**Homework 02**

**AA203: Optimal and learning-based Control**

Ricardo Luiz Moreira Paschoeto

[rp304154@stanford.edu](mailto:rp304154@stanford.edu)

**Problem 1:** Introduction to Q-learning

a) For learning rate = 0.2

Gráfico

Descrição gerada automaticamente

Figure 1 - Q-values for state-actions

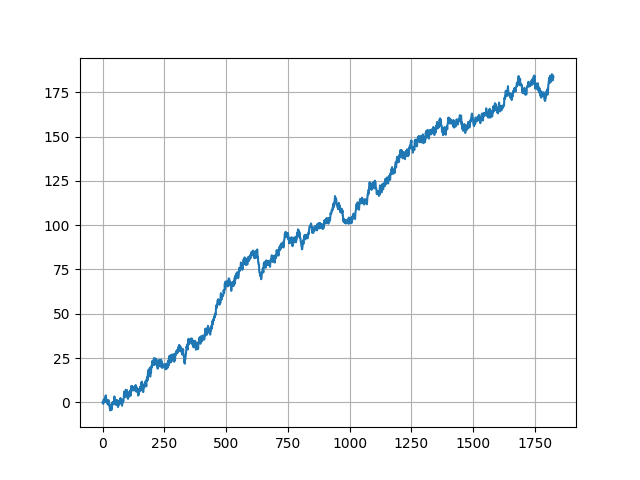


Figure 2 - Aggregate profit over 5 years.

Python code:

def policy(state, Q):

    return np.argmax(Q[state])\*2

def e\_greedy(state,Q):

    if np.random.random() < epsilon:

        a = random\_policy()

        index = int(a/2)

        return random\_policy(), index

    else:

        index = np.argmax(Q[state])

        a = index\*2

        return a, index

def q\_learning():

    q\_values = np.zeros((len(sim.valid\_states),len(sim.valid\_actions)))

    for n in range(N):

        s = sim.reset()

        for t in range(len(data)):

            x0\_hist.append(copy.deepcopy(q\_values[0]))

            x1\_hist.append(copy.deepcopy(q\_values[1]))

            x2\_hist.append(copy.deepcopy(q\_values[2]))

            x3\_hist.append(copy.deepcopy(q\_values[3]))

            x4\_hist.append(copy.deepcopy(q\_values[4]))

            x5\_hist.append(copy.deepcopy(q\_values[5]))

            a, index = e\_greedy(s,q\_values)

            sp,r = sim.step(a)

            td = r + gamma\*np.max(q\_values[sp]) - q\_values[s,index]

            q\_values[s,index] += alpha\*td

            s = sp

    return q\_values

def simulation(Q):

    s = sim.reset()

    r\_hist.append(0)

    for t in range(T):

        a = policy(s,Q)

        sp, r = sim.step(a)

        r\_hist.append(r)

        s = sp

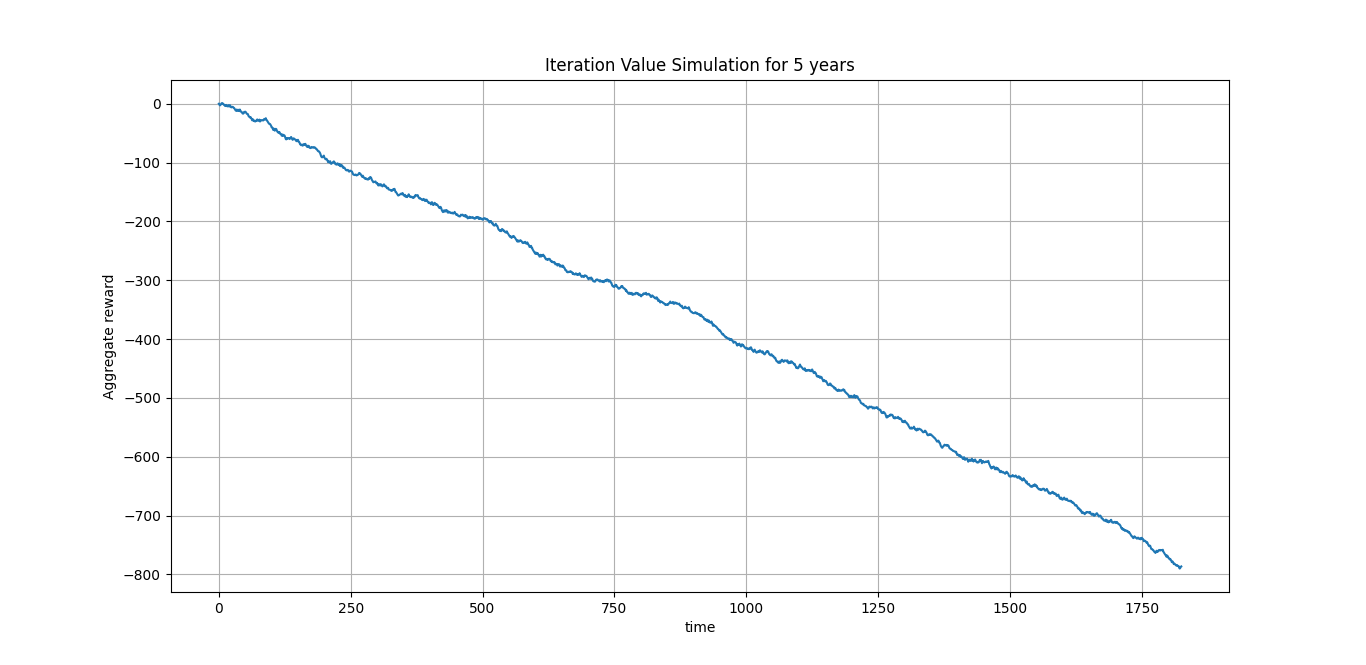
b) Result for *Q* from Iteration Values Iteration along 5 years

Figure - Simulation over 5 years

Python code:

iteration = 0

def value\_iteration(sim, epsilon=0.001):

    V = np.zeros((len(sim.valid\_states)))

    q\_values = np.zeros((len(sim.valid\_states),len(sim.valid\_actions)))

    def next\_step(V, x):

        nv = 0

        policy = 0

        for a in sim.valid\_actions:

            for d, prob in enumerate(sim.demand\_probs):

                next\_state = sim.transition(x,a, d)

                r = sim.get\_reward(x, a, d)

                v = sim.demand\_probs[next\_state] \* (r + gamma \* V[next\_state])

                if nv < v:

                    nv = v

                    policy = a

        return nv, policy

    while True:

        delta = 0

        for x in sim.valid\_states:

            prev\_v = V[x]

            best\_v, best\_a = next\_step(V, x)

            V[x] = best\_v

            delta = max(delta, np.abs(prev\_v - V[x]))

            q\_values[x][int(best\_a/2)] = V[x]

        print(delta)

        if delta < epsilon:

            break

    return q\_values

c) Q-learning presented better results.

After read explanations about this behavior, where the Q-learning perform better than value iteration, my conclusion is how each algorithm works, Q-learning operate over a finite-horizon of five years while the Value Iteration operate over a infinite horizon. One of solutions to help us analyze this outperformance is doing the simulation with bigger values of gamma, forcing the VI policy cares about rewards in distant future. Q-learning already have this behavior, it cares about future reward.

**Problem 2:** Cart-pole swing-up

**a)** *A, B = jax.jacfwd(f, (0, 1))(s, u)*

**,b)**

(I)

=> (II)

, => (III)

. => (IV)

Substituting (II),(III) and (IV) in (I)

Manipulating and considering *R, Q and*  are symmetric,

where,

And the terms,

.

c) Python code:

    def costs():

        c = np.zeros((N,m)) # immediate state cost

        cs = np.zeros((N, n)) # dc / dx

        cu = np.zeros((N, m)) # dc / du

        cuu = np.zeros((N, m, m)) # d^2 c / du^2

        css = np.zeros((N, n, n)) # d^2 c / dx^2

        cus = np.zeros((N, m, n))  # d^2 c / du / dx == 0 Don't have cross terms (s,u)

        for k in range(N-1):

            qkT = np.dot((s[k] - s\_goal).T, Q) # (sk\_bar - s\_goal).T\*Q

            rkT = u[k] @ R # uk.T\*R

            c[k] = (0.5)\*np.dot((s[k] - s\_goal).T, np.dot(Q, (s[k] - s\_goal))) + (0.5)\*np.dot(u[k].T, np.dot(R, u[k])) # (1/2)\*(sk\_bar - s\_star).T \* Qk \* (sk\_bar - s\_star) + (1/2)\*(uk\_bar.T \* Rk \* uk\_bar)

            cs[k] = qkT # qk.T

            cu[k] = rkT # rk.T

            cuu[k] = np.array(R) # just R

            css[k] = np.array(Q) # just Q

        qNT = (s[-1] - s\_goal).T @ Qf # (sN\_bar - s\_star).T\*Qf

        c[-1] = (0.5)\*(s[-1] - s\_goal).T @ Qf @ (s[-1] - s\_goal)  # (1/2)\*(sN\_bar - s\_goal).T\*QN\*(sN\_bar - s\_goal)

        cs[-1] = qNT # qN.T ????

        css[-1] = Qf # final cost

        return np.array(c), np.array(cs), np.array(cu), np.array(cuu), np.array(css), cus

    # iLQR loop

    for \_ in range(max\_iters):

        # Linearize the dynamics at each step `k` of `(s\_bar, u\_bar)`

        A, B = jax.vmap(linearize, in\_axes=(None, 0, 0))(f, s[:-1], u)

        A, B = np.array(A), np.array(B)

        # WRITE YOUR CODE BELOW ###############################################

        # Update the arrays `L`, `l`, `s`, and `u`.

        # for storing quadratized cost function

        # Foward pass:

        for k in range(N):

            s[k + 1] = f(s[k], u[k])

        # Compute cost

        c, cs, cu, cuu, css, cus = costs()

        # Backward pass:

        v, v\_bold, V = c[-1].copy(), cs[-1].copy(), css[-1]

        l = np.zeros((N, m))

        L = np.zeros((N, m, n))

        for k in range(N - 1, -1, -1):

            Qk = c[k] + v

            Qs = cs[k] + (A[k].T @ v\_bold)

            Qu = cu[k] + (B[k].T @ v\_bold)

            Qss = css[k] + (A[k].T @ V @ A[k])

            Quu = cuu[k] + (B[k].T @ V @ B[k])

            Qus = cus[k] + (B[k].T @ V @ A[k])

            l[k] = -np.linalg.inv(Quu) @ Qu

            L[k] = -np.linalg.inv(Quu) @ Qus

            v = Qk - 0.5\*(l[k].T @ Quu @ l[k])

            v\_bold = Qs - (L[k].T @ Quu @ l[k])

            V = Qss - (L[k].T @ Quu @ L[k])

        for k in range(N - 1):

            u[k] = u\_bar[k] + l[k] + np.dot(L[k], s[k] - s\_bar[k])

            s[k + 1] = f(s[k], u[k])

        #######################################################################

        print(np.max(np.abs(u - u\_bar)))

        if np.max(np.abs(u - u\_bar)) < eps:

            converged = True

            break

        else:

            u\_bar[:] = u

            s\_bar[:] = s

    if not converged:

        raise RuntimeError('iLQR did not converge!')

    return s\_bar, u\_bar, L, l

d) Python code:

        if simulate\_continuous\_time\_dynamics:

            # WRITE YOUR CODE BELOW ###########################################

            # Update `u[k]` using the final LQR policy `L`, `l` output by

            # `ilqr` above to track the planned trajectory when we simulate the

            # continuous-time dynamics.

            u[k] = u\_bar[k] + l[k] + np.dot(L[k], s[k] - s\_bar[k])

            ###################################################################

            s[k+1] = odeint(lambda s, t: f(s, u[k]), s[k], t[k:k+2])[1]

e) Plot Results:

Ícone

Descrição gerada automaticamente com confiança baixa

Figure 4 - Results for cart-pole iLQR

**Problem 3:** Cart-pole swing-up with limited actuation

a) the discrete dynamic is,

Linearing around

Where,

.

,

Subject to

.

b) *A, B, c = jax.jacfwd(lambda x:f(x, u))(x), jax.jacfwd(lambda u:f(x, u))(u), f(x, u).*

c)Python code:

def scp\_iteration(f,Q,R,Q\_N,s\_bar,u\_bar,s\_star,s0,N,dt,rho,uLB,uUB):

    ###########################################################################

    # WRITE YOUR CODE HERE

    # implement one iteration of scp

    # HINT: See slides 34-38 of Recitation 1.

    n = Q.shape[0]

    m = R.shape[0]

    S = {}

    U = {}

    u = np.zeros((N, m))

    s = np.zeros((N+1, n))

    #print("s shape = " + str(s.shape))

    cost\_terms = []

    constraints = []

    #A, B, c = linearize(f, s\_bar, u\_bar)

    for t in range(N):

        S[t] = cvx.Variable(n)

        U[t] = cvx.Variable(m)

        cost\_terms.append(cvx.quad\_form(S[t] - s\_star, Q)) # State cost

        cost\_terms.append(cvx.quad\_form(U[t], R)) # Control cost

        constraints.append(U[t] <= uUB)

        constraints.append(U[t] >= uLB)

        constraints.append(cvx.norm(U[t] - u\_bar[t], "inf") <= rho) # Box constraint 1

        constraints.append(cvx.norm(S[t] - s\_bar[t], "inf") <= rho) # Box constraint 2

        if t == 0: # S[0] == s0 # Initial condition

            constraints.append(S[t] == s0)

        if t > 0:

           A, B, c = linearize(f, s\_bar[t - 1], u\_bar[t - 1])

           constraints.append(A @ (S[t - 1] - s\_bar[t - 1]) + B @ (U[t - 1] - u\_bar[t - 1]) + c == S[t])

    S[t + 1] = cvx.Variable(n)

    A, B, c = linearize(f, s\_bar[t], u\_bar[t])

    constraints.append(A @ (S[t] - s\_bar[t]) + B @ (U[t] - u\_bar[t]) + c == S[t+1])

    cost\_terms.append(cvx.quad\_form(S[t + 1] - s\_star, Q\_N))

    obj = cvx.Minimize(cvx.sum(cost\_terms))

    problem = cvx.Problem(obj, constraints)

    problem.solve()

    for k in range(len(U)):

       u[k, :] = U[k].value

    for t in range(len(S)):

       s[t, :] = S[t].value

    ###########################################################################

    return s,u

d)

Texto, Quadro de comunicações

Descrição gerada automaticamente

Figure 5 - Results for cart-pole scp.