UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR ROBOTICS
AND COGNITIVE SYSTEMS

Course: Reinforcement Learning (SS2021)
Dr. Christoph Metzner & Honghu Xue, M.Sc.
Assignment Report Nr. II
Date: 6. Juli 2021

Team Nr. 007
Robin Luckey
Moritz Gerwin
Frederic Dlugi
Franek Stark
Contact: frederic.dlugi@student.uni-luebeck.de

1 Task: Theoretical understanding on Function Approximation approaches

1.1

1.1.1 When using function approximation, what are the conditions for convergence? (Hint: Please consider the case of on/off-policy, tabular case/function approximation, bootstrapping/Monte-Carlo estimation)

Function approximation minimizes an error between a target function $f : S \to A$ and a model-function with specified parameters $f_m : S \times W \to A$. There are multiple methods for function approximation, for example: neural networks.
Conditions for convergence:
Instability and divergence can arise, if all elements of the following so called "Deadly Triad" are combined:

- Function approximation e.g. using ANNs.
- Bootstrapping (Using estimates to update targets), e.g. TD methods. MC-Methods do not have this problem.
- Off-policy training e.g. in dynamic programming, which is off-policy

Further more, in general these conditions must be met:

- The model function needs to be sufficiently complex.
- There need to be sufficient data in the training process. (Long enough training and long enough episodes)

on/off-policy
In on- and off-policy cases alike all inputs and outputs of the function need to be observed to perfectly converge to the correct function. Moreover a function can only be approximated for observed values of the function. This does present a problem especially for off-policy since the exploration of the function depends on the behaviour-policy. In on-policy processes the exploration of the input space needs to be implemented specifically in the training process. In the off-policy case, importance sampling, using the state-action marginals and only using the state part using the chain rule can help since otherwise the importance weight decrease exponentially.

tabular case/function approximation
Approximating a tabular function with discrete output values can be as hard as a continuous function approximation (tabular case=classification in ML). The difference between the continuous function case and the tabular function problem is to find similarities in the input and give a similar output for similar inputs. In contrast to the tabular case value based methods and Q-iteration do generally not converge to an optimal solution in the non tabular case. (Lecture 10)

bootstrapping/Monte-Carlo estimation
Bootstrapping methods like semi-gradient TD(0) do not have a convergence guarantee, where gradient-MC is guaranteed to converge to a local minimum when approximating a function. The difference between these two methods is, that semi-gradient-TD(0) updates the weights for the model function after each step in the episode and therefore has a high bias. Gradient-MC on the other hand updates the weights for the model function after the episode and therefore has a lower bias.

1.1.2 Does the algorithm Deep Q-Network (DQN) has convergence guarantee?

As in all neural-network based algorithms the function approximation convergence can only be determined for a specified function with a specified network. Therefore no general convergence guarantee is given for DQN. But any non-linear function can be approximated through a sufficiently large (deep/wide) neural network. Therefore it is possible to converge to any function with an infinitely large neural network in DQN. "This suggests that, despite lacking any theoretical convergence guarantees, our method is able to train large neural networks using a reinforcement learning signal and stochastic gradient descent in a stable manner."[1]

1.2

1.2.1 How do semi-gradient methods differ from true gradient descent when updating the weights w in function approximation?

In function approximation we try to estimate the state value function by reducing the MSE in respect to an estimate of said value function:

$$w_{t+1} = w_t + \alpha * [U_t - \hat{v}(S_t, w_t)] \nabla \hat{v}(S_t, w_t) \tag{1}$$

If the estimate U_t is unbiased the stochastic gradient descent is guaranteed to converge to a local minimum. The main difference is that in true gradient descent methods, the value function is independent of the weights. This is not the case with semi-gradient methods, as they are using a biased estimate of the value function. E.g. Bootstrapping Methods are called semi gradient descent: They take into account the effect of changing the weight vector w_t on the estimate, but ignore its effect on the target vector w_t, which implies that they will be biased and that they will not produce a true gradient-descent method. They do converge reliably in important cases such as the linear case

1.2.2 How does semi-gradient TD(0) algorithm differ from gradient Monte Carlo algorithm? Do both of them converge to the global optimum in linear function approximation? Also mention the specific term to which semi-gradient TD(0) algorithm converges.

Gradient-MC and semi-gradient-TD(0) try to approximate a differentiable function $\hat{v} : S \times \mathbb{R}^d \to \mathbb{R}$ where $\mathbb{R}^d$ is a weight vector that describes model parameters. Like MC and TD(0) the main difference between the two algorithms is, that gradient-MC updates the weight-vector after each episode and is therefore unbiased and semi-gradient-TD(0) updates the weight-vector after each step and is therefore biased. The main strength of gradient-MC is a convergence guarantee for a local minimum, where semi-gradient-TD(0) only has a convergence guarantee for linear functions (see Equation 2). We use semi-gradient-TD(0) because even tough it does not converge as robustly as gradient-MC it learns much faster and can be used in continual and online, without waiting for an episode to end.

$$A = X^T D(I - \gamma P)X \tag{2}$$

1.2.3 What is the advantage of least squares TD over semi-gradient TD(0) algorithm?

Least squares TD is the most data efficient form of linear TD(0). It therefore learns the fastest. It also does not need a step-size parameter, but it does require an $\epsilon > 0$ and has higher complexity as well as higher memory costs compared to semi-gradient-TD(0).

1.3

1.3.1 What is the benefit of replay buffer in Deep Q-Network (DQN)?

The replay buffer can be interpreted as batches in supervised leaning of neural networks. Using multiple state transitions for each evaluation of the gradient decent leads to more stable training and mitigates overestimation of actions. Since we sample randomly from the buffer the samples are i.i.d. Which is important as taking consecutive transitions these correlate somehow.

1.3.2 What is the purpose of the target network in DQN? Mention 2 options on the update of the target network in DQN.

In basic DQN there is only one network. This network gets updated after each step in the episode, this leads to two problems: One problem of DQN is the so called ”Moving Target Problem”, since we simultaneously optimize the policy and the model using the same network, we try to use regression on a target that is then changed in the same step.
The other is called “catastrophic forgetting“ this effect is caused by relearning the entire network after each step and can lead to problems even with a very small learning rate.
To mitigate these effects, we use a target network. The target network is used to chose actions for a specified interval. Parallel to that a second network gets updated for each step. This allows for collecting more samples in between updates.
Update Options: One option is that after a specified number of steps to copy the weights of this network to the target network: $\phi' \leftarrow \phi$. Another option is called ”Polyak averaging” where the target network is updated every step with following rule: $\phi' \leftarrow \tau\phi' + (1 - \tau)\phi$

1.4

1.4.1 Explain what is the maximization bias in DQN or q-learning?

The maximization bias in Q-Leaning is a technical way of saying that the algorithm overestimates the value function estimates and action value estimates. This happens because the same samples are being used to decide which action is the best and to estimate the value of that action. The problem is more pronounced in noisy or stochastic environments since a few missteps can lead to the overestimated action being used as a target. The problem can also occur if the model in DQN is too small. It is therefore also an expression of under fitting.

1.4.2 How to mitigate this problem? Show the formulae for updating the Q-values in the mitigated method.

A possible mitigation against the maximization bias is to used two different action-value estimates (see Equation 3). The best action will then be chosen from the first action-value estimate and the target can be chosen from the second action-value estimate.

$$Q_1(S_t, a) \leftarrow Q_1(S_t, a) + \alpha(R_{t+1} + \gamma Q_2(S_{t+1}, \arg\max_{a'} Q_1(S_{t+1}, a')) - Q_1(S_t, a))) \tag{3}$$

This update rule is used in DDQN.

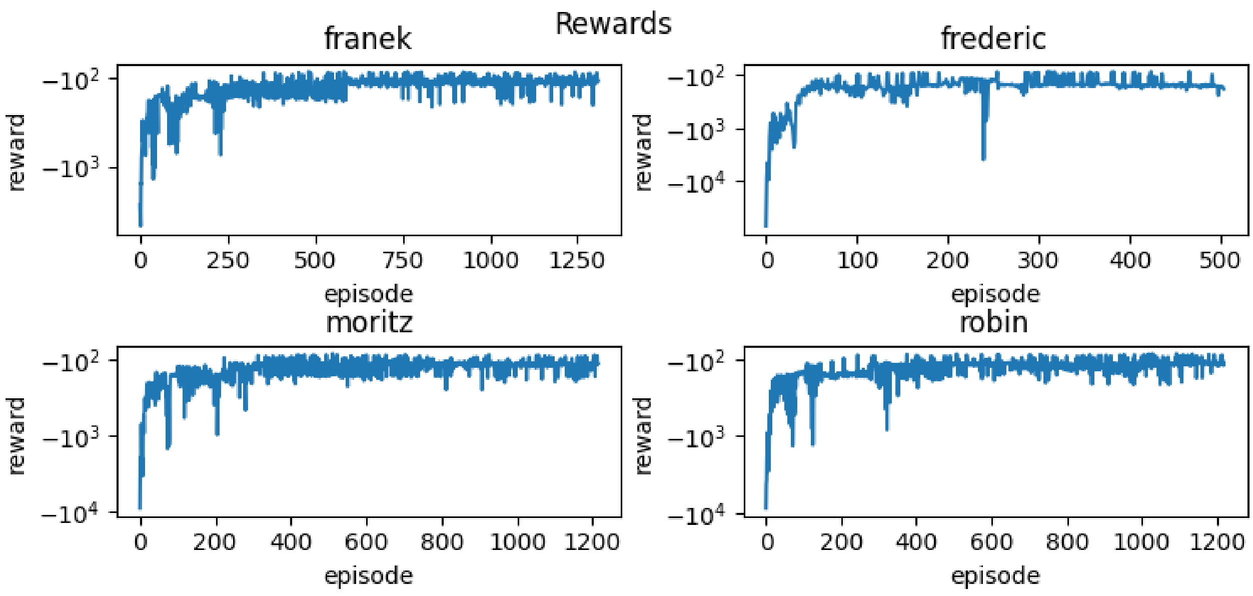

Figure 1 Episodic rewards of DQN in Mountaincar-v0

2 Task: Programming part on DQN and DDQN

2.1

See figure 1

2.2

See figure 2 and 3
The two networks are named running network and target network. They share the network architecture but only the running network will be learned via the traning error. The target network weights will be then replaced every 3000 timesteps by the weights from the running network.
The second network is intoduced at the loss function, since in classic DQN we are using the same values both to select and to evaluate an action. This might result in overoptimistic value estimates. As a solution to this issue one decouples the selection from the evaluation by using two seperate networks for the task. This results in the following formula for the loss:

$$Y_t = R_{t+1} + \gamma Q(S_{t+1}, argmax\ Q(S_{t+1}, a; \theta_t); \theta'_t) \tag{4}$$

The network architecture is a fully connected layer as an input layer which maps from the input neurons onto 64 hidden neurons. (Input neurons are the state variables so we have one input for each state variable). Then Followed by a PReLU layer. One more fully connected layer which maps the 64 hidden neurons onto another layer of 64 hidden neurons again followed by a PReLU layer. Finally we have a Fully connected Layer which maps the 64 hidden neurons onto the output neurons. Each output neurons is representing the estimated Q value for one action. So there are as many output neurons as we have actions in the respective environment.

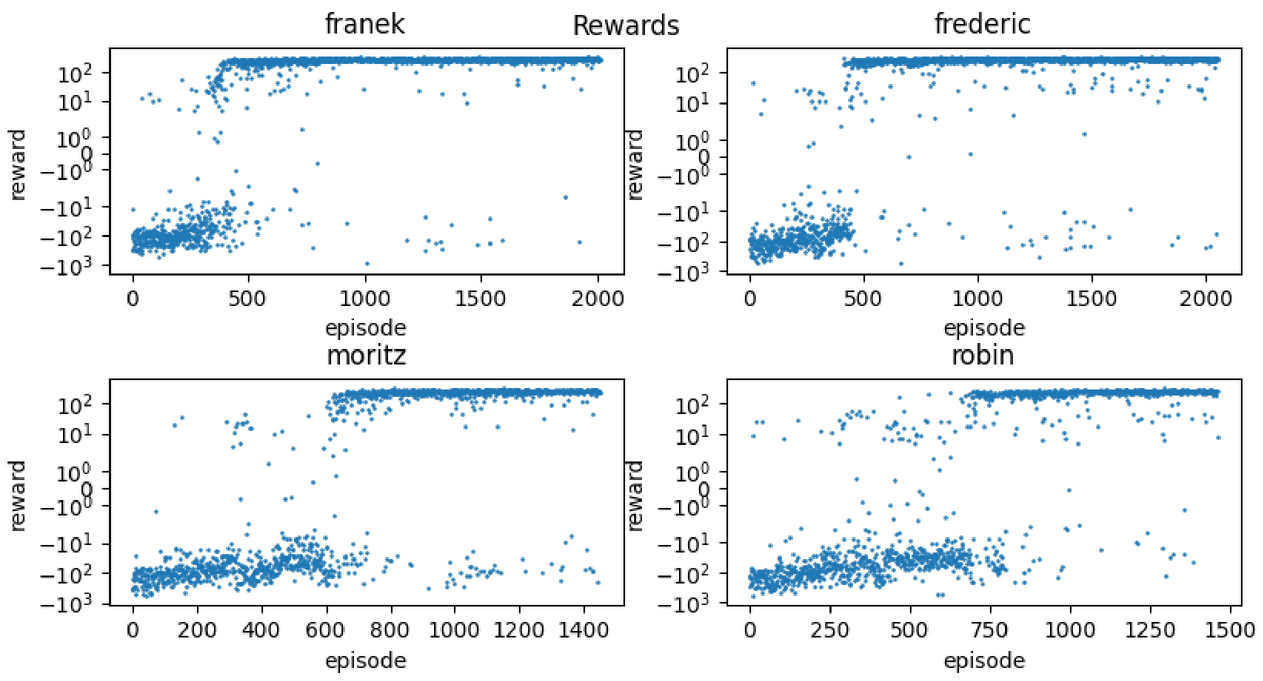

Figure 2 Episodic rewards of DDQN in LunarLander-v2

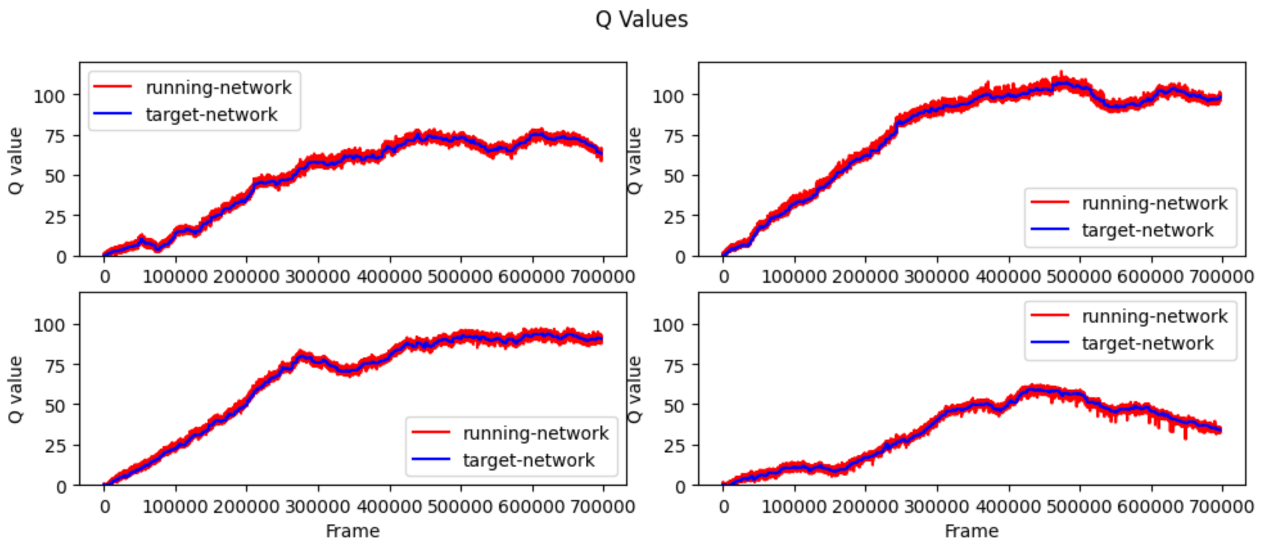

Figure 3 Approximated $\hat{q}(s_0, a*, w)$ of DDQN in LunarLander-v2

3 Task: Going Deeper

3.1 If you look into the trend of estimated Q-value of the initial state s0 in the programming task of DQN in "Mountaincar-v0",

3.1.1 What can be the lowest possible q-value for $\hat{q}(s_0, a*, w)$ in the case of $\gamma = 0.99$?

From the formula for the expected Q-Value under a policy π we know that:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right]$$

Assuming know the worst policy π_{bad} which won't solve the task we know from the *MountainCar-V0-* Environment that it gives a reward of $R = -1$ per step. (Assuming that the 200-step-termination rule is removed) Therefore we have:

$$\hat{q}_{\pi_{\text{bad}}}(s,a) = \sum_{k=0}^{\infty} 0.99^k * (-1) \approx -100$$

Looking into the Plot 5 you can see that -100 is the minimum value that we get in the experiment which needs over 100000 frames to complete the first episode.

3.1.2 Why or in which case can $\hat{q}(s_0, a*, w)$ approach this lowest possible value in DQN/DDQN?

At start of the training process we take many time steps to finish the first episode. Until we finish, the $\hat{q}(s_0, a*, w)$ gets updated to lower values for each time step as the environment returns -1 for each step. If it takes a long time to finish the first episode we will reach this value. That is around 100000 frames as can be seen in the top right plot in figure 5.

3.1.3 Does the final converged value of $\hat{q}(s_0, a*, w)$ corresponds roughly to its optimal value, reason that with mathematical computation.

In the dwintion of MountainCar-v0 environment it is stated that the task is treated to be solved if the average reward is -110 over the last 100 runs. Therefore we assume the average optimal number of steps to be 110. The optimal value of $\hat{q}(s_0, a*, w)$ is equal to the the Return of 110 time steps (See Equation 3.1.3).

$$\hat{q}(s_0, a*, w) = \sum_{k=0}^{110} 0.99^k * (-1) \approx -67.2$$

In our implementation we converge to ~ 67 as one can see in figure 5.

3.2 Again in the task of DQN on 'Mountaincar-v0'

3.2.1 Provide the figure showing the q-loss function and estimated q-value of the initial state. In this task, one figure per group is sufficient.

See figure 5 and 4.

3.2.2 Explain why q-loss approaches zero at the early phase, and then experience a sudden increase? Mention which kind of sample/experience leads to the first occurrence of high q-loss?

We observed that the loss was zero until the learning finished the first episode. Then the loss increased and meanwhile the Q-value increases as well. The increasing loss can be explained when looking at our loss function

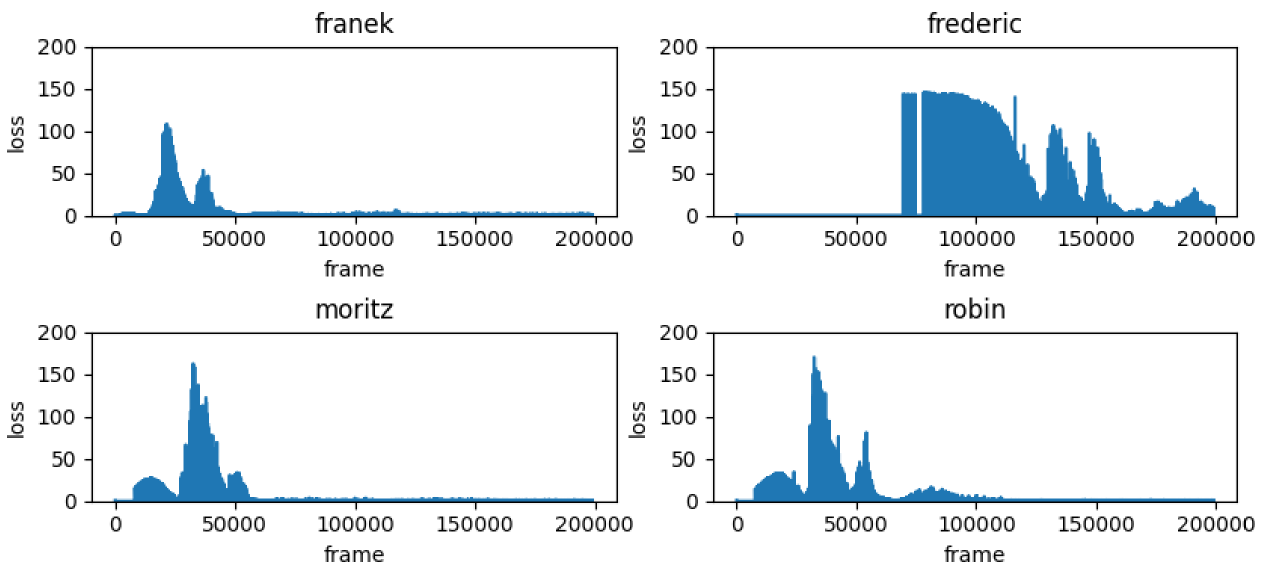

Figure 4 Training losses during training of DQN in Mountaincar-v0

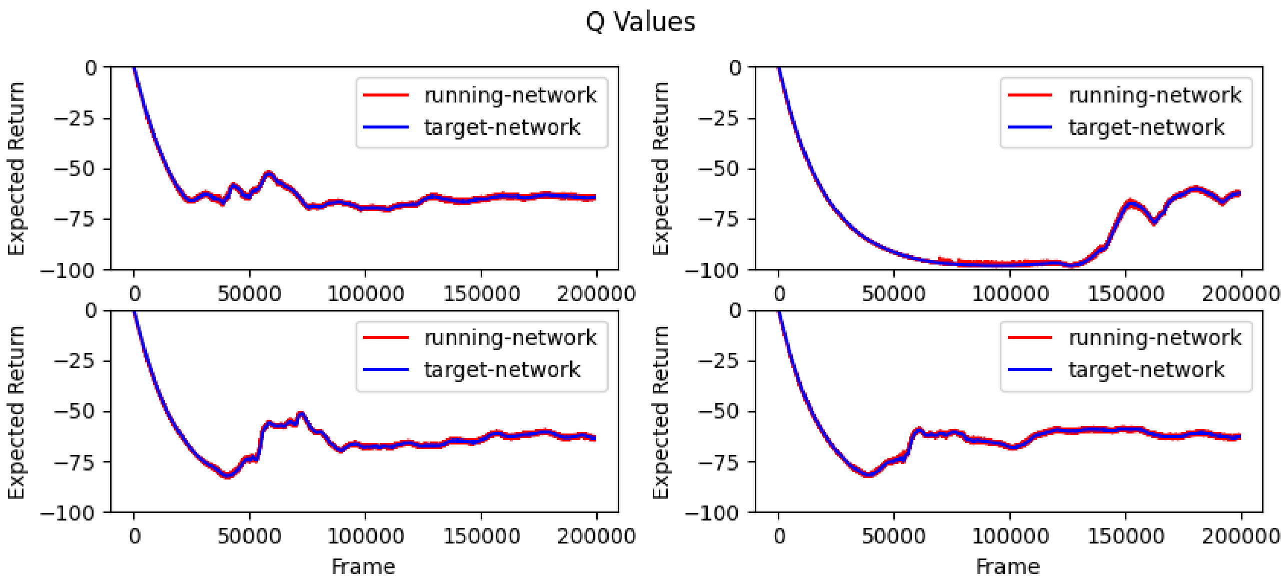

Figure 5 Approximated $\hat{q}(s_0, a*, w)$ of DQN in Mountaincar-v0

for DQN:

$$y = \begin{cases} r & \text{if episode done} \\ r + \gamma \max_{a'} Q_{\text{target}}(s, a) & \text{if episode not done} \end{cases}$$

$$\text{loss} = (y - Q_{\text{running}}(s, a))^2$$

We can see that until an episode is finished y contains the reward plus the maximum q value of our network. Our loss is then calculated as the MSE of y and the q value of the current state action pair. In the beginning those two values are very similar since we did not gain any experience of finishing an episode. Moreover for each step we only decrease the value by -1 so the error is only 1. As explained above and can bee seen in figure 5 the q values are decreasing to minium -100. However, once we do finish an episode y becomes -1 since that is the reward for that specific time step, but the q value that our network returns is still very low, therefore we receive a very high loss once we finish an episode. After that we can see that the loss is further increasing. This is due to the fact, that we are using the target network to calculate y and the running network to calculate the loss. If we were doing hard updates every iteration this would not matter since the weights would be the same, but this is not how the target network is updated. We are doing soft updates using polyak averaging with $\tau = 0.005$ and therefore the weights of the target network are only slowly adjusting to changes in the running network. Moreover as we have this high loss one time the q function will be different in the next steps. Therefore the behaviour of the agent changes and it starts to finish many episodes which results in a real 'traning' behaviour with high losses at the beginning as the Q value where to low.

Literatur

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.