Artificial Intelligence Foundation – JC3001

Lecture 45: Reinforcement Learning - II

Prof. Aladdin Ayesh (aladdin.ayesh@abdn.ac.uk)

Dr. Binod Bhattarai (binod.bhattarai@abdn.ac.uk)

Dr. Gideon Ogunniye, (g.ogunniye@abdn.ac.uk)

October 2025

Material adapted from:

Russell and Norvig (AIMA Book): Chapter 22

Sutton and Barto (Reinforcement Learning: An Introduction 2nd ed.)

David Silver (UCL)

Michael Littman (Brown University) and Charles Isbell (GA Tech)

Course Progression

- Part 1: Introduction
  1. Introduction to AI ✓
  2. Agents ✓
- Part 2: Problem-solving
  1. Search 1: Uninformed Search ✓
  2. Search 2: Heuristic Search ✓
  3. Search 3: Local Search ✓
  4. Search 4: Adversarial Search ✓
- Part 3: Reasoning and Uncertainty
  1. Reasoning 1: Constraint Satisfaction ✓
  2. Reasoning 2: Logic and Inference ✓
  3. Probabilistic Reasoning 1: BNs ✓
  4. Probabilistic Reasoning 2: HMMs ✓
- Part 4: Planning
  1. Planning 1: Intro and Formalism ✓
  2. Planning 2: Algos and Heuristics ✓
  3. Planning 3: Hierarchical Planning ✓
  4. Planning 4: Stochastic Planning ✓
- Part 5: Learning
  1. Learning 1: Intro to ML ✓
  2. Learning 2: Regression ✓
  3. Learning 3: Neural Networks ✓
  4. **Learning 4: Reinforcement Learning**
- Part 6: Conclusion
  1. Ethical Issues in AI
  2. Conclusions and Discussion

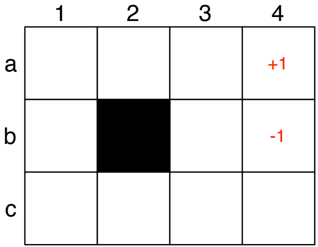
Objectives

- Bandit Problems ✓
- Reinforcement Learning based Agents ✓
- Tabular Reinforcement Learning
- Function Generalization

Outline

Passive Temporal Difference Learning

- We start with a policy π

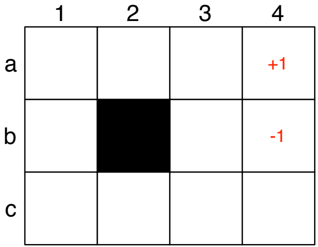


Algorithm consists of:

if s' is new **then** $U[s'] \leftarrow r'$
if s is not null **then**
  increment $N_s[s]$
  $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

Passive Temporal Difference Learning

- We start with a policy π
- $U(s)$ — Utility for s (init. null)

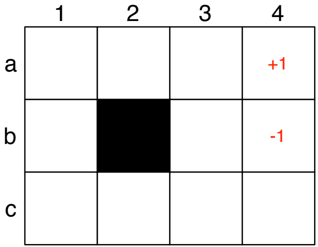


Algorithm consists of:

if s' is new **then** $U[s'] \leftarrow r'$
if s is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

Passive Temporal Difference Learning

- We start with a policy π
- $U(s)$ — Utility for s (init. null)
- $N(s)$ — # times we visited s

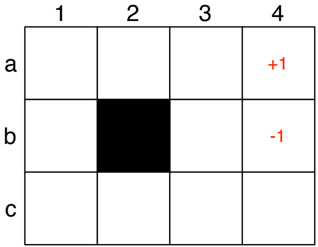


Algorithm consists of:

if s' is new **then** $U[s'] \leftarrow r'$
if s is not null **then**
  increment $N_s[s]$
  $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

Passive Temporal Difference Learning

- We start with a policy π
- $U(s)$ — Utility for s (init. null)
- $N(s)$ — # times we visited s



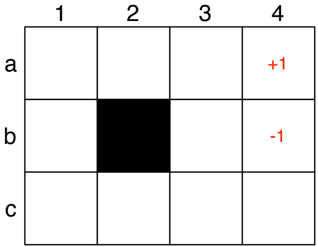
Algorithm consists of:

- Follow π from the initial state to an end state (multiple times);

if s' is new **then** $U[s'] \leftarrow r'$
if s is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

Passive Temporal Difference Learning

- We start with a policy π
- $U(s)$ — Utility for s (init. null)
- $N(s)$ — # times we visited s



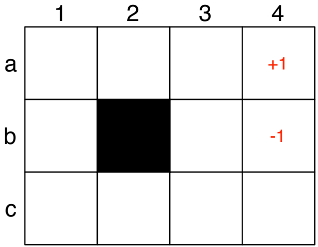
Algorithm consists of:

- Follow π from the initial state to an end state (multiple times);
- Keep track of the number of times we visited each state;

if s' is new **then** $U[s'] \leftarrow r'$
if s is not null **then**
  increment $N_s[s]$
  $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

Passive Temporal Difference Learning

- We start with a policy π
- $U(s)$ — Utility for s (init. null)
- $N(s)$ — # times we visited s



Algorithm consists of:

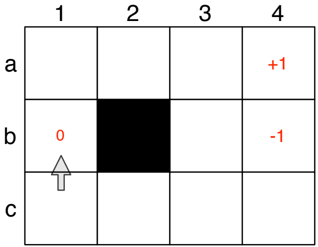
- Follow π from the initial state to an end state (multiple times);
- Keep track of the number of times we visited each state;
- Update the utility of the states as we go, using the following algorithm:

if s' is new **then** $U[s'] \leftarrow r'$
if s is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

Assume that rewards for non-terminals is 0

Passive Temporal Difference Learning

- We start with a policy π
- $U(s)$ — Utility for s (init. null)
- $N(s)$ — # times we visited s



Algorithm consists of:

- Follow π from the initial state to an end state (multiple times);
- Keep track of the number of times we visited each state;
- Update the utility of the states as we go, using the following algorithm:

if s' is new **then** $U[s'] \leftarrow r'$

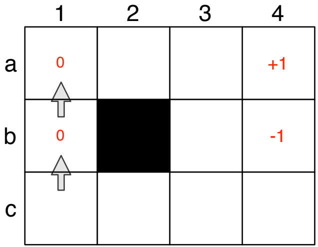
if s is not null **then**

increment $N_s[s]$

$U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

Passive Temporal Difference Learning

- We start with a policy π
- $U(s)$ — Utility for s (init. null)
- $N(s)$ — # times we visited s



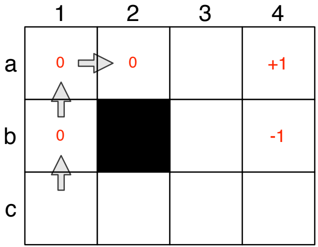
Algorithm consists of:

- Follow π from the initial state to an end state (multiple times);
- Keep track of the number of times we visited each state;
- Update the utility of the states as we go, using the following algorithm:

if s' is new **then** $U[s'] \leftarrow r'$
if s is not null **then**
increment $N_s[s]$
$U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

Passive Temporal Difference Learning

- We start with a policy π
- $U(s)$ — Utility for s (init. null)
- $N(s)$ — # times we visited s



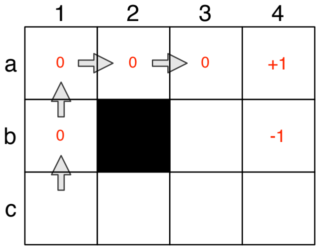
Algorithm consists of:

- Follow π from the initial state to an end state (multiple times);
- Keep track of the number of times we visited each state;
- Update the utility of the states as we go, using the following algorithm:

if s' is new **then** $U[s'] \leftarrow r'$
if s is not null **then**
  increment $N_s[s]$
  $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

Passive Temporal Difference Learning

- We start with a policy π
- $U(s)$ — Utility for s (init. null)
- $N(s)$ — # times we visited s
- α — Learning rate, e.g. $\frac{1}{N[s]+1}$



Algorithm consists of:

- Follow π from the initial state to an end state (multiple times);
- Keep track of the number of times we visited each state;
- Update the utility of the states as we go, using the following algorithm:

if s' is new **then** $U[s'] \leftarrow r'$

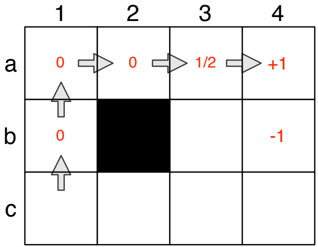
if s is not null **then**

increment $N_s[s]$

$U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

Passive Temporal Difference Learning

- We start with a policy π
- $U(s)$ — Utility for s (init. null)
- $N(s)$ — # times we visited s
- α — Learning rate, e.g. $\frac{1}{N[s]+1}$



Algorithm consists of:

- Follow π from the initial state to an end state (multiple times);
- Keep track of the number of times we visited each state;
- Update the utility of the states as we go, using the following algorithm:

if s' is new **then** $U[s'] \leftarrow r'$

if s is not null **then**

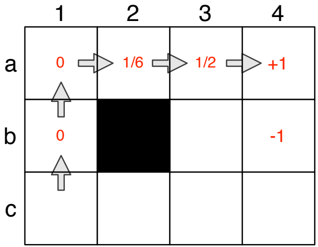
increment $N_s[s]$

$U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

$U[a3] \leftarrow 0 + 1/2(0 + 1 - 0)$ — for $\gamma = 1$

Passive Temporal Difference Learning

- We start with a policy π
- $U(s)$ — Utility for s (init. null)
- $N(s)$ — # times we visited s
- α — Learning rate, e.g. $\frac{1}{N[s]+1}$



Algorithm consists of:

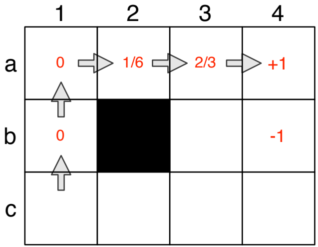
- Follow π from the initial state to an end state (multiple times);
- Keep track of the number of times we visited each state;
- Update the utility of the states as we go, using the following algorithm:

if s' is new **then** $U[s'] \leftarrow r'$
if s is not null **then**
increment $N_s[s]$
$U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

$U[a2] \leftarrow 0 + 1/3(0 + 1/2 - 0)$ — for $\gamma = 1$

Passive Temporal Difference Learning

- We start with a policy π
- $U(s)$ — Utility for s (init. null)
- $N(s)$ — # times we visited s
- α — Learning rate, e.g. $\frac{1}{N[s]+1}$



Algorithm consists of:

- Follow π from the initial state to an end state (multiple times);
- Keep track of the number of times we visited each state;
- Update the utility of the states as we go, using the following algorithm:

if s' is new **then** $U[s'] \leftarrow r'$

if s is not null **then**

increment $N_s[s]$

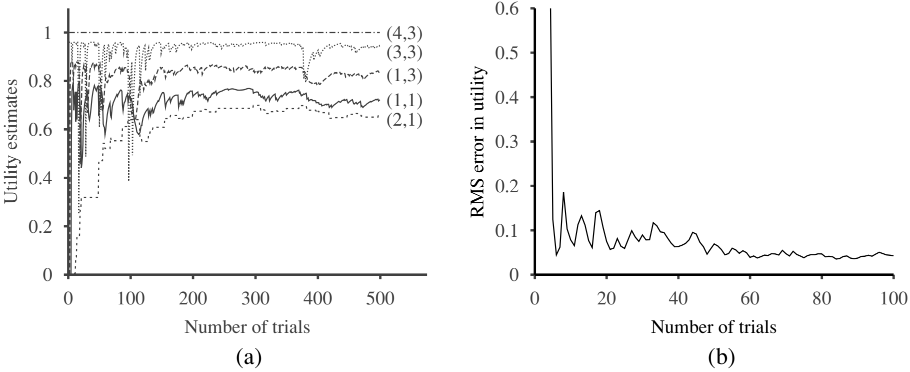
$U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

$U[a3] \leftarrow 1/2 + 1/3(0 + 1 - 1/2)$ — for $\gamma = 1$

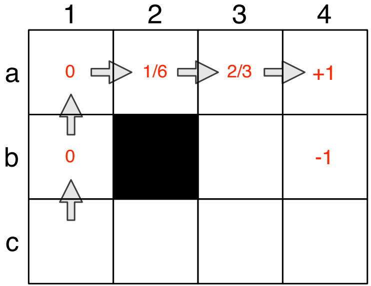
function PASSIVE-TD-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
    U, a table of utilities, initially empty
    N_s, a table of frequencies for states, initially zero
    s, a, r, the previous state, action, and reward, initially null

  if s' is new **then** $U[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$
  if s'.TERMINAL? **then** $s, a, r \leftarrow$ null **else** $s, a, r \leftarrow s', \pi[s'], r'$
  return a

Passive Agent Results

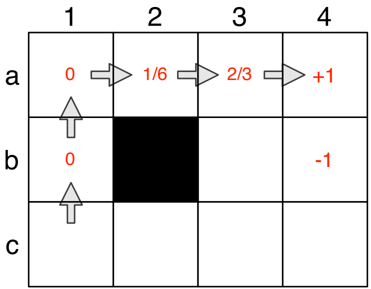


Problems with Passive RL



T	F	
		Long Convergence
		Limited by Policy
		Missing States
		Poor Estimate

Problems with Passive RL



T	F	
X		Long Convergence
X		Limited by Policy
X		Missing States
X		Poor Estimate

Active Greedy TD

- Active reinforcement learning seeks to use the reward information learned by the passive algorithm to generate a new policy

Active Greedy TD

- Active reinforcement learning seeks to use the reward information learned by the passive algorithm to generate a new policy
- **Active Greedy** Temporal Difference learning

Active Greedy TD

- Active reinforcement learning seeks to use the reward information learned by the passive algorithm to generate a new policy
- **Active Greedy** Temporal Difference learning
  - Works exactly like passive TD learning, but

Active Greedy TD

- Active reinforcement learning seeks to use the reward information learned by the passive algorithm to generate a new policy
- **Active Greedy** Temporal Difference learning
  - Works exactly like passive TD learning, but
  - After a certain number of iterations, it recomputes a new policy π

Active Greedy TD

- Active reinforcement learning seeks to use the reward information learned by the passive algorithm to generate a new policy
- **Active Greedy** Temporal Difference learning
  - Works exactly like passive TD learning, but
  - After a certain number of iterations, it recomputes a new policy $\not\pi \rightarrow \pi_2$
  - This new policy is computed by **solving the MDP** using the estimated utilities

Active Greedy TD

- Active reinforcement learning seeks to use the reward information learned by the passive algorithm to generate a new policy
- **Active Greedy** Temporal Difference learning
  - Works exactly like passive TD learning, but
  - After a certain number of iterations, it recomputes a new policy $\not\pi \rightarrow \pi_2$
  - This new policy is computed by **solving the MDP** using the estimated utilities

$$\pi(s) = \arg\max_a \sum_{s'} P(s'|s,a) * V(s')$$

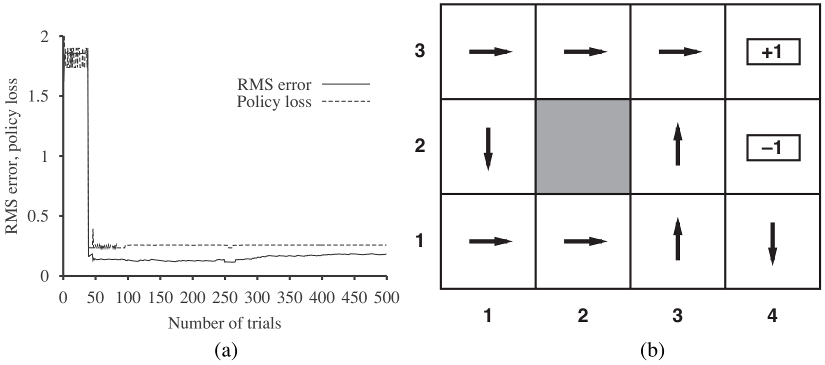
  - Use the new policy instead of the old one for passive TD

Active Greedy TD

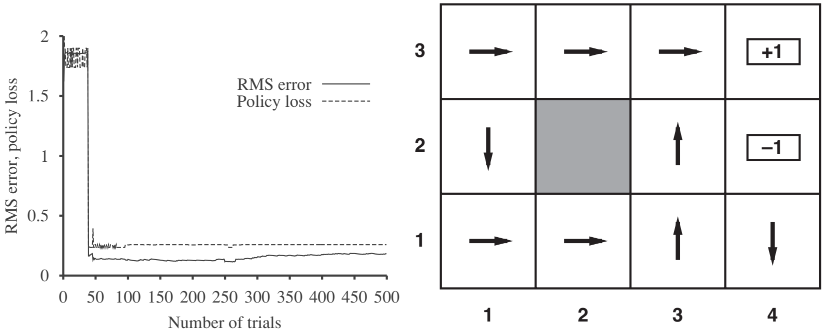
- Active reinforcement learning seeks to use the reward information learned by the passive algorithm to generate a new policy
- **Active Greedy** Temporal Difference learning
  - Works exactly like passive TD learning, but
  - After a certain number of iterations, it recomputes a new policy $\not\pi \rightarrow \pi_2$
  - This new policy is computed by **solving the MDP** using the estimated utilities

$$\pi(s) = \arg\max_a \sum_{s'} P(s'|s,a) * U(s')$$

  - Use the new policy instead of the old one for passive TD



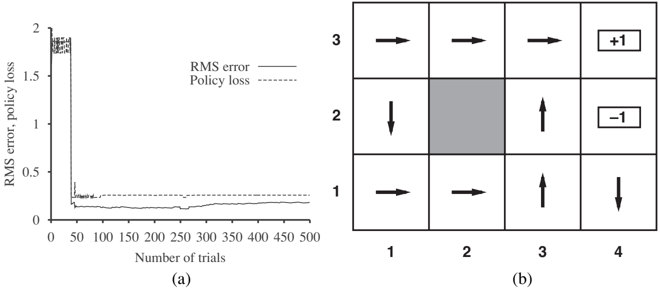
Is this policy optimal?



(a) (b)

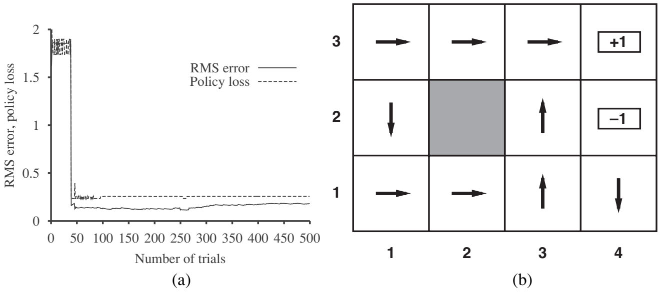
Is this policy optimal? No

Exploration vs. Exploitation



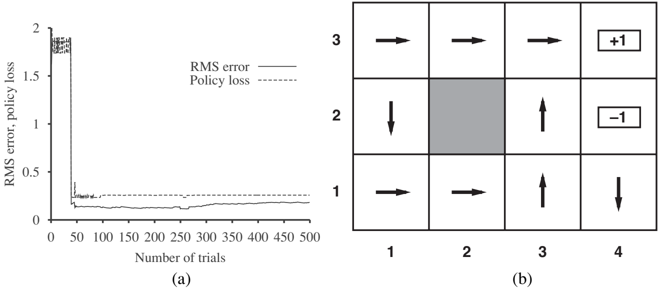
- Even after changing policy, we are still stuck to something close to the original policy

Exploration vs. Exploitation



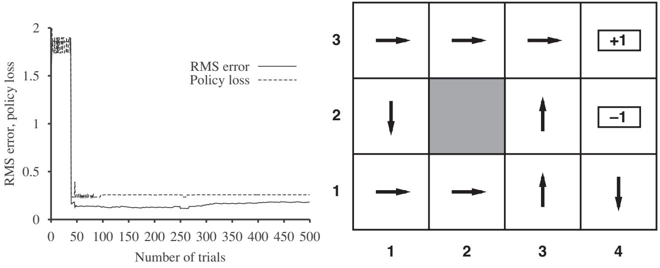
- Even after changing policy, we are still stuck to something close to the original policy
- So we need to balance:

Exploration vs. Exploitation



- Even after changing policy, we are still stuck to something close to the original policy
- So we need to balance:
  - exploiting the values we discovered

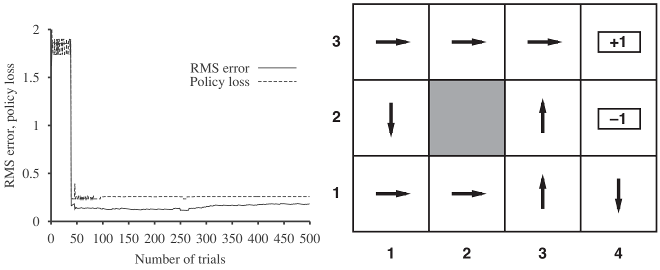
Exploration vs. Exploitation



(a) (b)

- Even after changing policy, we are still stuck to something close to the original policy
- So we need to balance:
  - exploiting the values we discovered
  - explore different actions to discover new opportunities

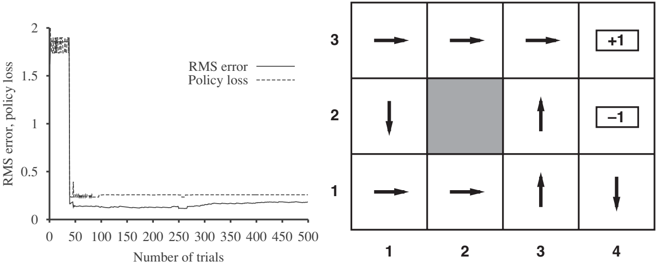
Exploration vs. Exploitation



(a) (b)

- Even after changing policy, we are still stuck to something close to the original policy
- So we need to balance:
  - exploiting the values we discovered
  - explore different actions to discover new opportunities

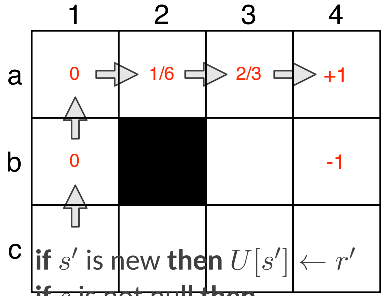
Exploration vs. Exploitation



(a) (b)

- Even after changing policy, we are still stuck to something close to the original policy
- So we need to balance:
  - exploiting the values we discovered
  - explore different actions to discover new opportunities
    one possibility is to choose random actions occasionally

Errors in Utility



if s' is new **then** $U[s'] \leftarrow r'$

if s is not null **then**

increment $N_s[s]$

$U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

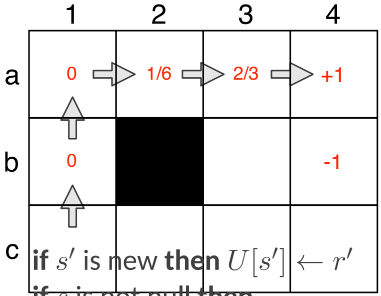
Possible problems with these methods

- We might not have sampled enough N
- We might have started with a very poor policy π

What can happen if we vary the following parameters?

Sampling		Policy		
T	F	T	F	
				Underestimate U
				Overestimate U
				Can we improve with higher N

Errors in Utility



if s' is new **then** $U[s'] \leftarrow r'$
if s is not null **then**
increment $N_s[s]$
$U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

Possible problems with these methods

- We might not have sampled enough N
- We might have started with a very poor policy π

What can happen if we vary the following parameters?

Sampling		Policy		
T	F	T	F	
X		X		Underestimate U
X			X	Overestimate U
X			X	Can we improve with higher N

if s' is new **then** $U[s'] \leftarrow r'$
if s is not null **then**
  increment $N_s[s]$
  $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

- Mitigate the negative effects of having explored too little using an optimistic estimate function

if s' is new **then** $U[s'] \leftarrow r'$
if s is not null **then**
increment $N_s[s]$
$U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

- Mitigate the negative effects of having explored too little using an optimistic estimate function

if s' is new **then** $U[s'] \leftarrow r'$

if s is not null **then**

  increment $N_s[s]$

  $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$

- Mitigate the negative effects of having explored too little using an optimistic estimate function

$$f(u,n) = \begin{cases} R^{+} & if\, n < N_e \\ u & otherwise \end{cases}$$

  where:
  - N_e is an exploration threshold, and
  - R^{+} is the best reward we expect to receive
- With this update, we avoid underestimating a state until we have explored the domain sufficiently

if s' is new **then** $U[s'] \leftarrow r'$

if s is not null **then**

increment $N_s[s]$

$U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - f(U[s], N_s[s]))$

- Mitigate the negative effects of having explored too little using an optimistic estimate function

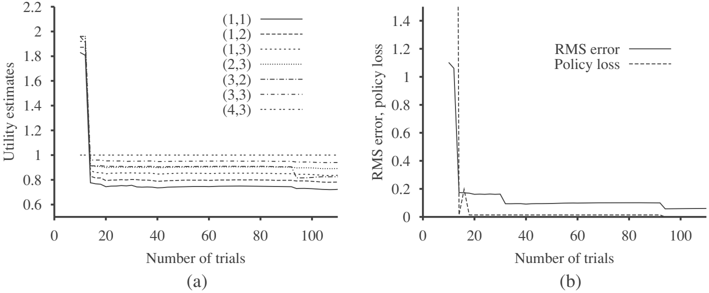
$$f(u,n) = \begin{cases} R^+ & if\, n < N_e \\ u & otherwise \end{cases}$$

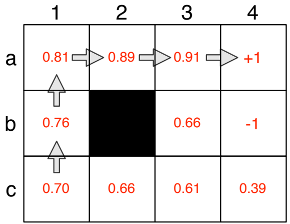
where:

- N_e is an exploration threshold, and
- R^+ is the best reward we expect to receive

- With this update, we avoid underestimating a state until we have explored the domain sufficiently

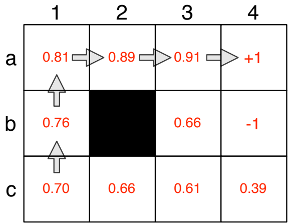
Exploration Agent Result





- Once we have learned the utility of the states, we simply apply the Bellman equation to select the best policy:

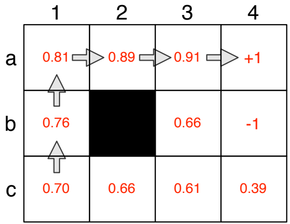
$$\pi * (s) = \arg\max_a \sum_{s'} P(s'|s,a) * U(s')$$



- Once we have learned the utility of the states, we simply apply the Bellman equation to select the best policy:

$$\pi * (s) = \arg\max_a \cancel{\sum_{s'} P(s'|s,a) * U(s')}$$

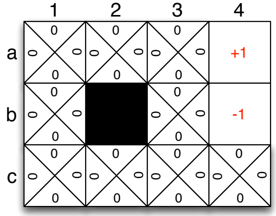
- But what if we do not know the transition model $P(s'|s,a)$?



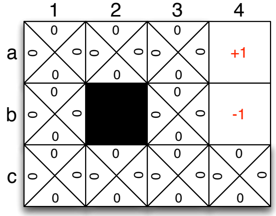
- Once we have learned the utility of the states, we simply apply the Bellman equation to select the best policy:

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

- But what if we do not know the transition model $P(s'|s, a)$?
- We can use a method called Q learning, that learns a different value $Q(s, a)$, from which we can derive the optimal policy

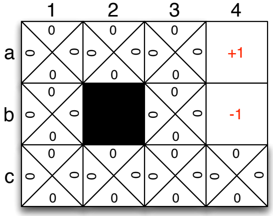


- Instead of storing rewards for each state, we store rewards for each action we took at each state
  this is Q value $Q(s, a)$



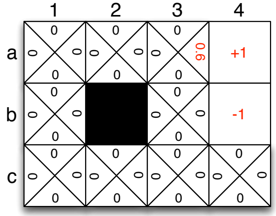
- Instead of storing rewards for each state, we store rewards for each action we took at each state
  this is Q value $Q(s, a)$
- The update algorithm is quite similar to TD, but with the following update function:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$



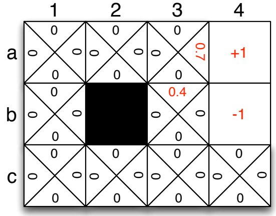
- Instead of storing rewards for each state, we store rewards for each action we took at each state
  this is Q value $Q(s, a)$
- The update algorithm is quite similar to TD, but with the following update function:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$



- Instead of storing rewards for each state, we store rewards for each action we took at each state
  this is Q value $Q(s, a)$
- The update algorithm is quite similar to TD, but with the following update function:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s,a))$$



- Instead of storing rewards for each state, we store rewards for each action we took at each state
  this is Q value $Q(s, a)$
- The update algorithm is quite similar to TD, but with the following update function:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

function Q-LEARNING-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r'
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, r, the previous state, action, and reward, initially null

  if TERMINAL?(s) **then** $Q[s, None] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$
    $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
  $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a']), r'$
  return a

- Exploratory Q-learning agent: same exploration function as TD-learning
- Computes the best action at each call through $\arg\max$

- SARSA – State-Action-Reward-State-Action is a close relative to Q-Learning
- Algorithm is exactly the same, but the update rule is slightly different

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a))$$

  - Where a' – action actually taken in s' (notice $\max$ is gone)
- Subtle differences:

- SARSA – State-Action-Reward-State-Action is a close relative to Q-Learning
- Algorithm is exactly the same, but the update rule is slightly different

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a))$$

  - Where a' – action actually taken in s' (notice $\max$ is gone)
- Subtle differences:
  - Q-learning backs up the best action (**off-policy**)

- SARSA – State-Action-Reward-State-Action is a close relative to Q-Learning
- Algorithm is exactly the same, but the update rule is slightly different

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a))$$

  - Where a' – action actually taken in s' (notice $\max$ is gone)
- Subtle differences:
  - Q-learning backs up the best action (**off-policy**)
  - SARSA backs up the action actually taken (**on-policy**)

- SARSA – State-Action-Reward-State-Action is a close relative to Q-Learning
- Algorithm is exactly the same, but the update rule is slightly different

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a))$$

  - Where a' – action actually taken in s' (notice $\max$ is gone)
- Subtle differences:
  - Q-learning backs up the <u>best</u> action (**off-policy**)
  - SARSA backs up the action actually taken (**on-policy**)
  - Algorithms are **identical** when there is **no exploration**

Outline

- TD Learning
- Q Learning
- Feature Generalization

- TD-RL and Q-learning use a monolithic lookup table
  - Will work for 2D maze-like environments, and up to ≈ 10000 states
  - Not good enough for, e.g. chess, backgammon ($10^{20} - -10^{40}$ states)

- TD-RL and Q-learning use a monolithic lookup table
  - Will work for 2D maze-like environments, and up to ≈ 10000 states
  - Not good enough for, e.g. chess, backgammon ($10^{20} - -10^{40}$ states)
- We could generalize using **function approximation**
  - Use an alternative representation for the Q-function, e.g. a linear function of features:

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s)$$

  - For direct utility estimation: exactly like **supervised learning**

- TD-RL and Q-learning use a monolithic lookup table
  - Will work for 2D maze-like environments, and up to ≈ 10000 states
  - Not good enough for, e.g. chess, backgammon ($10^{20} - -10^{40}$ states)
- We could generalize using **function approximation**
  - Use an alternative representation for the Q-function, e.g. a linear function of features:

$$\hat{U}_{\theta}(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s)$$

  - For direct utility estimation: exactly like **supervised learning**
- More recently, use a deep neural network to approximate Q
  - Not exactly like supervised learning

- For example, consider the utilities for our 4×3 grid, its only features are the x and y coordinates, so:

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

Generalization with Features

- For example, consider the utilities for our 4×3 grid, its only features are the x and y coordinates, so:

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

If $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$, then $\hat{U}_\theta(1, 1) = 0.8$

- For example, consider the utilities for our 4×3 grid, its only features are the x and y coordinates, so:

$$\hat{U}_{\theta}(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

If $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$, then $\hat{U}_{\theta}(1, 1) = 0.8$
Suppose we run a trial and obtain $\hat{U}_{\theta}(1, 1) = 0.4$,
then we need to correct for error

- For example, consider the utilities for our 4×3 grid, its only features are the x and y coordinates, so:
$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$
If $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$, then $\hat{U}_\theta(1, 1) = 0.8$
Suppose we run a trial and obtain $\hat{U}_\theta(1, 1) = 0.4$,
then we need to correct for error
- Let $u_j(s)$ be the observed utility of s at trial j
With an error function $E_j(s) = (\hat{U}_\theta(s) - u_j(s))^2/2$, we can move each parameter θ_i following the rate of change defined by $\delta E_j/\delta\theta_i$, using:

$$\theta_i \leftarrow \theta_i - \alpha \frac{\delta E_j}{\delta \theta_i} = \theta_i - \alpha(u_j(s) - \hat{U}_\theta(s)) \frac{\delta \hat{U}_\theta(s)}{\delta \theta_i}$$

Generalization with Features (Update Rules)

$$\theta_i \leftarrow \theta_i - \alpha(u_j(s) - \hat{U}_\theta(s))\frac{\delta \hat{U}_\theta(s)}{\delta \theta_i}$$

- This is called the Widrow-Hoff rule or delta rule for online least squares
- If we apply this rule to our linear function $\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$, we obtain three simple update rules

$$\begin{aligned}
\theta_0 &\leftarrow \theta_0 + \alpha(u_j(s) - \hat{U}_\theta(s)), \\
\theta_1 &\leftarrow \theta_1 + \alpha(u_j(s) - \hat{U}_\theta(s))x, \\
\theta_2 &\leftarrow \theta_2 + \alpha(u_j(s) - \hat{U}_\theta(s))y.
\end{aligned}$$

- So, for every transition we make in a single state, we update the utilities of **every other state**

Generalization with Features (TD Updates)

- Given a collection of samples, we can estimate the utility of some states without ever visiting them
- We can apply this same principle for TD-RL
- And for Q-learning:

Generalization with Features (TD Updates)

- Given a collection of samples, we can estimate the utility of some states without ever visiting them
- We can apply this same principle for TD-RL

$$\theta_i \leftarrow \theta_i + \alpha \left[R(s) + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s) \right] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

- And for Q-learning:

Generalization with Features (TD Updates)

- Given a collection of samples, we can estimate the utility of some states without ever visiting them
- We can apply this same principle for TD-RL

$$\theta_i \leftarrow \theta_i + \alpha \left[R(s) + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s) \right] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

- And for Q-learning:

$$\theta_i \leftarrow \theta_i + \alpha \left[R(s) + \gamma \max \hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a) \right] \frac{\partial \hat{Q}_\theta(s, a)}{\partial \theta_i}$$

Convergence of Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

Convergence of Prediction Algorithms

- TD does not follow the gradient of any objective function
- This is why TD can diverge when off-policy or using non-linear function approximation
- **Gradient TD** follows true gradient of projected Bellman error

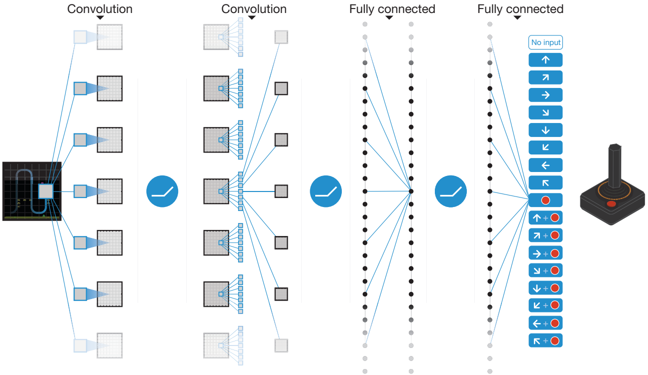
On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD	✓	✓	✗
	Gradient TD	✓	✓	✓
Off-Policy	MC	✓	✓	✓
	TD	✓	✗	✗
	Gradient TD	✓	✓	✓

Convergence of Control Algorithms (Active)

Algorithm	Table Lookup	Linear	Non-Linear
MC-Control	✓	(✓)	✗
SARSA	✓	(✓)	✗
Q-Learning	✓	✗	✗
Gradient Q-Learning	✓	✓	✗

(✓) = chatters around near-optimal value function

Deep Q-Networks (DQN) was one of the first such NN-based function approximations proven at scale:
Mnih, Volodymyr, et al. **Human-level control through deep reinforcement learning.** Nature 518.7540 (2015): 529.



Reinforcement Learning Summary

- How to solve an MDP without P and R, using interaction
  - TD-learning
  - Q-Learning (a family of algorithms)
- The balance between exploration and exploitation
  - Idea: instead of exploration function, use $\hat{Q} \leftarrow$ awesome
  - Start really optimistic and exploration will only drive values down
- How to learn the utility of individual features

Any Questions.