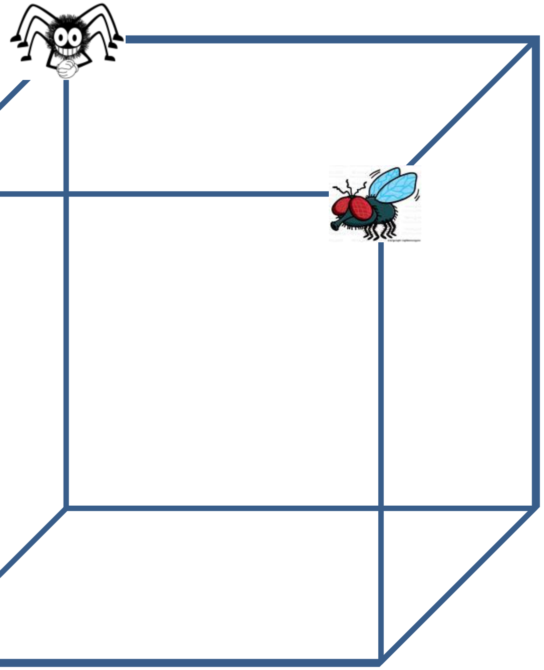
Reinforcement Learning

Spring 2018

RL!

Recap: Model-Free Assumption



- Can see the fly
- Know the distance to the fly
- Know possible actions (get closer/farther)
- But have no idea of how the fly will respond
  - Will it move, and if so, to what corner

Recap: Model-Free Methods

- AKA model-free **reinforcement learning**

- How do you find the value of a policy, without knowing the underlying MDP?
  - Model-free *prediction*
- How do you find the optimal policy, without knowing the underlying MDP?
  - Model-free *control*

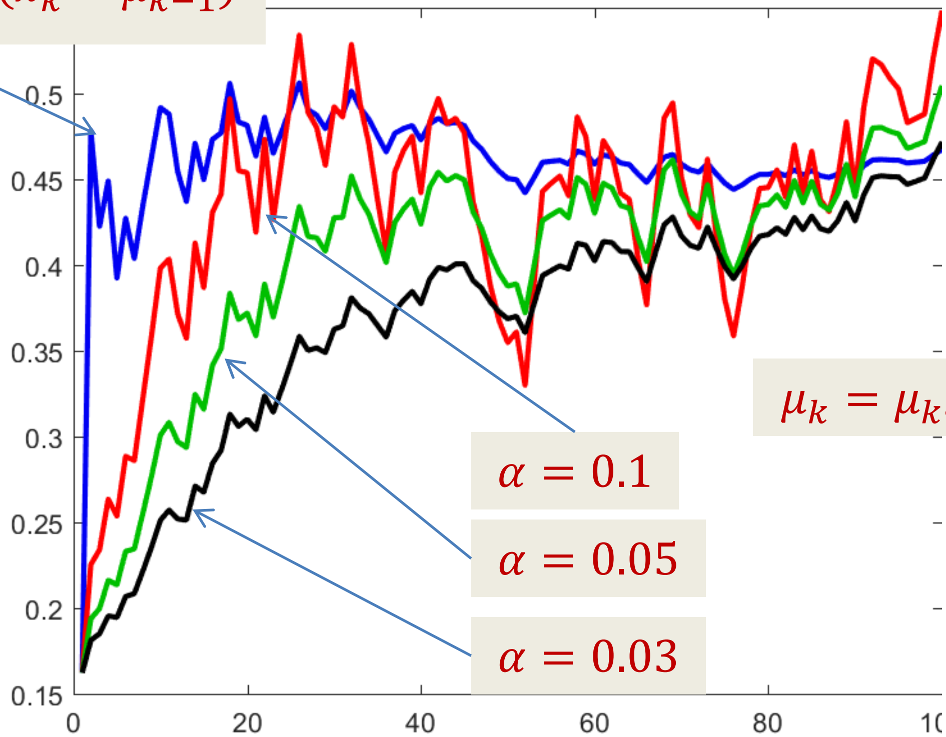
Recap: Methods

- *Monte-Carlo* Learning

- *Temporal-Difference* Learning
  - *TD(1)*
  - *TD(K)*
  - *TD*(λ)

Recap: Incremental Updates

$$\mu_k = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$



$$\mu_k = \mu_{k-1} + \alpha(x_k - \mu_{k-1})$$

- Correct equation is *unbiased* and converges to true value
- Equation with α is *biased* (early estimates can be expected to be wrong) but *converges* to true value

Recap: TD(1)

- For all s Initialize: $v_\pi(s) = 0$
- For every episode e
  - For every time $t = 1 \ldots T_e$
    - $v_\pi(S_t) = v_\pi(S_t) + \alpha\big(R_{t+1} + \gamma v_\pi(S_{t+1}) - v_\pi(S_t)\big)$
- There's a "lookahead" of one state, to know which state the process arrives at at the next time
- But is otherwise online, with continuous updates

Recap: TD(N) with lookahead

$$v_\pi(S_t) = v_\pi(S_t) + \alpha\delta_t(N)$$

- Where

$$\delta_t(N) = R_{t+1} + \sum_{i=1}^{N} \gamma^i R_{t+1+i} + \gamma^{N+1} v_\pi(S_{t+N}) - v_\pi(S_t)$$

- $\delta_t(N)$ is the TD *error* with *N* step lookahead

Recap: TD(λ)

$$G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t(n)$$

- Combine the predictions from all lookaheads with an exponentially falling weight
  - Weights sum to 1.0

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t^{\lambda} - V(S_t)\right)$$

Recap: TD(λ)

- Maintain an eligibility trace for *every* state

$$E_0(s) = 0$$
$$E_t(s) = \lambda\gamma E_{t-1}(s) + 1(S_t = s)$$

- Computes total weight for the state until the present time

Recap: TD(λ)

- At every time, update the value of *every state* according to its eligibility trace

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

$$V(s) \leftarrow V(s) + \alpha \delta_t E_t(s)$$

- Any state that was visited will be updated
  - Those that were not will not be, though

Model-Free Methods

- AKA model-free **reinforcement learning**

- How do you find the value of a policy, without knowing the underlying MDP?
  - Model-free *prediction*
- How do you find the optimal policy, without knowing the underlying MDP?
  - Model-free *control*

Value vs. Action Value

- The solution we saw so far only computes the *value function*

- Not sufficient, even if we knew the optimal values
  - To select the optimal action given the optimal values, we will need extra information, namely transition probabilities
  - Which we do not have

- Instead, we use the same method to compute the optimal *action value* functions
  - Optimal policy in any state : Choose the action that has the largest *optimal* action value

Value vs. Action Value

- Given only value functions, the optimal policy must be estimated as:

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \; \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s')$$

  - Needs knowledge of transition probabilities
- Given action value functions, we can find it as:

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \, Q(s, a)$$

- This is *model free* (no need for knowledge of model parameters)

TD(1) with action-values

- For all s, a, initialize:

$$q_\pi(s,a) = 0$$

- For every episode e
  - For every time $t = 1 \ldots T_e$

$$\hat{A}_{t+1} \sim \pi(S_{t+1})$$

$$\delta_t = R_{t+1} + \gamma q_\pi(S_{t+1}, \hat{A}_{t+1}) - q_\pi(S_t, A_t)$$

$$q_\pi(S_t, A_t) = q_\pi(S_t, A_t) + \alpha \delta_t$$

TD(λ) with action-values

For all s, a, initialize:

$$q_\pi(s,a) = 0$$
$$E_t(s,a) = 0$$

- For every episode e
  - For every time $t = 1 \dots T_e$

$$E_t(s,a) = \lambda\gamma E_{t-1}(s,a) + 1(S_t = s \wedge A_t = a)$$
$$\hat{A}_{t+1} \sim \pi(S_{t+1})$$
$$\delta_t = R_{t+1} + \gamma q_\pi(S_{t+1}, \hat{A}_{t+1}) - q_\pi(S_t, A_t)$$
$$q(s,a) \leftarrow q(s,a) + \alpha\delta_t E_t(s,a)$$

Optimal Policy: Control

- We learned how to estimate the state value functions for an MDP whose transition probabilities are unknown ***for a given policy***

- *How do we find the optimal policy?*

Problem of optimal control

- From a series of episodes of the kind:

$$S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, R_4, \dots, S_T$$

- Find the optimal action value function $q_*(s, a)$
  - The optimal policy can be found from it
- Ideally do this *online*
  - So that we can continuously improve our policy from *ongoing experience*

Control: Greedy Policy

- Recall the steps in policy iteration:
  - Start with any policy $\pi^{(0)}$
  - Iterate ($k = 0$ … convergence)
    - Find the value function $v_{\pi^{(k)}}(s)$ using DP
    - Find the greedy policy

$$\pi^{(k+1)}(\mathrm{s}) = \mathrm{argmax}_a \left(R_s^a + \gamma \sum_{s'} P_{s,s'}^a v_{\pi^{(k)}}(s') \right)$$

- Can we adapt this for model-free control?

Control: Greedy Policy

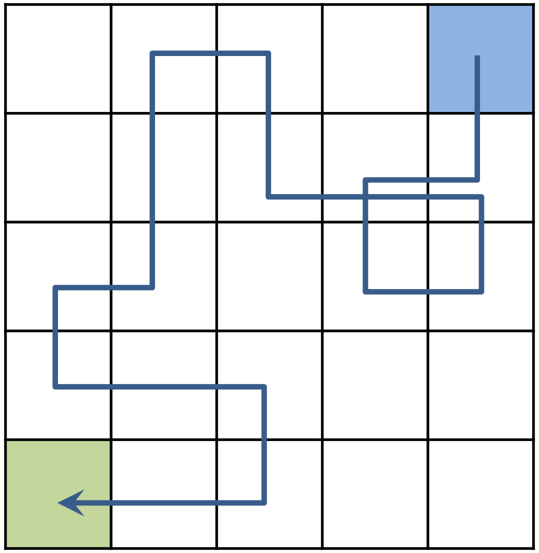
- Our proposed algorithm:
  - Start with any policy $\pi^{(0)}$
  - Iterate ($k = 0 \ldots$ convergence)
    - Estimate the action-value function $q_{\pi^{(k)}}(s, a)$ using TD-learning
    - Find the greedy policy

$$\pi^{(k+1)}(\mathrm{s}) = \mathrm{argmax}_a \left(q_{\pi^{(k)}}(s, a) \right)$$

- Let's see if this works…

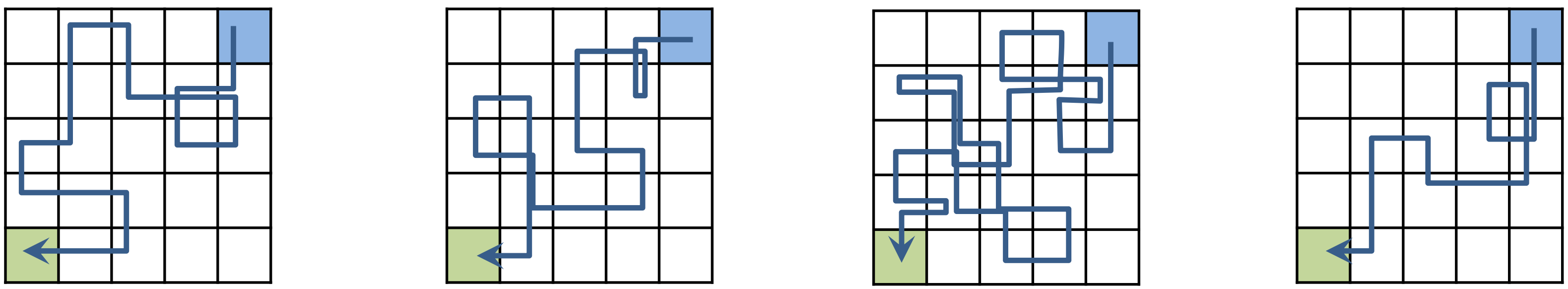
Gridworld Example

- States: Location on a 5x5 grid of cells
- Actions: Move up, down, left or right
- The game starts on the top right corner and ends on the lower left corner. State transitions are deterministic.



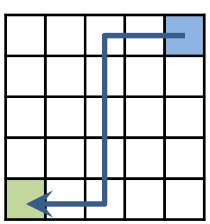
Gridworld: Iteration 1

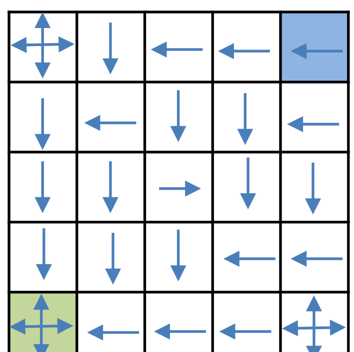
- Initialize with a uniform random policy and collect sample episodes. Use TD-learning to estimate action-values.



- Find the greedy policy

True optimal route:



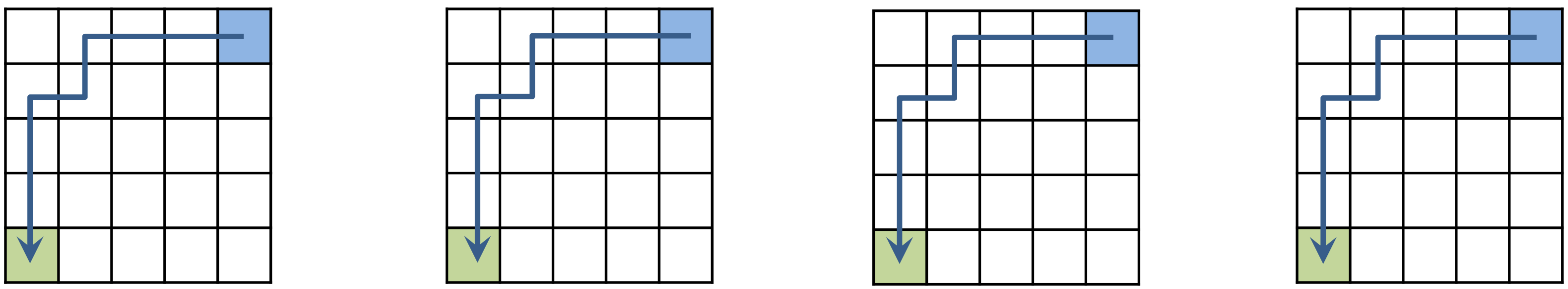


Ignore state-action pairs that haven't been visited when performing argmax.

We're getting close. Nice!

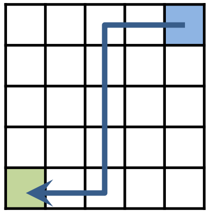
Gridworld: Iteration 2

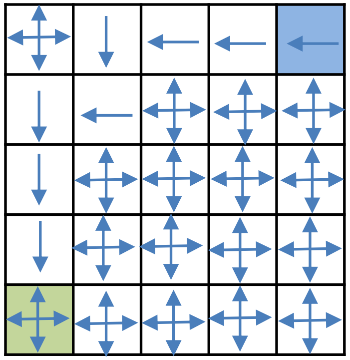
- Use the previous policy and collect sample episodes. Use TD-learning to estimate action-values.



- Find the greedy policy

True optimal route:





Err… what just happened?

Exploration vs. Exploitation

- The original policy iteration algorithm can update the values of *all states* because all the rewards and transition probabilities are known.
- Our model-free control algorithm gathers sample data *by following a policy.*
  - Can't learn about state-action pairs that weren't encountered
  - Will never learn about alternate policies that may turn out to be better
- Solution: Follow our current policy $1 - \epsilon$ of the time
  - But choose a random action ϵ of the time
  - The "epsilon-greedy" policy

GLIE Monte Carlo

- **Greedy in the limit with infinite exploration**
- Start with some random initial policy π
- Produce the episode

$$S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, R_4, \dots, S_T$$

- Process the episode using the following online update rules:

$$N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}\left(G_t - Q(S_t, A_t)\right)$$

- Compute the ϵ-greedy policy for each state

$$\pi(a|s) = \begin{cases} 1 - \epsilon & for\ a = arg\max_{a'} Q(s, a') \\ \dfrac{\epsilon}{N_a - 1} & otherwise \end{cases}$$

- Repeat

GLIE Monte Carlo

- **Greedy in the limit with infinite exploration**
- Start with some random initial policy π
- Produce the episode

$$S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, R_4, \ldots, S_T$$

- Process the episode using the following online update rules:

$$N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t))$$

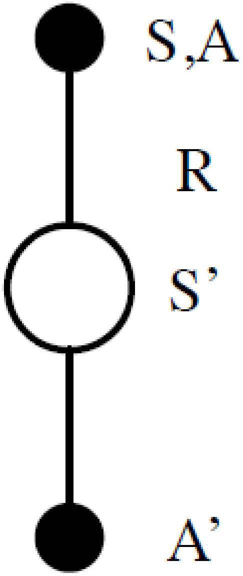
- Compute the ϵ-greedy policy for each state

$$\pi(a|s) = \begin{cases} 1 - \epsilon & for\ a = arg\max_{a'} Q(s, a') \\ \dfrac{\epsilon}{N_a - 1} & otherwise \end{cases}$$

- Repeat

On-line version of GLIE: SARSA

- Replace G_t with an estimate
- TD(1) or TD(λ)
  - Just as in the prediction problem
- **TD(1) → SARSA**



$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$$

SARSA

- Initialize $Q(s, a)$ for all s, a
- Start at initial state S_1
- Select an initial action A_1
- For t = 1.. Terminate
  - Get reward R_t
  - Let system transition to new state S_{t+1}
  - Draw A_{t+1} according to ϵ -greedy policy

$$P(\pi(s) = a) = \begin{cases} 1 - \epsilon \quad for\ a = arg\max_{a'} Q(s, a') \\ \dfrac{\epsilon}{N_a - 1} \ otherwise \end{cases}$$

  - Update

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\Big(R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big)$$

SARSA

- Initialize $Q(s, a)$ for all s, a
- Start at initial state S_1
- Select an initial action A_1
- For t = 1.. Terminate
  - Get reward R_t
  - Let system transition to new state S_{t+1}
  - Draw A_{t+1} according to ϵ -greedy policy

$$P(\pi(s) = \ldots (s, a')$$

  - Update

$$Q(S_t, A_t) = Q \ldots - Q(S_t, A_t))$$

Similar to our proposed algorithm!
Though here, we're making the greedy update to our policy after each action.

This means we no longer need to explicitly store π(a|s); we can infer it using the Q-values.

SARSA(λ)

- Again, the TD(1) estimate can be replaced by a TD(λ) estimate
- Maintain an eligibility trace for every state-action pair:

$$E_0(s,a) = 0$$
$$E_t(s,a) = \lambda\gamma E_{t-1}(s,a) + 1(S_t = s, A_t = a)$$

- Update every state-action pair visited so far

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta_t E_t(s,a)$$

SARSA(λ)

- For all s, a initialize $\mathrm{Q}(s, a)$
- For each episode e
  - For all s, a initialize $E(s, a) = 0$
  - Initialize S_1, A_1
  - For $t = 1 \ldots \mathrm{Termination}$
    - Observe R_{t+1}, S_{t+1}
    - Choose action A_{t+1} using policy obtained from Q
    - $\delta = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$
    - $E(S_t, A_t) \mathrel{+}= 1$
    - For all s, a
      - $Q(s, a) = Q(s, a) + \alpha \delta E(s, a)$
      - $E(s, a) = \lambda \gamma E(s, a)$

Closer look at SARSA

- SARSA: From any state-action (S, A), accept reward (R), transition to next state (S'), choose next action (A')

- Use TD rules to update:

$$\delta = R + \gamma Q(S', A') - Q(S, A)$$

- Problem: what's the best policy to use to choose A'?

Closer look at SARSA

- SARSA: From any state-action (S, A), accept reward (R), transition to next state (S'), choose next action (A')

- Problem: which policy do we use to choose A'
- If we choose the *current judgment of the best action* at S' we will become too greedy
  - Fail to explore the space of possibilities
- If we choose a *sub-optimal* policy to follow, we will never find the best policy
  - E.g. We don't want to be ϵ-greedy at test-time!

Generalization of SARSA

- Pick a random initial policy π.
- Repeatedly create episodes.
  - For each time step t in the current episode:
    - Start at state S_t **(S)**
    - Carry out action $A_t = \pi(S_t)$ **(A)**
    - Get reward R_{t+1} **(R)**
    - Reach state S_{t+1} **(S)**
    - Estimate optimal future action $\hat{a}^*_{S_{t+1}}$ **(A)**
    - Estimate optimal future return $Q\left(S_{t+1}, \hat{a}^*_{S_{t+1}}\right)$
    - Update $Q(\boldsymbol{S}, \boldsymbol{a})$ using R_{t+1} and $Q\left(S_{t+1}, \hat{a}^*_{S_{t+1}}\right)$
    - Update the current policy

Generalization of SARSA

- Pick a random initial policy π.
- Repeatedly create episodes.
  - For each time step t in the current episode:
    - Start at state S_t **(S)**
    - Carry out action $A_t = \pi(S_t)$
    - Get reward R_{t+1}
    - Reach state S_{t+1}
    - l future action $\hat{a}^*_{S_{t+1}}$ **(A)**
    - l future return $Q\left(S_{t+1}, \hat{a}^*_{S_{t+1}}\right)$
    - using R_{t+1} and $Q\left(S_{t+1}, \hat{a}^*_{S_{t+1}}\right)$
    - Update the current policy

← Used to explore the environment
Are there any reasons to choose A_t to be the optimal action?

Used to estimate optimal return →
Are there any reasons to make $\hat{a}^*_{S_{t+1}}$ the same as A_{t+1}?

On-policy vs. Off-policy

- It’s possible learn to what the best actions should be, even if we don’t always follow those actions.
  - E.g. learning by observation
- We learn by following a more exploratory policy
- In the process, we look for a hypothetical optimal policy…the one that we’d want to follow at test-time.

$$S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, R_4, \ldots, S_T$$

$$\hat{a}^*_{S_2}? \qquad \hat{a}^*_{S_3}?$$

- The actions we actually follow to get samples (e.g. A_t) are not the same as our best estimates of the optimal actions (e.g. $\hat{a}^*_{S_t}$)
  - Hence this is an “off-policy” method

Solution: Off-policy learning

- Use data to improve your choice of actions, but follow different ("off-policy") actions to collect data.

$$S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, R_4, \ldots, S_T$$

$$\hat{a}^*_{S_2}? \qquad \hat{a}^*_{S_3}?$$

- E.g. Use $\hat{a}^*_{S_{t+1}} = \text{argmax}_a\big(Q(S_{t+1}, a)\big)$
- But, actually follow the *epsilon-greedy* policy
  - The hypothetical action is better than the one you actually took, but you still explore (non-greedy)
- This is the basis for the most popular RL algorithm, Q-Learning

Q-Learning (TD-1)

- Pick initial values for Q.
- Repeatedly create episodes.
  - For each time step t in the current episode:
    - Start at state S_t
    - Carry out action $A_t = \pi_{\epsilon-\mathrm{greedy}}(S_t)$
    - Get reward R_{t+1}
    - Reach state S_{t+1}
    - Estimate optimal future action
      $$\hat{a}^*_{S_{t+1}} = \mathrm{argmax}_a\big(Q(S_{t+1}, a)\big)$$
    - Estimate optimal future return $Q\left(S_{t+1}, \hat{a}^*_{S_{t+1}}\right)$
    - Update $Q(S_t, A_t) =$
      $Q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma Q\left(S_{t+1}, \hat{a}^*_{S_{t+1}}\right) - Q(S_t, A_t)\right)$

The Q-learning algorithm generalizes to TD(λ) too

Off-policy vs. On-policy

- Optimal greedy policy:

$$\pi(a|s) = \begin{cases} 1 \quad for\ a = arg\max_{a\prime} Q(s, a') \\ 0\ otherwise \end{cases}$$

- Exploration policy

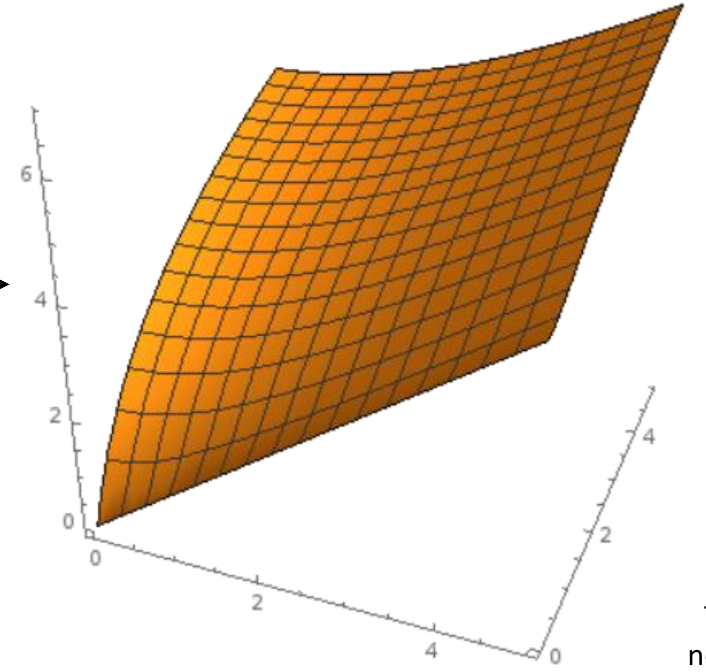
$$\pi(a|s) = \begin{cases} 1 - \epsilon \quad for\ a = arg\max_{a\prime} Q(s, a') \\ \dfrac{\epsilon}{N_a - 1}\ otherwise \end{cases}$$

- Ideally ϵ should decrease with time

Continuous State Space

- Tabular methods won't work if our state space is infinite or huge
- E.g. position on a [0, 5] x [0, 5] square, instead of a 5x5 grid.

4.4	4.5	4.8	5.3	5.9
3.9	4.0	4.4	4.9	5.6
3.2	3.4	3.8	4.0	5.1
2.2	2.4	3.0	3.7	4.6
0	1.0	2.0	3.0	4.0



The graphs show the negative value function

Parameterized Functions

- Instead of using a table of Q-values, we use a parametrized function

$$Q(s,a) = f(s,a|\theta)$$

- Instead of writing values to the table, we fit the parameters to minimize the prediction error of the "Q function"

$$\theta_{k+1} \leftarrow \theta_k - \eta \nabla_\theta \left(Div\left(f(s,a|\theta_k), Q_{\mathrm{s,a}}^{\mathrm{new}}\right)\right)$$

Parameterized Functions

- Instead of using a table of Q-values, we use a parametrized function

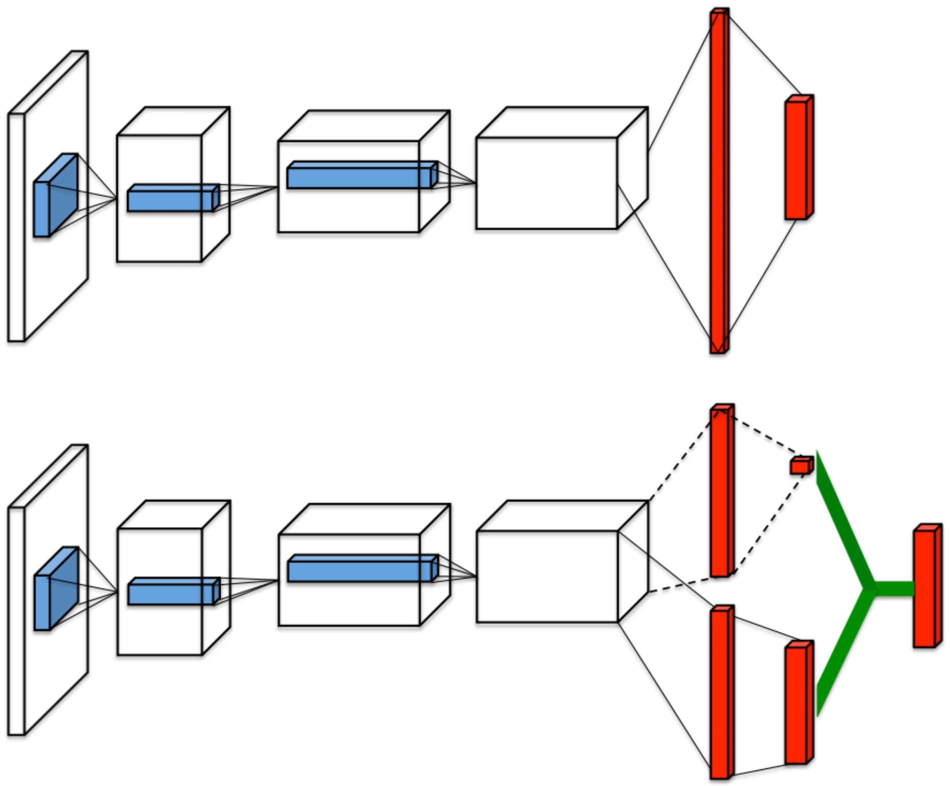
$$Q(s,a) = f(s,a|\theta)$$

- This can be a simple linear function…

$$f(\mathbf{s},\mathbf{a}|\boldsymbol{\theta}) = \boldsymbol{\theta}^T[\mathbf{s};\mathbf{a}]$$

Parameterized Functions

- Or a massive convolutional network…



Parameterized Functions

- Fundamental issue: limited capacity
  - A table of Q values will never forget any values that you write into it
  - But, modifying the parameters of a Q-function will affect its *overall* behavior
    - Fitting the parameters to match one (s, a) pair can change the function's output at (s', a').
    - If we don't visit (s', a') for a long time, the function's output can diverge considerably from the values previously stored there.

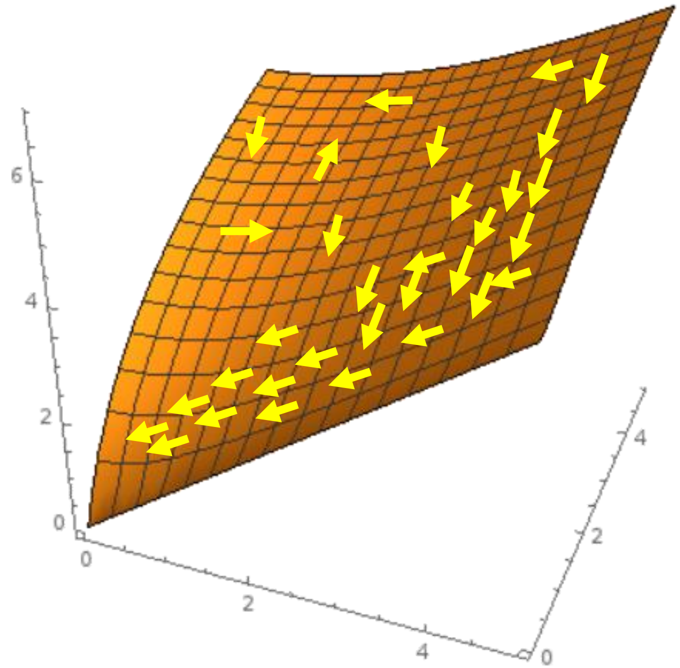
Full Capacity

- Q-learning works well with Q-tables
  - The sample data is going to be heavily biased toward optimal actions $\left(s, \pi^*(s)\right)$, or close approximations thereof.
  - But still, ϵ-greedy policy will ensure that we will visit all state-action pairs arbitrarily many times if we explore long enough.
  - The action-value for uncommon inputs will still converge, just more slowly.

Limited Capacity

- The Q-function will fit more closely to more common inputs, even at the expense of lower accuracy for less common inputs.

- Just exploring the whole state-action space isn't enough. We also need to visit those states often enough so the function computes accurate q-values before they are "forgotten".



Action-replay

- The raw data obtained from Q-learning is:
  - Highly correlated: current data can look very different from data from several episodes ago if the policy changed significantly.
  - Very unevenly distributed: only ϵ probability of choosing suboptimal actions.
- Instead, create a *replay buffer* holding past experiences, so we can train the Q-function using this data.

Action-replay

- Pseudocode:

```
for B steps:
    (R_{t+1}, S_{t+1}) = make_action(A_t)
    replay_buffer.add(S_t, A_t, R_{t+1}, S_{t+1})

TD_update(replay_buffer.sample(B),
          q_function)
```

- We have control over how the experiences are added, sampled and deleted.
  - Can make the samples look independent
  - Can emphasize old experiences more
  - Can change frequency depending on accuracy

Action-replay

- What is the best way to sample?
  - On the one hand, our function has limited capacity, so we should let it optimize more strongly for the common case
  - On the other hand, our function needs explore uncommon examples just enough to compute accurate action-values, so it can avoid missing out on better policies
- A trade-off!

Moving target

- We already have moving targets in online SARSA and Q-learning, since we're using the action-values to compute the updates to the action-values.
- The problem is much worse with Q-functions though. Optimizing the function at one state-action pair affects *all other state-action pairs*.
  - The target value is fluctuating at all inputs in the function's domain, and all updates will shift the target value across the entire domain.

Separate target function

- Solution: Create two copies of the Q-function.
  - The “target copy” is frozen and used to compute the target Q-values.
  - The “learner copy” will be trained on the targets.

$$Q_{\text{learner}}(S_t, A_t) \leftarrow_{\text{fit}} R_{t+1} + \gamma \max_{\text{a}} \left(Q_{\text{target}}(S_{t+1}, a) \right)$$

- Just need to periodically update the target copy to match the learner copy.

Deep Q Network

- Create a neural network function which takes in a state and outputs the Q values for all possible actions

$$DQN(s) = [Q(s, a) \mid a \in A]$$

  - Note: This is equivalent to a function that takes in both the state and the action to produce one Q value. But this design lets us iterate over all possible actions more efficiently.

- Create a replay buffer and a frozen "target network"
- Perform Q-learning as usual, except:
  - The Q-value updates use samples from the reply buffer
  - The new Q-value is computed using the target network
  - The target network is periodically updated

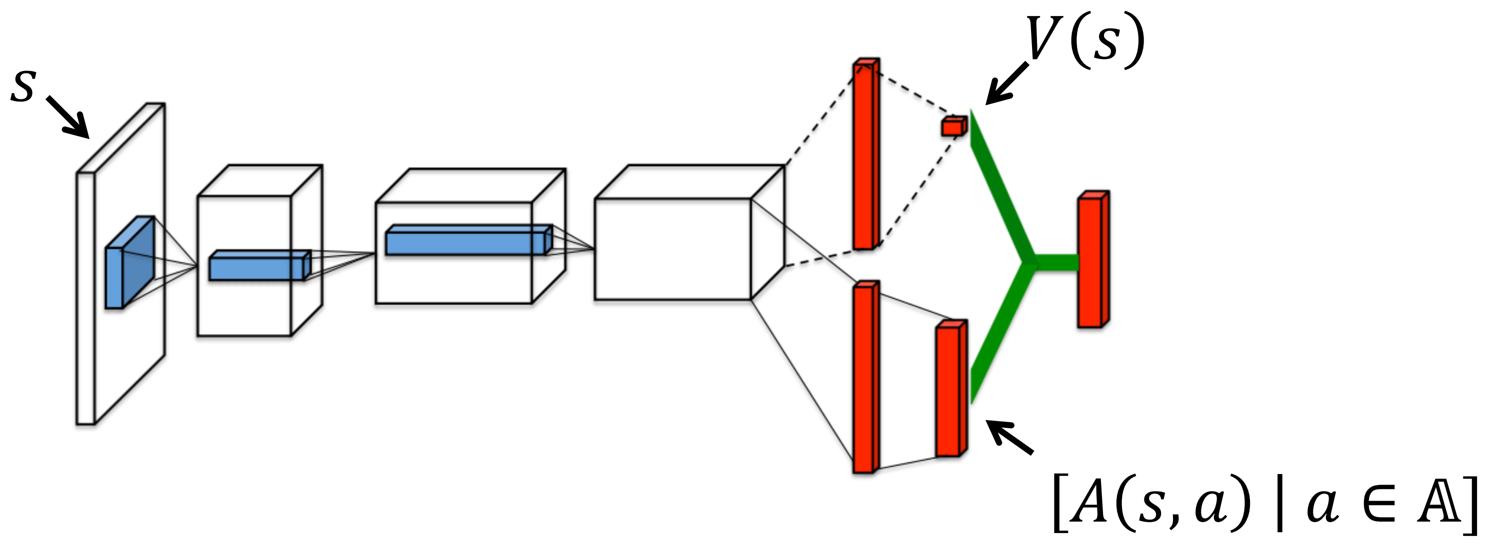
Performance

	Breakout	R. Raid	Enduro	Sequest	S. Invaders
DQN	**316.8**	**7446.6**	**1006.3**	**2894.4**	**1088.9**
Naive DQN	3.2	1453.0	29.1	275.8	302.0
Linear	3.0	2346.9	62.0	656.9	301.3

Replay	○	○	×	×
Target	○	×	○	×
Breakout	**316.8**	240.7	10.2	3.2
River Raid	**7446.6**	4102.8	2867.7	1453.0
Seaquest	**2894.4**	822.6	1003.0	275.8
Space Invaders	**1088.9**	826.3	373.2	302.0

Other optimizations

- Dualing DQN:
  - Decompose $Q(s,a) = f\big(V(s), A(s,a)\big)$
    - V is the value function, and A is known as the advantage function.
  - Easier to learn since you can get good estimates with $A(s,a) = \text{some constant } A(a)$ and $f(x,y) = x + y$



Other optimizations

- Problem: $\max_a \left(Q_{DQN}(S_{t+1}, a) \right)$ can be biased toward large values if Q_{DQN} is noisy.

- Solution: Double DQN
  - Train two DQN's with the same parameters, but different initialization.
  - Instead of $\max_a \left(Q_{DQN1}(S_{t+1}, a) \right)$, do:
    $$Q_{DQN2}\left(S_{t+1}, \operatorname{argmax}_a \left(Q_{DQN1}(S_{t+1}, a) \right) \right)$$
  - Similar story for $\max_a \left(Q_{DQN2}(S_{t+1}, a) \right)$

- Alternative: $Q_{\text{learner}}\left(\operatorname{argmax}_a \left(Q_{\text{target}}(S_{t+1}, a) \right) \right)$

Direct Policy Estimation

- It's also possible to make a deep neural network that directly produces a distribution over actions given a state
  - Also known as a policy network, or the policy gradient method
  - Useful when the action space is also large or continuous
- This approach is explained in more depth in the recitation

Summary

- Model-free control
- Exploration vs Exploitation
- Off-policy vs On-policy learning
- Q-learning
- Parameterized Functions
- Action-replay
- Target functions
- Deep Q Networks