Lecture: Machine Learning for Data Science

Winter semester 2021/22

Lecture 19: Reinforcement Learning (Approximate learning)

Prof. Dr. Eirini Ntoutsi

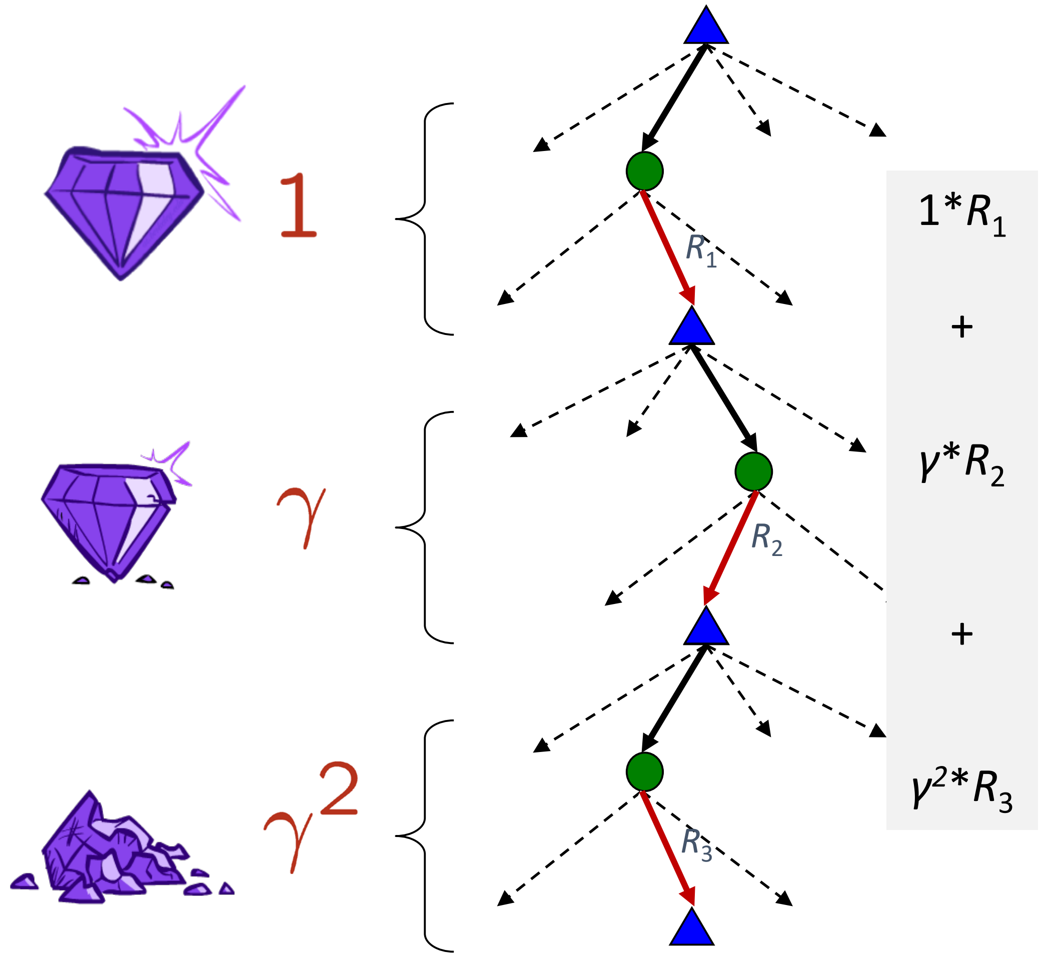
Outline

- Recap (with some additions)
- Approximate learning
- Things you should know from this lecture & reading material

Solving MDPs

See lecture 16

- Input: the MDP formulation
  - Set of states *S*
  - Start state S_0
  - Set of actions *A*
  - Transitions *P(s'|s,a)* (or *T(s,a,s')*)
  - Rewards *R(s,a,s')* (and discount γ)
- Output
  - An optimal policy: $\pi^*: S \rightarrow A$ that maps each state to an action, $\pi^*(s)$
  - If followed by the agent π^* will yield the maximum expected total reward or utility
- Utility = sum of (discounted rewards)



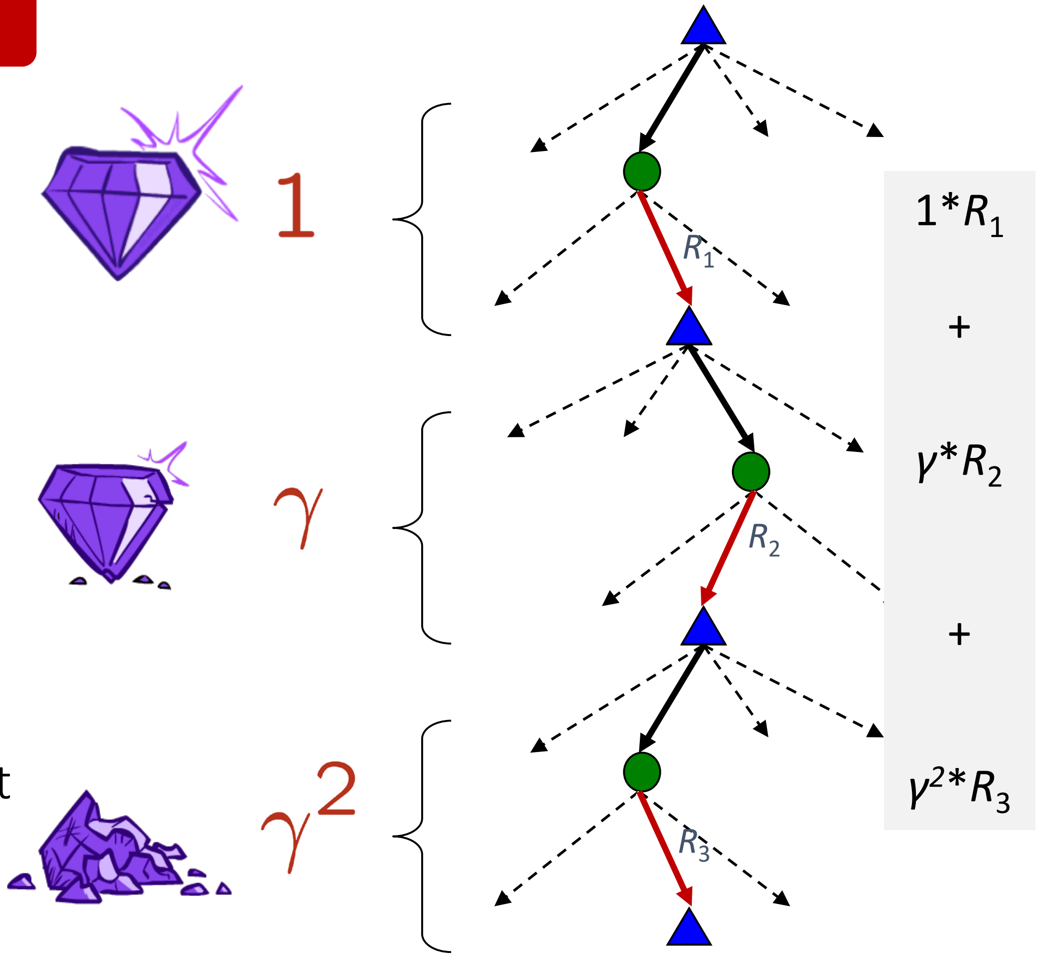
Value function (A major component of an RL agent)

See lecture 16

- How we decide among possible actions/states?
- The value(utility) of a state $s \rightarrow V(s)$ (called V-value)
  - It is a prediction of future reward
  - Used to evaluate the goodness/badness of states
- It is the expected value of the state

$$v_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s \right]$$

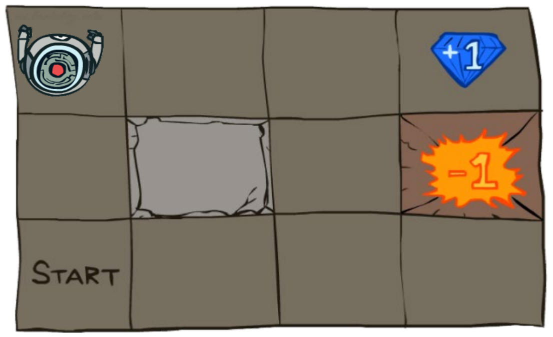
- But due to stochastic nature of the environment, a different environment history might be generated started from s_0
  - For a sequence $s_0, s_1, \ldots s_n$
    - $U[s_o, s_1, \ldots] = R(s_o) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots + \gamma^n R(s_n)$

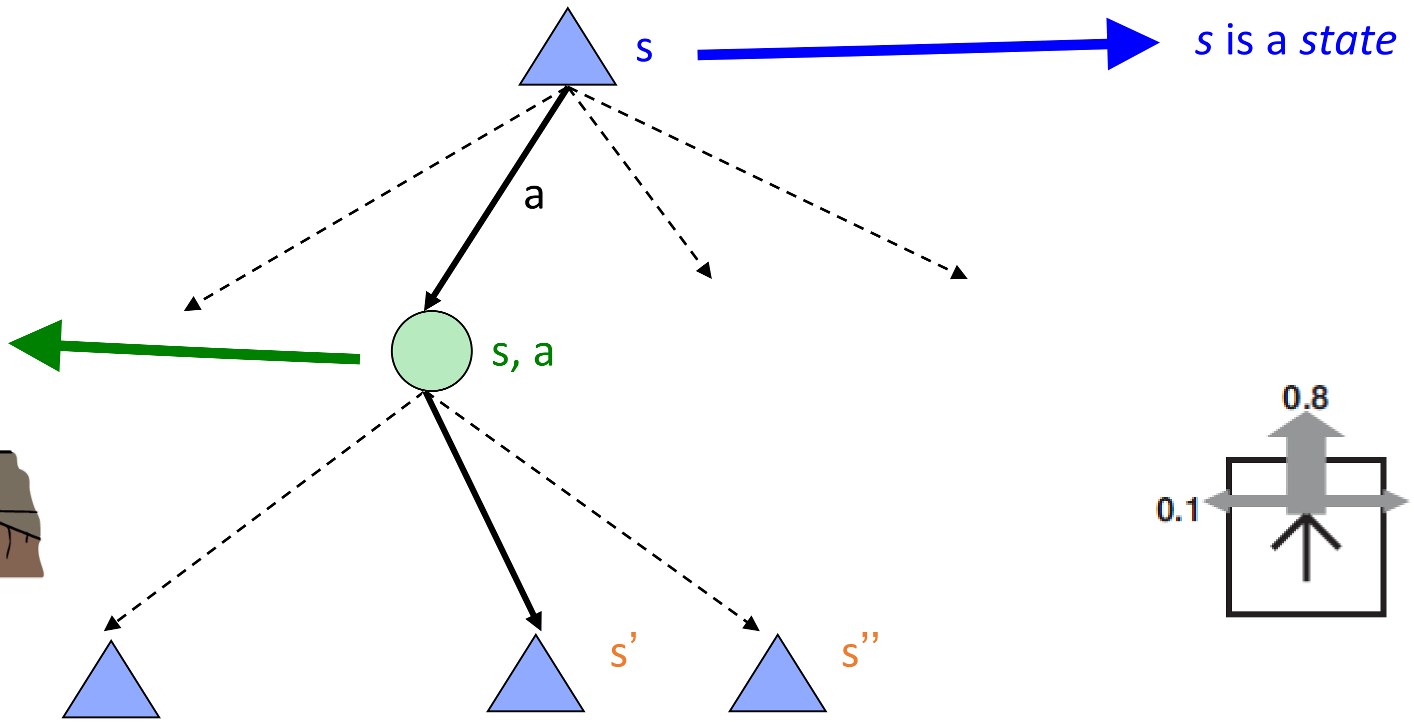


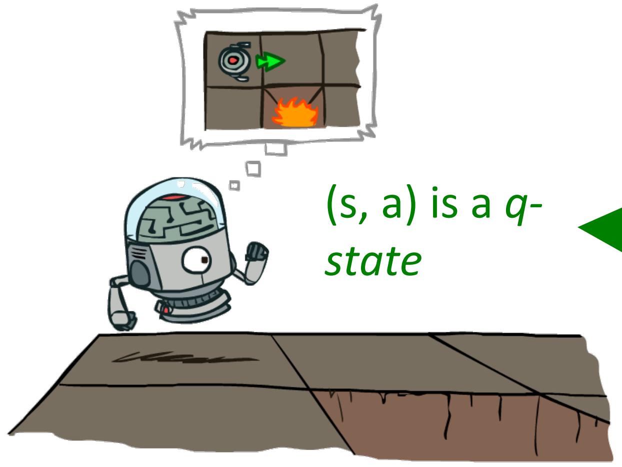
Q-states

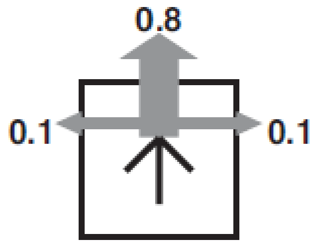
See lecture 16

- The combination of a state *s* and action *a,* denoted by *(s,a)*, is called a Q-state
    - It represents being in state *s* and having taken action *a*
    - still, due to uncertainty, we don't know what the outcome of the action will be
- The value of a q-state is called Q-value









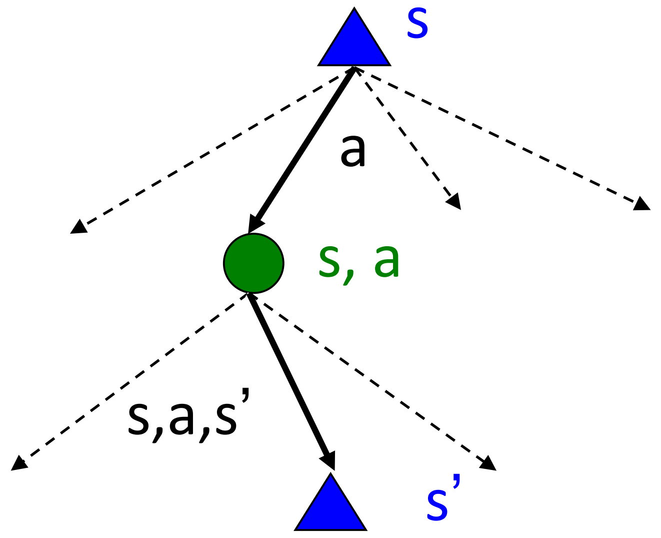
Optimal V* and Q* quantities

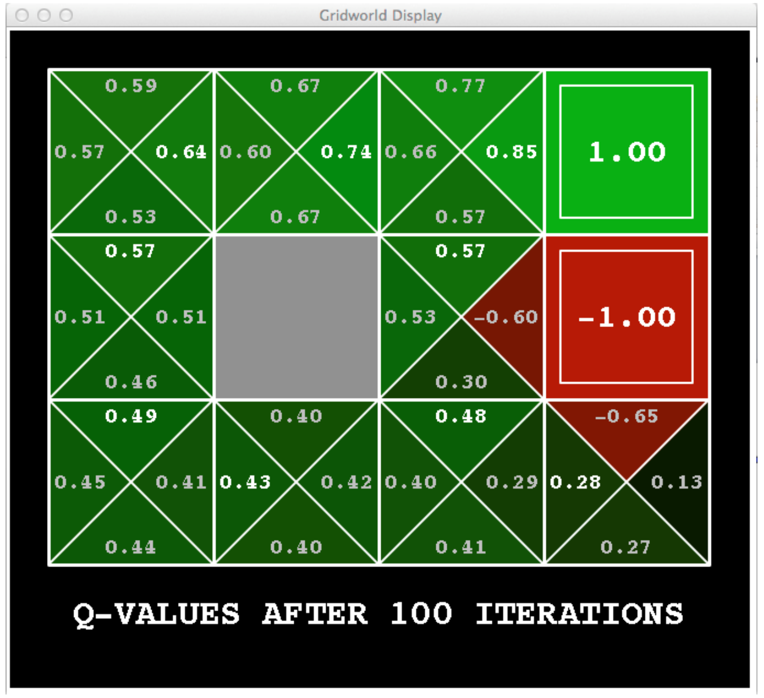
See lecture 16

- The value (utility) of a state *s*:
  - *V*(s)* = expected utility starting in *s* and acting optimally from that point onwards
- The value (utility) of a q-state *(s,a)*:
  - *Q*(s,a)* = expected utility starting out having taken action *a* from state *s* and acting optimally from that point onwards

- Acting optimally: a rational agent should choose the action that maximizes the expected utility of the subsequent state
  - π*(s) = optimal action from state *s*

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s,a)$$





Bellman equations

See lecture 16

- The utility of a state is the immediate reward *R(s,a,s')* for that state plus the expected discounted utility of the next state *γV*(s')*, assuming that the agent chooses the optimal action.

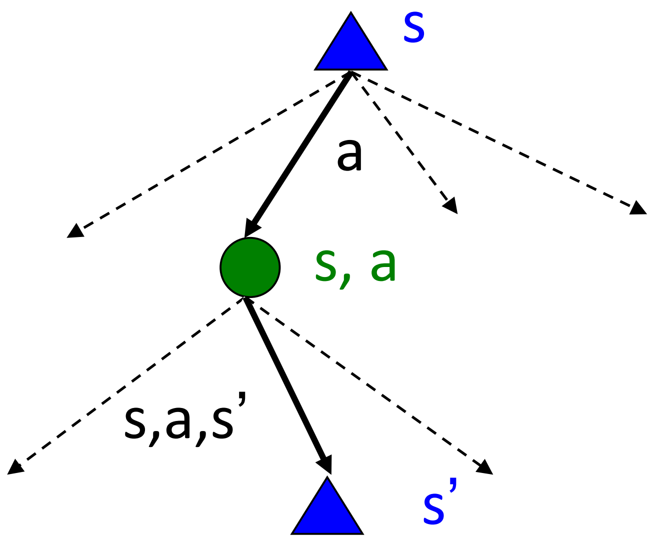
$$V^*(s) = \max_a \underbrace{\sum_{s'} T(s,a,s') \left[\underbrace{R(s,a,s') + \gamma V^*(s')}_{} \right]}_{}$$

  - Due to the non-deterministic actions, we should compute the expected value
  - We assume the agent acts optimally

- Also defined for Q-states *(s,a)*

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

$$V^*(s) = \max_a Q^*(s,a)$$



From MDPs to Reinforcement Learning

See lecture 17

- In Reinforcement Learning (RL)
  - We still have an MDP
    - A set of states $s \in S$
    - A set of actions (per state) A
    - A transition model T(s,a,s')
    - A reward function R(s,a,s')
  - Still looking for a policy $\pi(s)$

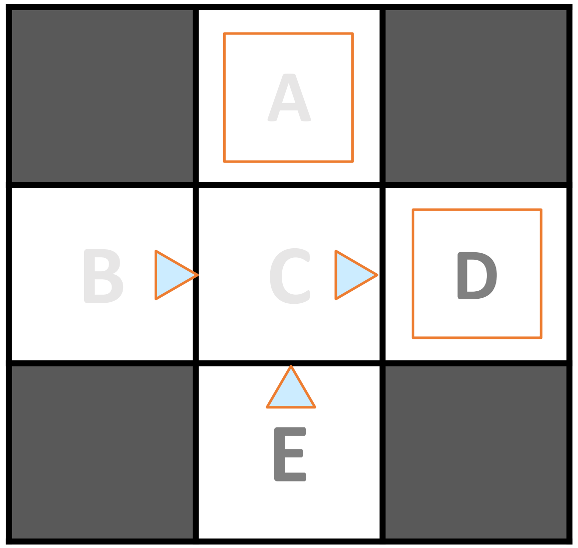
- New twist: we don't know T, R
  - i.e. we don't know which states are good or what the actions do
- So, we must actually try out actions and states to learn

So RL can solve MDP problems when we don't know the MDP

Collecting experience

See lecture 17

- The agent collects experience/data via its interaction with the environment
- Tuples *(s,a,s',r)* are known as samples
- A collection of samples until arriving at a terminal state is known as episode
  - (s_0,a_0,s_1,r_0) (s_1,a_1,s_2,r_1) … $(s_{n-1},a_{n-1},s_n,r_{n-1})$



Observed Episodes (Training)

Episode 1

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

Episode 2

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

Episode 3

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

Episode 4

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

Key ideas for learning

See lecture 17

- Online vs Offline/Batch learning
  - Learn while exploring the world, or learn from fixed batch of data?
- Active vs. Passive Learning
  - Does the learner actively choose actions to gather experience or, is a fixed policy π provided?
- Model-based vs. Model-free Learning
  - Do we estimate $T(s,a,s')$ and $R(s,a,s')$, or just learn values/policy directly?

- What we will (quickly ☹) cover in the next 3-4 lectures
  - Model-based learning
  - Model-free learning
    - Passive RL (policy evaluation using: episodes (Monte Carlo policy evaluation) or samples (TD-learning)
    - Active RL (TD Q-learning)
  - Value-function approximation (Approximate (deep) Q-learning)

Model-based learning

See lecture 17

- Model-based idea:
  - (Step 1) Learn an approximate model of *T*, *R* based on experiences/data
  - (Step 2) Solve the MDP based on the learned *T, R*

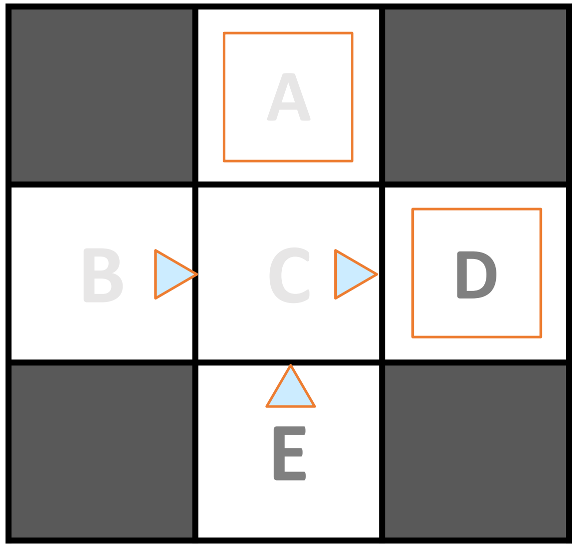
- **Step 1:** Learn empirical MDP model
  - Count outcomes *s'* for each q-state *(s, a)*
  - Normalize to give an estimate of $\hat{T}(s,a,s')$
  - Discover each $\widehat{R}(s,a,s')$ estimate when we experience *(s, a, s')*

- **Step 2:** Solve the learned MDP
  - For example, use value iteration or policy iteration (see previous slides)

Example: Model-based learning – Step 1: Learn the empirical model

See lecture 17

Input Policy π



Assume: γ = 1

Observed Episodes (Training)

Episode 1

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

Episode 2

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

Episode 3

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

Episode 4

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

Learned Model

$\hat{T}(s, a, s')$

T(B, east, C) = 1.00
T(C, east, D) = 0.75
T(C, east, A) = 0.25
…

$\hat{R}(s, a, s')$

R(B, east, C) = -1
R(C, east, D) = -1
R(D, exit, x) = +10
…

Assumption: the reward is deterministic

Model-free learning

See lecture 18

- Model-based learning: learns an approximate model of T, R based on experience and uses this model to solve a (conventional) MDP.
- Model-free learning: learns the v-values of states or q-values of state-action pairs directly, without constructing a model of the rewards and transitions in the MDP

- We will cover
  - Direct policy evaluation or Monte-Carlo policy evaluation (i.e., policy evaluation using episodes)
  - Temporal difference (TD) learning (i.e., policy evaluation using samples)
  - Q-learning

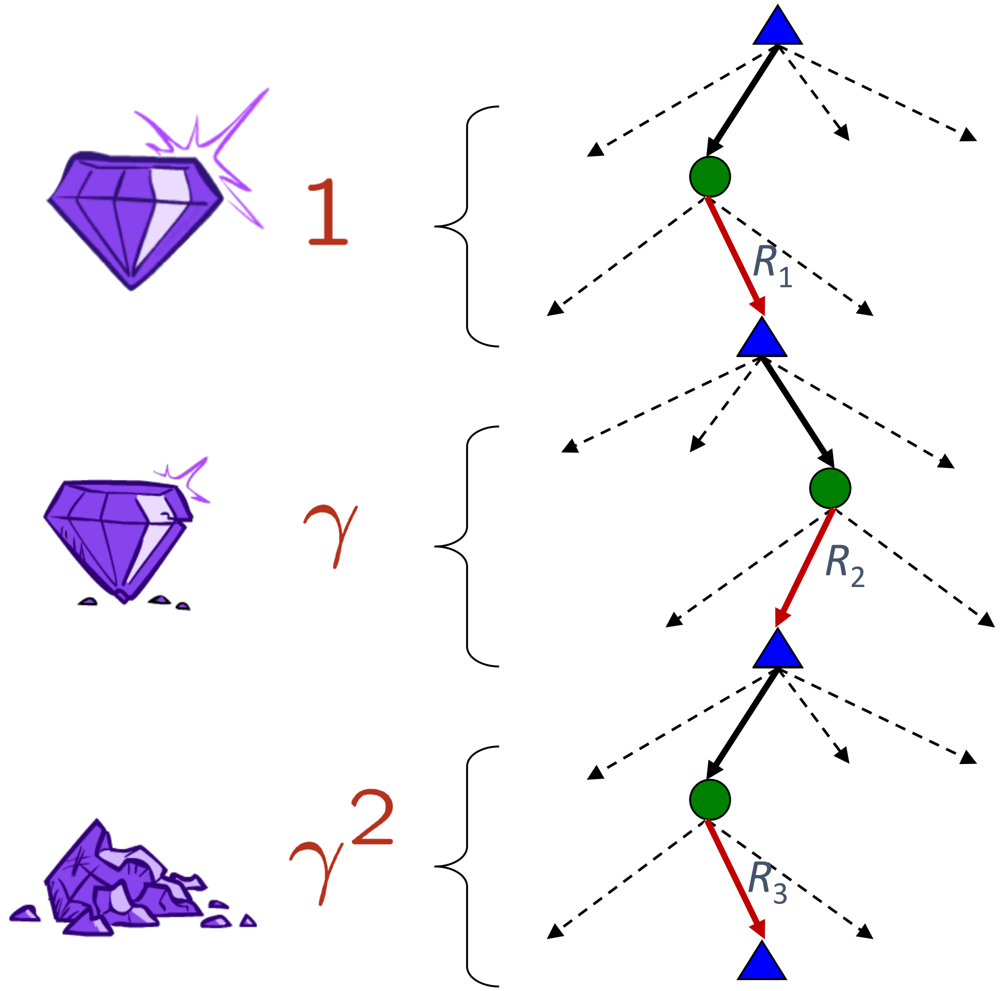
Passive RL: An agent is given a policy to follow and learns the values of the states under that policy as its experience grows

Active RL: The agent can use the feedback it receives to iteratively update its policy

Monte-Carlo policy evaluation (MC-learning)

See lecture 18

- **Goal**: Compute values for each state under a policy π
- **Idea**: Have the agent learn from experience while following π
- The experience comes in form of full episodes
  - Monte Carlo sampling
- More concretely:
  - Act according to π
  - Every time you visit a state *s*, keep track of its utility (sum of discounted rewards) as well as of the number of visits
    - $U[s_o, s_1, \ldots] = R(s_o) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots + \gamma^n R(s_n)$
  - Average those samples to get the estimated value of *s*
- This is called direct policy evaluation

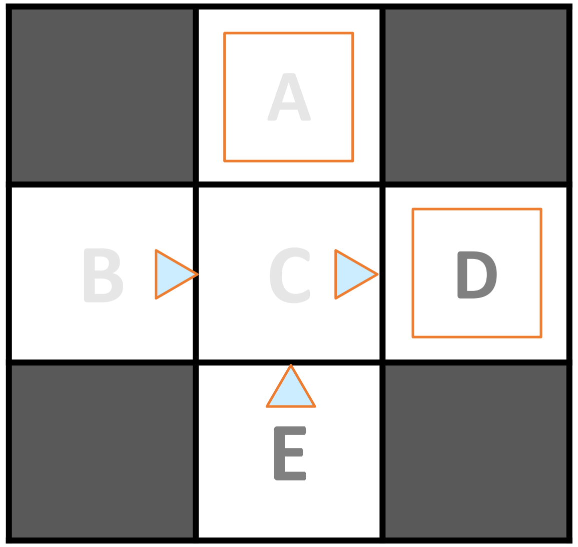


Example: Monte-Carlo policy evaluation

See lecture 18

$U[s_o, s_1, \ldots] = R(s_o) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots + \gamma^n R(s_n)$

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

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

Episode 2

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

Episode 3

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

Episode 4

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

Output Values

Episodes	**1**
A	
B	-1-1+10=8
C	-1+10=9
D	10
E	

Example: Monte-Carlo policy evaluation

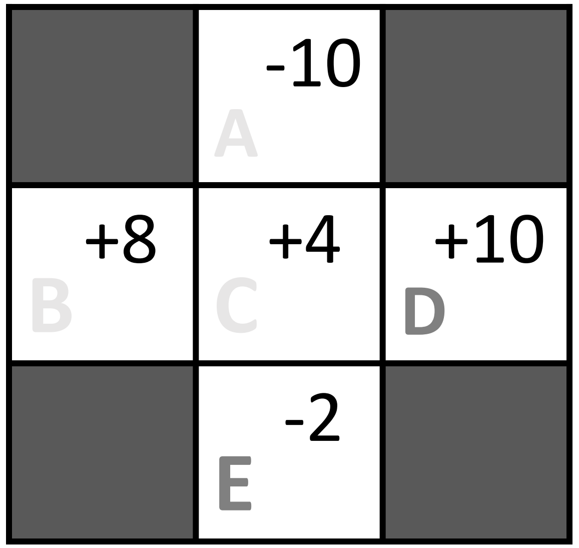
See lecture 18

- We can now estimate the values for each state s

Episodes	1	2	3	4
A				-10
B	8	8		
C	9	9	9	-11
D	10	10	10	
E			8	-12

s	Total Reward	Times Visited	$V^{\pi}(s)$
A	-10	1	-10
B	16	2	8
C	16	4	4
D	30	3	10
E	-4	2	-2

Output Values

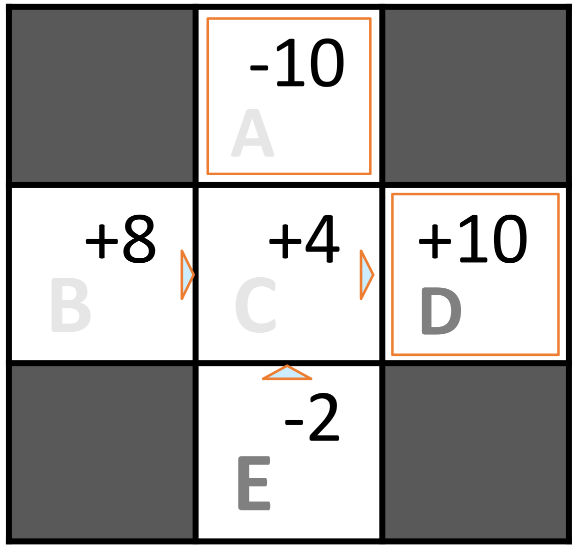


See lecture 18

Monte-Carlo policy evaluation: Discussion

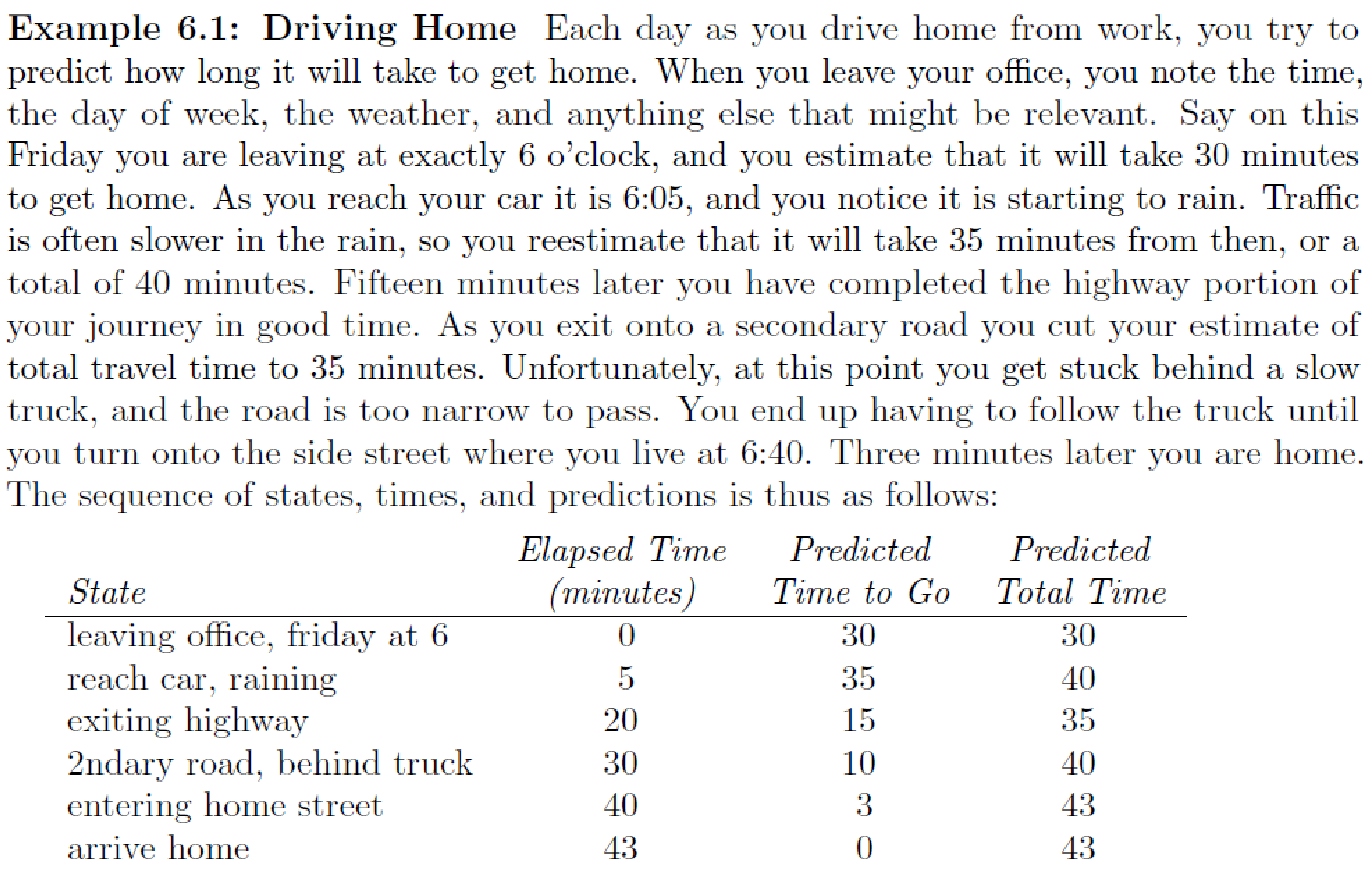
- Batch approach: It collects data and learns the values at the end of training
    - Can be also maintained online
- Passive approach: A fixed policy π is provided
- Model-free approach: learn directly the values of the states
- What is good about direct evaluation?
    - Easy to understand
    - Doesn't require any knowledge of T, R
    - It eventually computes the correct average values, using just sample transitions
- What is bad about direct evaluation?
    - Each state must be learned separately
        - No share of information between states (unlike Bellman equations)
    - The method needs (complete) episodes (Monte Carlo sampling)

Output Values



From Monte-Carlo evaluation to Temporal Difference-learning

- In Monte-Carlo evaluati
  - We must wait till the
- In TD-learning the idea i
- Driving home example



Example 6.1: Driving Home Each day as you drive home from work, you try to predict how long it will take to get home. When you leave your office, you note the time, the day of week, the weather, and anything else that might be relevant. Say on this Friday you are leaving at exactly 6 o'clock, and you estimate that it will take 30 minutes to get home. As you reach your car it is 6:05, and you notice it is starting to rain. Traffic is often slower in the rain, so you reestimate that it will take 35 minutes from then, or a total of 40 minutes. Fifteen minutes later you have completed the highway portion of your journey in good time. As you exit onto a secondary road you cut your estimate of total travel time to 35 minutes. Unfortunately, at this point you get stuck behind a slow truck, and the road is too narrow to pass. You end up having to follow the truck until you turn onto the side street where you live at 6:40. Three minutes later you are home. The sequence of states, times, and predictions is thus as follows:

State	*Elapsed Time (minutes)*	*Predicted Time to Go*	*Predicted Total Time*
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

Source: Barto and Sutton book, Chapter 6

TD Learning policy evaluation

See lecture 18

- **Main idea:** learn from every experience (sample)
  - Update V(s) each time we get a new experience/sample, i.e., transition (s, a, s', r)
  - Policy still fixed, so a=π(s), so we are still doing evaluation!
- How do we update V(s) based on just one sample?
  - **Main idea:** Correct the old estimate $V^{\pi}(s)$ with the new sample value
    - Estimate value of $V^{\pi}(s)$ *based on sample: (s, a, s', r)*

$$sample = R(s, \pi(s), s') + \gamma V^{\pi}(s')$$

    - Update: Move current value estimate *V(s)* values towards value of whatever successor occurs (*s'*)
      - The learning rate *α* controls how to combine our current estimate with the new sampled estimate, $0 \leq \alpha \leq 1$

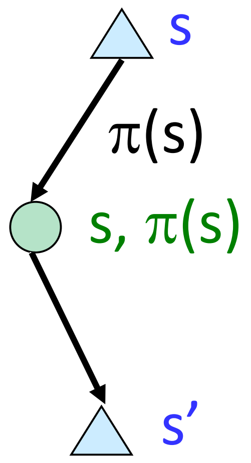
$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha)sample$$

    - Same formula different form

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(\boxed{sample - V^{\pi}(s)})$$

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha[difference]$$

$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\left[R(s, \pi(s), s') + \gamma V^{\pi}(s')\right]$$

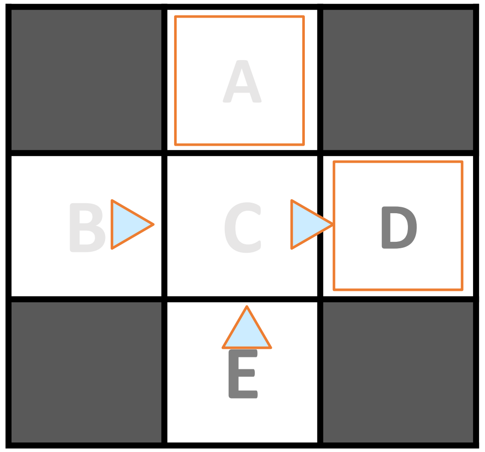


Example: Temporal Difference Learning

See lecture 18

$V^\pi(C) = (1-\alpha)V^\pi(C)+\alpha[R(C,\text{east},D)+\gamma V^\pi(D)]$ $=(1-0.5)*0+0.5[-2+8]=3$ Only C is updated

States

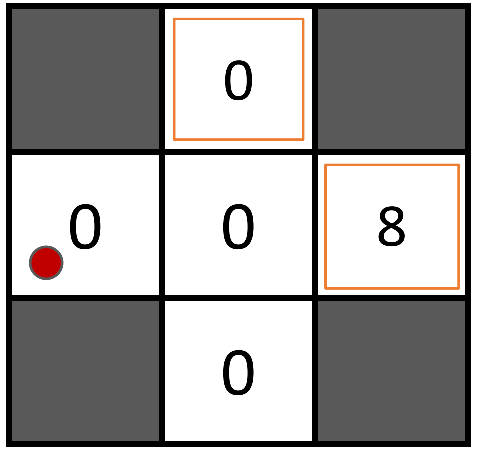


Assume: $\gamma = 1$, $\alpha = 1/2$

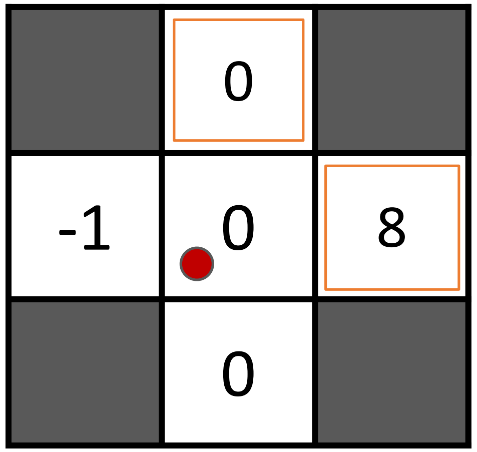
Observed Transitions (samples)

B, east, C, -2

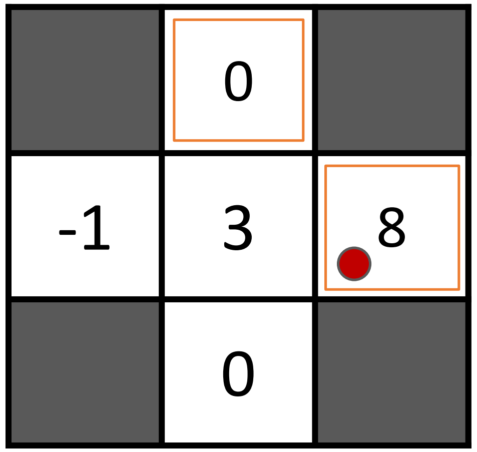
C, east, D, -2



Current estimates



Updated estimates



Updated estimates

$$V^\pi(s) \leftarrow (1-\alpha)V^\pi(s) + \alpha\left[R(s,\pi(s),s') + \gamma V^\pi(s')\right]$$

$V^\pi(B) = (1-\alpha)V^\pi(B)+\alpha[R(B,\text{east},C)+\gamma V^\pi(C)]$ $=(1-0.5)*0+0.5[-2+0]=-1$ Only B is updated

TD Learning policy evaluation: discussion

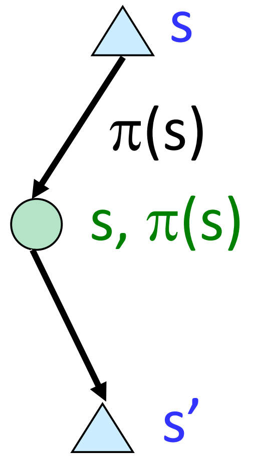
See lecture 18

- So, with a simple update rule $V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + (\alpha)sample$ we are able to
  - Learn with each sample → online learning approach
    - Don’t need to wait till the end of the episode as with Monte-Carlo learning
    - Works also for non-terminating environments
  - Give exponentially less weight to older, potentially less accurate sample estimates
  - Converge to learning the true value states much faster comparing to Monte-Carlo policy evaluation

- TD learning is an example of on-policy learning:
  - A policy π is followed and information from policy-dependent sampling of the value function is not used immediately to improve the policy

Other important aspects (not covered in this lecture)

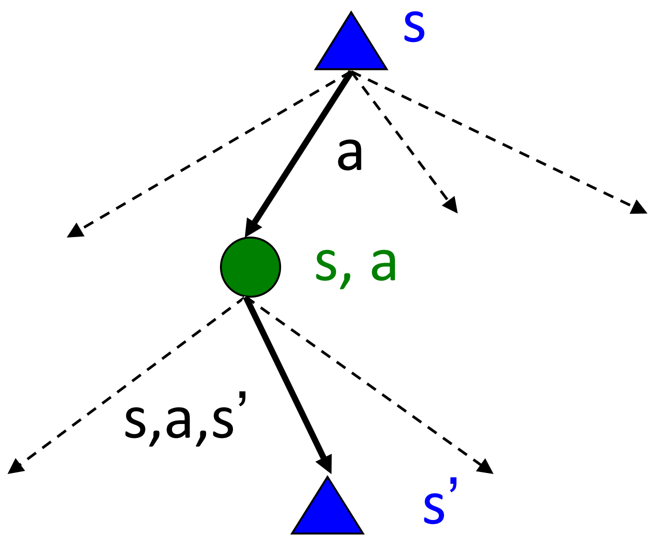
- What is discussed thus far is the so-called TD(0)
  - This is the simplest form of TD learning, where after each step the value function is updated with information from the next state
  - So, the update relies on information from just 1 step
- The idea of TD(λ) is to use more information from n steps
  - Like a combination of TD(0) – where we just look in 1 step - and MC-learning – where we look till the end of the episode.
  - There are two different perspectives of TD(λ)
    - the forward view and
    - the backward view (eligibility traces).



$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha\left[R(s,\pi(s),s') + \gamma V^{\pi}(s')\right]$$

From TD V learning to TD Q-learning

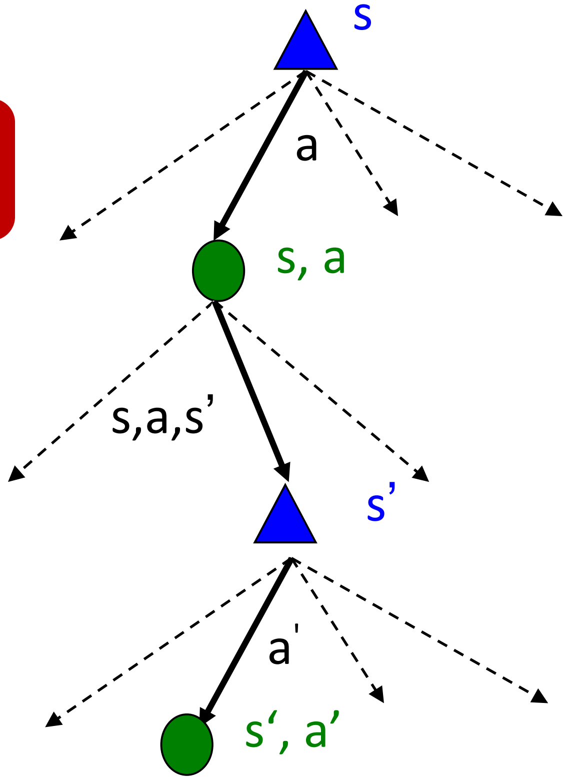
- TD value learning is a model-free way to do policy evaluation, mimicking Bellman updates with running sample averages
- But still we learn V-values and therefore cannot convert them into a (new) policy
  - ➔TD Q-learning



Temporal Difference Q-learning

See lecture 18

In lecture 18 please correct s'' into s' in the figure and formulas



- **Main idea:** learn from every experience (sample)
  - Update Q(s,a) each time we get a new experience/sample, i.e., transition (s, a, s', r)
- How do we update Q(s,a) based on just one sample?
  - **Main idea:** Correct the old estimate with the new sample value
  - Consider your old estimate *Q(s,a)*
  - Consider your new sample estimate:

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

  - Update the old estimate using the new sample estimate (*a* is the learning rate):

$$Q(s,a) \leftarrow (1-a)Q(s,a) + \alpha[sample]$$

  - Same formula different form

$$Q(s,a) \leftarrow Q(s,a) + \alpha(\boxed{sample - Q(s,a)}) \qquad Q(s,a) \leftarrow Q(s,a) + \alpha[difference]$$

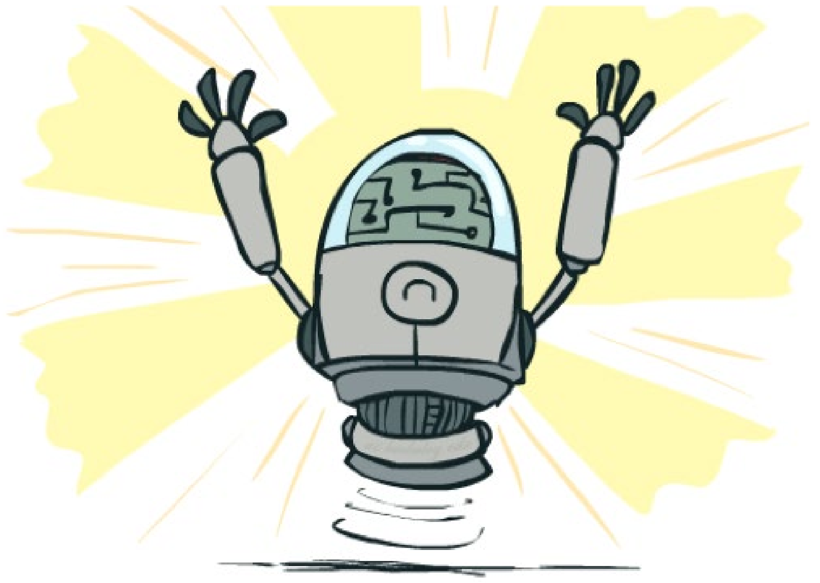
$$Q(s,a) \leftarrow (1-a)Q(s,a) + \alpha[R(s,a,s') + \gamma \max_{a'} Q(s',a')]$$

Temporal-difference Q-Learning: discussion

See lecture 18

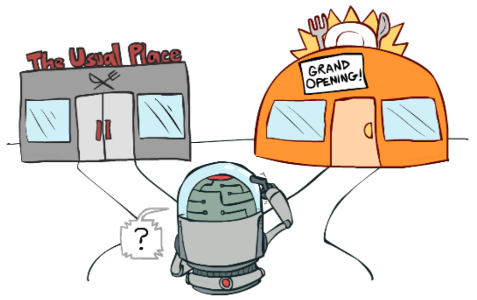
- Q-learning converges to optimal policy -- even if you're acting suboptimally!
  - This is called off-policy learning
- Caveats:
  - You have to explore enough
  - You have to eventually make the learning rate *a* small enough

  … but not decrease it too quickly



Exploration vs Exploitation dilemma

See lecture 18

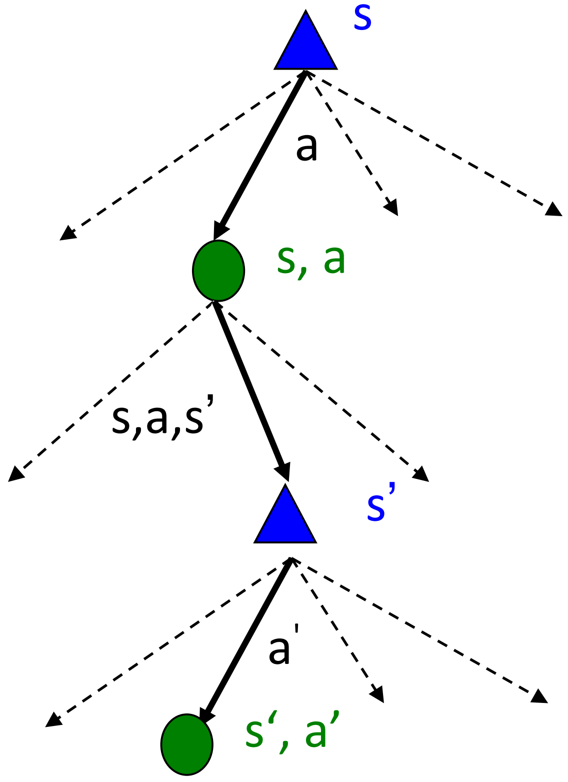


- Exploitation: Make best decision given current information
- Exploration: Gather more information that might lead us to better decisions in the future
- Different exploration schemes
  - ε-greedy policies: Explore with probability ε and exploit with probability 1-ε, 0≤ε ≤1
    - Explore means select a random action; exploit means select based on current policy, i.e., a = $\arg_a \max Q(s,a)$
    - Fixed ε-greedy policy vs ε-decaying greedy policy
  - Exploration functions: give some preference to unvisited states, by modify q-value update to consider the frequency of visiting a particular state

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \cdot [R(s,a,s') + \gamma \max_{a'} f(s',a')]$$

    - An example of an exploration function $f(s,a) = Q(s,a) + \frac{k}{N(s,a)}$

Other important aspects

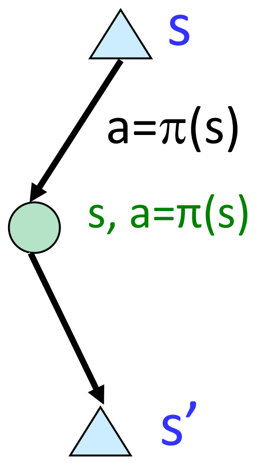


- In TD Q-learning we followed the optimal policy (i.e., max operation)

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[R(s,a,s') + \gamma \max_{a'} Q(s',a')]$$

- Another approach, called SARSA, is to choose an action a' following the same policy

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[R(s,a,s') + \gamma\, Q(s',a')$$



- Q-learning considers the best possible case if you get to the next state, while SARSA considers the reward if we follow the current policy at the next state.
  - If our policy is greedy, they are the same.
- Q-learning is off-policy learning; SARSA is on-policy learning

Key methods for both MDP and RL

- Policy evaluation
  - Given a fixed policy π calculate the utility of each state s if π was to be executed
  - For full MDP, see lecture 16-17
  - For RL, see lecture 18
    - With full episodes (see Monte-Carlo policy evaluation)
    - With samples (TD-learning)
- Policy iteration
  - Policy improvement following a 2 step approach until convergence
    - Step 1: Policy evaluation: calculate utilities for some fixed policy $\pi()$(not optimal utilities!) until convergence
    - Step 2: Policy improvement: with fixed values from step 1, find the best action according to one-step look-ahead and follow that
  - For full MDP, see lecture 16-17

Outline

- Recap (with some additions)
- Approximate learning
- Things you should know from this lecture & reading material

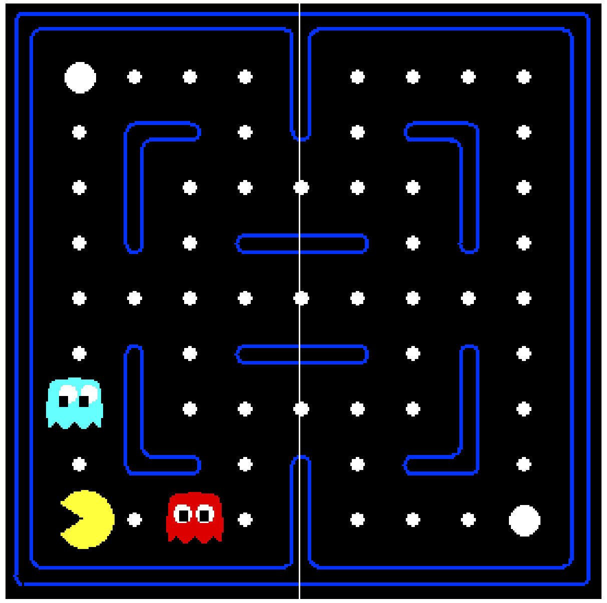
Generalizing across states

- Vanilla Q-Learning keeps a look-up table of all states
- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
    - Training samples, training time, …
  - Too many states to hold the q-tables in memory

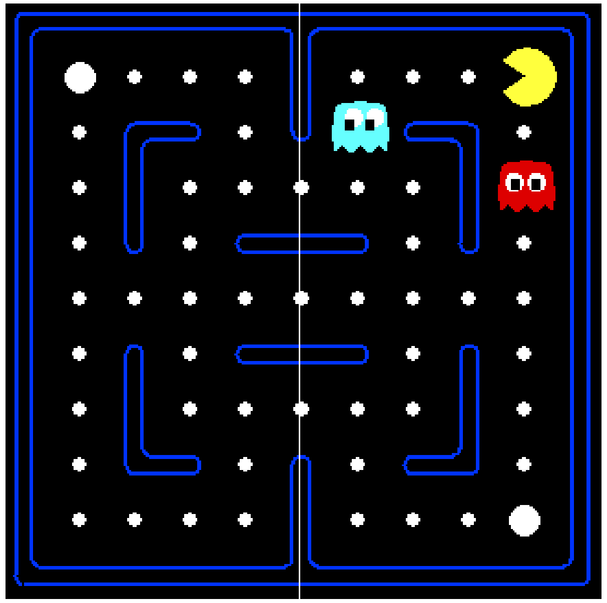
- Solution for large MDPs:
  - Estimate value function with function approximation
  - Generalize from seen states to unseen states
  - This is the fundamental idea in machine learning

Example: Pacman

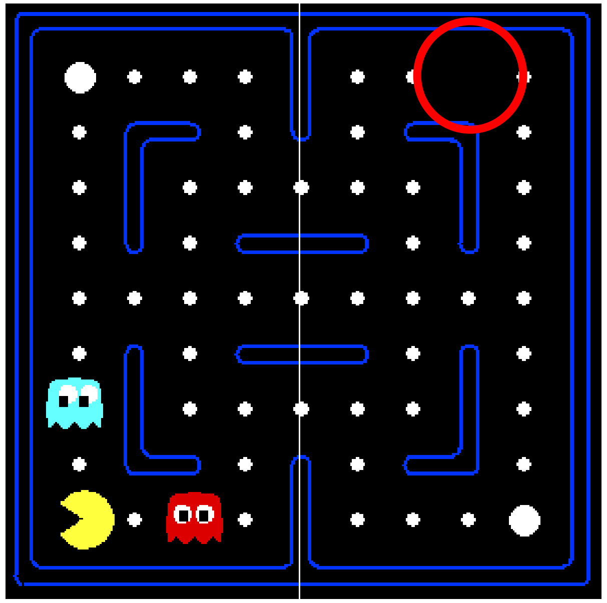
Let's say we discover through experience that this state is bad:



In naïve/vanilla Q-learning, we know nothing about this state



Or even this one!

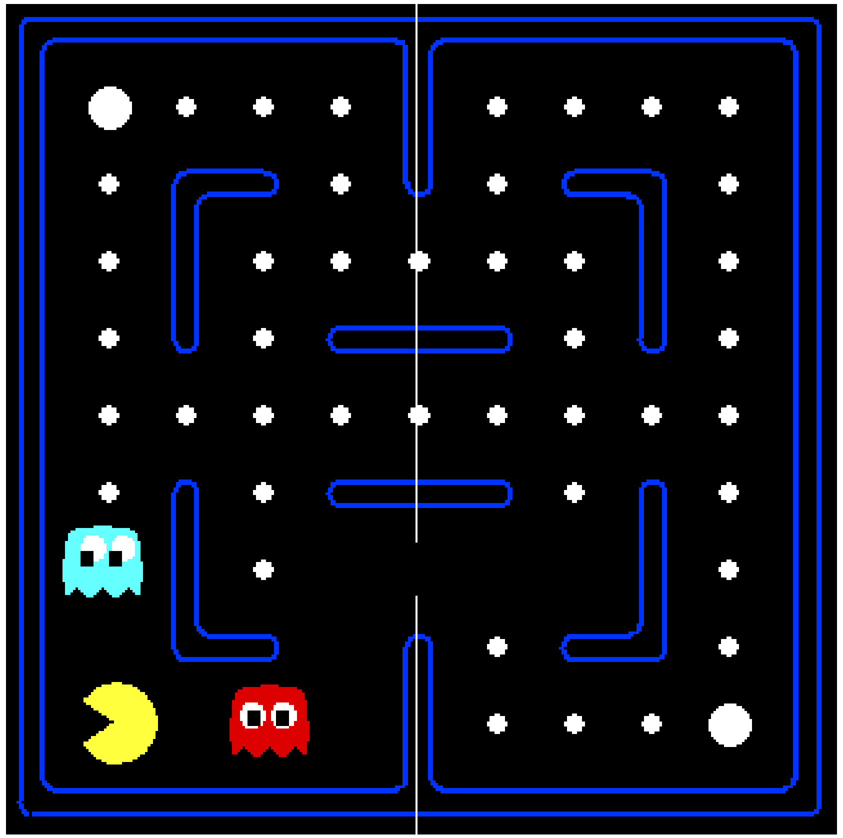


Generalizing across states

- Solution for large MDPs:
  - Estimate value function with function approximation
  - Generalize from seen states to unseen states
  - This is the fundamental idea in machine learning
- How do we represent states (or state-action pairs)?
  - Feature-based state representation
  - Features can be
    - Manually engineered → hand-crafted features (traditional ML)
    - Automatically obtained via ML → learned features (deep learning)

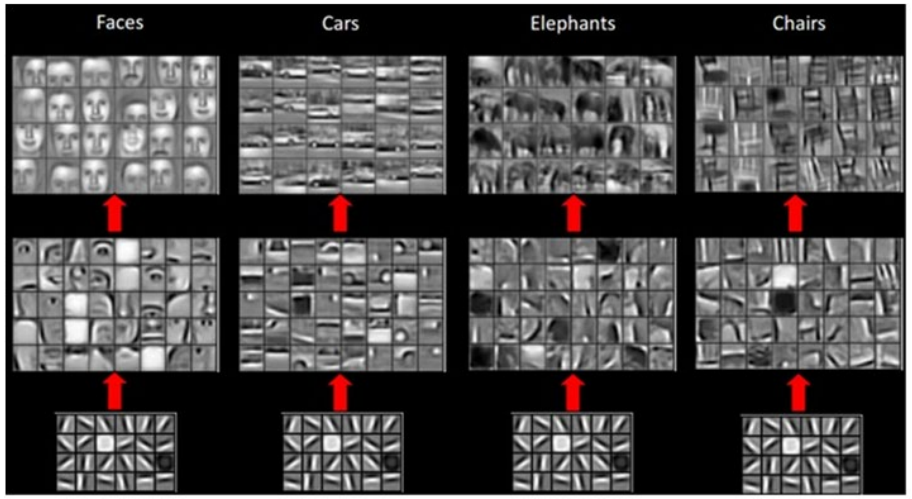
Feature-based state representation

- Idea: describe states using features (properties of the state) → feature representation
- Hand-crafted features: features manually engineered
- The feature representation depends on the problem per se
    - Example features for Packman:
        - Distance to closest ghost → f_1
        - Distance to closest dot → f_2
        - Number of ghosts → f_3
        - 1 / (dist to dot)2 → f_4
        - Is Pacman in a tunnel? (0/1) → f_5
        - … → f_n
- A similar representation could be also used for q-states (s,a)
    - Can also describe a q-state (s, a) with features (e.g. action moves closer to food)



Feature-based state representation

- **Learned features**: automatically obtained from a machine learning algorithm
  - Very useful for e.g., images, text etc
- Such features are typically obtained using Deep Learning
  - E.g., a Convolutional Neural Network (CNN) tries to extract the most useful features for classifying an image

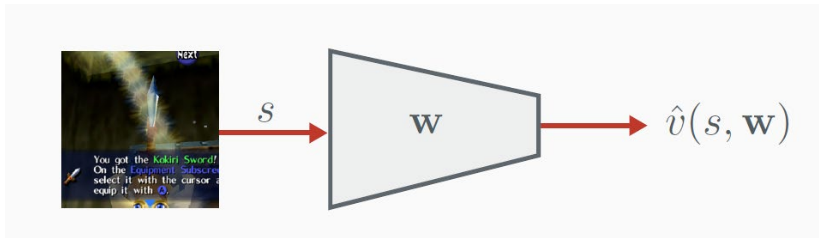


Input data: images

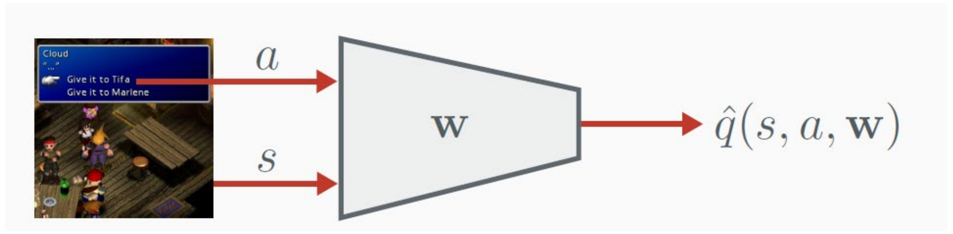
Different approaches

w are the function parameters

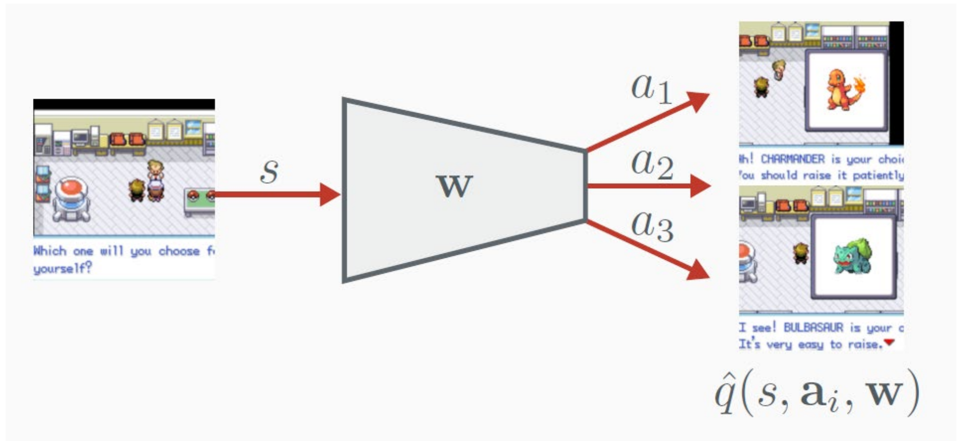
- States s to v-values



- Q-states (s,a) to q-values



- States s to q-values



Which function approximator?

- Approximate Q-learning: Use a function approximator to estimate the action-value function
$$Q(s,a,\theta) \approx Q^*(s,a)$$
function parameters
    - Goal: find the model (parameters θ) that minimizes the loss
- There are many function approximators, e.g.
    - Linear combinations of features
    - Neural networks
    - Decision trees
    - Nearest neighbors
- Which one to choose?
    - Typically differentiable function approximators like linear combination of features and NNs

Disclaimer: I use either w or θ to denote the model parameters

w typically refers to a NN model

Disclaimer w.r.t. function approximation: I am also using the term (ML) model sometimes.

- If the function approximator is a deep neural network => deep q-learning!

Challenges with function approximation

- Naïve idea: use ML to learn such a model/function
- This seems like a regression problem where the class attribute is the value of a state

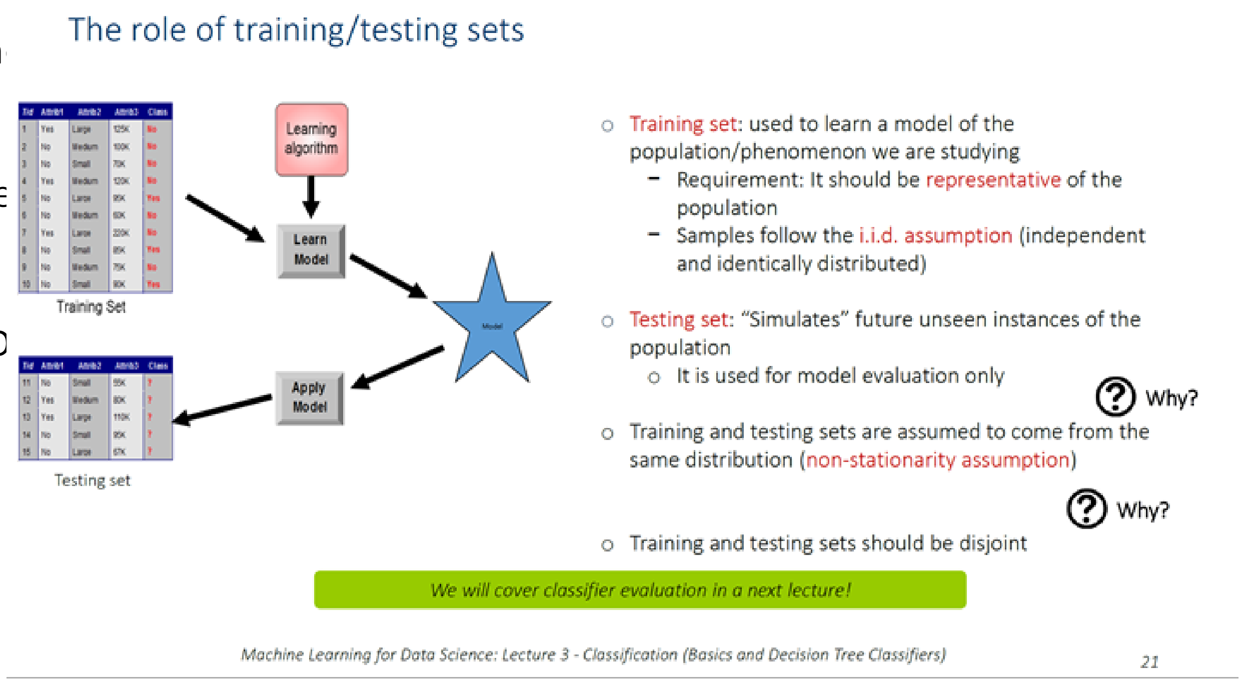
House ID	Size (feet)	Old	Price
1	500	10	100K
2	1000	20	500K
3	2000	50	300K
4	300	15	200K

What is the predicted price for a house of a certain size and age?

- Challenge: What are the target values in our case?
  - Solution: Substitute them with a target

Challenges with function aproximation

- Challenges w.r.t. training of such a model
  - Data is no-stationary
    - As you explore and discover new rewards, th
  - Data is not i.i.d.
    - When playing a game, all subsequent experie
- But thus far, we assumed that stationarity ho
  - Recall esp. lecture 3 and 6
- Solution: Change training: experience replay



The role of training/testing sets

Tid	Attrib1	Attrib2	Attrib3	Class
1	Yes	Large	125K	No
2	No	Medium	100K	No
3	No	Small	70K	No
4	Yes	Medium	120K	No
5	No	Large	95K	Yes
6	No	Medium	60K	No
7	Yes	Large	220K	No
8	No	Small	85K	Yes
9	No	Medium	75K	No
10	No	Small	90K	Yes

Training Set

Tid	Attrib1	Attrib2	Attrib3	Class
11	No	Small	55K	?
12	Yes	Medium	80K	?
13	Yes	Large	110K	?
14	No	Small	95K	?
15	No	Large	67K	?

Testing set

- Training set: used to learn a model of the population/phenomenon we are studying
  - Requirement: It should be representative of the population
  - Samples follow the i.i.d. assumption (independent and identically distributed)
- Testing set: "Simulates" future unseen instances of the population
  - It is used for model evaluation only

Why?

- Training and testing sets are assumed to come from the same distribution (non-stationarity assumption)

Why?

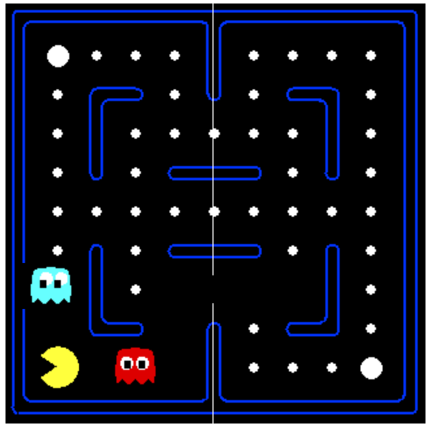
- Training and testing sets should be disjoint

We will cover classifier evaluation in a next lecture!

Machine Learning for Data Science: Lecture 3 - Classification (Basics and Decision Tree Classifiers) 21

Approximate Q-learning with linear functions

- In the simplest case, we can define a value function as a linear combination of the different feature values (w_i is the weight of feature f_i):



$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \quad = \sum_{i=1}^{n} w_i f_i(s)$$

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a) \quad = \sum_{i=1}^{n} w_i f_i(s,a)$$

- How about the model parameters θ (these are feature weights w_i)?
- This seems like a regression problem where the class attribute is the value of a state
  - But we don't have the target values
  - Solution: (as mentioned already) substitute the target value (see next slide)

Approximate Q-learning with linear functions

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

- Q-learning with linear Q-functions:
  - Sample/instance *(s,a,r,s')*
  - Sample estimate $sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$
  - Value difference:

This represents the target value

$$difference = sample - Q(s,a) = \boxed{[R(s,a,s') + \gamma \max_{a'} Q(s',a')]} - Q(s,a)$$

  - Exact Q-learning

$$Q(s,a) \leftarrow Q(s,a) + \alpha\,[\text{difference}]$$

  - Approximate Q-learning

$$w_i \leftarrow w_i + \alpha\,[\text{difference}]\, f_i(s,a)$$

We don't maintain the values, rather we tune the weights of the features

Approximate Q-learning with linear functions

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

- Looking at the weights update rule

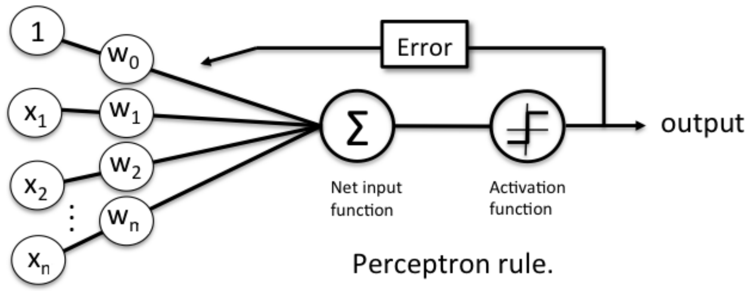
The target value

$$w_i \leftarrow w_i + \alpha\,[\text{difference}]\, f_i(s,a)$$

difference = sample-Q(s,a) = $\boxed{[R(s,a,s') + \gamma \max_{a'} Q(s',a')]}$*-Q(s,a)*

- It is similar to the perceptron update rule (lecture 8)
  - Goal: learn a classifier that minimizes L2 loss

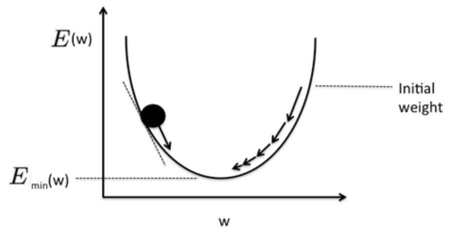
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n \qquad E[\vec{w}] \equiv \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$$



Perceptron rule.

- Idea: At each step, we take a step into the opposite direction of the gradient, and the step size is determined by the value of the learning rate η as well as the slope of the gradient.

$$\Delta\vec{w} = -\eta\nabla E[\vec{w}] \qquad \nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

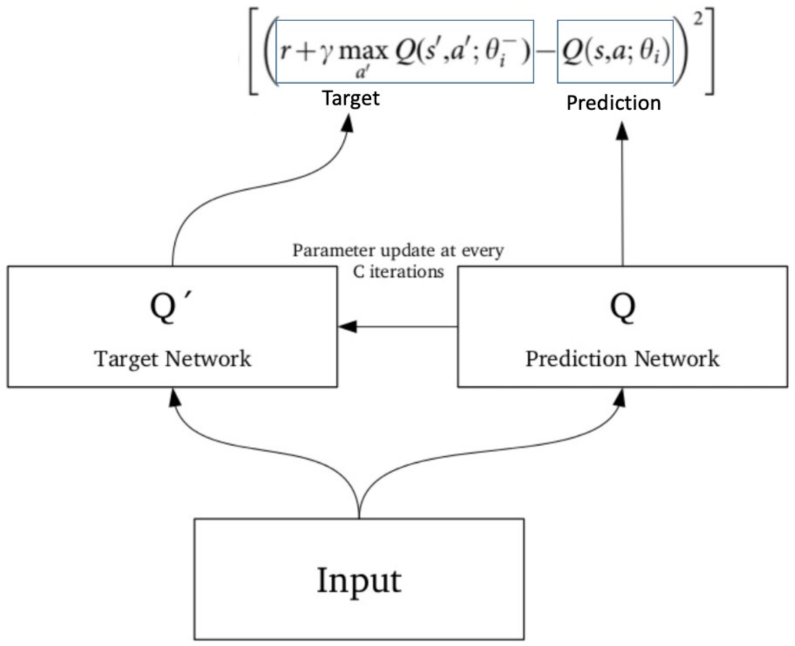


- Perceptron training rule (slide 28 lecture8):

$$\Delta w_i = \eta(t - o)x_i$$

Approximate Q-learning

- **Target network:** Note that the parameters from the previous iteration θ_{i-1} are fixed and not updated. In practice we use a snapshot of the network parameters from a few iterations ago instead of the last iteration. This copy is called the *target network*.



$\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$
Target
Prediction
Parameter update at every C iterations
Q´
Target Network
Q
Prediction Network
Input

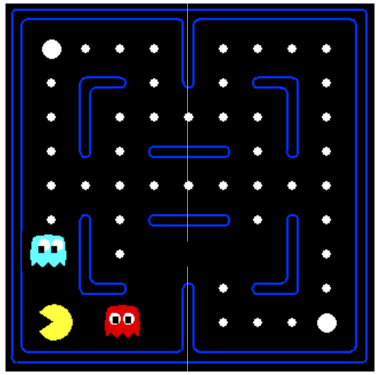
Source: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

Example: Q-Pacman

$$w_i \leftarrow w_i + \alpha\,[\text{difference}]\,f_i(s,a)$$

$$Q(s,a) = 4.0 f_{DOT}(s,a) - 1.0 f_{GST}(s,a)$$

s



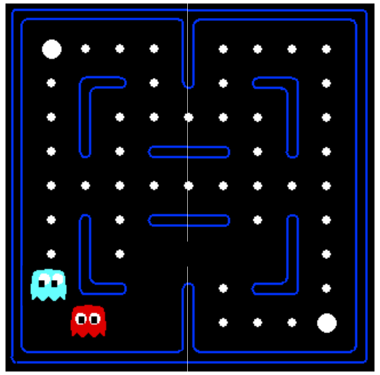
$f_{DOT}(s, \text{NORTH}) = 0.5$

$f_{GST}(s, \text{NORTH}) = 1.0$

$a = \text{NORTH}$

$r = -500$

s'



Current estimate $Q(s, \text{NORTH}) = +1$ i.e., 4*0.5-1*1

$Q(s', \cdot) = 0$

Sample estimate $r + \gamma \max_{a'} Q(s', a') = -500 + 0$

$\text{difference} = -501$ Update weights

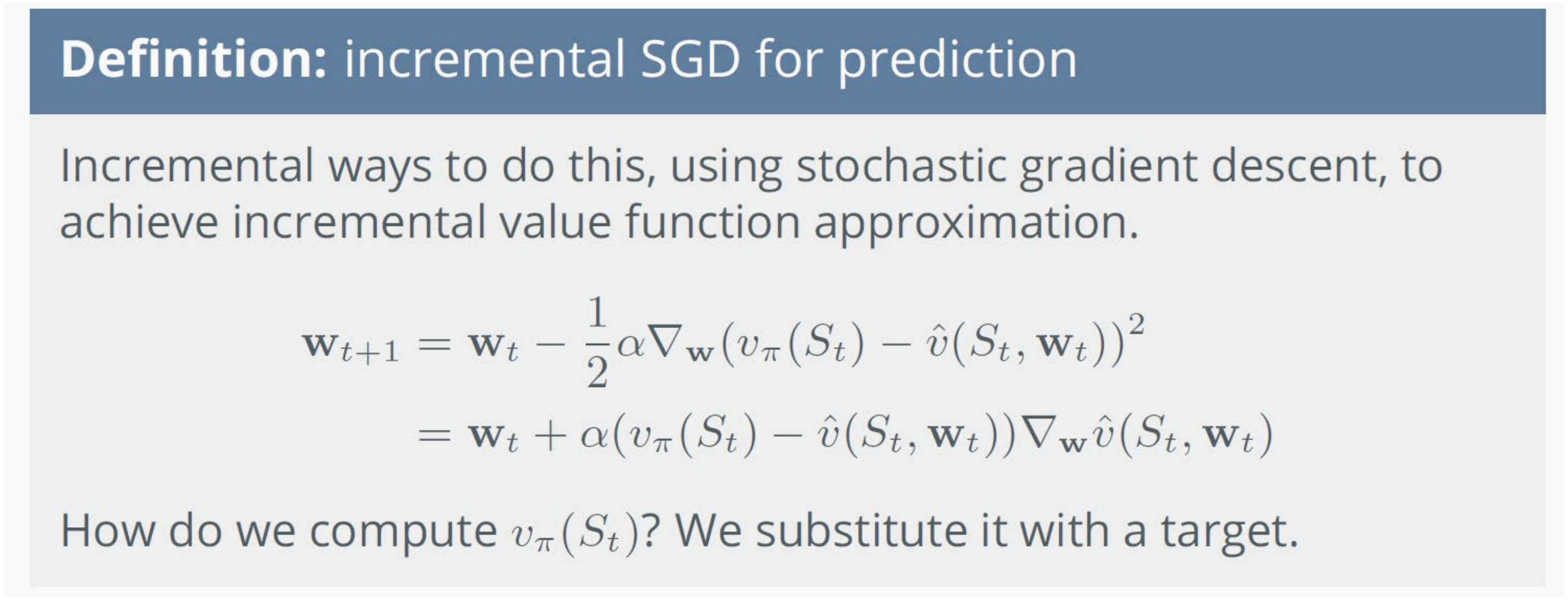
$w_{DOT} \leftarrow 4.0 + \alpha\,[-501]\,0.5$

$w_{GST} \leftarrow -1.0 + \alpha\,[-501]\,1.0$

α=0,4%

$$Q(s,a) = 3.0 f_{DOT}(s,a) - 3.0 f_{GST}(s,a)$$

Incremental methods stochastic gradient descent for V-values

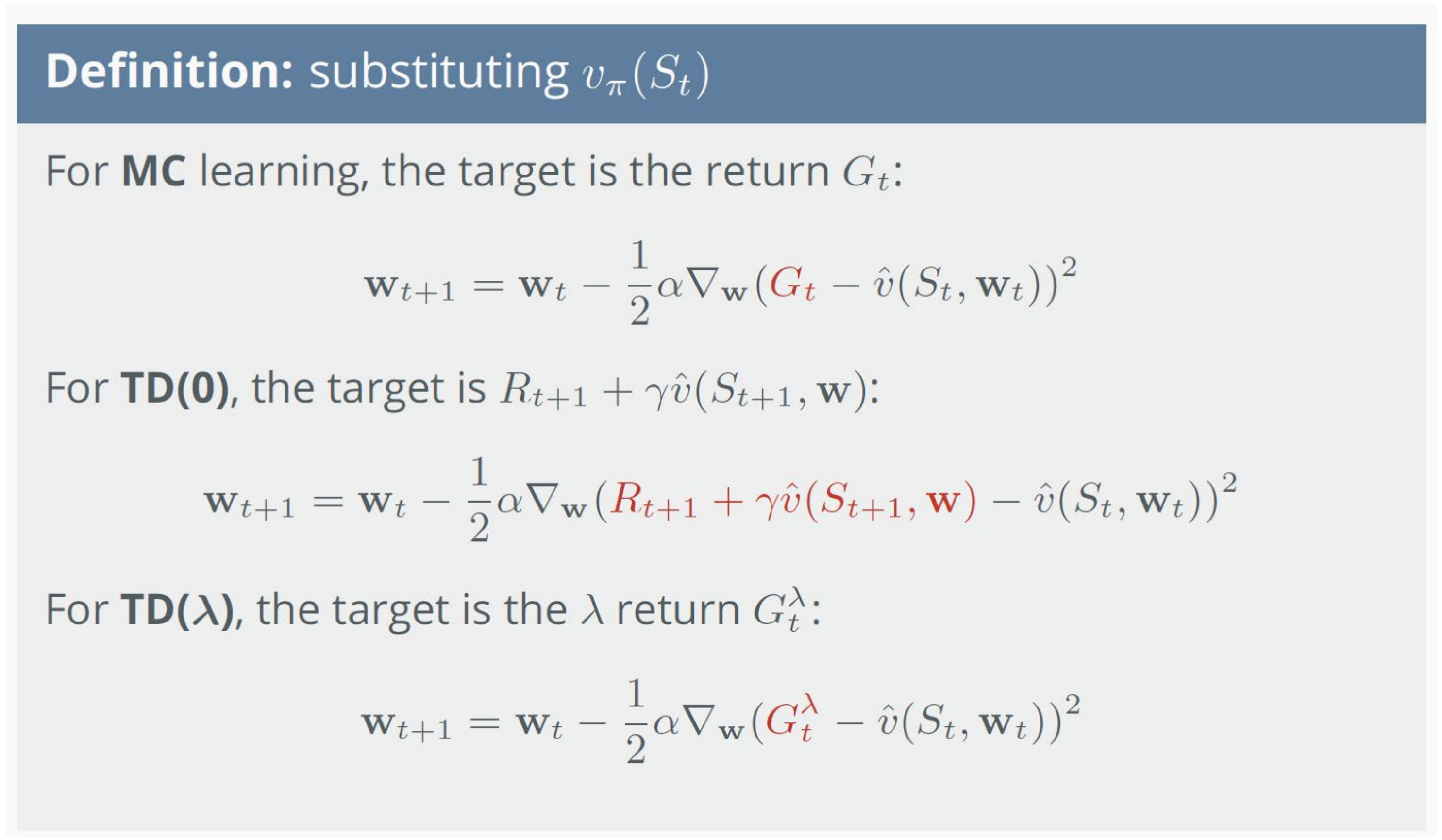

Definition: incremental SGD for prediction

Incremental ways to do this, using stochastic gradient descent, to achieve incremental value function approximation.

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}\left(v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right)^2 \\ &= \mathbf{w}_t + \alpha\left(v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right)\nabla_{\mathbf{w}}\hat{v}(S_t, \mathbf{w}_t)\end{aligned}$$

How do we compute $v_\pi(S_t)$? We substitute it with a target.

Incremental methods stochastic gradient descent for V-values



Definition: substituting $v_\pi(S_t)$

For **MC** learning, the target is the return G_t:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}(G_t - \hat{v}(S_t, \mathbf{w}_t))^2$$

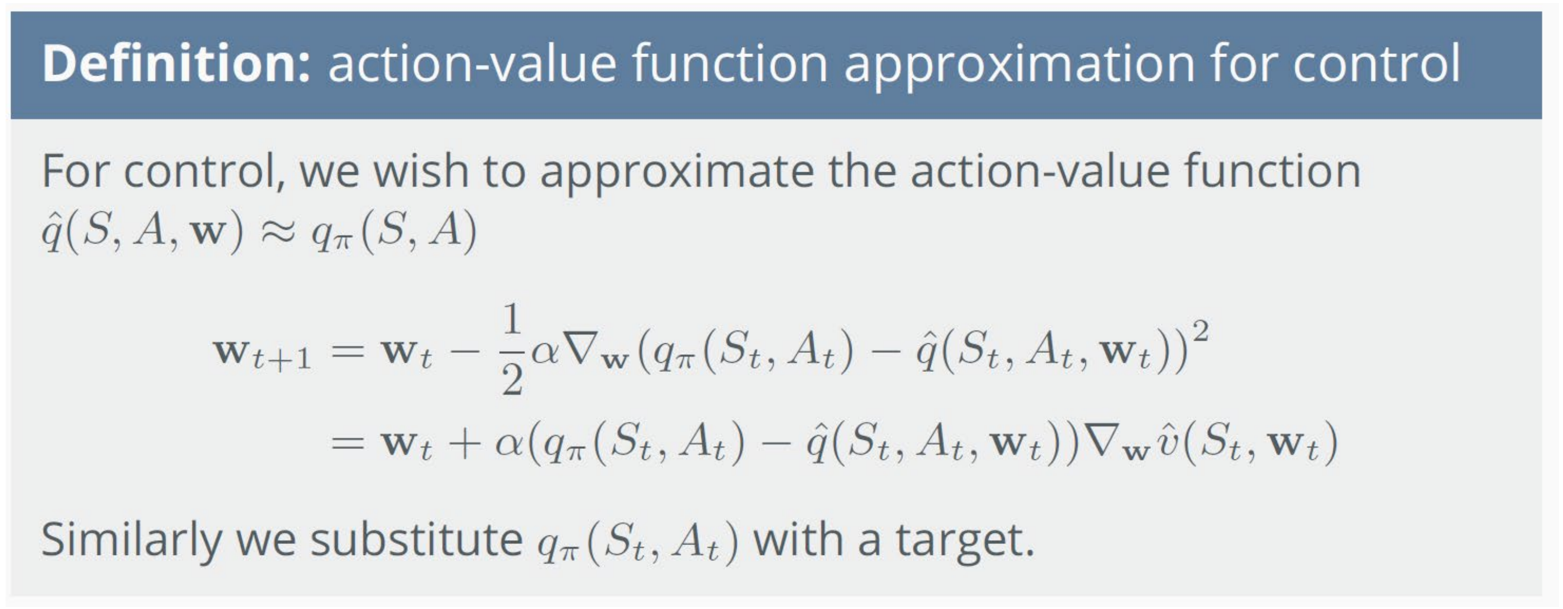
For **TD(0)**, the target is $R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w})$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}_t))^2$$

For **TD(λ)**, the target is the λ return G_t^λ:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}(G_t^\lambda - \hat{v}(S_t, \mathbf{w}_t))^2$$

Incremental methods stochastic gradient descent for Q-values



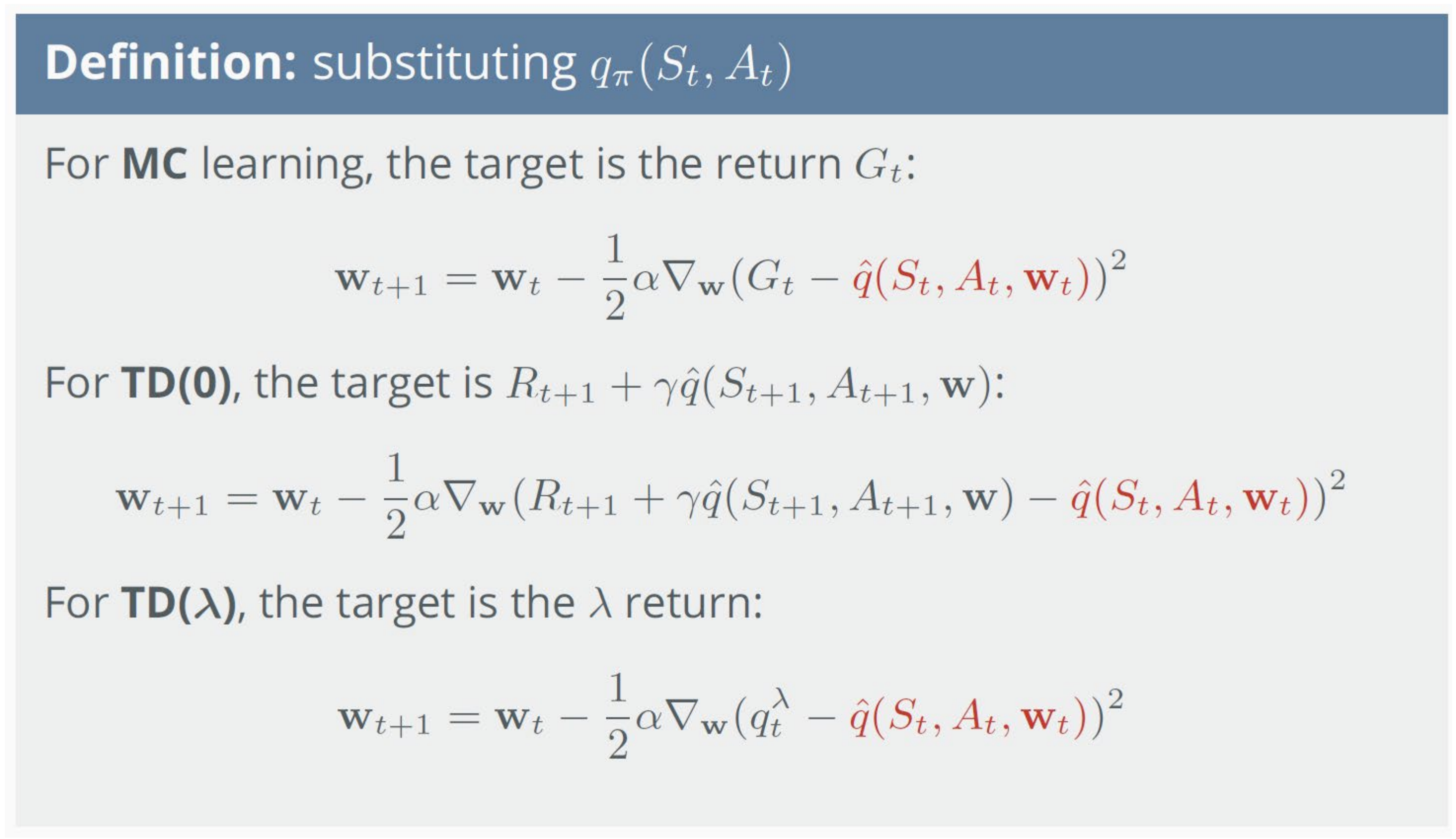
Definition: action-value function approximation for control

For control, we wish to approximate the action-value function $\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$

$$
\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha \nabla_{\mathbf{w}} (q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, \mathbf{w}_t))^2 \\
&= \mathbf{w}_t + \alpha (q_\pi(S_t, A_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}_t)
\end{aligned}
$$

Similarly we substitute $q_\pi(S_t, A_t)$ with a target.

Incremental methods stochastic gradient descent for Q-values



Definition: substituting $q_\pi(S_t, A_t)$

For **MC** learning, the target is the return G_t:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}(G_t - \hat{q}(S_t, A_t, \mathbf{w}_t))^2$$

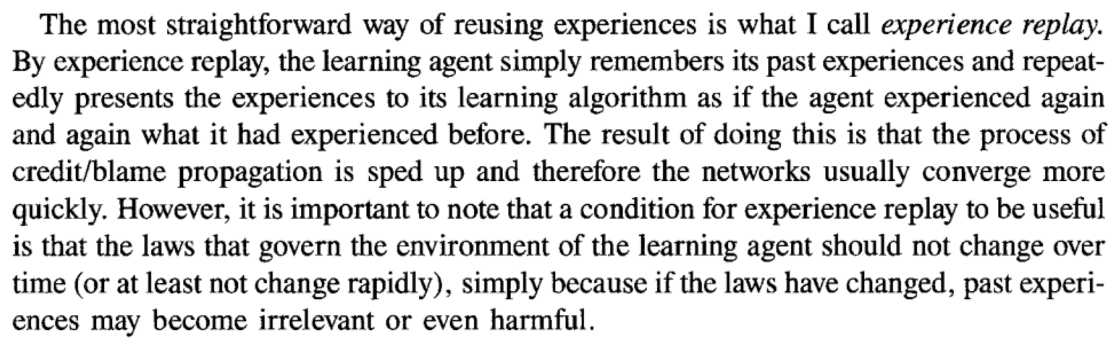
For **TD(0)**, the target is $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w})$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}_t))^2$$

For **TD(λ)**, the target is the λ return:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}(q_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}_t))^2$$

Experience replay



The most straightforward way of reusing experiences is what I call *experience replay*. By experience replay, the learning agent simply remembers its past experiences and repeatedly presents the experiences to its learning algorithm as if the agent experienced again and again what it had experienced before. The result of doing this is that the process of credit/blame propagation is sped up and therefore the networks usually converge more quickly. However, it is important to note that a condition for experience replay to be useful is that the laws that govern the environment of the learning agent should not change over time (or at least not change rapidly), simply because if the laws have changed, past experiences may become irrelevant or even harmful.

- Incremental methods have several problems:
  - They are not sample efficient
  - They have strongly correlated updates that break the i.i.d. assumptions of popular SGD algorithms
  - They may rapidly forget rare experiences that would be useful later on
- Solution: Reuse experience/data and replay them (Experience replay)
  - Store experience in an experience replay buffer
  - Replay experience
    - (Vanilla) experience replay: all experiences are equally important [see more]
    - Prioritized experience replay: reply important transitions/samples more frequently [see more]

Outline

- Recap (with some additions)
- Approximate learning
- Things you should know from this lecture & reading material

Summary

- Function approximation allows generalization to new unseen states
- Making the training stable is a problem and there are solutions/tricks towards this
- A fast evolving domain with many new ideas/methods e.g.:
  - Double Q-learning [link] and [link]
  - Dueling Network Architectures [link]
  - …

Overview and Reading

- Overview
  - Approximate learning

- Reading
  - Chapter 23, AI book 4th edition
  - Chapter 6 (Sections 6.1, 6.2, 6.5), RL book
  - Experience reply: https://link.springer.com/content/pdf/10.1007/BF00992699.pdf
  - Prioritized experience reply: https://arxiv.org/abs/1511.05952
  - Introduction to Reinforcement Learning with David Silver (DeepMind)
  - Stanford CS234: Reinforcement Learning with Emma Brunskill

Hands on experience

- Work on project 3

Thank you

Questions/Feedback/Wishes?

Acknowledgements

- The slides are based on
    - CS 188 | Introduction to Artificial Intelligence, Berkeley
    - Artificial Intelligence: A modern approach (Russel and Norvig), 4th edition
    - Chris G. Willcocks RL lecture: https://cwkx.github.io/teaching.html