Master in applied sciences
Computer science and engineering
Year 2018-2019

Introduction to Complex problem solving

Assignement 3

INFO8003-1
Damien Ernst

Olivier Moitroux
s152122

Liège

Février 2019

Domain representation

The **origin** of the grid has been put in the **bottom left** of the board so that, in the provided domain instance of the assignment, the red cell is considered to be at position (0,1). The value of y increases as we go up (0,1) while the value of x increases by going to the right (1,0). The value of the reward `grid`[0, 0] is thus considered to be -20.

-3	1	-5	0	19
6	3	8	9	10
5	-8	4	1	-8
6	-9	4	19	-5
-20	-17	-4	-3	9

If more information relative to the implementation is expected, please refer to more details explanation in the assignment 1.

Part I

$\gamma = 0.99$

6 Q-learning in a Batch Setting

6.1 Convergence speed of $\hat{Q}$ towards Q

6.1.1 Chosen framework

Assignement was not clear about the framework we had to use to display convergence. I chose this one:

- No experience replay for q-learning
- 1 single trajectory of length increasing up to convergence for q-learning.
- As many updates in q-learning as there are transitions in the generated trajectory.
- Q is computed like in question 4 with N=500. $\hat{Q}$ computed by q-learning described above.

$\hat{Q}$ is computed with in an increasing trajectory length. The Q considered is always Q_{500}, that is, the Q computed in question 4 at convergence. The convergence is highlighted by considering the sum of square error between all elements of $Q(x, u)$ (that is a $width \times height \times 4$ matrix computed in Q4) and of $\hat{Q}$, computed with q-learning (same dimensions as Q(x, u) of course). To do so, I store a $width \times height \times 4$ matrix of square error and I only update the error on the element that is updated in the q-learning. Then, I only recompute the sum for this given length of trajectory for the plot. Let's note we could also use the bound on the error like explained in the course to stop when appropriate accuracy is reached. Other metrics could be used as well.

6.1.2 Plots and discussion

The evolution of the sum of the square error between $\hat{Q}$ and Q is reported in figure 1 for deterministic and stochastic settings.

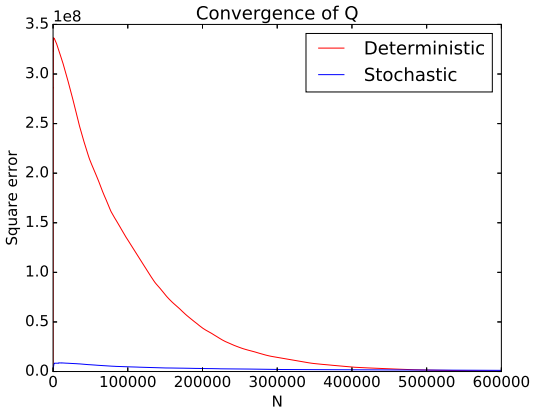


Figure 1 – Evolution of sum of square error between $Q(x, u)$ and $\hat{Q}(x, u)$, $\forall x \in X, \forall u \in U$.

At a first sight, the results are a bit surprising. Indeed, it seems like using the q-learning algorithm with a very small trajectory length leads to a huge sum of square error between $\hat{Q}$ and Q in the deterministic case. This might be explained by the fact that Q is initialised to 0 everywhere and that the information inside the history might not even update Q at the beginning if new value are below 0 (see max in the algorithm). In the stochastic setting, the agent tends to get back to [0, 0], which for me, gets a reward of -20. For this reason, the expected cumulative result is smaller in the stochastic case and thus, closer to the early stages of the q-learning algorithm. Still, I'm surprised to reach such high values.

Other than that, the q-learning algorithm seems to stabilise after 500000 updates in the deterministic case and ealier at around 400000 for the stochastic case. The convergence in the deterministic case is way steeper due to the fact that the information gai

Idea of interesting plots

These plots were not asked but could confirm our intuitions and reasoning for the next sections.

- Plotting the sum of square error between $J^n_{\hat{u}^*}$, $n = 0, \ldots, N$ and J_{u^*500} of question 4.
- Highlight, like I did in previous assignement, the effect of considering several trajectories instead of a long one of increasing length.
- Make the same plots with random replay and playing with the batch-size.

6.1.3 Expected cumulative reward computed with q-learning.

Once convergence is achieved, we can derive $\hat{u}^*$ from $\hat{Q}$ and display $J^N_{\hat{u}^*}$. I considered N=600000 because q-learning is very fast anyway and 1 trajectory with no experience replay (nothing stated in the assignment about the configuration to use). Table 1 shows the optimal policy and figure 2 $J^N_{\hat{u}^*}$ besides $J^N_{\hat{u}*}$, that was computed in question 4 of previous assignment. Let's highlight that figure 2c and 2d don't have very close values, leading us to the conclusion that the stochastic setting might be difficult to tackle with q-learning. A correct tuning of the parameters is thus important. Let's note that using the start position [3, 3] better helps in

the stochastic setting to accumulate information about states around [5, 5] (in opposition to start at [0, 0] f.e.). In the same way, considering more smaller trajectories also benefit (see my plots/justifications that have been discussed in the last assignment). This won't be shown, again, in the report but can be easily verified by tweaking values in the code.

↓	→	→	→	→
→	→	→	→	↑
↑	→	↑	↑	↑
↑	→	→	↑	↑
↑	→	↑	↑	←

Deterministic

↓	↓	↓	→	↑
→	→	→	→	→
↑	↑	↓	↓	↑
↑	→	←	←	↓
↑	→	↑	↑	↓

Stochastic, $\beta = 0.2$

Table 1 – Optimal policy $\hat{\mu}^*$ by q-learning (N=600000)

```
Expected cumulative reward of optimal policy:
[[1767.7243 1783.6172 1807.5653 1826.4805 1826.6141]
 [1780.2010 1796.1281 1807.3812 1817.4734 1826.5073]
 [1767.7762 1781.5558 1796.2676 1807.1087 1816.9887]
 [1754.0511 1774.3254 1788.8675 1788.8108 1789.7836]
 [1741.4957 1751.8190 1774.2978 1788.8282 1766.8551]]
```

(a) Deterministic: $J^N_{\hat{\mu}^*}$

```
[[ 1831.52989208  1846.58327896  1870.39327896  1889.39327896
   1889.39327896]
 [ 1843.96958796  1859.67237896  1870.48327896  1880.39327896
   1889.39327896]
 [ 1831.42382487  1844.96958796  1859.67237896  1870.48327896
   1880.39327896]
 [ 1818.00351941  1838.40312487  1853.03958796  1852.67237896
   1853.48327896]
 [ 1805.71741701  1815.91302641  1838.40312487  1853.03958796
   1831.50919208]]
```

(b) Deterministic: $J^N_{\mu^*}$

```
Expected cumulative reward of optimal policy:
[[188.2131 189.5997 197.5258 197.7263 194.3334]
 [192.3373 198.5971 199.0298 196.7170 196.1319]
 [188.2863 194.5730 198.4449 212.6047 196.8181]
 [189.6406 199.5346 202.3155 201.3929 202.9487]
 [191.7618 186.3829 199.8955 210.1213 200.3684]]
```

(c) Stochastic: $J^N_{\hat{\mu}^*}$ (β=0.2)

```
[[ 296.6329785   301.13313707  314.29031983  329.49031983  329.49031983]
 [ 301.12569759  309.83801903  315.78791983  322.29031983  329.49031983]
 [ 296.62553902  301.92569759  309.83801903  315.78791983  322.29031983]
 [ 292.26141343  301.65046062  309.49049759  304.23801903  309.49793707]
 [ 289.60316165  289.04115133  301.65046062  309.49049759  296.0579001 ]]
```

(d) Stochastic: $J^N_{\mu^*}$ ($\beta = 0.2$)

Figure 2 – Left: $J^N_{\hat{\mu}^*}$, computed with q-learning ($\alpha = 0.05$, no experience replay, N=600000, 1 trajectory of length N.). Right: $J^N_{\mu^*}$ (N=500), see prev. assignment Q4.

6.2 Intelligent agent with ϵ-greedy policy.

6.2.1 Analysis of effect of batch size

Framework:

- Constants:
  - 100 episodes
  - 1000 trajectories
  - $\Longleftrightarrow$ 100.1000 = 100000 total transitions considered for learning.
  - $\alpha = 0.05$
- Parameter: batch-size
  - Experience replay, batchsize = 1
  - Experience replay, batchsize = 10
  - Experience replay, batchsize = 100

Results and discussion.

Changing the batch-size means we can sample i.i.d. several tuples in the form $[x_i, u_i, r_i, x_i +$

1] $i \in [1, T]$ at once. Of course, we fix the total number of sampling for the q-learning so that the only parameter is the batch-size. **By increasing the batch-size, there is a gain in information**. Indeed, by fixing the batch-size to x, we update directly x times Q and we are guaranteed to take x different tuples.

In opposition, with experience replay and a batch-size of 1, we are more likely to take the same tuple several times (less exploration): we don't extract as much from the trajectory. This effect is quite noticeable when we consider very few episodes that are very short. Indeed, the q-learning algorithm considers inital condition of zero and more elements stays to zero with small batch-size. Making plots would not have been too diffucult but too time consuming (not asked anyway). So I'll only display the J's in figure 3.

```
[[711.0049 726.1177 750.7168 772.6269 769.8908]
 [719.8364 737.5629 752.2938 761.4858 773.5113]
 [705.8083 719.0594 739.1045 747.3800 763.0768]
 [692.0734 707.4589 726.7886 725.8579 738.6758]
 [680.6433 684.4007 711.0738 725.4963 710.6284]]
```

(a) Batch-size = 1 (deterministic)

```
[[90.9968 82.8577 75.9949 66.4167 48.2889]
 [98.8950 96.3068 91.2330 80.9245 60.3834]
 [98.9421 97.6432 94.4632 98.2334 81.6499]
 [100.0977 100.0516 105.0647 97.6095 92.9628]
 [99.1509 90.6719 99.7244 105.3467 86.1715]]
```

(b) Batch-size = 1 (stochastic, $\beta = 0.2$)

```
[[1836.4176 1851.5956 1875.4362 1894.4442 1894.4400]
 [1848.9746 1864.6845 1875.5012 1885.4211 1894.4382]
 [1836.4139 1849.9686 1864.6661 1875.4783 1885.4372]
 [1822.9825 1843.4053 1858.0477 1857.6736 1858.5313]
 [1810.6860 1820.8853 1843.3960 1858.0388 1836.3894]]
```

(c) Batch-size = 10 (deterministic)

```
[[212.9183 217.4583 229.0895 241.2657 239.8526]
 [215.8132 223.1823 235.3364 238.8594 241.0774]
 [217.1414 223.2119 226.4800 232.6352 240.0209]
 [210.1300 221.2895 232.8734 225.3856 221.5787]
 [211.0074 208.3126 222.3678 234.9633 224.4123]]
```

(d) Batch-size = 10 (stochastic, $\beta = 0.2$)

```
[[1842.0305 1857.1900 1881.0000 1900.0000 1900.0000]
 [1854.5763 1870.2791 1881.0900 1891.0000 1900.0000]
 [1842.0305 1855.5763 1870.2791 1881.0900 1891.0000]
 [1828.6102 1849.0098 1863.6463 1863.2791 1864.0900]
 [1816.3241 1826.5197 1849.0098 1863.6463 1842.0098]]
```

(e) Batch-size = 100 (deterministic)

```
[[377.7551 389.3849 411.3629 429.7166 437.6816]
 [381.9684 387.7163 402.8690 411.8921 426.2950]
 [378.0769 387.3729 396.1410 400.1493 413.9736]
 [373.1617 390.8111 396.6597 388.1771 395.9807]
 [372.8528 369.2166 382.7817 391.0147 379.0536]]
```

(f) Batch-size = 100 (stochastic, $\beta = 0.2$)

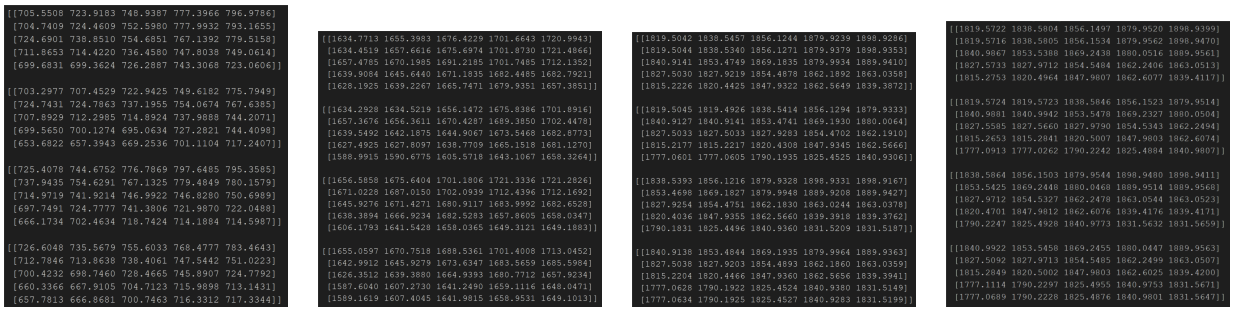
Figure 3 – Effect of batch-size on $J^N_{\hat{\mu}^*}$ when we consider experience replay. Number of samples = N is equal to 100000.

By playing around with more values than the ones exposed (1-10-100), it turns out that a small batch-size results in a very slow convergence. On the other hand, performance-wise, it is very fast to compute. Increasing a little bit the batch-size greatly benefit the speed of convergence and offer already good results with a batch-size of 10. However, this benefit is not much improved when considering significantly larger value of batch-sizes. It even turns out that large batch-sizes result in slow computations.

With regards to the distribution of the expected cumulative reward for each position in the grid, we can highlight the fact that, in the stochastic setting, the larger the batch-size, the larger is $J^N_{\hat{u}^*}(5,\, 5)$ compared to the other elements. For example, in figure 3(b), the expected cumulative result of [5, 5] is the smallest value in the grid, while its immediate reward is 19=Br. By increasing the batch-size up to 100, we observe in figure 3(f) that is has reached the highest value in the grid, thanks to a more equitable sampling among the transitions present in the trajectory.

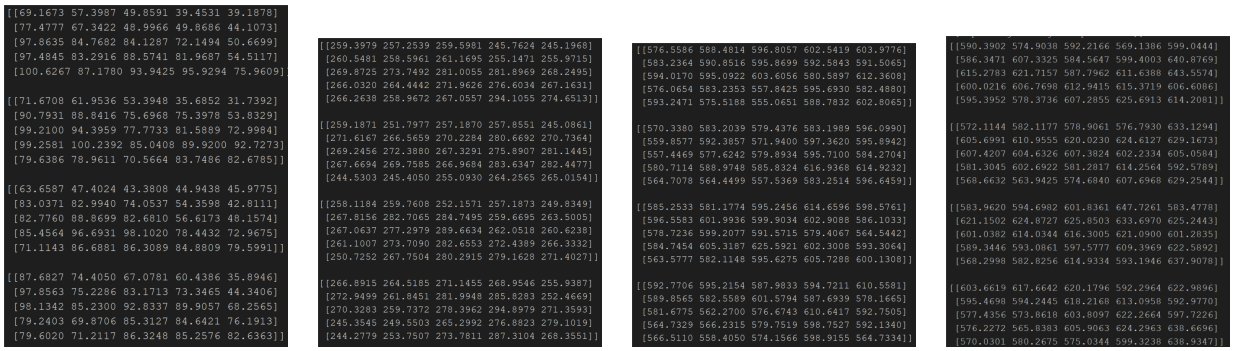
6.3 Analysis of effect of learning rate α

The protocol used is the one from the assignment with alpha as our parameter. The different $\hat{Q}$ are displayed in figure 4 and figure 5 for deterministic and stochastic settings. In the same way, the expected cumulative rewards of the extracted optimal policy are displayed for each settings in figure 6.



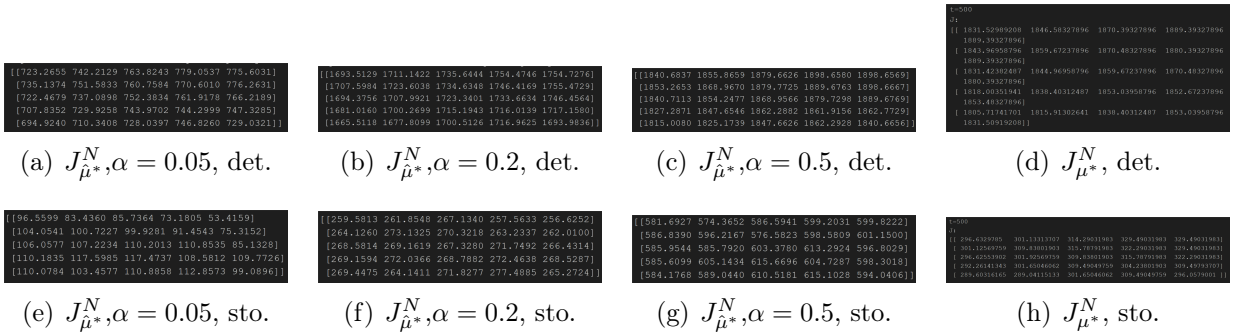
(a) $\hat{Q}$,$\alpha = 0.05$ (b) $\hat{Q}$,$\alpha = 0.2$ (c) $\hat{Q}$,$\alpha = 0.5$ (d) $\hat{Q}$,$\alpha = 0.5 \rightarrow 0$

Figure 4 – Effect of alpha on $\hat{Q}$ in deterministic case. [↑, ←, →, ↓] from top to bottom.



(a) $\hat{Q}$,$\alpha = 0.05$ (b) $\hat{Q}$,$\alpha = 0.2$ (c) $\hat{Q}$,$\alpha = 0.5$ (d) $\hat{Q}$,$\alpha = 0.5 \rightarrow 0$

Figure 5 – Effect of alpha on $\hat{Q}$ in stochastic case ($\beta = 0.2$). [↑, ←, →, ↓] from top to bottom.



(a) $J^N_{\hat{\mu}^*}$,$\alpha = 0.05$, det. (b) $J^N_{\hat{\mu}^*}$,$\alpha = 0.2$, det. (c) $J^N_{\hat{\mu}^*}$,$\alpha = 0.5$, det. (d) $J^N_{\mu^*}$, det.

(e) $J^N_{\hat{\mu}^*}$,$\alpha = 0.05$, sto. (f) $J^N_{\hat{\mu}^*}$,$\alpha = 0.2$, sto. (g) $J^N_{\hat{\mu}^*}$,$\alpha = 0.5$, sto. (h) $J^N_{\mu^*}$, sto.

Figure 6 – Effect of alpha on J. $\beta = 0.2$ for stochastic case.

The larger alpha, the larger the impact of the update in the q-learning. In a deterministic setting, increasing alpha helps the algorithm converge faster from what we can see in figure 4. Indeed an $\alpha = 0.05$ might be too conservative, thus resulting in a uselessly slow convergence.

In the stochastic setting, the value of alpha has a big impact on the final result. An alpha too high might lead to divergence. From the result obtained, a value of alpha close to 0.2 seems to offer acceptable results. It would have been perhaps interesting to plot the square error with $J_{\hat{\mu}^* N}$ by varying the value of alpha. Fianally, there isn't much difference between a setting where we consider $\alpha_k = 0,\ \forall k$ and when we consider an alpha that decrease linearly towards 0.

6.4 Implement another exploration technique[1].

A widely used methods for balancing exploration/exploitation is a policy that selects at each time step a random action with a fixed probability, $0 \leq \epsilon \leq 1$, instead of selecting greedily one of the learned optimal actions with respect to the Q-function.

$$\pi(s) = \begin{cases} \text{random action from } U(s), & \text{if } \zeta < \epsilon \\ \text{argmax}_{u \in U(s)} Q(s,u), & \text{Otherwise} \end{cases} \tag{1}$$

My code was no thought from the start to handle such kind of policies (the policy parameter for my simulation is a pointer to function). All the changes are not difficult but quite time consuming. For this reason, I only implemented it (`exploratoryPolicy()`) but didn't test it. Anyway, we can see that this policy offer a tradeoff between exploration and exploitation of (detected) high-reward paths. This tradeoff can be controlled via ϵ. In the previous stages, the trajectories were everytime generated by a random policy. Using (1) with $\epsilon < 1$ would encourage the agent to exploit more paths that are (currently) thought to give better rewards. This way, we might not discover some interesting cell with high rewards. It is more conservative in a sense.

Part II

$\gamma = 0.4$

With a smaller discount factor, the agent will favour closer but perhaps smaller rewards rather than long paths leading to a future excellent reward. In the same idea, the cumulative results will be smaller as new immediate rewards will rapidly vanish.

4 Expected return of a policy

One just need to change the variable `GAMMA`.

5 Optimal policy

Considering all the plots that have been made for each report, I'll just generate the more interesting results. If the full results were expected, one just needs to change `discntFact` in the main script to the new value and the remaining is automatic. The same applies for `plot.py`.

[1] See *Adaptive ϵ-greedy Exploration in ReinforcementLearning Based on Value Differences*, Michel Tokic

```
Deterministic setting:
t=500
J:
[[10.0000 9.0907 15.2267 31.6667 31.6667]
 [10.0000 15.2267 18.0667 22.6667 31.6667]
 [10.0000 10.0907 15.2267 24.5238 22.6667]
 [10.0000 13.8095 24.5238 13.8095 24.5238]
 [10.0000 1.5238 13.8095 24.5238 15.0000]]

u:
[['down' 'down' 'down' 'righ' 'up']
 ['left' 'righ' 'righ' 'righ' 'up']
 ['up' 'righ' 'up' 'down' 'up']
 ['left' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'righ']]


Stochastic setting:
t=500
J:
[[1.3333 -0.0437 4.5302 16.6275 16.6275]
 [1.3333 4.5302 6.3235 9.4275 16.6275]
 [1.3333 0.7563 4.5302 12.3494 9.4275]
 [1.3333 3.2585 12.3494 3.2585 12.3494]
 [1.3333 -6.0506 3.2585 12.3494 4.8627]]

u:
[['down' 'down' 'down' 'righ' 'up']
 ['left' 'righ' 'righ' 'righ' 'up']
 ['up' 'righ' 'up' 'down' 'up']
 ['left' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'righ']]
```

Figure 7 – μ^* and $J_{\mu^*}^N$ for each initial state x. N=500 in both deterministic and stochastic settings.

6 Q-learning in batch setting

Like expected, the convergence arise way sooner !

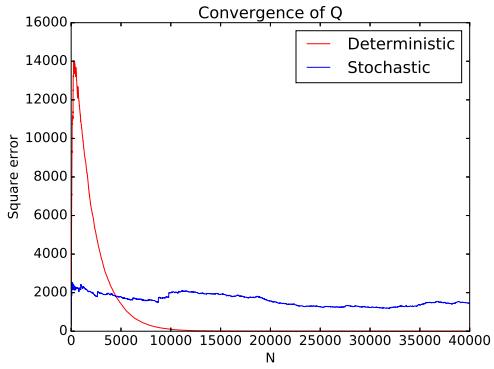


Figure 8 – Convergence of Q (equivalent of figure 1 with gamma = 0.4)

Question 6 with $\gamma = 0.4$

```
Q6: Q-learning in batch setting

*******************************

1) Simulation of q-learning algorithm, 100 episodes of length 1000, starting in [3, 3] by applying a ra

1.a) alpha = 0.05
----------------

1.a.i) Deterministic setting:
Expected cumulative reward of optimal policy:
```

```
[[10.0000 9.0907 15.2267 31.6667 31.6667]
 [10.0000 15.2267 18.0667 22.6667 31.6667]
 [10.0000 10.0907 15.2267 24.5238 22.6667]
 [10.0000 13.8095 24.5238 13.8095 24.5238]
 [10.0000 1.5238 13.8095 24.5238 15.0000]]


Corresponding optimal policy:
[['down' 'down' 'down' 'righ' 'up']
 ['left' 'righ' 'righ' 'righ' 'up']
 ['up' 'righ' 'up' 'down' 'up']
 ['left' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'down']]

1.a.ii) Stochastic setting:
Expected cumulative reward of optimal policy:
[[2.6963 1.7177 2.3660 8.8219 12.7837]
 [1.3912 1.3954 6.0665 9.8261 13.8524]
 [4.2062 2.5416 5.3797 16.9032 10.0045]
 [3.6810 6.4285 16.5621 3.7236 11.9227]
 [-1.2166 -4.7157 5.2528 10.5159 6.9900]]


Corresponding optimal policy:
[['down' 'down' 'down' 'righ' 'righ']
 ['left' 'righ' 'righ' 'righ' 'up']
 ['up' 'righ' 'down' 'down' 'up']
 ['up' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'down']]

1.b) alpha = 0.2
----------------

1.b.i) Deterministic setting:
Expected cumulative reward of optimal policy:
[[10.0000 9.0907 15.2267 31.6667 31.6667]
 [10.0000 15.2267 18.0667 22.6667 31.6667]
 [10.0000 10.0907 15.2267 24.5238 22.6667]
 [10.0000 13.8095 24.5238 13.8095 24.5238]
 [10.0000 1.5238 13.8095 24.5238 15.0000]]


Corresponding optimal policy:
[['down' 'down' 'down' 'righ' 'up']
 ['left' 'righ' 'righ' 'righ' 'up']
 ['up' 'righ' 'up' 'down' 'up']
 ['left' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'righ']]

1.b.ii) Stochastic setting:
Expected cumulative reward of optimal policy:
[[5.2661 3.0231 4.0239 13.6202 18.8060]
 [0.7489 3.1313 8.8356 10.7541 25.1118]
 [5.0085 4.8098 8.5829 20.7414 15.2833]
 [0.7045 7.0767 11.4094 4.0826 11.0682]
 [0.4795 -1.7601 7.0224 17.7679 3.9597]]


Corresponding optimal policy:
[['down' 'down' 'down' 'down' 'down']
```

```
 ['left' 'righ' 'righ' 'left' 'up']
 ['up' 'left' 'up' 'down' 'up']
 ['up' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'righ']]

1.a) alpha = 0.5
----------------

1.c.i) Deterministic setting:
Expected cumulative reward of optimal policy:
[[10.0000 9.0907 15.2267 31.6667 31.6667]
 [10.0000 15.2267 18.0667 22.6667 31.6667]
 [10.0000 10.0907 15.2267 24.5238 22.6667]
 [10.0000 13.8095 24.5238 13.8095 24.5238]
 [10.0000 1.5238 13.8095 24.5238 15.0000]]


Corresponding optimal policy:
[['down' 'down' 'down' 'righ' 'up']
 ['left' 'righ' 'righ' 'righ' 'up']
 ['up' 'righ' 'up' 'down' 'up']
 ['left' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'righ']]

1.c.ii) Stochastic setting:
Expected cumulative reward of optimal policy:
[[7.7330 6.4023 8.2341 30.8754 30.0656]
 [5.0079 9.2056 12.6448 14.7653 21.5975]
 [9.2723 6.5256 11.0820 22.8140 -1.4815]
 [1.1520 12.0694 22.8239 11.2308 19.4273]
 [3.4269 -1.5160 10.3606 22.7875 14.9768]]


Corresponding optimal policy:
[['down' 'down' 'righ' 'righ' 'righ']
 ['righ' 'righ' 'righ' 'righ' 'righ']
 ['down' 'righ' 'down' 'down' 'up']
 ['up' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'righ']]

1.a) alpha decreasing linearly
----------------

1.c.i) Deterministic setting:
Expected cumulative reward of optimal policy:
[[10.0000 9.0907 15.2267 31.6667 31.6667]
 [10.0000 15.2267 18.0667 22.6667 31.6667]
 [10.0000 10.0907 15.2267 24.5238 22.6667]
 [10.0000 13.8095 24.5238 13.8095 24.5238]
 [10.0000 1.5238 13.8095 24.5238 15.0000]]


Corresponding optimal policy:
[['down' 'down' 'down' 'righ' 'up']
 ['left' 'righ' 'righ' 'righ' 'up']
 ['up' 'righ' 'up' 'down' 'up']
 ['left' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'righ']]
[[1.0000 4.6363 1.0907 12.6667 31.6667]
 [1.0000 4.6363 1.0907 12.6667 31.6667]
```

```
 [10.0000 9.0907 15.2267 18.0667 22.6667]
 [9.0000 -3.9637 10.0907 10.8095 1.0667]
 [10.0000 -3.4762 13.8095 24.5238 4.8095]]

[[1.0000 1.0000 4.6363 1.0907 12.6667]
 [10.0000 10.0000 9.0907 15.2267 18.0667]
 [9.0000 9.0000 -3.9637 10.0907 10.8095]
 [10.0000 10.0000 -3.4762 13.8095 24.5238]
 [-16.0000 -16.0000 -16.3905 1.5238 6.8095]]

[[4.6363 1.0907 12.6667 31.6667 31.6667]
 [9.0907 15.2267 18.0667 22.6667 22.6667]
 [-3.9637 10.0907 10.8095 1.0667 1.0667]
 [-3.4762 13.8095 24.5238 4.8095 4.8095]
 [-16.3905 1.5238 6.8095 15.0000 15.0000]]

[[10.0000 9.0907 15.2267 18.0667 22.6667]
 [9.0000 -3.9637 10.0907 10.8095 1.0667]
 [10.0000 -3.4762 13.8095 24.5238 4.8095]
 [-16.0000 -16.3905 1.5238 6.8095 15.0000]
 [-16.0000 -16.3905 1.5238 6.8095 15.0000]]


1.c.ii) Stochastic setting:
Expected cumulative reward of optimal policy:
[[8.8947 2.7781 10.1388 25.2769 31.5251]
 [7.2838 9.4973 14.6396 22.0269 31.0267]
 [8.8095 7.3726 8.9653 14.5219 22.1129]
 [8.5267 4.2773 10.9394 9.6199 14.3541]
 [0.7482 -2.0071 0.5917 22.6793 14.4993]]


Corresponding optimal policy:
[['down' 'down' 'down' 'righ' 'up']
 ['left' 'righ' 'righ' 'righ' 'up']
 ['up' 'righ' 'down' 'down' 'up']
 ['up' 'righ' 'righ' 'left' 'down']
 ['up' 'righ' 'up' 'up' 'righ']]
Deterministic: Batch1
Expected cumulative reward of optimal policy:
[[10.0000 9.0907 15.2267 31.6667 31.6667]
 [10.0000 15.2267 18.0667 22.6667 31.6667]
 [10.0000 10.0907 15.2267 24.5238 22.6667]
 [10.0000 13.8095 24.5238 13.8095 24.5238]
 [10.0000 1.5238 13.8095 24.5238 15.0000]]


Corresponding optimal policy:
[['down' 'down' 'down' 'righ' 'righ']
 ['left' 'righ' 'righ' 'righ' 'up']
 ['up' 'righ' 'up' 'down' 'up']
 ['left' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'down']]
Stochastic: Batch1
Expected cumulative reward of optimal policy:
[[3.7588 -1.6807 3.4400 15.7522 11.7797]
 [2.1960 3.4533 2.6733 12.9969 14.1840]
 [4.1975 0.5979 1.7023 12.3552 8.3686]
 [0.1077 3.6132 11.0404 1.3677 15.8848]
 [1.4705 -5.6387 2.4179 16.2660 7.0704]]
```

```
Corresponding optimal policy:
[['down' 'down' 'down' 'righ' 'righ']
 ['down' 'righ' 'righ' 'righ' 'up']
 ['down' 'left' 'down' 'down' 'up']
 ['up' 'righ' 'righ' 'up' 'left']
 ['up' 'righ' 'up' 'up' 'righ']]
Deterministic: Batch10
Expected cumulative reward of optimal policy:
[[10.0000 9.0907 15.2267 31.6667 31.6667]
 [10.0000 15.2267 18.0667 22.6667 31.6667]
 [10.0000 10.0907 15.2267 24.5238 22.6667]
 [10.0000 13.8095 24.5238 13.8095 24.5238]
 [10.0000 1.5238 13.8095 24.5238 15.0000]]


Corresponding optimal policy:
[['down' 'down' 'down' 'righ' 'up']
 ['left' 'righ' 'righ' 'righ' 'up']
 ['up' 'righ' 'up' 'down' 'up']
 ['left' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'righ']]
Stochastic: Batch10
Expected cumulative reward of optimal policy:
[[1.8172 -0.6248 2.2034 11.8197 17.0160]
 [2.1408 5.0508 2.1100 6.6810 9.6285]
 [2.7964 1.6744 5.1554 9.9312 6.2705]
 [4.4155 7.8763 16.2130 5.1266 11.1017]
 [6.2715 -5.2509 3.4604 9.7152 7.0726]]


Corresponding optimal policy:
[['down' 'down' 'down' 'righ' 'up']
 ['righ' 'righ' 'righ' 'left' 'up']
 ['down' 'righ' 'up' 'down' 'up']
 ['up' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'down']]
Deterministic: Batch100
Expected cumulative reward of optimal policy:
[[10.0000 9.0907 15.2267 31.6667 31.6667]
 [10.0000 15.2267 18.0667 22.6667 31.6667]
 [10.0000 10.0907 15.2267 24.5238 22.6667]
 [10.0000 13.8095 24.5238 13.8095 24.5238]
 [10.0000 1.5238 13.8095 24.5238 15.0000]]


Corresponding optimal policy:
[['down' 'down' 'down' 'righ' 'up']
 ['left' 'righ' 'righ' 'righ' 'up']
 ['up' 'righ' 'up' 'down' 'up']
 ['left' 'righ' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'righ']]
Stochastic: Batch100
Expected cumulative reward of optimal policy:
[[2.4465 0.2912 4.7482 17.0682 21.8820]
 [3.4003 3.2955 5.4534 11.7766 22.9268]
 [4.0921 1.5500 4.5114 9.0486 11.3238]
 [1.7023 0.9362 14.0407 5.0345 12.1158]
 [0.8815 -6.0684 3.1835 16.3987 6.3874]]
```

```
Corresponding optimal policy:
[['down' 'down' 'down' 'righ' 'righ']
 ['left' 'righ' 'righ' 'righ' 'up']
 ['down' 'left' 'up' 'down' 'up']
 ['left' 'left' 'righ' 'left' 'left']
 ['up' 'righ' 'up' 'up' 'down']]
```
