An illustrated Overview of Reinforcement Learning

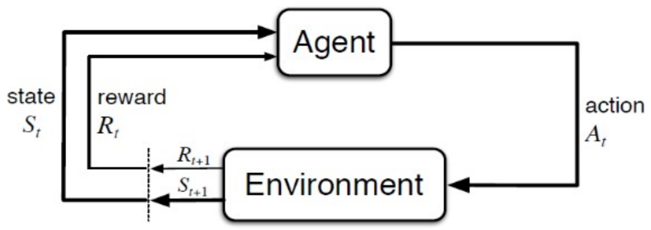
Deepak Babu P R,Sr. Data Scientist

Prof. Shalabh Bhatnagar, IISc - Indian Institute of Science.

Reinforcement Learning (RL) is a learning paradigm different from traditional machine learning (supervised and unsupervised). The learning problem considered here mimics humans learning from interactions using trial-and-error and has historically been used in the context of planning/decision making related problems like robotics and autonmous driving. We can apply RL in cases where there is a need to make sequential decisions to optimize a metric. Let's consider an example of news recommendation problem. Users are presented list of articles that are of potential interest to them and user decides to either click(and read) vs. not-click. How do we(called agent) organize or rank the articles to maximize click-thru rates(CTR)? Is there a strategy(called policy in RL terminology) that leads to better relevance and hence improve CTR ? Using supervised learning for this problem would attempt to learn the P(click|article) independently for all articles. With RL, at each position we are trying to learn P(click|article1, article2, article3) considering the sequence of articles already seen by the user. We are maximizing for total future cumulative rewards(clicks) that we expect to recieve by recommending this article given the fact that the user has already seen article 1,2 and 3. Intuitively, it makes sense to consider the sequence of articles already seen because 100s of articles might be relevant to a user across various topics and if we end up showing articles from one topic we might risk user getting fatigue and quiting the page. Instead, RL tries to arrive at the optimal policy that balances diversity in topics that are of interest to the user maximizing potential future clicks from that user

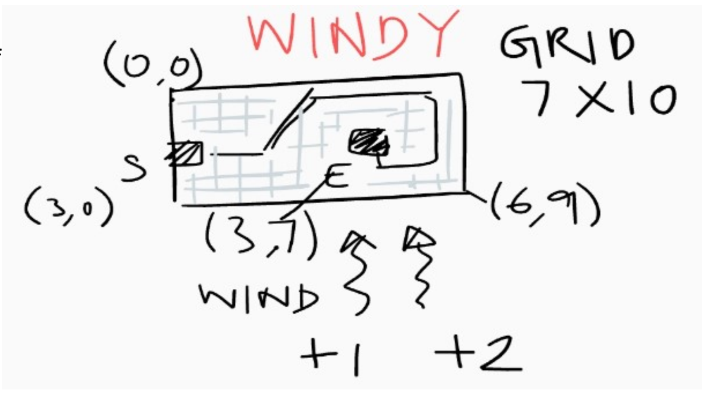
Today, RL has been used in a couple of industries with varied success and hasnt yet become mainstream technique yet. RL has been applied to game playing with good success, one of the early wins being game of AlphaGo (https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf) in which a RL agent beats expert human player. Consumer Internet, Quantitative Trading, Resource management, Advertising and Manufacturing are other areas where RL has been successfully applied. One of the reason RL has been extensively applied to game playing is because of our ability to collect lots of data through simulation of the game. With real-world problems, the data collection is far more expensive and in some cases infeasible which is one of the reasons preventing wide-spread adoption of RL.

Next, we consider the mathematical representation of Reinforcement Learning problem called Markov Decision Process(MDP).MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. Thus MDPs involve delayed reward and the need to tradeoff immediate and delayed reward.An MDP involves following components



- **Agent** is learner and decision maker.
- **Environment** comprises everything the agent interacts with.
- Everytime agent interacts with environment by taking **actions**, the environment responds by changing its **state**
- For every action taken, the agent recieves a scalar **reward** indicating goodness of action taken. Reward can be instantaneous or delayed.
  The MDP and agent together thereby give rise to a sequence or trajectory that begins like this
  S0,A0,R1, S1,A1,R2, S2,A2,R3, . . .
  Below are some important considerations while designing a RL problem. Evaluating each of these questions will narrow down the solution framework for the problem.
  - What are the states and actions in my env ?
  - Are my interactions with environment continous and infinite or Is there a natural end state for interactions? continous vs. episodic | discounted vs. non-discounted
  - Are there infinitely many (large number) of states ? Tabular vs. Approximate methods
  - Do we know the system model ? or Do we know how my environment behaves to actions in different states ?> model-based(DP) vs. model-free(MC,TD)
  - Are the rewards delayed or instantaneous ? Does agent involve trial-and-error actions ? Is there a sequential decision making ? supervised learning vs. reinforcement learning
  - How should I design my reward scheme ? what metric should I base my rewards upon ?
  - How far-sighted should your agent be? choice of discount factor
  - Does the agent learn and act on the strategy for taking actions(policy) in environment simultaneously ? on-policy vs. off-policy
  - Are your states complex to represent/featurize? mostly Deep Reinforcement Learning

Windy Grid World is a modification to the famous grid world problem. The problem concerns navigating a grid of 7 X 10 cells by taking a sequence of actions(UP,DOWN,LEFT,RIGHT) at each cell. The goal is to reach the end state/cell - (3,7).An action from a state that leads to (3,7) gets a reward of 0 and every other transition results in a reward of -1. We have additional constraint of wind: There is wind blowing upwards in the columns 4-9 and the wind shift the agent by +1 or +2 steps depending on column the agent moved. Find the optimal path (least cost) path possible to reach (3,7) from (3,0).



We will use python openAI gym (https://gym.openai.com/) for simulating the windygrid environment. Gym is a RL toolkit by open AI which provides hundreds of environments for researchers and developers to develop novel RL algorithms. Each environment is a definition of dynamics of a system which can be used for simulation with option for visualizing the environment. Enviroment provides ways to record rewards from taking actions in a state. In our case, resetting the environment takes the states to (3,0), our start state. The target state is (3,7). We have 70 possible states the agent can be in at anytime. Each state is represented as (row,column) index and stored as 2D arrays. In every state, there are 4 possible actions (0-UP, 1-RIGHT, 2-DOWN, 3-LEFT)

Most of the current literature of RL focus on different problems to demonstrate the various RL techniques which makes comparison harder. To address, we will use the windy grid world to demostrate DP, MC and TD based algorithms. We will learn the Q function and V function along with optimal policy pi*

Most of the current literature of RL focus on different problems to demonstrate the various RL techniques which makes comparison harder. To address, we will use the windy grid world to demostrate DP, MC and TD based algorithms. We will learn the Q function and V function along with optimal policy pi*

```
In [59]: %run helper.py
         import gym
         import gym_gridworlds
         env = gym.make('WindyGridworld-v0')
         env.reset()

Out[59]: (3, 0)
```

Dynamic Programming

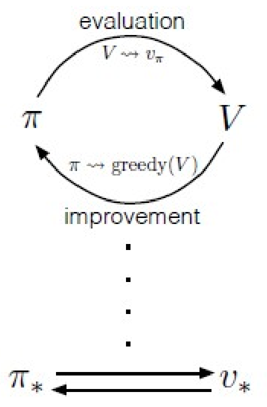
DP is a model-based RL technique which assumes the complete knowledge of the system is readily available. i.e transition probabilities and Rewards/incentive scheme. Below code, we try to solve the Bellman Equation iteratively using Generailzed Policy Iteration. We start with random policy and evaluate wrt. to it. Based on value function learnt as part of evaluation, we will update the policy to being greedy wrt. value function. Next, we again evaluate the new policy to get an updated value function and update the policy to be greedy wrt current value function and this proceeds recursively until policy doesnt change(we call this convergence criteria). At this time, we have optimal policy and hence can evalaute once to get optimal value function and optimal action-value function.

In [87]:

```
import numpy as np
#initialize
policy = np.ones([70, 4]) / 4
V = np.zeros(70)
Q = np.zeros([70,4])

#step1 : policy evaluation(complete)
overall = 0
while True :
    cnt = 0
    while True:
        gamma = 1
        cnt = cnt + 1
        #theta = 1e-8
        theta = 0.01
        delta = 0
        oldpolicy = policy
        Vtmp = np.zeros(70)
        for s in range(70):
            ## store old value of V[s] to calculate delta
            v = V[s]
            v_new = 0
            for a in range(4):
                env.S = (s/10,s%10)
                ## Iterate over possible results of taking action a
                ns,r,done, info = env.step(a)
                if ns == (3,7):
                    r = 0
                v_new += policy[s][a]*(r+gamma*V[10 * ns[0] + ns[1]])
            delta = max(delta, abs(v-v_new))
            Vtmp[s] = v_new
        #print delta
        V =  Vtmp
        if delta < theta or cnt == 1:
            break

    #step2 :Derive Q from V
    for s in range(70):
        for a in range(4):
            q_a = 0
            env.S = (s/10,s%10)
            ns,r,done, info = env.step(a)
            if ns == (3,7):
                r = 0
            q_a += (r + gamma*V[10 * ns[0] + ns[1]])
            Q[s][a] = q_a

    #step 3: policy improvement
    policy = np.zeros([70, 4]) / 4
    for s in range(70):
        Q_s = Q[s]
        argmax = np.argwhere(Q_s == np.amax(Q_s)).flatten().tolist()
        ## Create Stochastic policy if multiple actions yield best results
        prob = 1.0/len(argmax)
        for index in argmax:
            policy[s][index] = prob

    #step 4: check if policy converged
    if np.array_equal(oldpolicy,policy):
        print "converged after ",overall," iterations"
        break
    else:
        None #print "iterate"
        overall = overall + 1
    oldpolicy = policy
```

```
converged after  127  iterations
```

The above code accomplishes the following steps (iteratively approaching optimality)


evaluation
$V \rightsquigarrow v_\pi$
π
V
$\pi \rightsquigarrow \text{greedy}(V)$
improvement
π_*
v_*

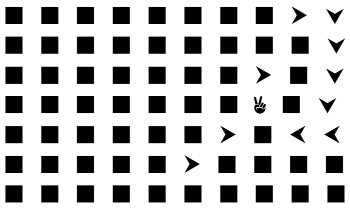
$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

step-0 : Initialization of policy to be equiprobable (ie. in each state the policy mentions taking any action with same prob.), value-function(V) is set to 0. action-value function(Q) is set to 0.
step-1 : Policy Evaluation(PE) tries to arrive at as estimate of value function under the assumption of policy initialized.Vπ(s)=∑aπ(s,a)∑s′Pass′[Rass′+γVπ(s′)]
step-2 : Get Q-function from V-function.Qπ(s,a)=Eπ{rt+γVπ(st+1)|st=s,at=a}=∑s′Pass′[Rass′+γVπ(s′)]
step-3 : Policy Improvement(PI) updates the policy greedily based on value-function(V and Q) estimated as part of step-1 and step-2.π′(s)=argmaxaQ(s,a)
step-4 : Check for convergence - Does the policy (prev iter) == policy (current) ? If yes, stop the loop and call it a day ! converged. If not,go back to step and repeat until convergence.

Below code visualizes the optimal policy by showing optimal path to be followed from start to end(textually and visually). Total cost incurred is also shown. NOTE: The arrows might look wierd, but do note that there is the effect of wind acting upwards which makes the agent shift more than one cells at a time.

```
In [88]: visualize_path(Q,(5,5))
```

```
(5, 5) => (4, 6) => (2, 7) => (0, 8) => (0, 9) => (1, 9) => (2, 9) => (3, 9) =
> (4, 9) => (4, 8) => (3, 7)
-10 10
```



Monte Carlo Technique

It is unrealistic to assume knowledge about dynamics of the system as done in DP. Most of the real world problems, involve unknown system model and we still need to figure out the optimal path. Monte Carlo technique views system model as black-box and instead tries to sample with random initial states and average rewards from there. This is called "Monte Carlo with Exploring starts". MC-ES makes two unrealistic asumptions (i) exploring starts (ii) infintie episodes. To address the exploring starts assumption, on-policy and off-policy methods are used which leverage epsilon-soft policy.
key considerations

1. Episodic Tasks only
2. Require large Episodes
3. Can be first-visit or every-visit based
   MC techniques are the most easy and natural ways to observe the rewards by exploring the environment. It uses simple averaging of rewards (w or w/o discounting) to arrive value of state. V

In [89]:
```
import numpy as np
import random
observation = env.reset()
states = range(0,70,1)
actions = [0,1,2,3]
total_rewards = np.zeros([70, env.action_space.n])
epi_cnt =  np.zeros([70, env.action_space.n])
Q = np.zeros([70, env.action_space.n])
Qold = np.zeros([70, env.action_space.n])
observation = env.reset()
r = 0
done = 0
epi=0
for episode in range(4000):
    #current_state = env.reset()
    #random exploring starts
    current_state = env.reset()
    x = random.randint(0,70)
    current_state = ((x/10)-1,x%10)
    s_a = []
    r = []
    steps = 0
    states = range(0,70,1)
    actions = [0,1,2,3]
    while not done:
            steps = steps + 1
            if steps > 800:
                states = []
                actions = []
                break
            if random.uniform(0, 1) <= 0.7:
                action = env.action_space.sample()
            else:
                action = np.argmax(Q[10 * current_state[0] + current_state[1]])
            next_state,reward,done, info = env.step(action)
            s_a.append((10 * current_state[0] + current_state[1],action))
            current_state = next_state
            r.append(reward)
    #udpate reward and states fist-visit
    #print steps
    for s,a in cartesian([states,actions]):
        try:
            indx = s_a.index((s,a))
        except Exception as e:
            #print(e)
            continue
        cnt = 1
        rew = sum(r[indx:])
        total_rewards[s,a] = total_rewards[s,a] + rew
        epi_cnt[s,a] = epi_cnt[s,a] + 1.0
    done = False
    Q = (total_rewards/epi_cnt)
```

```
C:\Anaconda2\lib\site-packages\ipykernel\__main__.py:52: RuntimeWarning: inval
id value encountered in divide
```

In [116]:
```
visualize_path(Q)
```

```
(3, 0) => (3, 1) => (3, 2) => (2, 2) => (2, 3) => (1, 4) => (0, 5) => (0, 6) =
> (0, 7) => (0, 8) => (0, 9) => (1, 9) => (2, 9) => (3, 9) => (4, 9) => (4, 8)
=> (3, 7)
-16 16
■ ■ ■ ■ ■ ➤ ➤ ➤ ➤ ⮟
■ ■ ■ ■ ➤ ■ ■ ■ ■ ⮟
■ ■ ➤ ➤ ■ ■ ■ ■ ■ ⮟
➤ ➤ ⮝ ■ ■ ■ ■ ✌ ■ ⮟
■ ■ ■ ■ ■ ■ ■ ■ ⮜ ⮜
■ ■ ■ ■ ■ ■ ■ ■ ■ ■
■ ■ ■ ■ ■ ■ ■ ■ ■ ■
```

What if we start from (4,2). what would be the optimal path ?

In [115]:
```
visualize_path(Q,(4,2))
```

```
(4, 2) => (4, 3) => (2, 3) => (1, 4) => (0, 5) => (0, 6) => (0, 7) => (0, 8) =
> (0, 9) => (1, 9) => (2, 9) => (3, 9) => (4, 9) => (4, 8) => (3, 7)
-14 14
■ ■ ■ ■ ■ ➤ ➤ ➤ ➤ ⮟
■ ■ ■ ■ ➤ ■ ■ ■ ■ ⮟
■ ■ ■ ➤ ■ ■ ■ ■ ■ ⮟
■ ■ ■ ■ ■ ■ ■ ✌ ■ ⮟
■ ■ ➤ ⮝ ■ ■ ■ ■ ⮜ ⮜
■ ■ ■ ■ ■ ■ ■ ■ ■ ■
■ ■ ■ ■ ■ ■ ■ ■ ■ ■
```

Temporal Differencing : Q-learning

TD methods attempt to unify key advantages from both DP and MC. It retains advantages of DP (bootstrapping or step-wise update) and idea of sampling from MC. Below we apply an off-policy technique called Q-learning. On-policy counterpart of Q-learning is called SARSA for state-action-reward-state-action. The actual learning happens through exploration in state-action space and uses e-greedy algorithm. It is to be noted, the value-action function is updated immediately after each step unlike MC techniques which needed waiting till the end of episode.

In [119]:
```
import numpy as np
import random
observation = env.reset()
Q = np.zeros([70, env.action_space.n])
observation = env.reset()
r = 0
for episode in range(300):
    current_state = env.reset()
    for step in range(8000):
            #print observation
            if random.uniform(0, 1) < 0.5:
                action = env.action_space.sample()
            else:
                action = np.argmax(Q[10 * current_state[0] + current_state[1]])
            # #######print current_state
            # print action
            next_state,reward,done, info = env.step(action)
            Q[10 * current_state[0] + current_state[1],action] = 0.8 *  Q[10 *
current_state[0] + current_state[1],action] + 0.2 * (reward + np.max(Q[10 * nex
t_state[0] + next_state[1]]))
            current_state = next_state
            #print observation, reward, done, info
            ######print action,next_state,reward
            #r = r + reward
            if done:
                #print("Finished after {} timesteps".format(t+1))
                #print(Q)
                break

#total cost to get to the target
tot_reward = 0
current_state = env.reset()
cnt = 0
for step in range(100):
    #print current_state
    next_action = np.argmax(Q[10 * current_state[0] + current_state[1]])
    next_state,reward,done, info = env.step(next_action)
    tot_reward = tot_reward + reward
    current_state = next_state
    cnt = cnt + 1
    if done:
        #print("Finished after {} timesteps".format(t+1))
        break
print tot_reward,cnt
```

```
-15 15
```

In [120]:
```
visualize_path(Q)
```

```
(3, 0) => (3, 1) => (3, 2) => (3, 3) => (2, 4) => (1, 5) => (0, 6) => (0, 7) =
> (0, 8) => (0, 9) => (1, 9) => (2, 9) => (3, 9) => (4, 9) => (4, 8) => (3, 7)
-15 15
■ ■ ■ ■ ■ ■ ➤ ➤ ➤ ⮟
■ ■ ■ ■ ■ ➤ ■ ■ ■ ⮟
■ ■ ■ ■ ➤ ■ ■ ■ ■ ⮟
➤  ➤ ➤ ➤ ■ ■ ■ ✌ ■ ⮟
■ ■ ■ ■ ■ ■ ■ ■ ⮜ ⮜
■ ■ ■ ■ ■ ■ ■ ■ ■ ■
■ ■ ■ ■ ■ ■ ■ ■ ■ ■
```

All the methods discussed so far fall under so called 'tabular methods' which works well in case of finite managable set of states. If we extend the same windy grid problem as being a grid of 10000 X 10000, the tabular methods doesnt scale well. In such scenarios, approximate methods are employed which uses supervised learning to learn value function given a set of features describing the state. If deep NNs are used in value function approximation, then it falls under deep Reinforcement Learning topic.

Conclusion

As seen from simulations and optimal policy derived from DP, MC and TD algorithms, the best strategy to navigate the grid from (3,0) would be to take extreme outer cells and approach round about of target cell (3,7)

References

- **Book : Reinforcement Learning: An Introduction (http://incompleteideas.net/book/the-book-2nd.html), Second Edition, Feb 2018**
  Richard S. Sutton and Andrew G. Barto
- **REINFORCEjs : Interactive Visualization of Rl Algorithms (https://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_dp.html)**
  Andrej Karpathy (@karpathy)
- **RL Algorithms toolkit (https://gym.openai.com/)**
  OpenAI Gym, Python
- **Overview of Reinforcement Learning (https://medium.com/deep-math-machine-learning-ai/ch-12-1-model-free-reinforcement-learning-algorithms-monte-carlo-sarsa-q-learning-65267cb8d1b4)**
  Madhu Sanjeevi
- **Introduction: Reinforcement Learning with OpenAI Gym (https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2)**
  Ashish Rana
- **Deep Reinforcement Learning: Pong from Pixels (http://karpathy.github.io/2016/05/31/rl/)**
  Andrej Karpathy
- **Reinforcement Learning course (https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ)**
  David Silver, DeepMind