Pre-requisites:

Last week's videos lectures and homework assignment.

Recap:

Thanks to Rayan Wong for summarizing it for us. (https://medium.com/@taggatle/01-reinforcement-learning-move-37-introduction-c3449dac2d54)

Now that we are familiar with the basics, it's time to move forward. Let us revisit the same problem we discussed last week. Robotic vacuum cleaner famously known as Roomba is a machine that cleans the floor. Roomba needs to start, clean, avoid obstacles and find the charging station. These 4 states describe the possible positions of the robot and the action describes the direction of motion. The robot can move left/right/up/down. The first (Battery Full) and the final (Charging) states are the terminal states. The goal is to find an optimal policy that maximizes the return from any initial states.

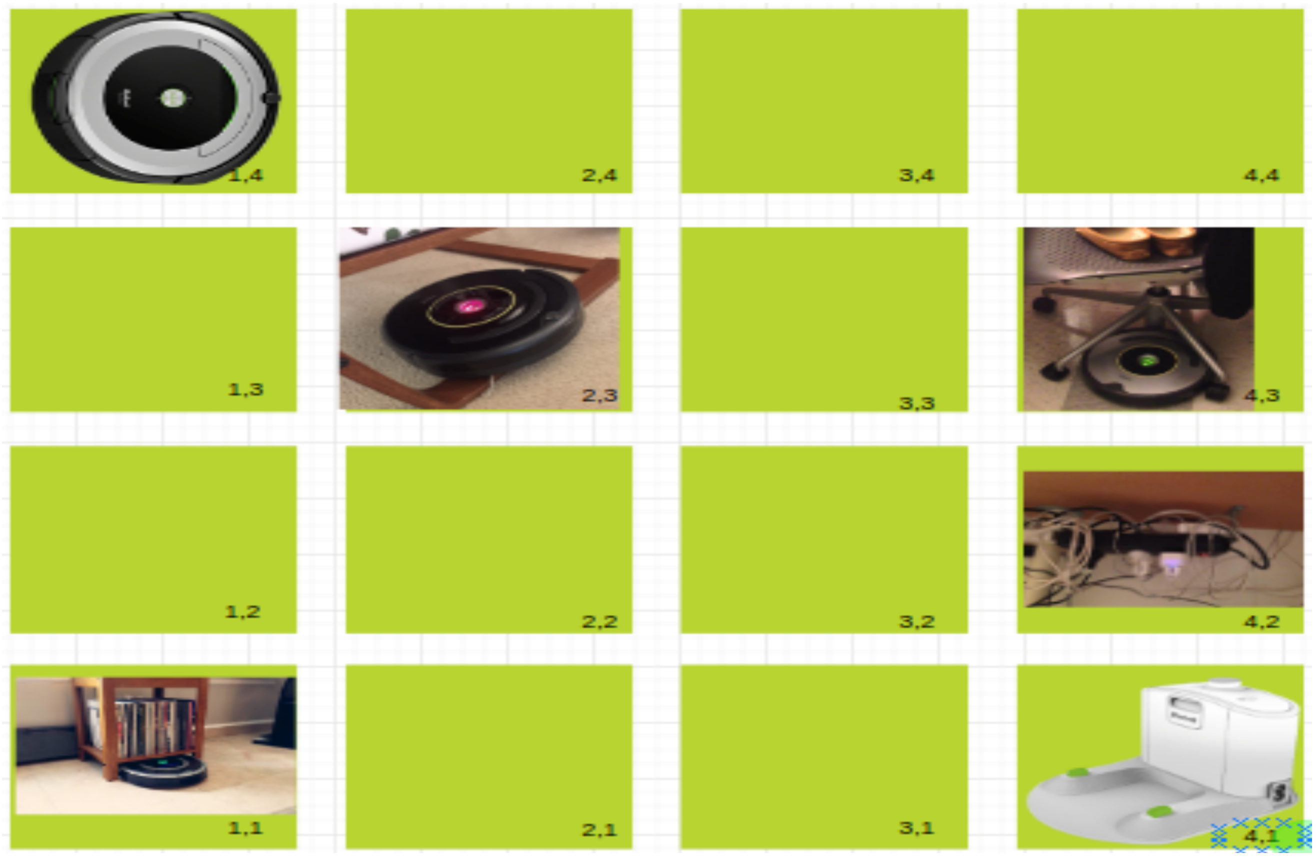


Fig 1

Above figure shows us the map of possible states that the Roomba can navigate through in form of a grid. For each step, Roomba gets a reward of 0 except when it reaches the goal to get a reward value of 1.

Let us use OpenAI gym and try to solve this problem. FrozenLake-v0 environment will be a best fit for this problem. The reward and next state models are stochastic p (r_t+1 | s_t , a_t), p(s_t+1 | s_t , a_t) for this environment. To solve this problem, we should find an optimal policy that can reach the goal more than 70% of the times. (Please try to come up with a solution before looking at the solutions)

Let's pull our sleeves and try a brute force method:

We got 16 states and 4 possible moves that gives us 4^16 = 4294967296 possible policies to choose from. It is a computationally intensive task to evaluate all of them so we are going to choose few cases randomly and select the best among them.

```
import numpy
import time
import gym
"""
      Args:
         policy: [S, A] shaped matrix representing the policy.
         env: OpenAI gym env.
         render: boolean to turn rendering on/off.
"""
#Execution
def execute(env, policy, episodeLength=100, render=False):
   totalReward = 0
   start = env.reset()
   for t in range(episodeLength):
      if render:
         env.render()
      action = policy[start]
      start, reward, done, _ = env.step(action)
      totalReward += reward
      if done:
         break
   return totalReward

#Evaluation
def evaluatePolicy(env, policy, n_episodes=100):
   totalReward = 0.0
   for _ in range(n_episodes):
      totalReward += execute(env, policy)
   return totalReward / n_episodes

#Function for a random policy
def gen_random_policy():
   return numpy.random.choice(4, size=((16)))

```

```
if __name__ == '__main__':
    env = gym.make('FrozenLake-v0')
    ## Policy search
    n_policies = 1000
    startTime = time.time()
    policy_set = [gen_random_policy() for _ in range(n_policies)]
    policy_score = [evaluatePolicy(env, p) for p in policy_set]
    endTime = time.time()
    print("Best score = %0.2f. Time taken = %4.4f seconds" %(numpy.max(policy_score) ,
  endTime - startTime))
```

Output for the above script:

<u>**Best score = 0.44. Time taken = 10.2633 seconds**</u>

The above script searches the environment for best policy in a random set of 1000 solutions and evaluates them. The best policy score we get is 0.44 in 10.2633 seconds. It means that the chance of agent reaching the goal is 44%. It is not even close to our goal. Random search does not work well for complex problems where the search space is huge.

How can we get this better??? Let's Dive Deeper...

The goal of the Roomba is to pick the best policy that will maximize the total rewards received from the environment.
t_0: Roomba observes the environment state s_0 and picks an action a_0, then upon performing its action, environment state becomes s_1 and the Roomba receives a reward r_1.

t_1: Roomba observes current state s_1 and picks an action a_1, then upon acting its action, environment state becomes s_2 and it receives a reward r_2.

t_2: Roomba observes current state s_2 and picks an action a_2, then upon acting its action, environment state becomes s_3 and it receives a reward r_3.

Thus the total reward received by the Roomba in response to the actions selected by its policy is going to be:

Total reward = r_1 + r_2 + r_3 + r_4 + r_5 + ...

However, it is common to use a discount factor to give higher weight if the rewards are nearby and lower weight if the rewards are far in the future.

Total discounted reward = r_1 + γ r_2 + γ^2 r_3 + γ^3 r_4 + γ^4 r_5+ …

$$total\ discounted\ reward = \sum_{i=1}^{T} \gamma^{i-1} r_i$$

**T is the length of the episode. It can be infinity if there is huge length for the episode.*

Why discount factor???

The idea of using discount factor is to prevent the total reward from going to infinity (because 0 <= γ <= 1). It also models the agent behavior when the agent prefers immediate rewards than rewards that are potentially received later in the future.

In our example Roomba can take two paths to reach goal state; one of them is longer but gives higher reward and there is also a shorter path with smaller reward. We need the Roomba to cover and clean maximum distance before it reaches the goal. By adjusting the γ value we can control which path Roomba should prefer.

Value Function:

This term is very popular in reinforcement learning. It's denoted as **V(s)** and this represents how good a state is for an agent to be in. If an agent uses a given policy **π** to select actions, then the corresponding value function is given by:

$$V^{\pi}(s) = \mathbb{E}[\sum_{i=1}^{T} \gamma^{i-1} r_i] \quad \forall s \in \mathbb{S}$$

*∀ - *Universal Quantification* (https://en.wikipedia.org/wiki/Universal_quantification)

Amongst all the possible value functions there will be an optimal value function that has higher value than other functions for all states.

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \quad \forall s \in \mathbb{S}$$

The optimal policy **π*** is the policy that corresponds to the optimal value function.

$$\pi^{*} = \arg\max_{\pi} V^{\pi}(s) \quad \forall s \in \mathbb{S}$$

Q Function:

Q is a function of a state-action pair and returns real value. **" Q : S x A --> R ".** An optimal Q-function **Q*(s,a)** means the expected total reward received by an agent starting in a state **s** and picks an action **a** will then move optimally forward. This can also be put as the indication for how good it is for an agent to pick action **a** while being in state **s.**

Since **V*(s)** is the maximum expected total reward when starting from state a, it will be the maximum of Q*(s,a) over all possible actions.

$$V^{*}(s) = \max_{a} Q^{*}(s, a) \quad \forall s \in \mathbb{S}$$

When a known optimal Q-function Q*(s,a) exists, the optimal policy can be attained by choosing the action a that declares the maximum Q*(s,a) for a particular state *s*.

$$\pi^*(s) = \arg\max_a Q^*(s,a) \quad \forall s \in \mathbb{S}$$

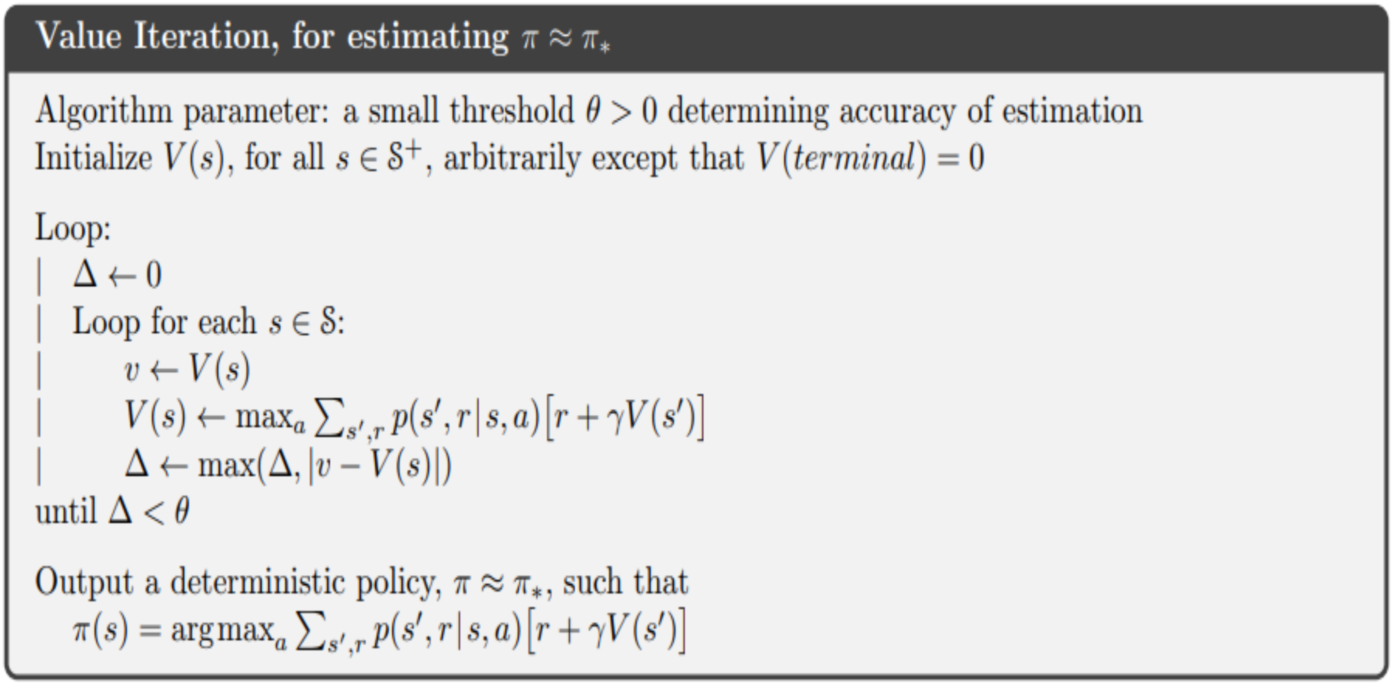
Bellman equation using dynamic programming gives a recursive definition for the optimal Q-function. The Q*(s,a) equals the summation of immediate reward after performing action *a* while in state *s* and the discounted expected future reward after transition to a next state *s'* .

$$Q^*(s,a) = R(s,a) + \gamma\, \mathbb{E}_{s'}[V^*(s')]$$
$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in \mathbb{S}} p(s'|s,a) V^*(s')$$
$$\text{Since,}$$
$$V^*(S) = \max_a Q^*(s,a)$$
$$V^*(S) = \max_a \left[R(s,a) + \gamma \sum_{s' \in \mathbb{S}} p(s'|s,a) V^*(s') \right]$$

***please refer to last week's portion for more details.*

Value Iteration:

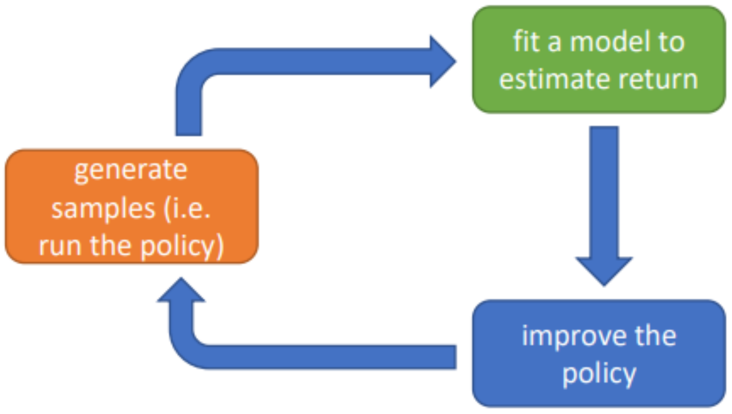
Value Iteration computes the optimal state value function by iteratively improving the estimate of **V(s).** The algorithm initializes **V(s)** to arbitrary random values then it repeatedly updates the **Q(s,a)** and **V(s)** values until they converge. Value iteration is guaranteed to converge to the optimal values.



```
Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
|  $\Delta \leftarrow 0$
|  Loop for each $s \in \mathcal{S}$:
|      $v \leftarrow V(s)$
|      $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
|      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
   $\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
```

Fig 2. *credits: Richard S. Sutton and Andrew G. Barto*

$$Q^{\pi}(\mathbf{s},\mathbf{a}) \leftarrow r(\mathbf{s},\mathbf{a}) + \gamma E_{\mathbf{s}'\sim p(\mathbf{s}'|\mathbf{s},\mathbf{a})}[V^{\pi}(\mathbf{s}')]$$


fit a model to estimate return
improve the policy
generate samples (i.e. run the policy)

$$V^{\pi}(\mathbf{s}) \leftarrow \max_{\mathbf{a}} Q^{\pi}(\mathbf{s},\mathbf{a})$$

*Fig 3. credits: Prof.*Sergey Levine

Fun part now: *let's take the above problem and try to solve it using Value Iteration method.*

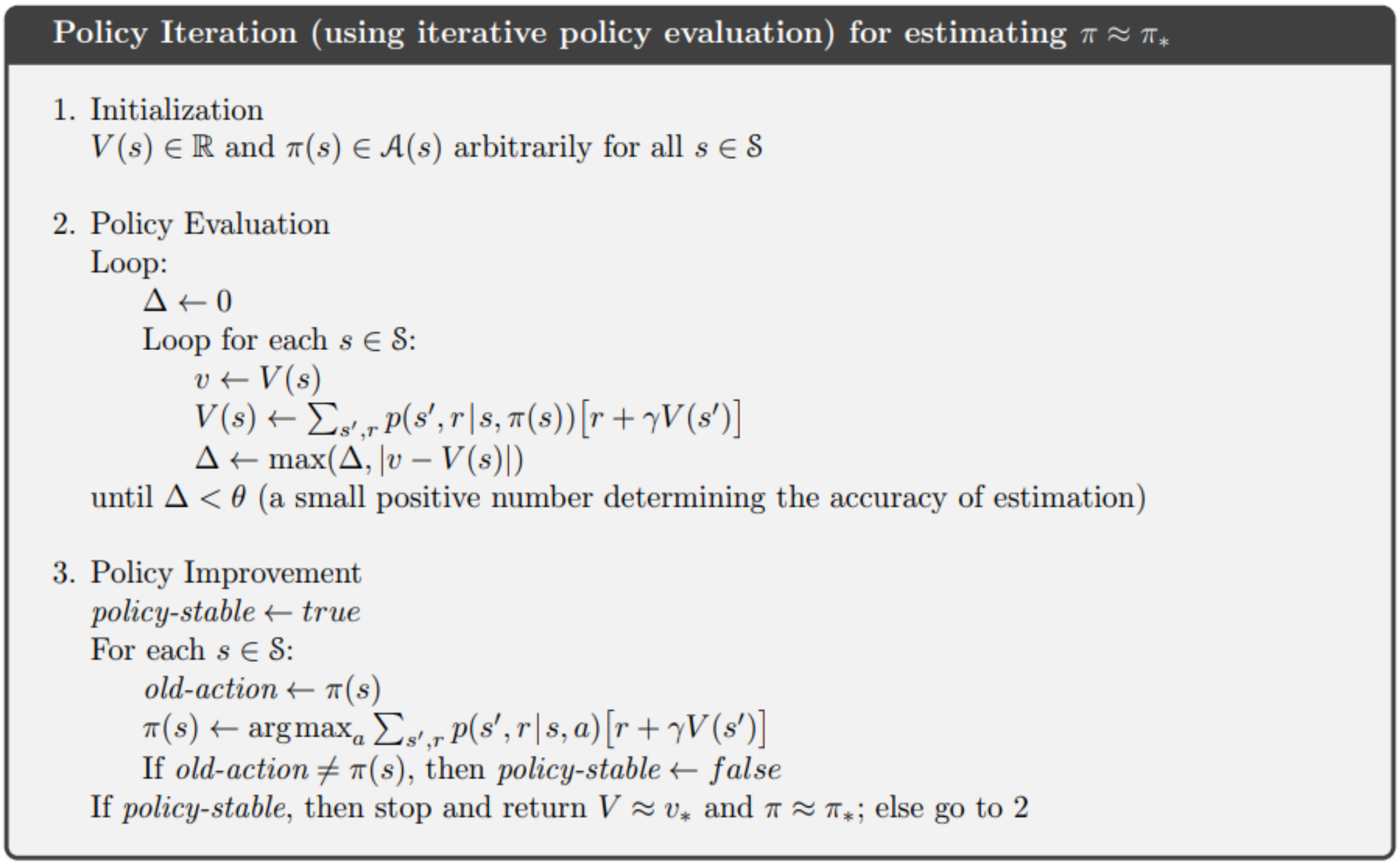
```python
import numpy
import gym
import time
"""
        Args:
            policy: [S, A] shaped matrix representing the policy.
            env: OpenAI gym env.
                env.P represents the transition probabilities of the environment.
                env.P[s][a] is a list of transition tuples (prob, next_state, reward, done).
                env.nS is a number of states in the environment.
                env.nA is a number of actions in the environment.
            gamma: Gamma discount factor.
            render: boolean to turn rendering on/off.
"""
def execute(env, policy, gamma=1.0, render=False):
    start = env.reset()
    totalReward = 0
    stepIndex = 0
    while True:
        if render:
            env.render()
        start, reward, done, _ = env.step(int(policy[start]))
        totalReward += (gamma ** stepIndex * reward)
        stepIndex += 1
        if done:
            break
    return totalReward

# Evaluates a policy by running it n times.returns:average total reward
def evaluatePolicy(env, policy, gamma=1.0, n=100):
    scores = [
        execute(env, policy, gamma=gamma, render=False)
        for _ in range(n)]
    return numpy.mean(scores)

# choosing the policy given a value-function
def calculatePolicy(v, gamma=1.0):
    policy = numpy.zeros(env.env.nS)
    for s in range(env.env.nS):
        q_sa = numpy.zeros(env.action_space.n)
        for a in range(env.action_space.n):
            for next_sr in env.env.P[s][a]:
                # next_sr is a tuple of (probability, next state, reward, done)
                p, s_, r, _ = next_sr
                q_sa[a] += (p * (r + gamma * v[s_]))
        policy[s] = numpy.argmax(q_sa)
    return policy

# Value Iteration Agorithm
def valueIteration(env, gamma=1.0):
    value = numpy.zeros(env.env.nS)  # initialize value-function
    max_iterations = 10000
    eps = 1e-20
    for i in range(max_iterations):
        prev_v = numpy.copy(value)
        for s in range(env.env.nS):
            q_sa = [sum([p * (r + prev_v[s_]) for p, s_, r, _ in env.env.P[s][a]]) for a in range(env.env.nA)]
            value[s] = max(q_sa)
        if (numpy.sum(numpy.fabs(prev_v - value)) <= eps):
            print('Value-iteration converged at # %d.' % (i + 1))
            break
    return value

if __name__ == '__main__':
    gamma = 1.0
    env = gym.make("FrozenLake-v0")
    optimalValue = valueIteration(env, gamma);
    startTime = time.time()
    policy = calculatePolicy(optimalValue, gamma)
    policy_score = evaluatePolicy(env, policy, gamma, n=1000)
    endTime = time.time()
    print("Best score = %0.2f. Time taken = %4.4f seconds" % (numpy.mean(policy_score), endTime - startTime))
```

Output:

```
Value-iteration converged at # 1373.
Best score = 0.76. Time taken = 0.3675 seconds.
```

Policy Iteration:

The value-iteration algorithm keeps improving the value function at each iteration, until the value-function converges. Since the agent only cares about the finding the optimal policy, sometimes the optimal policy will converge before the value function. Therefore, we have another algorithm called policy-iteration. Instead of repeatedly improving the value-function estimate, it will re-define the policy at each step and compute the value according to this new policy until the policy converges. Policy iteration is also guaranteed to converge to the optimal policy and it often takes less iterations to converge than the value-iteration algorithm.



```
Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
       $\Delta \leftarrow 0$
       Loop for each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   *policy-stable* $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
       *old-action* $\leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
       If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

Fig 4. *credits: Richard S. Sutton and Andrew G. Barto*

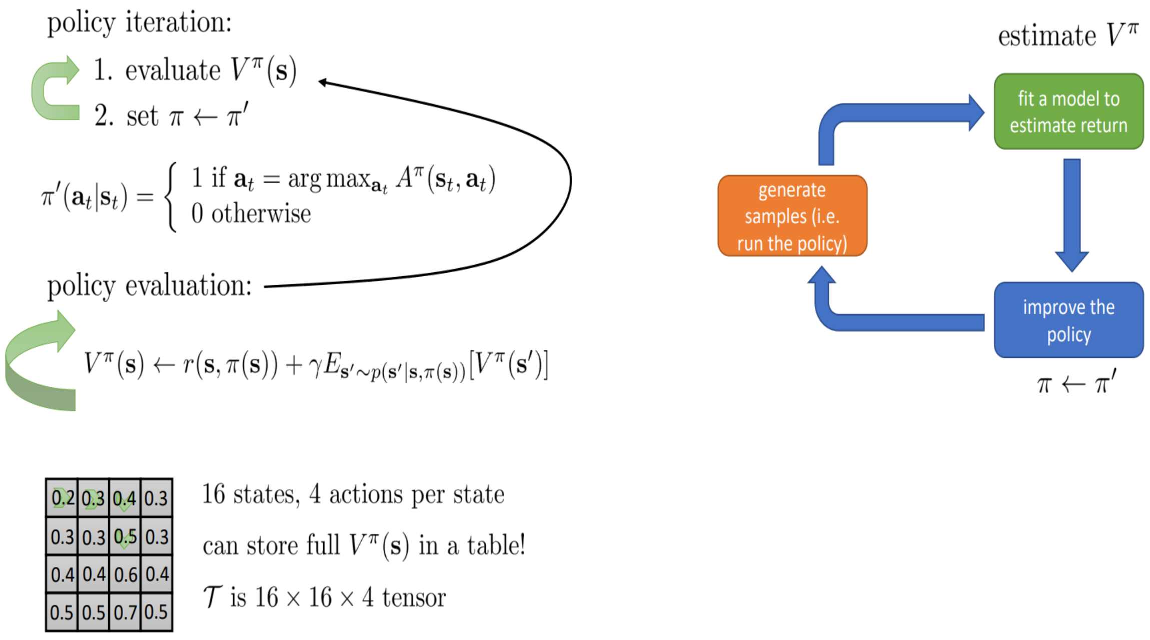

policy iteration:
1. evaluate $V^\pi(\mathbf{s})$
2. set $\pi \leftarrow \pi'$
$\pi'(\mathbf{a}_t|\mathbf{s}_t) = \begin{cases} 1 \text{ if } \mathbf{a}_t = \arg\max_{\mathbf{a}_t} A^\pi(\mathbf{s}_t,\mathbf{a}_t) \\ 0 \text{ otherwise} \end{cases}$
policy evaluation:
$V^\pi(\mathbf{s}) \leftarrow r(\mathbf{s},\pi(\mathbf{s})) + \gamma E_{\mathbf{s}'\sim p(\mathbf{s}'|\mathbf{s},\pi(\mathbf{s}))}[V^\pi(\mathbf{s}')]$
estimate V^π
fit a model to estimate return
generate samples (i.e. run the policy)
improve the policy
$\pi \leftarrow \pi'$
0.2 0.3 0.4 0.3
0.3 0.3 0.5 0.3
0.4 0.4 0.6 0.4
0.5 0.5 0.7 0.5
16 states, 4 actions per state
can store full $V^\pi(\mathbf{s})$ in a table!
$\mathcal{T}$ is $16 \times 16 \times 4$ tensor

*Fig 5. credits: Prof.*Sergey Levine

Fun part again: *let's take the same problem and try to solve it using Policy Iteration method.*

```
import numpy
```

```python
import gym
import time
"""
    Args:
        policy: [S, A] shaped matrix representing the policy.
        env: OpenAI gym env.
            env.P represents the transition probabilities of the environment.
            env.P[s][a] is a list of transition tuples (prob, next_state, reward, done).
            env.nS is a number of states in the environment.
            env.nA is a number of actions in the environment.
        gamma: Gamma discount factor.
        render: boolean to turn rendering on/off.
"""
#executes an episode
def execute(env, policy, gamma = 1.0, render = False):
    start = env.reset()
    totalReward = 0
    stepIndex = 0
    while True:
        if render:
            env.render()
        start, reward, done , _ = env.step(int(policy[start]))
        totalReward += (gamma ** stepIndex * reward)
        stepIndex += 1
        if done:
            break
    return totalReward

def evaluatePolicy(env, policy, gamma = 1.0, n = 100):
    scores = [execute(env, policy, gamma, False) for _ in range(n)]
    return numpy.mean(scores)

#Extract the policy given a value-function
def extractPolicy(v, gamma = 1.0):
    policy = numpy.zeros(env.env.nS)
    for s in range(env.env.nS):
        q_sa = numpy.zeros(env.env.nA)
        for a in range(env.env.nA):
            q_sa[a] = sum([p * (r + gamma * v[s_]) for p, s_, r, _ in  env.env.P[s][a]])
        policy[s] = numpy.argmax(q_sa)
    return policy

#Iteratively calculates the value-function under policy.
def CalcPolicyValue(env, policy, gamma=1.0):
    value = numpy.zeros(env.env.nS)
    eps = 1e-10
    while True:
        previousValue = numpy.copy(value)
        for states in range(env.env.nS):
            policy_a = policy[states]
            value[states] = sum([p * (r + gamma * previousValue[s_]) for p, s_, r, _ in env.env.P[states][policy_a]])
        if (numpy.sum((numpy.fabs(previousValue - value))) <= eps):
            # value converged
            break
    return value

#PolicyIteration algorithm
def policyIteration(env, gamma = 1.0):
    policy = numpy.random.choice(env.env.nA, size=(env.env.nS))  # initialize a random policy
    maxIterations = 1000
    gamma = 1.0
    for i in range(maxIterations):
        oldPolicyValue = CalcPolicyValue(env, policy, gamma)
        newPolicy = extractPolicy(oldPolicyValue, gamma)
        if (numpy.all(policy == newPolicy)):
            print ('Policy Iteration converged at %d.' %(i+1))
            break
        policy = newPolicy
    return policy

if __name__ == '__main__':
```

```python
env_name  = 'FrozenLake-v0'
env = gym.make(env_name)
start = time.time()
optimalPolicy = policyIteration(env, gamma = 1.0)
scores = evaluatePolicy(env, optimalPolicy, gamma = 1.0)
end = time.time()
print("Best score = %0.2f. Time taken = %4.4f seconds" %(numpy.max(scores) , end - start))
```

Output:

```
Policy-Iteration converged at step 3.
Best score = 0.78. Time taken = 0.1491 seconds.
```

Value-Iteration vs Policy-Iteration:
Both value-iteration and policy-iteration algorithms can be used for *offline planning* where the agent is assumed to have prior knowledge about the effects of its actions on the environment (they assume the MDP model is known). Comparing each other, policy-iteration is computationally efficient as it often takes considerably fewer number of iterations to converge although each iteration is more computationally expensive.

Let us make it better next week!!!

Credits:

http://learning.mpi-sws.org/mlss2016/slides/2016-MLSS-RL.pdf

http://incompleteideas.net/book/bookdraft2018jan1.pdf

http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-7.pdf

https://gym.openai.com/evaluations/eval_4VyQBhXMRLmG9y9MQA5ePA

http://web.cs.ucla.edu/~malzantot/

http://www.planetb.ca/syntax-highlight-word