



# 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](mailto:vimalsp@wilp.bits-pilani.ac.in)),
2. Dr. V Chandra Sekhar ([chandrasekhar.v@wilp.bits-pilani.ac.in](mailto: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 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](mailto:vimalsp@wilp.bits-pilani.ac.in)),  
2.Dr. V Chandra Sekhar ([chandrasekhar.v@wilp.bits-pilani.ac.in](mailto: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

| <u>Paradigm</u>     |  Supervised Learning |  Unsupervised Learning |  Reinforcement Learning |
|---------------------|--|---|--|
| <u>Objective</u>    | $p_{\theta}(y x)$  | $p_{\theta}(x)$   | $\pi_{\theta}(a s)$  |
| <u>Applications</u> | <ul style="list-style-type: none"><li>→ Classification</li><li>→ Regression</li></ul>                  | <ul style="list-style-type: none"><li>→ Inference</li><li>→ Generation</li></ul>                          | <ul style="list-style-type: none"><li>→ Prediction</li><li>→ Control</li></ul>                             |

## Types of Learning

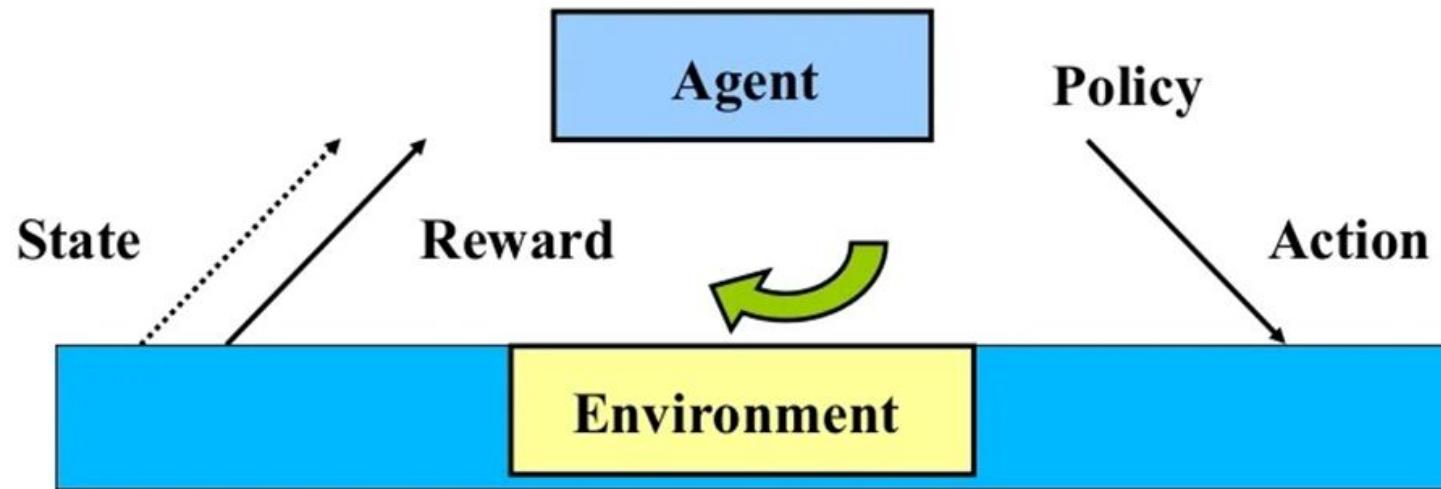
| <b>Criteria</b>         | <b>Supervised ML</b>                                  | <b>Unsupervised ML</b>                              | <b>Reinforcement ML</b>                   |
|-------------------------|---|---|---|
| <i>Definition</i>       | Learns by using labelled data                     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



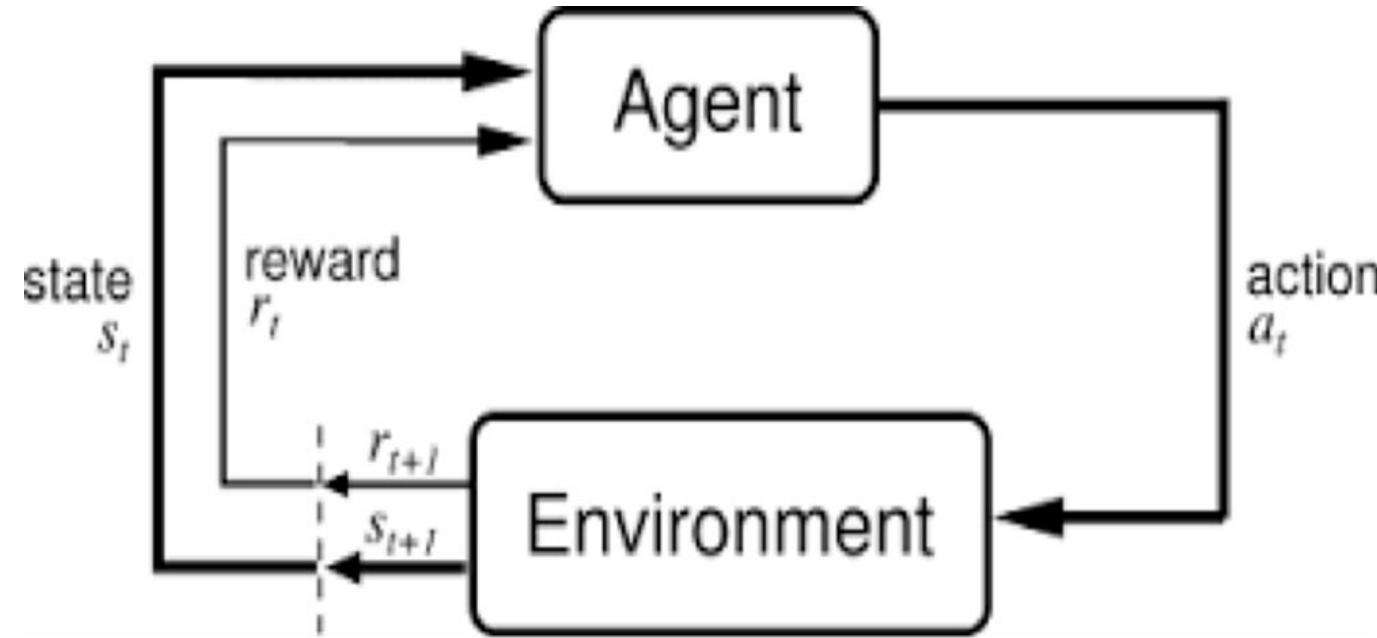
# Characteristics of RL

- 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:  $\gamma$ . If  $\gamma$  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.*



**BITS Pilani**  
Pilani | Dubai | Goa | Hyderabad

**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  $t = 0, 1, 2, \dots$ , 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  $G_t$

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \\ &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \end{aligned}$$

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



# Elements of Reinforcement Learning

- **Policy ( $\pi$ )**

- 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  $\pi$ , then  $\pi(a/s)$  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  $\pi(a/s)$  at time  $t$  with policy  $\pi$ .



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

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



# Elements of Reinforcement Learning

## •Action value function

- determines the goodness of the action taken by the agent from a given state for policy  $\pi$ .
- This function gives the value which is the expected return starting from state  $s$  at time step  $t$ , with action  $a$ , and following policy  $\pi$  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.

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

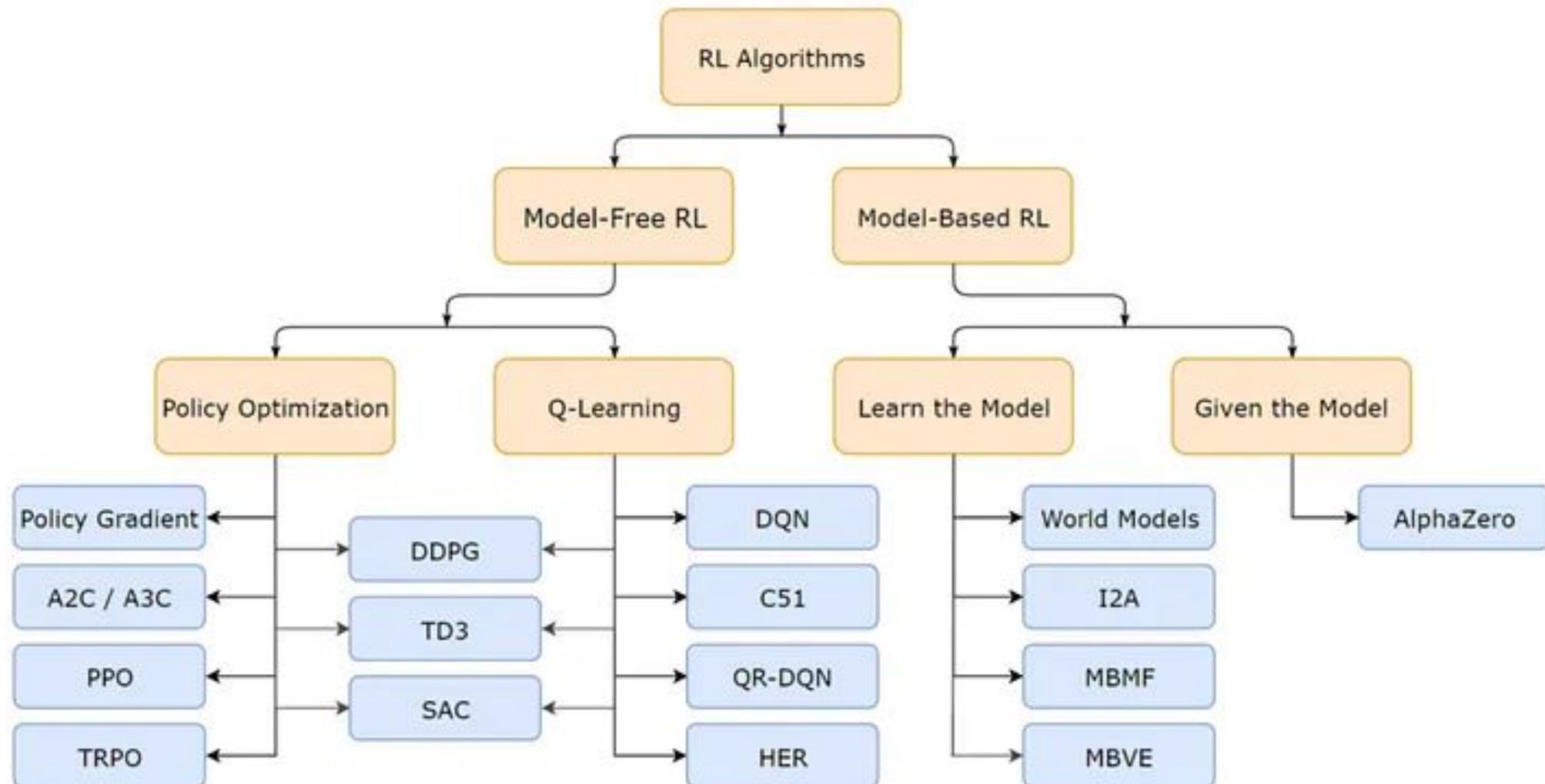


# Elements of Reinforcement Learning

- **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





# 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

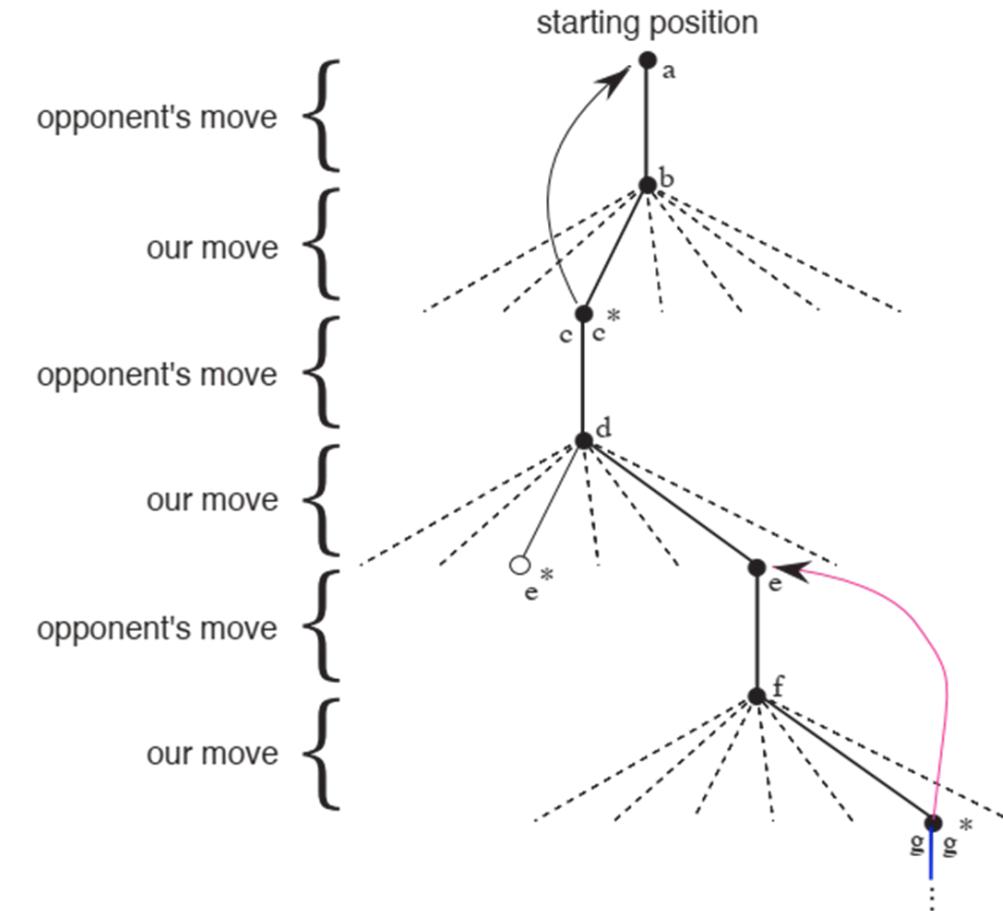
- We set up a table of numbers, one for each possible state of the game. Each number will be the latest estimate of the probability of our winning from that state.
- We treat this estimate as the state's value, and the whole table is the learned value function.
- State A has higher value than state B, 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)

## Session #2-3: Multi-armed Bandits

### Instructors :

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



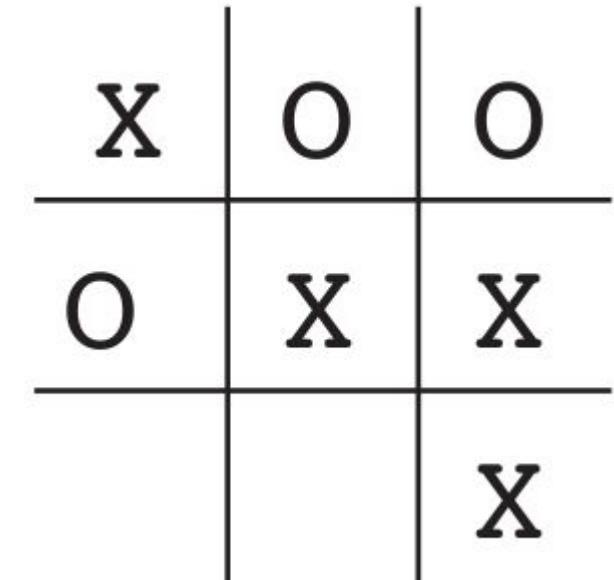
# Tic-Tac-Toe

|   |   |   |
|---|---|---|
| X | O | O |
| O | X | X |
|   |   | X |

# Tic-Tac-Toe

| 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



Set up a table of states initial values

# Tic-Tac-Toe ( prev. class)

States

Initial Values

|       |     |
|-------|-----|
| $[X]$ | 0.5 |
|-------|-----|

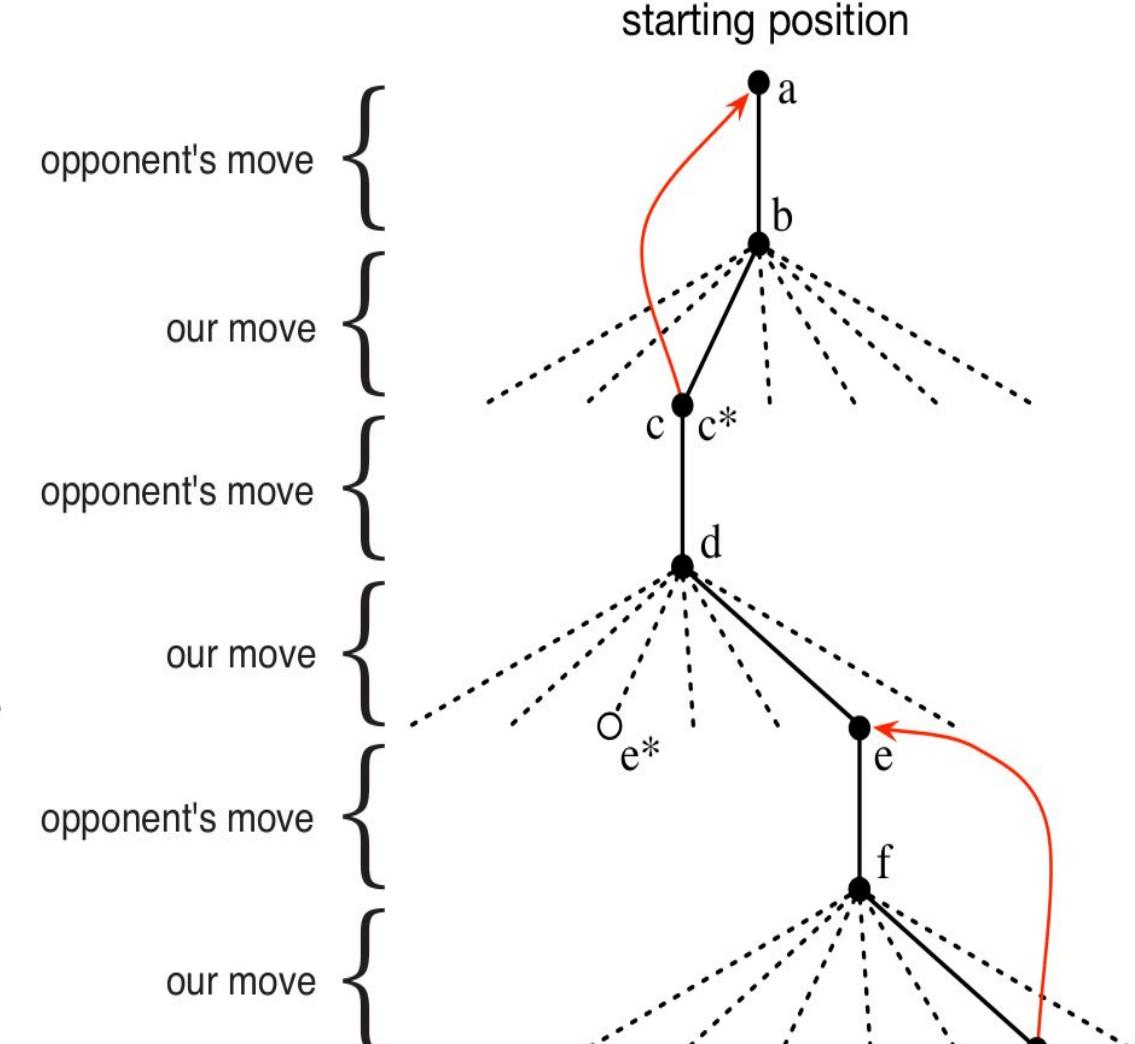
|                            |     |
|----------------------------|-----|
| $[X \ O \ O]$<br>$\quad X$ | 0.5 |
|----------------------------|-----|

|                                    |     |
|------------------------------------|-----|
| $[X \ O \ O]$<br>$\quad X \quad X$ | 1.0 |
|------------------------------------|-----|

|                                    |   |
|------------------------------------|---|
| $[X \ O]$<br>$\quad X \quad X \ O$ | 0 |
|------------------------------------|---|

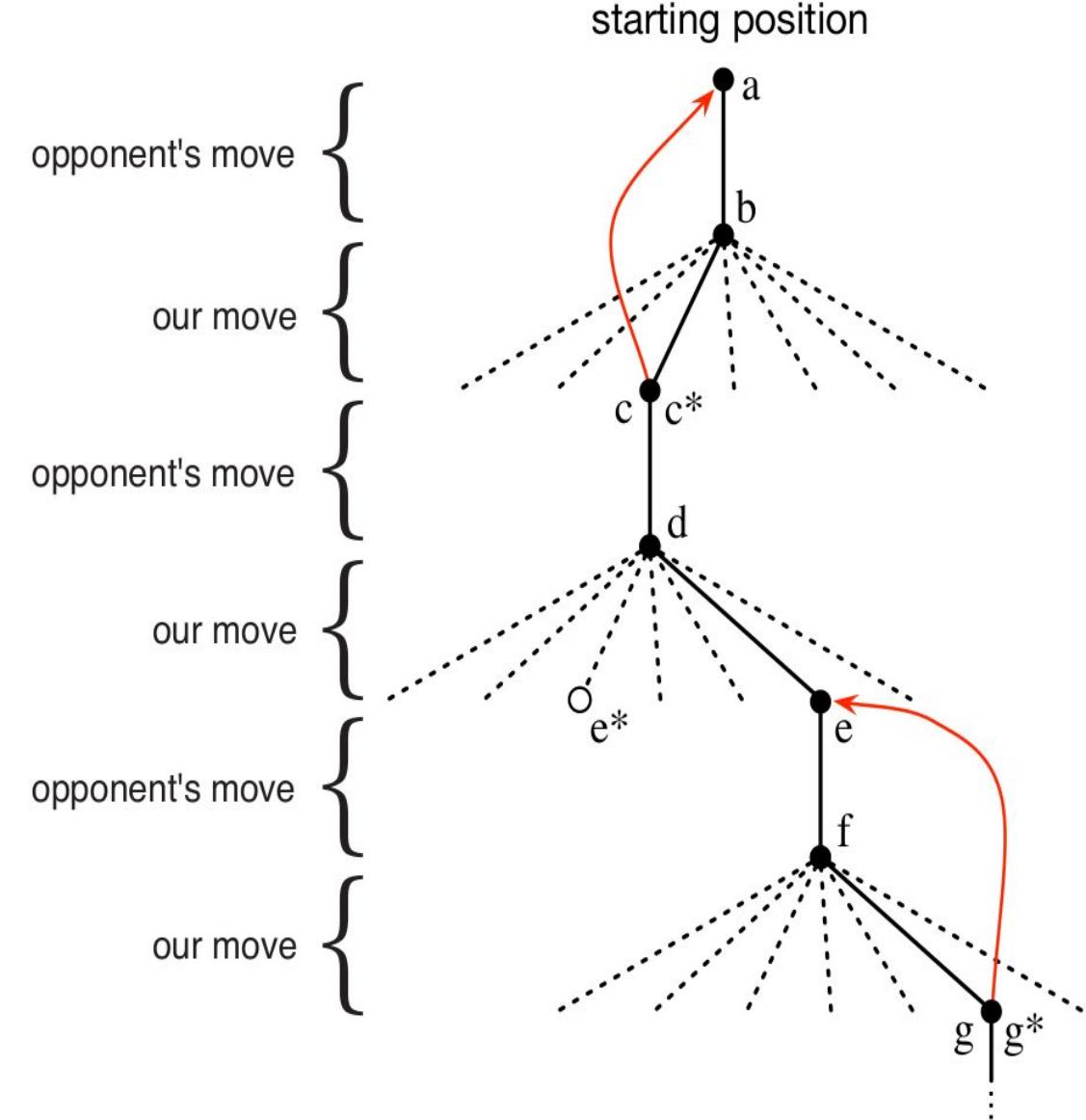
...

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



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

# Tic-Tac-Toe ( prev. class)



Temporal Difference Learning Rule

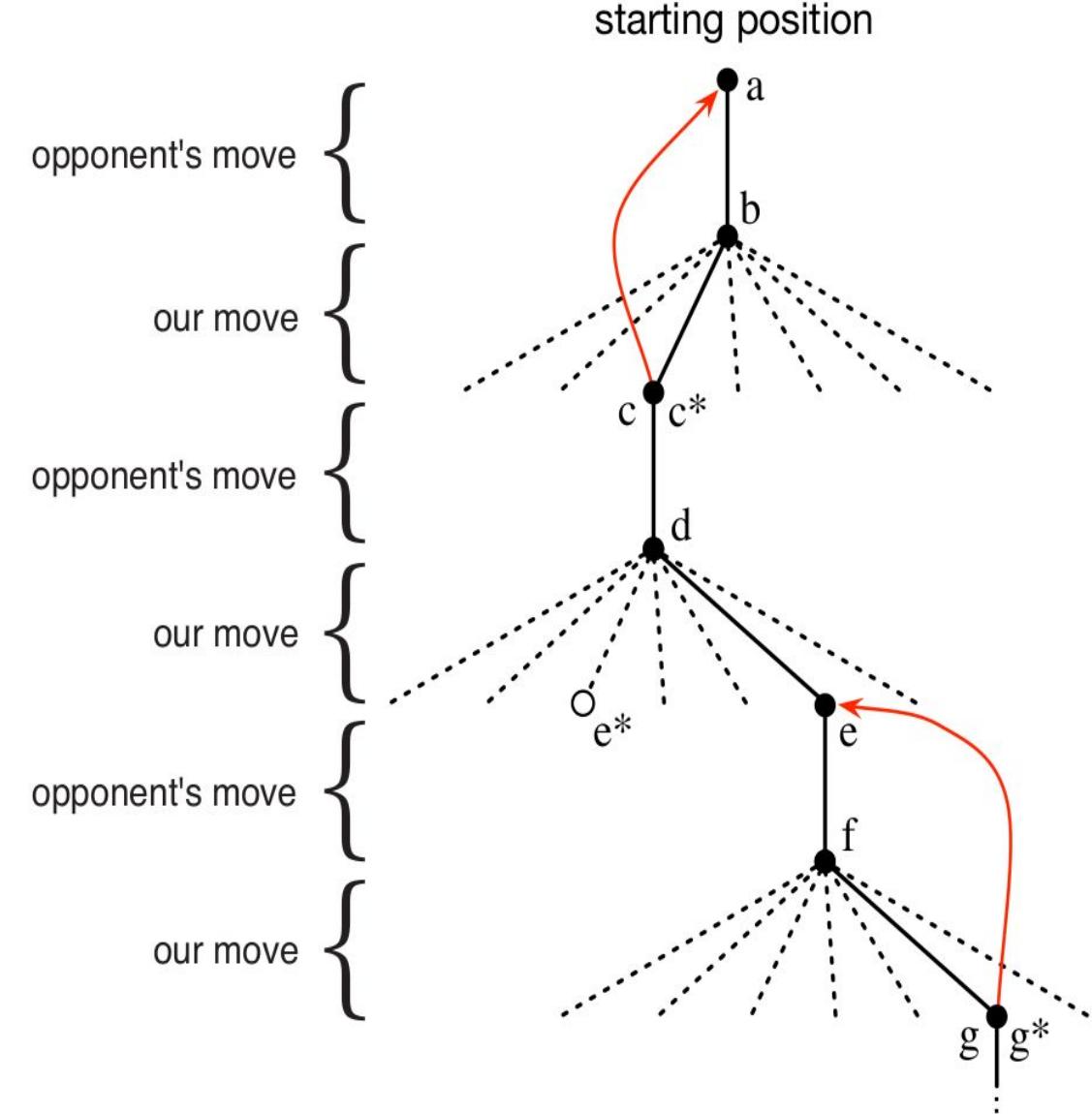
$$V(S_t) \leftarrow V(S_t) + \alpha [V(S_{t+1}) - V(S_t)]$$

$\alpha$ - Step Size Parameter

# Tic-Tac-Toe ( prev. class)

Questions:

- (1) What happens if  $\alpha$  is gradually made to 0 over many games with the opponent?
- (2) What happens if  $\alpha$  is gradually reduced over many games, but never made 0?
- (3) What happens if  $\alpha$  is kept constant throughout its life time?



## Temporal Difference Learning Rule

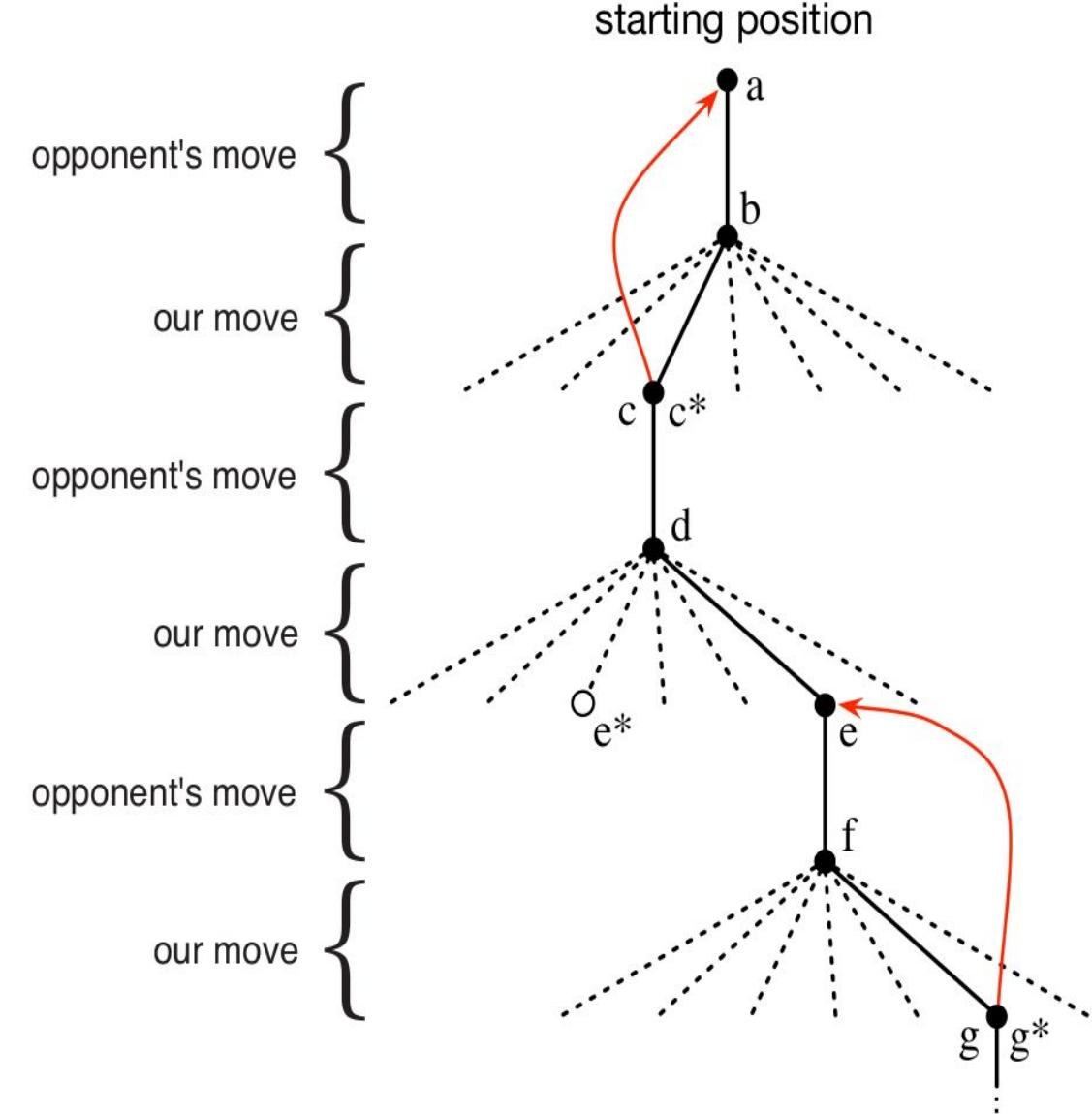
$$V(S_t) \leftarrow V(S_t) + \alpha [V(S_{t+1}) - V(S_t)]$$

$\alpha$ - Step Size Parameter

# Tic-Tac-Toe ( 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
  - o Use the values of states most of the time (exploration) and explore rest of the time.



## Temporal Difference Learning Rule

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

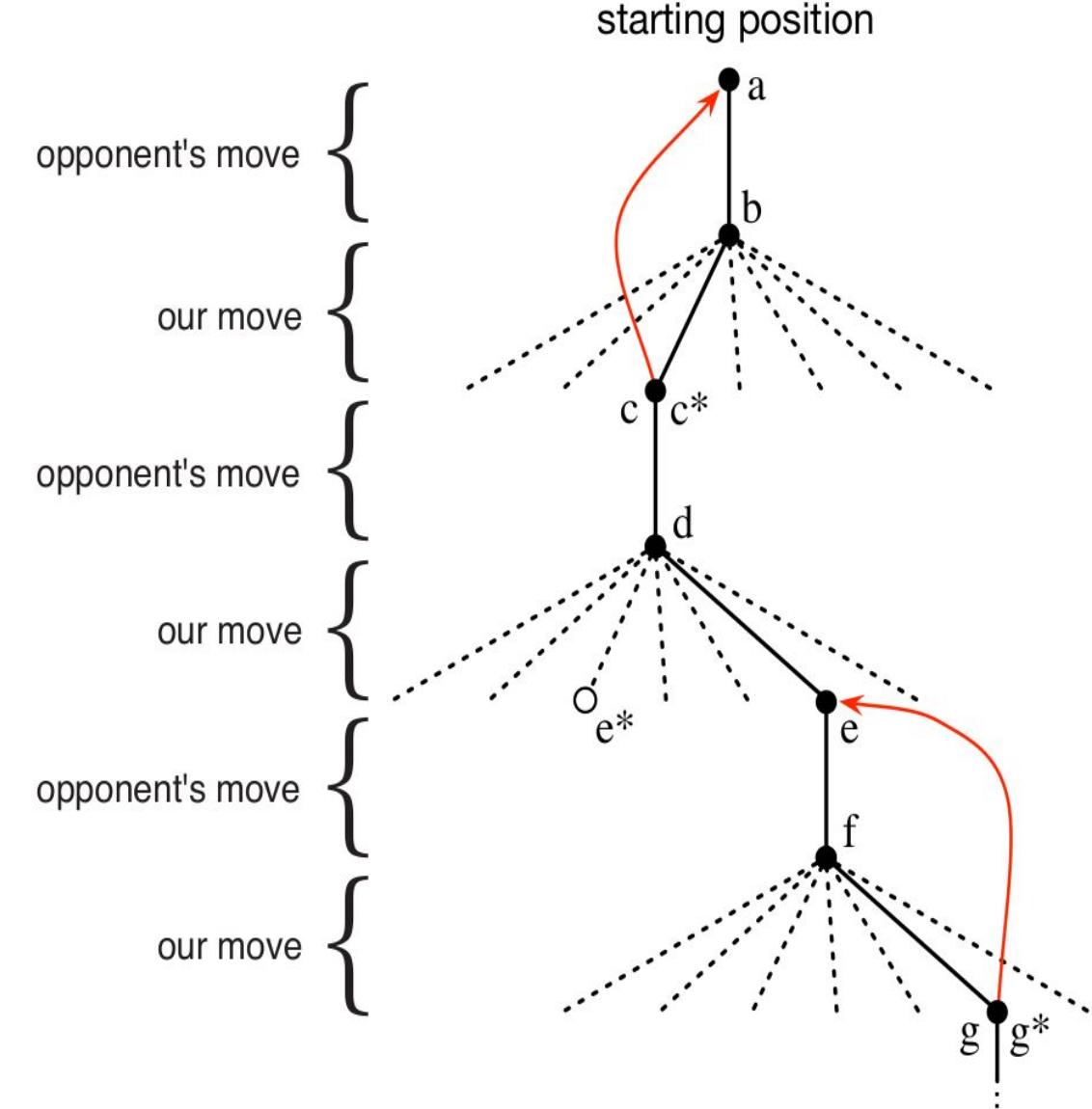
$\alpha$ - Step Size Parameter

# Tic-Tac-Toe ( 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

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

$\alpha$ - 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 ones**?
- 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 ones**?
- 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

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

# 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



$$E[a] = -\$0.5$$



$$E[b] = -\$0.2$$



$$E[c] = \$0.1$$



$$E[d] = \$0.11$$

# K-armed Bandit Problem



-1, -1, 5  
 $\hat{E}[a] = 1$



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



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



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

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

# K-armed Bandit Problem



-1, -1, 5  
 $\hat{E}[a] = 1$



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



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



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

# K-armed Bandit Problem

- How to maintain the estimate of expected rewards for each action?  
Average the rewards actually received !!!

$$\begin{aligned} Q_t(a) &\doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} \\ &= \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}} \end{aligned}$$

- How to use the estimate in selecting the right action?

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

# K-armed Bandit Problem

- 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 !!!**

- Exploration vs. Exploitation?

## $\epsilon$ -Greedy Action Selection / near-greedy action selection

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

# K-armed Bandit Problem



$\{-1, -1, 5\}$

$$N_{11}(a)=3$$

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

$$Q_{11}(a)=1$$



$\{-0.2, -0.2\}$

$$N_{11}(b)=2$$

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

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



$\{-0.5, -0.5, -0.5\}$

$$N_{11}(c)=3$$

$$q_*(c)=\$0.1$$

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



$\{-2, -2\}$

$$N_{11}(d)=2$$

$$\text{q*}(d)=\$1$$

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

# K-armed Bandit Problem

## Greedy Action



# K-armed Bandit Problem

Action to Explore



$\{-1, -1, 5\}$

$$N_{11}(a)=3$$

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

$$Q_{11}(a)=1$$



$\{-0.2, -0.2\}$

$$N_{11}(b)=2$$

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

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



$\{-0.5, -0.5, -0.5\}$

$$N_{11}(c)=3$$

$$q_*(c)=\$0.1$$

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



$\{-2, -2\}$

$$N_{11}(d)=2$$

$$\text{q*}(d)=\$1$$

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

# K-armed Bandit Problem

## $\epsilon$ -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  $\epsilon$ -greedy action selection an infinite number of times. This ensures that all the  $Q_t(a)$  converge to  $q_*(a)$ .
- Easy to implement / optimize for epsilon / yields good results



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

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

$p(\text{greedy action})$

$$= p(\text{greedy action AND greedy selection}) + p(\text{greedy action AND random selection})$$

$$= p(\text{greedy action} \mid \text{greedy selection}) p(\text{greedy selection})$$

$$+ p(\text{greedy action} \mid \text{random selection}) p(\text{random selection})$$

$$= p(\text{greedy action} \mid \text{greedy selection}) (1-\epsilon) + p(\text{greedy action} \mid \text{random selection}) (\epsilon)$$

$$= p(\text{greedy action} \mid \text{greedy selection}) (0.5) + p(\text{greedy action} \mid \text{random selection}) (0.5)$$

$$= (1)(0.5) + (0.5)(0.5)$$

$$= 0.5 + 0.25$$

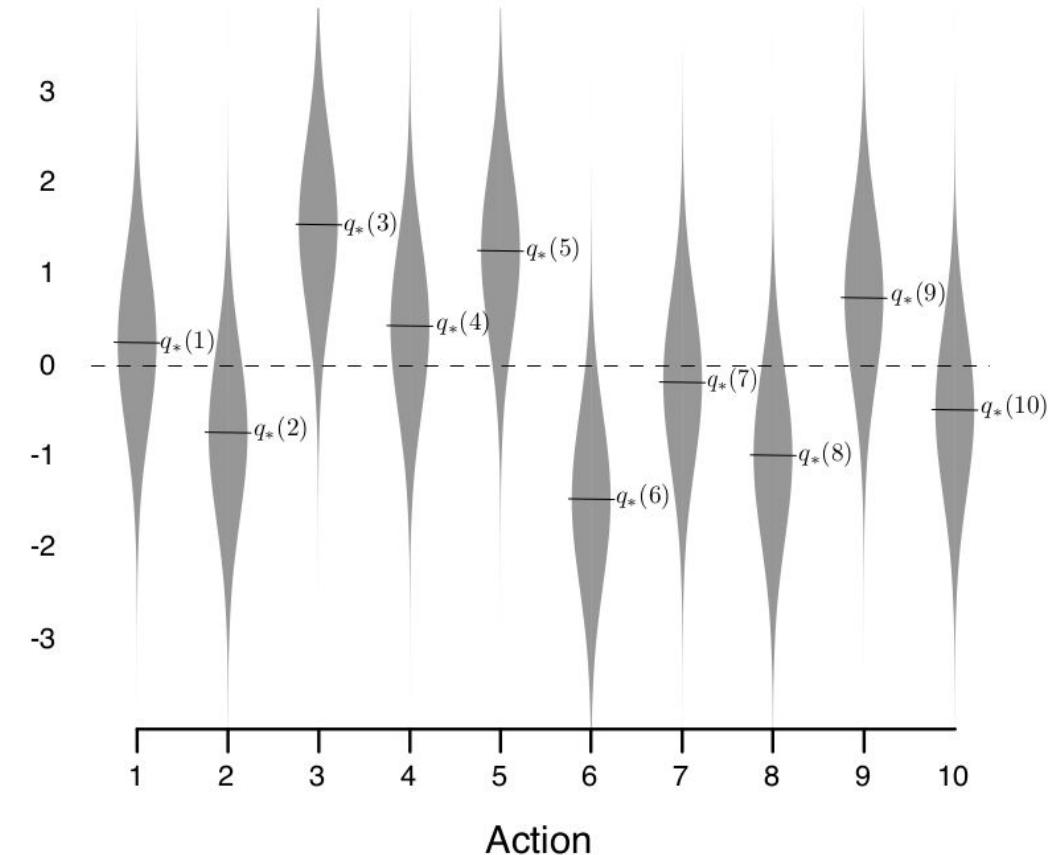
$$= 0.75$$

# 10-armed Testbed

## Example:

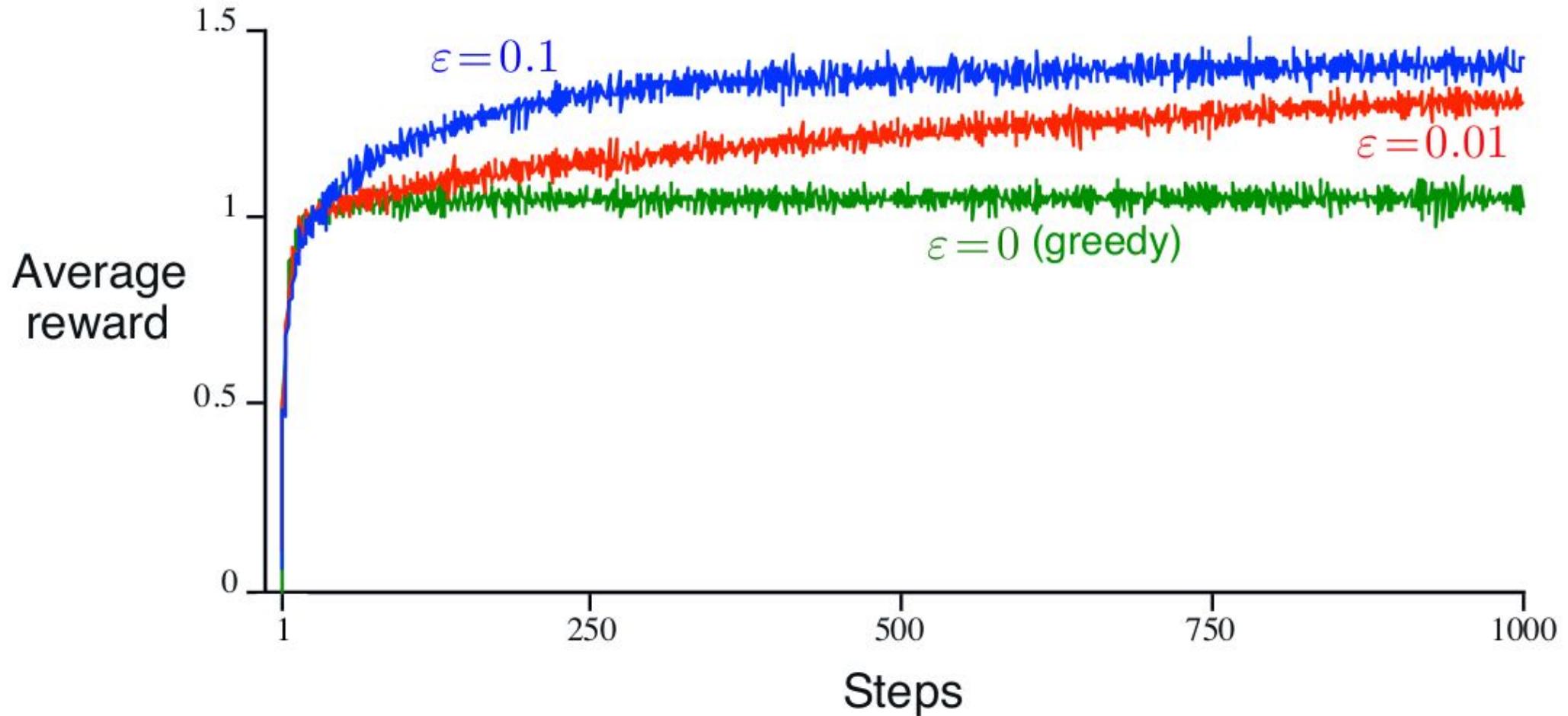
- A set of 2000 randomly generated k -armed bandit problems with  $k = 10$
- Action values were selected according to a normal (Gaussian) distribution with mean 0 and variance 1.
- While selecting action  $A_t$  at time step t, the actual reward,  $R_t$ , was selected from a normal distribution with 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

Reward distribution

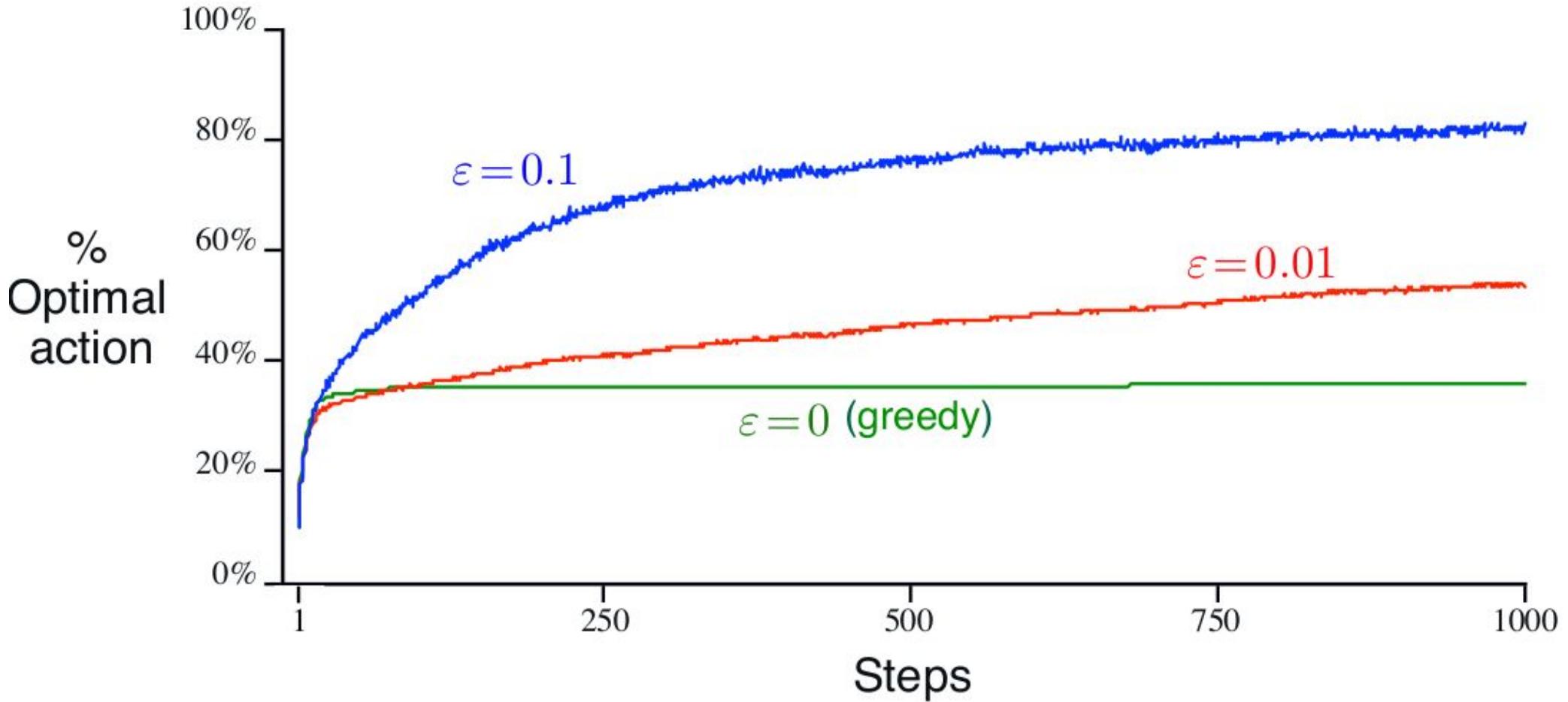


An example bandit problem from the 10-armed testbed

# Average performance of $\epsilon$ -greedy action-value methods on the 10-armed testbed



# Average performance of $\epsilon$ -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  $\epsilon$ -greedy action selection, sample-average action-value estimates, and initial estimates of  $Q_1(a) = 0$ , for all  $a$ .

Suppose the initial sequence of actions and rewards is  $A_1 = 1, R_1 = 1, A_2 = 2, R_2 = 1, A_3 = 2, R_3 = 2, A_4 = 2, R_4 = 2, A_5 = 3, R_5 = 0$ .

On some of these time steps the  $\epsilon$  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;

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

- Given  $Q_n$  and the  $n$ th reward,  $R_n$ , the new average of all  $n$  rewards can be computed as follows

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

# Incremental Implementation

## Note:

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

## Discussion:

Const vs. Variable step size?

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

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize} [\text{Target} - \text{OldEstimate}]$$

# Bandit Algorithm with Incremental Update/ $\epsilon$ -greedy selection

Initialize, for  $a = 1$  to  $k$ :

$$Q(a) \leftarrow 0$$

$$N(a) \leftarrow 0$$

Loop forever:

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

$$R \leftarrow \text{bandit}(A)$$

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

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



# Non-stationary Problem

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

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

# Non-stationary Problem

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

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

**Exponential recency-weighted average**

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

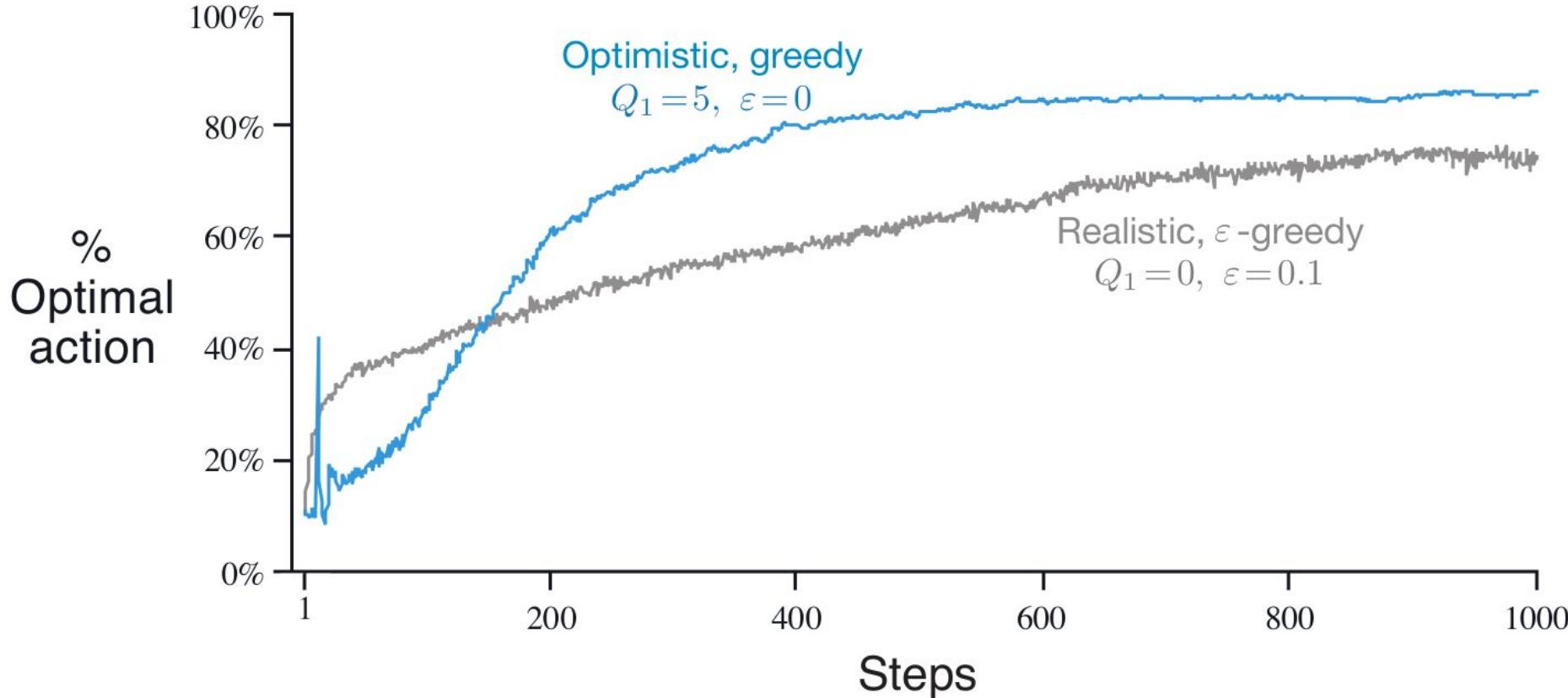
# Optimistic Initial Values

- 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  $\alpha$ , 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



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

## 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).

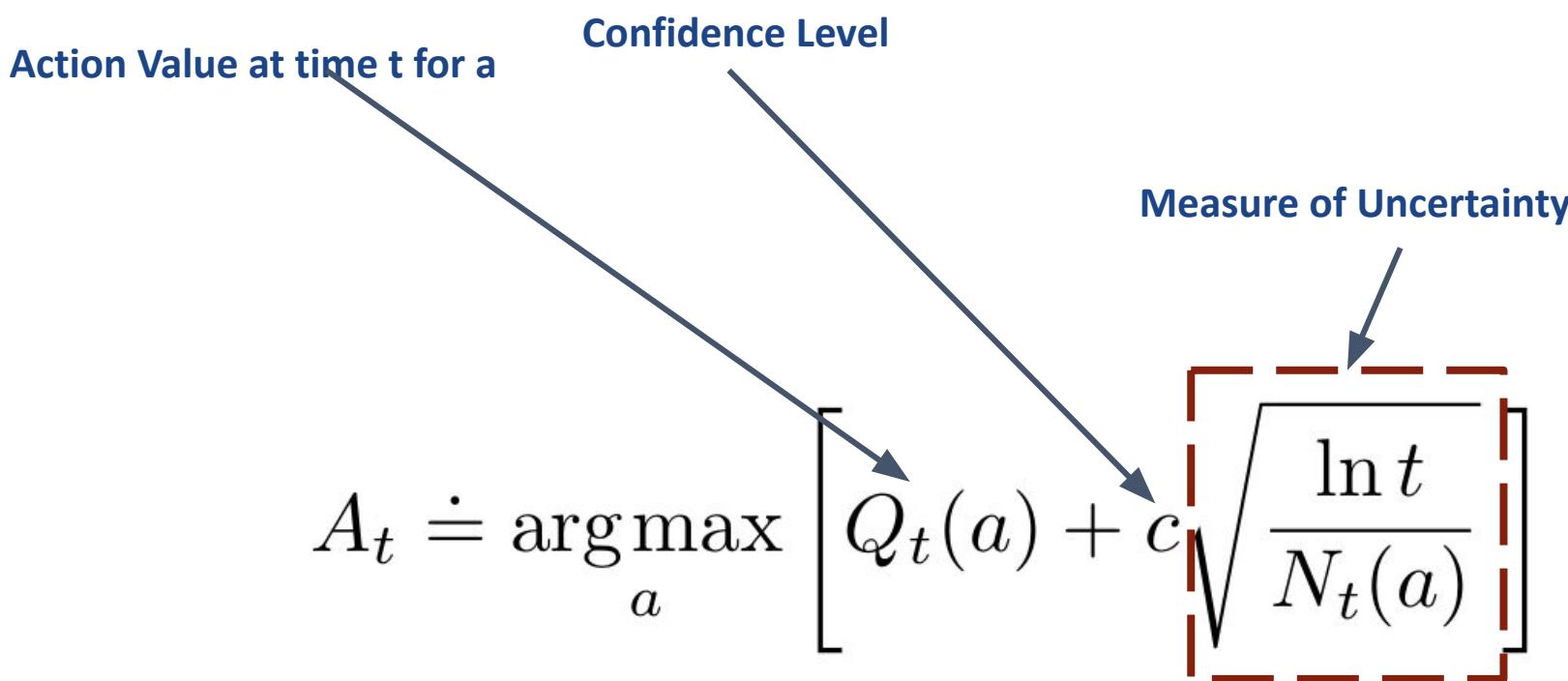
# Upper-Confidence-Bound Action Selection

- **$\epsilon$ -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.

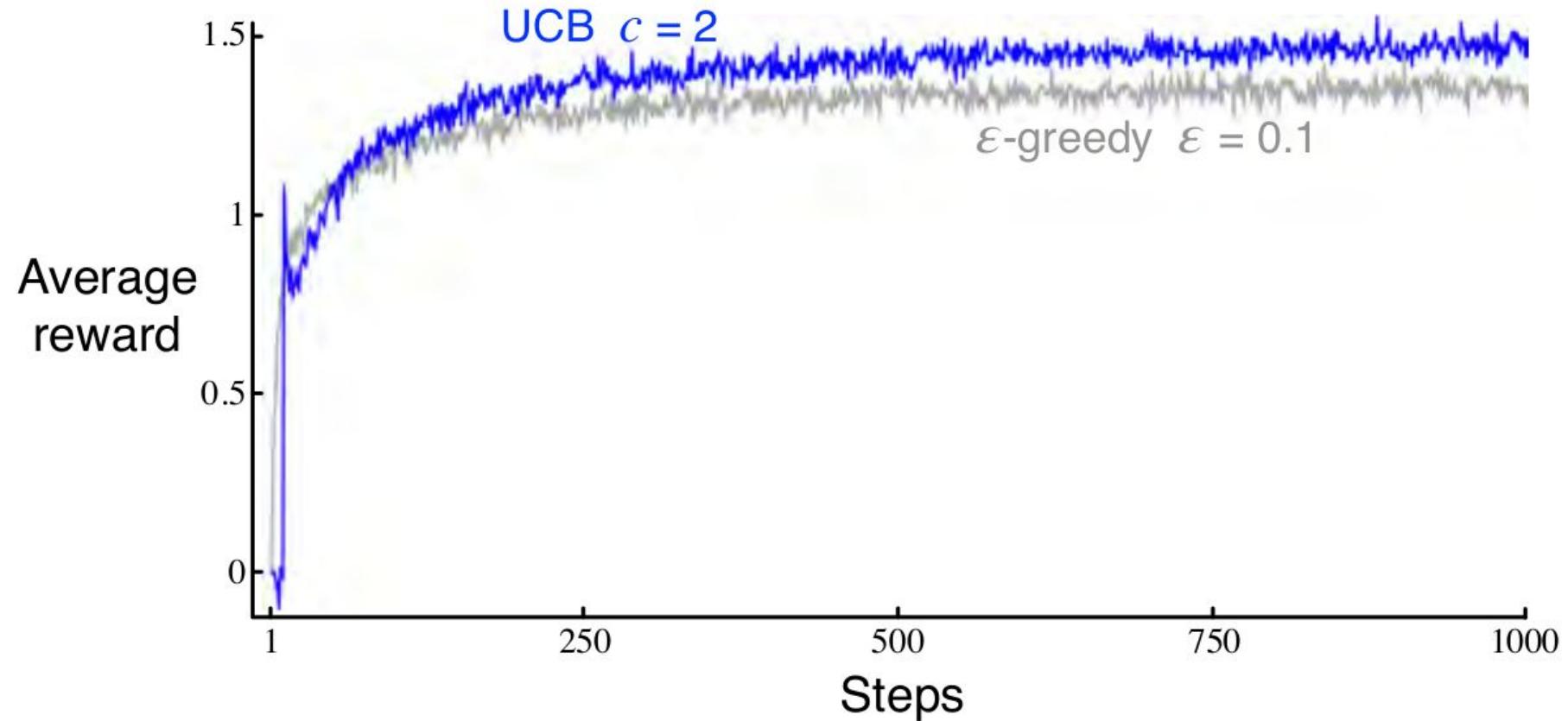
$$A_t \doteq \arg \max_a \left[ Q_t(a) + c \sqrt{\frac{\ln t}{N_t(a)}} \right]$$

# 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



# Upper-Confidence-Bound Action Selection



UCB often performs well, as shown here, but is more difficult than " $\epsilon$ -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 choose
  - 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?
  - $A_t = \underset{a}{\operatorname{argmax}}[H_t(a)]$ 
    - Hardmax results in no exploration -- deterministic action selection



# 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:  $A_t = \underset{a}{\operatorname{argmax}}[H_t(a_1), H_t(a_2), H_t(a_3)]$ 
  - 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

- $H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \bar{R}_t)$   $\frac{\partial \Pr(A_t)}{\partial H(A_t)}$  ?
- $\forall a \neq A_t, H_{t+1}(a) = H_t(a) + \alpha(R_t - \bar{R}_t)$   $\frac{\partial \Pr(a)}{\partial H(a)}$
- Update the preferences based on observed reward and a baseline reward ( $\bar{R}_t$ )
- 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

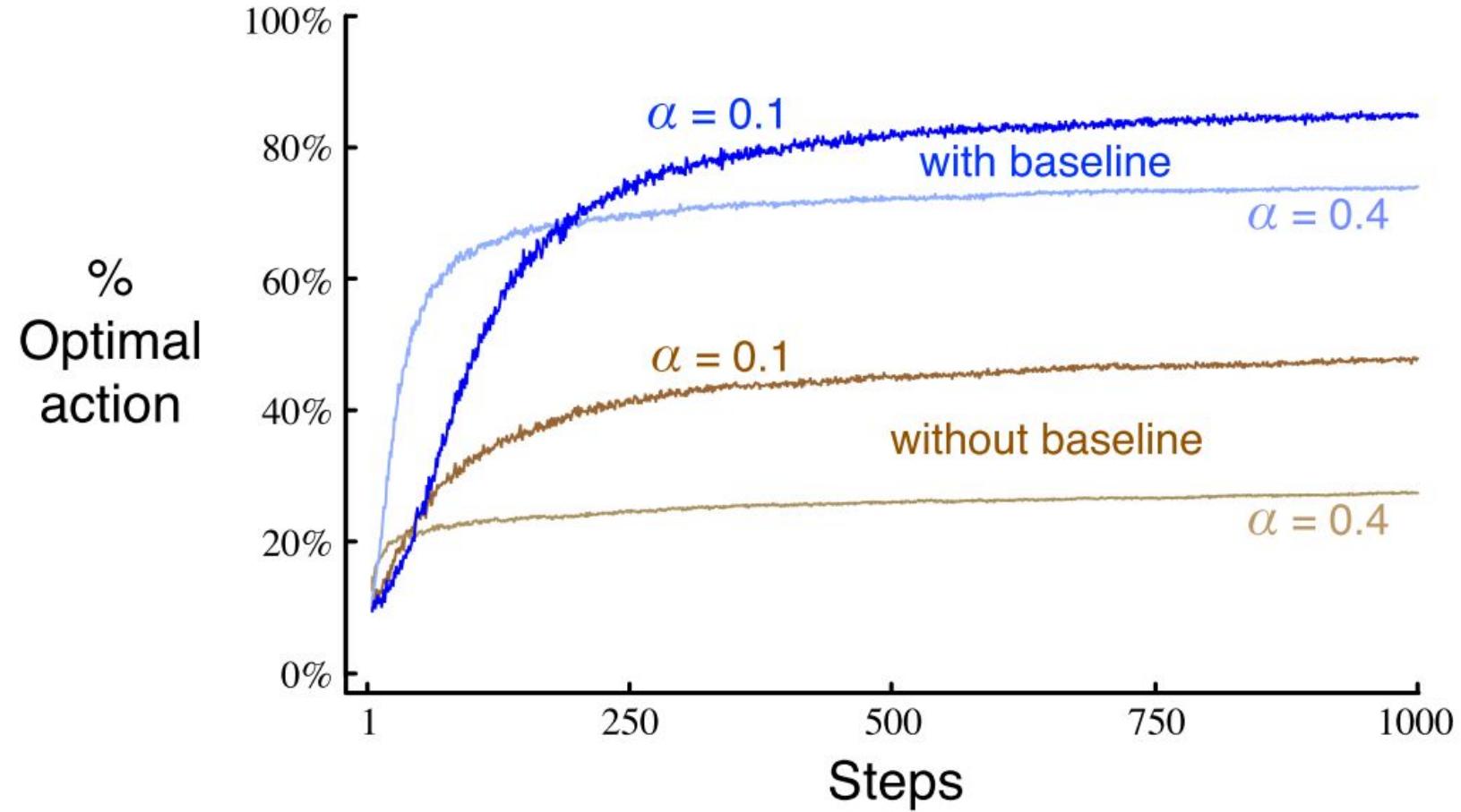
On each step, after selecting action  $A_t$  and receiving the reward  $R_t$ ,  
Update the action preferences :

$$H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \Pr(A_t))$$

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

$$H_{t+1}(A_t) \doteq H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)), \quad \text{and}$$

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



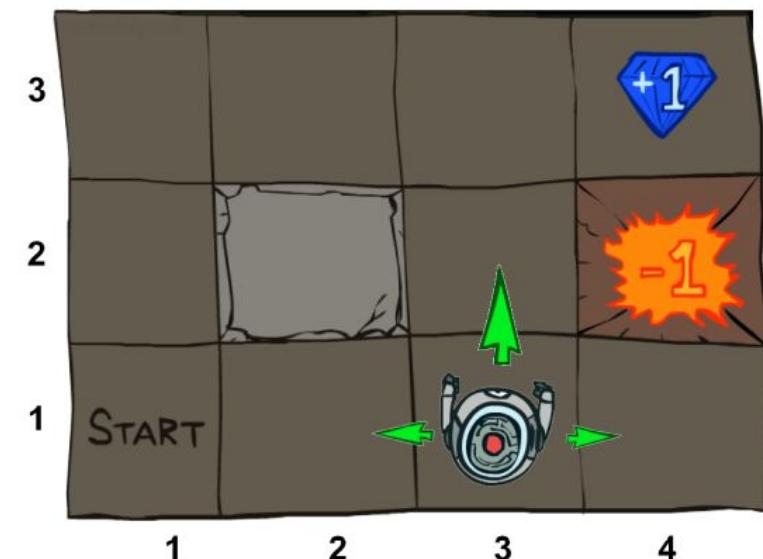


# What did we learn?

- **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, 2<sup>nd</sup> 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 !



## Session #4: Markov Decision Processes

### Instructors :

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

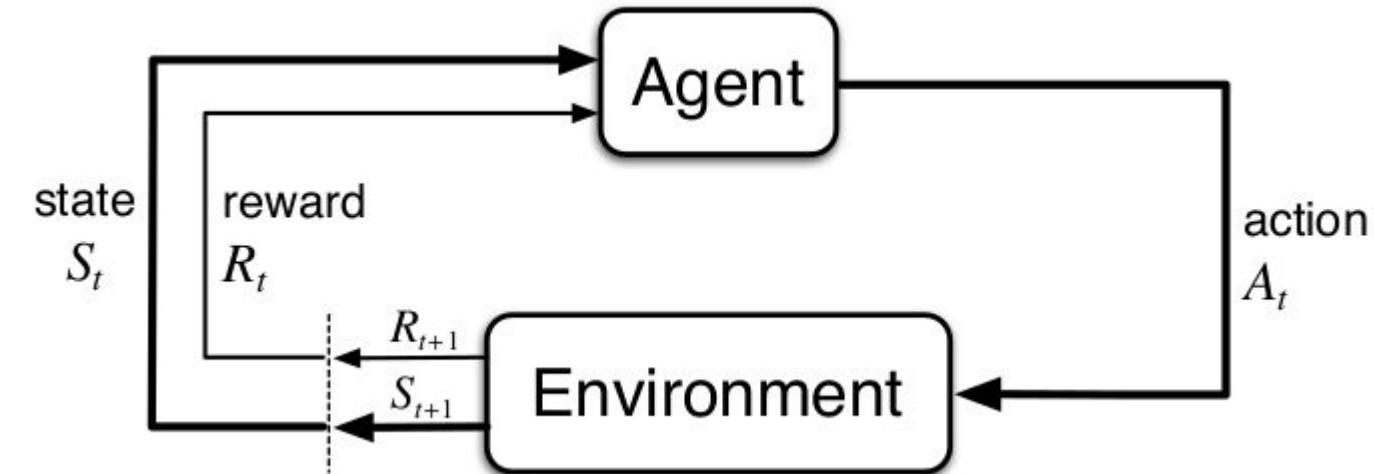
## Announcement !!!

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}

# 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

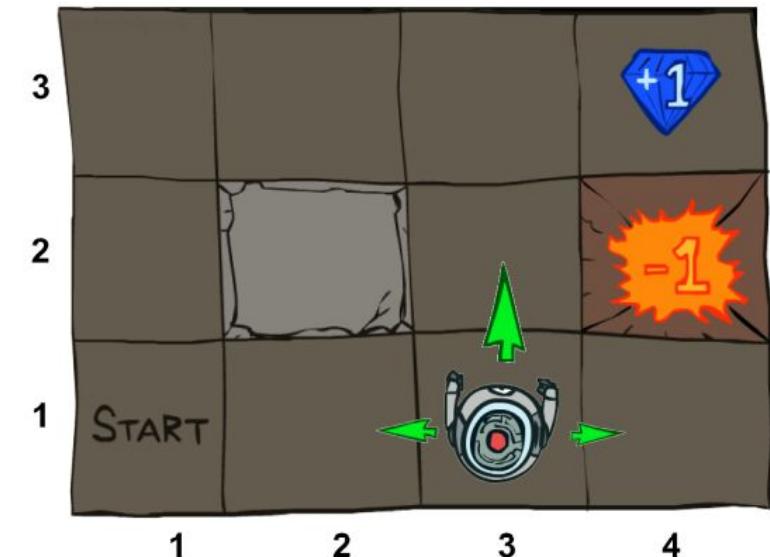


## Note:

- Interaction occurs in discrete time steps
- $$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots$$

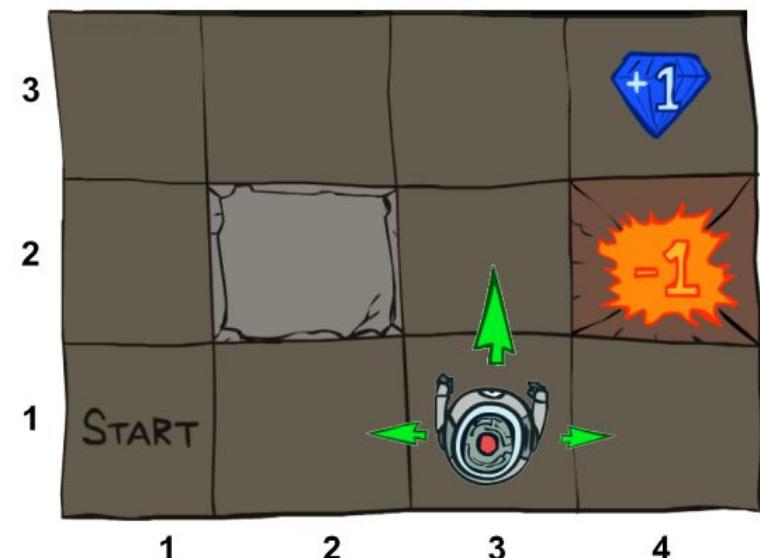
# Grid World Example

- 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

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

## State-transition probabilities

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

## Expected rewards for state-action-next-state triples

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



# 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).*

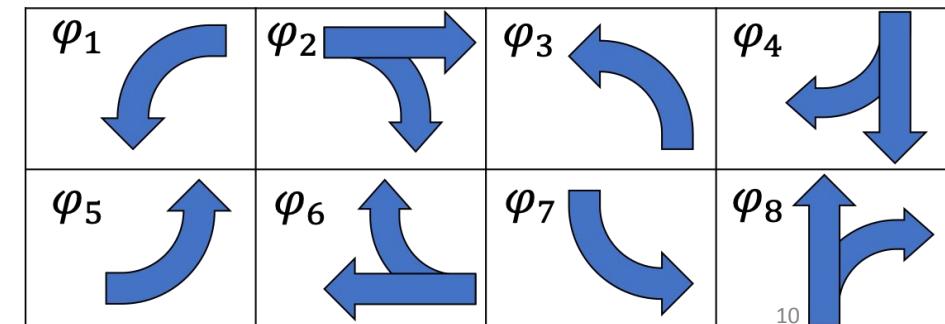
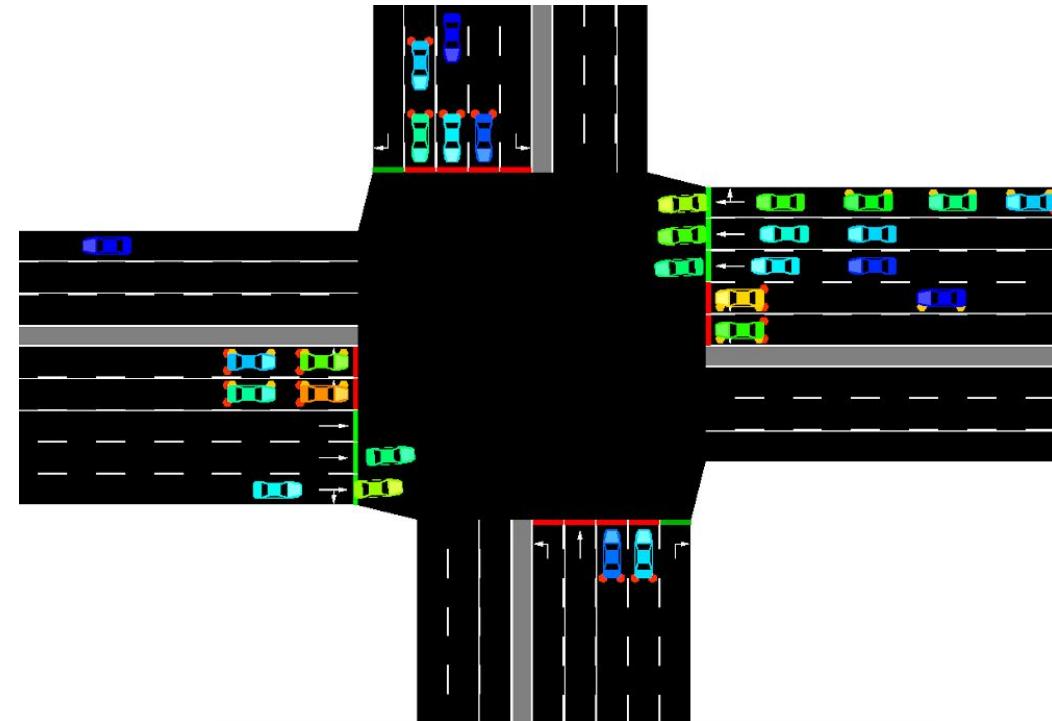
# MDP Formalization : Video Games

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



# 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



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



# MDP Formalization : Recycling Robot (Detailed Ex.)

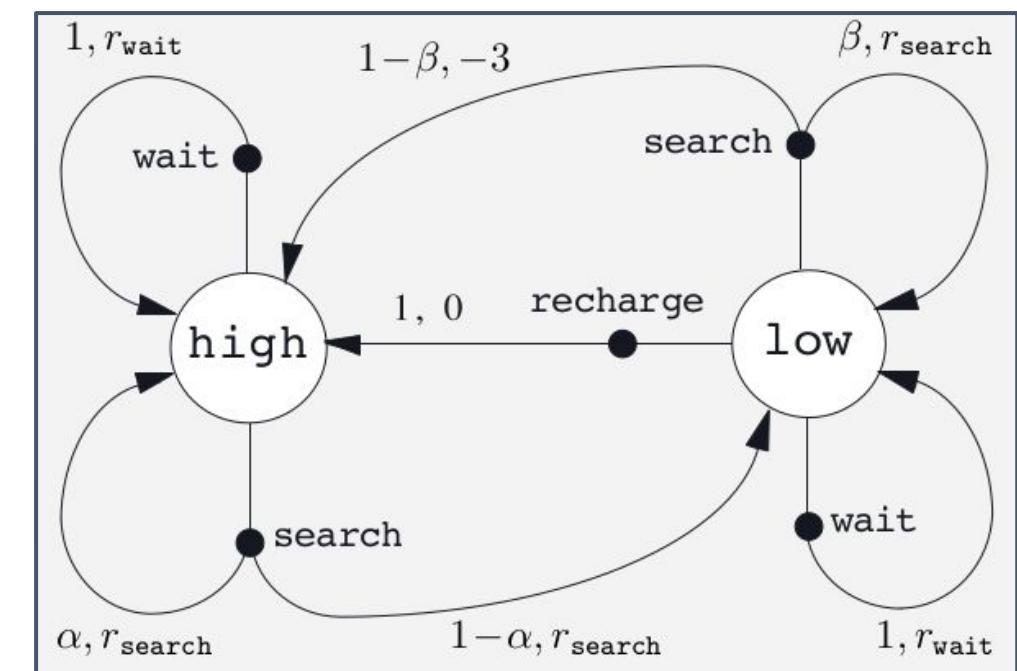
- *State-transition probabilities (contd...):*

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

# MDP Formalization : Recycling Robot (Detailed Ex.)

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

| $s$  | $a$      | $s'$ | $p(s'   s, a)$ | $r(s, a, s')$       |
|------|----------|------|----------------|---------------------|
| high | search   | high | $\alpha$       | $r_{\text{search}}$ |
| high | search   | low  | $1 - \alpha$   | $r_{\text{search}}$ |
| low  | search   | high | $1 - \beta$    | -3                  |
| low  | search   | low  | $\beta$        | $r_{\text{search}}$ |
| high | wait     | high | 1              | $r_{\text{wait}}$   |
| high | wait     | low  | 0              | -                   |
| low  | wait     | high | 0              | -                   |
| low  | wait     | low  | 1              | $r_{\text{wait}}$   |
| 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 ( $G_t$ ) 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*

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

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

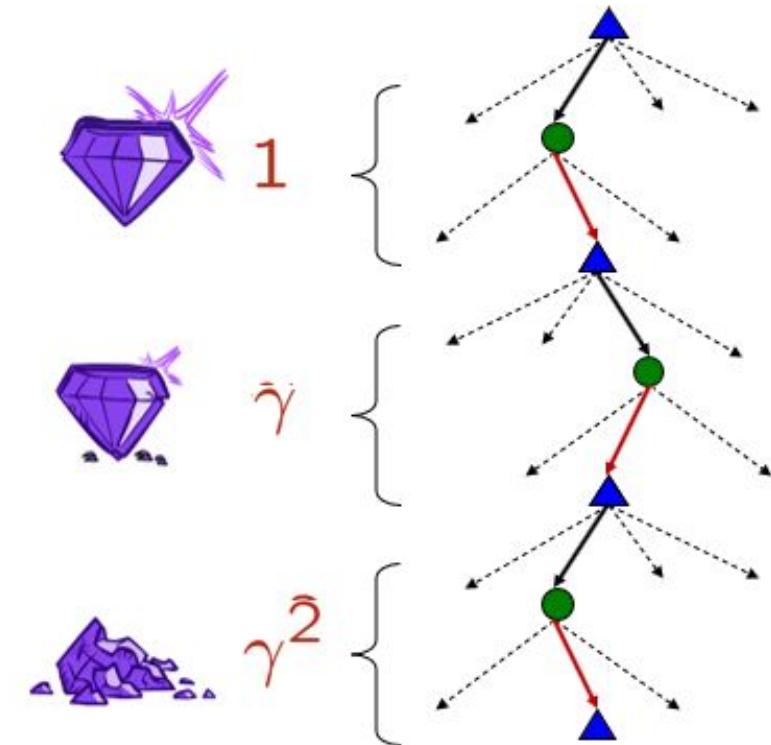
# Returns & Episodes

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

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

**Note:**  $\gamma$  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  $\gamma$  is 0?*
- *What if  $\gamma$  is 1?*
- *Computing discounted rewards incrementally*

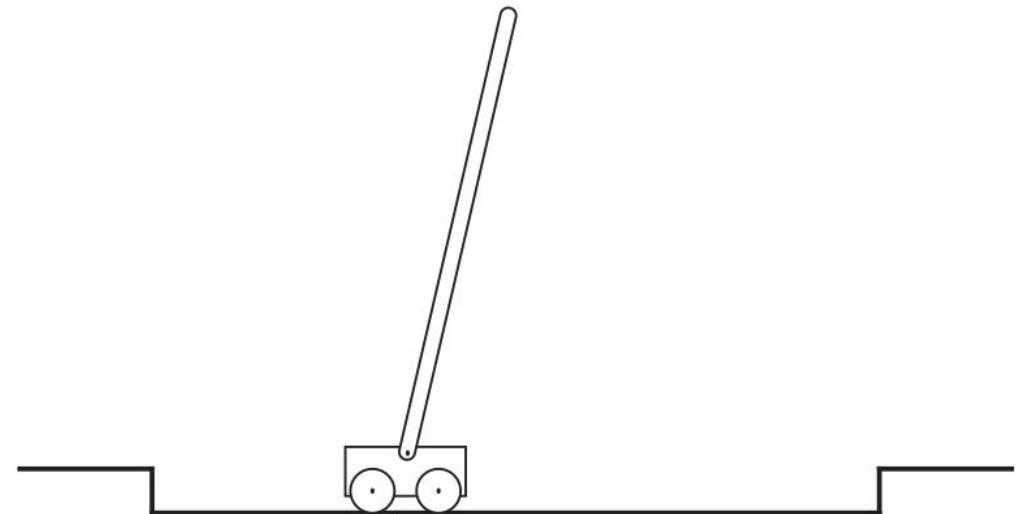
$$\begin{aligned}G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\&= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots) \\&= R_{t+1} + \gamma G_{t+1}\end{aligned}$$

- *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*

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

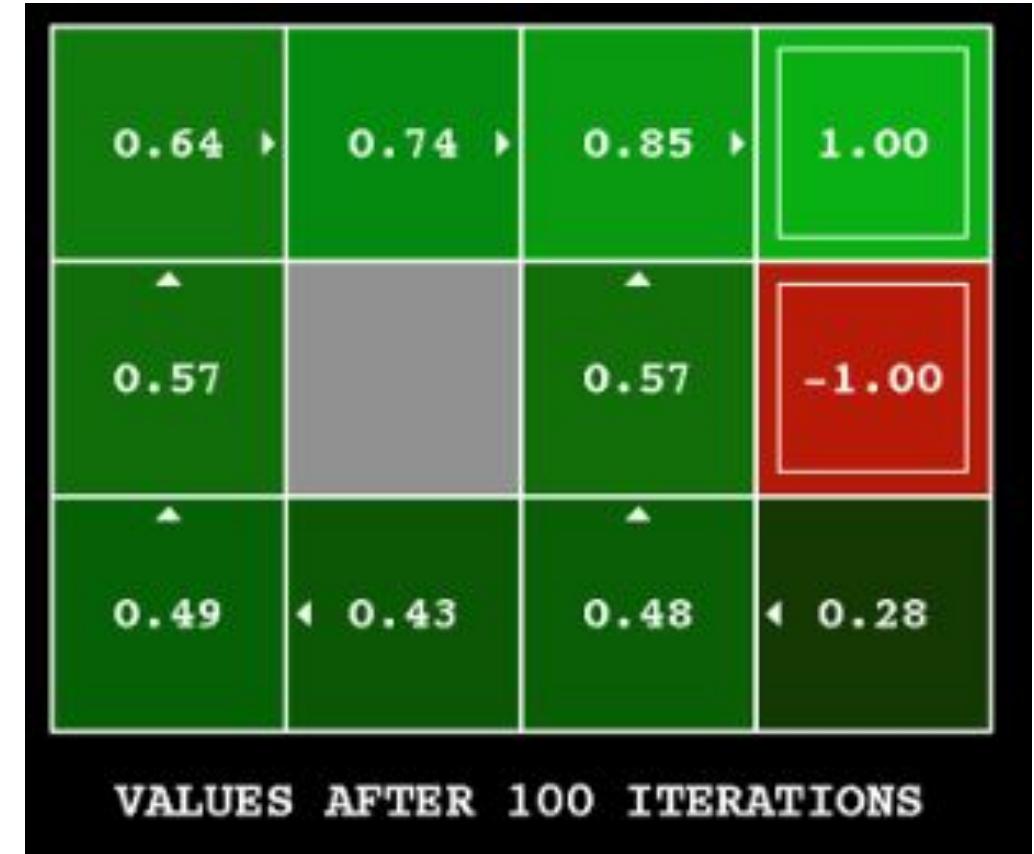
# Returns & Episodes

- **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 $\pi$

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

## Action-value function for policy $\pi$

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

# Defining Value Functions

State Value function in terms of Action-value function for policy  $\pi$

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

Action Value function in terms of State value function for policy  $\pi$

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

May skip to the next slide !

# Bellman Equation for $V_\pi$

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

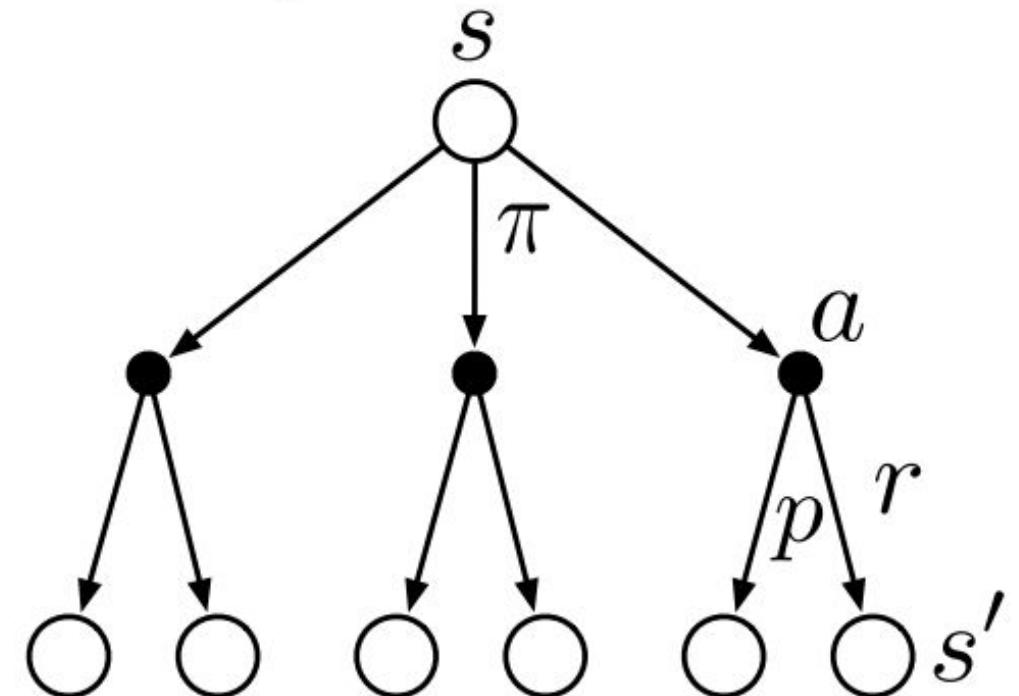
$$\begin{aligned}v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) \left[ r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \right] \\&= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) \left[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

# Bellman Equation for $V_\pi$

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

**Value of the start state must equal**

- (1) the (discounted) value of the expected next state,  
plus
- (1) the reward expected along the way

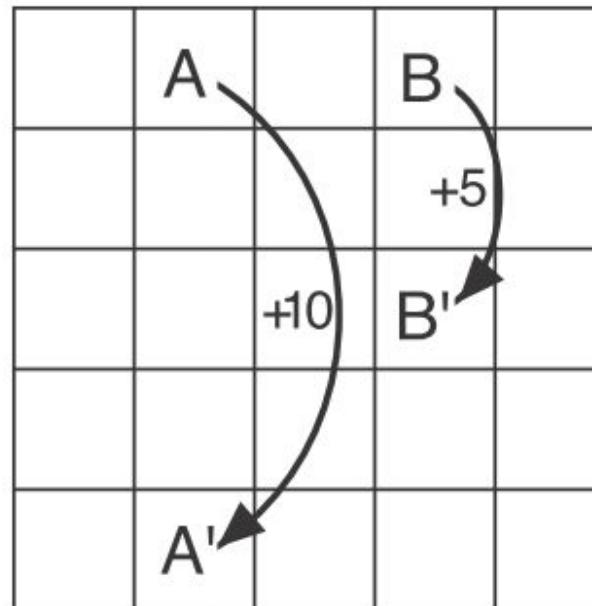


Backup Diagram

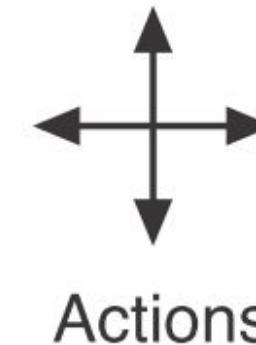
# Understanding $V_\pi(s)$ with Gridworld

## 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

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

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

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

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

|      |      |      |      |      |
|------|------|------|------|------|
| 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  $\rightarrow$  optimal policy (denoted as  $\pi_*$ )
  - 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*

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a)$$

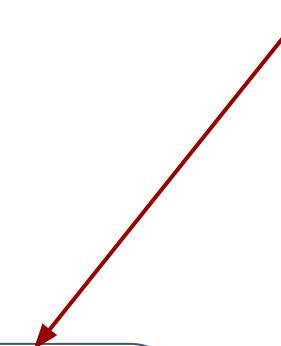
$$= \max_a \mathbb{E}_{\pi_*}[G_t \mid S_t = s, A_t = a]$$

$$= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a]$$

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

$$= \max_a \sum_{s', r} p(s', r \mid s, a) [r + \gamma v_*(s')].$$

**Bellman optimality equation for  $V_*$**





# Optimal Policies and Optimal Value Functions

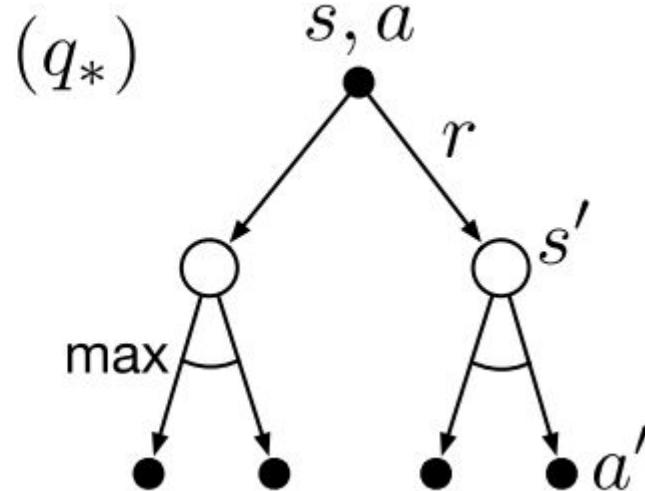
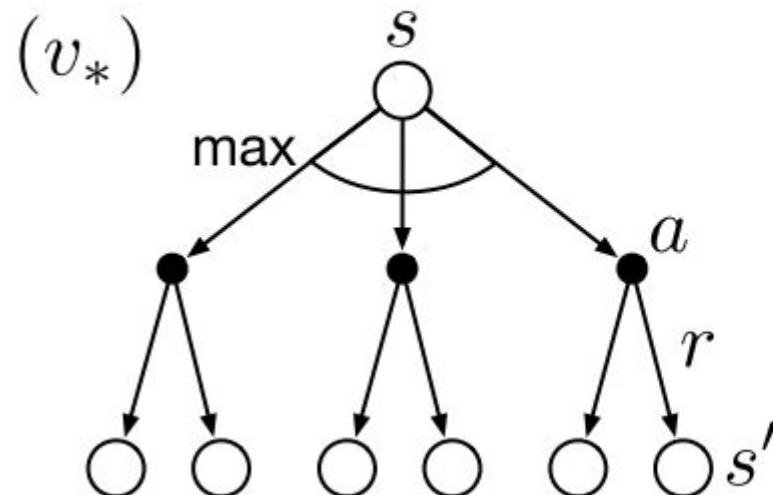
Bellman optimality equation - expresses that *the value of a state under an optimal policy must equal the expected return for the best action from that state*

## Bellman optimality equation for $q_*$

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

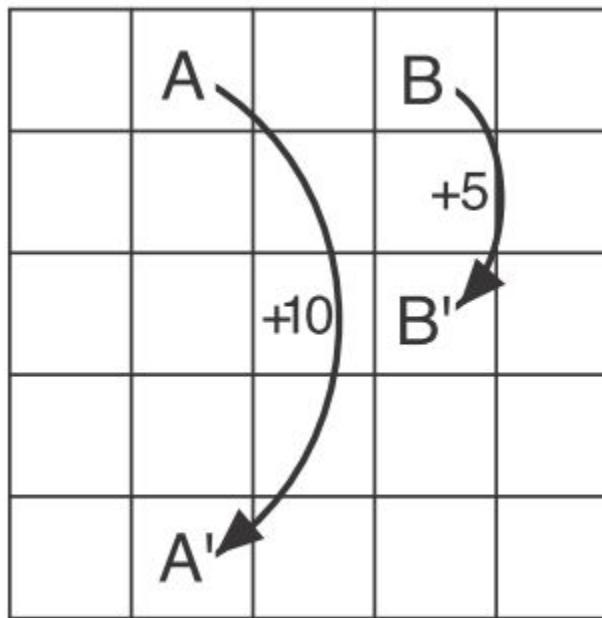
# Optimal Policies and Optimal Value Functions

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  $q^*$

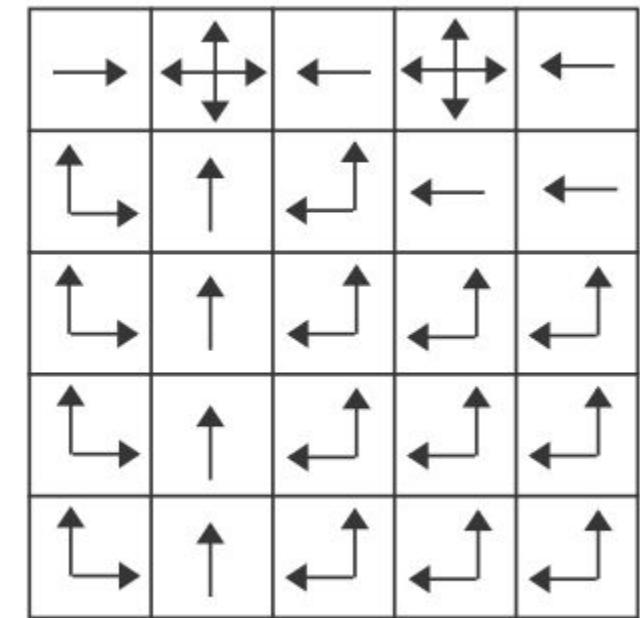
# Optimal solutions to the gridworld example



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 |

$v_*$



$\pi_*$

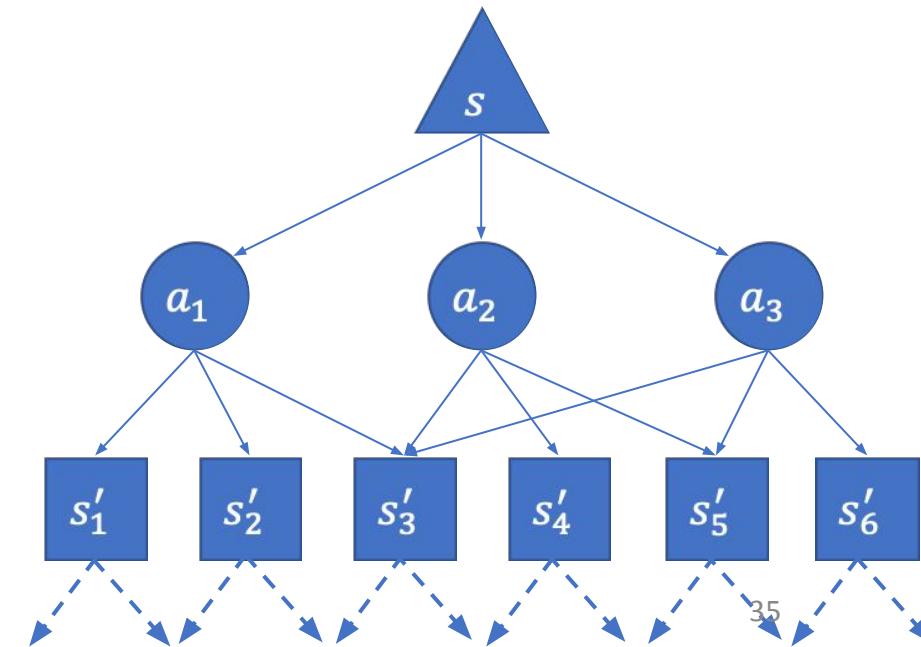


# Break for 5 mins

# 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,  $\pi^*$
- $\pi^*(s) = ?$
- $= \underset{a}{\operatorname{argmax}} [\sum_{s'} P(s'|s, a) R(s, a, s')]$ 
  - Must also optimize over the future (next steps)
- $= \underset{a}{\operatorname{argmax}} [\sum_{s'} P(s'|s, a) (R(s, a, s') + \mathbb{E}_{\pi^*}[G|s'])]$ 
 $v^*(s')$





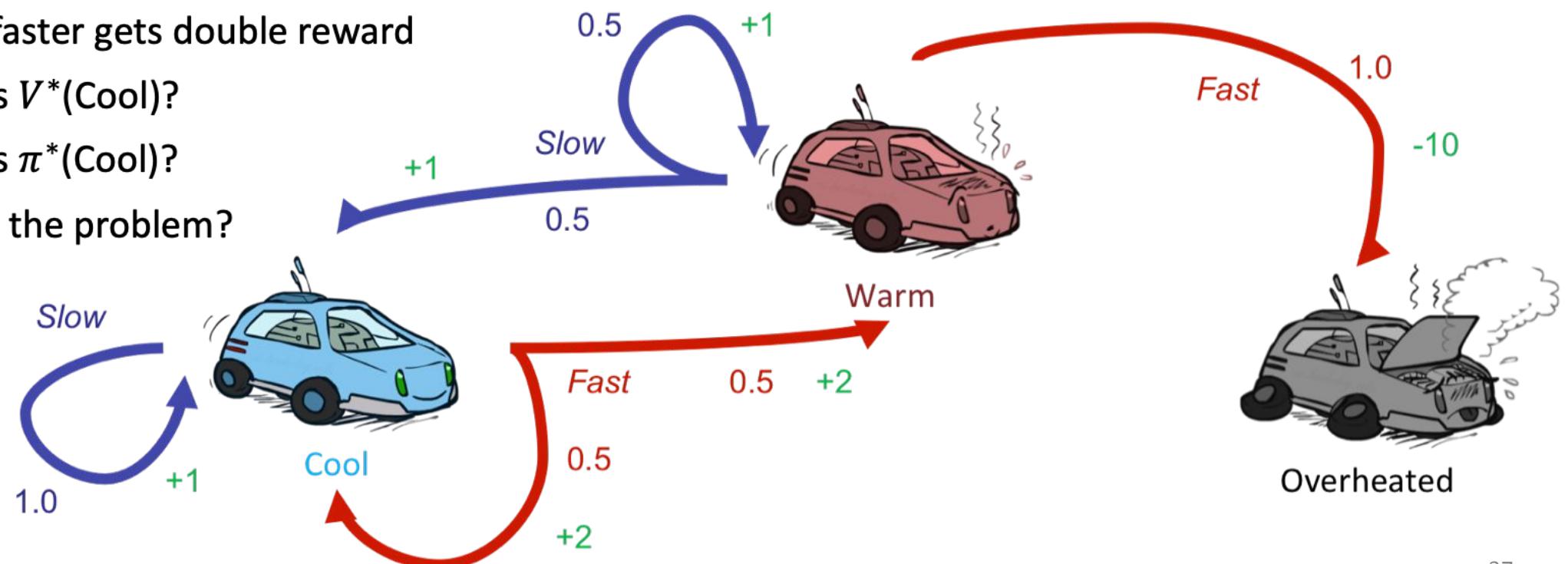
# Notation

- $\pi^*$  - 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  $\pi^*$ 
  - $= \mathbb{E}_{\pi^*} [G | s]$

# Race car example

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

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

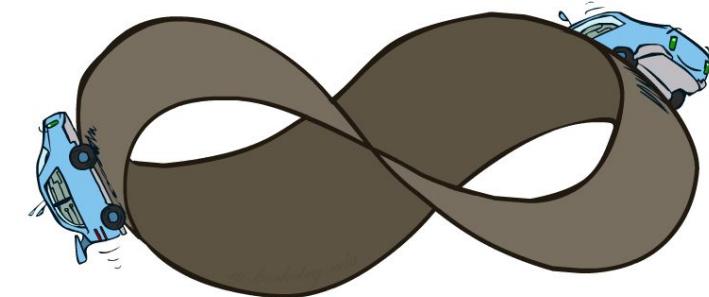


# Infinite Utilities?!

- 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 ( $\pi$  depends on time left)
  - Discounting: use  $0 < \gamma < 1$

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

- Smaller  $\gamma$  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?
    - $\tau_1 = +1, +1, +1, +1, +1, +1\dots$   $\sum_{t=0}^{\infty} 1 = \sum_{t=0}^{\infty} 2 = \infty$
    - $\tau_2 = +2, +2, +2, +2, +2, +2\dots$
  - Let's decay future rewards exponentially by a factor,  $0 \leq \gamma < 1$

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

- Now  $\tau_1$  yields higher utility than  $\tau_2$

# 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



$\gamma$

Worth Next Step

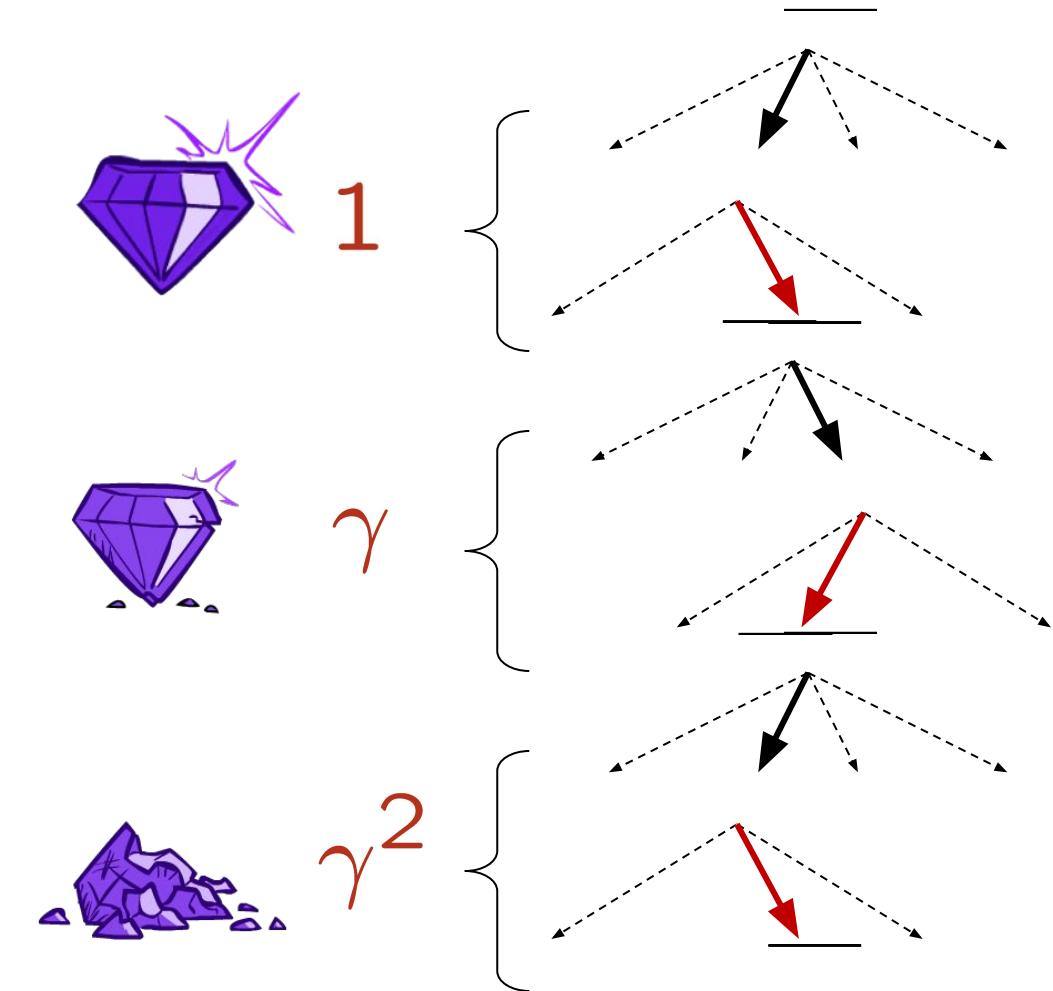


$\gamma^2$

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
  - $G(r=[1,2,3]) = 1*1 + 0.5*2 + 0.25*3$
  - $G([1,2,3]) < G([3,2,1])$

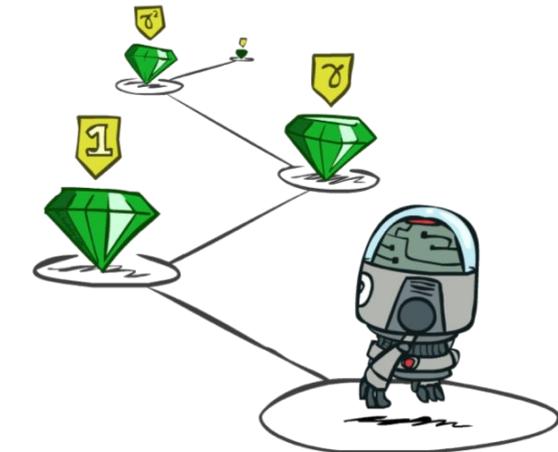


# Quiz: Discounting

- Given grid world:

|    |   |   |   |   |
|----|---|---|---|---|
| 10 |   |   |   | 1 |
| a  | b | c | d | e |

- 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  $\gamma = 1$ , what is the optimal policy?
- Quiz 2: For  $\gamma = 0.1$ , what is the optimal policy?
- Quiz 3: For which  $\gamma$  are West and East equally good when in state d?

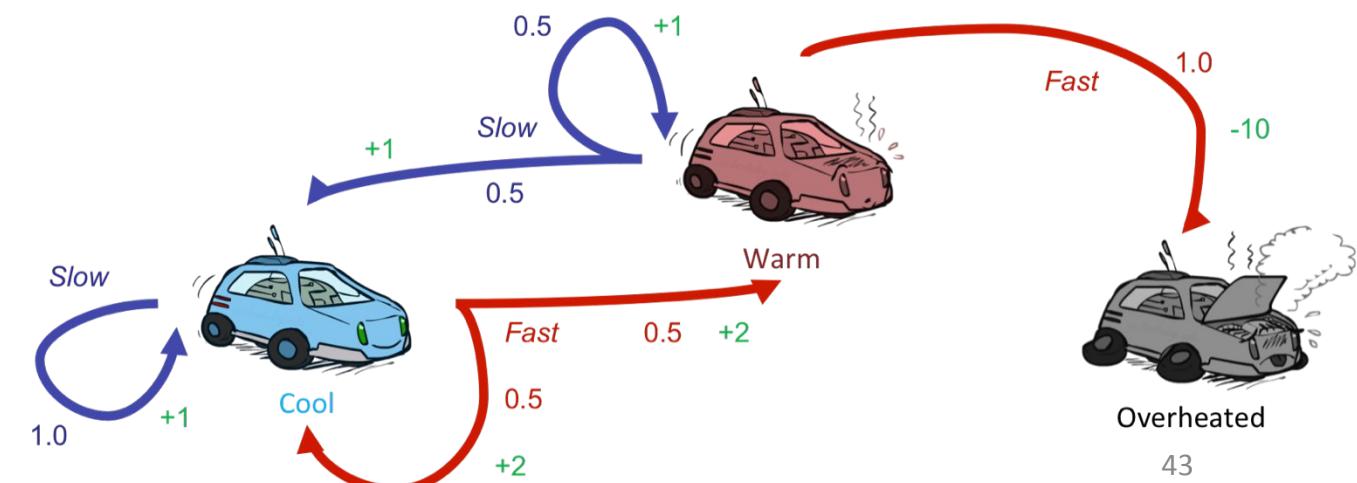


|    |   |   |   |   |
|----|---|---|---|---|
| 10 |   |   |   | 1 |
| a  | b | c | d | e |

|    |   |   |   |   |
|----|---|---|---|---|
| 10 |   |   |   | 1 |
| a  | b | c | d | e |

# Race car example

- Consider a discount factor,  $\gamma = 0.9$
- What is  $v^*(Cool)$
- $= \max_a [r(s, a) + \sum_{s'} p(s'|s, a) \gamma v^*(s')]$
- $= \max[1 + 0.9 \cdot 1v^*(Cool), 2 + 0.9 \cdot 0.5v^*(Cool) + 0.9 \cdot 0.5v^*(Warm)]$ 
  - 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  $s \in \mathcal{S}$ :

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

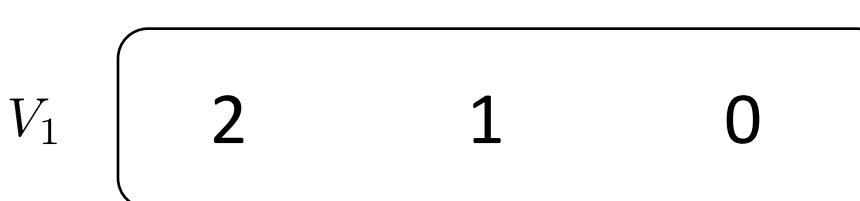
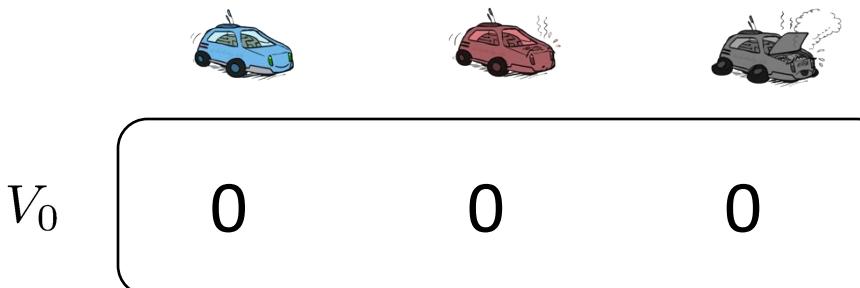
$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until  $\Delta < \theta$  (a small positive number)

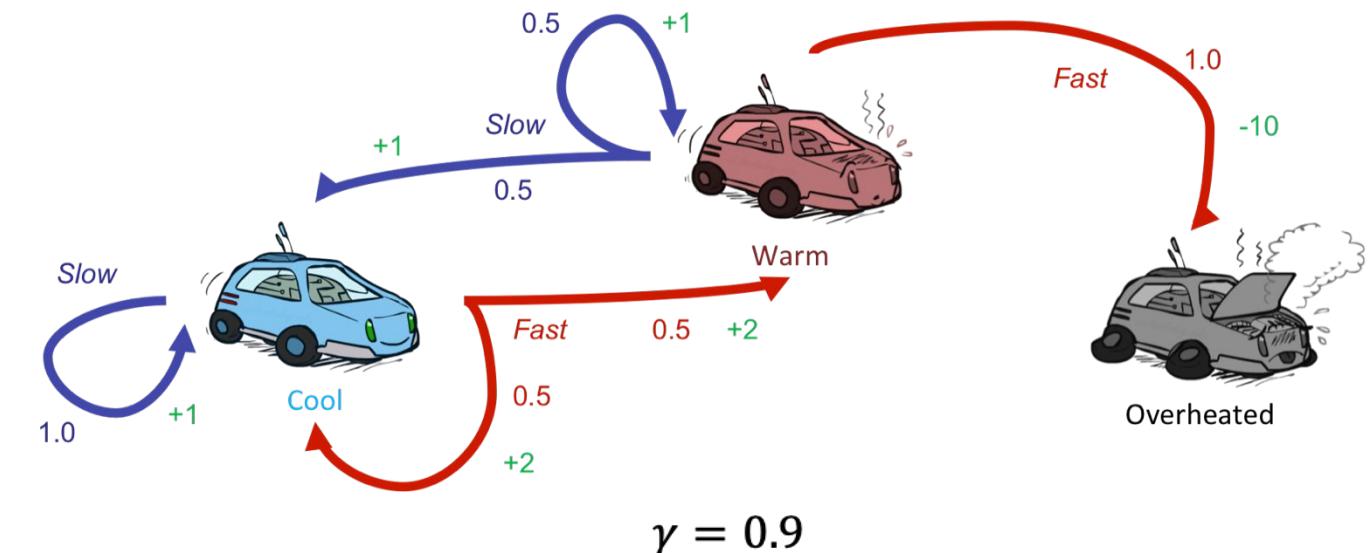
Output a deterministic policy,  $\pi \approx \pi_*$ , such that

$$\pi(s) = \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

# Value Iteration



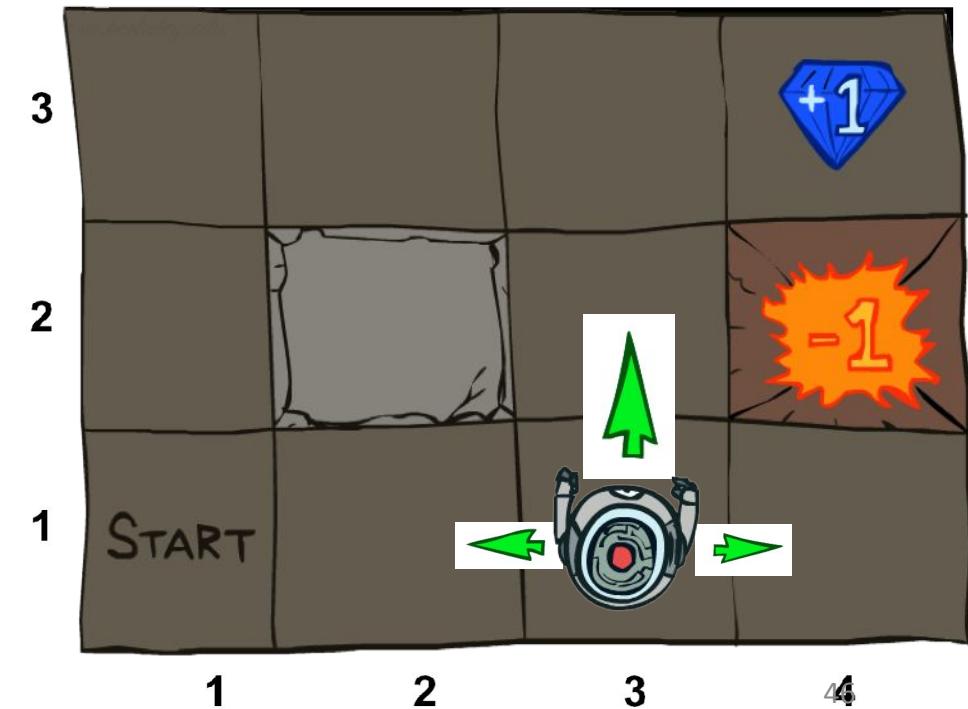
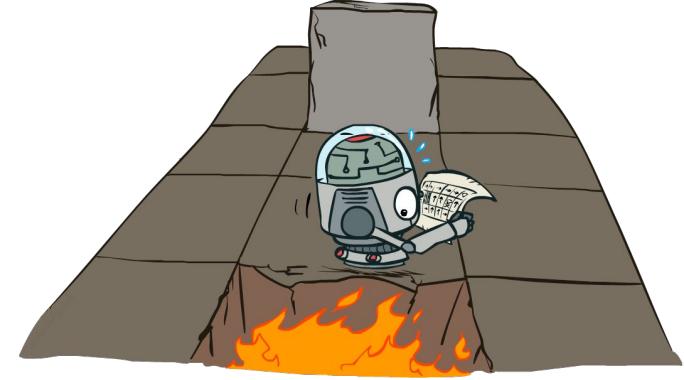
Check this computation on paper.



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

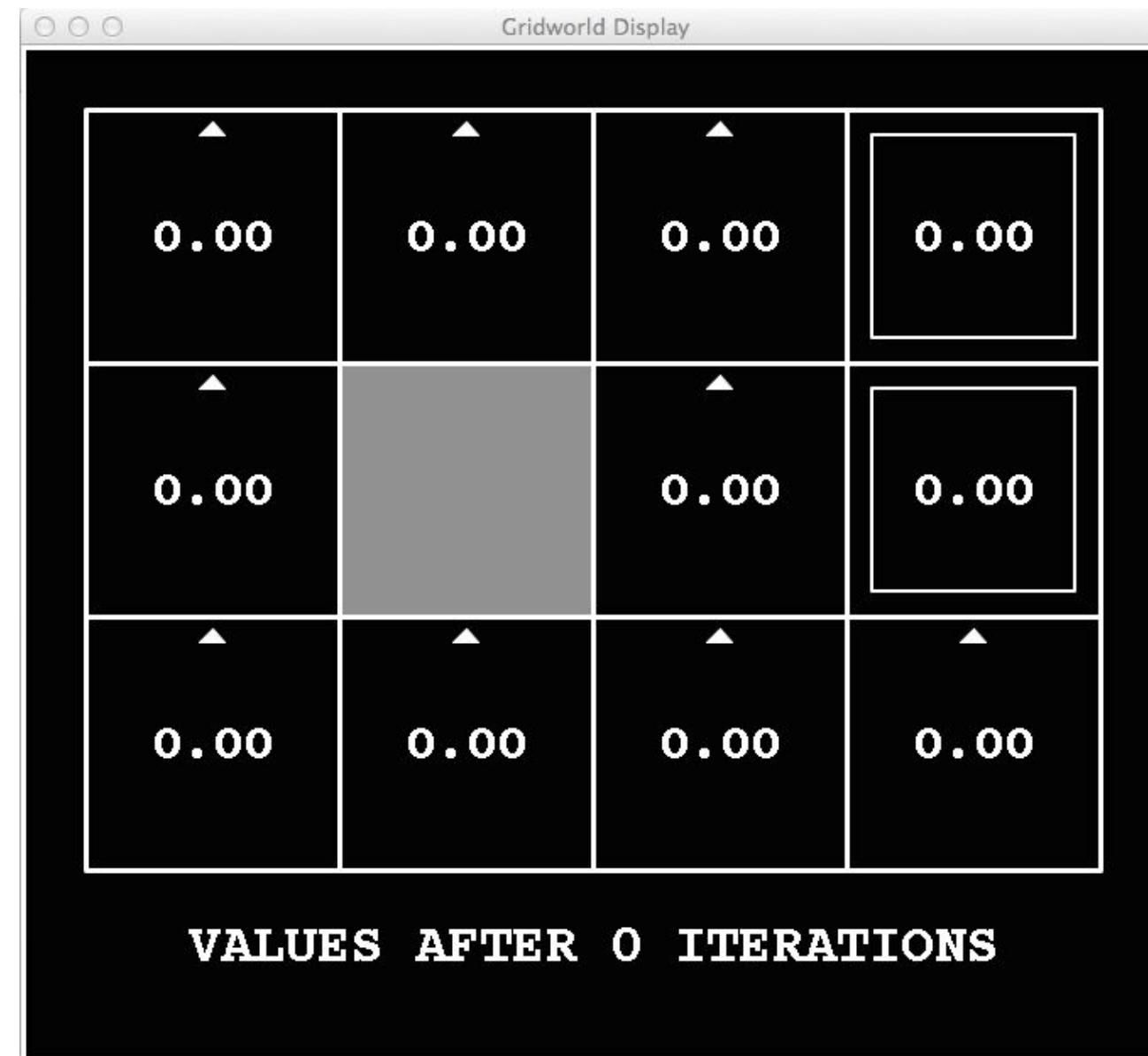
# Example: Grid World

- 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



k=0

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



k=1

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



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

k=2



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

k=3



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

k=4



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

k=5



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

k=6



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

k=7



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

k=8



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

k=9



k=10

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



k=11

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



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

k=12



k=100

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



# Problems with Value Iteration

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

- 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

For the home assignment set:

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

# Issue 2: A policy cannot be easily extracted

- Given state values, what is the appropriate policy?
  - $\pi(s) \leftarrow \operatorname{argmax}_a [R(s, a) + \sum_{s'} P(s'|s, a) \gamma V_k(s')]$
  - 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  $\pi^*$



# Q-learning

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



# Q-learning as value iteration

- $V^*(s) := \max_a [\sum_{s'} P(s'|s, a) (R(s, a, s') + \gamma V^*(s'))]$
- $V^*(s) := \max_a [Q^*(s, a)]$
- $Q^*(s, a) := \sum_{s'} P(s'|s, a) (R(s, a, s') + \gamma V^*(s'))$
- $Q^*(s, a) = \sum_{s'} P(s'|s, a) (R(s, a, s') + \gamma \max_a [Q^*(s', a)])$
- 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

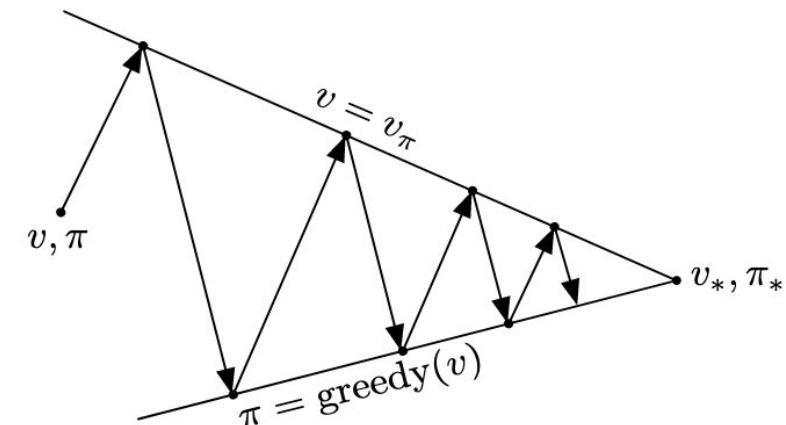
# Issue 3: The policy often converges long before the values

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



# Policy Iteration

1. **Compute  $V_\pi$ :** calculate state value for some fixed policy (not necessarily the optimal values,  $V_\pi \neq V^*$ )
2. **Update  $\pi$ :** 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  $\pi^*$ 
    - $\forall s, V_{k>0}(s) \leq V_{k+1}(s)$  i.e.,  $\pi_i$  improves monotonically with  $i$
    - A fixed point,  $\forall s, V_k(s) = V_{k+1}(s)$ , implies  $\pi^*$

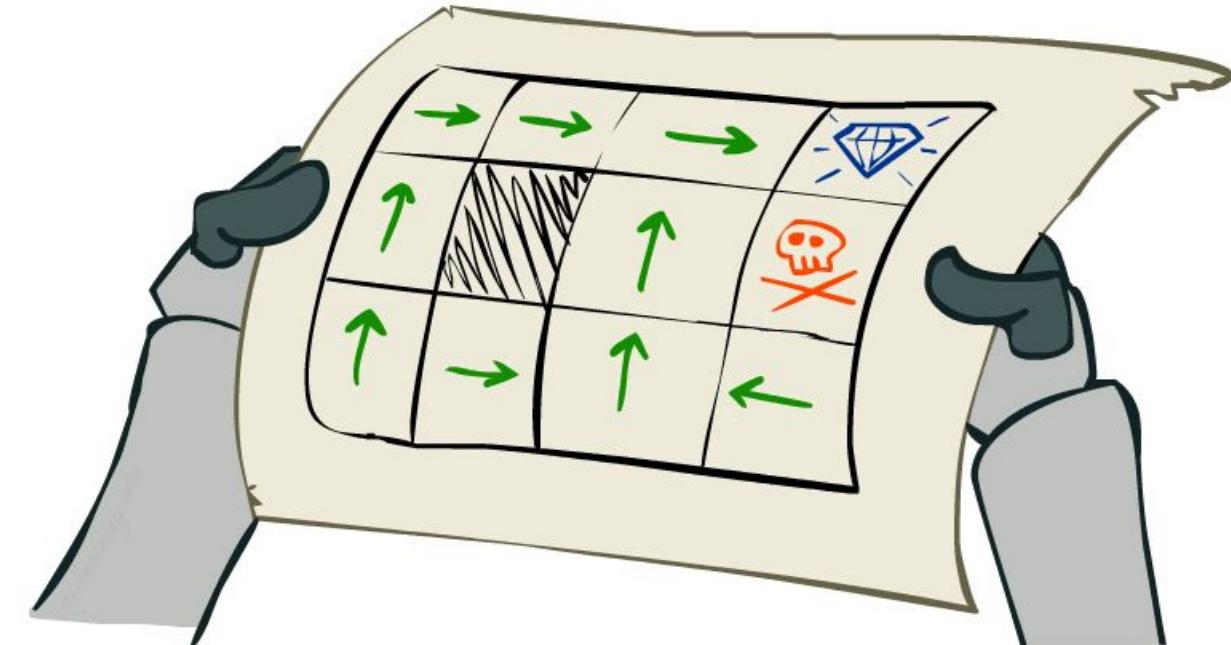


# Policy Evaluation

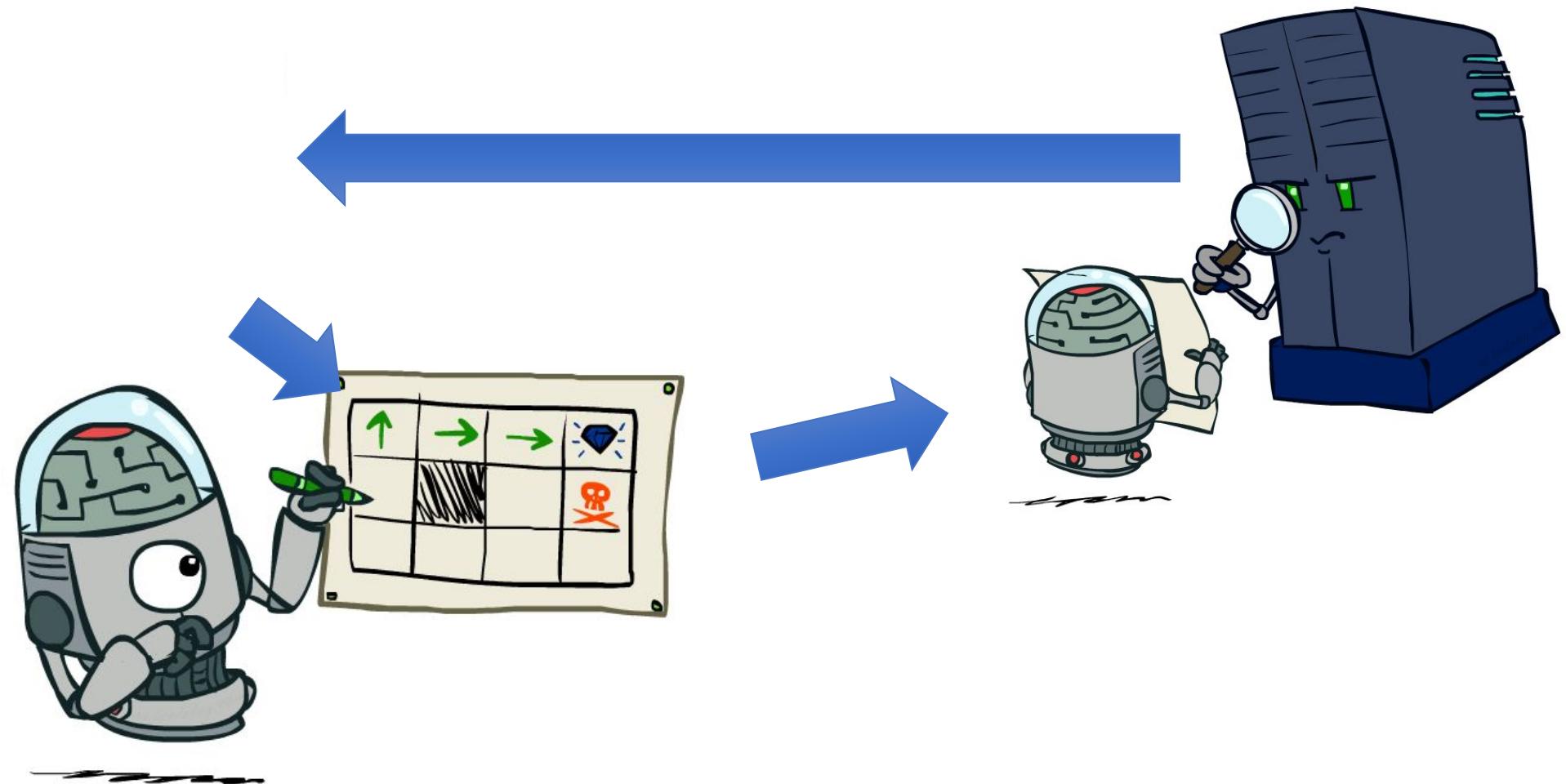
- Why is calculating  $V_\pi$  easier than calculating  $V^*$ ?
  - Turns non-linear Bellman equations into linear equations
- $v^*(s) = \max_a [\sum_{s'} P(s'|s, a) (R(s, a, s') + \gamma v^*(s'))]$
- $v_\pi(s) = \sum_{s'} P(s'|s, \pi(s)) (R(s, \pi(s), s') + \gamma v^*(s'))$
- Solve a set of linear equations 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



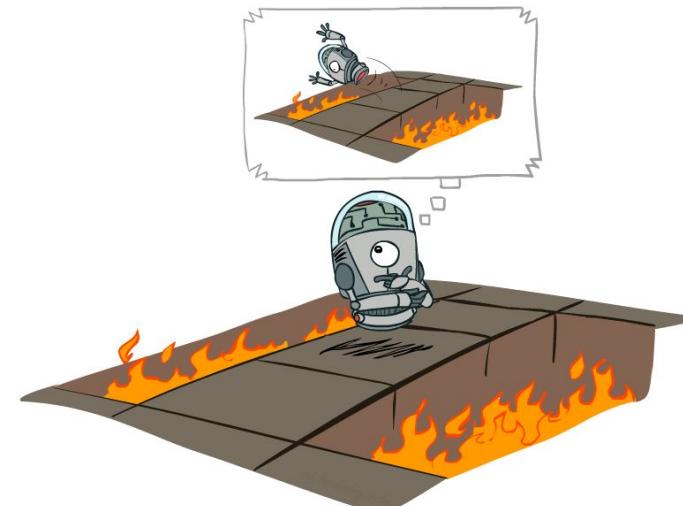


# Comparison

- 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 continuous action space as a distribution over the discrete 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

- $\pi^*$  - a policy that yields the maximal expected sum of rewards
- $V^*(s)$  - the expected sum of rewards from being at  $s$  then following  $\pi^*$
- $V_\pi(s)$  - the expected sum of rewards from being at  $s$  then following  $\pi$
- $Q^*(s, a)$  - the expected sum of rewards from being at  $s$ , taking action  $a$  and then following  $\pi^*$
- $Q_\pi(s, a)$  - the expected sum of rewards from being at  $s$ , taking action  $a$  and then following  $\pi$
- $G_t$  - 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, 2<sup>nd</sup> Ed., Sutton & Barto



**BITS Pilani**  
Pilani | Dubai | Goa | Hyderabad

Thank you



## Session #6-7: Monte Carlo Methods

### Instructors :

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

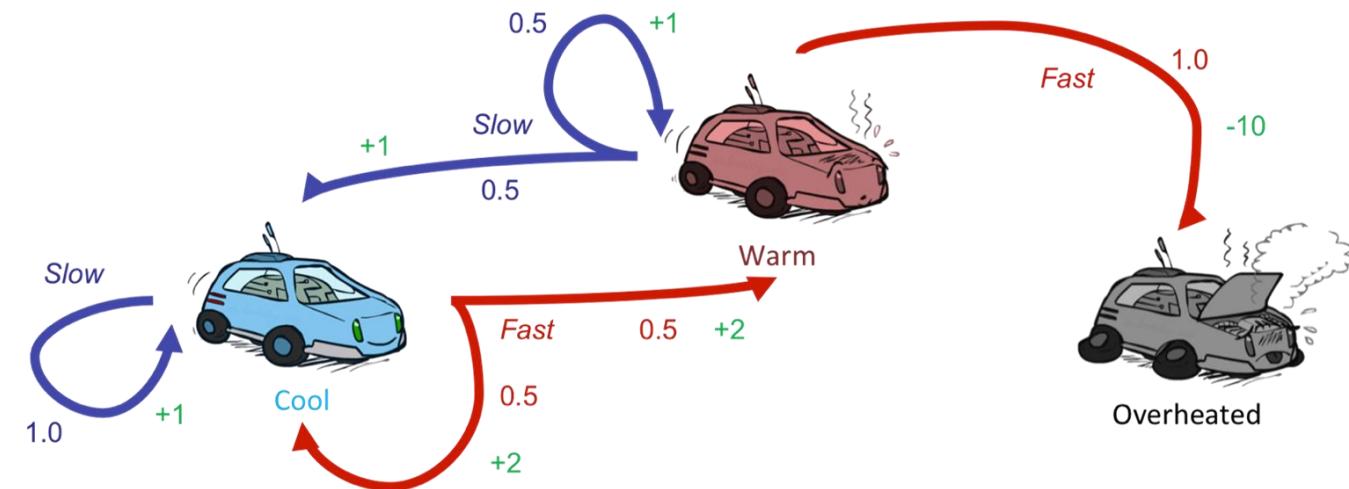


# Agenda for the class

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

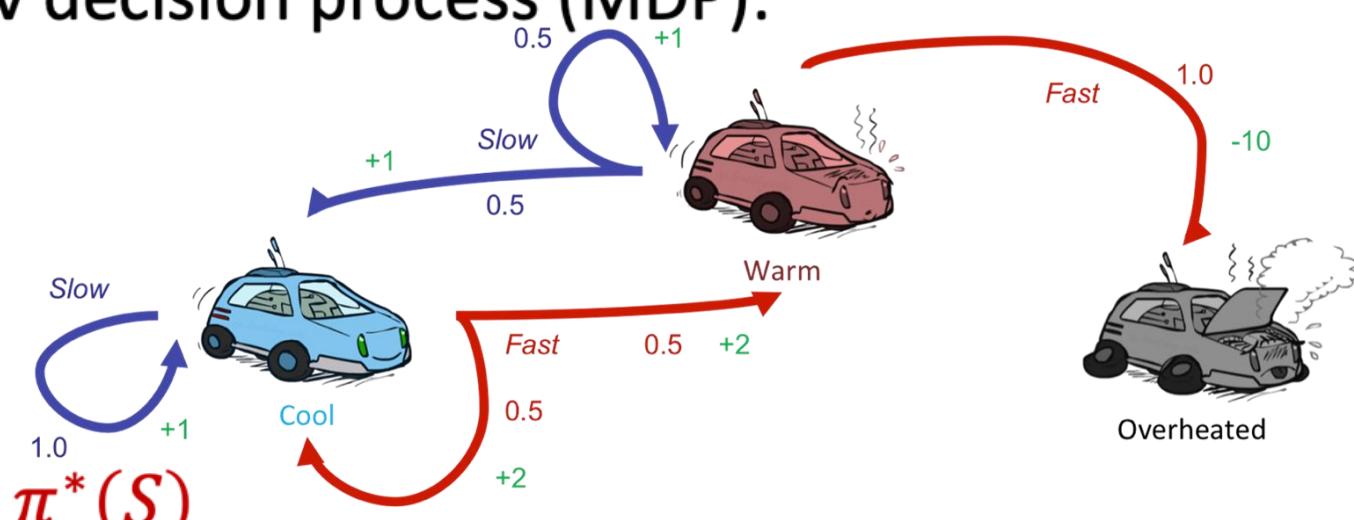
# Introduction

- 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  $s \in S$
  - A set of actions  $A$
  - A model  $P(s'|s, a)$
  - A reward function  $R(s, a, s')$
  - A discount factor  $\gamma$
  - Still looking for the best policy  $\pi^*(S)$



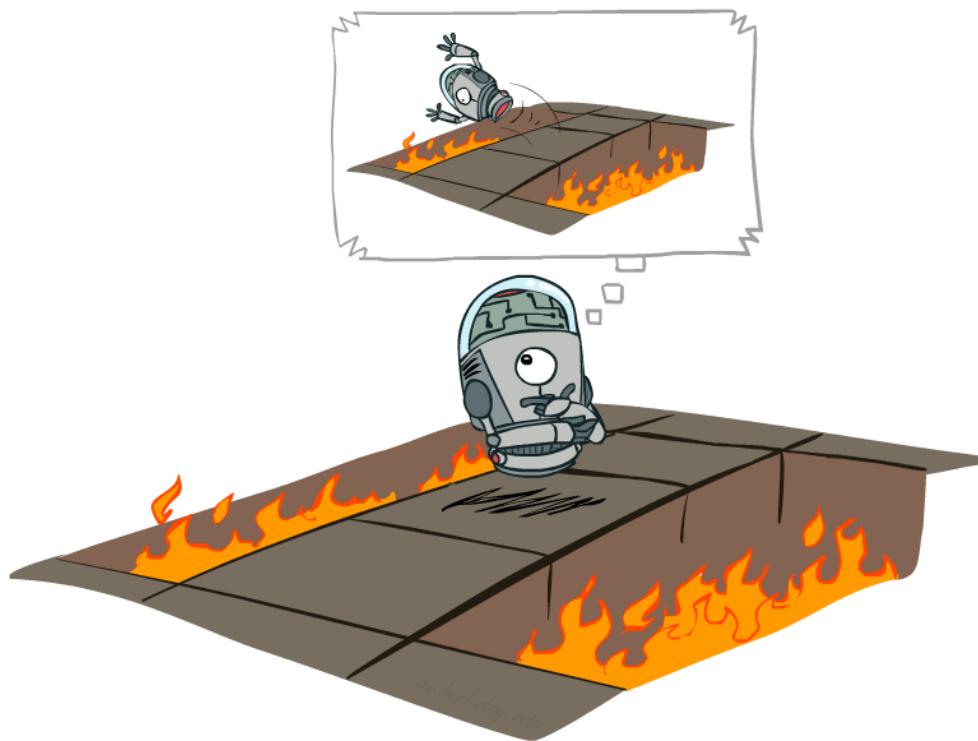
# Introduction

- Still assume an underlying Markov decision process (MDP):
  - A **set of states**  $s \in S$
  - A **set of actions**  $A$
  - A **model**  $P(s'|s, a)$
  - A **reward function**  $R(s, a, s')$
  - A discount factor  $\gamma$
  - 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  $\pi$

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

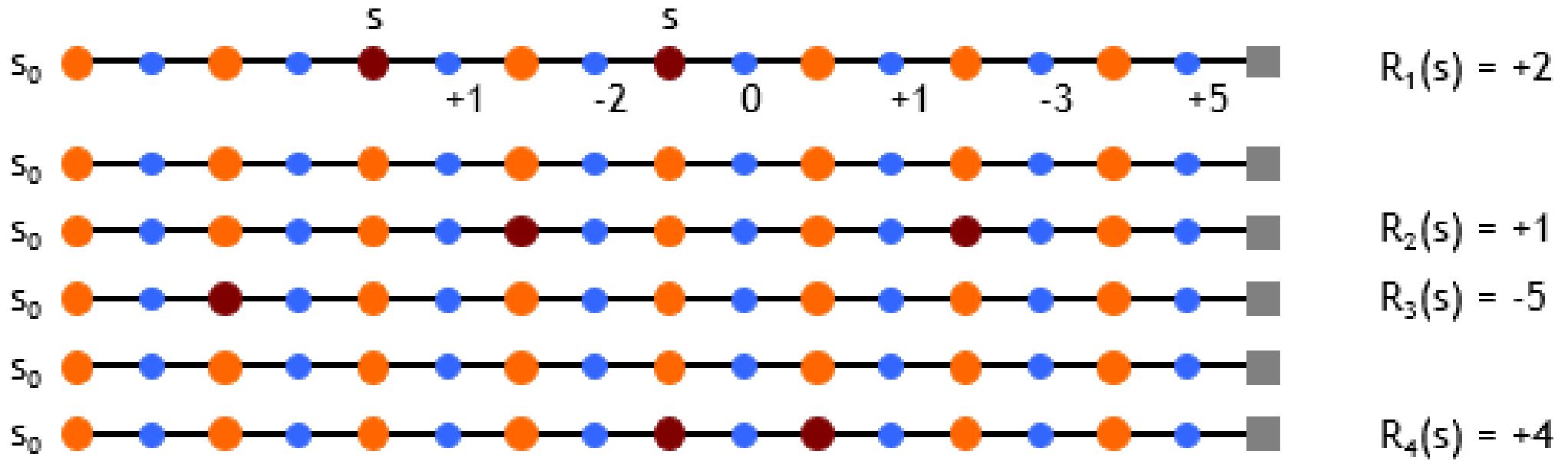
$R \leftarrow$  return following the first occurrence of  $s$

Append  $R$  to  $Returns(s)$

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

# Ex-1:First-visit Monte-Carlo Policy Evaluation

## [estimate $V^\pi(s)$ ]

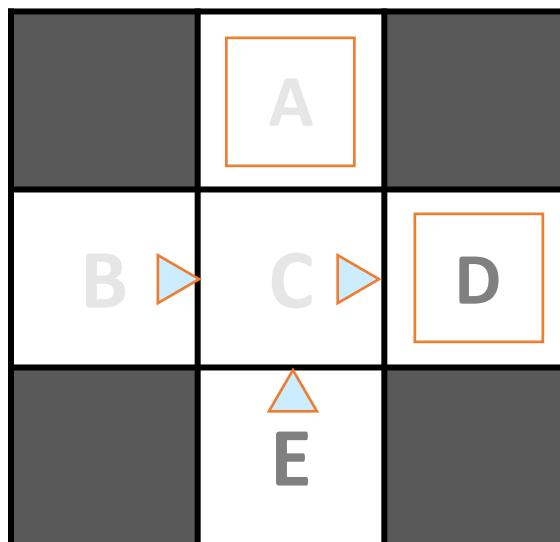


$$V^\pi(s) \approx (2 + 1 - 5 + 4)/4 = 0.5$$

# Ex-2: First-visit Monte-Carlo Policy Evaluation

## [estimate $V\pi(s)$ ]

Input Policy  $\pi$



Assume:  $\gamma = 1$

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

|   |     |    |
|---|-----|----|
|   | -10 |    |
| A | +8  | +4 |
| B | C   | D  |
|   | -2  |    |
| E |     |    |

# 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

|         |          |          |
|---------|----------|----------|
|         | -10<br>A |          |
| +8<br>B | +4<br>C  | +10<br>D |
|         | -2<br>E  |          |

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

# Must explore!

- Hard policy (insufficient):  $\pi(s) = a$ ,  $\pi: S \rightarrow \mathcal{A}$
- Soft policy:  $\pi(a|s) = [0,1]$ ,  $\pi: S \times \mathcal{A} \rightarrow p$ 
  - 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  $\varepsilon$ 
  - $\forall a \neq A^*$ ,  $\pi(s, a) = \frac{\varepsilon}{|\mathcal{A}(s)|}$
  - Else select the greedy action:  $\pi(s, A^*) = 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|}$

# $\varepsilon$ -greedy MC control

**On-policy first-visit MC control (for  $\varepsilon$ -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  $\varepsilon$ -soft policy

Repeat forever:

(a) Generate an episode using  $\pi$

(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  $a \in \mathcal{A}(s)$ :

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

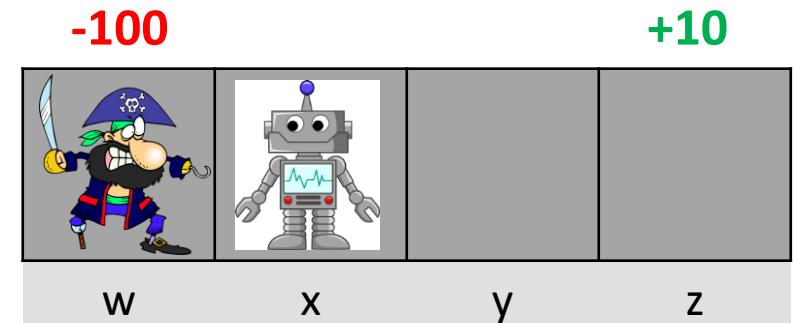
$$\gamma = 0.9$$

# MC control - example

- $Q = \begin{array}{|c|c|c|c|} \hline 5 & 4,3 & 2,1 & 0 \\ \hline w & x & y & z \\ \hline \end{array}$

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

- $\pi(a|s) = (1 - \varepsilon) \cdot \begin{array}{|c|c|c|c|} \hline \text{exit} & & & \text{exit} \\ \hline w & x & y & z \\ \hline \end{array}$
- $\varepsilon \cdot \text{Random}$



On-policy first-visit MC control (for  $\varepsilon$ -soft policies),

Initialize, for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ :

$Q(s, a) \leftarrow \text{arbitrary}$

$Returns(s, a) \leftarrow \text{empty list}$

$\pi(a|s) \leftarrow \text{an arbitrary } \varepsilon\text{-soft policy}$

Repeat forever:

(a) Generate an episode using  $\pi$

(b) For each pair  $s, a$  appearing in the episode:

$G \leftarrow \text{the return that follows the first occurrence of } s, a$

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 tie-breaking rule)

For all  $a \in \mathcal{A}(s)$ :

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

$$\gamma = 0.9$$

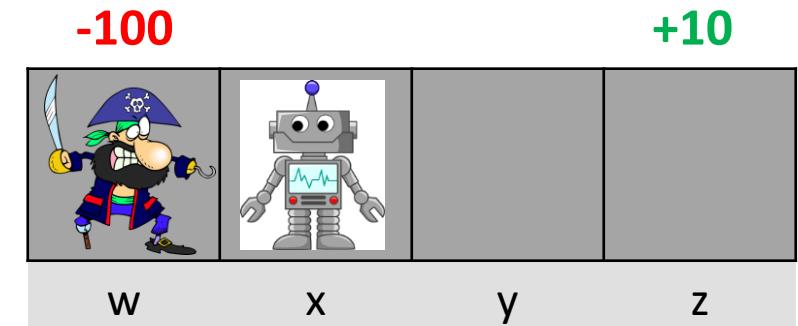
# MC control - example

- $Q = \begin{array}{|c|c|c|c|} \hline 5 & 4,3 & 2,1 & 0 \\ \hline w & x & y & z \\ \hline \end{array}$

- $\overline{Returns} = \begin{array}{|c|c|c|c|} \hline - & -, - & -, - & - \\ \hline w & x & y & z \\ \hline \end{array}$

- $\pi(a|s) = (1 - \varepsilon) \cdot \begin{array}{|c|c|c|c|} \hline \text{exit} & & & \text{exit} \\ \hline w & x & y & z \\ \hline \end{array}$
- $\varepsilon \cdot \text{Random}$

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



## On-policy first-visit MC control (for $\varepsilon$ -soft policies),

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :

$$Q(s, a) \leftarrow \text{arbitrary}$$

$$Returns(s, a) \leftarrow \text{empty list}$$

$$\pi(a|s) \leftarrow \text{an arbitrary } \varepsilon\text{-soft policy}$$

Repeat forever:

(a) Generate an episode using  $\pi$

(b) For each pair  $s, a$  appearing in the episode:

$$G \leftarrow \text{the return that follows the first occurrence of } s, a$$

Append  $G$  to  $Returns(s, a)$

$$Q(s, a) \leftarrow \text{average}(Returns(s, a))$$

(c) For each  $s$  in the episode:

$$A^* \leftarrow \arg \max_a Q(s, a) \quad (\text{with tie-breaking rule})$$

For all  $a \in \mathcal{A}(s)$ :

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

$$\gamma = 0.9$$

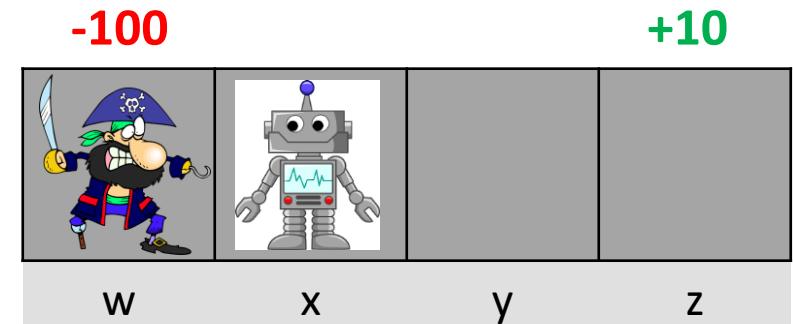
# MC control - example

- $Q = \begin{array}{|c|c|c|c|} \hline 5 & 4,3 & 2,1 & 0 \\ \hline w & x & y & z \\ \hline \end{array}$

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

- $\pi(a|s) = (1 - \varepsilon) \cdot \begin{array}{|c|c|c|c|} \hline \text{exit} & & & \text{exit} \\ \hline w & x & y & z \\ \hline \end{array}$   
  - $\varepsilon \cdot \text{Random}$

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



## On-policy first-visit MC control (for $\varepsilon$ -soft policies),

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :

$Q(s, a) \leftarrow \text{arbitrary}$

$Returns(s, a) \leftarrow \text{empty list}$

$\pi(a|s) \leftarrow \text{an arbitrary } \varepsilon\text{-soft policy}$

Repeat forever:

(a) Generate an episode using  $\pi$

(b) For each pair  $s, a$  appearing in the episode:

$G \leftarrow \text{the return that follows the first occurrence of } s, a$

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 tie-breaking rule)

For all  $a \in \mathcal{A}(s)$ :

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

$$\gamma = 0.9$$

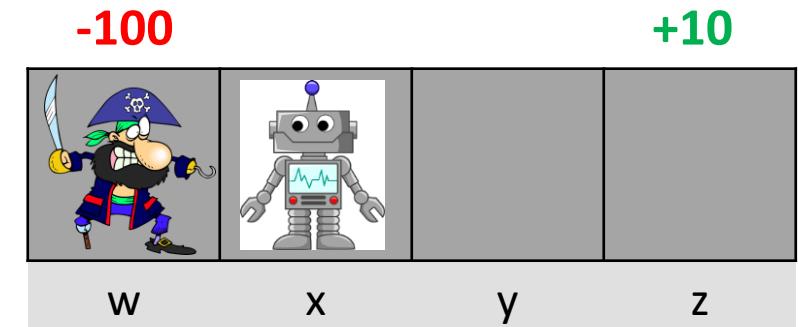
# MC control - example

- $Q = \begin{array}{|c|c|c|c|} \hline -100 & -90,3 & 2,1 & 0 \\ \hline w & x & y & z \\ \hline \end{array}$

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

- $\pi(a|s) = (1 - \varepsilon) \cdot \begin{array}{|c|c|c|c|} \hline \text{exit} & & & \text{exit} \\ \hline w & x & y & z \\ \hline \end{array}$   
  - $\varepsilon \cdot \text{Random}$

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



## On-policy first-visit MC control (for $\varepsilon$ -soft policies),

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :

$Q(s, a) \leftarrow \text{arbitrary}$

$Returns(s, a) \leftarrow \text{empty list}$

$\pi(a|s) \leftarrow \text{an arbitrary } \varepsilon\text{-soft policy}$

Repeat forever:

(a) Generate an episode using  $\pi$

(b) For each pair  $s, a$  appearing in the episode:

$G \leftarrow \text{the return that follows the first occurrence of } s, a$

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 tie)

For all  $a \in \mathcal{A}(s)$ :

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

$$\gamma = 0.9$$

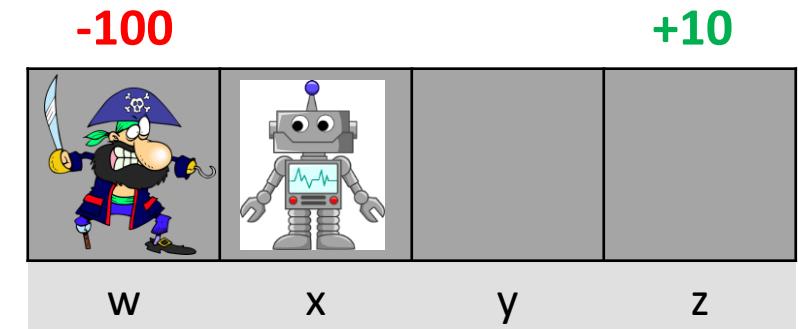
# MC control - example

- $Q = \begin{array}{|c|c|c|c|} \hline -100 & -90,3 & 2,1 & 0 \\ \hline w & x & y & z \\ \hline \end{array}$

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

- $\pi(a|s) = (1 - \varepsilon) \cdot \begin{array}{|c|c|c|c|} \hline \text{exit} & & & \text{exit} \\ \hline w & x & y & z \\ \hline \end{array}$

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



On-policy first-visit MC control (for  $\varepsilon$ -soft policies),

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :

$Q(s, a) \leftarrow \text{arbitrary}$

$Returns(s, a) \leftarrow \text{empty list}$

$\pi(a|s) \leftarrow \text{an arbitrary } \varepsilon\text{-soft policy}$

Repeat forever:

(a) Generate an episode using  $\pi$

(b) For each pair  $s, a$  appearing in the episode:

$G \leftarrow \text{the return that follows the first occurrence of } s, a$

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 tie-breaking)

For all  $a \in \mathcal{A}(s)$ :

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

$$\gamma = 0.9$$

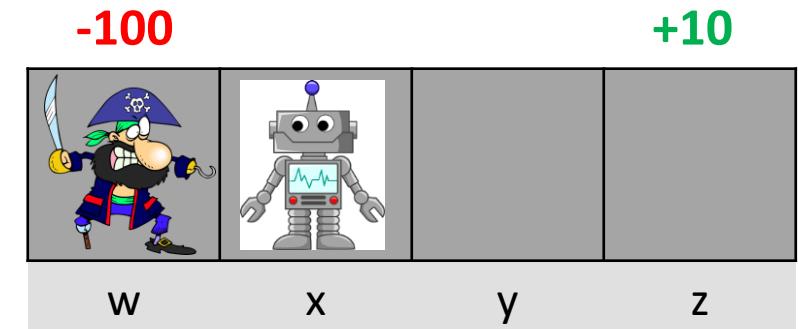
# MC control - example

- $Q = \begin{array}{|c|c|c|c|} \hline -100 & -90,3 & 2,1 & 0 \\ \hline w & x & y & z \\ \hline \end{array}$

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

- $\pi(a|s) = (1 - \varepsilon) \cdot \begin{array}{|c|c|c|c|} \hline \text{exit} & & & \text{exit} \\ \hline w & x & y & z \\ \hline \end{array}$   
  - $\varepsilon \cdot \text{Random}$

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



On-policy first-visit MC control (for  $\varepsilon$ -soft policies),

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :

$Q(s, a) \leftarrow \text{arbitrary}$

$Returns(s, a) \leftarrow \text{empty list}$

$\pi(a|s) \leftarrow \text{an arbitrary } \varepsilon\text{-soft policy}$

Repeat forever:

(a) Generate an episode using  $\pi$

(b) For each pair  $s, a$  appearing in the episode:

$G \leftarrow \text{the return that follows the first occurrence of } s, a$

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 tie)

For all  $a \in \mathcal{A}(s)$ :

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

$$\gamma = 0.9$$

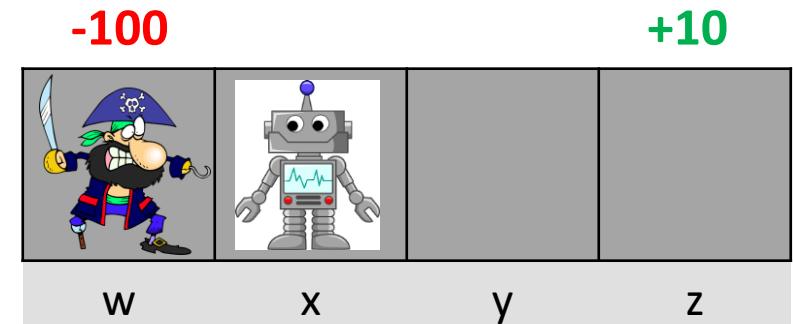
# MC control - example

- $Q = \begin{array}{|c|c|c|c|} \hline -100 & -90,3 & 2,1 & 0 \\ \hline w & x & y & z \\ \hline \end{array}$

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

- $\pi(a|s) = (1 - \varepsilon) \cdot \begin{array}{|c|c|c|c|} \hline \text{exit} & & & \text{exit} \\ \hline w & x & y & z \\ \hline \end{array}$   
  - $\varepsilon \cdot \text{Random}$

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



## On-policy first-visit MC control (for $\varepsilon$ -soft policies),

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :

$$Q(s, a) \leftarrow \text{arbitrary}$$

$$Returns(s, a) \leftarrow \text{empty list}$$

$$\pi(a|s) \leftarrow \text{an arbitrary } \varepsilon\text{-soft policy}$$

Repeat forever:

(a) Generate an episode using  $\pi$

(b) For each pair  $s, a$  appearing in the episode:

$$G \leftarrow \text{the return that follows the first occurrence of } s, a$$

Append  $G$  to  $Returns(s, a)$

$$Q(s, a) \leftarrow \text{average}(Returns(s, a))$$

(c) For each  $s$  in the episode:

$$A^* \leftarrow \arg \max_a Q(s, a) \quad (\text{with tie-breaking rule})$$

For all  $a \in \mathcal{A}(s)$ :

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

$$\gamma = 0.9$$

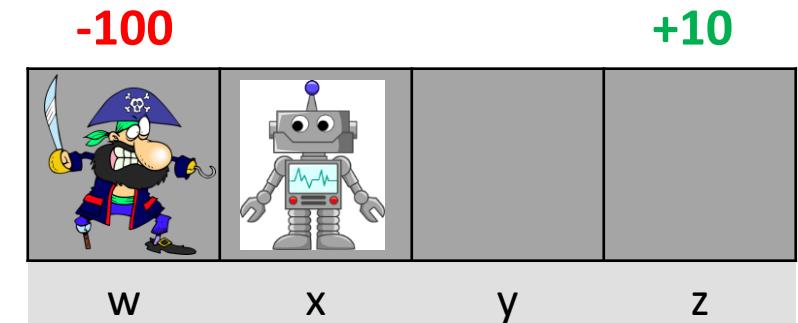
# MC control - example

- $Q = \begin{array}{|c|c|c|c|} \hline -100 & -90, -72.9 & -81, 1 & 0 \\ \hline w & x & y & z \\ \hline \end{array}$

- $\overline{Returns} = \begin{array}{|c|c|c|c|} \hline -100 & -90, -72.9 & -81, - & - \\ \hline w & x & y & z \\ \hline \end{array}$

- $\pi(a|s) = (1 - \varepsilon) \cdot \begin{array}{|c|c|c|c|} \hline \text{exit} & & & \text{exit} \\ \hline w & x & y & z \\ \hline \end{array}$   
  - $\varepsilon \cdot \text{Random}$

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



## On-policy first-visit MC control (for $\varepsilon$ -soft policies),

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :

$Q(s, a) \leftarrow \text{arbitrary}$

$Returns(s, a) \leftarrow \text{empty list}$

$\pi(a|s) \leftarrow \text{an arbitrary } \varepsilon\text{-soft policy}$

Repeat forever:

(a) Generate an episode using  $\pi$

(b) For each pair  $s, a$  appearing in the episode:

$G \leftarrow \text{the return that follows the first occurrence of } s, a$

Append  $G$  to  $Returns(s, a)$

$Q(s, a) \leftarrow \text{average}(Returns(s, a))$

(c) For each  $s$  in the episode:

$A^* \leftarrow \arg \max_a Q(s, a)$

For all  $a \in \mathcal{A}(s)$ :

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

(with tie

$$\gamma = 0.9$$

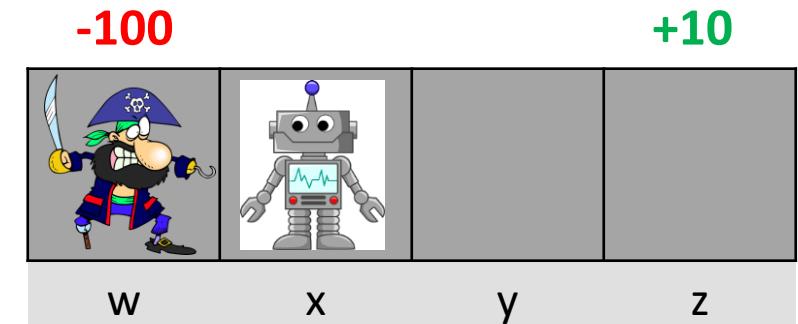
# MC control - example

- $Q = \begin{array}{|c|c|c|c|} \hline -100 & -90, -72.9 & -81, 1 & 0 \\ \hline w & x & y & z \\ \hline \end{array}$

- $\overline{Returns} = \begin{array}{|c|c|c|c|} \hline -100 & -90, -72.9 & -81, - & - \\ \hline w & x & y & z \\ \hline \end{array}$

- $\pi(a|s) = (1 - \varepsilon) \cdot \begin{array}{|c|c|c|c|} \hline \text{exit} & & & \text{exit} \\ \hline w & x & y & z \\ \hline \end{array}$   
  - $\varepsilon \cdot \text{Random}$

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



## On-policy first-visit MC control (for $\varepsilon$ -soft policies),

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :

$Q(s, a) \leftarrow \text{arbitrary}$

$Returns(s, a) \leftarrow \text{empty list}$

$\pi(a|s) \leftarrow \text{an arbitrary } \varepsilon\text{-soft policy}$

Repeat forever:

(a) Generate an episode using  $\pi$

(b) For each pair  $s, a$  appearing in the episode:

$G \leftarrow \text{the return that follows the first occurrence of } s, a$

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 tie-breaking rule)

For all  $a \in \mathcal{A}(s)$ :

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

$$\gamma = 0.9$$

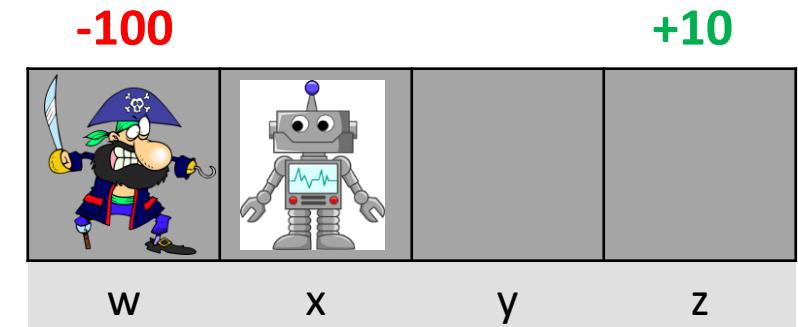
# MC control - example

- $Q = \begin{array}{|c|c|c|c|} \hline -100 & -90, -72.9 & -81, 1 & 0 \\ \hline w & x & y & z \\ \hline \end{array}$

- $\overline{Returns} = \begin{array}{|c|c|c|c|} \hline -100 & -90, -72.9 & -81, - & - \\ \hline w & x & y & z \\ \hline \end{array}$

- $\pi(a|s) = (1 - \varepsilon) \cdot \begin{array}{|c|c|c|c|} \hline \text{exit} & & & \text{exit} \\ \hline w & x & y & z \\ \hline \end{array}$   
  - $\varepsilon \cdot \text{Random}$

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



## On-policy first-visit MC control (for $\varepsilon$ -soft policies),

Initialize, for all  $s \in \mathcal{S}$ ,  $a \in \mathcal{A}(s)$ :

$Q(s, a) \leftarrow \text{arbitrary}$

$Returns(s, a) \leftarrow \text{empty list}$

$\pi(a|s) \leftarrow \text{an arbitrary } \varepsilon\text{-soft policy}$

Repeat forever:

(a) Generate an episode using  $\pi$

(b) For each pair  $s, a$  appearing in the episode:

$G \leftarrow \text{the return that follows the first occurrence of } s, a$

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 tie-breaking rule)

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, 2<sup>nd</sup> 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 #9: Temporal Difference Learning



# Agenda for the class

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

# Solving MDPs so far

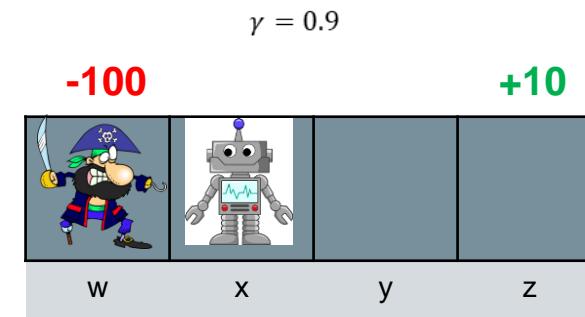
## Dynamic programming

- 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

- $Q(z, \text{exit}) = 10$
  - $Q(y, \rightarrow) = 0 + \gamma \max_a Q(z, a)$
  - $Q(x, \rightarrow) = 0 + \gamma \max_a Q(y, a)$
- $$q^*(s, a) = \sum_{s'} p(s'|s, a) (r(s, a, s') + \gamma \max_a [q^*(s', a)])$$



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

# Fuse DP and MC

## Dynamic programming

- Off policy
- local learning, propagating values from neighbors  
(Bootstrapping)
- Model based

## Monte-Carlo

- On-policy (though importance 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  
(Bootstrapping)
- Model free, online learning

# Temporal difference learning

- $Q(s, a) = Q(s, a) + \alpha \left( R_{t+1} + \gamma \max_{a'}[Q(s', a')] - Q(s, a) \right)$ 
  - $V(s) = V(s) + \alpha(R_{t+1} + \gamma V(s') - V(s))$
- Update estimate based on other estimates
- Model free
- Online, incremental learning
- Guaranteed to converge to the true value!
  - Some conditions on the step size,  $\alpha$  (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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

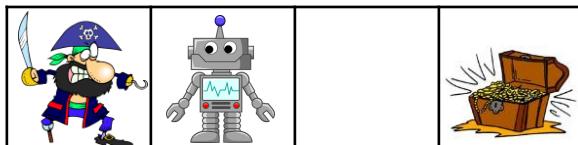
$$S \leftarrow S'; A \leftarrow A';$$

    until  $S$  is terminal

$$\gamma = 0.9$$

**-100**

**+10**



# 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

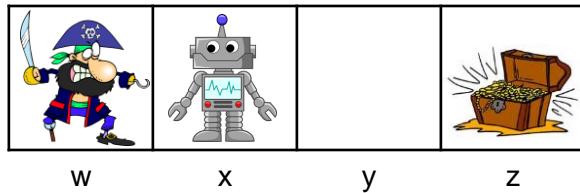
$$S \leftarrow S'; A \leftarrow A';$$

    until  $S$  is terminal

$$\gamma = 0.9$$

**-100**

**+10**



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

# 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

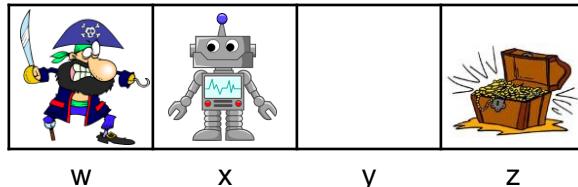
$$S \leftarrow S'; A \leftarrow A';$$

    until  $S$  is terminal

$$\gamma = 0.9$$

**-100**

**+10**



|     |              |          |     |             |
|-----|--------------|----------|-----|-------------|
| $x$ | $\leftarrow$ | <b>0</b> | $w$ | <b>null</b> |
| S   | A            | R        | S'  | A'          |

|          |            |            |          |
|----------|------------|------------|----------|
| <b>0</b> | <b>0,0</b> | <b>0,0</b> | <b>0</b> |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

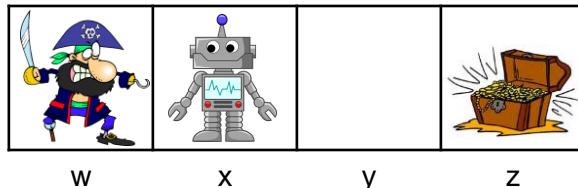
$$S \leftarrow S'; A \leftarrow A';$$

    until  $S$  is terminal

$$\gamma = 0.9$$

**-100**

**+10**



|   |   |   |    |      |
|---|---|---|----|------|
| x | ← | 0 | w  | exit |
| S | A | R | S' | A'   |

$Q =$

|   |     |     |   |
|---|-----|-----|---|
| 0 | 0,0 | 0,0 | 0 |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

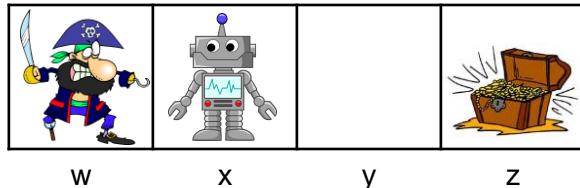
$$S \leftarrow S'; A \leftarrow A';$$

    until  $S$  is terminal

$$\gamma = 0.9$$

**-100**

**+10**



|     |              |          |     |             |
|-----|--------------|----------|-----|-------------|
| $x$ | $\leftarrow$ | <b>0</b> | $w$ | <i>exit</i> |
| S   | A            | R        | S'  | A'          |

$Q =$

|          |            |            |          |
|----------|------------|------------|----------|
| <b>0</b> | <b>0,0</b> | <b>0,0</b> | <b>0</b> |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

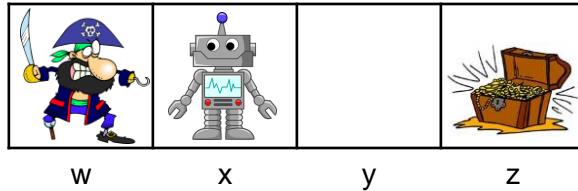
$$S \leftarrow S'; A \leftarrow A';$$

    until  $S$  is terminal

$$\gamma = 0.9$$

**-100**

**+10**



|   |      |   |    |      |
|---|------|---|----|------|
| w | exit | 0 | w  | exit |
| S | A    | R | S' | A'   |

$Q =$

|   |     |     |   |
|---|-----|-----|---|
| 0 | 0,0 | 0,0 | 0 |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

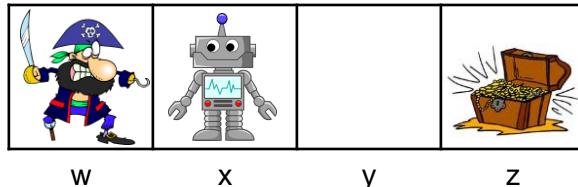
$$S \leftarrow S'; A \leftarrow A';$$

    until  $S$  is terminal

$$\gamma = 0.9$$

**-100**

**+10**



|   |      |      |     |      |
|---|------|------|-----|------|
| w | exit | -100 | ter | exit |
| S | A    | R    | S'  | A'   |

$$Q = \begin{matrix} & \boxed{0} & \boxed{0,0} & \boxed{0,0} & \boxed{0} \\ w & & x & & z \end{matrix}$$

# 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

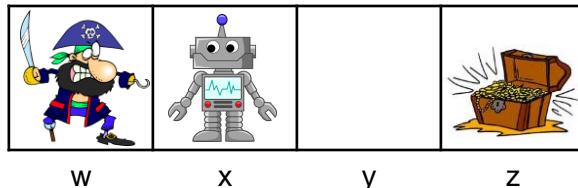
$$S \leftarrow S'; A \leftarrow A';$$

    until  $S$  is terminal

$$\gamma = 0.9$$

**-100**

**+10**



|   |      |      |     |    |
|---|------|------|-----|----|
| w | exit | -100 | ter | .  |
| S | A    | R    | S'  | A' |

$$Q = \begin{bmatrix} -100 & 0,0 & 0,0 & 0 \\ w & x & y & z \end{bmatrix}$$

# 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

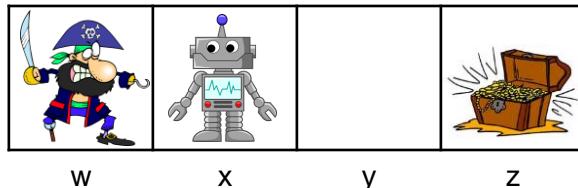
$$S \leftarrow S'; A \leftarrow A';$$

    until  $S$  is terminal

$$\gamma = 0.9$$

**-100**

**+10**



|     |              |             |            |      |
|-----|--------------|-------------|------------|------|
| $x$ | $\leftarrow$ | <b>-100</b> | <i>ter</i> | .    |
| S   | A            | R           | $S'$       | $A'$ |

|             |            |            |          |
|-------------|------------|------------|----------|
| <b>-100</b> | <b>0,0</b> | <b>0,0</b> | <b>0</b> |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

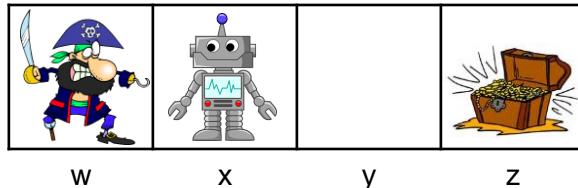
$$S \leftarrow S'; A \leftarrow A';$$

    until  $S$  is terminal

$$\gamma = 0.9$$

**-100**

**+10**



|     |              |          |     |             |
|-----|--------------|----------|-----|-------------|
| $x$ | $\leftarrow$ | <b>0</b> | $w$ | <i>exit</i> |
| S   | A            | R        | S'  | A'          |

|             |            |            |          |
|-------------|------------|------------|----------|
| <b>-100</b> | <b>0,0</b> | <b>0,0</b> | <b>0</b> |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $A$ , observe  $R, S'$

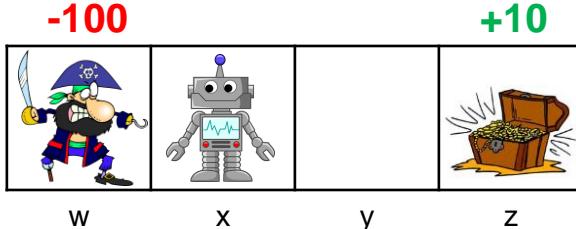
        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

$$S \leftarrow S'; A \leftarrow A';$$

    until  $S$  is terminal

$$\gamma = 0.9$$



|     |              |     |      |             |
|-----|--------------|-----|------|-------------|
| $x$ | $\leftarrow$ | $0$ | $w$  | <i>exit</i> |
| S   | A            | R   | $S'$ | $A'$        |

|        |     |       |     |
|--------|-----|-------|-----|
| $-100$ | $-$ | $0,0$ | $0$ |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

    Repeat (for each step of episode):

        Take action  $A$ , observe  $R, S'$

        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

$$S \leftarrow S'; A \leftarrow A';$$

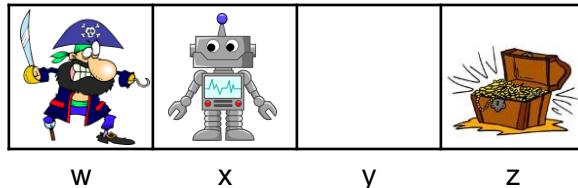
    until  $S$  is terminal

And so on...

$$\gamma = 0.9$$

**-100**

**+10**



|   |      |   |    |      |
|---|------|---|----|------|
| w | exit | 0 | w  | exit |
| S | A    | R | S' | A'   |

|      |   |     |   |
|------|---|-----|---|
| -100 | - | 0,0 | 0 |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

    until  $S$  is terminal

$$\gamma = 0.9$$

+10

-100

|   |   |   |   |
|---|---|---|---|
|   |   |   |   |
| w | x | y | z |

|     |      |      |      |
|-----|------|------|------|
| $x$ | null | null | null |
| S   | A    | R    | S'   |

|   |     |     |   |
|---|-----|-----|---|
| 0 | 0,0 | 0,0 | 0 |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

    until  $S$  is terminal

$$\gamma = 0.9$$

+10

-100

|   |   |   |   |
|---|---|---|---|
|   |   |   |   |
| w | x | y | z |

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

|     |       |       |     |
|-----|-------|-------|-----|
| $0$ | $0,0$ | $0,0$ | $0$ |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$$S \leftarrow S'$$

    until  $S$  is terminal

$$\gamma = 0.9$$

+10

-100

|   |   |   |   |
|---|---|---|---|
|   |   |   |   |
| w | x | y | z |

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

|     |       |       |     |
|-----|-------|-------|-----|
| $0$ | $0,0$ | $0,0$ | $0$ |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$$S \leftarrow S'$$

    until  $S$  is terminal

$$\gamma = 0.9$$

+10

-100

|   |   |   |   |
|---|---|---|---|
|   |   |   |   |
| w | x | y | z |

|     |               |     |     |
|-----|---------------|-----|-----|
| $y$ | $\rightarrow$ | $0$ | $y$ |
| S   | A             | R   | S'  |

|     |       |       |     |
|-----|-------|-------|-----|
| $0$ | $0,0$ | $0,0$ | $0$ |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

    until  $S$  is terminal

$$\gamma = 0.9$$

+10

-100

|   |   |   |   |
|---|---|---|---|
|   |   |   |   |
| w | x | y | z |

|   |   |   |    |
|---|---|---|----|
| y | → | 0 | z  |
| S | A | R | S' |

|   |     |     |   |
|---|-----|-----|---|
| 0 | 0,0 | 0,0 | 0 |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

    until  $S$  is terminal

$$\gamma = 0.9$$

+10

-100

|   |   |   |   |
|---|---|---|---|
|   |   |   |   |
| w | x | y | z |

|          |          |          |          |
|----------|----------|----------|----------|
| <b>z</b> | <b>→</b> | <b>0</b> | <b>z</b> |
| S        | A        | R        | S'       |

|          |            |            |          |
|----------|------------|------------|----------|
| <b>0</b> | <b>0,0</b> | <b>0,0</b> | <b>0</b> |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

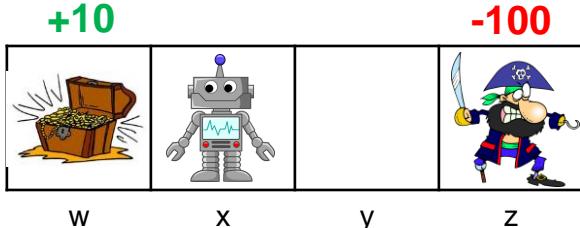
$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

    until  $S$  is terminal

$$\gamma = 0.9$$

|     |   |   |      |
|-----|---|---|------|
| +10 |   |   | -100 |
| w   | x | y | z    |



|     |             |      |      |
|-----|-------------|------|------|
| $z$ | <i>exit</i> | -100 | .    |
| S   | A           | R    | $S'$ |

$$Q = \begin{matrix} & \begin{matrix} 0 & 0,0 & 0,0 & 0 \end{matrix} \\ \begin{matrix} w & x & y & z \end{matrix} & \end{matrix}$$

# 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$$S \leftarrow S'$$

    until  $S$  is terminal

$$\gamma = 0.9$$

+10

-100

|   |   |   |   |
|---|---|---|---|
|   |   |   |   |
| w | x | y | z |

|          |             |             |    |
|----------|-------------|-------------|----|
| <b>z</b> | <b>exit</b> | <b>-100</b> | .  |
| S        | A           | R           | S' |

|          |            |            |             |
|----------|------------|------------|-------------|
| <b>0</b> | <b>0,0</b> | <b>0,0</b> | <b>-100</b> |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

    until  $S$  is terminal

$$\gamma = 0.9$$

+10

-100

|   |   |   |   |
|---|---|---|---|
|   |   |   |   |
| w | x | y | z |

|     |             |      |    |
|-----|-------------|------|----|
| $x$ | <i>exit</i> | -100 | .  |
| S   | A           | R    | S' |

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

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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

    until  $S$  is terminal

$$\gamma = 0.9$$

+10

-100

|   |   |   |   |
|---|---|---|---|
|   |   |   |   |
| w | x | y | z |

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

|     |       |       |        |
|-----|-------|-------|--------|
| $0$ | $0,0$ | $0,0$ | $-100$ |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

    until  $S$  is terminal

$$\gamma = 0.9$$

+10

-100

|   |   |   |   |
|---|---|---|---|
|   |   |   |   |
| w | x | y | z |

|   |   |   |    |
|---|---|---|----|
| y | → | 0 | z  |
| S | A | R | S' |

|   |     |     |      |
|---|-----|-----|------|
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

    until  $S$  is terminal

$$\gamma = 0.9$$

+10

-100

|   |   |   |   |
|---|---|---|---|
|   |   |   |   |
| w | x | y | z |

|   |   |   |    |
|---|---|---|----|
| y | → | 0 | z  |
| S | A | R | S' |

|   |     |       |      |
|---|-----|-------|------|
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

    until  $S$  is terminal

$$\gamma = 0.9$$

+10

-100

|   |   |   |   |
|---|---|---|---|
|   |   |   |   |
| w | x | y | z |

|     |               |     |     |
|-----|---------------|-----|-----|
| $x$ | $\rightarrow$ | $0$ | $z$ |
| S   | A             | R   | S'  |

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

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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

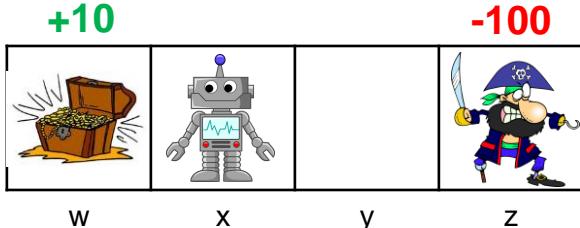
        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

    until  $S$  is terminal

$$\gamma = 0.9$$



|     |   |   |      |
|-----|---|---|------|
| $x$ | → | 0 | $y$  |
| S   | A | R | $S'$ |

|   |     |       |      |
|---|-----|-------|------|
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

    Initialize  $S$

    Repeat (for each step of episode):

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

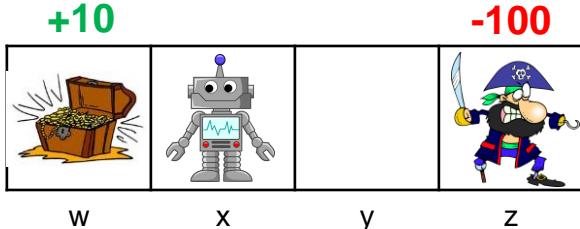
        Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

    until  $S$  is terminal

$$\gamma = 0.9$$



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

|     |        |          |        |
|-----|--------|----------|--------|
| $0$ | $0, 0$ | $0, -90$ | $-100$ |
| 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  $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

Initialize  $S$

Repeat (for each step of episode):

Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

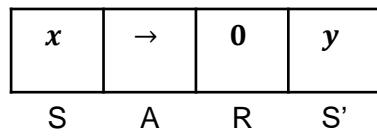
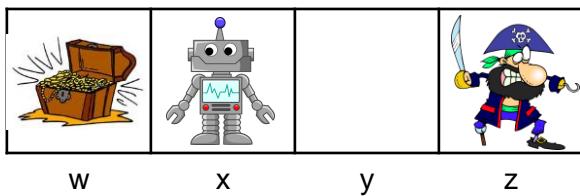
Take action  $A$ , observe  $R, S'$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$

$S \leftarrow S'$

until  $S$  is terminal

And so on...



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





## Required Readings

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



**BITS** Pilani  
Pilani | Dubai | Goa | Hyderabad

Thank you



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](mailto:vimalsp@wilp.bits-pilani.ac.in)),
2. Prof. Sangeetha Viswanathan ([sangeetha.viswanathan@pilani.bits-pilani.ac.in](mailto:sangeetha.viswanathan@pilani.bits-pilani.ac.in))

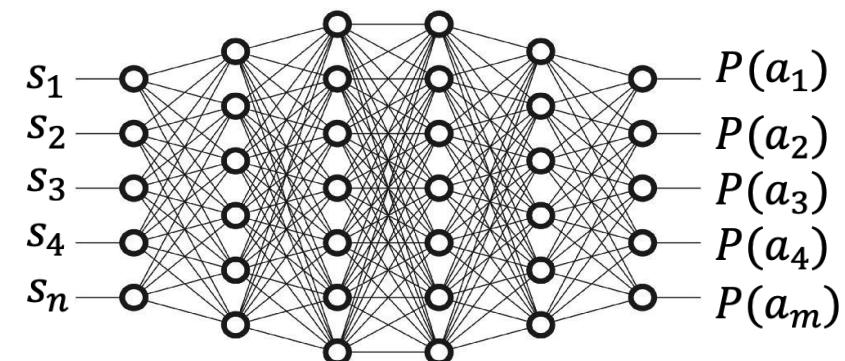


# Agenda for the classes

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

# Notation

- The policy is a parametrized function:  $\pi_\theta(a|s)$ 
  - For policy gradient we need a continuous, differentiable policy... (Softmax activation can be useful for discrete action spaces)
  - $\pi(a|s)$  is assumed to be differentiable with respect to  $\theta$
  - E.g., a DNN where  $\theta$  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)

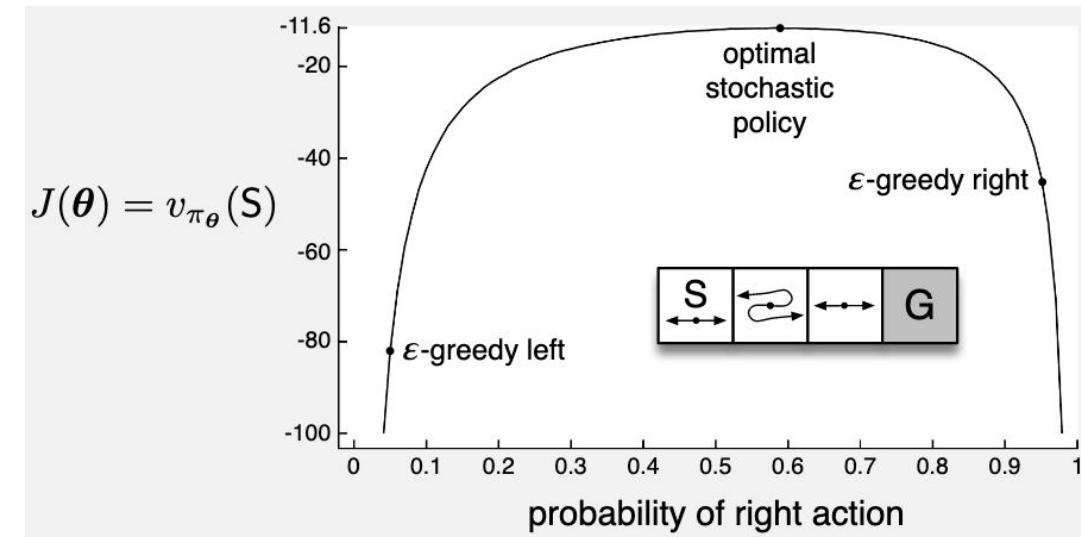


# Advantages of PG

- The policy convergence over time as opposed to an epsilon-greedy value-based approach
- Naturally applies to continuous 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
- $\mathcal{A} = \{\text{left}, \text{right}\}$
- Left goes left and right goes right except in the middle state where they are reversed
- States are identical from the policy's perspective
- $\pi^* = [0.41 \quad 0.59]$





# 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  $\theta$
- 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  $q_\pi$ , denoted  $\widehat{q}$ , 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  $\pi(a|s, \theta)$

Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$

Repeat forever:

Generate an episode  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ , following  $\pi(\cdot|\cdot, \theta)$

For each step of the episode  $t = 0, \dots, T - 1$ :

$G \leftarrow$  return from step  $t$

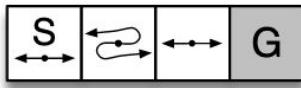
$$\theta \leftarrow \theta + \alpha \gamma^t G \boxed{\nabla_{\theta} \ln \pi(A_t | S_t; \theta)}$$

How<sup>12</sup> did we get here  
from  $\frac{\nabla_{\theta} \pi(A_t | S_t; \theta)}{\pi(A_t | S_t; \theta)}$ ?

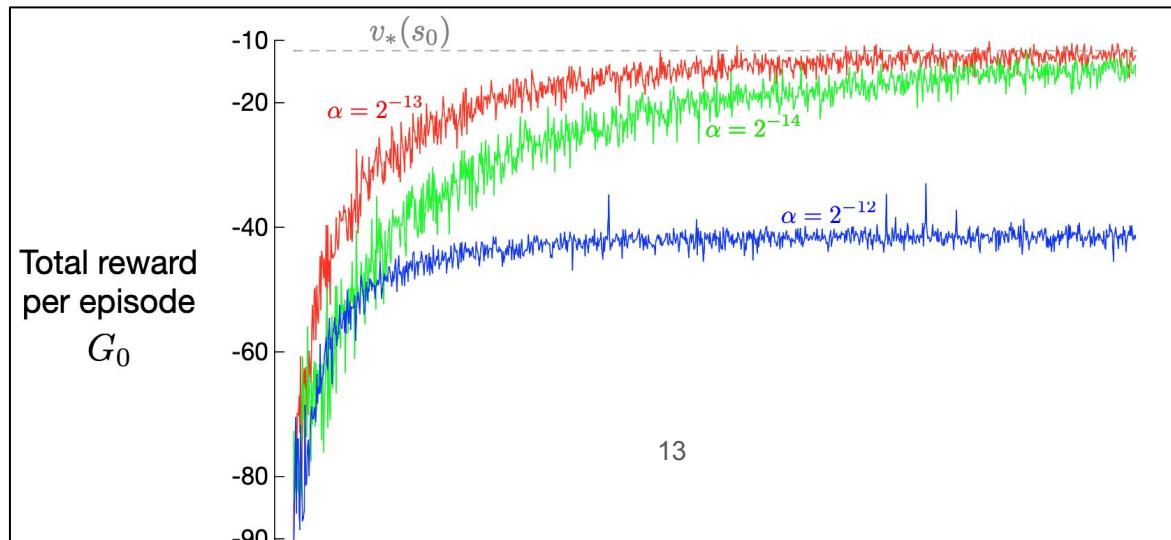
$$\nabla \ln(f(x)) = \frac{\nabla f(x)}{f(x)}$$

# 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  $\alpha$

1000

# 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:  $\theta_{t+1} = \theta_t + \alpha \textcolor{red}{G}_t \nabla_\theta \ln \pi(A_t | S_t; \theta)$
- 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!
- $\nabla J(\theta) \propto \sum_s \mu(s) \sum_a (q_\pi(s, a) - b(s)) \nabla \pi(a|s)$
- 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  $a$
- $\sum_a b(s) \nabla \pi(a|s) = b(s) \nabla \sum_a \pi(a|s) = b(s) \nabla 1 = 0$  15 (actions' probabilities sum to 1)
- REINFORCE with baseline:  $\theta_{t+1} = \theta_t + \alpha(G_t - b(S_t)) \nabla_\theta \ln \pi_\theta(A_t | S_t)$

## 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

- $\nabla J(\theta) \propto \sum_s \mu(s) \sum_a (q_\pi(s, a) - v_\pi(s)) \nabla \pi(a|s)$
- 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  $\pi(a|s, \theta)$

Input: a differentiable state-value parameterization  $\hat{v}(s, w)$

Parameters: step sizes  $\alpha^\theta > 0$ ,  $\alpha^w > 0$

Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$  and state-value weights  $w \in \mathbb{R}^d$

Repeat forever:

Generate an episode  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ , following  $\pi(\cdot | \cdot, \theta)$

For each step of the episode  $t = 0, \dots, T - 1$ :

$$G_t \leftarrow \text{return from step } t$$

$$\delta \leftarrow G_t - \hat{v}(S_t, w)$$

$$w \leftarrow w + \alpha^w \gamma^t \delta \nabla_w \hat{v}(S_t, w)$$

$$\theta \leftarrow \theta + \alpha^\theta \gamma^t \delta \nabla_\theta \ln \pi(A_t | S_t, \theta)$$

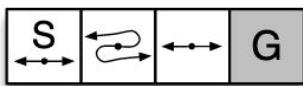
18

Each approximator has its  
unique learning rate

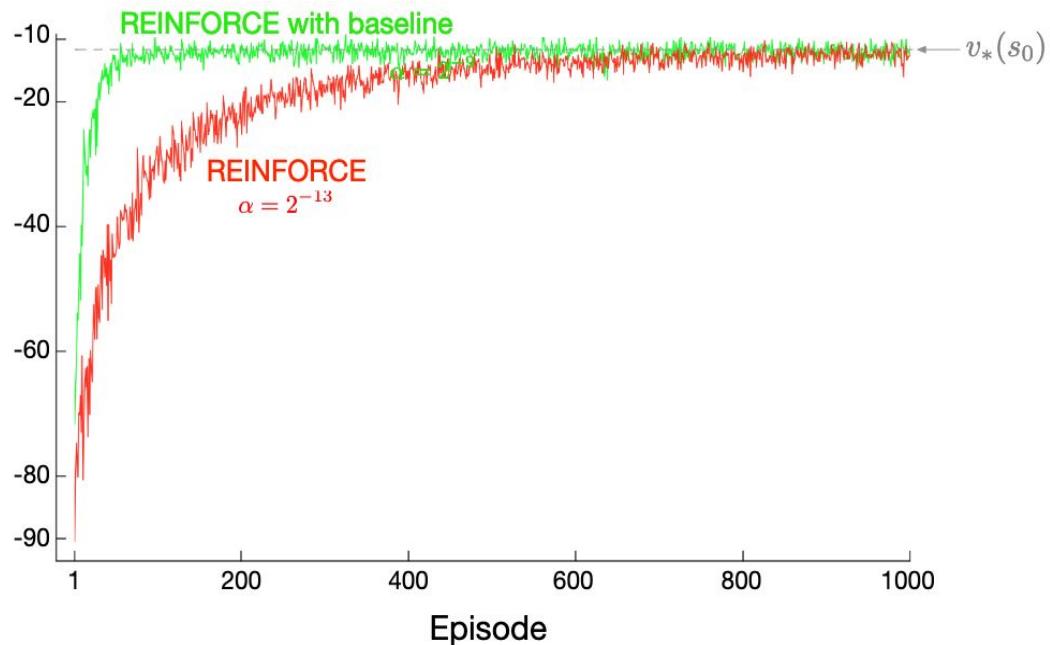
# REINFORCE with baseline

- The crazy corridor domain

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



Total reward  
per episode  
 $G_0$



# Policy Gradient

- $\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)}$
- PG theorem:  $\widehat{\nabla J(\theta_t)} \propto (q_\pi(S_t, A_t) - b(S_t)) \nabla_\theta \log \pi(A_t | S_t; \theta)$ 
  - REINFORCE+baseline:  $\theta_{t+1} = \theta_t + \alpha(G_t - b(S_t)) \nabla_\theta \log \pi(A_t | S_t; \theta)$
- **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<sup>20</sup>(high variance) though unbiased
  - Unstable learning

## Add a critic

- $\widehat{\nabla J(\theta_t)} \propto (q_{\pi}(S_t, A_t) - b(S_t)) \frac{\nabla_{\theta} \pi(A_t | S_t; \theta)}{\pi(A_t | S_t; \theta)}$
- 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?
    - $\theta_{t+1} = \theta_t + \alpha(G_t - \hat{v}_\pi(S_t)) \nabla_\theta \ln \pi(A_t | S_t; \theta)$
    - 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)<sup>22</sup>
  - 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  $\pi$ ?
  - $A_\pi(s, a) = q_\pi(s, a) - v_\pi(s)$
- This resembles PG with baseline but not the same as  $q_\pi$  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
  - $\hat{q}_\pi(s_t, a_t) - \hat{v}_\pi(s_t; w) = \hat{\mathbb{E}}[r_{t+1} + \gamma \hat{v}(s_{t+1}; w)] - \hat{v}(s_t; w)$
- $\theta_{t+1} = \theta_t + \alpha(r_{t+1} + \gamma \hat{v}(s_{t+1}; w) - \hat{v}(s_t; w)) \nabla_\theta \ln \pi(a_t | s_t; \theta)$ 

$\delta - \text{TD error}$
- 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  $\pi(a|s, \theta)$

Input: a differentiable state-value parameterization  $\hat{v}(s, w)$

Parameters: step sizes  $\alpha^\theta > 0$ ,  $\alpha^w > 0$

Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$  and state-value weights  $w \in \mathbb{R}^d$

Repeat forever:

    Initialize  $S$  (first state of episode)

$I \leftarrow 1$

    While  $S$  is not terminal:

$$A \sim \pi(\cdot|S, \theta)$$

    Take action  $A$ , observe  $S', R$

$$\delta \leftarrow R + \gamma \hat{v}(S', w) - \hat{v}(S, w) \quad (\text{if } S' \text{ is terminal, then } \hat{v}(S', w) \doteq 0)$$

$$w \leftarrow w + \alpha^w \delta \nabla_w \hat{v}(S, w)$$

$$\theta \leftarrow \theta + \alpha^\theta I \delta \nabla_\theta \ln \pi(A|S, \theta)$$

$$I \leftarrow \gamma I$$

$$S \leftarrow S'$$

Keep track of  
accumulated discount

## One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization  $\pi(a|s, \theta)$

Input: a differentiable state-value parameterization  $\hat{v}(s, w)$

Parameters: step sizes  $\alpha^\theta > 0$ ,  $\alpha^w > 0$

Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$  and state-value weights  $w \in \mathbb{R}^d$

Repeat forever:

    Initialize  $S$  (first state of episode)

$I \leftarrow 1$

    While  $S$  is not terminal:

$A \sim \pi(\cdot|S, \theta)$

        Take action  $A$ , observe  $S', R$

$\delta \leftarrow R + \gamma \hat{v}(S', w) - \hat{v}(S, w)$

            (if  $S'$  is terminal, then  $\hat{v}(S', w) \doteq 0$ )

$w \leftarrow w + \alpha^w \delta \nabla_w \hat{v}(S, w)$

$\theta \leftarrow \theta + \alpha^\theta I \delta \nabla_\theta \ln \pi(A|S, \theta)$

$I \leftarrow \gamma I$

$S \leftarrow S'$

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  $\pi(a|s, \theta)$

Input: a differentiable state-value parameterization  $\hat{v}(s, w)$

Parameters: step sizes  $\alpha^\theta > 0$ ,  $\alpha^w > 0$

Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$  and state-value weights  $w \in \mathbb{R}^d$

Repeat forever:

    Initialize  $S$  (first state of episode)

$I \leftarrow 1$

    While  $S$  is not terminal:

$A \sim \pi(\cdot|S, \theta)$

        Take action  $A$ , observe  $S', R$

$\delta \leftarrow R + \gamma \hat{v}(S', w) - \hat{v}(S, w)$

$w \leftarrow w + \alpha^w \delta \nabla_w \hat{v}(S, w)$

$\theta \leftarrow \theta + \alpha^\theta I \delta \nabla_\theta \ln \pi(A|S, \theta)$

$I \leftarrow \gamma I$

$S \leftarrow S'$

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  $S', R$

$\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S, \boldsymbol{\theta})$

$I \leftarrow \gamma I$

$S \leftarrow S'$

(if  $S'$  is terminal, then  $\hat{v}(S', \mathbf{w}) \doteq 0$ )

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  $\pi(a|s, \theta)$

Input: a differentiable state-value parameterization  $\hat{v}(s, w)$

Parameters: step sizes  $\alpha^\theta > 0$ ,  $\alpha^w > 0$

Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$  and state-value weights  $w \in \mathbb{R}^d$

Repeat forever:

    Initialize  $S$  (first state of episode)

$I \leftarrow 1$

    While  $S$  is not terminal:

$A \sim \pi(\cdot|S, \theta)$

        Take action  $A$ , observe  $S', R$

$\delta \leftarrow R + \gamma \hat{v}(S', w) - \hat{v}(S, w)$

$w \leftarrow w + \alpha^w \delta \nabla_w \hat{v}(S, w)$

$\theta \leftarrow \theta + \alpha^\theta I \delta \nabla_\theta \ln \pi(A|S, \theta)$

$I \leftarrow \gamma I$

$S \leftarrow S'$

(if  $S'$  is terminal, then  $\hat{v}(S', w) \doteq 0$ )

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  $\pi_\theta$  and the critic  $\hat{V}_w$
- For each episode:
  - Init empty episode memory
  - $s = \text{init env}$
  - For each time step:
    - $a \sim \pi(s)$
    - $s', r_t \sim \text{env}(s, a)$
    - 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  $\pi(a|s, \theta)$

Input: a differentiable state-value parameterization  $\hat{v}(s, w)$

Parameters: trace-decay rates  $\lambda^\theta \in [0, 1]$ ,  $\lambda^w \in [0, 1]$ ; step sizes  $\alpha^\theta > 0$ ,  $\alpha^w > 0$

Initialize policy parameter  $\theta \in \mathbb{R}^{d'}$  and state-value weights  $w \in \mathbb{R}^d$

Repeat forever (for each episode):

    Initialize  $S$  (first state of episode)

$z^\theta \leftarrow \mathbf{0}$  ( $d'$ -component eligibility trace vector)

$z^w \leftarrow \mathbf{0}$  ( $d$ -component eligibility trace vector)

$I \leftarrow 1$

    While  $S$  is not terminal (for each time step):

$A \sim \pi(\cdot|S, \theta)$

        Take action  $A$ , observe  $S', R$

$\delta \leftarrow R + \gamma \hat{v}(S', w) - \hat{v}(S, w)$      (if  $S'$  is terminal, then  $\hat{v}(S', w) \doteq 0$ )

$z^w \leftarrow \gamma \lambda^w z^w + \nabla_w \hat{v}(S, w)$

$z^\theta \leftarrow \gamma \lambda^\theta z^\theta + I \nabla_\theta \ln \pi(A|S, \theta)$

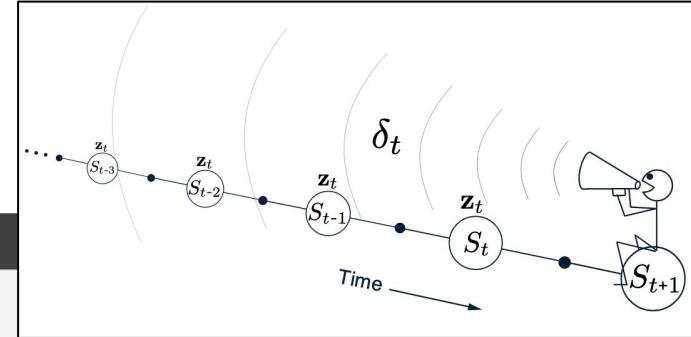
$w \leftarrow w + \alpha^w \delta z^w$

$\theta \leftarrow \theta + \alpha^\theta \delta z^\theta$

$I \leftarrow \gamma I$

$S \leftarrow S'$

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:  $\nabla_{\theta} J(\theta)$
- Such approximation is made possible when by the policy gradient theorem
- Should we reinforce policies 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:

- $\bullet \widehat{\nabla J(\theta_t)} = \boxed{G_t} \nabla_{\theta} \ln \pi(A_t | S_t; \theta)$

- Q Actor-Critic:

- $\bullet \widehat{\nabla J(\theta_t)} = \boxed{\hat{q}(S_t, A_t; w)} \nabla_{\theta} \ln \pi(A_t | S_t; \theta)$

- REINFORCE + baseline:

- $\bullet \widehat{\nabla J(\theta_t)} = (\boxed{G_t} - \boxed{\hat{v}(S_t; w)}) \nabla_{\theta} \ln \pi(A_t | S_t; \theta)$

- Advantage Actor-Critic:

- $\bullet \widehat{\nabla J(\theta_t)} = (\boxed{R_{t+1} + \gamma \hat{v}(S_{t+1}; w)} - \boxed{\hat{v}(S_t; w)}) \nabla_{\theta} \ln \pi(A_t | S_t; \theta)$



## Required Readings

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



**BITS** Pilani  
Pilani | Dubai | Goa | Hyderabad

Thank you



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](mailto:vimalsp@wilp.bits-pilani.ac.in)),
2. Prof. Sangeetha Viswanathan ([sangeetha.viswanathan@pilani.bits-pilani.ac.in](mailto:sangeetha.viswanathan@pilani.bits-pilani.ac.in))

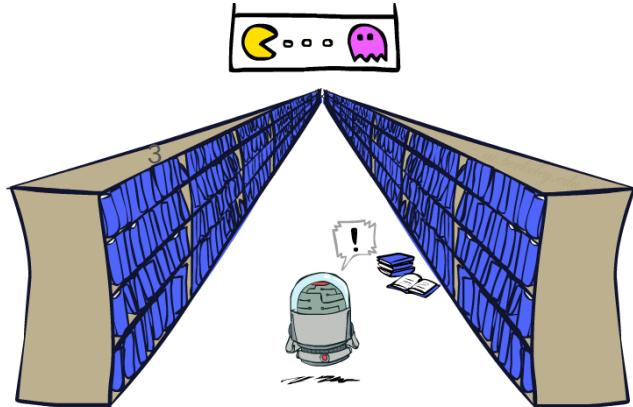
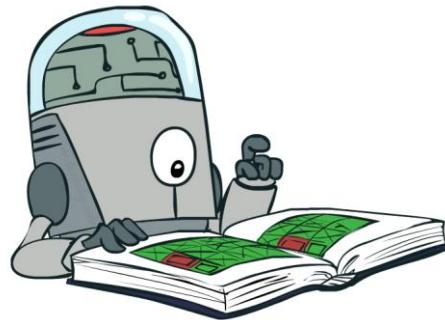


# Agenda for the classes

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

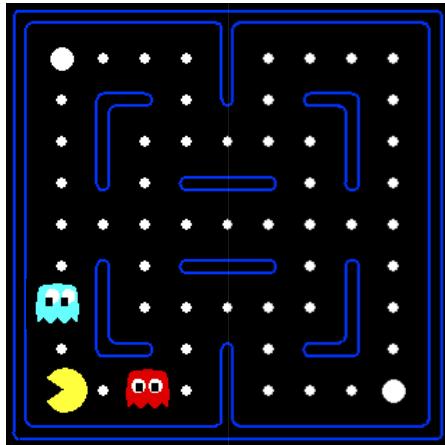
# Generalizing Across States

- 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

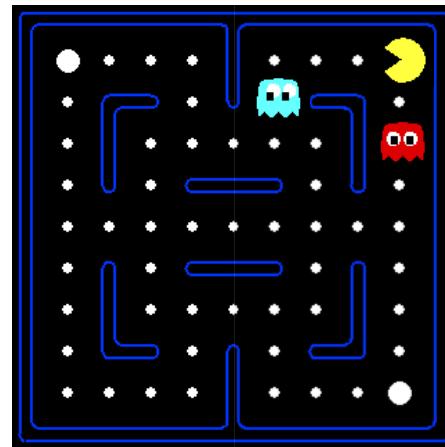


# Example: Pacman

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



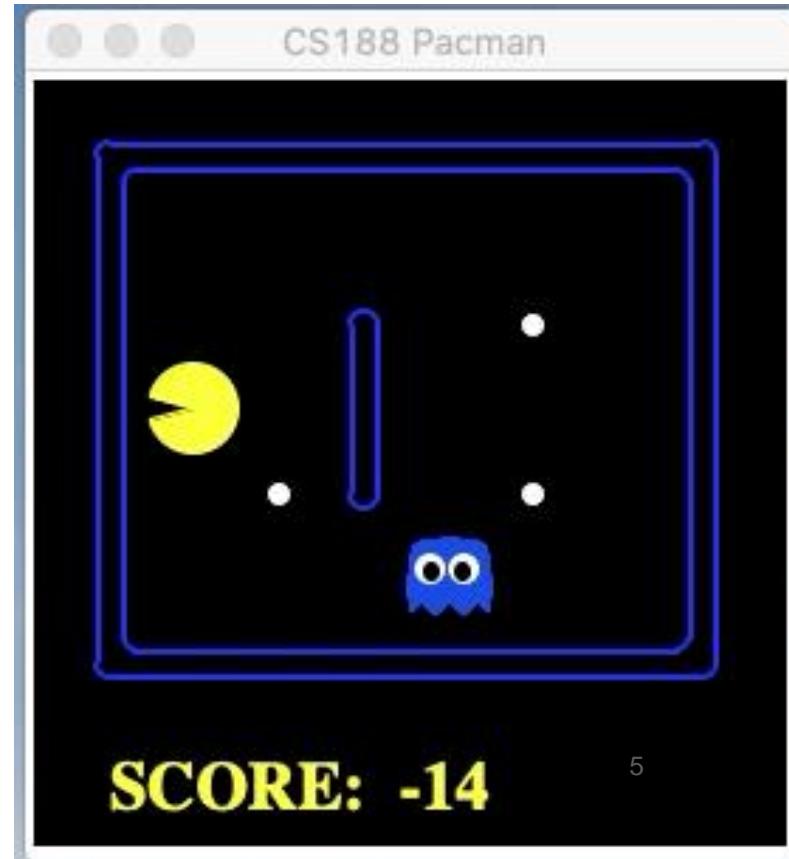
In naïve tabular-learning, we know nothing about this state:



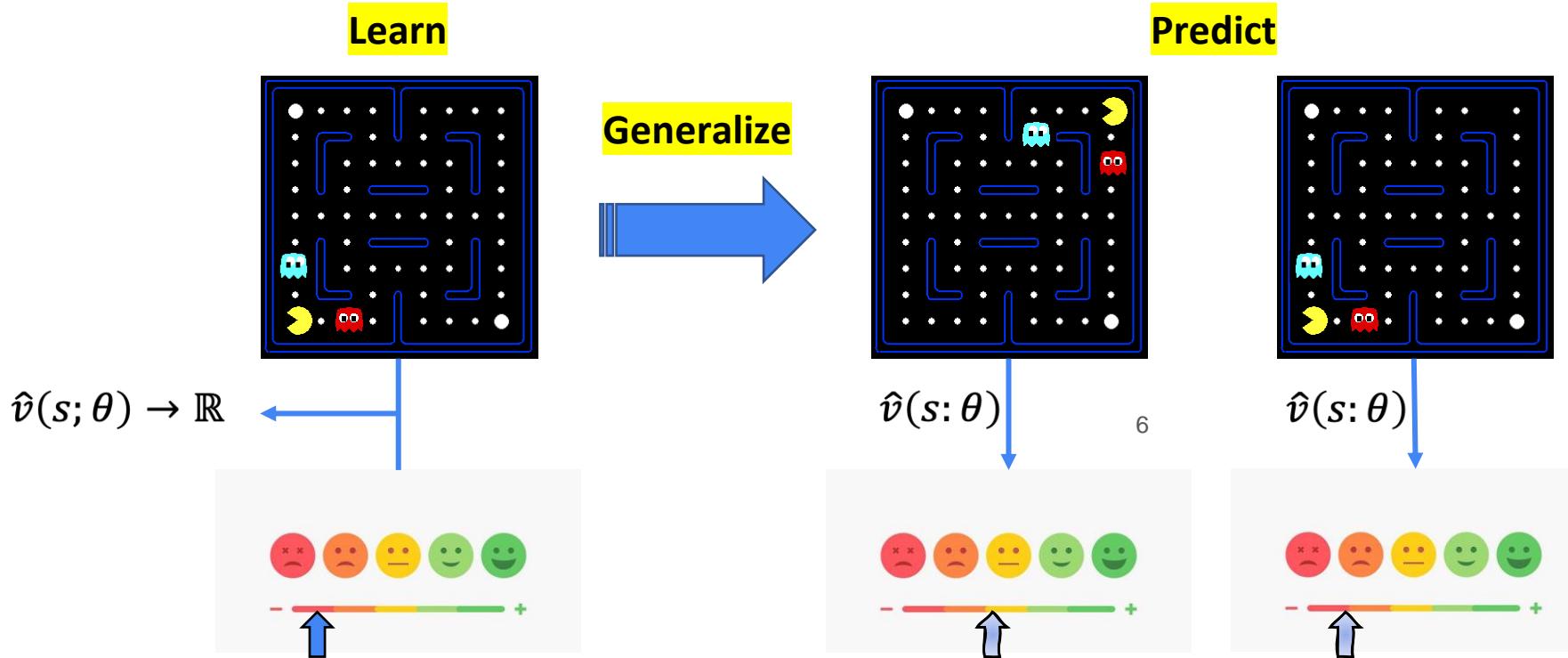
Or even this one!



- Naïve Q-learning
- After 50 training episodes



# Learn an approximation function

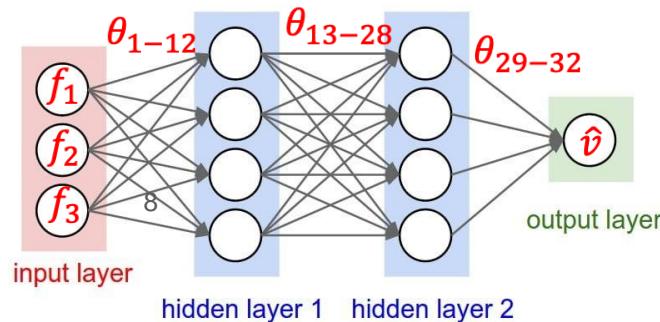


- Q-learning with function approximator



# Parameterized function approximator

- Assume that each state is vector of features  $(f_1, f_2, \dots, f_n)$ , 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  $\theta$  to reflect this information?



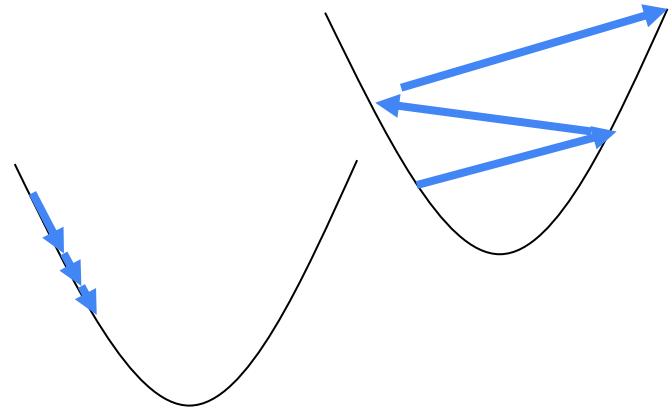
# Gradient Descent

- Given:  $v(S_1) = 5, v(S_2) = 8, v(S_3) = 2$
- We want to set  $\theta$  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  $\theta_i$  = how to change  $\theta_i$  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  $\theta_i$  towards a direction that minimizes loss
  - $\theta_i = \theta_i - \frac{\partial \text{loss}}{\partial \theta_i}$
- More generally  $\theta = \theta - \alpha \nabla \text{loss}$ 
  - $\nabla \text{loss} = \left( \frac{\partial \text{loss}}{\partial \theta_1}, \frac{\partial \text{loss}}{\partial \theta_2}, \dots, \frac{\partial \text{loss}}{\partial \theta_n} \right)$
  - $\alpha$  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  $\pi$  to be evaluated

Input: a differentiable function  $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$

Initialize value-function weights  $\mathbf{w}$  as appropriate (e.g.,  $\mathbf{w} = \mathbf{0}$ )

Repeat forever:

Generate an episode  $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$  using  $\pi$

For  $t = 0, 1, \dots, T - 1$ :

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$$

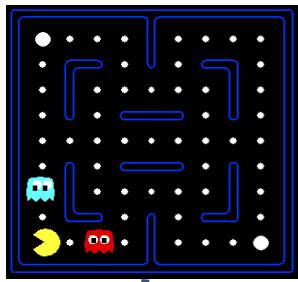
w are the tunable parameters of the value approximation function

18

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

# Example

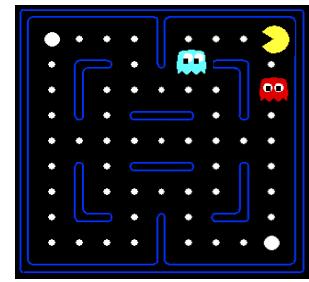
$$f(S) = [2,2,1]$$



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:  $\theta = [0,0,0]$
- $\theta = \theta + \alpha(G_t - \hat{v}(s; \theta))\nabla \hat{v}(s; \theta)$
- $\theta = [0,0,0] + 0.1(10 - [0,0,0] \cdot [2,2,1])[2,2,1]$ 
  - $\theta = [2,2,1]$
- $\hat{v}(S') = f(S') \cdot \theta = [2,4,1] \cdot [2,2,1] = 13$

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



# 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:  $\hat{v}(s_t) = \hat{v}(s_t) + \alpha(r_t + \gamma \hat{v}(s_{t+1}) - \hat{v}(s_t))$
  - Approximation TD learning:  $\theta = \theta + \alpha(r_t + \gamma \hat{v}(s_{t+1}; \theta) - \hat{v}(s_t; \theta)) \nabla \hat{v}(s_t; \theta)$
- 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  $\pi$  to be evaluated

Input: a differentiable function  $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$  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  $R, S'$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})] \nabla \hat{v}(S, \mathbf{w})$

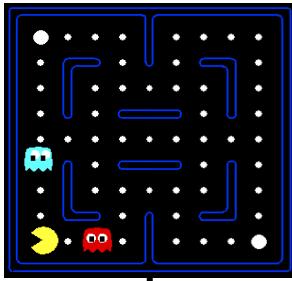
$S \leftarrow S'$

    until  $S'$  is terminal

What's the difference  
from the tabular case?

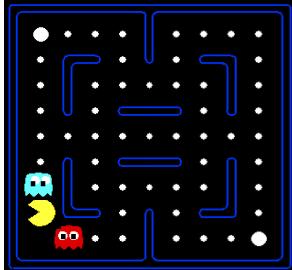
# Example

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



$$R = +10$$

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



- $S = \{f_1(S), f_2(S), f_3(S)\}$

- $f_{1,2}$ =distance to ghost 1,2,  $f_3$ =distance to food

- $\hat{v}(s) = \sum_i \theta_i f_i(s)$

- init:  $\theta = [0,0,0]$

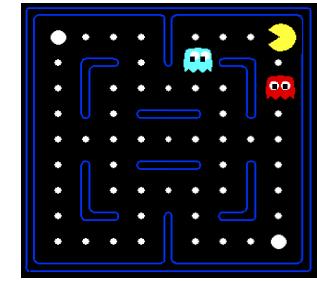
- $\theta = \theta + \alpha(R + \gamma \hat{v}(S'; \theta) - \hat{v}(S; \theta)) \nabla \hat{v}(S; \theta)$

- $\theta = [0,0,0] + 0.1(10 + [1,2,1] \cdot [0,0,0] - [2,3,1] \cdot [0,0,0]) [2,3,1]$

- $\theta = [2,3,1]$

- $\hat{v}(U) = f(U) \cdot \theta = [2,4,1] \cdot [2,3,1] = 17^{23}$

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



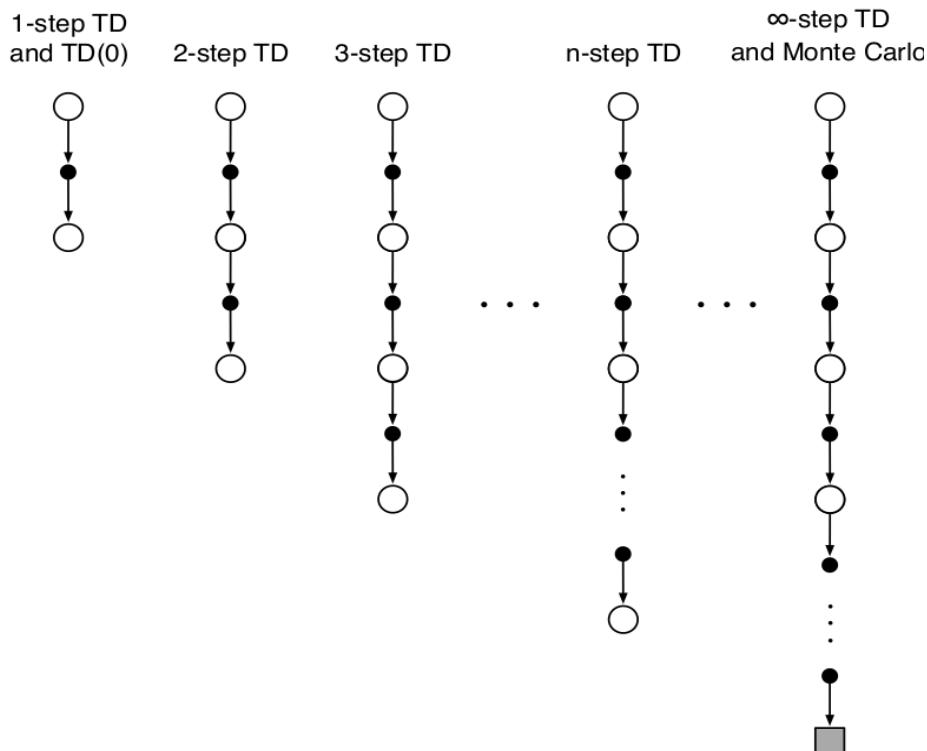
# [Review] n-step TD Prediction

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:

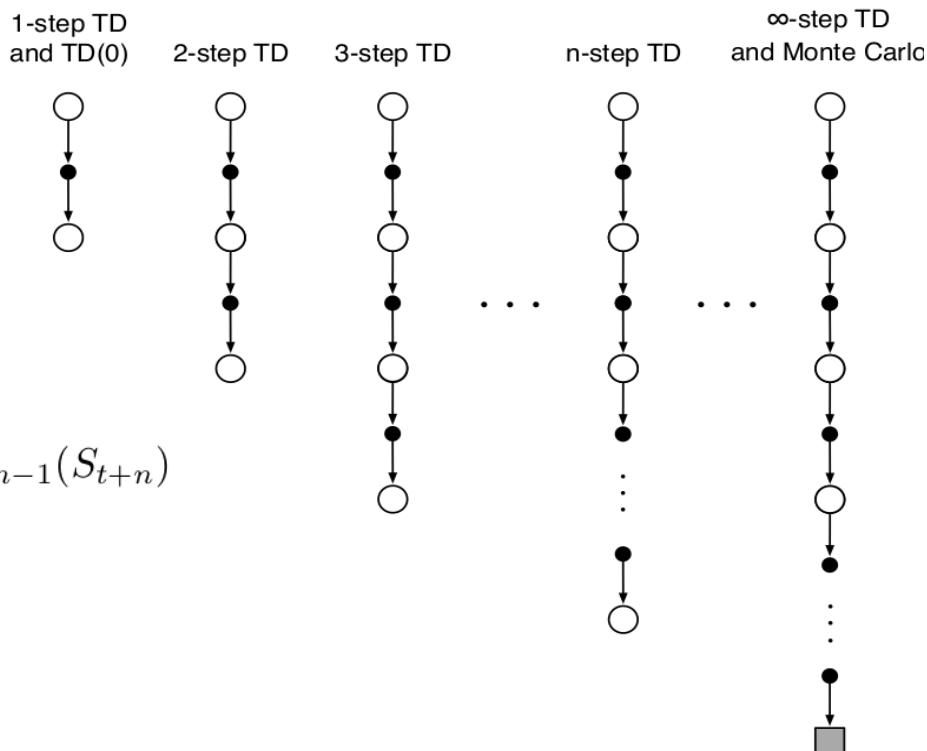
$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1})$$

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:

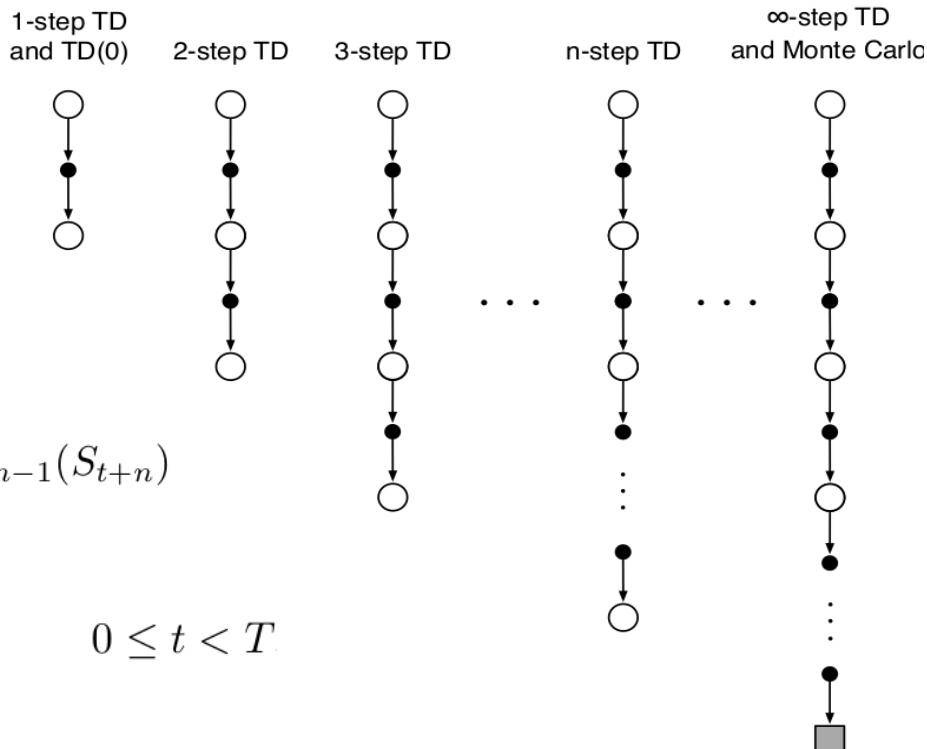
$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$

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:

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



Ref Section 7.1 of TB

# [Review] n-step TD Prediction

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

Input: a policy  $\pi$

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  $t = 0, 1, 2, \dots$ :

    | 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  $T \leftarrow t + 1$

      |  $\tau \leftarrow t - n + 1$  ( $\tau$  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  $\pi$  to be evaluated

Input: a differentiable function  $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$  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  $t \equiv 0, 1, 2, \dots$ :

If  $t < T$ , then:

Take an action according to  $\pi(\cdot|S_t)$

Observe and store the next reward

If  $S_{t+1}$  is terminal, then  $T \leftarrow t + 1$

$$\tau \leftarrow t - r$$

$$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} B_i$$

If  $\tau + n \leq T$ , then:  $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$

$$(G_{\tau_1, \tau_1 + \tau_0})$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{v}(S_\tau, \mathbf{w})] \nabla \hat{v}(S_\tau, \mathbf{w})$$

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  $\pi$  to be evaluated

Input: a differentiable function  $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$  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  $S_0 \neq \text{terminal}$

$$T \leftarrow \infty$$

For  $t = 0, 1, 2, \dots$ :

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  $T \leftarrow t + 1$

$\tau \leftarrow t - n + 1$  ( $\tau$  is the time whose state's estimate is being updated)

If  $\tau > 0$ :

$$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$$

If  $\tau + n < T$ , then:  $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n}, \mathbf{w})$

$$(G_{\tau;\tau+n})$$

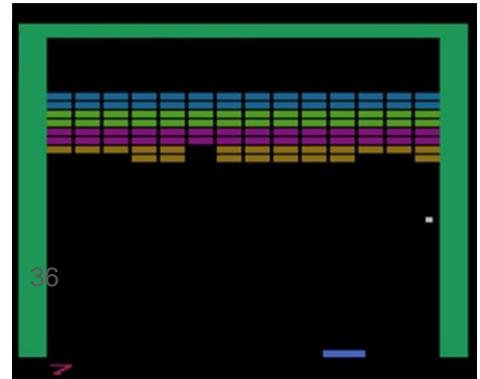
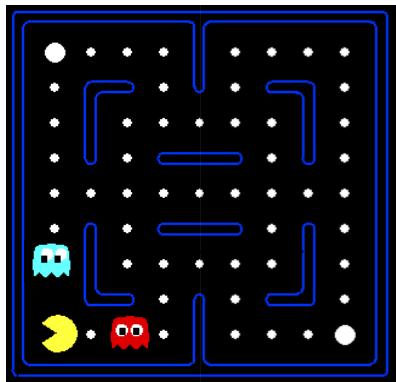
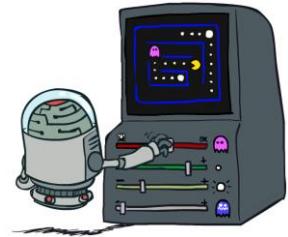
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{v}(S_{\tau}, \mathbf{w})] \nabla \hat{v}(S_{\tau}, \mathbf{w})$$

Until  $\tau \equiv T - 1$

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

# Feature selection

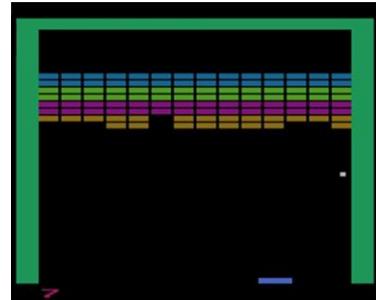
- Assume a linear function approximator  $\hat{v}(f(s); \theta) = f(s) \cdot \theta$
- What relevant features should represent states?



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:**  $f_i(s) = \prod_{j=1}^k x_j^{c_{i,j}}$

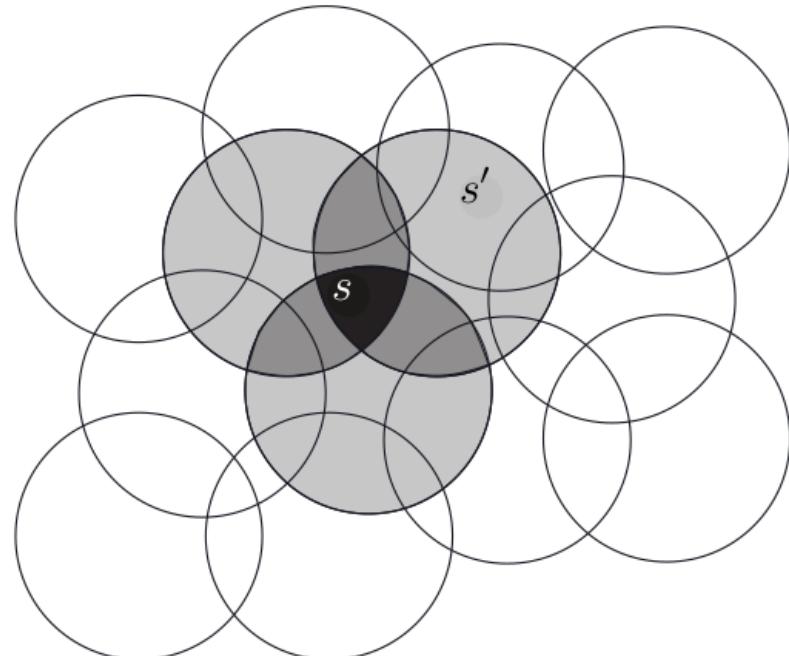
- where each  $c_{i,j}$  is an integer in the set  $\{0, 1, \dots, n\}$  for an integer  $n \geq 0$
- These features makeup the order- $n$  polynomial basis for dimension  $k$ , which contains  $(n + 1)^k$  different features

- **Fourier Basis:**  $f_i(s) = \cos(\pi X^\top c^i)$

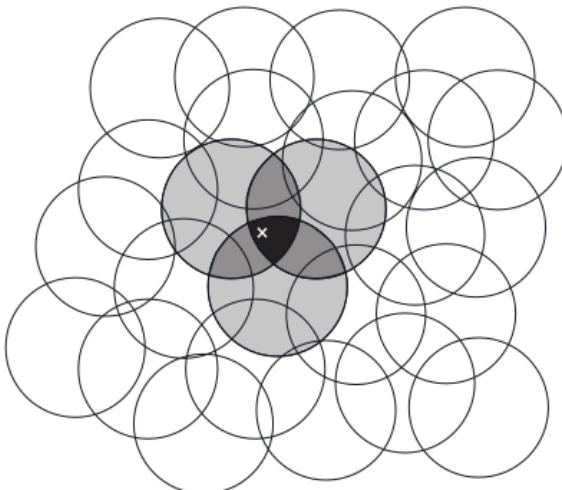
- Where  $c^i = (c_1^i, \dots, c_k^i)^\top$ , with  $c_j^i \in \{0, \dots, n\}$  for  $j = \{1, \dots, k\}$  and  $i = \{0, \dots, (n + 1)^k\}$
- This defines a feature for each of the  $(n + 1)^k$  possible integer vectors  $c^i$
- The inner product  $X^\top c^i$  has the effect of assigning an integer 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

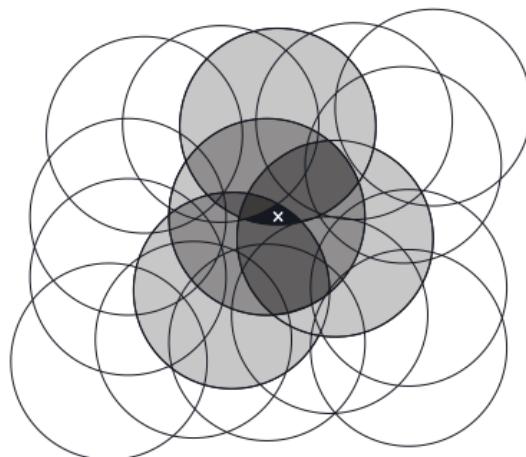
- Natural representation of the state set is continuous
- In 2-d, features corresponding to circles in state space
- Coding of a state:
  - If the state is inside a circle, then the corresponding feature has the value 1
  - otherwise the feature is 0
- Corresponding to each circle is a single weight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



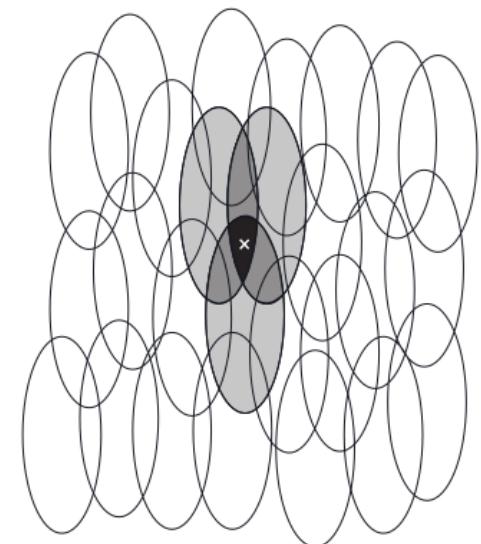
# Automatic features extraction for linear approximator - Coarse Coding



Narrow generalization

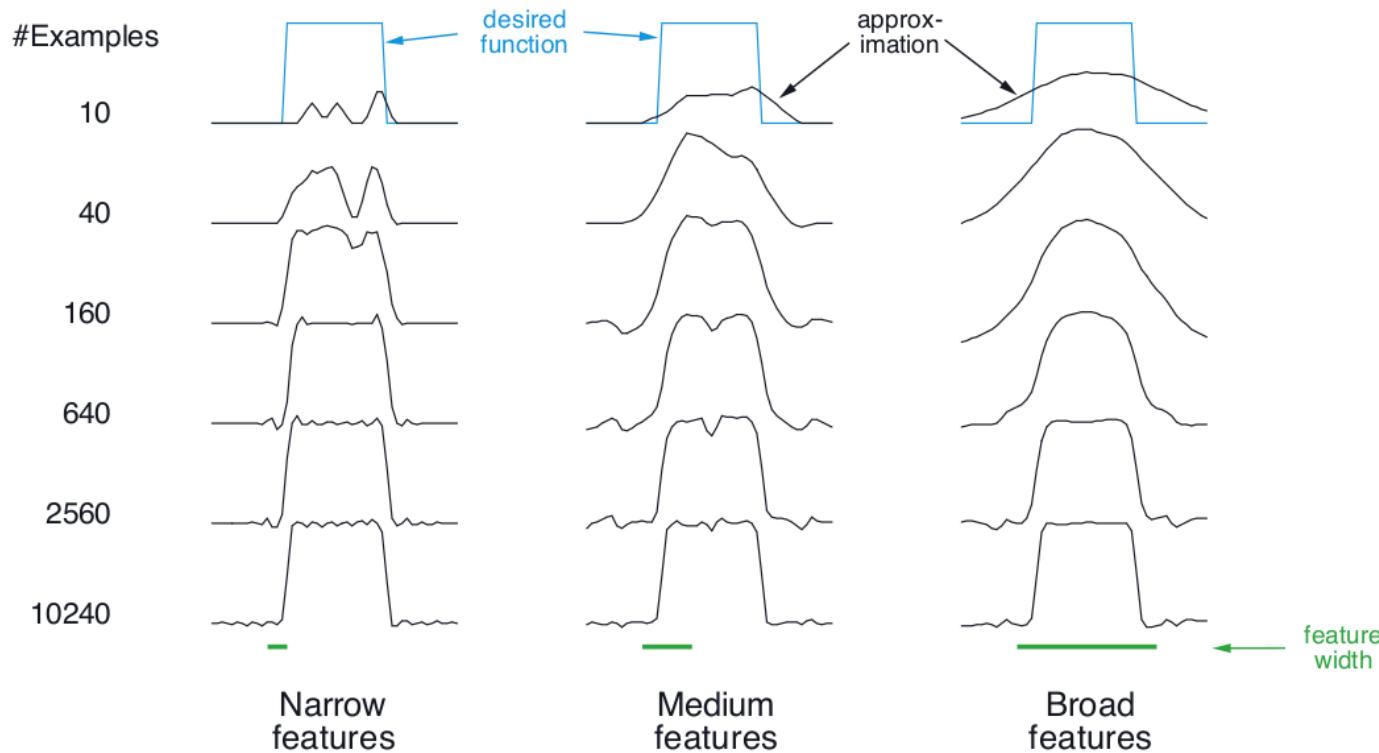


Broad generalization

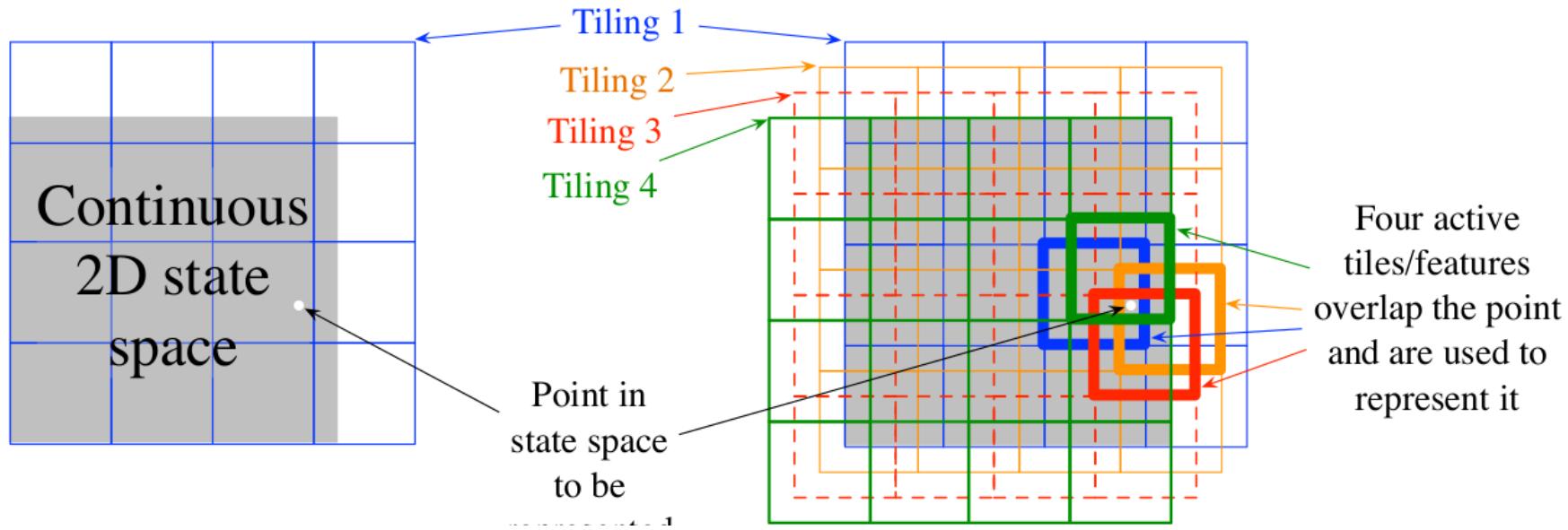


Asymmetric generalization

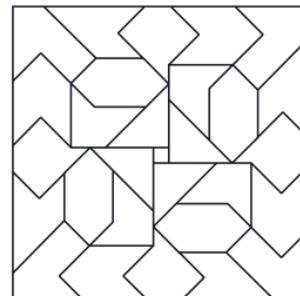
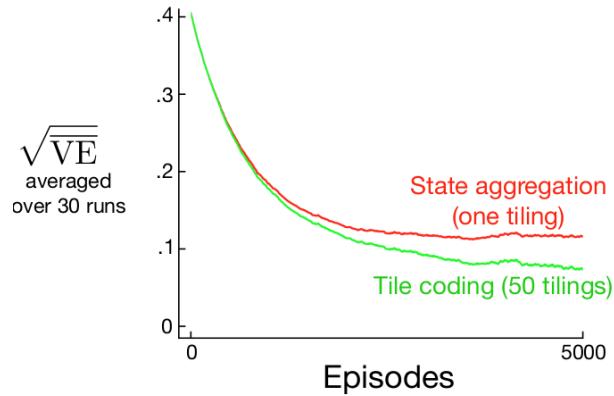
# Automatic features extraction for linear approximator - Coarse Coding



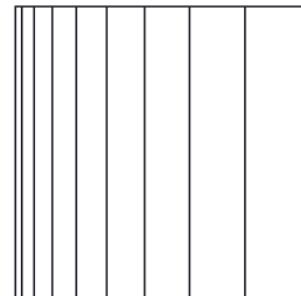
# Automatic features extraction for linear approximator - Tile Coding



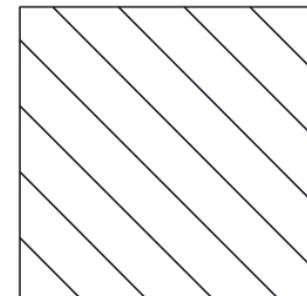
# Automatic features extraction for linear approximator - Tile Coding



Irregular

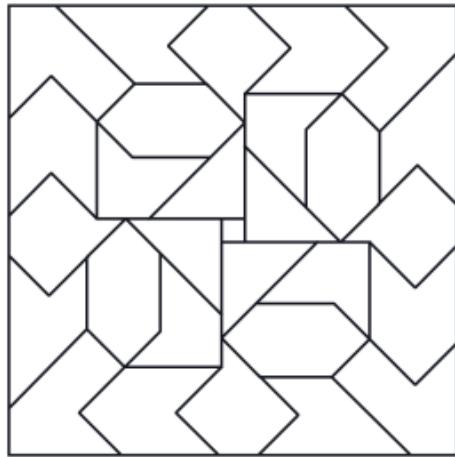


Log stripes

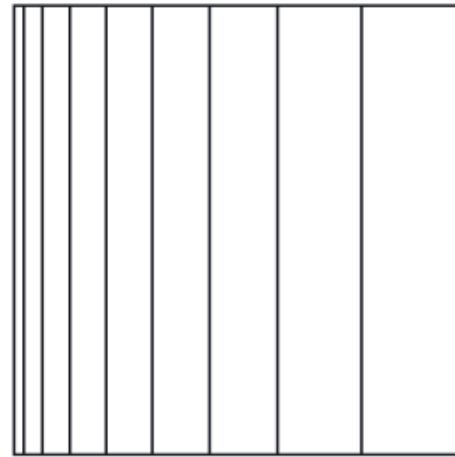


Diagonal stripes

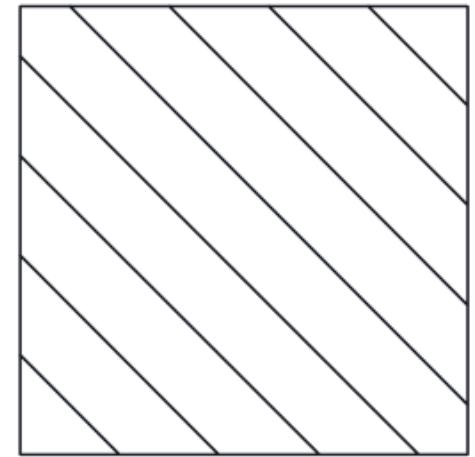
# Automatic features extraction for linear approximator - Tile Coding



Irregular



Log stripes

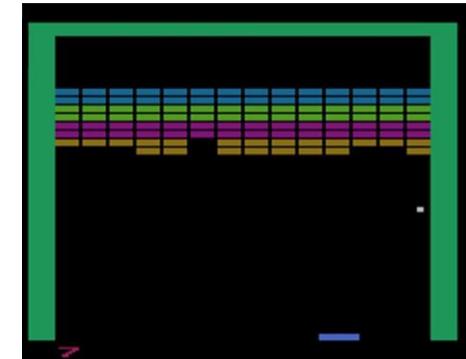


Diagonal stripes

# 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...

45



# 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

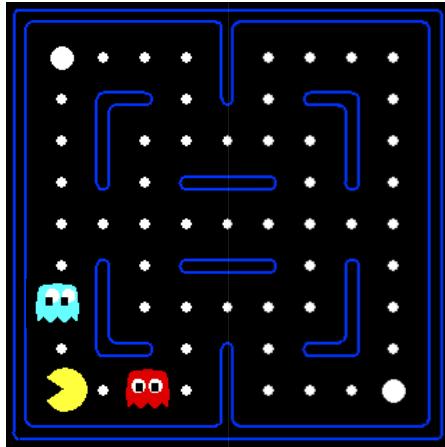
# The age of deep learning

- 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

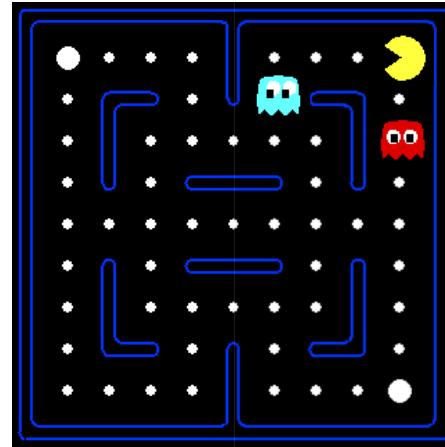


# Example: Pacman

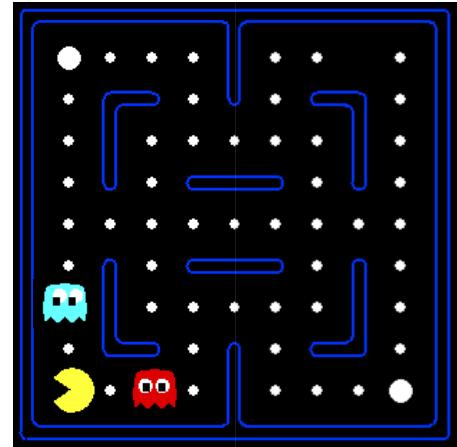
Let's say we discover through experience that this state is bad:



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



Or even this one!



We must generalize our knowledge!

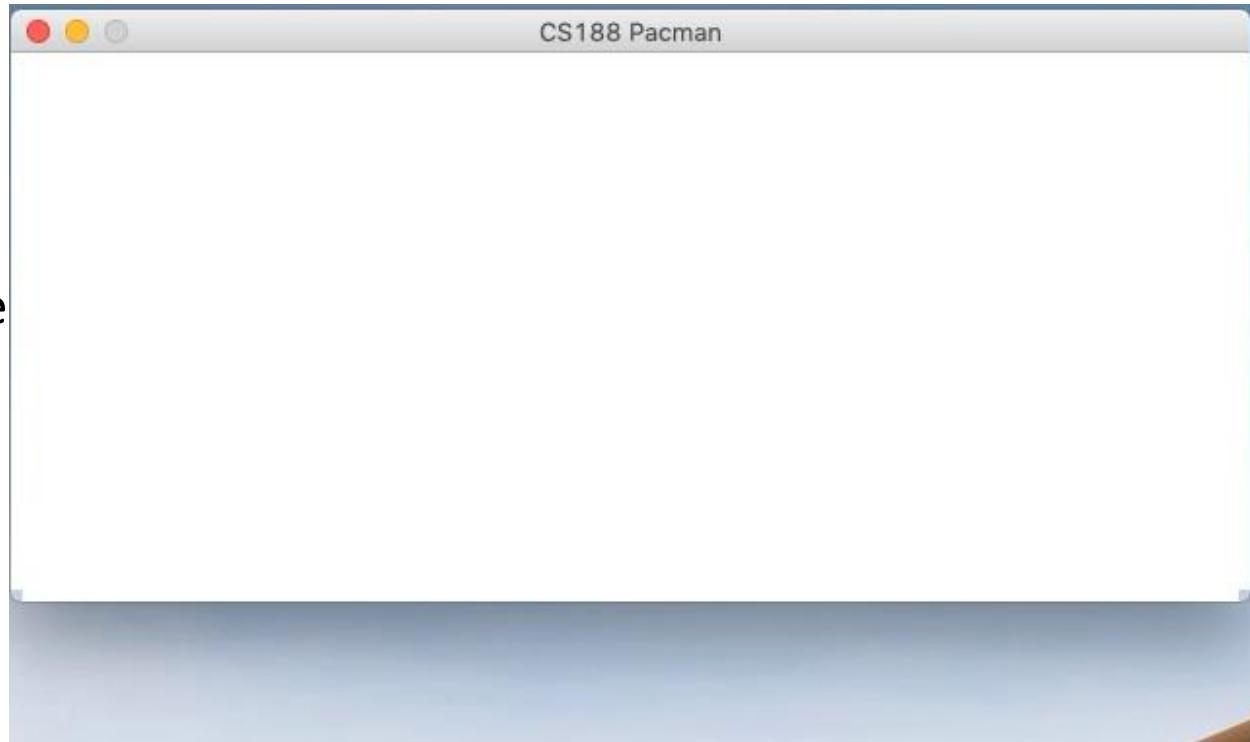
- Naïve Q-learning
- After 50 training episodes



- 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.,  $\epsilon$ -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]

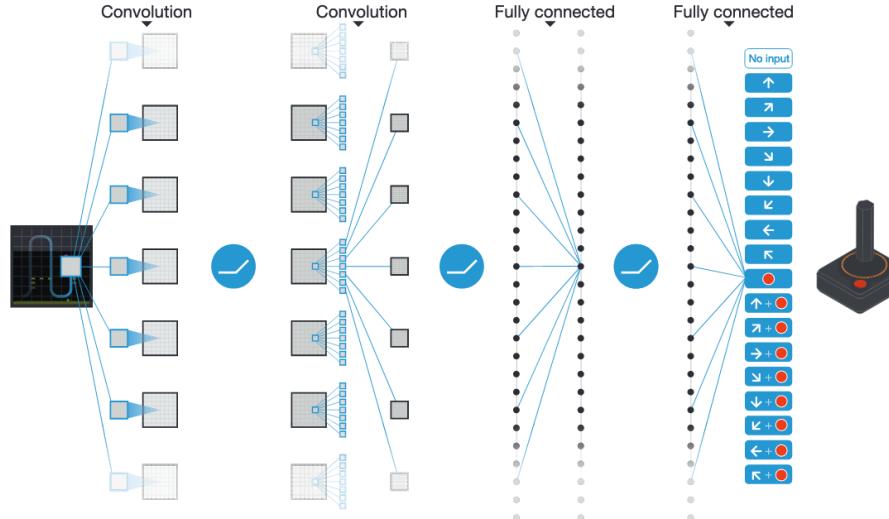
- DQN addresses problems of correlated data and non-stationary distributions
  - Use an experience replay mechanism
  - Randomly samples and trains on previous transitions
  - Results in a smoother training distribution over many past behaviors

# 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
  - $\theta_{i-1}$  is considered fix when optimizing the loss function (helps when taking the derivative with respect to  $\theta$  and with stability)

# Deep Q learning [Mnih et al. 2015]

- $L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[ (y_i - \hat{Q}(s, a; \theta_i))^2 \right]$  Independent of  $\theta_i$  (because  $\theta_{i-1}$  is considered fix )
- $\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot), s' \sim \mathcal{E}} [(r + \gamma \max_{a'} \hat{Q}(s', a'; \theta_{i-1}) - \hat{Q}(s, a; \theta_i)) \nabla_{\theta_i} \hat{Q}(s, a; \theta_i)]$



# Q-learning with experience replay

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

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  $\varepsilon$  select a random action  $a_t$

        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

        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  $\theta$

        Every  $C$  steps reset  $\hat{Q} = Q$

**End For**

**End For**

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

# Q-learning with experience replay

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$

**For** episode = 1,  $M$  **do** ← Play  $m$  episodes (full games)

    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$

**For**  $t = 1, T$  **do**

        With probability  $\varepsilon$  select a random action  $a_t$

        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

        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  $\theta$

        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  $\theta$

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  $\varepsilon$  select a random action  $a_t$

        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

        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  $\theta$

        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  $\theta$

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** ← For each time step during the episode

        With probability  $\varepsilon$  select a random action  $a_t$

        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

        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  $\theta$

        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  $\theta$

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  $\varepsilon$  select a random action  $a_t$   
    otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

    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  $\theta$

    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  $\theta$

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  $\varepsilon$  select a random action  $a_t$

    otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

    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  $\theta$

    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  $\theta$

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  $\varepsilon$  select a random action  $a_t$

    otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

    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  $\theta$

    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  $\theta$

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  $\varepsilon$  select a random action  $a_t$

        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

        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  $\theta$

        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  $\theta$

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  $\varepsilon$  select a random action  $a_t$

    otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

    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  $\theta$

    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,  $a = \underset{a}{\operatorname{argmax}} Q(s, a; \theta)$ , 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

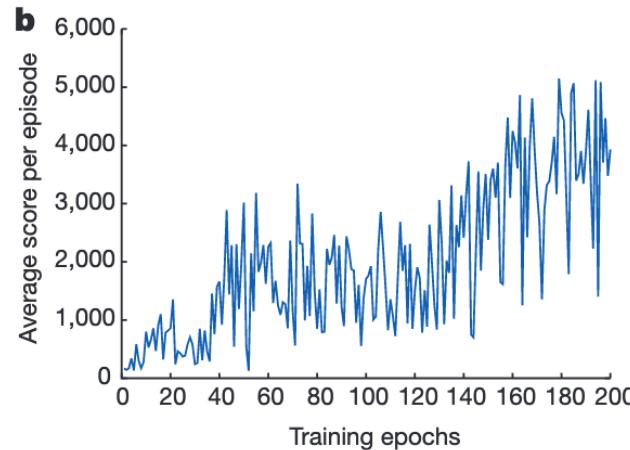
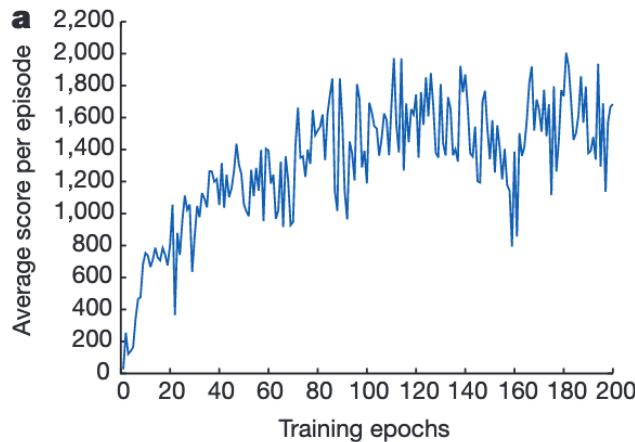
- DQN only stores the last N experience tuples in the replay memory
  - Old transitions are overwritten
- Samples uniformly at random from the buffer when performing updates
- Is there room for improvement?
  - Important transitions?
    - Prioritized sweeping
  - Prioritize deletions from the replay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

**Before training  
peaceful swimming**



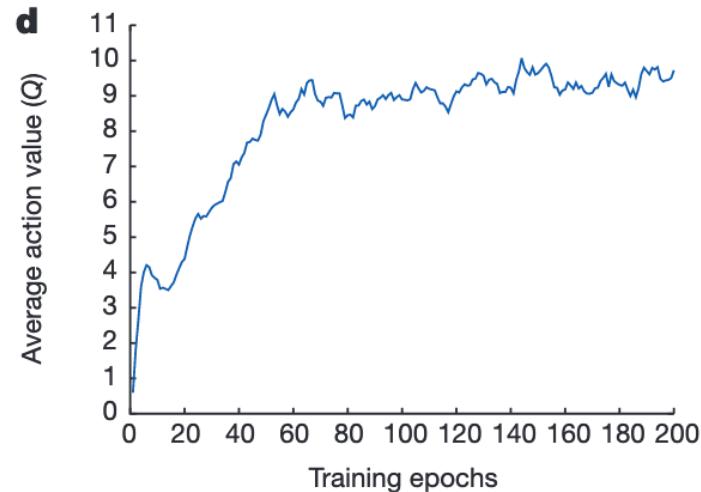
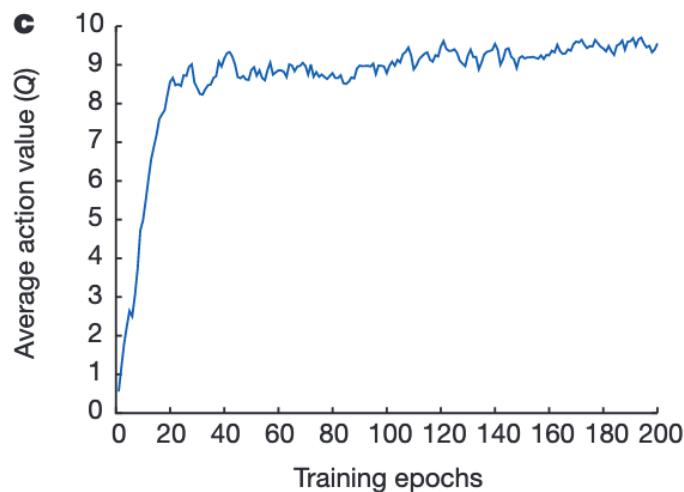
# Results: DQN

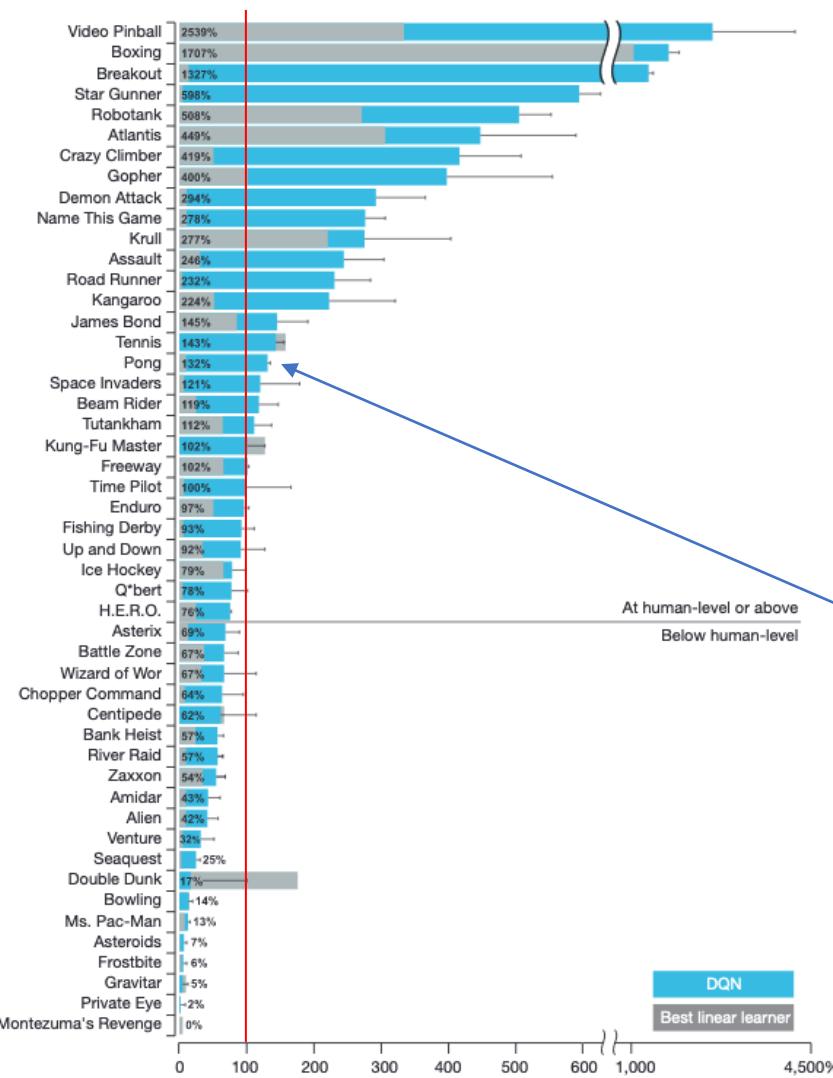
- Each point is the average score achieved per episode after the agent is run with an epsilon-greedy policy ( $\varepsilon = 0.05$ ) for 520k frames on SpaceInvaders (a) and Seaquest (b)



# 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 from the other

# Double Deep Q networks (DDQN)

Initialize replay memory  $D$  to capacity  $N$

Initialize action-value function  $Q$  with random weights  $\theta$

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  $\varepsilon$  select a random action  $a_t$

    otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

    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  $\theta$

    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  $\theta$

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  $\varepsilon$  select a random action  $a_t$

        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

        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  $\theta$

        Every  $C$  steps reset  $\hat{Q} = Q$

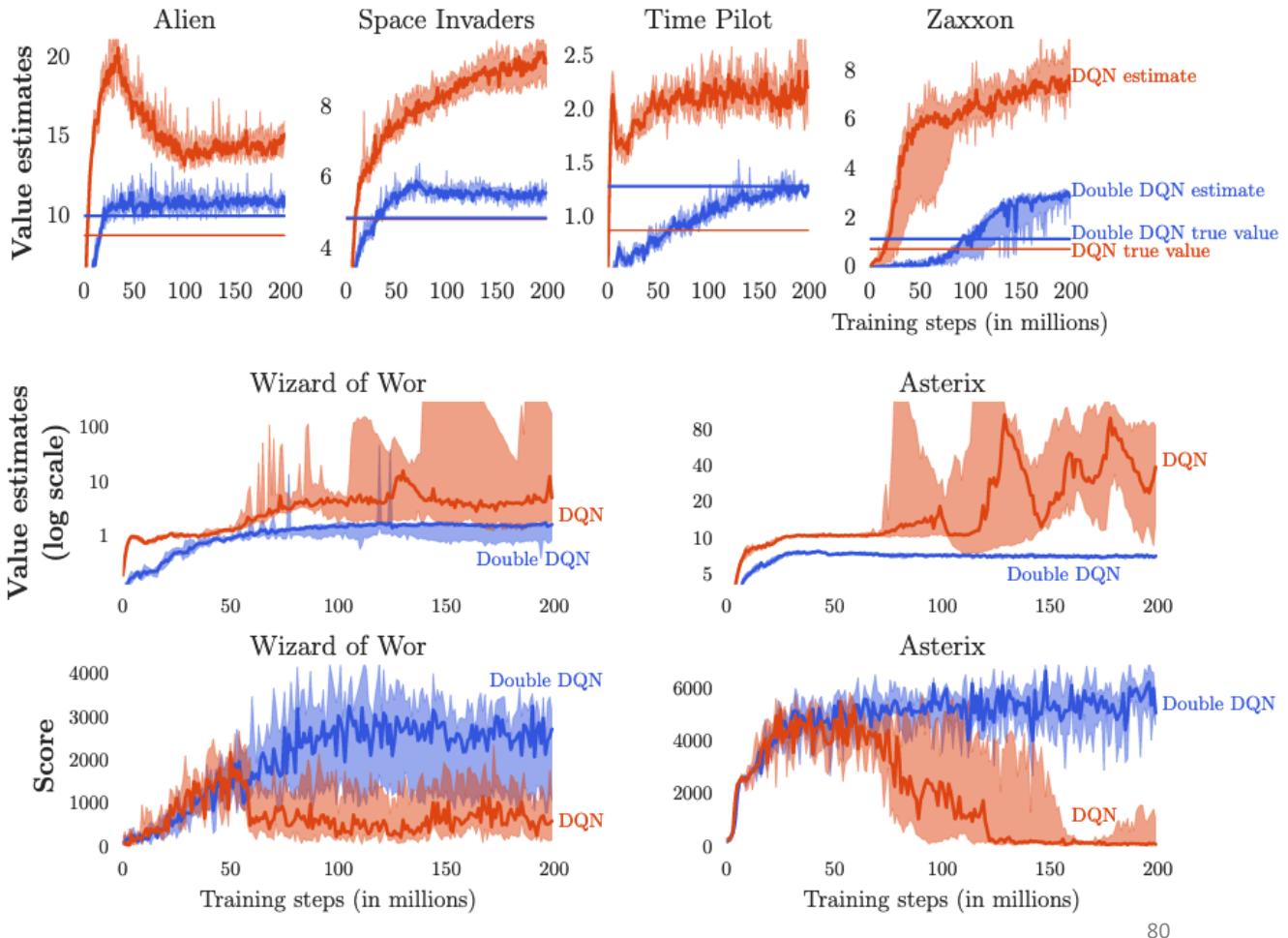
**End For**

**End For**

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

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

- 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, 2<sup>nd</sup> Ed., Sutton & Barto



**BITS** Pilani  
Pilani | Dubai | Goa | Hyderabad

Thank you

# Human-level control through deep reinforcement learning

Volodymyr Mnih<sup>1\*</sup>, Koray Kavukcuoglu<sup>1\*</sup>, David Silver<sup>1\*</sup>, Andrei A. Rusu<sup>1</sup>, Joel Veness<sup>1</sup>, Marc G. Bellemare<sup>1</sup>, Alex Graves<sup>1</sup>, Martin Riedmiller<sup>1</sup>, Andreas K. Fidjeland<sup>1</sup>, Georg Ostrovski<sup>1</sup>, Stig Petersen<sup>1</sup>, Charles Beattie<sup>1</sup>, Amir Sadik<sup>1</sup>, Ioannis Antonoglou<sup>1</sup>, Helen King<sup>1</sup>, Dharshan Kumaran<sup>1</sup>, Daan Wierstra<sup>1</sup>, Shane Legg<sup>1</sup> & Demis Hassabis<sup>1</sup>

The theory of reinforcement learning provides a normative account<sup>1</sup>, deeply rooted in psychological<sup>2</sup> and neuroscientific<sup>3</sup> perspectives on animal behaviour, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations. Remarkably, humans and other animals seem to solve this problem through a harmonious combination of reinforcement learning and hierarchical sensory processing systems<sup>4,5</sup>, the former evidenced by a wealth of neural data revealing notable parallels between the phasic signals emitted by dopaminergic neurons and temporal difference reinforcement learning algorithms<sup>3</sup>. While reinforcement learning agents have achieved some successes in a variety of domains<sup>6–8</sup>, their applicability has previously been limited to domains in which useful features can be handcrafted, or to domains with fully observed, low-dimensional state spaces. Here we use recent advances in training deep neural networks<sup>9–11</sup> to develop a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. We tested this agent on the challenging domain of classic Atari 2600 games<sup>12</sup>. We demonstrate that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks.

We set out to create a single algorithm that would be able to develop a wide range of competencies on a varied range of challenging tasks—a central goal of general artificial intelligence<sup>13</sup> that has eluded previous efforts<sup>8,14,15</sup>. To achieve this, we developed a novel agent, a deep Q-network (DQN), which is able to combine reinforcement learning with a class of artificial neural network<sup>16</sup> known as deep neural networks. Notably, recent advances in deep neural networks<sup>9–11</sup>, in which several layers of nodes are used to build up progressively more abstract representations of the data, have made it possible for artificial neural networks to learn concepts such as object categories directly from raw sensory data. We use one particularly successful architecture, the deep convolutional network<sup>17</sup>, which uses hierarchical layers of tiled convolutional filters to mimic the effects of receptive fields—inspired by Hubel and Wiesel's seminal work on feedforward processing in early visual cortex<sup>18</sup>—thereby exploiting the local spatial correlations present in images, and building in robustness to natural transformations such as changes of viewpoint or scale.

We consider tasks in which the agent interacts with an environment through a sequence of observations, actions and rewards. The goal of the

agent is to select actions in a fashion that maximizes cumulative future reward. More formally, we use a deep convolutional neural network to approximate the optimal action-value function

$$Q^*(s,a) = \max_{\pi} \mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots | s_t = s, a_t = a, \pi],$$

which is the maximum sum of rewards  $r_t$  discounted by  $\gamma$  at each time-step  $t$ , achievable by a behaviour policy  $\pi = P(a|s)$ , after making an observation ( $s$ ) and taking an action ( $a$ ) (see Methods)<sup>19</sup>.

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as  $Q$ ) function<sup>20</sup>. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to  $Q$  may significantly change the policy and therefore change the data distribution, and the correlations between the action-values ( $Q$ ) and the target values  $r + \gamma \max_{a'} Q(s', a')$ . We address these instabilities with a novel variant of Q-learning, which uses two key ideas. First, we used a biologically inspired mechanism termed experience replay<sup>21–23</sup> that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution (see below for details). Second, we used an iterative update that adjusts the action-values ( $Q$ ) towards target values that are only periodically updated, thereby reducing correlations with the target.

While other stable methods exist for training neural networks in the reinforcement learning setting, such as neural fitted Q-iteration<sup>24</sup>, these methods involve the repeated training of networks *de novo* on hundreds of iterations. Consequently, these methods, unlike our algorithm, are too inefficient to be used successfully with large neural networks. We parameterize an approximate value function  $Q(s,a;\theta_i)$  using the deep convolutional neural network shown in Fig. 1, in which  $\theta_i$  are the parameters (that is, weights) of the Q-network at iteration  $i$ . To perform experience replay we store the agent's experiences  $e_t = (s_t, a_t, r_t, s_{t+1})$  at each time-step  $t$  in a data set  $D_t = \{e_1, \dots, e_t\}$ . During learning, we apply Q-learning updates, on samples (or minibatches) of experience  $(s, a, r, s') \sim U(D)$ , drawn uniformly at random from the pool of stored samples. The Q-learning update at iteration  $i$  uses the following loss function:

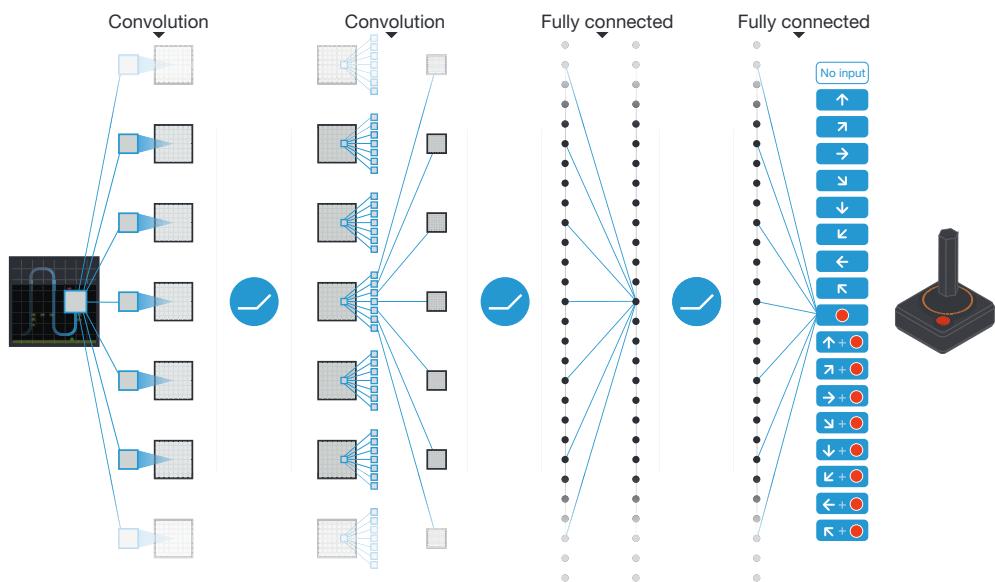
$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[ \left( r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right]$$

in which  $\gamma$  is the discount factor determining the agent's horizon,  $\theta_i$  are the parameters of the Q-network at iteration  $i$  and  $\theta_i^-$  are the network parameters used to compute the target at iteration  $i$ . The target network parameters  $\theta_i^-$  are only updated with the Q-network parameters ( $\theta_i$ ) every  $C$  steps and are held fixed between individual updates (see Methods).

To evaluate our DQN agent, we took advantage of the Atari 2600 platform, which offers a diverse array of tasks ( $n = 49$ ) designed to be

<sup>1</sup>Google DeepMind, 5 New Street Square, London EC4A 3TW, UK.

\*These authors contributed equally to this work.

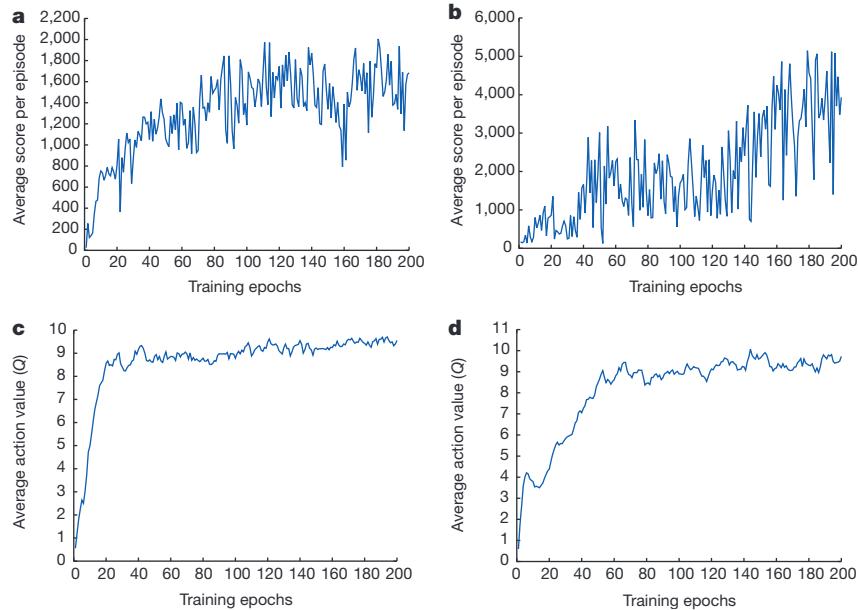


**Figure 1 | Schematic illustration of the convolutional neural network.** The details of the architecture are explained in the Methods. The input to the neural network consists of an  $84 \times 84 \times 4$  image produced by the preprocessing map  $\phi$ , followed by three convolutional layers (note: snaking blue line

difficult and engaging for human players. We used the same network architecture, hyperparameter values (see Extended Data Table 1) and learning procedure throughout—taking high-dimensional data ( $210 \times 160$  colour video at 60 Hz) as input—to demonstrate that our approach robustly learns successful policies over a variety of games based solely on sensory inputs with only very minimal prior knowledge (that is, merely the input data were visual images, and the number of actions available in each game, but not their correspondences; see Methods). Notably, our method was able to train large neural networks using a reinforcement learning signal and stochastic gradient descent in a stable manner—illustrated by the temporal evolution of two indices of learning (the agent’s average score-per-episode and average predicted Q-values; see Fig. 2 and Supplementary Discussion for details).

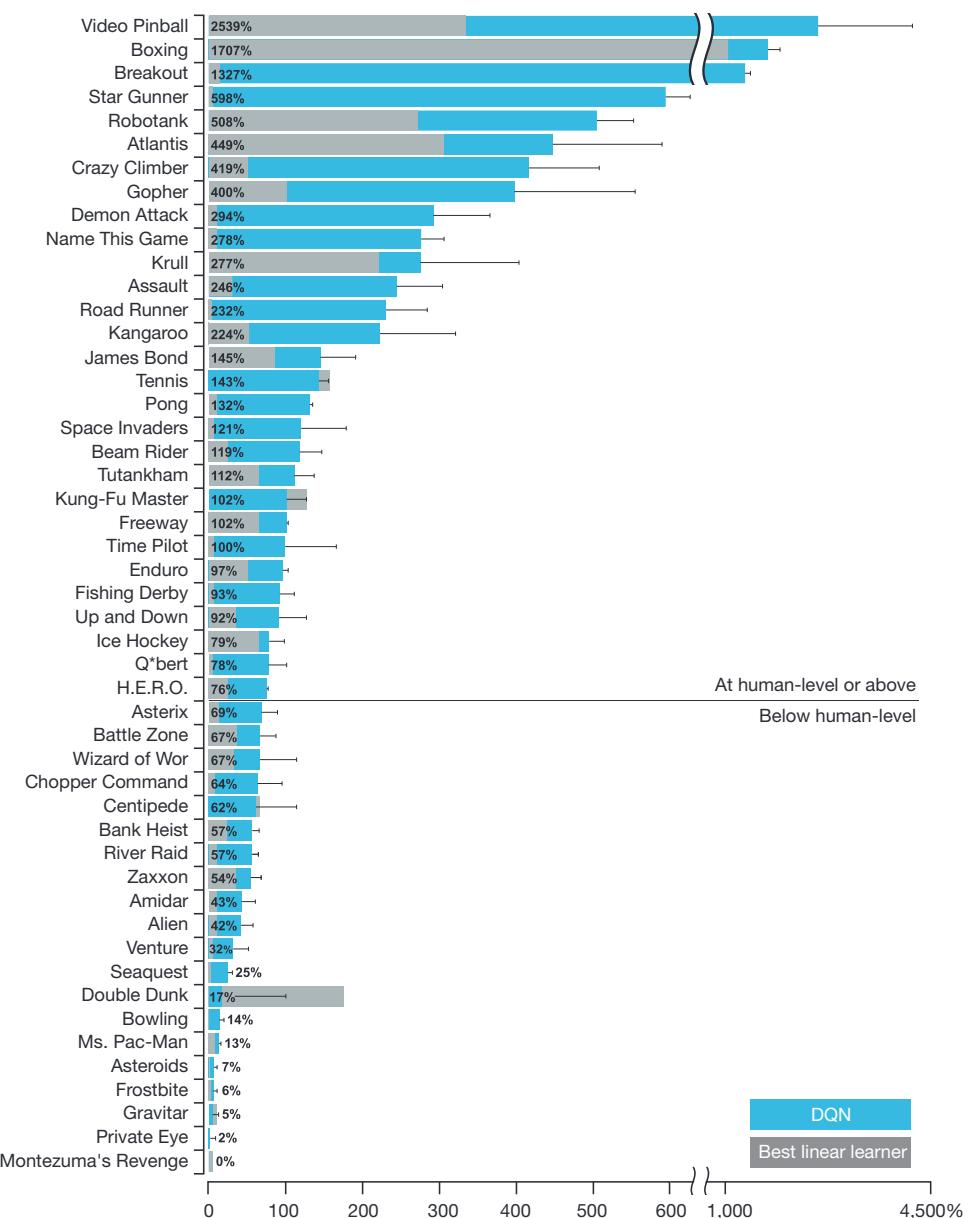
symbolizes sliding of each filter across input image) and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is,  $\max(0, x)$ ).

We compared DQN with the best performing methods from the reinforcement learning literature on the 49 games where results were available<sup>12,15</sup>. In addition to the learned agents, we also report scores for a professional human games tester playing under controlled conditions and a policy that selects actions uniformly at random (Extended Data Table 2 and Fig. 3, denoted by 100% (human) and 0% (random) on  $y$  axis; see Methods). Our DQN method outperforms the best existing reinforcement learning methods on 43 of the games without incorporating any of the additional prior knowledge about Atari 2600 games used by other approaches (for example, refs 12, 15). Furthermore, our DQN agent performed at a level that was comparable to that of a professional human games tester across the set of 49 games, achieving more than 75% of the human score on more than half of the games (29 games);



**Figure 2 | Training curves tracking the agent’s average score and average predicted action-value.** **a**, Each point is the average score achieved per episode after the agent is run with  $\varepsilon$ -greedy policy ( $\varepsilon = 0.05$ ) for 520 k frames on Space Invaders. **b**, Average score achieved per episode for Seaquest. **c**, Average predicted action-value on a held-out set of states on Space Invaders. Each point

on the curve is the average of the action-value  $Q$  computed over the held-out set of states. Note that  $Q$ -values are scaled due to clipping of rewards (see Methods). **d**, Average predicted action-value on Seaquest. See Supplementary Discussion for details.



**Figure 3 | Comparison of the DQN agent with the best reinforcement learning methods<sup>15</sup> in the literature.** The performance of DQN is normalized with respect to a professional human games tester (that is, 100% level) and random play (that is, 0% level). Note that the normalized performance of DQN, expressed as a percentage, is calculated as:  $100 \times (\text{DQN score} - \text{random play score}) / (\text{human score} - \text{random play score})$ . It can be seen that DQN

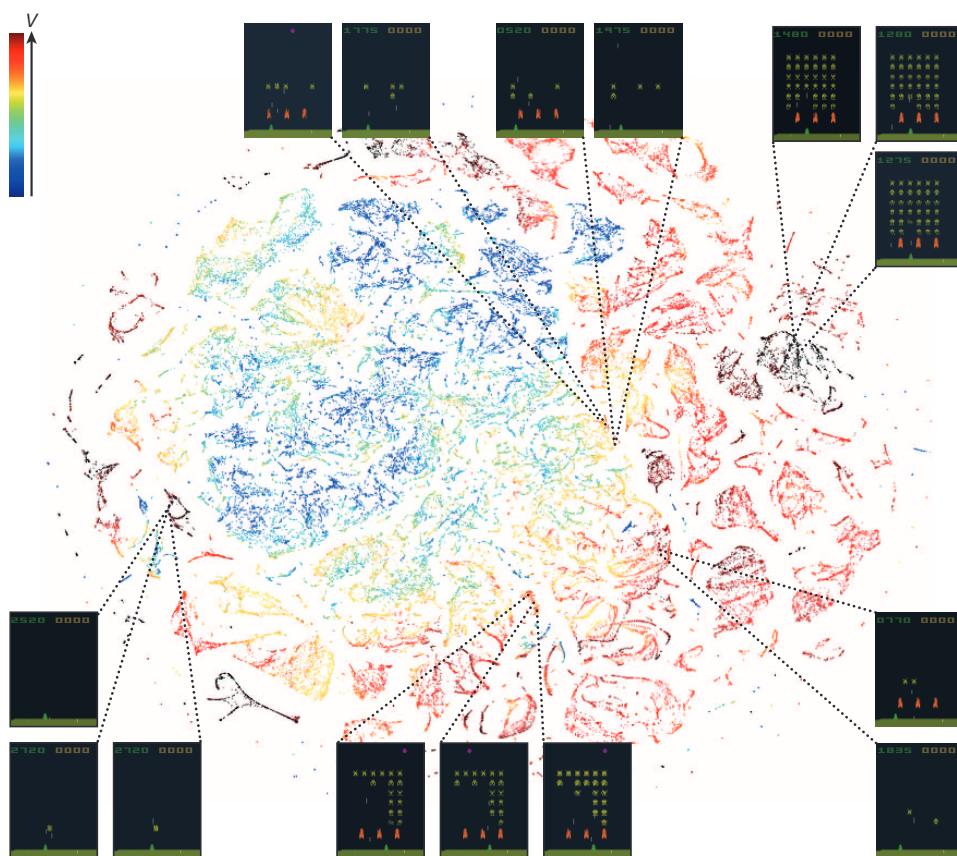
outperforms competing methods (also see Extended Data Table 2) in almost all the games, and performs at a level that is broadly comparable with or superior to a professional human games tester (that is, operationalized as a level of 75% or above) in the majority of games. Audio output was disabled for both human players and agents. Error bars indicate s.d. across the 30 evaluation episodes, starting with different initial conditions.

see Fig. 3, Supplementary Discussion and Extended Data Table 2). In additional simulations (see Supplementary Discussion and Extended Data Tables 3 and 4), we demonstrate the importance of the individual core components of the DQN agent—the replay memory, separate target Q-network and deep convolutional network architecture—by disabling them and demonstrating the detrimental effects on performance.

We next examined the representations learned by DQN that underpinned the successful performance of the agent in the context of the game Space Invaders (see Supplementary Video 1 for a demonstration of the performance of DQN), by using a technique developed for the visualization of high-dimensional data called ‘t-SNE’<sup>25</sup> (Fig. 4). As expected, the t-SNE algorithm tends to map the DQN representation of perceptually similar states to nearby points. Interestingly, we also found instances in which the t-SNE algorithm generated similar embeddings for DQN representations of states that are close in terms of expected reward but

perceptually dissimilar (Fig. 4, bottom right, top left and middle), consistent with the notion that the network is able to learn representations that support adaptive behaviour from high-dimensional sensory inputs. Furthermore, we also show that the representations learned by DQN are able to generalize to data generated from policies other than its own—in simulations where we presented as input to the network game states experienced during human and agent play, recorded the representations of the last hidden layer, and visualized the embeddings generated by the t-SNE algorithm (Extended Data Fig. 1 and Supplementary Discussion). Extended Data Fig. 2 provides an additional illustration of how the representations learned by DQN allow it to accurately predict state and action values.

It is worth noting that the games in which DQN excels are extremely varied in their nature, from side-scrolling shooters (River Raid) to boxing games (Boxing) and three-dimensional car-racing games (Enduro).



**Figure 4 | Two-dimensional t-SNE embedding of the representations in the last hidden layer assigned by DQN to game states experienced while playing Space Invaders.** The plot was generated by letting the DQN agent play for 2 h of real game time and running the t-SNE algorithm<sup>25</sup> on the last hidden layer representations assigned by DQN to each experienced game state. The points are coloured according to the state values ( $V$ , maximum expected reward of a state) predicted by DQN for the corresponding game states (ranging from dark red (highest  $V$ ) to dark blue (lowest  $V$ )). The screenshots corresponding to a selected number of points are shown. The DQN agent

predicts high state values for both full (top right screenshots) and nearly complete screens (bottom left screenshots) because it has learned that completing a screen leads to a new screen full of enemy ships. Partially completed screens (bottom screenshots) are assigned lower state values because less immediate reward is available. The screens shown on the bottom right and top left and middle are less perceptually similar than the other examples but are still mapped to nearby representations and similar values because the orange bunkers do not carry great significance near the end of a level. With permission from Square Enix Limited.

Indeed, in certain games DQN is able to discover a relatively long-term strategy (for example, Breakout: the agent learns the optimal strategy, which is to first dig a tunnel around the side of the wall allowing the ball to be sent around the back to destroy a large number of blocks; see Supplementary Video 2 for illustration of development of DQN's performance over the course of training). Nevertheless, games demanding more temporally extended planning strategies still constitute a major challenge for all existing agents including DQN (for example, Montezuma's Revenge).

In this work, we demonstrate that a single architecture can successfully learn control policies in a range of different environments with only very minimal prior knowledge, receiving only the pixels and the game score as inputs, and using the same algorithm, network architecture and hyperparameters on each game, privy only to the inputs a human player would have. In contrast to previous work<sup>24,26</sup>, our approach incorporates 'end-to-end' reinforcement learning that uses reward to continuously shape representations within the convolutional network towards salient features of the environment that facilitate value estimation. This principle draws on neurobiological evidence that reward signals during perceptual learning may influence the characteristics of representations within primate visual cortex<sup>27,28</sup>. Notably, the successful integration of reinforcement learning with deep network architectures was critically dependent on our incorporation of a replay algorithm<sup>21–23</sup> involving the storage and representation of recently experienced transitions. Convergent evidence suggests that the hippocampus may support the physical

realization of such a process in the mammalian brain, with the time-compressed reactivation of recently experienced trajectories during offline periods<sup>21,22</sup> (for example, waking rest) providing a putative mechanism by which value functions may be efficiently updated through interactions with the basal ganglia<sup>22</sup>. In the future, it will be important to explore the potential use of biasing the content of experience replay towards salient events, a phenomenon that characterizes empirically observed hippocampal replay<sup>29</sup>, and relates to the notion of 'prioritized sweeping'<sup>30</sup> in reinforcement learning. Taken together, our work illustrates the power of harnessing state-of-the-art machine learning techniques with biologically inspired mechanisms to create agents that are capable of learning to master a diverse array of challenging tasks.

**Online Content** Methods, along with any additional Extended Data display items and Source Data, are available in the online version of the paper; references unique to these sections appear only in the online paper.

Received 10 July 2014; accepted 16 January 2015.

1. Sutton, R. & Barto, A. Reinforcement Learning: An Introduction (MIT Press, 1998).
2. Thorndike, E. L. Animal Intelligence: Experimental studies (Macmillan, 1911).
3. Schultz, W., Dayan, P. & Montague, P. R. A neural substrate of prediction and reward. *Science* **275**, 1593–1599 (1997).
4. Serre, T., Wolf, L. & Poggio, T. Object recognition with features inspired by visual cortex. *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern. Recognit.* 994–1000 (2005).
5. Fukushima, K. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biol. Cybern.* **36**, 193–202 (1980).

6. Tesauro, G. Temporal difference learning and TD-Gammon. *Commun. ACM* **38**, 58–68 (1995).
7. Riedmiller, M., Gabel, T., Hafner, R. & Lange, S. Reinforcement learning for robot soccer. *Auton. Robots* **27**, 55–73 (2009).
8. Diuk, C., Cohen, A. & Littman, M. L. An object-oriented representation for efficient reinforcement learning. *Proc. Int. Conf. Mach. Learn.* 240–247 (2008).
9. Bengio, Y. Learning deep architectures for AI. *Foundations and Trends in Machine Learning* **2**, 1–127 (2009).
10. Krizhevsky, A., Sutskever, I. & Hinton, G. ImageNet classification with deep convolutional neural networks. *Adv. Neural Inf. Process. Syst.* **25**, 1106–1114 (2012).
11. Hinton, G. E. & Salakhutdinov, R. R. Reducing the dimensionality of data with neural networks. *Science* **313**, 504–507 (2006).
12. Bellemare, M. G., Naddaf, Y., Veness, J. & Bowling, M. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.* **47**, 253–279 (2013).
13. Legg, S. & Hutter, M. Universal Intelligence: a definition of machine intelligence. *Minds Mach.* **17**, 391–444 (2007).
14. Genesereth, M., Love, N. & Pell, B. General game playing: overview of the AAAI competition. *AI Mag.* **26**, 62–72 (2005).
15. Bellemare, M. G., Veness, J. & Bowling, M. Investigating contingency awareness using Atari 2600 games. *Proc. Conf. AAAI Artif. Intell.* 864–871 (2012).
16. McClelland, J. L., Rumelhart, D. E. & Group, T. P. R. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* (MIT Press, 1986).
17. LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **86**, 2278–2324 (1998).
18. Hubel, D. H. & Wiesel, T. N. Shape and arrangement of columns in cat's striate cortex. *J. Physiol.* **165**, 559–568 (1963).
19. Watkins, C. J. & Dayan, P. Q-learning. *Mach. Learn.* **8**, 279–292 (1992).
20. Tsitsiklis, J. & Roy, B. V. An analysis of temporal-difference learning with function approximation. *IEEE Trans. Automat. Contr.* **42**, 674–690 (1997).
21. McClelland, J. L., McNaughton, B. L. & O'Reilly, R. C. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychol. Rev.* **102**, 419–457 (1995).
22. O'Neill, J., Pleydell-Bouverie, B., Dupret, D. & Csicsvari, J. Play it again: reactivation of waking experience and memory. *Trends Neurosci.* **33**, 220–229 (2010).
23. Lin, L.-J. Reinforcement learning for robots using neural networks. Technical Report, DTIC Document (1993).
24. Riedmiller, M. Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method. *Mach. Learn.: ECML*, **3720**, 317–328 (Springer, 2005).
25. Van der Maaten, L. J. P. & Hinton, G. E. Visualizing high-dimensional data using t-SNE. *J. Mach. Learn. Res.* **9**, 2579–2605 (2008).
26. Lange, S. & Riedmiller, M. Deep auto-encoder neural networks in reinforcement learning. *Proc. Int. Jt. Conf. Neural. Netw.* 1–8 (2010).
27. Law, C.-T. & Gold, J. I. Reinforcement learning can account for associative and perceptual learning on a visual decision task. *Nature Neurosci.* **12**, 655 (2009).
28. Sigala, N. & Logothetis, N. K. Visual categorization shapes feature selectivity in the primate temporal cortex. *Nature* **415**, 318–320 (2002).
29. Bendor, D. & Wilson, M. A. Biasing the content of hippocampal replay during sleep. *Nature Neurosci.* **15**, 1439–1444 (2012).
30. Moore, A. & Atkeson, C. Prioritized sweeping: reinforcement learning with less data and less real time. *Mach. Learn.* **13**, 103–130 (1993).

**Supplementary Information** is available in the online version of the paper.

**Acknowledgements** We thank G. Hinton, P. Dayan and M. Bowling for discussions, A. Cain and J. Keene for work on the visuals, K. Keller and P. Rogers for help with the visuals, G. Wayne for comments on an earlier version of the manuscript, and the rest of the DeepMind team for their support, ideas and encouragement.

**Author Contributions** V.M., K.K., D.S., J.V., M.G.B., M.R., A.G., D.W., S.L. and D.H. conceptualized the problem and the technical framework. V.M., K.K., A.A.R. and D.S. developed and tested the algorithms. J.V., S.P., C.B., A.A.R., M.G.B., I.A., A.K.F., G.O. and A.S. created the testing platform. K.K., H.K., S.L. and D.H. managed the project. K.K., D.K., D.H., V.M., D.S., A.G., A.A.R., J.V. and M.G.B. wrote the paper.

**Author Information** Reprints and permissions information is available at [www.nature.com/reprints](http://www.nature.com/reprints). The authors declare no competing financial interests. Readers are welcome to comment on the online version of the paper. Correspondence and requests for materials should be addressed to K.K. ([korayk@google.com](mailto:korayk@google.com)) or D.H. ([demishassabis@google.com](mailto:demishassabis@google.com)).

## METHODS

**Preprocessing.** Working directly with raw Atari 2600 frames, which are  $210 \times 160$  pixel images with a 128-colour palette, can be demanding in terms of computation and memory requirements. We apply a basic preprocessing step aimed at reducing the input dimensionality and dealing with some artefacts of the Atari 2600 emulator. First, to encode a single frame we take the maximum value for each pixel colour value over the frame being encoded and the previous frame. This was necessary to remove flickering that is present in games where some objects appear only in even frames while other objects appear only in odd frames, an artefact caused by the limited number of sprites Atari 2600 can display at once. Second, we then extract the Y channel, also known as luminance, from the RGB frame and rescale it to  $84 \times 84$ . The function  $\phi$  from algorithm 1 described below applies this preprocessing to the  $m$  most recent frames and stacks them to produce the input to the Q-function, in which  $m = 4$ , although the algorithm is robust to different values of  $m$  (for example, 3 or 5).

**Code availability.** The source code can be accessed at <https://sites.google.com/a/deepmind.com/dqn> for non-commercial uses only.

**Model architecture.** There are several possible ways of parameterizing Q using a neural network. Because Q maps history-action pairs to scalar estimates of their Q-value, the history and the action have been used as inputs to the neural network by some previous approaches<sup>24,26</sup>. The main drawback of this type of architecture is that a separate forward pass is required to compute the Q-value of each action, resulting in a cost that scales linearly with the number of actions. We instead use an architecture in which there is a separate output unit for each possible action, and only the state representation is an input to the neural network. The outputs correspond to the predicted Q-values of the individual actions for the input state. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network.

The exact architecture, shown schematically in Fig. 1, is as follows. The input to the neural network consists of an  $84 \times 84 \times 4$  image produced by the preprocessing map  $\phi$ . The first hidden layer convolves 32 filters of  $8 \times 8$  with stride 4 with the input image and applies a rectifier nonlinearity<sup>31,32</sup>. The second hidden layer convolves 64 filters of  $4 \times 4$  with stride 2, again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that convolves 64 filters of  $3 \times 3$  with stride 1 followed by a rectifier. The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. The number of valid actions varied between 4 and 18 on the games we considered.

**Training details.** We performed experiments on 49 Atari 2600 games where results were available for all other comparable methods<sup>12,15</sup>. A different network was trained on each game: the same network architecture, learning algorithm and hyperparameter settings (see Extended Data Table 1) were used across all games, showing that our approach is robust enough to work on a variety of games while incorporating only minimal prior knowledge (see below). While we evaluated our agents on unmodified games, we made one change to the reward structure of the games during training only. As the scale of scores varies greatly from game to game, we clipped all positive rewards at 1 and all negative rewards at  $-1$ , leaving 0 rewards unchanged. Clipping the rewards in this manner limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games. At the same time, it could affect the performance of our agent since it cannot differentiate between rewards of different magnitude. For games where there is a life counter, the Atari 2600 emulator also sends the number of lives left in the game, which is then used to mark the end of an episode during training.

In these experiments, we used the RMSProp (see [http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture\\_slides\\_lec6.pdf](http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf)) algorithm with minibatches of size 32. The behaviour policy during training was  $\epsilon$ -greedy with  $\epsilon$  annealed linearly from 1.0 to 0.1 over the first million frames, and fixed at 0.1 thereafter. We trained for a total of 50 million frames (that is, around 38 days of game experience in total) and used a replay memory of 1 million most recent frames.

Following previous approaches to playing Atari 2600 games, we also use a simple frame-skipping technique<sup>15</sup>. More precisely, the agent sees and selects actions on every  $k$ th frame instead of every frame, and its last action is repeated on skipped frames. Because running the emulator forward for one step requires much less computation than having the agent select an action, this technique allows the agent to play roughly  $k$  times more games without significantly increasing the runtime. We use  $k = 4$  for all games.

The values of all the hyperparameters and optimization parameters were selected by performing an informal search on the games Pong, Breakout, Seaquest, Space Invaders and Beam Rider. We did not perform a systematic grid search owing to the high computational cost. These parameters were then held fixed across all other games. The values and descriptions of all hyperparameters are provided in Extended Data Table 1.

Our experimental setup amounts to using the following minimal prior knowledge: that the input data consisted of visual images (motivating our use of a convolutional deep network), the game-specific score (with no modification), number of actions, although not their correspondences (for example, specification of the up ‘button’) and the life count.

**Evaluation procedure.** The trained agents were evaluated by playing each game 30 times for up to 5 min each time with different initial random conditions (‘noop’; see Extended Data Table 1) and an  $\epsilon$ -greedy policy with  $\epsilon = 0.05$ . This procedure is adopted to minimize the possibility of overfitting during evaluation. The random agent served as a baseline comparison and chose a random action at 10 Hz which is every sixth frame, repeating its last action on intervening frames. 10 Hz is about the fastest that a human player can select the ‘fire’ button, and setting the random agent to this frequency avoids spurious baseline scores in a handful of the games. We did also assess the performance of a random agent that selected an action at 60 Hz (that is, every frame). This had a minimal effect: changing the normalized DQN performance by more than 5% in only six games (Boxing, Breakout, Crazy Climber, Demon Attack, Krull and Robotank), and in all these games DQN outperformed the expert human by a considerable margin.

The professional human tester used the same emulator engine as the agents, and played under controlled conditions. The human tester was not allowed to pause, save or reload games. As in the original Atari 2600 environment, the emulator was run at 60 Hz and the audio output was disabled: as such, the sensory input was equated between human player and agents. The human performance is the average reward achieved from around 20 episodes of each game lasting a maximum of 5 min each, following around 2 h of practice playing each game.

**Algorithm.** We consider tasks in which an agent interacts with an environment, in this case the Atari emulator, in a sequence of actions, observations and rewards. At each time-step the agent selects an action  $a_t$  from the set of legal game actions,  $\mathcal{A} = \{1, \dots, K\}$ . The action is passed to the emulator and modifies its internal state and the game score. In general the environment may be stochastic. The emulator’s internal state is not observed by the agent; instead the agent observes an image  $x_t \in \mathbb{R}^d$  from the emulator, which is a vector of pixel values representing the current screen. In addition it receives a reward  $r_t$ , representing the change in game score. Note that in general the game score may depend on the whole previous sequence of actions and observations; feedback about an action may only be received after many thousands of time-steps have elapsed.

Because the agent only observes the current screen, the task is partially observed<sup>33</sup> and many emulator states are perceptually aliased (that is, it is impossible to fully understand the current situation from only the current screen  $x_t$ ). Therefore, sequences of actions and observations,  $s_t = x_1, a_1, x_2, \dots, a_{t-1}, x_t$ , are input to the algorithm, which then learns game strategies depending upon these sequences. All sequences in the emulator are assumed to terminate in a finite number of time-steps. This formalism gives rise to a large but finite Markov decision process (MDP) in which each sequence is a distinct state. As a result, we can apply standard reinforcement learning methods for MDPs, simply by using the complete sequence  $s_t$  as the state representation at time  $t$ .

The goal of the agent is to interact with the emulator by selecting actions in a way that maximizes future rewards. We make the standard assumption that future rewards are discounted by a factor of  $\gamma$  per time-step ( $\gamma$  was set to 0.99 throughout), and define the future discounted return at time  $t$  as  $R_t = \sum_{t'=t}^T \gamma^{t'-t} r_{t'}$ , in which  $T$  is the time-step at which the game terminates. We define the optimal action-value function  $Q^*(s, a)$  as the maximum expected return achievable by following any policy, after seeing some sequence  $s$  and then taking some action  $a$ ,  $Q^*(s, a) = \max_{\pi} \mathbb{E}[R_t | s = s, a = a, \pi]$  in which  $\pi$  is a policy mapping sequences to actions (or distributions over actions).

The optimal action-value function obeys an important identity known as the Bellman equation. This is based on the following intuition: if the optimal value  $Q^*(s', a')$  of the sequence  $s'$  at the next time-step was known for all possible actions  $a'$ , then the optimal strategy is to select the action  $a'$  maximizing the expected value of  $r + \gamma Q^*(s', a')$ :

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

The basic idea behind many reinforcement learning algorithms is to estimate the action-value function by using the Bellman equation as an iterative update,  $Q_{i+1}(s, a) = \mathbb{E}_{s'} [r + \gamma \max_{a'} Q_i(s', a')] \mathbb{Q}_i$ . Such value iteration algorithms converge to the optimal action-value function,  $Q_i \rightarrow Q^*$  as  $i \rightarrow \infty$ . In practice, this basic approach is impractical, because the action-value function is estimated separately for each sequence, without any generalization. Instead, it is common to use a function approximator to estimate the action-value function,  $Q(s, a; \theta) \approx Q^*(s, a)$ . In the reinforcement learning community this is typically a linear function approximator, but

sometimes a nonlinear function approximator is used instead, such as a neural network. We refer to a neural network function approximator with weights  $\theta$  as a Q-network. A Q-network can be trained by adjusting the parameters  $\theta_i$  at iteration  $i$  to reduce the mean-squared error in the Bellman equation, where the optimal target values  $r + \gamma \max_{a'} Q^*(s', a')$  are substituted with approximate target values  $y = r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$ , using parameters  $\theta_i^-$  from some previous iteration. This leads to a sequence of loss functions  $L_i(\theta_i)$  that changes at each iteration  $i$ ,

$$\begin{aligned} L_i(\theta_i) &= \mathbb{E}_{s,a,r} [(y \mathbf{1}_a) - Q(s,a;\theta_i)]^2 \\ &= \mathbb{E}_{s,a,r,s'} [(y - Q(s,a;\theta_i))^2] + \mathbb{E}_{s,a,r} [V_{s'}[y]] : \end{aligned}$$

Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which are fixed before learning begins. At each stage of optimization, we hold the parameters from the previous iteration  $\theta_i^-$  fixed when optimizing the  $i$ th loss function  $L_i(\theta_i)$ , resulting in a sequence of well-defined optimization problems. The final term is the variance of the targets, which does not depend on the parameters  $\theta_i$  that we are currently optimizing, and may therefore be ignored. Differentiating the loss function with respect to the weights we arrive at the following gradient:

$$+_{\theta_i} L(\theta_i) = \mathbb{E}_{s,a,r,s'} \left[ \left( r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s,a;\theta_i) \right) +_{\theta_i} Q(s,a;\theta_i) \right] :$$

Rather than computing the full expectations in the above gradient, it is often computationally expedient to optimize the loss function by stochastic gradient descent. The familiar Q-learning algorithm<sup>19</sup> can be recovered in this framework by updating the weights after every time step, replacing the expectations using single samples, and setting  $\theta_i^- = \theta_{i-1}$ .

Note that this algorithm is model-free: it solves the reinforcement learning task directly using samples from the emulator, without explicitly estimating the reward and transition dynamics  $P(r,s'|a)$ . It is also off-policy: it learns about the greedy policy  $a = \operatorname{argmax}_a Q(s,a;\theta)$ , while following a behaviour distribution that ensures adequate exploration of the state space. In practice, the behaviour distribution is often selected by an  $\varepsilon$ -greedy policy that follows the greedy policy with probability  $1 - \varepsilon$  and selects a random action with probability  $\varepsilon$ .

**Training algorithm for deep Q-networks.** The full algorithm for training deep Q-networks is presented in Algorithm 1. The agent selects and executes actions according to an  $\varepsilon$ -greedy policy based on  $Q$ . Because using histories of arbitrary length as inputs to a neural network can be difficult, our Q-function instead works on a fixed length representation of histories produced by the function  $\phi$  described above. The algorithm modifies standard online Q-learning in two ways to make it suitable for training large neural networks without diverging.

First, we use a technique known as experience replay<sup>23</sup> in which we store the agent's experiences at each time-step,  $e_t = (s_t, a_t, r_t, s_{t+1})$ , in a data set  $D_t = \{e_1, \dots, e_t\}$ , pooled over many episodes (where the end of an episode occurs when a terminal state is reached) into a replay memory. During the inner loop of the algorithm, we apply Q-learning updates, or minibatch updates, to samples of experience,  $(s, a, r, s') \sim U(D)$ , drawn at random from the pool of stored samples. This approach has several advantages over standard online Q-learning. First, each step of experience is potentially used in many weight updates, which allows for greater data efficiency. Second, learning directly from consecutive samples is inefficient, owing to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates. Third, when learning on-policy the current parameters determine the next data sample that the parameters are trained on. For example, if the maximizing action is to move left then the training samples will be dominated by samples from the left-hand side; if the maximizing action then switches to the right then the training distribution will also switch. It is easy to see how unwanted feedback loops may arise and the parameters could get stuck in a poor local minimum, or even diverge catastrophically<sup>20</sup>. By using experience

replay the behaviour 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.

In practice, our algorithm only stores the last  $N$  experience tuples in the replay memory, and samples uniformly at random from  $D$  when performing updates. This approach is in some respects limited because the memory buffer does not differentiate important transitions and always overwrites with recent transitions owing to the finite memory size  $N$ . Similarly, the uniform sampling gives equal importance to all transitions in the replay memory. A more sophisticated sampling strategy might emphasize transitions from which we can learn the most, similar to prioritized sweeping<sup>30</sup>.

The second modification to online Q-learning aimed at further improving the stability of our method with neural networks is to use a separate network for generating the targets  $y_j$  in the Q-learning update. More precisely, every  $C$  updates we clone the network  $Q$  to obtain a target network  $\hat{Q}$  and use  $\hat{Q}$  for generating the Q-learning targets  $y_j$  for the following  $C$  updates to  $Q$ . 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. Generating the targets using an older set of parameters adds a delay between the time an update to  $Q$  is made and the time the update affects the targets  $y_j$ , making divergence or oscillations much more unlikely.

We also found it helpful to clip the error term from the update  $r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s,a;\theta_i)$  to be between  $-1$  and  $1$ . Because the absolute value loss function  $|x|$  has a derivative of  $-1$  for all negative values of  $x$  and a derivative of  $1$  for all positive values of  $x$ , clipping the squared error to be between  $-1$  and  $1$  corresponds to using an absolute value loss function for errors outside of the  $(-1, 1)$  interval. This form of error clipping further improved the stability of the algorithm.

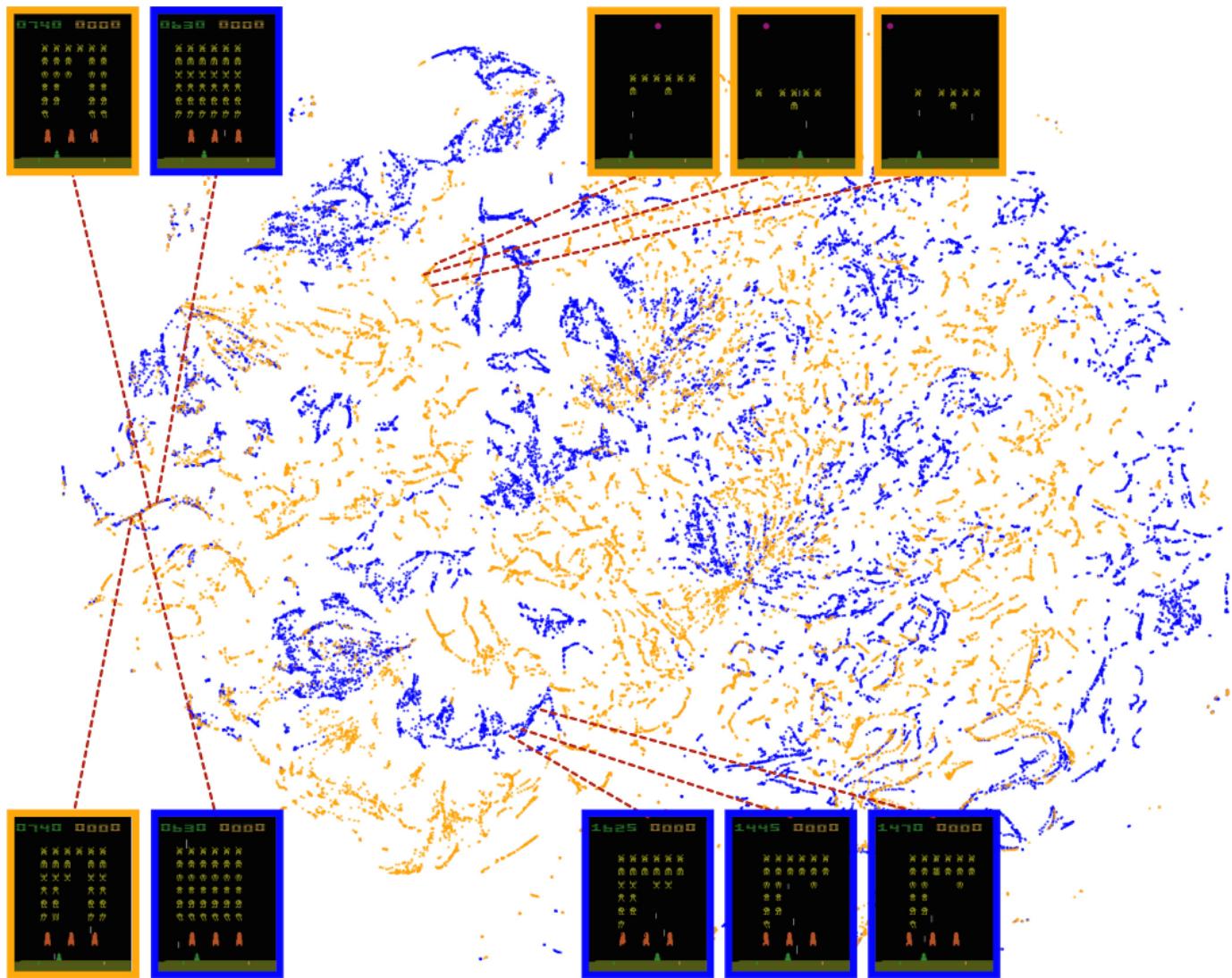
#### Algorithm 1: deep Q-learning with experience replay.

```

Initialize replay memory  $D$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights  $\theta$ 
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  $\varepsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$ 
        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  $\theta$ 
        Every  $C$  steps reset  $\hat{Q} = Q$ 
    End For
End For

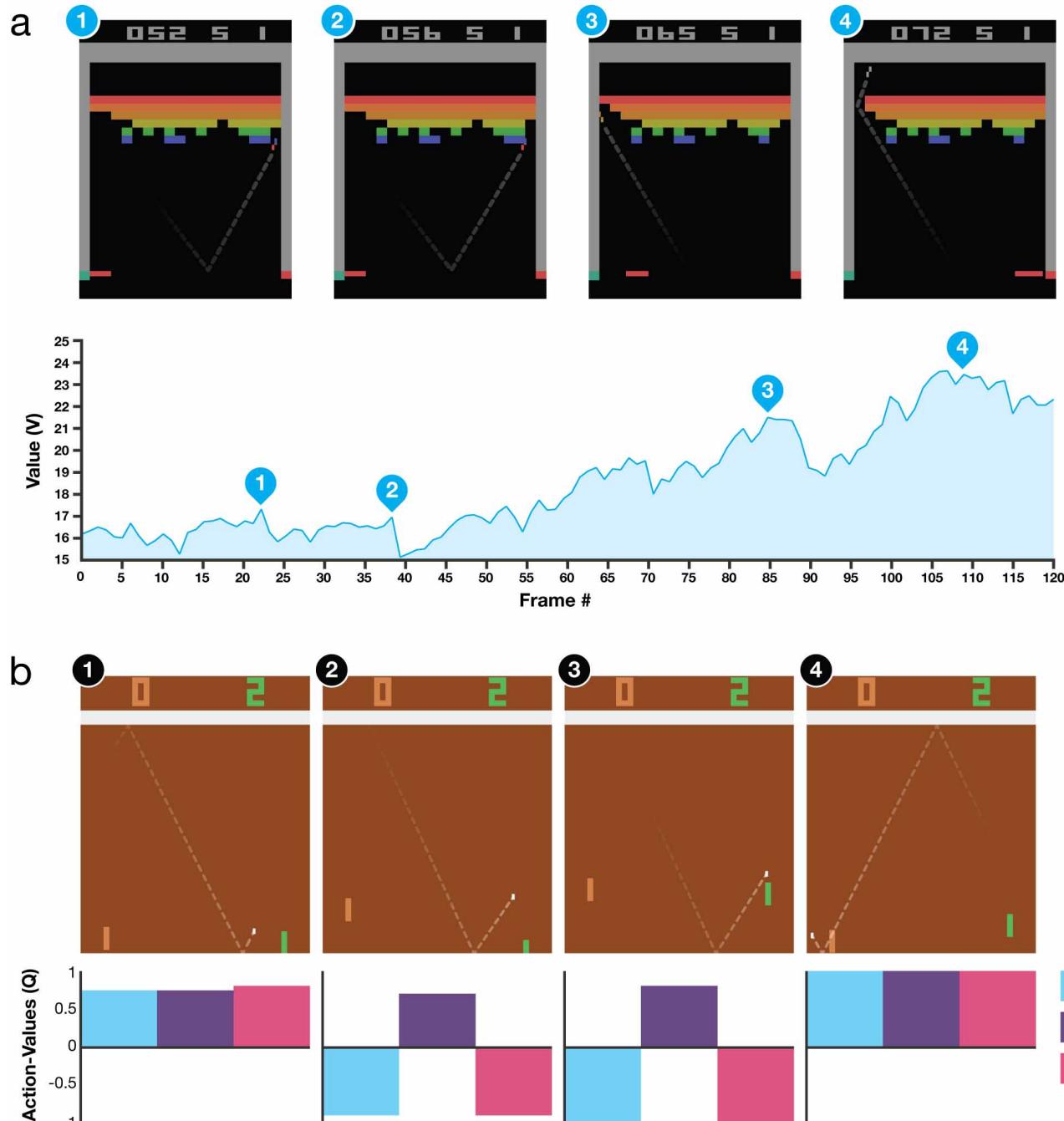
```

31. Jarrett, K., Kavukcuoglu, K., Ranzato, M. A. & LeCun, Y. What is the best multi-stage architecture for object recognition? *Proc. IEEE Int. Conf. Comput. Vis.* 2146–2153 (2009).
32. Nair, V. & Hinton, G. E. Rectified linear units improve restricted Boltzmann machines. *Proc. Int. Conf. Mach. Learn.* 807–814 (2010).
33. Kaelbling, L. P., Littman, M. L. & Cassandra, A. R. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* **101**, 99–134 (1994).



**Extended Data Figure 1 | Two-dimensional t-SNE embedding of the representations in the last hidden layer assigned by DQN to game states experienced during a combination of human and agent play in Space Invaders.** The plot was generated by running the t-SNE algorithm<sup>25</sup> on the last hidden layer representation assigned by DQN to game states experienced during a combination of human (30 min) and agent (2 h) play. The fact that there is similar structure in the two-dimensional embeddings corresponding to the DQN representation of states experienced during human play (orange

points) and DQN play (blue points) suggests that the representations learned by DQN do indeed generalize to data generated from policies other than its own. The presence in the t-SNE embedding of overlapping clusters of points corresponding to the network representation of states experienced during human and agent play shows that the DQN agent also follows sequences of states similar to those found in human play. Screenshots corresponding to selected states are shown (human: orange border; DQN: blue border).



**Extended Data Figure 2 | Visualization of learned value functions on two games, Breakout and Pong.** **a**, A visualization of the learned value function on the game Breakout. At time points 1 and 2, the state value is predicted to be  $\sim 17$  and the agent is clearing the bricks at the lowest level. Each of the peaks in the value function curve corresponds to a reward obtained by clearing a brick. At time point 3, the agent is about to break through to the top level of bricks and the value increases to  $\sim 21$  in anticipation of breaking out and clearing a large set of bricks. At point 4, the value is above 23 and the agent has broken through. After this point, the ball will bounce at the upper part of the bricks clearing many of them by itself. **b**, A visualization of the learned action-value function on the game Pong. At time point 1, the ball is moving towards the paddle controlled by the agent on the right side of the screen and the values of

all actions are around 0.7, reflecting the expected value of this state based on previous experience. At time point 2, the agent starts moving the paddle towards the ball and the value of the ‘up’ action stays high while the value of the ‘down’ action falls to  $-0.9$ . This reflects the fact that pressing ‘down’ would lead to the agent losing the ball and incurring a reward of  $-1$ . At time point 3, the agent hits the ball by pressing ‘up’ and the expected reward keeps increasing until time point 4, when the ball reaches the left edge of the screen and the value of all actions reflects that the agent is about to receive a reward of 1. Note, the dashed line shows the past trajectory of the ball purely for illustrative purposes (that is, not shown during the game). With permission from Atari Interactive, Inc.

**Extended Data Table 1 | List of hyperparameters and their values**

| Hyperparameter                  | Value   | Description  |
|---------------------------------|---------|--|
| minibatch size                  | 32      | Number of training cases over which each stochastic gradient descent (SGD) update is computed.   |
| replay memory size              | 1000000 | SGD updates are sampled from this number of most recent frames.  |
| agent history length            | 4       | The number of most recent frames experienced by the agent that are given as input to the Q network.  |
| target network update frequency | 10000   | The frequency (measured in the number of parameter updates) with which the target network is updated (this corresponds to the parameter C from Algorithm 1).                     |
| discount factor                 | 0.99    | Discount factor gamma used in the Q-learning update.   |
| action repeat                   | 4       | Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame.   |
| update frequency                | 4       | The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates. |
| learning rate                   | 0.00025 | The learning rate used by RMSProp.   |
| gradient momentum               | 0.95    | Gradient momentum used by RMSProp.   |
| squared gradient momentum       | 0.95    | Squared gradient (denominator) momentum used by RMSProp.   |
| min squared gradient            | 0.01    | Constant added to the squared gradient in the denominator of the RMSProp update.   |
| initial exploration             | 1       | Initial value of $\epsilon$ in $\epsilon$ -greedy exploration.   |
| final exploration               | 0.1     | Final value of $\epsilon$ in $\epsilon$ -greedy exploration.   |
| final exploration frame         | 1000000 | The number of frames over which the initial value of $\epsilon$ is linearly annealed to its final value.   |
| replay start size               | 50000   | A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.                              |
| no-op max                       | 30      | Maximum number of "do nothing" actions to be performed by the agent at the start of an episode.  |

The values of all the hyperparameters were selected by performing an informal search on the games Pong, Breakout, Seaquest, Space Invaders and Beam Rider. We did not perform a systematic grid search owing to the high computational cost, although it is conceivable that even better results could be obtained by systematically tuning the hyperparameter values.

**Extended Data Table 2 | Comparison of games scores obtained by DQN agents with methods from the literature<sup>12,15</sup> and a professional human games tester**

| Game                | Random Play | Best Linear Learner | Contingency (SARSA) | Human | DQN ( $\pm$ std)      | Normalized DQN (% Human) |
|---------------------|-------------|---------------------|---------------------|-------|-----------------------|--------------------------|
| Alien               | 227.8       | 939.2               | 103.2               | 6875  | 3069 ( $\pm$ 1093)    | 42.7%                    |
| Amidar              | 5.8         | 103.4               | 183.6               | 1676  | 739.5 ( $\pm$ 3024)   | 43.9%                    |
| Assault             | 222.4       | 628                 | 537                 | 1496  | 3359 ( $\pm$ 775)     | 246.2%                   |
| Asterix             | 210         | 987.3               | 1332                | 8503  | 6012 ( $\pm$ 1744)    | 70.0%                    |
| Asteroids           | 719.1       | 907.3               | 89                  | 13157 | 1629 ( $\pm$ 542)     | 7.3%                     |
| Atlantis            | 12850       | 62687               | 852.9               | 29028 | 85641 ( $\pm$ 17600)  | 449.9%                   |
| Bank Heist          | 14.2        | 190.8               | 67.4                | 734.4 | 429.7 ( $\pm$ 650)    | 57.7%                    |
| Battle Zone         | 2360        | 15820               | 16.2                | 37800 | 26300 ( $\pm$ 7725)   | 67.6%                    |
| Beam Rider          | 363.9       | 929.4               | 1743                | 5775  | 6846 ( $\pm$ 1619)    | 119.8%                   |
| Bowling             | 23.1        | 43.9                | 36.4                | 154.8 | 42.4 ( $\pm$ 88)      | 14.7%                    |
| Boxing              | 0.1         | 44                  | 9.8                 | 4.3   | 71.8 ( $\pm$ 8.4)     | 1707.9%                  |
| Breakout            | 1.7         | 5.2                 | 6.1                 | 31.8  | 401.2 ( $\pm$ 26.9)   | 1327.2%                  |
| Centipede           | 2091        | 8803                | 4647                | 11963 | 8309 ( $\pm$ 5237)    | 63.0%                    |
| Chopper Command     | 811         | 1582                | 16.9                | 9882  | 6687 ( $\pm$ 2916)    | 64.8%                    |
| Crazy Climber       | 10781       | 23411               | 149.8               | 35411 | 114103 ( $\pm$ 22797) | 419.5%                   |
| Demon Attack        | 152.1       | 520.5               | 0                   | 3401  | 9711 ( $\pm$ 2406)    | 294.2%                   |
| Double Dunk         | -18.6       | -13.1               | -16                 | -15.5 | -18.1 ( $\pm$ 2.6)    | 17.1%                    |
| Enduro              | 0           | 129.1               | 159.4               | 309.6 | 301.8 ( $\pm$ 24.6)   | 97.5%                    |
| Fishing Derby       | -91.7       | -89.5               | -85.1               | 5.5   | -0.8 ( $\pm$ 19.0)    | 93.5%                    |
| Freeway             | 0           | 19.1                | 19.7                | 29.6  | 30.3 ( $\pm$ 0.7)     | 102.4%                   |
| Frostbite           | 65.2        | 216.9               | 180.9               | 4335  | 328.3 ( $\pm$ 250.5)  | 6.2%                     |
| Gopher              | 257.6       | 1288                | 2368                | 2321  | 8520 ( $\pm$ 3279)    | 400.4%                   |
| Gravitar            | 173         | 387.7               | 429                 | 2672  | 306.7 ( $\pm$ 223.9)  | 5.3%                     |
| H.E.R.O.            | 1027        | 6459                | 7295                | 25763 | 19950 ( $\pm$ 158)    | 76.5%                    |
| Ice Hockey          | -11.2       | -9.5                | -3.2                | 0.9   | -1.6 ( $\pm$ 2.5)     | 79.3%                    |
| James Bond          | 29          | 202.8               | 354.1               | 406.7 | 576.7 ( $\pm$ 175.5)  | 145.0%                   |
| Kangaroo            | 52          | 1622                | 8.8                 | 3035  | 6740 ( $\pm$ 2959)    | 224.2%                   |
| Krull               | 1598        | 3372                | 3341                | 2395  | 3805 ( $\pm$ 1033)    | 277.0%                   |
| Kung-Fu Master      | 258.5       | 19544               | 29151               | 22736 | 23270 ( $\pm$ 5955)   | 102.4%                   |
| Montezuma's Revenge | 0           | 10.7                | 259                 | 4367  | 0 ( $\pm$ 0)          | 0.0%                     |
| Ms. Pacman          | 307.3       | 1692                | 1227                | 15693 | 2311 ( $\pm$ 525)     | 13.0%                    |
| Name This Game      | 2292        | 2500                | 2247                | 4076  | 7257 ( $\pm$ 547)     | 278.3%                   |
| Pong                | -20.7       | -19                 | -17.4               | 9.3   | 18.9 ( $\pm$ 1.3)     | 132.0%                   |
| Private Eye         | 24.9        | 684.3               | 86                  | 69571 | 1788 ( $\pm$ 5473)    | 2.5%                     |
| Q*Bert              | 163.9       | 613.5               | 960.3               | 13455 | 10596 ( $\pm$ 3294)   | 78.5%                    |
| River Raid          | 1339        | 1904                | 2650                | 13513 | 8316 ( $\pm$ 1049)    | 57.3%                    |
| Road Runner         | 11.5        | 67.7                | 89.1                | 7845  | 18257 ( $\pm$ 4268)   | 232.9%                   |
| Robotank            | 2.2         | 28.7                | 12.4                | 11.9  | 51.6 ( $\pm$ 4.7)     | 509.0%                   |
| Seaquest            | 68.4        | 664.8               | 675.5               | 20182 | 5286 ( $\pm$ 1310)    | 25.9%                    |
| Space Invaders      | 148         | 250.1               | 267.9               | 1652  | 1976 ( $\pm$ 893)     | 121.5%                   |
| Star Gunner         | 664         | 1070                | 9.4                 | 10250 | 57997 ( $\pm$ 3152)   | 598.1%                   |
| Tennis              | -23.8       | -0.1                | 0                   | -8.9  | -2.5 ( $\pm$ 1.9)     | 143.2%                   |
| Time Pilot          | 3568        | 3741                | 24.9                | 5925  | 5947 ( $\pm$ 1600)    | 100.9%                   |
| Tutankham           | 11.4        | 114.3               | 98.2                | 167.6 | 186.7 ( $\pm$ 41.9)   | 112.2%                   |
| Up and Down         | 533.4       | 3533                | 2449                | 9082  | 8456 ( $\pm$ 3162)    | 92.7%                    |
| Venture             | 0           | 66                  | 0.6                 | 1188  | 380.0 ( $\pm$ 238.6)  | 32.0%                    |
| Video Pinball       | 16257       | 16871               | 19761               | 17298 | 42684 ( $\pm$ 16287)  | 2539.4%                  |
| Wizard of Wor       | 563.5       | 1981                | 36.9                | 4757  | 3393 ( $\pm$ 2019)    | 67.5%                    |
| Zaxxon              | 32.5        | 3365                | 21.4                | 9173  | 4977 ( $\pm$ 1235)    | 54.1%                    |

Best Linear Learner is the best result obtained by a linear function approximator on different types of hand designed features<sup>12</sup>. Contingency (SARSA) agent figures are the results obtained in ref. 15. Note the figures in the last column indicate the performance of DQN relative to the human games tester, expressed as a percentage, that is,  $100 \times (DQN \text{ score} - \text{random play score}) / (\text{human score} - \text{random play score})$ .

**Extended Data Table 3 | The effects of replay and separating the target Q-network**

| Game           | With replay,<br>with target Q | With replay,<br>without target Q | Without replay,<br>with target Q | Without replay,<br>without target Q |
|----------------|-------------------------------|----------------------------------|----------------------------------|-------------------------------------|
| Breakout       | 316.8                         | 240.7                            | 10.2                             | 3.2                                 |
| Enduro         | 1006.3                        | 831.4                            | 141.9                            | 29.1                                |
| River Raid     | 7446.6                        | 4102.8                           | 2867.7                           | 1453.0                              |
| Seaquest       | 2894.4                        | 822.6                            | 1003.0                           | 275.8                               |
| Space Invaders | 1088.9                        | 826.3                            | 373.2                            | 302.0                               |

DQN agents were trained for 10 million frames using standard hyperparameters for all possible combinations of turning replay on or off, using or not using a separate target Q-network, and three different learning rates. Each agent was evaluated every 250,000 training frames for 135,000 validation frames and the highest average episode score is reported. Note that these evaluation episodes were not truncated at 5 min leading to higher scores on Enduro than the ones reported in Extended Data Table 2. Note also that the number of training frames was shorter (10 million frames) as compared to the main results presented in Extended Data Table 2 (50 million frames).

**Extended Data Table 4 | Comparison of DQN performance with linear function approximator**

| Game           | DQN    | Linear |
|----------------|--------|--------|
| Breakout       | 316.8  | 3.00   |
| Enduro         | 1006.3 | 62.0   |
| River Raid     | 7446.6 | 2346.9 |
| Seaquest       | 2894.4 | 656.9  |
| Space Invaders | 1088.9 | 301.3  |

The performance of the DQN agent is compared with the performance of a linear function approximator on the 5 validation games (that is, where a single linear layer was used instead of the convolutional network, in combination with replay and separate target network). Agents were trained for 10 million frames using standard hyperparameters, and three different learning rates. Each agent was evaluated every 250,000 training frames for 135,000 validation frames and the highest average episode score is reported. Note that these evaluation episodes were not truncated at 5 min leading to higher scores on Enduro than the ones reported in Extended Data Table 2. Note also that the number of training frames was shorter (10 million frames) as compared to the main results presented in Extended Data Table 2 (50 million frames).

# Deep Reinforcement Learning with Double Q-learning

**Hado van Hasselt** and **Arthur Guez** and **David Silver**  
Google DeepMind

## Abstract

The popular Q-learning algorithm is known to overestimate action values under certain conditions. It was not previously known whether, in practice, such overestimations are common, whether they harm performance, and whether they can generally be prevented. In this paper, we answer all these questions affirmatively. In particular, we first show that the recent DQN algorithm, which combines Q-learning with a deep neural network, suffers from substantial overestimations in some games in the Atari 2600 domain. We then show that the idea behind the Double Q-learning algorithm, which was introduced in a tabular setting, can be generalized to work with large-scale function approximation. We propose a specific adaptation to the DQN algorithm and show that the resulting algorithm not only reduces the observed overestimations, as hypothesized, but that this also leads to much better performance on several games.

The goal of reinforcement learning (Sutton and Barto, 1998) is to learn good policies for sequential decision problems, by optimizing a cumulative future reward signal. Q-learning (Watkins, 1989) is one of the most popular reinforcement learning algorithms, but it is known to sometimes learn unrealistically high action values because it includes a maximization step over estimated action values, which tends to prefer overestimated to underestimated values.

In previous work, overestimations have been attributed to insufficiently flexible function approximation (Thrun and Schwartz, 1993) and noise (van Hasselt, 2010, 2011). In this paper, we unify these views and show overestimations can occur when the action values are inaccurate, irrespective of the source of approximation error. Of course, imprecise value estimates are the norm during learning, which indicates that overestimations may be much more common than previously appreciated.

It is an open question whether, if the overestimations do occur, this negatively affects performance in practice. Overoptimistic value estimates are not necessarily a problem in and of themselves. If all values would be uniformly higher then the relative action preferences are preserved and we would not expect the resulting policy to be any worse. Furthermore, it is known that sometimes it is good to be optimistic: optimism in the face of uncertainty is a well-known

exploration technique (Kaelbling et al., 1996). If, however, the overestimations are not uniform and not concentrated at states about which we wish to learn more, then they might negatively affect the quality of the resulting policy. Thrun and Schwartz (1993) give specific examples in which this leads to suboptimal policies, even asymptotically.

To test whether overestimations occur in practice and at scale, we investigate the performance of the recent DQN algorithm (Mnih et al., 2015). DQN combines Q-learning with a flexible deep neural network and was tested on a varied and large set of deterministic Atari 2600 games, reaching human-level performance on many games. In some ways, this setting is a best-case scenario for Q-learning, because the deep neural network provides flexible function approximation with the potential for a low asymptotic approximation error, and the determinism of the environments prevents the harmful effects of noise. Perhaps surprisingly, we show that even in this comparatively favorable setting DQN sometimes substantially overestimates the values of the actions.

We show that the idea behind the Double Q-learning algorithm (van Hasselt, 2010), which was first proposed in a tabular setting, can be generalized to work with arbitrary function approximation, including deep neural networks. We use this to construct a new algorithm we call Double DQN. We then show that this algorithm not only yields more accurate value estimates, but leads to much higher scores on several games. This demonstrates that the overestimations of DQN were indeed leading to poorer policies and that it is beneficial to reduce them. In addition, by improving upon DQN we obtain state-of-the-art results on the Atari domain.

## Background

To solve sequential decision problems we can learn estimates for the optimal value of each action, defined as the expected sum of future rewards when taking that action and following the optimal policy thereafter. Under a given policy  $\pi$ , the true value of an action  $a$  in a state  $s$  is

$$Q_\pi(s, a) \equiv \mathbb{E}[R_1 + \gamma R_2 + \dots | S_0 = s, A_0 = a, \pi],$$

where  $\gamma \in [0, 1]$  is a discount factor that trades off the importance of immediate and later rewards. The optimal value is then  $Q_*(s, a) = \max_\pi Q_\pi(s, a)$ . An optimal policy is easily derived from the optimal values by selecting the highest-valued action in each state.

Estimates for the optimal action values can be learned using Q-learning (Watkins, 1989), a form of temporal difference learning (Sutton, 1988). Most interesting problems are too large to learn all action values in all states separately. Instead, we can learn a parameterized value function  $Q(s, a; \theta_t)$ . The standard Q-learning update for the parameters after taking action  $A_t$  in state  $S_t$  and observing the immediate reward  $R_{t+1}$  and resulting state  $S_{t+1}$  is then

$$\theta_{t+1} = \theta_t + \alpha(Y_t^Q - Q(S_t, A_t; \theta_t)) \nabla_{\theta_t} Q(S_t, A_t; \theta_t). \quad (1)$$

where  $\alpha$  is a scalar step size and the target  $Y_t^Q$  is defined as

$$Y_t^Q \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t). \quad (2)$$

This update resembles stochastic gradient descent, updating the current value  $Q(S_t, A_t; \theta_t)$  towards a target value  $Y_t^Q$ .

## Deep Q Networks

A deep Q network (DQN) is a multi-layered neural network that for a given state  $s$  outputs a vector of action values  $Q(s, \cdot; \theta)$ , where  $\theta$  are the parameters of the network. For an  $n$ -dimensional state space and an action space containing  $m$  actions, the neural network is a function from  $\mathbb{R}^n$  to  $\mathbb{R}^m$ . Two important ingredients of the DQN algorithm as proposed by Mnih et al. (2015) are the use of a target network, and the use of experience replay. The target network, with parameters  $\theta^-$ , is the same as the online network except that its parameters are copied every  $\tau$  steps from the online network, so that then  $\theta^-_t = \theta_t$ , and kept fixed on all other steps. The target used by DQN is then

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta^-_t). \quad (3)$$

For the experience replay (Lin, 1992), observed transitions are stored for some time and sampled uniformly from this memory bank to update the network. Both the target network and the experience replay dramatically improve the performance of the algorithm (Mnih et al., 2015).

## Double Q-learning

The max operator in standard Q-learning and DQN, in (2) and (3), uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, we can decouple the selection from the evaluation. This is the idea behind Double Q-learning (van Hasselt, 2010).

In the original Double Q-learning algorithm, two value functions are learned by assigning each experience randomly to update one of the two value functions, such that there are two sets of weights,  $\theta$  and  $\theta'$ . For each update, one set of weights is used to determine the greedy policy and the other to determine its value. For a clear comparison, we can first untangle the selection and evaluation in Q-learning and rewrite its target (2) as

$$Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta_t).$$

The Double Q-learning error can then be written as

$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta'_t). \quad (4)$$

Notice that the selection of the action, in the argmax, is still due to the online weights  $\theta_t$ . This means that, as in Q-learning, we are still estimating the value of the greedy policy according to the current values, as defined by  $\theta_t$ . However, we use the second set of weights  $\theta'_t$  to fairly evaluate the value of this policy. This second set of weights can be updated symmetrically by switching the roles of  $\theta$  and  $\theta'$ .

## Overoptimism due to estimation errors

Q-learning's overestimations were first investigated by Thrun and Schwartz (1993), who showed that if the action values contain random errors uniformly distributed in an interval  $[-\epsilon, \epsilon]$  then each target is overestimated up to  $\gamma \epsilon \frac{m-1}{m+1}$ , where  $m$  is the number of actions. In addition, Thrun and Schwartz give a concrete example in which these overestimations even asymptotically lead to sub-optimal policies, and show the overestimations manifest themselves in a small toy problem when using function approximation. Later van Hasselt (2010) argued that noise in the environment can lead to overestimations even when using tabular representation, and proposed Double Q-learning as a solution.

In this section we demonstrate more generally that estimation errors of any kind can induce an upward bias, regardless of whether these errors are due to environmental noise, function approximation, non-stationarity, or any other source. This is important, because in practice any method will incur some inaccuracies during learning, simply due to the fact that the true values are initially unknown.

The result by Thrun and Schwartz (1993) cited above gives an upper bound to the overestimation for a specific setup, but it is also possible, and potentially more interesting, to derive a lower bound.

**Theorem 1.** Consider a state  $s$  in which all the true optimal action values are equal at  $Q_*(s, a) = V_*(s)$  for some  $V_*(s)$ . Let  $Q_t$  be arbitrary value estimates that are on the whole unbiased in the sense that  $\sum_a (Q_t(s, a) - V_*(s)) = 0$ , but that are not all correct, such that  $\frac{1}{m} \sum_a (Q_t(s, a) - V_*(s))^2 = C$  for some  $C > 0$ , where  $m \geq 2$  is the number of actions in  $s$ . Under these conditions,  $\max_a Q_t(s, a) \geq V_*(s) + \sqrt{\frac{C}{m-1}}$ . This lower bound is tight. Under the same conditions, the lower bound on the absolute error of the Double Q-learning estimate is zero. (Proof in appendix.)

Note that we did not need to assume that estimation errors for different actions are independent. This theorem shows that even if the value estimates are on average correct, estimation errors of any source can drive the estimates up and away from the true optimal values.

The lower bound in Theorem 1 decreases with the number of actions. This is an artifact of considering the lower bound, which requires very specific values to be attained. More typically, the overoptimism increases with the number of actions as shown in Figure 1. Q-learning's overestimations there indeed increase with the number of actions,

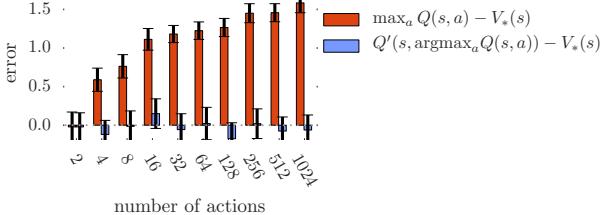


Figure 1: The orange bars show the bias in a single Q-learning update when the action values are  $Q(s, a) = V_*(s) + \epsilon_a$  and the errors  $\{\epsilon_a\}_{a=1}^m$  are independent standard normal random variables. The second set of action values  $Q'$ , used for the blue bars, was generated identically and independently. All bars are the average of 100 repetitions.

while Double Q-learning is unbiased. As another example, if for all actions  $Q_*(s, a) = V_*(s)$  and the estimation errors  $Q_t(s, a) - V_*(s)$  are uniformly random in  $[-1, 1]$ , then the overoptimism is  $\frac{m-1}{m+1}$ . (Proof in appendix.)

We now turn to function approximation and consider a real-valued continuous state space with 10 discrete actions in each state. For simplicity, the true optimal action values in this example depend only on state so that in each state all actions have the same true value. These true values are shown in the left column of plots in Figure 2 (purple lines) and are defined as either  $Q_*(s, a) = \sin(s)$  (top row) or  $Q_*(s, a) = 2 \exp(-s^2)$  (middle and bottom rows). The left plots also show an approximation for a single action (green lines) as a function of state as well as the samples the estimate is based on (green dots). The estimate is a  $d$ -degree polynomial that is fit to the true values at sampled states, where  $d = 6$  (top and middle rows) or  $d = 9$  (bottom row). The samples match the true function exactly: there is no noise and we assume we have ground truth for the action value on these sampled states. The approximation is inexact even on the sampled states for the top two rows because the function approximation is insufficiently flexible. In the bottom row, the function is flexible enough to fit the green dots, but this reduces the accuracy in unsampled states. Notice that the sampled states are spaced further apart near the left side of the left plots, resulting in larger estimation errors. In many ways this is a typical learning setting, where at each point in time we only have limited data.

The middle column of plots in Figure 2 shows estimated action value functions for all 10 actions (green lines), as functions of state, along with the maximum action value in each state (black dashed line). Although the true value function is the same for all actions, the approximations differ because we have supplied different sets of sampled states.<sup>1</sup> The maximum is often higher than the ground truth shown in purple on the left. This is confirmed in the right plots, which show the difference between the black and purple curves in orange. The orange line is almost always positive,

<sup>1</sup>Each action-value function is fit with a different subset of integer states. States  $-6$  and  $6$  are always included to avoid extrapolations, and for each action two adjacent integers are missing: for action  $a_1$  states  $-5$  and  $-4$  are not sampled, for  $a_2$  states  $-4$  and  $-3$  are not sampled, and so on. This causes the estimated values to differ.

indicating an upward bias. The right plots also show the estimates from Double Q-learning in blue<sup>2</sup>, which are on average much closer to zero. This demonstrates that Double Q-learning indeed can successfully reduce the overoptimism of Q-learning.

The different rows in Figure 2 show variations of the same experiment. The difference between the top and middle rows is the true value function, demonstrating that overestimations are not an artifact of a specific true value function. The difference between the middle and bottom rows is the flexibility of the function approximation. In the left-middle plot, the estimates are even incorrect for some of the sampled states because the function is insufficiently flexible. The function in the bottom-left plot is more flexible but this causes higher estimation errors for unseen states, resulting in higher overestimations. This is important because flexible parametric function approximators are often employed in reinforcement learning (see, e.g., Tesauro 1995; Sallans and Hinton 2004; Riedmiller 2005; Mnih et al. 2015).

In contrast to van Hasselt (2010) we did not use a statistical argument to find overestimations, the process to obtain Figure 2 is fully deterministic. In contrast to Thrun and Schwartz (1993), we did not rely on inflexible function approximation with irreducible asymptotic errors; the bottom row shows that a function that is flexible enough to cover all samples leads to high overestimations. This indicates that the overestimations can occur quite generally.

In the examples above, overestimations occur even when assuming we have samples of the *true* action value at certain states. The value estimates can further deteriorate if we bootstrap off of action values that are already overoptimistic, since this causes overestimations to propagate throughout our estimates. Although *uniformly* overestimating values might not hurt the resulting policy, in practice overestimation errors will differ for different states and actions. Overestimation combined with bootstrapping then has the pernicious effect of propagating the wrong relative information about which states are more valuable than others, directly affecting the quality of the learned policies.

The overestimations should not be confused with optimism in the face of uncertainty (Sutton, 1990; Agrawal, 1995; Kaelbling et al., 1996; Auer et al., 2002; Brafman and Tennenholtz, 2003; Szita and Lőrincz, 2008; Strehl et al., 2009), where an exploration bonus is given to states or actions with uncertain values. Conversely, the overestimations discussed here occur only after updating, resulting in overoptimism in the face of apparent certainty. This was already observed by Thrun and Schwartz (1993), who noted that, in contrast to optimism in the face of uncertainty, these overestimations actually can impede learning an optimal policy. We will see this negative effect on policy quality confirmed later in the experiments as well: when we reduce the overestimations using Double Q-learning, the policies improve.

<sup>2</sup>We arbitrarily used the samples of action  $a_{i+5}$  (for  $i \leq 5$ ) or  $a_{i-5}$  (for  $i > 5$ ) as the second set of samples for the double estimator of action  $a_i$ .

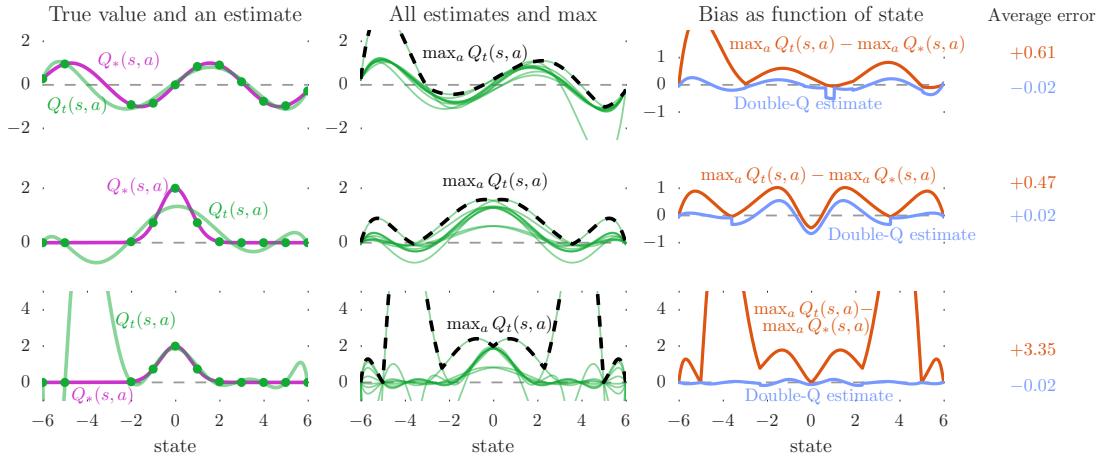


Figure 2: Illustration of overestimations during learning. In each state (x-axis), there are 10 actions. The **left column** shows the true values  $V_*(s)$  (purple line). All true action values are defined by  $Q_*(s, a) = V_*(s)$ . The green line shows estimated values  $Q(s, a)$  for one action as a function of state, fitted to the true value at several sampled states (green dots). The **middle column** plots show all the estimated values (green), and the maximum of these values (dashed black). The maximum is higher than the true value (purple, left plot) almost everywhere. The **right column** plots shows the difference in orange. The blue line in the right plots is the estimate used by Double Q-learning with a second set of samples for each state. The blue line is much closer to zero, indicating less bias. The three **rows** correspond to different true functions (left, purple) or capacities of the fitted function (left, green). (Details in the text)

## Double DQN

The idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. We therefore propose to evaluate the greedy policy according to the online network, but using the target network to estimate its value. In reference to both Double Q-learning and DQN, we refer to the resulting algorithm as Double DQN. Its update is the same as for DQN, but replacing the target  $Y_t^{\text{DQN}}$  with

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t), \theta_t^-).$$

In comparison to Double Q-learning (4), the weights of the second network  $\theta'_t$  are replaced with the weights of the target network  $\theta_t^-$  for the evaluation of the current greedy policy. The update to the target network stays unchanged from DQN, and remains a periodic copy of the online network.

This version of Double DQN is perhaps the minimal possible change to DQN towards Double Q-learning. The goal is to get most of the benefit of Double Q-learning, while keeping the rest of the DQN algorithm intact for a fair comparison, and with minimal computational overhead.

## Empirical results

In this section, we analyze the overestimations of DQN and show that Double DQN improves over DQN both in terms of value accuracy and in terms of policy quality. To further test the robustness of the approach we additionally evaluate the algorithms with random starts generated from expert human trajectories, as proposed by Nair et al. (2015).

Our testbed consists of Atari 2600 games, using the Arcade Learning Environment (Bellemare et al., 2013). The

goal is for a single algorithm, with a fixed set of hyperparameters, to learn to play each of the games separately from interaction given only the screen pixels as input. This is a demanding testbed: not only are the inputs high-dimensional, the game visuals and game mechanics vary substantially between games. Good solutions must therefore rely heavily on the learning algorithm — it is not practically feasible to overfit the domain by relying only on tuning.

We closely follow the experimental setting and network architecture outlined by Mnih et al. (2015). Briefly, the network architecture is a convolutional neural network (Fukushima, 1988; LeCun et al., 1998) with 3 convolution layers and a fully-connected hidden layer (approximately 1.5M parameters in total). The network takes the last four frames as input and outputs the action value of each action. On each game, the network is trained on a single GPU for 200M frames, or approximately 1 week.

## Results on overoptimism

Figure 3 shows examples of DQN’s overestimations in six Atari games. DQN and Double DQN were both trained under the exact conditions described by Mnih et al. (2015). DQN is consistently and sometimes vastly overoptimistic about the value of the current greedy policy, as can be seen by comparing the orange learning curves in the top row of plots to the straight orange lines, which represent the actual discounted value of the best learned policy. More precisely, the (averaged) value estimates are computed regularly during training with full evaluation phases of length  $T = 125,000$  steps as

$$\frac{1}{T} \sum_{t=1}^T \arg\max_a Q(S_t, a; \theta).$$

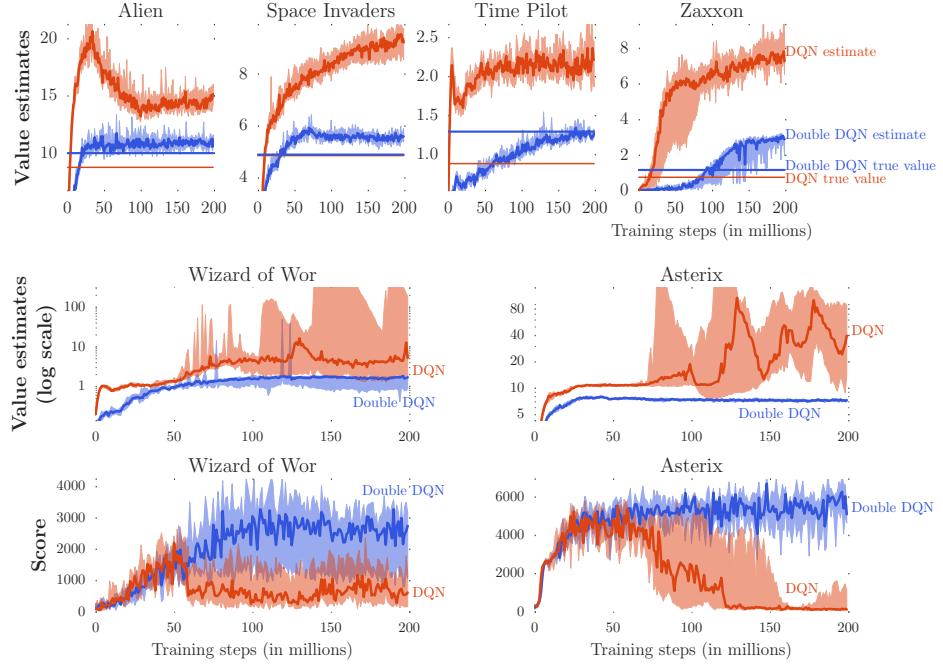


Figure 3: The **top** and **middle** rows show value estimates by DQN (orange) and Double DQN (blue) on six Atari games. The results are obtained by running DQN and Double DQN with 6 different random seeds with the hyper-parameters employed by Mnih et al. (2015). The darker line shows the median over seeds and we average the two extreme values to obtain the shaded area (i.e., 10% and 90% quantiles with linear interpolation). 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. The **middle** row shows the value estimates (in log scale) for two games in which DQN’s overoptimism is quite extreme. The **bottom** row shows the detrimental effect of this on the score achieved by the agent as it is evaluated during training: the scores drop when the overestimations begin. Learning with Double DQN is much more stable.

The ground truth averaged values are obtained by running the best learned policies for several episodes and computing the actual cumulative rewards. Without overestimations we would expect these quantities to match up (i.e., the curve to match the straight line at the right of each plot). Instead, the learning curves of DQN consistently end up much higher than the true values. The learning curves for Double DQN, shown in blue, are much closer to the blue straight line representing the true value of the final policy. Note that the blue straight line is often higher than the orange straight line. This indicates that Double DQN does not just produce more accurate value estimates but also better policies.

More extreme overestimations are shown in the middle two plots, where DQN is highly unstable on the games Asterix and Wizard of Wor. Notice the log scale for the values on the  $y$ -axis. The bottom two plots show the corresponding scores for these two games. Notice that the increases in value estimates for DQN in the middle plots coincide with decreasing scores in bottom plots. Again, this indicates that the overestimations are harming the quality of the resulting policies. If seen in isolation, one might perhaps be tempted to think the observed instability is related to inherent instability problems of off-policy learning with function approximation (Baird, 1995; Tsitsiklis and Van Roy, 1997; Sutton et al., 2008; Maei, 2011; Sutton et al., 2015). However, we see that learning is much more stable with Double DQN,

|        | DQN    | Double DQN |
|--------|--------|------------|
| Median | 93.5%  | 114.7%     |
| Mean   | 241.1% | 330.3%     |

Table 1: Summary of normalized performance up to 5 minutes of play on 49 games. Results for DQN are from Mnih et al. (2015)

suggesting that the cause for these instabilities is in fact Q-learning’s overoptimism. Figure 3 only shows a few examples, but overestimations were observed for DQN in all 49 tested Atari games, albeit in varying amounts.

## Quality of the learned policies

Overoptimism does not always adversely affect the quality of the learned policy. For example, DQN achieves optimal behavior in Pong despite slightly overestimating the policy value. Nevertheless, reducing overestimations can significantly benefit the stability of learning; we see clear examples of this in Figure 3. We now assess more generally how much Double DQN helps in terms of policy quality by evaluating on all 49 games that DQN was tested on.

As described by Mnih et al. (2015) each evaluation episode starts by executing a special no-op action that does not affect the environment up to 30 times, to provide different starting points for the agent. Some exploration during evaluation provides additional randomization. For Double DQN we used the exact same hyper-parameters as for DQN,

|        | DQN    | Double DQN | Double DQN (tuned) |
|--------|--------|------------|--------------------|
| Median | 47.5%  | 88.4%      | 116.7%             |
| Mean   | 122.0% | 273.1%     | 475.2%             |

Table 2: Summary of normalized performance up to 30 minutes of play on 49 games with human starts. Results for DQN are from Nair et al. (2015).

to allow for a controlled experiment focused just on reducing overestimations. The learned policies are evaluated for 5 mins of emulator time (18,000 frames) with an  $\epsilon$ -greedy policy where  $\epsilon = 0.05$ . The scores are averaged over 100 episodes. The only difference between Double DQN and DQN is the target, using  $Y_t^{\text{DoubleDQN}}$  rather than  $Y_t^{\text{DQN}}$ . This evaluation is somewhat adversarial, as the used hyperparameters were tuned for DQN but not for Double DQN.

To obtain summary statistics across games, we normalize the score for each game as follows:

$$\text{score}_{\text{normalized}} = \frac{\text{score}_{\text{agent}} - \text{score}_{\text{random}}}{\text{score}_{\text{human}} - \text{score}_{\text{random}}} . \quad (5)$$

The ‘random’ and ‘human’ scores are the same as used by Mnih et al. (2015), and are given in the appendix.

Table 1, under **no ops**, shows that on the whole Double DQN clearly improves over DQN. A detailed comparison (in appendix) shows that there are several games in which Double DQN greatly improves upon DQN. Noteworthy examples include Road Runner (from 233% to 617%), Asterix (from 70% to 180%), Zaxxon (from 54% to 111%), and Double Dunk (from 17% to 397%).

The Gorila algorithm (Nair et al., 2015), which is a massively distributed version of DQN, is not included in the table because the architecture and infrastructure is sufficiently different to make a direct comparison unclear. For completeness, we note that Gorila obtained median and mean normalized scores of 96% and 495%, respectively.

## Robustness to Human starts

One concern with the previous evaluation is that in deterministic games with a unique starting point the learner could potentially learn to remember sequences of actions without much need to generalize. While successful, the solution would not be particularly robust. By testing the agents from various starting points, we can test whether the found solutions generalize well, and as such provide a challenging testbed for the learned policies (Nair et al., 2015).

We obtained 100 starting points sampled for each game from a human expert’s trajectory, as proposed by Nair et al. (2015). We start an evaluation episode from each of these starting points and run the emulator for up to 108,000 frames (30 mins at 60Hz including the trajectory before the starting point). Each agent is only evaluated on the rewards accumulated after the starting point.

For this evaluation we include a tuned version of Double DQN. Some tuning is appropriate because the hyperparameters were tuned for DQN, which is a different algorithm. For the tuned version of Double DQN, we increased the number of frames between each two copies of the target network from 10,000 to 30,000, to reduce overestimations further because immediately after each switch DQN and Double DQN

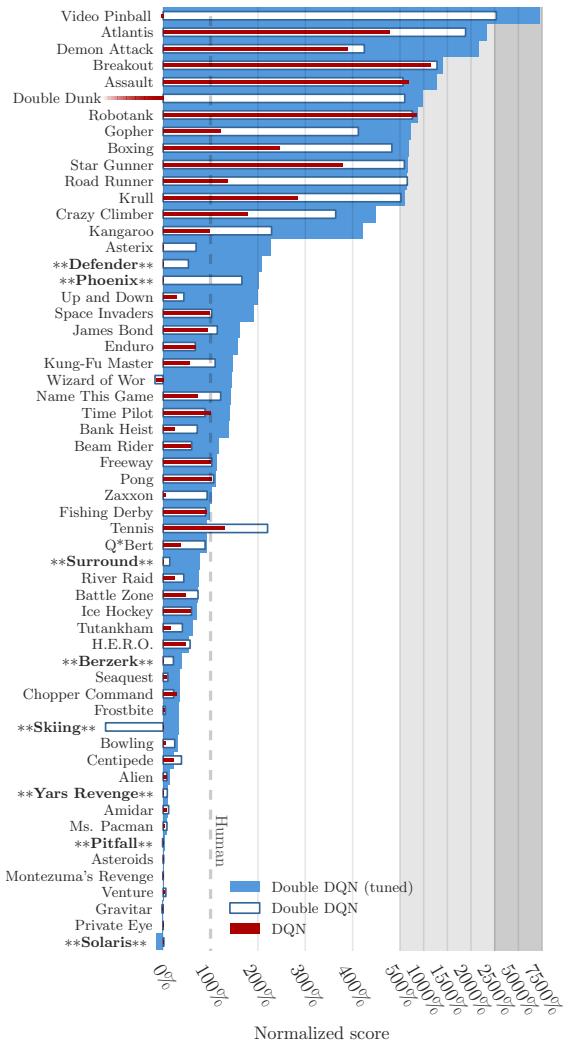


Figure 4: Normalized scores on 57 Atari games, tested for 100 episodes per game with human starts. Compared to Mnih et al. (2015), eight games additional games were tested. These are indicated with stars and a bold font.

both revert to Q-learning. In addition, we reduced the exploration during learning from  $\epsilon = 0.1$  to  $\epsilon = 0.01$ , and then used  $\epsilon = 0.001$  during evaluation. Finally, the tuned version uses a single shared bias for all action values in the top layer of the network. Each of these changes improved performance and together they result in clearly better results.<sup>3</sup>

Table 2 reports summary statistics for this evaluation on the 49 games from Mnih et al. (2015). Double DQN obtains clearly higher median and mean scores. Again Gorila DQN (Nair et al., 2015) is not included in the table, but for completeness note it obtained a median of 78% and a mean of 259%. Detailed results, plus results for an additional 8 games, are available in Figure 4 and in the appendix. On several games the improvements from DQN to Double DQN are striking, in some cases bringing scores much closer to

<sup>3</sup>Except for Tennis, where the lower  $\epsilon$  during training seemed to hurt rather than help.

human, or even surpassing these.

Double DQN appears more robust to this more challenging evaluation, suggesting that appropriate generalizations occur and that the found solutions do not exploit the determinism of the environments. This is appealing, as it indicates progress towards finding general solutions rather than a deterministic sequence of steps that would be less robust.

## Discussion

This paper has five contributions. First, we have shown why Q-learning can be overoptimistic in large-scale problems, even if these are deterministic, due to the inherent estimation errors of learning. Second, by analyzing the value estimates on Atari games we have shown that these overestimations are more common and severe in practice than previously acknowledged. Third, we have shown that Double Q-learning can be used at scale to successfully reduce this overoptimism, resulting in more stable and reliable learning. Fourth, we have proposed a specific implementation called Double DQN, that uses the existing architecture and deep neural network of the DQN algorithm without requiring additional networks or parameters. Finally, we have shown that Double DQN finds better policies, obtaining new state-of-the-art results on the Atari 2600 domain.

## Acknowledgments

We would like to thank Tom Schaul, Volodymyr Mnih, Marc Bellemare, Thomas Degris, Georg Ostrovski, and Richard Sutton for helpful comments, and everyone at Google DeepMind for a constructive research environment.

## References

- R. Agrawal. Sample mean based index policies with  $O(\log n)$  regret for the multi-armed bandit problem. *Advances in Applied Probability*, pages 1054–1078, 1995.
- P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.
- L. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning: Proceedings of the Twelfth International Conference*, pages 30–37, 1995.
- M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res. (JAIR)*, 47:253–279, 2013.
- R. I. Brafman and M. Tennenholtz. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *The Journal of Machine Learning Research*, 3:213–231, 2003.
- K. Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130, 1988.
- L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.
- Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- L. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3):293–321, 1992.
- H. R. Maei. *Gradient temporal-difference learning algorithms*. PhD thesis, University of Alberta, 2011.
- V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518 (7540):529–533, 2015.
- A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver. Massively parallel methods for deep reinforcement learning. In *Deep Learning Workshop, ICML*, 2015.
- M. Riedmiller. Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method. In J. Gama, R. Camacho, P. Brazdil, A. Jorge, and L. Torgo, editors, *Proceedings of the 16th European Conference on Machine Learning (ECML'05)*, pages 317–328. Springer, 2005.
- B. Sallans and G. E. Hinton. Reinforcement learning with factored states and actions. *The Journal of Machine Learning Research*, 5:1063–1088, 2004.
- A. L. Strehl, L. Li, and M. L. Littman. Reinforcement learning in finite MDPs: PAC analysis. *The Journal of Machine Learning Research*, 10:2413–2444, 2009.
- R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.
- R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the seventh international conference on machine learning*, pages 216–224, 1990.
- R. S. Sutton and A. G. Barto. *Introduction to reinforcement learning*. MIT Press, 1998.
- R. S. Sutton, C. Szepesvári, and H. R. Maei. A convergent  $O(n)$  algorithm for off-policy temporal-difference learning with linear function approximation. *Advances in Neural Information Processing Systems 21 (NIPS-08)*, 21:1609–1616, 2008.
- R. S. Sutton, A. R. Mahmood, and M. White. An emphatic approach to the problem of off-policy temporal-difference learning. *arXiv preprint arXiv:1503.04269*, 2015.
- I. Szita and A. Lőrincz. The many faces of optimism: a unifying approach. In *Proceedings of the 25th international conference on Machine learning*, pages 1048–1055. ACM, 2008.
- G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.
- S. Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. In M. Mozer, P. Smolensky, D. Touretzky, J. Elman, and A. Weigend, editors, *Proceedings of the 1993 Connectionist Models Summer School*, Hillsdale, NJ, 1993. Lawrence Erlbaum.
- J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.
- H. van Hasselt. Double Q-learning. *Advances in Neural Information Processing Systems*, 23:2613–2621, 2010.
- H. van Hasselt. *Insights in Reinforcement Learning*. PhD thesis, Utrecht University, 2011.
- C. J. C. H. Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge England, 1989.

## Appendix

**Theorem 1.** Consider a state  $s$  in which all the true optimal action values are equal at  $Q_*(s, a) = V_*(s)$  for some  $V_*(s)$ . Let  $Q_t$  be arbitrary value estimates that are on the whole unbiased in the sense that  $\sum_a (Q_t(s, a) - V_*(s)) = 0$ , but that are not all zero, such that  $\frac{1}{m} \sum_a (Q_t(s, a) - V_*(s))^2 = C$  for some  $C > 0$ , where  $m \geq 2$  is the number of actions in  $s$ . Under these conditions,  $\max_a Q_t(s, a) \geq V_*(s) + \sqrt{\frac{C}{m-1}}$ . This lower bound is tight. Under the same conditions, the lower bound on the absolute error of the Double Q-learning estimate is zero.

*Proof of Theorem 1.* Define the errors for each action  $a$  as  $\epsilon_a = Q_t(s, a) - V_*(s)$ . Suppose that there exists a setting of  $\{\epsilon_a\}$  such that  $\max_a \epsilon_a < \sqrt{\frac{C}{m-1}}$ . Let  $\{\epsilon_i^+\}$  be the set of positive  $\epsilon$  of size  $n$ , and  $\{\epsilon_j^-\}$  the set of strictly negative  $\epsilon$  of size  $m-n$  (such that  $\{\epsilon\} = \{\epsilon_i^+\} \cup \{\epsilon_j^-\}$ ). If  $n = m$ , then  $\sum_a \epsilon_a = 0 \implies \epsilon_a = 0 \forall a$ , which contradicts  $\sum_a \epsilon_a^2 = mC$ . Hence, it must be that  $n \leq m-1$ . Then,  $\sum_{i=1}^n \epsilon_i^+ \leq n \max_i \epsilon_i^+ < n \sqrt{\frac{C}{m-1}}$ , and therefore (using the constraint  $\sum_a \epsilon_a = 0$ ) we also have that  $\sum_{j=1}^{m-n} |\epsilon_j^-| < n \sqrt{\frac{C}{m-1}}$ . This implies  $\max_j |\epsilon_j^-| < n \sqrt{\frac{C}{m-1}}$ . By Hölder's inequality, then

$$\begin{aligned} \sum_{j=1}^{m-n} (\epsilon_j^-)^2 &\leq \sum_{j=1}^{m-n} |\epsilon_j^-| \cdot \max_j |\epsilon_j^-| \\ &< n \sqrt{\frac{C}{m-1}} n \sqrt{\frac{C}{m-1}}. \end{aligned}$$

We can now combine these relations to compute an upper-bound on the sum of squares for all  $\epsilon_a$ :

$$\begin{aligned} \sum_{a=1}^m (\epsilon_a)^2 &= \sum_{i=1}^n (\epsilon_i^+)^2 + \sum_{j=1}^{m-n} (\epsilon_j^-)^2 \\ &< n \frac{C}{m-1} + n \sqrt{\frac{C}{m-1}} n \sqrt{\frac{C}{m-1}} \\ &= C \frac{n(n+1)}{m-1} \\ &\leq mC. \end{aligned}$$

This contradicts the assumption that  $\sum_{a=1}^m \epsilon_a^2 < mC$ , and therefore  $\max_a \epsilon_a \geq \sqrt{\frac{C}{m-1}}$  for all settings of  $\epsilon$  that satisfy the constraints. We can check that the lower-bound is tight by setting  $\epsilon_a = \sqrt{\frac{C}{m-1}}$  for  $a = 1, \dots, m-1$  and  $\epsilon_m = -\sqrt{(m-1)C}$ . This verifies  $\sum_a \epsilon_a^2 = mC$  and  $\sum_a \epsilon_a = 0$ .

The only tight lower bound on the absolute error for Double Q-learning  $|Q'_t(s, \text{argmax}_a Q_t(s, a)) - V_*(s)|$  is zero. This can be seen by because we can have

$$Q_t(s, a_1) = V_*(s) + \sqrt{C \frac{m-1}{m}},$$

and

$$Q_t(s, a_i) = V_*(s) - \sqrt{C \frac{1}{m(m-1)}}, \text{ for } i > 1.$$

Then the conditions of the theorem hold. If then, furthermore, we have  $Q'_t(s, a_1) = V_*(s)$  then the error is zero. The remaining action values  $Q'_t(s, a_i)$ , for  $i > 1$ , are arbitrary.  $\square$

**Theorem 2.** Consider a state  $s$  in which all the true optimal action values are equal at  $Q_*(s, a) = V_*(s)$ . Suppose that the estimation errors  $Q_t(s, a) - Q_*(s, a)$  are independently distributed uniformly randomly in  $[-1, 1]$ . Then,

$$\mathbb{E} \left[ \max_a Q_t(s, a) - V_*(s) \right] = \frac{m-1}{m+1}$$

*Proof.* Define  $\epsilon_a = Q_t(s, a) - Q_*(s, a)$ ; this is a uniform random variable in  $[-1, 1]$ . The probability that  $\max_a Q_t(s, a) \leq x$  for some  $x$  is equal to the probability that  $\epsilon_a \leq x$  for all  $a$  simultaneously. Because the estimation errors are independent, we can derive

$$\begin{aligned} P(\max_a \epsilon_a \leq x) &= P(X_1 \leq x \wedge X_2 \leq x \wedge \dots \wedge X_m \leq x) \\ &= \prod_{a=1}^m P(\epsilon_a \leq x). \end{aligned}$$

The function  $P(\epsilon_a \leq x)$  is the cumulative distribution function (CDF) of  $\epsilon_a$ , which here is simply defined as

$$P(\epsilon_a \leq x) = \begin{cases} 0 & \text{if } x \leq -1 \\ \frac{1+x}{2} & \text{if } x \in (-1, 1) \\ 1 & \text{if } x \geq 1 \end{cases}$$

This implies that

$$\begin{aligned} P(\max_a \epsilon_a \leq x) &= \prod_{a=1}^m P(\epsilon_a \leq x) \\ &= \begin{cases} 0 & \text{if } x \leq -1 \\ \left(\frac{1+x}{2}\right)^m & \text{if } x \in (-1, 1) \\ 1 & \text{if } x \geq 1 \end{cases} \end{aligned}$$

This gives us the CDF of the random variable  $\max_a \epsilon_a$ . Its expectation can be written as an integral

$$\mathbb{E} \left[ \max_a \epsilon_a \right] = \int_{-1}^1 x f_{\max}(x) dx,$$

where  $f_{\max}$  is the probability density function of this variable, defined as the derivative of the CDF:  $f_{\max}(x) = \frac{d}{dx} P(\max_a \epsilon_a \leq x)$ , so that for  $x \in [-1, 1]$  we have  $f_{\max}(x) = \frac{m}{2} \left(\frac{1+x}{2}\right)^{m-1}$ . Evaluating the integral yields

$$\begin{aligned} \mathbb{E} \left[ \max_a \epsilon_a \right] &= \int_{-1}^1 x f_{\max}(x) dx \\ &= \left[ \left( \frac{x+1}{2} \right)^m \frac{mx-1}{m+1} \right]_{-1}^1 \\ &= \frac{m-1}{m+1}. \end{aligned} \quad \square$$

## Experimental Details for the Atari 2600 Domain

We selected the 49 games to match the list used by Mnih et al. (2015), see Tables below for the full list. Each agent step is composed of four frames (the last selected action is repeated during these frames) and reward values (obtained from the Arcade Learning Environment (Bellemare et al., 2013)) are clipped between -1 and 1.

## Network Architecture

The convolution network used in the experiment is exactly the one proposed by Mnih et al. (2015), we only provide details here for completeness. The input to the network is a 84x84x4 tensor containing a rescaled, and gray-scale, version of the last four frames. The first convolution layer convolves the input with 32 filters of size 8 (stride 4), the second layer has 64 layers of size 4 (stride 2), the final convolution layer has 64 filters of size 3 (stride 1). This is followed by a fully-connected hidden layer of 512 units. All these layers are separated by Rectifier Linear Units (ReLU). Finally, a fully-connected linear layer projects to the output of the network, i.e., the Q-values. The optimization employed to train the network is RMSProp (with momentum parameter 0.95).

## Hyper-parameters

In all experiments, the discount was set to  $\gamma = 0.99$ , and the learning rate to  $\alpha = 0.00025$ . The number of steps between target network updates was  $\tau = 10,000$ . Training is done over 50M steps (i.e., 200M frames). The agent is evaluated every 1M steps, and the best policy across these evaluations is kept as the output of the learning process. The size of the experience replay memory is 1M tuples. The memory gets sampled to update the network every 4 steps with minibatches of size 32. The simple exploration policy used is an  $\epsilon$ -greedy policy with the  $\epsilon$  decreasing linearly from 1 to 0.1 over 1M steps.

## Supplementary Results in the Atari 2600 Domain

The Tables below provide further detailed results for our experiments in the Atari domain.

| <b>Game</b>         | <b>Random</b> | <b>Human</b> | <b>DQN</b> | <b>Double DQN</b> |
|---------------------|---------------|--------------|------------|-------------------|
| Alien               | 227.80        | 6875.40      | 3069.33    | 2907.30           |
| Amidar              | 5.80          | 1675.80      | 739.50     | 702.10            |
| Assault             | 222.40        | 1496.40      | 3358.63    | 5022.90           |
| Asterix             | 210.00        | 8503.30      | 6011.67    | 15150.00          |
| Asteroids           | 719.10        | 13156.70     | 1629.33    | 930.60            |
| Atlantis            | 12850.00      | 29028.10     | 85950.00   | 64758.00          |
| Bank Heist          | 14.20         | 734.40       | 429.67     | 728.30            |
| Battle Zone         | 2360.00       | 37800.00     | 26300.00   | 25730.00          |
| Beam Rider          | 363.90        | 5774.70      | 6845.93    | 7654.00           |
| Bowling             | 23.10         | 154.80       | 42.40      | 70.50             |
| Boxing              | 0.10          | 4.30         | 71.83      | 81.70             |
| Breakout            | 1.70          | 31.80        | 401.20     | 375.00            |
| Centipede           | 2090.90       | 11963.20     | 8309.40    | 4139.40           |
| Chopper Command     | 811.00        | 9881.80      | 6686.67    | 4653.00           |
| Crazy Climber       | 10780.50      | 35410.50     | 114103.33  | 101874.00         |
| Demon Attack        | 152.10        | 3401.30      | 9711.17    | 9711.90           |
| Double Dunk         | -18.60        | -15.50       | -18.07     | -6.30             |
| Enduro              | 0.00          | 309.60       | 301.77     | 319.50            |
| Fishing Derby       | -91.70        | 5.50         | -0.80      | 20.30             |
| Freeway             | 0.00          | 29.60        | 30.30      | 31.80             |
| Frostbite           | 65.20         | 4334.70      | 328.33     | 241.50            |
| Gopher              | 257.60        | 2321.00      | 8520.00    | 8215.40           |
| Gravitar            | 173.00        | 2672.00      | 306.67     | 170.50            |
| H.E.R.O.            | 1027.00       | 25762.50     | 19950.33   | 20357.00          |
| Ice Hockey          | -11.20        | 0.90         | -1.60      | -2.40             |
| James Bond          | 29.00         | 406.70       | 576.67     | 438.00            |
| Kangaroo            | 52.00         | 3035.00      | 6740.00    | 13651.00          |
| Krull               | 1598.00       | 2394.60      | 3804.67    | 4396.70           |
| Kung-Fu Master      | 258.50        | 22736.20     | 23270.00   | 29486.00          |
| Montezuma's Revenge | 0.00          | 4366.70      | 0.00       | 0.00              |
| Ms. Pacman          | 307.30        | 15693.40     | 2311.00    | 3210.00           |
| Name This Game      | 2292.30       | 4076.20      | 7256.67    | 6997.10           |
| Pong                | -20.70        | 9.30         | 18.90      | 21.00             |
| Private Eye         | 24.90         | 69571.30     | 1787.57    | 670.10            |
| Q*Bert              | 163.90        | 13455.00     | 10595.83   | 14875.00          |
| River Raid          | 1338.50       | 13513.30     | 8315.67    | 12015.30          |
| Road Runner         | 11.50         | 7845.00      | 18256.67   | 48377.00          |
| Robotank            | 2.20          | 11.90        | 51.57      | 46.70             |
| Seaquest            | 68.40         | 20181.80     | 5286.00    | 7995.00           |
| Space Invaders      | 148.00        | 1652.30      | 1975.50    | 3154.60           |
| Star Gunner         | 664.00        | 10250.00     | 57996.67   | 65188.00          |
| Tennis              | -23.80        | -8.90        | -2.47      | 1.70              |
| Time Pilot          | 3568.00       | 5925.00      | 5946.67    | 7964.00           |
| Tutankham           | 11.40         | 167.60       | 186.70     | 190.60            |
| Up and Down         | 533.40        | 9082.00      | 8456.33    | 16769.90          |
| Venture             | 0.00          | 1187.50      | 380.00     | 93.00             |
| Video Pinball       | 16256.90      | 17297.60     | 42684.07   | 70009.00          |
| Wizard of Wor       | 563.50        | 4756.50      | 3393.33    | 5204.00           |
| Zaxxon              | 32.50         | 9173.30      | 4976.67    | 10182.00          |

Table 3: Raw scores for the no-op evaluation condition (5 minutes emulator time). DQN as given by Mnih et al. (2015).

| <b>Game</b>         | <b>DQN</b> | <b>Double DQN</b> |
|---------------------|------------|-------------------|
| Alien               | 42.75 %    | 40.31 %           |
| Amidar              | 43.93 %    | 41.69 %           |
| Assault             | 246.17 %   | 376.81 %          |
| Asterix             | 69.96 %    | 180.15 %          |
| Asteroids           | 7.32 %     | 1.70 %            |
| Atlantis            | 451.85 %   | 320.85 %          |
| Bank Heist          | 57.69 %    | 99.15 %           |
| Battle Zone         | 67.55 %    | 65.94 %           |
| Beam Rider          | 119.80 %   | 134.73 %          |
| Bowling             | 14.65 %    | 35.99 %           |
| Boxing              | 1707.86 %  | 1942.86 %         |
| Breakout            | 1327.24 %  | 1240.20 %         |
| Centipede           | 62.99 %    | 20.75 %           |
| Chopper Command     | 64.78 %    | 42.36 %           |
| Crazy Climber       | 419.50 %   | 369.85 %          |
| Demon Attack        | 294.20 %   | 294.22 %          |
| Double Dunk         | 17.10 %    | 396.77 %          |
| Enduro              | 97.47 %    | 103.20 %          |
| Fishing Derby       | 93.52 %    | 115.23 %          |
| Freeway             | 102.36 %   | 107.43 %          |
| Frostbite           | 6.16 %     | 4.13 %            |
| Gopher              | 400.43 %   | 385.66 %          |
| Gravitar            | 5.35 %     | -0.10 %           |
| H.E.R.O.            | 76.50 %    | 78.15 %           |
| Ice Hockey          | 79.34 %    | 72.73 %           |
| James Bond          | 145.00 %   | 108.29 %          |
| Kangaroo            | 224.20 %   | 455.88 %          |
| Krull               | 277.01 %   | 351.33 %          |
| Kung-Fu Master      | 102.37 %   | 130.03 %          |
| Montezuma's Revenge | 0.00 %     | 0.00 %            |
| Ms. Pacman          | 13.02 %    | 18.87 %           |
| Name This Game      | 278.29 %   | 263.74 %          |
| Pong                | 132.00 %   | 139.00 %          |
| Private Eye         | 2.53 %     | 0.93 %            |
| Q*Bert              | 78.49 %    | 110.68 %          |
| River Raid          | 57.31 %    | 87.70 %           |
| Road Runner         | 232.91 %   | 617.42 %          |
| Robotank            | 508.97 %   | 458.76 %          |
| Seaquest            | 25.94 %    | 39.41 %           |
| Space Invaders      | 121.49 %   | 199.87 %          |
| Star Gunner         | 598.09 %   | 673.11 %          |
| Tennis              | 143.15 %   | 171.14 %          |
| Time Pilot          | 100.92 %   | 186.51 %          |
| Tutankham           | 112.23 %   | 114.72 %          |
| Up and Down         | 92.68 %    | 189.93 %          |
| Venture             | 32.00 %    | 7.83 %            |
| Video Pinball       | 2539.36 %  | 5164.99 %         |
| Wizard of Wor       | 67.49 %    | 110.67 %          |
| Zaxxon              | 54.09 %    | 111.04 %          |

Table 4: Normalized results for no-op evaluation condition (5 minutes emulator time).

| <b>Game</b>         | <b>Random</b> | <b>Human</b> | <b>DQN</b> | <b>Double DQN</b> | <b>Double DQN (tuned)</b> |
|---------------------|---------------|--------------|------------|-------------------|---------------------------|
| Alien               | 128.30        | 6371.30      | 570.2      | 621.6             | 1033.4                    |
| Amidar              | 11.80         | 1540.40      | 133.4      | 188.2             | 169.1                     |
| Assault             | 166.90        | 628.90       | 3332.3     | 2774.3            | 6060.8                    |
| Asterix             | 164.50        | 7536.00      | 124.5      | 5285.0            | 16837.0                   |
| Asteroids           | 871.30        | 36517.30     | 697.1      | 1219.0            | 1193.2                    |
| Atlantis            | 13463.00      | 26575.00     | 76108.0    | 260556.0          | 319688.0                  |
| Bank Heist          | 21.70         | 644.50       | 176.3      | 469.8             | 886.0                     |
| Battle Zone         | 3560.00       | 33030.00     | 17560.0    | 25240.0           | 24740.0                   |
| Beam Rider          | 254.60        | 14961.00     | 8672.4     | 9107.9            | 17417.2                   |
| Berzerk             | 196.10        | 2237.50      |            | 635.8             | 1011.1                    |
| Bowling             | 35.20         | 146.50       | 41.2       | 62.3              | 69.6                      |
| Boxing              | -1.50         | 9.60         | 25.8       | 52.1              | 73.5                      |
| Breakout            | 1.60          | 27.90        | 303.9      | 338.7             | 368.9                     |
| Centipede           | 1925.50       | 10321.90     | 3773.1     | 5166.6            | 3853.5                    |
| Chopper Command     | 644.00        | 8930.00      | 3046.0     | 2483.0            | 3495.0                    |
| Crazy Climber       | 9337.00       | 32667.00     | 50992.0    | 94315.0           | 113782.0                  |
| Defender            | 1965.50       | 14296.00     |            | 8531.0            | 27510.0                   |
| Demon Attack        | 208.30        | 3442.80      | 12835.2    | 13943.5           | 69803.4                   |
| Double Dunk         | -16.00        | -14.40       | -21.6      | -6.4              | -0.3                      |
| Enduro              | -81.80        | 740.20       | 475.6      | 475.9             | 1216.6                    |
| Fishing Derby       | -77.10        | 5.10         | -2.3       | -3.4              | 3.2                       |
| Freeway             | 0.10          | 25.60        | 25.8       | 26.3              | 28.8                      |
| Frostbite           | 66.40         | 4202.80      | 157.4      | 258.3             | 1448.1                    |
| Gopher              | 250.00        | 2311.00      | 2731.8     | 8742.8            | 15253.0                   |
| Gravitar            | 245.50        | 3116.00      | 216.5      | 170.0             | 200.5                     |
| H.E.R.O.            | 1580.30       | 25839.40     | 12952.5    | 15341.4           | 14892.5                   |
| Ice Hockey          | -9.70         | 0.50         | -3.8       | -3.6              | -2.5                      |
| James Bond          | 33.50         | 368.50       | 348.5      | 416.0             | 573.0                     |
| Kangaroo            | 100.00        | 2739.00      | 2696.0     | 6138.0            | 11204.0                   |
| Krull               | 1151.90       | 2109.10      | 3864.0     | 6130.4            | 6796.1                    |
| Kung-Fu Master      | 304.00        | 20786.80     | 11875.0    | 22771.0           | 30207.0                   |
| Montezuma's Revenge | 25.00         | 4182.00      | 50.0       | 30.0              | 42.0                      |
| Ms. Pacman          | 197.80        | 15375.00     | 763.5      | 1401.8            | 1241.3                    |
| Name This Game      | 1747.80       | 6796.00      | 5439.9     | 7871.5            | 8960.3                    |
| Phoenix             | 1134.40       | 6686.20      |            | 10364.0           | 12366.5                   |
| Pit Fall            | -348.80       | 5998.90      |            | -432.9            | -186.7                    |
| Pong                | -18.00        | 15.50        | 16.2       | 17.7              | 19.1                      |
| Private Eye         | 662.80        | 64169.10     | 298.2      | 346.3             | -575.5                    |
| Q*Bert              | 183.00        | 12085.00     | 4589.8     | 10713.3           | 11020.8                   |
| River Raid          | 588.30        | 14382.20     | 4065.3     | 6579.0            | 10838.4                   |
| Road Runner         | 200.00        | 6878.00      | 9264.0     | 43884.0           | 43156.0                   |
| Robotank            | 2.40          | 8.90         | 58.5       | 52.0              | 59.1                      |
| Seaquest            | 215.50        | 40425.80     | 2793.9     | 4199.4            | 14498.0                   |
| Skiing              | -15287.40     | -3686.60     |            | -29404.3          | -11490.4                  |
| Solaris             | 2047.20       | 11032.60     |            | 2166.8            | 810.0                     |
| Space Invaders      | 182.60        | 1464.90      | 1449.7     | 1495.7            | 2628.7                    |
| Star Gunner         | 697.00        | 9528.00      | 34081.0    | 53052.0           | 58365.0                   |
| Surround            | -9.70         | 5.40         |            | -7.6              | 1.9                       |
| Tennis              | -21.40        | -6.70        | -2.3       | 11.0              | -7.8                      |
| Time Pilot          | 3273.00       | 5650.00      | 5640.0     | 5375.0            | 6608.0                    |
| Tutankham           | 12.70         | 138.30       | 32.4       | 63.6              | 92.2                      |
| Up and Down         | 707.20        | 9896.10      | 3311.3     | 4721.1            | 19086.9                   |
| Venture             | 18.00         | 1039.00      | 54.0       | 75.0              | 21.0                      |
| Video Pinball       | 20452.0       | 15641.10     | 20228.1    | 148883.6          | 367823.7                  |
| Wizard of Wor       | 804.00        | 4556.00      | 246.0      | 155.0             | 6201.0                    |
| Yars Revenge        | 1476.90       | 47135.20     |            | 5439.5            | 6270.6                    |
| Zaxxon              | 475.00        | 8443.00      | 831.0      | 7874.0            | 8593.0                    |

Table 5: Raw scores for the human start condition (30 minutes emulator time). DQN as given by Nair et al. (2015).

| <b>Game</b>         | <b>DQN</b> | <b>Double DQN</b> | <b>Double DQN (tuned)</b> |
|---------------------|------------|-------------------|---------------------------|
| Alien               | 7.08%      | 7.90%             | 14.50%                    |
| Amidar              | 7.95%      | 11.54%            | 10.29%                    |
| Assault             | 685.15%    | 564.37%           | 1275.74%                  |
| Asterix             | -0.54%     | 69.46%            | 226.18%                   |
| Asteroids           | -0.49%     | 0.98%             | 0.90%                     |
| Atlantis            | 477.77%    | 1884.48%          | 2335.46%                  |
| Bank Heist          | 24.82%     | 71.95%            | 138.78%                   |
| Battle Zone         | 47.51%     | 73.57%            | 71.87%                    |
| Beam Rider          | 57.24%     | 60.20%            | 116.70%                   |
| Berzerk             |            | 21.54%            | 39.92%                    |
| Bowling             | 5.39%      | 24.35%            | 30.91%                    |
| Boxing              | 245.95%    | 482.88%           | 675.68%                   |
| Breakout            | 1149.43%   | 1281.75%          | 1396.58%                  |
| Centipede           | 22.00%     | 38.60%            | 22.96%                    |
| Chopper Command     | 28.99%     | 22.19%            | 34.41%                    |
| Crazy Climber       | 178.55%    | 364.24%           | 447.69%                   |
| Defender            |            | 53.25%            | 207.17%                   |
| Demon Attack        | 390.38%    | 424.65%           | 2151.65%                  |
| Double Dunk         | -350.00%   | 600.00%           | 981.25%                   |
| Enduro              | 67.81%     | 67.85%            | 157.96%                   |
| Fishing Derby       | 91.00%     | 89.66%            | 97.69%                    |
| Freeway             | 100.78%    | 102.75%           | 112.55%                   |
| Frostbite           | 2.20%      | 4.64%             | 33.40%                    |
| Gopher              | 120.42%    | 412.07%           | 727.95%                   |
| Gravitar            | -1.01%     | -2.63%            | -1.57%                    |
| H.E.R.O.            | 46.88%     | 56.73%            | 54.88%                    |
| Ice Hockey          | 57.84%     | 59.80%            | 70.59%                    |
| James Bond          | 94.03%     | 114.18%           | 161.04%                   |
| Kangaroo            | 98.37%     | 228.80%           | 420.77%                   |
| Krull               | 283.34%    | 520.11%           | 589.66%                   |
| Kung-Fu Master      | 56.49%     | 109.69%           | 145.99%                   |
| Montezuma's Revenge | 0.60%      | 0.12%             | 0.41%                     |
| Ms. Pacman          | 3.73%      | 7.93%             | 6.88%                     |
| Name This Game      | 73.14%     | 121.30%           | 142.87%                   |
| Phoenix             |            | 166.25%           | 202.31%                   |
| Pit Fall            |            | -1.32%            | 2.55%                     |
| Pong                | 102.09%    | 106.57%           | 110.75%                   |
| Private Eye         | -0.57%     | -0.50%            | -1.95%                    |
| Q*Bert              | 37.03%     | 88.48%            | 91.06%                    |
| River Raid          | 25.21%     | 43.43%            | 74.31%                    |
| Road Runner         | 135.73%    | 654.15%           | 643.25%                   |
| Robotank            | 863.08%    | 763.08%           | 872.31%                   |
| Seaquest            | 6.41%      | 9.91%             | 35.52%                    |
| Skiing              |            | -121.69%          | 32.73%                    |
| Solaris             |            | 1.33%             | -13.77%                   |
| Space Invaders      | 98.81%     | 102.40%           | 190.76%                   |
| Star Gunner         | 378.03%    | 592.85%           | 653.02%                   |
| Surround            |            | 13.91%            | 76.82%                    |
| Tennis              | 129.93%    | 220.41%           | 92.52%                    |
| Time Pilot          | 99.58%     | 88.43%            | 140.30%                   |
| Tutankham           | 15.68%     | 40.53%            | 63.30%                    |
| Up and Down         | 28.34%     | 43.68%            | 200.02%                   |
| Venture             | 3.53%      | 5.58%             | 0.29%                     |
| Video Pinball       | -4.65%     | 2669.60%          | 7220.51%                  |
| Wizard of Wor       | -14.87%    | -17.30%           | 143.84%                   |
| Yars Revenge        |            | 8.68%             | 10.50%                    |
| Zaxxon              | 4.47%      | 92.86%            | 101.88%                   |

Table 6: Normalized scores for the human start condition (30 minutes emulator time).



AIMLC ZG512 -  
Deep Reinforcement Learning

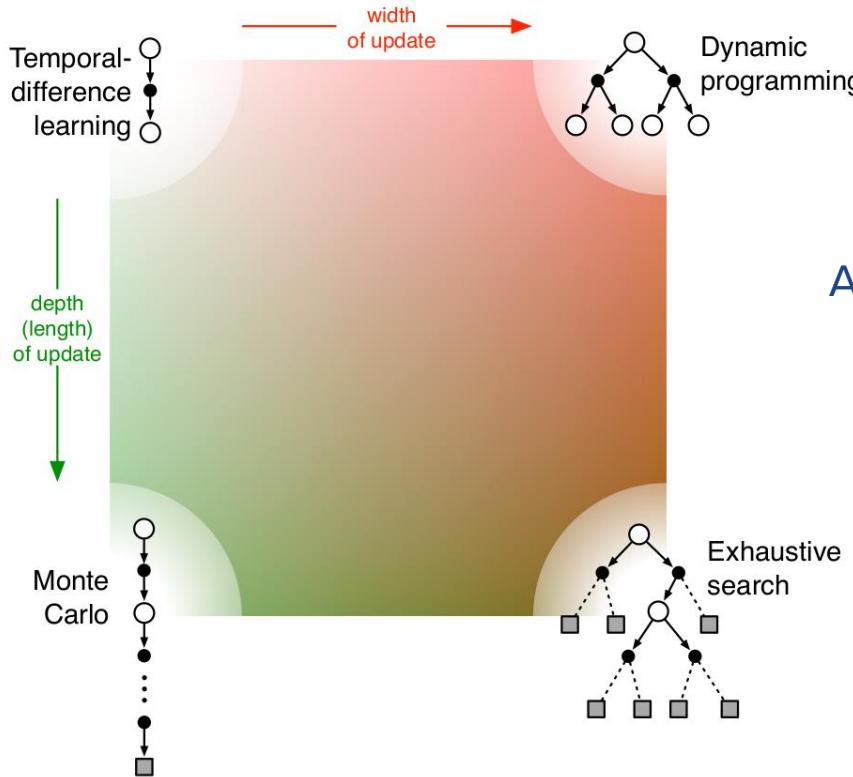
## Session #14: Model Based Algorithms



# 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 ]

# 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  $\pi$
- 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  $\pi$
- 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

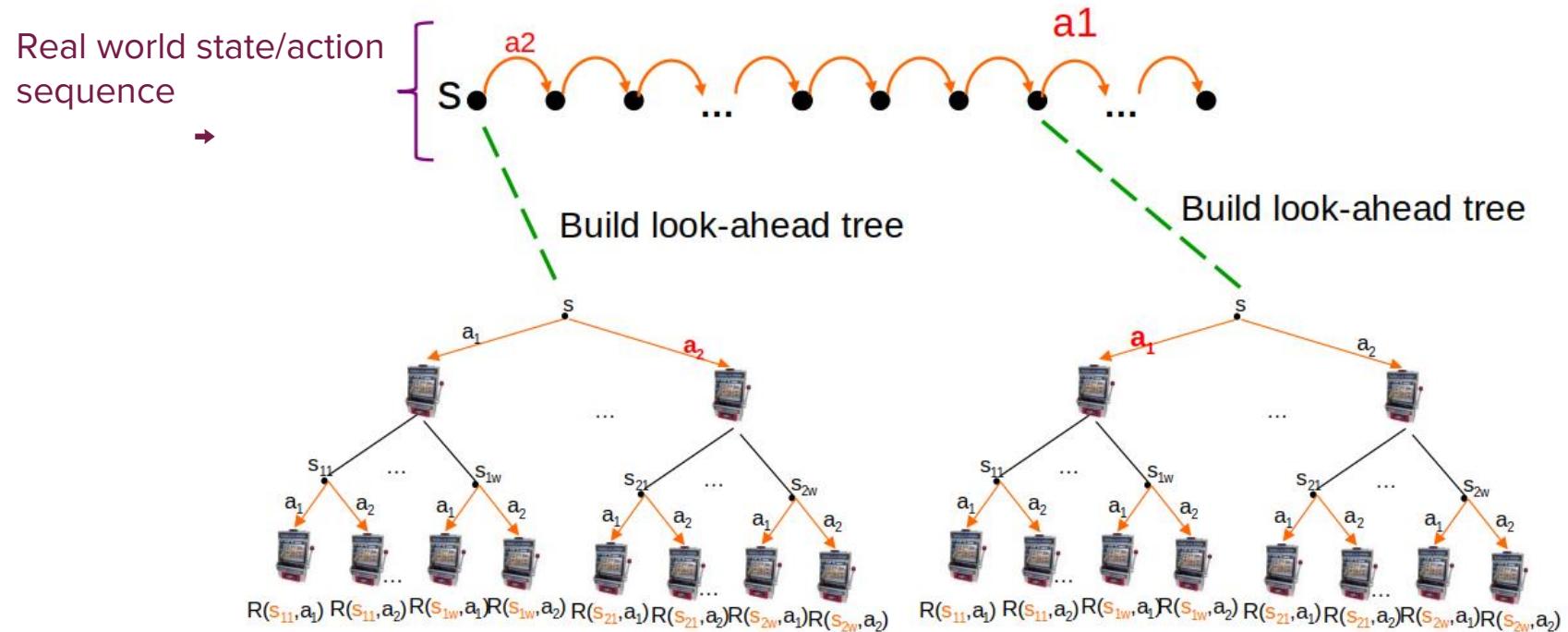


# Monte-Carlo Tree Search (MCTS)

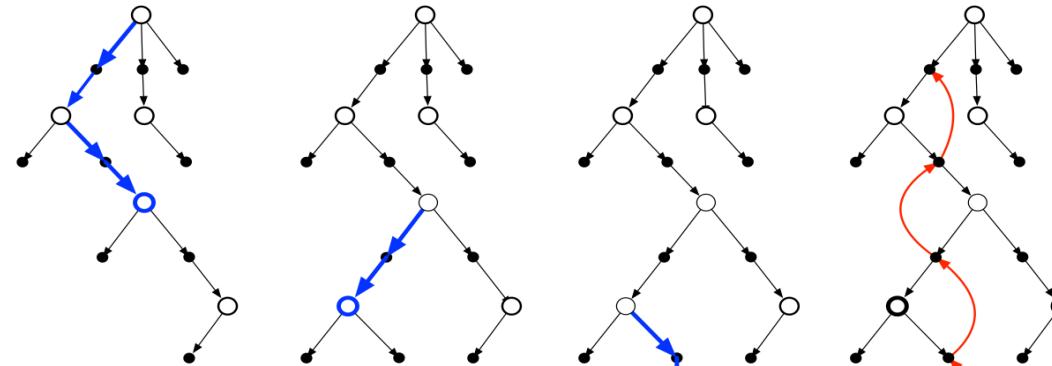
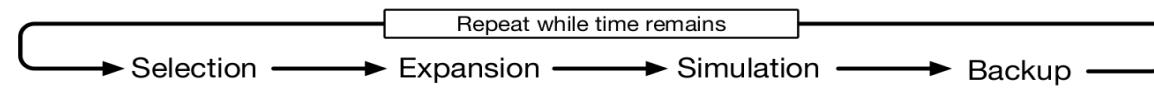
## How MCTS works?

- 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.

# Monte-Carlo Tree Search (MCTS)



# Monte-Carlo Tree Search (MCTS)

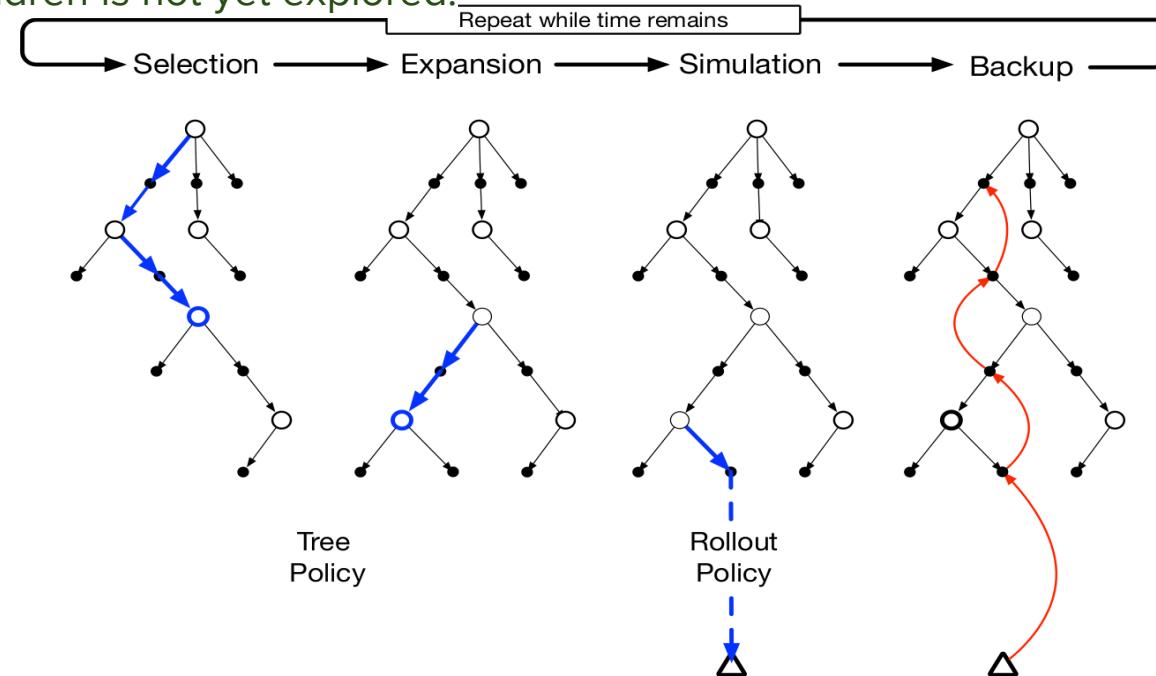


$$S_i = x_i + C \sqrt{\frac{\ln(t)}{n_i}}$$



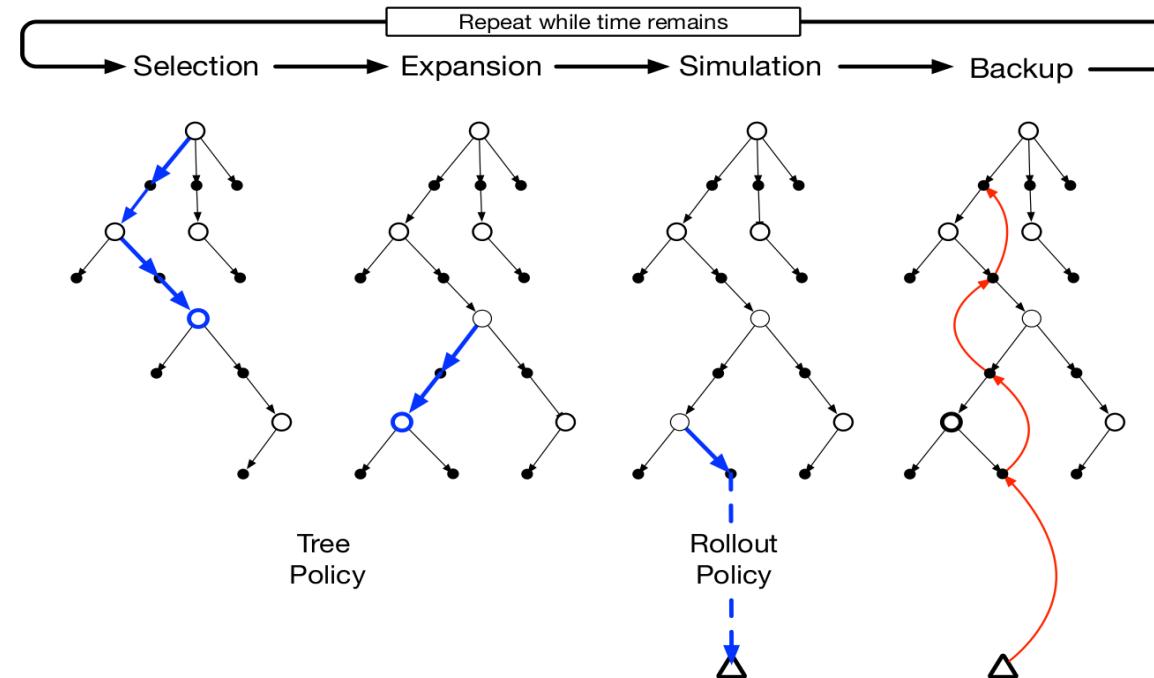
# Monte-Carlo Tree Search (MCTS) -- Selection

**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.



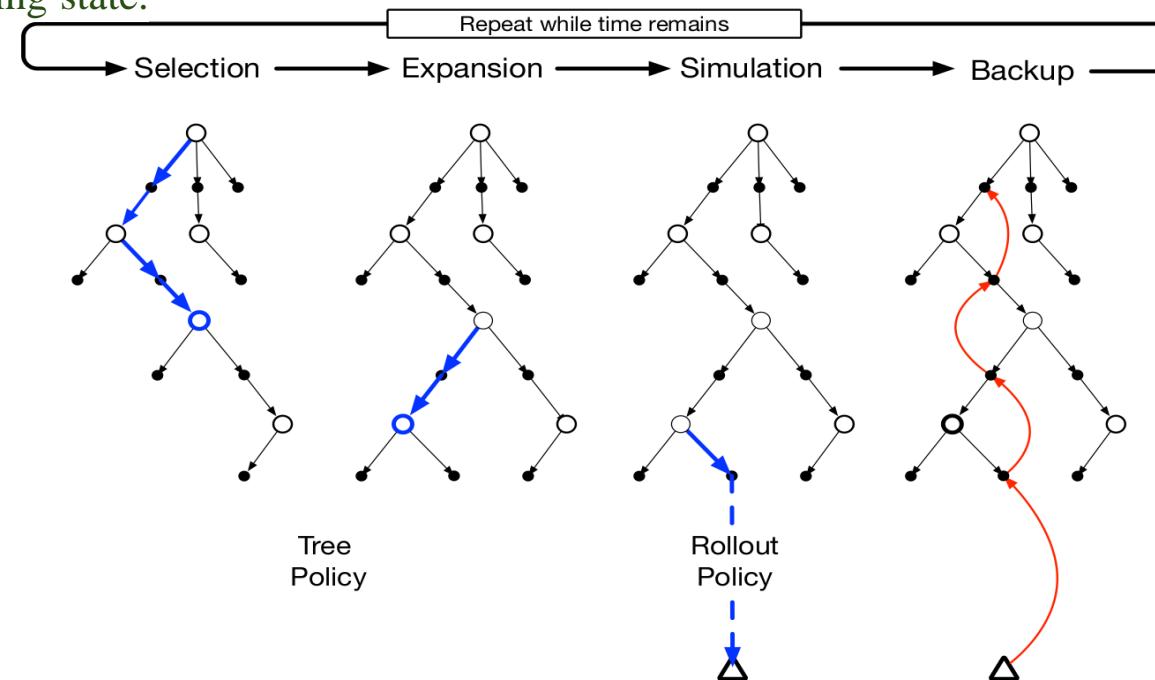
# Monte-Carlo Tree Search (MCTS) -- Expansion

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



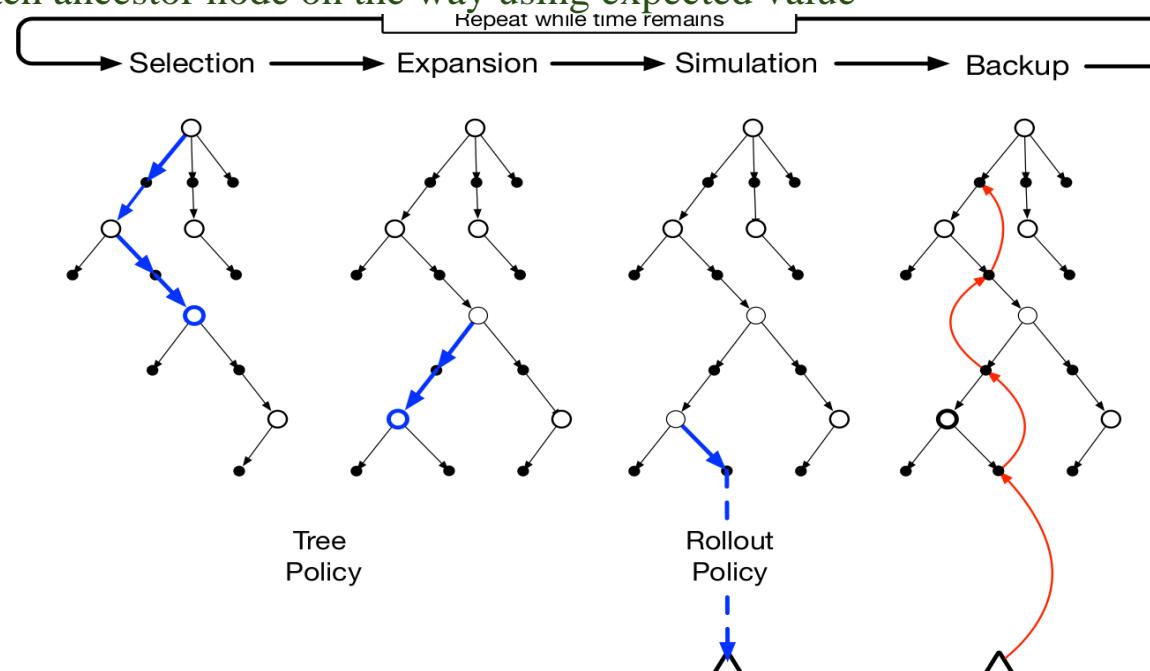
# Monte-Carlo Tree Search (MCTS) -- Simulation

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



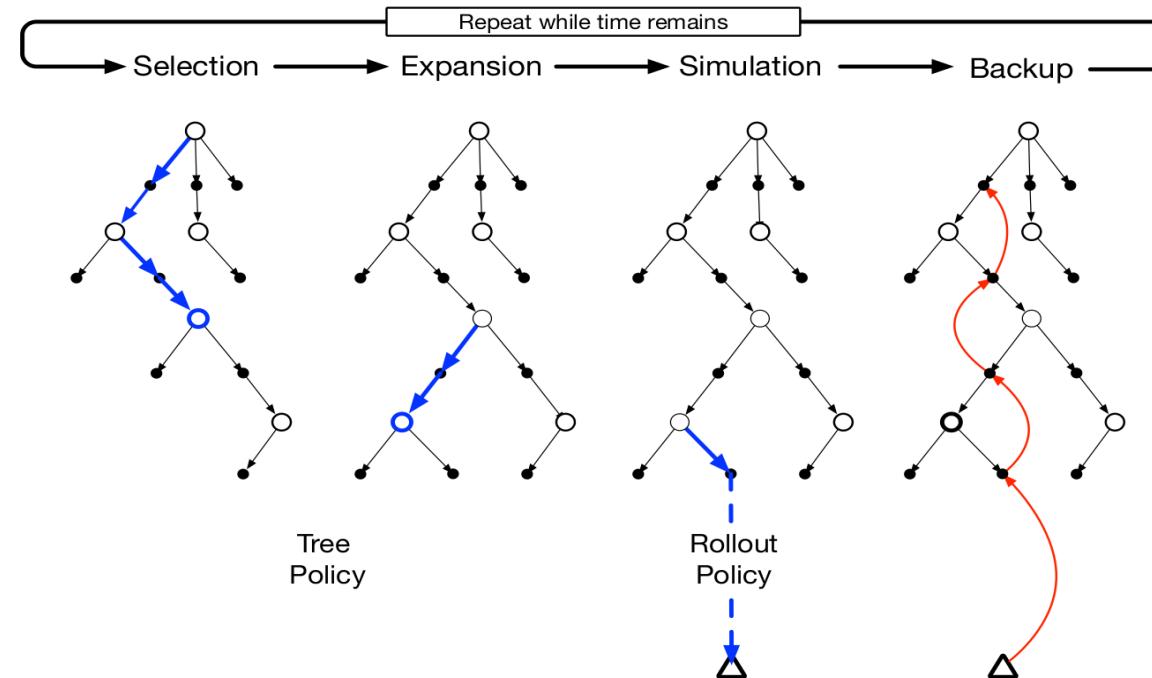
# Monte-Carlo Tree Search (MCTS) -- Backup

**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

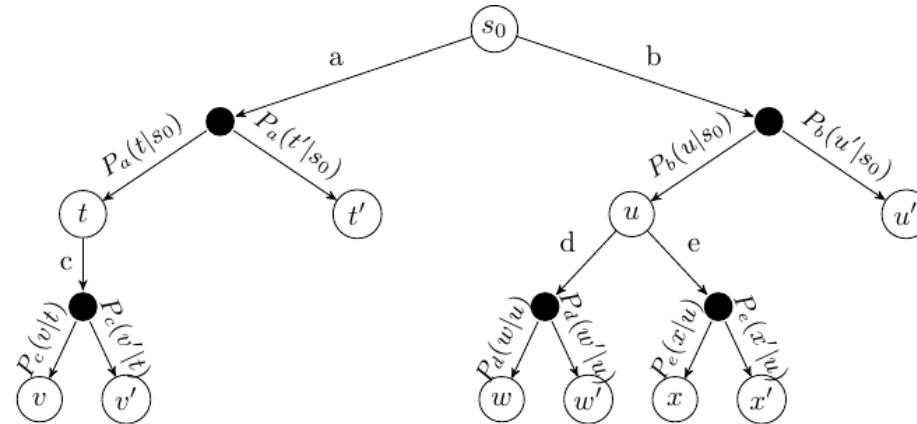


# Monte-Carlo Tree Search (MCTS) -- Summarizing

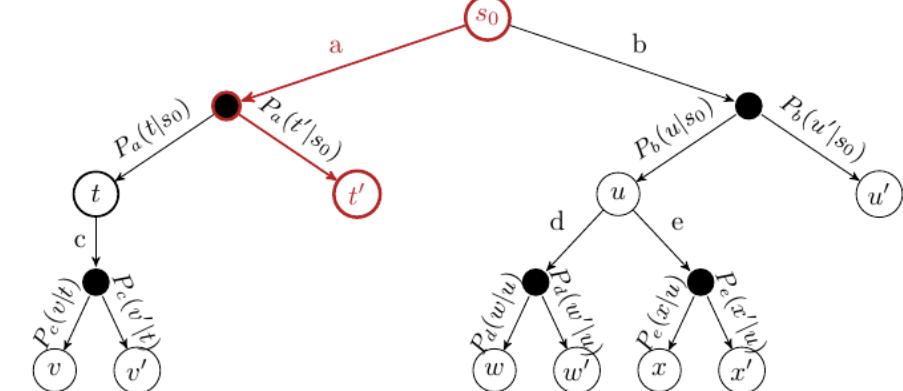
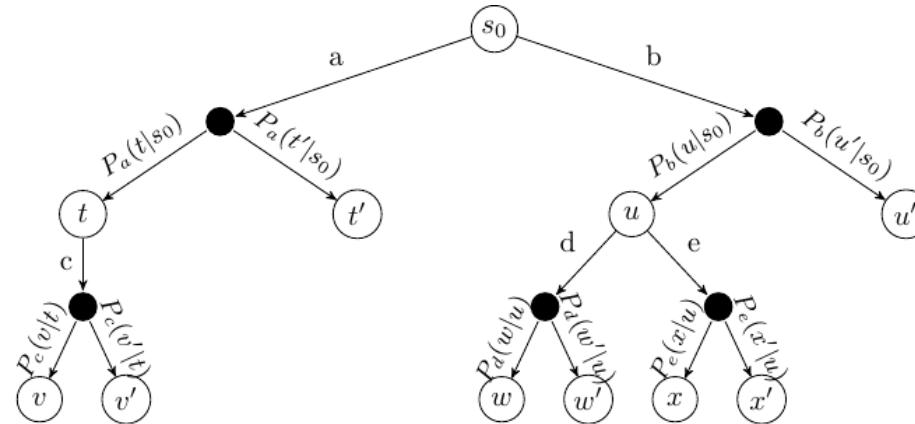
*Comments on the overall approach,,,*



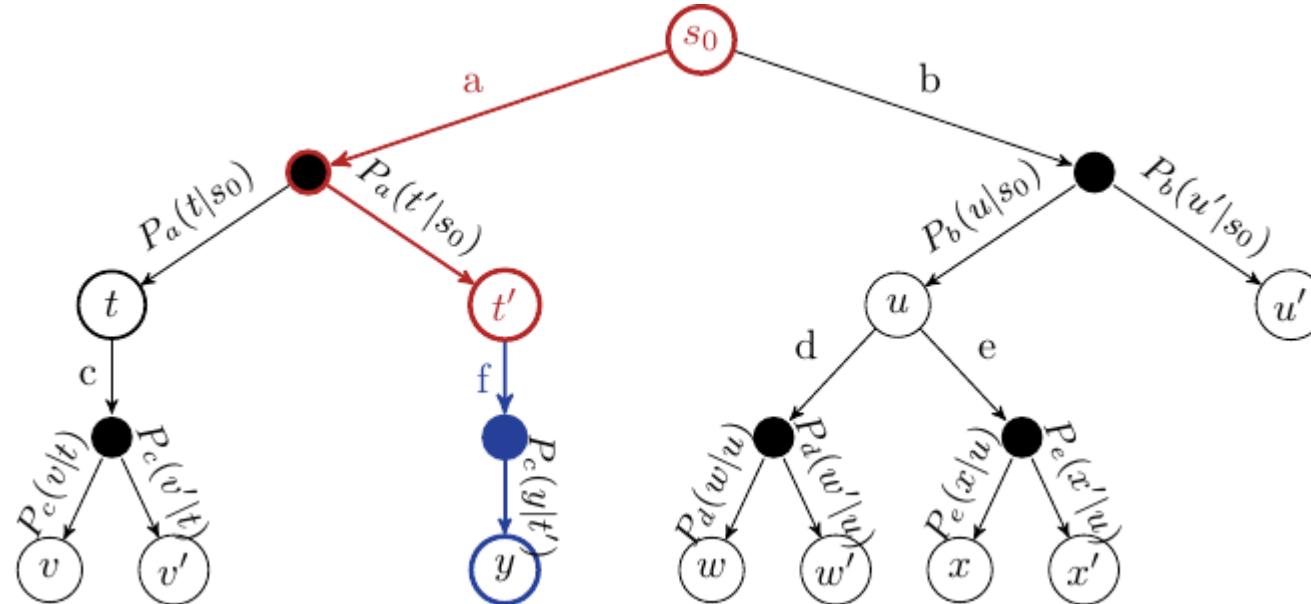
# Monte-Carlo Tree Search (MCTS) -- Selection



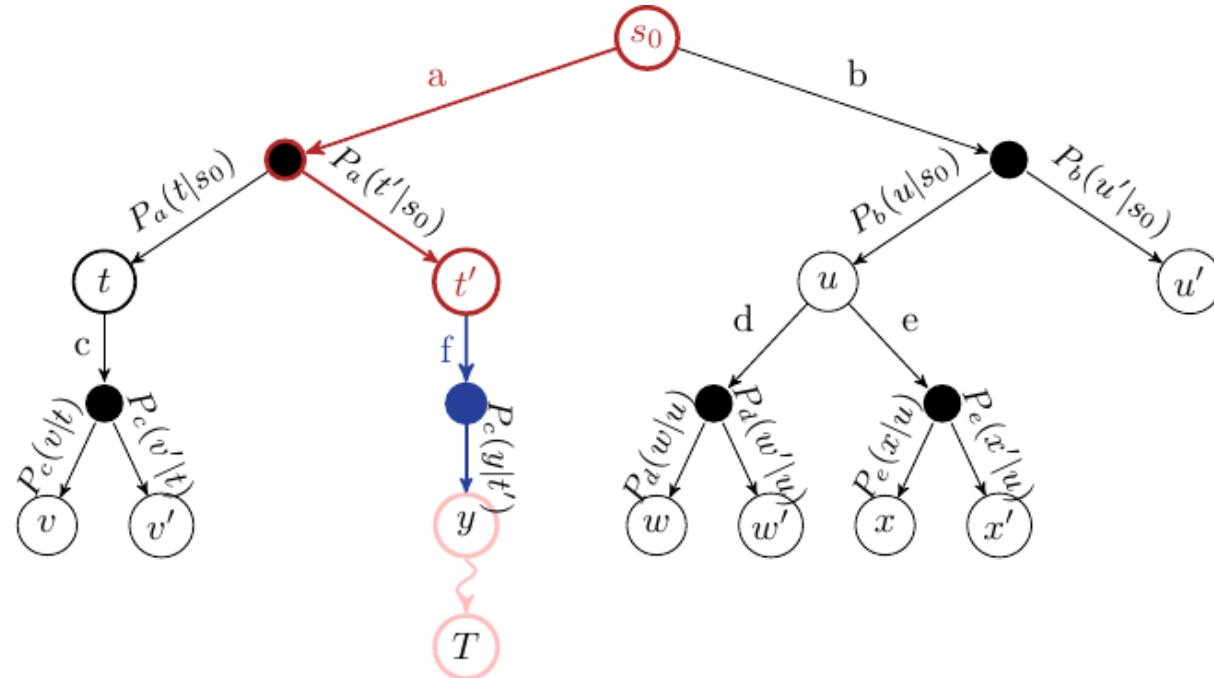
# Monte-Carlo Tree Search (MCTS) -- Selection



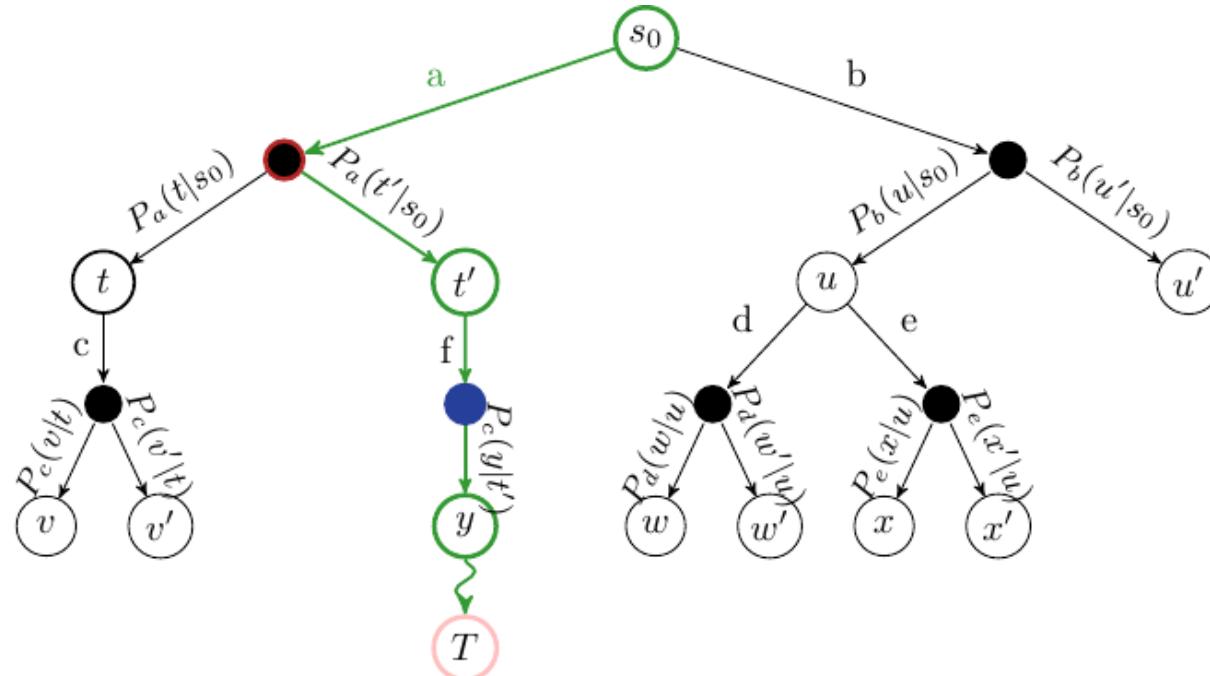
# Monte-Carlo Tree Search (MCTS) -- Expansion



# Monte-Carlo Tree Search (MCTS) -- Simulation



# Monte-Carlo Tree Search (MCTS) -- Backup



# Monte-Carlo Tree Search (MCTS)

## Algorithm – Monte-Carlo Tree Search

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

**Output:** updated Q-function  $Q$

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

# Monte-Carlo Tree Search (MCTS)

## Function – 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  $P_a(s' | s)$

$s \leftarrow s'$

**return**  $s$

# Monte-Carlo Tree Search (MCTS)

## 🔔 Function – 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  $P_a(s' | s)$  and observe reward  $r$

**return**  $s'$

# Monte-Carlo Tree Search (MCTS)

## ⚠ Procedure – Backpropagation( $s : S; a : A$ )

**Input:** state-action pair  $(s, a)$

**Output:** none

**do**

$$N(s, a) \leftarrow N(s, a) + 1$$

$$G \leftarrow r + \gamma G$$

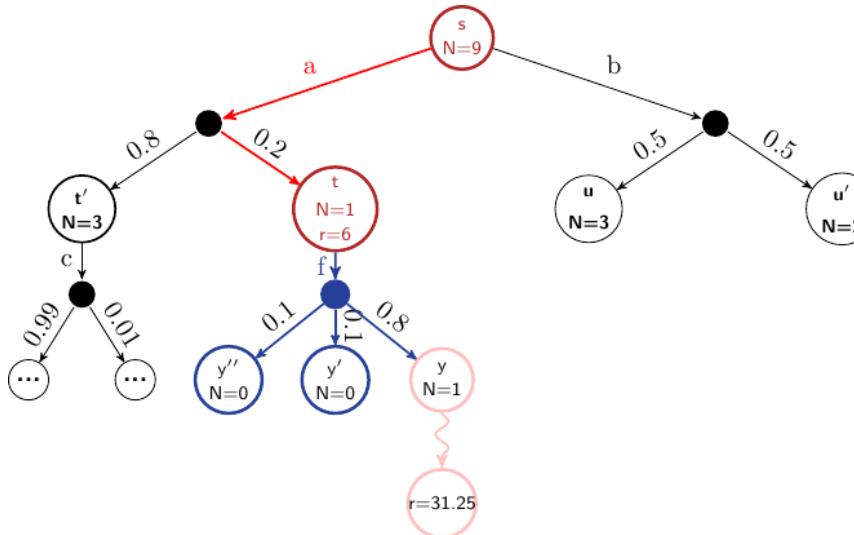
$$Q(s, a) \leftarrow Q(s, a) + \frac{1}{N(s, a)} [G - Q(s, a)]$$

$s \leftarrow$  parent of  $s$

$a \leftarrow$  parent action of  $s$

**while**  $s \neq s_0$

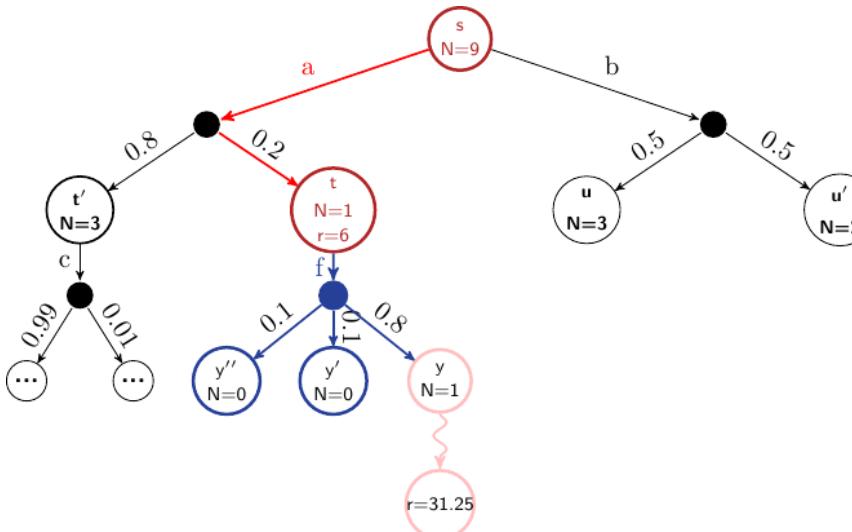
# Monte-Carlo Tree Search (MCTS)



Before backpropagation

$$\begin{aligned} Q(s, a) &= 18 \\ Q(t, f) &= 0 \end{aligned}$$

# Monte-Carlo Tree Search (MCTS)



Before backpropagation

$$\begin{aligned} Q(s, a) &= 18 \\ Q(t, f) &= 0 \end{aligned}$$

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

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

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

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



# Upper-Confidence-Bound Action Selection

- **$\epsilon$ -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

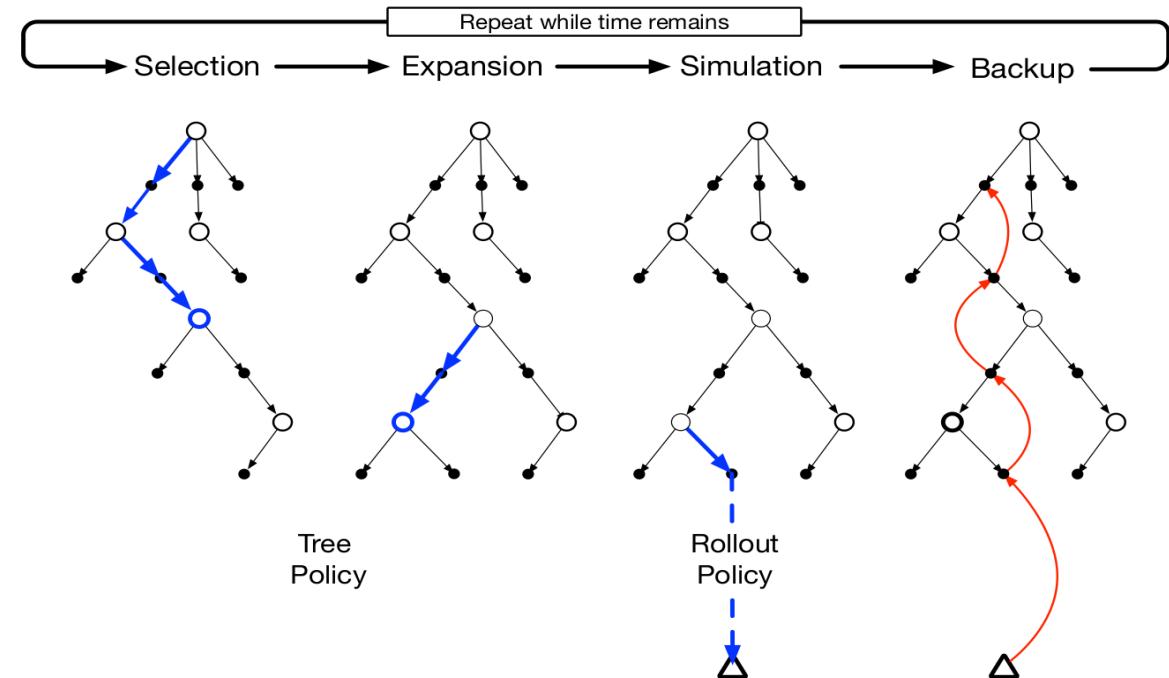
Measure of Uncertainty

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

# Monte-Carlo Tree Search (MCTS)

Can the selection of action in Tree policy use UCB?

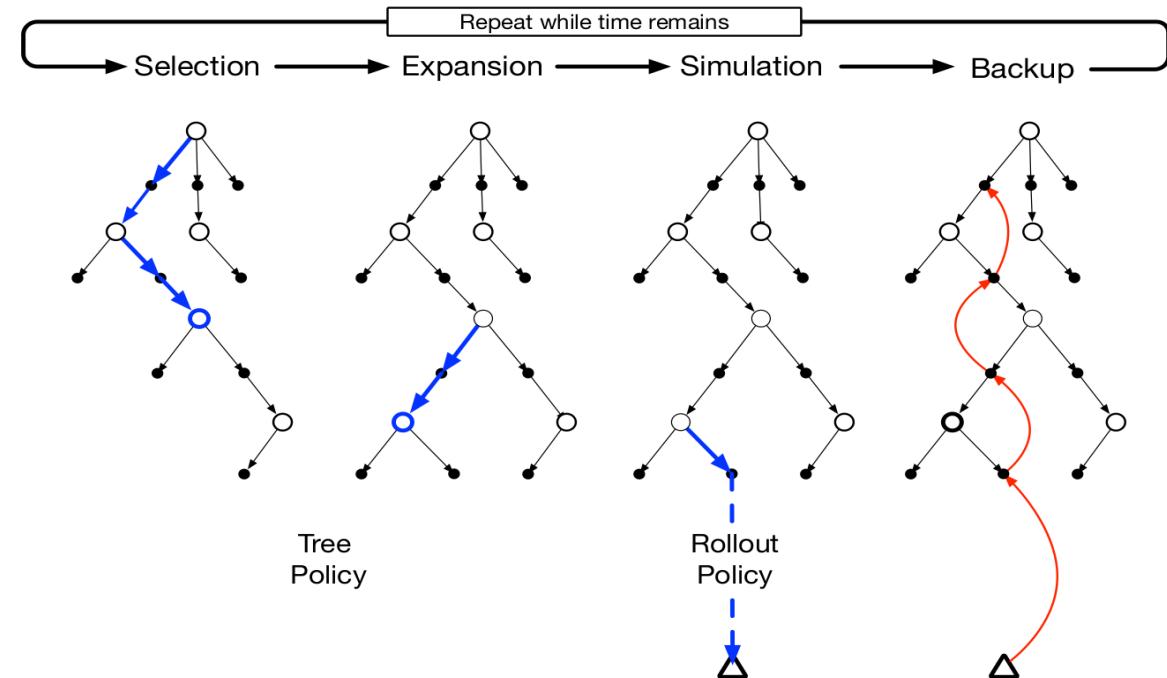
$$S_i = x_i + C \sqrt{\frac{\ln(t)}{n_i}}$$



# Monte-Carlo Tree Search (MCTS)

Can the selection of action in Tree policy use UCB?

Upper Confidence Trees (UCT):  
MCTS with UCB for Tree policy



# Required Readings and references

1. <https://rl-lab.com/#play>
2. <https://www.aionlinecourse.com/tutorial/machine-learning/upper-confidence-bound-%28ucb%29>
3. <https://towardsdatascience.com/monte-carlo-tree-search-in-reinforcement-learning-b97d3e743d0f>
4. <https://gibberblot.github.io/rl-notes/single-agent/mcts.html>
5. <https://towardsdatascience.com/alphazero-chess-how-it-works-what-sets-it-apart-and-what-it-can-tell-us-4ab3d2d08867>
6. <https://medium.com/geekculture/muzero-explained-a04cb1bad4d4>
7. <https://towardsdatascience.com/everything-you-need-to-know-about-googles-new-planet-reinforcement-learning-network-144c2ca3f284>
8. <https://blog.research.google/2019/02/introducing-planet-deep-planning.html?m=1>



**BITS** Pilani  
Pilani | Dubai | Goa | Hyderabad

Thank you



AIMLC ZG512/ZG525 -

Deep Reinforcement Learning / Computer Vision

## Games

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

1



## Agenda for the class (1/2)

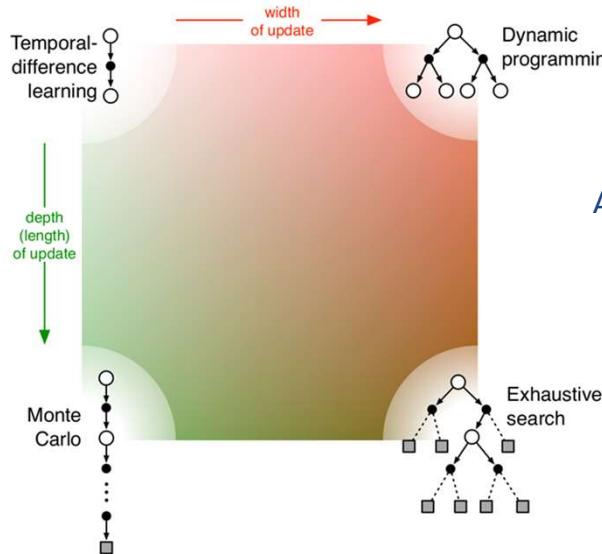
- Monte-Carlo Tree Search [ MCTS ]
- [AlphaGo](#)
- [AlphaGo Zero](#)
- [MuZero](#)

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

2



## Monte-Carlo Tree Search (MCTS)



A summary of pre-mid sem coverage !!!

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



## 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  $\pi$
- Aim of a rollout algorithm is to improve upon the rollout policy
  - Rollout policy could be completely random !!!

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



## 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
- 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

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



## Monte-Carlo Tree Search (MCTS)

### How MCTS works?

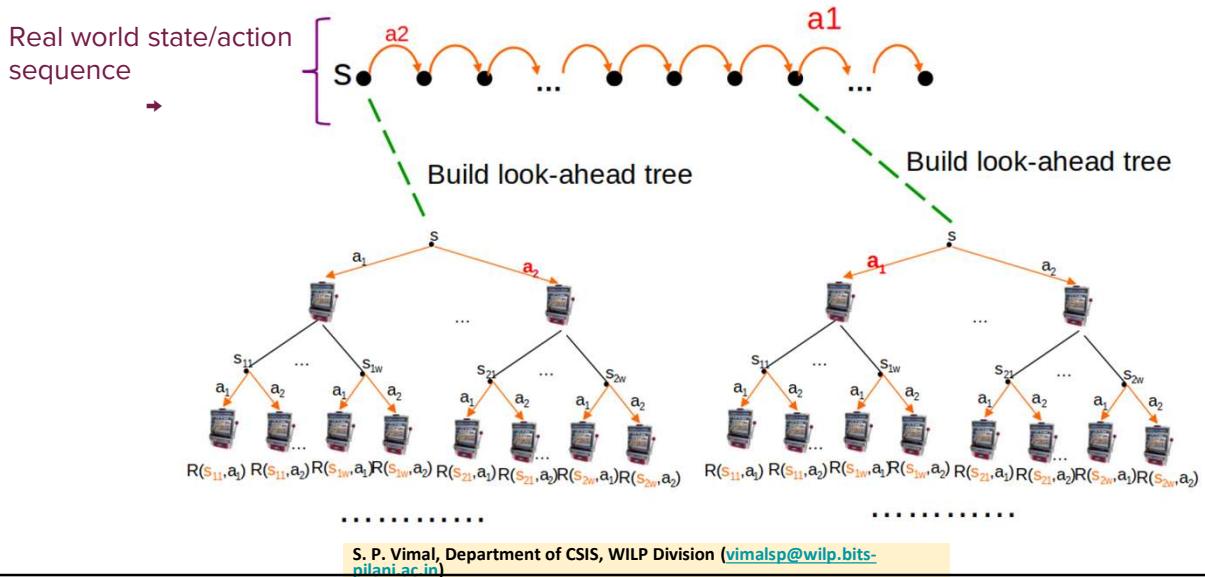
- MCTS is *executed* after encountering each new state ( $s$ )
  - [?] to select the agent's action for  $s$
- *Each execution is an iterative process* that simulates many trajectories starting from  $s$  and
  - running to a terminal state (or)
  - until discounting makes any further reward negligible 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.

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

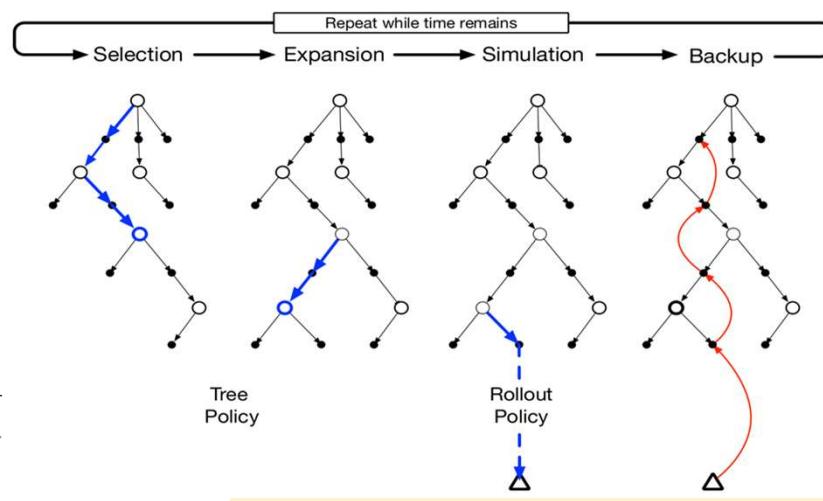


Slide from Prof. Alan Fern, Oregon State University

## Monte-Carlo Tree Search (MCTS)



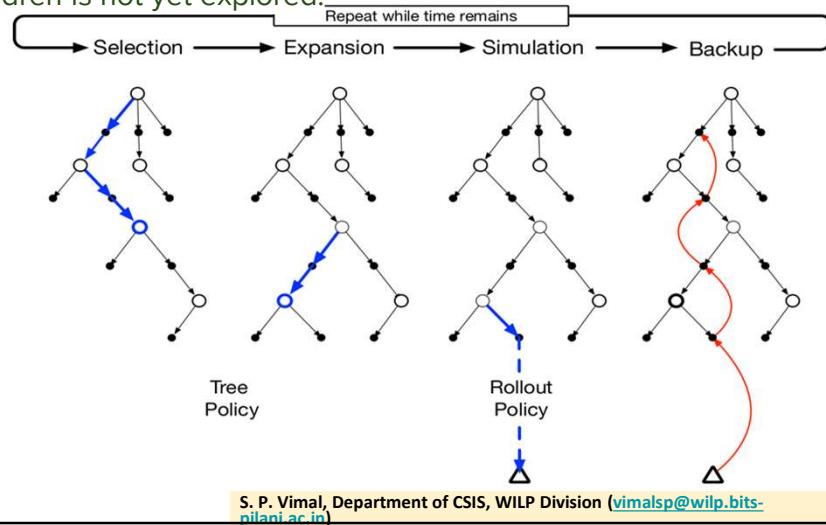
## Monte-Carlo Tree Search (MCTS)





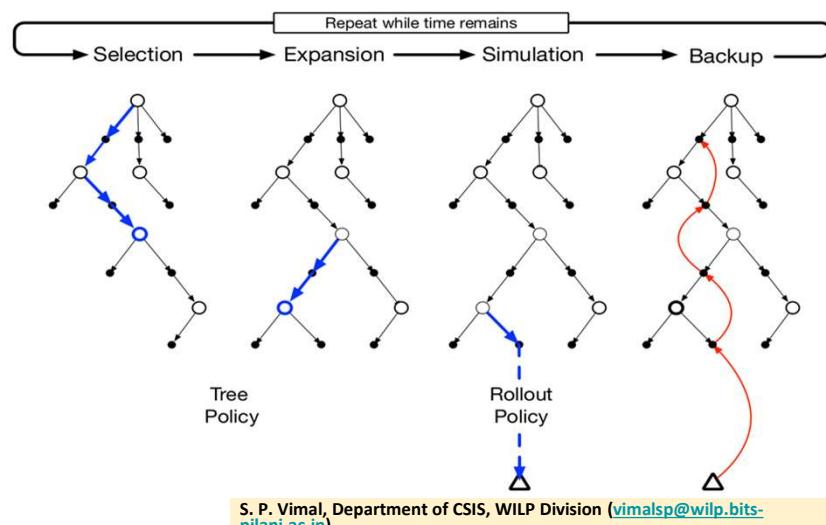
## Monte-Carlo Tree Search (MCTS) -- Selection

**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

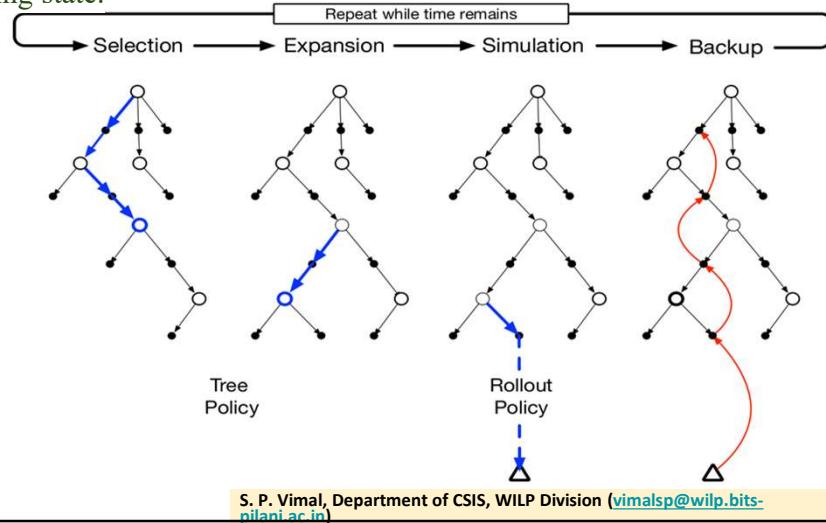
**Expand:** Expand this node by applying one available action (as defined by the MDP) from the node.





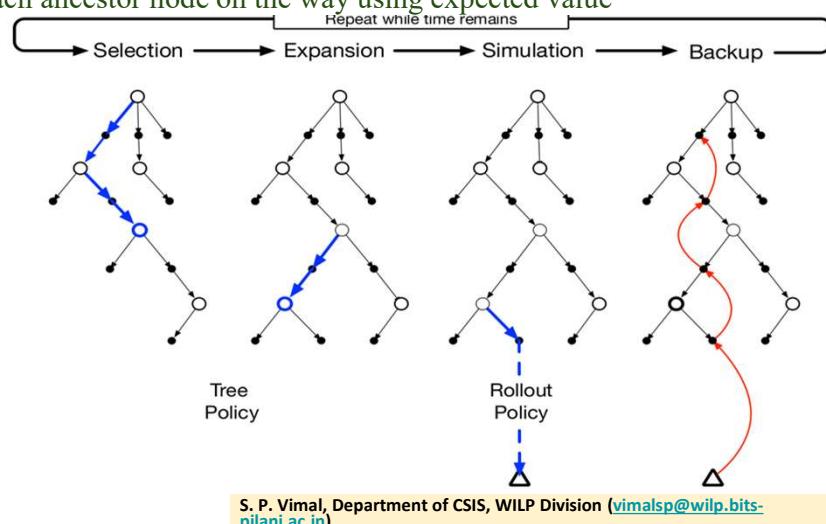
## Monte-Carlo Tree Search (MCTS) -- Simulation

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



## Monte-Carlo Tree Search (MCTS) -- Backup

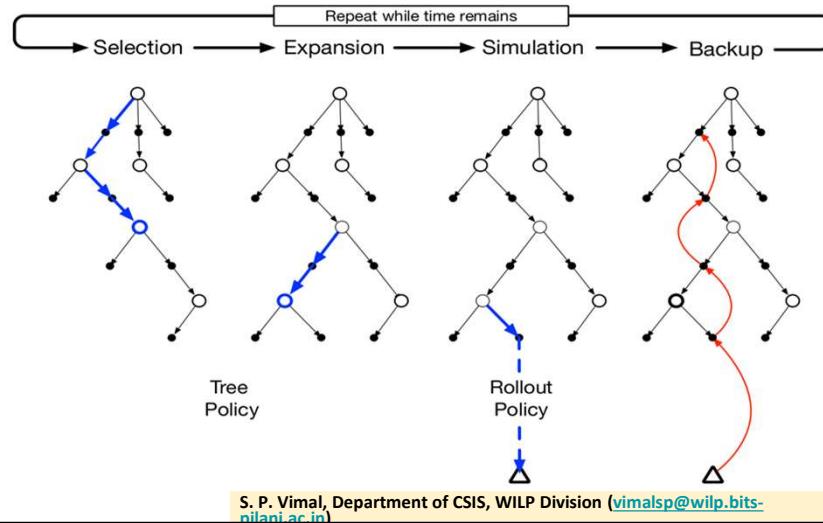
**Backup/ Backpropagate:** The value of the node is *back propagated* to the root node, updating the value of each ancestor node on the way using expected value





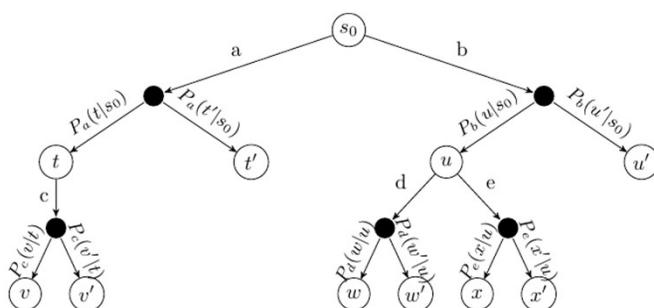
## Monte-Carlo Tree Search (MCTS) -- Summarizing

*Comments on the overall approach,,,,*

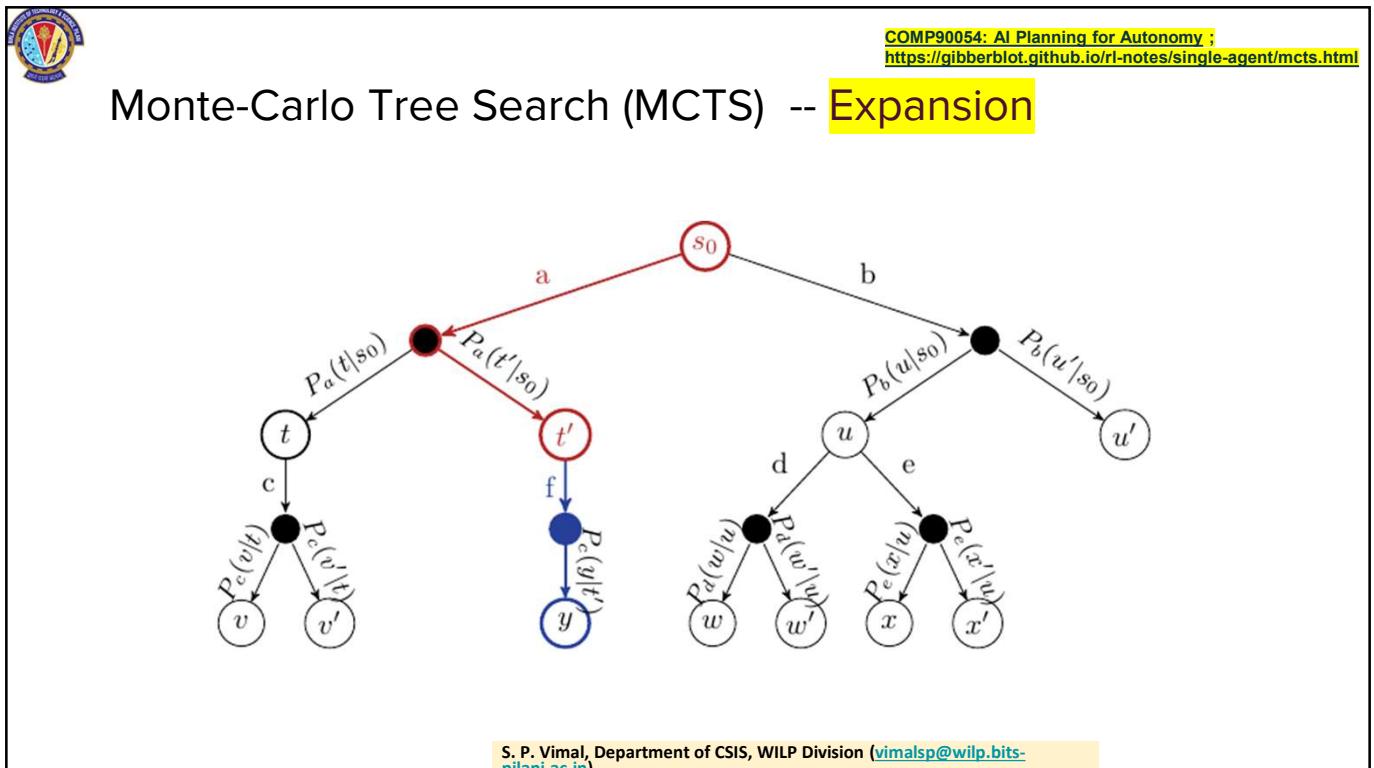
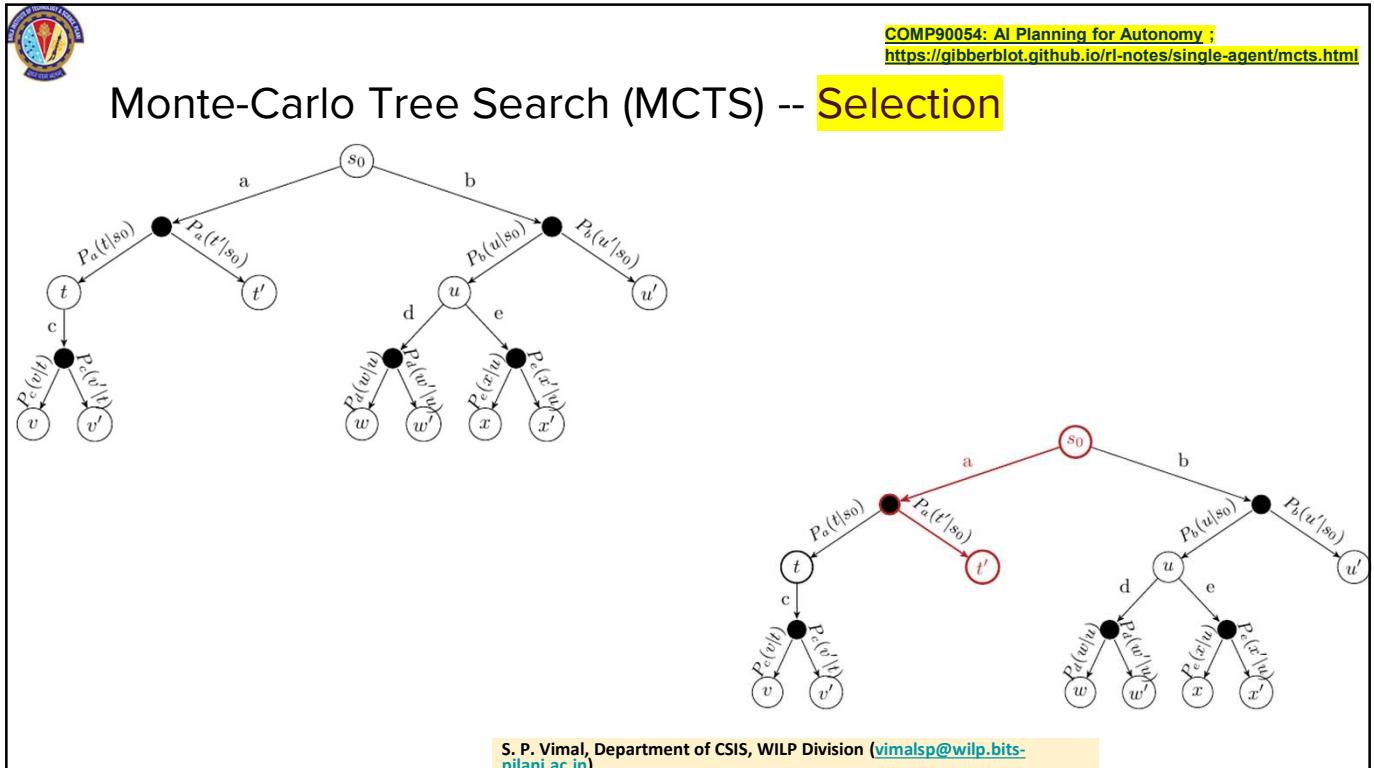


COMP90054: AI Planning for Autonomy ;  
<https://gibberblot.github.io/rl-notes/single-agent/mcts.html>

## Monte-Carlo Tree Search (MCTS) -- Selection

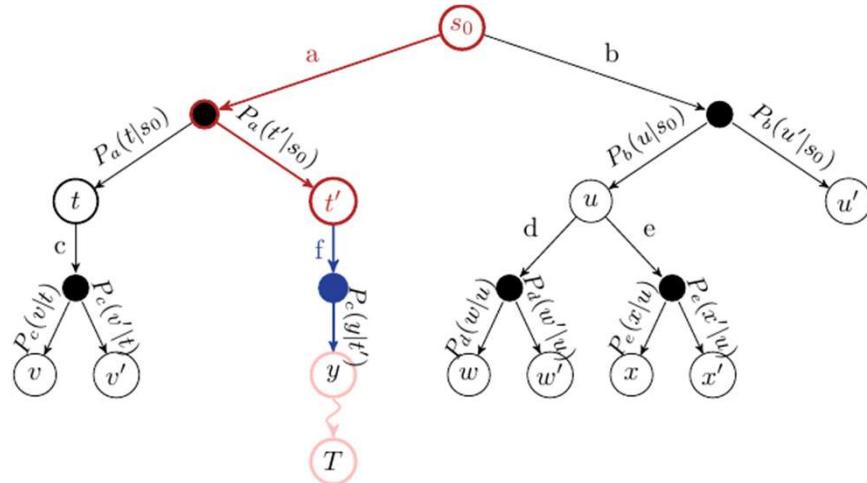


S. P. Vimal, Department of CSIS, WILP Division ([vimalsp@wilp.bits-pilani.ac.in](mailto:vimalsp@wilp.bits-pilani.ac.in))





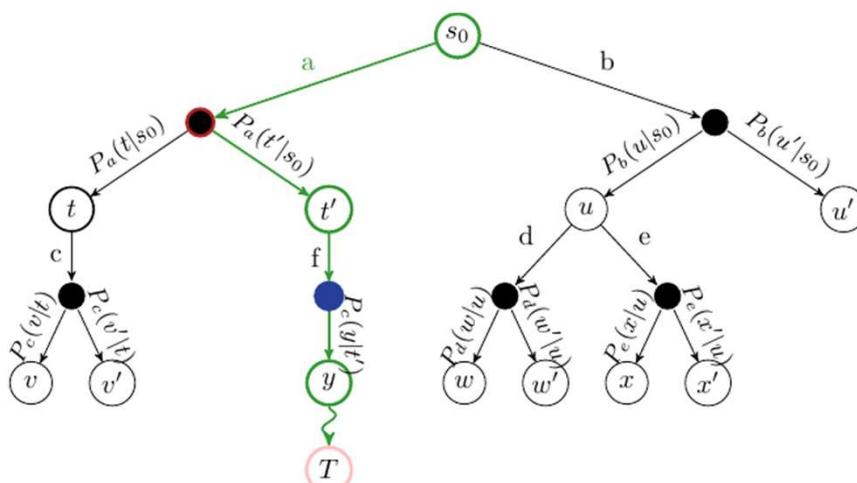
## Monte-Carlo Tree Search (MCTS) -- Simulation



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



## Monte-Carlo Tree Search (MCTS) -- Backup



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



## Monte-Carlo Tree Search (MCTS)

### Algorithm – Monte-Carlo Tree Search

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

```

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

```

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



## Monte-Carlo Tree Search (MCTS)

### Function – 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  $P_a(s' | s)$ 
     $s \leftarrow s'$ 
return  $s$ 

```

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



## Monte-Carlo Tree Search (MCTS)

### 🔔 Function – 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  $P_a(s' | s)$  and observe reward  $r$

**return**  $s'$

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



## Monte-Carlo Tree Search (MCTS)

### 🔔 Procedure – Backpropagation( $s : S; a : A$ )

**Input:** state-action pair  $(s, a)$

**Output:** none

**do**

```

 $N(s, a) \leftarrow N(s, a) + 1$ 
 $G \leftarrow r + \gamma G$ 
 $Q(s, a) \leftarrow Q(s, a) + \frac{1}{N(s, a)} [G - Q(s, a)]$ 
 $s \leftarrow \text{parent of } s$ 
 $a \leftarrow \text{parent action of } s$ 
while  $s \neq s_0$ 

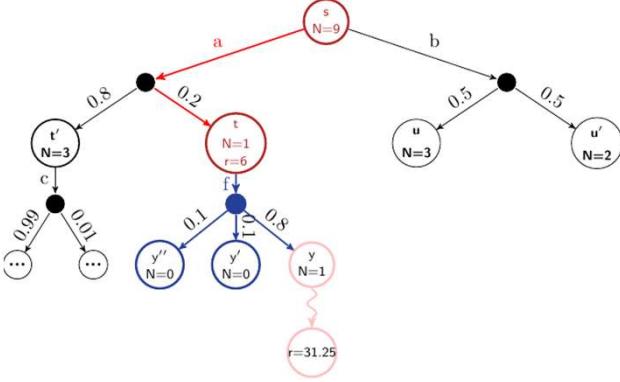
```

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



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

## Monte-Carlo Tree Search (MCTS)



Before backpropagation

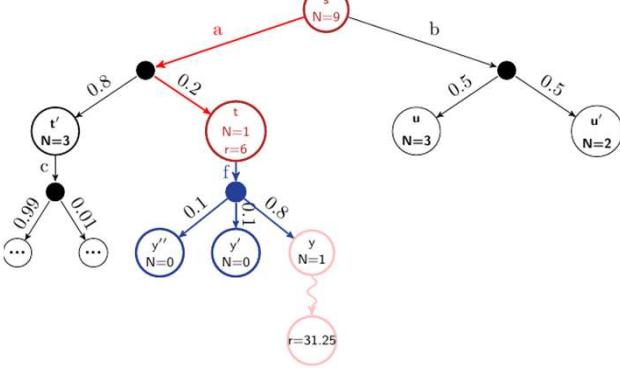
$$\begin{aligned} Q(s, a) &= 18 \\ Q(t, f) &= 0 \end{aligned}$$

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



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

## Monte-Carlo Tree Search (MCTS)



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

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

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

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

Before backpropagation

$$\begin{aligned} Q(s, a) &= 18 \\ Q(t, f) &= 0 \end{aligned}$$

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



## Upper-Confidence-Bound Action Selection

- $\epsilon$ -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](mailto:vimalsp@wilp.bits-pilani.ac.in))

25



## 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

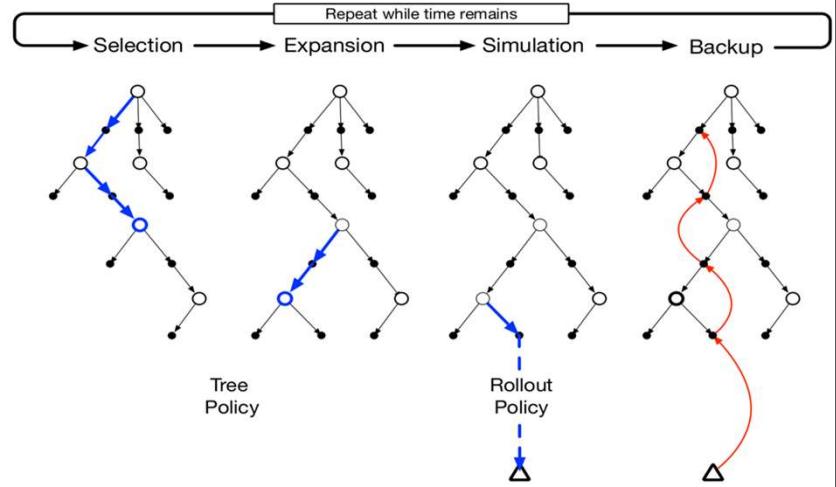
Action Value at time  $t$  for  $a$       Confidence Level  
 $A_t \doteq \arg \max_a \left[ Q_t(a) + c \sqrt{\frac{\ln t}{N_t(a)}} \right]$   
 Measure of Uncertainty

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

26



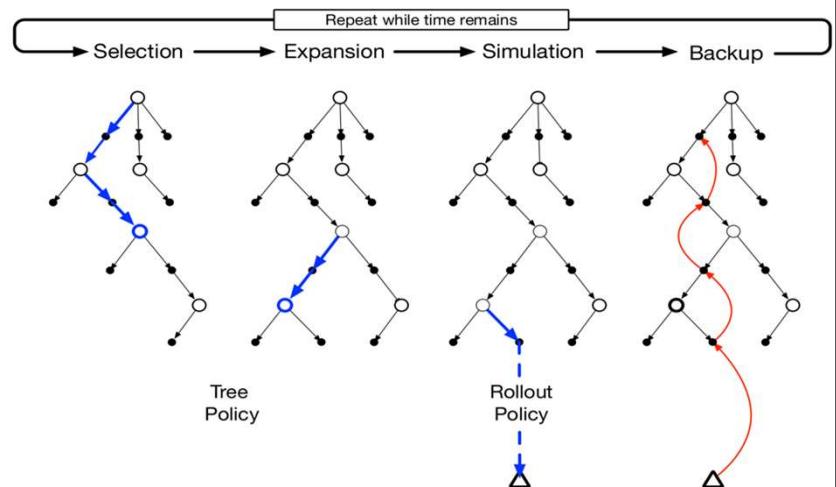
## Monte-Carlo Tree Search (MCTS)



## Monte-Carlo Tree Search (MCTS)

Can the selection of action in Tree policy use UCB?

Upper Confidence Trees (UCT):  
MCTS with UCB for Tree policy





## AlphaGo

### To discuss:

- How & Why AlphaGo was significant?
- Working of AlphaGo
- Why is Go a difficult game for AI?
- Role of MCTS in the AlphaGo

## ARTICLE

doi:10.1038/nature16961

### Mastering the game of Go with deep neural networks and tree search

David Silver<sup>1\*</sup>, Aja Huang<sup>1\*</sup>, Chris J. Maddison<sup>1</sup>, Arthur Guez<sup>1</sup>, Laurent Sifre<sup>1</sup>, George van den Driessche<sup>1</sup>, Julian Schrittwieser<sup>1</sup>, Ioannis Antonoglou<sup>1</sup>, Veda Panneershelvam<sup>1</sup>, Marc Lanctot<sup>1</sup>, Sander Dieleman<sup>1</sup>, Dominik Grewe<sup>1</sup>, John Nham<sup>2</sup>, Nal Kalchbrenner<sup>1</sup>, Ilya Sutskever<sup>2</sup>, Timothy Lillicrap<sup>1</sup>, Madeleine Leach<sup>1</sup>, Koray Kavukcuoglu<sup>1</sup>, Thore Graepel<sup>1</sup> & Demis Hassabis<sup>1</sup>

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. Here we introduce a new approach to computer Go that uses ‘value networks’ to evaluate board positions and ‘policy networks’ to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of state-of-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play. We also introduce a new search algorithm that combines Monte Carlo simulation with value and policy networks. Using this search algorithm, our program AlphaGo achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion by 5 games to 0. This is the first time that a computer program has defeated a human professional player in the full-sized game of Go, a feat previously thought to be at least a decade away.

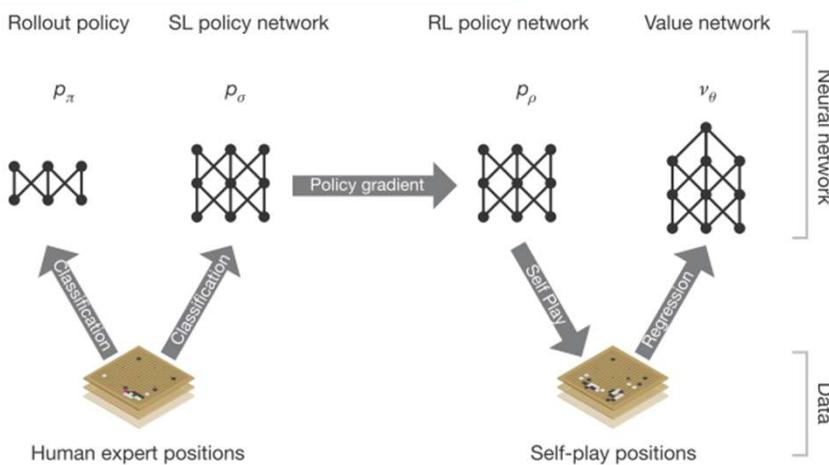
S. P. Vimal, Department of CSIS, WILP Division ([vimalsp@wilp.bits-pilani.ac.in](mailto:vimalsp@wilp.bits-pilani.ac.in))

29



## AlphaGo

### #1 - Components of the Pipeline



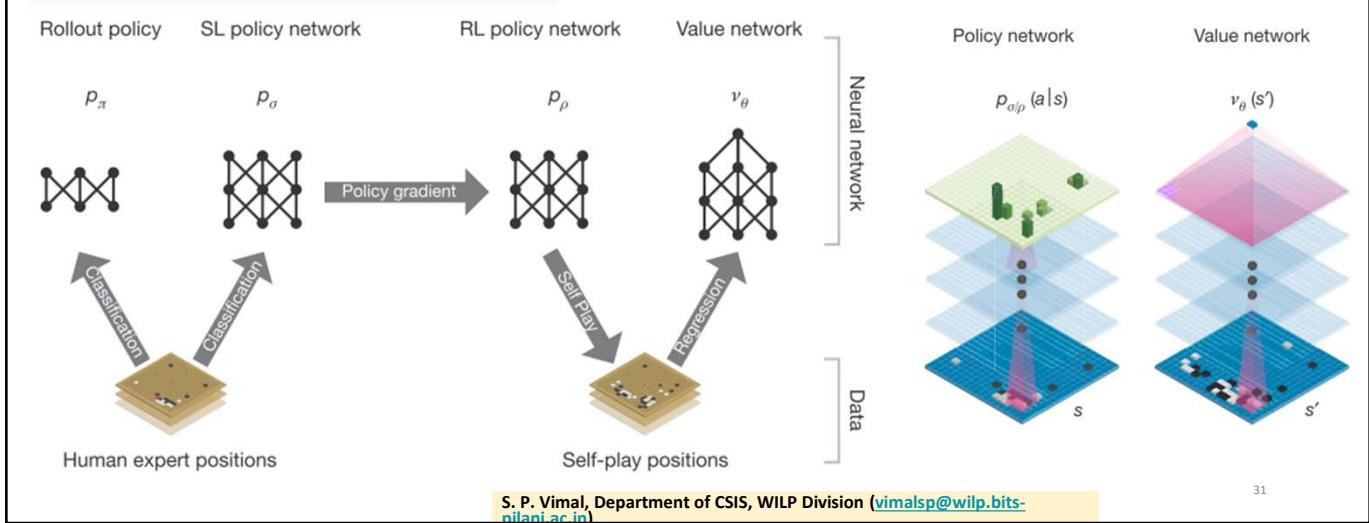
S. P. Vimal, Department of CSIS, WILP Division ([vimalsp@wilp.bits-pilani.ac.in](mailto:vimalsp@wilp.bits-pilani.ac.in))

30



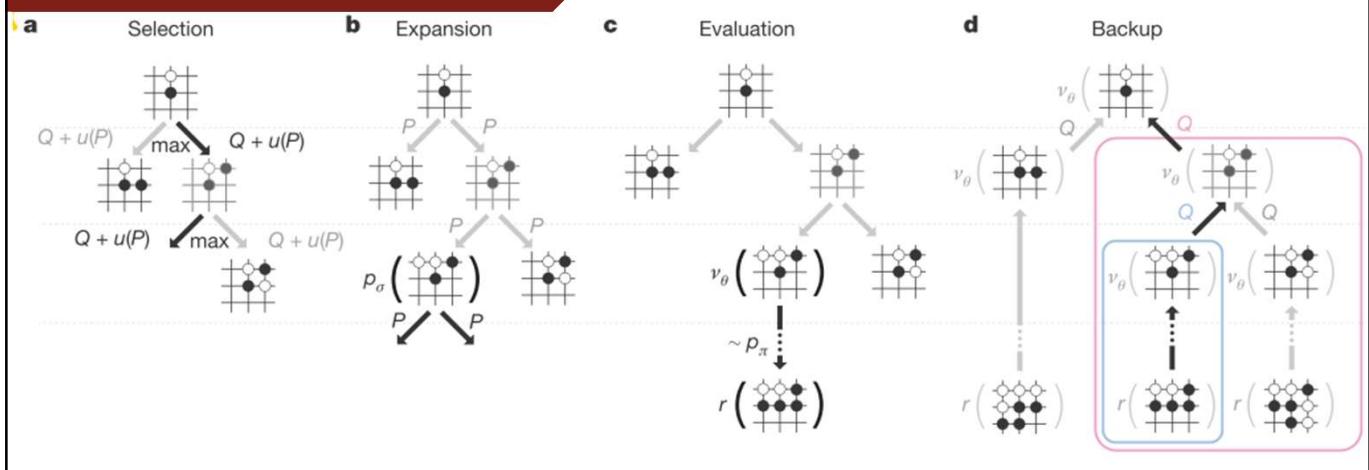
## AlphaGo

### #1 - Components of the Pipeline



## AlphaGo

### #2 - MCTS in AlphaGo





## AlphaGo Zero

# ARTICLE

doi:10.1038/nature24270

### To discuss:

- How AlphaGo Zero is different from AlphaGo?
- In what way AlphaGo Zero is significant to the field of AI?

## Mastering the game of Go without human knowledge

David Silver<sup>1\*</sup>, Julian Schrittwieser<sup>1\*</sup>, Karen Simonyan<sup>1\*</sup>, Ioannis Antonoglou<sup>1</sup>, Aja Huang<sup>1</sup>, Arthur Guez<sup>1</sup>, Thomas Hubert<sup>1</sup>, Lucas Baker<sup>1</sup>, Matthew Lai<sup>1</sup>, Adrian Bolton<sup>1</sup>, Yutian Chen<sup>1</sup>, Timothy Lillicrap<sup>1</sup>, Fan Hui<sup>1</sup>, Laurent Sifre<sup>1</sup>, George van den Driessche<sup>1</sup>, Thore Graepel<sup>1</sup> & Demis Hassabis<sup>1</sup>

A long-standing goal of artificial intelligence is an algorithm that learns, *tabula rasa*, superhuman proficiency in challenging domains. Recently, AlphaGo became the first program to defeat a world champion in the game of Go. The tree search in AlphaGo evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play. Here we introduce an algorithm based solely on reinforcement learning, without human data, guidance or domain knowledge beyond game rules. AlphaGo becomes its own teacher: a neural network is trained to predict AlphaGo's own move selections and also the winner of AlphaGo's games. This neural network improves the strength of the tree search, resulting in higher quality move selection and stronger self-play in the next iteration. Starting *tabula rasa*, our new program AlphaGo Zero achieved superhuman performance, winning 100–0 against the previously published, champion-defeating AlphaGo.

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



## AlphaGo Zero

### #1 - How AlphaGo Zero is different?

1. Trained solely by self-play RL [ self-play ]
2. Only black & white board position as input
3. Uses only single neural network (policy & value)
4. Simpler tree search without Monte-carlo rollouts

Our program, AlphaGo Zero, differs from AlphaGo Fan and AlphaGo Lee<sup>12</sup> in several important aspects. First and foremost, it is trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data. Second, it uses only the black and white stones from the board as input features. Third, it uses a single neural network, rather than separate policy and value networks. Finally, it uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte Carlo rollouts. To achieve these results, we introduce a new reinforcement learning algorithm that incorporates lookahead search inside the training loop, resulting in rapid improvement and precise and stable learning. Further technical differences in the search algorithm, training procedure and network architecture are described in Methods.

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

34



## AlphaGo Zero

### #1 - How AlphaGo Zero is different?

1. Trained solely by self-play RL [ self-play]
2. Only black & white board position as input
3. Uses only single neural network (policy & value)
4. Simpler tree search without Monte-carlo rollouts

#### About the network:

Our new method uses a deep neural network  $f_\theta$  with parameters  $\theta$ . This neural network takes as an input the raw board representation  $s$  of the position and its history, and outputs both move probabilities and a value,  $(p, v) = f_\theta(s)$ . The vector of move probabilities  $p$  represents the probability of selecting each move  $a$  (including pass),  $p_a = \Pr(a|s)$ . The value  $v$  is a scalar evaluation, estimating the probability of the current player winning from position  $s$ . This neural network combines the roles of both policy network and value network<sup>12</sup> into a single architecture. The neural network consists of many residual blocks<sup>4</sup> of convolutional layers<sup>16,17</sup> with batch normalization<sup>18</sup> and rectifier nonlinearities<sup>19</sup> (see Methods).

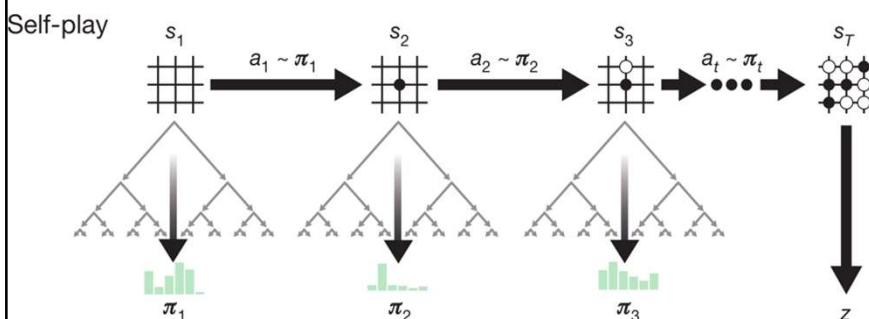
S. P. Vimal, Department of CSIS, WILP Division ([vimalsp@wilp.bits-pilani.ac.in](mailto:vimalsp@wilp.bits-pilani.ac.in))

35



## AlphaGo Zero

### #2 - Self-play Pipeline



1. In each position  $s_t$ , an MCTS  $a_t$  is executed using the latest neural network  $f_\theta$
2. Moves are selected according to the search probabilities computed by the MCTS,  $a_t \sim \pi_t$ .
3. The terminal position  $s_T$  is scored according to the rules of the game to compute the game winner  $z$

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

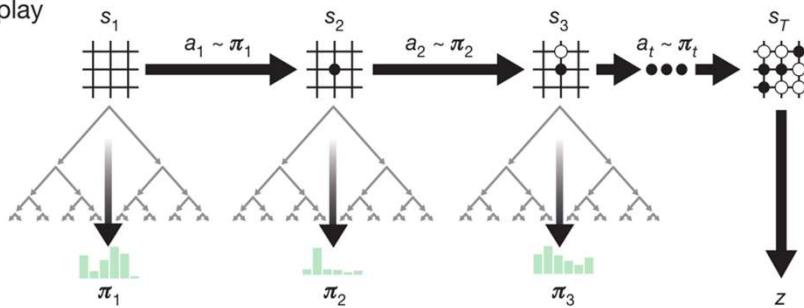
36



## AlphaGo Zero

### #2 - Self-play Pipeline

Self-play



repeatedly in a policy iteration procedure<sup>22,23</sup>; the neural network's parameters are updated to make the move probabilities and value ( $p, v = f_\theta(s)$ ) more closely match the improved search probabilities and self-play winner ( $\pi, z$ ); these new parameters are used in the next iteration of self-play to make the search even stronger. Figure 1 illustrates the self-play training pipeline.

1. In each position  $s_t$ , an MCTS  $a_t$  is executed using the latest neural network  $f_\theta$
1. Moves are selected according to the search probabilities computed by the MCTS,  $a_t \sim \pi_t$ .
2. The terminal position  $s_T$  is scored according to the rules of the game to compute the game winner  $z$

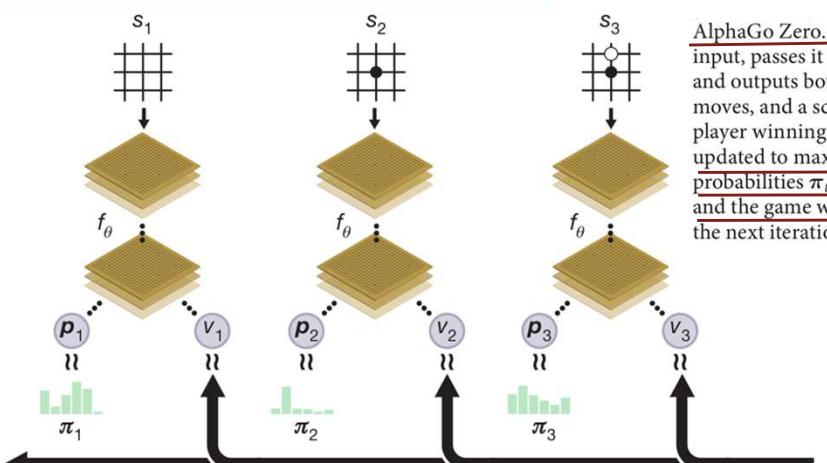
S. P. Vimal, Department of CSIS, WILP Division ([vimalsp@wilp.bits-pilani.ac.in](mailto:vimalsp@wilp.bits-pilani.ac.in))

37



## AlphaGo Zero

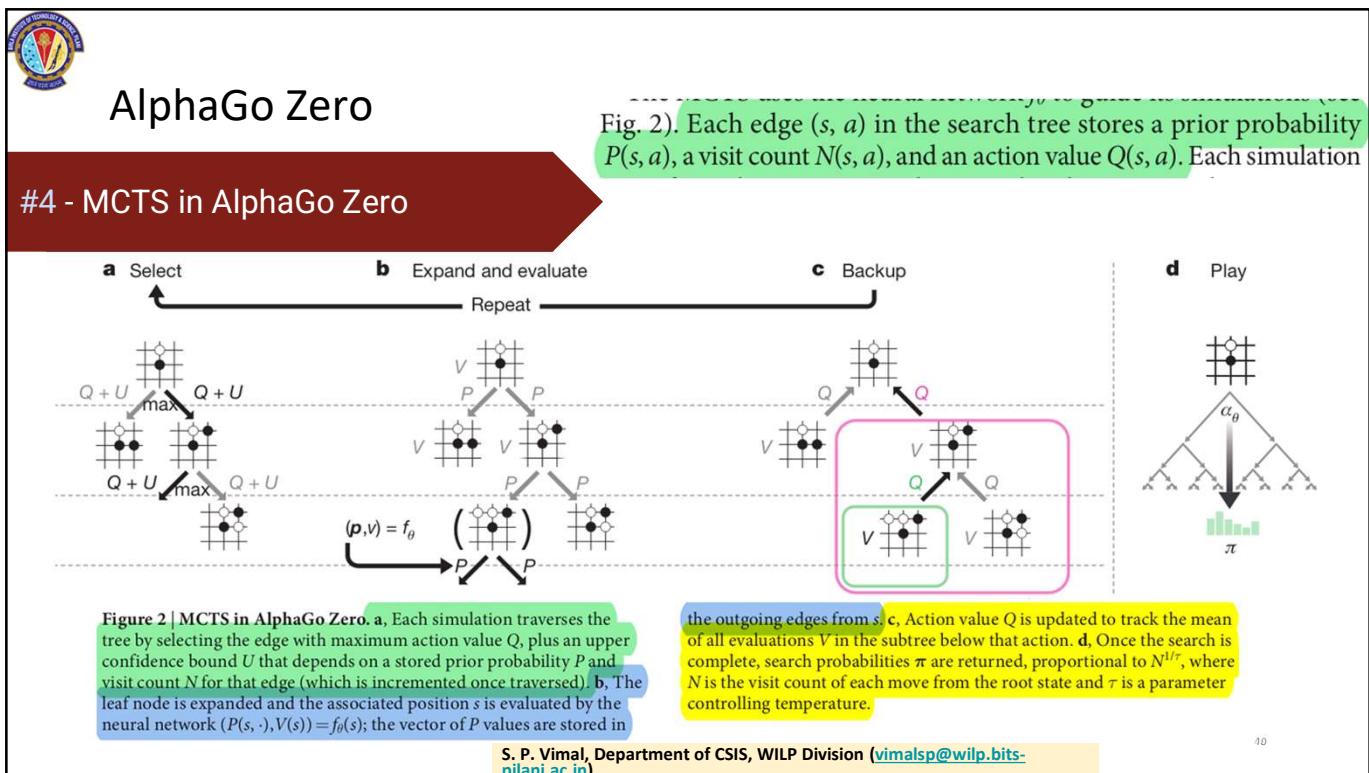
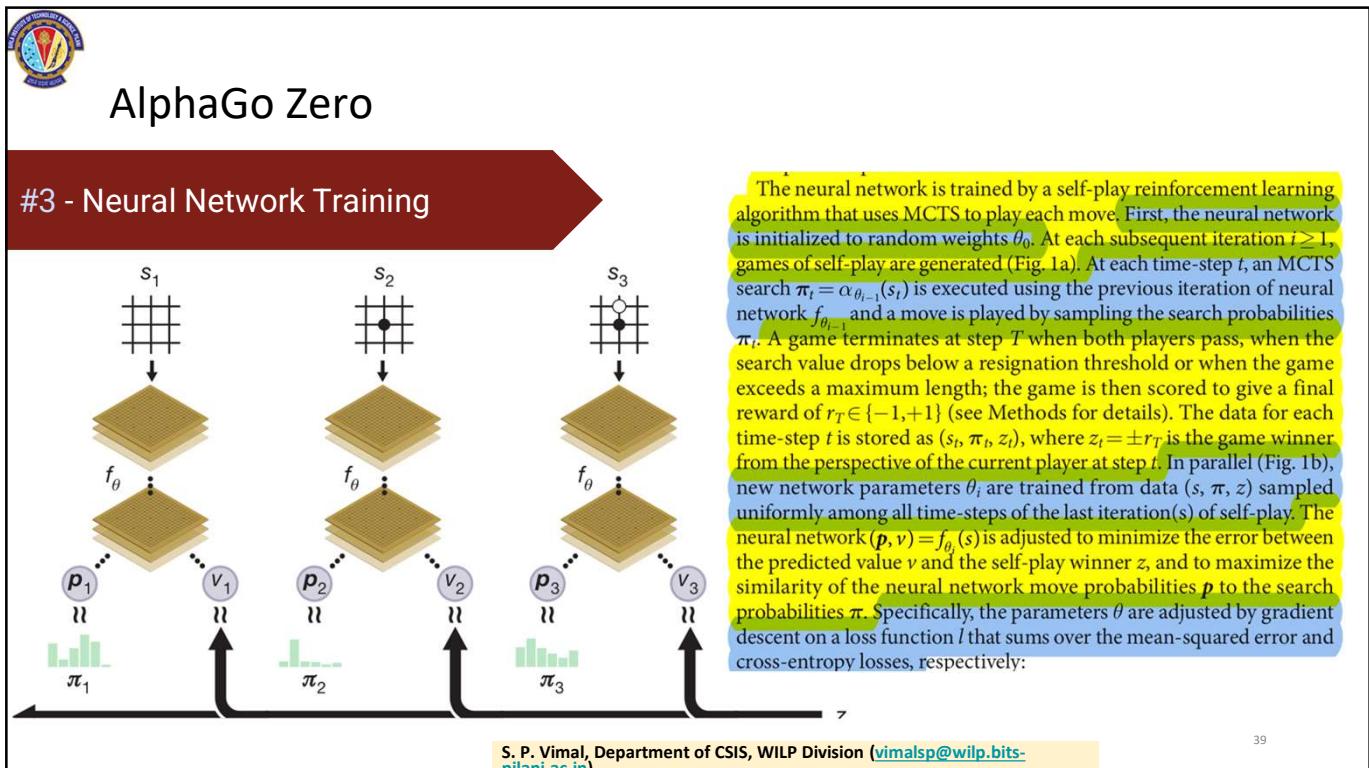
### #3 - Neural Network Training



Neural network training in AlphaGo Zero. The neural network takes the raw board position  $s_t$  as its input, passes it through many convolutional layers with parameters  $\theta$ , and outputs both a vector  $p_t$ , representing a probability distribution over moves, and a scalar value  $v_t$ , representing the probability of the current player winning in position  $s_t$ . The neural network parameters  $\theta$  are updated to maximize the similarity of the policy vector  $p_t$  to the search probabilities  $\pi_t$ , and to minimize the error between the predicted winner  $v_t$  and the game winner  $z$  (see equation (1)). The new parameters are used in the next iteration of self-play as in a.

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

38

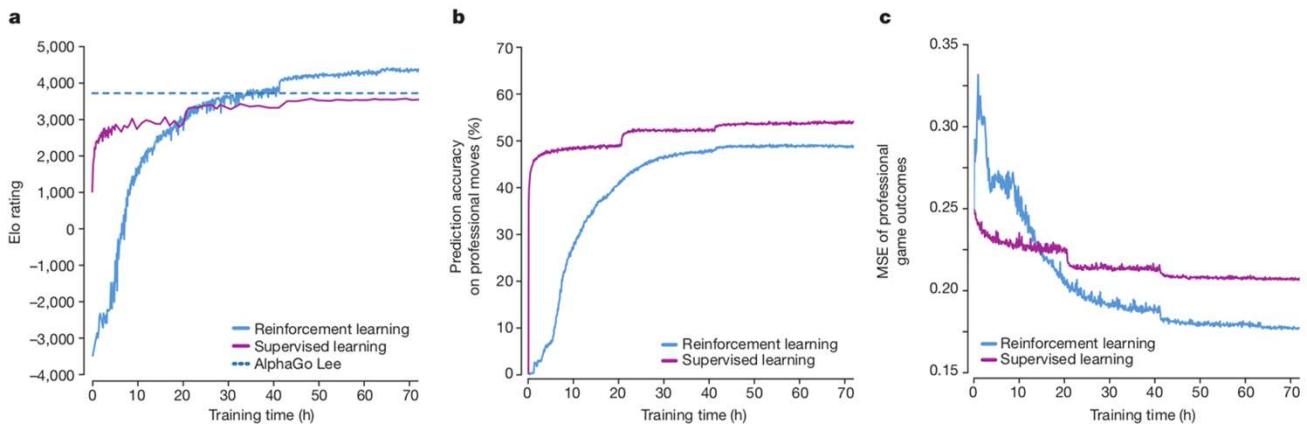




## AlphaGo Zero

### #5 - Some Analysis

Over the course of training, 4.9 million games of self-play were generated, using 1,600 simulations for each MCTS, which corresponds to approximately 0.4s thinking time per move. Parameters were updated from 700,000 mini-batches of 2,048 positions. The neural network contained 20 residual blocks (see Methods for further details).



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

41



## AlphaGo Zero

### #5 - Some Analysis

#### Conclusion

Our results comprehensively demonstrate that a pure reinforcement learning approach is fully feasible, even in the most challenging of domains: it is possible to train to superhuman level, without human examples or guidance, given no knowledge of the domain beyond basic rules. Furthermore, a pure reinforcement learning approach requires just a few more hours to train, and achieves much better asymptotic performance, compared to training on human expert data. Using this approach, AlphaGo Zero defeated the strongest previous versions of AlphaGo, which were trained from human data using handcrafted features, by a large margin.

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

42



## MuZero

### Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model

Julian Schrittwieser,<sup>1\*</sup> Ioannis Antonoglou,<sup>1,2\*</sup> Thomas Hubert,<sup>1\*</sup>  
 Karen Simonyan,<sup>1</sup> Laurent Sifre,<sup>1</sup> Simon Schmitt,<sup>1</sup> Arthur Guez,<sup>1</sup>  
 Edward Lockhart,<sup>1</sup> Demis Hassabis,<sup>1</sup> Thore Graepel,<sup>1,2</sup> Timothy Lillicrap,<sup>1</sup>  
 David Silver<sup>1,2\*</sup>

<sup>1</sup>DeepMind, 6 Pancras Square, London N1C 4AG.

<sup>2</sup>University College London, Gower Street, London WC1E 6BT.

\*These authors contributed equally to this work.

#### Abstract

Constructing agents with planning capabilities has long been one of the main challenges in the pursuit of artificial intelligence. Tree-based planning methods have enjoyed huge success in challenging domains, such as chess and Go, where a perfect simulator is available. However, in real-world problems the dynamics governing the environment are often complex and unknown. In this work we present the *MuZero* algorithm which, by combining a tree-based search with a learned model, achieves superhuman performance in a range of challenging and visually complex domains, without any knowledge of their underlying dynamics. *MuZero* learns a model that, when applied iteratively, predicts the quantities most directly relevant to planning: the reward, the action-selection policy, and the value function. When evaluated on 57 different Atari games - the canonical video game environment for testing AI techniques, in which model-based planning approaches have historically struggled - our new algorithm achieved a new state of the art. When evaluated on Go, chess and shogi, without any knowledge of the game rules, *MuZero* matched the superhuman performance of the *AlphaZero* algorithm that was supplied with the game rules.

43

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



## Required Readings and references

1. <https://rl-lab.com/#play>
2. <https://www.aionlinecourse.com/tutorial/machine-learning/upper-confidence-bound-%28ucb%29>
3. <https://towardsdatascience.com/monte-carlo-tree-search-in-reinforcement-learning-b97d3e743d0f>
4. <https://gibberblot.github.io/rl-notes/single-agent/mcts.html>
5. <https://towardsdatascience.com/alphazero-chess-how-it-works-what-sets-it-apart-and-what-it-can-tell-us-4ab3d2d08867>
6. <https://medium.com/geekculture/muzero-explained-a04cb1bad4d4>
7. <https://towardsdatascience.com/everything-you-need-to-know-about-googles-new-planet-reinforcement-learning-network-144c2ca3f284>
8. <https://blog.research.google/2019/02/introducing-planet-deep-planning.html?m=1>

44



Thank you



# Multi Agent Reinforcement Learning

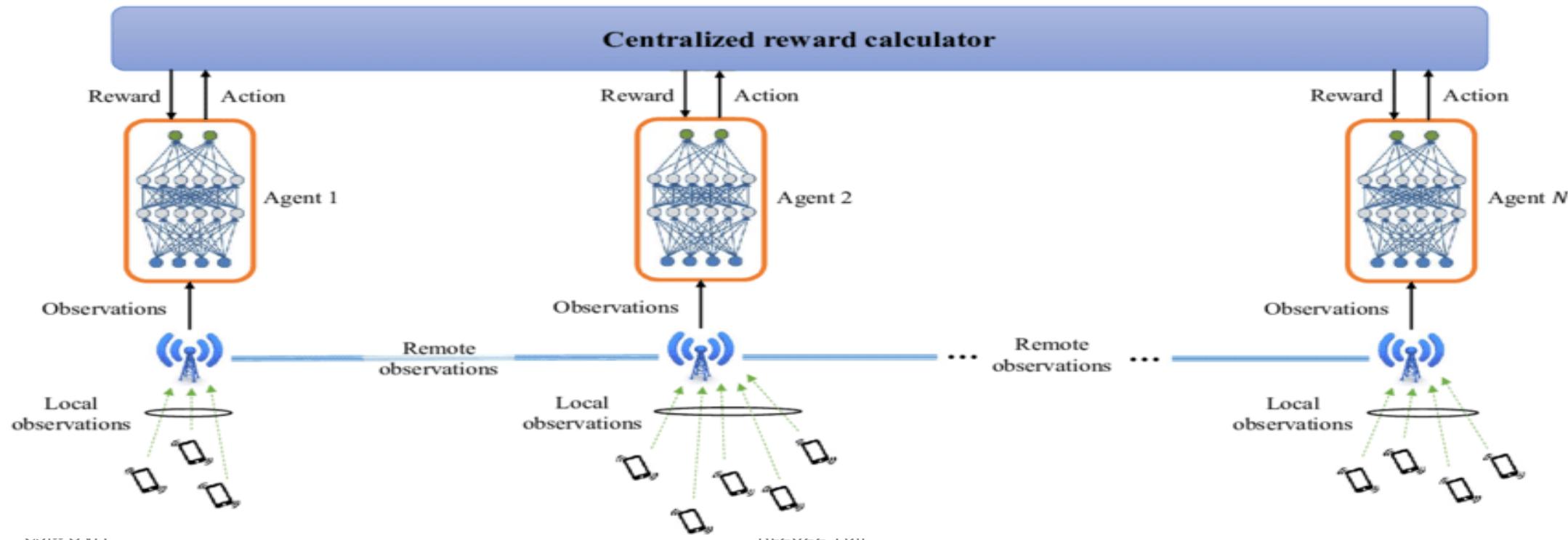
Dr.Chandra Sekhar Vorugunti

# 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:

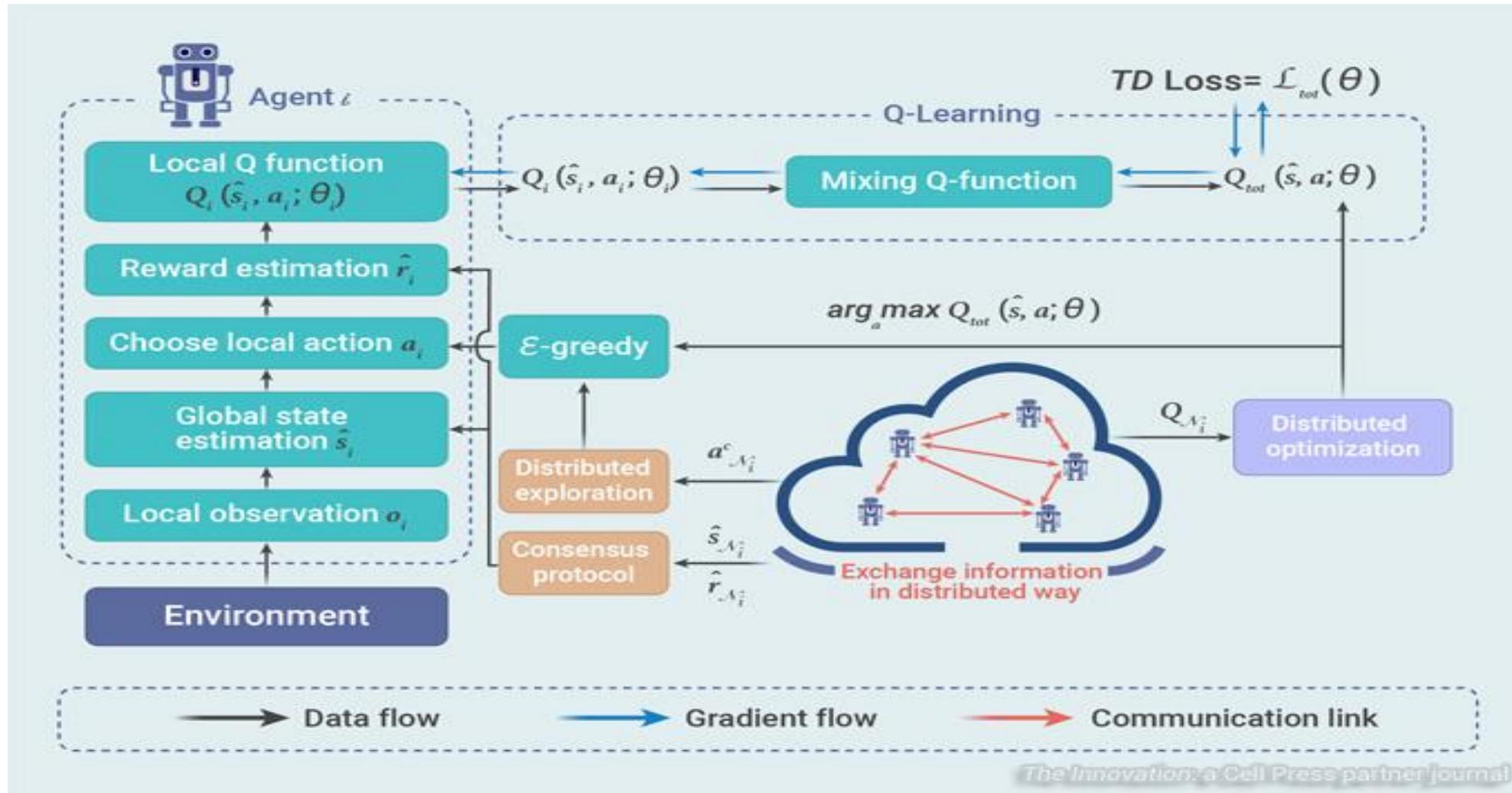
**Cooperative**: All agents working towards a common goal

**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)

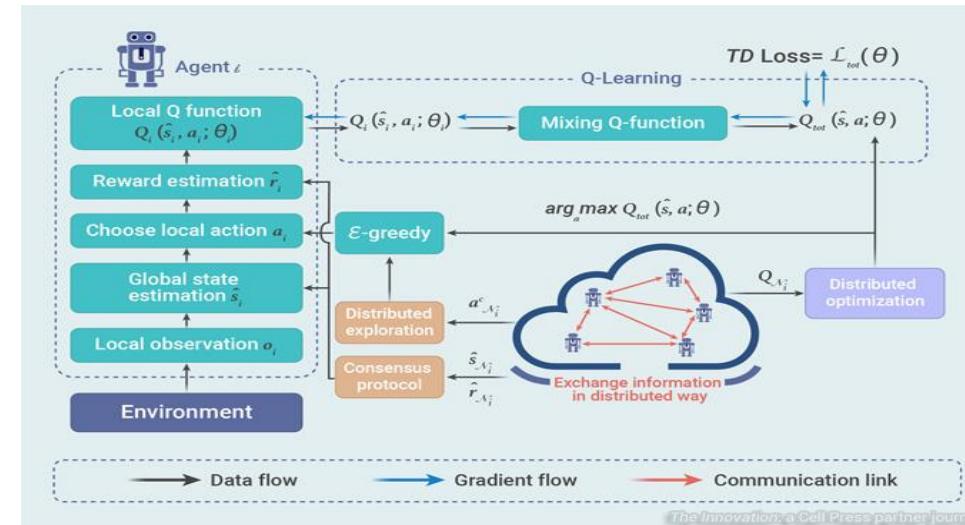


**Local Observation  $\mathbf{o}_i$ :** Each agent  $i$  receives an observation from the environment, which may represent a **partial** view of the entire state space.

**Global State Estimation  $\mathbf{s}_i$ :** 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  $\mathbf{a}_i$ :** Based on its state estimation, the agent selects an action to take. This decision is often made using an  $\epsilon$ -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  $\theta_i$ .
- **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  $\theta$  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  $\epsilon$ -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  $\theta_i$ , estimates the expected cumulative reward of taking action  $a_i$  in state  $s_i$  and following policy  $\pi$  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  $\alpha$  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  $f$ :

$$Q_{tot}(s, \mathbf{a}) = f(Q_1(s, a_1), \dots, Q_N(s, a_N))$$

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  $\theta_i$  is calculated as the squared difference between the predicted Q-values and the target Q-values:

$$\mathcal{L}(\theta_i) = (\hat{r}_i + \gamma \max_{a'_i} Q_i(s'_i, a'_i; \theta_i^-) - Q_i(s_i, a_i; \theta_i))^2$$

Here,  $\theta_i^-$  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  $\epsilon$  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  $\theta_i$  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:

$$Q_i^{new}(s_i, a_i) \leftarrow Q_i(s_i, a_i) + \alpha [\hat{r}_i + \gamma \max_{a'_i} Q_i(s'_i, a'_i) - Q_i(s_i, a_i)]$$

This equation indicates that the Q-value is adjusted towards the reward plus the discounted maximum future reward, modulated by a learning rate  $\alpha$ .

# Multi Agent Reinforcement Learning (MARL)



The expected return when taking action  $a$  in state  $s$  is calculated as:

$$Q(s, a) = \mathbb{E}[R_1 + \gamma R_2 + \gamma^2 R_3 + \dots | S_0 = s, A_0 = a]$$

where:

- $\mathbb{E}$  is the expectation operator.
- $R_t$  is the reward received after  $t$  time steps.
- $\gamma$  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  $\pi^*$  guides the selection of future actions.

## Q-function Update:

- The Q-function is updated using the Temporal Difference (TD) learning rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

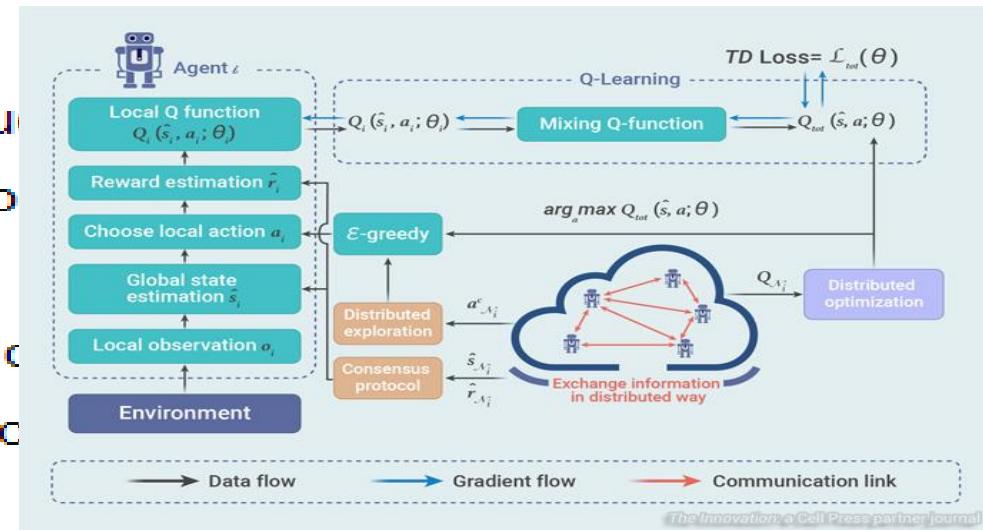
where  $\alpha$  is the learning rate, and  $\max_{a'} Q(s', a')$  is the maximum predicted Q-value for the next state  $s'$  over all possible actions.

# Multi Agent Reinforcement Learning (MARL)



## 7. Mixing Q-function

In a collaborative MARL setting, there might be a global value function for joint actions. It's a function of individual Q-values and possibly a mixing function:  $Q_{tot}(s, a; \Theta) = f(Q_1(s, a_1; \theta_1), \dots, Q_N(s, a_N; \theta_N), s)$ , where  $\Theta$  denotes the collective parameters of all agents, and  $f$  mixes individual Q-values in a manner that accounts for the joint action space.



## 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:

AIMLC ZG512 -  
Deep Reinforcement Learning

**BITS Pilani**  
Pilani | Dubai | Goa | Hyderabad

**Session 17: Special Topics in DRL**

**Introducing - Safety in Reinforcement Learning**

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

**Safety**

**BIG QUESTION:**  
How do we guarantee safety when we deploy RL in real-world applications?

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

**Agenda for the session**

- What do we mean by **Safety** in RL?
- Safe RL – Problem
- Some Applications (Autonomous Driving)

**Source for the session & Recommended Reading:**  
A Review of Safe Reinforcement Learning: Methods, Theory and Applications, 2023, Shangding Gu, Long Yang, Yali Du, Guang Chen, Florian Walter, Jun Wang, Yaodong Yang, Alois Knoll [ Available from <https://arxiv.org/pdf/2205.10330.pdf> ]

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

**Safety**

**BIG QUESTION:**  
How do we guarantee **safety** when we deploy RL in real-world applications?

What is Safety?

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

 **Safety**

**Some Definitions of Safety:**

- (1) the condition of being protected from or unlikely to cause danger, risk, or injury [oxford]
- (2) being protected from harm or other dangers
- (3) controlling recognized dangers to attain an acceptable level of risk

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

 **Safety**

**2H3W problems:**

- (1) **Safety Policy.** How can we perform policy optimisation to search for a safe policy?
- (2) **Safety Complexity.** How much training data is required to find a safe policy?
- (3) **Safety Applications.** What is the up to date progress of safe RL applications?
- (4) **Safety Benchmarks.** What benchmarks can we use to fairly and holistically examine safe RL performance?
- (5) **Safety Challenges.** What are the challenges faced in future safe RL research?

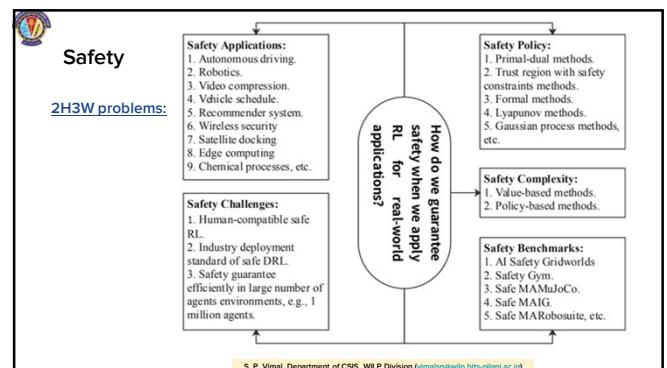
S. P. Vimal, Department of CSIS, WILP Division ([vimalsp@wilp.bits-pilani.ac.in](mailto:vimalsp@wilp.bits-pilani.ac.in))

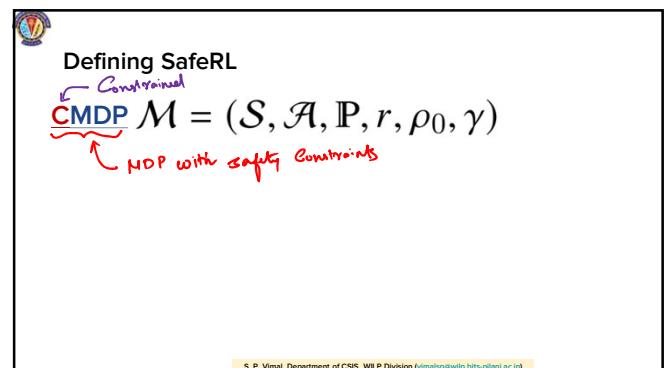
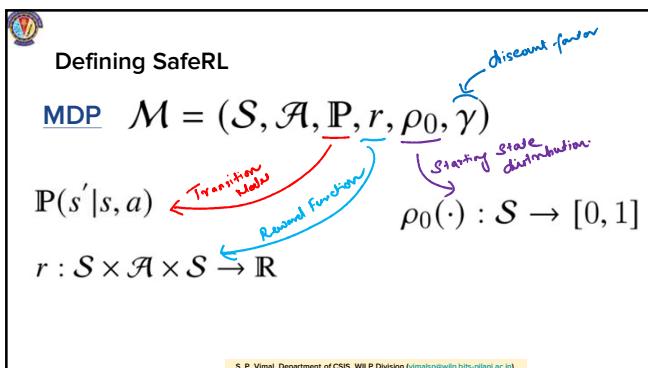
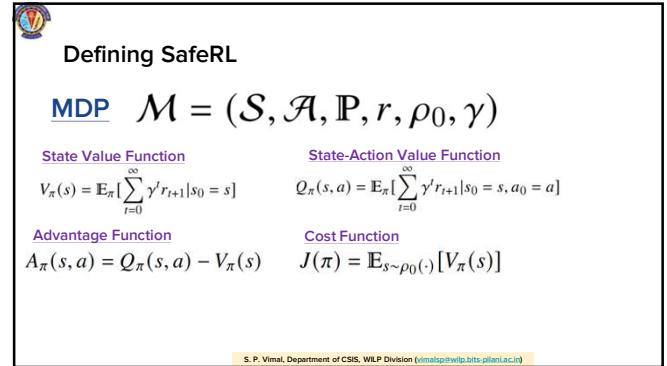
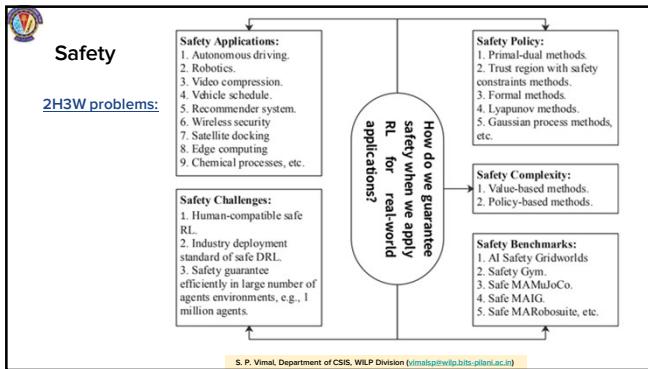
 **Safety**

**Some Definitions of Safety in the RL Sense:**

- (1) humans need to label environmental states as "safe" or "unsafe," and agents are considered "safe" if "they never reach unsafe states".
- (2) agents are safe if "they act, reason, and generalize obeying human desires"

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





**Defining SafeRL**

CMDP  $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathbb{P}, r, \rho_0, \gamma)$

$\nwarrow$  NDP with safety constraints

$\{(c_i, b_i)\}_{i=1}^m$

Cost Value function for constraint type  $i$

Safety Constraint Bound.

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

**Defining SafeRL**

CMDP

Now define  $V_\pi^{ci}$ ,  $Q_\pi^{ci}$ ,  $A_\pi^{ci}$  for each  $c_i$

$V_\pi^{ci}(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t c_i(s_t, a_t) | s_0 = s \right]$

$\{(c_i, b_i)\}_{i=1}^m$

Cost Value function for constraint type  $i$

Safety Constraint Bound.

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

**Defining SafeRL**

CMDP

Now define  $V_\pi^{ci}$ ,  $Q_\pi^{ci}$ ,  $A_\pi^{ci}$  for each  $c_i$

$\{(c_i, b_i)\}_{i=1}^m$

Cost Value function for constraint type  $i$

Safety Constraint Bound.

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

**Defining SafeRL**

CMDP

Now define  $V_\pi^{ci}$ ,  $Q_\pi^{ci}$ ,  $A_\pi^{ci}$  for each  $c_i$

$V_\pi^{ci}(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t c_i(s_t, a_t) | s_0 = s \right]$

$V_\pi(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r_{t+1} | s_0 = s \right]$

Compare!

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

**Defining SafeRL**

**CMDP**

Now define  $V_\pi^{c_i}$ ,  $Q_\pi^{c_i}$ ,  $A_\pi^{c_i}$  for each  $c_i$

$\{(c_i, b_i)\}_{i=1}^m$

$V_\pi^{c_i}(s) = \mathbb{E}_\pi [\sum_{t=0}^{\infty} \gamma^t c_i(s_t, a_t) | s_0 = s]$

$C_i(\pi) = \mathbb{E}_{s \sim \rho_0(\cdot)} [V_\pi^{c_i}(s)]$

Expected Cost function

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

**Defining SafeRL**

**CMDP**

Constraint Board

$V_\pi^{c_i}(s) = \mathbb{E}_\pi [\sum_{t=0}^{\infty} \gamma^t c_i(s_t, a_t) | s_0 = s]$

$\Pi_C = \cap_{i=1}^m \{\pi \in \Pi_S \text{ and } C_i(\pi) \leq b_i\}$

$C_i(\pi) = \mathbb{E}_{s \sim \rho_0(\cdot)} [V_\pi^{c_i}(s)]$

Feasible Policy Set

Expected Cost function

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

**Defining SafeRL**

**CMDP**

$V_\pi^{c_i}(s) = \mathbb{E}_\pi [\sum_{t=0}^{\infty} \gamma^t c_i(s_t, a_t) | s_0 = s]$

$C_i(\pi) = \mathbb{E}_{s \sim \rho_0(\cdot)} [V_\pi^{c_i}(s)]$

Expected Cost function

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

**Defining SafeRL**

**CMDP**

$\Pi_C = \cap_{i=1}^m \{\pi \in \Pi_S \text{ and } C_i(\pi) \leq b_i\}$

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

**Defining SafeRL**

**CMDP**

$$\max_{\pi \in \Pi_S} J(\pi), \text{ such that } c(\pi) \leq b$$

$\Pi_C = \cap_{i=1}^m \{\pi \in \Pi_S \text{ and } C_i(\pi) \leq b_i\}$

$$\pi^* = \arg \max_{\pi \in \Pi_C} J(\pi)$$

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

**SafeRL – Categorization**

**Model Based SafeRL**

**Model-free SafeRL**

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

**Defining SafeRL**

**CMDP**

$$\max_{\pi \in \Pi_S} J(\pi), \text{ such that } c(\pi) \leq b$$

$\Pi_C = \cap_{i=1}^m \{\pi \in \Pi_S \text{ and } C_i(\pi) \leq b_i\}$

$$\pi^* = \arg \max_{\pi \in \Pi_C} J(\pi)$$

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

**Applications | Autonomous Driving**

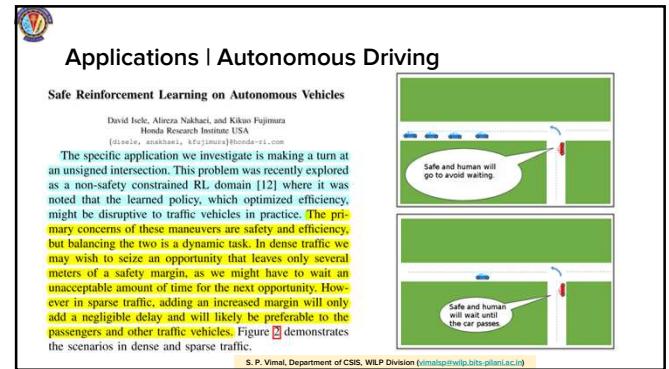
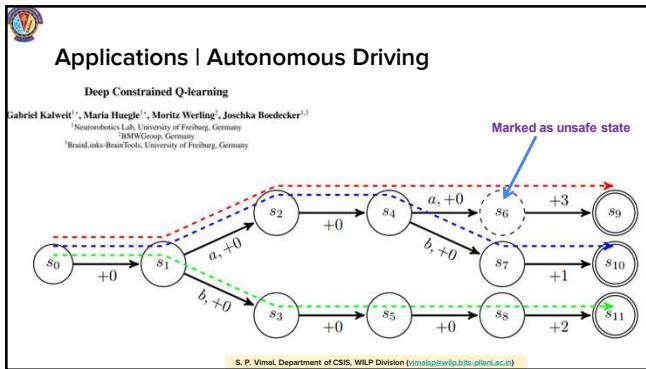
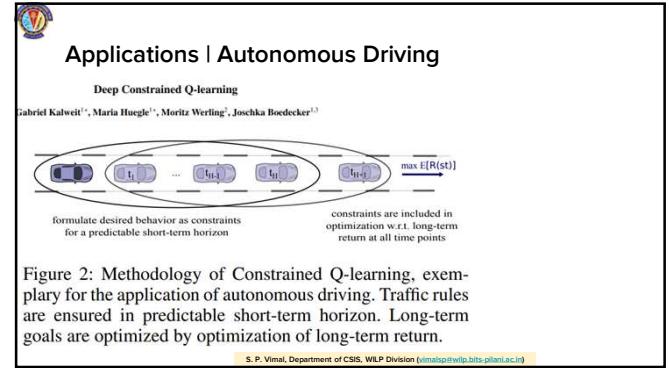
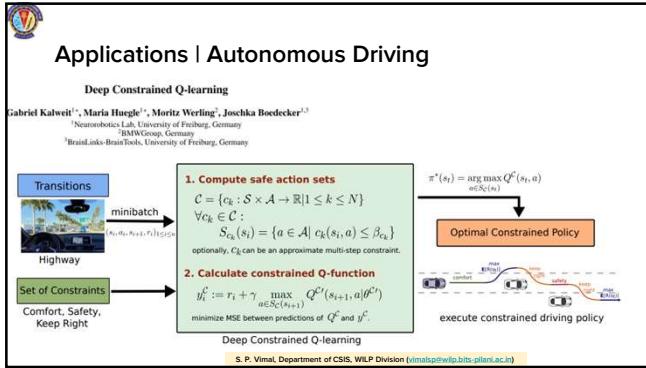
**Safe Reinforcement Learning on Autonomous Vehicles**

David Jeet, Alireza Nakhaei, and Kikuo Fujimura  
Honda Research Institute USA  
[\[djeete, anakhaei, kfujimura\]@honda-ri.com](mailto:[djeete, anakhaei, kfujimura]@honda-ri.com)

*Abstract*— There have been numerous advances in reinforcement learning, but the typical framework’s exploration of the learned policies can lead to adoption of these methods in many safety critical applications. Recent work in safe reinforcement learning uses idealized models to achieve their guarantees, but these models do not easily accommodate the stochasticity or high-dimensionality of real world systems. We investigate how prediction provides a general and intuitive framework to constrain exploration, and show how it can be used to safely learn intersection handling behaviors on an autonomous vehicle.



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

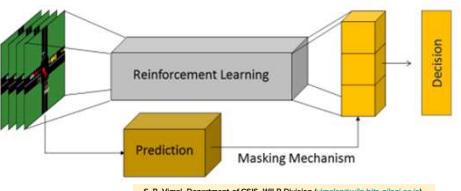


 Applications | Autonomous Driving

Safe Reinforcement Learning on Autonomous Vehicles

David Isele, Alireza Nakhaei, and Kikuo Fujimura  
Honda Research Institute USA  
[{disele, anakhaei, krfujimura}@honda-ri.com](mailto:{disele, anakhaei, krfujimura}@honda-ri.com)

<https://arxiv.org/pdf/1910.00399.pdf>



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

 BITS Pilani  
Hon / Dubai / Goa / Hyderabad

Thank you

31



Required Readings and references

1. Deep Constrained Q-learning Gabriel Kalweit, Maria Huegle, Moritz Werling, Joschka Boedecker
2. Safe Reinforcement Learning on Autonomous Vehicles David Isele, Alireza Nakhaei, and Kikuo Fujimura Honda Research Institute USA
3. A Review of Safe Reinforcement Learning: Methods, Theory and Applications, 2023, <https://arxiv.org/pdf/2205.10330.pdf>

30

# Sample Problems in Temporal Difference Learning

# Temporal Difference Learning (TD Learning)

- TD learning is an unsupervised technique to predict a variable's expected value in a sequence of states.
- TD uses a simple mathematical trick to replace a complex reasoning about the future with a simple learning procedure that can produce the same results.
- Instead of calculating the total future reward, TD tries to predict the combination of immediate reward and its own reward prediction at the next moment in time.

# Important Algorithms of TD Learning

## 1. SARSA (State – Action – Reward – State – Action)

- SARSA is an on-policy learning method, as it uses an  $\epsilon$ -greedy strategy for all the steps. It updates the Q-value for a certain action based on the obtained reward from taking that action and the reward from the state after that assuming it keeps following the policy. The formula that updates the Q-value is as follows:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

- In on-policy learning, the agent learns from the experiences it gathers while following its current policy. The policy being updated and the policy being followed are the same.

# Important Algorithms of TD Learning

## 2. Q - Learning

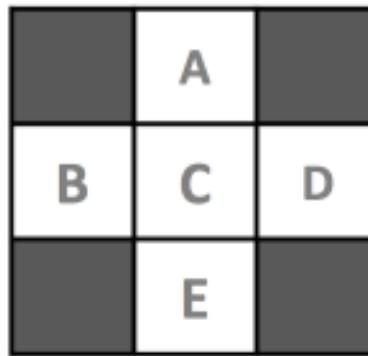
- Q-Learning is an off-policy learning method. It updates the Q-value for a certain action based on the obtained reward from the next state and the maximum reward from the possible states after that. It is off-policy because it uses an  $\epsilon$ -greedy strategy for the first step and a greedy action selection strategy for the second step.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

- In off-policy learning, the agent learns from experiences generated by following a different (or "off") policy from the one it's currently improving. The policy being updated may be different from the policy being followed.

# Sample Problem

- Consider the Gridworld. We would like to use TD learning to find the values of these states.



- Suppose we observe the following  $(s, a, s', R(s, a, s'))^*$  transitions and rewards: (B, East, C, 2), (C, South, E, 4), (C, East, A, 6), (B, East, C, 2). The initial value of each state is 0. Let  $\gamma = 1$  and  $\alpha = 0.5$ .
- \*Note that the  $R(s, a, s_0)$  in this notation refers to observed reward, not a reward value computed from a reward function.

- What are the learned values for each state from TD learning after all four observations?

**Solution using TD-Learning :**

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

For (B, East, C, 2), we update

$$\begin{aligned} V \pi (B): V \pi (B) &\leftarrow V \pi (B) + \alpha(R(s, a, s') + \gamma V \pi (C) - V \pi (B)) \\ &= 0 + 0.5(2 + 1 * 0 - 0) = 1. \end{aligned}$$

For (C, South, E, 4), we update

$$\begin{aligned} V \pi (C): V \pi (C) &\leftarrow V \pi (C) + \alpha(R(s, a, s') + \gamma V \pi (E) - V \pi (C)) \\ &= 0 + 0.5(4 + 1 * 0 - 0) = 2. \end{aligned}$$

For (C, East, A, 6), we update

$$\begin{aligned}V\pi(C): V\pi(C) &\leftarrow V\pi(C) + \alpha(R(s, a, s') + \gamma V\pi(A) - V\pi(C)) \\&= 2 + 0.5(6 + 1 * 0 - 2) = 4.\end{aligned}$$

For (B, East, C, 2), we update

$$\begin{aligned}V\pi(B): V\pi(B) &\leftarrow V\pi(B) + \alpha(R(s, a, s') + \gamma V\pi(C) - V\pi(B)) \\&= 1 + 0.5(2 + 1 * 4 - 1) = 3.5.\end{aligned}$$

The final values are,

$$V(B) = 3.5 \text{ and } V(C) = 4$$

Learned values for other states are 0, i.e.,  $V(A) = V(D) = V(E) = 0$ .

Solution using SARSA :

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

Update for transition (B, East, C, 2)

$$Q(\text{B, East}) \leftarrow Q(\text{B, East}) + 0.5 (2 + 1 * 0 - 0) = 1$$



Update for transition (C, South, E, 4)  $\begin{matrix} (\text{C, East}) & (\text{B, East}) \end{matrix}$

$$Q(\text{C, South}) \leftarrow Q(\text{C, South}) + 0.5 (4 + 1 * 0 - 0) = 2$$



$\begin{matrix} (\text{E, South}) & (\text{C, South}) \end{matrix}$

Update for transition (C, East, A, 6)

$$Q(C, \text{East}) \leftarrow Q(C, \text{East}) + 0.5 (6 + 1 * 0 - 0) = 3$$

Update for transition (C, South, E, 4)

$$\begin{aligned} Q(B, \text{East}) &\leftarrow Q(B, \text{East}) + 0.5 (2 + 1 * 3 - 1) \\ &= 1 + 2 = 3 \end{aligned}$$

The final values are,

$$Q(C, \text{South}) = 2, Q(C, \text{East}) = 3 \text{ and } Q(B, \text{East}) = 3$$

- What are the learned Q - values from Q-Learning after all four observations? Use the same  $\gamma = 1$  and  $\alpha = 0.5$ .

The update rule for Q-Learning is,

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

Update for transition (B, East, C, 2)

$$Q(\text{B, East}) \leftarrow Q(\text{B, East}) + 0.5 (2 + 1 * \max(0,0,0,0) - 0) = 1$$

Update for transition (C, South, E, 4)

$$Q(\text{C, South}) \leftarrow Q(\text{C, South}) + 0.5 (4 + 1 * \max(0,0,0,0) - 0) = 2$$

Update for transition (C, East, A, 6)

$$Q(C, \text{East}) \leftarrow Q(C, \text{East}) + 0.5 (6 + 1 * \max(0,0,0,0) - 0) = 3$$

Update for transition (C, South, E, 4)

$$Q(B, \text{East}) \leftarrow Q(B, \text{East}) + 0.5 (2 + 1 * \max(3,0,0,0) - 1) = 3$$

The final learned Q-Values for each state-action pair after all four observations are

$$Q(C, \text{South}) = 2$$

$$Q(C, \text{East}) = 3$$

$$Q(B, \text{East}) = 3$$

- In class you would have seen two formulations for TD Learning

$$V \pi(s) \leftarrow (1 - \alpha)V \pi(s) + (\alpha)\text{sample} \quad (1)$$

$$V \pi(s) \leftarrow V \pi(s) + \alpha(\text{sample} - V \pi(s)) \quad (2)$$

Mathematically, these 2 equations are equivalent. However, they represent two conceptually different ways of understanding TD value updates. How could we intuitively explain each of these equations?

Equation (1) : This equation represents the update rule for updating the value function  $V \pi(s)$  using a weighted average between the current estimate  $V \pi(s)$  and a sampled value (often a target value or a reward).

- The parameter  $\alpha$  controls the weight given to the sampled value relative to the current estimate.
- $(1 - \alpha)V\pi(s)$  : It represents the weight given to the current estimate  $V\pi(s)$ . It indicates how much we trust our current estimate. A higher  $\alpha$  means less trust in the current estimate and more weight is given to the sampled value.
- $(\alpha)\text{sample}$ : It represents the weighted sampled value. It indicates the amount by which we update our current estimate based on the sampled value. A higher  $\alpha$  value means more influence of the sampled value on the update.

$$\text{Equation (2)} : V \pi(s) \leftarrow V \pi(s) + \alpha(\text{sample} - V \pi(s))$$

This equation represents the update rule for updating the value function  $V \pi(s)$  by directly adjusting the current estimate towards the sampled value. The parameter  $\alpha$  controls the step size of the adjustment.

$\alpha(\text{sample} - V \pi(s))$ : This term represents the adjustment to the current estimate based on the difference between the sampled value and the current estimate. A higher  $\alpha$  means a larger adjustment to the current estimate.

$V \pi(s)$ : This term represents the current estimate of the value function. The adjustment is applied directly to this estimate, shifting it closer to the sampled value.

- In conclusion,

The first equation takes a weighted average between our current values and our new sample.

The second equation updates our current values towards the new sample value, scaled by a factor of our learning rate  $\alpha$ .