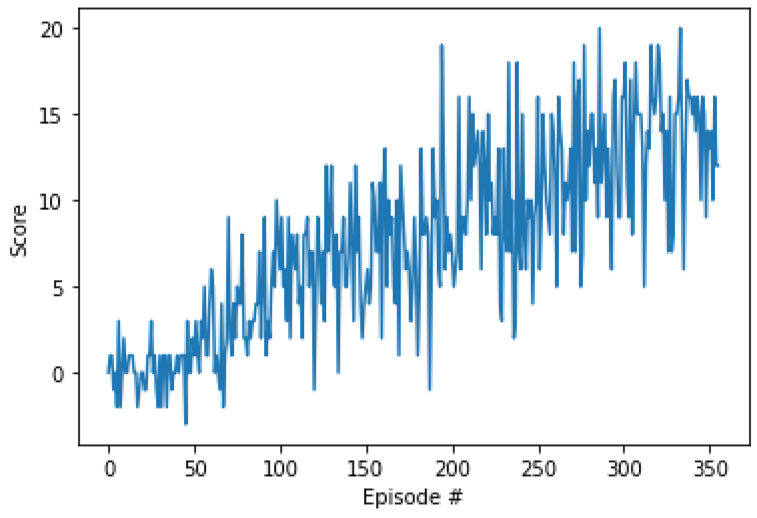
Report

In my implementation I used the code of a dqn-agent given during the lectures, but modified it so that the agent is able to solve the envirnoment much faster (within **256** steps).
The jupyter-notebook „Run.ipynb“is the main file, where we load the environment, set up the agent and are able to monitor the progress. When the agent reaches an average reward of 13 over the 100 episodes, it stops, saves the model and returns a plot of the learning curve of the agent.
The agent itself uses an epsilon-greedy policy to choose the next action. We start with an $\epsilon = 1.$ and use an exponential decay with decay rate 0.99. The agent has a replay puffer of size 10000, where it stores the different experiences it makes. Additionally, we save the TD-error for every experience to learn using priorized experience replay. Please note, that here we implemented the structure of a double dqn to stabilize the learning process. For the priorized experience replay we use the following hyperparameters:

$$\alpha = 0.1\,, \qquad \epsilon = 0.01\,, \qquad \beta = 0.6\,,$$

while we raise β with every step by a value of 0.01.
The network itself has a duelling architecture where we start with 3 dense layers with 64 neurons, followed by two parts, one for the state value and one for the advantage value. The part for the state value has one layer with 64 neurons, and then just one neuron, the advantage value part has one layer with 64 neurons, and one with 4 neurons, where we ensure that the average of the neurons is always zero. Afterwards we add the output values for both parts to get the Q-values for each action. In the network we use after each layer a ReLu-activation function. We train our network with a learning rate of $5 * 10^{-5}$ with Adam optimizer and a batch size of 64.
As a result we get for the average reward the following plot:



As we can see is our agent able to solve the environment within 256 steps.

For the future, we can implement the other 3 possible improvements: Multi-step bootstrap targets, Distributional DQN and Noisy DQN. Another possibility is to improve the hyperparameters further. Here is a lot of space for improvements.