TBMI26: A4

A report by

Adam Nyberg Rasmus Johns
Adany869 Rasjo813

Theory

Different parts of Q-learning

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

The components of the Q-learning in our implementation was:

- State: The different states in the state space.
- Actions: The actions which can be performed.
- Q-lookup table: The combination of state and actions. “If in state S, perform action A”.
- Reward: The calculated value used to incentivize good behaviour.
- Alpha: The rate at which old knowledge will be replaced by new knowledge.
- Gamma: How long- /short-sighted the program will be.
- Max QValue: The value multiplied by gamma, resulting in a weight representing the future.

Before applying Q-learning, we had to prevent the robot from leaving the world. To do so, the program sets all outbounding actions adjacent to corresponding borders to -Inf penalty.

Describe the differences between the worlds explored by the robot. Any surprises?

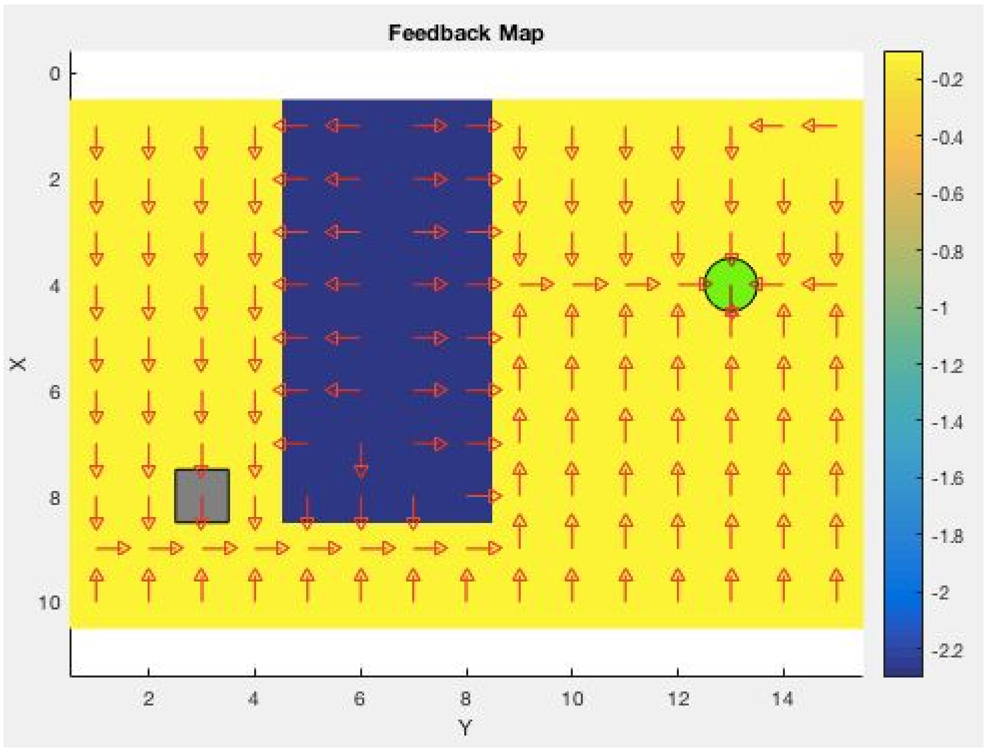
Difference between world 1 and 2 are only that world 2 is not deterministic i.e. have some noise introduced. World 3 and 4 are a little different because they have a narrow path in between water that is optimal. The difference between world 3 and 4 are that in world 4 the correct action is not always applied and therefore noise added. Not really any surprises between the worlds**.**

World 1

We used the following parameters:

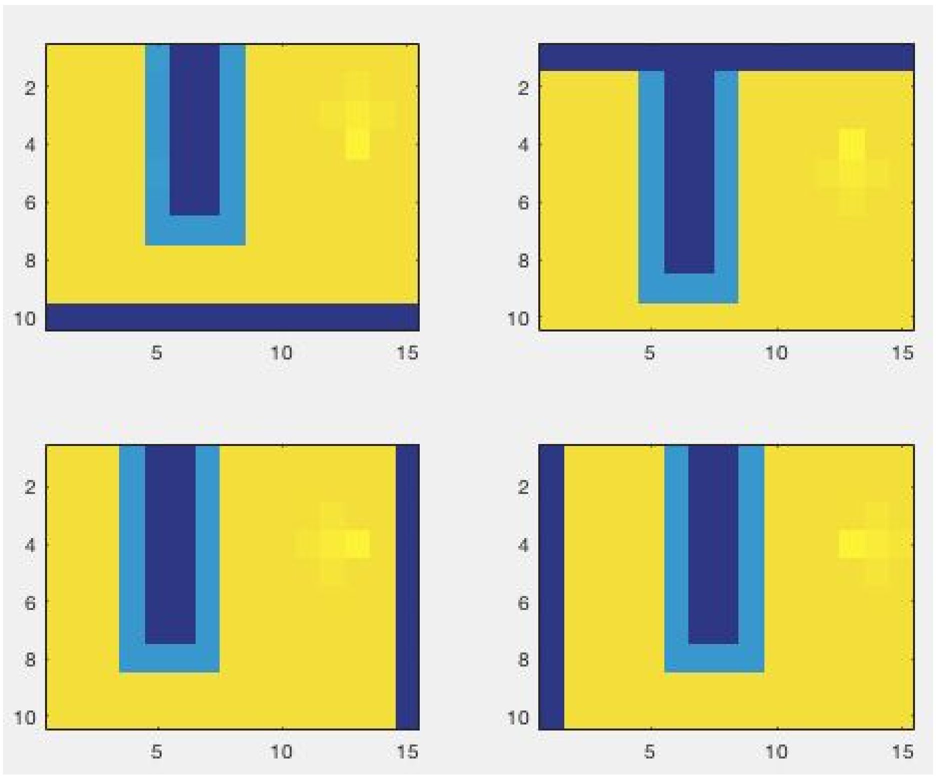
Alpha	**Gamma**	**Epsilon**	**Total iterations**	**Goal iterations**
1	0.5	0.4	26 885	1 000

Because we knew that this world is fully deterministic we set alpha=1. We then got the following V-function.



V-function

We can also see our corresponding Q-function below.



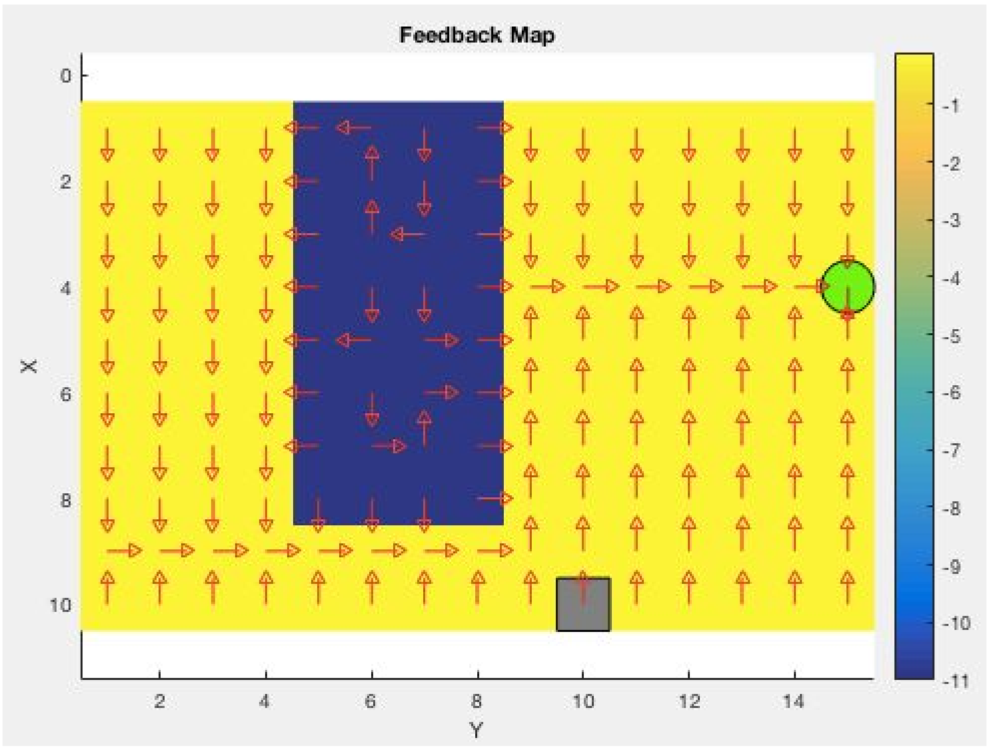
Q-function

World 2

For this world, we set alpha to a slightly lower value in order to counter the randomness in the world. We used the following parameters:

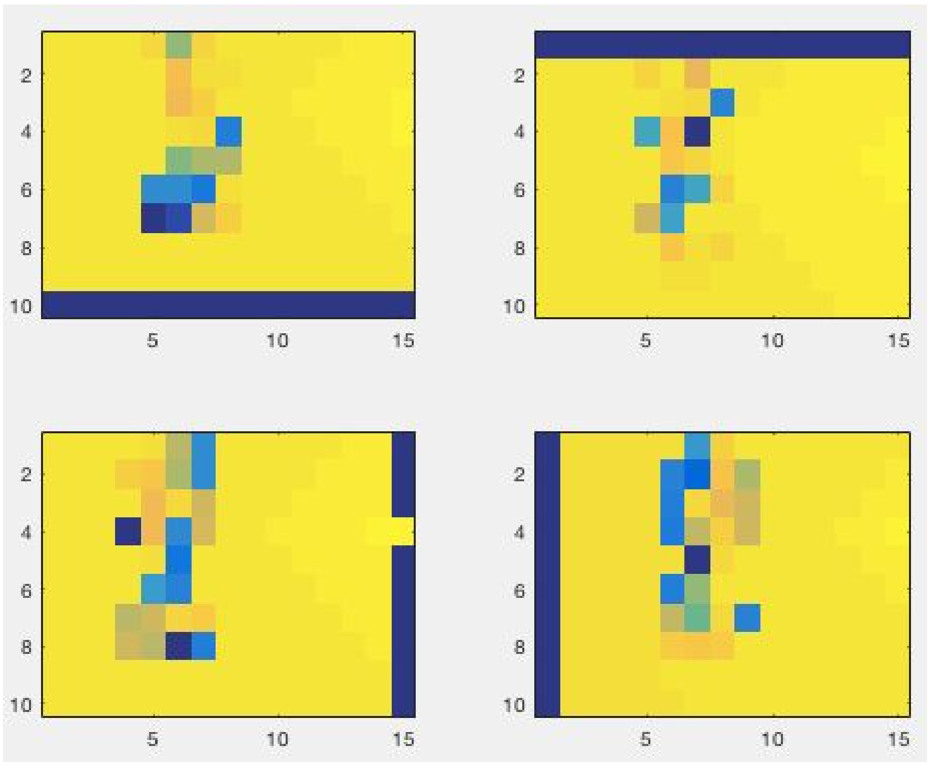
Alpha	Gamma	Epsilon	Total iterations	Goal iterations
0.7	0.8	0.3	950 526	4 000

That gave us the following V-function.



V-function

Below we have the Q-function plotted for all positions and directions. The blue border indicates the direction.



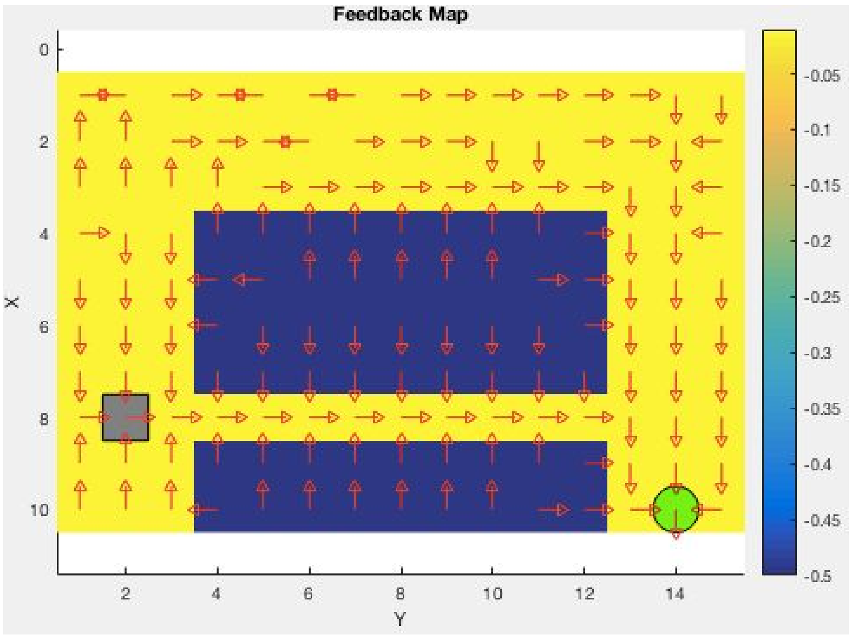
Q-function

Dijkstra’s pathfinding algorithm finds the cheapest possible path (if it succeeds finding a path). However, too much noise (as seen in Suddenly Irritating Blob), could cause the algorithm to fail. Static worlds should be navigated successfully with Dijkstra’s shortest path finding algorithm.

World 3

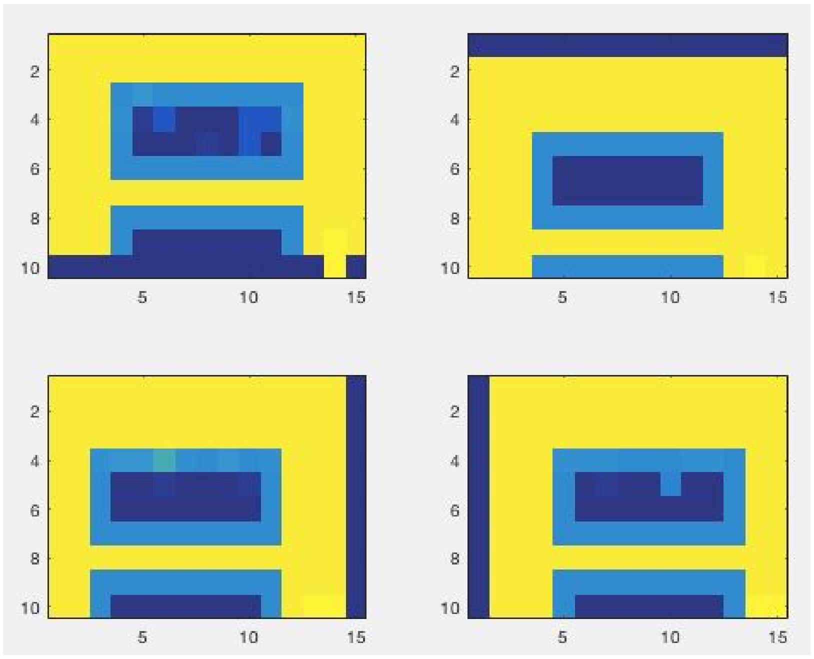
Since this world is deterministic we used a high alpha.

Alpha	Gamma	Epsilon	Total iterations	Goal iterations
0.8	0.4	0.3	255 302	10 000



V-function

We think the reason that we have some colliding arrows in the top left corner is because that area is not being explored as much as the other parts of the world.



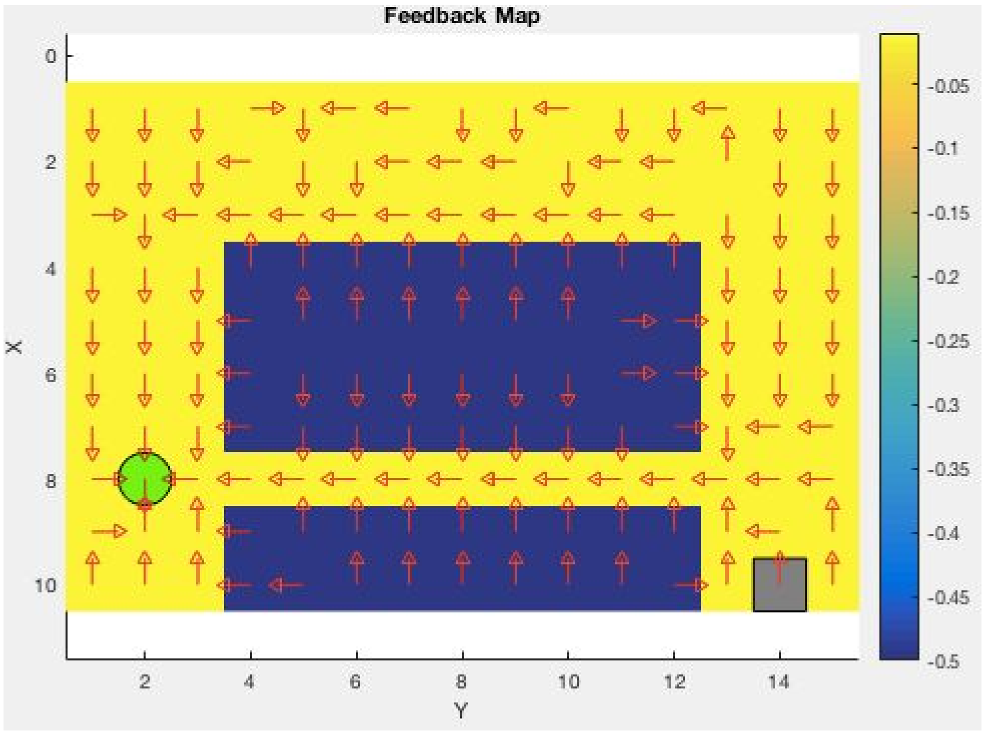
Q-function

World 4

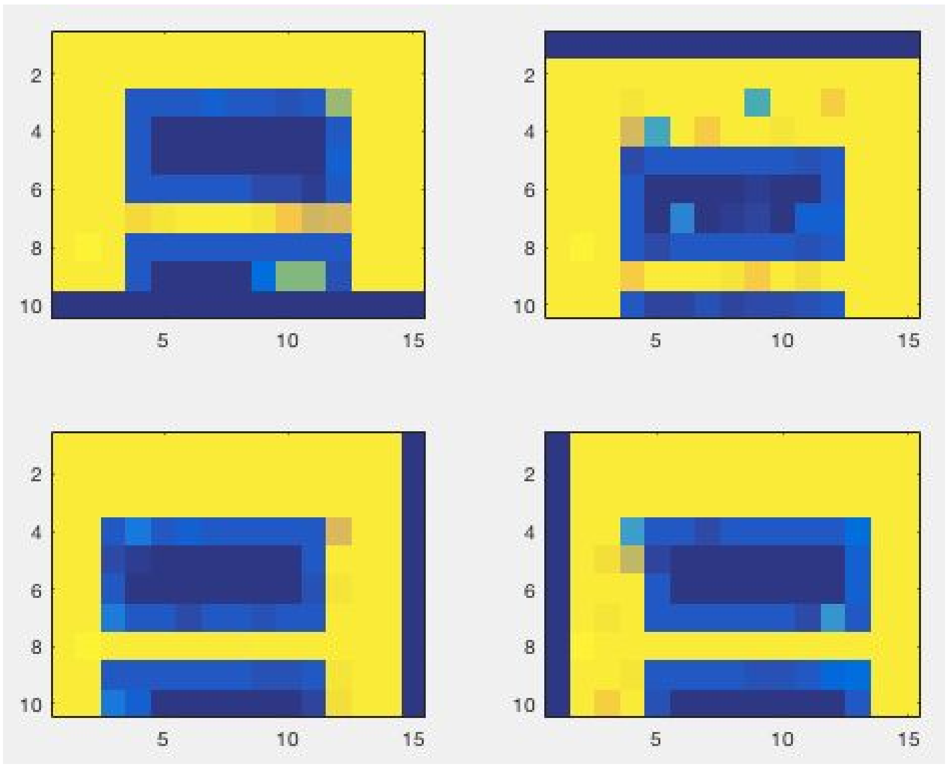
To the naked eye, this world looks similar to World 3. The only clear difference is that the agent and the goal has switched spawning positions. However, in this world, there is a giant underlying difference: every third actions the agent performs, it runs the risk of randoming another action. When this happens, the agent is unaware and believes it has executed the instructed action.

In order to make our agent “smarter” than that, we made it check its position relative to the action it wanted to perform. If the desired action combined with the agent’s last position does not match the agent’s current position, meaning the agent believes something to have gone wrong, it simply does not update its Q-table. By doing this, the agent at least keeps a sane mind by finding the best way (disregarding the randomness of the world)..

Alpha	Gamma	Epsilon	Total iterations	Goal iterations
0.6	0.1	0.25	178 823	5 000



V-function



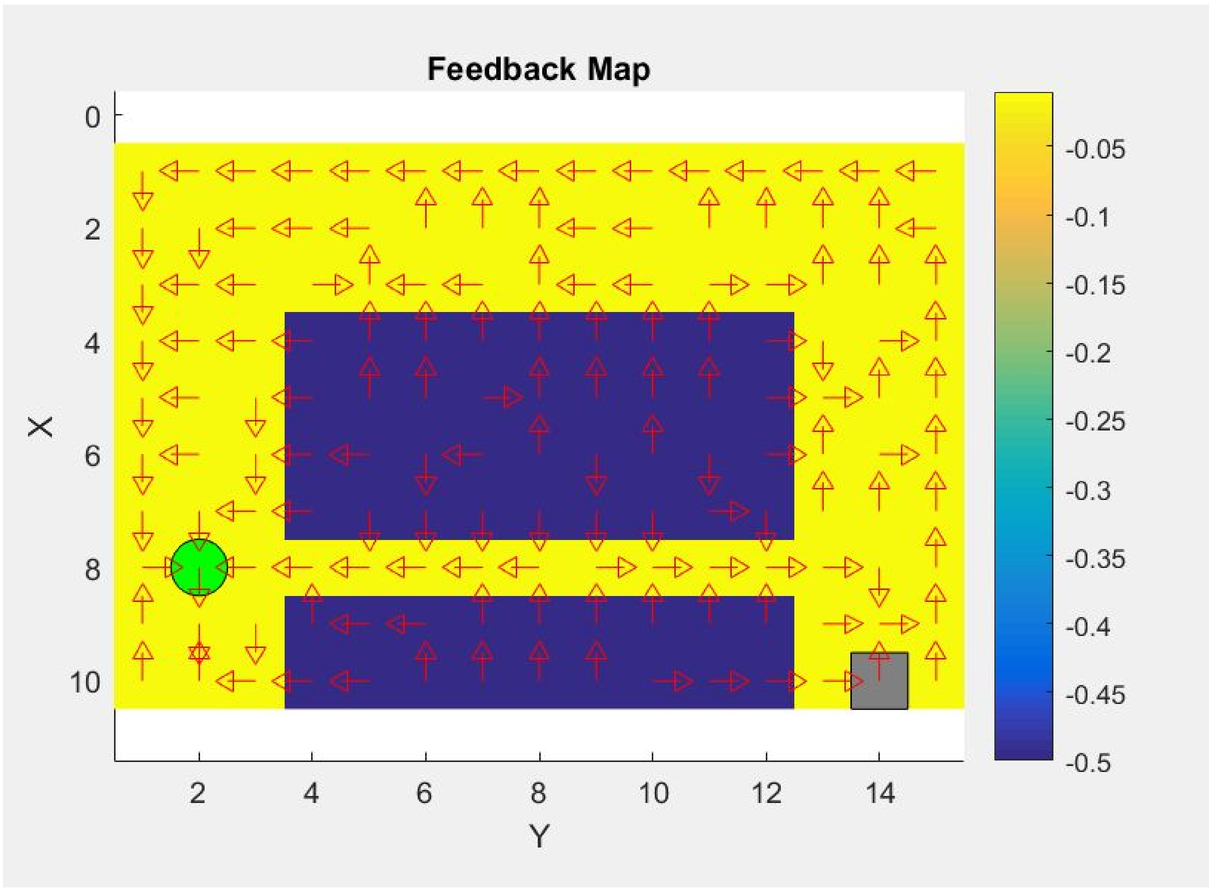
Q-function

The problem with this approach is that the agent find a “super path” which goes between the water. In reality, this path is terrible and dangerous. After all, there is a huge risk that the agent performs an incorrect action and steps out into the water if it takes that path.

If we do not implement this solution to the stochastic problem, the agent never takes the path in between the water. Instead, it prioritizes its exit out of the water as fast as possible.

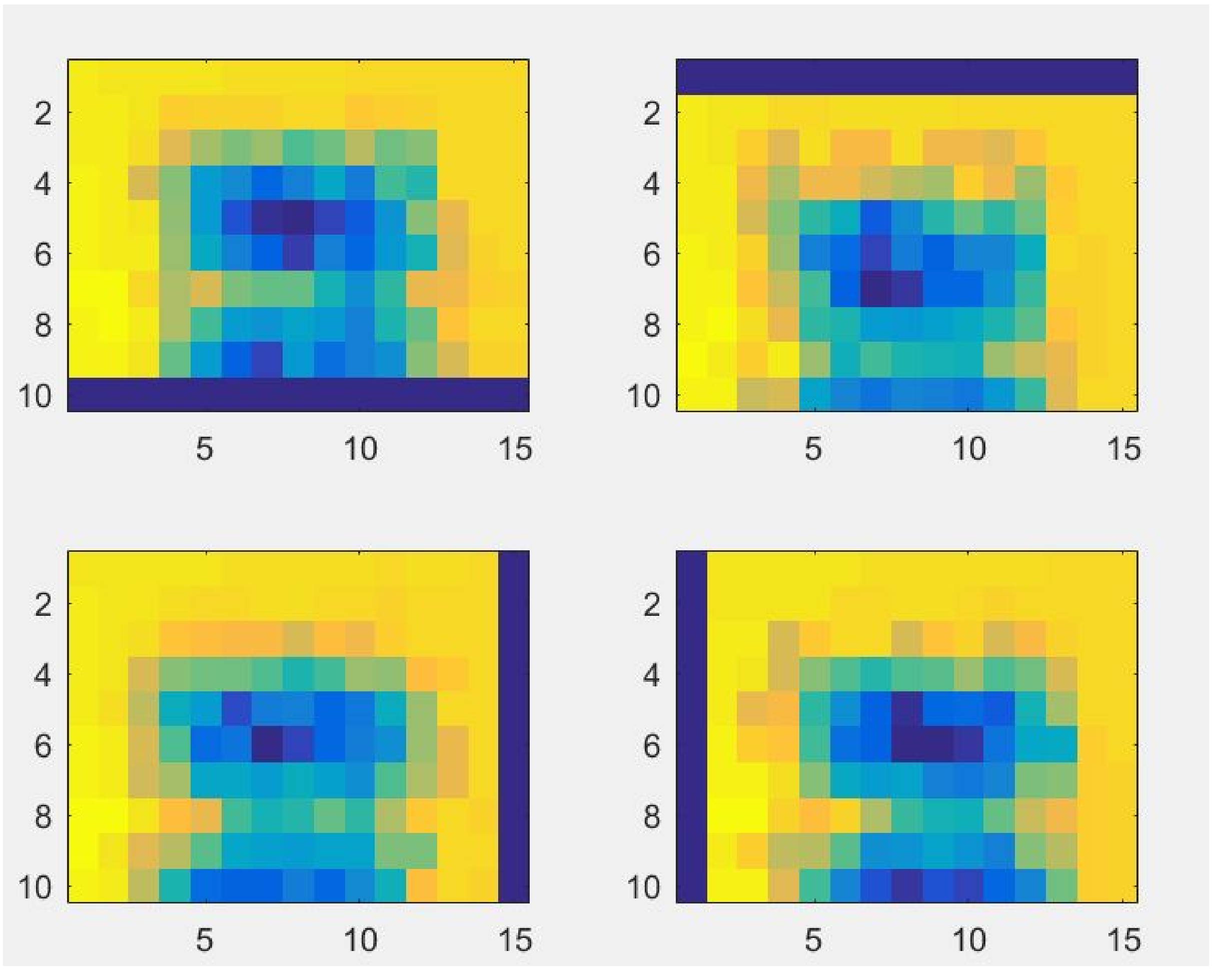
For this test, we used the following variables:

Alpha	**Gamma**	**Epsilon**	**Total iterations**	**Goal iterations**
0.3	0.95	Decreasing from 0.9 to 0.2 at an exponential rate	707 038	5 000



Feedback Map
X
Y
0
2
4
6
8
10
2
4
6
8
10
12
14
-0.05
-0.1
-0.15
-0.2
-0.25
-0.3
-0.35
-0.4
-0.45
-0.5

V-function



Q-function

General

A problem which can be discretized into states and a good reward function can possibly be solved with reinforcement learning. For example, many computer games offer such premises. Computer games often have a clear scoring system or goal, which can be a strong component in a reward function, and can often be split into well defined states.

Conclusions

This lab highlights many interesting aspects of Q-learning. It shows that a problem which can be represented by states (position in a grid in our case) and has some sort of feedback function can successfully be solved by reinforcement learning. We also saw the importance of the different components of Q-learning. Even more important, we saw how crucial it is to tweak these values according to one’s knowledge of the specific problem: in a random world, alpha should be lowered; in a world where a greedy approach pays off, beta should be lowered; in a world where the agent performs a mission it has already completed several times before, perhaps the learning rate should be tuned down to a really low number.

The lab also shows that the V-function is just a maximization of the Q-function for every position.