Deep Reinforcement Learning

BITS Pilani

Prof. Vimal SP
CSIS

BITS Pilani

<AIMLCZG512, Deep Reinforcement Learning>
Lecture No. 1,2,3

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

Paradigm	Supervised Learning	Unsupervised Learning	Reinforcement Learning
Objective	$p_\theta(y\|x)$	$p_\theta(x)$	$\pi_\theta(a\|s)$
Applications	→ Classification → Regression	→ Inference → Generation	→ Prediction → Control

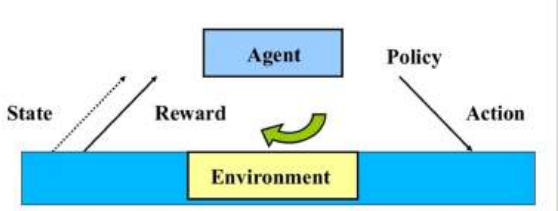
Types of Learning

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

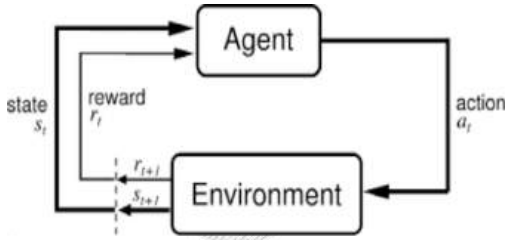
Characteristics of RL

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

Elements of Reinforcement Learning



Elements of Reinforcement Learning



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

Elements of Reinforcement Learning

•**Agent**

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

•**Action (A)** -

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

Elements of Reinforcement Learning

•**State (S)**

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

•**Rewards (R)**

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

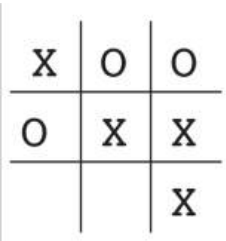
Elements of Reinforcement Learning

•**Environment**

- Outside world of an agent or physical world in which the agent operates.

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**.*

Tic-Tac-Toc



Tic-Tac-Toc

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

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

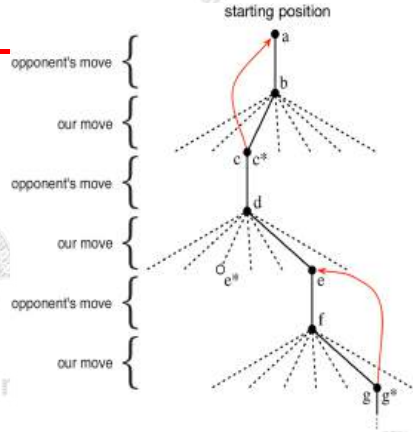
X	O	O
O	X	X
		X

Set up a table of states initial values

Tic-Tac-Toc

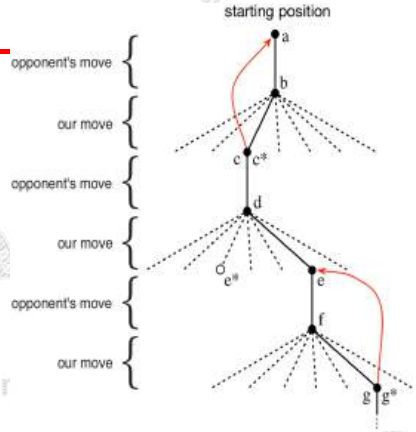
States	Initial Values
$\begin{bmatrix} X & & \\ & & \\ & & \end{bmatrix}$	0.5
$\begin{bmatrix} X & O & O \\ & X & \\ & & \end{bmatrix}$	0.5
$\begin{bmatrix} X & O & O \\ & X & \\ & & X \end{bmatrix}$	1.0
$\begin{bmatrix} X & & O \\ X & & O \\ & X & O \end{bmatrix}$	0
...	...

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



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

Tic-Tac-Toc



Temporal Difference Learning Rule

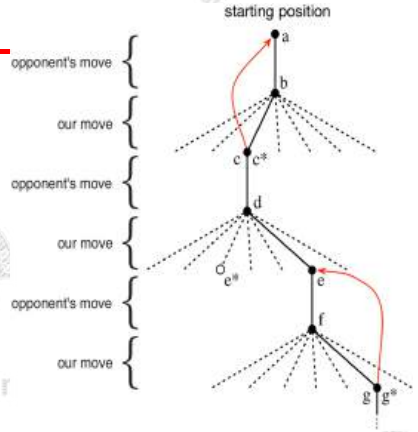
$$V(S_t) \leftarrow V(S_t) + \alpha\left[V(S_{t+1}) - V(S_t)\right]$$

α- Step Size Parameter

Tic-Tac-Toc

Questions:

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



Temporal Difference Learning Rule

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

α- Step Size Parameter

Tic-Tac-Toc

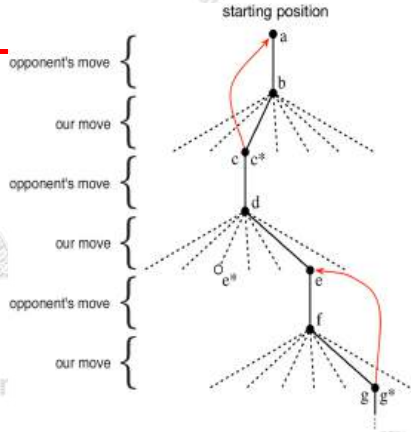
Key Takeaways:

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

(2) We have a clear goal

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

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



Temporal Difference Learning Rule

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

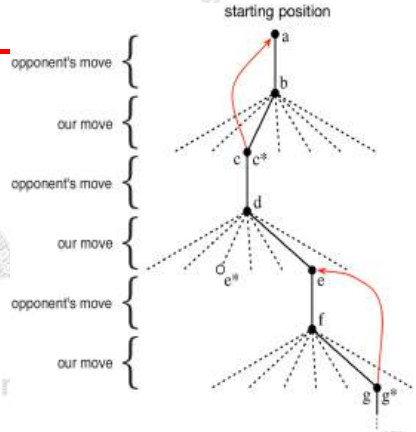
α- Step Size Parameter

Tic-Tac-Toc

Reading Assigned:

Identify how this reinforcement learning solution is different from solutions using minimax algorithm and genetic algorithms.

Post your answers in the discussion forum;



Temporal Difference Learning Rule

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

α- Step Size Parameter

K-armed Bandit Problem

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?

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

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*

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

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

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

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 :-)

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

K-armed Bandit Problem

$\mathbb{E}[a] = -\$0.5$ $\mathbb{E}[b] = -\$0.2$ $\mathbb{E}[c] = \$0.1$ $\mathbb{E}[d] = \$0.11$

K-armed Bandit Problem

-1, -1, 5 $\widehat{\mathbb{E}}[a] = 1$ -0.2, -0.2 $\widehat{\mathbb{E}}[b] = -0.2$ -0.5, -0.5, -0.5 $\widehat{\mathbb{E}}[c] = -0.5$ -2, -2 $\widehat{\mathbb{E}}[d] = -2$

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

K-armed Bandit Problem

-1, -1, 5 $\widehat{\mathbb{E}}[a] = 1$ -0.2, -0.2 $\widehat{\mathbb{E}}[b] = -0.2$ -0.5, -0.5, -0.5 $\widehat{\mathbb{E}}[c] = -0.5$ -2, -2 $\widehat{\mathbb{E}}[d] = -2$

K-armed Bandit Problem

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

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

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

Greedy Action Selection $\quad A_t \doteq \underset{a}{\operatorname{argmax}}\, Q_t(a)$

K-armed Bandit Problem

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

   Greedy Action Selection

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

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

3. Exploration vs. Exploitation?

   ε-Greedy Action Selection / near-greedy action selection

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

K-armed Bandit Problem

{-1, -1, 5}	{-0.2, -0.2}	{-0.5, -0.5, -0.5}	{-2, -2}
$N_{11}(a)=3$	$N_{11}(b)=2$	$N_{11}(c)=3$	$N_{11}(d)=2$
$q_{\cdot}(a)=-\$0.5$	$q_{\cdot}(b)=-\$0.2$	$q_{\cdot}(c)=\$0.1$	q-(d)=$1
$Q_{11}(a)=1$	$Q_{11}(b)=-0.2$	$Q_{11}(c)=-0.5$	$Q_{11}(d)=-2$

K-armed Bandit Problem

Greedy Action

{-1, -1, 5}	{-0.2, -0.2}	{-0.5, -0.5, -0.5}	{-2, -2}
$N_{11}(a)=3$	$N_{11}(b)=2$	$N_{11}(c)=3$	$N_{11}(d)=2$
$q_{\cdot}(a)=-\$0.5$	$q_{\cdot}(b)=-\$0.2$	$q_{\cdot}(c)=\$0.1$	q-(d)=$1
$Q_{11}(a)=1$	$Q_{11}(b)=-0.2$	$Q_{11}(c)=-0.5$	$Q_{11}(d)=-2$

K-armed Bandit Problem

Action to Explore

{-1, -1, 5}	{-0.2, -0.2}	{-0.5, -0.5, -0.5}	{-2, -2}
$N_{11}(a)=3$	$N_{11}(b)=2$	$N_{11}(c)=3$	$N_{11}(d)=2$
$q_{\cdot}(a)=-\$0.5$	$q_{\cdot}(b)=-\$0.2$	$q_{\cdot}(c)=\$0.1$	q-(d)=$1
$Q_{11}(a)=1$	$Q_{11}(b)=-0.2$	$Q_{11}(c)=-0.5$	$Q_{11}(d)=-2$

K-armed Bandit Problem

ε-Greedy Action Selection / near-greedy action selection

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

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

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

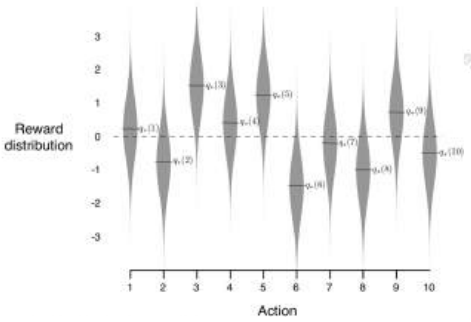
Ex-1: In ε-greedy action selection, for the case of two actions and ε = 0.5, what is the probability that *the greedy action* is selected?

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

10-armed Testbed

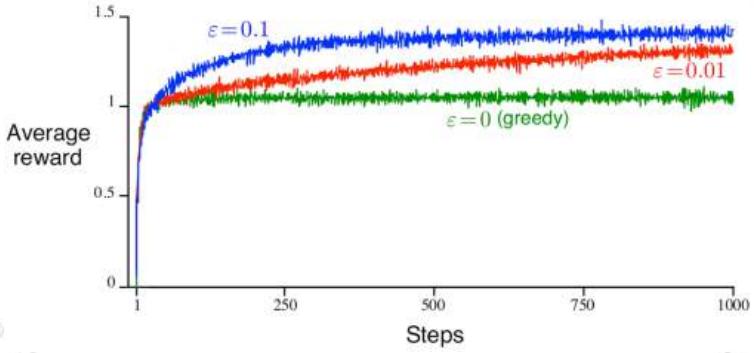
Example:

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

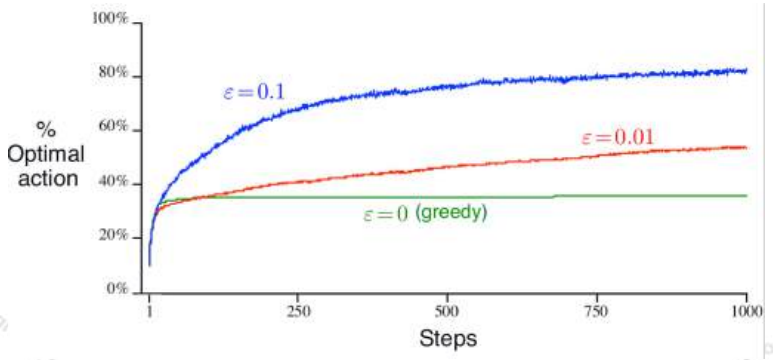


An example bandit problem from the 10-armed testbed

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



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



Discussion on Exploration vs. Exploitation

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

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

Ex-2:

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

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

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

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

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

Incremental Implementation

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

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

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

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

Incremental Implementation

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

Note:

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

Discussion:

Const vs. Variable step size?

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

Bandit Algorithm with Incremental Update/ ε-greedy selection

Initialize, for $a = 1$ to k:
  $Q(a) \leftarrow 0$
  $N(a) \leftarrow 0$

Loop forever:
  $A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$
  $R \leftarrow bandit(A)$
  $N(A) \leftarrow N(A) + 1$
  $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\big[R - Q(A)\big]$

What are problems whose solutions are modelled as MAB?

Non-stationary Problem

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

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

Non-stationary Problem

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

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

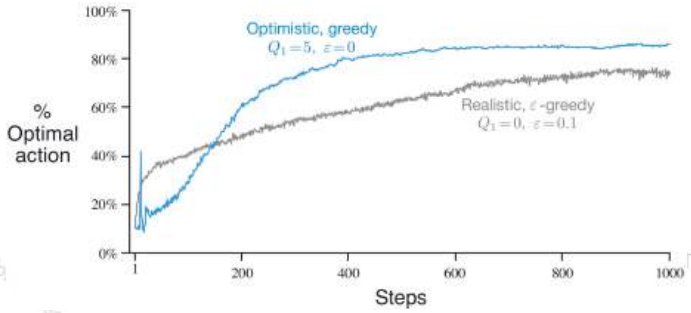
$$\begin{aligned}
Q_{n+1} &= Q_n + \alpha\left[R_n - Q_n\right] \\
&= \alpha R_n + (1-\alpha)Q_n \\
&= \alpha R_n + (1-\alpha)\left[\alpha R_{n-1} + (1-\alpha)Q_{n-1}\right] \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 \alpha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

Exponential recency-weighted average

Optimistic Initial Values

- All the above discussed methods are ***biased*** by their initial estimates
- For sample average method the bias disappears once all actions have been selected at least once
- For methods with constant α, the bias is permanent, though decreasing over time
- Initial action values can also be used as a simple way of encouraging exploration.
- In 10 armed testbed, set initial estimate to +5 rather than 0.
  - This can encourage action-value methods to explore.
    - Whichever actions are initially selected, the reward is less than the starting estimates;
    - the learner switches to other actions, being disappointed with the rewards it is receiving.
    - The result is that all actions are tried several times before the value estimates converge.

Optimistic Initial Values



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

- **ε**-greedy action selection forces the non-greedy actions to be tried,
  - Indiscriminately, with no preference for those that are nearly greedy or particularly uncertain
- It would be better to select among the non-greedy actions according to their potential for actually being optimal
  - Take ir being r

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

es are to stimates.

Upper-Confidence-Bound Action Selection

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

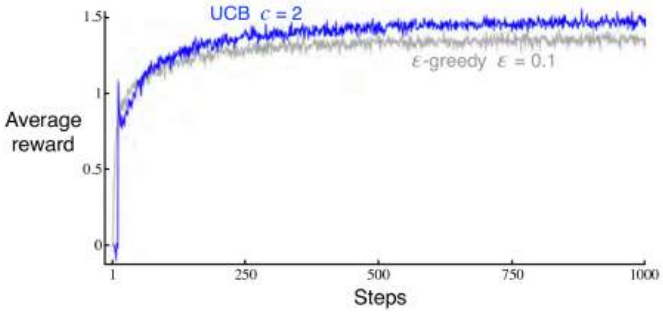
Action Value at time t for a

Confidence Level

Measure of Uncertainty

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

Upper-Confidence-Bound Action Selection



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

Policy-based algorithms

- Forget about action-value (Q) estimates, we don't really care about them
- We care about what actions to chose
  - Le assign a preference to each action and tweak its value
- Define $H_t(a)$ as a numerical preference value associated with action a
- Which action is selected?
  - $A_t = \underset{a}{\text{argmax}}[H_t(a)]$
    - Hardmax results in no exploration -- deterministic action selection
  - S ax!

Softmax function

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

Softmax function

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

Gradient ascend

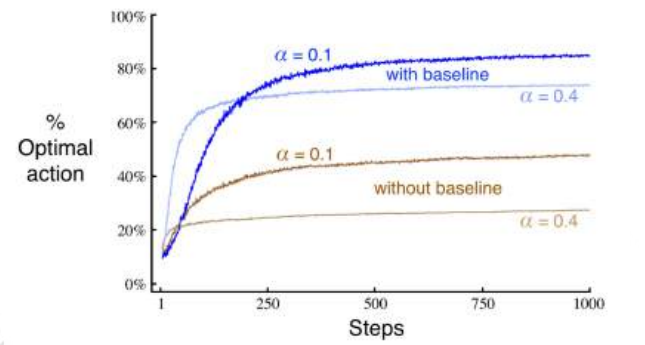
- $H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \overline{R_t})\frac{\partial \Pr(A_t)}{\partial H(A_t)}$?
- $\forall a \neq A_t, H_{t+1}(a) = H_t(a) + \alpha(R_t - \overline{R_t})\frac{\partial \Pr(A_t)}{\partial H(a)}$
- Update the preferences based on observed reward and a baseline reward ($\overline{R_t}$)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

Update Rule

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

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

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

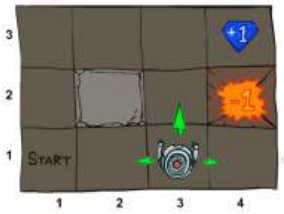


What did we learn?

- **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 !!!



Deep Reinforcement Learning

BITS Pilani

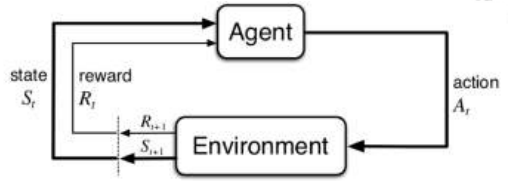
Prof. Vimal SP
CSIS

BITS Pilani

<AIMLCZG512, Deep Reinforcement Learning>
Lecture No. 3,4,5

Agent-Environment Interface

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

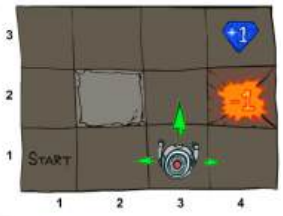


Note:
- Interaction occurs in discrete time steps

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

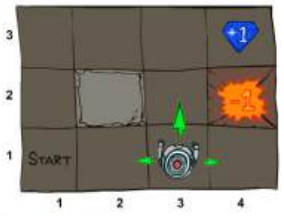
Grid World Example

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



Markov Decision Processes

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



Markov Decision Processes

Model Dynamics

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

State-transition probabilities

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

Expected rewards for state–action–next-state triples

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

Markov Decision Processes - Discussion

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

Markov Decision Processes - Discussion

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

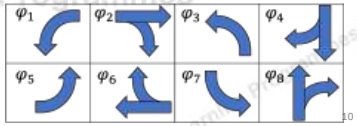
MDP Formalization : Video Games

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

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

MDP Formalization : Traffic Signal Control

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



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

MDP Formalization : Recycling Robot (Detailed Ex.)

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

MDP Formalization : Recycling Robot (Detailed Ex.)

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

MDP Formalization : Recycling Robot (Detailed Ex.)

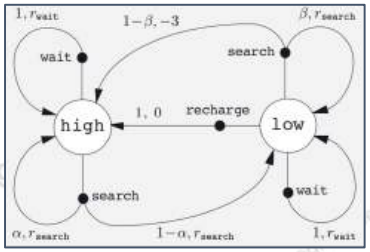
- *State-transition probabilities (contd…):*

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

MDP Formalization : Recycling Robot (Detailed Ex.)

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

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



Note on Goals & Rewards

- *Reward Hypothesis:*

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

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

Returns & Episodes

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

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

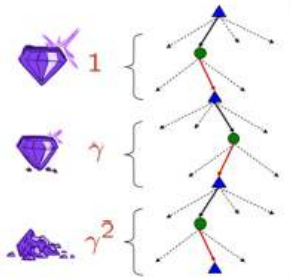
  - *Applicable when agent-environment interaction breaks into episodes*
  - *Ex: Playing Game, Trips through maze etc. [called episodic tasks]*

Returns & Episodes

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

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

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



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

Returns & Episodes

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

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

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

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

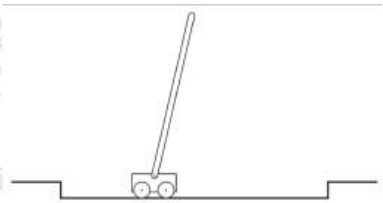
Returns & Episodes

➔ *Objective: To apply forces to a cart moving along a track so as to keep a pole hinged to the cart from falling over*

➔ *Discuss:*

➔ *Consider the task as episodic, that is try/maintain balance until failure. What could be the reward function?*

➔ *Repeat prev. assuming task is continuous.*



Policy

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

Defining Value Functions

State-value function for policy π

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

Action-value function for policy π

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

Defining Value Functions

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

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

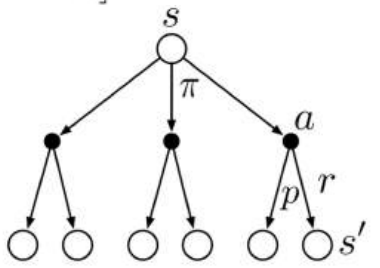
Action Value function in terms of State value function for policy π

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

Bellman Equation for V_π

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

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

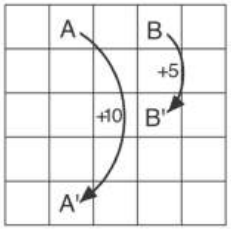


Backup Diagram

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

Actions

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

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

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

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

Understanding $V_\pi(s)$ with Gridworld

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

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

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

Ex-1

Recollect the reward function used for Gridworld as below:

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

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

Optimal Policies and Optimal Value Functions

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

Bellman optimality equation for V_*

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

Optimal Policies and Optimal Value Functions

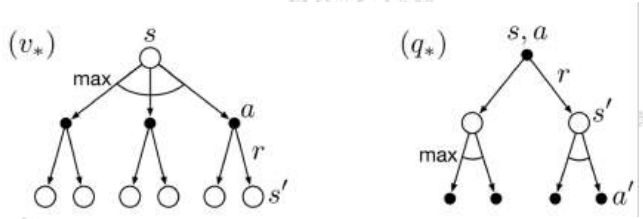
Bellman optimality equation - expresses that *the value of a state under an optimal policy must equal the expected return for the best action from that state*

Bellman optimality equation for q_*

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

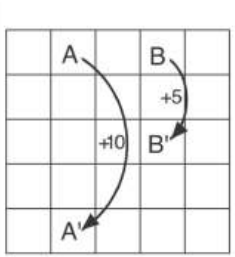
Optimal Policies and Optimal Value Functions

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*

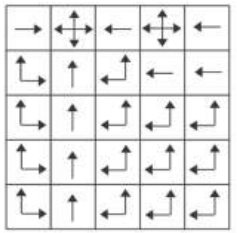
Optimal solutions to the gridworld example



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_*



π_*

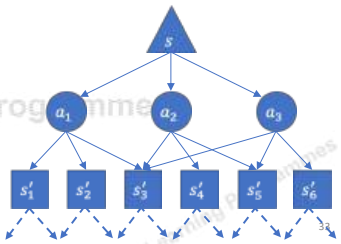
Backup diagrams for v* and q*

MDP - Objective

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

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

$v^*(s')$



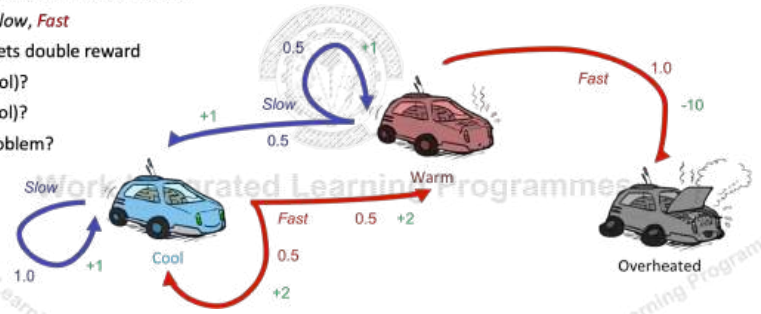
Notation

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

Race car example

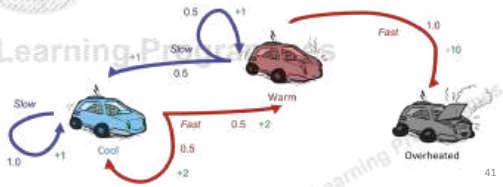
$$\max_a \left[\sum_{s'} P(s'|s,a)\,(R(s,a,s') + v^*(s'))\right]$$

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



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

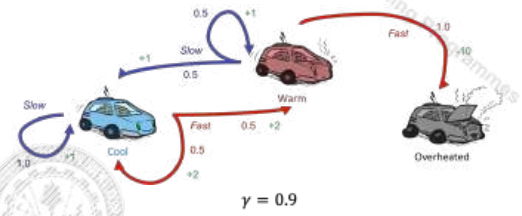
Value iteration

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

Value Iteration



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

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

Check this computation on paper.

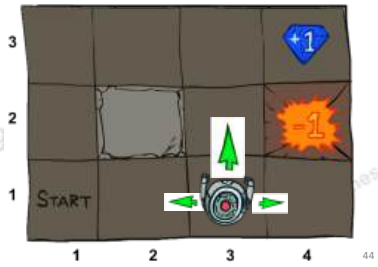
Example: Grid World

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

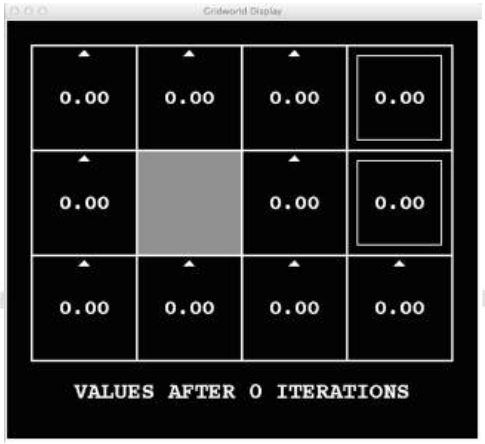
The agent receives rewards each time step
- Small negative reward each step (battery drain)
- Big rewards come at the end (good or bad)

Goal: maximize sum of (discounted) rewards



k=0

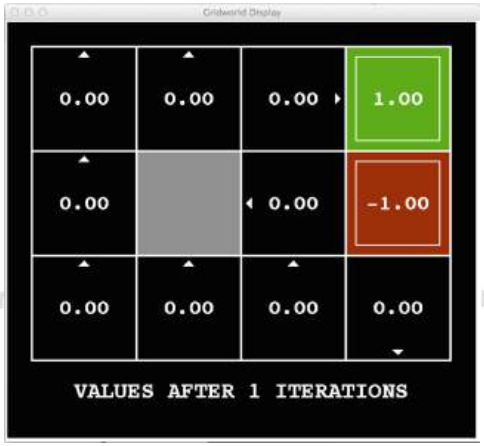
$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right]$$



Noise = 0.2
Discount = 0.9
Living reward = 0

k=1

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



Noise = 0.2
Discount = 0.9
Living reward = 0

k=2

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

Noise = 0.2
Discount = 0.9
Living reward = 0

k=3

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

Noise = 0.2
Discount = 0.9
Living reward = 0

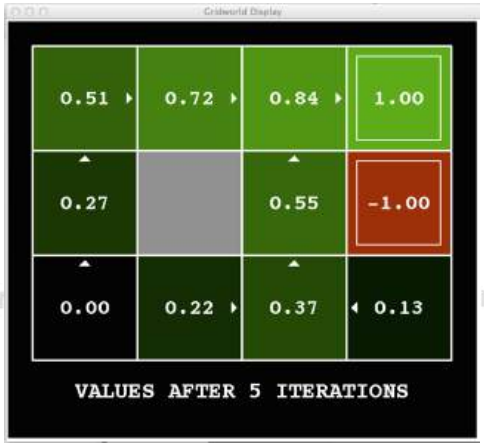
k=4

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

Noise = 0.2
Discount = 0.9
Living reward = 0

k=5

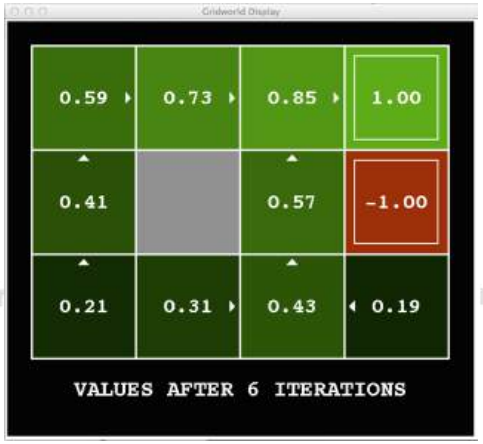
$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$



Noise = 0.2
Discount = 0.9
Living reward = 0

k=6

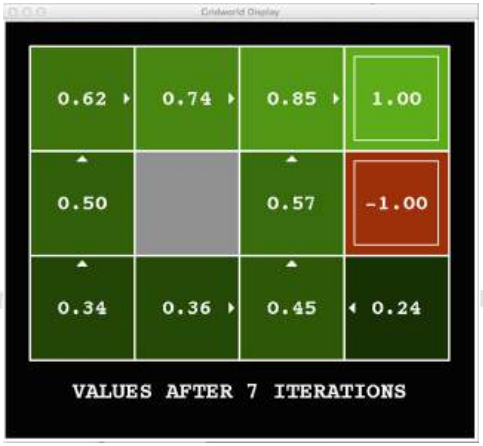
$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$



Noise = 0.2
Discount = 0.9
Living reward = 0

k=7

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



Noise = 0.2
Discount = 0.9
Living reward = 0

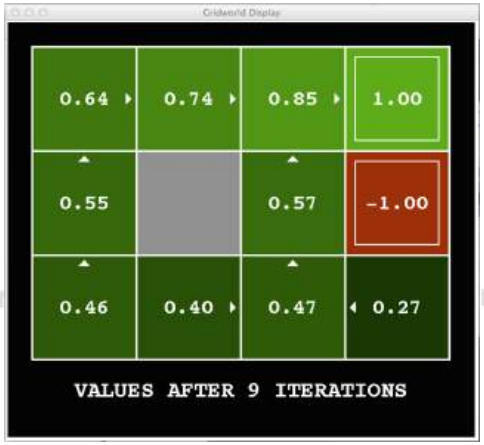
k=8

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

Noise = 0.2
Discount = 0.9
Living reward = 0

k=9

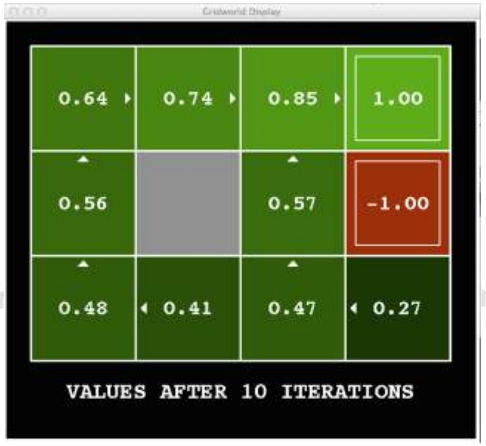
$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$



Noise = 0.2
Discount = 0.9
Living reward = 0

k=10

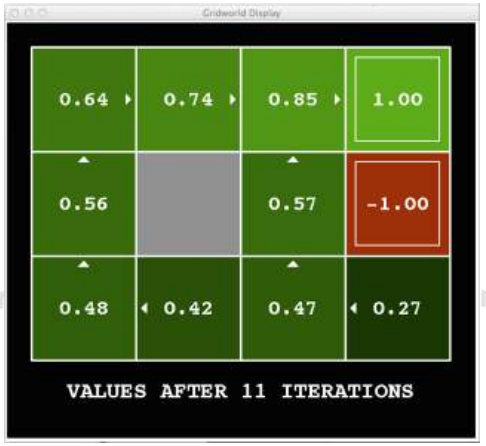
$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$



Noise = 0.2
Discount = 0.9
Living reward = 0

k=11

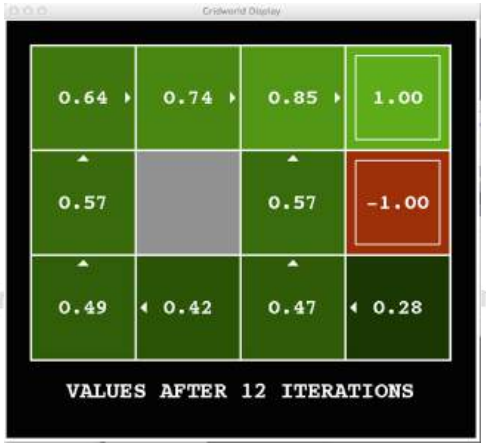
$$v_{k+1}(s) \doteq \max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma v_k(s')\right]$$



Noise = 0.2
Discount = 0.9
Living reward = 0

k=12

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



Noise = 0.2
Discount = 0.9
Living reward = 0

k=100

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

Noise = 0.2
Discount = 0.9
Living reward = 0

Problems with Value Iteration

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

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

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

Solutions (briefly, more later…)

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

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

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

Asynchronous Value Iteration

states should be prioritized for an update?

A single update per iteration

Algorithm 3 Prioritized Value Iteration

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

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

For the home assignment set:

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

Double the work?

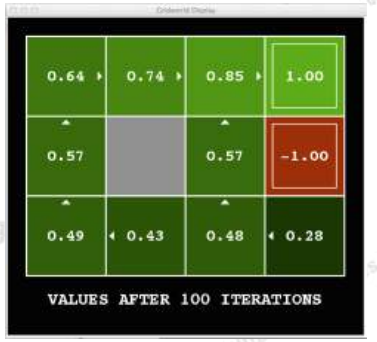
For the home assignment set:

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

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

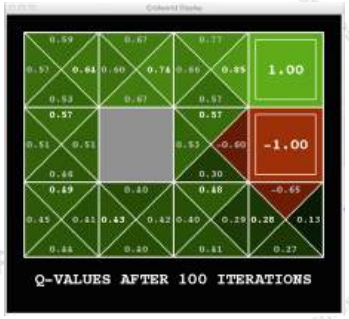
Issue 2: A policy cannot be easily extracted

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



Q-learning

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



Q-learning as value iteration

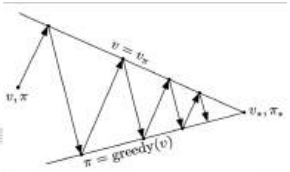
- $V^*(s) \coloneqq \max_a[\sum_{s'} P(s'|s, a)\ (R(s, a, s') + \gamma V^*(s'))]$
- $V^*(s) \coloneqq \max_a[Q^*(s, a)]$
- $Q^*(s, a) \coloneqq \sum_{s'} P(s'|s, a)\ (R(s, a, s') + \gamma V^*(s'))$
- $Q^*(s, a) = \sum_{s'} P(s'|s, a)\ (R(s, a, s') + \gamma \max_a[Q^*(s', a)])$
- Solve iteratively
  -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

Issue 3: The policy often converges long before the values

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

Policy Iteration

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



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

Policy Evaluation

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

Policy value as a Linear program

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

Policy iteration

Comparison

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

Issue 4: requires knowing the model and the reward function

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

Offline optimization

Online Learning

Issue 5: requires discrete (finite) set of actions

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

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

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

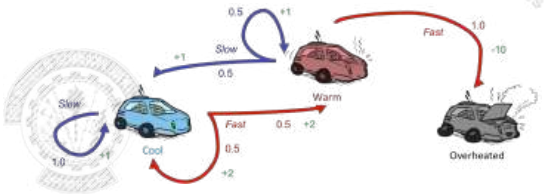
Notation

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

<AIMLCZG512, Deep Reinforcement Learning>
Lecture No. 6,7,8

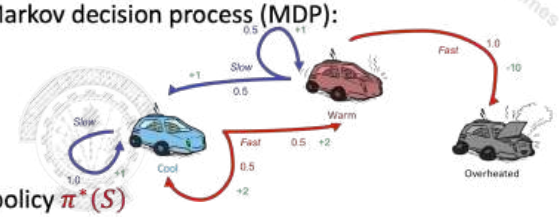
Introduction

- 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 γ
  - Still looking for the best policy $\pi^*(S)$

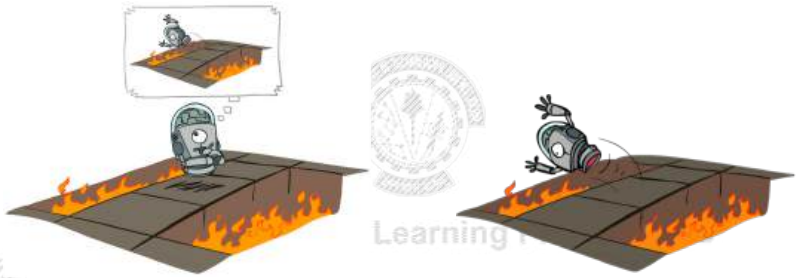


Introduction

- 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 γ
  - Still looking for the best policy $\pi^*(S)$

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

(Aside) Offline vs. Online (RL)



Offline Optimization

Online Learning

Monte Carlo Methods

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

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

Initialize:

$\pi \leftarrow$ policy to be evaluated

$V \leftarrow$ an arbitrary state-value function

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

Repeat forever:

(a) Generate an episode using π

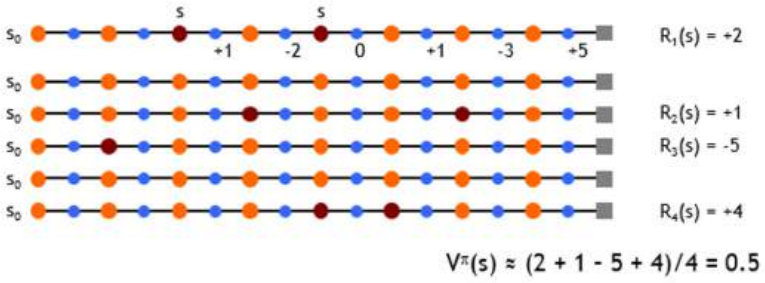
(b) For each state s appearing in the episode:

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

Append R to $Returns(s)$

$V(s) \leftarrow$ average($Returns(s)$)

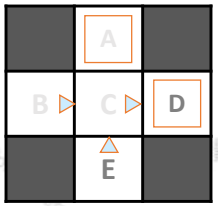
Ex-1:First-visit Monte-Carlo Policy Evaluation [estimate Vπ(s)]



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

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

Input Policy π



Assume: γ = 1

Observed Episodes (Training)

Episode 1

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

Episode 2

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

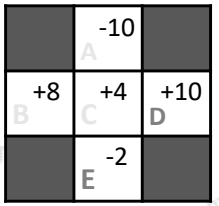
Episode 3

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

Episode 4

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

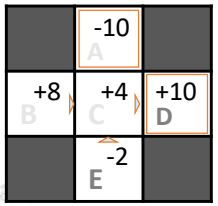
Output Values



Problems with MC Evaluation

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

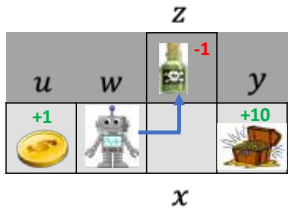
Output Values



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

What about exploration?

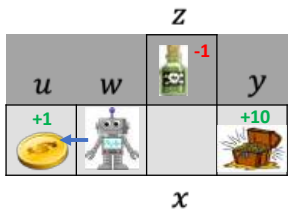
- $S = \{u, w, x, y, z\}$
- $A = \{N, E, S, W, exit\}$
- Reward:
  - $r(u, exit) = 1$
  - $r(z, exit) = -1$
  - $r(y, exit) = 10$



State	$\pi(s)$	$Q_\pi(N)$	$Q_\pi(E)$	$Q_\pi(S)$	$Q_\pi(W)$	$Q_\pi(exit)$
u	$exit$	NA	NA	NA	NA	0
w	E	0	~~-1~~	0	0	NA
x	N	~~-1~~	0	0	0	NA
y	$exit$	NA	NA	NA	NA	0
z	$exit$	NA	NA	NA	NA	~~-1~~

What about exploration?

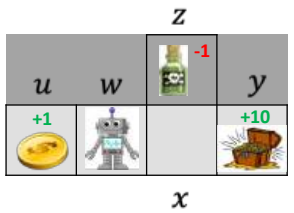
- $S = \{u, w, x, y, z\}$
- $A = \{N, E, S, W, exit\}$
- Reward:
  - $r(u, exit) = 1$
  - $r(z, exit) = -1$
  - $r(y, exit) = 10$



State	$\pi(s)$	$Q_\pi(N)$	$Q_\pi(E)$	$Q_\pi(S)$	$Q_\pi(W)$	$Q_\pi(exit)$
u	$exit$	NA	NA	NA	NA	~~1~~
w	~~N~~ W	0	-1	0	~~-1~~	NA
x	~~N~~ E	-1	0	0	0	NA
y	$exit$	NA	NA	NA	NA	0
z	$exit$	NA	NA	NA	NA	-1

What about exploration?

- $S = \{u, w, x, y, z\}$
- $A = \{N, E, S, W, exit\}$
- Reward:
  - $r(u, exit) = 1$
  - $r(z, exit) = -1$
  - $r(y, exit) = 10$



State	$\pi(s)$	$Q_\pi(N)$	$Q_\pi(E)$	$Q_\pi(S)$	$Q_\pi(W)$	$Q_\pi(exit)$
u	$exit$	NA	NA	NA	NA	1
w	W	0			1	NA
x	E	-			0	NA
y	$exit$	N			NA	0
z	$exit$	NA	NA	NA	NA	-1

We converged on a local optimum!

ε-greedy MC control

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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

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

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

MC control - example

- $Q =$

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

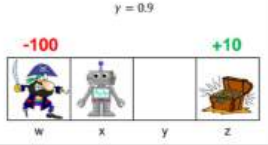
- $\overline{Returns} =$

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

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

exit	←	←	exit
w	x	y	z

$\gamma = 0.9$



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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

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

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

MC control - example

- $Q =$

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

- $\overline{Returns} =$

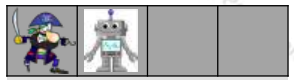
-	-,-	-,-	-
w	x	y	z

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

exit	→	←	exit
w	x	y	z

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

$\gamma = 0.9$



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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

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

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

MC control - example

- $Q =$

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

- $\overline{Returns} =$

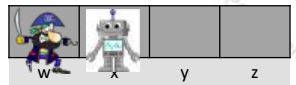
-100	-90,0	-,-	-
w	x	y	z

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

exit	→	←	exit
w	x	y	z

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

$\gamma = 0.9$



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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

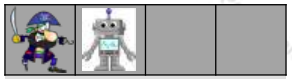
For all $a \in \mathcal{A}(S_t)$:

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

MC control - example

$\gamma = 0.9$

-100 +10



w x y z

- $Q =$

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

- $\overline{Returns} =$

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

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

exit	→	←	exit
w	x	y	z

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

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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken arbitrarily)

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

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

MC control - example

$\gamma = 0.9$

-100 +10

w x y z

- $Q =$

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

- $\overline{Returns} =$

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

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

exit	→	←	exit
w	x	y	z

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

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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken arbitrarily)

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

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

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

MC control - example

$\gamma = 0.9$

-100 +10

w x y z

- $Q =$

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

- $\overline{Returns} =$

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

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

exit	→	←	exit
w	x	y	z

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

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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken arbitrarily)

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

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

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

MC control - example

$\gamma = 0.9$

-100 +10

w x y z

- $Q =$

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

- $\overline{Returns} =$

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

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

exit	→	←	exit
w	x	y	z

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

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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken arbitrarily)

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

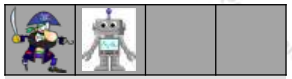
$$\pi(a|S_t) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

MC control - example

$\gamma = 0.9$

-100 +10



w x y z

- $Q =$

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

- $\overline{Returns} =$

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

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

exit	→	←	exit
w	x	y	z

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

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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken arbitrarily)

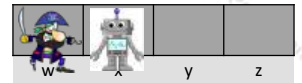
For all $a \in \mathcal{A}(S_t)$:

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

MC control - example

$\gamma = 0.9$

-100 +10



w x y z

- $Q =$

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

- $\overline{Returns} =$

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

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

exit	→	←	exit
w	x	y	z

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

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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken arbitrarily)

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

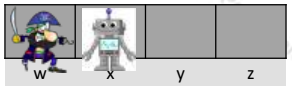
$$\pi(a|S_t) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

MC control - example

$\gamma = 0.9$

-100 +10



w x y z

- $Q =$

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

- $\overline{Returns} =$

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

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

exit	→	→	exit
w	x	y	z

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

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

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$A^* \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken arbitrarily)

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

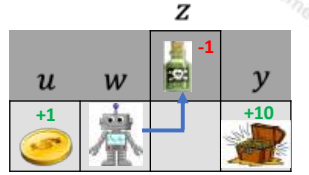
$$\pi(a|S_t) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

On-policy learning

Estimation True value

- $Q_\pi(w, \rightarrow) = q_\pi(w, \rightarrow) = -1$ (correct!)
- $q^*(w, \rightarrow) = ?$
- $q_{\pi \neq b} = ?$
- Observation drawn from π are useful for evaluating q_π
- Once the policy is changed these observations are irrelevant
- This is not **sample efficient**!



State	$\pi(s)$	$Q_\pi(\uparrow)$	$Q_\pi(\rightarrow)$
u	exit	NA	NA
w	→	0	-1
x	↑	-1	0
y	exit	NA	NA
z	exit	NA	NA

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

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

Off-policy learning

- We would like to use observations drawn from some policy b to evaluate q_π where $\pi \neq b$, specifically, we want to evaluate q_{π^*}
- We strive for full utilization of previous experience
- Off-policy learning allows us to optimize a ***target policy*** while following another ***behavior policy***
- **Pros:** sample efficient
- **Cons:** greater variance in value estimations

Off-policy learning conditions

- **Objective**: use episodes from b to estimate values for π
- For off-policy learning we must assume ***coverage***
  - $\forall s, a,\ \pi(a|s) > 0 \Longrightarrow b(a|s) > 0$
- If this is true, then by running b repeatedly we will eventually discover all possible trajectories for π
- If coverage is violated for some s, a, then no inference is possible regarding that state-action value

Importance sampling

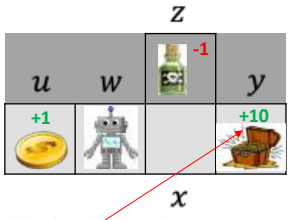
- Given a trajectory τ drawn by running b
- We can **define** (not compute) the probability $\Pr\{\tau|b\}$
- We can also **define** $\Pr\{\tau|\pi\}$
- Define the ***importance sampling ratio*** as: $\rho_t = \frac{\Pr\{\tau_t|\pi\}}{\Pr\{\tau_t|b\}}$
- Can we **compute** ρ without a model, $p(S_{K+1}|S_K, A_K)$?
- $\rho_t = \frac{\prod_{k=t}^{T-1} \pi(A_K|S_K) p(S_{K+1}|S_K,A_K)}{\prod_{k=t}^{T-1} b(A_K|S_K) p(S_{K+1}|S_K,A_K)} = \prod_{k=t}^{T-1} \frac{\pi(A_K|S_K)}{b(A_K|S_K)}$ **YES!**

Importance sampling

- **Fact**: $\mathbb{E}_{\tau\sim b}[G_t|S_t = s] = v_b(s)$
- Importance sampling allows us to compute an unbiased estimate of $v_\pi(s)$ by running b
- **Claim**: $\mathbb{E}_{\tau\sim b}[\rho_t G_t|S_t = s] = v_\pi(s)$
- We set $v_\pi(s)$ to be a weighted average of observed returns (weighted by the importance ratio)
- Assume visiting state s over M episodes using policy b
  - s is first visited during time step t^m during each episode, m $\in M$
- $v_\pi(s) = \frac{\sum_{m\in M} \rho_{t^m} G_{t^m}}{M}$

(Ordinary) Importance sampling - example

- $b(s|a) = \begin{cases}\rightarrow, p(0.5)\\ \leftarrow, p(0.5)\end{cases}$
- $\pi(s|a) = \begin{cases}\rightarrow, p(0.99)\\ \leftarrow, p(0.01)\end{cases}$
- $\tau_1 = \{w, \rightarrow, 0, x, \rightarrow, 0, y, exit, 10\}$
- $v_\pi(w) = \frac{\sum_{m\in M} \rho_{t^m} G_{t^m}}{M} = \frac{0.99}{0.5} * \frac{0.99}{0.5} * 10 = 3.96 * 10$
- $\tau_2 = \{w, \leftarrow, 0, u, exit, 1\}$
- $v_\pi(w) = \frac{\sum_{m\in M} \rho_{t^m} G_{t^m}}{M} = \frac{3.96*10+0.02*1}{2}$

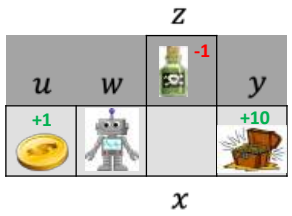


39.6 ??
Ordinary Importance sampling is unbiased yet high variance

Weighted importance sampling

- $v_\pi(s) = \frac{\sum_{m\in M}[\rho_{t^m} G_{t^m}]}{\sum_{m\in M} \rho_{t^m}}$
- $b(s|a) = \begin{cases}\rightarrow, p(0.5)\\ \leftarrow, p(0.5)\end{cases}$
- $\pi(s|a) = \begin{cases}\rightarrow, p(0.99)\\ \leftarrow, p(0.01)\end{cases}$
- $\tau_1 = \{w, \rightarrow, 0, x, \rightarrow, 0, y, exit, 10\}$
- $v_\pi(w) = \frac{\sum_{m\in M}[\rho_{t^m} G_{t^m}]}{\sum_{m\in M} \rho_{t^m}} = \frac{3.96*10}{3.96}$
- $\tau_2 = \{w, \leftarrow, 0, u, exit, 1\}$
- $v_\pi(w) = \frac{\sum_{m\in M}[\rho_{t^m} G_{t^m}]}{\sum_{m\in M} \rho_{t^m}} = \frac{3.96*10+0.02*1}{3.98}$

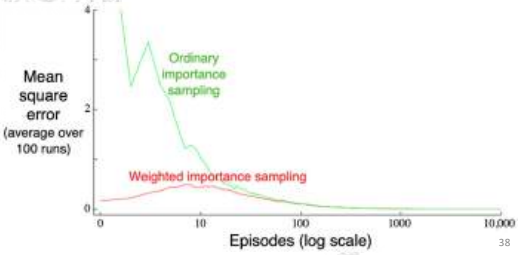
Trick: normalize by the sum of importance ratios



Ordinary Importance sampling is unbiased while the weighted version is biased (initially). Ordinary Importance sampling results in high variance while the weighted version has a bounded variance

Ordinary vs weighted importance sampling

- Estimating a black-jack state
- Target policy: hit on 19 or below
- Behavior policy: random (uniform)
- Both approaches converge to the true value
- weighted importance sampling is much better initially



Revisit : Incremental Implementation (Slide taken from Multi-armed Bandit)

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

Note:

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

Discussion:

Const vs. Variable step size?

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

MC control + importance sampling

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $C(s,a) \leftarrow 0$
  $\pi(s) \leftarrow \operatorname{argmax}_a Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
  $b \leftarrow$ any soft policy
  Generate an episode using b:
    $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
  $G \leftarrow 0$
  $W \leftarrow 1$
  For $t = T-1, T-2, \ldots$ down to 0:
    $G \leftarrow \gamma G + R_{t+1}$
    $C(S_t,A_t) \leftarrow C(S_t,A_t) + W$
    $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t,A_t)]$
    $\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t,a)$ (with ties broken consistently)
    If $A_t \neq \pi(S_t)$ then exit For loop
    $W \leftarrow W\frac{1}{b(A_t|S_t)}$

Accumulated reward after step t

MC control + importance sampling

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $C(s,a) \leftarrow 0$
  $\pi(s) \leftarrow \operatorname{argmax}_a Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
  $b \leftarrow$ any soft policy
  Generate an episode using b:
    $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
  $G \leftarrow 0$
  $W \leftarrow 1$
  For $t = T-1, T-2, \ldots$ down to 0:
    $G \leftarrow \gamma G + R_{t+1}$
    $C(S_t,A_t) \leftarrow C(S_t,A_t) + W$
    $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t,A_t)]$
    $\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t,a)$ (with ties broken consistently)
    If $A_t \neq \pi(S_t)$ then exit For loop
    $W \leftarrow W\frac{1}{b(A_t|S_t)}$

1 over Joint probability over observed actions from following b after step t. This equals the IS ratio here because the target policy is deterministic.

MC control + importance sampling

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
  $Q(s,a) \leftarrow$ arbitrary
  $C(s,a) \leftarrow 0$
  $\pi(s) \leftarrow \operatorname{argmax}_a Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
  $b \leftarrow$ any soft policy
  Generate an episode using b:
    $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
  $G \leftarrow 0$
  $W \leftarrow 1$
  For $t = T-1, T-2, \ldots$ down to 0:
    $G \leftarrow \gamma G + R_{t+1}$
    $C(S_t,A_t) \leftarrow C(S_t,A_t) + W$
    $Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t,A_t)]$
    $\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t,a)$ (with ties broken consistently)
    If $A_t \neq \pi(S_t)$ then exit For loop
    $W \leftarrow W\frac{1}{b(A_t|S_t)}$

Going back in time

MC control + importance sampling

Discount future rewards and add immediate reward

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + importance sampling

Cumulative sum of IS weights affiliated with S_t, A_t (for weighted IS)

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + importance sampling

Incremental update of Q values (waited moving average)

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + importance sampling

Update target policy (greedy)

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + importance sampling

Sinse π is deterministic, once we diverge from it all IS weights of earlier actions will be 0

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + importance sampling

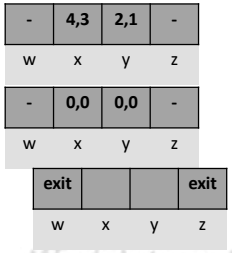
Update the joint prob by multiplying by ρ_t. Notice that $\pi(S_t) = 1$ in this example (deterministic target policy)

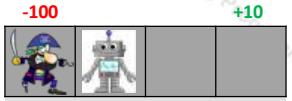
Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example



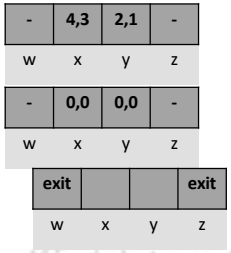


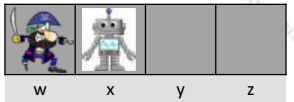
Off-po

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example





Off-po

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

-

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

Off-po

Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

-

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

Off-po

Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

-

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

Off-po

Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

-

-100	4,3	2,1	5
w	x	y	z
1	0,0	0,0	0
w	x	y	z
exit			exit
w	x	y	z

Off-po

Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

-

-100	4,3	2,1	5
w	x	y	z
1	0,0	0,0	0
w	x	y	z
exit			exit
w	x	y	z

Off-po

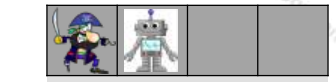
Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

-

-100	4,3	2,1	5
w	x	y	z
1	0,0	0,0	0
w	x	y	z
exit			exit
w	x	y	z



Off-po

Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

MC control + IS example

-

-100	4,3	2,1	5
w	x	y	z
1	0,0	0,0	0
w	x	y	z
exit			exit
w	x	y	z

Off-po

Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

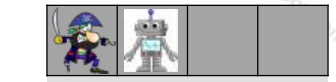
Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

MC control + IS example

-

-100	4,3	2,1	5
w	x	y	z
1	2,0	0,0	0
w	x	y	z
exit			exit
w	x	y	z



Off-po

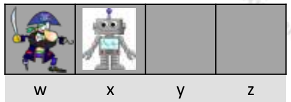
Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

MC control + IS example

-100 +10



-100	-90,3	2,1	5
w	x	y	z

1	2,0	0,0	0
w	x	y	z

exit			exit
w	x	y	z

•

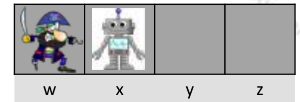
Off-pol

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

-100 +10



-100	-90,3	2,1	5
w	x	y	z

1	2,0	0,0	0
w	x	y	z

exit			exit
w	x	y	z

•

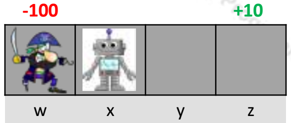
Off-pol

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

-100 +10



-100	-90,3	2,1	5
w	x	y	z

1	2,0	0,0	0
w	x	y	z

exit			exit
w	x	y	z

•

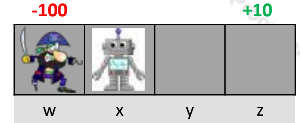
Off-pol

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

-100 +10



-100	-90,3	2,1	5
w	x	y	z

1	2,0	0,0	0
w	x	y	z

exit			exit
w	x	y	z

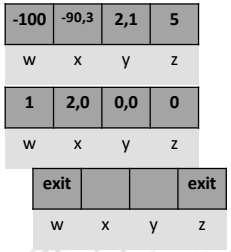
•

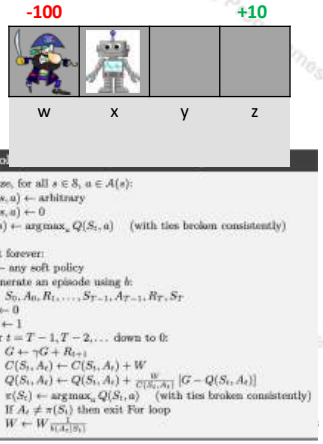
Off-pol

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)

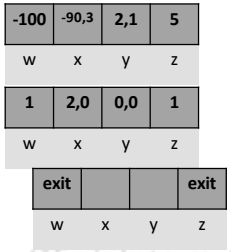
Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

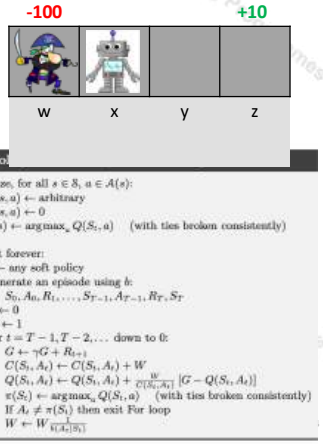
MC control + IS example

- 

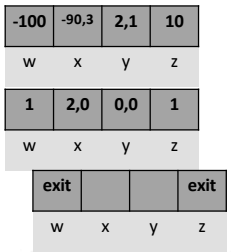


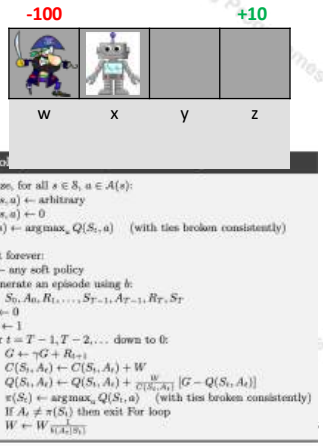
MC control + IS example

- 

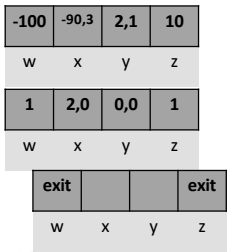


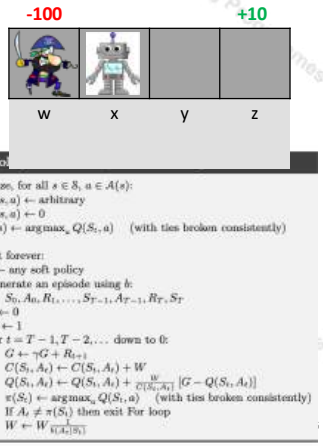
MC control + IS example

- 



MC control + IS example

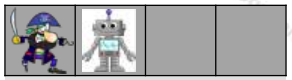
- 



MC control + IS example

-

-100	-90,3	2,1	10
w	x	y	z
1	2,0	0,0	1
w	x	y	z
exit			exit
w	x	y	z



Off-po

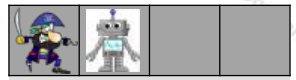
Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

-

-100	-90,3	2,1	10
w	x	y	z
1	2,0	0,0	1
w	x	y	z
exit			exit
w	x	y	z



Off-po

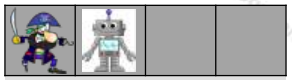
Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

-

-100	-90,3	2,1	10
w	x	y	z
1	2,0	0,2	1
w	x	y	z
exit			exit
w	x	y	z



Off-po

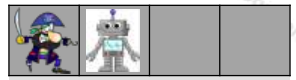
Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

-

-100	-90,3	2,9	10
w	x	y	z
1	2,0	0,2	1
w	x	y	z
exit			exit
w	x	y	z



Off-po

Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

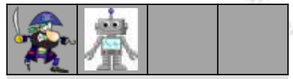
•

-100	-90,3	2,9	10
w	x	y	z

1	2,0	0,2	1
w	x	y	z

exit			exit
w	x	y	z

-100 +10



w x y z

Off-pol

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

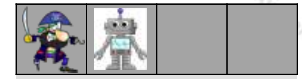
•

-100	-90,3	2,9	10
w	x	y	z

1	2,0	0,2	1
w	x	y	z

exit			exit
w	x	y	z

-100 +10



w x y z

Off-pol

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

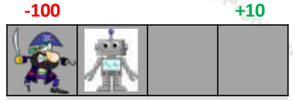
•

-100	-90,3	2,9	10
w	x	y	z

1	2,0	0,2	1
w	x	y	z

exit			exit
w	x	y	z

-100 +10



w x y z

Off-pol

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

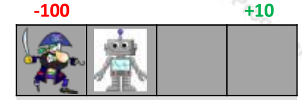
•

-100	-90,3	2,9	10
w	x	y	z

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

exit			exit
w	x	y	z

-100 +10



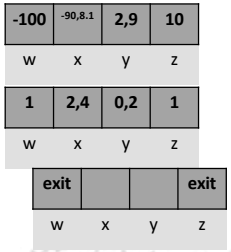
w x y z

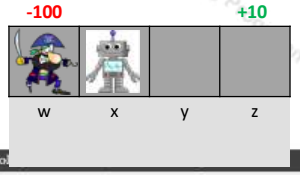
Off-pol

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t,A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example



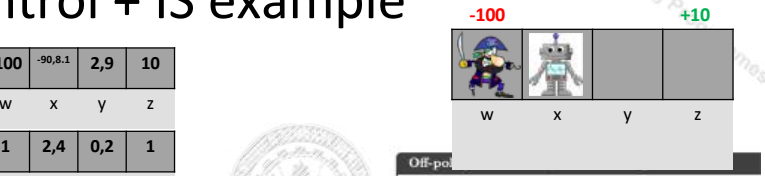


Off-po

Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example

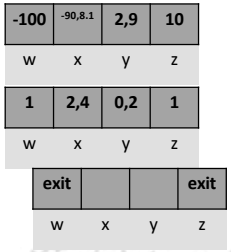


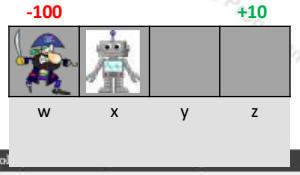
Off-po

Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

MC control + IS example





Off-po

Initialize, for all $s \in S$, $a \in A(s)$:
$Q(s,a) \leftarrow$ arbitrary
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(S_t,a)$ (with ties broken consistently)

Repeat forever:
$b \leftarrow$ any soft policy
Generate an episode using b:
$S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$
$G \leftarrow 0$
$W \leftarrow 1$
For $t = T-1, T-2, \ldots$ down to 0:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit For loop
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

What did we learn?

- Online evaluation through the Monte-Carlo approach
  - Update V or Q estimations based on observed returns
- On policy Monte-Carlo control
  - Beware of local optimum. Must explore!
  - Consider using a soft policy $\pi(a|s)$
  - Assign soft policy values that mimic an ε-greedy strategy
  - On policy learning is not sample efficient!
- Off policy Monte-Carlo control
  - Define target policy (e.g., $\arg\max_a Q(S_t, a)$) that can be different than the behavior policy
  - Utilize weighted importance sampling to train a target policy
  - $v_\pi(s) = \frac{\sum_{m \in M}[\rho_t m G_t^m]}{\sum_{m \in M} \rho_t m}$

Deep Reinforcement Learning

BITS Pilani

Prof. Vimal SP
CSIS

BITS Pilani

<AIMLCZG512, Deep Reinforcement Learning>
Lecture No. 9

Solving MDPs so far

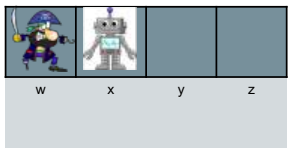
Dynamic programming
Off policy
local learning, propagating values from neighbors (Bootstrapping)
Model based

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

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

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

$\gamma = 0.9$



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

Fuse DP and MC

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

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

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

Temporal difference learning

•In Monte-Carlo, G_t is an actual return from the complete episode. Now, if we replace G_t with an estimated return $R(t+1)+V(S_{t+1})$, this is what *TD(0)* would look like:

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

Where:

•$R(t+1)+V(S_{t+1})$ is called **TD target value**

•$R(t+1)+V(S_{t+1})- V(S_t)$ is called **TD error**.

•MC uses accurate return Gt to update value, while TD uses the Bellman Optimality Equation to estimate value, and then updates the estimated value with the target value.

Temporal Difference learning

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  until S is terminal

SARSA: On-policy TD Control

- learn an action-value function rather than a state-value function

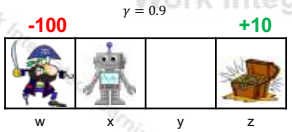
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big].$$

- This update is done after every transition from a nonterminal state S_t. If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero.
- Quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$

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

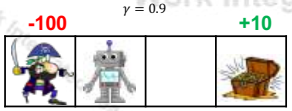


SARSA: On-policy TD Control

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

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

$\gamma = 0.9$



$Q =$

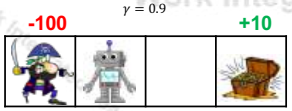
0	0,0	0,0	0
w	x	y	z

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

$\gamma = 0.9$



x	←	0	w	null
S	A	R	S'	A'

$Q =$

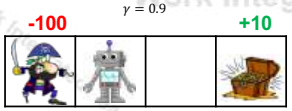
0	0,0	0,0	0
w	x	y	z

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

$\gamma = 0.9$



x	←	0	w	exit
S	A	R	S'	A'

$Q =$

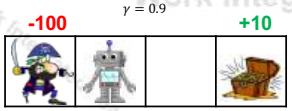
0	0,0	0,0	0
w	x	y	z

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

$\gamma = 0.9$



x	←	0	w	exit
S	A	R	S'	A'

$Q =$

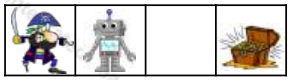
0	0,0	0,0	0
w	x	y	z

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

$\gamma = 0.9$



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

$Q =$

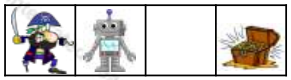
0	0,0	0,0	0
w	x	y	z

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

$\gamma = 0.9$



w	exit	−100	ter	exit
S	A	R	S'	A'

$Q =$

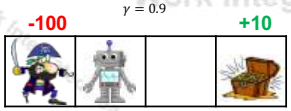
0	0,0	0,0	0
w	x	y	z

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

$\gamma = 0.9$



w	exit	−100	ter	·
S	A	R	S'	A'

$Q =$

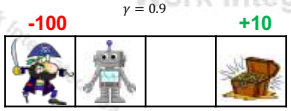
-100	0,0	0,0	0
w	x	y	z

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

$\gamma = 0.9$



x	←	−100	ter	·
S	A	R	S'	A'

$Q =$

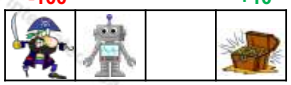
-100	0,0	0,0	0
w	x	y	z

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

$\gamma = 0.9$



x	←	0	w	exit
S	A	R	S'	A'

$Q =$

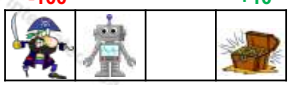
-100	0,0	0,0	0
w	x	y	z

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

$\gamma = 0.9$



x	←	0	w	exit
S	A	R	S'	A'

$Q =$

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

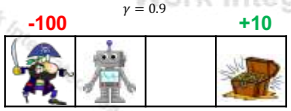
SARSA: On-policy TD Control

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

And so on…

$\gamma = 0.9$



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

$Q =$

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

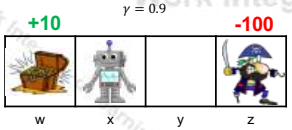
Q-learning: Off-policy TD Control

- 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

-learning: Off-policy TD Control

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

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



x	null	null	null
S	A	R	S'

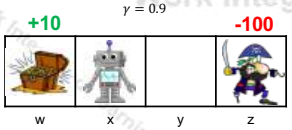
$Q =$

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

-learning: Off-policy TD Control

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

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



x	→	0	y
S	A	R	S'

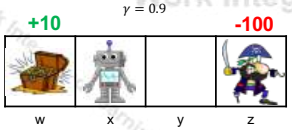
$Q =$

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

-learning: Off-policy TD Control

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

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



x	→	0	y
S	A	R	S'

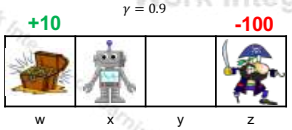
$Q =$

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

-learning: Off-policy TD Control

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

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



y	→	0	y
S	A	R	S'

$Q =$

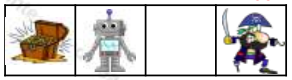
0	0,0	0,0	0
w	x	y	z

-learning: Off-policy TD Control

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

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

$\gamma = 0.9$



y	→	0	z
S	A	R	S'

$Q =$

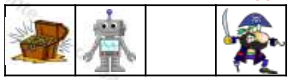
0	0,0	0,0	0
w	x	y	z

-learning: Off-policy TD Control

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

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

$\gamma = 0.9$



z	→	0	z
S	A	R	S'

$Q =$

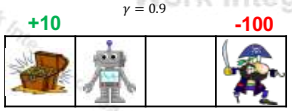
0	0,0	0,0	0
w	x	y	z

-learning: Off-policy TD Control

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

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

$\gamma = 0.9$



z	exit	−100	·
S	A	R	S'

$Q =$

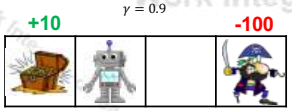
0	0,0	0,0	0
w	x	y	z

-learning: Off-policy TD Control

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

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

$\gamma = 0.9$



z	exit	−100	·
S	A	R	S'

$Q =$

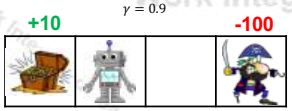
0	0,0	0,0	-100
w	x	y	z

-learning: Off-policy TD Control

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

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

$\gamma = 0.9$



x	exit	−100	·
S	A	R	S'

$Q =$

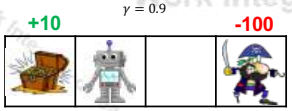
0	0,0	0,0	-100
w	x	y	z

-learning: Off-policy TD Control

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

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

$\gamma = 0.9$



x	→	0	y
S	A	R	S'

$Q =$

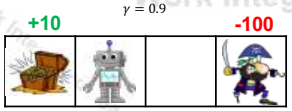
0	0,0	0,0	-100
w	x	y	z

-learning: Off-policy TD Control

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

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

$\gamma = 0.9$



y	→	0	z
S	A	R	S'

$Q =$

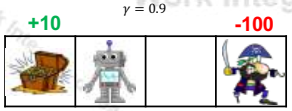
0	0,0	0,0	-100
w	x	y	z

-learning: Off-policy TD Control

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

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

$\gamma = 0.9$



y	→	0	z
S	A	R	S'

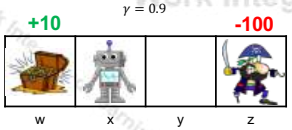
$Q =$

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

-learning: Off-policy TD Control

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

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



x	→	0	z
S	A	R	S'

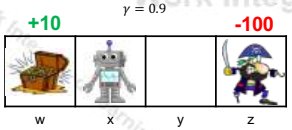
$Q =$

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

-learning: Off-policy TD Control

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

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



x	→	0	y
S	A	R	S'

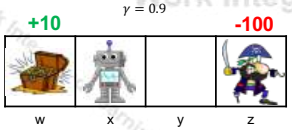
$Q =$

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

-learning: Off-policy TD Control

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

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



x	→	0	y
S	A	R	S'

$Q =$

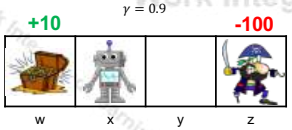
0	0,0	0,-90	-100
w	x	y	z

-learning: Off-policy TD Control

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

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

And so on…



x	→	0	y
S	A	R	S'

$Q =$

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

Expected SARSA

- Instead of the maximum over next state–action pairs it uses the expected value, taking into account how likely each action is under the current policy

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big]$$
$$= Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big],$$

Maximization Bias & Double Learning

- Maximum over estimated values is used implicitly as an estimate of the

maximum value, which can lead to a significant positive bias.

- Solution – Learn two estimates – Double Learning
- Double learning doubles the memory requirements, but does not

increase the amount of computation per step.

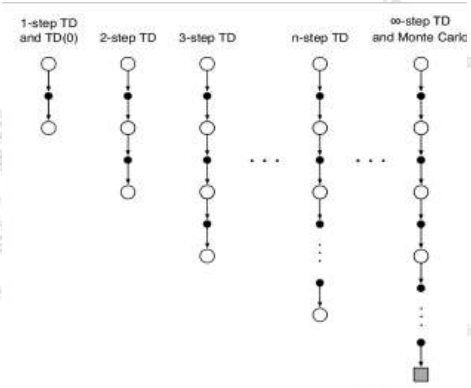
n-step TD Prediction

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

n-step TD Prediction

One-step return:

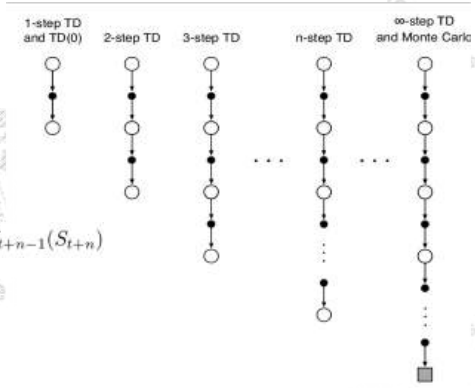
$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1})$$

Two-step return:

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

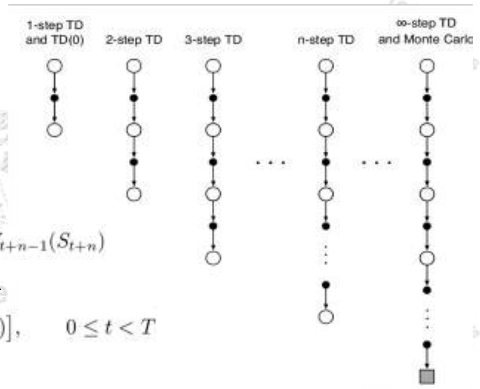
n-step return:

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



Ref Section 7.1 of TB

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:

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

Ref Section 7.1 of TB

n-step TD Prediction

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

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

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

Deep Reinforcement Learning

Prof. Vimal SP
CSIS

BITS Pilani

BITS Pilani

<AIMLCZG512, Deep Reinforcement Learning>
Lecture No. 4,5,6,7,8,9 - Incremental example

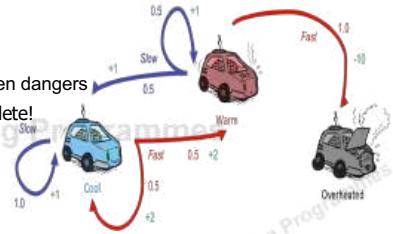
1. MDP
2. DP
3. MC
4. TD

Race Car Laps

A race car completes laps while managing engine temperature and speed decisions. Here each step is a segment of a lap in the track where the car learns to trades off lap time against overheating risk. The more the lap the car completes it performance betters. Car shuts down only when its engine get overheated (optional terminal state).

Key Characteristics:

- The agent must act under **uncertainty** — percepts are local, and the agent must reason about hidden dangers
- Assume there is no fuel constraints or No.of.Laps to complete!
- At every time step the next potential engine temperature depends on the current engine state not the past history

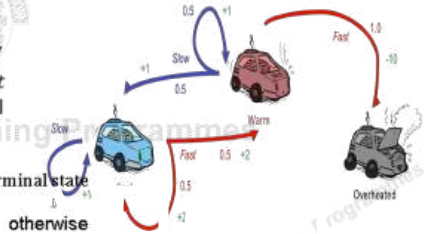


1. States (S)
   - S = (Cool(**C**), Warm(**W**), Overheated(**O**)) #Engine Temperature
   - Finite State MDP
2. Actions (A)
   - A = {Go Slow(**S**), Go Fast **(F)** }
3. Rewards (R)

$$R(s,a,s') = \begin{cases} +1 & \text{if } s' \text{ is reached by going } Slow \\ +2 & \text{if } s' \text{ is reached by going } Fast \\ -10 & \text{if } s' \text{ is terminal} \end{cases}$$

4. State Transition Probabilities (P)

$$P(s'|s,a) = \begin{cases} 0.5 & \text{if action lead to different non-terminal state} \\ 1 & \text{otherwise} \end{cases}$$

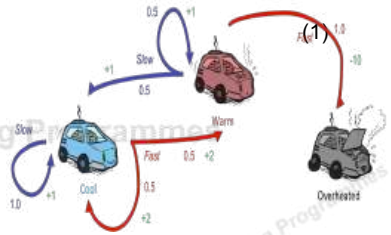


1. MDP
2. **DP**
3. MC
4. TD

Scenario

In a modified Race car, there is a training track where agent is given a known map and risk factors(dynamics) in advance. Agent knows exactly where it is and its action how actions result in engine state change (KNOWN MODEL).

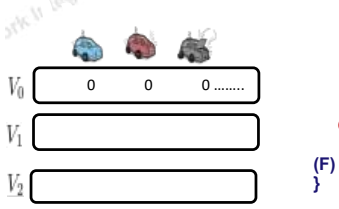
With full knowledge, the agent can compute optimal policies by trying through all states.

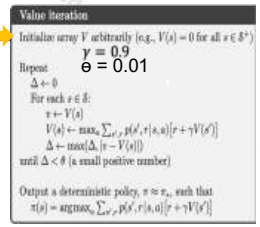


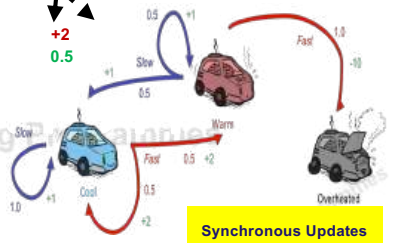
1. Given
   - P(s` | s, a) , R(s, a, s`) , P(s`, r | s, a)
2. To find
   - Policy π
3. Solution
   - Policy Iteration (PI): Prefer this in a relatively smaller state space where intermediate termination provides explicit interpretable policy
   - Value Iteration (VI): Prefer this in a relatively larger state space where fast convergence to optimal policy is obtained at the end
4. Idea
   - PI : One Step Look ahead □ Evaluate □ Improve
   - VI : One Step Look ahead □ Control

VI can be thought of truncated PI where evaluation is done fully before improvement.

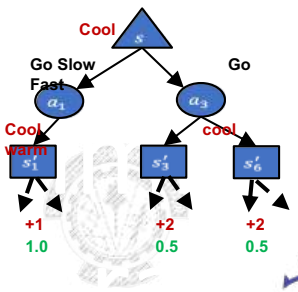
Dynamic Programming Solution







Synchronous Updates



Value iteration

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

$\gamma = 0.9$

$\theta = 0.01$

Repeat

$\Delta \leftarrow 0$

For each $s \in S$:

$v \leftarrow V(s)$

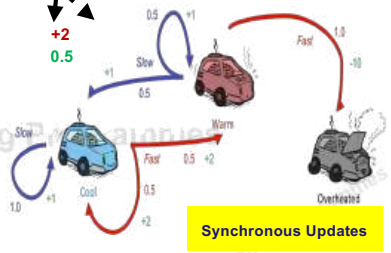
$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$

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

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

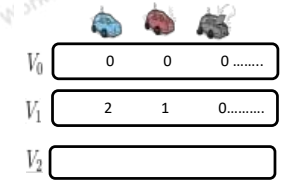
Output a deterministic policy, $\pi \approx \pi_*$, such that

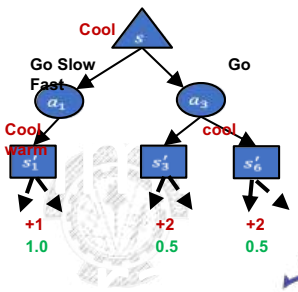
$\pi(s) = \text{argmax}_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$



Synchronous Updates

innovate achieve lead





Value iteration

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

$\gamma = 0.9$

$\theta = 0.01$

Repeat

$\Delta \leftarrow 0$

For each $s \in S$:

$v \leftarrow V(s)$

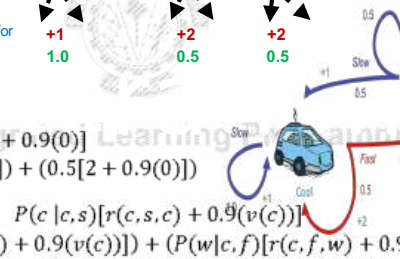
$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$

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

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

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

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



Synchronous Updates

- Below is done only for state : "Cool". Repeat this for all the states "Warm", "Overheated" per iteration

$$V(cool) = \max \begin{cases} slow: 1 \\ fast: 2 \end{cases}$$

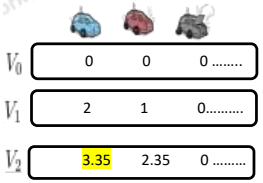
$$V(cool) = \max \begin{cases} 1 * [1 + 0.9(0)] \\ (0.5 * [2 + 0.9(0)]) + (0.5[2 + 0.9(0)]) \end{cases}$$

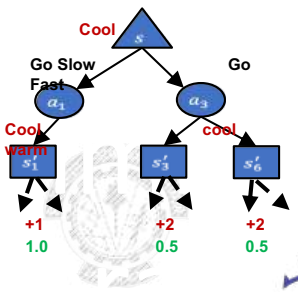
$$V(cool) = \max \begin{cases} P(c|c,s)[r(c,s,c) + 0.9(v(c))] \\ (P(c|c,f)[r(c,f,c) + 0.9(v(c))]) + (P(w|c,f)[r(c,f,w) + 0.9(v(w))]) \end{cases}$$

Deep Reinforcement Learning

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

innovate achieve lead





Value iteration

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

$\gamma = 0.9$

$\theta = 0.01$

Repeat

$\Delta \leftarrow 0$

For each $s \in S$:

$v \leftarrow V(s)$

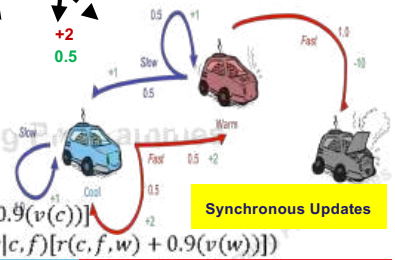
$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$

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

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

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

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



Synchronous Updates

- Below is done only for state : "Cool". Repeat this for all the states "Warm", "Overheated" per iteration

$$V(cool) = \max \begin{cases} slow: 2.8 \\ fast: 3.35 \end{cases}$$

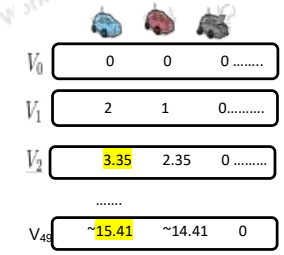
$$V(cool) = \max \begin{cases} 1 * [1 + 0.9(2)] \\ (0.5 * [2 + 0.9(2)]) + (0.5[2 + 0.9(1)]) \end{cases}$$

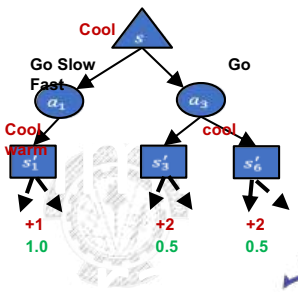
$$V(cool) = \max \begin{cases} P(c|c,s)[r(c,s,c) + 0.9(v(c))] \\ (P(c|c,f)[r(c,f,c) + 0.9(v(c))]) + (P(w|c,f)[r(c,f,w) + 0.9(v(w))]) \end{cases}$$

Deep Reinforcement Learning

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

innovate achieve lead





Value iteration

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

$\gamma = 0.9$

$\theta = 0.01$

Repeat

$\Delta \leftarrow 0$

For each $s \in S$:

$v \leftarrow V(s)$

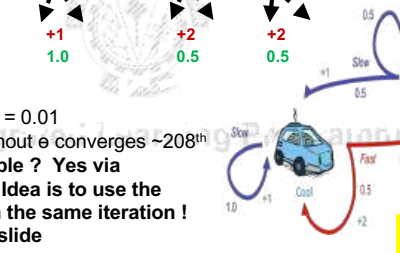
$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$

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

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

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

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

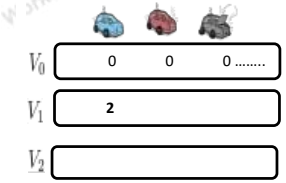


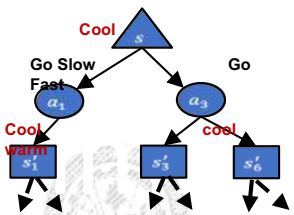
Synchronous Updates

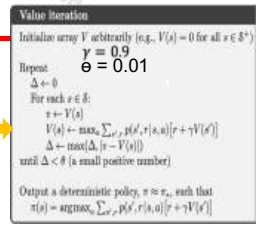
- Δ = ~0.009 less than threshold θ = 0.01
- Algorithm converges slowly. Without θ converges ~208th
- **Can faster convergence possible ? Yes via Asynchronous Value Updates. Idea is to use the recent values V(S`) even within the same iteration ! One iteration is shown in next slide**

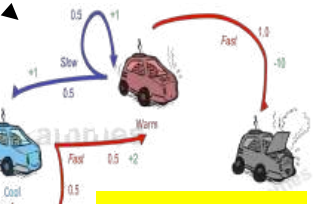
Deep Reinforcement Learning

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956









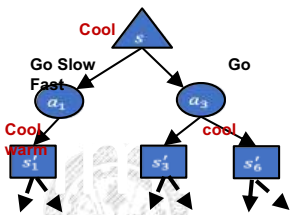
- Below is done only for state : "Cool". In this same iteration while updating the V(Warm) use recent value of V(Cool) = 2 not 0 .

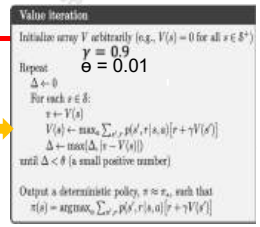
$$V(cool) = \max\begin{cases} slow: 1 \\ fast: 2 \end{cases}$$

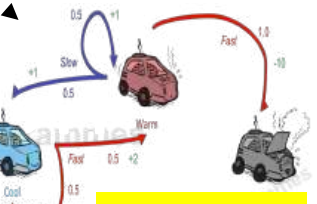
$$V(cool) = \max\begin{cases} 1 * [1 + 0.9(0)] \\ (0.5 * [2 + 0.9(0)]) + (0.5[2 + 0.9(0)]) \end{cases}$$

$$V(cool) = \max\begin{cases} P(c\,|c,s)[r(c,s,c) + 0.9(v(c))] \\ (P(c\,|c,f)[r(c,f,c) + 0.9(v(c))]) + (P(w|c,f)[r(c,f,w) + 0.9(v(w))]) \end{cases}$$

Asynchronous Updates







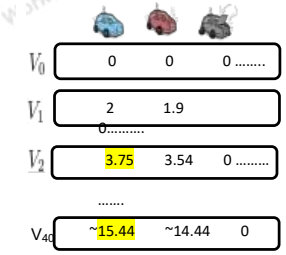
- Below is done only for state : "Cool". In this same iteration while updating the V(Warm) use recent value of V(Cool) = 2 .

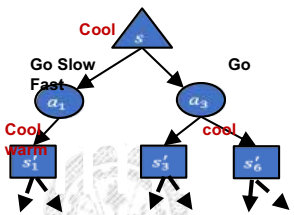
$$V(warm) = \max\begin{cases} fast: -10 \\ slow: 1.9 \end{cases}$$

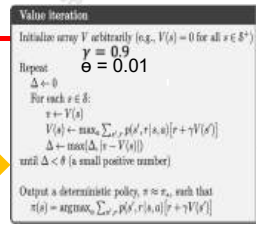
$$V(warm) = \max\begin{cases} 1 * [-10 + 0.9(0)] \\ (0.5 * [1 + 0.9(2)]) + (0.5[1 + 0.9(0)]) \end{cases}$$

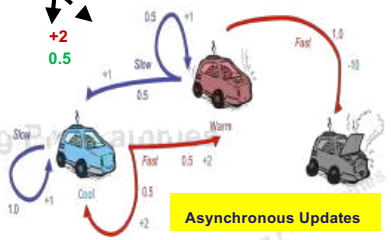
$$V(warm) = \max\begin{cases} P(o\,|w,f)[r(w,f,o) + 0.9(v(o))] \\ (P(c\,|w,s)[r(w,s,c) + 0.9(v(c))]) + (P(w|w,s)[r(w,s,w) + 0.9(v(w))]) \end{cases}$$

Asynchronous Updates



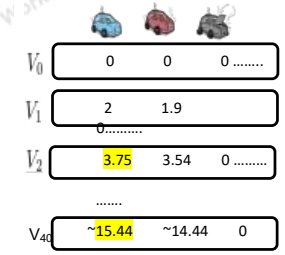


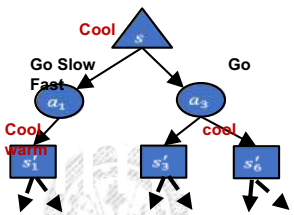


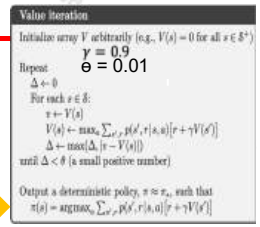


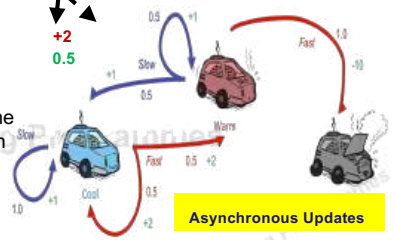
- Δ = ~0.008 less than threshold ɵ = 0.01
- Algorithm converges fast. Without ɵ converges at ~157th iteration!

Asynchronous Updates









State	Best Action
Cool	Fast
Warm	Slow
Overheated	-

- After convergence extract the policy by choosing the action that lead to the converged maximum value here from V40th iteration

Asynchronous Updates

Dynamic Programming Solution

In many cases, sometimes policies is observed to converge faster than value!

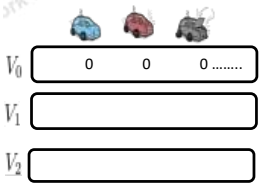
So why wait for entire value updates?

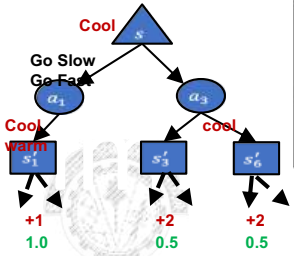
$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*,$$

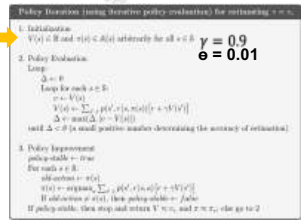
1. Given
   - P(s` | s, a) , R(s, a, s`) , P(s`, r | s, a)
2. To find
   - Policy π
3. Solution
   - Policy Iteration (PI): Prefer this in a relatively smaller state space where intermediate termination provides explicit interpretable policy
   - Value Iteration (VI): Prefer this in a relatively larger state space where fast convergence to optimal policy is obtained at the end
4. Idea
   - PI : One Step Look ahead Evaluate Improve
   - VI : One Step Look ahead Control

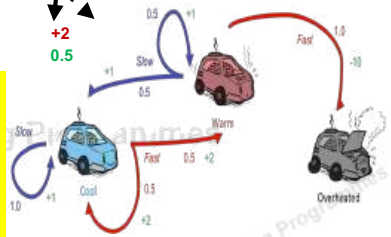
 Number of state sweeps (updates) can be controlled if required (similar to mini-batch /stochastic processing). Evaluate & Improve steps are alternatively are done .

VI can be thought of truncated PI where evaluation is done fully before improvement.







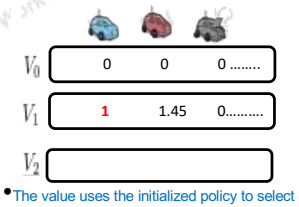


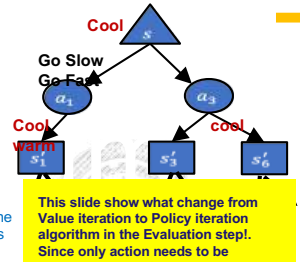
π_0

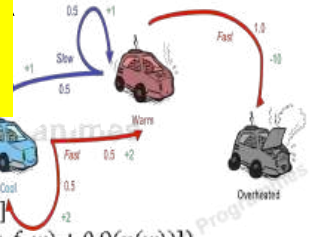
State	Best Action
Cool	Slow
Warm	Slow
Overheated	-

We will use asynchronous for updates in this example. Note that even though above backup diagram depicts all possible transition, this algorithm computes value from only one of the action as given by the initialized policy at the start of every iteration!!!

Policy Evaluation Starts







- The value uses the initialized policy to select only one action GREEDILY. Hence below cancelled equations differentiated PI vs VI

$$V(cool) = \max \begin{cases} slow: 1 \\ fast: 2 \end{cases}$$

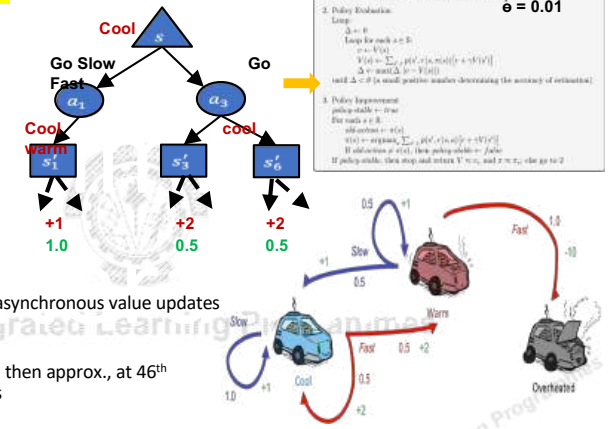
This slide show what change from Value iteration to Policy iteration algorithm in the Evaluation step!. Since only action needs to be assessed, MAX of all actions outcome is no longer required in this PE cycle!

$$V(cool) = \max \begin{cases} 1 * [1 + 0.9(0)] \\ (0.5 * [2 + 0.9(0)]) + (0.5[2 + 0.9(0)]) \end{cases}$$

$$V(cool) = \max \begin{cases} P(c|c,s)[r(c,s,c) + 0.9(v(c))] \\ (P(c|c,f)[r(c,f,c) + 0.9(v(c))]) + (P(w|c,f)[r(c,f,w) + 0.9(v(w))]) \end{cases}$$

Policy Evaluation Completes

V_0	0	0	0 .....
V_1	1	1.45	0.....
V_2	1.9	2.51	
	0.....		
V_{225}	~10	~ 10	0



- In PE, either synchronous or asynchronous value updates can be followed
- If Threshold of 0.01 is applied then approx., at 46th iteration algorithm converges
- Assume that Asynchronous method is used & Thresholding is not applied. In this case at approx., near 225th iteration values converge

Policy Improvement

V_0	0	0	0 .....
V_1	1	1.45	0.....
V_2	1.9	2.51	
	0.....		
V_{225}	~10	~ 10	0

$$V(cool) = \max\begin{cases} slow: 10 \\ fast: 11 \end{cases}$$

$$V(cool) = \max\begin{cases} 1 * [1 + 0.9(10)] \\ (0.5 * [2 + 0.9(10)]) + (0.5[2 + 0.9(10)]) \end{cases}$$

$$V(cool) = \max\begin{cases} P(c\,|c,s)[r(c,s,c) + 0.9(v(c))] \\ (P(c\,|c,f)[r(c,f,c) + 0.9(v(c))]) + (P(w|c,f)[r(c,f,w) + 0.9(v(w))]) \end{cases}$$

After PE is done, PI starts where the last value from PE is used to find the next best action. In this PI phase all actions are analysed and best action that MAXimises the value is chosen to update the policy

Policy Improvement

V_0	0	0	0 .....
V_1	1	1.45	0.....
V_2	1.9	2.51	
	0.....		
V_{225}	~10	~ 10	0

$$V(warm) = \max\begin{cases} fast: -10 \\ slow: 10 \end{cases}$$

$$V(warm) = \max\begin{cases} 1 * [-10 + 0.9(0)] \\ (0.5 * [1 + 0.9(10)]) + (0.5[1 + 0.9(10)]) \end{cases}$$

$$V(warm) = \max\begin{cases} P(o\,|w,f)[r(w,f,o) + 0.9(v(o))] \\ (P(c\,|w,s)[r(w,s,c) + 0.9(v(c))]) + (P(w|w,s)[r(w,s,w) + 0.9(v(w))]) \end{cases}$$

After PE is done, PI starts where the last value from PE is used to find the next best action. In this PI phase all actions are analysed and best action that MAXimises the value is chosen to update the policy

Policy Improvement Completes

V_0	0	0	0 .....
V_1	1	1.45	0.....
V_2	1.9	2.51	
	0.....		
V_{225}	~10	~ 10	0

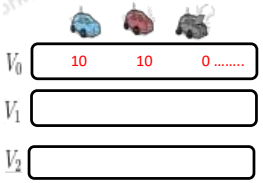
$$V(warm) = \max\begin{cases} fast: -10 \\ slow: 10 \end{cases}$$

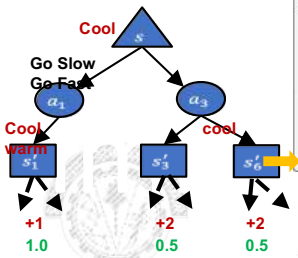
π_1

State	Best Action
Cool	Fast
Warm	Slow
Overheated	-

$$V(cool) = \max\begin{cases} slow: 10 \\ fast: 11 \end{cases}$$

Continue another cycle of PE□PI



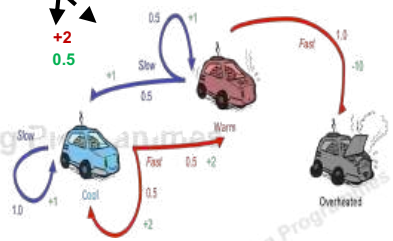


$\gamma = 0.9$
$\theta = 0.01$

π_1

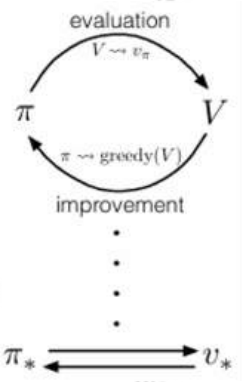
State	Best Action
Cool	Fast
Warm	Slow
Overheated	-

After this cycle of algorithm converges with values :
V(cool) = 15.5
V(Warm) = 14.5



Observation

- Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement).
- In policy iteration, these two processes alternate, each completing before the other begins, **but this is not really necessary.**



Solution : GPI Generalized Policy Iteration

- Generalized policy iteration (GPI) to refer to the general idea of letting policy-evaluation and policy improvement processes interact, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as GPI.

Idea (Do this as an exercise for given problem)

- In one cycle perform only one iteration of Policy Evaluation (ie.,Value Updates) without waiting for convergence immediately follow by Policy Improvement.

1. MDP
2. DP
3. MC
4. TD

Race Car Laps

In a modified Race car, there is a training track where agent is given a known (sometimes incomplete) map and risk factors(dynamics) may not be known in advance. Agent knows exactly where it is but its actions impact on the engine state change is difficult to predict (UNKNOWN MODEL).

With partial knowledge, the agent can compute optimal policies by trying through random experiments (Sampling). Car shuts down when its engine get overheated (optional terminal state) or Car stops once the predefined no.of.laps are completed.

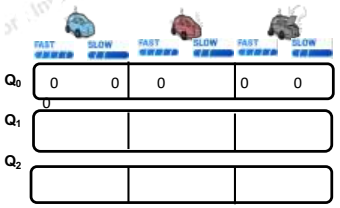
Key Characteristics:

- The agent must act under **uncertainty** — percepts are local, and the agent must reason about hidden dangers
- Assume there are fuel constraints or/and No.of.Laps to complete!
- At every time step **there is no way clear indication of Impact of next** potential engine temperature depending on the current engine state. The entire past changes might lead to any possible state change. Hence **immediate reward is not always available**.

1. States (S)
   - S = (Cool(**C**), Warm(**W**), Overheated(**O**)) #Engine Temperature
   - Finite State MDP
2. Actions (A)
   - A = {Go Slow(**S**), Go Fast(**F**)}
3. Rewards (R) and/or State Transition Probabilities (P) may be non-stationary
4. Policy still needs estimation of V(S) , Q(S,A) i.e., MDP components but its is not a necessity that the process is expected to follow a Markov Property!! ☺

1. Given
   - S, A, optionally R(s, a, s`)
2. To find
   - Policy π
3. Solution
   - Sample Episodes
   - Compute estimates of the value by discounted returns
     - Simple average
     - Weighted average
     - Normalized weighted average
4. Idea
   - On Policy :
     - Generate Sample/Experience → Evaluate using Return → Improve
   - Off Policy :
     - Refer to Sample/Experience generated by another reference(behavior) policy → Evaluate using Return → Improve

- **Monte Carlo Method**



ε = 0.2
γ = 0.6

π_0

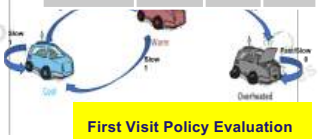
State	Best Action	P(F)	P(S)
Cool	Slow	0.4	0.6
Warm	Fast	0.6	0.4
Overheated	-	-	-

First Visit Policy Evaluation

Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

π_0

State	Best Action	P(F)	P(S)
Cool	Slow	0.4	0.6
Warm	Fast	0.6	0.4
Overheated	-	-	-



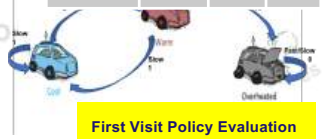
First Visit Policy Evaluation

Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

t=3: G_3 = -10
t=2: G_2 = 2 + (0.6)(-10) = -4 #Not used in First Visit
t=1: G_1 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4
t=0: G_0 = 2+ (0.6*1) + (0.6*0.6*2) + (0.6*0.6*0.6-10) = 1.16

π_0

State	Best Action	P(F)	P(S)
Cool	Slow	0.4	0.6
Warm	Fast	0.6	0.4
Overheated	-	-	-



First Visit Policy Evaluation

Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

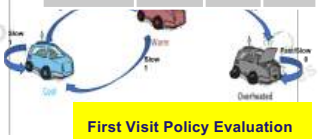
t=3: G_3 = -10
t=2: G_2 = 2 + (0.6)(-10) = -4 #Not used in First Visit
t=1: G_1 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4
t=0: G_0 = 2+ (0.6*1) + (0.6*0.6*2) + (0.6*0.6*0.6-10) = 1.16

Episode 1:

t=3: Q(W, F) = -10
t=2: Q(C, F) = -4
t=1: Q(W, S) = -1.4
t=0: Q(C, F) = 1.16

π_0

State	Best Action	P(F)	P(S)
Cool	Slow	0.4	0.6
Warm	Fast	0.6	0.4
Overheated	-	-	-



First Visit Policy Evaluation

Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

t=3: G_3 = -10
t=2: G_2 = 2 + (0.6)(-10) = -4 #Not used in First Visit
t=1: G_1 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4
t=0: G_0 = 2+ (0.6*1) + (0.6*0.6*2) + (0.6*0.6*0.6-10) = 1.16

Episode 1:

t=3: Q(W, F) = -10
t=2: Q(C, F) = -4
t=1: Q(W, S) = -1.4
t=0: Q(C, F) = 1.16

π_1

State	Best Action	P(F)	P(S)
Cool	Slow	0.2	0.8
Warm	Slow	0.2	0.8
Overheated	-	-	-

Policy Improvement

Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

t=3: G_3 = -10
t=2: G_2 = 2 + (0.6)(-10) = -4 #Not used in First Visit
t=1: G_1 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4
t=0: G_0 = 2+ (0.6*1) + (0.6*0.6*2) + (0.6*0.6*0.6-10) = 1.16

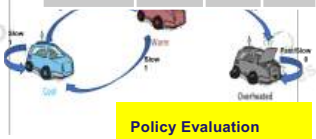
Episode 2: C, S, 1, C, F, 2, W, F, -10, O
G=0
t=2: G_2 = -10
t=1: G_1 = 2 + (0.6)(-10) = -4
t=0: G_0 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4

Episode 1:

t=3: Q(W, F) = -10
t=2: Q(C, F) = -4
t=1: Q(W, S) = -1.4
t=0: Q(C, F) = 1.16

π_1

State	Best Action	P(F)	P(S)
Cool	Slow	0.2	0.8
Warm	Slow	0.2	0.8
Overheated	-	-	-



Policy Evaluation

Episode 3: C, S, 1, C, S, 1, C, F, 2, W, F, -10, O
G=0

t=3: G_3 = -10
t=2: G_2 = 2 + (0.6)(-10) = -4
t=1: G_1 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4
t=0: G_0 = 1+ (0.6*1) + (0.6*0.6*2) + (0.6*0.6*0.6-10) = 0.16

Episode 3:

t=3: Q(W, F) = -10
t=2: Q(C, F) = avg(-4,1.16,-4) = -2.28
t=1: Q(C, S) = -1.4
t=0: Q(C, S) = avg(0.16,-1.4)
= -0.62

π_1

State	Best Action	P(F)	P(S)
Cool	Slow	0.2	0.8
Warm	Slow	0.2	0.8
Overheated	-	-	-

First Visit Policy Evaluation

Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

t=3: G_3 = -10
t=2: G_2 = 2 + (0.6)(-10) = -4 #Not used in First Visit
t=1: G_1 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4
t=0: G_0 = 2+ (0.6*1) + (0.6*0.6*2) + (0.6*0.6*0.6-10) = 1.16

Episode 2: C, S, 1, C, F, 2, W, F, -10, O
G=0
t=2: G_2 = -10
t=1: G_1 = 2 + (0.6)(-10) = -4
t=0: G_0 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4

Episode 1:

t=3: Q(W, F) = -10
t=2: Q(C, F) = -4
t=1: Q(W, S) = -1.4
t=0: Q(C, F) = 1.16

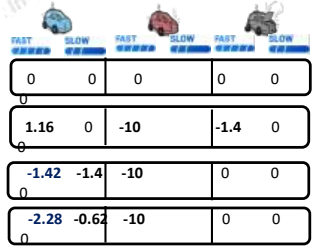
Episode 2:

t=2: Q(W, F) = -10
t=1: Q(C, F) = avg(1.16,-4) = -1.42
t=0: Q(C, S) = -1.4

π_1

State	Best Action	P(F)	P(S)
Cool	Slow	0.2	0.8
Warm	Slow	0.2	0.8
Overheated	-	-	-

Policy PE & PI



Episode 3: C, S, 1, C, S, 1, C, F, 2, W, F, -10, O
G=0

t=3: G_3 = -10
t=2: G_2 = 2 + (0.6)(-10) = -4
t=1: G_1 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4
t=0: G_0 = 1+ (0.6*1) + (0.6*0.6*2) + (0.6*0.6*0.6-10) = 0.16

Episode 3:

t=3: Q(W, F) = -10
t=2: Q(C, F) = avg(-4,1.16,-4) = -2.28
t=1: Q(C, S) = -1.4
t=0: Q(C, S) = avg(0.16,-1.4)
= -0.62

π_2

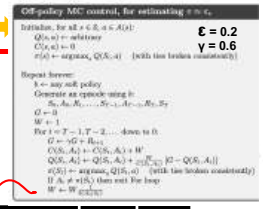
State	Best Action	P(F)	P(S)
Cool	Slow	0.2	0.8
Warm	Slow	0.2	0.8
Overheated	-	-	-

After PE PI no change in π.
Policy converges towards safer actions

1. Given
   - S, A, optionally R(s, a, s`)
2. To find
   - Policy π . This is referred to as **TARGET** policy
3. Solution
   - Sample Episodes
   - Compute estimates of the value by discounted returns
     - Simple average
     - Weighted average
     - Normalized weighted average
4. Idea
   - On Policy :
     - Generate Sample/Experience → Evaluate using Return → Improve
   - Off Policy :
     - Refer to Sample/Experience generated by another reference(**BEHAVIOUR**) policy → Evaluate using Return → Improve

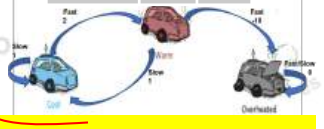
Monte Carlo Method

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956



b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-

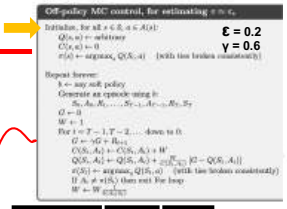


Assume deterministic Target policy. Hence $\pi(A_t | S_t) = 1$
But the behavior policy (b) is Stochastic as defined

Deep Reinforcement Learning

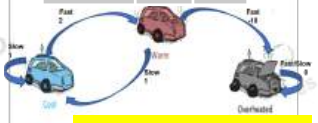
BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Ep	
G	
W	
t	
A_t	
$\pi(S_t)$	



b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-



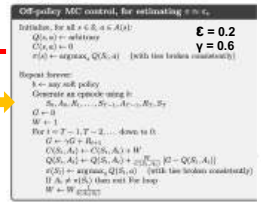
Every Visit Policy Evaluation

Deep Reinforcement Learning

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Ep	1
G	0
W	1
t	
A_t	
$\pi(S_t)$	

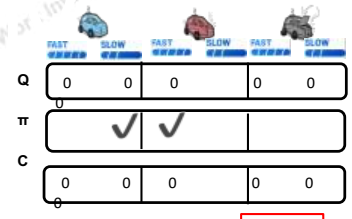
Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0



b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-

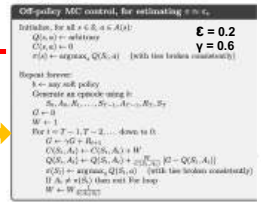
First Visit Policy Evaluation



Ep	1
G	0
W	1
t	3
A_t	F
$\pi(S_t)$	F

Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

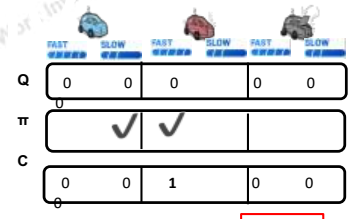
t=3: G_3 = -10



b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-

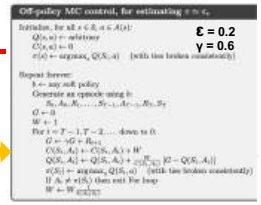
First Visit Policy Evaluation



Ep	1
G	-10
W	1
t	3
A_t	F
$\pi(S_t)$	F

Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

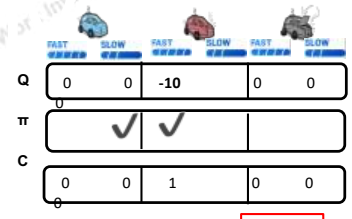
t=3: G_3 = -10



b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-

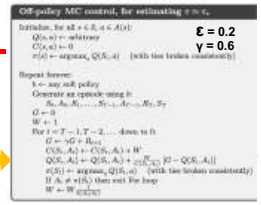
Compute Return & Update the Count for Averaging



Ep	1
G	-10
W	1
t	3
A_t	F
$\pi(S_t)$	F

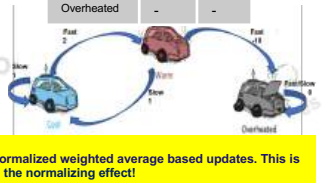
Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

t=3: G_3 = -10

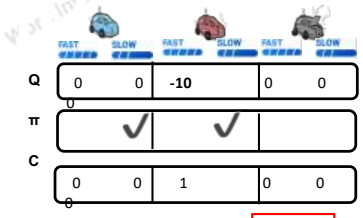


b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-



Compute the action value Q(W,F) using incremental update form adapted to perform normalized weighted average based updates. This is the WEIGHTED IMPORTANCE SAMPLING METHOD, which bounds the variance due to the normalizing effect!

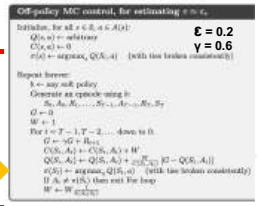


Ep	1
G	-10
W	1
t	3
A_t	F
$\pi(S_t)$	S

Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

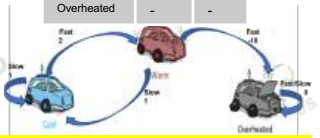
t=3: G_3 = -10

Policy Improvement

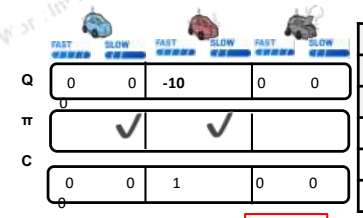


b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-



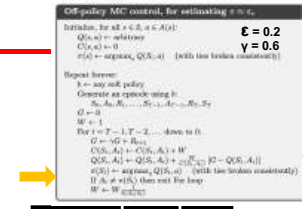
The best action is not the currently encountered action in the sample!



Ep	1
G	-10
W	1
t	3
A_t	F
$\pi(S_t)$	S

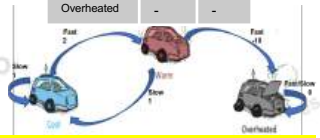
Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

t=3: G_3 = -10

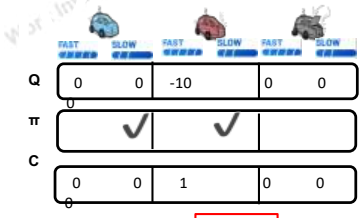


b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-



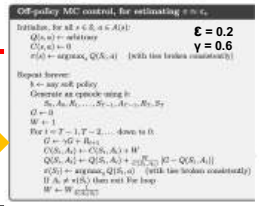
Exits to the next time step without W update!



Ep	1
G	-10
W	1
t	2
A_t	F
$\pi(S_t)$	S

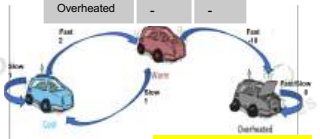
Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

t=3: G_3 = -10
t=2: G_2 = 2 + (0.6)(-10) = -4 #WILL BE used in EVERY Visit

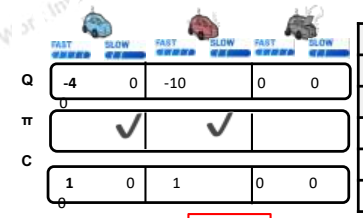


b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-



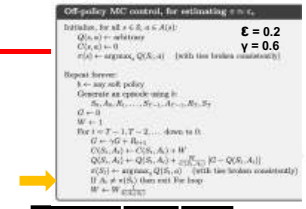
Continue ……..



Ep	1
G	-4
W	1
t	2
A_t	F
$\pi(S_t)$	S

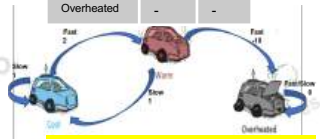
Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

t=3: G_3 = -10
t=2: G_2 = 2 + (0.6)(-10) = -4

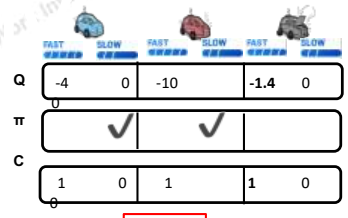


b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-



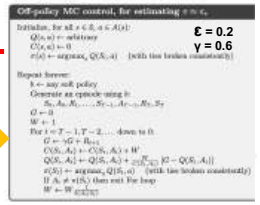
No change in W! Exit to next t



Ep	1
G	-1.4
W	1
t	1
A_t	S
$\pi(S_t)$	S

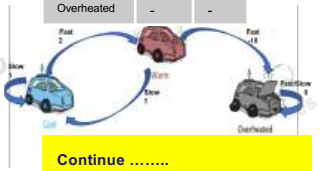
Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

t=3: G_3 = -10
t=2: G_2 = 2 + (0.6)(-10) = -4
t=1: G_1 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4



b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-



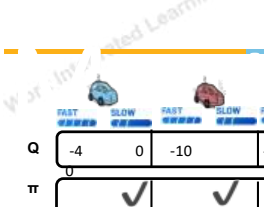
Continue ……..

Ep	1
G	-1.4
W	2.5
t	1
A_t	S
$\pi(S_t)$	S

W = 1/b(S|W)
= 1/0.4
=2.5

Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

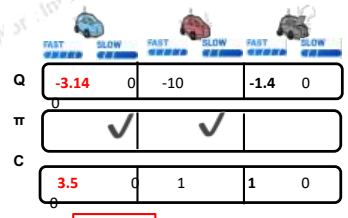
t=3: G_3 = -10
t=2: G_2 = 2 + (0.6)(-10) = -4
t=1: G_1 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4



b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-

At t=1 the best action & current action is same .Hence the Global Weight for WIS is updated

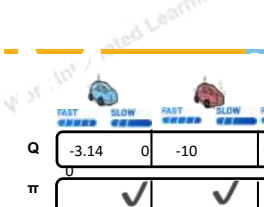


Ep	1
G	-1.4
W	2.5
t	0
A_t	F
$\pi(S_t)$	S

Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O
G=0

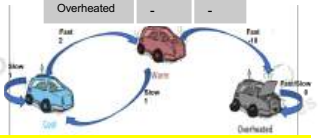
t=3: G_3 = -10
t=2: G_2 = 2 + (0.6)(-10) = -4
t=1: G_1 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4
t=0: G_0 = 2+ (0.6*1) + (0.6*0.6*2) + (0.6*0.6*0.6-10) = 1.16

$$Q(C,F) = -4 + \frac{2.5}{3.5}[1.16 - (-4)] = \sim -3.14$$



b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-



Continuing to next t=0……..
After update observe that the episode terminates

Ep	2
G	0 □-10
W	1
t	2
A_t	F
$\pi(S_t)$	S

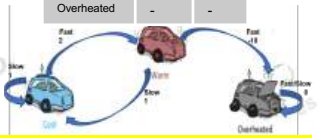
Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O

Episode 2: C, S, 1, C, F, 2, W, F, -10, O
G=0
t=2: G_2 = -10

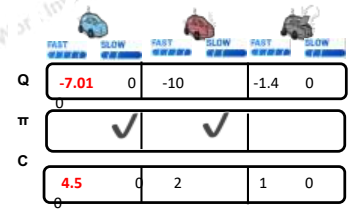
$$Q(W,F) = -10 + \frac{1}{2}[-10 - (-10)] = -10$$

b

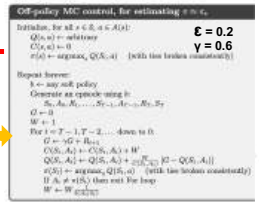
State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-



Generate new episode from "b" . Continue ……
Results of every time-step in Episode 2 is shown

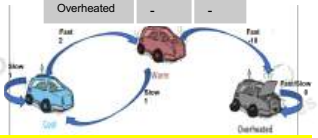


Ep	2
G	-10 -4
W	1
t	1
A_t	F
$\pi(S_t)$	S



b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-

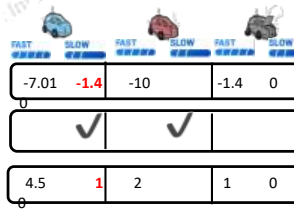


Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O

Episode 2: C, S, 1, C, F, 2, W, F, -10, O
G=0
t=2: G_2 = -10
t=1: G_1 = 2 + (0.6)(-10) = -4

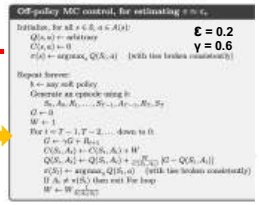
$Q(W,F) = -3.14 + \frac{4.5}{1}[-4 - (-3.14)] = -7.01$

Continue .....
Results of every time-step in Episode 2 is shown



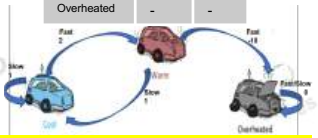
Ep	2
G	-4 -1.4
W	1.67
t	0
A_t	S
$\pi(S_t)$	S

W = 1/b(S|W)
= 1/0.6
=~1.67



b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-



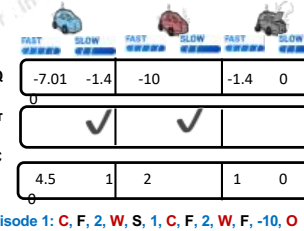
Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O

Episode 2: C, S, 1, C, F, 2, W, F, -10, O
G=0
t=2: G_2 = -10
t=1: G_1 = 2 + (0.6)(-10) = -4
t=0: G_0 = 1+ (0.6*2) + (0.6*0.6*-10) = -1.4

$Q(W,F) = 0 + \frac{1}{1}[-1.4 - (0)] = -1.4$

This completes processing second episode.
Continue next episode

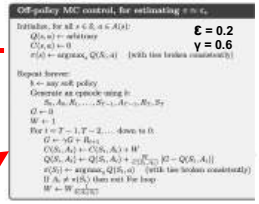
1. Given
   - S, A, optionally R(s, a, s`)
2. To find
   - Policy π . This is referred to as **TARGET** policy
3. Solution
   - Sample Episodes
   - Compute estimates of the value by discounted returns
     - Simple average
     - Weighted average (OIS : Ordinary Importance Sampling)
     - Normalized weighted average (WIS : Weighted Importance Sampling)
4. Idea
   - On Policy :
     - Generate Sample/Experience → Evaluate using Return → Improve
   - Off Policy :
     - Refer to Sample/Experience generated by another reference(**BEHAVIOUR**) policy → Evaluate using Return → Improve



Ep	
G	
W	
t	
A_t	
$\pi(S_t)$	

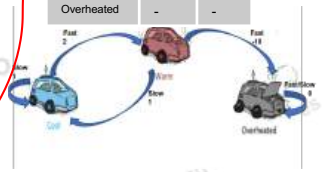
Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O

Episode 2: C, S, 1, C, F, 2, W, F, -10, O



b

State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-



Important Note : The Ordinary Importance Sampling (OIS) is unbiased but will produce high variance. The same given algorithm can be adapted to perform ordinary importance sampling by changing only the count update equation to : C(St,At) → C(St,At) + 1

Q	-7.01	-1.4	-10	-1.4	0
π		✓	✓		
C	4.5	1	2	1	0

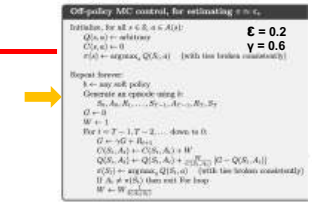
Ep	3
G	0
W	1
t	
A_t	
$\pi(S_t)$	

Episode 1: C, F, 2, W, S, 1, C, F, 2, W, F, -10, O

Episode 2: C, S, 1, C, F, 2, W, F, -10, O

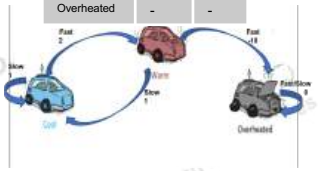
Episode 3: C, S, 1, C, S, 1, C, F, 2, W, F, -10, O

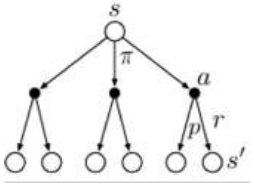
Students are advised to take this episode #3 as an exercise for practice .....



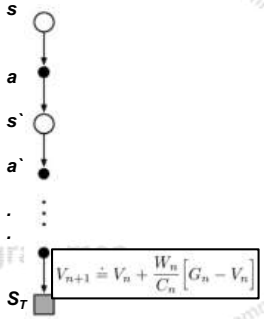
b

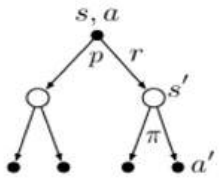
State	P(F)	P(S)
Cool	0.4	0.6
Warm	0.6	0.4
Overheated	-	-



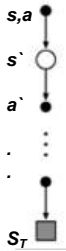


$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$$





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



$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \frac{W}{C(S_t,A_t)}\left[G - Q(S_t,A_t)\right]$$

1. MDP
2. DP
3. MC
4. TD

(Modified) Race Car Example

In a modified Race car, there is a training track where agent is given a known (sometimes incomplete) map and risk factors(dynamics) may not be known in advance. Agent knows exactly where it is but its actions impact on the engine state change is difficult to predict (UNKNOWN MODEL).

With partial knowledge, the agent can compute optimal policies by trying through random experiments (Sampling). Car sometimes may shuts down when its engine get overheated (optional terminal state) otherwise Race is allowed to proceed indefinitely (NO predefined no.of.laps to complete)

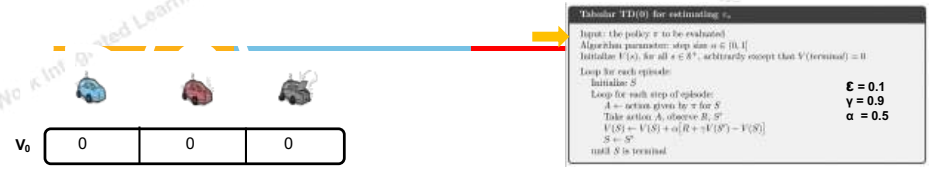
Key Characteristics:

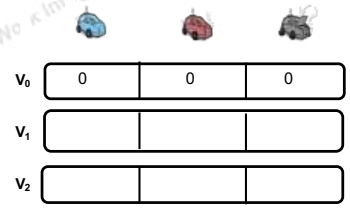
- The agent must act under **uncertainty** — percepts are local, and the agent must reason about hidden dangers
- Assume there are neither fuel constraints nor No.of.Laps to complete!
- At every time step **there is an indication (direct/indirect) of Impact of next** potential engine temperature depending on the current engine state. I**mmediate reward is available**.

1. Given
   - S, A, R(s, a, s`), (optionally)P(a | s)
2. To find
   - Policy π
3. Solution
   - Sample Episodes
   - Compute estimates of the value by discounted next state estimates (bootstrapping)
     - Simple average
     - Weighted average
     - Normalized weighted average
4. Idea
   - On Policy :
     - Generate Sample/Experience Evaluate using bootstrapping Improve
   - Off Policy :
     - Refer to Sample/Experience generated by another reference(behavior) policy Evaluate using bootstrapping Improve

Temporal Difference Learning Methods

Policy Evaluation (PE) Algorithm

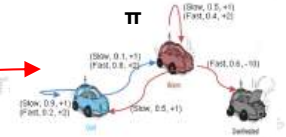




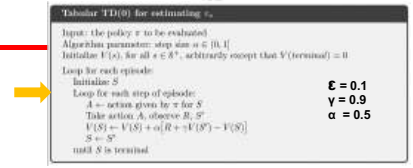
Episode 1: C

Important Note :

- In few textbooks/ running incremental upcoming examples, dynamics are given so **only to *simulate trajectories*** and show updates step by step. TD itself doesn't require knowing the full transition model — it only needs sampled transitions (s,a,r,s') while training from scratch ie., when policy is not known/learnt yet.
- The dynamics are provided for clarity and reproducibility, not because TD requires them. **In practice, TD works model-free, just like Monte Carlo.**



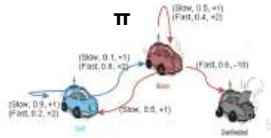
Use case : A driving policy is already learnt which needs to be evaluated for monitoring or certification/approval by safety engineers.

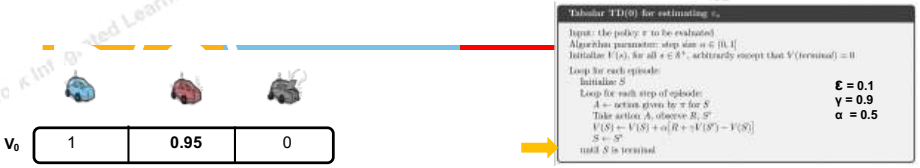


V_0	1	0	0
V_1			
V_2			

Episode 1 C, F, 2, W

Target = R+ γV(W) = 2+(0.9 * 0) = 2
TD Error = Target – Old Estimate = 2-V(C) = 2-0 = 2
Update the value : V(C) = 0 + 0.5(2) = 1

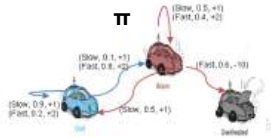




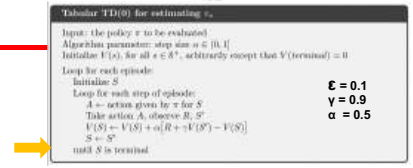
V_0	1	0.95	0
V_1			
V_2			

Episode 1: C, F, 2 W, S, 1, C

V(W) = 0 + 0.5 [1+ (0.9*1)-0] = 0.95



In practice, we can define an episode cutoff (e.g., after 2 or10 steps or when Overheated occurs). If Overheated isn't reached, still the episode an be programmed to end artificially. That's how learning continues.

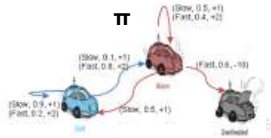


V_0	1	0.95	0
V_1	1.95	2.40	1.40
V_2			

Episode 1: C, F, 2, W, S, 1, C
Episode 2: C, F, 2, C, F, 2, W, S, 1, W

t=0 : V(C) = 1 + 0.5 [2+ (0.9*1)-1] = 1.95
t=1 : V(C) = 1.95 + 0.5 [2+ (0.9*0.95)-1.95] = ~2.4

t=2 : V(W) = 0.95 + 0.5 [1+ (0.9*0.95)-0.95] = ~1.40



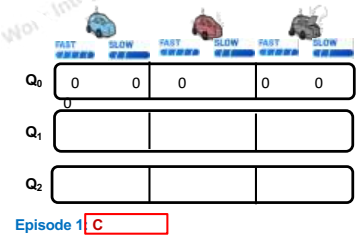
Continued for 1 more episode…..

- **Temporal Difference Learning Methods**

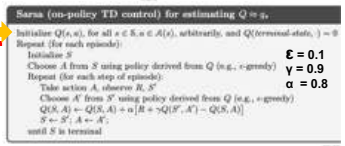
- **On Policy Control Algorithms**

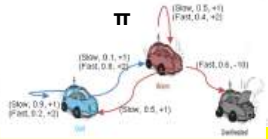
- **Temporal Difference Learning Methods**

- **On Policy Control Algorithms**

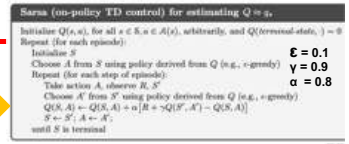


Episode 1: C



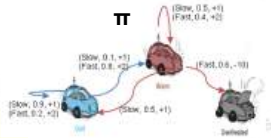


Use case : A driving policy is already learnt but there are few unforeseen changes. Policy needs to be retrained to reflect the actual risk observed via exploration

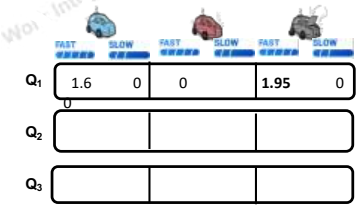


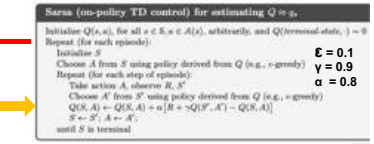
Episode 1: C, F, 2, W, S

Q(C,F) = 0 + 0.8 [2+ (0.9*0)-0] = 1.6



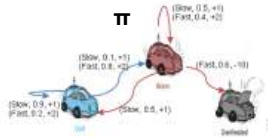
From State "C" & "W", Action "F" & "S" are derived respectively by applying E-greedy on current policy from Q values learnt till now.
In the shown time step : (S,A,R,S,A) = (C,F,2,W,S)



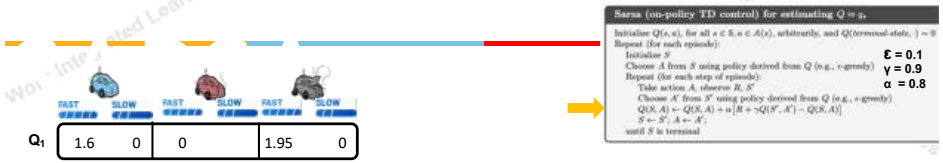


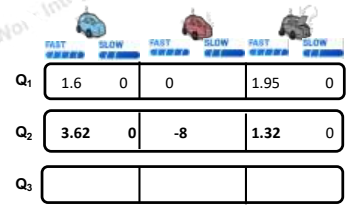
Episode 1: C, F, 2, W, S, 1, C , F

Q(W,S) = 0 + 0.6 [1+ (0.9*1.6)-0] = ~1.95



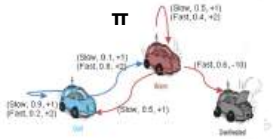
Terminating episode 1 after 2 steps. In practice, we can define an episode cutoff (e.g., after 2 or10 steps or when Overheated occurs). If Overheated isn't reached, still the episode an be programmed to end artificially. That's how learning continues.

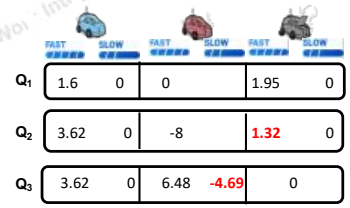




Episode 1: C, F, 2, W, S, 1, C , F
Episode 2: C, F, 2, C, F, 2, W, S, 1, W , S, 1, W , F, -10, O

t=0 : Q(C,F) = 1.6 + 0.8 [2+ (0.9*1)-1.6] = 3.07
t=1 : Q(C,F) = 3.07 + 0.8 [2+ (0.9*1.95)-3.07] = ~3.62
t=2 : Q(W,S) = 1.95 + 0.8 [1+ (0.9*1.95)-1.95] = ~2.59
t=3 : Q(W,S) = 2.59 + 0.8 [1+ (0.9*0)-2.59] = ~1.32
t=4 : Q(W,F) = 0 + 0.8 [-10+ (0.9*0)-0] = ~-8



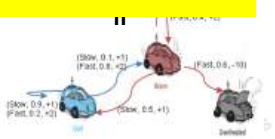


Continued for two more episodes….
Observe the difference in value updates w.r.t Q(W,S) between successive time steps in episodes 3 & 4.
At the worst case on larger learning rate eg., 0.8 the pull is even larger
This large pull is an effect of the VARIANCE produced
Solution to tackle : Expected SARSA

Episode 1: C, F, 2, W, S, 1, C , F
Episode 2: C, F, 2, C, F, 2, W, S, 1, W , S, 1, W , F, -10, O

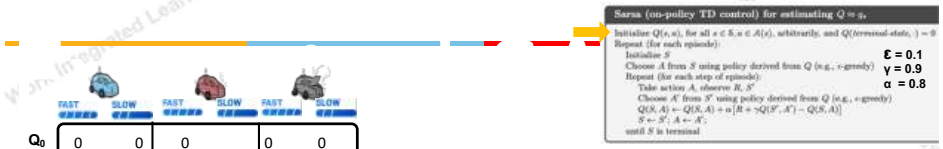
Episode 3: W, S, +1, W, F, +10, O
t=0 : Q(W,S) = 1.32 + 0.8 [1+ (0.9*-8)-1.32] = -4.69
t=1 : Q(W,F) = -8 + 0.8 [10+ (0.9*0)+8] = ~+6.4

Episode 4: W, S, +1, W, F
t=0 : Q(W,S) = -4.69 + 0.8 [1+ (0.9*6.4)+4.69] = ~+4.47

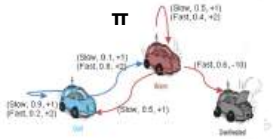


- **Temporal Difference Learning Methods**

- **On Policy Control Algorithms**



Episode 1 C



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., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

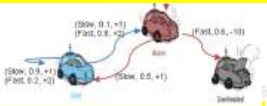
ε = 0.1
γ = 0.9
α = 0.8

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big]$$
$$\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big], \quad (6.9)$$

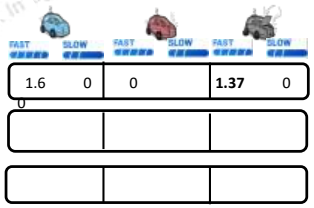
Episode 1: C, F, 2, W, S

Q(C,F) = 0 + 0.8 [2+ (0.9*(0.5*Q(W,S)+0.5*Q(W,F))-0] = 1.6

Important note from TextBook:
In these results Expected Sarsa was used on-policy, but in general it might use a policy different from the target policy to generate behavior, in which case it becomes an off-policy algorithm. For example, suppose the target is the greedy policy while behavior is more exploratory; then Expected Sarsa is exactly Q-learning.



Actions "F" & "S` are derived by applying E-greedy on policy from state "C" &"W", For above highlighted timestep: (S,A,R,S) = (C,F,2,W)
During bootstrapping the expected values are used.
For simplicity in this episode its assumed that all actions are equiprobable for state "W"

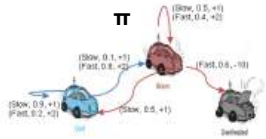


Sarsa (on-policy TD control) for estimating $Q \approx q_*$

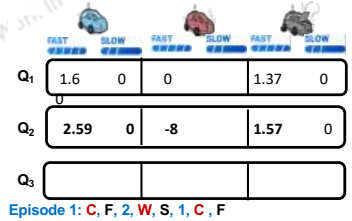
ε = 0.1
γ = 0.9
α = 0.8

Episode 1: C, F, 2, W, S, 1, C, F

Q(W,S) = 0 + 0.8 [1+ (0.9*(0.5*Q(C,S)+0.5*Q(C,F)))-0] = ~1.37



Assumed that all actions are equiprobable from a state in this episode

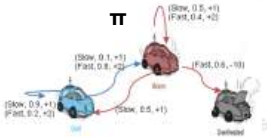


Sarsa (on-policy TD control) for estimating $Q \approx q_*$

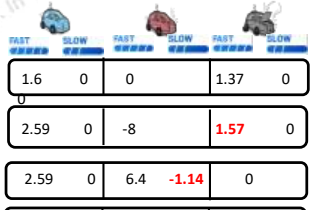
ε = 0.1
γ = 0.9
α = 0.8

Episode 1: C, F, 2, W, S, 1, C , F
Episode 2: C, F, 2, C, F, 2, W, S, 1, W , S, 1, W , F, -10, O

t=0 : Q(C,F) = 1.6+0.8(2+0.9(0.5*1.6+0.5*0)−1.6)=~2.496
t=1 : Q(C,F) = 2.496+0.8(2+0.9(0.5*1.37+0.5*0)−2.496)=~2.59
t=2 : Q(W,S) = 1.37+0.8(1+0.9(0.5*1.37+0.5*0)−1.37)=~1.57
t=3 : Q(W,S) = 1.57 + 0.8 [1+0.9(0.5*1.57+0.5*0)−1.57] = ~1.68
t=4 : Q(W,F) = 0 + 0.8 [-10+ (0.9*0)-0] = ~-8



Continued for one more episode….
Assumed likelihood for all the states "_" for simplicity
:P(S|_) = 0.05, P(F|_) = 0.5



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

ε = 0.1
γ = 0.9
α = 0.8

Episode 1: C, F, 2, W, S, 1, C , F
Episode 2: C, F, 2, C, F, 2, W, S, 1, W , S, 1, W , F, -10, O

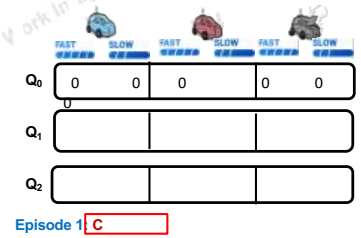
Episode 3: W, S, +1, W, F, +10, O
t=0 : Q(W,S) = 1.57 + 0.8 [1+ (0.9*((-8*0.5) + (1.57*0.5)))-1.57] = -1.14
t=1 : Q(W,F) = -8 + 0.8 [10+ (0.9*0)+8] = ~+6.4

Episode 4: W, S, +1, W, F
t=0 : Q(W,S) = -1.14 + 0.8 [1+ (0.9*((-1.14*0.5)+(6.4*0.5)))+1.14] = ~+2.46

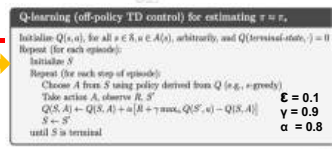
Continued for one more episode….
Assumed likelihood for all the states "_" for simplicity
:P(S|_) = 0.05, P(F|_) = 0.5

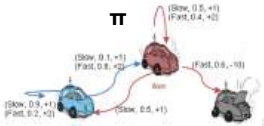
- **Temporal Difference Learning Methods**

- **Off Policy Control Algorithms**

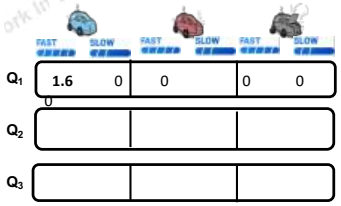


Episode 1: C



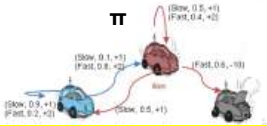


Target Policy : Greedy (MAX) is used to evaluate /update the Q values
Behavior Policy : ε-greedy is used to select actions based on the policy derived from Q - Value

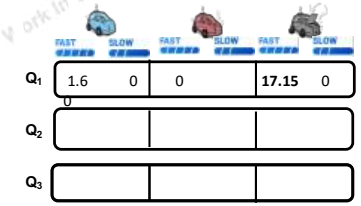


Episode 1: C, F, 2, W, S

Q(C,F) = 0 + 0.8 [2+ (0.9* MAX{Q(W,S),Q(W,F)})-0] = 1.6

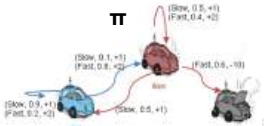


Behavior Policy : ε-greedy is used to select actions based on the policy derived from Q - Value **. Here action "F" for state "C"**
Target Policy : Greedy (MAX) is used to evaluate /update the Q values.

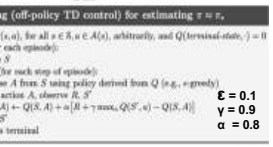


Episode 1: C, F, 2, W, S, 20, C, F

Q(W,S) = 0 + 0.8 [20+ (0.9*(MAX{Q(C,S), Q(C,F)})-0] = ~17.15

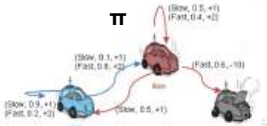


Terminating episode 1 after 2 steps. In practice, we can define an episode cutoff (e.g., after 2 or10 steps or when Overheated occurs). If Overheated isn't reached, still the episode an be programmed to end artificially. That's how learning continues.

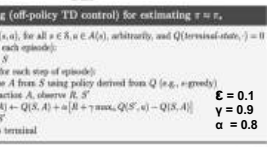


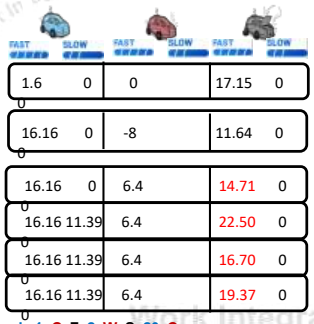
Episode 1: C, F, 2, W, S, 20, C , F
Episode 2: C, F, 2, C, F, 4, W, S, -5, W , S, 1, W , F, -10, O

t=0 : Q(C,F) = 1.6+0.8(2+0.9(MAX{1.6,0})−1.6)=~3.07
t=1 : Q(C,F) = 3.07+0.8(4+0.9(MAX{0,17.15})−3.07)=~16.16
t=2 : Q(W,S) = 17.15+0.8(-5+0.9(MAX{0,17.15})−17.15)=~11.78
t=3 : Q(W,S) = 11.78 + 0.8 [1+0.9(MAX{0,11.78})−11.78] = ~11.64
t=4 : Q(W,F) = 0 + 0.8 [-10+ (0.9*0)-0] = ~-8



Target Policy : Greedy (MAX) is used to evaluate /update the Q values
Behavior Policy : ε-greedy is used to select actions based on the policy derived from Q - Value





Episode 1: C, F, 2, W, S, 20, C
Episode 2: C, F, 2, C, F, 4, W, S, -5, W , S, 1, W , F, -10, O

Episode 3: W, S, +5, W, F, +10, O
Episode 4: C, S, +1, W, S, +10, C
Episode 5: W, S, -5, W
Episode 6: W, S, +5, W

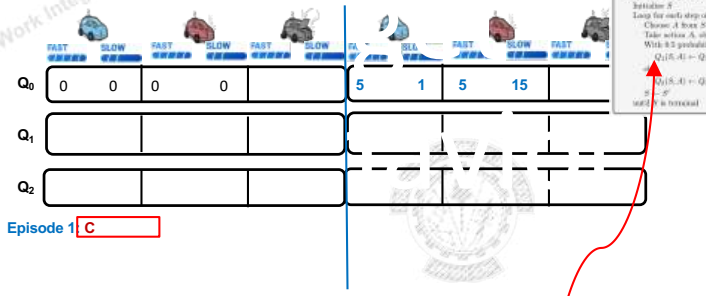
Observe the difference in value updates w.r.t Q(W,S) between successive time steps in episodes 3 to 6.
The true value is approx. Q(W,S) = 15
Due to MAX bias the values are overestimated and this is more pronounced in the presence of NOISE
Sometimes this diverges from true value.
Solution to tackle : Double Q-Learning

*** But in near-deterministic , less noisy environment , Q-learning helps in faster convergence!***

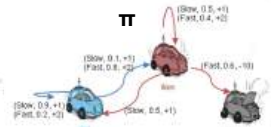
Few more episodes are completed and the results of the update per Q(State, Action) is shown . Students are advised to practice this.

- **Temporal Difference Learning Methods**

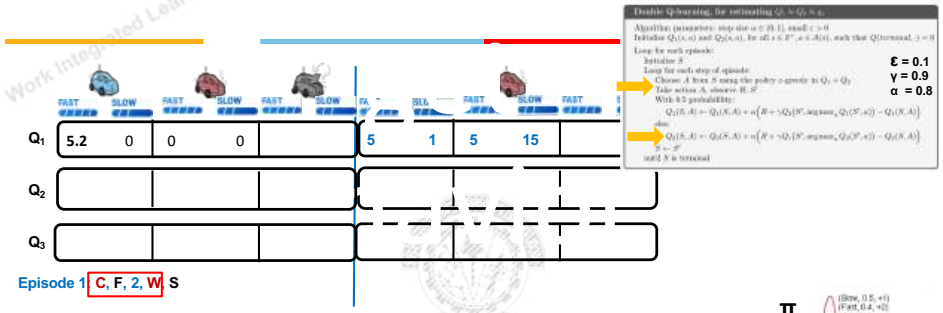
- **Off Policy Control Algorithms**



Episode 1 C

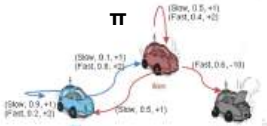


Target Policy : Q2 - Table
Behavior Policy : Q1-Table is available from other sources/experience/logged data. Above algorithm learns to update this as well.

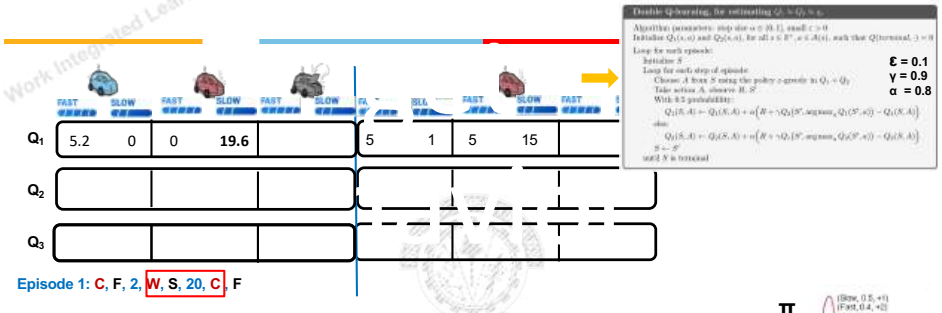


Episode 1: C, F, 2, W, S

Q2(C,F)
= 0 + 0.8 [2+ (0.9* Q1(W, ARGMAX{Q2(W,S),Q2(W,F)}))-0]
= 0 + 0.8 [2+ (0.9* Q1(W, F)-0] *#Here both S & F had same values hence randomly "F" is chosen*
= 0 + 0.8 [2+ (0.9* 5) - 0]
= 5.2



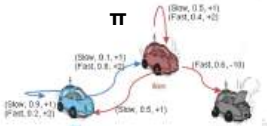
Target Policy : Q2 - Table
Behavior Policy : Q1-Table is available from other sources/experience/logged data. Above algorithm learns to update this as well.



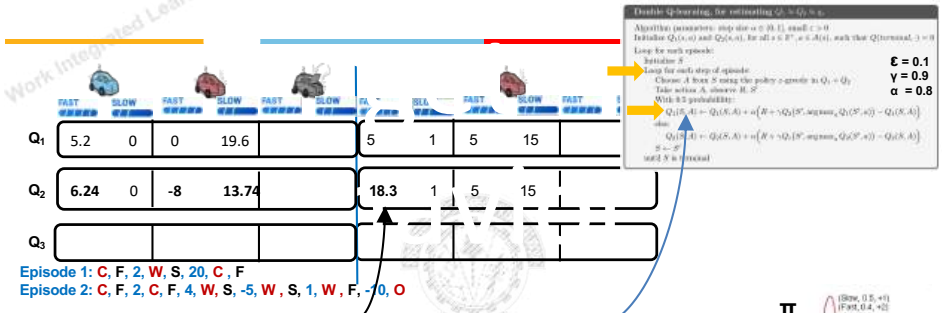
Episode 1: C, F, 2, W, S, 20, C, F

Q2(W,S) = 0 + 0.8 [20+ (0.9*(Q1(C, ARGMAX{Q2(C,S), Q2(C,F)}))-0]
= 0 + 0.8 [20+ (0.9*(Q1(C, F))-0]
= 0 + 0.8 [20+ (0.9*5)-0]

= ~19.6



Terminating episode 1 after 2 steps. In practice, we can define an episode cutoff (e.g., after 2 or10 steps or when Overheated occurs). If Overheated isn't reached, still the episode an be programmed to end artificially. That's how learning continues.

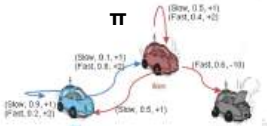


Episode 1: C, F, 2, W, S, 20, C , F
Episode 2: C, F, 2, C, F, 4, W, S, -5, W , S, 1, W , F, -10, O

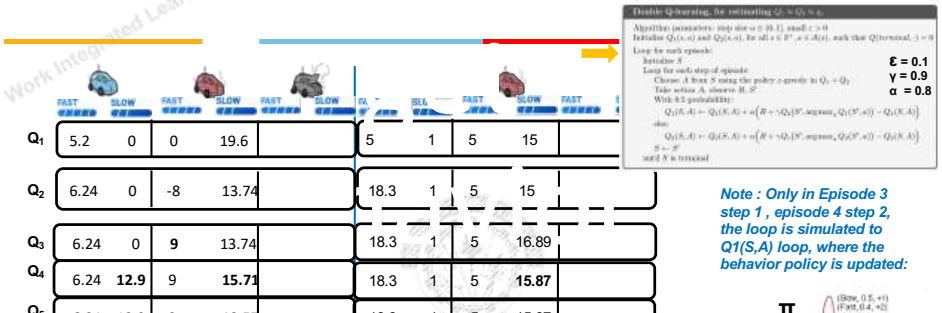
t=0 : Q2(C,F) = 5.2+0.8(2+0.9(Q1(C, ARGMAX{Q2(C,F), Q2(C,S)})) −5.2)=~6.24

For step 2 : The loop goes to Q1(S,A) where the behavior policy is updated:
t=1 : Q1(C,F) = 5+0.8(4+0.9(Q2(W, ARGMAX{ Q1(W,S) , Q1(W,F) }))−5)
= 5+0.8(4+0.9(Q2(W, ARGMAX{ 15, 5}))−5)
= 5+0.8(4+0.9(Q2(W, S))−5)
= 5+0.8(4+0.9(19.6)−5) =~18.3

Only two time step is shown here .
Other time steps updated target policy ie., Q2 table and the result is tabulated above.



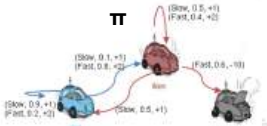
Target Policy : Q2 - Table
Behavior Policy : Q1-Table is available from other sources/experience/logged data. Above algorithm learns to update this as well.



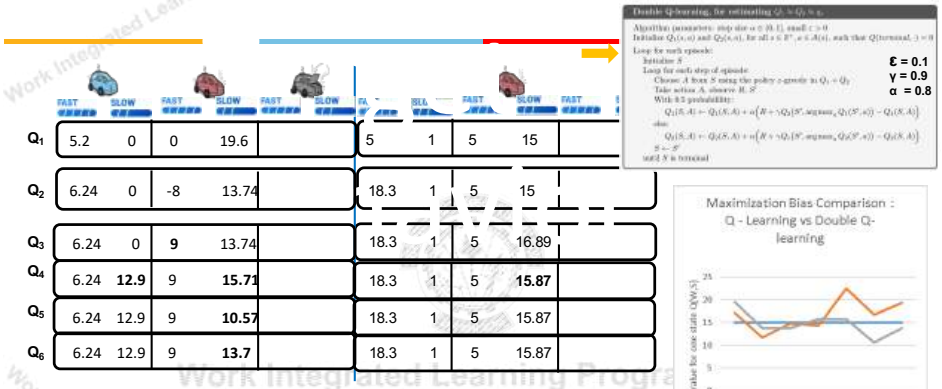
Episode 1: C, F, 2, W, S, 20, C
Episode 2: C, F, 2, C, F, 4, W, S, -5, W , S, 1, W , F, -10, O

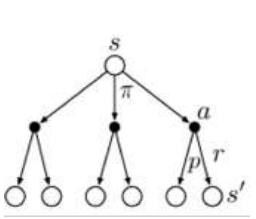
Episode 3: W, S, +5, W, F, +10, O
Episode 4: C, S, +1, W, S, +10, C
Episode 5: W, S, -5, W
Episode 6: W, S, +5, W

Note : Only in Episode 3 step 1 , episode 4 step 2, the loop is simulated to Q1(S,A) loop, where the behavior policy is updated:

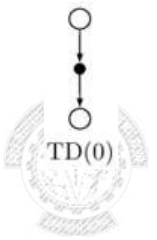


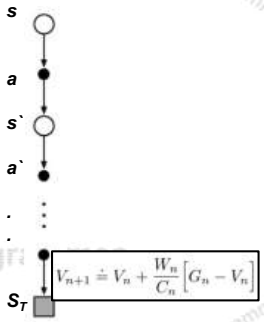
Two more episodes are completed and the results of the update per Q(State, Action) is shown . Students are advised to practice this.





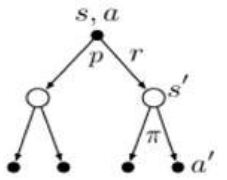
$$V(S) \leftarrow V(S) + \alpha\left[R + \gamma V(S') - V(S)\right]$$





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

$$V_{n+1} \doteq V_n + \frac{W_n}{C_n}\left[G_n - V_n\right]$$

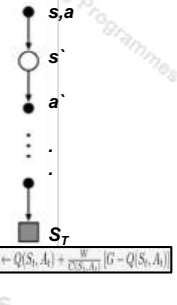


$$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$$

Sarsa

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

Q-learning



$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \frac{W}{C(S_t,A_t)}\left[G - Q(S_t,A_t)\right]$$

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

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\left[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1},A_{t+1}) \mid S_{t+1}] - Q(S_t,A_t)\right]$$
$$\leftarrow Q(S_t,A_t) + \alpha\left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1},a) - Q(S_t,A_t)\right]$$

BITS Pilani

<AIMLCZG512, Deep Reinforcement Learning>
Lecture No. 10,11,12

Agenda for the classes

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

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

Generalizing Across States

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

Example: Pacman

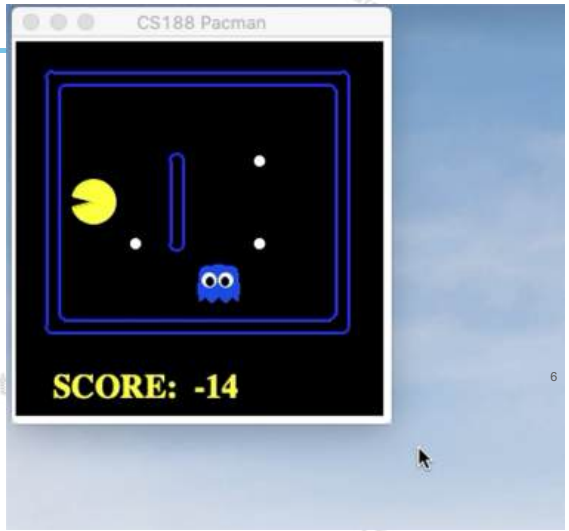
Let's say we discover through experience that this state is bad:

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

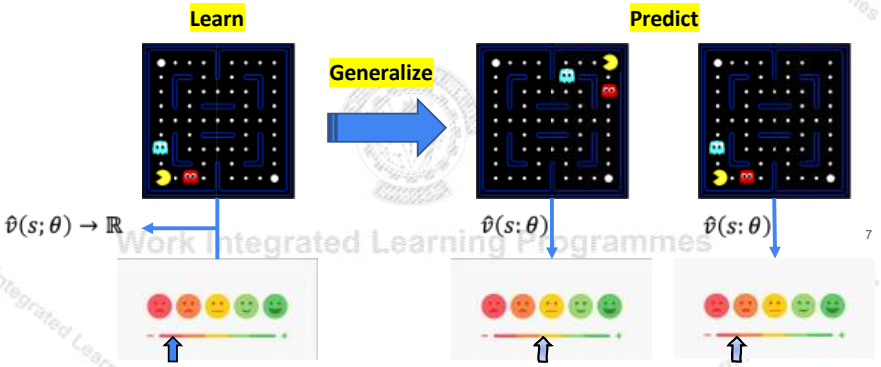
Or even this one!

Demo

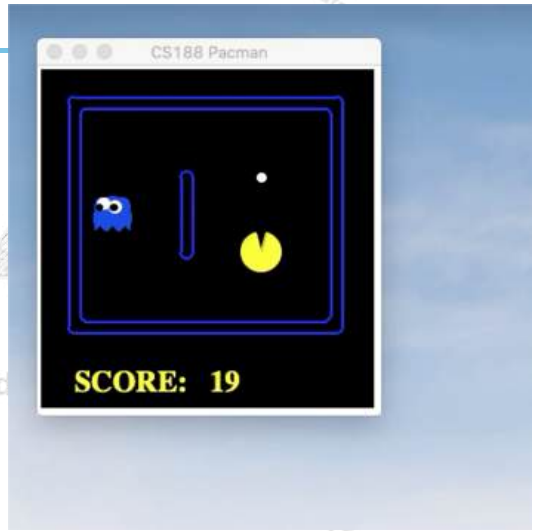
- Naïve Q-learning
- After 50 training episodes



Learn an approximation function

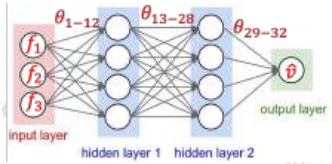


- Q-learning with function approximator



Parameterized function approximator

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

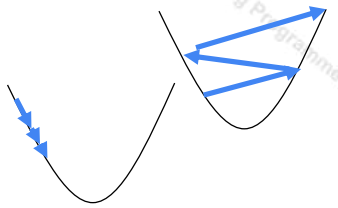


Gradient Descent

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

Gradient Decent

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



SGD for Monte Carlo estimation

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

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

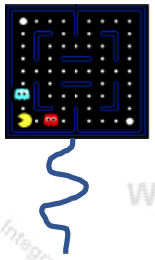
Initialize value-function weights $\mathbf{w}$ as appropriate (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat forever:
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  For $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\left[G_t - \hat{v}(S_t,\mathbf{w})\right]\nabla\hat{v}(S_t,\mathbf{w})$

w are the tunable parameters of the value approximation function

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

Example

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



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

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

10

Learning approximation with bootstrapping

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

Semi-gradient methods

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

Semi-gradient TD(0)

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

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

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

What's the difference from the tabular case?

Example

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

$R = +10$

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

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

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

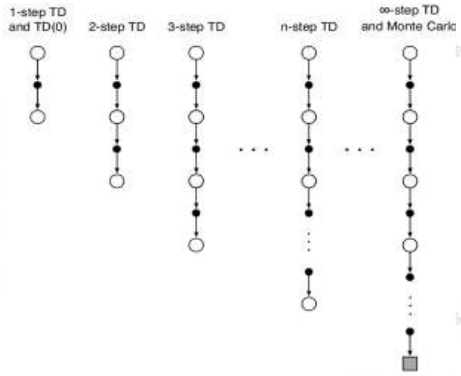
[Review] n-step TD Prediction

One-step return:

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

Two-step return:

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



Ref Section 7.1 of TB

[Review] n-step TD Prediction

One-step return:

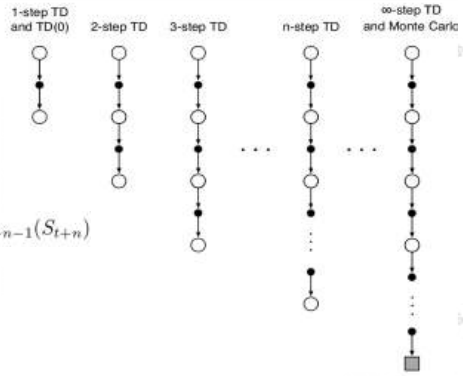
$$G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1})$$

Two-step return:

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

n-step return:

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



Ref Section 7.1 of TB

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

[Review] n-step TD Prediction

One-step return:

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

Two-step return:

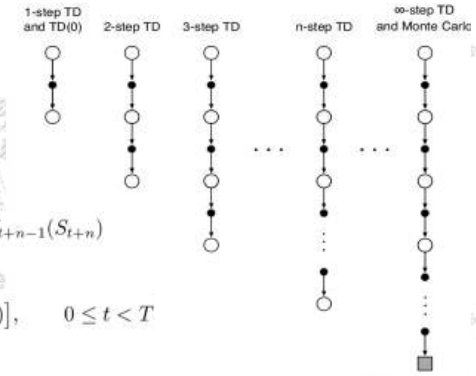
$$G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$

n-step return:

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

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

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



Ref Section 7.1 of TB

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

[Review] n-step TD Prediction

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

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

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

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

n-step return

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

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

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

- Again, only a simple modification over the tabular setting

n-step return

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

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

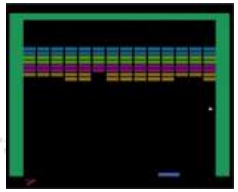
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
    If $t < T$, then:
      Take an action according to $\pi(\cdot|S_t)$
      Observe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

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

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Feature selection

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

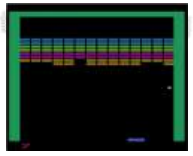


Features are domain depended requiring expert knowledge

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

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?



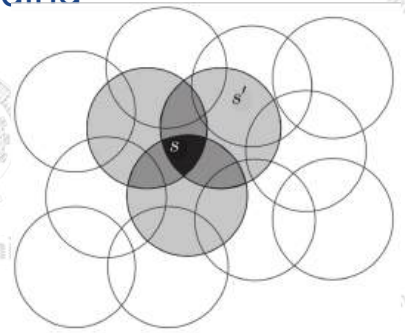
BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Automatic features extraction for linear approximator

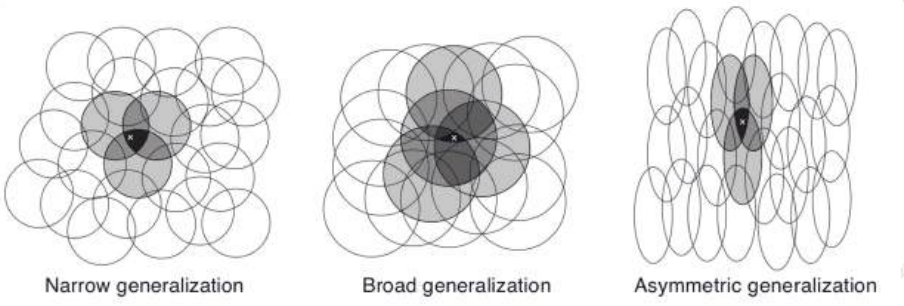
- **Polynomials:** $f_i(s) = \prod_{j=1}^{k} x_j^{c_{i,j}}$
  - where each $c_{i,j}$ is an integer in the set $\{0,1,\dots,n\}$ for an integer $n \geq 0$
  - These features makeup the order-n polynomial basis for dimension k, which contains $(n+1)^k$ different features
- **Fourier Basis:** $f_i(s) = \cos(\pi X^\top c^i)$
  - 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

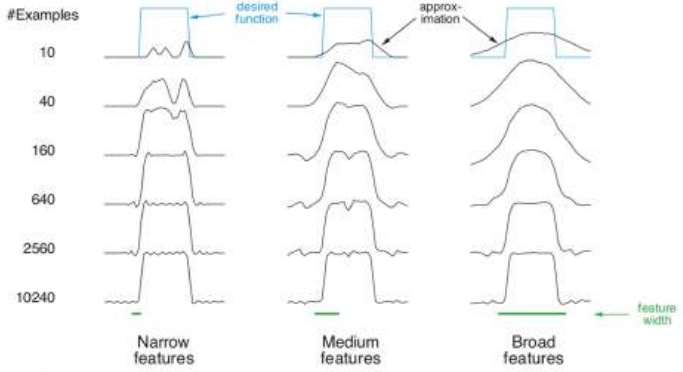
- Natural representation of the state set is continuous
- In 2-d, features corresponding to circles in state space
- Coding of a state:
  - If the state is inside a circle, then the corresponding feature has the value 1
  - otherwise the feature is 0
- Corresponding to each circle is a single weight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



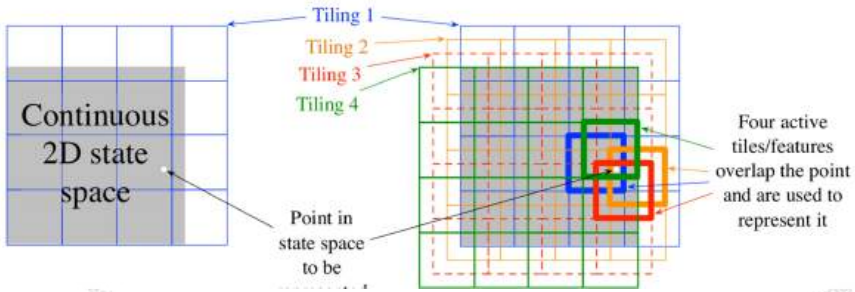
Automatic features extraction for linear approximator - Coarse Coding



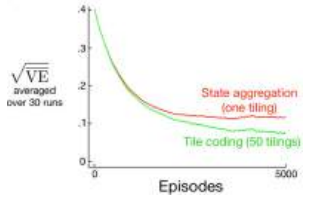
Automatic features extraction for linear approximator - Coarse Coding

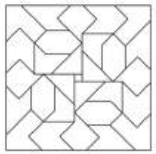


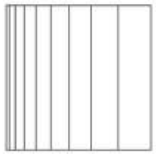
Automatic features extraction for linear approximator - Tile Coding



Automatic features extraction for linear approximator - Tile Coding

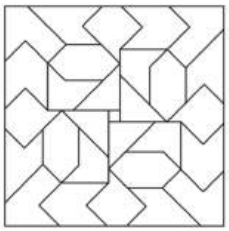


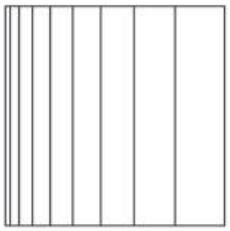

Irregular

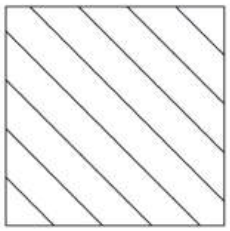

Log stripes


Diagonal stripes

Automatic features extraction for linear approximator - Tile Coding

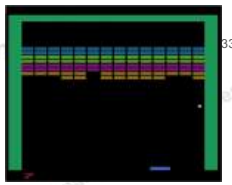

Irregular


Log stripes


Diagonal stripes

Automatic features extraction for linear approximator

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



What did we learn?

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

T-Rex Chrome Dino Game

➔ State Space : Larger in Dimension

➔ Action Space : { 0 - No Ops , 1 – Jump , 2 - Duck }

➔ Choice : Value based function approximation

Require : Sample efficient learning with fast convergence	Vs	Require : Online stable learning
Choice : Off policy – Q learning		Choice : On policy - SARSA

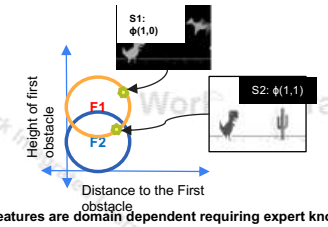
T-Rex Chrome Dino Game
Feature selection

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

➔ State Space : Larger in Dimension

➔ Feature Construction

Lets use coarse coding to construct feature for below sample states S1 & S2:



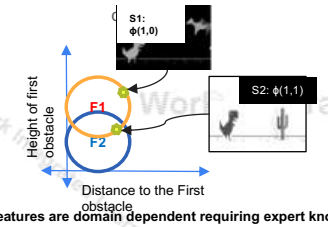
Features are domain dependent requiring expert knowledge

T-Rex Chrome Dino Game
Initialize Hyper parameters

➔ State Space : Larger in Dimension

➔ Lets use the assumed hyper parameters & value initializat
double Q –learning!

➔ By default let the policy select action based on ϵ-greedy. Assume that only actions {0,1} are



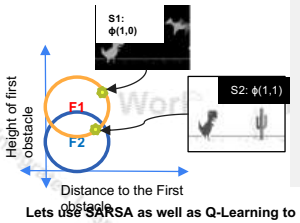
Initial weights:
θ=[1.0,0.5]
θ`=[0.5,1.0]

Discount γ=0.2
Learning rate α=0.9
TD(n) : n=0

Features are domain dependent requiring expert knowledge

T-Rex Chrome Dino Game
Start the Episode

- State Space : Larger in Dimension
- Assume TD(0) 1st time step
- SARSA = (S1, action : 0, reward : 2points, S2, action : 0)



To estimate Q Value its desirable to encode the actions as a part of the feature vector! For simplicity lets append the action value with the state vector designed
Eg.,
S1: ϕ(1,0,0) if no action is performed .
S2: ϕ(1,1,0) if no action is performed
S3: ϕ(1,1,1) if "jump" action is performed
……
The parameters are modified to include the design

Initial weights:
θ=[1.0,0.5,1.0]
θ`=[0.5,1.0,1.0]

Discount γ=0.2
Learning rate α=0.9
TD(n) : n=0

Lets use SARSA as well as Q-Learning to perform one iteration of GD on the parameters θ .

On Policy Approximation for Action Value
SARSA

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

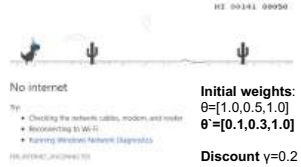
$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\Big[\big(r + \gamma \quad \hat{Q}(s', a'; \theta_i \;) - \hat{Q}(s, a; \theta_i)\big)\nabla_{\theta_i}\hat{Q}(s, a; \theta_i)\Big]$$

Off Policy Approximation for Action Value
Original Q-learning

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

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

T-Rex Chrome Dino Game
Estimate the VE: Value Error



- SARSA = (S1, action : 0, reward : 2points, S2, action : 0)
- Error = Target – Estimate
- Current Estimate = Q(s1,a=0,θ) = 1

Initial weights:
θ=[1.0,0.5,1.0]
θ`=[0.1,0.3,1.0]

Discount γ=0.2
Learning rate α=0.9
TD(n) : n=0
S1: ϕ(1,0,0)
S2: ϕ(1,1,0)

On policy – SARSA

Error = r + γ Q(s2',a'=0,θ) - Q(s1,a=0,θ)

$$= 2 + 0.2\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix} - \begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix}$$

= 2.3-1 = 1.3

Vs

Off policy – Q learning

Error = r + γ **MAX {**

Q(s2',a'=0,θ),

Q(s2',a'=1,θ)

}

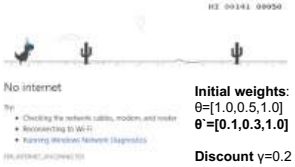
- Q(s1,a=0,θ)

$$= 2 + 0.2\ \mathbf{MAX\{}\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix}, \begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix}\mathbf{\}} - \begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix}$$

= 2 + 0.2 **MAX{**1.5,2.5 **}** – 1

= 2.5-1 = 1.5

T-Rex Chrome Dino Game
Update the Model Parameters



- SARSA = (S1, action : 0, reward : 2points, S2, action : 0)
- Error = Target – Estimate
- Current Estimate = Q(s1,a=0,θ) = 1

Initial weights:
θ=[1.0,0.5,1.0]
θ`=[0.1,0.3,1.0]

Discount γ=0.2
Learning rate α=0.9
TD(n) : n=0
S1: ϕ(1,0,0)
S2: ϕ(1,1,0)

On policy – SARSA

Error = 1.3

Update the parameter using SGD:

$\theta \leftarrow \theta + \alpha \cdot Error \cdot \nabla\theta Q(s1,a=0,\theta)$

$= \begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix} + 0.9*1.3* \begin{bmatrix}1\\0\\0\end{bmatrix} = \begin{bmatrix}\mathbf{2.17}\\0.5\\1.0\end{bmatrix}$

Vs

Off policy – Q learning

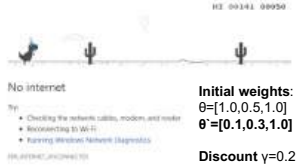
Error = 1.5

Update the parameter using SGD:

$\theta \leftarrow \theta + \alpha \cdot Error \cdot \nabla\theta Q(s1,a=0,\theta)$

$= \begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix} + 0.9*1.5* \begin{bmatrix}1\\0\\0\end{bmatrix} = \begin{bmatrix}\mathbf{2.35}\\0.5\\1.0\end{bmatrix}$

T-Rex Chrome Dino Game
Optimize the Performance



- SARSA = (S1, action : 0, reward : 2points, S2, action : 0)
- Error = Target – Estimate
- Current Estimate = Q(s1,a=0,θ) = 1

Initial weights:
θ=[1.0,0.5,1.0]
θ`=[0.1,0.3,1.0]

Discount γ=0.2
Learning rate α=0.9
TD(n) : n=0
S1: ϕ(1,0,0)
S2: ϕ(1,1,0)

Understanding Maximization bias

Off policy – Q learning

Error = 1.5

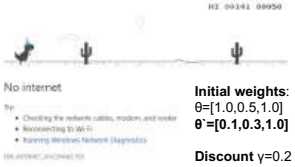
Update the parameter using SGD:

$\theta \leftarrow \theta + \alpha \cdot Error \cdot \nabla\theta Q(s1,a=0,\theta)$

$= \begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix} + 0.9*1.5* \begin{bmatrix}1\\0\\0\end{bmatrix} = \begin{bmatrix}\mathbf{2.35}\\0.5\\1.0\end{bmatrix}$

Observation:
Though it may try to converge faster, it is biased due to MAX function resulting in Maximization bias

T-Rex Chrome Dino Game
Optimize the Performance



- SARSA = (S1, action : 0, reward : 2points, S2, action : 0)
- Error = Target – Estimate
- Current Estimate = Q(s1,a=0,θ) = 1

Initial weights:
θ=[1.0,0.5,1.0]
θ`=[0.1,0.3,1.0]

Discount γ=0.2
Learning rate α=0.9
TD(n) : n=0
S1: ϕ(1,0,0)
S2: ϕ(1,1,0)

Understanding Maximization bias

True value of Q(S2,0) = Q(S2,1) = 2.0

The θ=[1.0,0.5,1.0] produces noisy estimates for :

Q(S2,0) = 1.5 & Q(S2,1) = 2.5, resulting in 2.5 as target value which is an overestimation due to bias.

But given above true value target ~2.1 is expected.

Solution: Decouple the action selection from action evaluation using two independent models!!

Off policy – Q learning

Error = r + γ **MAX {**

Q(s2',a'=0,θ),

Q(s2',a'=1,θ)

}

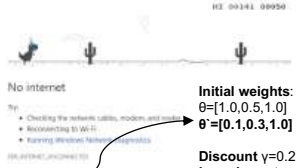
- Q(s1,a=0,θ)

$= 2 + 0.2\ \mathbf{MAX}\{ \begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix}, \begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix} \} - \begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix}$

$= 2 + 0.2\ \mathbf{MAX}\{1.5, 2.5\} - 1$

$= 2.5 - 1 = 1.5$

T-Rex Chrome Dino Game
Optimize the Performance



- SARSA = (S1, action : 0, reward : 2points, S2, action : 0)
- Error = Target – Estimate
- Current Estimate = Q(s1,a=0,θ) = 1

Initial weights:
θ=[1.0,0.5,1.0]
θ`=[0.1,0.3,1.0]

Discount γ=0.2
Learning rate α=0.9
TD(n) : n=0
S1: ϕ(1,0,0)
S2: ϕ(1,1,0)

Understanding Maximization bias

True value of Q(S2,0) = Q(S2,1) = 2.0

The θ=[1.0,0.5,1.0] produces noisy estimates for :

Q(S2,0) = 1.5 & Q(S2,1) = 2.5, resulting in 2.5 as target value which is an overestimation due to bias.

But given above true value target ~2.1 is expected.

Solution: Decouple the action selection from action evaluation using two independent models!!

Off policy – Double Q - Learning

Error = r + γ **MAX {**

Q(s2',a'=0,θ), Q(s2',a'=1,θ)

}

- Q(s1,a=0,θ)

Instead of **MAX** function in above target,

✓ Use θ for action a` selection in the next state S2

✓ Use **θ`** for estimating Q(s2',a',θ`)

✓ Role of model can be reversed

T-Rex Chrome Dino Game Optimize the Performance

- SARSA = (S1, action : 0, reward : 2points, S2, action : 0)
- Error = Target – Estimate
- Current Estimate = Q(s1,a=0,θ) = 1

Effect of the Double Q - Learning

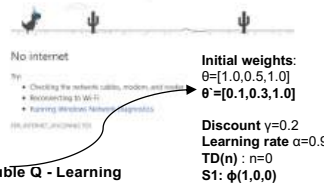
True value of Q(S2,0) = Q(S2,1) = 2.0

Use θ for action a` selection in the next state S2

: **ARGMAX {** Q(s2',a'=0,θ), Q(s2',a'=1,θ) **}**

= **ARGMAX {** $\begin{bmatrix}1\\1\\0\end{bmatrix}\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix}, \begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix}$ **}**

= **ARGMAX {** 1.5, 2.5**}** → Action a1



Initial weights:
θ=[1.0,0.5,1.0]
θ`=[0.1,0.3,1.0]

Discount γ=0.2
Learning rate α=0.9
TD(n) : n=0
S1: ϕ(1,0,0)
S2: ϕ(1,1,0)

Off policy – Double Q - Learning

Error = r + γ **MAX {**

Q(s2',a'=0,θ), Q(s2',a'=1,θ)

}

- Q(s1,a=0,θ)

Instead of **MAX** function in above target,

- ✓ Use θ for action a` selection in the next state S2
- ✓ Use **θ`** for estimating Q(s2',a',θ`)
- ✓ Role of model can be reversed

T-Rex Chrome Dino Game Optimize the Performance

- SARSA = (S1, action : 0, reward : 2points, S2, action : 0)
- Error = Target – Estimate
- Current Estimate = Q(s1,a=0,θ) = 1

Effect of the Double Q - Learning

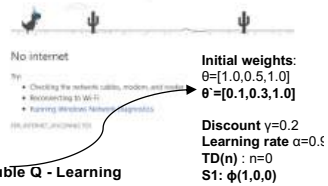
True value of Q(S2,0) = Q(S2,1) = 2.0

Use θ for action a` selection in the next state S2:

Action a1

Use **θ`** for estimating Q(s2',a',θ`)= $\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}0.1\\0.3\\1.0\end{bmatrix}$ = 1.4

2+0.2(1.4) = 2.28 value is a closer target to 1.0 than 2.5!This reduces overestimation problem



Initial weights:
θ=[1.0,0.5,1.0]
θ`=[0.1,0.3,1.0]

Discount γ=0.2
Learning rate α=0.9
TD(n) : n=0
S1: ϕ(1,0,0)
S2: ϕ(1,1,0)

Off policy – Double Q - Learning

Error = r + γ **MAX {**

Q(s2',a'=0,θ), Q(s2',a'=1,θ)

}

- Q(s1,a=0,θ)

Instead of **MAX** function in above target,

- ✓ Use θ for action a` selection in the next state S2
- ✓ Use **θ`** for estimating Q(s2',a',θ`)
- ✓ Role of model can be reversed

T-Rex Chrome Dino Game Optimize the Performance

- SARSA = (S1, action : 0, reward : 2points, S2, action : 0)
- Error = Target – Estimate
- Current Estimate = Q(s1,a=0,θ) = 1

Understanding Maximization bias

True value of Q(S2,0) = Q(S2,1) = 0.5

Use θ for action a` selection in the next state S2:

Action a1

Use **θ`** for estimating Q(s2',a',θ`)= $\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}0.1\\0.3\\1.0\end{bmatrix}$ = 1.4

Initial weights:
θ=[1.0,0.5,1.0]
θ`=[0.1,0.3,1.0]

Discount γ=0.2
Learning rate α=0.9
TD(n) : n=0
S1: ϕ(1,0,0)
S2: ϕ(1,1,0)

Off policy – Double Q - Learning

Error = r + γ Q(s2',a'=argmax(Q(s2',a',**θ**)) ,**θ`**)

- Q(s1,a=0,θ)

= 2 + 0.2 (1.4) – $\begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix}$

= 2.28 – 1

=1.28

Update the parameter using SGD:

θ←θ+α·Error·∇θQ(s1,a=0,θ)

= $\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix}$ + 0.9*1.28* $\begin{bmatrix}1\\0\\0\end{bmatrix}$ = $\begin{bmatrix}2.152\\0.5\\1.0\end{bmatrix}$ = $\begin{bmatrix}θ1\\θ2\\θ3\end{bmatrix}$

T-Rex Chrome Dino Game Optimize the Performance

- SARSA = (S1, action : 0, reward : 2points, S2, action : 0)
- Error = Target – Estimate
- Current Estimate = Q(s1,a=0,θ) = 1

Every consecutive sequence of states in this game is mostly highly correlated!! And if non-linear function approximated is used off policy learnt parameter this affects with diverging updates leading to unstable learning.

Solution : Use Experience Replay to store & use past experience (ie.,SARSA). Instead of SGD use , mini Batch GD by randomly sampling from the buffer. This can be used in DQN as well as DDQN.

Initial weights:
θ=[1.0,0.5,1.0]
θ`=[0.1,0.3,1.0]

Discount γ=0.2
Learning rate α=0.9
TD(n) : n=0
S1: ϕ(1,0,0)
S2: ϕ(1,1,0)

Off policy – Double Q - Learning

Error = r + γ Q(s2',a'=argmax(Q(s2',a',**θ**)) ,**θ`**)

- Q(s1,a=0,θ)

= 2 + 0.2 (1.4) – $\begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix}$

= 2.28 – 1

=1.28

Update the parameter using SGD:

θ←θ+α·Error·∇θQ(s1,a=0,θ)

= $\begin{bmatrix}1.0\\0.5\\1.0\end{bmatrix}$ + 0.9*1.28* $\begin{bmatrix}1\\0\\0\end{bmatrix}$ = $\begin{bmatrix}2.152\\0.5\\1.0\end{bmatrix}$ = $\begin{bmatrix}θ1\\θ2\\θ3\end{bmatrix}$

Deep Reinforcement Learning

BITS Pilani

Prof. Vimal SP
CSIS

BITS Pilani

<AIMLCZG512, Deep Reinforcement Learning>
Lecture No. 12,13

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

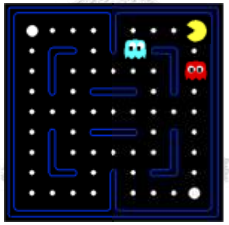
The age of deep learning

- 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

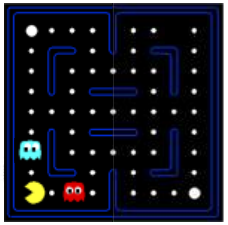
Example: Pacman

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

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

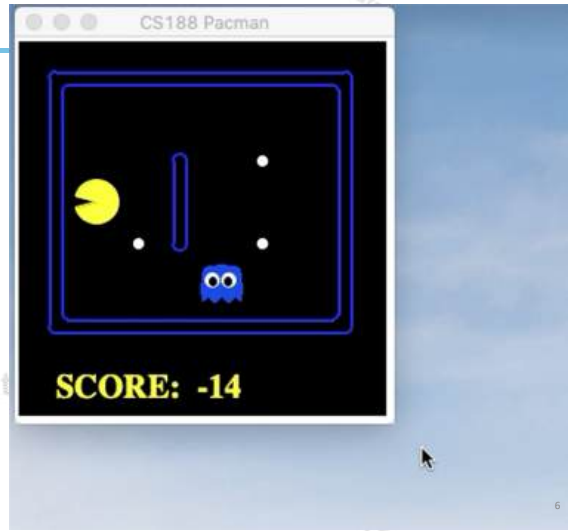


Or even this one!

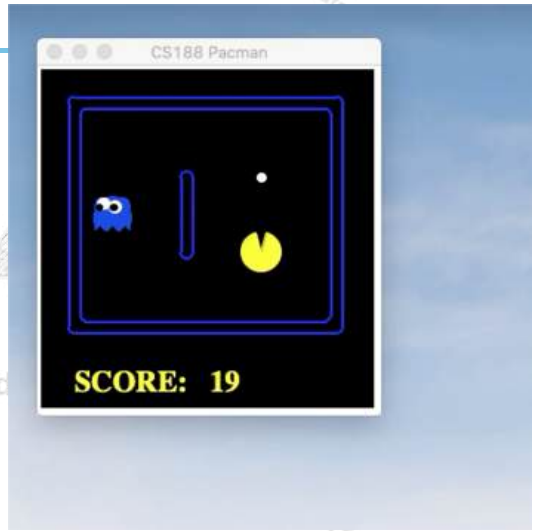


We must generalize our knowledge!

- Naïve Q-learning
- After 50 training episodes



- Generalize Q-learning with function approximator



Generalizing with Deep learning

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

Deep Q learning [Mnih et al. 2015]

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

Original Q-learning

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

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

Deep Q learning [Mnih et al. 2015]

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

Deep Q learning [Mnih et al. 2015]

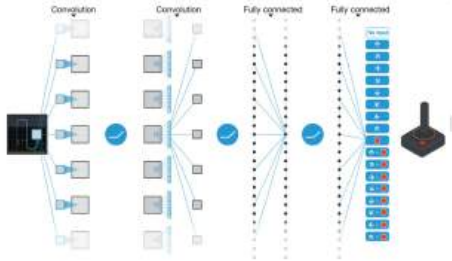
- Learn a function approximator (Q network), $\hat{Q}(s, a; \theta) \approx Q^*(s, a)$
- Value propagation: $Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a')\right]$
  - $\mathcal{E}$ represents the environment (underlying MDP)
- Update $\hat{Q}(s, a; \theta)$ at each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

Deep Q learning [Mnih et al. 2015]

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

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

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



Q-learning with experience replay

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

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

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

Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**

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

Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**

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

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

Q-learning with experience replay

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

For each time step during the episode

Q-learning with experience replay

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

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

Q-learning with experience replay

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

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

Q-learning with experience replay

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

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

Q-learning with experience replay

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

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

Q-learning with experience replay

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

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

Deep Q learning

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

Experience replay

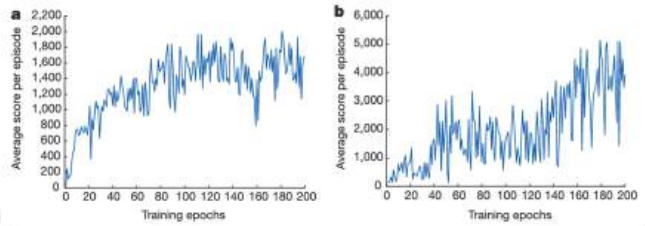
- Utilizing experience replay has several advantages
  - Each step of experience is potentially used in many weight updates, which allows for greater data efficiency
  - Learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples 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

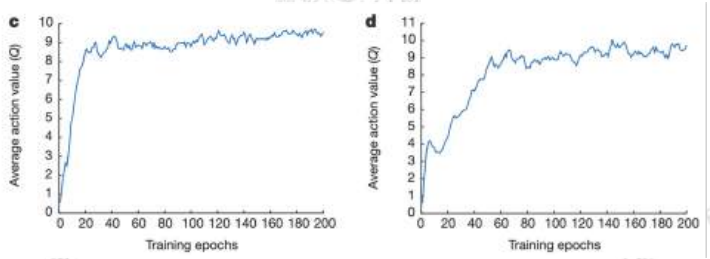
Results: DQN

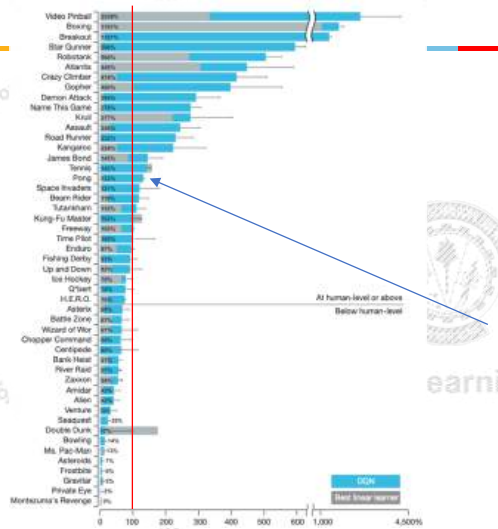
- Each point is the average score achieved per episode after the agent is run with an epsilon-greedy policy ($\varepsilon = 0.05$) for 520k frames on SpaceInvaders (a) and Seaquest (b)



Results: DQN

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





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

Maximization bias

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

Double Deep Q networks (DDQN)

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

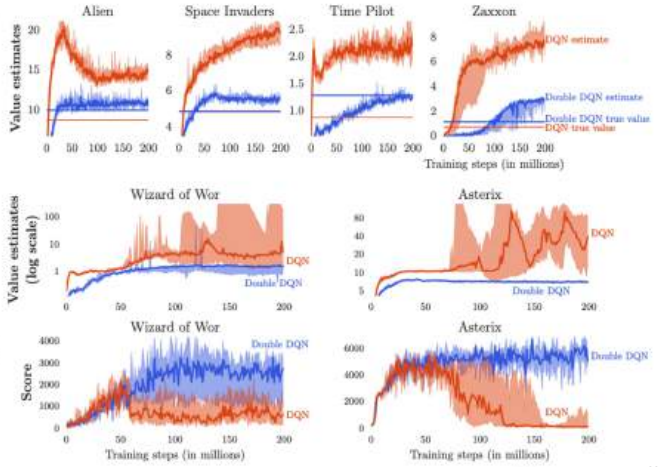
We have two available Q networks. Let's use them for double learning

Double Deep Q networks (DDQN)

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

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

- 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

Deep Reinforcement Learning

BITS Pilani

Prof. Vimal SP
CSIS

<AIMLCZG512, Deep Reinforcement Learning>
Lecture No. 13,14

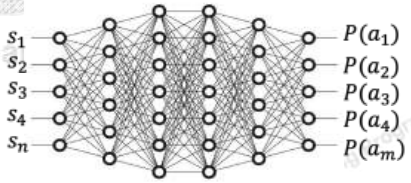
Agenda for the classes

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

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

Notation

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



Improving the policy

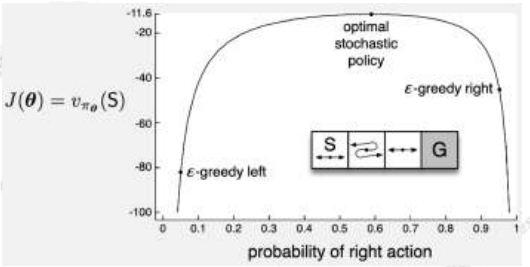
- SGD: $\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)}$
- Where $\widehat{\nabla J(\theta_t)}$ is a stochastic estimate, whose expectation approximates the gradient of the performance measure
- All methods that follow this general schema we call policy gradient methods
  - Might also learn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

Advantages of PG

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

Stochastic policy - Example – Short corridor with switched actions

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



Evaluate the gradient in performance

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

Monte-Carlo Policy Gradient

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

REINFORCE [Williams, 1992]

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

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

REINFORCE [Williams, 1992]

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

REINFORCE [Williams, 1992]

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

REINFORCE [Williams, 1992]

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

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

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

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

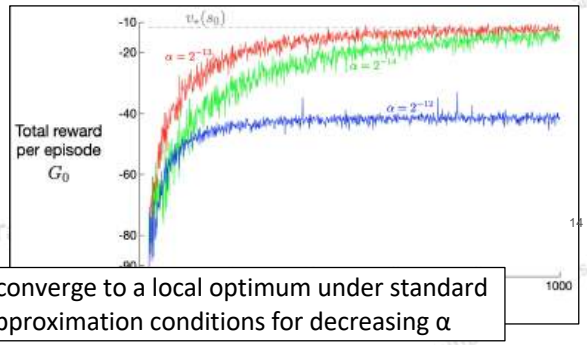
How did we get here from $\frac{\nabla_\theta \pi(A_t|S_t;\theta)}{\pi(A_t|S_t;\theta)}$?

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

REINFORCE [Williams, 1992]

- The crazy corridor domain

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



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

Calibrating REINFORCE

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

PG with baseline

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

PG with baseline

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

PG with baseline

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

REINFORCE with baseline

REINFORCE with Baseline (episodic)

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

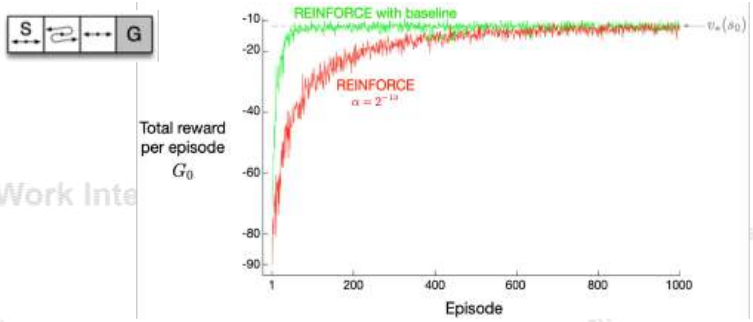
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots,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

Each approximator has its unique learning rate

REINFORCE with baseline

- The crazy corridor domain

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



Policy Gradient

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

Add a critic

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

Critic's duties

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

   adjusting the actor's (policy) gradient

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

Benefits from a critic

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

Actor+critic

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

Advantage function

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

One-step actor-critic

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

δ – TD error

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

One-step Actor–Critic (episodic)

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

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

Keep track of accumulated discount

One-step Actor–Critic (episodic)

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

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

Follow the current policy

One-step Actor–Critic (episodic)

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

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

Compute the TD error

One-step Actor–Critic (episodic)

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

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

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

One-step Actor–Critic (episodic)

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

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

Update the actor with discounting. Early actions matter more.

One-step Actor–Critic (episodic)

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

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

Update accumulated discount and progress to the next state

One-step Actor–Critic (episodic)

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

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

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

One-step Actor–Critic (episodic)

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

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

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

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 $\hat{v}(S',\mathbf{w}) = 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} I\delta\nabla_{\mathbf{w}}\hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla_{\boldsymbol{\theta}}\ln\pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

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

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


Advantage Actor-Critic (A2C)

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

What did we learn?

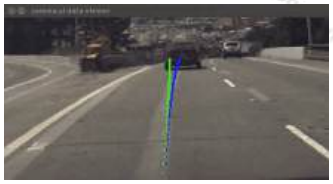
- In many domains it's more efficient to learn the policy directly
  - Instead of deriving one from state or action values
- Applicable for continuous action spaces and stochastic policies
- Approximate the change to the policy that results in the highest increase in the expected return: $\widehat{\nabla_\theta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

What did we learn?

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

Lane Keeping Assistant

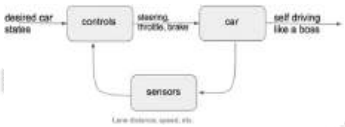
Objective : Policy to keep in the vehicle in line (center of the lane!)



Eg.,

State = function (Distance of Lane edges, Speed of Vehicle, etc.,)

Action = function (Steering Wheel Turn Angle, Amount(Force) of Acceleration/Deceleration, etc.,)



Challenge 1:

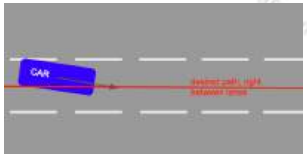
Continuous Large State Space & Action Space

Solution : Policy based function approximation

Source Reference: Example inspired from AI Researcher & Scientist : Eder Santana

(Simplified) Lane Keeping Assistant

Objective : Policy to keep in the vehicle in line (center of the lane!)



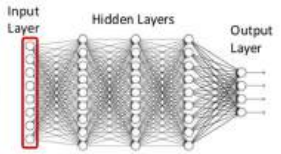
Eg.,

State Ɛ {S0: Left, S1:Center, S2: Right}

Action Ɛ [-1.0, +1.0] (Steer Angle is any value between)

-1=> Full Left , 0=> Straight(no turn) , +1=> Full right

Reward Ɛ $\begin{Bmatrix} +1 & if\ vehicle\ is\ in\ the\ center \\ 0 & if\ its\ in\ left\ or\ right \end{Bmatrix}$



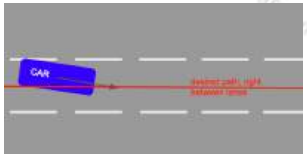
Input Layer : One hot encoding of the states.

For S1 = $[X1\ X2\ X3]^T = [0\ 1\ 0]^T$

Source Reference: Example inspired from AI Researcher & Scientist : Eder Santana

(Simplified) Lane Keeping Assistant

Objective : Policy to keep in the vehicle in line (center of the lane!)



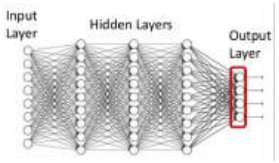
Eg.,

State Ɛ {S0: Left, S1:Center, S2: Right}

Action Ɛ [-1.0, +1.0] (Steer Angle is any value between)

-1=> Full Left , 0=> Straight(no turn) , +1=> Full right

Reward Ɛ $\begin{Bmatrix} +1 & if\ vehicle\ is\ in\ the\ center \\ 0 & if\ its\ in\ left\ or\ right \end{Bmatrix}$



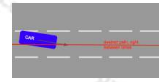
Output Layer : Parameters of the action distribution

Steer Angle: let it be a ~ N (μ, σ)

For simplicity, let's Assume σ = 0.2 (a constant) & only $\mu = \theta^T X$, a linear approximation is a function the network parameter θ

Note : Output can also be designed with P(ai) for discrete larger action space!. Apply soft max in this case

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane

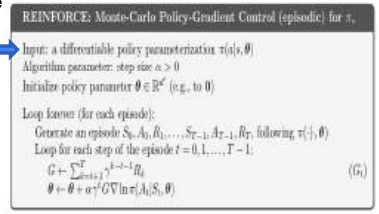
How the Policy Network Parameter θ's are learnt?

> Update by Gradient Ascent

> Refer to the below gradient of the Gaussian

> Initialize the parameter for $\mu = \theta^T X = \theta_1 X_1 + \theta_2 X_2 +$

:

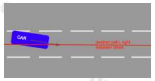

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_θ

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Refer to TB-1 Page section 13.7 → $\nabla \ln \pi(a|s,\boldsymbol{\theta}_\mu) = \frac{\nabla \pi(a|s,\boldsymbol{\theta}_\mu)}{\pi(a|s,\boldsymbol{\theta})} = \frac{1}{\sigma(s,\boldsymbol{\theta})^2}\big(a - \mu(s,\boldsymbol{\theta})\big)\mathbf{x}_\mu(s)$

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane

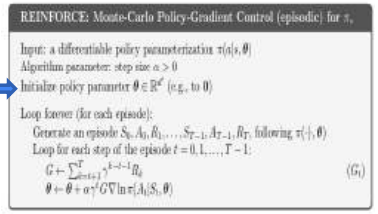
How the Policy Network Parameter θ's are learnt?

> Update by Gradient Ascent

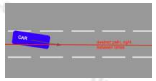
> Refer to the below gradient of the Gaussian

> Initialize the parameter for $\mu = \theta^T X = \theta_1 X_1 + \theta_2 X_2 +$:

Let θ^T = [-0.5 0 0.5]



(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane!)

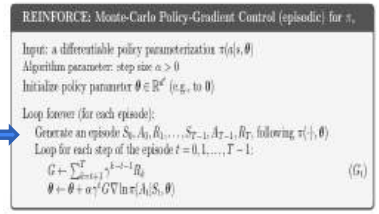
How the Policy Network Parameter θ's are learnt?

> Update by Gradient Ascent

> Refer to the below gradient of the Gaussian

> Initialize the parameter for $\mu = \theta^T X = \theta_1 X_1 + \theta_2 X_2 + \theta_3$

Let θ^T = [-0.5 0 0.5]



Trajectory 1: (S0, a0, r1, S1) = (Left, -0.3, +1, Center)

Start with S0 = $[1\ 0\ 0]^T$

Apply $\mu = \theta^T X$ = -0.5 derived in Gaussian to derive Action from above policy

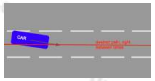
a0= -0.3 (Steer left slightly)

Next State: S1 = $[0\ 1\ 0]^T$

Reward = +1

Refer to TB-1 Page section 13.7 → $\nabla \ln \pi(a|s,\boldsymbol{\theta}_\mu) = \frac{\nabla \pi(a|s,\boldsymbol{\theta}_\mu)}{\pi(a|s,\boldsymbol{\theta})} = \frac{1}{\sigma(s,\boldsymbol{\theta})^2}(a - \mu(s,\boldsymbol{\theta}))\mathbf{x}_\mu(s)$

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane!)

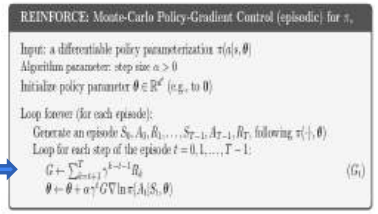
How the Policy Network Parameter θ's are learnt?

> Update by Gradient Ascent

> Refer to the below gradient of the Gaussian

> Initialize the parameter for $\mu = \theta^T X = \theta_1 X_1 + \theta_2 X_2 + \theta_3 X_3$:

Let θ^T = [-0.5 0 0.5]



Trajectory 1: (S0, a0, r1, S1) = (Left, -0.3, +1, Center)

Start with S0 = $[1\ 0\ 0]^T$

Apply $\mu = \theta^T X$ = -0.5 derived in Gaussian to derive Action from above policy

a0= -0.3 (Steer left slightly)

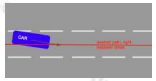
Next State: S1 = $[0\ 1\ 0]^T$

Reward = +1

(Assume episode terminates) → G = +1 with γ = 1[No discount]

Refer to TB-1 Page section 13.7 → $\nabla \ln \pi(a|s,\boldsymbol{\theta}_\mu) = \frac{\nabla \pi(a|s,\boldsymbol{\theta}_\mu)}{\pi(a|s,\boldsymbol{\theta})} = \frac{1}{\sigma(s,\boldsymbol{\theta})^2}(a - \mu(s,\boldsymbol{\theta}))\mathbf{x}_\mu(s)$

(Simplified) Lane Keeping Assistant



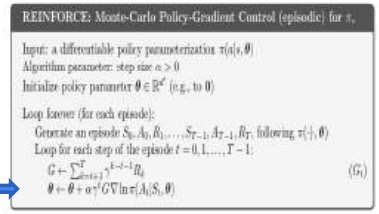
Objective : Policy to keep in the vehicle in line (center of the lane!)

How the Policy Network Parameter θ's are learnt?

> Compute the gradient, return

> Update θ with eg., α = 0.01

θ_{old}^T = [-0.5 0 0.5]



Trajectory 1:

(S0, a0, r1, S1) = (Left, -0.3, +1, Center)

G = +1

$\theta_{new}^T = \theta_{old}^T + \alpha\ \mathbf{G}\ \nabla \ln \pi\ (A_t|S_t, \theta_{old})$

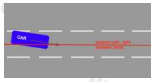
$= [-0.5\ 0\ 0.5]^T + 0.01\ \mathbf{(1)}\ (\frac{(-0.3-(-0.5))}{(0.2)^2})\ [1\ 0\ 0]^T$

$= [-0.5\ 0\ 0.5]T + 0.01\ \mathbf{(5)}\ [1\ 0\ 0]T$

$= [\mathbf{-0.45}\ 0\ 0.5]$

Refer to TB-1 Page section 13.7 → $\nabla \ln \pi(a|s,\boldsymbol{\theta}_\mu) = \frac{\nabla \pi(a|s,\boldsymbol{\theta}_\mu)}{\pi(a|s,\boldsymbol{\theta})} = \frac{1}{\sigma(s,\boldsymbol{\theta})^2}(a - \mu(s,\boldsymbol{\theta}))\mathbf{x}_\mu(s)$

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane!)

With θ_{old}^T = [-0.45 0 0.5],

after repeating the exercise for one more say ,

Trajectory 2: (S0, a0, r1, S2) = (Left, -0.9, 0, Right)

Update is as follows :

Trajectory 2: *(assume that the episode terminates after S2)*

(S0, a0, r1, S2) = (Left, -0.9, 0, Right)

G = 0

$\theta_{new}^T = \theta_{old}^T + \alpha$ **G** $\nabla \ln \pi (A_t|S_t, \theta_{old})$

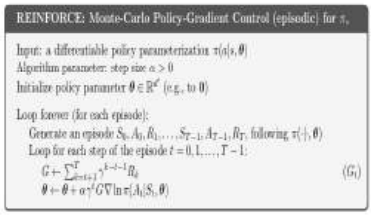
$= [-0.45\ 0\ 0.5]^T + 0.01$ **(0)** $(\frac{(-0.9-(-0.45))}{(0.2)^2})\ [1\ 0\ 0]^T$

$= [-0.45\ 0\ 0.5]T + 0.01$ **(0)** $[1\ 0\ 0]T$

$= [$**-0.45** $0\ 0.5]$

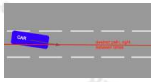
~~$= [0.5\ 0\ 0.5]T + 0.01$ **(11.25)** $[1\ 0\ 0]T$~~

~~$=$ **[0.3875** $0\ 0.5]$~~



Observation :
Gradient update vanishes or tend to vary due to full dependence on reward
Solution : Induce bias using baseline. Use value network w

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane!)

How the Policy Network Parameter θ's are learnt?

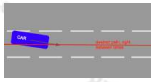
> Compute the gradient, return

> Update θ with eg., α_θ = 0.01

θ_{old}^T = [-0.5 0 0.5]

Typically the value network w will also be learnt as per the algorithm.
For simplicity assume that it estimated values for S0:V(left)=0.4, V(center)=0.9, V(right)=0.2

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane!)

How the Policy Network Parameter θ's are learnt?

> Compute the gradient, return

> Update θ with eg., α_θ = 0.01

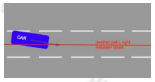
θ_{old}^T = [-0.5 0 0.5]

Typically the value network w will also be learnt as per the algorithm.
For simplicity assume that it estimated values for S0:V(left)=0.4, V(center)=0.9, V(right)=0.2

Trajectory 1:

(S0, a0, r1, S1) = (Left, -0.3, +1, Center)

(Simplified) Lane Keeping Assistant



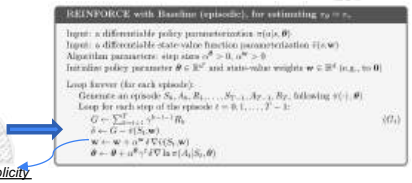
Objective : Policy to keep in the vehicle in line (center of the lane!)

How the Policy Network Parameter θ's are learnt?

> Compute the gradient, return

> Update θ with eg., α_θ = 0.01

θ_{old}^T = [-0.5 0 0.5]



Typically the value network w will also be learnt as per the algorithm. For simplicity assume that it estimated values for S0:V(left)=0.4, V(center)=0.9, V(right)=0.2

Trajectory 1:

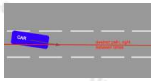
(S0, a0, r1, S1) = (Left, -0.3, +1, Center)

G = +1

$\theta_{new}^T = \theta_{old}^T + \alpha$ **(G-V(S$_t$))** $\nabla \ln \pi (A_t|S_t, \theta_{old})$

$= [-0.5\ 0\ 0.5]^T + 0.01$ **(1-0.4)** $(\frac{(-0.3-(-0.5))}{(0.2)^2})\ [1\ 0\ 0]^T$

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane!)

With θ_{old}^T = [-0.47 0 0.5],

after repeating the exercise for one more say ,

Trajectory 2: (S0, a0, r1, S2) = (Left, -0.9, 0, Right)

Update is as follows :

Trajectory 2: *(assume that the episode terminates after S2)*

(S0, a0, r1, S2) = (Left, -0.9, 0, Right)

G = 0

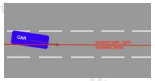
$\theta_{new}^T = \theta_{old}^T + \alpha\, \mathbf{G}\, \nabla \ln \pi\,(A_t|S_t, \theta_{old})$

$= [-0.47\ 0\ 0.5]^T + 0.01\ \mathbf{(0\text{-}0.4)}\ (\frac{(-0.9-(-0.45))}{(0.2)^2})\ [1\ 0\ 0]^T$

$= [-0.47\ 0\ 0.5]T + 0.01\ \mathbf{(4.3)}\ [1\ 0\ 0]T$

$= [\mathbf{-0.427}\ 0\ 0.5]$

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane!)

With θ_{old}^T = [-0.47 0 0.5],

after repeating the exercise for one more say ,

Trajectory 2: (S0, a0, r1, S2) = (Left, -0.9, 0, Right)

Update is as follows :

Trajectory 2: *(assume that the episode terminates after S2)*

(S0, a0, r1, S2) = (Left, -0.9, 0, Right)

G = 0

$\theta_{new}^T = \theta_{old}^T + \alpha\, \mathbf{G}\, \nabla \ln \pi\,(A_t|S_t, \theta_{old})$

$= [-0.47\ 0\ 0.5]^T + 0.01\ \mathbf{(0\text{-}0.4)}\ (\frac{(-0.9-(-0.45))}{(0.2)^2})\ [1\ 0\ 0]^T$

$= [-0.47\ 0\ 0.5]T + 0.01\ \mathbf{(4.3)}\ [1\ 0\ 0]T$

$= [\mathbf{-0.427}\ 0\ 0.5]$

Observation :

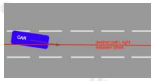
> Relatively less variance
> But Monte Carlo based updates may not be suited for lengthier episodes/continuing tasks
> Need online learning with faster updates

Solution : Use Temporal Difference .

Use value network w as critic
Policy net θ as actor for policy improvement
Gradient Descent with TD error based advantage function

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane!)

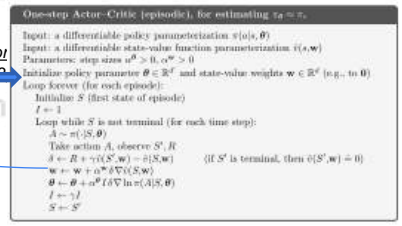
How the Policy Network Parameter θ's are learnt?

> Compute the gradient, return

> Update θ with eg., α_θ = 0.01

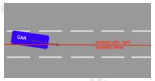
θ_{old}^T = [-0.5 0 0.5]

Typically the value network w will also be learnt as per the algo
For simplicity assume that it estimated values for S0:V(left)=0.3
V(center)=0.9, V(right)=0.2



BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane!)

How the Policy Network Parameter θ's are learnt?

> Compute the gradient, return

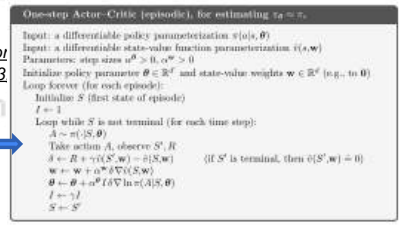
> Update θ with eg., α_θ = 0.01

θ_{old}^T = [-0.5 0 0.5]

Typically the value network w will also be learnt as per the algo
For simplicity assume that it estimated values for S0:V(left)=0.3
V(center)=0.9, V(right)=0.2

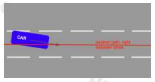
Trajectory 1:

(S0, a0, r1, S1) = (Left, -0.3, +1, Center)



BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane!)

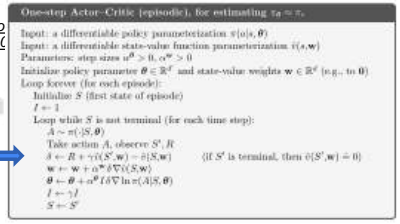
How the Policy Network Parameter θ's are learnt?

> Compute the gradient, return

> Update θ with eg., $\alpha_\theta = 0.01$

θ_{old}^T = [-0.5 0 0.5]

Typically the value network w will also be learnt as per the algorithm. Fo simplicity assume that it estimated values for S0:V(left)=0.3, V(center)=(V(right)=0.2



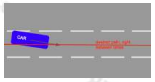
Trajectory 1:

(S0, a0, r1, S1) = (Left, -0.3, +1, Center)

$\theta_{new}^T = \theta_{old}^T + \alpha$ **(r + V(S_{t+1})-V(S_t))** $\nabla \ln \pi (A_t|S_t, \theta_{old})$

$= [-0.5\ 0\ 0.5]^T + 0.01$ **(1+0.9-0.3)** $(\frac{(-0.3-(-0.5))}{(0.2)^2})$ $[1\ 0\ 0]^T$

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane!)

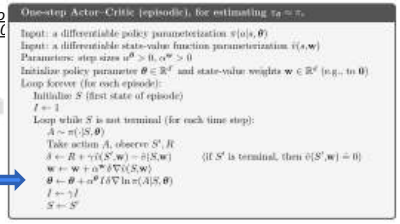
How the Policy Network Parameter θ's are learnt?

> Compute the gradient, return

> Update θ with eg., $\alpha_\theta = 0.01$

θ_{old}^T = [-0.5 0 0.5]

Typically the value network w will also be learnt as per the algorithm. Fo simplicity assume that it estimated values for S0:V(left)=0.3, V(center)=(V(right)=0.2



Trajectory 1:

(S0, a0, r1, S1) = (Left, -0.3, +1, Center)

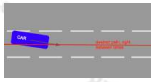
$\theta_{new}^T = \theta_{old}^T + \alpha$ **(r + V(S_{t+1})-V(S_t))** $\nabla \ln \pi (A_t|S_t, \theta_{old})$

$= [-0.5\ 0\ 0.5]^T + 0.01$ **(1+0.9-0.3)** $(\frac{(-0.3-(-0.5))}{(0.2)^2})$ $[1\ 0\ 0]^T$

$= [-0.5\ 0\ 0.5]T + 0.01$ **(8)** $[1\ 0\ 0]T$

$=$ **[-0.42** $0\ 0.5]$

(Simplified) Lane Keeping Assistant



Objective : Policy to keep in the vehicle in line (center of the lane!)

With θ_{old}^T = [-0.42 0 0.5],

after repeating the exercise for one more say ,

Trajectory 2: (S0, a0, r1, S2) = (Left, -0.9, 0, Right)

Update is as follows :

Trajectory 2: *(assume that the episode terminates after S2)*

(S0, a0, r1, S2) = (Left, -0.9, 0, Right)

This iteration is added only for additional practice!

G = 0

$\theta_{new}^T = \theta_{old}^T + \alpha$ **(r + V(S_{t+1})-V(S_t))** $\nabla \ln \pi (A_t|S_t, \theta_{old})$

$= [-0.42\ 0\ 0.5]^T + 0.01$ **(0+0.2-0.3)** $(\frac{(-0.9-(-0.42))}{(0.2)^2})$ $[1\ 0\ 0]^T$

$= [-0.42\ 0\ 0.5]T + 0.01$ **(1.2)** $[1\ 0\ 0]T$

$=$ **[-0.408** $0\ 0.5]$

BITS Pilani

<AIMLCZG512, Deep Reinforcement Learning>
Lecture No. 14,15

Agenda for the classes

- ➔ Introduction
- ➔ Upper-Confidence-bound Action Selection
- ➔ Monte-Carlo Tree Search
- ➔ AlphaGo Zero
- ➔ MuZero, PlaNet

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

Model Based Algorithms

- Learn the model of an environment's transition dynamics or make use of a known dynamic model
- Once an agent has a model of the environment, P(s'|s,a), it can "imagine" what will happen in the future by predicting the trajectory for a few time steps.
- If the environment is in state s, an agent can estimate how the state will change if it makes a sequence of actions a1, a2, . . . , an by repeatedly applying P(s'|s,a) all without actually producing an action to change the environment.
- Hence, the predicted trajectory occurs in the agent's "head" using a model.

Model Based Algorithms

- Most commonly applied to games with a target state, such as winning or losing in a game of chess, or navigation tasks with a goal state s*.
- Do not model any rewards

Model based algorithms - Advantages

- Very appealing : it can play out scenarios and understand the consequences of its actions without having to actually act in an environment.
- Require many fewer samples of data to learn good policies since having a model enables an agent to supplement its actual experiences with imagined ones.

Model based algorithms - challenges

- Models are hard to come by.
- An environment with a large state space and action space can be very difficult to model; doing so may even be intractable, especially if the transitions are extremely complex
- Models are only useful when they can accurately predict the transitions of an environment many steps into the future - prediction errors

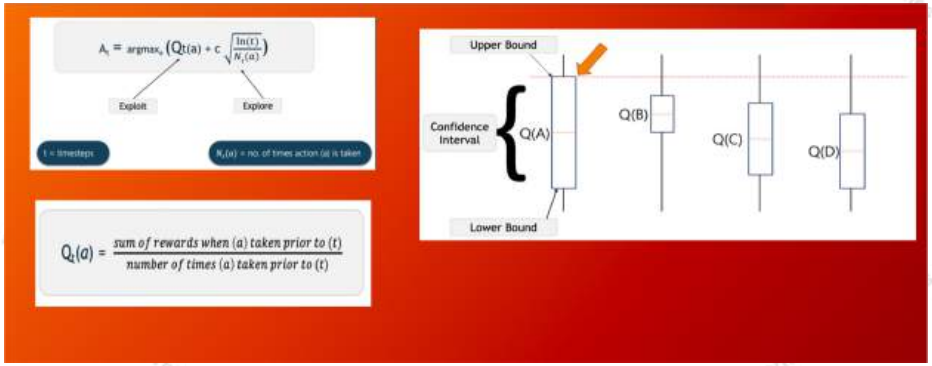
Upper Confidence bound action selection

UCB is a deterministic algorithm for Reinforcement Learning that focuses on exploration and exploitation based on a confidence boundary that the algorithm assigns to each machine on each round of exploration. These boundary decreases when a machine is used more in comparison to other machines.

The **Upper Confidence Bound** follows the principle of optimism in the face of uncertainty which implies that if we are uncertain about an action, we should optimistically assume that it is the correct action.

Initially, UCB explores more to systematically reduce uncertainty but its exploration reduces over time. Thus we can say that UCB obtains greater reward on average than other algorithms such as Epsilon-greedy, Optimistic Initial Values, etc.

UCB Algorithm



Upper Confidence bound action selection

Steps followed in Upper Confidence Bound

1. *At each round **n**, we consider two numbers for machine m.*
   -> $N_m(n)$ = number of times the machine m was selected up to round n.
   -> $R_m(n)$ = number of rewards of the machine m up to round n.
2. *From these two numbers we have to calculate,*
   a. The average reward of machine m up to round n, $r_m(n) = R_m(n) / N_m(n)$.
   b. The confidence interval [$r_m(n) — \Delta_m(n), r_m(n)+\Delta_m(n)$] at round n with, $\Delta_m(n)$= sqrt(1.5 * log(n) / $N_m(n)$)
3. *We select the machine **m** that has the maximum UCB,* ($r_m(n)+\Delta_m(n)$)

Monte Carlo Tree Search

- Monte Carlo Tree Search (MCTS) is a heuristic search set of rules that has won big attention and reputation within the discipline of synthetic intelligence, specially in the area of choice-making and game playing.
- MCTS combines the standards of Monte Carlo strategies, which rely upon random sampling and statistical evaluation, with tree-primarily based search techniques.
- **Idea :**

  - build a seek tree incrementally by using simulating more than one random performs (regularly known as rollouts or playouts) from the current recreation nation.

  - These simulations are carried out until a terminal state or a predefined intensity is reached.

  - The results of these simulations are then backpropagated up the tree, updating the records of the nodes visited at some stage in the play, which includes the wide variety of visits and the win ratios.

Monte Carlo Tree Search

- It is a probabilistic and heuristic driven search algorithm that combines the classic tree search implementations alongside machine learning principles of reinforcement learning.
- ***exploration-exploitation trade-off :*** exploits the actions and strategies that is found to be the best till now but also must continue to explore the local space of alternative decisions and find out if they could replace the current best.
- exploration expands the tree's breadth more than its depth.But it quickly becomes inefficient in situations with large number of steps or repetitions.
- Exploitation sticks to a single path that has the greatest estimated value. This is a greedy approach and this will extend the tree's depth more than its breadth.

Monte Carlo Tree Search

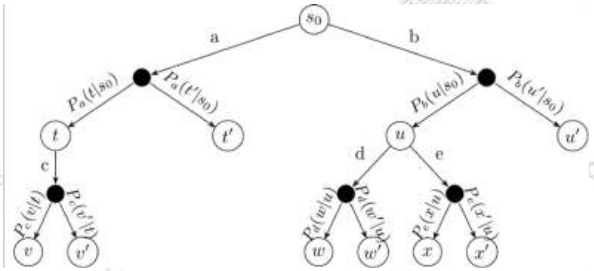
Why use Monte Carlo Tree Search (MCTS) ?

1. Handling Complex and Strategic Games
2. Unknown or Imperfect Information
3. Learning from Simulations
4. Optimizing Exploration and Exploitation
5. Scalability and Parallelization
6. Applicability Beyond Games
7. Domain Independence

Monte Carlo Tree Search

Nodes are the building blocks of the search tree and are formed based on the outcome of a number of simulations.

MDPs can be represented as trees (or graphs), called *ExpectiMax* trees:



The letters *a*-*e* represent actions, and letters *s*-*x* represent states. White nodes are state nodes, and the small black nodes represent the probabilistic uncertainty: the 'environment' choosing which outcome from an action happens, based on the transition function.

Monte Carlo Tree Search – Overview

The algorithm is online, which means the action selection is interleaved with action execution. Thus, MCTS is invoked every time an agent visits a new state.

Fundamental features:

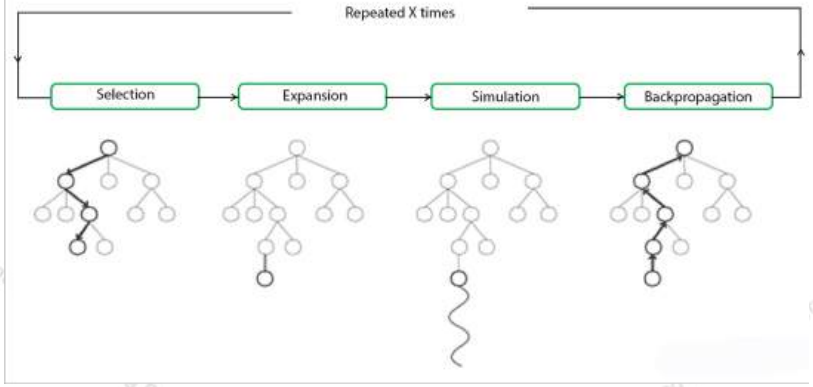
1. The Q-value Q(s,a) for each is approximated using *random simulation*.
2. For a single-agent problem, an ExpectiMax *search tree* is built incrementally
3. The search terminates when some pre-defined computational budget is used up, such as a time limit or a number of expanded nodes. Therefore, it is an *anytime* algorithm, as it can be terminated at any time and still give an answer.
4. The best performing action is returned.
   - This is complete if there are *no* dead–ends.
   - This is optimal if an entire search can be performed (which is unusual – if the problem is that small we should just use a dynamic programming technique such as value iteration).

The Framework of MCTS

The basic framework is to build up a tree using simulation. The states that have been evaluated are stored in a search tree. The set of evaluated states is *incrementally* built be iterating over the following four steps:

- ***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.
- ***Expand*****:** Expand this node by applying one available action (as defined by the MDP) from the node.
- ***Simulation*****:** From one of the outcomes of the expanded, perform a complete random simulation of the MDP to a terminating state. This therefore assumes that the simulation is finite, but versions of MCTS exist in which we just execute for some time and then estimate the outcome.
- ***Backpropagate*****:** Finally, the value of the node is *backpropagated* to the root node, updating the value of each ancestor node on the way using expected value

MCTS Framework



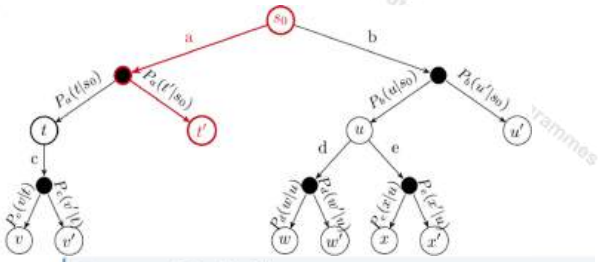
In a basic MCTS algorithm we incrementally build of the search tree. Each node in the tree stores:

1. a set of children nodes;
2. pointers to its parent node and parent action; and
3. the number of times it has been visited.

Algorithm – Monte-Carlo Tree Search

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

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



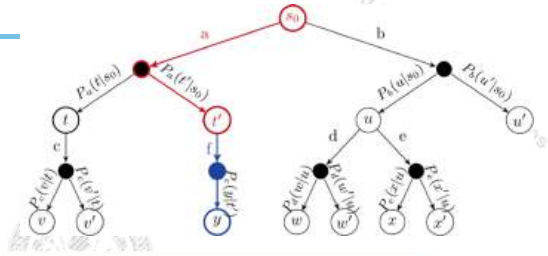
Selection: The first loop progressively selects a branch in the tree using a multi-armed bandit algorithm using Q(s|a). The outcome that occurs from an action is chosen according to P(s'|s,a) defined in the MDP.

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 $P_a(s' \mid s)$
  $s \leftarrow s'$
return s

Expansion

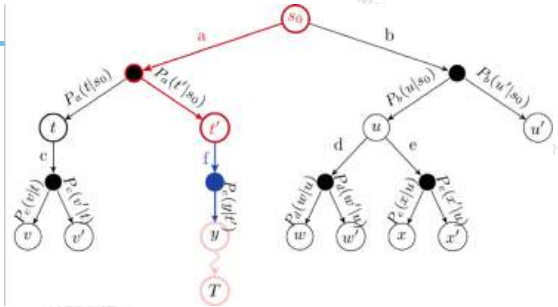
Select an action q to apply in state s, either randomly or using an heuristic. Get an outcome state s' from applying action a in state s according to the probability distribution p(s'|s). Expand a new environment node and a new state node for that outcome.



Function – Expand$(s : S)$

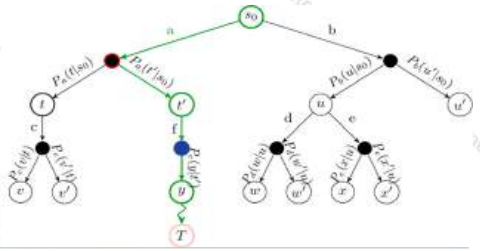
Input: state s
Output: expanded state s'
Select an action a from s to apply
Expand one outcome s' according to the distribution $P_a(s' \mid s)$ and observe reward r
return s'

Simulation: Perform a randomised simulation of the MDP until we reach a terminating state. That is, at each choice point, randomly select an possible action from the MDP, and use transition probabilities P(s'|s) to choose an outcome for each action.



Heuristics can be used to improve the random simulation by guiding it towards more promising states. G is the cumulative discounted reward received from the simulation starting at s' until the simulation terminates.
To avoid memory explosion, we discard all nodes generated from the simulation. In any non-trivial search, we are unlikely to ever need them again.

Backpropagation: The reward from the simulation is backpropagated from the selected node to its ancestors recursively. We must not forget the *discount factor*! For each state s and action a selected in the Select step, update the cumulative reward of that state.

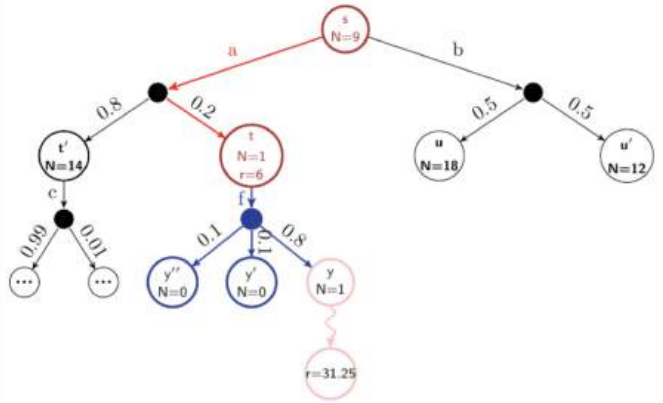


Procedure – Backpropagation$(s : S; a : A)$

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 $s \neq s_0$

Example : Backpropagation

Consider the following ExpectiMax tree that has been expanded several times. Assume $\gamma = 0.8$, $r = X$ represents reward X received at a state, V represents the value of the state (the value $\max_{a' \in children} Q(s, a')$) and the length of the simulation is 14. After the simulation step, but before backpropagation, our tree would look like this:



In the next iteration, the red actions are selected, and the blue node is expanded to its three children nodes. A simulation is run from y, which terminates after 3 steps. A reward of 31.25 is received in the terminal state. This would mean that the discounted reward returned at node y would be $\gamma \times 31.25$, which is 20.

Before backpropagation, we have the following:

$$\begin{aligned} Q(s,a) &= 18 \\ Q(t,f) &= 0 \end{aligned}$$

We leave out the remainder of the Q-function as it is not relevant to the example.

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

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

$$\begin{aligned} Q(t,f) &= Q(t,f) + \tfrac{1}{N(t,f)}[r + \gamma G - Q(t,f)] \\ &= 0 + \tfrac{1}{2}[0 + 0.8 \cdot 20 - 0] \\ &= 8 \end{aligned}$$

$$\begin{aligned} Q(s,a) &= Q(s,a) + \tfrac{1}{N(s,a)}[r + \gamma G - Q(s,a)] \\ &= 18 + \tfrac{1}{5}[6 + 0.8 \cdot (0.8 \cdot 20) - 18] \\ &= 18 + \tfrac{1}{5}[6 + 12.8 - 18] \\ &= 18.16 \end{aligned}$$

Action selection

Once we have run out of computational time, we select the action that maximises are expected return, which is simply the one with the highest Q-value from our simulations:

$$\text{argmax}_{a \in A(s)} Q(s_0, a)$$

We execute that action and wait to see which outcome occurs for the action.

Once we see the outcome state, which we will call s', we start the process all over again, except with $s_0 \leftarrow s'$.

However, importantly, we can *keep* the sub-tree from state s', as we already have done simulations from that state. We discard the rest of the tree (all child of s_0 other than the chosen action) and incrementally build from s'.

Issues in Monte Carlo Tree Search:

1. Exploration-Exploitation Trade-off
2. Sample Efficiency
3. High Variance
4. Heuristic Design
5. Computation and Memory Requirements
6. Overfitting
7. Domain-specific Challenges

	Value iteration	MCTS
Cost	Higher cost (exhaustive)	Lower cost (does not solve for entire state space)
Coverage/ Robustness	Higher (works from any state)	Lower (works only from initial state or state reachable from initial state)

AlphaZero

- Combining MCTS and TD learning: Alpha Zero
- Alpha Zero (or more accurately its predecessor AlphaGo) made headlines when it beat Go world champion Lee Sodol in 2016.
- It uses a combination of MCTS and (deep) reinforcement learning to learn a policy.

AlphaZero - Overview

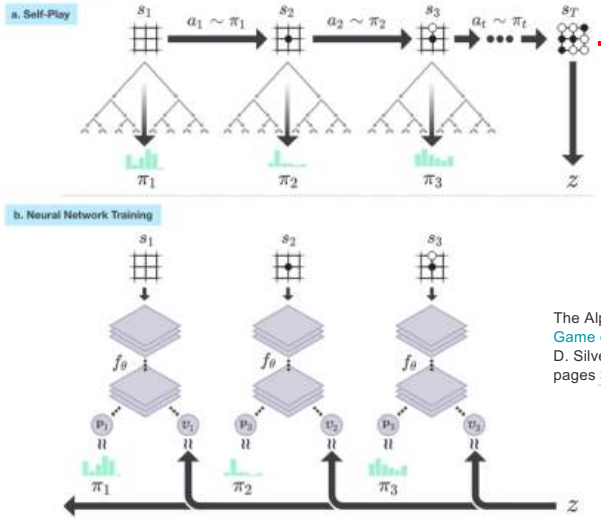
1. AlphaZero uses a deep neural network to estimate the Q-function. More accurately, it gives an estimate of the probability of selecting action a in state s, (P(a|s) and the *value* of the state (V(s)), which represents the probability of the player winning from s.
2. It is trained via *self-play*. Self-play is when the same policy is used to generate the moves of both the learning agent and any of its opponents. In AlphaZero, this means that initially, both players make random moves, but both also learn the same policy and use it to select subsequent moves.
3. In short, AlphaZero is a **game-playing program** that, through a combination of self-play and neural network reinforcement learning (more on that later), is able to learn to play games such as chess and Go from scratch ─ that is, after being fed nothing more than the rules of said games.

AlphaZero

A DNN initialised to return 2 randomly generated outputs: v(s), and p(s). The network will later be trained to take in position s and predict v(s) and p(s) from it.

At each move, AlphaZero:

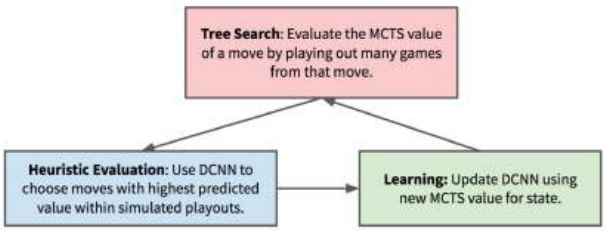
1. Executes an MCTS search using UCB-like selection: $Q(s,a) + P(s,a)/1 + N(s,a)$, which returns the probabilities of playing each move.
2. The neural network is used to guide the MCTS by influencing $Q(s,a)$.
3. The final result of a simulated game is used as the reward for each simulation.
4. After a set number of MCTS simulations, the best move is chosen for self-play.
5. Repeat steps 1-4 for each move until the self-play game ends.
6. Then, feedback the result of the self-play game to update the Q function for each move.



The AlphaZero framework. Mastering the Game of Go without Human Knowledge. D. Silver, et al. Nature volume 550, pages 354–359 (2017)

Implications of AlphaZero

- Increase the probability of a win come the end of the game
- AlphaZero's evaluation function is the product of a highly sophisticated neural network's experience accumulated over millions of games
- AlphaZero does not make use of any human knowledge, we can expect it to come up with brand-new ideas prev



MuZero

- AlphaZero (2017) - learning to play on its own, without any human data or domain knowledge, or even by mastering three different games (Go, Chess, Shogi) with only one algorithm being only provided with the known set of rules.
- MuZero (2020) - earnt to master these games(Go, Chess, Shogi, Atari) without being provided known rules
- Big step forward towards general-purpose algorithms

MuZero

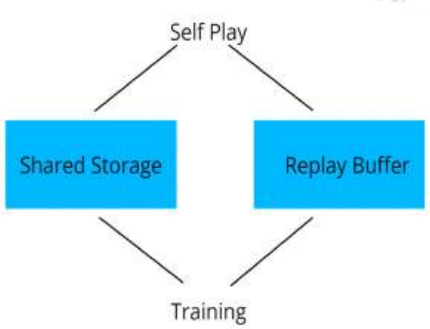
MuZero models three elements that are key to the planner:

- **Value**: how good is the current state?
- **Policy**: which action should be the next one?
- **Reward**: how good was the last action taken?

MuZero - overview

- The MuZero algorithm aims to accurately predict characteristics and details of thefuture which it deems important for planning.
- The algorithm initially receives an input, for instance an image of a chess board, which is translated into a hidden state.
- The hidden state then undergoes iterations based on the previous hidden state and a proposed subsequent plan of action.
- Each time the hidden state is updated the model predicts three variables: policy, value function and immediate reward.
- The policy is the next move to be played, the value function is the predicted winner, and the immediate reward is the strength of the move (if it improves the player's position).
- The model is then trained to accurately predict the values of the three aforementioned variables.

- The MuZero algorithm does not receive any rules of the game, for instance in a chess game, the algorithm is unaware of the legal moves or what constitutes as win, draw or a loss.
- MuZero only receives the rewards of its actions when a game is terminated, either by winning, drawing, or losing. However, during its learning, MuZero receives rewards periodically based on how well it is doing in its own mini-games.



Training and Loss function

- function train_network repeatedly trains the neural network by making use of the replayBuffer.
- The train_network function works by looping the total number of training steps, which is set to one million by default.
- The function then samples a batch at every step and uses that data to update the neural network.
- The tuples within a batch are: the current state of the game, a list of actions taken from the current position and lastly the targets used to train the neural networks.
- The targets used to train the neural networks are calculated by using Temporal Difference (TD) Learning .
- The loss function of MuZero is responsible for how the weights of the neural network are updated.

PlaNet

- Deep Planning Network - Google AI & DeepMind Initiative
- PlaNet agent was tasked with 'planning' a sequence of actions to achieve a goal like pole balancing, teaching a virtual entity (human or cheetah) to walk, or keeping a box rotating by hitting it in a specific location.
- Common goals between these tasks that the PlaNet needed to achieve:

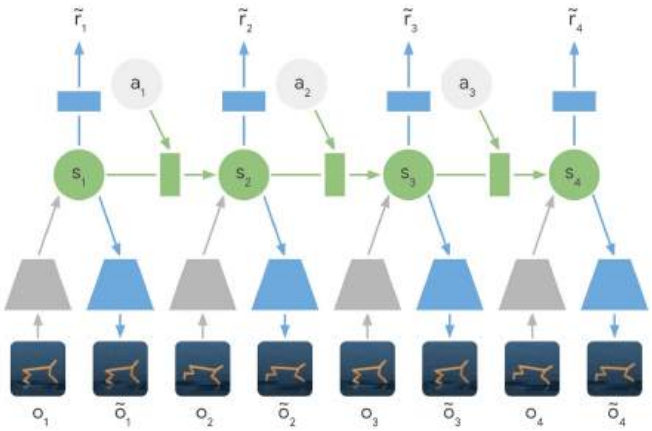
1. The Agent needs to predict a variety of possible futures (for robust planning)
2. The Agent needs to update the plan based on the outcomes/rewards of a recent action
3. The Agent needs to retain information over many time steps

So how did the Google AI team achieve these goals?

PlaNet Overview

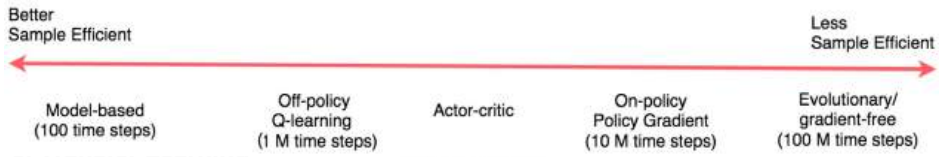
1. **Learning with a latent dynamics model** — PlaNet learns from a series of hidden or latent states *instead of images* to predict the latent state moving forward.
2. **Model-based planning** — PlaNet works without a policy network and instead makes decisions based on continuous planning.
3. **Transfer learning** — The Google AI team trained a single PlaNet agent to solve all six different tasks.

1) **Latent Dynamics Model -** Key benefits to using compact latent state spaces are that it allows the agent to learn more abstract representations like the objects' positions and velocities and



Learned Latent Dynamics Model — Instead of using the input images directly, the encoder networks (gray trapezoids) compress the images' information into hidden states (green circles). These hidden states are then used to predict future images (blue trapezoids) and rewards (blue rectangle).

2. Model based planning



- one agent for all tasks

The agent is randomly placed into different environments without knowing the task, so it needs to infer the task from its image observations. Without changes to the hyper parameters, the multi-task agent achieves the same mean performance as individual agents.

Deep Reinforcement Learning

BITS Pilani

Prof. Vimal SP
CSIS

BITS Pilani

<AIMLCZG512, Deep Reinforcement Learning>
Lecture No. 16

Agenda for the classes

- ➔ Introduction
- ➔ Imitation Learning Via Supervised Learning
- ➔ Behavior Cloning
- ➔ DAGGER
- ➔ Inverse Reinforcement Learning, GAIL
- ➔ Applications

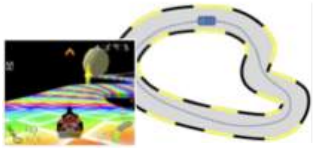
Imitation Learning in a Nutshell

Given: demonstrations or demonstrator

Goal: train a policy to mimic demonstrations

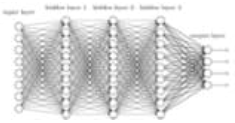
Images from Stephane Ross

Ingredients of Imitation Learning



Demonstrations or Demonstrator

Environment / Simulator



Policy Class

Loss Function

Learning Algorithm

Some Interesting Examples

- ALVINN https://www.ri.cmu.edu/publications/alvinn-an-auton`omous-land-vehicle-in-a-neural-network/
  Dean Pomerleau et al., 1989-1999 https://www.youtube.com/watch?v=ilP4aPDTBPE
- Helicopter Acrobatics
  Learning for Control from Multiple Demonstrations - Adam Coates, Pieter Abbeel, Andrew Ng, ICML 2008
  An Application of Reinforcement Learning to Aerobatic Helicopter Flight - Pieter Abbeel, Adam Coates, Morgan Quigley, Andrew Y. Ng, NIPS 2006
  https://www.youtube.com/watch?v=0JL04JJjocc
- Ghosting (Sports Analytics) - Next Slide.

Ghosting

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

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

Notation & Set-up

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

Action: a (sometimes y)

Policy: π_θ (sometimes h)

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

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

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

Notation & Set-up

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

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

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

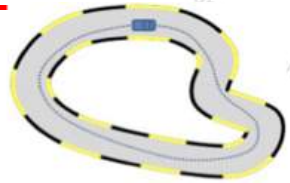
1. Sample s_0 from P_0 (distribution over initial states), initialize t = 1.
2. Sample action a_i from $\pi(s_{t-1})$
3. Sample next state s_t from applying a_t to s_{t-1} (requires access to environment)
4. Repeat from Step 2 with t=t+1

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

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

Example #1: Racing Game

(Super Tux Kart)



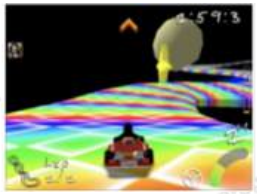
s = game screen

a = turning angle

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

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

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



Images from Stephane Ross

Example #2: Basketball Trajectories

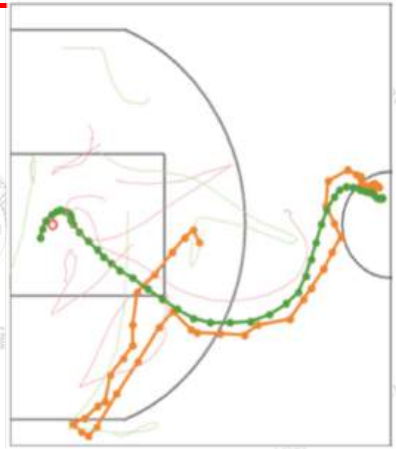
s = location of players & ball

a = next location of player

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

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

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

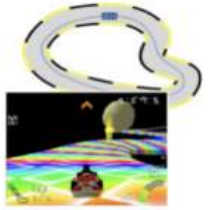


Behavioral Cloning = Reduction to Supervised Learning (Ignoring regularization for brevity.)

Define $P^* = P(s|\pi^*)$ (distribution of states visited by expert)

(recall $P(s|\pi^*) = (1/T)\sum_t P_t(s|\pi^*)$)

(sometimes abuse notation: $P^* = P(s,a^*=\pi^*(s)|\pi^*)$)



Learning objective:

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

Interpretations:

1. Assuming perfect imitation so far, learn to continue imitating perfectly
2. Minimize 1-step deviation error along the expert trajectories

Images from Stephane Ross

Behavioral Cloning vs. Imitation Learning

Behavioral Cloning (Supervised Learning):

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

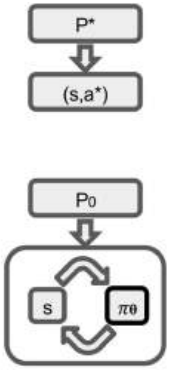
Distribution provided exogenously

Training Loss

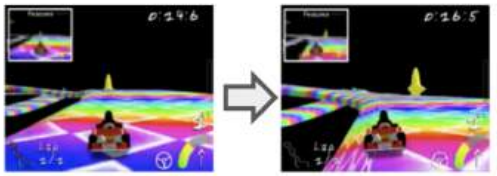
(General) Imitation Learning:

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

Distribution depends on rollout.
$P(s|\theta)$ = state distribution of π_θ



Limitations of Behavioral Cloning

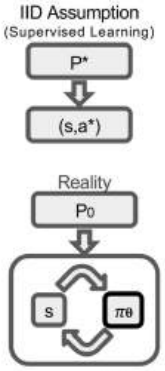


π_θ makes a mistake

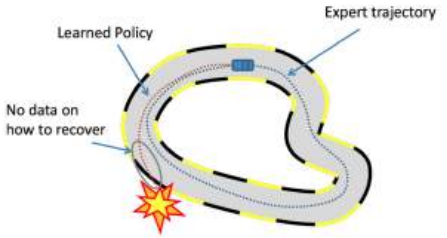
New state sampled not from P*!

Worst case is catastrophic!

Images from Stephane Ross



Limitations of Behavioral Cloning



Compounding Errors

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

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

When to use Behavioral Cloning?

Advantages

- Simple
- Simple
- Efficient

Disadvantages

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

Use When:

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

Don't Use When:

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

Types of Imitation Learning

Behavioral Cloning

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

Works well when P* close to Pθ

Inverse RL

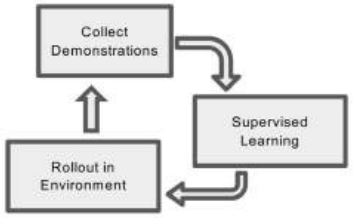
Learn r such that:

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

RL problem

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

Direct Policy Learning
via Interactive Demonstrator



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

Types of Imitation Learning

	Direct Policy Learning	Reward Learning	Access to Environment	Interactive Demonstrator	Pre-collected Demonstrations
Behavioral Cloning	Yes	No	No	No	Yes
Direct Policy Learning (Interactive IL)	Yes	No	Yes	Yes	Optional
Inverse Reinforcement Learning	No	Yes	Yes	No	Yes

Interactive Expert

Can query expert at any state

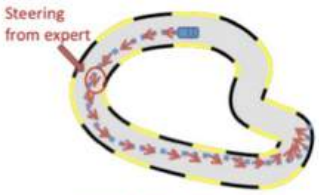
Construct loss function

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

Typically applied to rollout trajectories

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

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



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

Expert provides feedback on state visited by policy

Images from Stephane Ross

Interactive Expert

Can query expert at any state

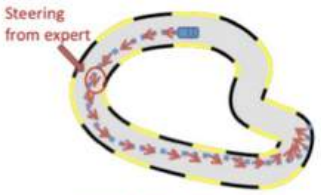
Construct loss function

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

Typically applied to rollout trajectories

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

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



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

Expert provides feedback on state visited by policy

Images from Stephane Ross

DAGGER: Dataset Aggregation

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

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

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

State = function (Charge Levels: High, Low)

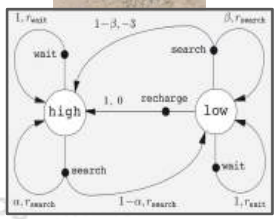
Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

Objective $\hat{\pi}^* = \arg\min_{\pi} \sum_{\zeta \in \mathcal{D}} \sum_{x \in \zeta} L(\pi(x), \pi^*(x))$

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	π_θ(State)
High	??
Low	??



Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

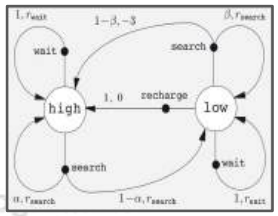
State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	π_θ(State)
High	Search
Low	Wait



Behavioral Cloning Exp-1:

Train the SL model using training data

P(Search | High) = 0.7, P(Wait | High) = 0.3

P(Search | Low) = 0.3, P(Wait | Low) = 0.6, P(Recharge | Low) = 0.3

Evaluate model Loss

$L(\theta_n) = \sum_i 1\,(\pi_{expert}(State) \neq \pi_{\theta n}(State)) = 0+1 = 1$

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

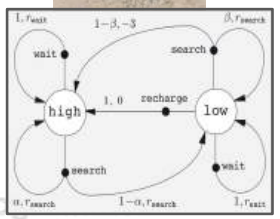
State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	π_θ(State)
High	Search
Low	Wait



Behavioral Cloning Exp-1:

Behave (Rollout) using π_{θ_n}

τ: (High, Search, +1, Low) → (Low, Wait, 0.5, Low) → (Low, Wait, 0.5, Low).....

- r_{search}=1
- r_{wait}=0.5
- No Charge = −3
- Recharge = 0

Loss

$L(\theta_n) = \sum_i 1\,(\pi_{expert}(State) \neq \pi_{\theta_n}(State)) = 0+1+1+.....(\infty) =$ **High Error**

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

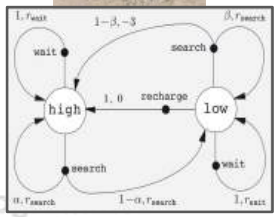
State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	π_θ(State)
High	Search
Low	Wait



Behavioral Cloning Exp-1:

Behave (Rollout) using π_{θ_n}

τ: (High, Search, Low) → (Low, Wait, Low) → (Low, Wait, Low).....

Observation: Compounding Error due to Distributional Shift

Loss

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

$L(\theta_n)=\sum_i 1\,(\pi_{expert}(State) \neq \pi_{\theta_n}(State)) = 0+1+1+\ldots\ldots(\infty)$ = **High Error**

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

DAGGER

Behavioral Cloning with Data augmentation Exp-2:

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	π_θ(State)
High	Search
Low	Wait

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

SL Training initializes the agent policy

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

DAGGER

Behavioral Cloning with Data augmentation Exp-2:

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	π_θ(State)
High	Search
Low	Wait

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

Behave (Rollout) using π_{θ_n}

$\beta = 0.6$ → $\pi_{\theta i} = 0.6\ \pi_{expert} + 0.4\ \pi_{agent}$

E → A → E

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

DAGGER

Behavioral Cloning with Data augmentation Exp-2:

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	π_θ(State)
High	Search
Low	Wait

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

Behave (Rollout) using π_{θ_n}

$\beta = 0.6$ → $\pi_{\theta i} = 0.6\ \pi_{expert} + 0.4\ \pi_{agent}$

E → A → E

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

τ: (High, Search, +1, Low) → (Low, Wait, 0.5, Low) → (Low, Recharge, 0, High)

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

DAGGER

Behavioral Cloning with Data augmentation Exp-2:

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	π_{θ}(State)
High	Search
Low	Wait

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

Behave (Rollout) using π_{θ_n}

τ: (High, Search, Low) → (Low, Wait, Low) → (Low, Recharge, High)

E: Search Recharge Recharge

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

DAGGER

Behavioral Cloning with Data augmentation Exp-2:

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	π_{θ}(State)
High	Search
Low	Wait

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

X	Y
.....	.....
High	Search
Low	Recharge
Low	Recharge

Behave (Rollout) using π_{θ_n}

τ: (High, Search, Low) → (Low, Wait, Low) → (Low, Recharge, High)

E: Search Recharge Recharge

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

DAGGER

Behavioral Cloning with Data augmentation Exp-2:

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	$\pi_{\theta(n+1)}$(State)
High	Search
Low	Recharge

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

X	Y
.....	.....
High	Search
Low	Recharge
Low	Recharge

Behave (Rollout) using π_{θ_n}

τ: (High, Search, Low) → (Low, Wait, Low) → (Low, Recharge, High)

E: Search Recharge Recharge

Validation Loss

$L(\theta_{n+1}) = \sum_j 1\ (\pi_{expert}(State) \neq \pi_{\theta j}(State)) = 0$ (Calculate for every iteration)

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

DAGGER

Behavioral Cloning with Data augmentation Exp-2:

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	$\pi_{\theta(n+1)}$(State)
High	Search
Low	Recharge

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

X	Y
.....	.....
High	Search
Low	Recharge
Low	Recharge
High	Search
Low	Recharge
Low	Recharge

Behave (Rollout) using $\pi_{\theta_{n+1}}$

τ: (High, Search, Low) → (Low, Recharge, High) → (Low, Recharge, High)

E: Search Recharge Recharge

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

DAGGER

Behavioral Cloning with Data augmentation Exp-2:

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	$\pi_{\theta(n+1)}$(State)
High	Search
Low	Recharge

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

X	Y
.....	.....
High	Search
Low	Recharge
Low	Recharge
High	Search
Low	Recharge
Low	Recharge

Behave (Rollout) using $\pi_{\theta_{n+1}}$

τ: (High, Search, Low) → (Low, Recharge, High) → (Low, Recharge, High)

Observation: What if the π_{expert} is suboptimal? Trusts that expert's behavior is reward maximizing!!!

E: Search Recharge Recharge

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

Variations of DAGGER (Few Ideas!)

Behavioral Cloning with Data augmentation Exp-2:

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	$\pi_{\theta(n+1)}$(State)
High	Search
Low	Wait

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

X	Y	Reward
.....	.....	.....
High	Search	+1
Low	Wait	0.5
Low	Recharge	0

Behave (Rollout) using π_{θ_n}

E → A → E

τ: (High, Search, +1, Low) → (Low, Wait, 0.5, Low) → (Low, Recharge, 0, High)

E: Search Recharge Recharge

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

Recycling Robot

Objective : Policy to search for cans efficiently based on current charge level of the battery

State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}

Variations of DAGGER (Few Ideas!)

Behavioral Cloning with Data augmentation Exp-2:

Expert	π_{expert}(State)
High	Search
Low	Recharge

Agent	$\pi_{\theta(n+1)}$(State)
High	Search
Low	Wait

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

X	Y	Reward
.....	.....	.....
High	Search	+1
Low	Wait	0.5
Low	Recharge	0
High	Search	+1
Low	Wait	0
Low	~~Wait~~ Recycle	0

Behave (Rollout) using π_{θ_n}

E → A → A

τ: (High, Search, +1, Low) → (Low, Wait, 0, Low) → (Low, Wait, 0, Low)

E: Search Recharge Recharge

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Bo

Recycling Robot

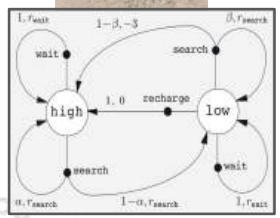
Objective : Policy to search for cans efficiently based on current charge level of the battery

Eg.,

State = function (Charge Levels: High, Low)

Action = function (Charge level based reactions)

- A(high) = {search, wait}
- A(low) = {search, wait, recharge}



Observation:
What is the π_{expert} intent? What if the agent's works in dynamic environment?

Solution : Learn Reward functions using Feature Expectation by parameterizing the R(s,a)

Source Reference: Example inspired from Recycle Robot example in Chapter 3 of Text Book-1

Inverse RL

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

Inverse RL

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

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Inverse RL

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

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

optimize this to explain the data

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

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

Inverse RL

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

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

optimize this to explain the data

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

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

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Inverse RL

The imitation learning perspective

Standard imitation learning:
- copy the *actions* performed by the expert
- no reasoning about outcomes of actions

Human imitation learning:
- copy the *intent* of the expert
- might take very different actions!

Inverse RL

The reinforcement learning perspective

what is the reward?

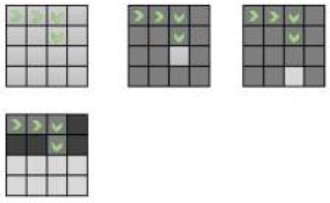
Inverse RL

Infer **reward functions** from **demonstrations**

$r(\mathbf{s}, \mathbf{a})$

by itself, this is an **underspecified** problem

many reward functions can explain the **same** behavior



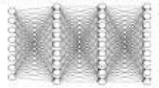
Inverse RL

"forward" reinforcement learning	inverse reinforcement learning
given:	given:
states $\mathbf{s} \in \mathcal{S}$, actions $\mathbf{a} \in \mathcal{A}$	states $\mathbf{s} \in \mathcal{S}$, actions $\mathbf{a} \in \mathcal{A}$
(sometimes) transitions $p(\mathbf{s}'\|\mathbf{s}, \mathbf{a})$	(sometimes) transitions $p(\mathbf{s}'\|\mathbf{s}, \mathbf{a})$
reward function $r(\mathbf{s}, \mathbf{a})$	samples $\{\tau_i\}$ sampled from $\pi^\star(\tau)$
learn $\pi^\star(\mathbf{a}\|\mathbf{s})$	learn $r_\psi(\mathbf{s}, \mathbf{a})$ (reward parameters)
	...and then use it to learn $\pi^\star(\mathbf{a}\|\mathbf{s})$

linear reward function:

$r_\psi(\mathbf{s}, \mathbf{a}) = \sum_i \psi_i f_i(\mathbf{s}, \mathbf{a}) = \psi^T \mathbf{f}(\mathbf{s}, \mathbf{a})$

neural net reward function:



$r_\psi(\mathbf{s}, \mathbf{a})$
parameters ψ

Inverse RL

Objective : Policy to search for cans efficiently based on current charge level of the battery

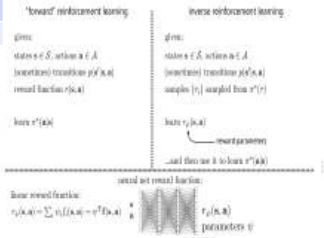
Feature Representation:

F1 : 1 if action = Search

F2 : 1 if battery status = Low

F3 : 1 if action = Recharge

	F1	F2	F3
(H,S)	1	0	0
(H,W)	0	0	0
(H,R)	0	0	1
(L,S)	1	1	0
(L,W)	0	1	0
(L,R)	0	1	1



Imitation Learning Exp-3:

$\tau 1$: (H,S)→(L,R)→(H,S)

$\tau 2$: (H,S)→(L,R)→(H,S)

Inverse RL

Objective : Policy to search for cans efficiently based on current charge level of the battery

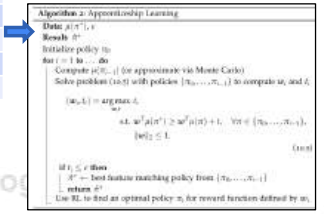
Feature Representation:

F1 : 1 if action = Search

F2 : 1 if battery status = Low

F3 : 1 if action = Recharge

	F1	F2	F3
(H,S)	1	0	0
(H,W)	0	0	0
(H,R)	0	0	1
(L,S)	1	1	0
(L,W)	0	1	0
(L,R)	0	1	1



Imitation Learning Exp-3: Standard Apprenticeship algorithm

$\tau 1$: (H,S)→(L,R)→(H,S) [1,0,0] → [0,1,1] → [1,0,0]

$\tau 2$: (H,S)→(L,R) [1,0,0] → [0,1,1]

Note: For simplicity only two demo is shown . But in reality large dataset of demo is used to compute μ_{π^*}

μ_{π^*} = [0.6, 0.4, 0.4]

ϵ = 0.1

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Inverse RL

Objective : Policy to search for cans efficiently based on current charge level of the battery

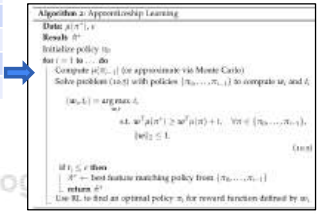
Feature Representation:

F1 : 1 if action = Search

F2 : 1 if battery status = Low

F3 : 1 if action = Recharge

	F1	F2	F3
(H,S)	1	0	0
(H,W)	0	0	0
(H,R)	0	0	1
(L,S)	1	1	0
(L,W)	0	1	0
(L,R)	0	1	1



Imitation Learning Exp-3: Standard Apprenticeship algorithm

$\tau 1$: (H,W)→(L,S) [0,0,0] → [1,1,0]

$\tau 2$: (H,R)→(L,W) [0,0,1] → [0,1,0]

$\tau 3$: (H,S)→(L,R) [1,0,0] → [0,1,1]

μ_{π^*} = [0.6, 0.4, 0.4]

$\mu_{\pi 0}$ = [0.33, 0.5, 0.33]

$\theta = \frac{(\mu_{\pi^*} - \mu_{\pi 0})}{Norm((\mu_{\pi^*} - \mu_{\pi 0}))} = \frac{[0.267, -0.1, 0.067]}{0.292}$

ϵ = 0.1 < t = f(0.292)

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Inverse RL

Objective : Policy to search for cans efficiently based on current charge level of the battery

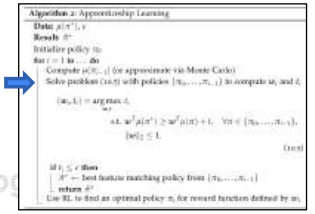
Feature Representation:

F1 : 1 if action = Search

F2 : 1 if battery status = Low

F3 : 1 if action = Recharge

	F1	F2	F3
(H,S)	1	0	0
(H,W)	0	0	0
(H,R)	0	0	1
(L,S)	1	1	0
(L,W)	0	1	0
(L,R)	0	1	1



Imitation Learning Exp-3: Standard Apprenticeship algorithm

$\tau 1$: (H,W)→(L,S) [0,0,0] → [1,1,0]

$\tau 2$: (H,R)→(L,W) [0,0,1] → [0,1,0]

$\tau 3$: (H,S)→(L,R) [1,0,0] → [0,1,1]

Reward = $\theta_0^T f$ = [0.914,−0.342,0.229] ^{T} * [1,0,0] = ~0.914

μ_{π^*} = [0.6, 0.4, 0.4]

$\mu_{\pi 0}$ = [0.33, 0.5, 0.33]

$\theta = \frac{(\mu_{\pi^*} - \mu_{\pi 0})}{Norm((\mu_{\pi^*} - \mu_{\pi 0}))} = \frac{[0.267, -0.1, 0.067]}{0.292}$

= [0.914,−0.342,0.229]

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Inverse RL

Agent	$\pi_{\theta(n+1)}$(State) Greedy
High	Search
Low	Search

Objective : Policy to search for cans efficiently based on current charge level of the battery

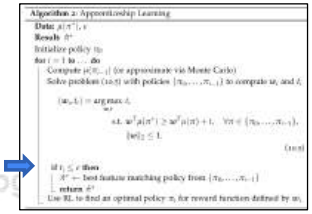
Feature Representation:

F1 : 1 if action = Search

F2 : 1 if battery status = Low

F3 : 1 if action = Recharge

	F1	F2	F3
(H,S)	1	0	0
(H,W)	0	0	0
(H,R)	0	0	1
(L,S)	1	1	0
(L,W)	0	1	0
(L,R)	0	1	1



Imitation Learning Exp-3: Standard Apprenticeship algorithm

τ1: (H,W)→(L,S) [0,0,0] → [1,1,0]

τ2: (H,R)→(L,W) [0,0,1] → [0,1,0]

τ3: (H,S)→(L,R) [1,0,0] → [0,1,1]

Reward = $\theta_0{}^T f$ = [0.914,−0.342,0.229] ^{T} * [1,0,0] = **~0.914**

μ_{π^*} = [0.6, 0.4, 0.4]

$\mu_{\pi 0}$ = [0.33, 0.5, 0.33]

$\theta = \frac{(\mu_{\pi^*} - \mu_{\pi 0})}{Norm((\mu_{\pi^*} - \mu_{\pi 0}))} = \frac{[0.267,-0.1,0.067]}{0.292}$

= [0.914,−0.342,0.229]

ϵ = 0.1 < t = f(0.292) ..... continue

Inverse RL

Agent	$\pi_{\theta(n+1)}$(State) Greedy
High	Search
Low	Search

Objective : Policy to search for cans efficiently based on current charge level of the battery

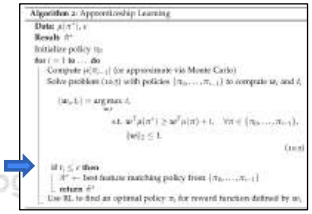
Feature Representation:

F1 : 1 if action = Search

F2 : 1 if battery status = Low

F3 : 1 if action = Recharge

	F1	F2	F3
(H,S)	1	0	0
(H,W)	0	0	0
(H,R)	0	0	1
(L,S)	1	1	0
(L,W)	0	1	0
(L,R)	0	1	1



Imitation Learning Exp-3: Standard Apprenticeship algorithm

τ1: (H,S)→(L,S) [1,0,0] → [1,1,0]

τ2: (H,S)→(L,S) [1,0,0] → [1,1,0]

τ3: (H,S)→(L,S) [1,0,0] → [1,1,0]

μ_{π^*} = [0.6, 0.4, 0.4]

Inverse RL

Agent	$\pi_{\theta(n+1)}$(State) Greedy
High	Search
Low	Search

Objective : Policy to search for cans efficiently based on current charge level of the battery

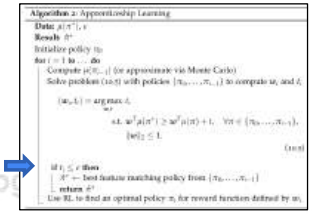
Feature Representation:

F1 : 1 if action = Search

F2 : 1 if battery status = Low

F3 : 1 if action = Recharge

	F1	F2	F3
(H,S)	1	0	0
(H,W)	0	0	0
(H,R)	0	0	1
(L,S)	1	1	0
(L,W)	0	1	0
(L,R)	0	1	1



Imitation Learning Exp-3: Standard Apprenticeship algorithm

τ1: (H,S)→(L,S) [1,0,0] → [1,1,0]

τ2: (H,S)→(L,S) [1,0,0] → [1,1,0]

τ3: (H,S)→(L,S) [1,0,0] → [1,1,0]

μ_{π^*} = [0.6, 0.4, 0.4]

$\mu_{\pi 1}$ = [1, 0.5, 0]

$\theta = \frac{(\mu_{\pi^*} - \mu_{\pi 0})}{Norm((\mu_{\pi^*} - \mu_{\pi 0}))} = \frac{[-0.4,-0.1,0.4]}{0.574}$

= [−0.697,−0.174,0.697]

ϵ = 0.1 < t = f(0.574) ..... continue

Inverse RL

Agent	$\pi_{\theta(n+1)}$(State) Greedy
High	Search
Low	Search

Objective : Policy to search for cans efficiently based on current charge level of the battery

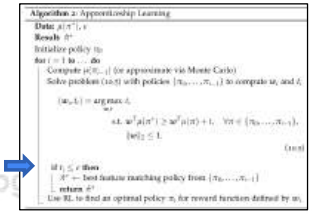
Feature Representation:

F1 : 1 if action = Search

F2 : 1 if battery status = Low

F3 : 1 if action = Recharge

	F1	F2	F3
(H,S)	1	0	0
(H,W)	0	0	0
(H,R)	0	0	1
(L,S)	1	1	0
(L,W)	0	1	0
(L,R)	0	1	1



Imitation Learning Exp-3: Standard Apprenticeship algorithm

τ1: (H,S)→(L,S) [1,0,0] → [1,1,0]

τ2: (H,S)→(L,S) [1,0,0] → [1,1,0]

τ3: (H,S)→(L,S) [1,0,0] → [1,1,0]

Reward = $\theta_0{}^T f$ = [−0.697,−0.174,0.697] ^{T} * [1,0,0] = **-0.697**

But for (H,R) ie., [0,0,1] Reward = 0.697

μ_{π^*} = [0.6, 0.4, 0.4]

$\mu_{\pi 1}$ = [1, 0.5, 0]

$\theta = \frac{(\mu_{\pi^*} - \mu_{\pi 0})}{Norm((\mu_{\pi^*} - \mu_{\pi 0}))} = \frac{[-0.4,-0.1,0.4]}{0.574}$

= [−0.697,−0.174,0.697]

ϵ = 0.1 < t = f(0.574) ..... continue

Inverse RL

Agent	$\pi_{\theta(n+1)}$(State) Greedy
High	Recharge
Low	Recharge

Objective : Policy to search for cans efficiently based on current charge level of the battery

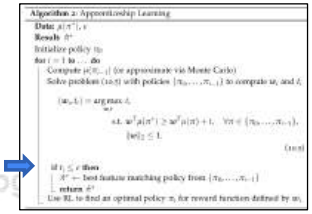
Feature Representation:

F1 : 1 if action = Search

F2 : 1 if battery status = Low

F3 : 1 if action = Recharge

	F1	F2	F3
(H,S)	1	0	0
(H,W)	0	0	0
(H,R)	0	0	1
(L,S)	1	1	0
(L,W)	0	1	0
(L,R)	0	1	1



Imitation Learning Exp-3: Standard Apprenticeship algorithm

$\tau 1$: (H,S)→(L,S) [1,0,0] → [1,1,0]

$\tau 2$: (H,S)→(L,S) [1,0,0] → [1,1,0]

$\tau 3$: (H,S)→(L,S) [1,0,0] → [1,1,0]

$\mu_{\pi^*} =$ [0.6, 0.4, 0.4]

$\mu_{\pi 1} =$ **[1, 0.5, 0]**

$\theta = \frac{(\mu_{\pi^*} - \mu_{\pi 0})}{Norm((\mu_{\pi^*} - \mu_{\pi 0}))} = \frac{[-0.4,-0.1,0.4]}{0.574}$

= [−0.697,−0.174,0.697]

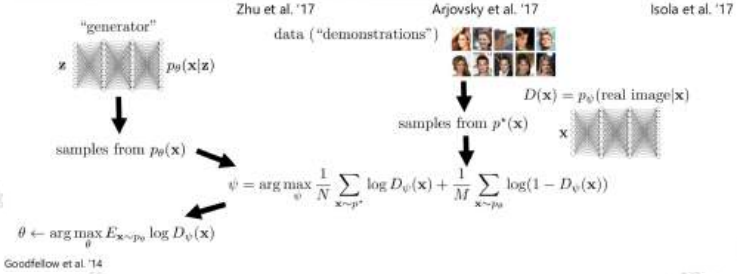
ε = 0.1 < t = f(0.574) ..... continue

Reward = $\theta_0^T f$ = [−0.697,−0.174,0.697]^T * [1,0,0] = **-0.697**

But for (H,R) ie., [0,0,1] Reward = 0.697

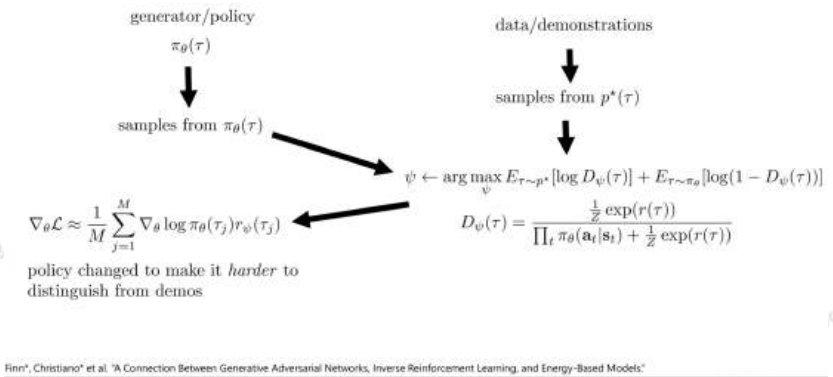
GAN

Zhu et al. '17 Arjovsky et al. '17 Isola et al. '17



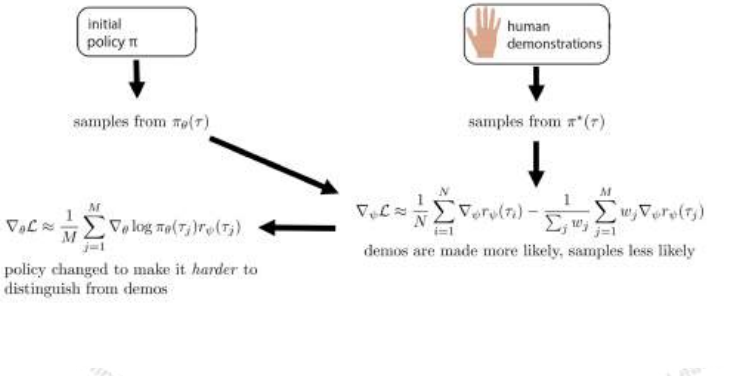
Goodfellow et al. '14

Inverse RL as GAN



Finn*, Christiano* et al. "A Connection Between Generative Adversarial Networks, Inverse Reinforcement Learning, and Energy-Based Models."

Inverse RL as GAN



Deep Reinforcement Learning

BITS Pilani

Prof. Vimal SP
CSIS

BITS Pilani

<AIMLCZG512, Deep Reinforcement Learning>
AlphaGo_Zero Research paper

Mastering the Game of Go without Human Knowledge

David Silver*, Julian Schrittwieser*, Karen Simonyan*, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, Demis Hassabis.

DeepMind, 5 New Street Square, London EC4A 3TW.

*These authors contributed equally to this work.

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 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*.

Much progress towards artificial intelligence has been made using supervised learning systems that are trained to replicate the decisions of human experts [1–4]. However, expert data is often expensive, unreliable, or simply unavailable. Even when reliable data is available it may impose a ceiling on the performance of systems trained in this manner [5]. In contrast, reinforcement learning systems are trained from their own experience, in principle allowing them to exceed human capabilities, and to operate in domains where human expertise is lacking. Recently, there has been rapid progress towards this goal, using deep neural networks trained by reinforcement learning. These systems have outperformed humans in computer games such as Atari [6,7] and 3D virtual environments [8–10]. However, the most challenging domains in terms of human intellect – such as the game of Go, widely viewed as a grand challenge for artificial intelligence [11] – require precise and sophisticated lookahead in vast search spaces. Fully general methods have not previously achieved human-level performance in these domains.

AlphaGo was the first program to achieve superhuman performance in Go. The published version [12], which we refer to as *AlphaGo Fan*, defeated the European champion Fan Hui in October 2015. *AlphaGo Fan* utilised two deep neural networks: a policy network that outputs move probabilities, and a value network that outputs a position evaluation. The policy network was trained initially by supervised learning to accurately predict human expert moves, and was subsequently refined by policy-gradient reinforcement learning. The value network was trained to predict the winner of games played by the policy network against itself. Once trained, these networks were combined with a Monte-Carlo Tree Search (MCTS) [13–15] to provide a lookahead search, using the policy network to narrow down the search to high-probability moves, and using the value network (in conjunction with Monte-Carlo rollouts using a fast rollout policy) to evaluate positions in the tree. A subsequent version, which we refer to as *AlphaGo Lee*, used a similar approach (see Methods), and defeated Lee Sedol, the winner of 18 international titles, in March 2016.

Our program, *AlphaGo Zero*, differs from *AlphaGo Fan* and *AlphaGo Lee* [12] 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 only uses 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.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

1 Reinforcement Learning in *AlphaGo Zero*

Our new method uses a deep neural network f_θ with parameters θ. 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, $(\mathbf{p}, v) = f_\theta(s)$. The vector of move probabilities $\mathbf{p}$ represents the probability of selecting each move (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 [12] into a single architecture. The neural network consists of many residual blocks [4] of convolutional layers [16,17] with batch normalisation [18] and rectifier non-linearities [19] (see Methods).

The neural network in *AlphaGo Zero* is trained from games of self-play by a novel reinforcement learning algorithm. In each position s, an MCTS search is executed, guided by the neural network f_θ. The MCTS search outputs probabilities $\boldsymbol{\pi}$ of playing each move. These search probabilities usually select much stronger moves than the raw move probabilities $\mathbf{p}$ of the neural network $f_\theta(s)$; MCTS may therefore be viewed as a powerful *policy improvement* operator [20,21]. Self-play with search — using the improved MCTS-based policy to select each move, then using the game winner z as a sample of the value — may be viewed as a powerful *policy evaluation* operator. The main idea of our reinforcement learning algorithm is to use these search operators repeatedly in a policy iteration procedure [22,23]: the neural network's parameters are updated to make the move probabilities and value $(\mathbf{p}, v) = f_\theta(s)$ more closely match the improved search probabilities and self-play winner $(\boldsymbol{\pi}, z)$; 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.

The Monte-Carlo tree search uses the neural network f_θ to guide its simulations (see Figure 2). Each edge (s, a) in the search tree stores a prior probability $P(s, a)$, a visit count $N(s, a)$, and an action-value $Q(s, a)$. Each simulation starts from the root state and iteratively selects moves that maximise an upper confidence bound $Q(s, a) + U(s, a)$, where $U(s, a) \propto P(s, a)/(1 + N(s, a))$ [12,24], until a leaf node s' is encountered. This leaf position is expanded and evaluated just

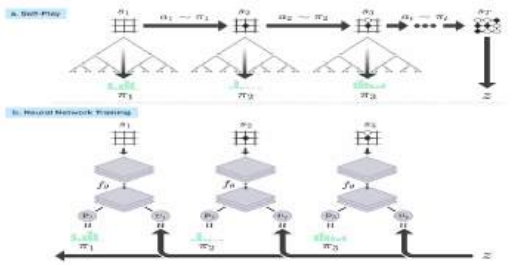


Figure 1: **Self-play reinforcement learning in *AlphaGo Zero*.** **a** The program plays a game $s_1, ..., s_T$ against itself. In each position s_t, a Monte-Carlo tree search (MCTS) α_θ is executed (see Figure 2) using the latest neural network f_θ. Moves are selected according to the search probabilities computed by the MCTS, $a_t \sim \pi_t$. The terminal position s_T is scored according to the rules of the game to compute the game winner z. **b** 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 θ, and outputs both a vector $\mathbf{p}_t$, 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 θ are updated so as to maximise the similarity of the policy vector $\mathbf{p}_t$ to the search probabilities $\boldsymbol{\pi}_t$, and to minimise 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 **a**.

Figure 2: **Monte-Carlo tree search in *AlphaGo Zero*.** **a** Each simulation traverses the tree by selecting the edge with maximum action-value Q, plus an upper confidence bound U that depends on a stored prior probability P and visit count N for that edge (which is incremented once traversed). **b** The leaf node is expanded and the associated position s is evaluated by the neural network $(P(s, \cdot), V(s)) = f_\theta(s)$; the vector of P values are stored in the outgoing edges from s. **c** Action-values Q are updated to track the mean of all evaluations V in the subtree below that action. **d** Once the search is complete, search probabilities $\boldsymbol{\pi}$ are returned, proportional to $N^{1/\tau}$, where N is the visit count of each move from the root state and τ is a parameter controlling temperature.

once by the network to generate both prior probabilities and evaluation, $(P(s', \cdot), V(s')) = f_\theta(s')$. Each edge (s, a) traversed in the simulation is updated to increment its visit count $N(s, a)$, and to update its action-value to the mean evaluation over these simulations, $Q(s, a) = 1/N(s, a) \sum_{s'|s,a \to s'} V(s')$, where $s, a \to s'$ indicates that a simulation eventually reached s' after taking move a from position s.

MCTS may be viewed as a self-play algorithm that, given neural network parameters θ and a root position s, computes a vector of search probabilities recommending moves to play, $\boldsymbol{\pi} = \alpha_\theta(s)$, proportional to the exponentiated visit count for each move, $\pi_a \propto N(s, a)^{1/\tau}$, where τ is a temperature parameter.

The neural network is trained by a self-play reinforcement learning algorithm that uses MCTS to play each move. First, the neural network is initialised to random weights θ_0. At each subsequent iteration $i \geq 1$, games of self-play are generated (Figure 1a). At each time-step t, an MCTS search $\boldsymbol{\pi}_t = \alpha_{\theta_{i-1}}(s_t)$ is executed using the previous iteration of neural network $f_{\theta_{i-1}}$, and a move is played by sampling the search probabilities $\boldsymbol{\pi}_t$. A game terminates at step T when

both players pass, when the search value drops below a resignation threshold, or when the game exceeds a maximum length; the game is then scored to give a final reward of $r_T \in \{-1, +1\}$ (see Methods for details). The data for each time-step t is stored as $(s_t, \boldsymbol{\pi}_t, z_t)$ where $z_t = \pm r_T$ is the game winner from the perspective of the current player at step t. In parallel (Figure 1b), new network parameters θ_i are trained from data $(s, \boldsymbol{\pi}, z)$ sampled uniformly among all time-steps of the last iteration(s) of self-play. The neural network $(\mathbf{p}, v) = f_{\theta_i}(s)$ is adjusted to minimise the error between the predicted value v and the self-play winner z, and to maximise the similarity of the neural network move probabilities $\mathbf{p}$ to the search probabilities $\boldsymbol{\pi}$. Specifically, the parameters θ are adjusted by gradient descent on a loss function l that sums over mean-squared error and cross-entropy losses respectively,

$$(\mathbf{p}, v) = f_\theta(s), \qquad l = (z - v)^2 - \boldsymbol{\pi}^\top \log \mathbf{p} + c||\theta||^2 \tag{1}$$

where c is a parameter controlling the level of L2 weight regularisation (to prevent overfitting).

2 Empirical Analysis of AlphaGo Zero Training

We applied our reinforcement learning pipeline to train our program *AlphaGo Zero*. Training started from completely random behaviour and continued without human intervention for approximately 3 days.

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).

Figure 3a shows the performance of *AlphaGo Zero* during self-play reinforcement learning, as a function of training time, on an Elo scale [25]. Learning progressed smoothly throughout training, and did not suffer from the oscillations or catastrophic forgetting suggested in prior literature

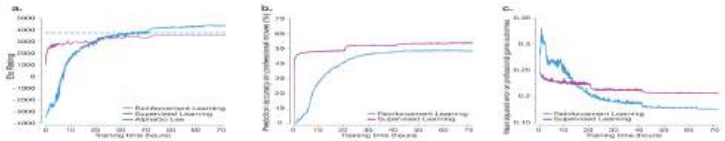


Figure 3: **Empirical evaluation of *AlphaGo Zero*.** **a** Performance of self-play reinforcement learning. The plot shows the performance of each MCTS player α_{θ_i} from each iteration i of reinforcement learning in *AlphaGo Zero*. Elo ratings were computed from evaluation games between different players, using 0.4 seconds of thinking time per move (see Methods). For comparison, a similar player trained by supervised learning from human data, using the *KGS* data-set, is also shown. **b** Prediction accuracy on human professional moves. The plot shows the accuracy of the neural network f_{θ_i}, at each iteration of self-play i, in predicting human professional moves from the *GoKifu* data-set. The accuracy measures the percentage of positions in which the neural network assigns the highest probability to the human move. The accuracy of a neural network trained by supervised learning is also shown. **c** Mean-squared error (MSE) on human professional game outcomes. The plot shows the MSE of the neural network f_{θ_i}, at each iteration of self-play i, in predicting the outcome of human professional games from the *GoKifu* data-set. The MSE is between the actual outcome $z \in \{-1, +1\}$ and the neural network value v, scaled by a factor of $\frac{1}{4}$ to the range $[0, 1]$. The MSE of a neural network trained by supervised learning is also shown.

[26–28]. Surprisingly, *AlphaGo Zero* outperformed *AlphaGo Lee* after just 36 hours; for comparison, *AlphaGo Lee* was trained over several months. After 72 hours, we evaluated *AlphaGo Zero* against the exact version of *AlphaGo Lee* that defeated Lee Sedol, under the 2 hour time controls and match conditions as were used in the man-machine match in Seoul (see Methods). *AlphaGo Zero* used a single machine with 4 Tensor Processing Units (TPUs) [29], while *AlphaGo Lee* was distributed over many machines and used 48 TPUs. *AlphaGo Zero* defeated *AlphaGo Lee* by 100 games to 0 (see Extended Data Figure 5 and Supplementary Information).

To assess the merits of self-play reinforcement learning, compared to learning from human data, we trained a second neural network (using the same architecture) to predict expert moves in the *KGS* data-set; this achieved state-of-the-art prediction accuracy compared to prior work [13, 30–33] (see Extended Data Table 1 and 2 respectively). Supervised learning achieved better initial performance, and was better at predicting the outcome of human professional games (Figure 3). Notably, although supervised learning achieved higher move prediction accuracy, the self-learned player performed much better overall, defeating the human-trained player within the first 24 hours of training. This suggests that *AlphaGo Zero* may be learning a strategy that is qualitatively different to human play.

To separate the contributions of architecture and algorithm, we compared the performance of the neural network architecture in *AlphaGo Zero* with the previous neural network architecture used in *AlphaGo Lee* (see Figure 4). Four neural networks were created, using either separate policy and value networks, as in *AlphaGo Lee*, or combined policy and value networks, as in *AlphaGo Zero*; and using either the convolutional network architecture from *AlphaGo Lee* or the residual network architecture from *AlphaGo Zero*. Each network was trained to minimise the same loss function (Equation 1) using a fixed data-set of self-play games generated by *AlphaGo Zero* after 72 hours of self-play training. Using a residual network was more accurate, achieved lower error, and improved performance in *AlphaGo* by over 600 Elo. Combining policy and value together into a single network slightly reduced the move prediction accuracy, but reduced the value error and boosted playing performance in *AlphaGo* by around another 600 Elo. This is partly due to

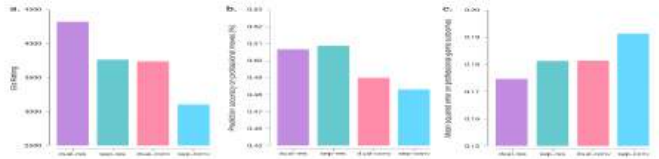


Figure 4: **Comparison of neural network architectures in *AlphaGo Zero* and *AlphaGo Lee*.** Comparison of neural network architectures using either separate ("sep") or combined policy and value networks ("dual"), and using either convolutional ("conv") or residual networks ("res"). The combinations "dual-res" and "sep-conv" correspond to the neural network architectures used in *AlphaGo Zero* and *AlphaGo Lee* respectively. Each network was trained on a fixed data-set generated by a previous run of *AlphaGo Zero*. **a** Each trained network was combined with *AlphaGo Zero*'s search to obtain a different player. Elo ratings were computed from evaluation games between these different players, using 5 seconds of thinking time per move. **b** Prediction accuracy on human professional moves (from the *GoKifu* data-set) for each network architecture. **c** Mean-squared error on human professional game outcomes (from the *GoKifu* data-set) for each network architecture.

improved computational efficiency, but more importantly the dual objective regularises the network to a common representation that supports multiple use cases.

3 Knowledge Learned by AlphaGo Zero

AlphaGo Zero discovered a remarkable level of Go knowledge during its self-play training process. This included fundamental elements of human Go knowledge, and also non-standard strategies beyond the scope of traditional Go knowledge.

Figure 5 shows a timeline indicating when professional *joseki* (corner sequences) were discovered (Figure 5a, Extended Data Figure 1); ultimately *AlphaGo Zero* preferred new *joseki* variants that were previously unknown (Figure 5b, Extended Data Figure 2). Figure 5c and the Supplementary Information show several fast self-play games played at different stages of training. Tournament length games played at regular intervals throughout training are shown in Extended Data Figure 3 and Supplementary Information. *AlphaGo Zero* rapidly progressed from entirely random moves towards a sophisticated understanding of Go concepts including *fuseki* (opening), *tesuji* (tactics), life-and-death, *ko* (repeated board situations), *yose* (endgame), capturing races, *sente* (initiative), shape, influence and territory, all discovered from first principles. Surprisingly, *shicho* ("ladder" capture sequences that may span the whole board) – one of the first elements of Go knowledge learned by humans – were only understood by *AlphaGo Zero* much later in training.

4 Final Performance of AlphaGo Zero

We subsequently applied our reinforcement learning pipeline to a second instance of *AlphaGo Zero* using a larger neural network and over a longer duration. Training again started from completely random behaviour and continued for approximately 40 days.

Over the course of training, 29 million games of self-play were generated. Parameters were updated from 3.1 million mini-batches of 2,048 positions each. The neural network contained

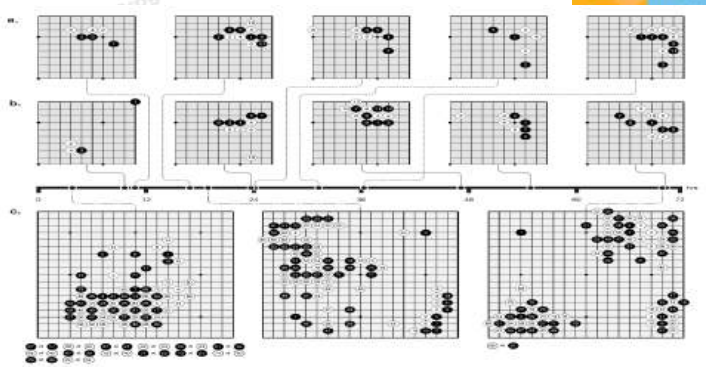


Figure 5: **Go knowledge learned by *AlphaGo Zero*.** **a** Five human *joseki* (common corner sequences) discovered during *AlphaGo Zero* training. The associated timestamps indicate the first time each sequence occured (taking account of rotation and reflection) during self-play training. Extended Data Figure 1 provides the frequency of occurrence over training for each sequence. **b** Five *joseki* favoured at different stages of self-play training. Each displayed corner sequence was played with the greatest frequency, among all corner sequences, during an iteration of self-play training. The timestamp of that iteration is indicated on the timeline. At 10 hours a weak corner move was preferred. At 47 hours the 3-3 invasion was most frequently played. This *joseki* is also common in human professional play; however *AlphaGo Zero* later discovered and preferred a new variation. Extended Data Figure 2 provides the frequency of occurence over time for all five sequences and the new variation. **c** The first 80 moves of three self-play games that were played at different stages of training, using 1,600 simulations (around 0.4s) per search. At 3 hours, the game focuses greedily on capturing stones, much like a human beginner. At 19 hours, the game exhibits the fundamentals of life-and-death, influence and territory. At 70 hours, the game is beautifully balanced, involving multiple battles and a complicated *ko* fight, eventually resolving into a half-point win for white. See Supplementary Information for the full games.

40 residual blocks. The learning curve is shown in Figure 6a. Games played at regular intervals throughout training are shown in Extended Data Figure 4 and Supplementary Information.

We evaluated the fully trained *AlphaGo Zero* using an internal tournament against *AlphaGo Fan*, *AlphaGo Lee*, and several previous Go programs. We also played games against the strongest existing program, *AlphaGo Master* – a program based on the algorithm and architecture presented in this paper but utilising human data and features (see Methods) – which defeated the strongest human professional players 60–0 in online games [34] in January 2017. In our evaluation, all programs were allowed 5 seconds of thinking time per move; *AlphaGo Zero* and *AlphaGo Master* each played on a single machine with 4 TPUs; *AlphaGo Fan* and *AlphaGo Lee* were distributed over 176 GPUs and 48 TPUs respectively. We also included a player based solely on the raw neural network of *AlphaGo Zero*; this player simply selected the move with maximum probability.

Figure 6b shows the performance of each program on an Elo scale. The raw neural network, without using any lookahead, achieved an Elo rating of 3,055. *AlphaGo Zero* achieved a rating of 5,185, compared to 4,858 for *AlphaGo Master*, 3,739 for *AlphaGo Lee* and 3,144 for *AlphaGo Fan*.

Finally, we evaluated *AlphaGo Zero* head to head against *AlphaGo Master* in a 100 game match with 2 hour time controls. *AlphaGo Zero* won by 89 games to 11 (see Extended Data Figure 6) and Supplementary Information.

5 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.

Humankind has accumulated Go knowledge from millions of games played over thousands of years, collectively distilled into patterns, proverbs and books. In the space of a few days, starting *tabula rasa*, *AlphaGo Zero* was able to rediscover much of this Go knowledge, as well as novel strategies that provide new insights into the oldest of games.

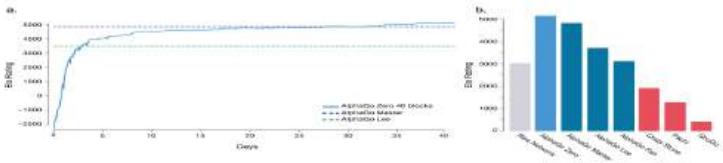


Figure 6: **Performance of *AlphaGo Zero*.** **a** Learning curve for *AlphaGo Zero* using larger 40 block residual network over 40 days. The plot shows the performance of each player α_{θ_i} from each iteration i of our reinforcement learning algorithm. Elo ratings were computed from evaluation games between different players, using 0.4 seconds per search (see Methods). **b** Final performance of *AlphaGo Zero*. *AlphaGo Zero* was trained for 40 days using a 40 residual block neural network. The plot shows the results of a tournament between: *AlphaGo Zero*, *AlphaGo Master* (defeated top human professionals 60-0 in online games), *AlphaGo Lee* (defeated Lee Sedol), *AlphaGo Fan* (defeated Fan Hui), as well as previous Go programs *Crazy Stone*, *Pachi* and *GnuGo*. Each program was given 5 seconds of thinking time per move. *AlphaGo Zero* and *AlphaGo Master* played on a single machine on the Google Cloud; *AlphaGo Fan* and *AlphaGo Lee* were distributed over many machines. The raw neural network from *AlphaGo Zero* is also included, which directly selects the move a with maximum probability p_a, without using MCTS. Programs were evaluated on an *Elo* scale [25]: a 200 point gap corresponds to a 75% probability of winning.

Methods

Reinforcement learning Policy iteration [20,21] is a classic algorithm that generates a sequence of improving policies, by alternating between *policy evaluation* – estimating the value function of the current policy – and *policy improvement* – using the current value function to generate a better policy. A simple approach to policy evaluation is to estimate the value function from the outcomes of sampled trajectories [35,36]. A simple approach to policy improvement is to select actions greedily with respect to the value function [20]. In large state spaces, approximations are necessary to evaluate each policy and to represent its improvement [22,23].

Classification-based reinforcement learning [37] improves the policy using a simple Monte-Carlo search. Many rollouts are executed for each action; the action with the maximum mean value provides a positive training example, while all other actions provide negative training examples; a policy is then trained to classify actions as positive or negative, and used in subsequent rollouts. This may be viewed as a precursor to the policy component of *AlphaGo Zero*'s training algorithm when $\tau \to 0$.

A more recent instantiation, classification-based modified policy iteration (CBMPI), also performs policy evaluation by regressing a value function towards truncated rollout values, similar to the value component of *AlphaGo Zero*; this achieved state-of-the-art results in the game of Tetris [38]. However, this prior work was limited to simple rollouts and linear function approximation using handcrafted features.

The *AlphaGo Zero* self-play algorithm can similarly be understood as an approximate policy iteration scheme in which MCTS is used for both policy improvement and policy evaluation. Policy improvement starts with a neural network policy, executes an MCTS based on that policy's recommendations, and then projects the (much stronger) search policy back into the function space of the neural network. Policy evaluation is applied to the (much stronger) search policy: the outcomes of self-play games are also projected back into the function space of the neural network. These projection steps are achieved by training the neural network parameters to match the search

probabilities and self-play game outcome respectively.

Guo et al. [7] also project the output of MCTS into a neural network, either by regressing a value network towards the search value, or by classifying the action selected by MCTS. This approach was used to train a neural network for playing Atari games; however, the MCTS was fixed — there was no policy iteration — and did not make any use of the trained networks.

Self-play reinforcement learning in games Our approach is most directly applicable to zero-sum games of perfect information. We follow the formalism of alternating Markov games described in previous work [12], noting that algorithms based on value or policy iteration extend naturally to this setting [39].

Self-play reinforcement learning has previously been applied to the game of Go. *NeuroGo* [40,41] used a neural network to represent a value function, using a sophisticated architecture based on Go knowledge regarding connectivity, territory and eyes. This neural network was trained by temporal-difference learning [42] to predict territory in games of self-play, building on prior work [43]. A related approach, *RLGO* [44], represented the value function instead by a linear combination of features, exhaustively enumerating all 3×3 patterns of stones; it was trained by temporal-difference learning to predict the winner in games of self-play. Both *NeuroGo* and *RLGO* achieved a weak amateur level of play.

Monte-Carlo tree search (MCTS) may also be viewed as a form of self-play reinforcement learning [45]. The nodes of the search tree contain the value function for the positions encountered during search; these values are updated to predict the winner of simulated games of self-play. MCTS programs have previously achieved strong amateur level in Go [46,47], but used substantial domain expertise: a fast *rollout policy*, based on handcrafted features [48,13], that evaluates positions by running simulations until the end of the game; and a *tree policy*, also based on handcrafted features, that selects moves within the search tree [47].

Self-play reinforcement learning approaches have achieved high levels of performance in other games: chess [49–51], checkers [52], backgammon [53], othello [54], Scrabble [55] and most recently

poker [56]. In all of these examples, a value function was trained by regression [54–56] or temporal-difference learning [49–53] from training data generated by self-play. The trained value function was used as an evaluation function in an alpha-beta search [49–54], a simple Monte-Carlo search [55–57], or counterfactual regret minimisation [56]. However, these methods utilised handcrafted input features [49–53,56] or handcrafted feature templates [54,55]. In addition, the learning process used supervised learning to initialise weights [58], hand-selected weights for piece values [49,51,52], handcrafted restrictions on the action space [56], or used pre-existing computer programs as training opponents [49,59] or to generate game records [51].

Many of the most successful and widely used reinforcement learning methods were first introduced in the context of zero-sum games: temporal-difference learning was first introduced for a checkers-playing program [59], while MCTS was introduced for the game of Go [13]. However, very similar algorithms have subsequently proven highly effective in video games [6–8,10], robotics [60], industrial control [61–63], and online recommendation systems [64,65].

AlphaGo versions We compare three distinct versions of *AlphaGo*:

1. *AlphaGo Fan* is the previously published program [12] that played against Fan Hui in October 2015. This program was distributed over many machines using 176 GPUs.
2. *AlphaGo Lee* is the program that defeated Lee Sedol 4–1 in March, 2016. It was previously unpublished but is similar in most regards to *AlphaGo Fan* [12]. However, we highlight several key differences to facilitate a fair comparison. First, the value network was trained from the outcomes of fast games of self-play by AlphaGo, rather than games of self-play by the policy network; this procedure was iterated several times – an initial step towards the tabula rasa algorithm presented in this paper. Second, the policy and value networks were larger than those described in the original paper – using 12 convolutional layers of 256 planes respectively – and were trained for more iterations. This player was also distributed over many machines using 48 TPUs, rather than GPUs, enabling it to evaluate neural networks faster during search.

3. *AlphaGo Master* is the program that defeated top human players by 60–0 in January, 2017 [34]. It was previously unpublished but uses the same neural network architecture, reinforcement learning algorithm, and MCTS algorithm as described in this paper. However, it uses the same handcrafted features and rollouts as *AlphaGo Lee* [12] and training was initialised by supervised learning from human data.
4. *AlphaGo Zero* is the program described in this paper. It learns from self-play reinforcement learning, starting from random initial weights, without using rollouts, with no human supervision, and using only the raw board history as input features. It uses just a single machine in the Google Cloud with 4 TPUs (*AlphaGo Zero* could also be distributed but we chose to use the simplest possible search algorithm).

Domain Knowledge Our primary contribution is to demonstrate that superhuman performance can be achieved without human domain knowledge. To clarify this contribution, we enumerate the domain knowledge that *AlphaGo Zero* uses, explicitly or implicitly, either in its training procedure or its Monte-Carlo tree search; these are the items of knowledge that would need to be replaced for *AlphaGo Zero* to learn a different (alternating Markov) game.

1. *AlphaGo Zero* is provided with perfect knowledge of the game rules. These are used during MCTS, to simulate the positions resulting from a sequence of moves, and to score any simulations that reach a terminal state. Games terminate when both players pass, or after $19 \cdot 19 \cdot 2 = 722$ moves. In addition, the player is provided with the set of legal moves in each position.
2. *AlphaGo Zero* uses Tromp-Taylor scoring [66] during MCTS simulations and self-play training. This is because human scores (Chinese, Japanese or Korean rules) are not well-defined if the game terminates before territorial boundaries are resolved. However, all tournament and evaluation games were scored using Chinese rules.
3. The input features describing the position are structured as a 19×19 image; i.e. the neural network architecture is matched to the grid-structure of the board.

4. The rules of Go are invariant under rotation and reflection; this knowledge has been utilised in *AlphaGo Zero* both by augmenting the data set during training to include rotations and reflections of each position, and to sample random rotations or reflections of the position during MCTS (see Search Algorithm). Aside from komi, the rules of Go are also invariant to colour transposition; this knowledge is exploited by representing the board from the perspective of the current player (see Neural network architecture)

AlphaGo Zero does not use any form of domain knowledge beyond the points listed above. It only uses its deep neural network to evaluate leaf nodes and to select moves (see section below). It does not use any rollout policy or tree policy, and the MCTS is not augmented by any other heuristics or domain-specific rules. No legal moves are excluded – even those filling in the player's own eyes (a standard heuristic used in all previous programs [67]).

The algorithm was started with random initial parameters for the neural network. The neural network architecture (see Neural Network Architecture) is based on the current state of the art in image recognition [4,18], and hyperparameters for training were chosen accordingly (see Self-Play Training Pipeline). MCTS search parameters were selected by Gaussian process optimisation [68], so as to optimise self-play performance of *AlphaGo Zero* using a neural network trained in a preliminary run. For the larger run (40 block, 40 days), MCTS search parameters were re-optimised using the neural network trained in the smaller run (20 block, 3 days). The training algorithm was executed autonomously without human intervention.

Self-Play Training Pipeline *AlphaGo Zero's* self-play training pipeline consists of three main components, all executed asynchronously in parallel. Neural network parameters θ_i are continually *optimised* from recent self-play data; *AlphaGo Zero* players α_{θ_i} are continually *evaluated*; and the best performing player so far, α_{θ_*}, is used to generate new *self-play* data.

Optimisation Each neural network f_{θ_i} is optimised on the Google Cloud using TensorFlow, with 64 GPU workers and 19 CPU parameter servers. The batch-size is 32 per worker, for a total mini-batch size of 2,048. Each mini-batch of data is sampled uniformly at random from

all positions from the most recent 500,000 games of self-play. Neural network parameters are optimised by stochastic gradient descent with momentum and learning rate annealing, using the loss in Equation 1. The learning rate is annealed according to the standard schedule in Extended Data Table 3. The momentum parameter is set to 0.9. The cross-entropy and mean-squared error losses are weighted equally (this is reasonable because rewards are unit scaled, $r \in \{-1, +1\}$) and the L2 regularisation parameter is set to $c = 10^{-4}$. The optimisation process produces a new checkpoint every 1,000 training steps. This checkpoint is evaluated by the evaluator and it may be used for generating the next batch of self-play games, as we explain next.

Evaluator To ensure we always generate the best quality data, we evaluate each new neural network checkpoint against the current best network f_{θ_*} before using it for data generation. The neural network f_{θ_i} is evaluated by the performance of an MCTS search α_{θ_i} that uses f_{θ_i} to evaluate leaf positions and prior probabilities (see Search Algorithm). Each evaluation consists of 400 games, using an MCTS with 1,600 simulations to select each move, using an infinitesimal temperature $\tau \to 0$ (i.e. we deterministically select the move with maximum visit count, to give the strongest possible play). If the new player wins by a margin of $> 55\%$ (to avoid selecting on noise alone) then it becomes the best player α_{θ_*}, and is subsequently used for self-play generation, and also becomes the baseline for subsequent comparisons.

Self-Play The best current player α_{θ_*}, as selected by the evaluator, is used to generate data. In each iteration, α_{θ_*} plays 25,000 games of self-play, using 1,600 simulations of MCTS to select each move (this requires approximately 0.4s per search). For the first 30 moves of each game, the temperature is set to $\tau = 1$; this selects moves proportionally to their visit count in MCTS, and ensures a diverse set of positions are encountered. For the remainder of the game, an infinitesimal temperature is used, $\tau \to 0$. Additional exploration is achieved by adding Dirichlet noise to the prior probabilities in the root node s_0, specifically $P(s, a) = (1 - \epsilon)p_a + \epsilon\eta_a$, where $\boldsymbol{\eta} \sim \mathrm{Dir}(0.03)$ and $\epsilon = 0.25$; this noise ensures that all moves may be tried, but the search may still overrule bad moves. In order to save computation, clearly lost games are resigned. The resignation threshold v_{resign} is selected automatically to keep the fraction of false positives (games that could have been

won if AlphaGo had not resigned) below 5%. To measure false positives, we disable resignation in 10% of self-play games and play until termination.

Supervised Learning For comparison, we also trained neural network parameters θ_{SL} by supervised learning. The neural network architecture was identical to *AlphaGo Zero*. Mini-batches of data $(s, \boldsymbol{\pi}, z)$ were sampled at random from the *KGS* data-set, setting $\pi_a = 1$ for the human expert move a. Parameters were optimised by stochastic gradient descent with momentum and learning rate annealing, using the same loss as in Equation 1, but weighting the mean-squared error component by a factor of 0.01. The learning rate was annealed according to the standard schedule in Extended Data Table 3. The momentum parameter was set to 0.9, and the L2 regularisation parameter was set to $c = 10^{-4}$.

By using a combined policy and value network architecture, and by using a low weight on the value component, it was possible to avoid overfitting to the values (a problem described in prior work [12]). After 72 hours the move prediction accuracy exceeded the state of the art reported in previous work [12,30–33], reaching 60.4% on the *KGS* test set; the value prediction error was also substantially better than previously reported [12]. The validation set was composed of professional games from *GoKifu*. Accuracies and mean squared errors are reported in Extended Data Table 1 and Extended Data Table 2 respectively.

Search Algorithm *AlphaGo Zero* uses a much simpler variant of the asynchronous policy and value MCTS algorithm (APV-MCTS) used in *AlphaGo Fan* and *AlphaGo Lee*.

Each node s in the search tree contains edges (s, a) for all legal actions $a \in \mathcal{A}(s)$. Each edge stores a set of statistics,

$$\{N(s, a), W(s, a), Q(s, a), P(s, a)\},$$

where $N(s, a)$ is the visit count, $W(s, a)$ is the total action-value, $Q(s, a)$ is the mean action-value, and $P(s, a)$ is the prior probability of selecting that edge. Multiple simulations are executed in parallel on separate search threads. The algorithm proceeds by iterating over three phases (a–c in Figure 2), and then selects a move to play (d in Figure 2).

Select (Figure 2a). The selection phase is almost identical to *AlphaGo Fan* [12]; we recapitulate here for completeness. The first *in-tree phase* of each simulation begins at the root node of the search tree, s_0, and finishes when the simulation reaches a leaf node s_L at time-step L. At each of these time-steps, $t < L$, an action is selected according to the statistics in the search tree, $a_t = \underset{a}{\mathrm{argmax}}\,(Q(s_t, a) + U(s_t, a))$, using a variant of the PUCT algorithm [24],

$$U(s, a) = c_{\mathrm{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

where c_{puct} is a constant determining the level of exploration; this search control strategy initially prefers actions with high prior probability and low visit count, but asymptotically prefers actions with high action-value.

Expand and evaluate (Figure 2b). The leaf node s_L is added to a queue for neural network evaluation, $(d_i(p), v) = f_\theta(d_i(s_L))$, where d_i is a dihedral reflection or rotation selected uniformly at random from $i \in [1..8]$.

Positions in the queue are evaluated by the neural network using a mini-batch size of 8; the search thread is locked until evaluation completes. The leaf node is expanded and each edge (s_L, a) is initialised to $\{N(s_L, a) = 0, W(s_L, a) = 0, Q(s_L, a) = 0, P(s_L, a) = p_a\}$; the value v is then backed up.

Backup (Figure 2c). The edge statistics are updated in a backward pass through each step $t \le L$. The visit counts are incremented, $N(s_t, a_t) = N(s_t, a_t) + 1$, and the action-value is updated to the mean value, $W(s_t, a_t) = W(s_t, a_t) + v$, $Q(s_t, a_t) = \frac{W(s_t, a_t)}{N(s_t, a_t)}$. We use virtual loss to ensure each thread evaluates different nodes [69].

Play (Figure 2d). At the end of the search *AlphaGo Zero* selects a move a to play in the root position s_0, proportional to its exponentiated visit count, $\pi(a|s_0) = N(s_0, a)^{1/\tau} / \sum_b N(s_0, b)^{1/\tau}$, where τ is a temperature parameter that controls the level of exploration. The search tree is reused at subsequent time-steps: the child node corresponding to the played action becomes the new root node; the subtree below this child is retained along with all its statistics, while the remainder of

the tree is discarded. *AlphaGo Zero* resigns if its root value and best child value are lower than a threshold value v_{resign}.

Compared to the MCTS in *AlphaGo Fan* and *AlphaGo Lee*, the principal differences are that *AlphaGo Zero* does not use any rollouts; it uses a single neural network instead of separate policy and value networks; leaf nodes are always expanded, rather than using dynamic expansion; each search thread simply waits for the neural network evaluation, rather than performing evaluation and backup asynchronously; and there is no tree policy. A transposition table was also used in the large (40 block, 40 day) instance of *AlphaGo Zero*.

Neural Network Architecture The input to the neural network is a $19 \times 19 \times 17$ image stack comprising 17 binary feature planes. 8 feature planes X_t consist of binary values indicating the presence of the current player's stones ($X_t^i = 1$ if intersection i contains a stone of the player's colour at time-step t; 0 if the intersection is empty, contains an opponent stone, or if $t < 0$). A further 8 feature planes, Y_t, represent the corresponding features for the opponent's stones. The final feature plane, C, represents the colour to play, and has a constant value of either 1 if black is to play or 0 if white is to play. These planes are concatenated together to give input features $s_t = [X_t, Y_t, X_{t-1}, Y_{t-1}, ..., X_{t-7}, Y_{t-7}, C]$. History features X_t, Y_t are necessary because Go is not fully observable solely from the current stones, as repetitions are forbidden; similarly, the colour feature C is necessary because the *komi* is not observable.

The input features s_t are processed by a residual tower that consists of a single convolutional block followed by either 19 or 39 residual blocks [4].

The convolutional block applies the following modules:

1. A convolution of 256 filters of kernel size 3×3 with stride 1
2. Batch normalisation [18]
3. A rectifier non-linearity

Each residual block applies the following modules sequentially to its input:

1. A convolution of 256 filters of kernel size 3×3 with stride 1
2. Batch normalisation
3. A rectifier non-linearity
4. A convolution of 256 filters of kernel size 3×3 with stride 1
5. Batch normalisation
6. A skip connection that adds the input to the block
7. A rectifier non-linearity

The output of the residual tower is passed into two separate "heads" for computing the policy and value respectively. The policy head applies the following modules:

1. A convolution of 2 filters of kernel size 1×1 with stride 1
2. Batch normalisation
3. A rectifier non-linearity
4. A fully connected linear layer that outputs a vector of size $19^2 + 1 = 362$ corresponding to logit probabilities for all intersections and the pass move

The value head applies the following modules:

1. A convolution of 1 filter of kernel size 1×1 with stride 1
2. Batch normalisation
3. A rectifier non-linearity

4. A fully connected linear layer to a hidden layer of size 256
5. A rectifier non-linearity
6. A fully connected linear layer to a scalar
7. A tanh non-linearity outputting a scalar in the range $[-1, 1]$

The overall network depth, in the 20 or 40 block network, is 39 or 79 parameterised layers respectively for the residual tower, plus an additional 2 layers for the policy head and 3 layers for the value head.

We note that a different variant of residual networks was simultaneously applied to computer Go [34] and achieved amateur dan-level performance; however this was restricted to a single-headed policy network trained solely by supervised learning.

Neural Network Architecture Comparison Figure 4 shows the results of a comparison between network architectures. Specifically, we compared four different neural networks:

1. *dual-res:* The network contains a 20-block residual tower, as described above, followed by both a policy head and a value head. This is the architecture used in *AlphaGo Zero*.
2. *sep-res:* The network contains two 20-block residual towers. The first tower is followed by a policy head and the second tower is followed by a value head.
3. *dual-conv:* The network contains a non-residual tower of 12 convolutional blocks, followed by both a policy head and a value head.
4. *sep-conv:* The network contains two non-residual towers of 12 convolutional blocks. The first tower is followed by a policy head and the second tower is followed by a value head. This is the architecture used in *AlphaGo Lee*.

Each network was trained on a fixed data-set containing the final 2 million games of self-play data generated by a previous run of *AlphaGo Zero*, using stochastic gradient descent with

the annealing rate, momentum, and regularisation hyperparameters described for the supervised learning experiment; however, cross-entropy and mean-squared error components were weighted equally, since more data was available.

Evaluation We evaluated the relative strength of *AlphaGo Zero* (Figure 3a and 6) by measuring the Elo rating of each player. We estimate the probability that player a will defeat player b by a logistic function $p(a \text{ defeats } b) = \frac{1}{1+\exp(c_{elo}[e(b)-e(a)])}$, and estimate the ratings $e(\cdot)$ by Bayesian logistic regression, computed by the *BayesElo* program [28] using the standard constant $c_{elo} = 1/400$.

Elo ratings were computed from the results of a 5 second per move tournament between *AlphaGo Zero*, *AlphaGo Master*, *AlphaGo Lee*, and *AlphaGo Fan*. The raw neural network from *AlphaGo Zero* was also included in the tournament. The Elo ratings of *AlphaGo Fan*, *Crazy Stone*, *Pachi* and *GnuGo* were anchored to the tournament values from prior work [12], and correspond to the players reported in that work. The results of the matches of *AlphaGo Fan* against Fan Hui and *AlphaGo Lee* against Lee Sedol were also included to ground the scale to human references, as otherwise the Elo ratings of *AlphaGo* are unrealistically high due to self-play bias.

The Elo ratings in Figure 3a, 4a and 6a were computed from the results of evaluation games between each iteration of player α_θ during self-play training. Further evaluations were also performed against baseline players with Elo ratings anchored to the previously published values [12].

We measured the head-to-head performance of *AlphaGo Zero* against *AlphaGo Lee*, and the 40 block instance of *AlphaGo Zero* against *AlphaGo Master*, using the same player and match conditions as were used against Lee Sedol in Seoul, 2016. Each player received 2 hours of thinking time plus 3 byoyomi periods of 60 seconds per move. All games were scored using Chinese rules with a *komi* of 7.5 points.

Data Availability The datasets used for validation and testing are the GoKifu dataset (available from *http://gokifu.com/*) and the KGS dataset (available from *https://u-go.net/gamerecords/*).

	KGS train	*KGS* test	*GoKifu* validation
Supervised learning (20 block)	62.0	60.4	54.3
Supervised learning (12 layer [12])	59.1	55.9	-
Reinforcement learning (20 block)	-	-	49.0
Reinforcement learning (40 block)	-	-	51.3

Extended Data Table 1: Move prediction accuracy. Percentage accuracies of move prediction for neural networks trained by reinforcement learning (i.e. *AlphaGo Zero*) or supervised learning respectively. For supervised learning, the network was trained for 3 days on *KGS* data (amateur games); comparative results are also shown from Silver et al [12]. For reinforcement learning, the 20 block network was trained for 3 days and the 40 block network was trained for 40 days. Networks were also evaluated on a validation set based on professional games from the *GoKifu* data set.

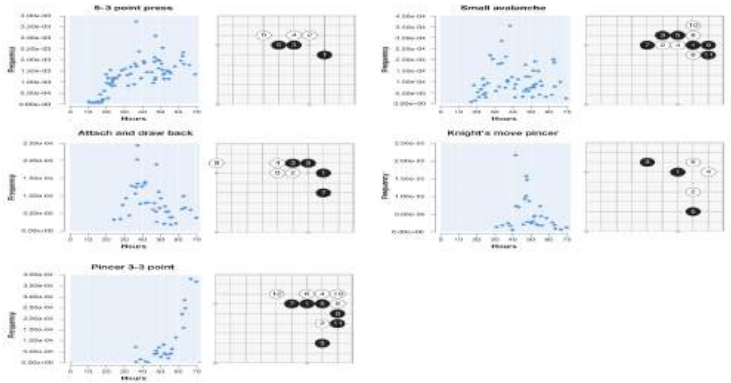
	KGS train	*KGS* test	*GoKifu* validation
Supervised learning (20 block)	0.177	0.185	0.207
Supervised learning (12 layer [12])	0.19	0.37	-
Reinforcement learning (20 block)	-	-	0.177
Reinforcement learning (40 block)	-	-	0.180

Extended Data Table 2: Game outcome prediction error. Mean squared error on game outcome predictions for neural networks trained by reinforcement learning (i.e. *AlphaGo Zero*) or supervised learning respectively. For supervised learning, the network was trained for 3 days on *KGS* data (amateur games); comparative results are also shown from Silver et al [12]. For reinforcement learning, the 20 block network was trained for 3 days and the 40 block network was trained for 40 days. Networks were also evaluated on a validation set based on professional games from the *GoKifu* data set.

Thousands of steps	Reinforcement learning	Supervised learning
0–200	10^{-2}	10^{-1}
200–400	10^{-2}	10^{-2}
400–600	10^{-3}	10^{-3}
600–700	10^{-4}	10^{-4}
700–800	10^{-4}	10^{-5}
>800	10^{-4}	-

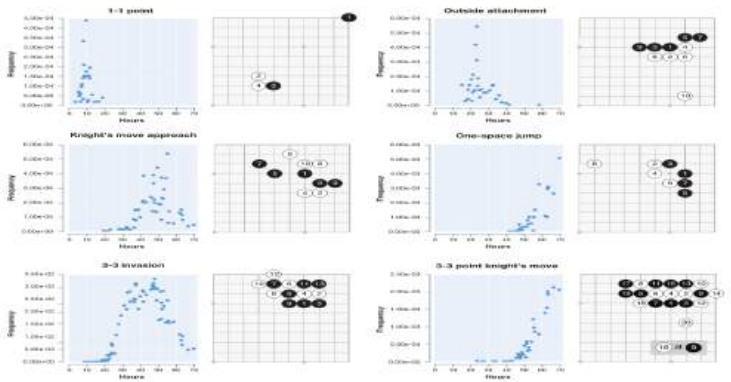
Extended Data Table 3: Learning rate schedule. Learning rate used during reinforcement learning and supervised learning experiments, measured in thousands of steps (mini-batch updates).

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956



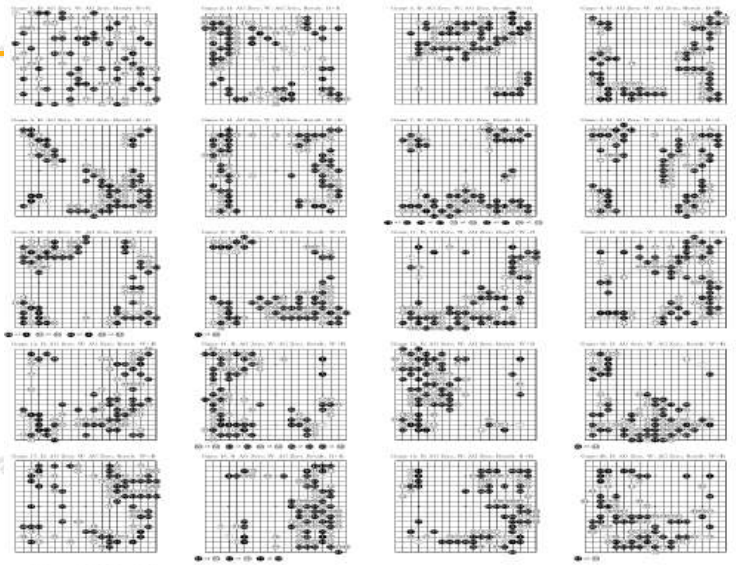
Extended Data Figure 1: Frequency of occurence over time during training, for each *joseki* from Figure 5a (corner sequences common in professional play that were discovered by *AlphaGo Zero*). The corresponding *joseki* are reproduced in this figure as insets.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

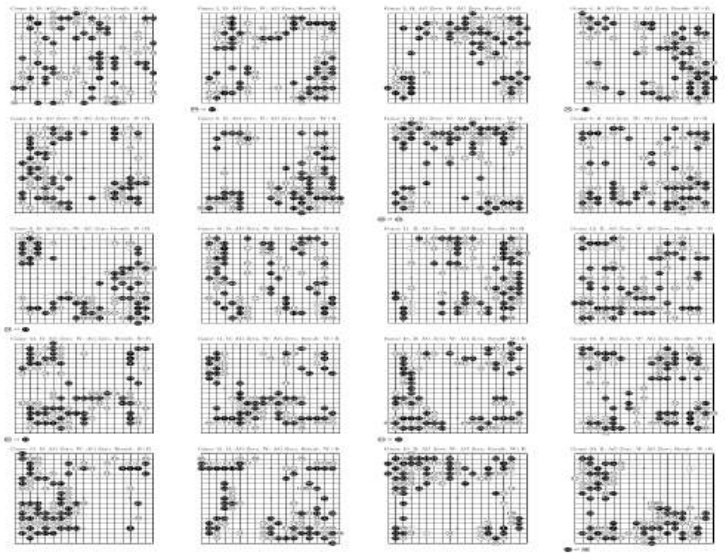


Extended Data Figure 2: Frequency of occurence over time during training, for each *joseki* from Figure 5b, (corner sequences that *AlphaGo Zero* favoured for at least one iteration), and one additional variation. The corresponding *joseki* are reproduced in this figure as insets.

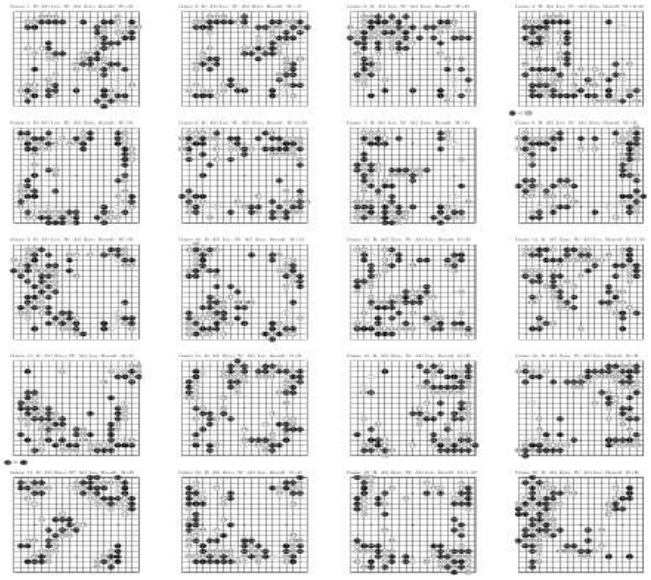
BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956



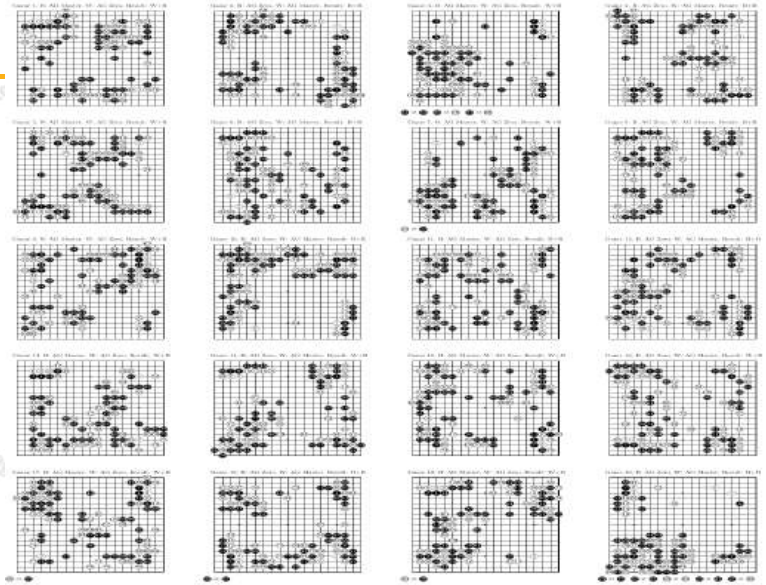
Extended Data Figure 3: *AlphaGo Zero* (20 block) self-play games. The 3 day training run was subdivided into 20 periods. The best player from each period (as selected by the evaluator) played a single game against itself, with 2 hour time controls. 100 moves are shown for each game; full games are provided in Supplementary Information.



Extended Data Figure 4: *AlphaGo Zero* (40 block) self-play games. The 40 day training run was subdivided into 20 periods. The best player from each period (as selected by the evaluator) played a single game against itself, with 2 hour time controls. 100 moves are shown for each game; full games are provided in Supplementary Information.



Extended Data Figure 5: Tournament games between *AlphaGo Zero* (20 block, 3 day) versus *AlphaGo Lee* using 2 hour time controls. 100 moves of the first 20 games are shown; full games are provided in Supplementary Information.



Extended Data Figure 6: *AlphaGo Zero* (40 block, 40 day) versus *AlphaGo Master* tournament games using 2 hour time controls. 100 moves of the first 20 games are shown; full games are provided in Supplementary Information.