ELEC-E8125 - Reinforcement learning project report

Julius Hietala (317447)

December 2019

1 Introduction

Teaching agents to play Atari games only based on pixels of the game environment has been one of the most publicly appealing applications of reinforcement learning in the past years. Since the first paper based on a DQN agent [4], many different approaches have been proposed and tried since to solve the same problem. This work will be exploring one of those approaches.

The goal of this project was to implement an agent that is able to solve a similar problem, namely one that is able to play the game of pong [3] using some kind of a reinforcement learning [6] method. As in the original DQN paper, the agent was supposed to be able to play only by processing the pixels of frames given by the game environment.

2 External resources used

Using inspiration from external resources such as articles and code was allowed during the project, as long as they are referenced. The code submitted along with this report includes references to articles/external code as comments for parts that were inspired from somewhere else and/or that are not obvious. Reasoning and motivation for these choices will be described in the next sections, although here is a recap of them:

- Hyperparameter and neural network choices, neural network weight initialization, gradient clipping: https://github.com/rgilman33/baselines-A2C/blob/master/baselines-A2C.ipynb
- Loss function: https://www.datahubbs.com/two-headed-a2c-network-in-pytorch/
- Calculating discounted rewards for episodes was copied from the *ELEC-E8125 - Reinforcement learning* Homework 5 utility code
- Performing frame stacking and resizing was copied from example agents provided by the *ELEC-E8125 - Reinforcement learning* teaching assistants

3 Approach and method

There are multiple different ways to go about solving the problem of playing pong from pixels. After reading articles about how others had solved the task, subjectively it seemed that *policy gradient* based methods had been more popular recently, leading me to explore similar options. One challenge in evaluating approaches beforehand was that all of the reference implementations were using a slightly different environment than what we had (reward scheme, random frame skipping), so the applicability of a particular solution was not evident before actually trying. Figure 1 is illustrating the pong game environment.

The main idea in policy gradients is to maximize a performance measure $J(\theta)$ with respect to the policy's $\pi(a|s,\theta)$ parameters θ, where usually $J(\theta)$ are the rewards experienced by an agent, as is the case in our setting. This setting is in contrast with *action-value* methods where the goal is to find values for state-action pairs and make action decisions based on them. Action value functions may still be used with together with policy gradient methods to learn the policy parameters, but are not required. Such methods combining both policy gradients and value functions are usually referred to as *actor-critic* methods.

In this setting, the maximization step of the performance measure J would be the following: $\theta_{t+1} = \theta_t + \alpha \nabla J(\theta)$, where α is the update step size and $\nabla J(\theta)$ is a stochastic estimate whose expectation approximates the true gradient of the performance measure J w.r.t. to the parameters θ. The *policy gradient theorem* states that the gradient $\nabla J(\theta)$ can be written to be proportional ($\propto$) to $\sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) = \mathbb{E}_\pi[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\theta)]$, where μ are the state probabilities, q is the action-value function, and π is the policy. This can further be written as $\mathbb{E}_\pi[G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}]$, where G_t is the return from timestep t onwards and A_t is a sample from π. This yields something called the REINFORCE update $\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$. The exact derivation steps can be verified from the book "Reinforcement learning: An introduction" chapter 13 [5], but intuitively this update takes a step towards the direction of the gradient of the probability of taking the action actually taken, multiplied by the return G_t (larger return increases the total more) and divided by the probability of taking the action actually taken (lower probability increases the total more) i.e. the update is maximal with a good return and low probability, and minimal with a bad return and high probability.

The REINFORCE method described above has theoretical convergence guarantees, but as a Monte Carlo method it might suffer from high variance and therefore slow learning. A remedy for this is to use a comparison of the state value to a baseline b i.e. making the update step $\theta_{t+1} = \theta_t + \alpha(G_t - b) \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} = \theta_t + \alpha(G_t - b) ln \nabla \pi(A_t|S_t,\theta)$ without changing the expectation. If the *state-value* function $v(S_t, w)$ is approximated as well, a natural choice for b would be that. This is the exact method that was used in this project.

My initial idea was to implement an actor-critic algorithm, where the key difference to the Monte Carlo REINFORCE with baseline is that instead of using the return G_t, actor-critic methods *bootstrap* that value as well using the func-

tion approximation $v(S_t, w)$, but after several tries with example approaches, it seemed that bootstrapping the state value was not working properly, making me modify the algorithm to utilize the actual returns G_t, effectively making it implement the REINFORCE with baseline algorithm [5]. In such an episodic setting, using a Monte Carlo method is not an issue although parameter updates can only happen at the end of episodes, which might make learning slower.

Figure 1: The pong game environment

4 Implementation

The agent was implemented using mainly the *pytorch* (https://pytorch.org/) library. The biggest benefits provided by the library were abstractions for convolutional neural networks for processing the input frames from the game, and automatic differentiation of the performance measure and making the parameter updates.

The agent was to be implemented as a python class, that had the following interface:

- load_model():void - a method for loading trained model weights for the agent
- reset(): void - a method for clearing the agent's memory after each game (frame stacking for example)
- get_action(frame:numpy array):int - a method that processes the pixels of the game frame and outputs an integer action (stay,up,down)

- get_name():string - a method for returning the name of the agent

The main problem was therefore to train the agent in such a way that the actions returned by get_action would maximize the experienced rewards. Given the game environment, processing the input frames with a convolutional neural network seemed the most appropriate. The structure of the network I ended up using was inspired by this example implementation: https://github.com/rgilman33/baselines-A2C/blob/master/baselines-A2C.ipynb. The reason to use almost an identical network to someone else's was to minimize the variables that needed tweaking in order to make the agent learn. Another aspect that was adopted from that example was how weights of the neural network layers are initialized. Namely, the weights of layers that were followed by ReLU-nonlinearities were scaled by "gains" with respect to the nonlinearity in order to modify the standard deviation, that supposedly helps in avoiding vanishing gradients (per this discussion: https://discuss.pytorch.org/t/what-is-gain-value-in-nn-init-calculate-gain-nonlinearity-param-none-and-how-to-use-these-gain-number-and-why/28131).

As mentioned in multiple articles and hinted by the course staff, stacking multiple consecutive frames of the game together and using those as the observations in order to get a temporal component into the policy and state value approximation seemed essential. I used a frame stack of 2 frames that were first resized and downsampled from 3-channel 200 x 200 RGB images to 1-channel 100 x 100 grayscale images before being stacked to a 2-channel 100 x 100 array (pytorch tensor). This implementation was directly taken from an example agent provided by the course staff as this was quite a "standard" preprocessing step.

The game environment was provided with a dummy opponent called SimpleAI, against which training could be performed and performance tested. A simplified training scheme looks like this for a single game:

1. Observe the environment
2. Sample an action from the policy (also returns the log-probability of that action, a state-value approximation, and the entropy of the policy distribution), and take that action
3. Observe the reward from the environment and store it along with the action log-probability, the state value approximation, and the entropy
    - If the game is not over, go back to step 1
    - If the game is over, move on to step 4
4. Once a game is over, perform stochastic gradient updates based on the stored values
5. After the update, a new game can be started by returning to step 1

For the step 4 above, we needed the experienced rewards, action log-probabilities, the state value approximation, and the entropy of the policy distribution. Reasons for all these terms will become apparent in the below subsections describing

the total loss, but in practice all of these terms are acquired from the from the convolutional neural network mentioned earlier, where the convolutional layers were shared, followed by two parallel fully connected layers, one for the policy distribution (soft-max distribution) and one for the state-value approximation. Networks like this are sometimes called *two-headed*.

The total performance measure (worded as loss) can be described with three different terms, that I will denote as *policy loss*, *value loss*, and *entropy loss*. Each of the terms were weighted by different weights. The parameters of the model were optimized with RMSprop [2]. The weights of the different losses and the hyperparameters of RMSprop were acquired from this example https://github.com/rgilman33/baselines-A2C/blob/master/baselines-A2C.ipynb without further analysis. To avoid vanishing or exploding gradients, *gradient clipping* [1] as also used from the same example. Below, I describe each loss term in more detail.

4.1 Policy loss

The policy loss is directly given by the REINFORCE with baseline update: $\theta_{t+1} = \theta_t + \alpha(G_t - v(S_t, \theta))ln\nabla\pi(A_t|S_t, \theta)$, where $v(S_t, \theta)$ (both v and π parameterized by θ) is the state-value approximation. The returns G_t were acquired from an utility function provided by the course staff for one of the homework problems during the course. The returns were also discounted, although the equations omit discounting here. Since pytorch is only able to minimize losses, we minimize the negative quantity $-(G_t - v(S_t, \theta))ln\pi(A_t|S_t, \theta)$.

4.2 Value loss

The goal is to also make the state-value approximation $v(S_t, \theta)$ as close to the true return G_t as possible. Therefore we minimize $(G_t - v(S_t, \theta))^2$ (effectively mean squared error).

4.3 Entropy loss

Including this term was purely inspired by this article: https://www.datahubbs.com/two-headed-a2c-network-in-pytorch/. The entropy of the policy distribution is the negative expectation of the log-probability mass function of the distribution i.e. $-\mathbb{E}_\pi[log\ \pi]$. Intuitively maximizing this quantity will encourage further exploration of the agent by making action probabilites closer to each other, thus promoting stochasticity.

5 Results and performance analysis

With the implementation described in the previous section, the agent was able to learn useful behavior in terms of rewards against Simple-AI using the hyperparameters described in Figure 2.

Figure 2: Hyperparameters

Hyperparameters	
Hyperparameter	Value
Discount (γ)	0.9
RMSprop learning rate	7e-4
RMSprop epsilon	1e-5
RMSprop alpha	0.99
State value loss coefficient	0.5
Entropy loss coefficient	1e-3
Frame stack size	2

I kept track of the running game winrate of 1000 games, and the agent was able to achieve running averages of 0.8 after 60 000 games. Figure 3 shows the performance plot against Simple-AI.

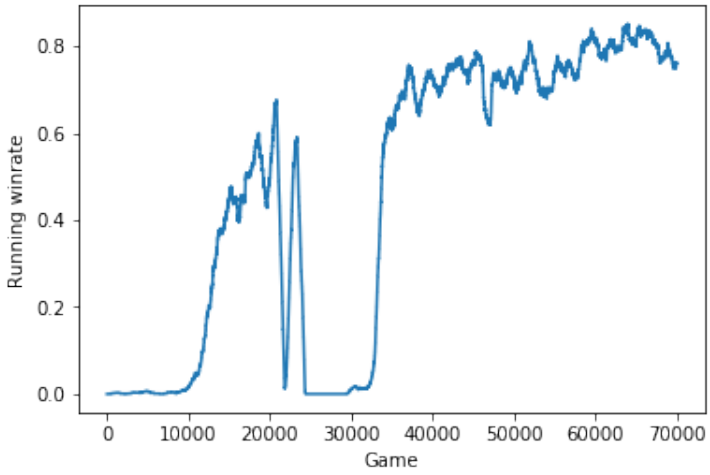

Figure 3: 1000 game winrate against Simple-AI

The average game length in timesteps was also tracked for past 1000 games (Figure 4) . I also kept track of the running average of 10 000 timesteps of the state-value (Figure 5), entropy (Figure 6), and action log-probabilities (Figure 7).

It looks like the running game lengths and the winrate go hand in hand. This is illustrated in Figure 8. A hypothesis I had was that by lowering the discount factor γ even further, the game lengths could be decreased while increasing or keeping the winrate i.e. making the agent win as quickly as possible, but anything lower than 0.9 did not seem to work properly. The state values also have significant increases/decreases with the same schedule as the winrate and game lengths.

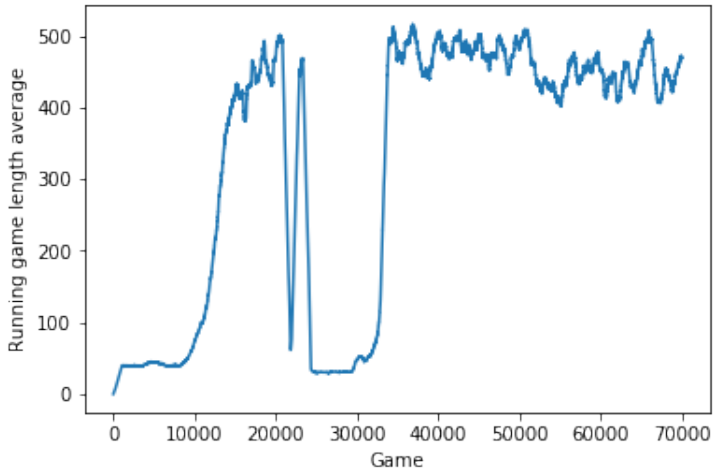

Figure 4: Past 1000 average game lengths

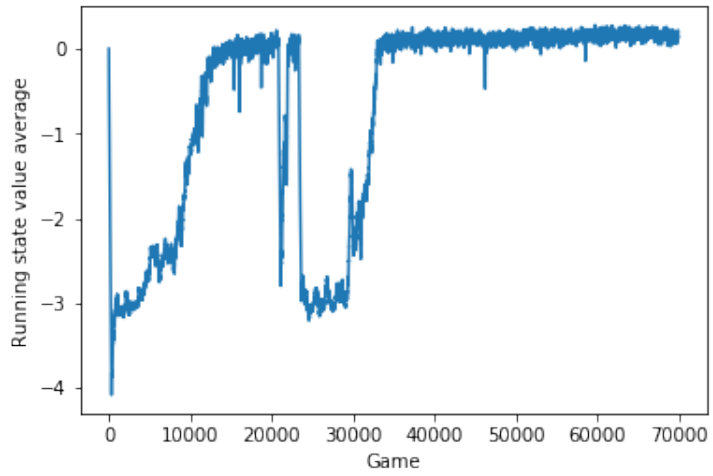

Figure 5: Running average of state values of past 10000 timesteps

Another observation regarding the relationships regarding the metrics is that the entropy and action log-probabilities are almost mirrors of each other. This implies that the actions sampled from the policy distribution were very close to the expectation/mean of the distribution.

The rate at which learning happens is very interesting. Intuitively someone might expect that learning would be highly linear/exponential, and once a local optimum is reached, the performance of the agent would only change a little. For the first 15 00 games or so this seems to be the case, after which the

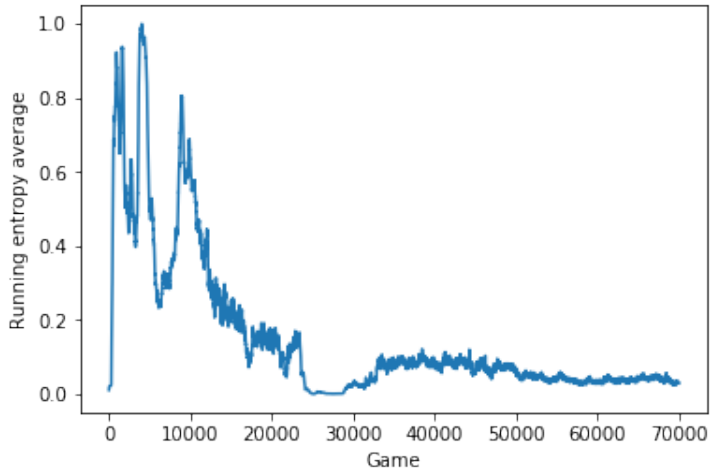

Figure 6: Running average of policy distribution entropies of past 10000 timesteps. Mirrors the log-probabilities plot.

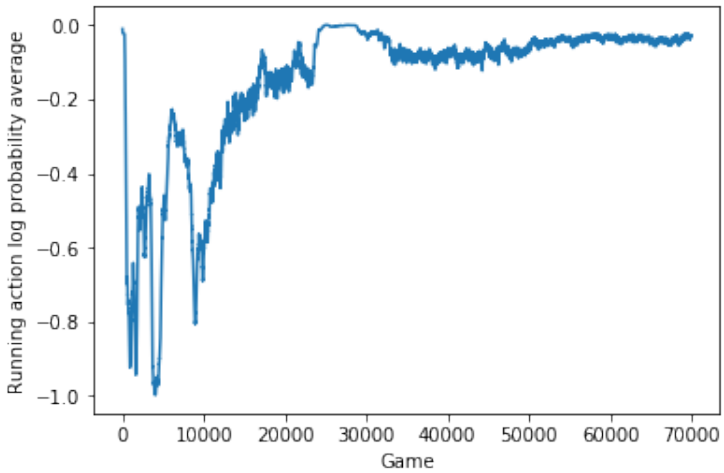

Figure 7: Running average of action log-probabilities of past 10000 timesteps. Mirrors the entropies plot.

performance deteriorates to almost 0% winrate at around 30 000 games, until it climbs back exponentially. This observation calls for the need to not only use the latest/final paramaters of the model but test for the best weigts afterward. Therefore weights of the model were saved at constant intervals and when new high winrates were experienced.

After the agent reached reasonable performance against Simple-AI, I decided

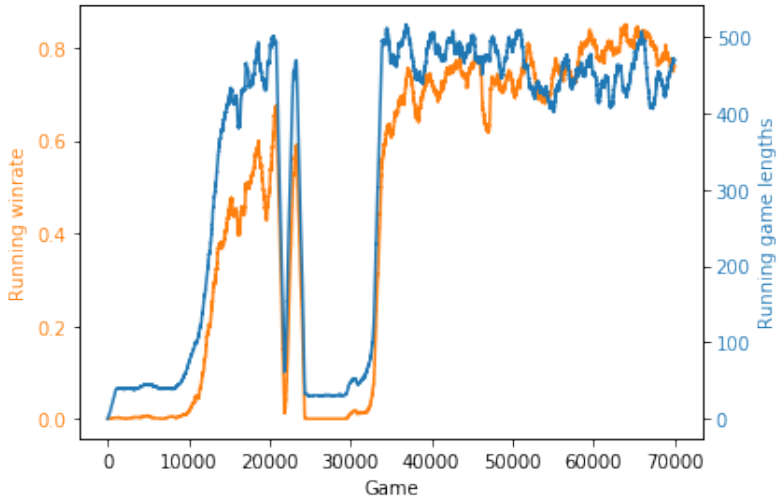

Figure 8: Comparison of winrate and game lengths

to implement a training scheme where the agent would play against Simple-AI, test agents provided by the course staff, and the agent itself (opponent is upgraded when new highest total winrate is found) in order to make it more robust and generalize to other opponents that just Simple-AI. The running winrate against all of these opponents is plotted in Figure 9. Validation runs of 100 games were run against Simple-AI every 1000 games to keep track how the original objective (beating Simple-AI) is still met. The validation run performance is plotted in Figure 10.

It seems like playing the other agents makes the agent worse against Simple-AI. This setting could have called for further simulations since the data is quite noisy when there are multiple different opponents. A sign of improvement could be seen at 80 000 games, but performance dropped again after that. Game lengths seem to again go hand in hand with the total winrate as can be seen in Figure 11.

The eventual weights submitted with the agent code were searched among all saved weight values by playing Simple-AI 1000 games and choosing the weights for which the running winrate stayed the highest. The chosen weights achieved a 0.907 winrate against Simple-AI, performance against other agents is unknown.

6 Conclusion

During this project, the task was to explore different reinforcement learning approaches for implementing an agent that is able to play the game of pong only by reading the pixels of the game environment frames. Many readily available examples and articles covering the solutions helped tremendously in finding a

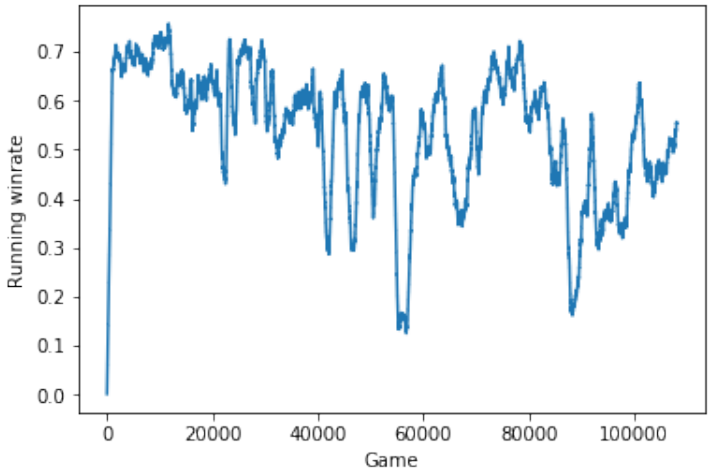

Figure 9: Winrate against multiple opponents

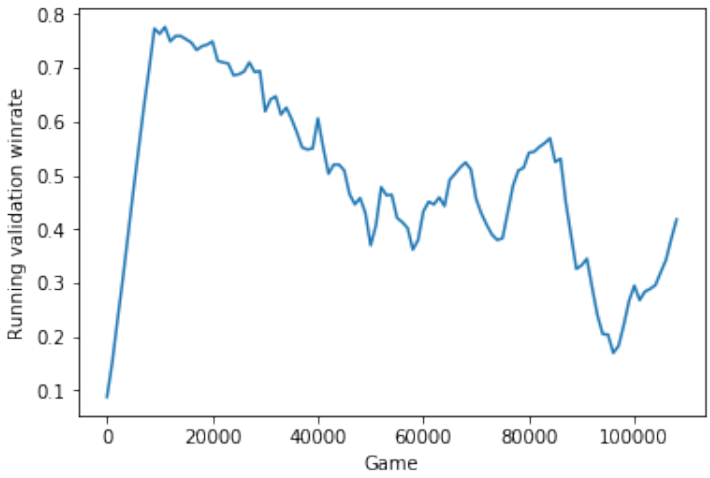

Figure 10: Validation against Simple-AI after playing other agents

solution that worked well enough against the provided example opponent.

The chosen approach for the problem was to use a policy gradient based agent, that tried to maximize the rewards that it experiences. This approach proved quite successful, and during test runs the agent was able to beat the example opponent over 90% of the games.

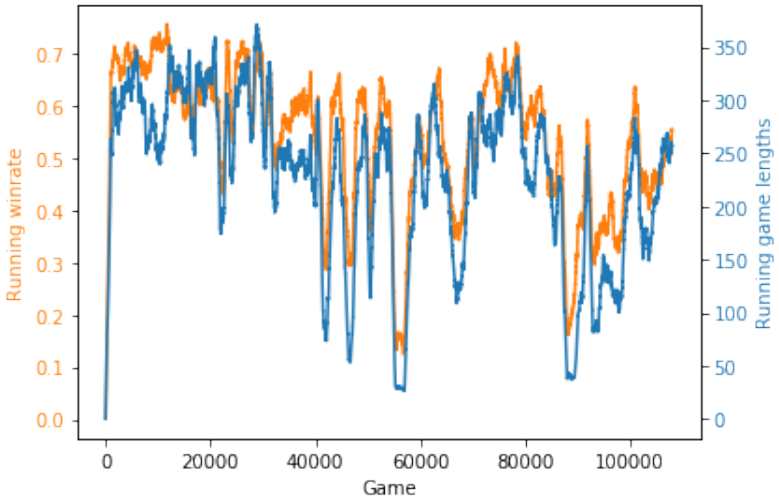

Figure 11: Comparison of winrate against multiple agents and game lengths

References

[1] Roger Grosse. Lecture 15: Exploding and vanishing gradients. *University of Toronto Computer Science*, 2017.

[2] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14:8, 2012.

[3] Andrej Karpathy. Deep Reinforcement Learning: Pong from Pixels, 2016. [Online; accessed 6-December-2019].

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[5] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[6] Wikipedia contributors. Reinforcement learning — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Reinforcement¡earningoldid = 928909260, 2019. [*Online*; *accessed*7 − *December* − 2019].