ELEC-E8125 - Reinforcement lerning

Jarkko Tulensalo, Jesse Miettinen

December 2019

1 Introduction

Reinforcement learning (RL) is a topic which includes many algorithms suitable for learning to play among other things, video games. Currently the best performing RL algorithms rely on deep learning methods [1], and thus they are sub-branched under deep reinforcement learning (DRL). In this project we use a DRL algorithm called deep Q-network (DQN), which basically maps pixel inputs from a video game to control action outputs with a convolutional neural network.

The video game we are trying to learn to control is the classical Atari's Pong -game, see figure 1. There are two players in the game controlling two paddles at opposing sides of the table. The players can choose between three actions: move up, move down and stay put. Their goal is to pass the ball back to the opponent's side of the table and score as much as possible by making the opponent miss the ball, see figure 2.

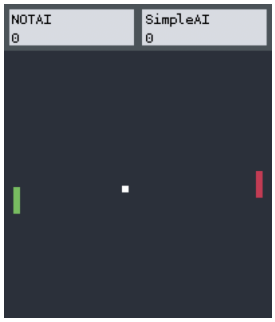

Figure 1: A snapshot from the gameplay.

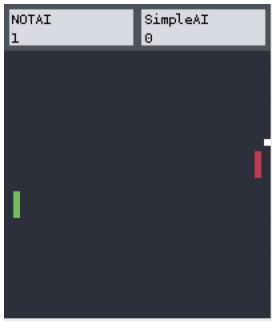

Figure 2: Player scores, as the opponent misses the ball.

The ultimate goal for this project is to use reinforcement learning to train an agent that plays and wins at high probability. The game environment is based on OPEN AI's gym environment. (https://gym.openai.com/envs/Pong-

v0/) The game is played as episodes, each episodes ends when one of the players misses the ball. If the agent scores, it receives a reward of 10 in the end of the episode, if the opponent scores, the agent receives a reward of -10. The best way to measure the performance during training is through average reward during recent episodes. Another good way to analyse the agent's performance is through average episode lengths. If the episodes are short, the agent probably does not follow the ball very well. On the other hand, if the episodes are long, the agent has probably learned the dynamics of the game and can pass most of the balls back.

2 Background and Review of external sources

2.1 OpenAI gym (Pong-v0)

As previously was discussed, Open AI's gym environment was the engine behind the whole pong-problem. This needs no more elaboration. However, there was also some modifications done for the pong-environment by the course staff. The environment we used was called "WimblePong". Wimblepong took care of rendering and was used as the interface between the agents and the environment. Course staff also provided a simple AI agent for training opponent.

2.2 Pytorch (CNN)

It is important to use function approximation when the state and/or action space gets high dimensional. In this case, the action space was very small including only 3 actions. However the paddles could be at least in 21 different positions each, which means that the state space for paddles is $21 * 21 = 441$. This was measured by moving the paddle in the highest possible state and then computing the steps taken for the most low state. Then again the ball's state space can be roughly estimated as 200 pixels * 200 pixels. It is not accurate but just a rough measure that the ball's state space is high, roughly $200 * 200 = 40000$, thus there are roughly $441 * 40000 = 17.64 * 10^6$ states in the game. Thus, function approximation is definately needed. We used the latest version of pytorch for the function approximation. (https://pytorch.org/) is an open source deep learning library inluding multiple algorithms for building and optimising convolutional neural networks (CNN) that are especially handy for image recognition. Thus we decided to use pytorch for the base of our deep Q-network (DQN).

2.3 Aalto RL-course - DQN agent

Aalto's RL course's 4th assignment included an example of DQN. We built our own DQN agent based on the structure of this example. Most of the functions such as network updates were used as they were. ReplayMemory from assignment 4 was also used almost as it was given.

2.4 Pickle and getkey functionalities

Getkey function was used for enabling human players to interface with the game. The function reads which key is pressed from the keyboard and then sends it for the environment. This function worked well for collecting expert demonstrations from human selected actions. Pickle is a software package including functions that enabled us to compress and save the content pushed in ReplayMemory and load them to another agent's memory later on.

2.5 dataset aggregation - dagger

The goal for aggregating the datasets with expert demonstration during the initialization of the replay memory is to jump start learning. With DQNs which are off-policy algorithms, this can be done. The experiences from these demonstrationss would then be sampled heavily during optimization steps in the beginning of DQN training. This should be biasing the trained agent to play more like the expert. The agent would naturally still do exploration in the beginning, but as there were little explored experiences stored, most of the samples drawn from the agent's memory should be "good" samples, in many cases showing how to win an episode and how to not lose. [4]

2.6 Weight initialisation

When initialising any neural networks, the parameters have to be set on some state. With extreme luck, the parameters would be set in a way that would not need optimization. However this situation is highly unlikely. Thus typically people leave the luck aside and initialise the bias parameters as zeros and weight parameters as random.

2.7 External sources for information

Hyperparameter tuning and preprocessing ideas froma blog post [2]. Weight intialisation was based on course material Actor-critic model and other suggestions found in previous studies. [5]. We used reference model as base how to set-up DQN layers in Atari games with Pytorch [3].

3 Approach and method

3.1 Data preprocessing

The feedback from the environment was a raw image of the game as a 200x200 image. In the task, the agent was only allowed to use the pixels of the observed images as inputs. We used three preprocessing techniques for each frame. First, changed the frames from color to greyscale. Second, we downsampled each frame to 100x100 image in order to speed up the computations. Finally we stacked two adjacent observed frames from the environment into a stacked array of (2, 100, 100). The stacked array we used as input to the agent.

We experimented with stack sizes of 2, 3 and 4. The original DQN Atari paper uses a stack size of 4 with frame skipping of 4 [6]. However, our stack of 4 did not manage to learn very well after 4 000 000 frames, as a stack of 2 had already a better behaviour after 3 000 000 frames. The random frame skipping in our environment definitely made it harder for the agent to learn from stacked frames. Using four frame stacks would have needed more frames to learn from variance, or even made it impossible.

3.2 Deep Q-learning network (DQN)

DQN's apply neural networks as function approximators; it uses two neural networks simultaneously. They are called policy net and target net, which are also updated separately. The target net was used to predict next state action value by predicting the next state values of the possible actions and then selecting the action maximising the next state value. The policy net was used to predict also the next state values for all possible actions. In the network update the loss -term was computed as the L1 -loss between the max value of the target net output and the policy net's prediction with the action chosen during gameplay. The policy net is then optimised with RMSprop-algorithm. The algorithm showing this structure is shown in figure 3. Note, that the last row in the Algorithm 1 differs, as we update the target network by simply copying the policy net's parameters every 2500'th frame.

The structure of the DQN has been put to a table. Briefly, there are 3 convolutional 2D-layers, which can process patterns in the images and two fully connected layers taking care of mapping the outputs to the action space. The structure is based on parameters found in literature for Atari games [3].

3.3 Experience replay

The basic idea of the replay memory was that it could store pre-defined amount of explored experiences. During the training, the agent plays episodes, from which experiences are then stored in ReplayMemory as tuples of (state, action, reward, next state, done). ReplayMemory can be used for drawing batches of previously collected experiences for policy optimization. We also added some extra functionalities that are discussed more briefly in the section 3.4.

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\ Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:
      $\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$

Figure 3: DQN algorithm used in the project.

Figure 4: DQN model

Graphs				
# layer	layer type	#out channels	kernel size	stride
Layer 1	Conv2d	32	8	4
Layer 2	Conv2d	64	4	2
Layer 3	Conv2d	64	3	1
Layer 4	Linear	512	-	-
Layer 5	Linear	3	-	-

3.4 Imitation learning

We programmed an interface and an extra agent, that applied actions chosen by humans to the game. It had it's own replay memory that the human actions and corresponding experiences were stored in. We call this set of stored human actions and environment responses expert demonstrations, since they included a lot of winning experiences. The replay memories of our agents were modified to include functions that could either save their currently stored experiences to a file or load experiences from a file. This way the agent could sample expert demonstrations from the memory when it does network updates.

3.5 Weight initialisation

We experimented with different weight initialisation techniques for the policy and target DQN neural network layers. We noticed that random initialisation had a downside that the agent preferred to jump instantly from game start to one of the other corners, up or down and stayed there. In literature, most common weight initialisation techniques were normal distribution and xavier uniform distribution. We experimented with both and found more promising results with normal distribution.

3.6 Hyperparameter tuning

We experimented with six different hyperparameters related to replay memory, optimiser, target update frequency and number of frame stacks. The experimented values and the chosen value are presented in Figure 6. In contrary to previous studies, we used replay buffer size of 100 000 due to memory limitations. Smaller replay batch size had a tendency to speed up the learning of the agent which is why we chose 32. There was a quite clear consensus in previous studies to use gamma parameter of 0.99. We found usage of both learning raters 1e-4 and 1e-3 in literature. We chose 1e-4 based on our own experiment. There were multiple parameters that were used for target update frequency depending on environment. Simpler environments worked with smaller target update frequency as more difficult environments required a longer one. We assumed that Pong environment is one of the simpler ones which is why we

chose a target frequency of 2500 frames. In contrary to almost all Atari DQN models, we preprocessed the frames with 2 stacks due to differences in our environment random frame skipping which we explained in Section 3.1.

Figure 5: Experimented hyperparameters

Parameters			
Parameter	Experimented values	Chosen value	Definition
replay buffer size	100 000, 1M	100 000	The size of replay memory buffer
replay batch size	32, 64	32	The size of batch taken from replay memory
gamma	0.99	0.99	Discount factor of the reward function.
learning rate	1e-4, 1e-3	1e-4	Learning rate of the RMSProp optimiser
target update	2500, 10000	2500	Target update frequency in number of frames
frame stacks	2, 3, 4	2	Number of frame stacks given to the agent.

3.7 Exploration

We used Greedy in the Limit with Infinite Exploration (GLIE) method during the model training. GLIE method has an advantage of starting with high exploration and gradually increasing the amount of exploitation and converging to a better optimal policy. Also, infinite exploration allows the model to avoid getting stuck in a local optimum. [**?**] We started by exploration factor 1 which decreased gradually to 0.1 until 10 000 episodes. Between 10 000 and 100 000 episodes, the model used an exploration factor of 0.1 as shown in Figure 6.

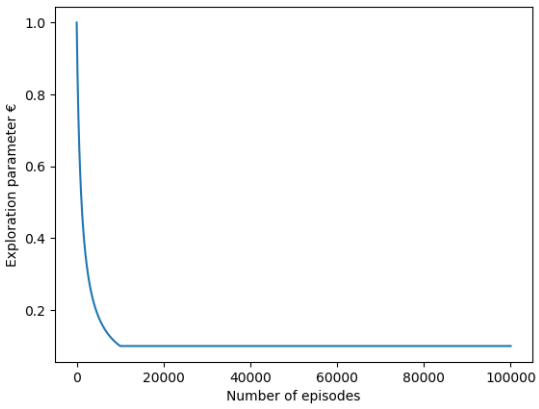

Figure 6: Exploration parameter ϵ per episode

4 Results and performance analysis

4.1 Main results

We performed a test where our trained agent played against the given pretrained SimpleAI agent for 500 games. The total score of the game between our trained agent and SimpleAi was 21 - 479 in the advantage of the SimpleAI. Figure 7 shows the final result of the test game.

In class competition, our agent finished 47th out of 55 competitors with a win rate of 23.7 % against the other classmate agents.

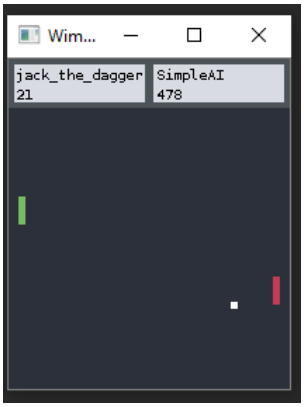

Figure 7: Final result after 500 games between our agent and given opponent SimpleAI.

4.2 Training results

We trained our model with 100 000 episodes wit a total amount of 5 300 000 frames. We monitored three parameters in order to follow the learning curve of our agent: number of wins, average rewards per 100 episodes and number of frames per episode. After the full training, our agent had won 153 times out of 1000 000 episodes which corresponds to a total win rate if 0.2 percentage. We also followed the average wins per 100 episodes but did not observe notable increase in wins per 100 episodes during the training time.

We noticed that the duration of each episode increased in number of frames during training, which inclines better performance of our model even if our win count did not increase substantially. This of course relies on assumption that the opponent SimpleAI wins the game and our agent is able to hang on longer. A quick win would show improved performance even if episode duration is small. Figure 8 shows the duration of the game in the first 2000 episodes in number of frames and Figure 9 shows after 70 000 episodes. We can observe that the average duration of the game increased from below 50 frames in 2 000 episodes to an average of 100 - 150 frames after 70 000 episodes. Also, the maximum duration of the game increased from 250 frames to 350 frames. We can also observe that the average duration of the game does not change notably between 10 000 and 70 000 episodes.

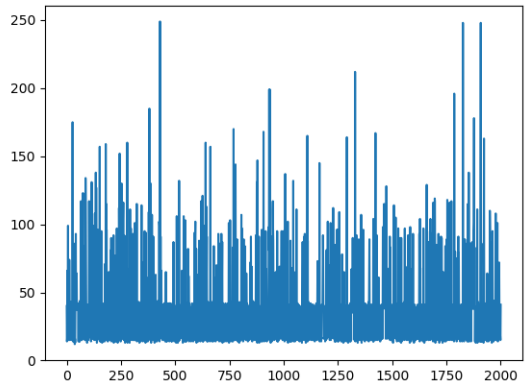

Figure 8: Duration of game in frames in first 2000 episodes.

Figure 10 shows how the rewards per episode progressed during 100 000 training episodes. Reward after a win equals 10 and after a loss -10. We can observe that in contrary to training time, the agent seems to win less game in the end of the training. We used Greedy in the Limit with Infinite Exploration method where the exploration factor

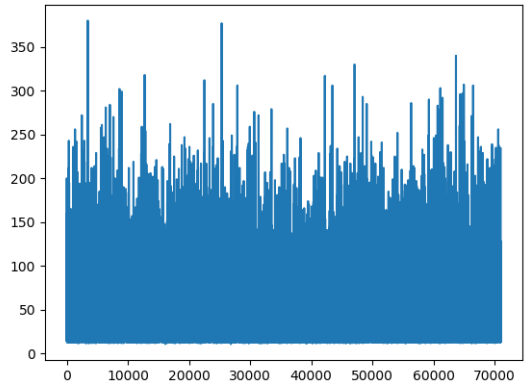

Figure 9: Duration of game in frames after 70 000 episodes.

decreased gradually to 0.1 until 10 000 episodes. It seems as the agent performed better in the first 10 000 episodes when the exploration factor was higher.

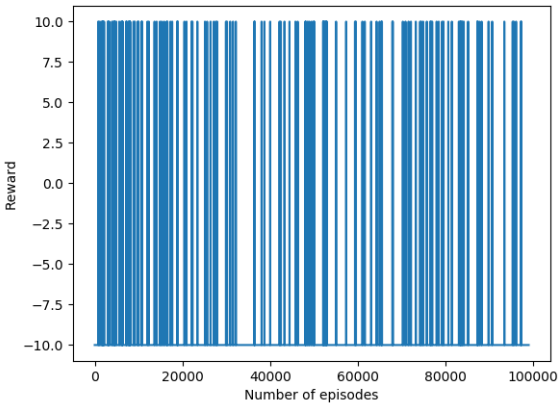

Figure 10: Reward per episode in 100 000 episodes.

4.3 Performance analysis

The performance of our agent was similar against the SimpleAI and in the class competition where our agent did not perform as well as previous studies suggest. Observing the behaviour of our agent in tests against the SimpleAI, we found that the agent had learnt that it could beat the opponent by bouncing the ball with a large angle and could play well short amount of times. Nevertheless, the agent still had problems in following the ball reliably. Sometimes our agent followed the game very well and fast, sometimes it went to a clearly wrong direction or stopped just before where it could have hit the ball.

Even from low win percentage, we could observe that our agent learnt over time. The initial agent had a tendency to jump to one of the corners. After 3 000 000 frames, our agent seemed to have learnt to follow the ball and win games once in a while.

We experimented with different parameters but did not seem to find a suitable combination to improve the per-

formance of our agent more. Initially, we followed the parameters that have been used in literature and in original Atari DQN paper. However, we noticed that common parameters used in literature for Atari games did not work as the random frame skipping in our environment was different to the original Pong environment. Also, we noticed that many of the papers we read, used even training of 1 000 000 episodes which we did not have time to do. Eventually, we decided to go with one set of parameters and used the last week to try to run as many episodes as possible. However, this did not result in a top-performing agent. What was curious, that our agent did not really improve performance between notably 10 000 and 100 000 episodes.

We also noticed in the end that it took about 3 000 000 frames in order to see if the agent actually works with the given parameters. This, however, took a simulation time of a day or two which meant we had a quite limited amount of experiments we could try. Also, we wonder if we limited some valid experiments in the beginning just because we did not see any progress in the first 1 000 000 frames.

We used imitation learning in order to speed up the training process. However, we noticed after the class competition that we had different preprocessing related to stacked frame selection during the imitation learning phase and the learning from playing the game. This means that our imitation learning more likely confused our agent rather than increased the training speed. After noticing the difference in preprocessing, we did not have time train the agent anymore.

We also found many different theories we could try to speed up the training or learning speed with more time. Parallesisation of GPU usage would decrease the training time and allow us to do more experiments on hyperparameters. We found DQN and prioritised experience replay suitable improvements to our model.

5 Conclusion

In this report, we present a reinforcement learning agent that can play the game of Pong by just using the pixels of the frame as input. Our proposed agent is a DQN model with experience replay. We show how expert demonstration can be used to increase the learning speed of an agent with experience replay memory. We present data preprocessing, weight initialisation and exploration techniques that increase the performance of a reinforced agent. We show 6 different hyperparameters that we experimented with and based on our findings show values that were used to participate in the class competition for our agent.

Our trained model did not perform as well as results from literature show for similar agents trained to play Atari games. This is why we suggest further work in hyperparameter tuning and more exploration. Also, we discuss further work in increasing the speed of training. We noticed that common parameters used in literature, could not be used directly with the given environment as the random frame skipping was different from the original Pong environment.

6 Team member contributions, information, and collaboration with others

We contributed to the project with equal share. We had informal conversation with Julius Hietala whose experiments seemed to work.

6.1 Jesse Miettinen (350417, jesse.miettinen@aalto.fi)

- Report: Sections 1-3.4

6.2 Jarkko Tulensalo (84071T, jarkko.tulensalo@aalto.fi)

- Report: Sections 3.1, 3.5-5

References

[1] M. Brundage M. Bharath A. Arulkumaran, K. Deisenroth. A brief survey of deep reinforcement learning. 2017.

[2] Adrien Ecoffet. Beat atari with deep reinforcement learning! (part 1: Dqn). [Online; accessed 12-December-2019].

[3] Shivantshu Gupta. Pytorch-double-dqn. [Online; accessed 12-December-2019].

[4] Jonathan Hui. Rl - dqn deep q-network. [Online; accessed 12-December-2019].

[5] Doshi Neerja. Deep learning best practices (1) — weight initialization. [Online; accessed 12-December-2019].

[6] Volodymyr et al. Playing atari with deep reinforcement learning, 2013. [Online; accessed 12-December-2019].