

CSCI 1470/2470
Spring 2024

Ritambhara Singh

April 19, 2024
Friday

Reinforcement Learning :Q-Learning

Deep Learning



Review: Q-value and V-value Tables (made up)

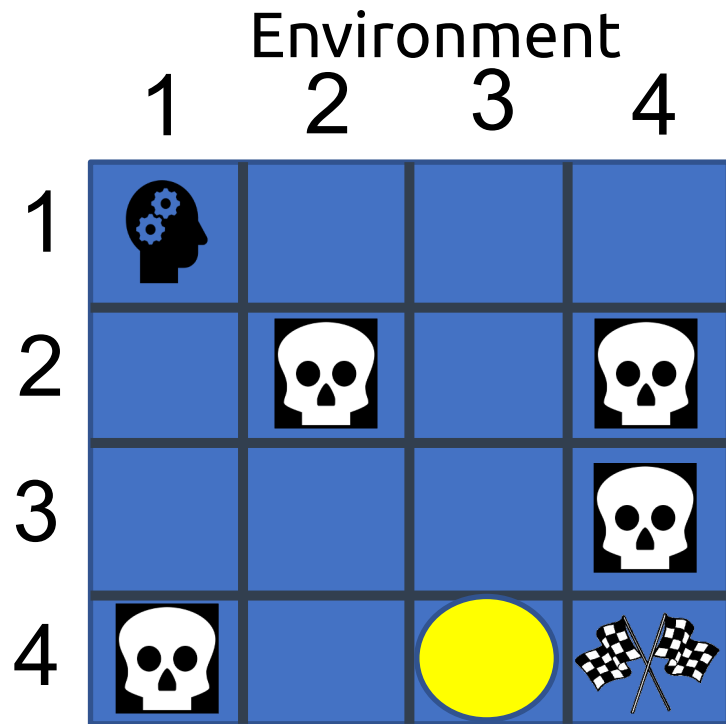
	Action #1	Action #2
State #1	0	-1
State #2	0.1	1
State #3	-1	-10
State #4	0	1.9
State #5	10	0
State #6	-10	-10

State	Value
State #1	0
State #2	1
State #3	-1
State #4	1.9
State #5	10
State #6	-10

Review: Value iteration pseudocode

1. For all s , set $V(s) := 0$.
2. Repeat until convergence:
 1. For all s :
 1. For all a , set $Q(s, a) := \sum_{s' \in S} T(s, a, s')[R(s, a, s') + \gamma V(s')]$
 2. $V(s) := \max_a Q(s, a)$
3. Return Q

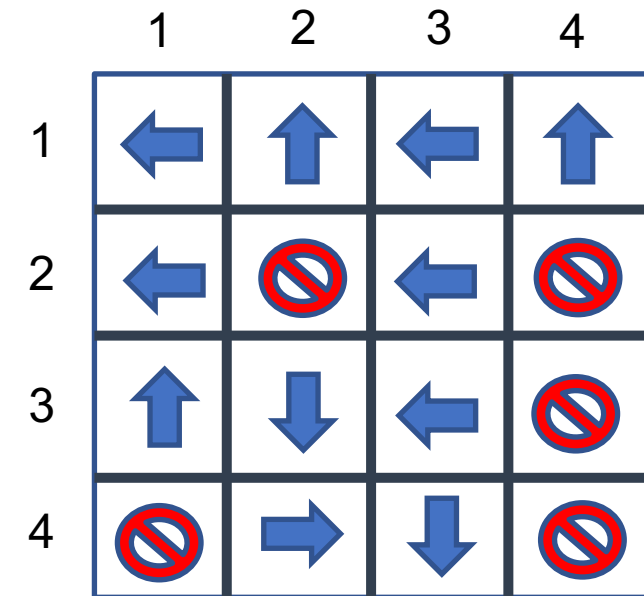
Review: Frozen Lake – final value table & optimal policy




Example Final Value Table

	1	2	3	4
1	0.068	0.061	0.074	0.055
2	0.092	0	0.112	0
3	0.145	0.247	0.3	0
4	0	0.38	0.639	0

Final Policy



Organizing RL problems/algorithms

	Know T and R	Don't know T and R
Simple/discrete	Value iteration	Q-Learning
Complex/continuous		Deep Q-Networks REINFORCE Actor-Critic

For a more complete taxonomy of RL algorithms, see https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below

Tabular Q-learning

Motivation: Why Not Value Iteration?

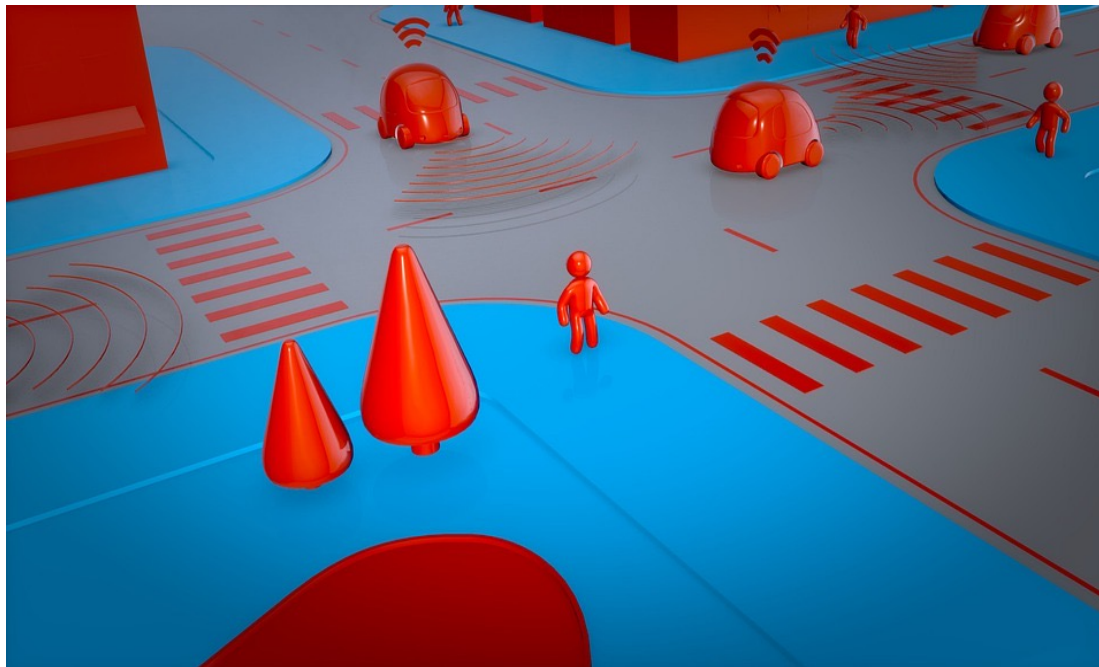
In value iteration, we assume that we know the transition and reward functions, but what if this isn't the case?

$$\{S, A, \cancel{B}, \cancel{V}, \gamma\}$$

How can we learn in this scenario?

Examples of Unknown T and/or R:

Self-Driving Cars



<https://pixabay.com/illustrations/self-driving-car-autonomous-4309836/>

Can't design T for a self-driving car.
Too complex to model directly

Frozen Lake

T,R = ?

S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	G

What if we don't know how slippery the lake is?
(i.e. T is unknown)

First Attempt

Start in Frozen Lake with no knowledge of T or R

S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	G

Take random actions to estimate T and R



Run value iteration using estimates of T and R

$$Q(s, a) = \sum_{s'} T(s, a, s') R(s, a, s') + \gamma V(s')$$

$$V(s) = \operatorname{argmax}_a (Q(s, a))$$

OpenAI Gym (<https://gym.openai.com/>)



Gym

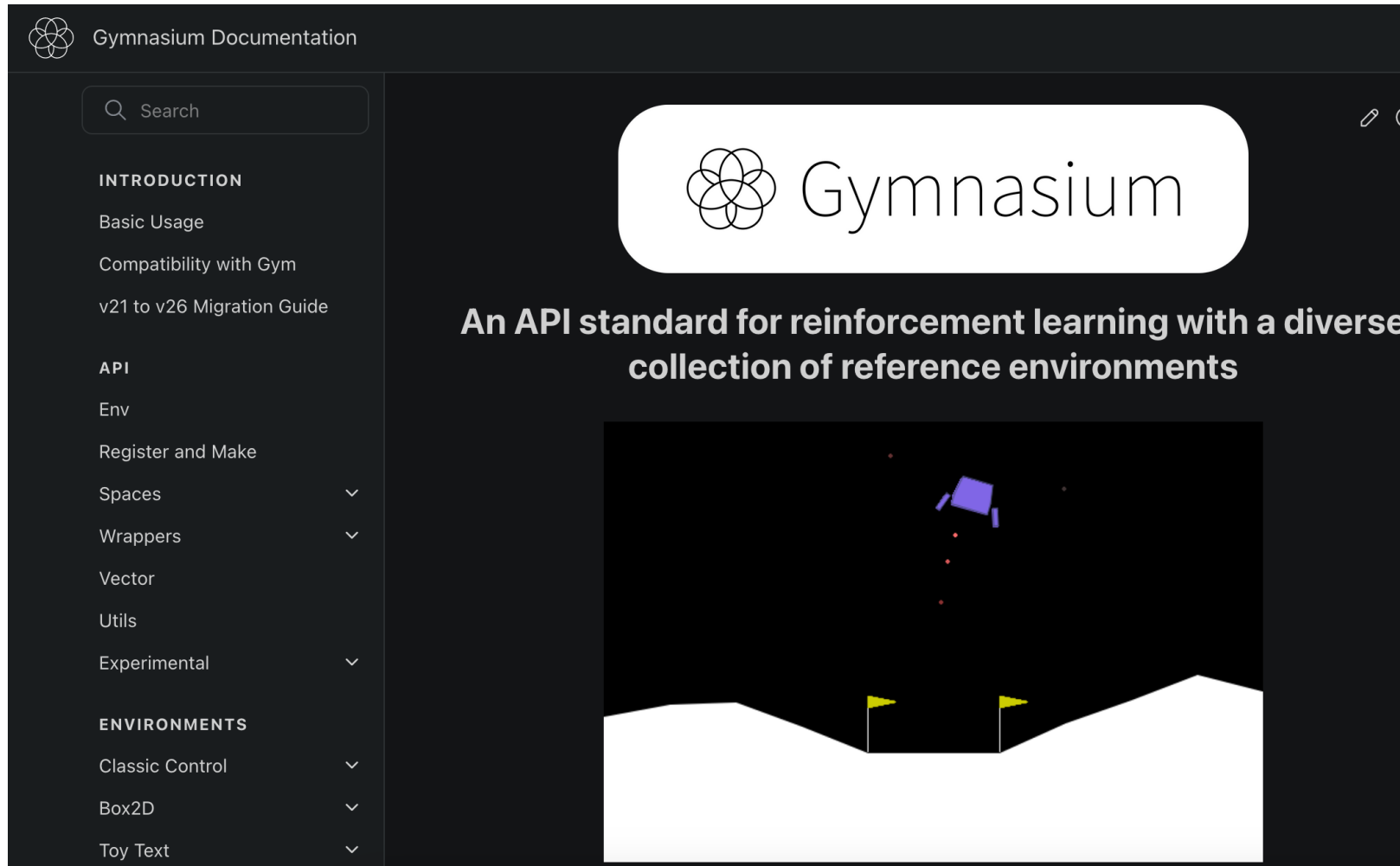
Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball.

[View documentation >](#)

[View on GitHub >](#)

Farama Foundation

(<https://gymnasium.farama.org>)



Gymnasium Documentation

Search

INTRODUCTION

- Basic Usage
- Compatibility with Gym
- v21 to v26 Migration Guide

API

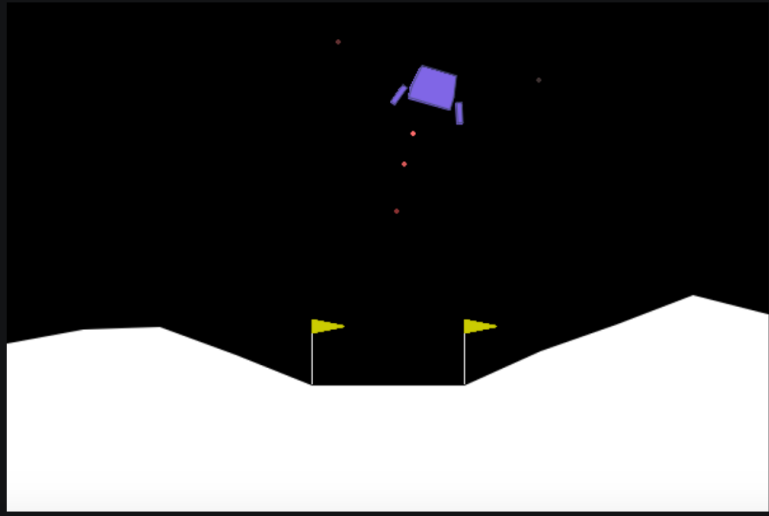
- Env
- Register and Make
- Spaces
- Wrappers
- Vector
- Utils
- Experimental

ENVIRONMENTS


- Classic Control
- Box2D
- Toy Text

Gymnasium

An API standard for reinforcement learning with a diverse collection of reference environments



Frozen Lake in OpenAI Gym




Gym Documentation

Search

INTRODUCTION

- Basic Usage
- API
- Core
- Spaces
- Wrappers
- Vector
- Utils

Frozen Lake



This environment is part of the [Toy Text environments](#). Please read that page first for general information.

Action Space	Discrete(4)
Observation Space	Discrete(16)
Import	<code>gym.make("FrozenLake-v1")</code>

Frozen Lake 'Wandering' Demo

https://colab.research.google.com/drive/1yBCDzAXlu9j0A2aT8fH1zm7beBgtB9yY#scrollTo=sI_x2TsCp15L

Problems with This Approach

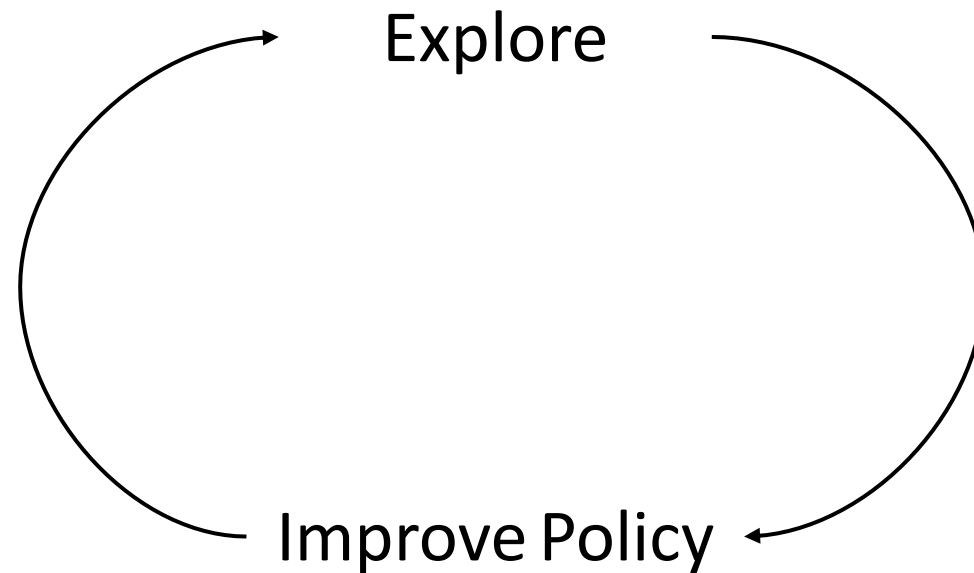
S 32709	F 13151	F 5951	F 2868
F 12335	H 7095	F 1760	H 1141
F 4646	F 1579	F 1015	H 252
H 1380	F 643	F 501	G 132

Values are the number of times each state was visited over 10000 episodes run in OpenAI Gym

So, what is the issue here?

We're extremely unlikely to reach the goal state through random wandering, so our estimates of T and R will probably be bad

How to Improve?



We can interleave policy improvement with wandering
(Get better at exploring *by exploring*)

But how do we improve our policy?

Q-Learning

- Every time we take an action, use the observed reward to update our estimate of Q
- $V(s)$ is still $\max_a Q(s, a)$, so it only matters how we update Q
- **Importantly, we use our estimates of Q at each step to pick what we think is the best action, instead of just moving randomly**
 - $\text{action} = \text{argmax}_a Q(s, a)$

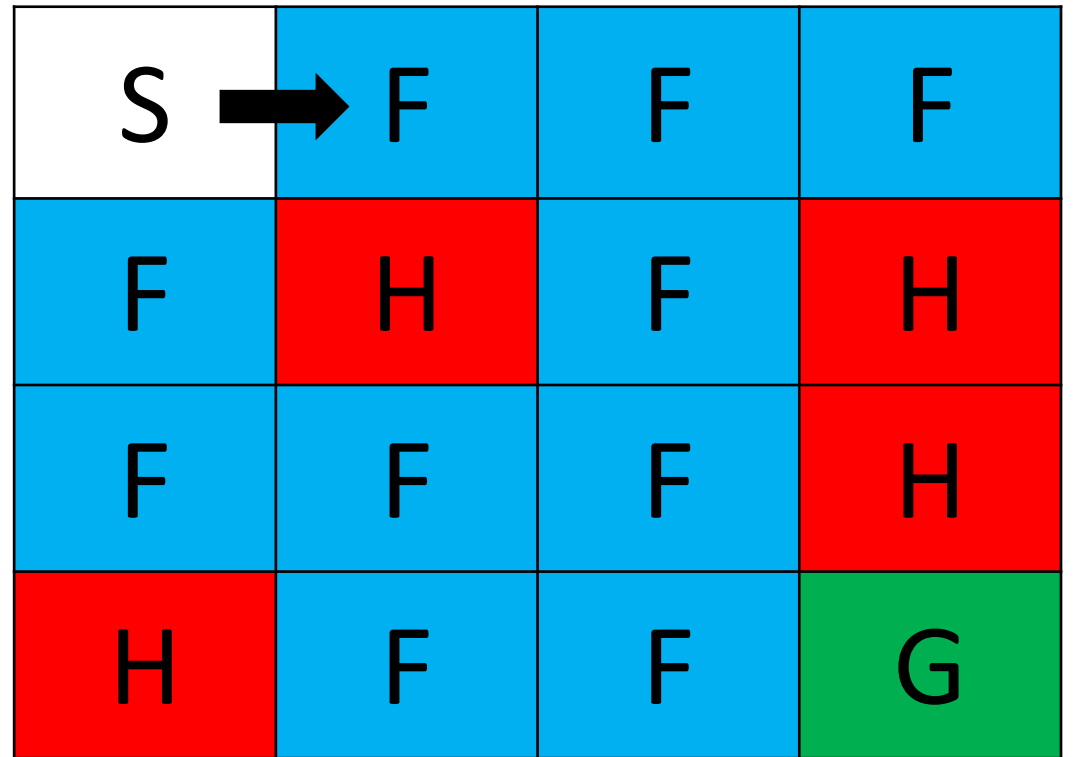
S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	G

$$Q^\pi(s, a) = E[R(s, a, s') + \gamma V^\pi(s')]$$

$$Q^\pi(s, a) = \sum_{s' \in S} P(s' | s, a) [R(s, a, s') + \gamma V^\pi(s')]]$$

Q-Learning

- Every time we take an action, use the observed reward to update our estimate of Q
- $V(s)$ is still $\max_a Q(s, a)$, so it only matters how we update Q
- Importantly, we use our estimates of Q at each step to pick what we think is the best action, instead of just moving randomly
 - $\text{action} = \operatorname{argmax}_a Q(s, a)$



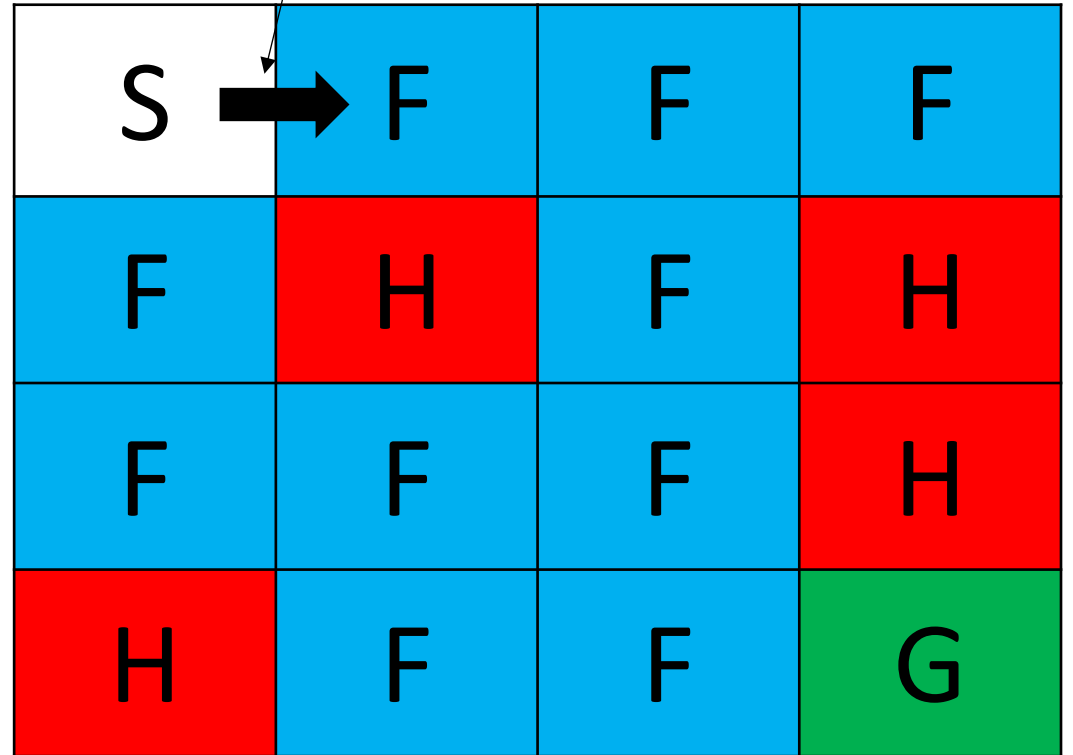
$$Q^\pi(s, a) = E[R(s, a, s') + \gamma V^\pi(s')]$$

$$Q^\pi(s, a) = \sum_{s' \in S} P(s' | s, a) [R(s, a, s') + \gamma V^\pi(s')]]$$

Q-Learning

- Every time we take an action, use the observed reward to update our estimate of Q
- $V(s)$ is still $\max_a Q(s, a)$, so it only matters how we update Q
- Importantly, we use our estimates of Q at each step to pick what we think is the best action, instead of just moving randomly
 - action = $\operatorname{argmax}_a Q(s, a)$

Use Q to pick action a_0 .
 Observe transition and get reward r_0 . Use r_0 to update $Q(s_0, a_0)$ immediately



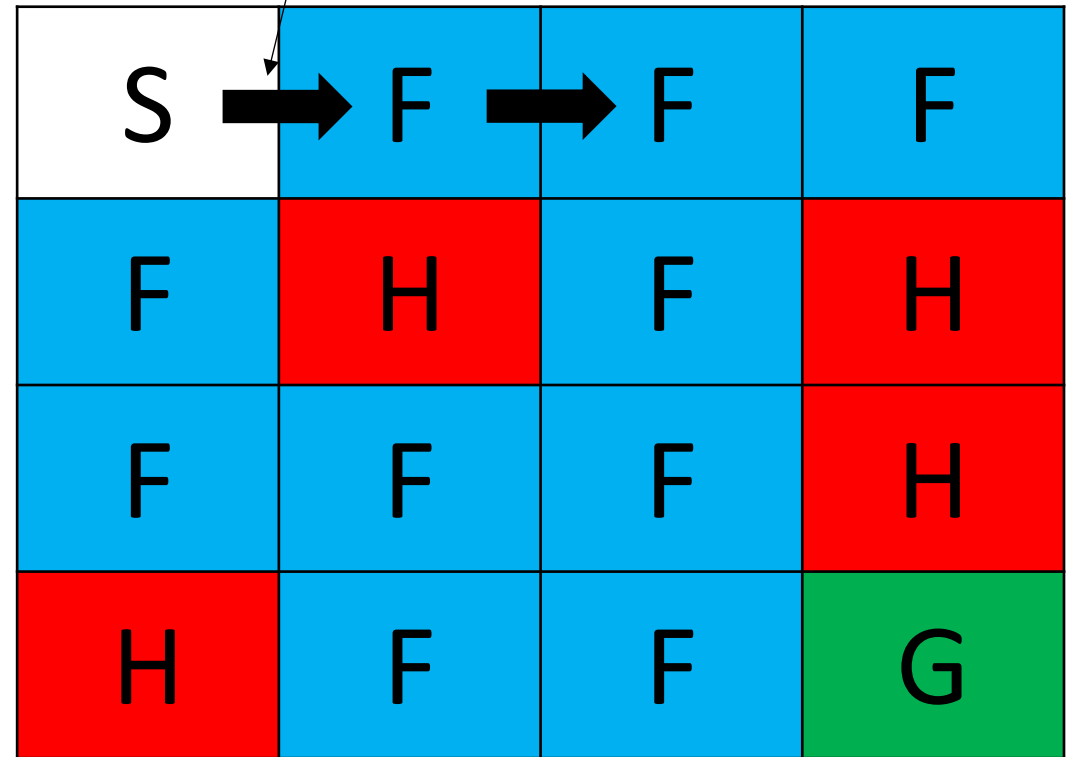
$$Q^\pi(s, a) = E[R(s, a, s') + \gamma V^\pi(s')]$$

$$Q^\pi(s, a) = \sum_{s' \in S} P(s'|s, a) [R(s, a, s') + \gamma V^\pi(s')]$$

Q-Learning

- Every time we take an action, use the observed reward to update our estimate of Q
- $V(s)$ is still $\max_a Q(s, a)$, so it only matters how we update Q
- Importantly, we use our estimates of Q at each step to pick what we think is the best action, instead of just moving randomly
 - $\text{action} = \text{argmax}_a Q(s, a)$

Use Q to pick action a_0 .
Observe transition and get reward r_0 . Use r_0 to update $Q(s_0, a_0)$ immediately



$$Q^\pi(s, a) = E[R(s, a, s') + \gamma V^\pi(s')]$$

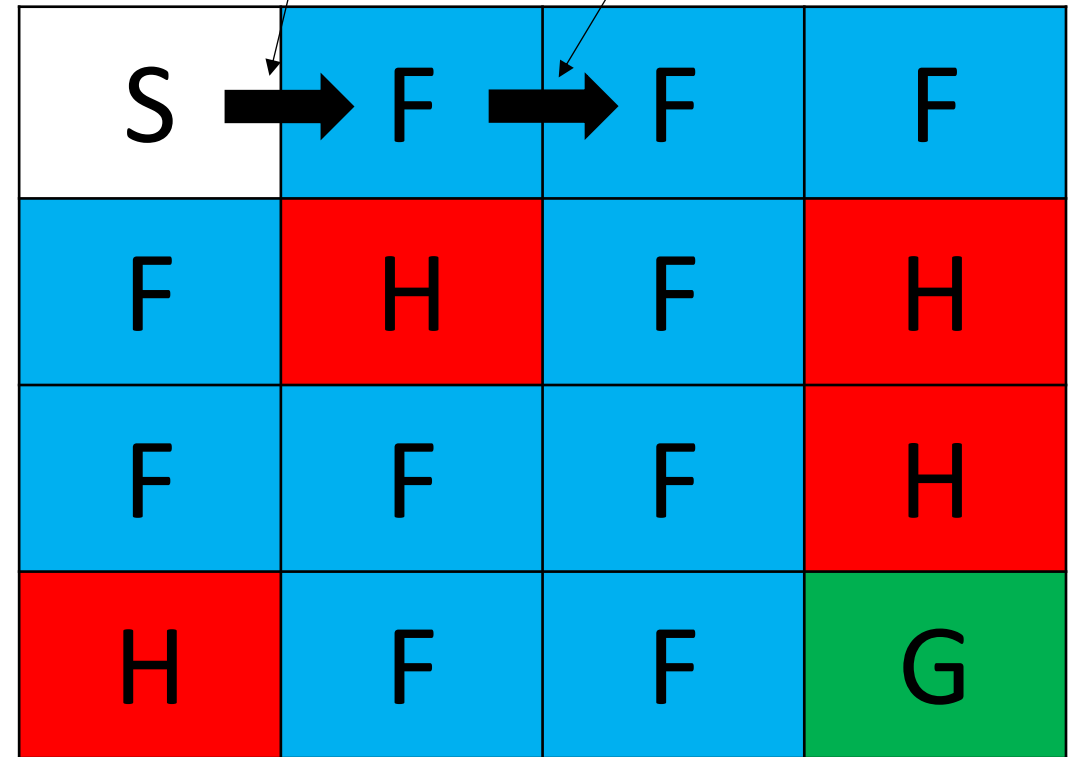
$$Q^\pi(s, a) = \sum_{s' \in S} P(s'|s, a) [R(s, a, s') + \gamma V^\pi(s')]]$$

Q-Learning

- Every time we take an action, use the observed reward to update our estimate of Q
- $V(s)$ is still $\max_a Q(s, a)$, so it only matters how we update Q
- Importantly, we use our estimates of Q at each step to pick what we think is the best action, instead of just moving randomly
 - $\text{action} = \text{argmax}_a Q(s, a)$

Use Q to pick action a_0 .
 Observe transition and get reward r_0 . Use r_0 to update $Q(s_0, a_0)$ immediately

Repeat process for s_1 .
 Update $Q(s_1, a_1)$ with r_1



$$Q^\pi(s, a) = E[R(s, a, s') + \gamma V^\pi(s')]$$

$$Q^\pi(s, a) = \sum_{s' \in S} P(s'|s, a) [R(s, a, s') + \gamma V^\pi(s')]$$

How do we Update Q?

Basic Strategy:

Take a weighted average of our old Q estimate with our new Q estimate

$$Q(s, a) = (1 - \alpha)Q_{old}(s, a) + \alpha Q_{new}(s, a)$$

Where the hyperparameter α controls how quickly we learn

But what should $Q_{new}(s, a)$ be?

Determining $Q_{new}(s, a)$

In value iteration, we took an expectation over all possible next states and actions to get a new estimate for $Q(s, a)$

$$Q(s, a) = \sum_{s'} T(s, a, s') (R(s, a, s') + \gamma V(s'))$$

How do we change this?

In Q-Learning, we only take one action and see one new state, and use that one transition to update our estimate for $Q(s, a)$, so our update rule becomes

$$Q(s, a) = R(s, a, s') + \gamma V(s')$$

i.e. the reward we observed for moving into state s' , plus whatever our current value function estimate thinks is the value of being in state s'

Any questions?



The Q-Learning Update Rule

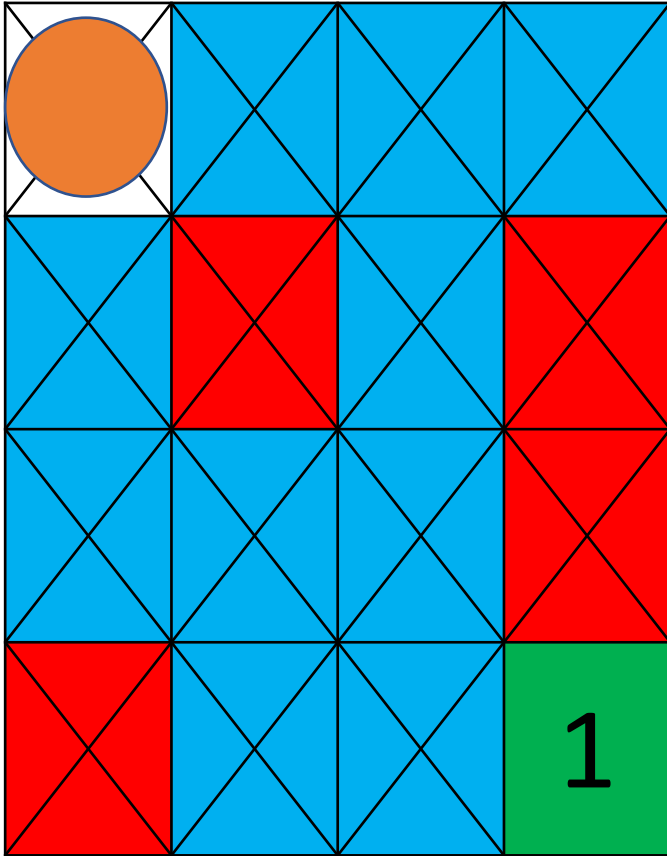
Combining our new estimate for $Q(s, a)$ with the weighted average equation, the final update rule for Q-Learning becomes

$$Q(s, a) = (1 - \alpha)Q_{old}(s, a) + \alpha Q_{new}(s, a)$$



$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(R(s, a, s') + \gamma V(s'))$$

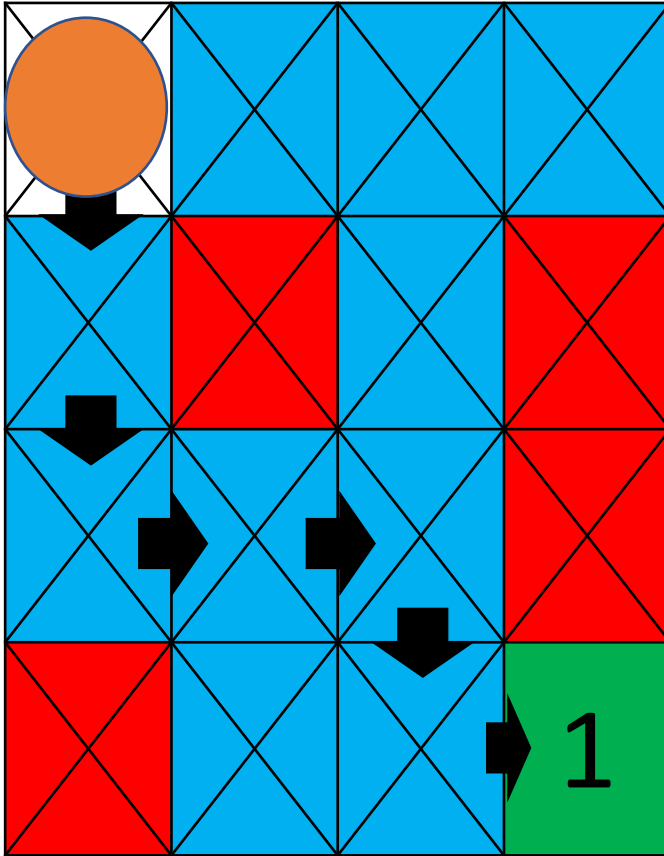
Q-Learning in Frozen Lake



- Initially, the agent is acting randomly, because all states have a value of zero
 - It would likely fall into the holes many times before reaching the goal state for the first time

- $\alpha = .5, \gamma = .99$
- Values in the grid are $Q(s,a)$
- Blank quadrants have a Q-value of 0

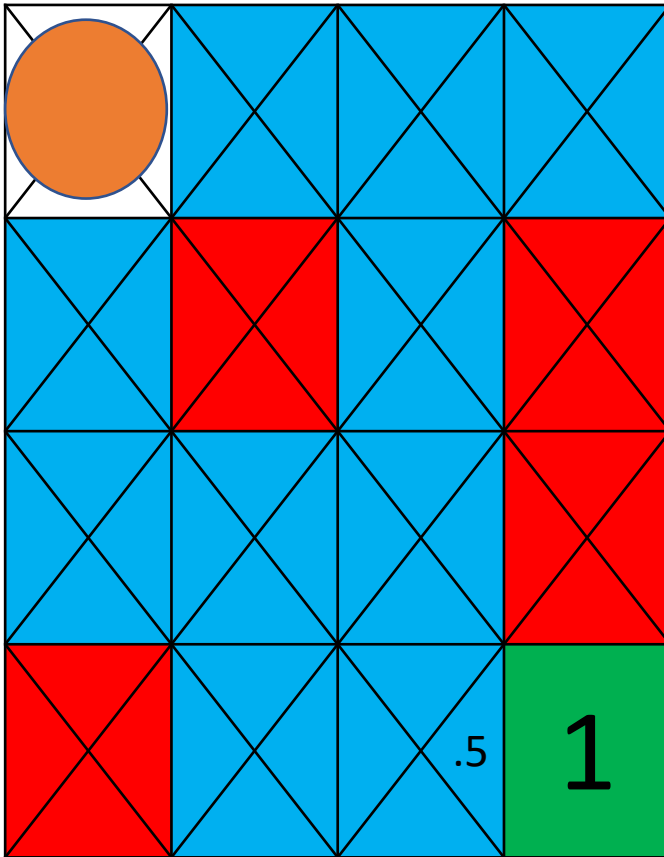
Q-Learning in Frozen Lake



- $\alpha = .5, \gamma = .99$
- Values in the grid are $Q(s,a)$
- Blank quadrants have a Q-value of 0

- Initially, the agent is acting randomly, because all states have a value of zero
 - It would likely fall into the holes many times before reaching the goal state for the first time, but for this example assume it got lucky
- At every transition in this path (each $\langle s,a,r,s' \rangle$ tuple), the agent sees a reward of zero up until the final one, so none of the Q-estimates change except the state before the goal
 - $Q(s,a) = (1 - \alpha)0 + \alpha(0) = 0$

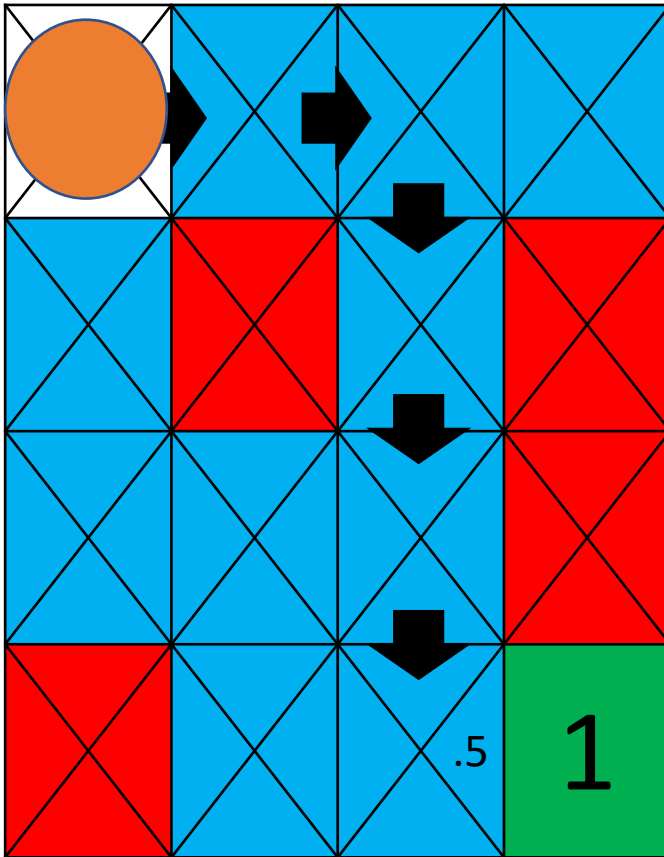
Q-Learning in Frozen Lake



- $\alpha = .5, \gamma = .99$
- Values in the grid are $Q(s,a)$
- Blank quadrants have a Q-value of 0

- Initially, the agent is acting randomly, because all states have a value of zero
 - It would likely fall into the holes many times before reaching the goal state for the first time, but for this example assume it got lucky
- At every transition in this path(each $\langle s,a,r,s' \rangle$ tuple), the agent sees a reward of zero up until the final one, so none of the Q-estimates change except the state before the goal
 - $Q(s,a) = (1 - \alpha)0 + \alpha(0) = 0$
- The last transition has a positive reward though, so the previous state action pair is changed
 - $Q(s,a) = (1 - \alpha)0 + \alpha(1) = .5$

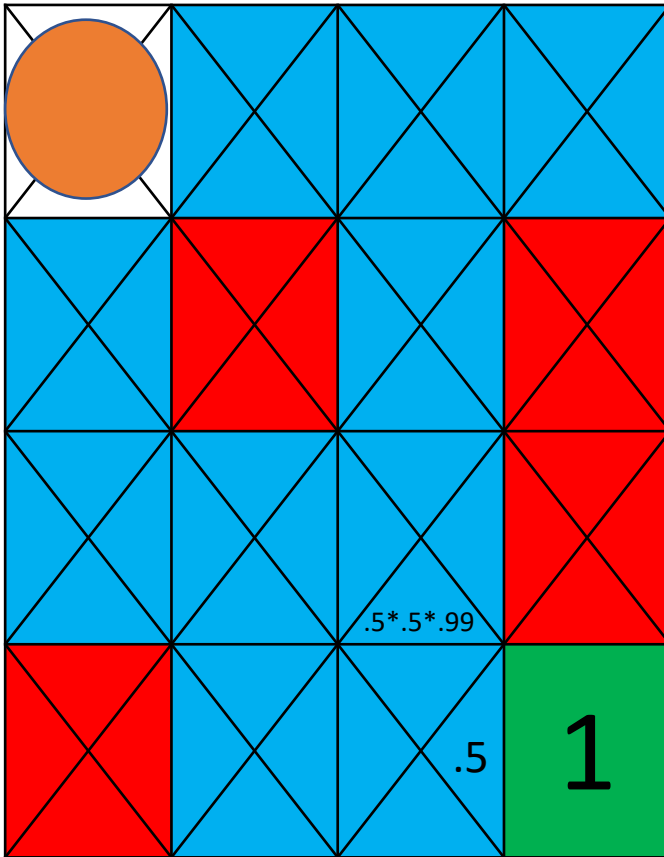
Q-Learning in Frozen Lake



- In another episode, the agent will still act randomly (since most values are still zero), unless it reaches the state next to the goal, which now has a positive value.
 - Assume it got lucky once again

- $\alpha = .5, \gamma = .99$
- Values in the grid are $Q(s,a)$
- Blank quadrants have a Q-value of 0

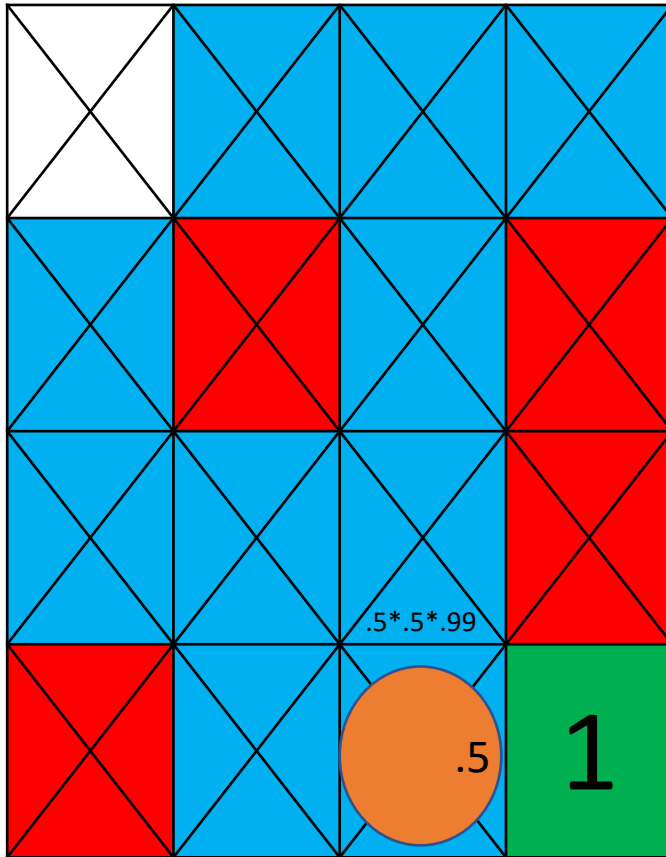
Q-Learning in Frozen Lake



- $\alpha = .5, \gamma = .99$
- Values in the grid are $Q(s,a)$
- Blank quadrants have a Q-value of 0

- In another episode, the agent will still act randomly (since most values are still zero), unless it reaches the state next to the goal, which now has a positive value.
 - Assume it got lucky once again
- When that happens, the previous state-action pair will be updated using the value of the state next to the goal
 - $Q(s, a) = (1 - \alpha)0 + \alpha(0 + \gamma * .5) = .5 * .5 * .99$

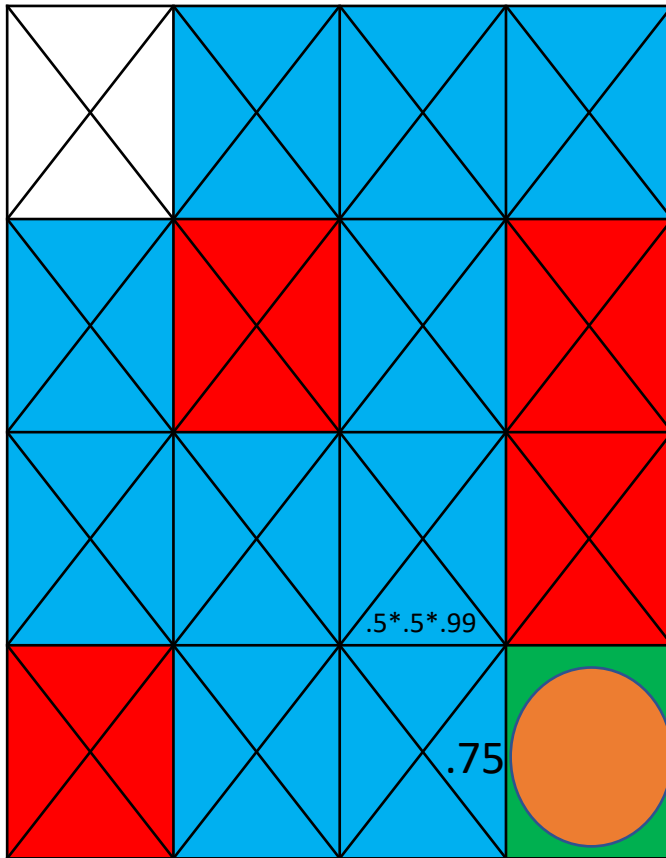
Q-Learning in Frozen Lake



- $\alpha = .5, \gamma = .99$
- Values in the grid are $Q(s,a)$
- Blank quadrants have a Q-value of 0

- In another episode, the agent will still act randomly (since most values are still zero), unless it reaches the state next to the goal, which now has a positive value.
 - Assume it got lucky once again
- When that happens, the previous state-action pair will be updated using the value of the state next to the goal
 - $Q(s, a) = (1 - \alpha)0 + \alpha(0 + \gamma * .5) = .5 * .5 * .99$
- When choosing its next action from the state next to the goal, the agent will choose $\text{argmax}_a Q(s, a)$, picking the action corresponding to the value of .5 (since all the other action-values for that state are zero).

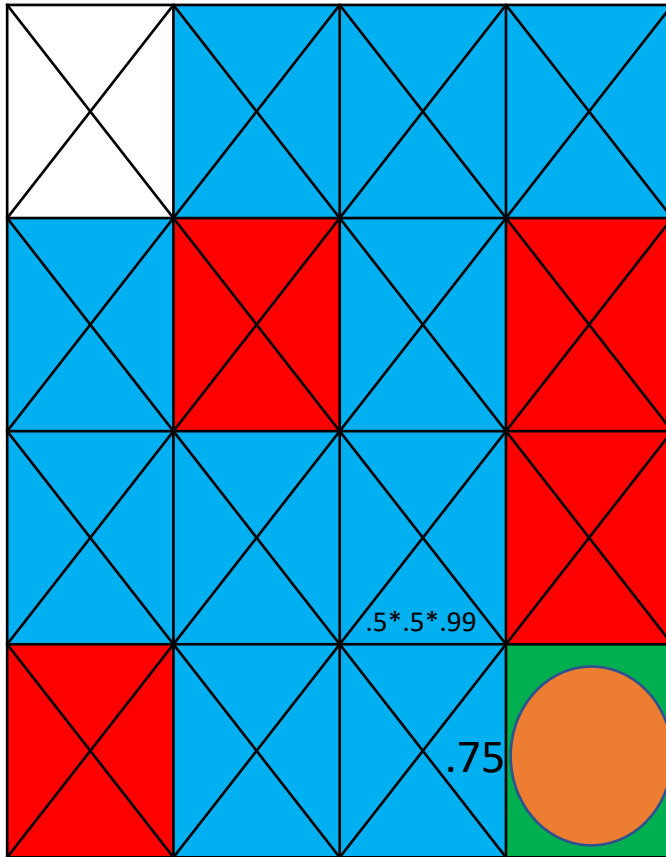
Q-Learning in Frozen Lake



- $\alpha = .5, \gamma = .99$
- Values in the grid are $Q(s,a)$
- Blank quadrants have a Q-value of 0

- In another episode, the agent will still act randomly (since most values are still zero), unless it reaches the state next to the goal, which now has a positive value.
 - Assume it got lucky once again
- When that happens, the previous state-action pair will be updated using the value of the state next to the goal
 - $Q(s, a) = (1 - \alpha)0 + \alpha(0 + \gamma * .5) = .5 * .5 * .99$
- When choosing its next action from the state next to the goal, the agent will choose $\text{argmax}_a Q(s, a)$, picking the action corresponding to the value of .5 (since all the other action-values for that state are zero).
- The value of the previous state is then updated again
 - $Q(s, a) = (1 - \alpha).5 + \alpha(1) = .75$

Q-Learning in Frozen Lake

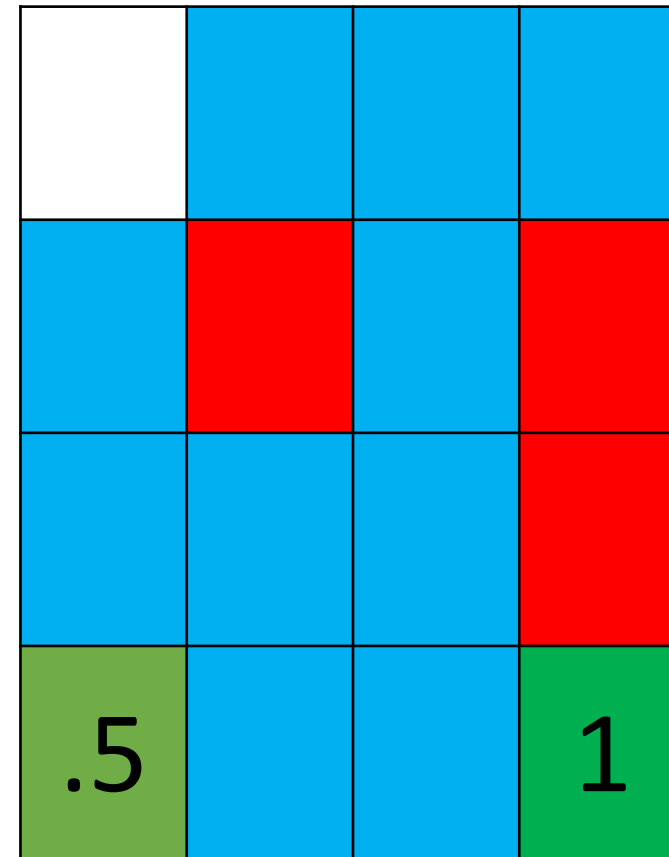


- $\alpha = .5, \gamma = .99$
- Values in the grid are $Q(s,a)$
- Blank quadrants have a Q-value of 0

- In another episode, the agent will still act randomly (since most values are still zero), unless it reaches the state next to the goal, which now has a positive value.
 - Assume it got lucky once again
- When that happens, the previous state-action pair will be updated using the value of the state next to the goal
 - $Q(s,a) = (1 - \alpha)0 + \alpha(0 + \gamma * .5) = .5 * .5 * .99$
- When choosing its next action from the state next to the goal, the agent will choose $\text{argmax}_a Q(s,a)$, picking the action corresponding to the value of .5 (since all the other action-values for that state are zero).
- The value of the previous state is then updated again
 - $Q(s,a) = (1 - \alpha).5 + \alpha(1) = .75$
- This process repeats over and over again until the Q-values converge to $Q^*(s,a)$, the optimal Q-values.

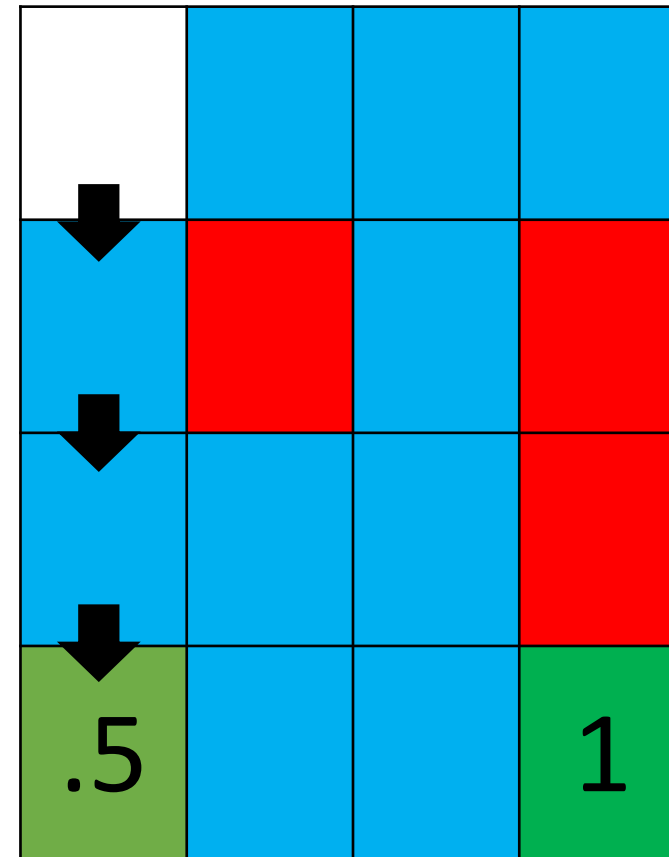
Problem: Exploration/Exploitation

- But what if Frozen Lake had two goal states?



Problem: Exploration/Exploitation

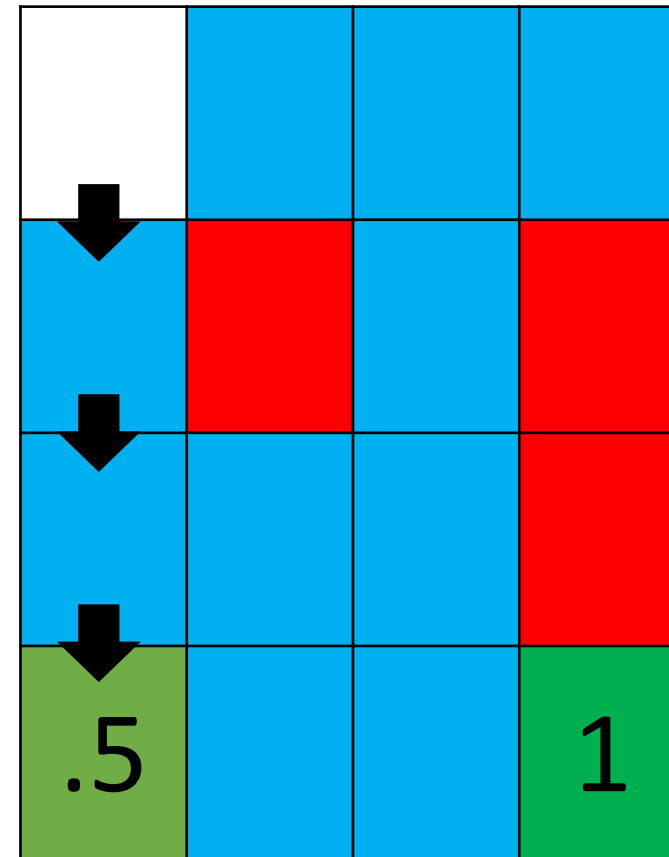
- But what if Frozen Lake had two goal states?
 - Q-Learning could learn a path to the .5 reward without ever getting to the 1 reward state.
 - Converges to a suboptimal solution



Problem: Exploration/Exploitation

Any ideas?

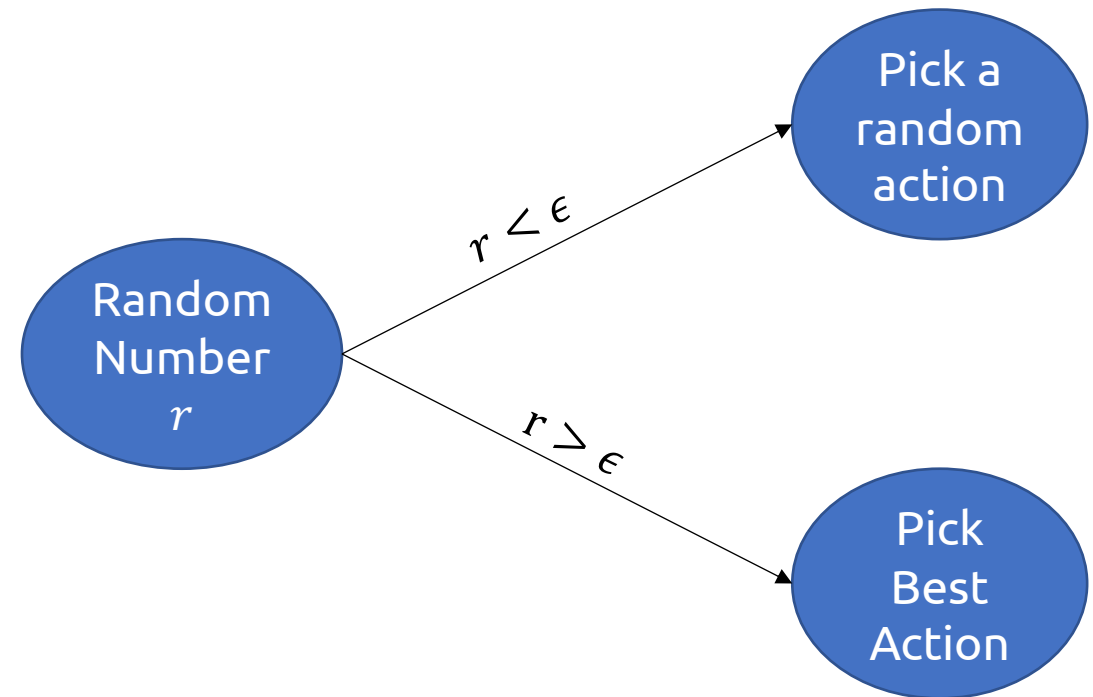
- But what if Frozen Lake had two goal states?
 - Q-Learning could learn a path to the .5 reward without ever getting to the 1 reward state.
 - Converges to a suboptimal solution
- How can we balance exploiting knowledge we already have with exploring unseen parts of the state space?



Solution: Epsilon-Greedy Policies

- Instead of always following our estimate of Q, we instead take random actions ϵ percent of the time, where ϵ is a hyperparameter
- We can also decrease ϵ over time, as our estimates of Q improve
 - Ex: after each episode, set $\epsilon = \frac{E}{(E+i)}$, where i is the number of episodes experienced so far
 - This lets the agent act less randomly over time

Action Selection Procedure



Any questions?




Q-Learning Update in Code

```
if np.random.rand(1) < epsilon:  
    act = env.action_space.sample()  
else:  
    act = np.argmax(Q[st])  
nst, rwd, done, _ = env.step(act)  
Q[st][act] = (1-alpha)Q[st][act] + alpha(rwd +  
gamma*V[nst])
```

Q-Learning Code Demo

https://colab.research.google.com/drive/1yBCDzAXlu9j0A2aT8fH1zm7beBgtB9yY#scrollTo=sI_x2TsCp15L

Organizing RL problems/algorithms

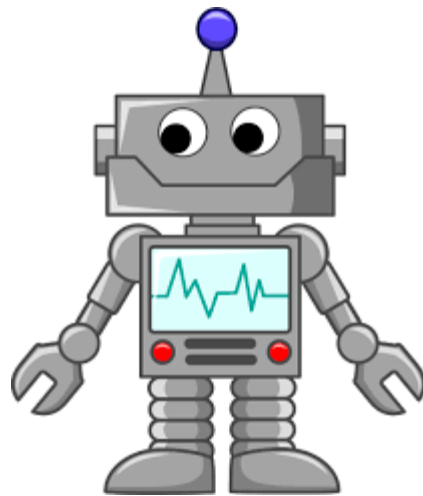
	Know T and R	Don't know T and R
Simple/discrete	Value iteration	Q-Learning
Complex/continuous		Deep Q-Networks REINFORCE Actor-Critic

For a more complete taxonomy of RL algorithms, see https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below

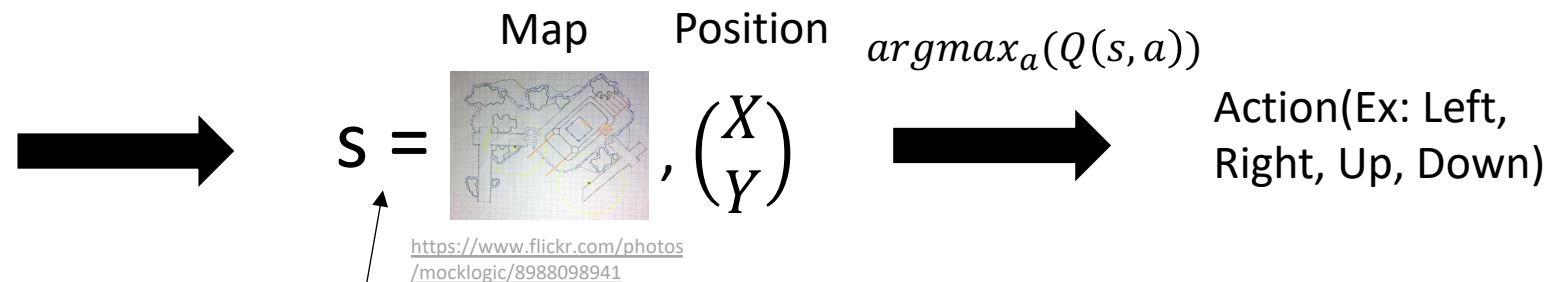
Deep Q-learning

Limitations of Tabular Q-Learning

- Tabular methods can't handle large or continuous state spaces
 - Can't have a table for infinite Q or V values
- Lots of problems have these kinds of state spaces
 - Robotic navigation: The state of the robot could be (map of environment, position), where position is a 2-D vector. Infinite possible states in this setup.
 - **Go: has $\sim 10^{170}$ states**



https://en.wikipedia.org/wiki/File:Cartoon_Robot.svg



Too many possible values for a table

Beyond Tabular Learning

In Q-learning, we are learning a function Q that maps from a state-action pair to a real number

$$Q: (s, a) \rightarrow \mathbb{R}$$

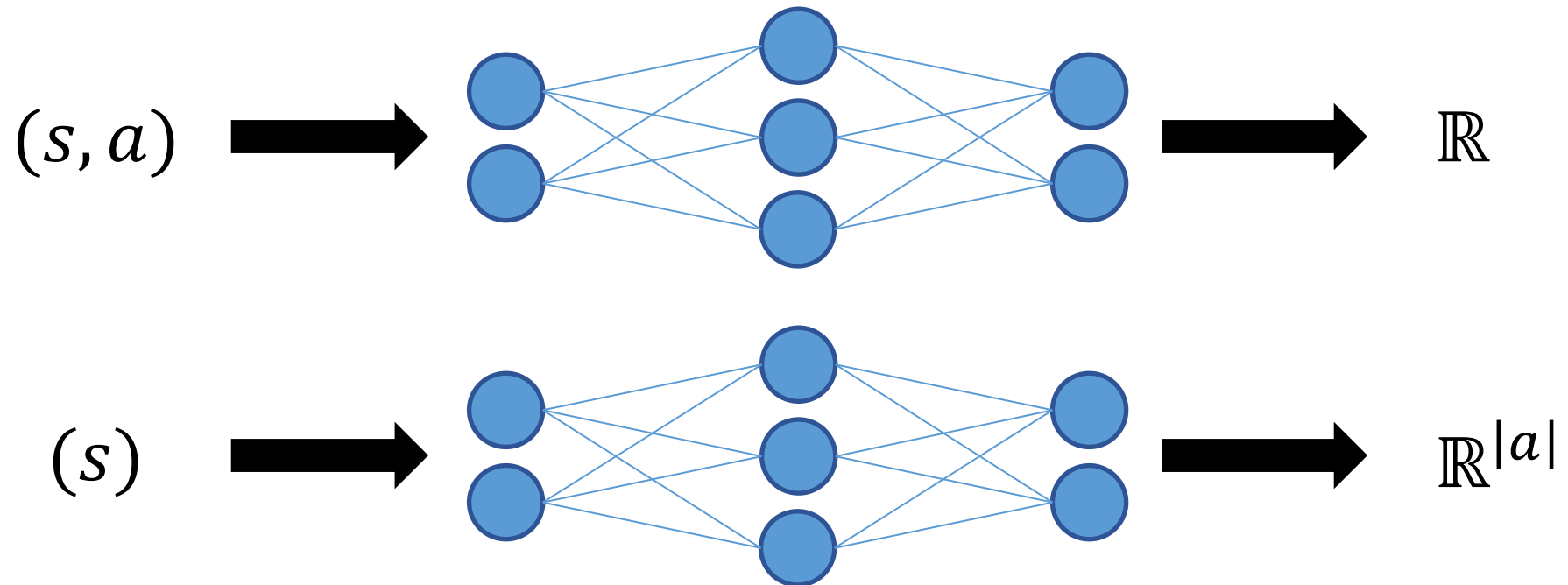
Or, equivalently, from a state to a vector of real numbers

$$Q: (s) \rightarrow \mathbb{R}^{|a|}$$

Instead of storing every (s,a) in a table to learn this function, we can learn a function to approximate Q using a (relatively) small set of parameters θ

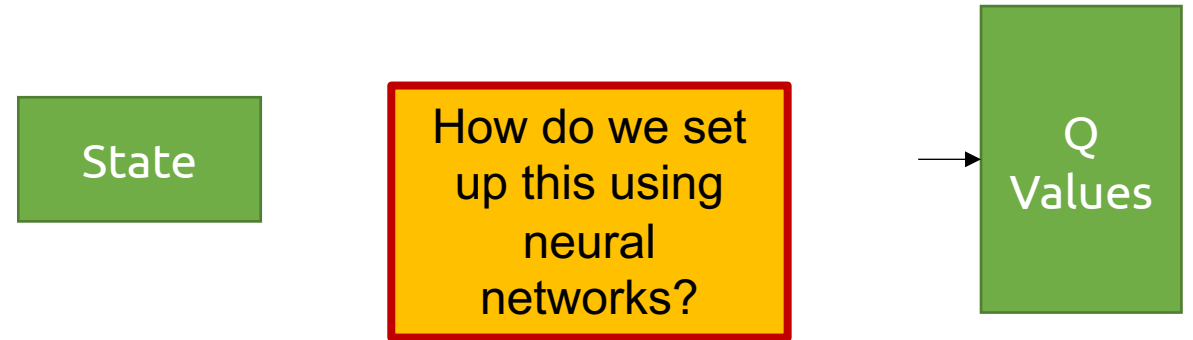
$$Q: (s, a, \theta) \rightarrow \mathbb{R}, \theta \ll |S \times A|$$

Neural Nets as Function Approximators



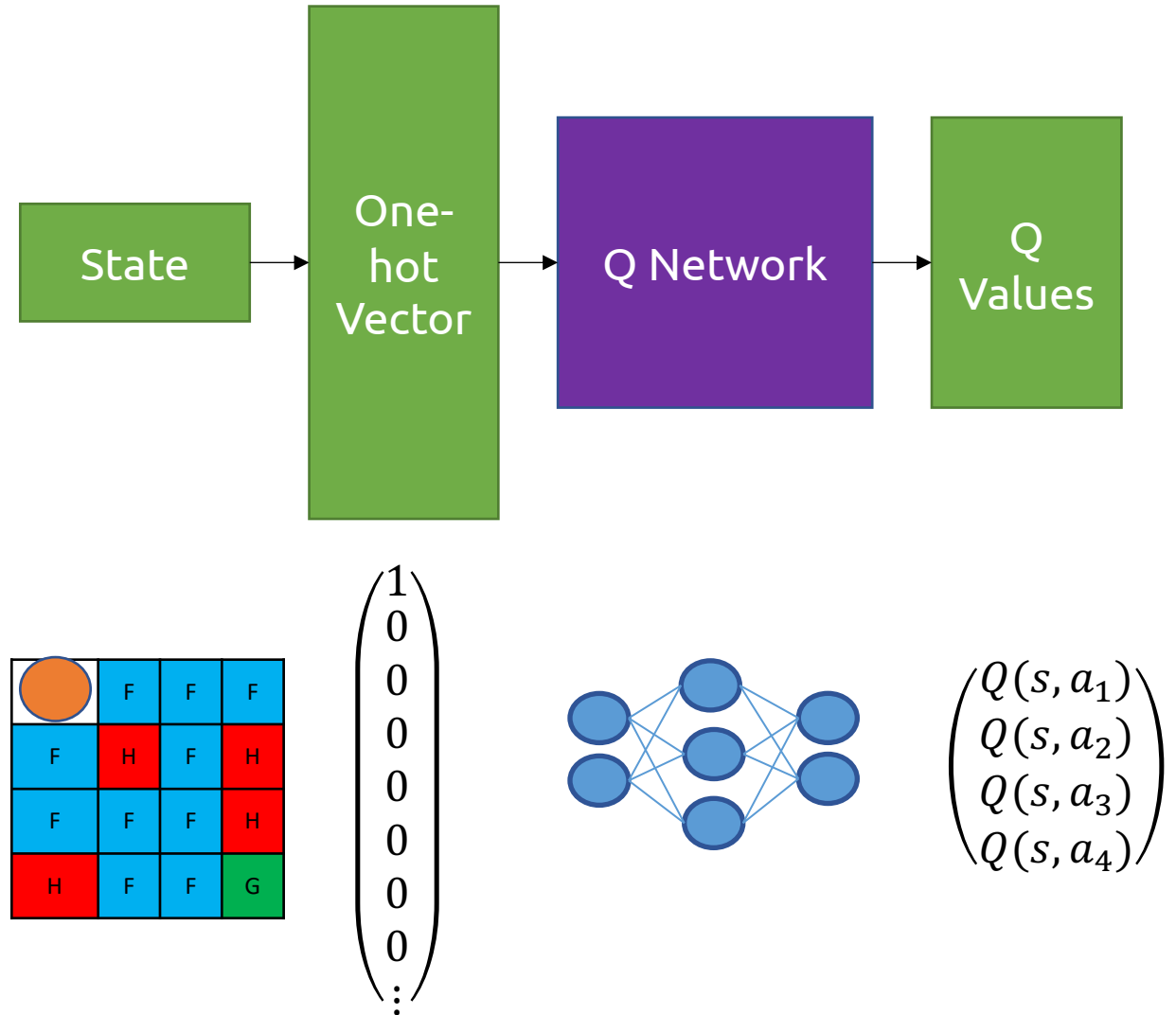
Neural networks are excellent function approximators, so we can use a deep network to learn this parameterized function

Frozen Lake Example



Frozen Lake Example

- Feed a one-hot representation of the state into a neural network to approximate the Q-value for each action
 - One-hot vector of length 16 for frozen lake
- Can also pass in a more complex state to the Q-network rather than a one hot representation.
 - Useful when the number of states is very large, making a one hot representation impractical
 - Or when states are continuous



TF Example Code for Frozen Lake

```
# Weights for Q-Network
Q = tf.Variable(tf.random.uniform([16,4],0,0.01))
# Q-value function
def qVals(inptSt):
    oneH = tf.one_hot(inptSt,16)
    qVals = tf.matmul([oneH],Q)
    return qVals
# argmax over q-values to get estimated best action
action = tf.argmax(qVals(st),1)
```

Atari Example

- 81x81 Images of the game
 - Even if each pixel is just on or off, that's $2^{81 \times 81}$ possible states, way too many for a table
 - They're actually colored, so real scenario is even worse
 - Need a different approach to learn Q and V values for large state spaces like these



<https://www.youtube.com/watch?v=TmPfTpitdgg>

How To Train This Q Network?

The original Q-learning update is not a minimization problem

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha(R(s, a, s') + \gamma V(s'))$$

So how can we transform this into a loss function we can use?

Recap

Tabular Q-learning

Wandering to estimate T and R

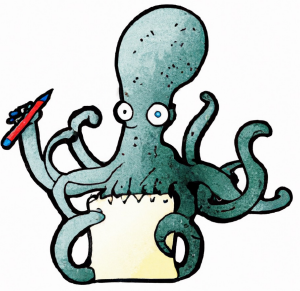
Q-learning: Explore + Improve

Epsilon-Greedy Policies

Limitations of Tabular Q-learning

Neural nets for Q approximation

Tensorflow code



Deep Q-learning

S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	G

$$Q^\pi(s, a) = E[R(s, a, s') + \gamma V^\pi(s')]$$

$$Q^\pi(s, a) = \sum_{s' \in \mathcal{S}} P(s'|s, a) [R(s, a, s') + \gamma V^\pi(s')]]$$

