**Assignment ATAI Part I –**

**N-Step and Double-Q DQN**

Frederik, Leon, Siebe, Dimi

**Deep Q-Network**

In this assignment a n-step Deep Q-Network is implemented. To be able to understand this network it is important to first understand how a normal Deep Q-network works. The Deep Q-network learns an optimal policy by combining neural networks and Q-learning for reinforcement learning. This neural network learns the Q values associated to each state/action pair by calculating the expected reward based on a mini-batch of the dataset. This mini-batch is selected randomly for every iteration and consists of state-action values (the predicted variables) and the ‘true’ Q-value (the target). As we don’t know the true Q-value we use an approximation taken from Q-learning which takes the first n immediate rewards and the estimated Q value from step n onwards. In a basic Deep Q-network only the first immediate reward is used whereafter the Q value of the action that achieves the highest long-term reward is added:

So for every iteration, a combination of the predicted variables and the target out is taken from the training dataset which is used to update the weights of the neural network Q(S,A) with the following formula:

\bold{w} = \bold{w}  + \; \alpha \;
\sum_{t \in B}  \frac{\partial \hat{q}_\bold{w}}{\partial \bold{w}}(S_t,A_t) 
[r_{t} + \gamma max_{a'}\hat{q}(S_{t+1},a',\color{green}{\bold{w'}}) -  \hat{q}(S_t,A_t,\bold{w})]\; 

In here a separate neural network is used for the target to make sure that the weights we are trying to learn are not part of the training formula and new policies are added to the experience using the epsilon-greedy policy to make sure that there is some exploration.

**Double Q-Learning in DQN**

The Double Q-learning is an extension of normal Q-learning, which solves the problem of the maximization bias, which comes from the overestimating of values in normal Q-learning. The problem is solved using two different estimates, one for the maximum action, and one providing the estimate of its value.

Another problem of vanilla Q-learning is the **moving target problem**. It appears when the policy being is constantly changing as the agent updates its estimates of the Q-values. This leads to problems in the convergence process and makes learning unstable.

All of those problems are fixed in our Double Q-learning in the DQN approach. The difference from normal Double Q-learning is that we use two different networks to determine the next action and the value.

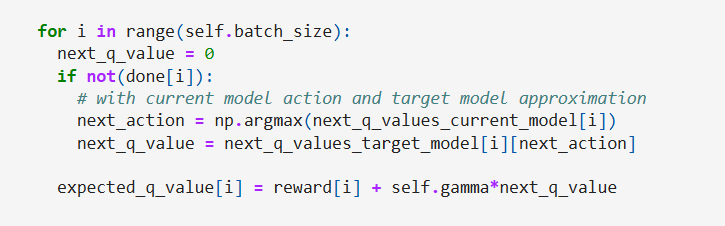
The formula:



Like shown in the formula we determine the next action by using the older network with weights w´to get the next action and the current network with weights w to get the value. Like this we can fix the moving target problem and the maximization bias problem.

**Changes in Code for Double Q-Learning:**

To implement this method we used the already existing solutions of Deep Q-Network. We figured out that the solutions for the Double-Q approach were already implemented as a comment. We commented out the calculation of the next\_q\_value for DQN and uncomment the part to calculate the next action and estimates for Double Q-Learning in DQN. The code is in the compute\_td\_loss\_our(self, target\_model) function:



**Evaluation for Double Q-Learning:**

Xxx

**N-Step Deep Q-Network**

The n-step Q learning method uses almost the same steps as described above, only changing the target. As mentioned above it takes the first n discounted immediate rewards and the (also discounted) estimated Q value for the best action in state S(t+n) which can be seen in the following formula:



It therefore must save in the dataset not only the first reward and the next state, but the sum of discounted n rewards and the next state at step n.

**Changes in Code for N-Step Deep Q-Network**

For our implementation of n-Step DQN, we used the same base as provided in the lecture and modified almost exclusively the ReplayBuffer class:

In this first part, we allow for the n\_step and gamma parameters to be given when initializing a ReplayBufferNStep object.

A screen shot of a computer

Description automatically generated

Then we modified the “push” method to allow for the “truncated” attribute to be passed. When push is executed, we first add the experience to the n\_step\_buffer deque. After that we check if the buffer is full or if we have encountered a truncated or done state. While technically in reverse order, we will first explain what happens when the buffer is full:

If the buffer is full, we will calculate the n-Step version of the first experience in the n-step buffer. We keep the state and action, but we replace the reward with the discounted sum of rewards – calculated by the \_calculate\_n\_stop\_return( ) method. This method also gives us the updated next state and done attribute.

If we encounter a true state for the truncated or done attributes, we immediately process all remaining experiences in the n-step buffer (until it is empty), before the environment is reset in the DQN training method.

A screenshot of a computer code

Description automatically generated

Finally, the calculation of the discounted return. We first set the reward, next\_state and done attributes to default values. Then we go through the experiences saved in the n-step buffer and for each add the rewards with corresponding discount (so first element will be ( 1 \* reward ), second will be (gamma \* reward), third will be ( (gamma \*\* 2) \* reward ) etc. We also constantly update the next\_state and done attributes, so we automatically have the correct values after we process all experiences. Then if we encounter a terminal state, we make sure to not continue adding rewards after it. At the end, we return the discounted sum of rewards and updated next\_state and done attributes.

A screen shot of a computer

Description automatically generated

The last changes are updating the ways in which the replay buffer in the DQN class is created (include n-Step and gamma) and the push method for it handled (include truncated when calling):

