TP2, Reinforcement learning

Mariia Drozdova

February 25, 2020

1 Exercise 0

File `replacement_model.py` contains a model of deterioration.

1.1 0.1 Identify a law of deterioration between two decisions, deterioration when the object should be replaced and associated punishment. Interpret a function of maintenance cost.

```
xmax = 10 # maximum level of use
xdead = 9 # level of death
Creplace = 40 # cost of replacement
Cdead = 10; # cost of death
```

Listing 1: Parameters in `replacement_model.py`

First, we notice that **x** here states for current level of use. When it is equal to **xdead**, the object should be recycled that has a cost **Cdead** (punishment). Buying new object will instead will cost **Creplace**. If the object is replaced when it is not dead, we do not have to pay **Cdead**.

```
beta = 2.0 # exponential distribution parameter

def sample_exp( beta ):
    return -1.0/beta*log( random() )

def next_state_and_reward( x, a):
    c=0
    if a==0: # replace
        y=sample_exp(beta)
        c+=Creplace + maintenance_cost(0,y)
    else: # keep
        y = x + sample_exp(beta)
        c += maintenance_cost(x, y)
    if y>xdead:
        y, c2 = next_state_and_reward ( y, 0 )
        c += Cdead + c2
    return y,c
```

Listing 2: The law for the level of use

The level of use which corresponds to the level of deterioration is a sum of values produced by `sample_exp`. This function returns samples from the exponential distribution of parameter beta. Sum of exponential variables with the same beta is gamma. A law of deterioration between two decisions is exponential.

```
def maintenance_cost(x,y):
    c=y-x
    if x < 2.5 < y:
        c=c+10
    if x < 5 < y:
        c=c+20
    if x < 7.5 < y:
        c=c+30
    return c
```

Listing 3: Maintenance cost function.

The maintenance cost can be interpreted as the following:

The cost is equal to the change of the level of use. If the level of use passes some predefined points, addition service should be applied. For the point of 2.5, the service costs 10, for 5 the cost is 20 and, finally, for 7.5 the cost is 30.

1.2 How to improve the solution?

A file `replacement.py` contains an already implemented algorithm of Discretized Value Iteration. We discretize the value space - the possible values of the level of use **x**. Then, we compute all possible transitions and rewards from this x depending on the taken action (**y0** and **y1**).

The file contains the following parameters:

```
N = 151 #  number of samples points
M = 10 # number of next_state_and_reward samples
K = 30 # number of updating alpha
```

Listing 4: Maintenance cost function.

N is responsible for discretisizing.

We construct the value function which associates each sample interval to the value which can be received from it. The algorithm updates these values though **K** iterations. For each iteration we run the algorithm for all sample points **N** making **M** next steps from each.

The figures 1 and 2 are the example of the output we get if we run only preimplemented algorithm.

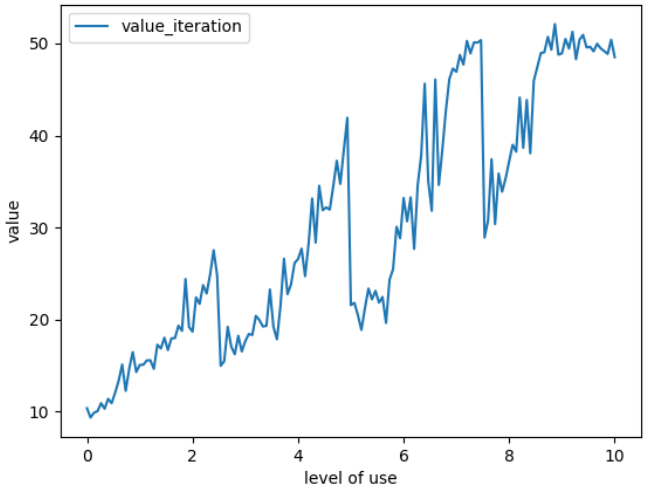

Figure 1: Values for the Discretized Value Iteration.

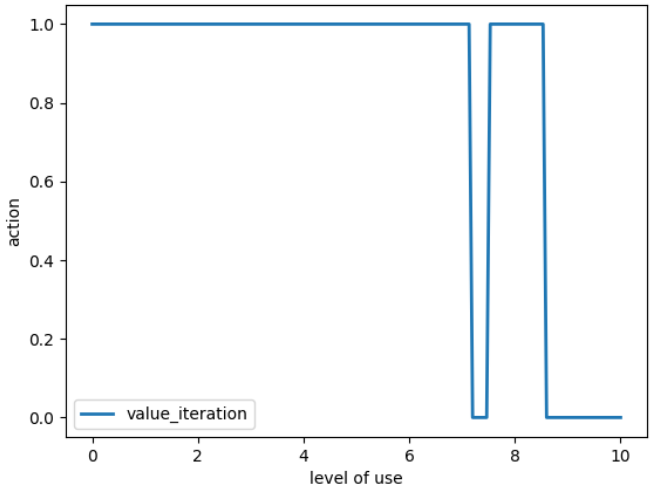

Figure 2: Actions for the Discretized Value Iterat

Thus, to improve the solution we can use more discretisized space. (bigger **N**) We can also augment the number of next steps (**M**) performed for each sample. The higher number of step leads to more precision.

2 Exercice 1 (Fitted Value-Iteration)

In this exercise we try to approximate the value function in the Fourier basis of functions $\phi_j(x) = \cos \frac{\pi j x}{x_{\max}}$ where $1 \leq j \leq d$. This time we will update the coefficients of the value function in this basis through solving minimization problem.

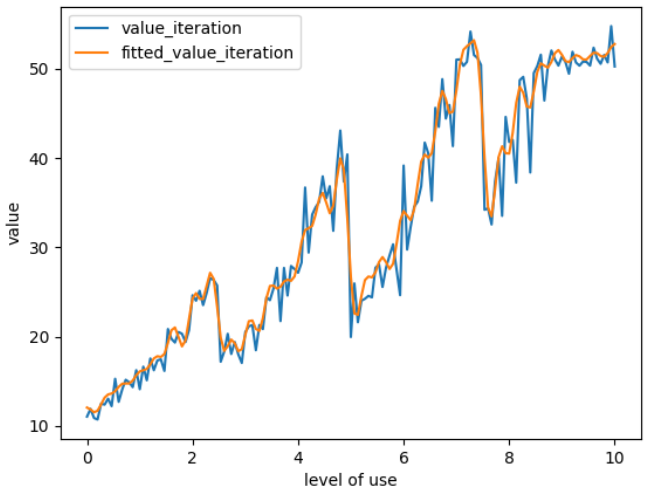

Figure 3: Values for the Fitted Value Iteration.

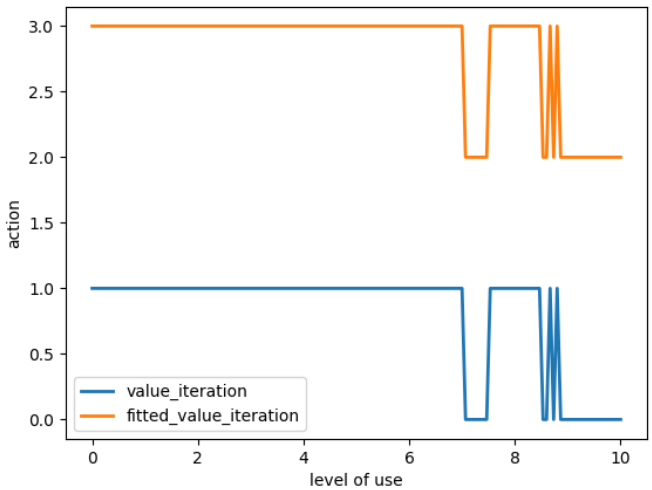

Figure 4: Taken actions for the Fitted Value Itera

The Figures 3 and 4 show the approximated value function and the action of this and previous technique. The value function is smooth in case of fitted value iteration as we have smooth basis. In terms of basis the first algorithm of value iteration can be rewritten as if we have a discrete euclidean basis of N vectors in N-dimensional space. The coefficients are corresponding values of each of N samples.

```
def return_cos(j, x, x_max=xmax):
    return np.cos(j*np.pi*x/x_max)
def compute_F(x):
    F = []
    for j in range(d):
        F.append(return_cos(j, x))
    F = np.array(F)
    return F
def return_value(alpha, x):
    res = 0
    for j in range(len(alpha)):
        res += alpha[j] * return_cos(j, x)
    return res
Fi = compute_F(x_samples).T

def fitted_value_iterate(alpha):
    alpha2,pol=[],[]
    Tvs = []
    for i in range(N):
        print((i+1),"/",N,"    \r", end=' ')
        sys.stdout.flush()
        Tv0,Tv1=0,0
        for j in range(M):
            value0 = return_value(alpha, y0_samples[i][j])
            value1 = return_value(alpha, y1_samples[i][j])
            Tv0 += r0_samples[i][j] + gamma * value0
            Tv1 += r1_samples[i][j] + gamma * value1
        if Tv0<=Tv1:
            Tv=Tv0/M
            pol.append(0)
        else:
            Tv=Tv1/M
            pol.append(1)
        Tvs.append(Tv)
    Tvs = np.array(Tvs)
    alpha2 = np.linalg.inv((np.transpose(Fi) @ Fi)) @ (np.transpose(Fi) @ Tvs)
    return alpha2,pol

def fitted_value_iteration(K):
    alpha=initiate_alpha()
    for i in range(K):
        alpha, pol = fitted_value_iterate(alpha)
        print(i,"              ",pol)
    values = return_value(alpha, x_samples)
    return values, pol
```

Listing 5: Fitted Value Iteration algorithm.

3 Exercice 2 (Fitted Q-Iteration)

In this exercise we approximate the $\mathbf{q}$ in the basis ϕ introduced in the previous section. We introduce two set of coefficients α_0 and α_1 corresponding to the two actions. Then, we solve two problems of minimization.

After getting q_0 and q_1 for each sample we find the minimal of them getting a value and an action for each sample.

The code can be found in the Listing 6. The output for all algorithms included into this project is displayed in the Figures 5 and 6. The curves for value are approximating the same law.

```
1
def fitted_q_iterate(alpha0, alpha1):
3     pol=[]
      Tvs0, Tvs1 = [],[]
5     for i in range(N):
          #print((i+1),"/",N,"    \r", end=' ')
7         sys.stdout.flush()
          Tv0,Tv1=0,0
9         for j in range(M):
              min_q0 = np.min(np.array([return_value(alpha0, y0_samples[i][j]),
    return_value(alpha1, y0_samples[i][j])]))
11            min_q1 = np.min(np.array([return_value(alpha0, y1_samples[i][j]),
    return_value(alpha1, y1_samples[i][j])]))
              Tv0 += r0_samples[i][j] + gamma * min_q0
13            Tv1 += r1_samples[i][j] + gamma * min_q1
          Tvs0.append(Tv0/M)
15        Tvs1.append(Tv1/M)
      Tvs0 = np.array(Tvs0).reshape(N, 1)
17    Tvs1 = np.array(Tvs1).reshape(N, 1)
      alpha0_2 = np.linalg.inv((np.transpose(Fi) @ Fi)) @ (np.transpose(Fi) @
    Tvs0)
19    alpha1_2 = np.linalg.inv((np.transpose(Fi) @ Fi)) @ (np.transpose(Fi) @
    Tvs1)
      return alpha0_2, alpha1_2, pol
21
def fitted_q_iteration(K):
23    alpha0, alpha1=initiate_alpha(), initiate_alpha()
      for i in range(K):
25        alpha0, alpha1, pol = fitted_q_iterate(alpha0, alpha1)
          print(i,"              ")#,pol
27
      qa0 = [return_value(alpha0, x_sample) for x_sample in x_samples]
29    qa1 = [return_value(alpha1, x_sample) for x_sample in x_samples]
      values = np.min(np.stack((qa0, qa1)), axis=0)
31    pol = np.argmin(np.stack((qa0, qa1)), axis=0)
      return values, pol
```

Listing 6: Fitted Q Iteration algorithm.

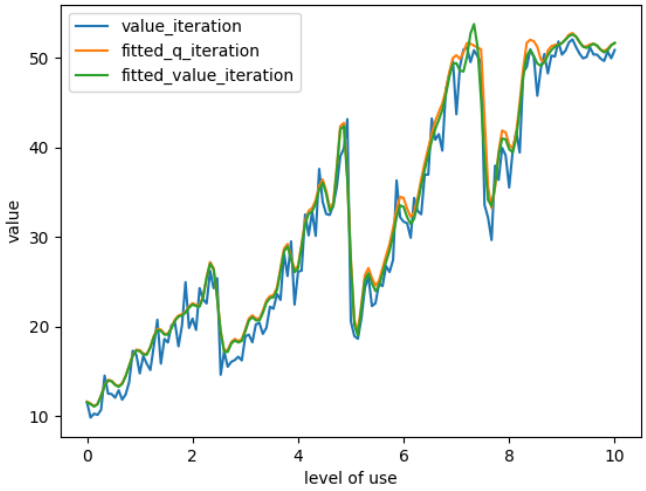

Figure 5: Values depending on the level of use.

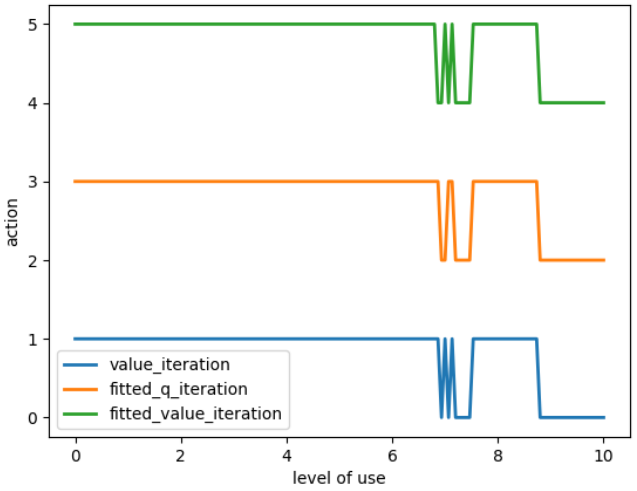

Figure 6: Actions depending on the level of use

Finally we test the three algorithms with **M** equal to one which means that we generate next step only once. It leads to big uncertainties for all algorithms with the worse result for discretized version. The difference between fitted value and q iteration is also becoming more visible. In case of value iteration we approximate two sets of coefficients which leads to bigger variance and error. Most of the time the value returned for q iteration is higher than the one returned for value iteration. Q-iteration has worse convergence properties that value iteration in our case. If we take bigger M, on the contrary the curves are getting closer to each other.

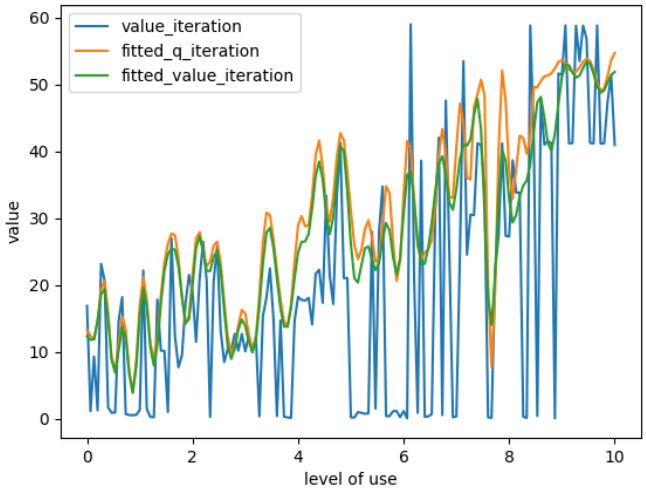

Figure 7: Values depending on the level of use with M=1.

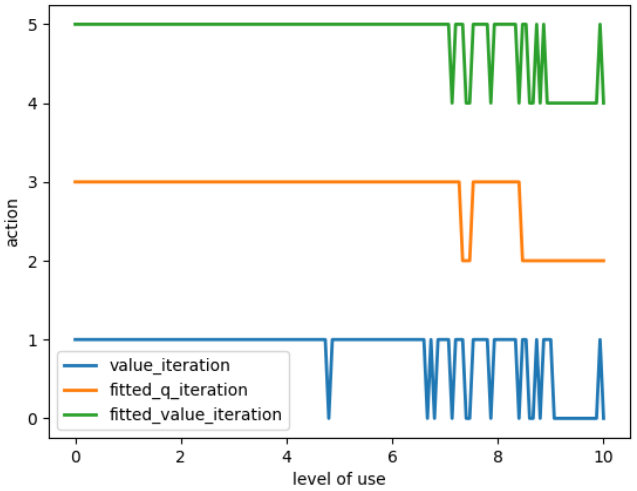

Figure 8: Actions depending on the level of use M=1.