Assignment _

Autonomously Learning Systems, WS2021/22

Team Members		
Last name	First name	Matriculation Number
Gaisberger	Patrick	11703581
Hamedl	Tobias	11808141

Task 1

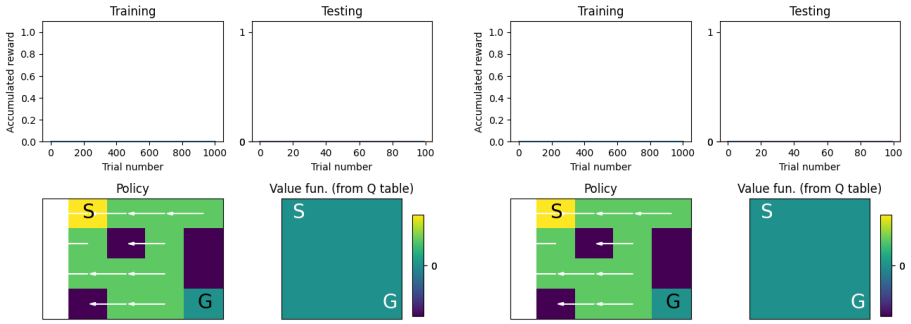
a)

See code.

b)

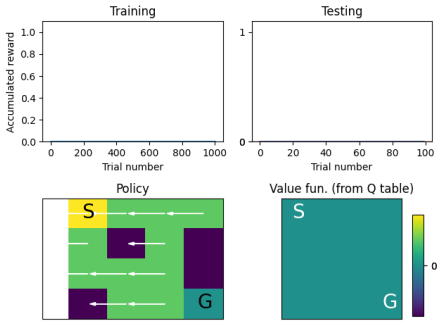
See code.

c)

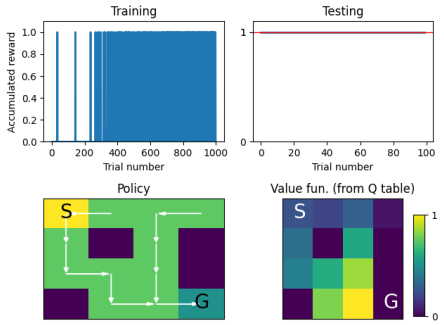


(a) $\epsilon = 0$

(b) $\epsilon = 0.01$



(c) $\epsilon = 0.1$

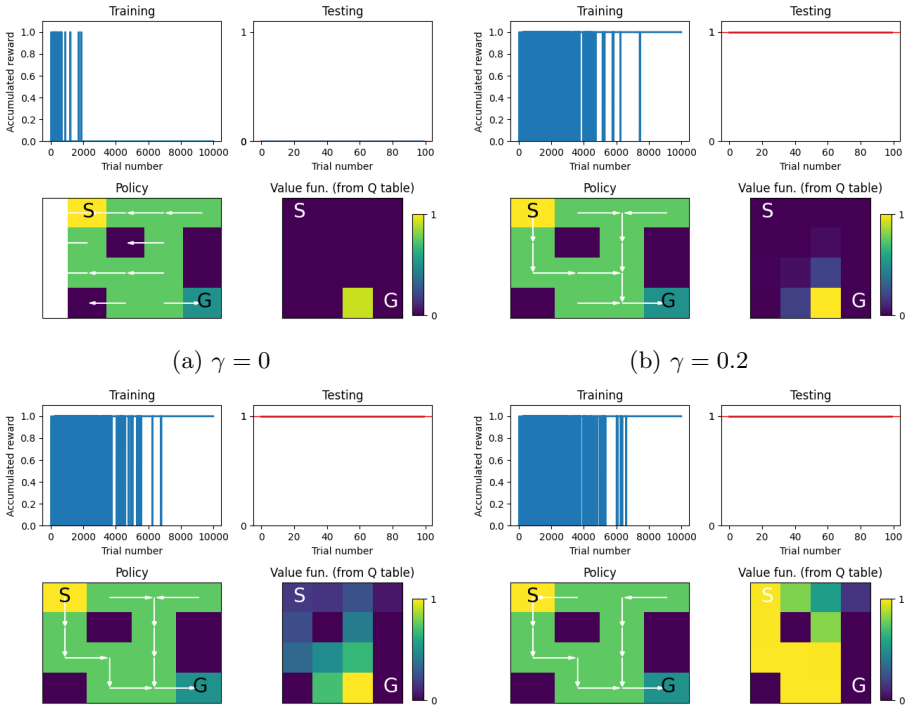


(d) $\epsilon = 1$

Figure 1: Frozen lake with different ϵ

In this case the best value for ϵ is 1, because for all other epsilons the average accumulated reward is mostly 0. The smaller the ϵ, the smaller the chance that a random action will be chosen. If we set our ϵ to 0 our action will be chosen deterministically every time. Hence, we won't explore anything. And for the small ones, the agent won't explore enought to solve the environment. One way to solve this environment with an *epsilon* $= 0$ would be to choose a stochastic policy. For environments with a randomness factor which forces the agent to

explore it is also possible to solve it with a 0 epsilon (No epsilon greedy).



(a) $\gamma = 0$

(b) $\gamma = 0.2$

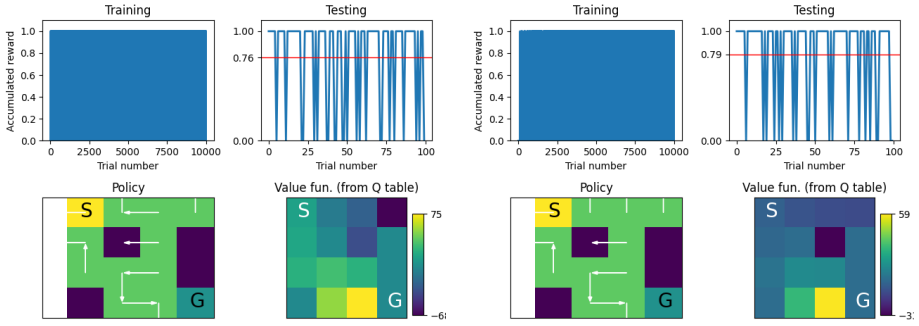
(c) $\gamma = 0.7$

(d) $\gamma = 1$

Figure 2: Frozen lake with $\epsilon = 1$ and different γ

We always get a working policy except with $\gamma = 0$. The reason why we get a working policy with $\gamma = 1$, is that we don't compute the exact Q-table (we converge towards it), hence we get greater Q-values for state-actions which are leading to the goal.

d)



(a) $\epsilon = 0.1$ and $\gamma = 1$ (b) $\epsilon = 1$ and $\gamma = 0.9$

Figure 3: Frozen lake with different ϵ and γ with slippery floor

The training performance will change, but the best policy will be found (even with $\epsilon = 0$). Because in this example the environment itself provides the randomness to explore the map. Giving a big negative-reward (punishment) prevents the agent from choosing the action, which led to this reward, again. The slightly negative reward prevents the agent from circling. The big positive reward leads the agent to choose the action again. (Except during the exploring phase, in which the agent chooses most of its actions randomly)

e)

$$
\begin{aligned}
\alpha &= 3e-1 \\
\epsilon &= 1.0 \\
\gamma &= 1.0 \\
\alpha_{decay} &= 0.9995 \\
\epsilon_{decay} &= 0.99991 \\
max\_train\_iterations &= 5000 \\
max\_test\_iterations &= 100 \\
max\_episode\_length &= 400
\end{aligned}
\tag{1}
$$

Result:

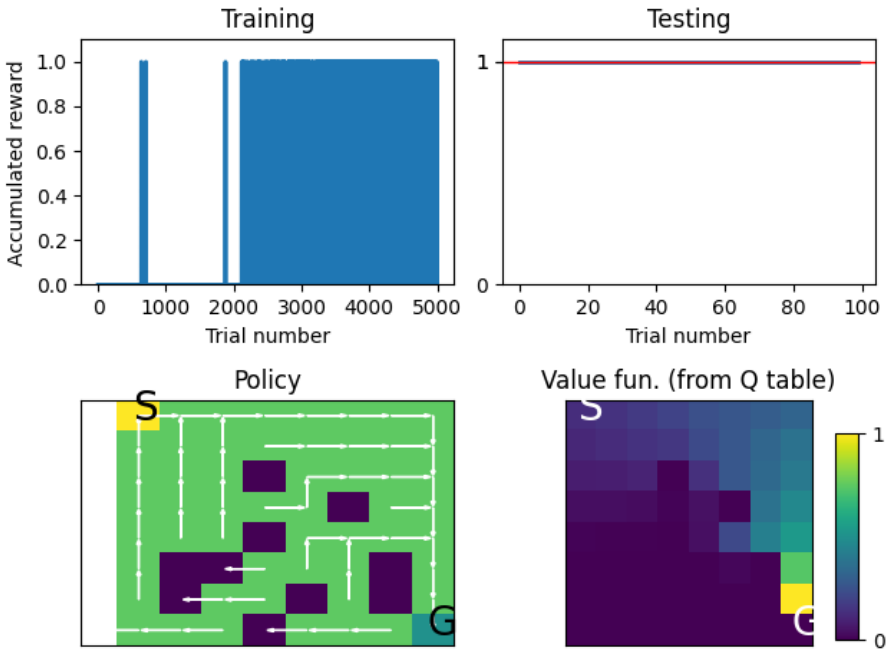


Figure 4: Frozen lake with optimized parameters and with a 8x8 map

As we can see the algorithm does not find the solution for all states. This is because these states are too unlikely to reach from the start position and are therefore never updated.

1 Task 2

a)

Given Formulas:

$$\begin{aligned} Q(s,a) &\leftarrow Q(s,a) + \alpha(y - Q(s,a)) \\ E &= \frac{1}{2}\left(Q(s,a) - y\right)^2 \end{aligned} \tag{2}$$

Gradient Descent Update:

$$\theta \leftarrow \theta - \alpha * \frac{\mathrm{d}E}{\mathrm{d}\theta} \tag{3}$$

$$Q_\theta(s,a) = \theta_a^T s \tag{4}$$

Let A be the **Action space**.

Solution for (with respect to θ) constant y

Since the partial derivation $\forall t \in A; t \neq a : \frac{\partial E}{\partial \theta_t} = 0$ we can simplify the GD update and only look at θ_a:

$$\begin{aligned} \textbf{GD: } \theta_a &\leftarrow \theta_a - \alpha * \frac{\mathrm{d}E}{\mathrm{d}\theta_a} \\ \frac{\partial E}{\partial \theta_a} &= \frac{1}{2}\left(Q_\theta(s,a) - y\right)^2 \frac{\partial}{\partial \theta_a} \\ &= (Q_\theta(s,a) - y)\frac{\partial Q_\theta(s,a)}{\partial \theta_a} \\ &= (Q_\theta(s,a) - y)s \\ \Rightarrow \theta_a &\leftarrow \theta_a - \alpha(Q_\theta(s,a) - y)s \end{aligned} \tag{5}$$

Solution for $y = r + \gamma Q_\theta(s', a')$:

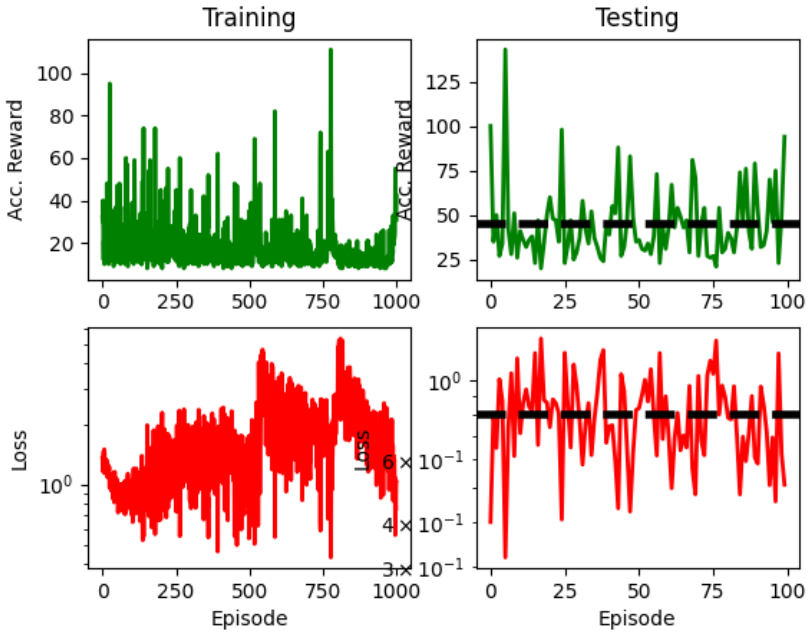
We assume that θ_a and θ'_a are independent. Hence, the partial derivative for θ_a stays the same and we look at $\theta_{a'}$:

$$\begin{aligned} \textbf{GD: } \theta_{a'} &\leftarrow \theta_{a'} - \alpha * \frac{\mathrm{d}E}{\mathrm{d}\theta_{a'}} \\ \frac{\partial E}{\partial \theta_{a'}} &= \frac{1}{2}\left(Q_\theta(s,a) - y\right)^2 \frac{\partial}{\partial \theta_{a'}} \\ &= (Q_\theta(s,a) - y)\frac{\partial - y}{\partial \theta_{a'}} \\ &= -(Q_\theta(s,a) - y)s' \\ \Rightarrow \theta_a &\leftarrow \theta_{a'} + \alpha(Q_\theta(s,a) - y)s' \end{aligned} \tag{6}$$

If y is assumed to be constant (first statement) with respect to θ the statement becomes true, since in the standard algorithms we update the Q table of the current state and action pair and not the future one.

b)

Linear SARSA cart-pole

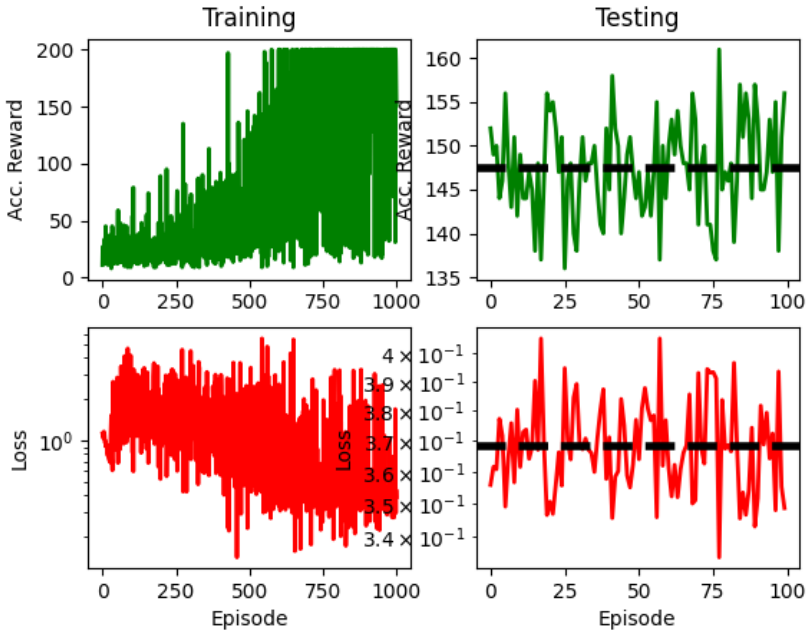


(Well there was no implementation of the linear model there, hence we implemented it) The code works almost the same as the code of task-1. Instead of implementing the update function, we provide an error-function and use pytorch to calculate the derivation. As we can see the algorithm does not perform very well due to the linear model (One could prob. adapt the reward function to solve this problem with this algorithm). Regarding the question that was a question written within the code, **"Detach the gradient of Q(s', a'). Why do we have to do that?"** Because as stated in the previous sub-task we would update Q also for the future state (with the value of the current one) and we wouldn't perform Q learning anymore (and the system would prob. go unstable). Hence, we have to detach it.

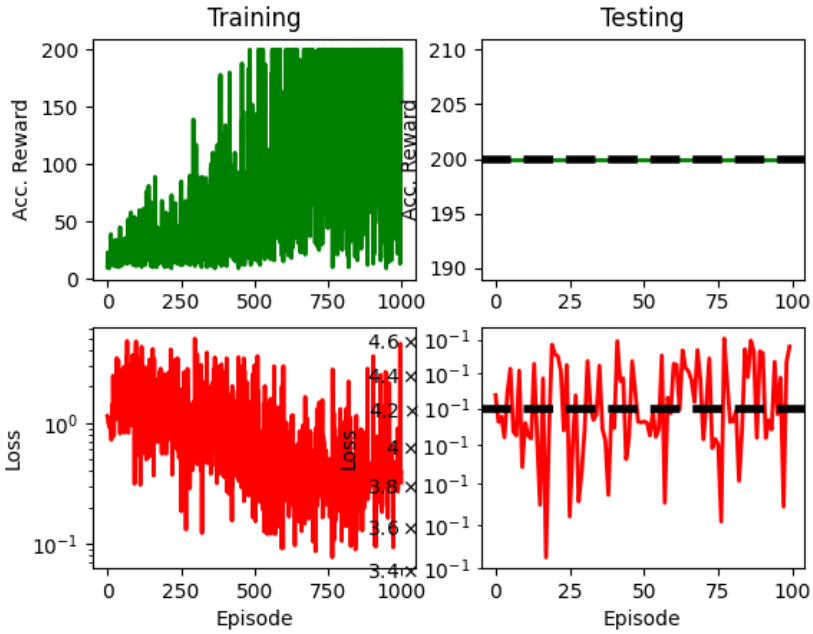
c)

We implemented the network as described in the assignment and tried different parameters.

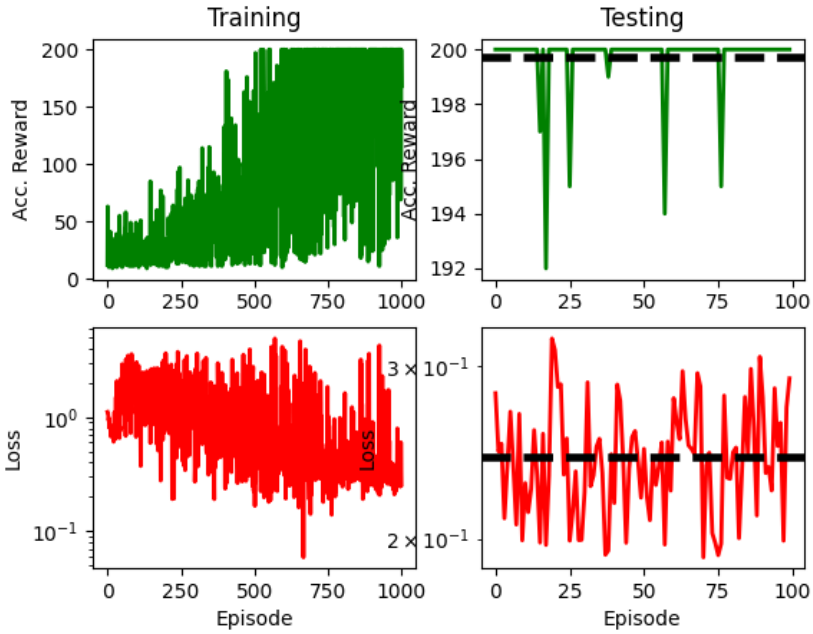
Result with the given parameters:



Result with the given parameters, but $\alpha = 3 * 10^{-3}$:



Result with the given parameters, but instead of ReLU Tanh:



Running the algorithm a few times once it encountered bad luck and didn't learn enough.

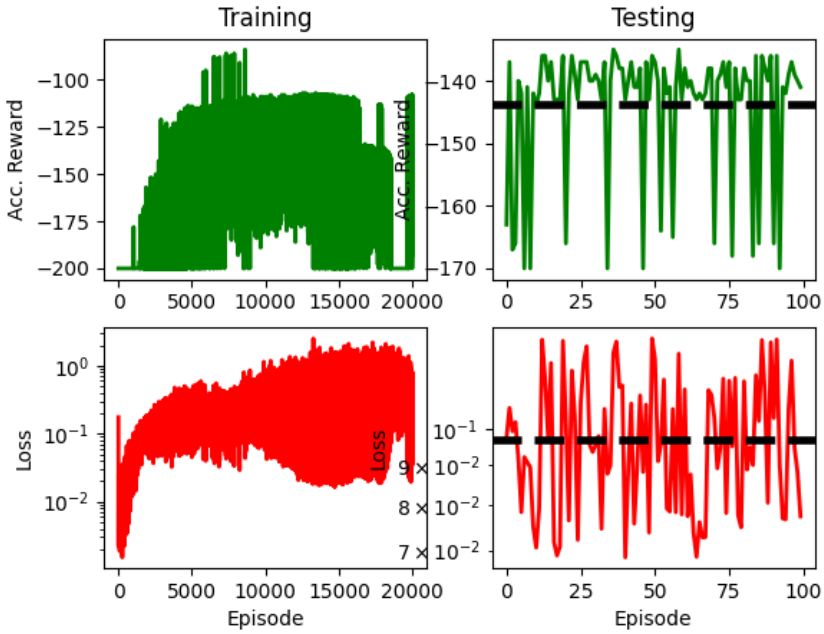
Conclusion

Changing the parameters and the activation function didn't have a big impact on the outcome. But, increasing alpha (learning rate for the optimizer) had a slightly positive impact.

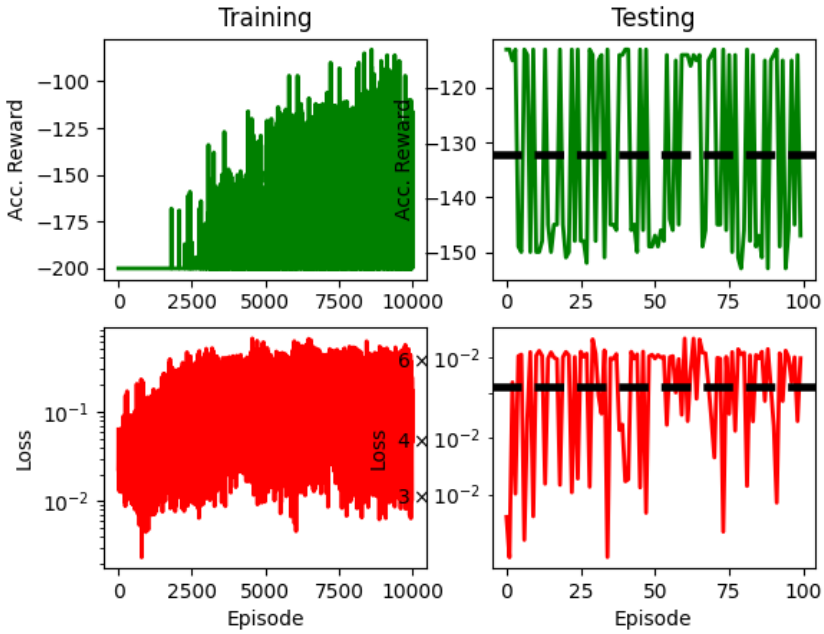
d) Solving Mountain Car-V0 with Q Learning

File names: ex2_mountain_car_non_lin_Q.py and ex2_mountain_car_linear_Q To solve this task (in a reasonable time), we had to adapt the reward function. (See code) The environment was tricky because it gives only a non-negative reward when the position is 0.5. But obviously, when the position is greater we reached the goal. Hence, we rewarded the algorithm for reaching a state >= 0.5. Another crucial point was include the velocity, we only applied the current position to get a slightly better approximation of the Q-model with the original reward. (Another possibility would be to save the model and change the reward back to its original and then train again with a tiny epsilon). We changed the parameters to explore a bit longer, since it is very unlikely to reach the goal with random actions. For convenience the original reward was plotted.

Linear Q


Training
Testing
Acc. Reward
−100
−125
−150
−175
−200
−140
−150
−160
−170
0
5000
10000
15000
20000
0
25
50
75
100
Loss
10^{0}
10^{-1}
10^{-2}
9×10^{-2}
8×10^{-2}
7×10^{-2}
Episode
Episode

Non Linear Q



As we can see the non-linear model performed much better (In average 10 iterations less to reach the goal). To achieve better results we could try state replay or maybe discretetizing the state.