Assignment 3

Reinforcement Learning, WS22

Team Members		
Last name	First name	Matriculation Number
Amering	Richard	1331945
Michael	Mitterlindner	11824770

1 Deep Q-Learning for Atari Space Invaders

In this task, a game called "Atari Space Invaders" from the OpenAI gym-package is solved using off-policy Q-learning with non-linear function approximation via deep neural networks. In the first Task a fixed target Q mechanism with experience replay should be implemented with the use of the third party Python library Pytorch. A code skeleton was provided to us via the Teach Center.

1.1 Using a Fixed Target Q-Network

The fist task is to create a so called fixed target neural network (off-policy evaluation of actions) and an online network (providing on-policy actions). The networks expect an observations of the state of the game (grayscale video frames of the game, 4 images of 84x84 resolution stacked together) and return a vector of Q-values for each possible action in that state. The action space of the game is $\Re^6$, with available actions being moving left and right, standing still, combined with shooting the laser cannon or not.

Since the game-output is actually of higher resolution and with 256bit color, the images need to be transformed to a 256-bit gray-scale image and resampled into a 84 x 84 frame.

Both networks are convolutional neural networks (CNNs). The architecture of the two networks is identical and looks as follows:

- Input layer: 84 x 84
- 2D Convolutional layer: 64 x 8, stride = 4
- 2D Convolutional layer: 64 x 4, stride = 2
- 2D Convolutional layer: 64 x 3, stride = 1
- Flatten
- Linear layer: 3136
- Linear layer: 512
- Output layer: 6 (number of possible actions)

Each layer (except the input and flatten-layers) use ReLU activation.

The parameters for the online network are updated every four game-frames (meaning every step) by gradient-decent using an Adam optimizer. The target network parameters are synced via deep copy to the online network parameters every 30,000 steps.

1.2 Experience Replay Memory

To store previous game experiences and increase sample efficiency, a replay buffer is implemented. While training the network, tuples of state, action, reward, next-state and information about episode-termination are stored in the buffer. These values correspond to the current step that the online-network chose, and the environments response. For optimization of the Q-network, a batch of tuples was randomly sampled from the buffer on each step and batch-gradient decent performed on them. The batch-size for this was 32.

The size of the buffer was usually set to around 5,000 tuples, depending on memory restrictions of our hardware.

1.3 Optimization Configurations

The main goal in this task is to optimize the configurations of the loss function and the parameter ϵ.

We investigated the performance of of two different loss functions, namely Huber- and MSE-loss. The figure 1 shows the behavior of these functions.[1]

- Mean-Squared Error loss [2]

  The MSE is a standard loss function and is calculated with the difference of the function value squared.

  $$L = l_{MSE}(x, y) = \frac{1}{n}(x_1 - y_1, ..., x_n - y_n)^\intercal \cdot (x_1 - y_1, ..., x_n - y_n)$$

- Huber loss [3]
  The Huber loss is calculated with the following formula:

  $$l_{Huber} = \begin{cases} \frac{1}{2}(x_n - y_n)^2, & if\ |x_n - y_n| < \delta \\ \delta * (|x_n - y_n| - \frac{\delta}{2}), & else \end{cases}$$

  The difference to the MSE is that outside of a certain threshold (δ) the loss function becomes a linear function while the MSE remains quadratic.

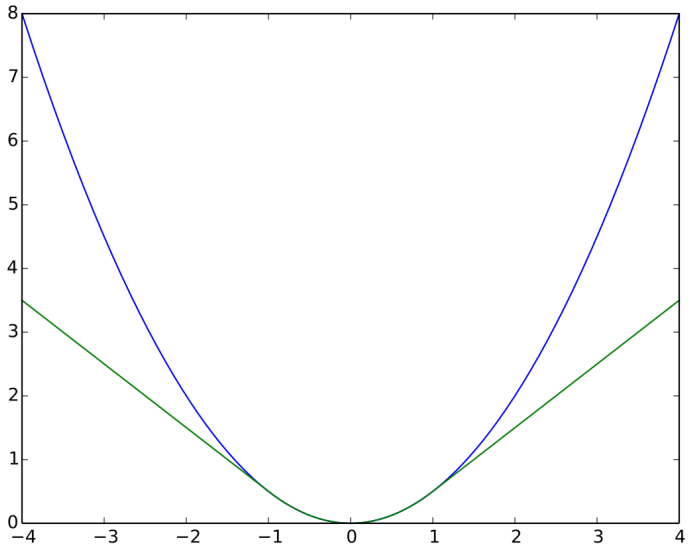

Figure 1: Huber loss (green) vs. MSE loss (blue)

[1] https://en.wikipedia.org/wiki/Huber$_l oss/media/File : Huber_l oss.svg$
[2] https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html
[3] https://pytorch.org/docs/stable/generated/torch.nn.HuberLoss.html

1.4 Results for the standard DQN-Model

1.4.1 Variation of epsilon

To study the influence of the parameter ϵ, five different initial values $\epsilon_0 = [0, 0.2, 0.5, 0.8, 1]$ were examined. ϵ was subjected to exponential decay with a decay factor of 0.999 per step, until a minimal value of 0.05 was reached, except for the case of $\epsilon_0 = 0$, which was not changed during training. When epsilon is equal to zero, no learning of the agent could be observed. $\epsilon = 0$ corresponds to the absolute greedy policy, where no exploration takes place and the actions are chosen through the network only. This happens because when epsilon is set to zero from the beginning on, the agent always performs the same actions as determined by the seeding of the initial starting position and the given weights and biases of the network, which are also seeded. In our case, the agent received the same reward in each episode by performing one single action. For the following examinations, we therefor only highlight ϵ values [0.2, 0.5, 0.8, 1].

Figure 2 shows the the episode-reward (red) and the loss (blue) over the amount of trained episodes. For this figure the model was the standard DQN model with different initial ϵ values.
The reward graphs clearly show that with a low initial ϵ, higher reward can be accumulated quicker, whereas with $\epsilon = 1$, this requires several training episodes until a stagnant reward is reached. Unfortunately, none of these variations could generate significantly more reward. The same picture can be seen in the loss graph, where none of the variations is convincingly better in the long run. We found that the agent, even after say 35,000 episodes, fails to learn that it should avoid incoming fire, and instead only chooses to shoot constantly and maybe move to one side. This grants an average reward of about 270 per episode, and is higher than just choosing random actions, but is still much lower than what could be achieved if the player was aware of the effects of being hit.

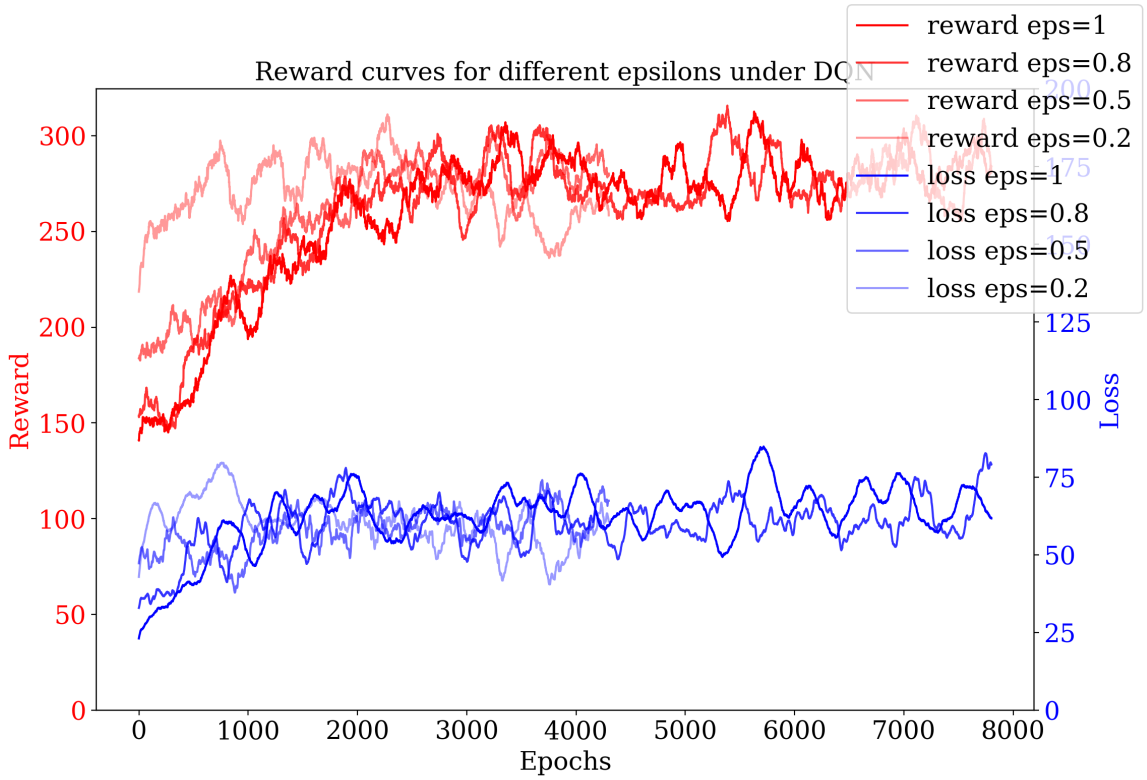

Figure 2: Epsilon variations of the standard model with MSE-loss

2 Combining Improvements for DQNs

2.1 Double Q-Learning: DDQN

The first improvement of the model is a switch to the double Q learning loss calculation. For this, the argmax function was applied to the online network, in the next state. The loss is then calculated with this action and the next-state with the target network.
Figure 3 shows the reward and the loss plotted over the trained episodes. Here it can also be seen that the DDQN model approaches the final value faster with a lower epsilon. This happens even faster than with the DQN model. Again, there is not much difference between the Epsilon values for either the Reward or the Loss in the long-run and therefore a lower Epsilon is actually a little better, because the network is training faster.

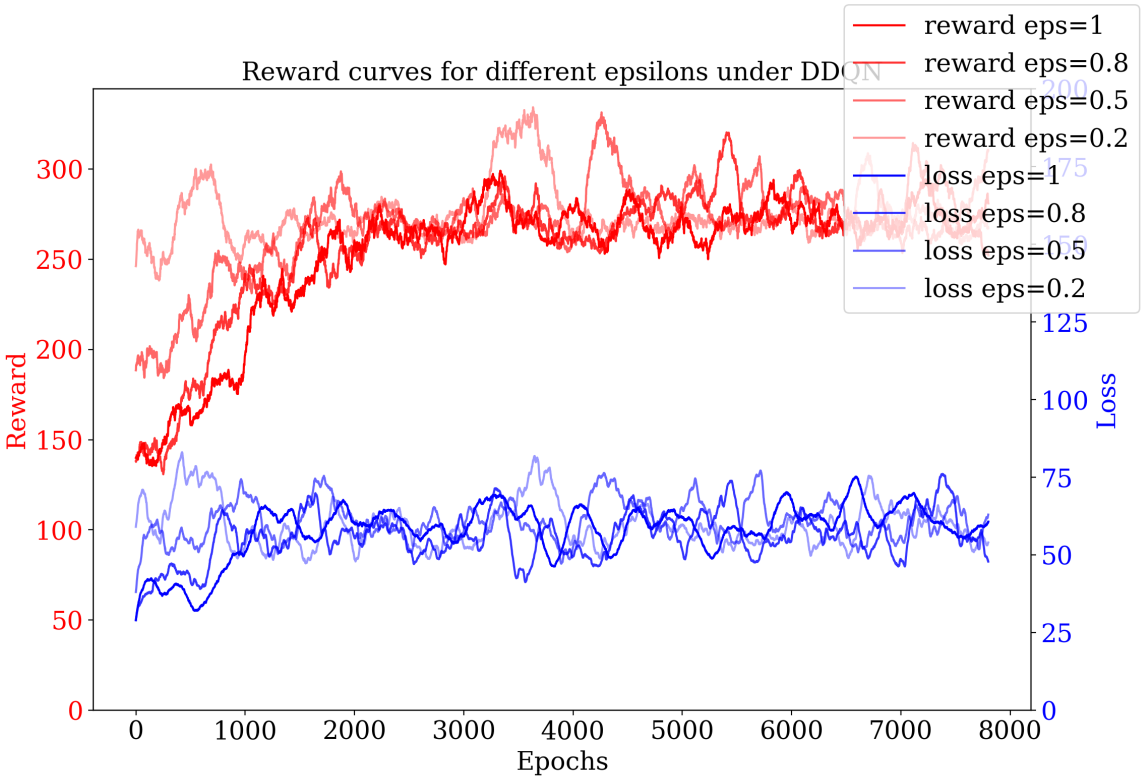

Figure 3: Epsilon variations of the DDQN model with MSE-loss

2.1.1 Variation of the loss function

As already mentioned above, two different loss functions were investigated: Mean squared error and Huber loss.

Therefore a DDQN model was trained with the Huber-loss function instead of the MSE-loss. The results for this training is shown in figure 4. In comparison to figure 3 the reward graph shows almost no outliers. But more conspicuous is the fact that the Huber-loss calculates the loss via a linear function and therefore the loss is over a magnitude lower than for the MSE-loss.

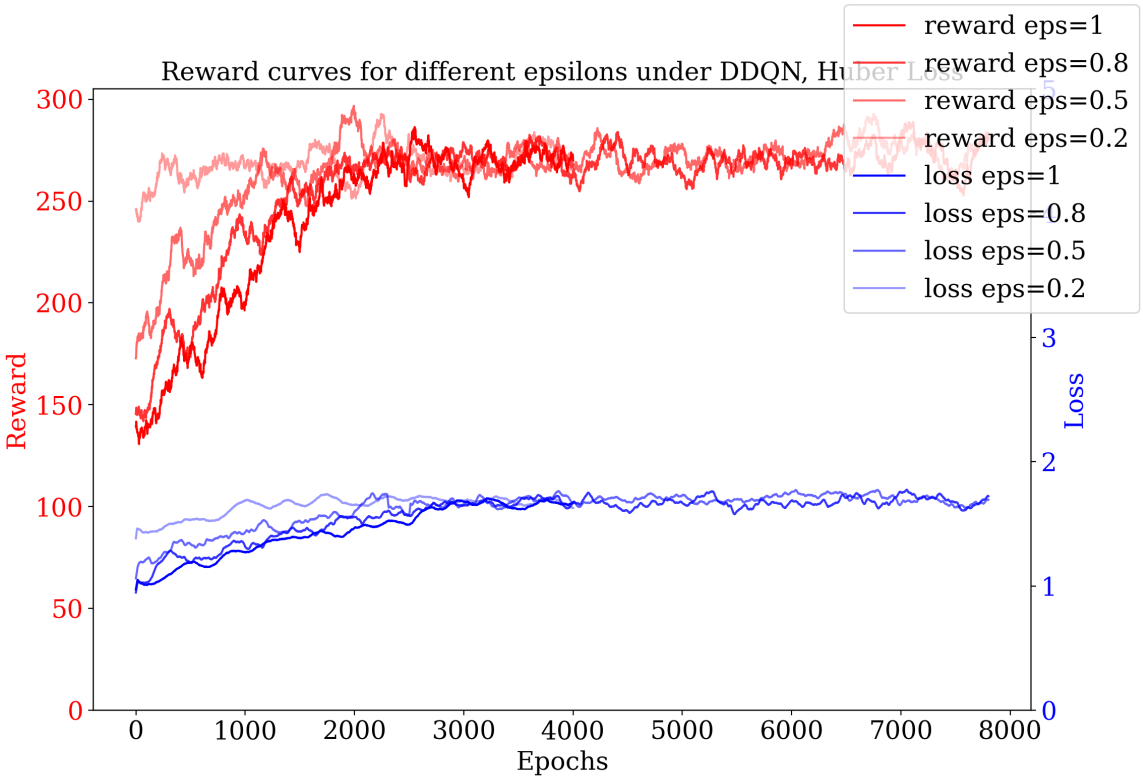

Figure 4: Epsilon variations of the DDQN model with Huber-loss

We showcase the effects of different loss functions in Figure 5. Here, a DQN model was trained with the MSE, and a DDQN model was trained with MSE and Huber loss. From our testing, we could not find any significant difference in training performance with either combination of learning models and cost-functions. In theory, the Huber loss function should be more resistant to extreme outliers, where the quadratic nature of the MSE produces far greater loss-values than the Huber loss. We could not find that this effect allowed our models to train faster.

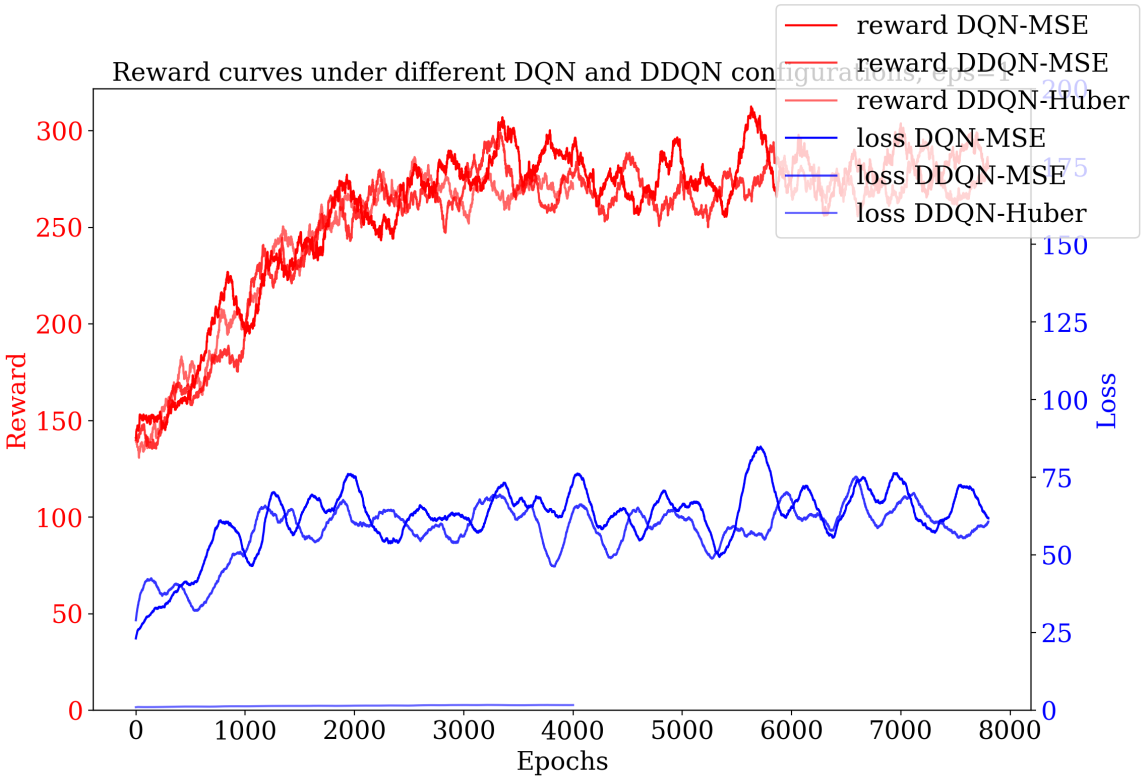

Figure 5: DQN and DDQN with MSE- and Huber-loss, $\epsilon_0 = 1$

2.2 Multi step rewards

The next improvement should happen through a multi-step learning model. For this, the reward is discounted and summed up over several time steps. We have chosen to run a DQN and DDQN model with three- and six-step reward accumulation, to be able to examine its effects more closely. We also compare these with the standard 1-step reward accumulation, which is the base case. Figure 6 shows different reward curves for these methods with the DDQN. In theory, multi-step reward accumulation could allow the agent to learn faster, since actions often grant a reward only after some time passes. For example, firing the cannon in the game could take a few game-frames, until the emitted laser hits an invader, if it is still higher up and therefor further away from the cannon. The multi-step methods carry over some of that future reward to the present actions, making it easier for the agent to see the connection. Unfortunately, we cant see this effect here, and the agent still seems to be unable to learn more sophisticated strategies than just moving to one side and firing the laser constantly, not avoiding any incoming fire. We don't show the multi-step results for the DQN model since they are identical to the DDQN results.

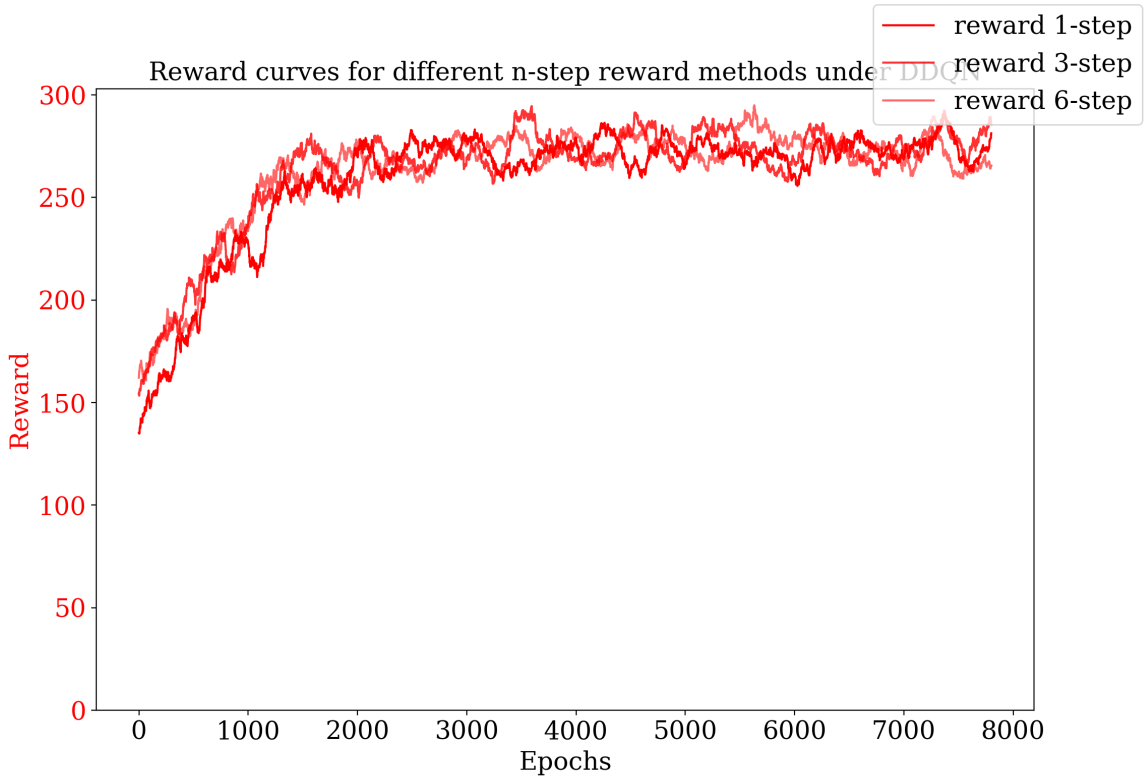

Figure 6: Reward curves for 1-,3- and 6-step reward accumulation, DDQN model with Huber-loss, $\epsilon_0 = 0.8$

2.3 Prioritized Experience sampling

All configurations so far sampled experience tuples from the buffer uniformly. However, some examples could be more valuable for training. For example, if the TD-error of an example is very large, the networks are bad at evaluating its Q-values. It would make sense to sample such experiences at a higher probability. This is called prioritized experience sampling. We implemented this by adding a weighting-buffer to the memory-buffer. This weighting-buffer holds a corresponding sampling weight for each experience tuple in the memory buffer. When sampling is performed, the examples are drawn with a probability according to the corresponding weight.

Figure 7 shows our results with the prioritized sampling strategy. Notably, the loss with prioritized sampling is much higher, although both configurations use the same loss function, Huber loss. This is expected, since samples that had bad predictions previously are more likely to be sampled again, therefore keeping the loss high. There are some spikes in the reward curves that suggest that prioritized sampling was able to reach higher rewards quicker, but this could just as likely be a random phenomenon, since it is common for RL agents to go up and down in terms of gathered reward during training.

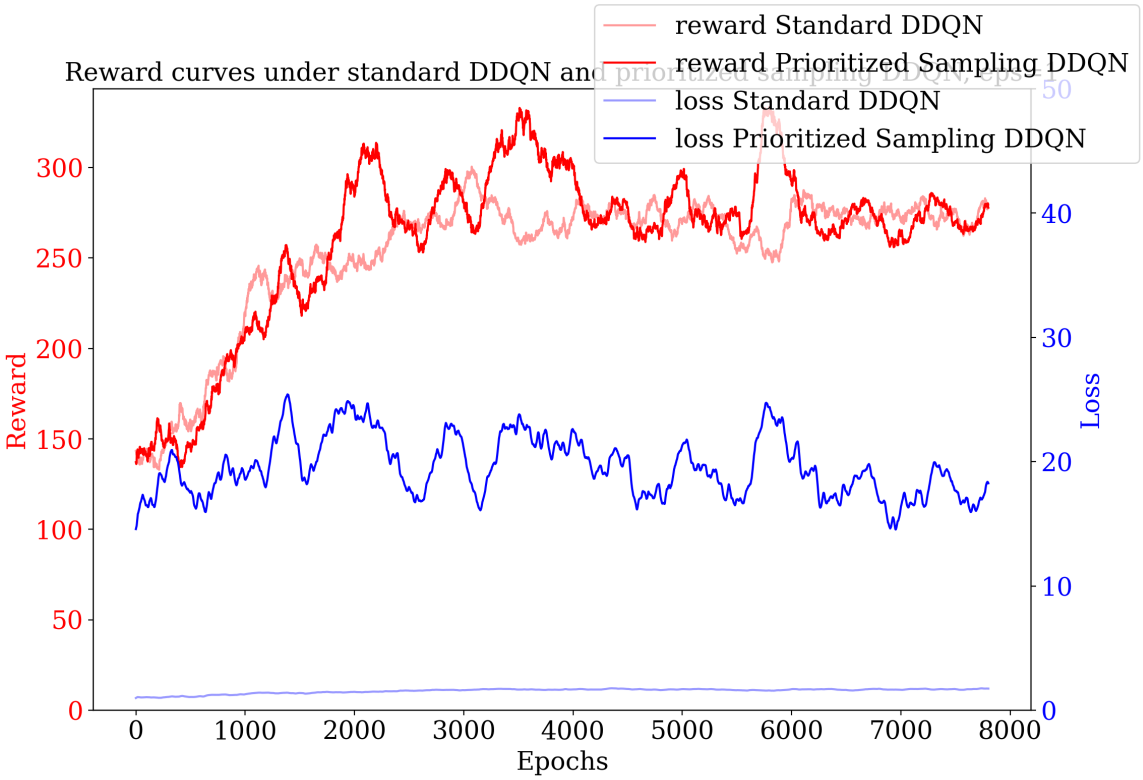

Figure 7: Reward curves for DDQN with standard and prioritized sampling, Huber-loss, $\epsilon_0 = 1$

2.4 Testing more complex network architectures

In addition to the improvements mentioned above, we investigated several architectures for our network. Since the above configurations did not give us satisfying results, we were expecting that maybe the network complexity is too low to learn the more intricate features of the game. Unfortunately none of the other network architectures performed significantly better than the one chosen. Figure 8 compares two different complexity-levels of the network architecture.

The architecture of the more complex network was as follows:

- Input layer: 84 x 84 x 4
- 2D Convolutional layer: 32 channels, kernelsize=8
- 2D MaxPooling, kernelsize=3
- 2D Convolutional layer: 64 channels, kernelsize=4
- 2D Convolutional layer: 64 channels, kernelsize=3
- 2D MaxPooling, kernelsize=3
- 2D Convolutional layer: 64 channels, kernelsize=3
- Flatten
- Linear layer: 1024

- Linear layer: 750
- Linear layer: 512
- Linear layer: 256
- Output layer: 6 (number of possible actions)

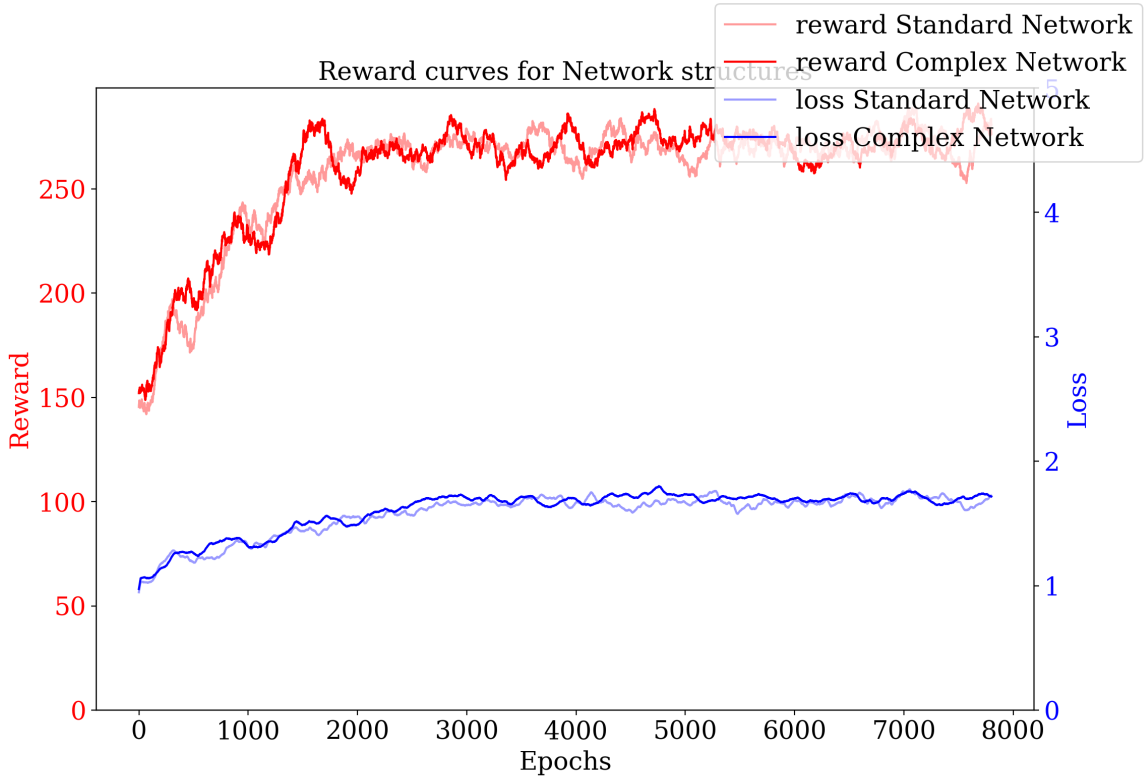

Figure 8: Reward curves for different network architectures, $\epsilon_0 = 0.8$

3 Discussion

Unfortunately, our agent did not show great learning behavior, because he only learns to shoot but not to dodge. Albeit many different hyperparameter configurations, learning models and architectures have been investigated, none of our agents was able to significantly surpass an average episodic reward of around 300. All agent were able to learn that shooting increases the reward, but none learned that surviving for longer time by evading incoming fire gives the chance to increase the reward even further. One reason we see for this behavior is that the environment does not yield negative rewards for being hit. Instead, the only reward given by the environment is positive, if invaders are being shot. The exploration due to the epsilon greedy policies seem to be insufficient to teach the agent the intricacies of the game. A possible solution is to change the reward function, or to include a reward for each time step the agent survives, giving a higher incentive to be more careful.