UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR ROBOTICS
AND COGNITIVE SYSTEMS

Course: Reinforcement Learning (SS2020)
Prof. Dr. Elmar Rueckert & Honghu Xue, M.Sc.
Assignment Report Nr. IV
Date: 24. Juni 2020

Team Nr. 1
Matthias Klatt
Abby Kandathil Abraham
Suneesh Omanakuttan Sumesh Nivas
Sameed Quais
Lakshith Nagarur Lakshminarayana Reddy
Contact: matthias.klatt@student.uni-luebeck.de

Task I, q.1:

on-policy, non-linear:

$$\omega_{t+1} = \omega_t + \alpha[v_\pi(S_t) - \hat{v}_\pi(S_t, \omega_t)]\nabla\hat{v}(S_t, \omega_t) \tag{1}$$

the conditions that this converge is: an decreasing positive α which fulfills the following conditions (for convergence that needs to be fulfilled every time α is involved):

$$\begin{aligned} \sum_{k=1}^{\infty} \alpha_k(a) &= \infty \\ \sum_{k=1}^{\infty} \alpha_k^2(a) &> \infty. \end{aligned} \tag{2}$$

MC: For MC approaches we need to approximate $v_\pi(S_t)$ so we define U_t as G_t the return of a MC approach with that we get the following equation:

$$\omega_{t+1} = \omega_t + \alpha[U_t - \hat{v}_\pi(S_t, \omega_t)]\nabla\hat{v}(S_t, \omega_t), \tag{3}$$

again α needs to fulfill the convergence conditions from equation 2 and it must decrease. U_t must to unbiased to guarantee convergence and that is the case for MC approaches but not for TD approaches.
bootstrapping: for bootstrapping approaches we have a biased U_t for the equation 3 and that leads to less robust convergence what means in general can it fail to convergence but if we approximate the function linear for example we can reach at least some near optimal policy.

on-policy, linear: Special case of a function approximation is the linear case here the update rule will change slightly, so first $\hat{v}(S_t, \omega_t)$ reduces to:

$$\hat{v}(S_t, \omega_t) = \omega^T \mathbf{x}(S_t) \tag{4}$$

and gradient of the approximate value function also:

$$\nabla\hat{v}(S_t, \omega_t) = \mathbf{x}(S_t) \tag{5}$$

with that the non-linear update rule 3 for example will change to:

$$\omega_{t+1} = \omega_t + \alpha[U_t - \hat{v}_\pi(S_t, \omega_t)]\mathbf{x}(S_t) \tag{6}$$

under the same conditions as for the non-linear case the MC approach can converge to a local optimum but because we are in the linear case the local optimum is also the global optimum.
bootstrapping: So for the TD(0) approach we will converge if the the system converges to a so called TD fixed point ω_{TD}, this point is defined as: $\omega_{TD} = \mathcal{A}^{-1}\mathbf{b}$ with $\mathbf{b} = \mathbb{E}[\mathcal{R}_{t+1}, \mathbf{x}(S_t)] \in \mathbb{R}$ and $\mathcal{A} = \mathbb{E}[\mathbf{x}(S_t)(\mathbf{x}(S_t) - \gamma\mathbf{x}(S_{t+1})^T]$ for the continuing case this will converge to $\overline{VE}(\omega_{TD}) = \frac{1}{1-\gamma}\min_\omega \overline{VE}(\omega)$ so in practice γ is chosen near 1 and that value will get large but if the discount factor is chosen very small the error can be very small but learning will take very long.

Task I, q.2:

for SGD methods we update ω with the following equation (for the on policy case):

$$\omega_{t+1} = \omega_t + \alpha[v_\pi(S_t) - \hat{v}_\pi(S_t, \omega_t)]\nabla\hat{v}(S_t, \omega_t), \tag{7}$$

or in MC approaches:

$$\omega_{t+1} = \omega_t + \alpha[U_t - \hat{v}_\pi(S_t, \omega_t)]\nabla\hat{v}(S_t, \omega_t), \tag{8}$$

where U_t is an unbiased estimate (so $\mathbb{E}[U_t|S_t = s] = v_\pi(S_t)$) because its the return of the MC approach. The same equation are used for updating the weights in bootstrapping approaches also U_t is defined as the n-step return and this return depends on the current value of ω_t and because of that it is biased because the target isn't independent of the weight vector ω_t. Because bootstrapping approaches aren't true gradient-descent methods they are called semi-gradient methods.

Task I, q.3:

what of loss needs to minimized in DQN: The update rule for the $\mathcal{Q}$-function is:

$$\mathcal{Q}_\pi(s, a) = r + \gamma\mathcal{Q}_\pi(s', \pi(s')), \tag{9}$$

With an so called temporal difference error δ which indicates the difference between the two sides of the equation 9:

$$\delta = \mathcal{Q}(s, a) - (r + \gamma \max_a \mathcal{Q}(s', a)), \tag{10}$$

and this error needs to minimise. So For DQN we use the Huber loss. Special about this loss function is that if the error is small the Huber loss function acts like the mean squared error function and if the error is large it acts like the mean absolute error. Because of his characteristics the Huber loss function is more robust to outliers when the estimate of $\mathcal{Q}$ is very noisy. If we calculate the Huber loss function over a batch of transitions ($\mathcal{B}$) sampled form replay memory it is defined as:

$$\begin{aligned} \mathcal{L} &= \frac{1}{|\mathcal{B}|} \sum_{(s,a,s',a',r)\in\mathcal{B}} \mathcal{L}(\delta), \\ \text{with } \mathcal{L}(\delta) &= \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1 \\ |\delta| - \frac{1}{2} & \text{else} \end{cases}. \end{aligned} \tag{11}$$

Benefit of replay buffer:
Using experience replay memory for training our DQN helps to prevent overfitting. It stores the transitions that the agent observes, allowing us to reuse this data later. By sampling from it randomly, the transitions that build up a batch are decorrelated. It has been shown that this greatly stabilizes and improves the DQN training procedure.

Purpose of target network in DQN:
The target network (first introduced in [2]) is essentially a copy of the training model at certain time steps so the target model updates less frequently. This second network is used to generate the target-$\mathcal{Q}$ values that will used to compute the loss for every action during training. The issue if we just use one network is that at every step of training, the $\mathcal{Q}$-network's value shift so the value estimations can easily spiral out of control. The network can become destabilized by falling into feedback loops between the target and estimated $\mathcal{Q}$ values. To mitigate that risk, the target network's weights are fixed, and only periodically (or slowly) updated. In this way training can proceed in a more stable manner.

Task I, q.4:

difference between DQN and fitted Q-iteration:
Beside the existence of the target network in DQN, Neural Fitted Q Iteration only uses the available historical observation and does not perform any exploration. In other words, there is no need to have an environment and there is just loop over train steps while in DQN there are two loops. DQN uses the target network, though fitted Q iteration uses the current policy.
Actually, Neural Fitted Q Iteration is considered as a batch-RL algorithm which assumes there is not any available environment.

Task I, q.5:

what are respectively the hard update and soft-update of the target network in DQN?
If we do a so called *hard update* for the target network of a DQN approach the target network parameters (θ^-) will set at each update to the parameters of the learning network (θ^+) so the update is done at certain time steps and if we update we update at once.
If we use so called *soft update* we do not set the parameters from the target network equal to the current network parameters of the learning parameters at once. We will use a parameter τ to determine how big the influence of the current parameters of the learning network are and how big is the influence of the current target network parameter. If we use a very small value for τ we need to update frequently to get an effect of the update. The update rule for *soft update* is the following:

$$\theta^- = \theta^+ \times \tau + \theta^- \times (1 - \tau). \tag{12}$$

Because τ is chosen very small we need to update frequently so our target network weights will converge to the learning network weights.

Task I, q.6:

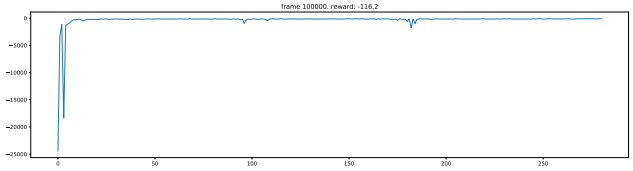
In Richard S. Sutton and Andrew G. Barto's book: Reinforcement Learning: an Introduction at says that: 'Maximization bias occurs when estimate the value function while taking max on it (that is what Q learning do), and maximization may not take on the true value which may introduce bias.' [3]. To solve this problem for DQN approaches or q-learning we need to decouple in two policies $Q_1(a)$ and $Q_2(a)$ and choose, say $Q_1(a)$ to choose maximizing action and $Q_2(a)$ to estimate its value. For DQN approaches we decouple by using two instead of one network for training so we have a learning and a training network.The training network is used to generate the target-Q values. That might slows down the training process but we got a more robust performance.

Task II, q.1:

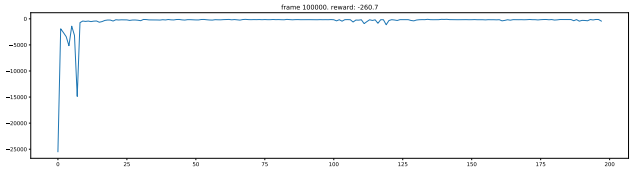
In the figure 1 you can see one learning curve per team member for the given DQN implementation task for the 'MountainCar-v0' environment.
We trained our DQN until we reaches 100000 frames after that we terminated, we thought that this would show how different the training solutions can be because of some random factors.

Task II, q.2:

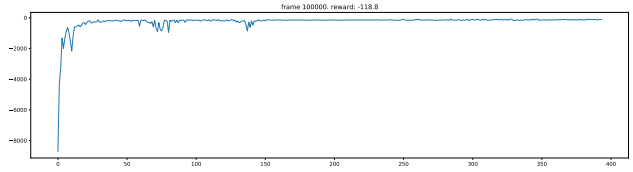
In figure 2 you can see one learning curve per team member for the given DDQN implementation task for the 'LunarLander-v2' environment. For this particular environment are four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine. So the actions are discrete but the states not.



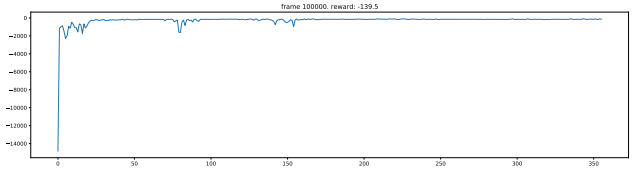
(a) *Matthias*



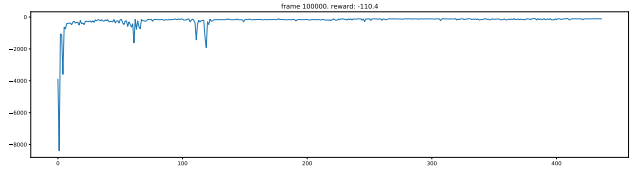
(b) *Abby*



(c) *Suneesh*



(d) *Sameed*



(e) *Lakshith*

Figure 1 The learning curves for the DQN task one learning curve from each team member

There are two slightly different approaches which are called double DQN the first were introduced in 2010 in [1] here we use to $\mathcal{Q}$-networks A and B like:

$$\begin{aligned}\mathcal{Q}_A(s,a) &= \mathcal{Q}_A(s,a) + \alpha(\mathcal{R} + \gamma\mathcal{Q}_B(s,a)(s', \arg\max \mathcal{Q}_A(s',a')) - \mathcal{Q}_A(s,a)) \\ \mathcal{Q}_B(s,a) &= \mathcal{Q}_B(s,a) + \alpha(\mathcal{R} + \gamma\mathcal{Q}_A(s,a)(s', \arg\max \mathcal{Q}_B(s',a')) - \mathcal{Q}_B(s,a)).\end{aligned} \tag{13}$$

The idea was to use this two $\mathcal{Q}$-networks to prevent unnecessarily large $\mathcal{Q}$ values but in DQN we already have two networks working together. So in 2016 a new double DQN approaches are published in [4]. The idea of the first published DQN approache that both $\mathcal{Q}$-network influence each other is still used but now the learning and target network are used again. The $\mathcal{Q}$ values are now updated like:

$$\mathcal{Q}(s,a;\theta^+) = \mathcal{R} + \gamma \max \mathcal{Q}(s', \arg\max \mathcal{Q}(s',a';\theta^+);\theta^-), \tag{14}$$

so the $\mathcal{Q}$ network influence each other, the learning network will select an action and is updated by the target network with that selected action.
In practice both networks have the same structure but different weight vectors in the beginning during training the weight vectors can get the same but mostly used is the soft update approach (see equation 12) so that the weights of the target network will converge slowly to the weight of the learning network.
In figure 2 are five learning curves over 120000 frames one per team member to show the learning result from our implemented DDQN approach.

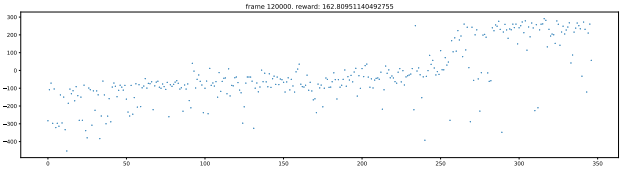
Literatur

[1] Hado V Hasselt. Double q-learning. In *Advances in neural information processing systems*, pages 2613–2621, 2010.

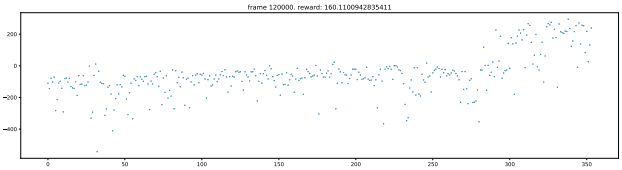
[2] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 2094–2100. AAAI Press, 2016.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: an Introduction*. Second edition.

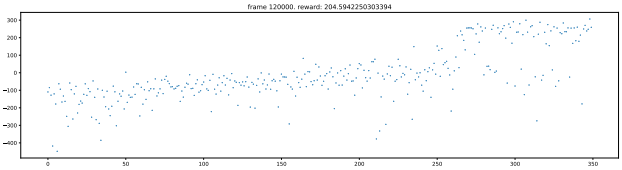
[4] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.



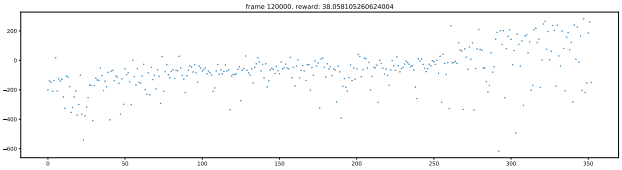
(a) *Matthias*



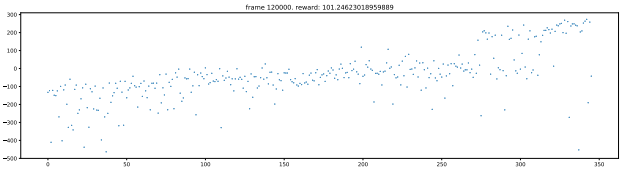
(b) *Abby*



(c) *Suneesh*



(d) *Sameed*



(e) *Lakshith*

Figure 2 The learning curves for the DDQN task one learning curve from each team member.