Project report

Oskari Järvi, Lauri Westerholm
ELEC-E8125 - Reinforcement Learning

December 7, 2020

1 Introduction

This paper is a report for ELEC-E8125 - Reinforcement Learning course project. During the project two different reinforcement learning approaches, DQN and PPO, were developed for a custom version of Pong, a traditional Atari game. In the customized game, a reinforcement learning agent plays against so called simple AI which uses absolute paddle and ball positions to select actions based on distance measures. The RL agent is required to use pixel inputs; each episode ends when the agent either wins or loses a single game round.

As primary project outcomes, the DQN approach was able to reach a good level of performance in the game. Its win-rate against the simple AI converged near 80%, whereas the PPO-based approach failed to properly learn the environment.

The remaining document is organized as follows: section 2 reviews external sources used during the project, section 3 provides details on RL agent architectures, section 4 overviews the training procedure for both approaches, and section 5 evaluates results of the DQN-based approach concentrating on the best achieved performance. The paper ends to conclusions which are presented in section 6.

2 Review of external sources

DQN is a variant to Q-learning [1, p.153] and it was first introduced by Mnih et al. [2] in 2013 when they presented a method to learn Atari games directly using pixel inputs. DQN uses similar temporal difference, TD(0), updates to regular Q-learning. The main differences between the approaches are that DQN utilizes a deep neural network, e.g. CNN, to extract features and it also implements experience replay with ϵ-greedy policy. In practice, the experience replay is accomplished by storing N previous state, action, reward and next state combinations to a buffer from which a batch is uniformly sampled each update step. The DQN agent randomized experience replay python code is based on Pytorch DQN example code [3].

Rainbow DQN publication by Hessel et al. [4] compared multiple methods which can be used to enhance performance of DQN-based RL agents. Specifically for Pong, the paper suggest that prioritized experience replay and multi-step updates should have a positive contribution to the agent performance. After reviewing the rainbow paper results, it was

decided to implement priority memory to the DQN agent; multi-step was not implemented primarily due to time constraints. Regardless of not being suggested to improve performance, DDQN-based updates were still implemented to the agent for comparison.

The DQN agent prioritized experience replay was implemented based on a publication by Schaul et al. [5]. In priority memory each batch is sampled from a weighted distribution where sampling probabilities are formed by previous TD-errors. Double Q-learning (DDQN) was introduced by van Hasselt et al. [6]. In a practical implementation of DDQN, the greedy policy is evaluated on the online network but Q-value estimates are computed with the target network. This procedure should lead to more stable network updates.

Additionally, Andrej Karpathy's blog post [7] provided ideas for pixel input pre-processing as it proposed to feed difference frames to the RL network.

Proximal Policy Optimization, PPO, is a policy gradient algorithm introduced in a paper [8] by Schulman et. al of OpenAI. The paper goes through the main parts of the algorithm, shows a performance comparison to other policy gradient algorithms and a comparison to other surrogate objectives with different hyperparameters. The essence of the PPO algorithm is trust region method called clipped surrogate objective. In a blog [9] by OpenAI, they simplify that this method "implements a way to do a Trust Region update which is compatible with Stochastic Gradient Descent, and simplifies the algorithm by removing the KL penalty and need to make adaptive updates". They also state that it is their default RL algorithm due to its relative simplicity and good performance.

Another paper that was studied to help with PPO is an empirical study by Andrychowicz et al. [10]. In this paper the research team goes through 250,000 on-policy RL agents in five continuous control environments. The goal of this study is to provide insights and recommendations for the training of RL agents. This includes, for example, choosing good hyperparameters. Even though this paper was studied, it was not utilized in this project as we could not get the PPO agent to learn.

3 Design of the agent architecture

3.1 Pre-processing and feature extraction

The customized Pong environment input frames are 200x200x3 arrays as the play area is 200 pixels wide, 200 pixels high and the images include RGB-channels. However, as there are only four different distinctive colors used, all the information necessary for a RL agent can be compressed into a single channel. Thus, during the pre-processing each frame is first converted into a gray-scale array by applying different weights to each dimension such a way that the end result for each pixel is a floating point value in the range of (0, 1). This approach significantly reduces the amount of redundant information making input frames more memory efficient.

Besides gray-scaling the input frames, the pre-processing also included input down-scaling. By doing max-pooling operations on the image dimensions, the original 200x200 arrays could be compressed into 100x100 arrays with minimal information loss. Again, the down-scaling was implemented to make the frames more memory efficient. Smaller input

dimensions typically also help in feature extraction as long as no necessary information is lost during the down-scaling process.

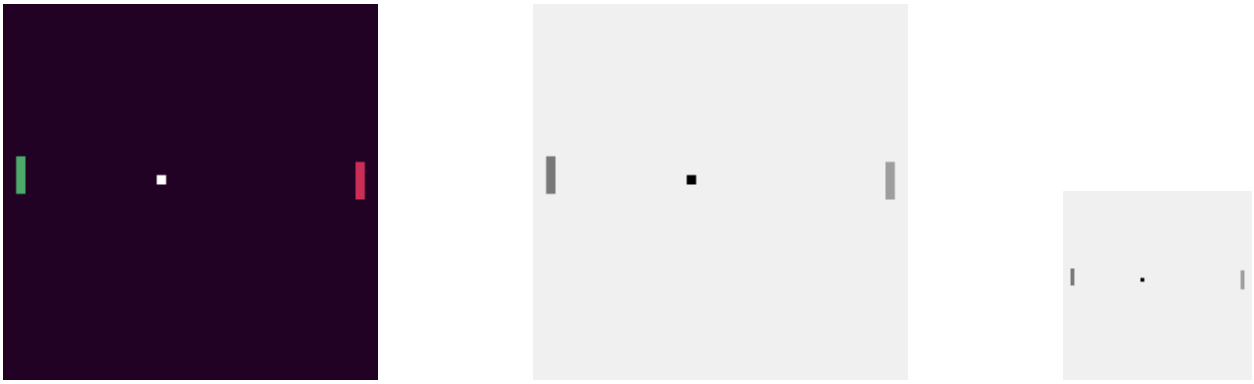

Figure 1: Frame pre-processing. 1. original, 2. gray-scaled, 3. gray and down-scaled.

In order to preserve information on how the objects move in the game, multiple frames were either stacked or difference frames created. The DQN agent frame stacking implementation supports stacking two or four frames forming an input frame with multiple channels. For the frame difference implementation the input remains one-channel but the one frame is composed off summing two game frames together with different weights. This keeps colors, excluding the background, separated. Finally the "difference frame" is scaled to the range of (0, 1).

For feature extraction, the decision was to use a quite simple CNN architecture with three convolutional layers (with ReLu activations) and two fully connected layers where the last fully connected layer is the network output layer. A CNN-based feature extraction network was selected as CNNs are widely known to operate well with image data. Furthermore, CNNs are significantly easier to implement than, for example, Autoencoders or GANs which could be also used for feature extraction. The CNN is not trained in supervised manner, rather it uses RL agent losses which typically tend to be noisy. Thus, using a relatively simple CNN should provide the best results.

3.2 RL agent

DQN algorithm was selected as one of the alternative approaches primarily due to its good performance on discrete environments and relative simplicity. As DQN is an off-policy method supporting replay memory, it is significantly more sample-efficient than on-policy approaches such as most of the policy gradient variants. Furthermore, having a deep neural network feature extraction layer is a critical for pixel-base learning. For such a large input state-space baseline algorithms, such as Q-learning, do not work well as there are too many state-action combinations. The main benefits of using policy gradient methods over DQN are typically in continuous action-spaces rather than in discrete environments like the customized Pong.

PPO was selected as the second algorithm and a policy gradient alternative to DQN. In the Karpathy blog [7] it is also stated that the current popular approach to RL problems

are policy gradient methods and that they usually achieve better results than DQN. This approach uses the same pre-processing methods for the game frames and difference frames that are calculated from two consequent frames as in the DQN implementation.

The neural network is the same architecture as described above for the DQN, however the output from the last linear layer is taken as an input for two separate additional linear layers. One linear layer calculates the weights for logits that are used in construction of a categorical distribution from which the actions for the agent are sampled from. The other layer approximates the value for each state.

The agent stores relevant information from each timestep into a batch by running the current policy in the environment. The number of timesteps taken for one policy is determined by a hyperparameter. When the desired number of steps have been ran, the advantage is calculated by getting the value each batch step and calculating the rewards-to-go for the batch. After that the policy is updated by maximizing the clipped surrogate objective.

4 Training methodology

4.1 DQN

Hyperparameter values for the best DQN-agent are shown in Table 1, the corresponding training rewards plot is shown in Figure 2.

The training was started with high exploration where the initial epsilon threshold was set to $\epsilon_0 = 0.99$. The epsilon was exponentially annealed during the training reaching the final value of $\epsilon_1 = 0.05$ in approximately 1200000 frames. Thus, the agent was still at the end of the training exploring quite significant amount. Smaller final epsilon values were also tested but they did not provide better results.

The discount factor γ was set to the value of 0.99. Rewards are sparse in the customized Pong environment as the agent is only rewarded at the end of an episode: a positive reward of 10 is given if the agent won, otherwise the agent is penalized with a reward of -10. Using a relatively high discount factor ensures that the Q-value updates are properly propagated towards the initial states.

The DQN-agent learning rate was initially set to the value of $lr_0 = 1 \times 10^{-4}$. With higher learning rates the network did not seem to learn properly which is probably due to too noisy updates being backpropagated to the feature extraction CNN. In order to fine-tune the network layers, the learning rate was dropped when the agent had been trained for 5000, 8000, 12000 episodes respectively: $lr(5000 - 8000) = 4\text{e}{-5}$, $lr(8000 - 12000) = 2\text{e}{-5}$, $lr(12000-) = 1\text{e}{-5}$. In total, the agent was trained for 20000 episodes which should be enough for the learning to converge as visualized in the rewards plot, Figure 2.

The network batch size was selected to be 64 as it provides a good combination of update steadiness, when compared to smaller batch sizes, while still being computationally easily achievable. Memory size was selected to be large enough to provide good sample-efficiency, the ultimate limiting factor was GPU memory size and the selected pre-processing scheme. As a result, the memory size needs to be adjusted for different configurations.

Besides DQN combined with priority memory, the agent was also trained with randomized replay buffer and DDQN updates. Typically, the DDQN updates were found to have no clear

Parameter	Value
ϵ_0	0.99
ϵ_1	0.05
γ	0.99
lr_0	1×10^{-4}
batch size	64
memory size	15000

Table 1: DQN hyperparameters.

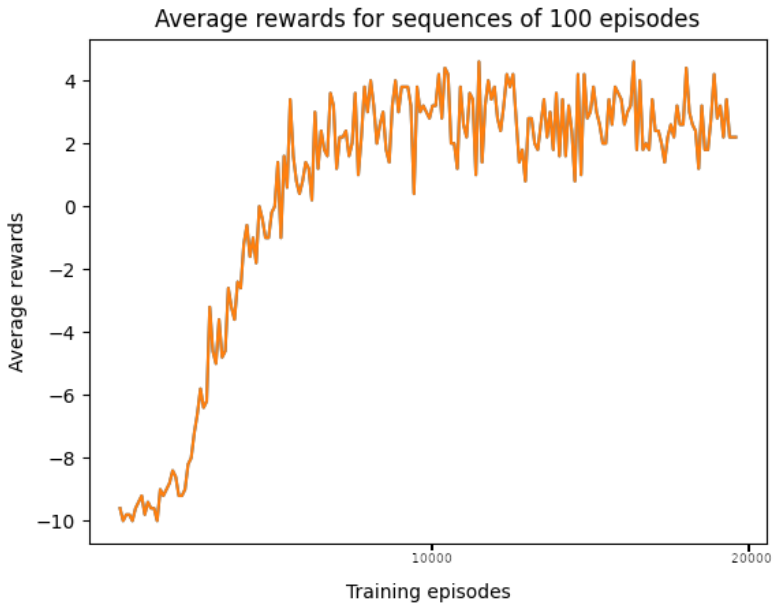

Figure 2: DQN training reward averages for 100 episode sequences. The training starts to significantly slow down around 8000 episodes: average rewards won't anymore steadily rise.

contribution to agent performance when compared to regular DQN updates. Only in the case where the target network was rarely updated the results using DDQN were worse than DQN. On the other hand, the use of priority memory ultimately produced a little better results than randomized replay memory. However, with the selected epsilon values, these differences did not clearly show in training plots visualizing average rewards.

Furthermore, different frame stacking and frame difference computation approaches were tested for the DQN-agent. The best results were produces by stacking four frames, which also contains the most information, but results obtained from stacking two frames or computing frame differences were extremely similar. From a computational perspective, stacking four frames obviously requires four times as much memory as the frame difference pre-processing. Thus, it can also limit viable configurations of other parameters: e.g. using four frame stacking with non-downscaled frames and replay memory size of 15000 frames requires approximately 25 GB of memory.

4.2 PPO

We could not get the PPO agent to learn. The agent does not seem to learn at all as shown in Figure 3. Average rewards for 100 episodes stay around -10. The model might randomly win in the beginning with pure chance, however the average rewards still converge to -10. As the implementation of the agent is clearly flawed in some way, the training for the agent was not fine tuned, i.e. no good hyperparameters were chosen.

We think that the reason why the implementation does not work is most likely due to a flawed handling of the data used in the training. This might result from some incorrect usage of PyTorch tensors or functions. The batch data collection function might not work as intended as well. We came to this conclusion as we could not find any problems with the overall architecture of the agent or the training procedure. We decided to leave this agent as it is and focus on the working architecture due to time constraints.

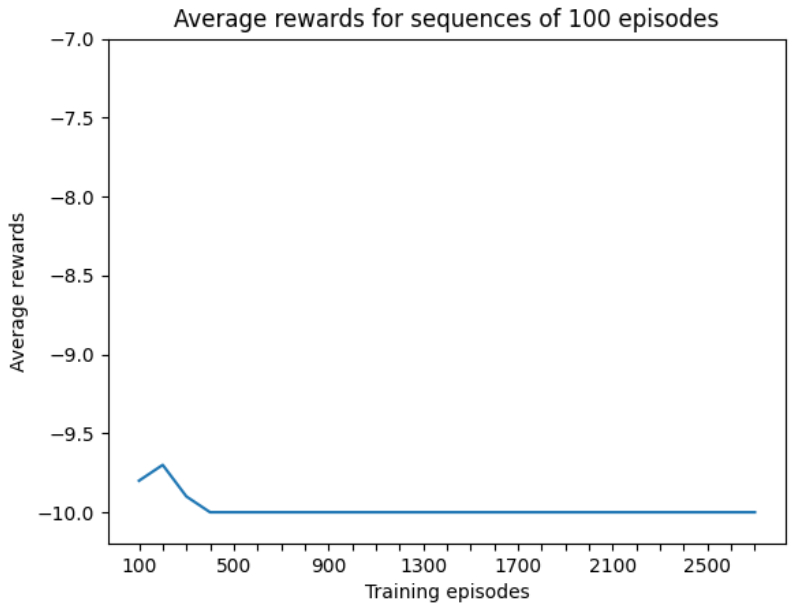

Figure 3: PPO training results. The model does not learn.

5 Evaluation of results

When the DQN-agent was tested against the simple AI, following results were observed after repeating the tests multiple times:

1. For 100 episodes WR = 80% ± 5%
2. For 1000 episodes WR converges to approx. 80% (± 1%)

From these results it is possible to see that the agent performs significantly better when there is no exploration involved in the choice of actions. By comparison, at the end of the training agent's WR was fluctuating near 70% which corresponds to average rewards of 4 in the training plot, Figure 2.

When evaluating the performance visually, two main reasons for a loss against the simple AI can be seen. The first reason why the agent misses the ball is because the model behaves too aggressively by aiming to hit the ball with the edge of the paddle. This behavior occasionally creates a scenario where the agent loses by missing the ball by a very small margin. The second reason why the agent loses is simply a wrong Q-value for a state. This can be seen from the fact that the behavior of the agent is illogical. For example, when the agent should move upwards it stays put. These problems might result from the nature of the DQN; the model chooses a policy and as the exploration rate gets smaller, the model does not find a better policy.

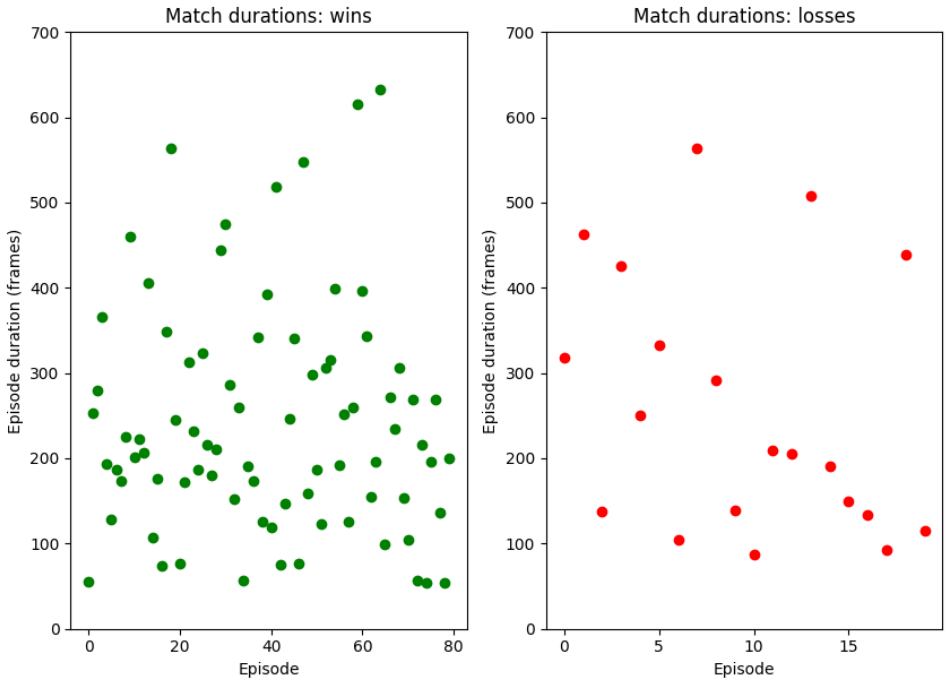

Figure 4: Episode duration comparison between wins and losses.

In Figure 4 the match durations for both wins and losses are shown. For 100 matches the majority of losses are focused around smaller match lengths which might result from the nature of the model being aggressive, i.e. this could imply that model plays in a high risk high reward manner; it wins most of the games fairly fast with few longer games and it also loses the majority of the losses in shorter games.

6 Conclusions

In this paper, two approaches to solve a RL problem in a custom Pong game environment were taken; DQN, an off-policy learning algorithm and PPO, an on-policy algorithm. The implementation PPO failed thus the main focus of this paper was the DQN algorithm and its results were evaluated.

The DQN model manages to win the majority of games against the simple AI, however there is still room for improvement. The Rainbow DQN [4] paper shows that the DQN algorithm in the Atari Pong environment benefits greatly from multi-step learning and a prioritized experience replay. Prioritized experience replay was implemented but the multi-step learning was not. Multi-step learning could be added to the existing algorithm with further development. Performance of the agent could also be improved by fine tuning the hyperparameters, however this is time intensive as training a single model takes quite some time.

With some further efforts the PPO algorithm could be fixed and a policy gradient alternative would probably also achieve better results. The reason why the agent does not learn is somewhat uncertain but a possible reason was presented in section 4.2.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[3] "Reinforcement learning (dqn) tutorial." `https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html`. Accessed: 2020-04-12.

[4] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," 2017.

[5] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

[6] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.

[7] "Deep reinforcement learning: Pong from pixels." `http://karpathy.github.io/2016/05/31/rl/`. Accessed: 2020-04-12.

[8] "Proximal policy optimization algorithms." https://arxiv.org/pdf/1707.06347.pdf. Accessed: 2020-05-12.

[9] "Proximal policy optimization." https://openai.com/blog/openai-baselines-ppo/. Accessed: 2020-05-12.

[10] "What matters in on-policy reinforcement learning? a large-scale empirical study." https://arxiv.org/pdf/2006.05990.pdf. Accessed: 2020-05-12.