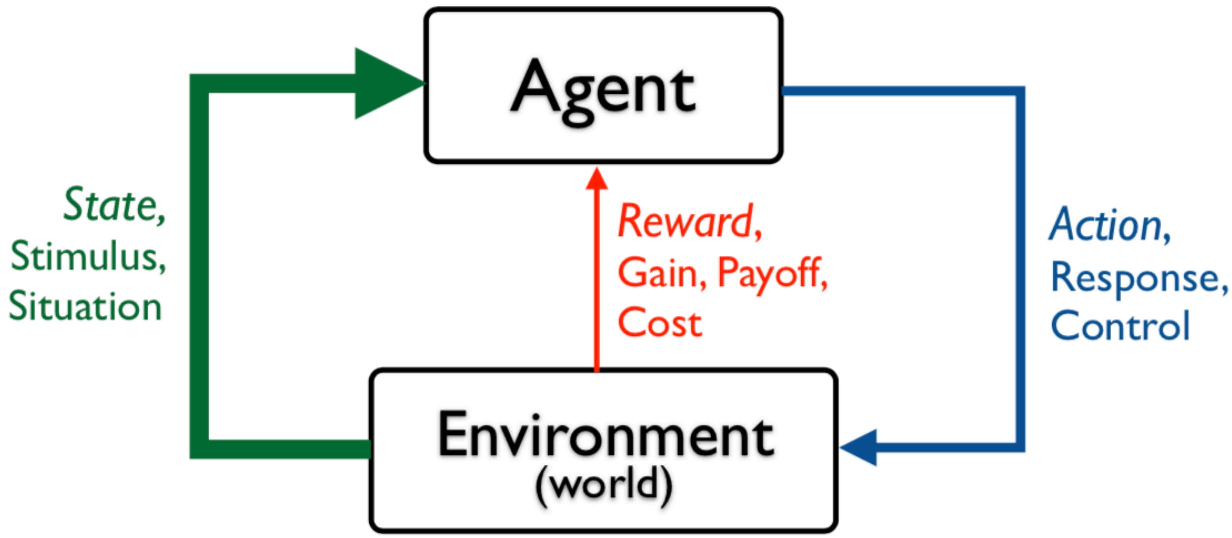
Day 3:
Reinforcement Learning

Questions about Monday's session?

- Utility theory
- Bandits
- MDPs
- Policy evaluation
- Policy iteration and value iteration

The RL interface



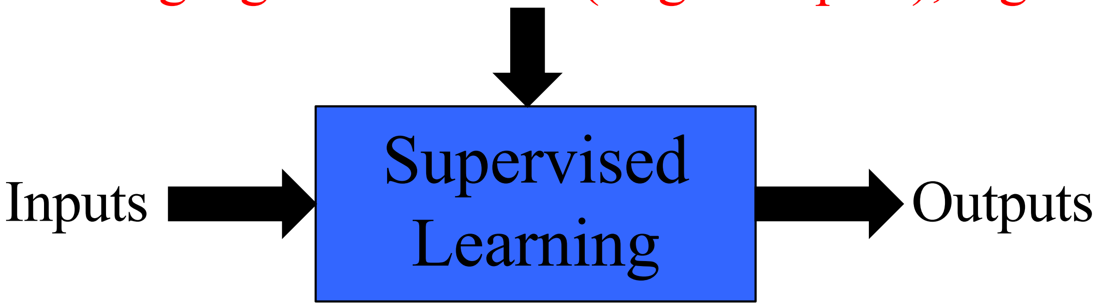
- Environment may be unknown, nonlinear, stochastic and complex
- Agent learns a policy mapping states to actions
  - Seeking to maximize its cumulative reward in the long run

When to use RL?

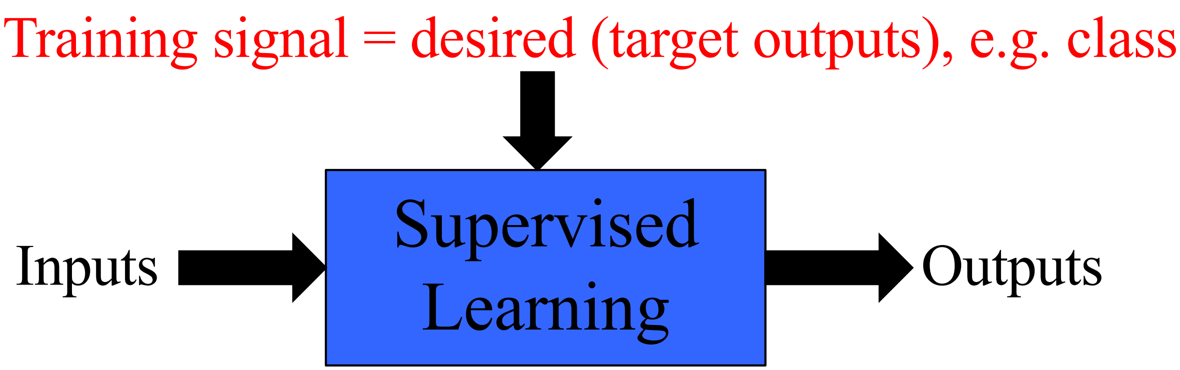
- Data in the form of trajectories.
- Need to make a sequence of (related) decisions.
- Observe (partial, noisy) feedback to state or choice of actions.
- There is a gain when optimizing action choice over a portion of the trajectory.

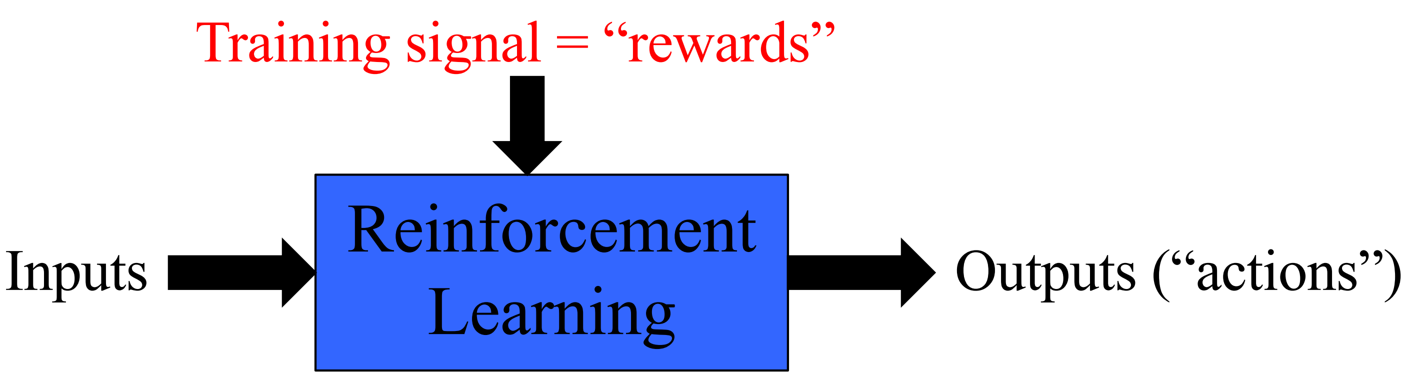
RL vs supervised learning

Training signal = desired (target outputs), e.g. class

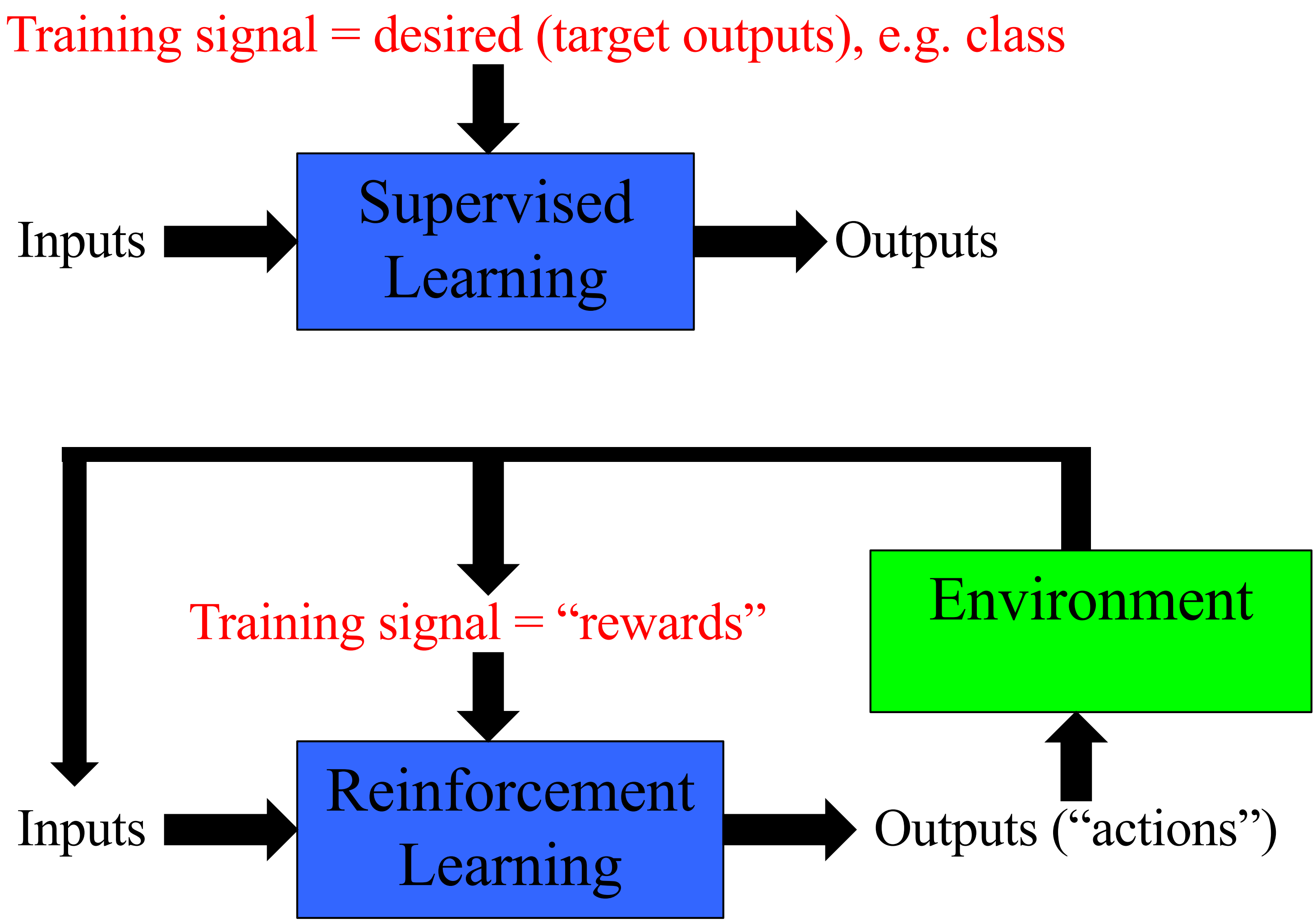


RL vs supervised learning

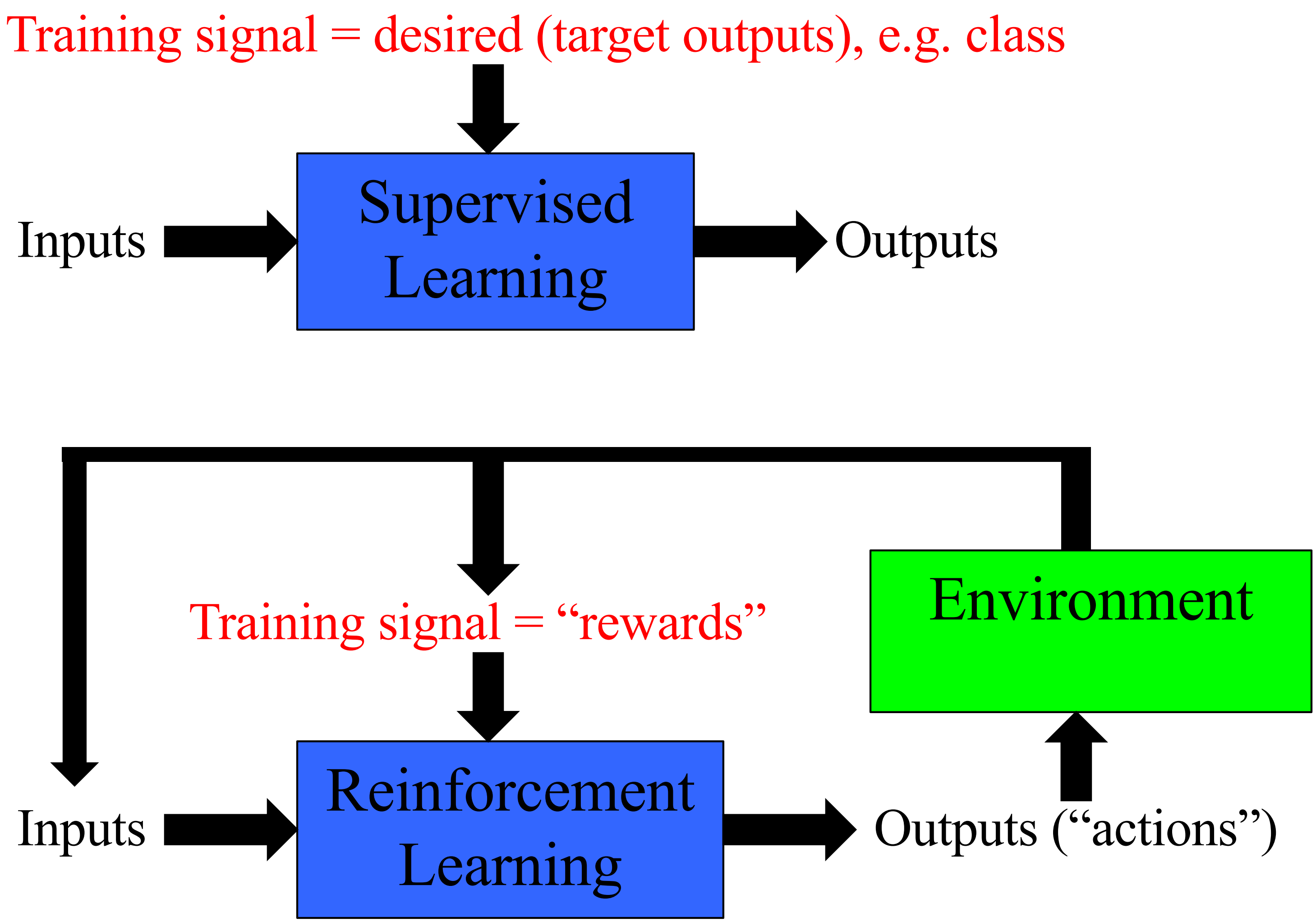




RL vs supervised learning



RL vs supervised learning



Challenges:

1. Need access to the environment.

2. Jointly learning AND planning from **correlated** samples.

3. Data distribution changes with action choice.

Learning the MDP model directly

- Suppose you have a policy for acquiring data (e.g. random exploration).
- Observe many transitions in the environment: $\langle s, a, r, s' \rangle$

Learning the MDP model directly

- Suppose you have a policy for acquiring data (e.g. random exploration).
- Observe many transitions in the environment: *<s, a, r, s'>*
- Learn an approximate model, *R(s,a), T(s,a,s')*.
  1. Use maximum likelihood to compute probabilities.
  2. Use supervised learning for the rewards.

Learning the MDP model directly

- Suppose you have a policy for acquiring data (e.g. random exploration).
- Observe many transitions in the environment: *<s, a, r, s'>*
- Learn an <u>approximate model</u>, *R(s,a), T(s,a,s')*.
  1. Use maximum likelihood to compute probabilities.
  2. Use supervised learning for the rewards.
- Pretend the approximate model is correct and use it for any dynamic programming method (value/policy iteration).
  - This approach is called <u>*model-based reinforcement learning*</u>.
  - Extensively used, especially in the robotics community.

Learning the MDP model directly

- Suppose you have a policy for acquiring data (e.g. random exploration).
- Observe many transitions in the environment: *<s, a, r, s'>*
- Learn an approximate model, $R(s,a)$, $T(s,a,s')$.
  1. Use maximum likelihood to compute probabilities.
  2. Use supervised learning for the rewards.
- Pretend the approximate model is correct and use it for any dynamic programming method (value/policy iteration).
  - This approach is called *model-based reinforcement learning*.
  - Extensively used, especially in the robotics community.

Can we avoid learning an approximate model?

Monte Carlo Methods

- Suppose we have an episodic task: the agent interacts with the environment in trials or episodes, which terminate at some point.

  E.g. Game playing.

- The agent behaves according to some policy π for a while, generating several trajectories.

How can we compute V^{π}?

Monte Carlo Methods

- Suppose we have an episodic task: the agent interacts with the environment in trials or episodes, which terminate at some point.
  E.g. Game playing.

- The agent behaves according to some policy π for a while, generating several trajectories.

How can we compute V^{π}?

- Compute $V^{\pi}(s)$ by averaging the observed returns after s on the trajectories in which s was visited.

Implementation of Monte Carlo Policy Evaluation

- Let U_i be the observed utility from state s for the i-th trajectory.
- Let $V_{n+1}(s)$ be the estimate of the value from some state s after observing $n+1$ trajectories starting at s.

$$V_{n+1}(s) = \frac{1}{n+1}\sum_{i=1}^{n+1} U_i(s)$$

Implementation of Monte Carlo Policy Evaluation

- Let U_i be the observed utility from state s for the i-th trajectory.
- Let $V_{n+1}(s)$ be the estimate of the value from some state s after observing $n+1$ trajectories starting at s.

$$V_{n+1}(s) = \frac{1}{n+1}\sum_{i=1}^{n+1} U_i(s)$$

$$= \frac{1}{n+1}\left(\sum_{i=1}^{n} U_i(s) + U_{n+1}(s)\right)$$

Implementation of Monte Carlo Policy Evaluation

- Let U_i be the observed utility from state s for the i-th trajectory.
- Let $V_{n+1}(s)$ be the estimate of the value from some state s after observing $n+1$ trajectories starting at s.

$$
\begin{aligned}
V_{n+1}(s) &= \frac{1}{n+1}\sum_{i=1}^{n+1} U_i(s) \\
&= \frac{1}{n+1}\left(\sum_{i=1}^{n} U_i(s) + U_{n+1}(s)\right) \\
&= \frac{n}{n+1}\frac{1}{n}\sum_{i=1}^{n} U_i(s) + \frac{1}{n+1}U_{n+1}(s) \\
&= \frac{n}{n+1}V_n(s) + \frac{1}{n+1}U_{n+1}(s)
\end{aligned}
$$

Implementation of Monte Carlo Policy Evaluation

- Let U_i be the observed utility from state s for the i-th trajectory.
- Let $V_{n+1}(s)$ be the estimate of the value from some state s after observing $n+1$ trajectories starting at s.

$$
\begin{aligned}
V_{n+1}(s) &= \frac{1}{n+1}\sum_{i=1}^{n+1} U_i(s) \\
&= \frac{1}{n+1}\left(\sum_{i=1}^{n} U_i(s) + U_{n+1}(s)\right) \\
&= \frac{n}{n+1}\frac{1}{n}\sum_{i=1}^{n} U_i(s) + \frac{1}{n+1}U_{n+1}(s) \\
&= \frac{n}{n+1}V_n(s) + \frac{1}{n+1}U_{n+1}(s) \\
&= V_n(s) + \frac{1}{n+1}\left(U_{n+1}(s) - V_n(s)\right)
\end{aligned}
$$

Monte Carlo Policy Evaluation - Reducing memory

- Monte Carlo policy evaluation requires keeping a count of how many times states have been visited. Can we avoid this?

- Instead, use a *learning rate* version:

$$V(s_t) \leftarrow V(s_t) + \alpha\left(U(s_t) - V(s_t)\right)$$

Temporal-Difference (TD) Prediction

- Monte Carlo uses as a target estimate for the value function the actual return, U_t:

$$V(s_t) \leftarrow V(s_t) + \alpha\left(U(s_t) - V(s_t)\right)$$

Temporal-Difference (TD) Prediction

- Monte Carlo uses as a target estimate for the value function the actual return, U_t:

$$V(s_t) \leftarrow V(s_t) + \alpha\big(U(s_t) - V(s_t)\big)$$

- The TD method uses instead an **estimate** of the return:

$$V(s_t) \leftarrow V(s_t) + \alpha\big(r_t + \gamma V(s_{t+1}) - V(s_t)\big)$$

  – Don't need to keep track of $U(s_t)$.
  – If $V(s_{t+1})$ were correct, this would be a dynamic programming target.

Comments on TD

- TD is a hybrid between dynamic programming (DP) and Monte Carlo (MC) evaluation.

- Like DP, TD *bootstraps* (computes the value of a state based on estimates of the successors).

- Like MC, TD estimates expected values by looking at *samples*.

TD Learning Algorithm

Initialize the value function, $V(s)=0, \ \forall s$

Repeat as many times as wanted:

(a) Pick a start state s for the current trial.

(b) Repeat for every time step t:

TD Learning Algorithm

Initialize the value function, $V(s)=0, \ \forall s$

Repeat as many times as wanted:

(a) Pick a start state s for the current trial.

(b) Repeat for every time step t:

   i. Choose action a based on policy π and the current state s.

   ii. Take action a, observed reward r and new state s'.

TD Learning Algorithm

Initialize the value function, $V(s)=0, \ \forall s$

Repeat as many times as wanted:

(a) Pick a start state s for the current trial.

(b) Repeat for every time step t:

i. Choose action a based on policy π and the current state s.

ii. Take action a, observed reward r and new state s'.

iii. Compute the TD error: $\delta \leftarrow r + \gamma V(s')-V(s)$

iv. Update the value function: $V(s) \leftarrow V(s) + \alpha_s \delta$

TD Learning Algorithm

Initialize the value function, $V(s)=0, \ \forall s$

Repeat as many times as wanted:

(a) Pick a start state s for the current trial.

(b) Repeat for every time step t:

i. Choose action a based on policy π and the current state s.

ii. Take action a, observed reward r and new state s'.

iii. Compute the TD error: $\delta \leftarrow r + \gamma V(s')-V(s)$

iv. Update the value function: $V(s) \leftarrow V(s) + \alpha_s \delta$

v. $s \leftarrow s'$

vi. If s' is not a terminal state, go to step (b).

Example

- Suppose you have a system with 2 states (A and B), you initially assume *V(A)=V(B)=0,* then observe (only) the following 6 episodes:
    1. B, 1
    2. B, 1
    3. B, 1
    4. B, 1
    5. B, 0
    6. A, 0; B (reward not seen yet)

What would you predict for *V(B)*? What would you predict for *V(A)*?

Example

- Suppose you have a system with 2 states (A and B), you initially assume *V(A)=V(B)=0,* then observe (only) the following 6 episodes:
  1. B, 1
  2. B, 1
  3. B, 1
  4. B, 1
  5. B, 0
  6. A, 0; B (reward not seen yet)

What would you predict for *V(B)*? What would you predict for *V(A)*?

  – V(B) = 4/5   (That's easy!)

Example

- Suppose you have a system with 2 states (A and B), you initially assume *V(A)=V(B)=0,* then observe (only) the following 6 episodes:
  1. B, 1
  2. B, 1
  3. B, 1
  4. B, 1
  5. B, 0
  6. A, 0; B (reward not seen yet)

What would you predict for *V(B)*? What would you predict for *V(A)*?

  – V(B) = 4/5 (That's easy!)
  – V(A) = 0 if you use Monte-Carlo (Haven't seen the return for trajectory 6 yet.)
  – V(A) = 0 + 4/5 if you use TD (Can use estimate of *V(B)*.)

Example (continued)

- Suppose you have a system with 2 states (A and B), you initially assume *V(A)=V(B)=0,* then observe (only) the following 6 episodes:
    1. B, 1
    2. B, 1
    3. B, 1
    4. B, 1
    5. B, 0
    6. A, 0; B 0

What would you predict for *V(B)*? What would you predict for *V(A)*?

– V(B) = 2/3 (Revised estimate.)

Summary: Methods of value prediction

- **Monte Carlo**:
  - This minimizes the sum-squared error on the training data.
  - In our example, we would predict *V(A)=0.*

- **Learning a model, then doing dynamic programming**:
  - Estimate a model from the data, then use this to compute the value.
  - In our example, we would estimate that *A* goes to *B* w/Pr=1, so *V(A)=0+4/6*.

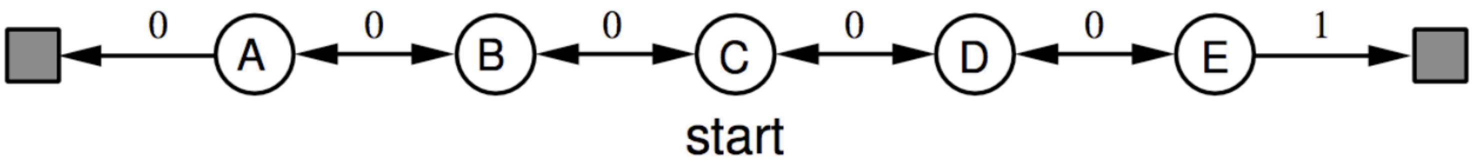
- **Temporal difference (TD):**
  - TD is a gradient algorithm: it adjusts the values based on current estimates of other values.
  - In our example, adjust V(*A)* towards current estimate for *B* (before the continuation from *B* is seen), so *V(A)=0+4/5*.
  - This is closer to dynamic programming than Monte Carlo.
  - TD estimates take into account *time sequence*.

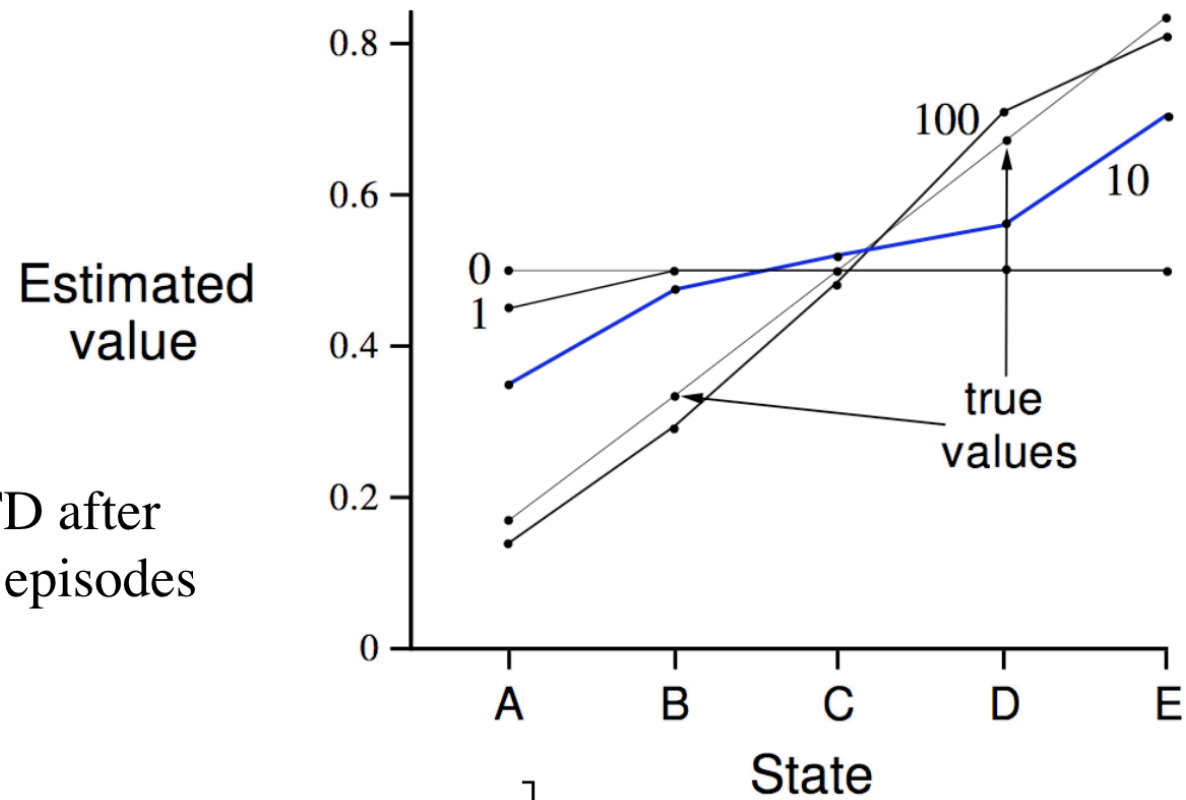
Pause

Advantages

- No model of the environment is required! TD only needs experience with the environment.
- MC methods have lower on past data, but higher error on future data.
- On-line, incremental learning:
  - Can learn before knowing the final outcome.
  - Less memory and peak computation are required.
- Both TD and MC converge (under mild assumptions), but TD usually learns faster.

Random walk example

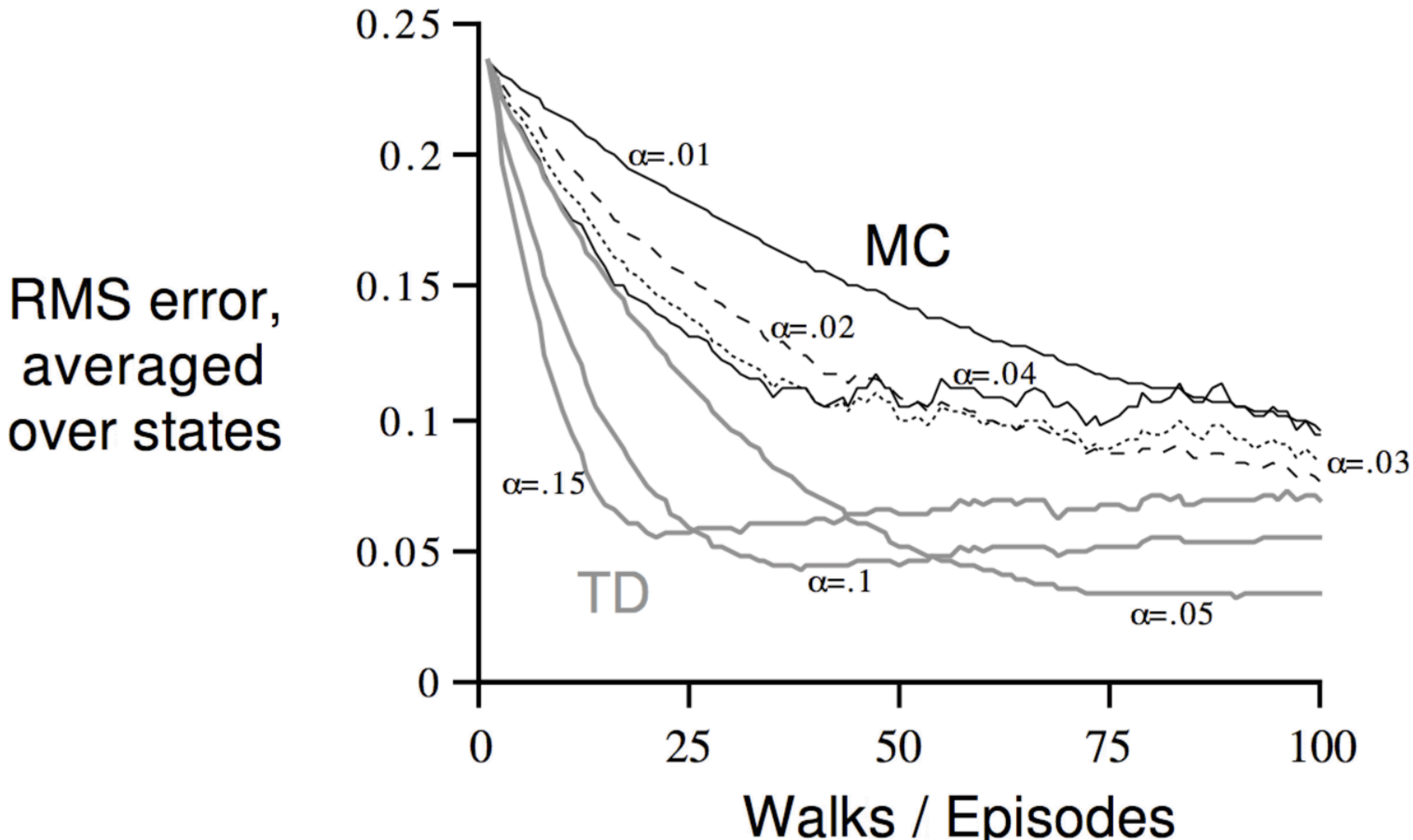




Values learned by TD after various numbers of episodes

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

TD and MC on the random walk example



Data averaged over
100 sequences of episodes

n-step TD

- Consider the *n*-step return:

$$G_t^{(n)} = r_t + \gamma r_{t+1} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^{n+1} V_{t+n}(s_{t+n})$$

- Of course this is <u>not available until time</u> $t+n$.

- The natural algorithm is thus to wait until then:

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha[G_t^{(n)} - V_{t+n-1}(S_t)\ ]$$

- This is called *n*-step TD.

Batch updating in TD and MC

Batch Updating: train completely on a finite amount of data, e.g., train repeatedly on 10 episodes until convergence.

Compute updates according to TD or MC, but only update estimates after each complete pass through the data.

For any finite Markov prediction task, under batch updating, TD converges for sufficiently small α.

Constant-α MC also converges under these conditions, but to a different answer!

Propagating value updates with TD

- Back to our simple example, you observed:
  1. B, 1
  2. B, 1
  3. B, 1
  4. B, 1
  5. B, 0
  6. A, 0; B (reward not seen yet)

  And estimated *V(B)*=4/5,

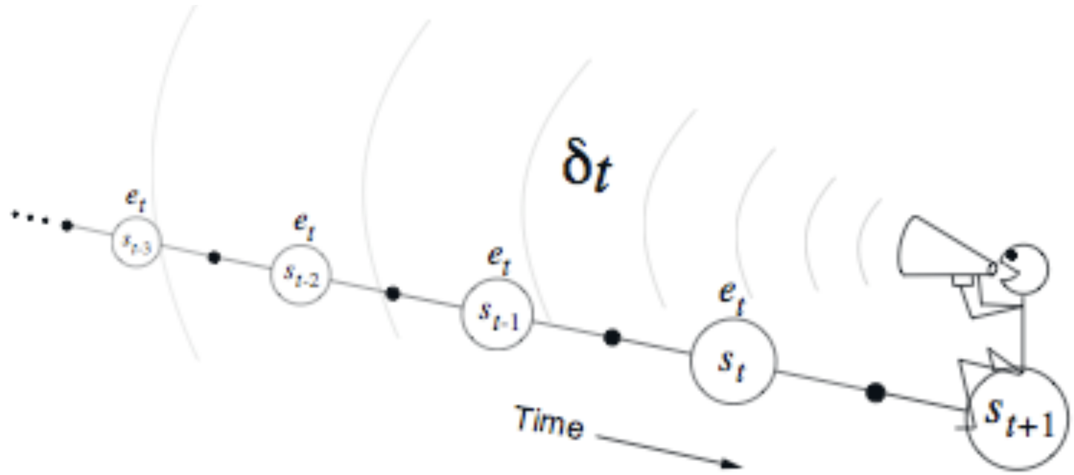
- Suppose you then see:
  7. A, 0, B, 0.

  Value of *A* is adjusted right away towards 4/5.

  But then the value if *B* is decreased from 4/5 to something like 4/6.

- It would be nice to propagate this information to *A* as well!

Eligibility Traces: TD(λ)

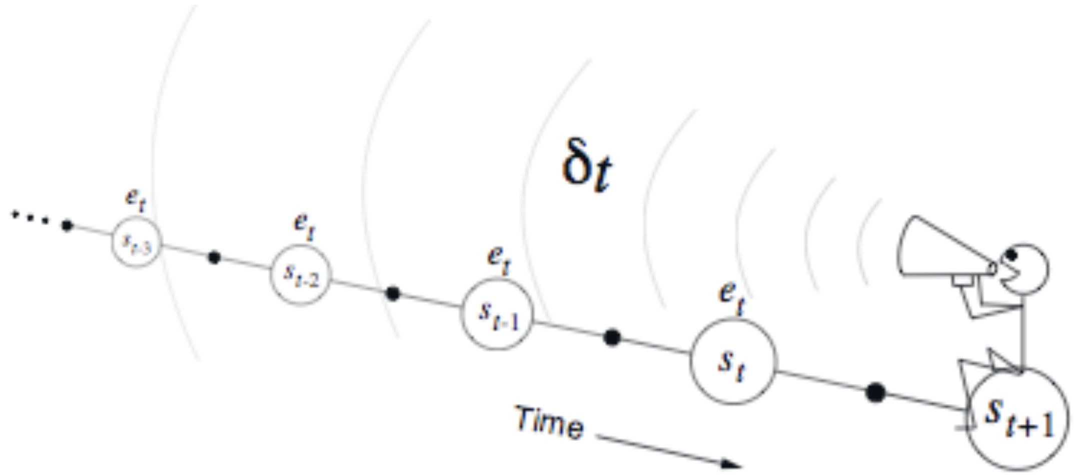


- On every time step *t*, we compute the TD error:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

- In addition to updating $V(s_t)$, shout δ_t backwards to past states.

Eligibility Traces: TD(λ)

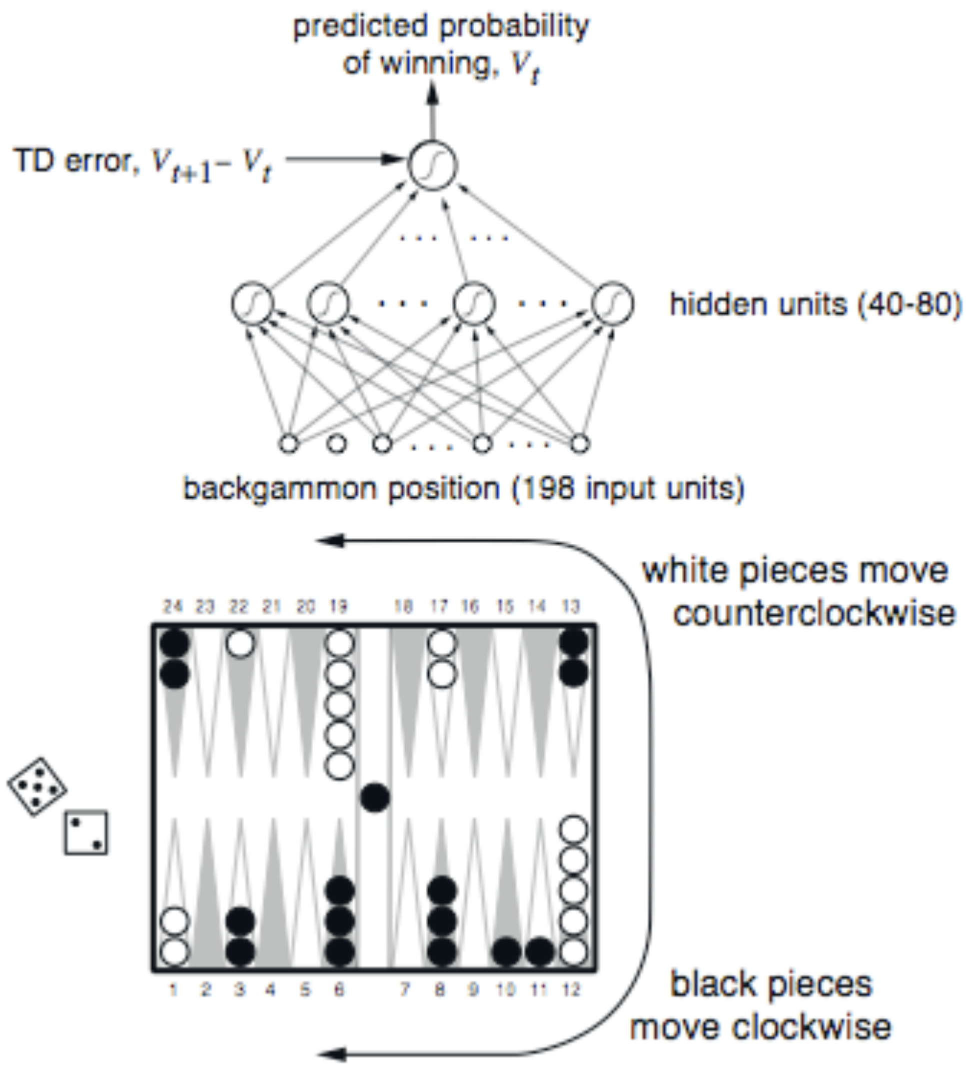


- On every time step t, we compute the TD error:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

- In addition to updating $V(s_t)$, shout δ_t backwards to past states.
- The strength of your voice decreases with temporal distance by $\gamma\lambda$, where $\lambda \in [0, 1]$ is a parameter.

TD-Gammon (Tesauro, 1992)



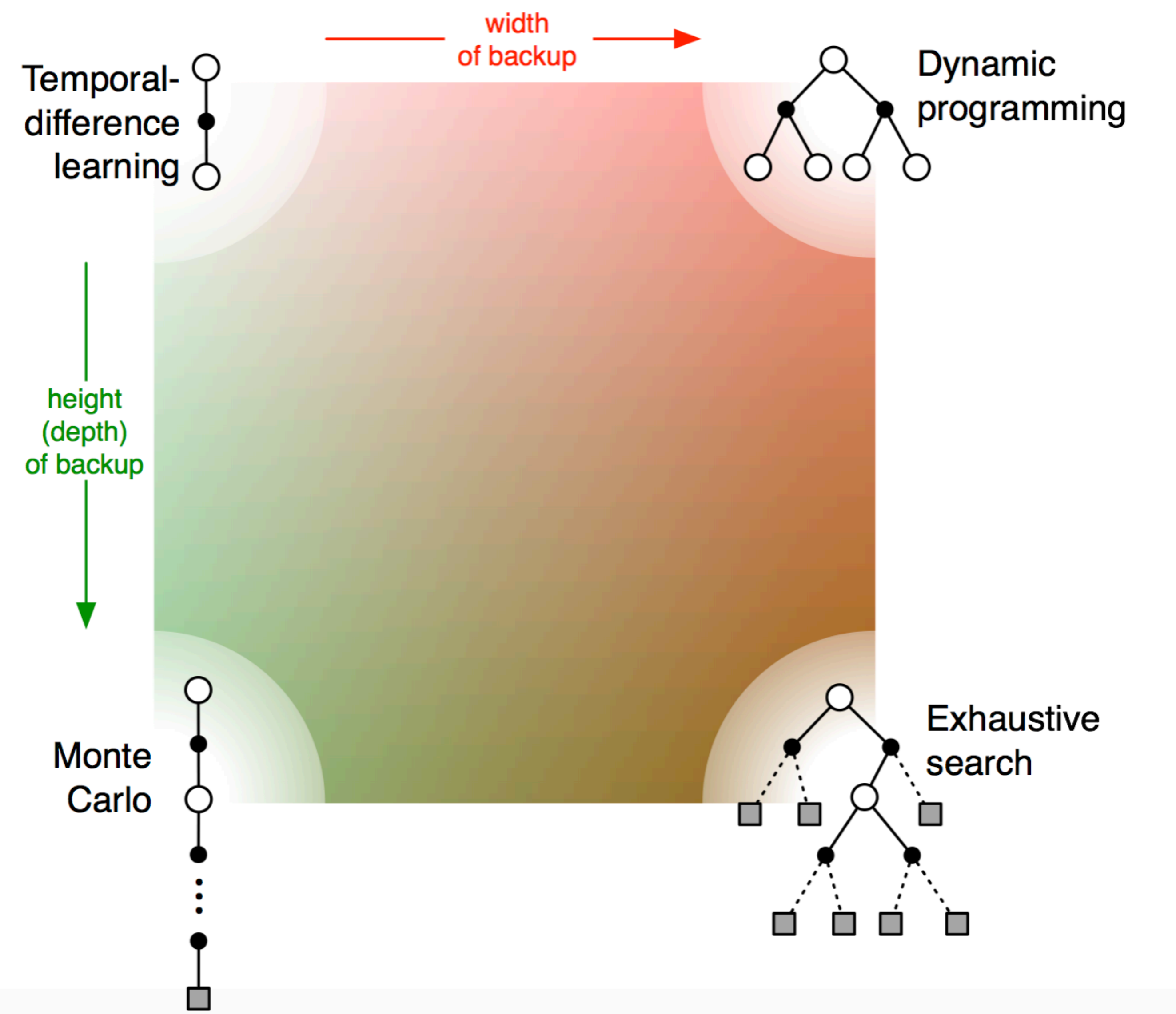
Reward function:

- +100 if win
- - 100 if lose
- 0 for all other states

Trained by playing 1.5×10^6 games against itself.

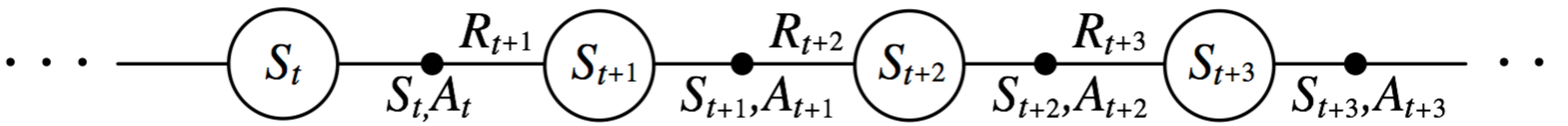
Enough to beat the best human player.

A unified view



Learning an action value function

Estimate q_π for the current policy π



After every transition from a nonterminal state, S_t, do this:

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

If S_{t+1} is terminal, then define $Q(S_{t+1}, A_{t+1}) = 0$

SARSA: On-policy TD control

Turn this into a control method by always updating the policy to be greedy with respect to the current estimate:

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, 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)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

On-policy vs off-policy learning

- Both MC and TD are <u>on-policy</u> algorithms.
- Policy induces a distribution over the states (data).
  - Data distribution **changes** every time you change the policy!
- Evaluating several policies with the same batch:
  - Need very big batch! Need policy to adequately cover all *(s,a)* pairs.
- Can use importance sampling to reweigh data samples to compute unbiased estimates of a new policy.
$$\rho_t = \frac{\pi(s_t, a_t)}{b(s_t, a_t)}$$
- Can we learn from data collected under a different policy?

  => Off-policy RL methods

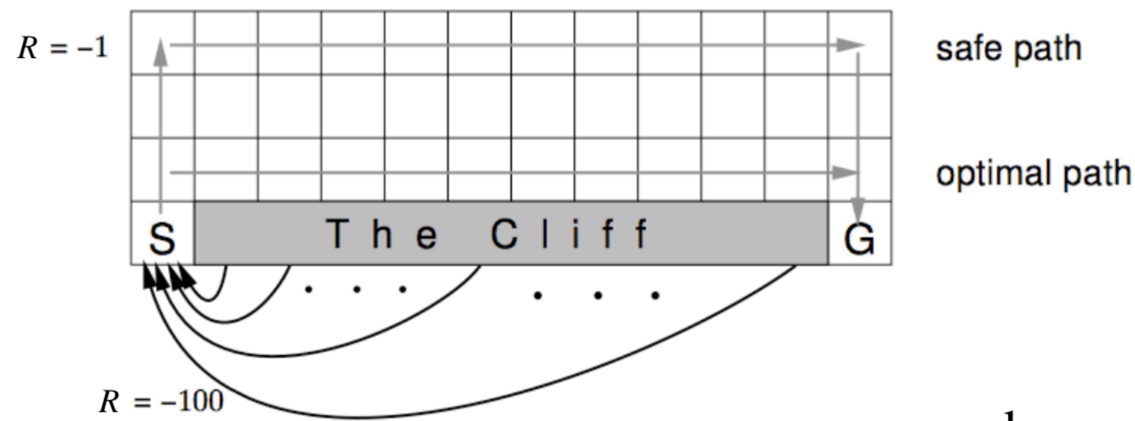
Q-learning: Off-policy TD control

- **Q-learning** (off-policy):

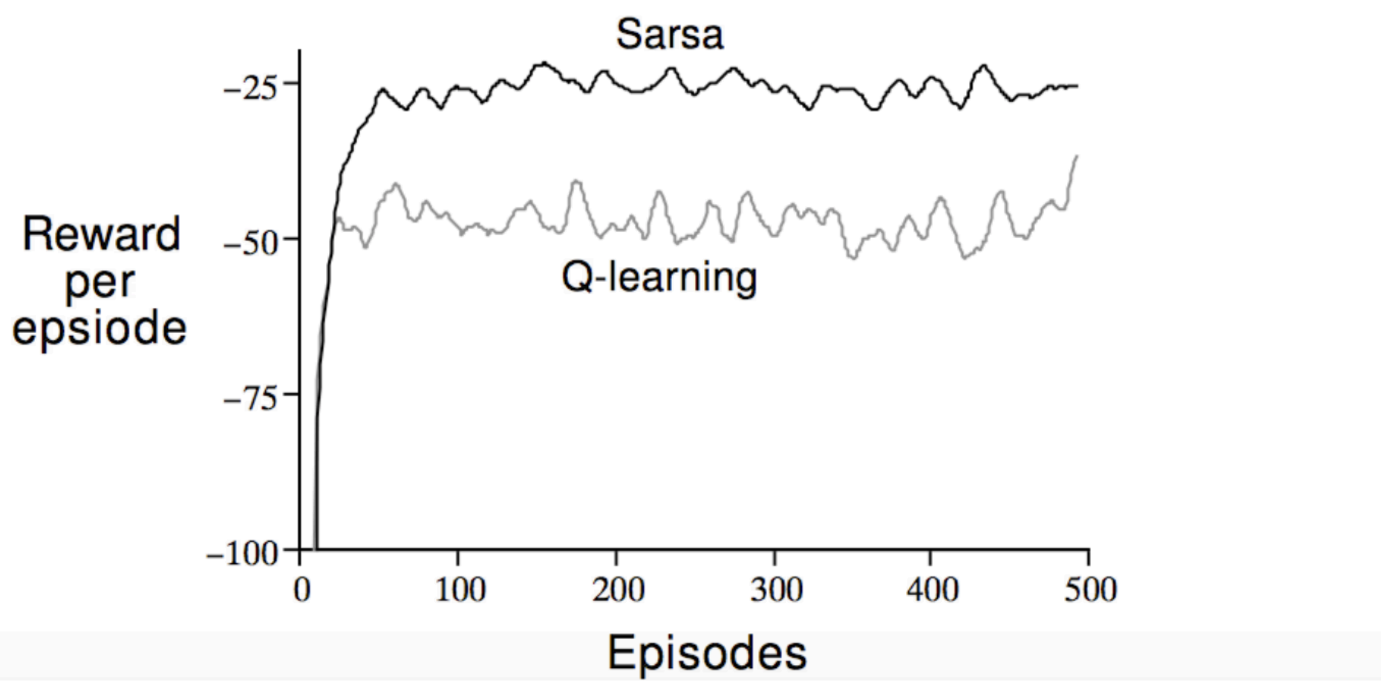
$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma max_a Q(s_{t+1}, a))$$

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, 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'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$;
  until S is terminal

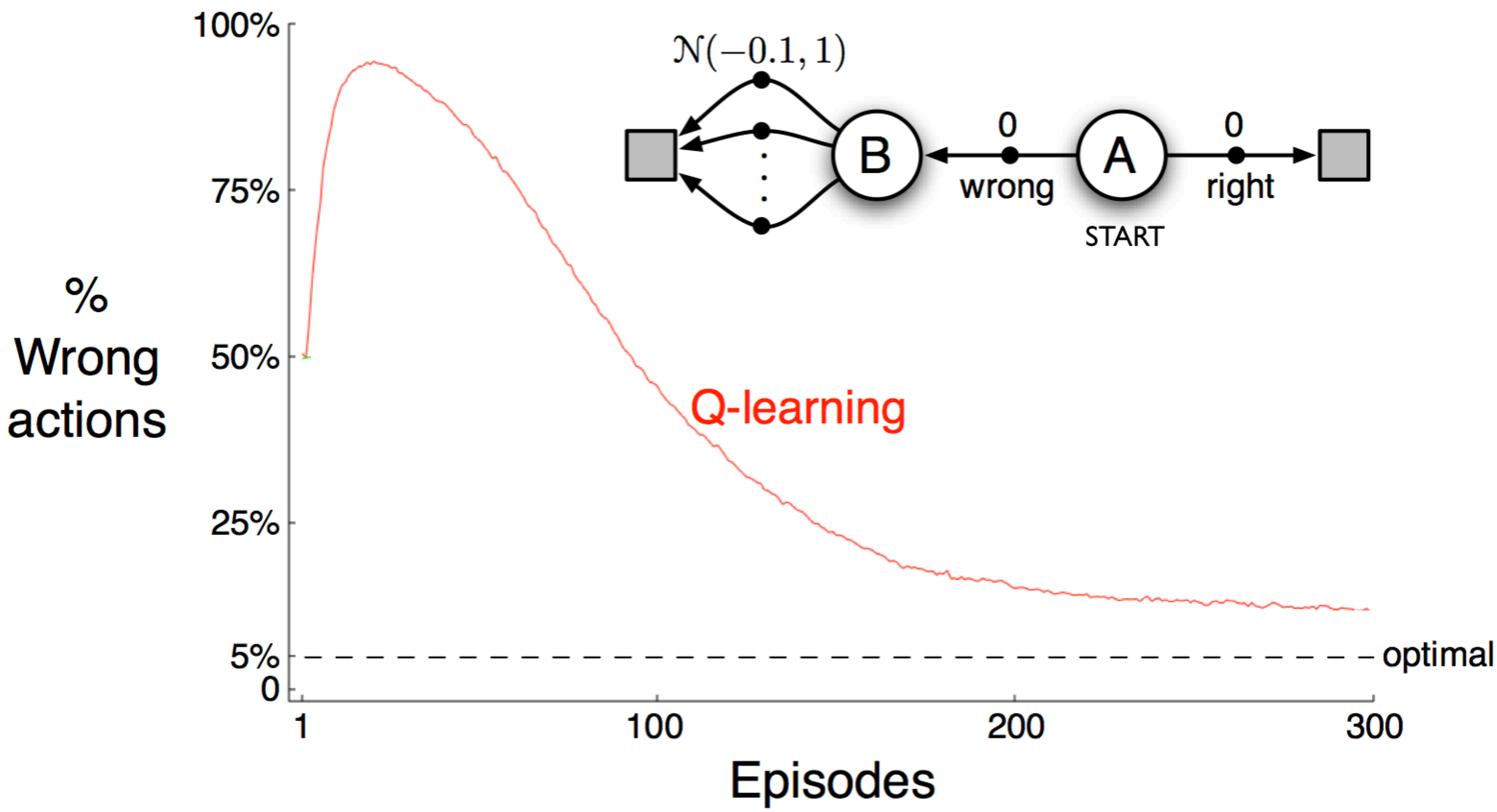
Example: Cliff walking



ε−greedy, $\varepsilon = 0.1$



Maximization bias example



Tabular Q-learning: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$

Double Q-learning

- Train 2 action-value functions, Q_1 and Q_2
- Do Q-learning on both, but
  - never on the same time steps (Q_1 and Q_2 are indep.)
  - pick Q_1 or Q_2 at random to be updated on each step
- If updating Q_1, use Q_2 for the value of the next state:

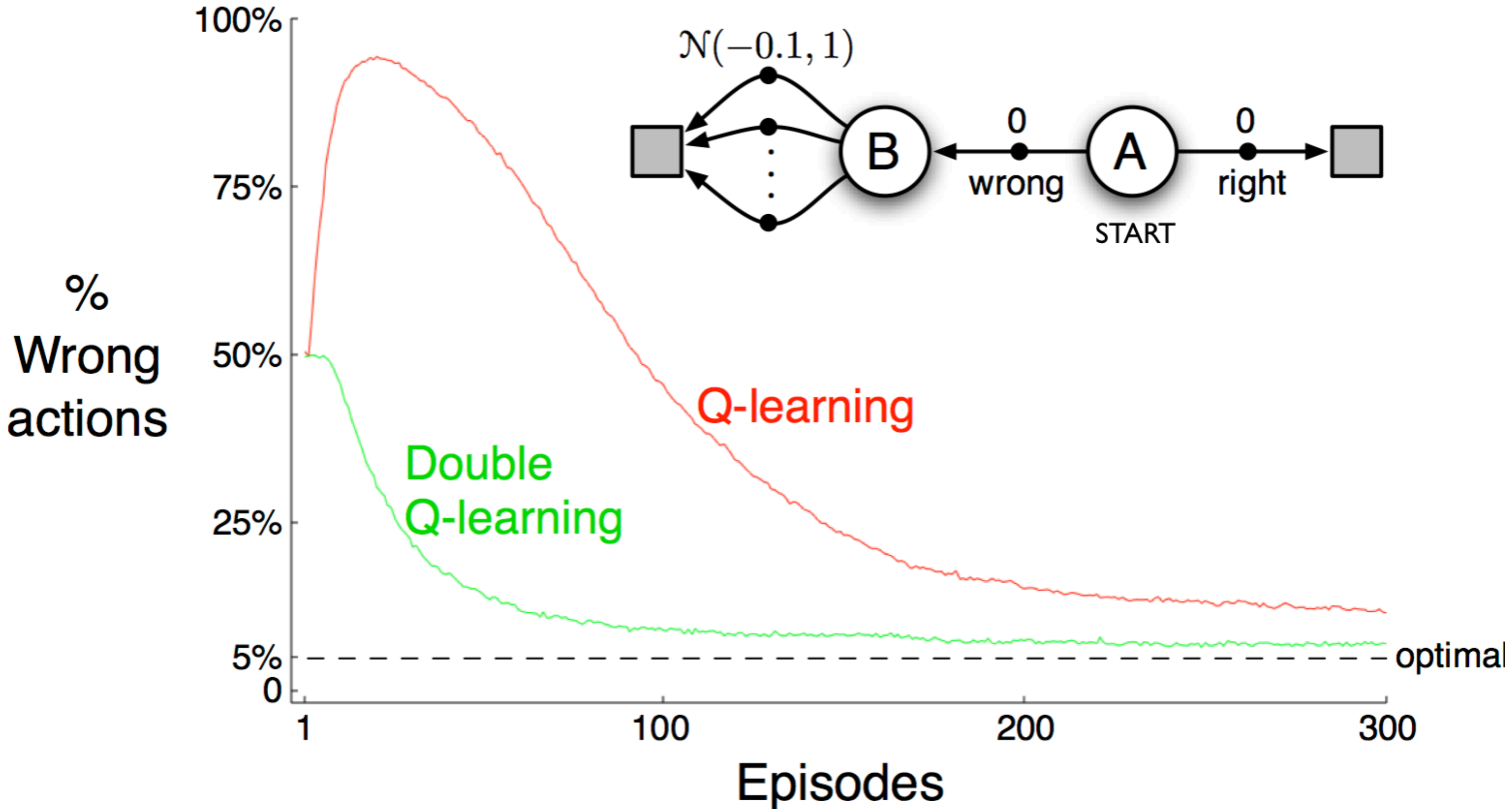
$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha\Big(R_{t+1} + Q_2\big(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t)\Big)$$

- Action selections are (say) ε-greedy with respect to the sum of Q_1 and Q_2

Double Q-learning

Initialize $Q_1(s,a)$ and $Q_2(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily
Initialize $Q_1(\textit{terminal-state}, \cdot) = Q_2(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q_1 and Q_2 (e.g., ε-greedy in $Q_1 + Q_2$)
    Take action A, observe R, S'
    With 0.5 probabilility:
      $$Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S',a)\big) - Q_1(S,A)\Big)$$
    else:
      $$Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S',a)\big) - Q_2(S,A)\Big)$$
    $S \leftarrow S'$;
  until S is terminal

Maximization bias example

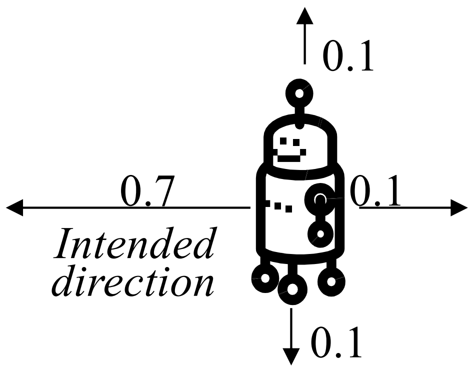


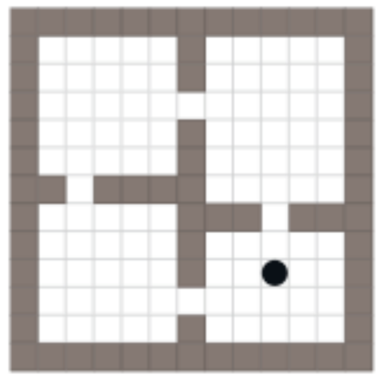
Key challenges in RL

- Designing the problem domain
  - State representation
  - Action choice
  - Cost/reward signal
- Acquiring data for training
  - Exploration / exploitation
  - High cost actions
  - Time-delayed cost/reward signal
- Function approximation
- Validation / confidence measures

Tabular vs Function approximation

- **Tabular**: Can store in memory a list of the states and their value.





** Can prove many more* ***theoretical properties*** *in this case, about convergence, sample complexity.*

- **Function approximation**: Too many states, continuous state spaces.

In large state spaces: Need approximation

Challenge: finding good features

$$\hat{Q}^\pi(s,a) = \sum_{i=1}^{d} \theta_i \underbrace{\phi_i(s,a)}_{\text{feature vector}}$$

Temporal-Difference with function approx.

- **Tabular TD(0)**:

$$V(s_t) \leftarrow V(s_t) + \alpha \left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right) \forall t = 0, 1, 2, \ldots$$

- **Gradient-descent TD(0)**:

$$\theta \leftarrow \theta + \alpha \left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right) \nabla_\theta V(s_t), \forall t = 0, 1, 2, \ldots$$

Use the **TD-error**, instead of the “supervised” error.

Fitted Q-iteration

- Use **supervised learning** to estimate the **Q-function** from a batch of training data.
  - Input: $x_i := \langle s_i, a_i \rangle, \;\; i=1..N$
  - Output: $y_i := r_i + \gamma \max_a Q_\theta(s_i', a)$
  - Loss: $\sum_i \| r_i + \gamma \max_a Q_\theta(s_i', a) - Q_\theta(s_i, a_i) \|^2$

- Regression with linear function, neural network, etc.
  (Can use other functions, e.g. random forests.)

Fitted Q-iteration

- Use **supervised learning** to estimate the **Q-function** from a batch of training data.
  - Input: $x_i := \langle s_i, a_i \rangle, \; i=1..N$
  - Output: $y_i := r_i + \gamma \max_a Q_\theta(s_i', a)$
  - Loss: $\sum_i \| r_i + \gamma \max_a Q_\theta(s_i', a) - Q_\theta(s_i, a_i) \|^2$

- Regression with linear function, neural network, etc.
  (Can use other functions, e.g. random forests.)

- **Important note**: Q_θ appears twice in the loss => Hard to learn!
  - And in addition, r can be very sparse.