Assignment 4: Q-Learning

Deep Learning

Micheal Denzler

December 17, 2019

1 Wrapper descriptions

- FrameStack wrapper: Returns a lazy array of the last 4 frames, which is called a state with each frame being one of its 4 channels. It has the methods to reset the environment to get back to an initial state, or make a next step within the environment to get the next state, reward, done, and info values.
- ScaledFloatFrame wrapper: Normalizes the frames from [0, 255] down to [0,1] which helps in terms of memory optimization.
- MaxAndSkipEnv wrapper: Repeats the given action 4 times and takes the element-wise maximum over neighbouring states. Adds up the rewards over each 4-skip group. When doing a step, it returns this 4-skip's max frame, total reward, done and info values.
- ClipRewardEnv wrapper: Bins rewards to -1, 0, 1 depending on the reward's signs. This allows to work with simplified -1, 0, 1 rewards during training.

2 Need for a target Q-Network

If we would work with the target network being the online network Q_{t-1} from the previous time step, we could easily run into oscillations. If there is a big update on $Q(s_t, a, \theta)$ this will also change the expectation for $Q(s_{t+1}, a, \theta)$ significantly. This strong change would then also impact target y_{t+1} and hence the entire expectation oscillates. If we in contrary use an older Q as the target, we can soften such oscillations which in the end also results in a higher probability for convergence as the system is more stable.

3 Necessity of ϵ-greedy policy

It is necessary to act according to an ϵ-greedy policy because we want to train on both exploration steps and exploitation steps.
If we would only act greedily and always take the next step with the highest expected reward, we would never explore new paths which might lead to a higher total reward in the long term. Furthermore we initialize the weights in Q randomly. Therefore initially the maximal action given Q is random as well. We have no guarantee that this action (= argmax(Q)) is actually the optimal one. Most probably it is not and we first need to explore the environment to improve our estimate about Q.

4 Computational Graph and Training

Before building and training the network, I got familiar with the Gym wrappers by following the tutorial. In this way I found out quickly how to wrap an environment and then initialize it, step through it, and end it. Also I saw the different value spaces I could work with, such as state, action, done and info. The states were in Lazy frames that I could simply pack into an numpy array with 4 channels, one for each 84x84 image in the frame.
The actions depend on the game. For Breakout I found 4 actions. The done value was also important because it would show me when I reached the end of a game and had to reset the environment. The only value I did not really use was the info, which would include further information about the game.

Then, as another preparation step before building the training architecture, I wrote the *wrap_atari_deepmind*() function. This function takes an environment ID and a boolean to decided whether to clip the rewards or not. The function then creates an atari calling the *make_atari*() method on the environment ID. This environment is then wrapped again using *wrap_deepmind*(), *clip_rewards* = *clip_rewards* and the further boolean parameters set as requested in the assignment. This environment is the returned by the *wrap_atari_deepmind*() function.

4.1 Computational graph

After that, I build the computational graph for the online network according to the architecture given in the assignment. My input X would be a [?, 84, 84, 4] placeholder since the image was wrapped to 84x84 with 4 channels. The output Y is a simple vector [?] with its length depending on the size of the batch I gave to the network. Further I also implemented an extra placeholder A = [?, 2] which, in combination with $tf.gather\_nd()$, would allow me to pick a specific $Q(s, a, \theta)$ value given an action. This A matrix I would then later use when picking the maximal $Q^\star(s, a, \theta^\star)$ to generate the target values.
The weights in the network layers I initialized with the variance scaling initializers and the biases with the zero initializers. During the convolutional layers I made sure I implemented the strides as requested. Then I flattened the convolutional layers' final output in order to feed it into the fully connected network.

The output Z of the final fully connected layer then had the dimension of [?, k], with k (number of activities) = 4 in case of the Breakout game. Hence, for each row in the batch, I would get an output $Q(s, a_i, \theta)$ for activity a_i, i = 1, ..., k.
The loss was then just the sum of squared differences between Y[j] and Z[j] at position A[j], j = 1, ..., batch size.

For the target network I rewrote the same architecture as for the online network. I just used different variable names to differentiate the two networks.

What I also included in the computation graph was the copying of the weight/bias values in the online network to the target network. This I did with simple tf.assign() calls, one for each weight/bias respectively.

Then I initialized the empty replay buffer with a deque of max length = M (10,000) and defined all the given constants.

4.2 Training algorithm

As the next step, I initialized the environment using *wrap_atari_deepmind*() with reward clipping and implemented the training algorithm. I have created a high-level pseudocode to explain the implementation.

while not yet more than 2 mio iterations done **do**
  reset environment to initial state
  while True **do**
    if random $< 1 - \epsilon$ **then**
      pick best action for exploitation
    else
      pick random action for exploration
    end if
    update ϵ as instructed
    make a step and add (state, action, reward, next state, termination) to replay buffer
    state $\leftarrow$ next state
    if buffer filled up **then**
      if every 4 iterations **then**
        pick 32 random samples out of buffer
        for k = 1, ..., 32 **do**
          create vectors for states, actions, rewards, next states, termination out of the 32 samples
        end for
        vector $Q_{target} \leftarrow$ max(run target network given next state)
        vector y = rewards + γQ_{target} (1 - termination)
        train online network using loss function given states, y, actions vectors
      end if

if every 10'000 iterations **then**
update target network weights given online network weights
end if
if every 100'000 iterations **then**
run 30 evaluations a 5 epochs each using $\epsilon_{eval} = 0.001$ and reward clipping = False
end if
end if
if termination **then**
increase episode count by one
break out of inner while loop
end if
end while
end while

To a far extend it was simply implementing the algorithm given in the exercise. One major additional step was the extra simulation run every 100'000 steps to get an evaluation score averaged over 30 plays.
Also challenging was to figure out how to compute the target y in order to get the loss and train the network. I solved this eventually with this extra placeholder A which would allow me to tell the network which action to pick since the output Z of the network was always an array $Q(s, a, \theta)$ for all possible activities.

4.3 Return during training

As shown in figure 1 and as I expected, while the network learned and the ϵ-greedy policy started to do more and more exploitation, the rewards per episode increased from around 1 in the beginning to roughly 12 during the last episodes.

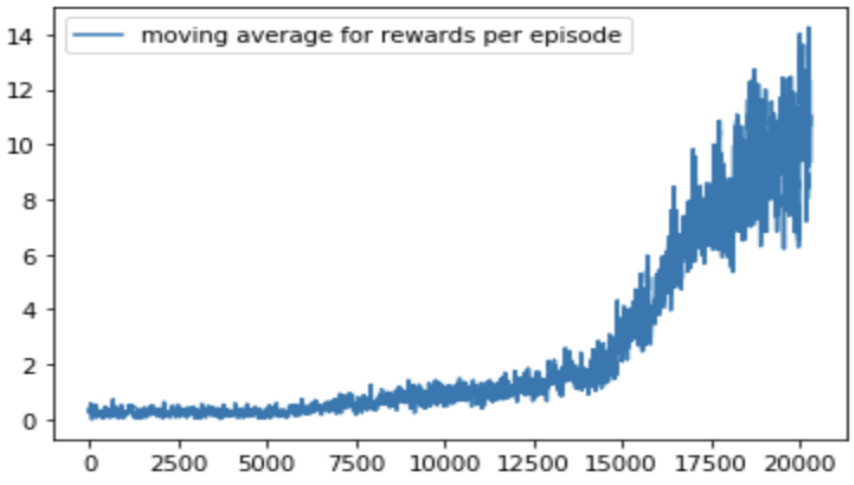

Figure 1: Return per episode, averaged over the last 30 episodes

4.4 Return during evaluation

As shown in figure 2 the average evaluation score increased from initially 1 to almost 350 in the last evaluation step. Since 1 play consisted of 5 episodes, this result means that my fully trained model achieved on average almost 70 points per episode.
Interesting was that after 200'000 iterations I never reached to suggested score of 10 or higher. My score was rather at 2-3. This rather low score I explain myself that due to the high exploration rate in the beginning, the network does not really learn yet how to behave when it gets further into the game. Hence getting high scores at that early point in time was rather rare.

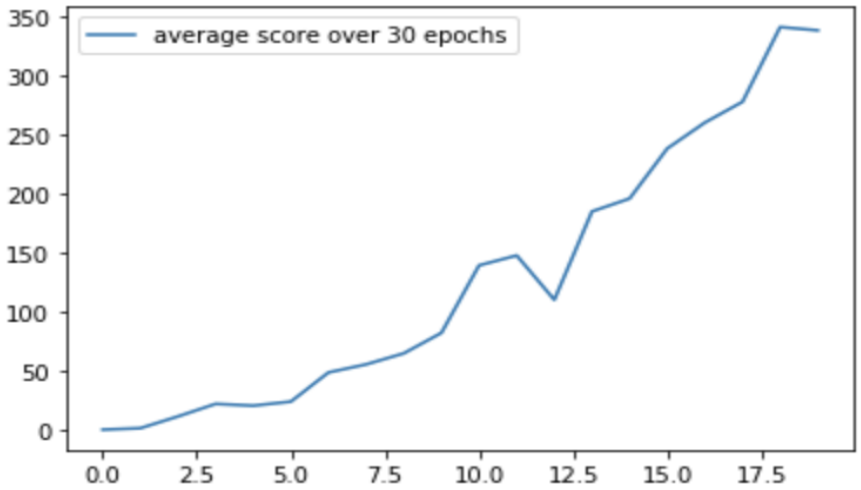

Figure 2: Average score over 30 epochs, taken every 100'000 iterations

4.5 Temporal-difference error

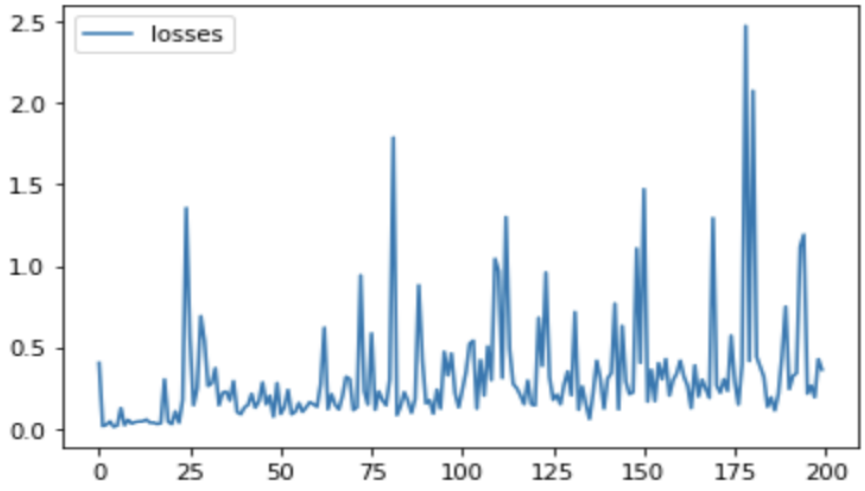

Figure 3: Temporal-difference error at each update step

As shown in figure 3 the loss neither decreased not increased over time. Only the variance of the loss seemed the have increased a bit over time, which might be caused by the expected Q being more and more tuned and hence every time an unexpected event occurred the loss was relatively large.

5 Render one episode

To render an episode of the game, I just had to wrap my environment with the Gym Monitor wrapper. Then I stepped through the game until I hit a termination state and closed the environment.
In figure 4 is a snapshot from the video. The full video of my agent interacting with the game environment is attached.

Figure 4: Snapshot of my agent playing Breakout

6 Updating the target network every 50, 000 steps

When I retrained my model by only updating the target network every 50'000 steps, I ended up with a much slower learning procedure (see figure 5). My explanation is that because I now use a, on average, much older Q_{target} to compute the loss, I slow down the training process. While it is important for stability and convergence to use a separate target Q-Network (section 2), if the target Q-Network is too old, it allows too little movement of the online Q function and therefore decelerates the learning process.
I would however expect that also with this slower settings, I would get to a result, just at much higher computational costs as I would need much more training iterations.

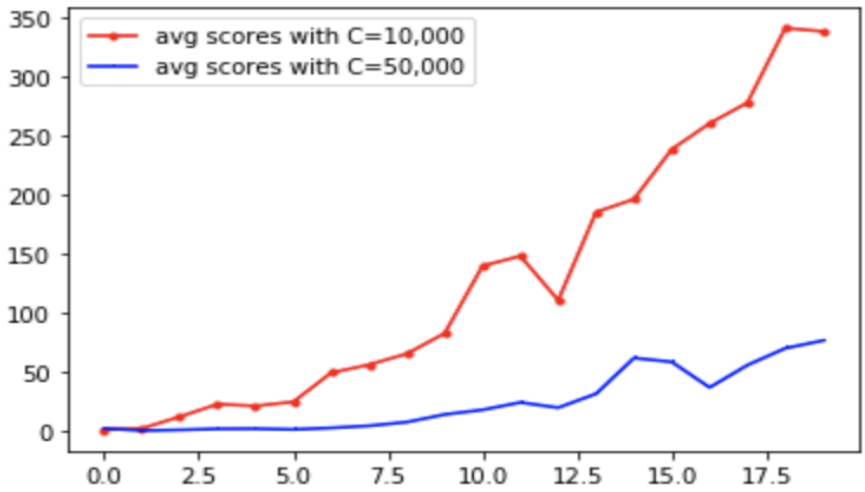

Figure 5: Average score updating the target networking ever 10'000 vs. every 50'000 steps

7 Bonus: Implementing a different Atari game

Finally, I then also applied the model to a different arati game. I chose "AssaultNoFrameskip-v4" because it is a totally different game with spaceships and levels. Also the action space was bigger, as the agent now could chose from 7 different actions.
I was delighted to see that despite Assault being very different to Breakout, my model could easily be applied also to this game.

In figure 6 is a snapshot from the video. The full video of my agent interacting with the game environment is attached.

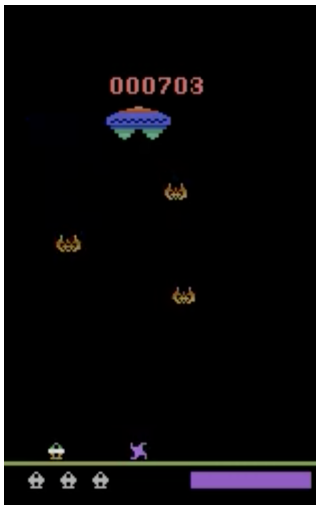

Figure 6: Snapshot of my agent playing Assault

In figure 7 one can see how the return during training evolved. Interesting is that in this case the return increased almost linearly whereas with the Breakout game the curve had more of

a hockey stick shape.

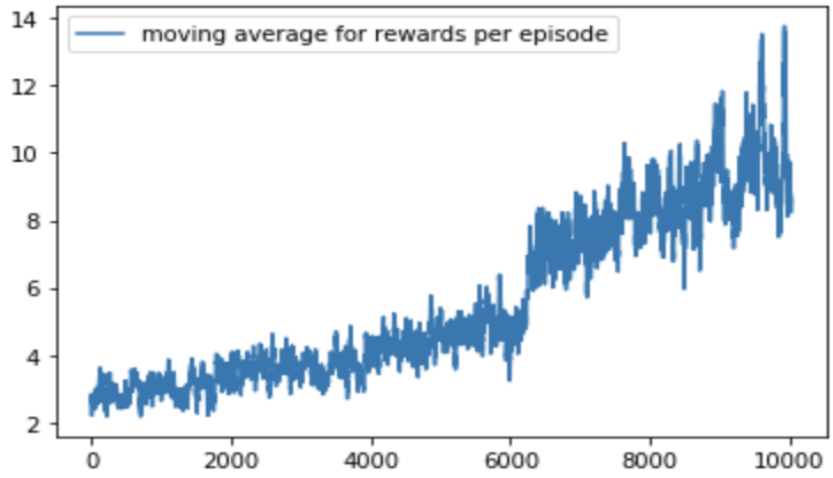

Figure 7: Return per episode, averaged over the last 30 episodes

In figure 8 one can observe how the averaged evaluation score evolved. The first interesting point is that here the scores are much higher. Compared to Breakout, not clipping the rewards to -1, 0, 1 seemed to have a stronger impact in this game as there are some rewards well above 1.

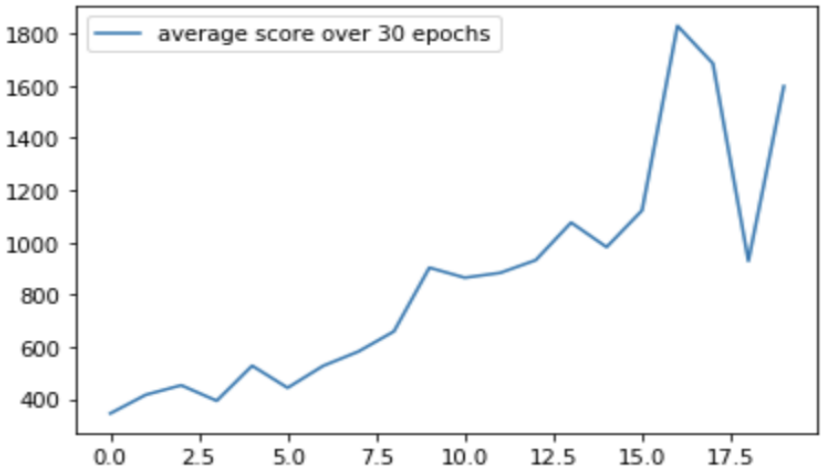

Figure 8: Return per episode, averaged over the last 30 episodes

Further I also found the dig after 1'800'00 iterations remarkable. It might well be that this is due to the high variance such reinforcement learning algorithms have by construction. Another explanation I however found is that in the video after 30 seconds the agent reaches a new level where the enemies start to behave very differently and in addition attack my agent from the side. I therefore guess that after 1'800'00 iterations the train agent reached this level more often which led to several large discrepancies between prediction and target

when being attacked from the side. This lead to a paradigm change in the Q function which let the performance of the evalution runs have this performance dig.

As shown in figure 9, as with Breakout the loss did not change much over the course of the training. Again it looks like the variance of the loss increases a little, although this time the increase looks much less significant as in the case of my Breakout agent.

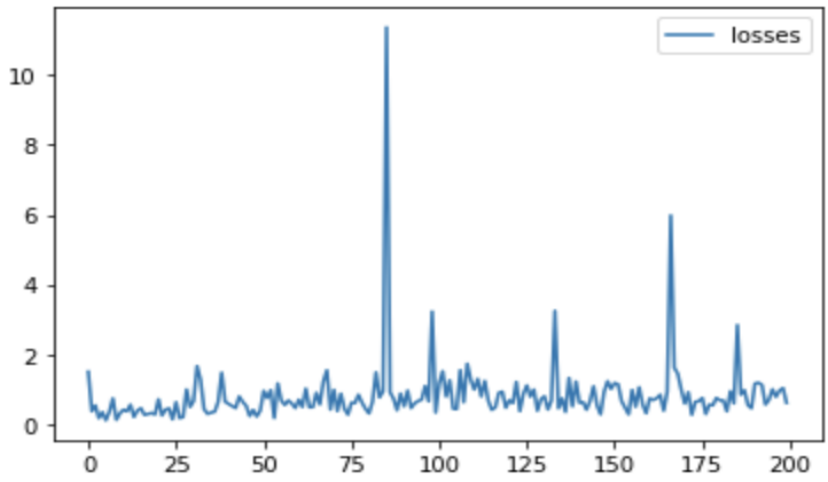

Figure 9: Temporal-difference error at each update step