Q learning

N.D. Van Foreest

July 4, 2017

I want to apply Q learning to the EOQ model. First I want to get the tabular Q learning algorithm to work. For this, I copy from Bertsekas, Vol 2, Section 6.4 on Q learning. Then I'll apply it to the EOQ model.

1 Q factors

Define the Q-factors as

$$Q^*(i,u) = \sum_j p_{ij}(u)\left[g(i,u,j) + \alpha J^*(j)\right],$$

where $g(i,u,j)$ is the reward of taking action u in state i and end up in state j, and $J^*(j)$ is the value of continuingin state j. With this expression for $Q^*(i,u)$ the dynamic programming (Bellman) equations take the form

$$J^*(i) = \max_{u\in U(i)}\left\{\sum_j p_{ij}(u)\left[g(i,u,j)+\alpha J^*(j)\right]\right\} = \max_{u\in U(i)} Q^*(i,u).$$

From this we can get an optimality equation for the Q factors:

$$\begin{aligned}
J^*(j) &= \max_{u'\in U(j)} Q^*(j,u') \iff \\
\alpha J^*(j) &= \alpha \max_{u'\in U(j)} Q^*(j,u') \iff \\
\sum_j p_{ij}(u)\alpha J^*(j) &= \sum_j p_{ij}(u)\alpha \max_{u'\in U(j)} Q^*(j,u') \iff \\
\sum_j p_{ij}(u)\left[g(i,u,j)+\alpha J^*(j)\right] &= \sum_j p_{ij}(u)\left[g(i,u,j)+\alpha \max_{u'\in U(j)} Q^*(j,u')\right] \iff \\
Q^*(i,u) &= \sum_j p_{ij}(u)\left[g(i,u,j)+\alpha \max_{u'\in U(j)} Q^*(j,u')\right].
\end{aligned}$$

We can use this to compute Q^* by value-iteration. Start with $Q(i,u)=0$ (or some other set of random values), and iterate according to

$$Q(i,u) := \sum_j p_{ij}(u)\left[g(i,u,j)+\alpha \max_{u'\in U(j)} Q(j,u')\right]. \tag{1}$$

Now we can do two things for Q-learning. We simulate j and $g(i,u,j)$ from the pair (i,u), take $\alpha=1$ to obtain a total cost during a finite horizon, and replace the above expectation by the update rule

$$Q(i,u) = g(i,u,j) + \max_{u'\in U(j)} Q(j,u').$$

To explore and exploit and we can choose to use $u' = \arg\max Q^*(j,u')$ with, for instance, probability 0.9 and otherwise take u' uniform in $U(j)$.

The other thing is to introduce a damping factor $\gamma \in [0,1]$ in (1) so that we obtain

$$\begin{aligned}
Q(i,u) &= (1-\gamma)Q(i,u) + \gamma \sum_j p_{ij}(u) \left[g(i,u,j) + \alpha \max_{u' \in U(j)} Q(j,u') \right] \\
&= Q(i,u) + \gamma \left(\sum_j p_{ij}(u) \left[g(i,u,j) + \alpha \max_{u' \in U(j)} Q(j,u') \right] - Q(i,u) \right).
\end{aligned}$$

Now we simulate j and $g(i,u,j)$ from the pair (i,u) and replace the above expectation by the update rule

$$Q(i,u) := Q(i,u) + \gamma \left(g(i,u,j) + \alpha \max_{u' \in U(j)} Q(j,u') - Q(i,u) \right). \tag{2}$$

To explore and exploit we can use the same trick as before.

2 An Application to an inventory system that satisfies the EOQ assumptions

Standard EOQ model under periodic review such that the review epochs are synchronized with the demand arrivals.. Orders are placed at the start of a period and arrive right away, i.e., immediate replenishments.

$D =$	Demand per period,
$h =$	Holding cost due at the end of the period,
$p =$	Selling price per unit,
$K =$	Ordering cost per order,

Let I_n be the inventory at the end of the period and Q_n the order size. Assuming that replenishments arrive at the start of the period, I_n satisfies the recursion

$$\begin{aligned}
S_n &= \min\{I_{n-1} + Q_n, D_n\}, \\
I_n &= \max\{I_{n-1} + Q_n - s_n, I_{\max}\},
\end{aligned}$$

where S_n is the sales during period n. Thus, in the notation of Section 1, the inventory $I_{n-1} = i$, the action $u_n = Q_n$ and $j = I_n$. With this, it is easy to compute $p_{ij}(u)$.

The end-of-period reward is given by

$$R_n = p s_n - h I_n - K \mathbb{1}_{Q_n > 0}.$$

Thus, $g(i,u,j) = R_n$.

The state space is $I_n \in \{0,1,\dots,I_{\max}\}$, the action space is $\{0,\dots,Q_{\max}\}$.

In the learning procedure we want to reduce the exploration rate while the number of episodes progresses. We take

$$\beta(i) = 0.8 + 0.2\frac{i}{N},$$

if i is the current episode of N episodes in total. I also tried the rule

$$\beta(i) = 0.8$$

for all i. This also gave good results.

Figure 1: piet

```
import numpy as np
import matplotlib.pylab as plt
from matplotlib2tikz import save as tikz_save

from matplotlib import style
style.use('ggplot')


D = 4  # demand per period
h = 1  # holding cost per unit per time
p = 20 # selling price per item
K = 10  # order cost
I_max = 20
Q_max = 10  # maximum order size

state_space = list(range(I_max+1))
action_space = list(range(Q_max))


def run(scenario):
    lr = scenario['lr']
    y = scenario['y']
    num_episodes = scenario['num_episodes']
    length_episode = scenario['length_episode']
    beta = scenario['beta']

    np.random.seed(3)
    Q = np.zeros([len(state_space), len(action_space)])

    # Trace of optimal decisions
    trace = np.zeros([num_episodes, len(state_space)])

    for i in range(num_episodes):
        I = 0
        thres = beta(i)
        for _ in range(length_episode):
            if np.random.rand(1) < thres:
                u = np.argmax(Q[I, :])
            else:
                u = np.random.choice(action_space)
            S = min(I + u, D)  # sales
            I1 = min(I + u - S, I_max)
            R = p*S - h*I1 - K*(u>0)

            #  Update Q-Table with new knowledge
            Q[I, u] += lr*(R + y*np.max(Q[I1, :]) - Q[I, u])
            I = I1

        # keep trace
        trace[i,:] = np.argmax(Q, axis=1)

    np.set_printoptions(precision=3)
    print("Final Q-Table Values")
    print(Q)
    print(np.argmax(Q, axis=1))

    EOQ = np.sqrt(2*K*D/h)
    print("EOQ: ", EOQ)

    fig = plt.figure()
    for i in action_space:
        plt.plot(trace[:,i], 'o-', markersize=2, label=i)
    plt.xlabel('episode')
```

```python
    # plt.title('Demand and demand during leadtime')
    plt.grid(True)
    plt.legend(loc='center right')
    plt.show()
    fname = 'eoq_learning_{}.tex'.format(scenario['name'])
    # tikz_save(fname, figureheight='5cm', figurewidth='12cm')

def max_total(scenario):
    # maximize the total reward during a finite horizon
    num_episodes = scenario['num_episodes']
    length_episode = scenario['length_episode']
    beta = scenario['beta']

    np.random.seed(3)
    Q = np.zeros([len(state_space), len(action_space)])

    for i in range(num_episodes):
        I = 0
        alpha = 0.99
        thres = beta(i)
        for _ in range(length_episode):
            if np.random.rand(1) < thres:
                u = np.argmax(Q[I, :])
            else:
                u = np.random.choice(action_space)
            D = np.random.randint(3,5)
            #D = 4
            S = min(I + u, D)  # sales
            I1 = min(I + u - S, I_max)
            R = p*S - h*I1 - K*(u>0)

            #  Update Q-Table with new knowledge
            Q[I, u] = R + alpha*np.max(Q[I1, :])
            I = I1
    return Q


# Set learning parameters
scenario_1 = {
    'name': "scenario_1",
    'lr': .85,
    'y': .99,
    'length_episode': 100,
    'num_episodes': 2000,
    'beta': lambda x: 0.8
    #'beta': lambda x: 0.8 + 0.2*x/2000
    }

#run(scenario_1)

scenario_2 = {
    'name': "scenario_2",
    'length_episode': 1000000,
    'num_episodes': 1,
    'beta': lambda x: 0.8
    #'beta': lambda x: 0.8 + 0.2*x/2000
    }

Q = max_total(scenario_2)
np.set_printoptions(precision=3)
print("Final Q-Table Values")
print(Q)
print(np.argmax(Q, axis=1))
```

3 First test

TBD: Include results.

4 (s,S) -policies

TBD: Include results.

5 Possible Next Steps

- One way that must give much better results is to use (1) for the iteration, rather than use (2). That must be a major improvement. Of course, this doesn't work for large problems, but is of interest to see how these two compare.
- (S,s) ?
- (Q,r) policy
- use neural network. Check the ice/hole/example on how to do this.