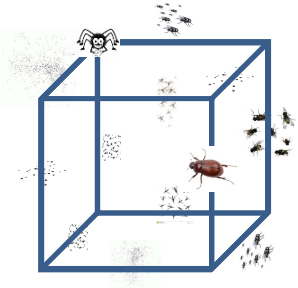
Deep Reinforcement Learning

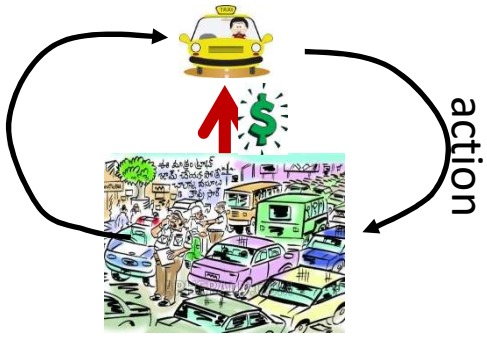
Spring 2019

RL!

Story so far

- Typical problem in life:
  - Agent is in some state
  - Agent takes an action
    - Chosen according to some policy
  - Agent gets a reward
  - Environment changes state in response to action





- Objective: Choose policy to maximize long-term return
  - Discounted sum of rewards from start to end

Approach: Define values

- Typical sequence

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

- Value of being any state (expected return) is the *expected* future return if you are at that state

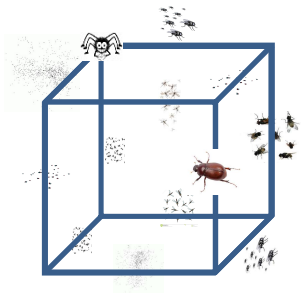
$$v_\pi(s) = E[G_t | S_t = s]$$
$$= E[R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T | S_t = s]$$

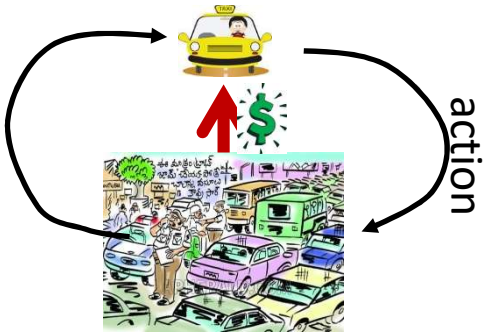
- Value of taking an action at any state

$$q_\pi(s, a) = E[G_t | S_t = s, A_t = a]$$
$$= E[R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T | S_t = s, , A_t = a]$$

- These are functions of the policy
- *Objective: Choose policy to maximize the return*
  - The value of every state under the policy

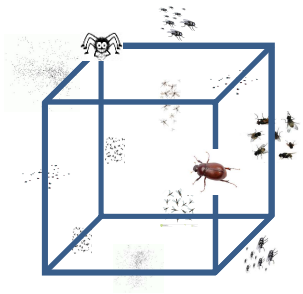
Different settings

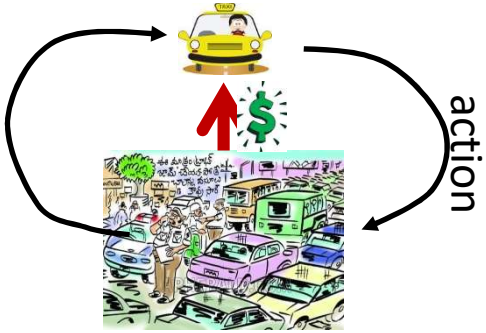




	Given policy, find the values of all states (or state-action pairs): PREDICTION	Find the optimal policy: CONTROL
Have model of how the environment will respond to an action at any state	MODEL BASED PLANNING	
Do not know how the environment will respond to an action at any state	MODEL FREE REINFORCEMENT LEARNING	

Different settings





	Given policy, find the values of all states (or state-action pairs): PREDICTION	Find the optimal policy: CONTROL
Have model of how the environment will respond to an action at any state	MODEL BASED PLANNING (MARKOV DECISION PROCESS)	
Do not know how the environment will respond to an action at any state	MODEL FREE REINFORCEMENT LEARNING	

Bellman *Expectation* Equations

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(R_s^a + \gamma \sum_{s'} P_{s,s'}^a v_\pi(s') \right)$$

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s'} P_{s,s'}^a \sum_{a' \in \mathcal{A}} \pi(a|s') q_\pi(s',a')$$

- *For given policy* how to compute
  - The value of being in any state
  - The value of being in any state and taking a particular action

Bellman *Optimality* Equations

$$v_*(s) = \max_a \; R_s^a + \gamma \sum_{s'} P_{s,s'}^a v_*(s')$$

$$q_*(s,a) = R_s^a + \gamma \sum_{s'} P_{s,s'}^a \max_{a'} q_*(s',a')$$

- How to compute
  - The value of being in any state
  - The value of being in any state and taking a particular action

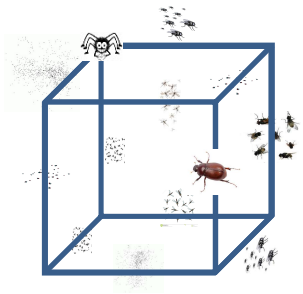
under the *optimal* policy

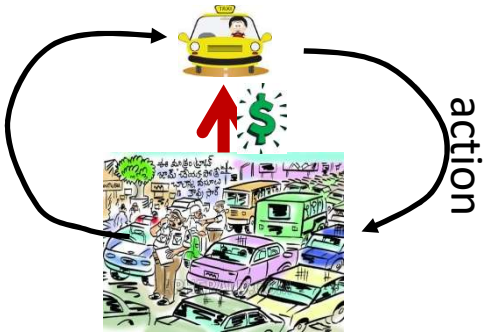
Solving an MDP

- Prediction: *Given a policy find value functions*
  - Using Bellman expectation equations

- Control: *Find the optimal policy*
  - Using policy iteration
    - Directly find optimal policy
  - Using value iteration
    - Find optimal values
      - Bellman optimality equation
    - Find policy from optimal values

Different settings

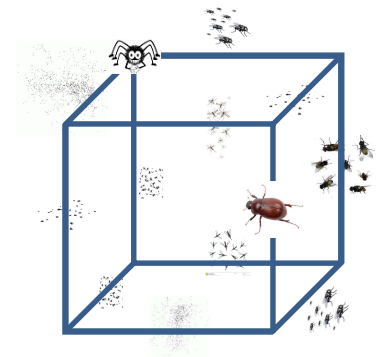




	Given policy, find the values of all states (or state-action pairs): PREDICTION	Find the optimal policy: CONTROL
Have model of how the environment will respond to an action at any state	MODEL BASED PLANNING	
Do not know how the environment will respond to an action at any state	MODEL FREE REINFORCEMENT LEARNING	

Reinforcement Learning

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(R_s^a + \gamma \sum_{s'} P_{s,s'}^a v_\pi(s') \right)$$



$$q_\pi(s,a) = R_s^a + \gamma \sum_{s'} P_{s,s'}^a \sum_{a' \in \mathcal{A}} \pi(a|s') q_\pi(s',a')$$

- In real-life problems the transition probabilities wont be known
  - No prior knowledge of how the environment will respond to an action
- Must still find optimal policy

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*

Solution: Actually run through the system

- Record many episodes of the kind
  - $episode(1) = S_{11}, A_{11}, R_{12}, S_{12}, A_{12}, R_{13}, \dots, S_{1T_1}$
  - $episode(2) = S_{21}, A_{21}, R_{22}, S_{22}, A_{22}, R_{23}, \dots, S_{2T_2}$
  - …
- Use these to estimate values $v_\pi(s)$ or action values $q_\pi(s, a)$ of states

Recap: Methods

- *Monte-Carlo* Learning

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

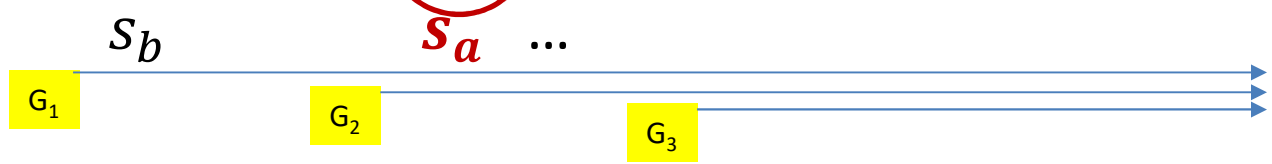
Recap: Methods

- *Monte-Carlo* Learning

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

Recap: Monte Carlo

- To estimate the value of any state, identify the instances of that state in the episodes:
  - $S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, R_4, \ldots$



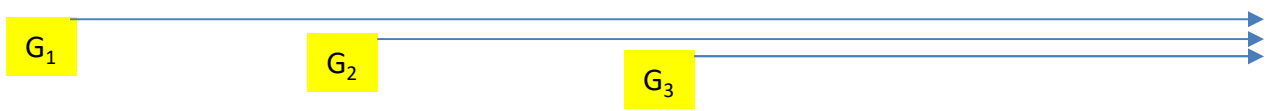
- Compute the average return from those instances

$$v_\pi(s_a) = avg(G_1, G_3, \ldots)$$

Monte Carlo: Estimating the Action Value function

- To estimate the value of any state-action pair, identify the instances of that state-action pair in the episodes:
  - $S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, R_4, \ldots$

$\boldsymbol{s_a}$ $\boldsymbol{a_x}$ s_b a_y $\boldsymbol{s_a}$ a_y …



- Compute the average return from those instances

$$q_\pi(s_a, a_x) = avg(G_1, \ldots)$$

Recap: Methods

- *Monte-Carlo* Learning

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

Concept behind TD learning

$$v_\pi(s) = E_\pi[r_s^a + \gamma E_{s'}[v_\pi(s')|a] \quad |s]$$

$$q_\pi(s,a) = E_{s'}\left[r_s^a + \gamma E_{s',a'}[q_\pi(s',a')] \quad |s,a\right]$$

- If we had the true value or action value functions, the above equations would be valid
- We can even write

$$v_\pi(s) = v_\pi(s) + (E_\pi[r_s^a + \gamma E_{s'}[v_\pi(s')|a] \quad |s] \; - v_\pi(s))$$

$$q_\pi(s,a) = q_\pi(s,a) + (E_{s'}\left[r_s^a + \gamma E_{s',a'}[q_\pi(s',a')] \quad |s,a\right] - q_\pi(s,a))$$

Concept behind TD learning

- If we had the true value or action value functions, the above equations would be valid

$$v_\pi(s) = v_\pi(s) + (E_\pi[r_s^a + \gamma E_{s'}[v_\pi(s')|a] \quad |s] \; - v_\pi(s))$$

$$q_\pi(s,a) = q_\pi(s,a) + (E_{s'}\left[r_s^a + \gamma E_{s',a'}[q_\pi(s',a')] \quad |s,a\right] - q_\pi(s,a))$$

- In practice we wont have the true value functions
- So we use the iterative update

$$v_\pi^{k+1}(s) = v_\pi^k(s) + \alpha(E_\pi[r_s^a + \gamma E_{s'}[v_\pi(s')|a] \quad |s] - v_\pi^k(s))$$

$$q_\pi^{k+1}(s,a) = q_\pi^k(s,a) + \alpha(E_{s'}\left[r_s^a + \gamma E_{s',a'}[q_\pi(s',a')] \quad |s,a\right] - q_\pi^k(s,a))$$

- It will converge to the true value for $\alpha \leq 1$

Concept behind TD learning

- Problem with this estimator:

$$v_\pi^{k+1}(s) = v_\pi^k(s) + \alpha(E_\pi[r_s^a + \gamma E_{s'}[v_\pi(s')|a] \quad |s] \; - v_\pi^k(s))$$

$$q_\pi^{k+1}(s,a) = q_\pi^k(s,a) + \alpha(E_{s'}\left[r_s^a + \gamma E_{s',a'}[q_\pi(s',a')] \quad |s,a\right] - q_\pi^k(s,a))$$

  - true values $v_\pi(s')$ and $q_\pi(s',a')$ are unknown
  - Transition probabilities are unknown, so expectations cannot be computed

- Instead we *bootstrap* with the empirical updates

$$v_\pi^{k+1}(s) = v_\pi^k(s) + \alpha(r_s^a + \gamma v_\pi^k(s') - v_\pi^k(s))$$

$$q_\pi^{k+1}(s,a) = q_\pi^k(s,a) + \alpha(r_s^a + \gamma q_\pi^k(s',a') - q_\pi^k(s,a))$$

Concept behind TD learning

- TD Estimator:

$$v_\pi^{k+1}(s) = v_\pi^k(s) + \alpha(r_s^a + \gamma v_\pi^k(s') - v_\pi^k(s))$$

$$q_\pi^{k+1}(s,a) = q_\pi^k(s,a) + \alpha(r_s^a + \gamma q_\pi^k(s',a') - q_\pi^k(s,a))$$

- Generally written as (only shown for action value estimator)

$$\delta = r_s^a + \gamma q_\pi^k(s',a') - q_\pi^k(s,a)$$

$$q_\pi^{k+1}(s,a) = q_\pi^k(s,a) + \alpha\delta$$

- δ is generally referred to as the TD error

Recap: TD(1)

- An "episode" is a run:
  $S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_3, R_4, \ldots$
- 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

TD(1) with action-values

- For all s, a, initialize:

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

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

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

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

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

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

- Simply knowing the value function is insufficient to find the optimal policy
- We must compute the optimal *action value* functions to find the optimal policy
  - 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 $t = 1 \ldots T_e$

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

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:

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

Problem of optimal control

- From a series of episodes of the kind:

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

G_1 G_2 G_3

  - Can also find *empirical returns ate each time* for the episode

- 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

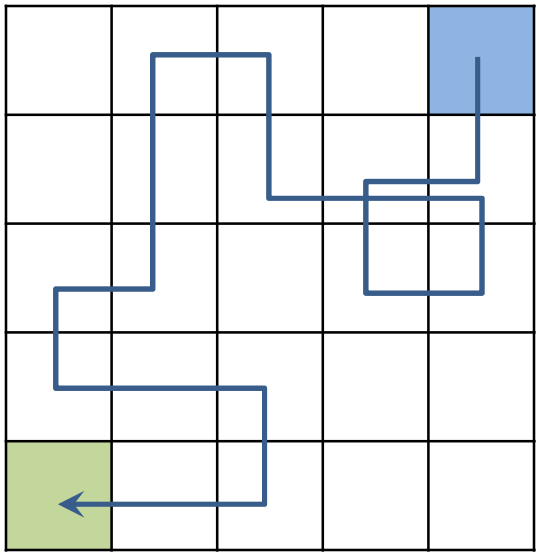
- Our proposed algorithm:
  - Start with any policy $\pi^{(0)}$
  - Iterate ($k = 0$ … convergence)
    - Estimate the action-value function $q_{\pi^{(k)}}(s, a)$ using TD-learning
    - Find the greedy policy

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

- Let's see if this works…

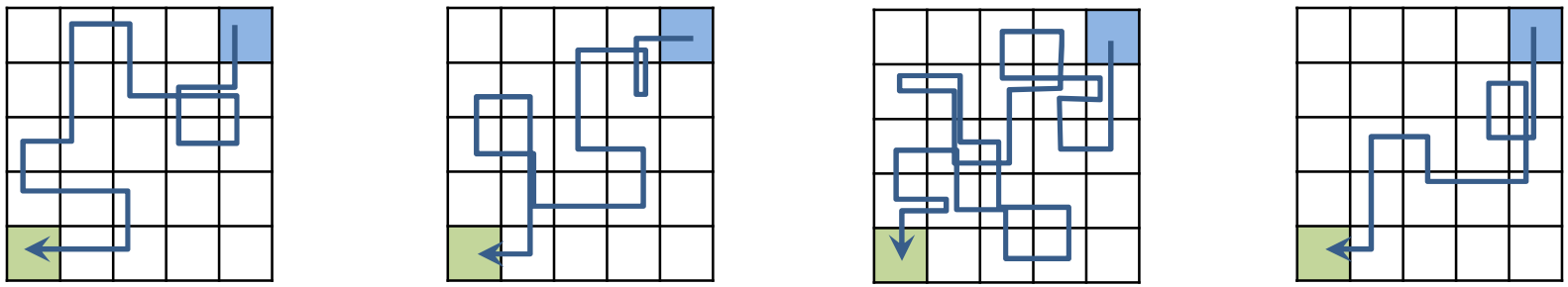
Gridworld Example

- 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.



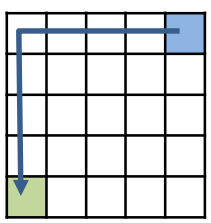
Gridworld: Iteration 1

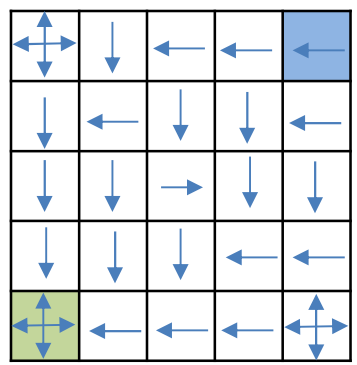
- Initialize with a uniform random policy and collect sample episodes. Use TD-learning to estimate action-values.



- Find the greedy policy

True optimal route:



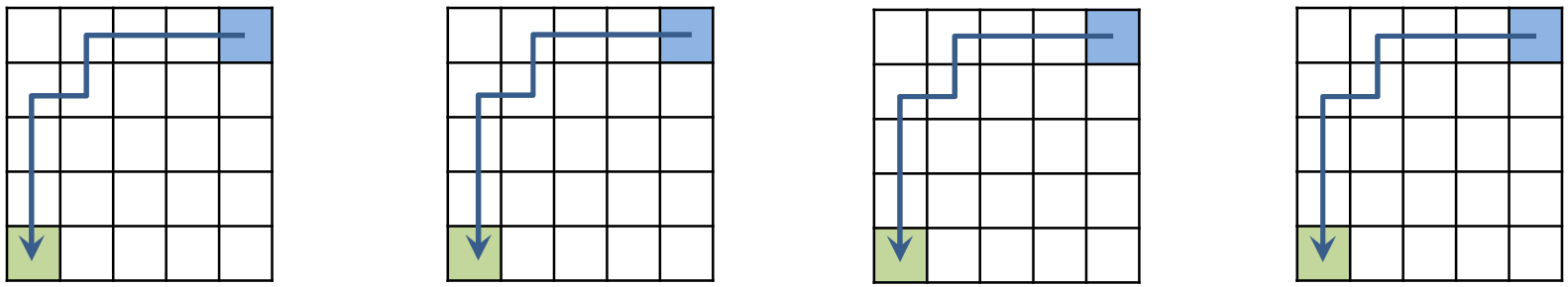


Ignore state-action pairs that haven't been visited when performing argmax.

We're getting close. Nice!

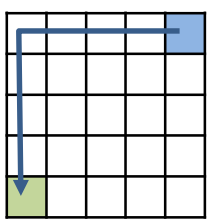
Gridworld: Iteration 2

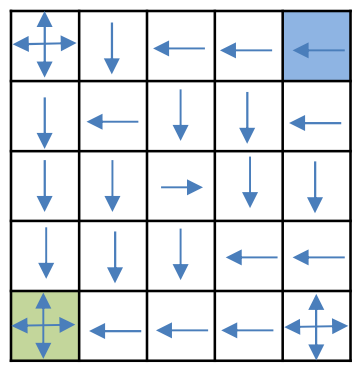
- Use the previous policy and collect sample episodes. Use TD-learning to estimate action-values.



- 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:

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

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

- 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

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:

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

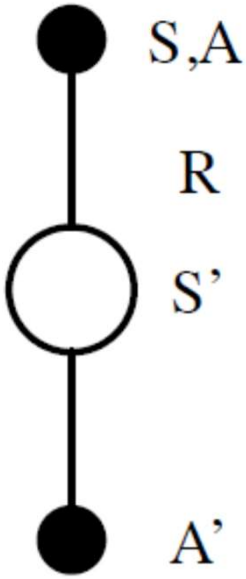
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t))$$

- 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



- ***Bootstrap:*** 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

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

  - 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

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

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.

  - Update

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

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}

← **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}?

future action $\hat{a}^*_{S_{t+1}}$ (**A**)

future return $Q\left(S_{t+1}, \hat{a}^*_{S_{t+1}}\right)$

Update $Q(S, a)$ using R_{t+1} and $Q\left(S_{t+1}, \hat{a}^*_{S_{t+1}}\right)$

    - Update the current policy

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.

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

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

- 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.

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

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

- E.g. Use $\hat{a}^*_{S_{t+1}} = \mathrm{argmax}_a\big(Q(S_{t+1}, a)\big)$
- 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 $A_t = \pi_{\epsilon-\text{greedy}}(S_t)$
    - Get reward R_{t+1}
    - Reach state S_{t+1}
    - Estimate optimal future action
      $$\hat{a}^*_{S_{t+1}} = \text{argmax}_a\big(Q(S_{t+1}, a)\big)$$
    - Estimate optimal future return $Q\big(S_{t+1}, \hat{a}^*_{S_{t+1}}\big)$
    - Update $Q(S_t, A_t) =$
      $Q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma Q\big(S_{t+1}, \hat{a}^*_{S_{t+1}}\big) - Q(S_t, A_t)\right)$

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

Off-policy vs. On-policy

- Optimal greedy policy:

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

- Exploration policy

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

- Ideally ϵ should decrease with time

Scaling up the problem..

- We've assumed a discrete set of states
- And a discrete set of actions

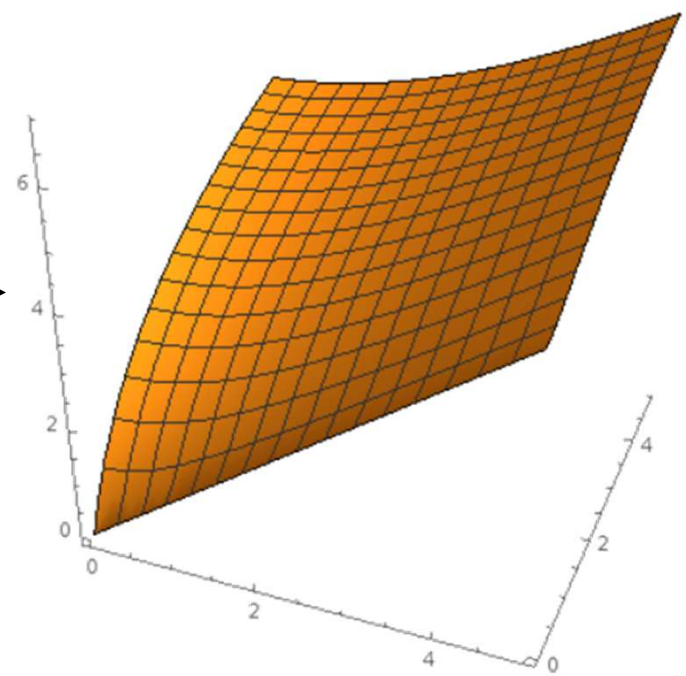
- Value functions can be stored as a table
  - One entry per state
- Action value functions can be stored as a table
  - One entry per state-action combination
- Policy can be stored as a table
  - One probability entry per state-action combination

- None of this is feasible if
  - The state space grows too large (e.g. chess)
  - Or the states are continuous valued

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"

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

Parameterized Functions

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

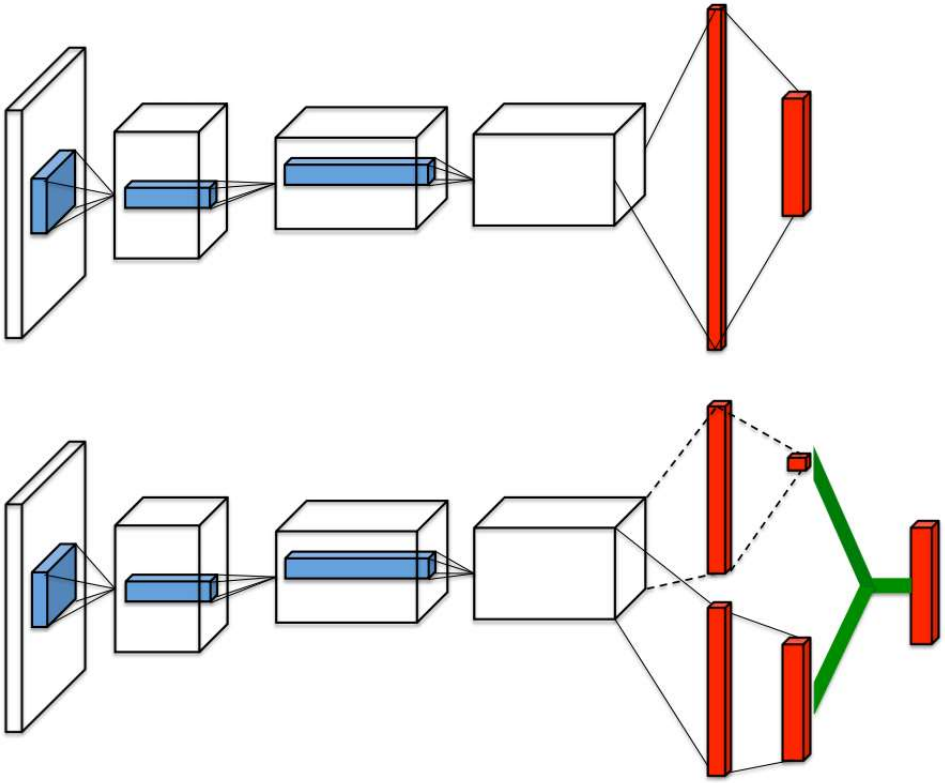
$$Q(s, a) = f(s, a|\theta)$$

- 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…



Target Q

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

→What is $Q_{\mathrm{s,a}}^{\mathrm{target}}$?

As in TD, use bootstrapping for the target :

$$Q_{\mathrm{s,a}}^{\mathrm{target}} = \mathcal{R}_s^a + \gamma \operatorname*{argmax}_{a' \in \mathcal{A}} f(s', a'|\theta_k)$$

And Div can be L2 distance

DQN (v0)

- Initialize θ_0
- For each episode e
  - Initialize S_1, A_1
  - For $t = 1 \ldots Termination$
    - Choose action A_t using ε –greedy policy obtained from θ_t
    - Observe R_{t+1}, S_{t+1}
    - Choose action $A_{tar} \quad = argmax_a f(S_{t+1}, a|\theta_t)$
    - $Q_{target} = R_{t+1} + \gamma Q\left(S_{t+1}, A_{tar} \quad \right)$
    - $\theta_{t+1} = \theta_t - \eta \nabla_\theta \left\| Q_{target} - f(S_t, A_t|\theta_t) \right\|_2^2$

Deep Q Network

- Note : $\nabla_\theta \left\| Q_{target} - f(S_t, A_t | \theta_t) \right\|_2^2$ does **not** consider Q_{target} as depending of θ_t (although it does). Therefore this is **semi-gradient descent**.
- If your function is a neural network, and the action set is finite of size $|A|$, then you can use a $|A|$-labels classification network that associates the probabilities of each action to an input

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.

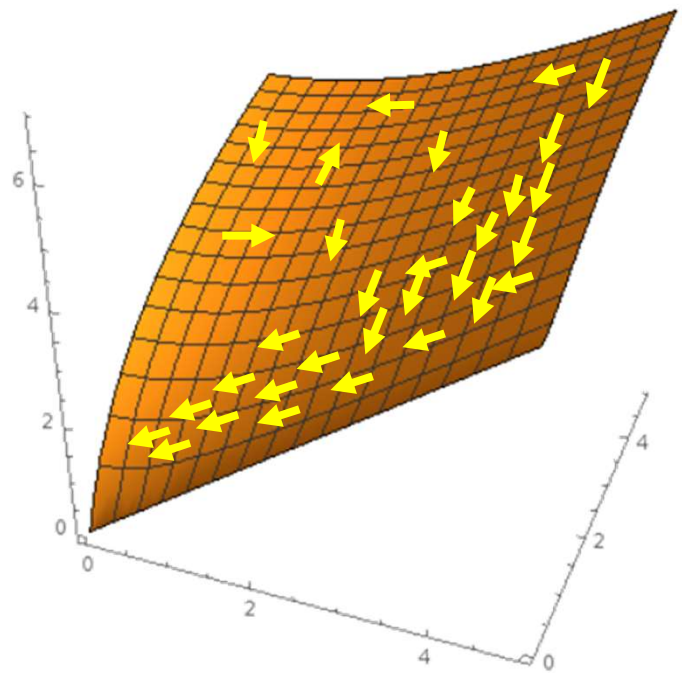
Tables have full capacity

- 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 of $f(s, a|\theta)$

- 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!

DQN (with Action-replay)

- Initialize θ_0
- Initialize buffer with some random episodes
- For each episode e
  - Initialize S_1, A_1
  - For $t = 1 \ldots Termination$
    - Choose action A_t using ε –greedy policy obtained from θ_t
    - Observe R_{t+1}, S_{t+1}
    - Add $S_t, A_t, R_{t+1}, S_{t+1}$ to the buffer
    - Sample from the buffer a batch of tuples S, A, R, S_{new}
    - Choose $A_{target} = argmax_a f(S_{\text{new}}, a|\theta_t)$
    - $Q_{target} = R + \gamma Q\left(S_{new}, A_{targe}\ \right)$
    - $\theta_{t+1} = \theta_t - \eta \nabla_\theta \left\| Q_{target} - f(S, A|\theta_t) \right\|_2^2$

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.

Frozen 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.

Fixed target DQN

- Initialize $\theta_0, \theta^* = \theta_0$
- Initialize buffer with some random episodes
- For each episode e
  - Initialize S_1, A_1
    - For $t = 1 \ldots Termination$
    - If $t\%k = 0$ then update $\theta^* = \theta_t$
    - Choose action A_t using ε –greedy policy obtained from θ_t
    - Observe R_{t+1}, S_{t+1}
    - Add $S_t, A_t, R_{t+1}, S_{t+1}$ to the buffer
    - Sample from the buffer a batch of tuples S, A, R, S_{new}
    - Choose $A_{ta} \quad = argmax_a f(S_{\text{new}}, a|\theta^*)$
    - $Q_{target} = R + \gamma f(S_{\text{new}}, A_{target}|\theta^*)$
    - $\theta_{t+1} = \theta_t - \eta \nabla_\theta \left\| Q_{target} - f(S, A|\theta_t) \right\|_2^2$

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

Overestimation of Q-values

- Choose $A_{target} = argmax_a f(S_{\text{new}}, a|\theta^*)$
- $Q_{target} = R + \gamma f(S_{\text{new}}, A_{target}|\theta^*)$

- But what if this action is not optimal ?

- If, in DQN (fixed target or not) in early training non-optimal actions are attributed higher Q-values than the optimal action…
  - Learning is difficult, due to **bias on chosen actions**

Double Q networks

- Solution : Create two Q-functions.
  - The “DQN network” compute the target action
  - The “target network” is used to compute the Q-value of the target action.
  - The “DQN network” is trained on the targets.

$$Q_{\mathrm{DQN}}(S_t, A_t) \leftarrow_{\mathrm{fit}} R_{t+1} + \gamma \left(Q_{\mathrm{target}} \left(S_{t+1}, argmax_a Q_{\mathrm{DQN}}(S_{t+1}, a) \right) \right)$$

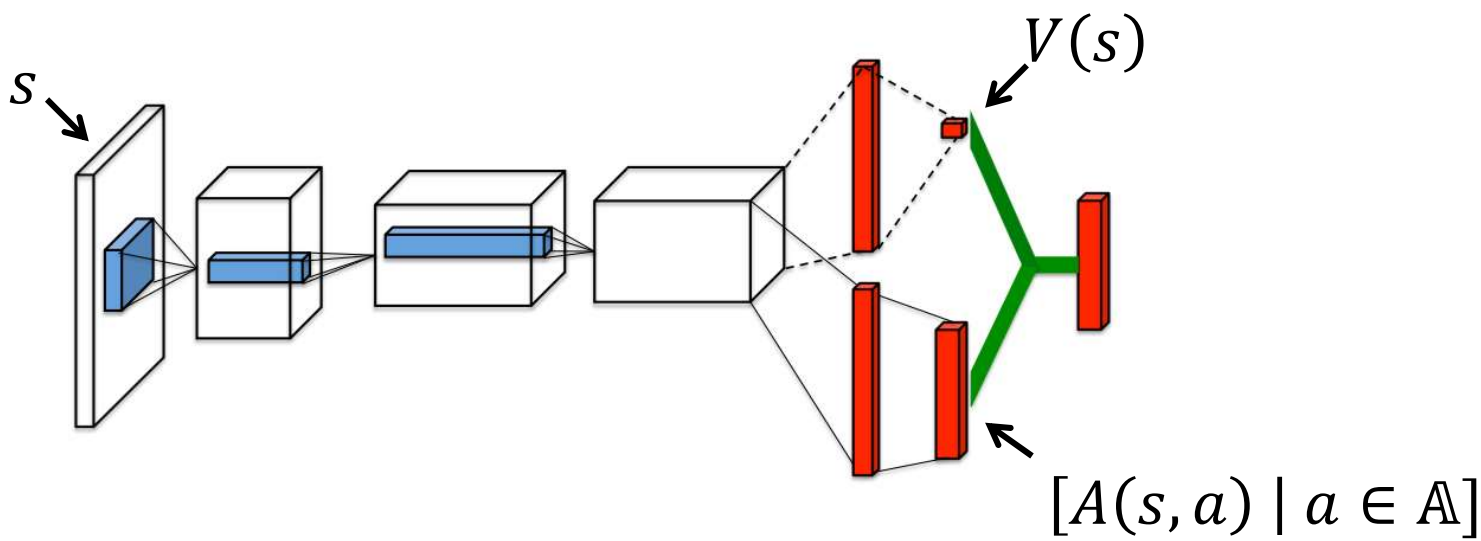
- Each network can play the role of the DQN or target network : **chosen randomly at each step**
- Action selections are epsilon-greedy with respect to the sum of both networks

Double DQN

- Initialize θ_0^1, θ_0^2
- Initialize buffer with some random episodes
- For each episode e
  - Initialize S_1, A_1
    - For $t = 1 \ldots Termination$
    - Choose action A_t using ε –greedy policy obtained from θ_t^1 and θ_t^2 , Observe R_{t+1}, S_{t+1}
    - Add $S_t, A_t, R_{t+1}, S_{t+1}$ to the buffer
    - Sample from the buffer a batch of tuples S, A, R, S_{new}
    - Assign randomly $\theta_0^1, \theta_0^2 \rightarrow \theta_0^{DQN}, \theta_0^{target}$
    - Choose $A_{target} = argmax_a f(S_{\text{ne}} \ , a|\theta_t^{DQN})$
    - $Q_{targ} \ = R + \gamma f(S_{\text{ne}} \ , A_{targe} \ |\theta_t^{target})$
    - $\theta_{t+1}^{DQN} = \theta_t^{DQN} - \eta^{DQN} \nabla_{\theta^{DQN}} \left\| Q_{target} - f\left(S, A \middle| \theta_t^{DQN}\right) \right\|_2^2$

Other Q-learning 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 $f(x,y) = x + y$



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**

Policy Network

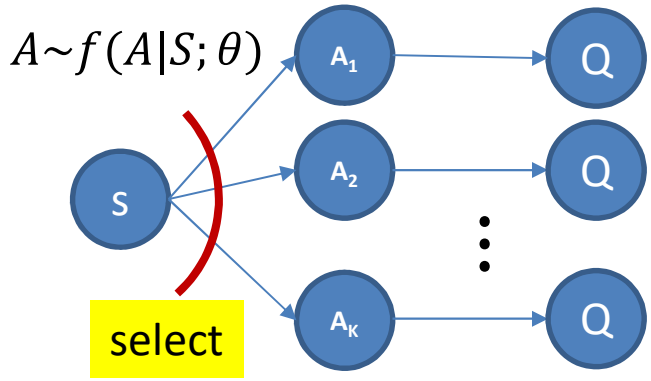
- Train a neural network to prescribe actions at each state:

$$f(A|S;\theta)$$

  - Input is S, output is probability distribution over A
  - Could be deterministic
- **Problem** : how to train such a network ?
- No golden truth
  - Unlike *value* functions, where there is a *target* value for the value at each state
    - Against which we can compute a loss

Maximizing return

- ***Learn policy to maximize expected return!***



- **Problem:** For discrete action space, the return is not differentiable with respect to policy function parameters
  - Selection is not a differentiable operation

Solution

- Recast differentiation as an *expectation* operation
  - Can now be approximated by sampling
  - Policy gradient method
- Compute expected returns using an action-value function approximator
  - Actor-critic methods

Solution

- Recast differentiation as an *expectation* operation
  - Can now be approximated by sampling
  - Policy gradient method
- Compute expected returns using an action-value function approximator
  - Actor-critic methods

How to choose policy

- In any run starting at a state S we get
  - $(S_1 = S,)\ A_1\ R_2\ S_2\ A_2\ R_3\ S_3\ A_3\ R_4\ \ldots$
- The trajectory T associated with the run is
  - $T\ =\ S\ A_1\ S_2\ A_2\ S_3\ A_3\ \ldots.$
- The total return over the run (at t=1) is
  - $G = R_2 + \gamma R_3 + \gamma^2 R_4 \ldots.$
- The choice of θ in $f(A|S;\theta)$ will modify the trajectory and thereby the return

The objective

- The probability of a trajectory T is a function of $f(A|S;\theta)$ and hence of θ
  - $T \sim P(T;\theta)$
- The probability of a return G is a function of the trajectory T
  - $G(T)$
- Objective: to maximize expected return

$$\underset{\theta}{\operatorname{argmax}}\, J(\theta) = \underset{\theta}{\operatorname{argmax}} \sum_{T} P(T;\theta)\, G(T)$$

Gradient of the objective

$$J(\theta) = \sum_T P(T;\theta)\, G(T)$$

$$\nabla_\theta J(\theta) = \sum_T \nabla_\theta P(T;\theta)\, G(T)$$

- A simple trick:

$$\nabla_\theta P(T;\theta) = P(T;\theta)\frac{\nabla_\theta P(T;\theta)}{P(T;\theta)} = P(T;\theta)\nabla_\theta \log P(T;\theta)$$

$$\nabla_\theta J(\theta) = \sum_T P(T;\theta)\nabla_\theta \log P(T;\theta)\, G(T)$$

$$\boxed{\nabla_\theta J(\theta) = E_{T\sim P(T;\theta)}\nabla_\theta \log P(T;\theta)\, G(T)}$$

Gradient of the objective

$$J(\theta) = \sum_T P(T;\theta)\, G(T)$$

$$\nabla_\theta J(\theta) = \sum_T \nabla_\theta P(T;\theta)\, G(T)$$

- A simple trick:

$$\nabla_\theta P(T;\theta) = P(T;\theta)\frac{\nabla_\theta P(T;\theta)}{P(T;\theta)} = P(T;\theta)\nabla_\theta \log P(T;\theta)$$

$$\nabla_\theta J(\theta) = \sum_T P(T;\theta)\nabla_\theta \log P(T;\theta)\, G(T)$$

$$\boxed{\nabla_\theta J(\theta) = E_{T\sim P(T;\theta)}\nabla_\theta \log P(T;\theta)\, G(T)}$$

The trajectory

- The trajectory T is

$$T = S_1\ A_1\ S_2\ A_2\ S_3\ A_3 \ldots$$

- The probability of T, under the policy function $f(A|S;\theta)$ is

$$P(T;\theta)\ =\ P(S_1)\ f(A_1|S_1;\theta)P(S_2|S_1,A_1)f(A_2|S_2;\theta)\ \ldots$$

- Taking logs

$$\log P(T;\theta)\ = \log P(S_1) + \sum_t \log P(S_{t+1}|S_t,A_t) + \sum_t \log f(A_t|S_t;\theta)$$

- Giving us the deriviative

$$\boxed{\nabla_\theta \log P(T;\theta) = \sum_t \nabla_\theta \log f(A_t|S_t;\theta)}$$

Gradient of the objective

$$\nabla_\theta J(\theta)$$

$$= E_{T \sim P(T;\theta)} \left(\sum_t \nabla_\theta \log f(A_t | S_t; \theta) \right) G(T)$$

- This is a simple expectation that can be approximated by sampling!

A simple extension

$$\nabla_\theta J(\theta) = E_{T\sim P(T;\theta)}\left(\sum_t \nabla_\theta \log f(A_t|S_t;\theta)\right) G(T)$$

- Better to compute the above instead as follows

$$\nabla_\theta J(\theta) = E_{T\sim P(T;\theta)} \sum_t \nabla_\theta \log f(A_t|S_t;\theta) G(t)$$

- This too can be estimated by sampling

Policy Gradients

- Record an episode (or episodes)

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

Policy Gradients

- Episode

$G(3)$

$G(2)$

$G(1)$

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

- Compute returns at each time

Policy Gradients

- Episode

$G(3)$

$G(2)$

$G(1)$

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

$\log f(A_1|S_1;\theta)$ $\log f(A_2|S_2;\theta)$ $\log f(A_3|S_3;\theta)$

- Compute returns at each time
- Compute log policy at each time

Policy Gradients

- Episode

$G(3)$

$G(2)$

$G(1)$

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

$\log f(A_1|S_1;\theta)$ $\log f(A_2|S_2;\theta)$ $\log f(A_3|S_3;\theta)$

$$\nabla_\theta J(\theta) \approx \frac{1}{T}\sum_t \nabla_\theta \log f(A_t|S_t;\theta)G(t)$$

- Compute returns at each time
- Compute log policy at each time
- Compute gradient

Policy Gradients

- Episode

$G(3)$

$G(2)$

$G(1)$

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

$\log f(A_1|S_1;\theta)$ $\log f(A_2|S_2;\theta)$ $\log f(A_3|S_3;\theta)$

$$\nabla_\theta J(\theta) \approx \frac{1}{T}\sum_t \nabla_\theta \log f(A_t|S_t;\theta)G(t)$$

$$\theta = \theta + \eta\nabla_\theta J(\theta)$$

- Compute returns at each time
- Compute log policy at each time
- Compute gradient
- Update network parameters
  - Ideally $\nabla_\theta J(\theta)$ is averaged over many episodes

Its like Maximum Likelihood

- The gradient actually looks like the derivative of a log likelihood function

$$\nabla_\theta J(\theta) = E_{T\sim P(T;\theta)} \nabla_\theta \log P(T;\theta)\, G(T)$$

- Can be written as

$$\nabla_\theta J(\theta) = E_{T\sim P(T;\theta)} \nabla_\theta \log P(T;\theta)^{G(T)}$$

- Maximization increases the probability of trajectories with greater return
  - If you see a trajectory you increase its probability

Its like Maximum Likelihood

- The gradient actually looks like the derivative of a log likelihood function

$$\nabla_\theta J(\theta) \approx \frac{1}{T} \sum_t \nabla_\theta \log f(A_t | S_t; \theta)^{G(t)}$$

- Maximization increases the probability of all seen actions
  - At the cost of the probability of unseen actions
  - Usual ML estimator

Merely seeing a trajectory isn't good

- We want to emphasize trajectories with high return and *reduce* the probability of low-return trajectories

- If an action results in more returns than the current average return for the state, we must improve its probability
  - If it results in less, we must decrease it

Its like Maximum Likelihood

- Subtract the *expected* return for the current state

$$\nabla_\theta J(\theta) \approx \frac{1}{T}\sum_t \nabla_\theta \log f(A_t|S_t;\theta)^{G(t)-v(S_t)}$$

- $a_t = G(t) - v(S_t)$ is the *advantage function*
  - How much advantage the current action has over the average
- Train $f(A_t|S_t;\theta)$ to maximize advantage
  - Typically approximate $v(S_t)$ by $\frac{1}{T}\sum_{t'} G(S_{t'})$

Reinforce

- Initialize θ
- For each episode e
  - Initialize S_1, A_1
  - For $t = 1 \ldots Termination$
    - Choose action A_t using ε –greedy policy obtained from θ
    - Observe R_{t+1}, S_{t+1}
  - Compute the returns $G(S_t)$, then the advantages a_t
  - Compute $J(\theta) = \frac{1}{T}\sum_t \log\big(\pi_\theta(A_t|S_t)\big)\, a_t$
  - $\theta \leftarrow \theta + \eta \nabla_\theta J(\theta)$

Solution

- Recast differentiation as an *expectation* operation
  - Can now be approximated by sampling
  - Policy gradient method
- Compute expected returns using an action-value function approximator
  - Actor-critic methods

Instability

- In Reinforce, the estimator for the expected return has high variance : rewards on one episode act as estimates for state value functions.

$$G(S_t) = \sum_{t' \geq t} \gamma^{t'-t} R_{t'+1}$$

- It also requires entire runs of episodes
  - Not online
- It can be made more stable through function approximation of the value function

Actor-Critic

- In actor-critic methods, two networks are used :
- The **actor** is the policy network : $f(S, A|\theta_a) = \pi_{\theta_a}(A|S)$ and is used to predict the next action
- The **critic** is a state value network : $\mathrm{g}(S|\theta_c) = V_{\theta_c}(S)$ and is used to guide the optimization direction of the actor

- To estimate the expected return based on an episode, we use N-step lookahead :

$$G(S_t) = \sum_{0 \leq k \leq N-1} \gamma^k R_{t+k+1} + \gamma^N V_{\theta_c}(S_{t+N})$$

Advantage Actor Critic (A2C)

Rethink the advantages

The critic can also be used as the “baseline” when computing the advantages :

$$a_t = G(S_t) - V_{\theta_c}(S_t)$$

The trajectory’s probability is improved if it is better than the trajectories previously followed.

The critic is trained on how well it predicted the return.

A2C

- Initialize θ_a, θ_c
- For each episode e
  - Initialize S_1, A_1
  - For $t = 1 \ldots Termination$
    - Choose action A_t using ε –greedy policy obtained from θ_a
    - Observe R_{t+1}, S_{t+1}
  - Compute the returns $G(S_t) = \sum_{0 \le k \le N-1} \gamma^k R_{t+k+1} + \gamma^N V_{\theta_c}(S_{t+N})$ if $\mathrm{t} + N < T$, else $\sum_{0 \le k \le T-t-1} \gamma^k R_{t+k+1}$
  - Compute the advantages $a_t = G(S_t) - V_{\theta_c}(S_t)$
  - Compute $f_a(\theta_a) = \frac{1}{T} \sum_t \log\big(\pi_\theta(A_t|S_t)\big)\, a_t$ **,** $L_c(\theta_c) = \frac{1}{T} \sum_t \Big(G(S_t) - V_{\theta_c}(S_t)\Big)^{\mathbf{2}}$
  - $\theta_a \leftarrow \theta_a + \eta_a \nabla_{\theta_a} L_a\,(\theta_a)$, $\theta_c \leftarrow \theta_c\ - \eta_c \nabla_{\theta_c} L_c\,(\theta_c)$,

Extensions

- A2C can be applied in a multi-thread environment on several episodes simultaneously, with a final mini-batch update
- **Asynchronous Advantage Actor-Critic** (A3C) (Deepmind, 2016): Each thread performs its updates without waiting for the others to end → **each thread keeps its own version of the parameters**. They upload their gradients asynchronously to a master server that performs batch updates
- Experience Replay can be adapted to A2C → ACER algorithm (Deepmind 2017)

Continuous action space

- We need to access action probabilities $\pi_\theta(A_t|S_t)$ for Reinforce and A2C.
- We have seen the discrete action space case (n labels + softmax) → Very large or **continuous space** ?
- You can use a network that **predict the parameters of a distribution** and sample an action from it. Ex : $A_t \sim \mathrm{N}(\mu, \sigma)$ with $\mu, \sigma = f(S_t|\theta)$ (similar to the encoder of a VAE) → Reinforce/A2C can be used (with the reparametrization trick).
- Most general case : $f(S_t|\theta) = A_t$. What algorithm can I use ?

Deep Deterministic policy gradients (DDPG)

- Hybrid between Q-learning and policy methods. Makes use of many tricks seen so far.
- An **actor** predicts the action : $f(S|\theta_a) = A$.
- A **critic** predicts the **action value** : $\mathrm{g}(S, A|\theta_c) = \mathrm{Q}_{\theta_c}(\mathrm{S}, \mathrm{A})$.
- Actor objective : maximize the Q-value → Gradient ascent with $\nabla_{\theta_a}\mathrm{g}(S, f(S|\theta_a)|\theta_c)$

Deep Deterministic policy gradients (DDPG)

- Critic objective : predict accurately the Q-value. Could be done with bootstrapping but like Double DQN, DDPG makes use of decoupled targets instead
- → Separate set of target actor and critic with parameters θ'_a, θ'_c
- Minimize $\left(\mathrm{g}(S_t, A_t|\theta_c) - R_{t+1} - \gamma g(S_{t+1}, f(S_{t+1}|\theta'_a)|\theta'_c)\right)^2$ wrt θ_c
- θ'_a, θ'_c are **slowly updated** as a moving average of $\theta_\mathrm{a}, \theta_c$
- DDPG also uses experience replay, and in training adds a noise to $f(S|\theta_a)$ for exploration.

Summary

- Parameterized Functions
- Action-replay
- Target functions
- Deep Q Networks
- Decoupled targets, Double DQN
- Policy gradients
- Reinforce
- Actor-Critic
- DDPG