Reinforcement Learning Project: Winning Atari Pong on pixels

Irina Nikulina, Nikita Karaev

***Abstract*—In this project we have explored Atari [7] Pong game environment from OpenAI gym. To win the game the agent has to bounce the ball past the other player enough times given raw pixels of the screen as input. We have implemented two reinforcement learning algorithms: Deep Q-Network and Policy Gradient [6] introducing a fully connected neural network and a convolution neural network agents.**

After training the agents for several thousand episodes against a manually coded AI algorithm provided by OpenAI we are able to conclude basing on reward progression analysis that in case of policy gradient approach CNN agent performs significantly better that the FCN agent, but the Deep Q-Network agent outperforms them both in this task.

I. Introduction

Results of applying reinforcement learning in gaming are truly amazing. Now AI outperforms human even in such complicated game as Go [10]. In this project we have focused on more simple Pong Atari game (for which the Pong-V0 environment is provided by OpenAI gym) to understand how agent can learn from only raw pixels to become a strong opponent.

Our objective is to explore the environment, to try different reinforcement learning strategies for this game and to compare their performance. Firstly, we have implemented a basic policy gradient algorithm with CNN and FCN agents. Secondly, we worked with Deep Q-Learning algorithm that is known to be very efficient in solving Atari environments.

II. Background and Related Work

London-based company DeepMind was the first to introduce the concept of Deep Reinforcement Learning [1] that was applied to Atari Games back in 2015. They introduced a new deep Q-Network agent that in contrast to existing reinforcement learning agents was able to learn directly from high-dimensional sensory input, such as pixels, and outperformed the existing algorithms.

DQN was able to solve a variety of Atari environments at superhuman level without being modified (using the same algorithm, parameters and architecture). The implemented DQN agent was also tested on Atari Pong game, and was able to solve the environment with a score of 21:3 in favor of the agent. This article largely inspired us to implement Deep Q-Network algorithm to explore its performance in Pong environment.

As another large source of inspiration for our project an approach described in [2] was used. The article provides an overview on implementation of Stochastic Policy Gradient algorithm as well as some guidelines for data processing. It's worth to mention that there is a large variety of RL algorithms based on policy gradient approach, such as Deterministic Policy Gradient, Asynchronous Advantage Actor-Critic (A3C), Off-Policy Policy Gradient, etc. Still we decided to implement its basic version described in the above-mentioned article.

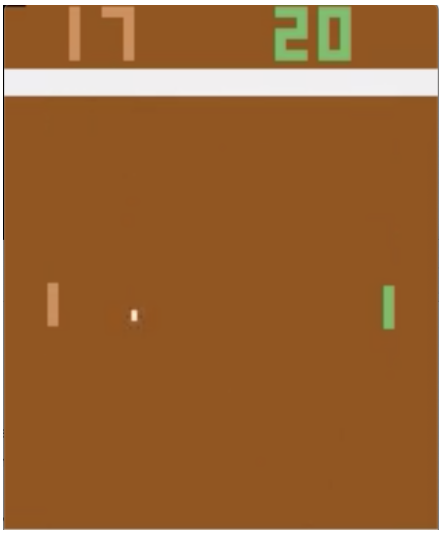

Fig. 1: Observation frame before pre-processing. Size of the frame is 210x160x3

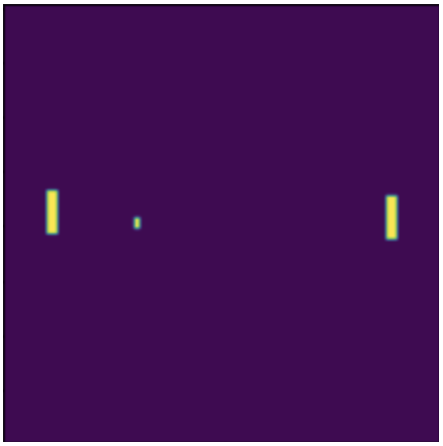

Fig. 2: Observation frame after pre-processing. Size of the frame is 80x80

III. The Environment

In this project we have worked with the existing gym environment **Pong-V0**. Let's consider its characteristics in more details :

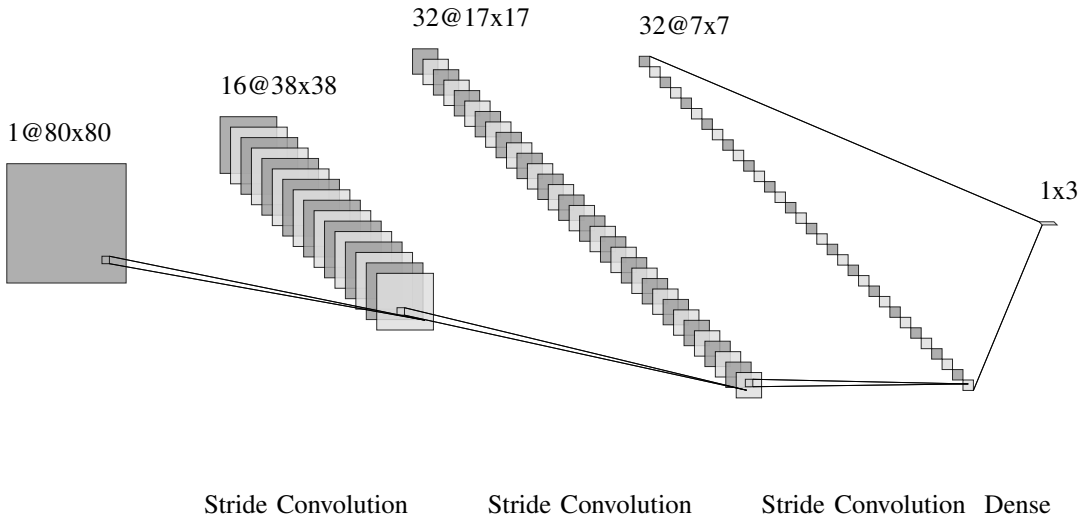

Fig. 3: Policy Gradient CNN architecture

- **State space** An observation consists of a pixel RGB image of size 210x160x3. The pixels corresponding to the current position of the ball and of the paddles as well as the current score are colored in different colors as shown on the Fig. 1.
  As the colors themselves do not include any crucial information, we convert the observation into binary color scale. In addition to this during the prepossessing (fully described in [2]) we crop the top and the bottom of the image, that are not part of the gaming space, and then we discard every second pixel. After this the size of an observation becomes 80x80 that is much simpler to manipulate (Fig. 2).
- **Action space** According to the OpenAI Gym documentation, the action space consists of 6 discrete numbers from 0 to 5. However, there are only 3 possible actions for the agent: go up, go down and stay still (no-op).
  There are two numbers corresponding to the same action because the action state is universal for many Atari games where a wider variety of actions is used. The actions not only allow agent to bounce the ball in order not to lose a point but also to change its trajectory depending on the part of paddle the ball is hit.
- **Reward function** The reward function equals to the sum of won and lost points of the player-agent for each moment of the game. The initial value of reward is zero. For every ball missed by the player the reward is reduced by one and increased by one if the agent has managed to bounce the ball past the other player. So the reward function is changing after every won or lost ball throughout the game that allows the agent to learn what actions have led to this or that change.
  The game is finished when one of the players scores 21 points, so the final value of reward function of the game takes values in the range from -21 to 21.

As we cannot fully foresee actions of the second player, the environment is stochastic and it is important to choose a proper policy. At every moment of time agent can see the entire state of the environment. In other words, the environment is fully observable.

The agent should not just somehow bounce the ball in order not to miss it, but to bounce it in a proper direction taking into account the position of the opponent to score a point.

IV. The Agent

A. Policy Gradient

In this project our goal was to build an agent that will choose an optimal action based on the current observation of the frame. To do this he follows policy gradient approach that aims at modeling and directly optimizing the policy.

We decided to compare three agents: the one whose actions are learned using a fully connected network and two others - a convolutional neural network taking 1 or 4 consecutive frames as input. In all the cases during the forward pass the network takes as an input a pre-processed frame and predicts for each of the three possible actions a probability of taking it, while the highest probability corresponds to the action that can bring the biggest reward in the future state.

As to the networks architecture - the fully connected network consists of one hidden layer of size 200. As observation is represented by an image of the current state of the game trying a CNN seemed to be an essential choice. The architecture of the network in this case is shown on the image below (Fig. 3). After receiving predictions for probability distribution for actions from a network the agent samples an action using categorical distribution and computes a gradient to perform a back propagation step. This approach allows to encourage actions that lead to higher rewards and increase their probabilities.

The idea behind using four consequent frames as an input to the CNN is that it could probably give the agent a more complete information about speed of the ball and its direction.

In our case an advantage of policy gradient is that following this approach we constantly improve the policy in contrast to Deep Q-Learning that we will consider later where we are improving estimates of Q-values of different pairs of states and

actions and the gradient guarantees a convergence. In addition to this, using policy gradient we do not need to explicitly handle exploration / exploitation trade-off (for example, by implementing ϵ-greedy policy) as it learns a stochastic policy. On the other hand, the gradient may slowly converge to a local maximum that may not coincide with the global one.

B. Deep Q-Network

The second algorithm we have decided to implement was Deep Q-Network, which is known for its efficiency in solving game environments. In contrast to table-based Q-Learning where agent gets Q-value for particular state and action from a table, Deep Q-Learning allows us to deal with an infinite space of states (just as in our case) approximating a Q-value using a convolutional neural network that is once again a good choice for visual inputs. Let's consider the main features of Deep Q-Network implementation.

As a *loss function* we use a smooth L1-loss that is quadratic for small values and linear for large values :

$$L = L_{1,smooth}(\delta) = L_{1,smooth}(target - pred)$$

$$L_{1,smooth}(\delta) = \begin{cases} \frac{1}{2}\delta^2, & \text{if } \delta < 1. \\ |\ \delta\ | - \frac{1}{2}, & \text{otherwise.} \end{cases} \quad (1)$$

where $target = R(s,a,s') + \gamma \max_a Q(s',a)$ and $pred = Q(s,a)$ is a Q-value approximated by neural network.

In contrast to supervised learning, the target is a variable term. To deal with this issue two neural networks can be used instead of one. The first neural network can be used to update parameters of the network as the agent is learning and the second one can serve to estimate the target. Both networks should have the same architecture, but the parameters of the target network would be frozen most of the time and updated with the weights of the first network every fixed number of iterations. We didn't test this approach and decided to focus on the basic DQN algorithm with only one network.

Each time agent interacts with the environment, we store experiences in a special *replay buffer*. Experience includes information about state, chosen action, new state that action has lead us to and the corresponding reward. During learning we sample randomly a batch of experiences from the buffer. It allows us to have a batch of uncorrelated experiences and to recall rare occurrences, that stabilizes and improves performance of DQN.

To deal with *exploration-exploitation trade off* we decided to use a well-known ϵ-greedy policy that allows agent to explore more random actions. More precisely, every time we may choose a random action with probability ϵ. This probability gradually decreases from ϵ_{start} to ϵ_{end} throughout the learning process, so that more experienced agent chooses less random actions.

This time we used either one or two consequent frames as an input for the network, because it could give the agent a more complete information about speed of the ball and its direction.

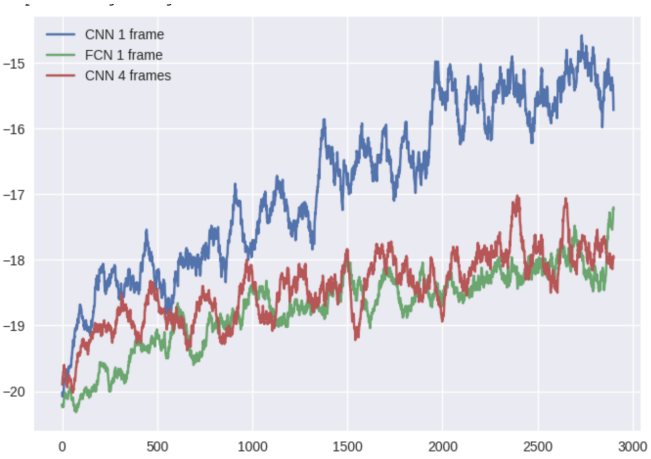

Fig. 4: Reward function graph for 3000 episodes (games) using a CNN architecture with 1-frame input (blue), with 4-frame input (red) and FCN architecture with 1-frame input (green). Averaged by the last 30 values.

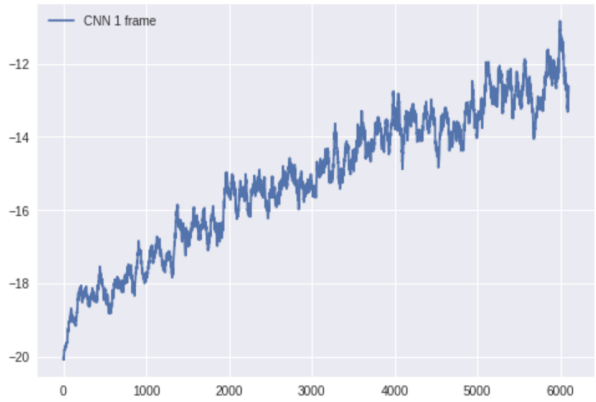

Fig. 5: Reward function values obtained during 6000-episode training of policy gradient CNN agent with 1-frame input into the network. Averaged by the last 30 values.

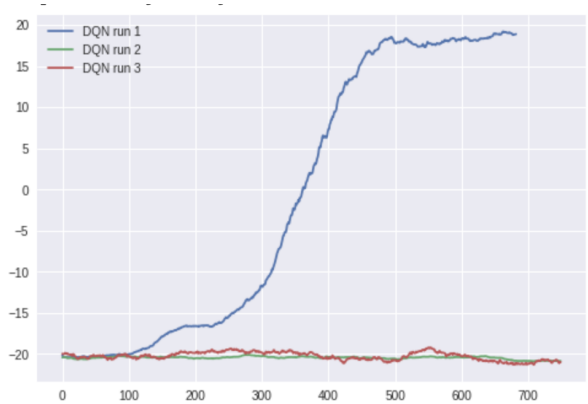

Fig. 6: Reward function values for DQN algorithm with different sets of parameters. Averaged by the last 50 values.

Fig. 7: Example of 4 preprocessed consecutive frames

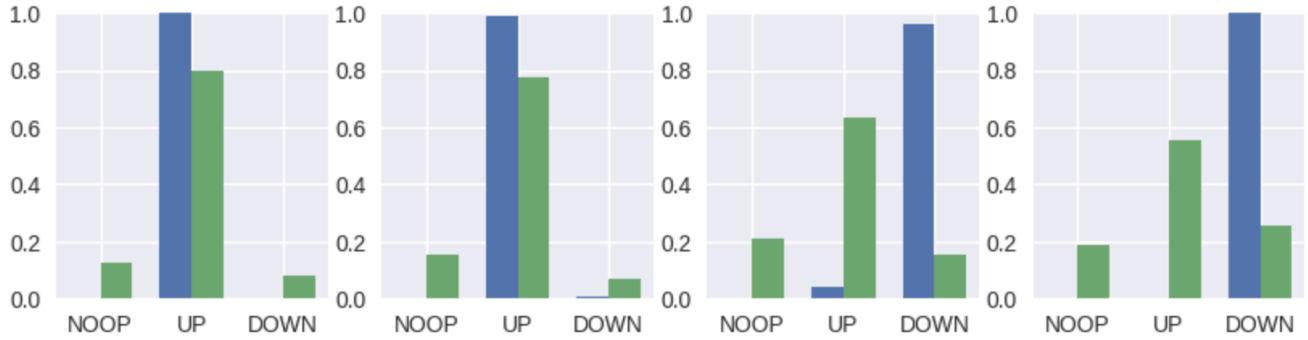

Fig. 8: Action prediction for 4 preprocessed consecutive frames by FCN agent (green) and CNN agent (blue)

V. Results and Discussion

A. Policy gradient

In this part of the work we considered agents that use convolutional and fully connected networks. In both cases the training process lasted about 2900 games (Reminder: game is finished when one of the players scores 21 points). For every game the resulting reward was recorded so it varies from -21 to 21.

As can be seen from graphs, CNN provides better performance in this task as it allows to have higher reward after shorter period of time (Fig. 4). This is an expected result as in general CNNs are more suitable for image processing and we work with states that are represented as pixel images. That's why the CNN 1-frame agent captures necessary features faster than the FCN one.

On the other hand, the CNN 4-frame agent shows roughly the same performance as the FCN agent that is also quite reasonable because it is definitely harder for a CNN to understand a sequence of images than only one image. Since feature extraction becomes more difficult, convolutional architecture doesn't have any advantage comparing to fully connected network and the agent learns with the same speed.

Let's take a look at 4 pre-processed consecutive frames (Fig. 7). For each frame we have tried to predict an action using either FCN or CNN. We can see that CNN predicts an action with a higher confidence, in this case with probability about 0.95, while the predictions of FCN are more dispersed but still the most probable action is correct and can be easily identified (Fig. 8).

In addition to this we have decided to explore outputs of every layer of CNN. In the first image of Fig. 9 we can see a pre-processed frame which is an input to the CNN. Three other pictures illustrate some outputs of 3 convolutional layers of the network. The regions colored with yellow represent zones with high activation level of neurons. It means that neurons corresponding to this part of the image with either moving paddles or approaching ball are more active that can be easily seen in the image corresponding to the filter of the third convolutional layer that will later allow fully connected layers to predict the action basing on where the ball is.

Seeing that 1-frame CNN performs better than other models, we decided to train it for another 3000 episodes so that the training process took us more than 10 hours. In the figure 5 we can see that even after 6000 episodes the agent still loses in most of the cases, even though he tends to linearly improve his performance.

We can conclude that for now this agent is not able to solve the environment. Based on the trend from the graphs, more training episodes are required to obtain an optimal policy capable of solving the environment.

B. Deep Q-Network

In the second part of the project we have implemented Deep Q-Network with CNN using PyTorch. We also trained an agent for 700 episodes and recorded final reward after each game to track down its progression. As we can see from the graph this algorithm has much better performance than Policy gradient.

Also we have tried to run the algorithm with 3 different sets of parameters that are presented in Table I. We have noticed that in our implementation of DQL the right tune of parameters

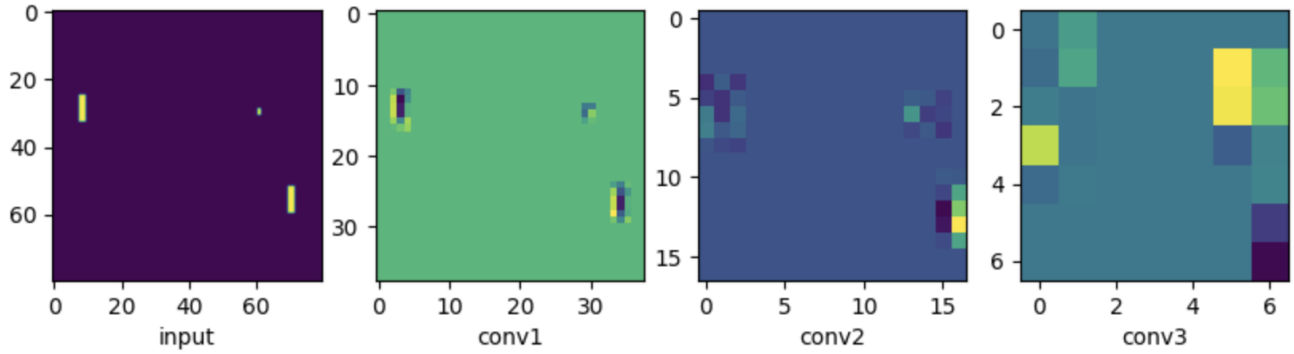

Fig. 9: From left to right: pixel input of the CNN, output of the first, second and third convolutional layers

	run 1	run 2	run 3
final reward	19.5	-20.2	-20.3
number of frames	1	1	2
action repeat	1	1	2
learning rate	10^{-5}	$2.5 * 10^{-4}$	10^{-3}
eps decay	$3 * 10^4$	10^5	$3 * 10^4$
train start frame	10^4	$5 * 10^4$	$5 * 10^4$
memory size	10^5	10^6	10^4
optimizer	Adam	RMSprop	RMSprop

TABLE I: 3 sets of parameters for 3 runs of Deep Q-Learning

is extremely important, otherwise the agent is simply not learning. We can see that already after 400 episodes the agent with right choice of parameters significantly outperforms other ones as we can see from Fig. 6.

VI. Conclusion and Future Work

In this project Atari Pong environment was explored. To start the agent has learned to play the game using stochastic policy gradient approach firstly predicting actions using a FC network and then a CNN. From the comparison of obtained results we can conclude that CNNs are more suitable for this task and provide better performance of the agent. Secondly, Deep Q-Network algorithm with CNN was implemented. It has proven to be considerably more efficient algorithm (than policy gradient) as it was able to solve the environment after 500 episodes under condition that parameters are well tuned, that is extremely important for this approach.

Policy gradient and Deep Q-Learning algorithms have a lot of improvements, for example Double Deep Q-Learning. It would be possible to implement the improved version of the algorithms as future work on the project. It would be also interesting to train the DQN agent on other Atari environment and to compare its performance with the obtained results.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236
[2] Deep Reinforcement Learning: Pong from Pixels, 2016. http://karpathy.github.io/2016/05/31/rl/
[3] Deep Q-Network with Pytorch, 2019. https://medium.com/@unnatsingh/deep-q-network-with-pytorch-d1ca6f40bfda/
[4] Reinforcement Learning (DQN) TutorialL. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html/
[5] https://www.geeksforgeeks.org/deep-q-learning/ Deep Q-Learning.
[6] As mentioned in the Lecture ”Reinforcement Learning III” *INF581 Advanced Topics in Artificial Intelligence*, 2020. https://arxiv.org/abs/1708.04782, 2017.
[7] Open AI Gym. Atari Environments. https://gym.openai.com/envs/#atari
[8] D. Mena et al. A family of admissible heuristics for A* to perform inference in probabilistic classifier chains. *Machine Learning*, vol. 106, no. 1, pp 143-169, 2017.
[9] O. Vinyals et al. StarCraft II: A New Challenge for Reinforcement Learning. https://arxiv.org/abs/1708.04782, 2017.
[10] Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016). https://doi.org/10.1038/nature16961
[11] D. Barber. Bayesian Reasoning and Machine Learning, *Cambridge University Press*, 2012.