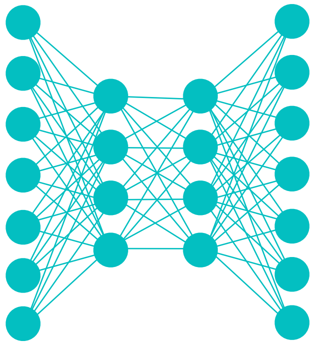
Lecture Notes for

Neural Networks and Machine Learning

Value Iteration, Variants and Tabular Q-Learning



Logistics and Agenda

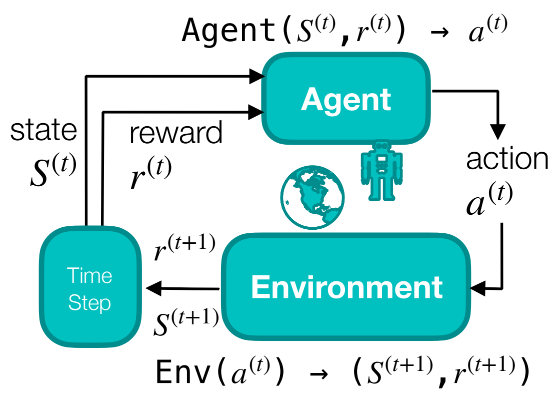
- Logistics
  - Grading Update
  - Final project is ONLY a presentation (no paper)
  - Sign up for a presentation slot ASAP
- Agenda
  - Paper Presentation
  - Markov Building Blocks
  - Value Iteration (and demo)
    - Q-Function Variant
  - Q-Learning
  - Deep Q-Learning (next time)

Paper Presentation

REASONING WITH LATENT THOUGHTS: ON THE POWER OF LOOPED TRANSFORMERS

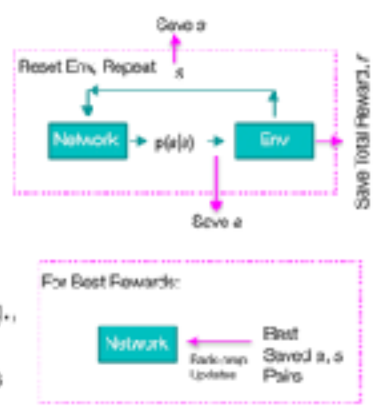
Nikunj Saunshi[1], **Nishanth Dikkala**[1], **Zhiyuan Li**[1,2], **Sanjiv Kumar**[1], **Sashank J. Reddi**[1]
{nsaunshi, nishanthd, lizhiyuan, sanjivk, sashank}@google.com
[1]Google Research, [2]Toyota Technological Institute at Chicago

Last Time



Recall: Frozen lake was not solvable using Cross Entropy Method

- Create a **random neural network**, with output $p(a|s)$
- Let it **interact** with the **environment** (randomly) for set of episodes (e.g., 20)
  - Use network output to sample from possible actions
  - Run episode to completion
  - Repeat
- Calculate reward for each episode
- Keep best episodes (some percentile, e.g., best five)
- For the given best episodes, develop loss function incentivizing the actions taken based upon the input observations

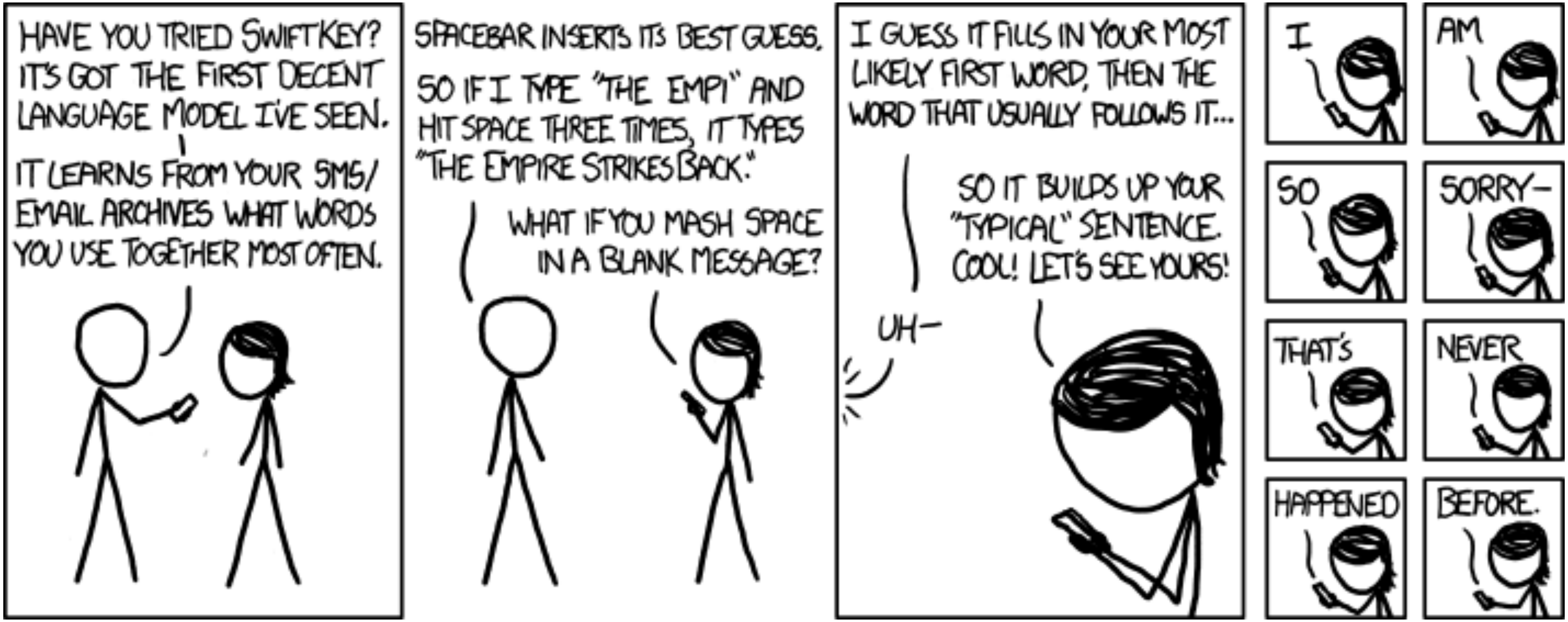


Repeat until desired performance!

- **State:** Every square in grid
- **Action:** Move to make {l,r,u,d}, *with probability*
- **Reward:** Goal, Death
- **Policy:** Given state, where should we move?
- **Optimal Policy:**

$$\pi^* = \arg\max_{\pi} \mathbf{E}\left[\sum_k \gamma^k R_{t+k+1} \,|\, \pi\right]$$

Markov Building Blocks



Markov Processes (MP)

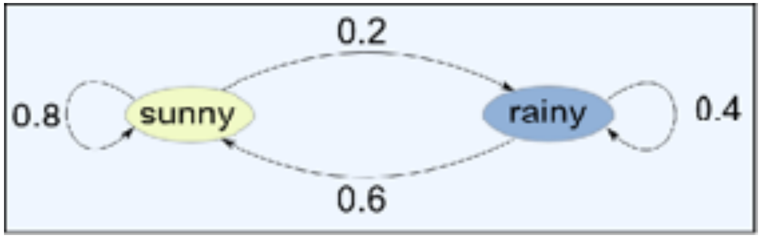
- **Definition**: Any process that can be explained (or simplified) through a sequential set of states that depend only on the previous state
- **Practical Meaning**: For N states, there will be the probability of transition to any other state, encoded through an NxN transition matrix of discrete probabilities
- State sequences are not deterministic, they are sampled from these distributions
- Despite **simplicity**, MP can model a number of real processes with **good enough** precision

Next State, s_{t+1}

Current State, s_t					
	0.1	0.2	0.1	0.6	0.0
	0.9	0.0	0.1	0.0	0.0
	0.0	0.4	0.0	0.4	0.2
	0.0	0.4	0.2	0.0	0.4
	0.0	0.0	0.6	0.0	0.4

MP Example from Maxim Lapan

	Sunny'	Rainy'
Sunny	0.8	0.2
Rainy	0.6	0.4

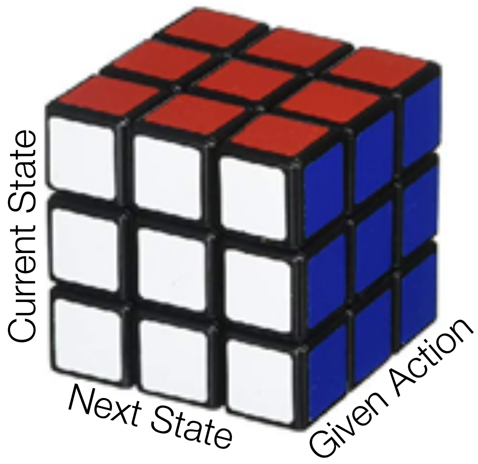


				...	
Sun+Summer					
Rainy+Summer					
Sun+Fall					
Rainy+Fall					
Sun+Else					
Rainy+Else					

Adding One Variable Can Have Drastic Effect on State Space Size

Markov Decision Processes (MDP)

- **New Definition**: any state to state transition can be altered by an action that is given by a Markov Process, so we can alter the MP with discrete actions (decisions)
- **Definition**: An MDP consists of:
  - Env. States, s_t
  - Actions for each time a_t
  - Reward function for each state, $r(s_t)$
  - A transition model, $P(s_{t+1}, s_t \mid a)$ a matrix of probabilities
    - Not ***guaranteed*** next state by given action, probabilistic



Markov Reward Process (MRP)

- **Total reward:** weighted sum of future rewards in sequence

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} \dots = \sum_k \gamma^k r_{t+k+1}$$

- γ defines future reward far- and short-sightedness
  - Common values are **0** (short), **0.9**, **0.99**, and **1** (far)
- G: Want to estimate and maximize this reward!
- This reward calculation, G, can be used to estimate the "**Value**" of each state based upon the average total reward a state *should* give, $V(s) = \mathbf{E}[G \mid s_t{=}s]$
- Typically, this value must be estimated from the model over fixed sequences, otherwise some reward values can become arbitrarily large by looping actions

MDPs and MRPs

- The million dollar question:
  How do we select a good action given a current state?
- **What we did** with Cross Entropy: setup a comparison of different actions we might take (*policy comparison*)
  - Where a *policy* is defined as $\pi(a, s) = P(a_t = a \mid s_t = s)$
  - Given the current state, we have a certain probability of selecting each action
  - Try different policies, select one with best average reward

- **What we will do now**: iteratively interact with environment and get an estimate of $V(s) = \mathbf{E}[G \mid s_t = s]$ called **value iteration**

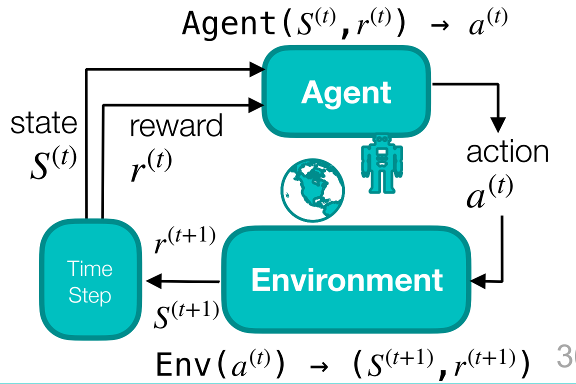
Value Iteration and Q-Learning

When you first start Training with Reinforcement Learning

Value Iteration Overview

0.0	0.0	0.0	0.0	0.0	0.0
$s^{(1)}$	$s^{(2)}$	$s^{(3)}$	… $s^{(i)}$ …		$s^{(N)}$

- Initialize $V(s)$ values to zero
- Interact with environment to estimate rewards from states
- Use $V(s)$ to select the next state (policy from state value)
- Estimate transition probabilities, state→action→next state
- Update $V(s)$ values using recurrence relation
- Keep repeating, updating transition probabilities and $V(s)$ until we get good rewards consistently
- $V(s)$ should follow the Bellman equation, keep updating it until it does
  - So what is this Bellman equation?



State Value Function

- $V(s_t)$ is derived from expected reward, G:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} \ldots = \sum_k \gamma^k r_{t+k+1}$$

- $V(s_t) = \mathbf{E}[G_t \mid s_t{=}s]$, expected Value of state over all future iterations
- **Important**: we can only calculate this exactly if we know:
  - all the rewards for all the states, actions, next states
  - the probabilities of transitioning to a given state from selecting an action
  - likelihood of successful action

We can also define the following recurrence relation:

$$V(s) = \mathbf{E}[G_t \mid s_t = s]$$

$$V(s) = \mathbf{E}[r_{t+1} + \gamma G_{t+1} \mid s_t = s, s_{t+1} = s']$$

$$V(s) = \mathbf{E}[r_{t+1} + \gamma V(s') \mid s_t = s]$$

The Bellman Equation

- For the case when each action is successful and state is discrete, ideal V has property, $a \rightarrow s$:

$$V^{ideal}(s) = \max_{a \in 1...A} (r_{a \rightarrow \hat{s}} + \gamma V(a \rightarrow \hat{s}))$$

current value is immediate reward plus value of next state with highest value because we will choose this next state and will be successful in reaching it

- In general, actions are probabilistic, we need to sum over possible transitions for ideal V, and property becomes:

$$V^{ideal}(s) = \max_{a \in A} \mathbf{E}[r_{s,a,\hat{s}} + \gamma V(\hat{s})] \approx \max_{a \in A} \sum_{\hat{s} \in S} p_{a,s \rightarrow \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma V(\hat{s}))$$

-probabilities of getting to next state x (current value is immediate reward plus value of next state)
-$p_{a,0 \rightarrow s}$ probability of getting to state s from state 0, given that you perform action a

- **Needs:** To select action with best value we need reward matrix, $r_{s,a,\hat{s}}$, action transition matrix $p_{a,s \rightarrow \hat{s}}$ and $V(\hat{s})$

Value Iteration with Bellman Equation

- **Direct**:
  - Initialize $V(s)$ to all zeros
  - Take a series of random steps, then follow policy
  - estimate $p_{a,s\rightarrow\hat{s}}$ via observed **Transitions**
  - Perform value iteration with Bellman Update:
    $$V(s) \leftarrow \max_{a \in A} \sum_{\hat{s} \in S} p_{a,s\rightarrow\hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma V(\hat{s}))$$
  - Repeat until $V(s)$ stops changing, or desired reward reached

With infinite time and exploration, this update will
Converge to Optimal Policy

Value Iteration Reinforcement Learning

M. Lapan Implementation for and Frozen Lake

```
Follow Along:
08a_Basics_Of_Reinforcement_Learning.ipynb
```

```python
from collections import defaultdict as defd
```

versatile dictionary, add keys by assignment

```python
class Val_Agent:
    def __init__(self, env):
        self.env = env
        self.state = self.env.reset()

        self.rewards  = defd(float)
        self.values   = defd(float)

        self.transits = defd(collections.Counter)
```

Index float with "tuple as key"

- Init $r_{s,a,\hat{s}}$ to zeros
- Init $V(s)$ to all zeros
- Init $p_{a,s\rightarrow\hat{s}}$ as counter

dict (key is tuple) containing dictionary of integer counters (key is state)

```python
    def play_n_random_steps(self, count):
        # play this save the observed rewards and actions
        for _ in range(count):
            # randomly sample the space
            action = self.env.action_space.sample()
            new_state, reward, is_done, _ = self.env.step(action)

            # track the reward  and transitions observed
            self.rewards[(self.state, action, new_state)] = reward
            self.transits[(self.state, action)][new_state] += 1

            self.state = self.env.reset() if is_done else new_state
```

Track observed
- rewards
- transition count

```python
def select_action(self, state):
    # for each action, get Value of next state and reward, then choose best
    best_action, best_value = None, None
    for action in range(self.env.action_space.n):
        action_value = self.calc_action_value(state, action)
        # if best action, save
        if best_value is None or best_value < action_value:
            best_value, best_action  = action_value, action
    return best_action
```

$$\boxed{V(s) \leftarrow \max_{a\in A}} \overbrace{\sum_{\hat{s}\in S} p_{a,s\rightarrow\hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma V(\hat{s}))}$$

```python
def calc_action_value(self, state, action):

    s_hat_counts = self.transits[(state, action)] # dict of observed transits
    total = sum(s_hat_counts.values()) # transition denominator

    action_value = 0.0

    # go through each possible target state and calculate value
    for tgt_state, count in s_hat_counts.items():
        # get rewards from playing these steps
        r_sas = self.rewards[(state, action, tgt_state)]

        action_value += (count / total) * (r_sas + GAMMA * self.values[tgt_state])
```

$$\sum_{\hat{s}\in S} \qquad p_{a,s\rightarrow\hat{s}} \qquad \cdot (r_{s,a,\hat{s}} + \gamma \qquad \cdot V(\hat{s}))$$

```python
    return action_value
```

```python
def play_episode(self, render=False):
    total_reward = 0.0
    state = self.env.reset()
    while True:
        # follow our policy based on Value
        action = self.select_action(state)
        new_state, reward, is_done, _ = self.env.step(action)

        self.rewards[(state, action, new_state)] = reward
        self.transits[(state, action)][new_state] += 1
        total_reward += reward

        if is_done:
            break
        state = new_state

    return total_reward
```

Update observations

$$V(s) \leftarrow \max_{a \in A} \sum_{\hat{s} \in S} p_{a,s \rightarrow \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma V(\hat{s}))$$

```python
def value_iteration(self):
    # update all the values according to Bellman equation
    for state in range(self.env.observation_space.n):

        state_values = [self.calc_action_value(state, action)
                        for action in range(self.env.action_space.n)]

        # get the max of this and save it for this state
        self.values[state] = max(state_values)
```

```python
env = gym.make("FrozenLake-v0")
agent = Val_Agent(env)

iter_no = 0
best_reward = 0.0
while True:
    iter_no += 1

    # interact randomly
    agent.play_n_random_steps(100)
    # update matrices
    agent.value_iteration()

    # interact using the value function
    reward = 0.0
    for _ in range(TEST_EPISODES):
        reward += agent.play_episode()
    reward /= TEST_EPISODES
```

```
Best reward updated 0.000 -> 0.050
Best reward updated 0.050 -> 0.250
Best reward updated 0.250 -> 0.300
Best reward updated 0.300 -> 0.500
Best reward updated 0.500 -> 0.600
Best reward updated 0.600 -> 0.700
Best reward updated 0.700 -> 0.800
Best reward updated 0.800 -> 0.850
Solved in 61 iterations!
```

```python
agent.play_episode(render=True)
```

```
  (Left)
SFFF
FHFH
FFFH
HFFG
  (Left)
SFFF
FHFH
FFFH
HFFG
  (Up)
SFFF
FHFH
FFFH
HFFG
  (Up)
SFFF
FHFH
FFFH
HFFG
  (Up)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Up)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Left)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
1.0
```

Value Iteration with Q-function Variant

$$V(s) = \max_{a \in A} \mathbf{E}[r_{s,a,\hat{s}} + \gamma V(\hat{s})] = \max_{a \in A} \sum_{\hat{s} \in S} p_{a,s \to \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma V(\hat{s}))$$

- Define intermediate function Q

$$Q(s,a) = \sum_{\hat{s} \in S} p_{a,s \to \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma V(\hat{s}))$$

- With some nice properties/relations:

$$V(s) = \max_{a \in A} Q(s,a)$$

$$Q(s,a) = \sum_{\hat{s} \in S} p_{a,s \to \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma \max_{\hat{a}} Q(\hat{s},\hat{a}))$$

Value Iteration via q-function

- **Q-Function Variant**:
  - Initialize Q(s,a) to all zeros
  - Take a series of random steps, then follow policy
  - estimate $p_{a,s\rightarrow\hat{s}}$ via observed **Transitions**
  - Perform value iteration with Bellman Update:

$$Q(s,a) \leftarrow \sum_{\hat{s}\in S} p_{a,s\rightarrow\hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma \max_{a'} Q(\hat{s},a'))$$

  - Repeat until Q is not changing

This is an **exact equivalent** to value iteration, but using a **slightly different tracking function** for value

Q-Function Iterati… Reinforcement Le…

```
Best reward updated 0.000 -> 0.350
Best reward updated 0.350 -> 0.400
Best reward updated 0.400 -> 0.450
Best reward updated 0.450 -> 0.650
Best reward updated 0.650 -> 0.800
Best reward updated 0.800 -> 0.850
Solved in 22 iterations!
```

```
def q_value_iteration(self):

  for state in range(self.env.observation_space.n):
    for action in range(self.env.action_space.n):

      s_hat_count = self.transits[(state, action)]
      total = sum(s_hat_count.values())

      q_action_value = 0.0
      for tgt_state, count in s_hat_count.items():
        r = self.rewards[(state, action, tgt_state)]
        p = (count / total)

        best_action = self.select_action(tgt_state)

        q_action_value += p * (r + GAMMA * self.q_values[(tgt_state, best_action)])
        self.q_values[(state, action)] = q_action_value
```

Loop over state and action space

Get counts for all next states

new: need best action for $\max_{a'} Q(\hat{s}, a')$

new: save Q at state / action pair

$$Q(s,a) \leftarrow \sum_{\hat{s}\in S} p_{a,s\rightarrow\hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma \max_{a'} Q(\hat{s}, a'))$$

Follow Along: 08a_Basics_Of_Reinforcement_Learning.ipynb

Tabular Q-Learning

Value Iteration Limitations

- Q and V can get really big for **large states and action spaces**
- Transition matrix, $p_{a,s \rightarrow \hat{s}}$, can get **gigantic** for large state and action spaces (and potentially intractable)
  - We will solve this by dropping the transition probabilities in Q function update
  - Helps make computation tractable, but optimization harder
- This **Variant** is known as Q-Learning
- (*not addressing yet…*) Q-table needs infinite inputs when the state spaces are **continuous**
  - We will eventually solve this by using a **neural network** to **approximate** the Q function
  - Q function already has the transitions simplified, so this is already in a good form for learning from NN

Creating a computable Q approximation

- Assume Q function can incorporate this

$$Q^{old}(s,a) = \sum_{\hat{s}\in S} p_{a,s\rightarrow\hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma \max_{a'} Q^{old}(\hat{s},a'))$$

$$Q^{old}(s,a) = \sum_{\hat{s}\in S} p_{a,s\rightarrow\hat{s}} \cdot r_{s,a,\hat{s}} + \gamma \max_{a'} Q^{old}(\hat{s},a') \cdot p_{a,s\rightarrow\hat{s}}$$

$$Q^{old}(s,a) = \sum_{\hat{s}\in S} p_{a,s\rightarrow\hat{s}} \cdot r_{s,a,\hat{s}} + \gamma \max_{a'} Q^{new}(\hat{s},a')$$

Leftmost term actually just sums over $\hat{s}$ so that reward has no dependence

$$Q^{old}(s,a) = r_{s,a} + \sum_{\hat{s}\in S} \gamma \max_{a'} Q^{new}(\hat{s},a')$$

What happens if our state space is absolutely gigantic. Summing over all possible states seems like a bad idea.

$$Q^{new}(s,a) = r_{s,a} + \gamma \max_{a'} Q^{new}(s',a')$$

Can we approximate it more simply?

We now call this the **Bellman Approximation**

Tabular Q-Learning Algorithm

- In update, **ignore the transition probability**, making use of the iterative nature of Q, Bellman Update:

$$Q(s_t, a_t) = r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 \ldots$$
$$Q(s_t, a_t) = r_0 + \gamma (r_1 + \gamma^2 r_2 + \gamma^3 r_3 \ldots)$$
$$Q(s_t, a_t) = r_0 + \gamma \max_a Q(s_{t+1}, a)$$

$$Q^{old}(s, a) \leftarrow \sum_{\hat{s} \in S} p_{a, s \rightarrow \hat{s}} \cdot (r_{s,a,\hat{s}} + \gamma \max_{a'} Q^{old}(\hat{s}, a'))$$

$$Q^{new}(s, a) \leftarrow r_{s,a} + \gamma \max_{a' \in A} Q^{new}(s', a')$$

- For stability, add momentum to the **Bellman approximation update** equation

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

- Algorithm, start with empty $Q(s,a)$:
  - Sample (with rand) from environment, (s, a, r, s')
  - Make Bellman Update with Momentum (eq. above)
  - Repeat until desired performance

Tabular Q-Learning Reinforcement Learning

M. Lapan Implementation for and Frozen Lake

```
Follow Along:
08a_Basics_Of_Reinforcement_Learning.ipynb
```

```python
from collections import defaultdict as defd

class QLearningAgent:
    def __init__(self,env):
        self.env = env
        self.state = self.env.reset()
        self.q_vals = defd(float)

    def sample_env(self):
        … play environment randomly for one step …
        return (old_state, action, reward, new_state)

    def best_value_and_action(self, state):
        # Go through Q(s,a), get Q(s',a'), best action, a'
        best_value, best_action = None, None
        for action in range(self.env.action_space.n):
            action_value = self.q_vals[(state, action)]
            if best_value is None or best_value < action_value:
                best_value, best_action = action_value, action

        return best_value, best_action

    def value_update(self, s, a, r, next_s):
        # update from one observation
        best_v, _ = self.best_value_and_action(next_s)
        new_Q = r + GAMMA * best_v
        old_Q = self.q_vals[(s, a)]
        self.q_vals[(s, a)] = old_Q * (1-ALPHA) + new_Q * ALPHA
```

Only save the $Q(s,a)$ as tuple dictionary

No need to track transitions and per state / action rewards

$$V(s) = \max_{a \in A} Q(s,a)$$

Update after each step
No other tracking…

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

```python
test_env = gym.make("FrozenLake-v0")
train_env = gym.make("FrozenLake-v0")
agent = QLearningAgent(train_env)

iter_no = 0
best_reward = 0.0
while True:
    iter_no += 1
    # sample one step
    s, a, r, next_s = agent.sample_env()

    # update Q
    agent.value_update(s, a, r, next_s)

    # test how well it works
    reward = 0.0
    for _ in range(TEST_EPISODES):
        reward += agent.play_episode(test_env)

    reward /= TEST_EPISODES
    if reward > 0.80:
        print("Solved in %d iterations!" % iter_no)
        break
```

```
Best reward updated 0.000 -> 0.300
Best reward updated 0.300 -> 0.350
Best reward updated 0.350 -> 0.400
Best reward updated 0.400 -> 0.450
Best reward updated 0.450 -> 0.500
Best reward updated 0.500 -> 0.600
Best reward updated 0.600 -> 0.650
Best reward updated 0.650 -> 0.700
Best reward updated 0.700 -> 0.750
Best reward updated 0.750 -> 0.800
Best reward updated 0.800 -> 0.850
Solved in 10103 iterations!
```

Conclusion:

- It still works, but wow it takes much longer to converge!!!
- Placing so much emphasis on the Q-function (to learn all variability) makes the optimization more difficult
- Update to Q noisy (approximation)

Next Time

- How can we use learning even in gigantic dimensional state space?
- Deep Q-Learning …

Lecture Notes for

Neural Networks and Machine Learning

Value Iteration

Next Time:
DeepQ-Learning

Reading: None