Decision Making and Reinforcement Learning

Module 8: Reinforcement Learning - Generalization

Tony Dear, Ph.D.

Department of Computer Science

School of Engineering and Applied Sciences

Topics

- n-step temporal difference prediction
- n-step SARSA, on-policy and off-policy
- Model-based RL and Dyna-Q
- Function approximation

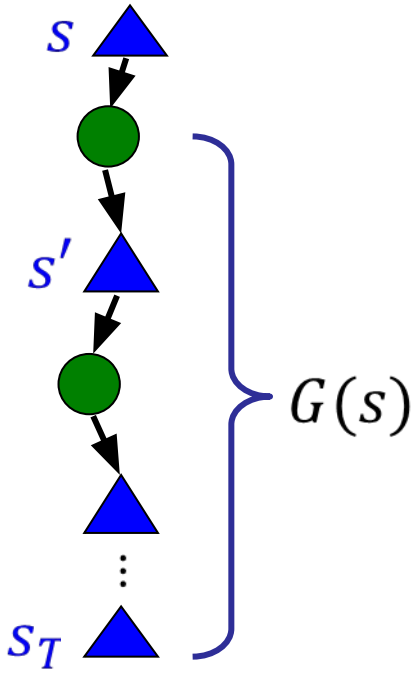
Learning Objectives

- **Generalize** Monte Carlo and TD(0) prediction methods to implement the n-step temporal difference method
- **Understand and implement** n-step SARSA for on-policy control and use importance sampling for off-policy control
- **Explain** the application of the Dyna-Q algorithm for model-based learning
- **Utilize** function approximation for reinforcement learning

Review: MC and TD

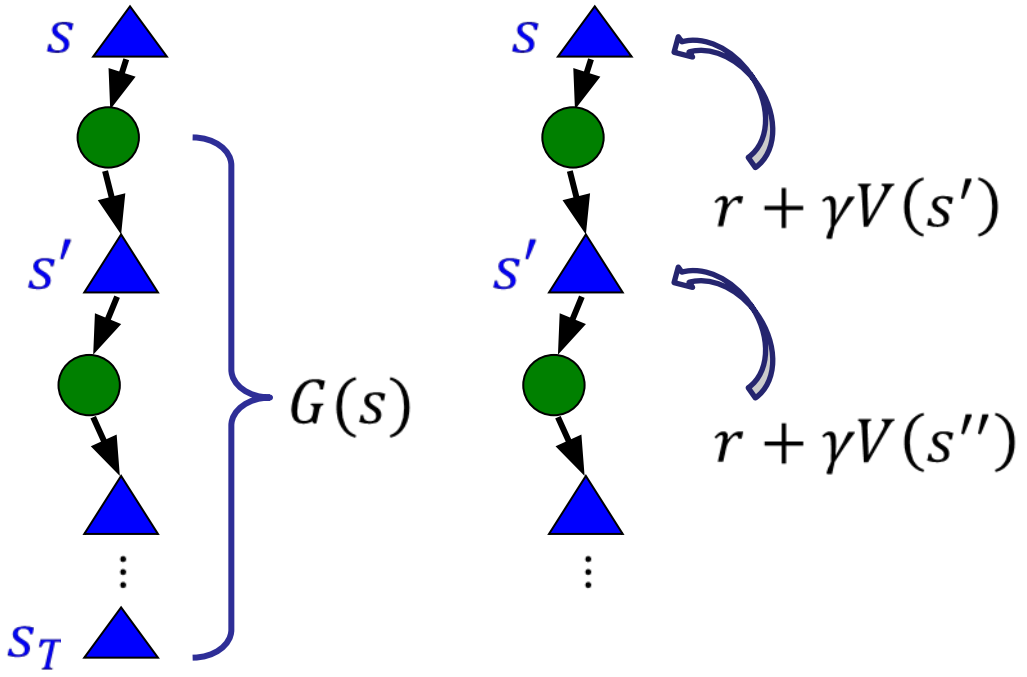
- We covered two different approaches for *prediction*—finding values of a policy

- **Monte Carlo** estimates a state value using the rewards of an entire *episode*



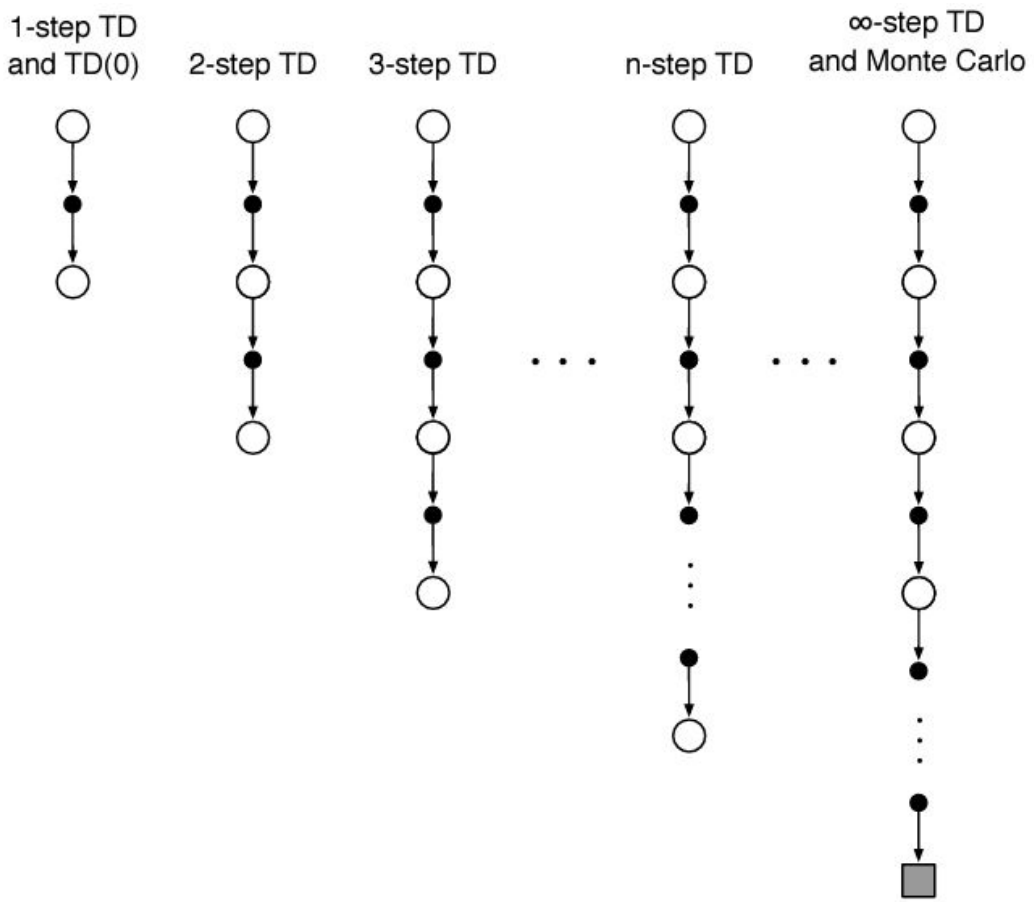
Review: MC and TD

- We covered two different approaches for *prediction*—finding values of a policy
- **Monte Carlo** estimates a state value using the rewards of an entire *episode*
- **Temporal difference** computes an *update* to a state value using the reward of a single *transition*



n-Step TD Prediction

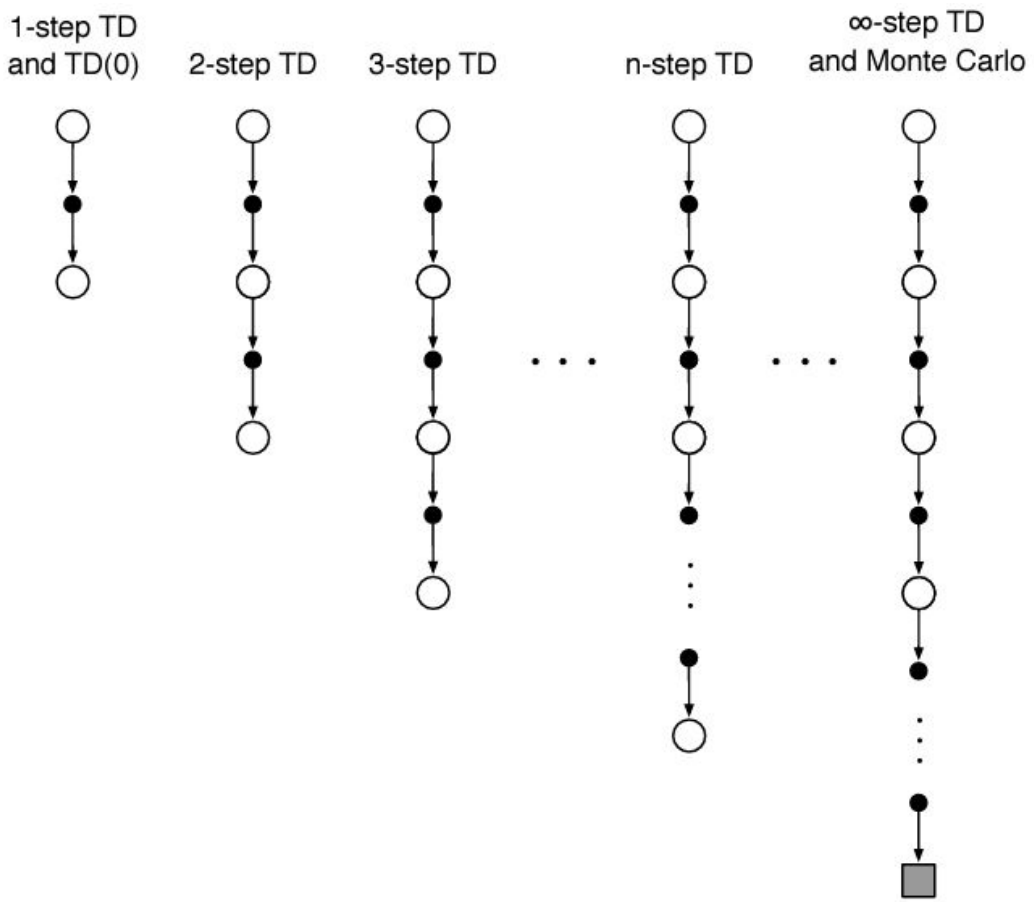
- $TD(0)$ and MC are just two extrema of a *family* of bootstrapping estimators



n-Step TD Prediction

- $TD(0)$ and MC are just two extrema of a *family* of bootstrapping estimators

- **n-step TD**: *Target value* consists of n steps of rewards followed by successor state value



n-Step TD Prediction

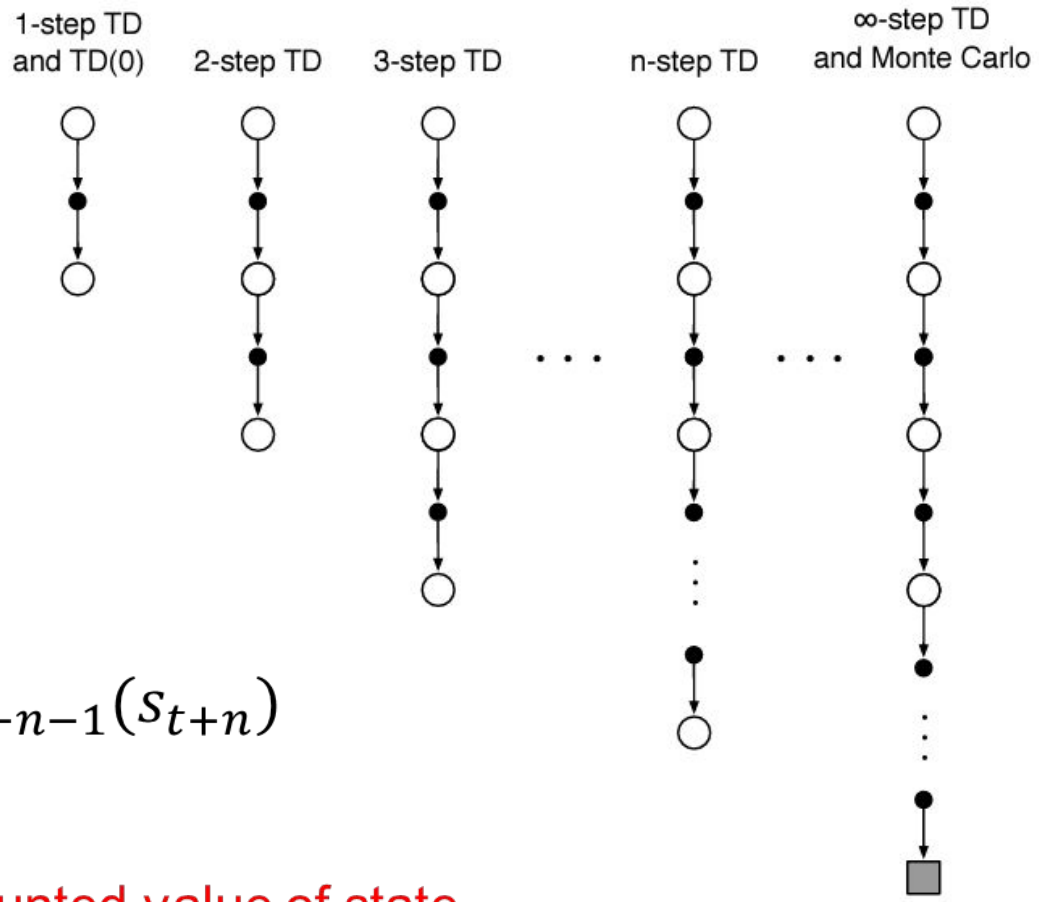
- $TD(0)$ and MC are just two extrema of a *family* of bootstrapping estimators

- **n-step TD**: *Target value* consists of n steps of rewards followed by successor state value

- Target value: $$G_{t:t+n} = \sum_{j=0}^{n-1} \gamma^j r_{j+t+1} + \gamma^n V_{t+n-1}(s_{t+n})$$

Discounted sum of rewards from r_{t+1} to r_{t+n}

Discounted value of state s_{t+n} at time $t+n-1$



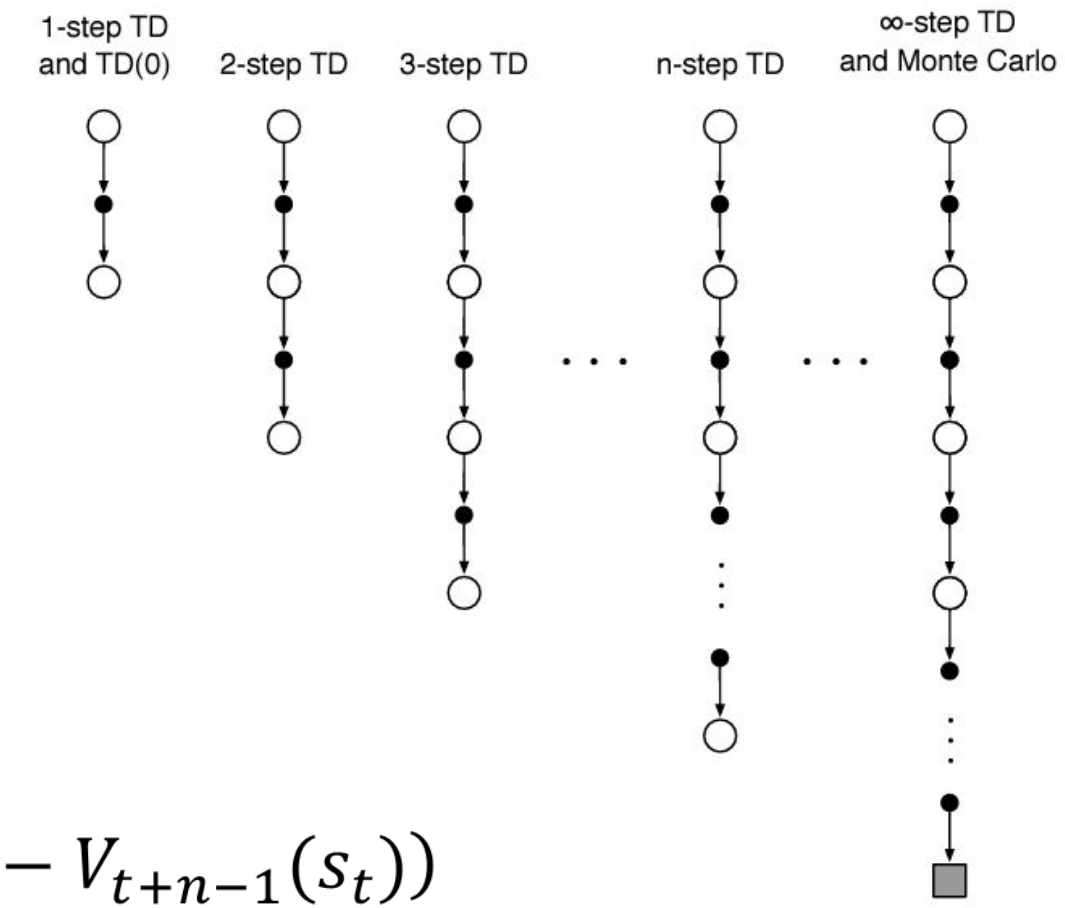
n-Step TD Prediction

- $TD(0)$ and MC are just two extrema of a *family* of bootstrapping estimators

- **n-step TD**: *Target value* consists of n steps of rewards followed by successor state value

- Temporal difference update:

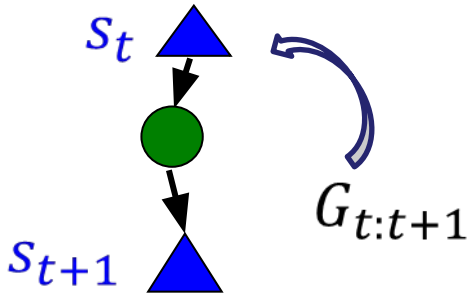
$$V_{t+n}(s_t) \leftarrow V_{t+n-1}(s_t) + \alpha(G_{t:t+n} - V_{t+n-1}(s_t))$$



Example: n-Step Updates

- $n = 1$ (TD(0)): $G_{t:t+1} = r_{t+1} + \gamma V_t(s_{t+1})$

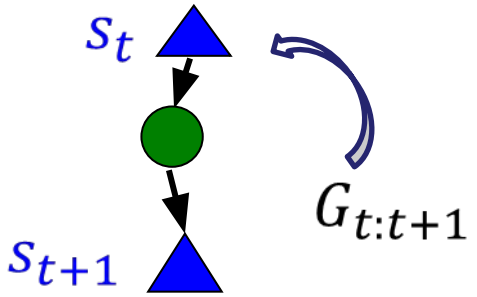
$$V_{t+1}(s_t) \leftarrow V_t(s_t) + \alpha(G_{t:t+1} - V_t(s_t))$$



Example: n-Step Updates

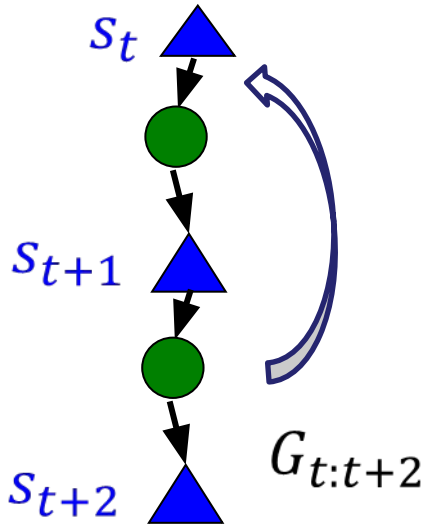
- $n = 1$ (TD(0)): $G_{t:t+1} = r_{t+1} + \gamma V_t(s_{t+1})$

$$V_{t+1}(s_t) \leftarrow V_t(s_t) + \alpha(G_{t:t+1} - V_t(s_t))$$



- $n = 2$: $G_{t:t+2} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_{t+1}(s_{t+2})$

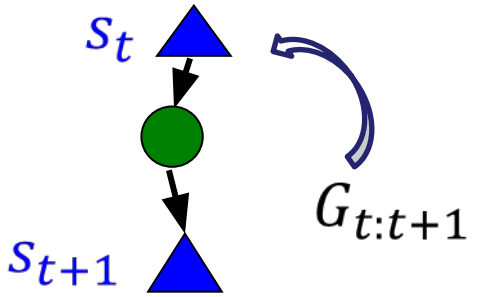
$$V_{t+2}(s_t) \leftarrow V_{t+1}(s_t) + \alpha(G_{t:t+2} - V_{t+1}(s_t))$$



Example: n-Step Updates

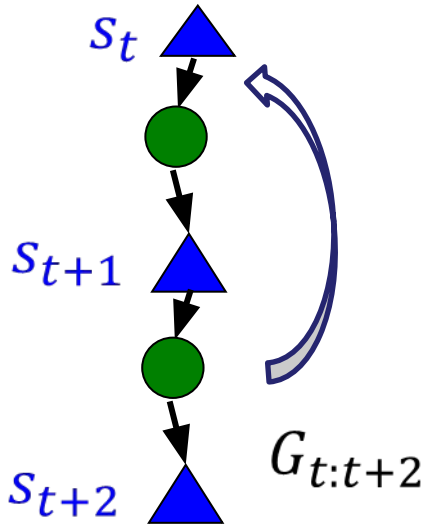
- $n = 1$ (TD(0)): $G_{t:t+1} = r_{t+1} + \gamma V_t(s_{t+1})$

$$V_{t+1}(s_t) \leftarrow V_t(s_t) + \alpha(G_{t:t+1} - V_t(s_t))$$



- $n = 2$: $G_{t:t+2} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_{t+1}(s_{t+2})$

$$V_{t+2}(s_t) \leftarrow V_{t+1}(s_t) + \alpha(G_{t:t+2} - V_{t+1}(s_t))$$



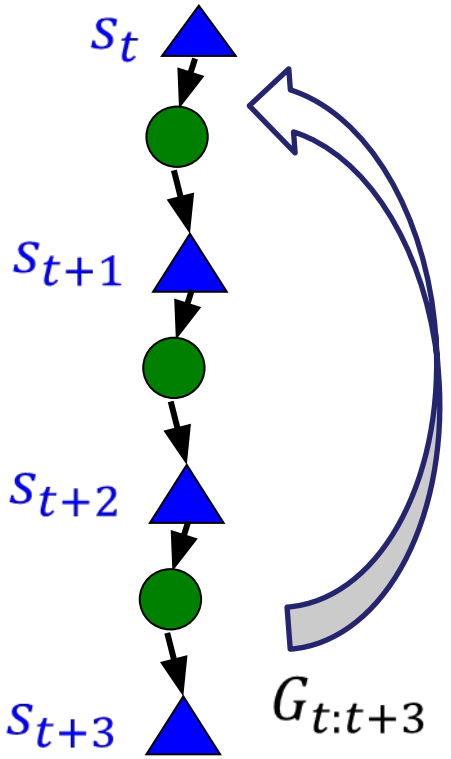
- $n = 3$?

Example: n-Step Updates

- $n = 3$:

$$G_{t:t+3} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V_{t+2}(s_{t+3})$$

$$V_{t+3}(s_t) \leftarrow V_{t+2}(s_t) + \alpha(G_{t:t+3} - V_{t+2}(s_t))$$



Example: Mini-Gridworld

- All values initialized to $V_0 = 0; \gamma = 0.8, \alpha = 0.5$
- Policy to evaluate: $\pi(s) = L$ for all states

- Observed state and reward sequence: $(A, +3, A, -2, B, +1, C, -2, B, +3, A)$

Example: Mini-Gridworld

- All values initialized to $V_0 = 0; \gamma = 0.8, \alpha = 0.5$
- Policy to evaluate: $\pi(s) = L$ for all states

- Observed state and reward sequence: $(A, +3, A, -2, B, +1, C, -2, B, +3, A)$
- 2-step TD update: $G_{t:t+2} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_{t+1}(s_{t+2})$

Time	
	0
	0
	0

Example: Mini-Gridworld

- All values initialized to $V_0 = 0; \gamma = 0.8, \alpha = 0.5$
- Policy to evaluate: $\pi(s) = L$ for all states

- Observed state and reward sequence: $(A, +3, A, -2, B, +1, C, -2, B, +3, A)$
- 2-step TD update:

$$G_{t:t+2} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_{t+1}(s_{t+2})$$

Time		
	0	
	0	0
	0	0

$$G_{0:2} = 3 + 0.8(-2) + 0.8^2 V_1(B) = 1.4$$

$$V_2(A) = V_1(A) + 0.5(G_{0:2} - V_1(A)) = 0.7$$

Example: Mini-Gridworld

- All values initialized to $V_0 = 0; \gamma = 0.8, \alpha = 0.5$
- Policy to evaluate: $\pi(s) = L$ for all states

- Observed state and reward sequence: $(A, +3, A, -2, B, +1, C, -2, B, +3, A)$
- 2-step TD update:

$$G_{t:t+2} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_{t+1}(s_{t+2})$$

Time			
	0		
	0	0	0
	0	0	0

$$G_{1:3} = -2 + 0.8(1) + 0.8^2 V_2(C) = -1.2$$

$$V_3(A) = V_2(A) + 0.5(G_{1:3} - V_2(A)) = -0.25$$

Example: Mini-Gridworld

- All values initialized to $V_0 = 0; \gamma = 0.8, \alpha = 0.5$
- Policy to evaluate: $\pi(s) = L$ for all states

- Observed state and reward sequence: $(A, +3, A, -2, B, +1, C, -2, B, +3, A)$
- 2-step TD update:

$$G_{t:t+2} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_{t+1}(s_{t+2})$$

Time				
	0			
	0	0	0	
	0	0	0	0

$$G_{2:4} = 1 + 0.8(-2) + 0.8^2 V_3(B) = -0.6$$

$$V_4(B) = V_3(B) + 0.5\big(G_{2:4} - V_3(B)\big) = -0.3$$

Example: Mini-Gridworld

- All values initialized to $V_0 = 0; \gamma = 0.8, \alpha = 0.5$
- Policy to evaluate: $\pi(s) = L$ for all states

- Observed state and reward sequence: $(A, +3, A, -2, B, +1, C, -2, B, +3, A)$
- 2-step TD update:

$$G_{t:t+2} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_{t+1}(s_{t+2})$$

Time					
	0				
	0	0	0		
	0	0	0	0	

$$G_{3:5} = -2 + 0.8(3) + 0.8^2 V_4(A) = 0.24$$

$$V_5(C) = V_4(C) + 0.5\big(G_{3:5} - V_4(C)\big) = 0.12$$

Error Reduction

- n-step TD still bootstraps but uses n "true" rewards, while using a state value estimate $V(s_{t+n})$ to *approximate* the rest

Error Reduction

- n-step TD still bootstraps but uses n "true" rewards, while using a state value estimate $V(s_{t+n})$ to *approximate* the rest

- Advantage: *Expectation* of using true rewards is guaranteed to be better for learning V^{π} than the corresponding *value estimate* (Watkins, 1989)

Error Reduction

- n-step TD still bootstraps but uses n "true" rewards, while using a state value estimate $V(s_{t+n})$ to *approximate* the rest

- Advantage: *Expectation* of using true rewards is guaranteed to be better for learning V^{π} than the corresponding *value estimate* (Watkins, 1989)

$$\max_{s_t} |E(G_{t:t+n}) - V^{\pi}(s_t)| \leq \gamma^n \max_{s_t} |V_{t+n-1}(s_t) - V^{\pi}(s_t)|$$

- Max error of expected n-step return is upper bounded by discounted max error of estimate V_{t+n-1}

n-Step SARSA

- n-step TD can be used for on-policy control as ***n*-step SARSA**
- We can follow an ε-greedy behavior policy

- As with prediction, combine next n steps of reward seen with the current Q value estimate for returns beyond n steps

n-Step SARSA

- n-step TD can be used for on-policy control as ***n*-step SARSA**
- We can follow an ε-greedy behavior policy

- As with prediction, combine next n steps of reward seen with the current Q value estimate for returns beyond n steps

$$G_{t:t+n} = \sum_{j=0}^{n-1} \gamma^j r_{j+t+1} + \gamma^n Q_{t+n-1}(s_{t+n}, a_{t+n})$$

$$Q_{t+n}(s_t, a_t) \leftarrow Q_{t+n-1}(s_t, a_t) + \alpha(G_{t:t+n} - Q_{t+n-1}(s_t, a_t))$$

Off-Policy Learning

- On-policy learning is suitable if the *target policy* π (what we wish to learn about) is the same as the *behavior policy* b (what the agent is following)

- Off-policy learning is suitable if they are different, e.g., π is pure greedy and b is ε-greedy with exploration

Off-Policy Learning

- On-policy learning is suitable if the *target policy* π (what we wish to learn about) is the same as the *behavior policy* b (what the agent is following)

- Off-policy learning is suitable if they are different, e.g., π is pure greedy and b is ε-greedy with exploration

- Recall that we can compute an **importance sampling ratio** of relative state-action likelihoods:

$$\rho_{t:h} = \prod_{k=t}^{h} \frac{\pi(a_k|s_k)}{b(a_k|s_k)}$$

Off-Policy Learning

- We can apply the IS ratio to state value updates for off-policy prediction:

$$V_{t+n}(s_t) \leftarrow V_{t+n-1}(s_t) + \alpha\rho_{t:t+n-1}(G_{t:t+n} - V_{t+n-1}(s_t))$$

- $\rho_{t:t+n-1}$ is the likelihood that subsequent n actions took place after s_t

Off-Policy Learning

- We can apply the IS ratio to state value updates for off-policy prediction:

$$V_{t+n}(s_t) \leftarrow V_{t+n-1}(s_t) + \alpha\rho_{t:t+n-1}(G_{t:t+n} - V_{t+n-1}(s_t))$$

- $\rho_{t:t+n-1}$ is the likelihood that subsequent n actions took place after s_t

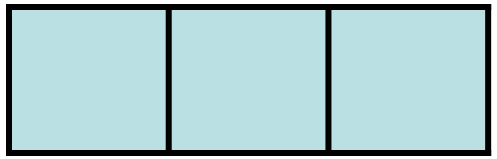
- Similar update for Q-values for off-policy control:

$$Q_{t+n}(s_t, a_t) \leftarrow Q_{t+n-1}(s_t, a_t) + \alpha\rho_{t+1:t+n}(G_{t:t+n} - Q_{t+n-1}(s_t, a_t))$$

- For Q-values, $\rho_{t+1:t+n}$ is computed starting with a_{t+1} up to a_{t+n}

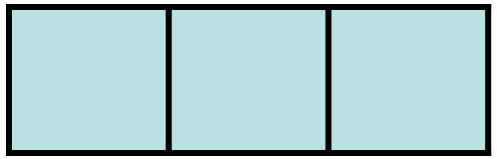
Example: Mini-Gridworld

- Suppose we have $Q_0(A, L) = 1.5, Q_0(A, R) = 1$
- Behavior policy is ε-greedy; $\alpha = 0.5, \gamma = 0.8$



Example: Mini-Gridworld

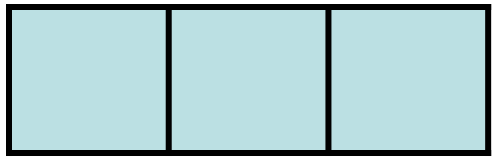
- Suppose we have $Q_0(A, L) = 1.5$, $Q_0(A, R) = 1$
- Behavior policy is ε-greedy; $\alpha = 0.5, \gamma = 0.8$



- Observed s-a-r sequence: $A, L, +3, A, R, +3, A, R$
- We will perform 2-step SARSA, so $Q_1 = Q_0$

Example: Mini-Gridworld

- Suppose we have $Q_0(A, L) = 1.5$, $Q_0(A, R) = 1$
- Behavior policy is ε-greedy; $\alpha = 0.5, \gamma = 0.8$

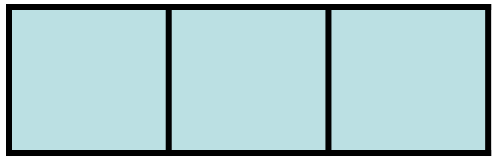


- Observed s-a-r sequence: $A, L, +3, A, R, +3, A, R$
- We will perform 2-step SARSA, so $Q_1 = Q_0$

- On-policy SARSA target: $G_{0:2} = 3 + 0.8(3) + 0.8^2 Q_1(A, R) = 6.4$
- Updated Q-value: $Q_2(A, L) = Q_1(A, L) + 0.5\big(G_{0:2} - Q_1(A, L)\big) = 3.95$

Example: Mini-Gridworld

- Suppose we have $Q_0(A, L) = 1.5$, $Q_0(A, R) = 1$
- Behavior policy is ε-greedy; $\alpha = 0.5, \gamma = 0.8$

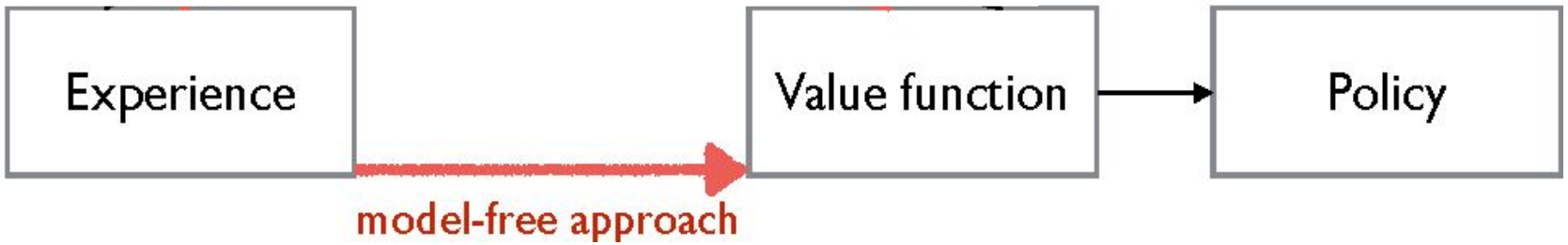


- Observed s-a-r sequence: $A, L, +3, A, R, +3, A, R$
- We will perform 2-step SARSA, so $Q_1 = Q_0$

- But if we were doing off-policy learning, we found that the IS ratio $\rho = 0$ since $(s, a) = (A, R)$ would *never* occur under a greedy target policy
- There is no Q-value update for off-policy learning!

Model-Based Methods

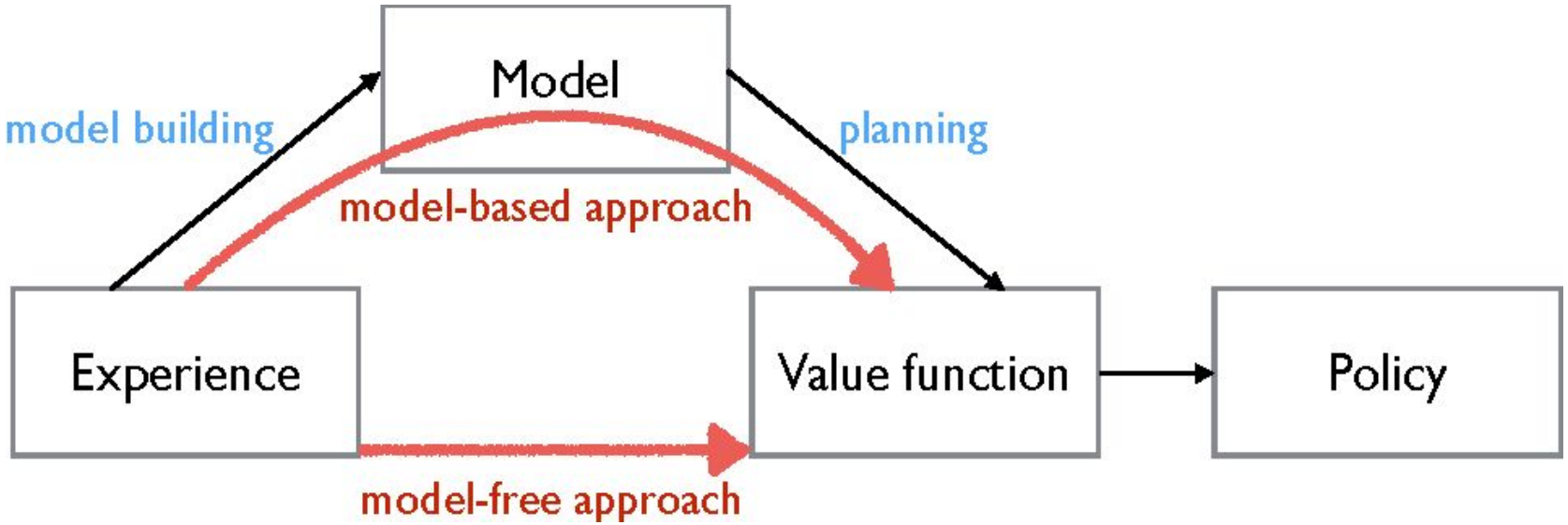
- With *model-free* methods, experience and samples are used in update and backup operations to *learn* value functions and policies



Model-Based Methods

- With *model-free* methods, experience and samples are used in update and backup operations to *learn* value functions and policies

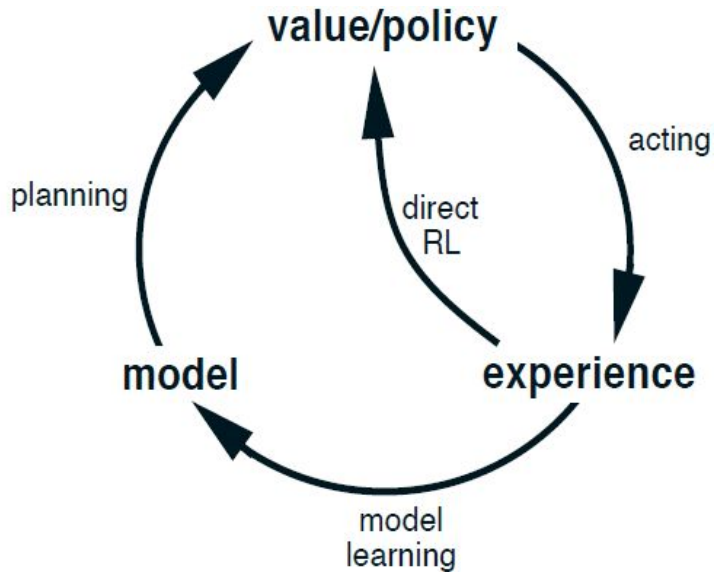
- With *model-based* methods, experience can be used to learn and update a model, and both can then be used to compute value functions and policies



Model-Based Methods

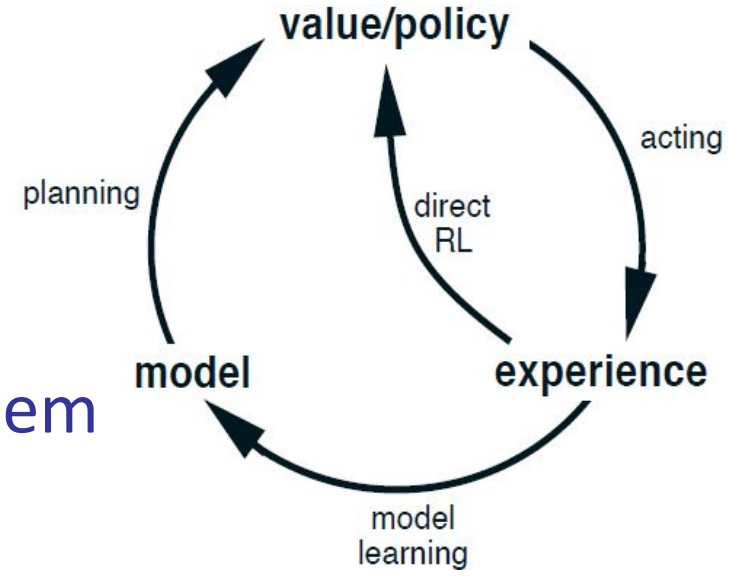
- While a model is typically just a fixed set of parameters used for planning, it can be also be viewed as a source of offline, simulated experience

- "Real" experience can be used to both compute value functions and policies as well as learn an underlying model



Model-Based Methods

- While a model is typically just a fixed set of parameters used for planning, it can be also be viewed as a source of offline, simulated experience
- "Real" experience can be used to both compute value functions and policies as well as learn an underlying model
- The model can then be applied to learning as well
- We solve both a *direct RL* and a *model learning* problem



Dyna-Q

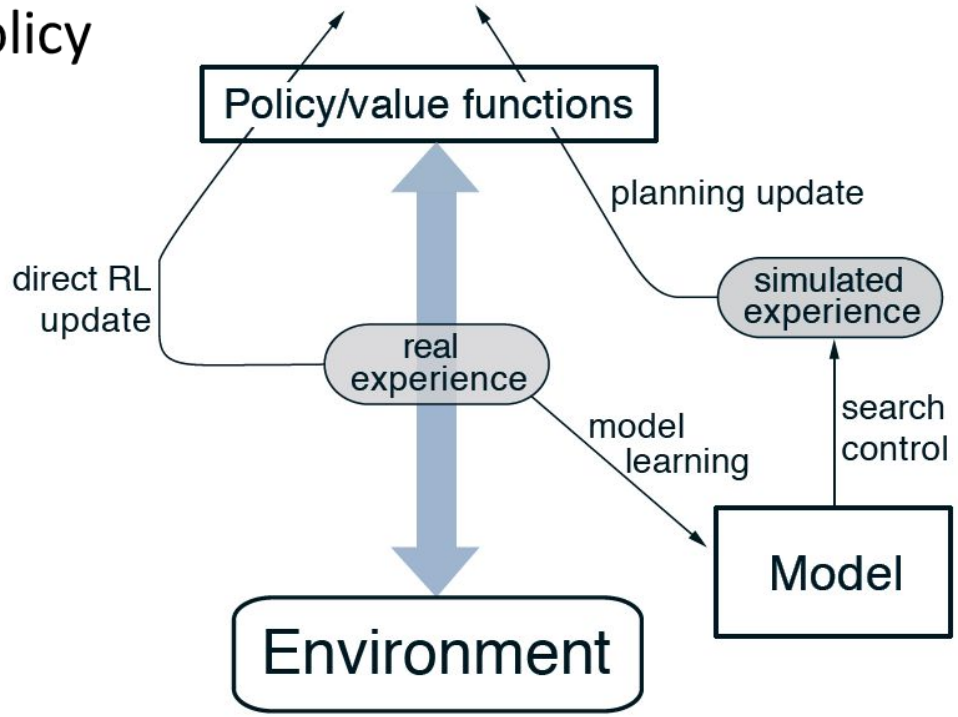
- The “Dyna-Q” algorithm integrates both real and simulated experience to plan:
- **Initialize** $Q(s,a)$ and $Model(s,a)$ $\forall s,a$

Dyna-Q

- The "Dyna-Q" algorithm integrates both real and simulated experience to plan:
- **Initialize** $Q(s,a)$ and $Model(s,a)$ $\forall s,a$
- **Loop**:
  - **Initialize** s if needed, action a using behavior policy
  - **Generate** sequence (s,a,r,s')
  - **Perform** Q-learning update on $Q(s,a)$

Dyna-Q

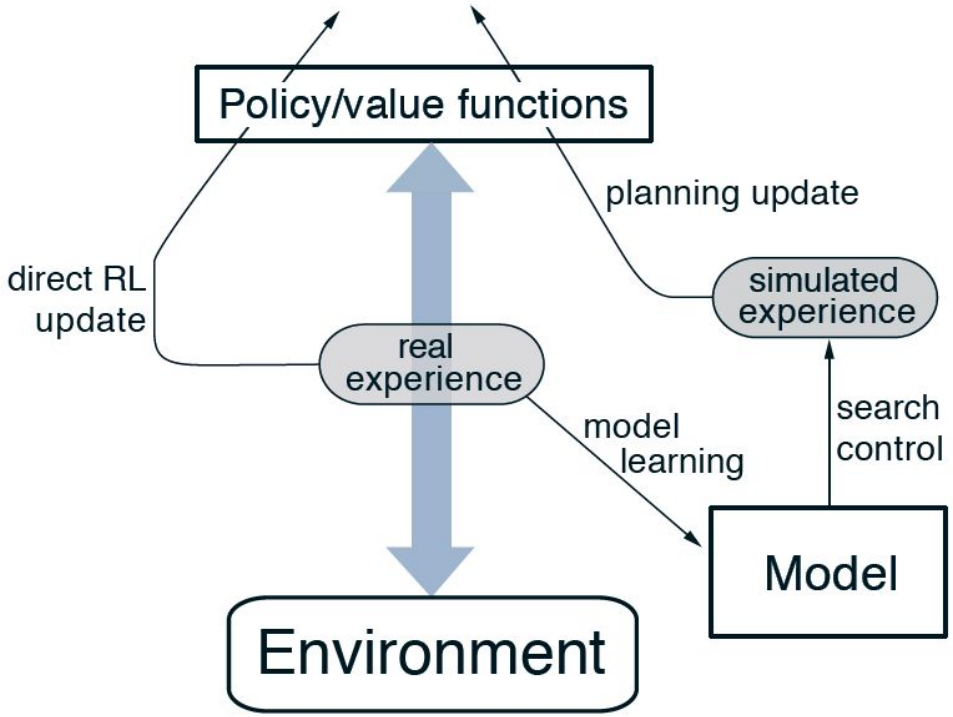
- The "Dyna-Q" algorithm integrates both real and simulated experience to plan:
- **Initialize** $Q(s,a)$ and $Model(s,a)$ $\forall s,a$
- **Loop**:
  - **Initialize** s if needed, action a using behavior policy
  - **Generate** sequence (s,a,r,s')
  - **Perform** Q-learning update on $Q(s,a)$
  - **Perform** model update: $Model(s,a) \leftarrow r,s'$
  - **Loop** as desired:
    - $\sigma,\alpha \leftarrow$ previously seen state-action pair
    - **Generate** $(\sigma,\alpha,\rho,\sigma')$ using $Model(\sigma,\alpha)$
    - **Perform** Q-learning update on $Q(\sigma,\alpha)$



Dyna-Q

- Model update estimates the transition and reward functions
- The planning part may be run as much as computational resources allow per iteration

- **Perform** model update: $Model(s, a) \leftarrow r, s'$
- **Loop** as desired:
  - $\sigma, \alpha \leftarrow$ previously seen state-action pair
  - **Generate** $(\sigma, \alpha, \rho, \sigma')$ using $Model(\sigma, \alpha)$
  - **Perform** Q-learning update on $O(\sigma, \alpha)$

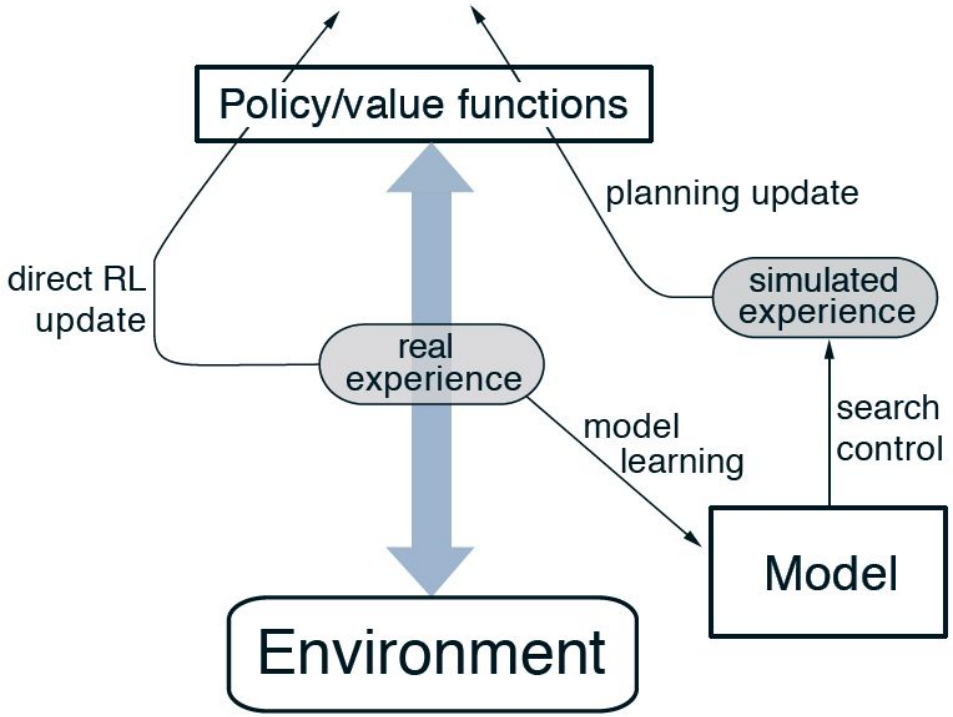


Dyna-Q

- Model update estimates the transition and reward functions
- The planning part may be run as much as computational resources allow per iteration

- The model is queried with previously seen inputs
- We can now learn with less real experience!

  - **Perform** model update: $Model(s, a) \leftarrow r, s'$
  - **Loop** as desired:
    - $\sigma, \alpha \leftarrow$ previously seen state-action pair
    - **Generate** $(\sigma, \alpha, \rho, \sigma')$ using $Model(\sigma, \alpha)$
    - **Perform** Q-learning update on $O(\sigma, \alpha)$



Model Sensitivity

- While a model can be a good substitute for limited experience, it can also produce suboptimality if it is incorrect

- Sources of model incorrectness: Insufficient samples used to learn it, incorrect initialization, nonstationary problems

Model Sensitivity

- While a model can be a good substitute for limited experience, it can also produce suboptimality if it is incorrect
- Sources of model incorrectness: Insufficient samples used to learn it, incorrect initialization, nonstationary problems
- Increased exploration can help correct model inaccuracies
- One idea: “Boost” a reward by an amount proportional to amount of time since the transition was last seen

Prioritized Sweeping

- Benefit of simulated experience is greatest when it leads to updates that were otherwise not done using real experience

- Instead of simulating randomly chosen state-actions, we can focus on those that have a high likelihood of producing value updates

Prioritized Sweeping

- Benefit of simulated experience is greatest when it leads to updates that were otherwise not done using real experience

- Instead of simulating randomly chosen state-actions, we can focus on those that have a high likelihood of producing value updates

- **Prioritized sweeping**: Use a queue to *prioritize* state-actions whose neighbors saw large updates from direct RL
- Can repeatedly simulate and update those state-actions until convergence

Function Approximation

- We have considered learning the *complete* value function for a given MDP or RL problem, assuming discrete state and action spaces

- V and Q can be represented in a *tabular* format using arrays or lists

Function Approximation

- We have considered learning the *complete* value function for a given MDP or RL problem, assuming discrete state and action spaces
- V and Q can be represented in a *tabular* format using arrays or lists
- This is not possible in problems with too many states or continuous states!
- We may need to do *function approximation*—estimate a complete value function using a fixed parameterization

Function Approximation

- Instead of directly learning $V^{\pi}(s)$, we learn $\hat{V}(s, \mathbf{w})$ by learning a set of weights $\mathbf{w}$

- $\mathbf{w}$ may be weights in some combination of features, neural network, decision tree, etc.

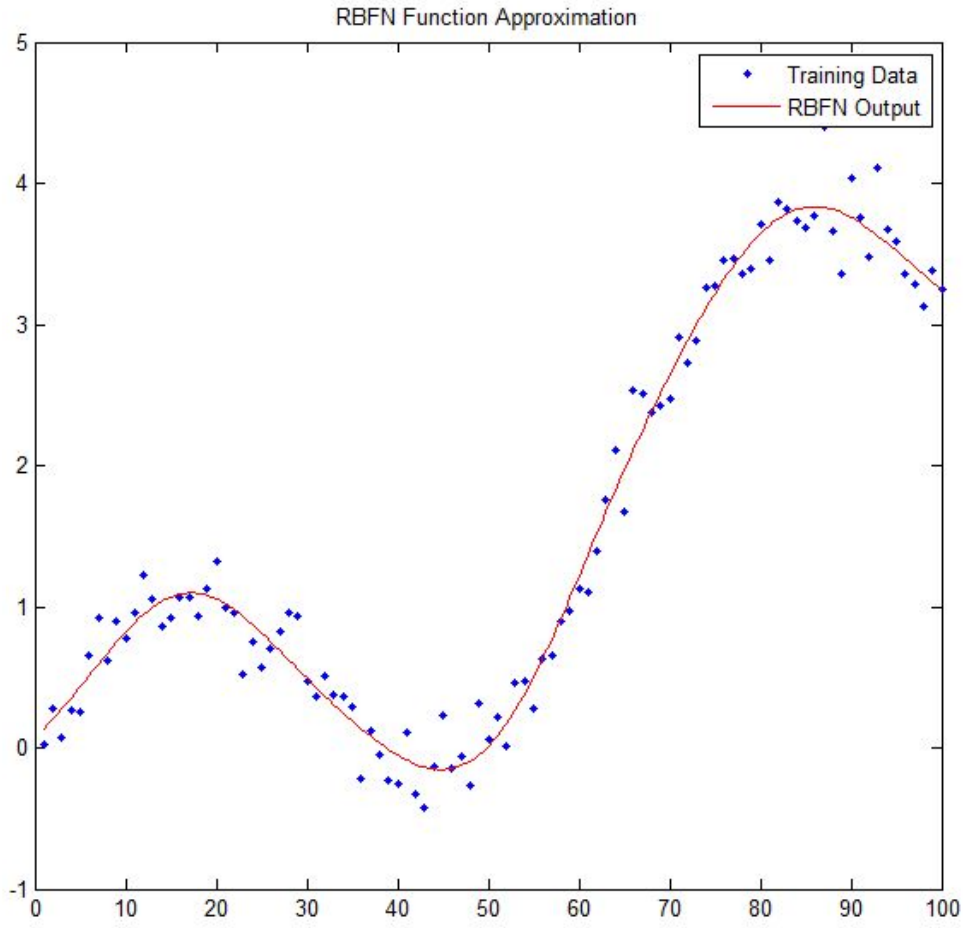
Function Approximation

- Instead of directly learning $V^{\pi}(s)$, we learn $\hat{V}(s, \mathbf{w})$ by learning a set of weights $\mathbf{w}$

- $\mathbf{w}$ may be weights in some combination of features, neural network, decision tree, etc.

- Our *objective* is to minimize an error function, e.g., $MSE(\mathbf{w}) = \frac{1}{n}\sum_s \left(V^{\pi}(s) - \hat{V}(s, \mathbf{w})\right)^2$

- $\hat{V}(s, \mathbf{w})$ is an *approximation* for $V^{\pi}(s)$



Stochastic Gradient Descent

- Suppose we have weights $\mathbf{w}_t$ at time t and we have the target U_t for state s_t
- We want to *shift* the weights so as to decrease and eventually minimize the error

Stochastic Gradient Descent

- Suppose we have weights $\mathbf{w}_t$ at time t and we have the target U_t for state s_t
- We want to *shift* the weights so as to decrease and eventually minimize the error

- **Stochastic gradient descent** adjusts the weights in the direction that reduces the error by the greatest amount with step size $0 < \alpha < 1$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left(U_t - \hat{V}(s_t, \mathbf{w}_t) \right) \nabla \hat{V}(s_t, \mathbf{w}_t)$$

Stochastic Gradient Descent

- Suppose we have weights $\mathbf{w}_t$ at time t and we have the target U_t for state s_t
- We want to *shift* the weights so as to decrease and eventually minimize the error

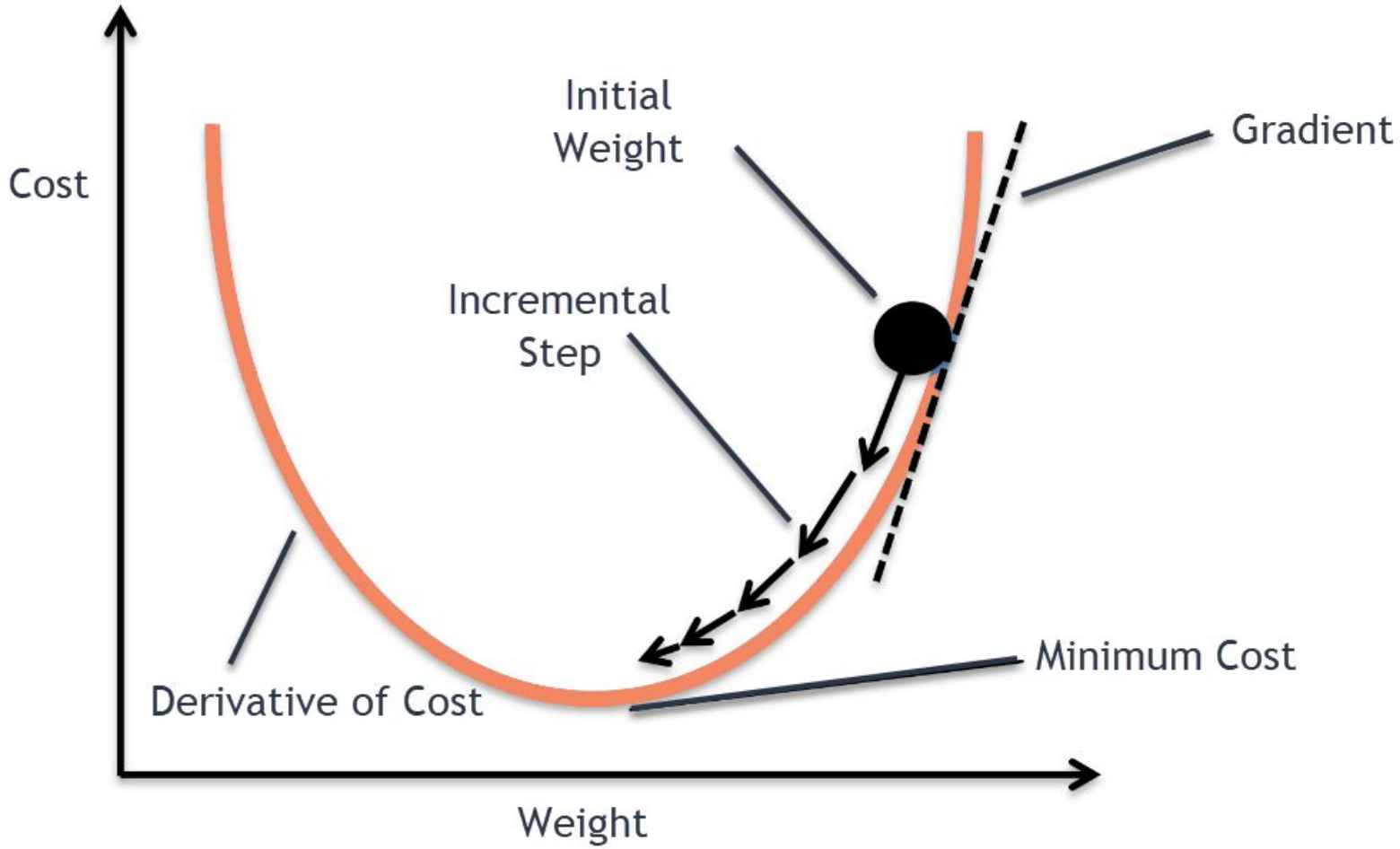
- **Stochastic gradient descent** adjusts the weights in the direction that reduces the error by the greatest amount with step size $0 < \alpha < 1$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left(U_t - \hat{V}(s_t, \mathbf{w}_t) \right) \nabla \hat{V}(s_t, \mathbf{w}_t)$$

- $\nabla \hat{V}$ is the *gradient* wrt $\mathbf{w}$, or the vector of partial derivatives of $\hat{V}$ wrt to each individual weight component:

$$\nabla \hat{V}(s_t, \mathbf{w}) = \left(\frac{\partial \hat{V}}{\partial w_1}, \frac{\partial \hat{V}}{\partial w_2}, \ldots, \frac{\partial \hat{V}}{\partial w_n} \right)$$

Visualizing Gradient Descent



Gradient Monte Carlo

- The episode return G_t is an *unbiased* estimate of $V^{\pi}(s_t)$
- The expectation of episode return values is equal to $V^{\pi}(s_t)$

Gradient Monte Carlo

- The episode return G_t is an *unbiased* estimate of $V^{\pi}(s_t)$
- The expectation of episode return values is equal to $V^{\pi}(s_t)$

- The SGD update rule can be directly used for MC prediction using G_t as the target with guarantee of convergence to local optimum:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left(G_t - \hat{V}(s_t, \mathbf{w}_t) \right) \nabla \hat{V}(s_t, \mathbf{w}_t)$$

- As with regular MC, episodic problem structure is required

Semi-Gradient Temporal Difference

- The zero bias property is not true for TD bootstrapping methods!
- Target values depend on each current instance of $\mathbf{w}_t$

Semi-Gradient Temporal Difference

- The zero bias property is not true for TD bootstrapping methods!
- Target values depend on each current instance of $\mathbf{w}_t$

- We can still get efficient and reliable results using a *semi-gradient* update

- Semi-gradient TD(0): $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left(r + \gamma \hat{V}(s_{t+1}, \mathbf{w}_t) - \hat{V}(s_t, \mathbf{w}_t) \right) \nabla \hat{V}(s_t, \mathbf{w}_t)$

Semi-Gradient Temporal Difference

- The zero bias property is not true for TD bootstrapping methods!
- Target values depend on each current instance of $\mathbf{w}_t$

- We can still get efficient and reliable results using a *semi-gradient* update

- Semi-gradient TD(0): $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left(r + \gamma \hat{V}(s_{t+1}, \mathbf{w}_t) - \hat{V}(s_t, \mathbf{w}_t) \right) \nabla \hat{V}(s_t, \mathbf{w}_t)$

- Semi-gradient SARSA:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left(r + \gamma \hat{Q}(s_{t+1}, a_{t+1}, \mathbf{w}_t) - \hat{Q}(s_t, a_t, \mathbf{w}_t) \right) \nabla \hat{Q}(s_t, a_t, \mathbf{w}_t)$$

Linear Feature Combinations

- Common function approximation: $\hat{V}$ is a weighted linear combination of *features* $\mathbf{x}$

$$\hat{V}(s, \mathbf{w}) = w_1 x_1(s) + w_2 x_2(s) + \cdots = \sum_i w_i x_i(s) = \mathbf{w}^\top \mathbf{x}(s)$$

Linear Feature Combinations

- Common function approximation: $\hat{V}$ is a weighted linear combination of *features* $\mathbf{x}$

$$\hat{V}(s, \mathbf{w}) = w_1 x_1(s) + w_2 x_2(s) + \cdots = \sum_i w_i x_i(s) = \mathbf{w}^\top \mathbf{x}(s)$$

- The gradient of $\hat{V}$ is simply $\nabla \hat{V}(s, \mathbf{w}) = \mathbf{x}(s)$

- The SGD update is then just $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left(U_t - \hat{V}(s_t, \mathbf{w}) \right) \mathbf{x}(s_t)$

Linear Feature Combinations

- Common function approximation: $\hat{V}$ is a weighted linear combination of *features* $\mathbf{x}$

$$\hat{V}(s, \mathbf{w}) = w_1 x_1(s) + w_2 x_2(s) + \cdots = \sum_i w_i x_i(s) = \mathbf{w}^\top \mathbf{x}(s)$$

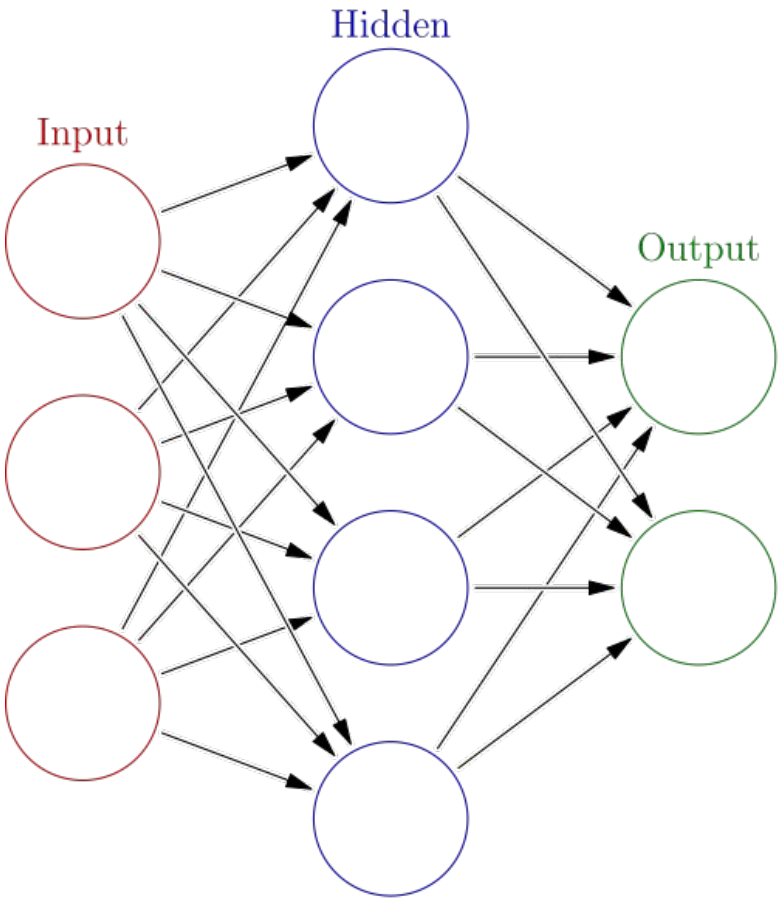
- The gradient of $\hat{V}$ is simply $\nabla \hat{V}(s, \mathbf{w}) = \mathbf{x}(s)$

- The SGD update is then just $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left(U_t - \hat{V}(s_t, \mathbf{w}) \right) \mathbf{x}(s_t)$

- Linear approximations do not have local optima—MC converges to the global optimum, while TD will converge to a *fixed point* near the global optimum

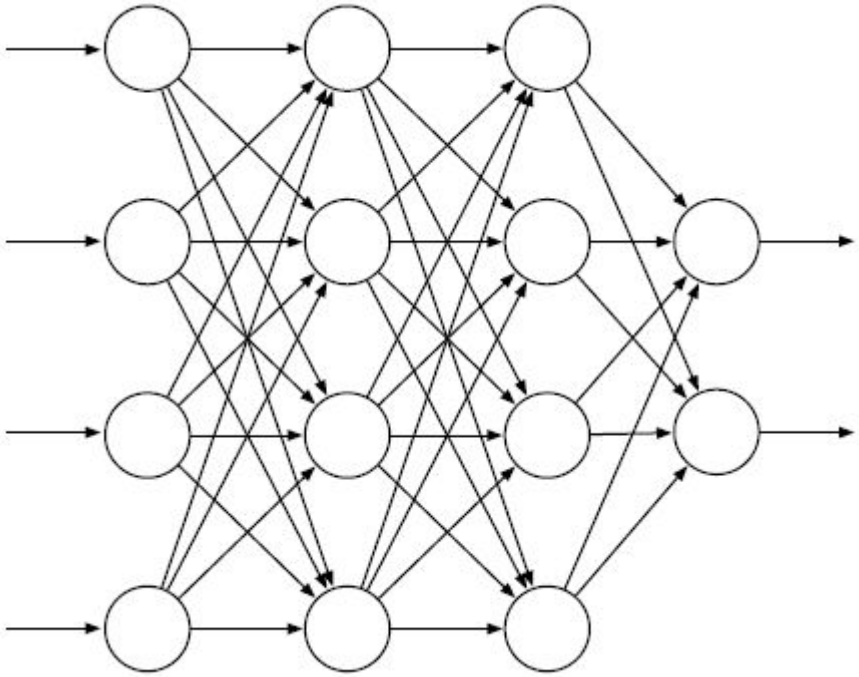
Neural Networks

- **Artificial neural networks** are a widely used tool for nonlinear function approximation
- Composed of several *layers* of neurons, including input, output, and *hidden layers*
- Each neuron outputs a nonlinear *activation function* of a weighted sum of inputs



Neural Networks

- ANNs typically have one or more hidden layers to create abstract representations of inputs
- More neurons can represent more information, but at the expense of training efficiency
- General goal is to learn the weights of the neuron connections
- Most methods like *backpropagation* run some kind of stochastic gradient descent



Summary

- Monte Carlo and temporal difference prediction and control methods lie on a continuum of n-step TD methods

- Model-based RL methods use raw data to learn a model
- Models can be used to generate simulated data and supplement real data

- Function approximations are important for generalization when tabular representations are not feasible
- Learning can be done using different forms of gradient descent