Playing Video Games with Deep Reinforcement Learning

Simon Janežič
University of Ljubljana, Faculty of Computer and Information Science
Email: sj8495@student.uni-lj.si

***Abstract*—Traditional reinforcement learning approaches don't work well when we want to use high-dimensional sensory inputs like vision. In this paper we present deep reinforcement learning and attempt to use it to teach an agent how to play a video game by self play, getting only raw pixel values and score as an input. Concretely, we explain and test the DQN algorithm on a Atari Breakout and Pong games. We compare our performance with results that DeepMind published in a paper that presented the mentioned algorithm.**

I. Introduction

Reinforcement learning is an area of machine learning concerned with training automatic software agents that are able to determine best action for each possible state in a given environment, in order to maximize their performance [1]. Simple reward feedback is required from the environment for the agent to learn its behavior.

These methods have been used to play video games for a long time, however traditional reinforcement learning approaches can only achieve good performance in a reasonable time if given some aggregated game-specific data about the current state as input. An example of such data would be y-positions of ball and user paddle in a Pong game. If however raw state data such as pixel values are given as input, their performance falls off. The reason is huge space of possible states we get if we use pixel values as state descriptors. Traditional reinforcement learning algorithms just weren't made to deal with that big of a number of dimensions. At least not in a reasonable amount of time.

The main motivation of using raw pixel values is that we're trying to somewhat replicate the conditions that humans have when we're learning to play games. We're only seeing pixels on a screen when playing a game. It also provides great playground for solving certain traditional games. For example, computers weren't able to play Go on a professional level before deep reinforcement learning methods were introduced.

Advances in deep learning offer themselves as a solution to our high dimensional problem. We are going to use convolutional neural networks that had many successes [2] with extracting high-level features from images. Video game frames are just images so it's only natural to try using these approaches for our problem as well. Specifically we are going to implement deep Q-network algorithm that was developed by DeepMind [3]. This method combines well known Q-learning algorithm with convolutional networks. The main idea is approximation of Q-value function using neural network. There are quite a few tricks that we have to use to actually make this work. We'll use Atari Breakout and Atari Pong as our use cases.

II. Related work

TD-gammon [4] is perhaps the first success of combining neural networks with reinforcement learning. It used algorithm that's similar to Q-learning and approximated value function using multilayer perceptron. It achieved super-human performance playing backgammon. Similar approaches were attempted for other games like chess and Go but with no success. Consequently, the general belief was that TD-gammon was an isolated case.

DeepMind developed deep Q-network (DQN) [3] method that uses Q-learning and convolutional neural networks for approximating Q-value function. DQN has achieved human or super-human performance on many of the Atari games, including Breakout and Pong. After the initial DQN paper they have also developed and published multiple improvements [5]–[7] to the algorithm presented in the original paper. Since we'll be implementing and testing this method including with a couple of mentioned improvements, we will use their results and compare them with ours.

DeepMind also developed AlphaGo program [8], the first computer program to ever beat a professional Go player in a fully-sized game of Go. Unlike DQN, AlphaGo combines policy gradient methods with deep neural networks. It uses value networks to evaluate board positions and policy networks to select moves. AlphaGo agent didn't learn everything by self-play as is the case with DQN and Atari. It also used supervised learning approaches to learn from human expert games. To actually achieve good enough performance to beat a professional Go player they also had to combine the algorithm with Monte-Carlo tree search, previous state-of-the art method for playing Go.

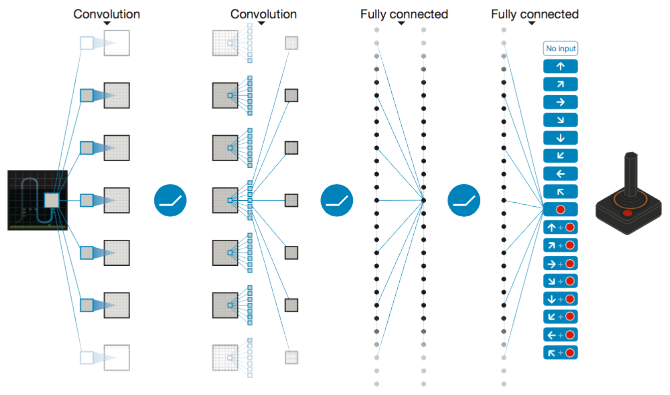

Fig. 1. Deep Q-network visualization. From left to right: Pixel input, convolutional and fully connected layers, output layer with a neuron for each possible action on a Atari controller. Source: Nature magazine

III. Methodology

A. Q-learning

With reinforcement learning we are trying to get the agent to pick the best actions inside given environment in order to maximize some reward. Let's call cumulative future reward from time t until the end of the episode (one game in our case) R_t and reward for time t r_t.

$$R_t = r_t + r_{t+1} + r_{t+2} + ... + r_n = r_t + R_{t+1}$$

Since we usually prefer getting some reward of equal value sooner rather than later, discount factor parameter is introduced. This parameter is typically called γ and determines the importance of future rewards. R_t can then be called discounted future reward.

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{n-t} r_n = r_t + \gamma R_{t+1}$$

If γ were set to 0 only immediate reward would be taken into account. If set to 1 some reward in the distant future would be just as valuable as equally sized immediate reward. Good starting value is usually around 0.9.

Key component of Q-learning algorithm [9] is a Q function that determines the quality of action a in state s. It can be written as the maximum discounted future reward if we were to preform action a in state s and then continue optimally till the end of the episode.

$$Q(s_t, a_t) = \max R_{t+1}$$

We can use Q function to determine the best action in a given state. The real problem is getting or learning this function. Similar to discounted future rewards, Q function of state s and action a can be expressed in terms of Q function of resulting state s'. The equation is called the Bellman equation. We denote it with Q_t since we'll see that it's used for calculating target Q-value.

$$Q_t(s,a) = r + \gamma \max_{a'} Q(s', a') \tag{1}$$

Basic Q-learning algorithm begins by initializing (usually with zeros) a table that contains Q-value for every possible state and every action we can take from that state. After that an agent iteratively explores the environment, at each step selecting best action based on current entries in that table and then updates appropriate entry using the following equation.

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s,a)) \tag{2}$$

We recognize the first two elements of equation after α from equation 1. This value can be interpreted as target Q-value for state s and action a based on current experience of this state and action. The newly introduced parameter α is called learning rate. It determines the magnitude of a single Q-value update. If set to 0 it wouldn't update Q-values at all. If set to 1 it would change Q values completely based on a single experience of that state and action. We want to generalize over multiple experiences of same states and actions so a good starting value is usually a low positive number (eg. 0.1).

Once we learn a behavior that brings us some reward, do we want to keep exploiting it or do we want to try and explore for potential better behavior. This is called exploration versus exploitation problem. Q-learning addresses this by making a random action every now and then. Additional ϵ parameter controls the probability of random move happening. This is called ϵ-greedy policy.

Using this procedure, Q-learning algorithm is guaranteed to converge. If that's the case why bother with deep Q-network when using pixel values as state descriptors at all? To give some intuition on why traditional Q-learning doesn't work well in this case we'll consider an artificial example. As we stated Q-learning needs to store a table of expected future rewards for all possible state and action pairs. In case of using pixels to describe states, this means storing expected future reward for every possible combination of pixels of a given game window size. If that window's size is 100x100 pixels and we're only dealing with grayscale values of pixels in 0-255 range we get 256^{10000} different states which is more than the estimated number of atoms in an observable universe. A large majority of these states will never appear in a specific game but it would still take many years for Q-learning algorithm to even visit each possible state at least once, let alone learn something from it.

B. Deep Q-network

We'll use deep Q-network to solve our problem with huge state space. Deep Q-network combines Q-learning algorithm with deep neural networks [3]. Since we're basically dealing with pictures, convolutional neural network will be used. The main idea is that instead of storing our Q-values in a table, we try approximating them with a convolutional network. Figure 1 visualizes this idea. Game frame is given to neural network as an input. Following is a series of convolutional and fully connected layers. The output on the final layer is approximated Q-value for every possible action. So performing forward pass on that neural network gives us $Q(s, a)$ value for every possible action a in a state s.

Again, the real problem is training this convolutional network to output appropriate values. As with basic Q-learning we begin by initializing that network and after that an agent iteratively explores the environment, at each step selecting the best action (based on forward pass output) or random action with probability ϵ. Lastly, at each step we also update the network to better fit our wanted behavior. We do that with well-known backpropagation. Of course for backpropagation to work we need to provide target values for each output neuron. In our case this means target Q-value for every possible action. In previous section we figured out that equation 1 can be interpreted as target Q-value based on current experience of some state and action. We can use this value as target Q-value of action that was selected in this step. We don't want to update the Q-values for actions that we didn't select, so we set target values for those neurons to the same value, the network has outputted for them. That way we get a neural network version of update step in equation 2.

To actually make this work in practice there's a couple more things that we need to do. We're only going to present the most important adjustments here. DeepMind's paper [3] contains full explanation of these approaches.

Learning on consecutive video game frames will likely get neural network stuck in a local optimum. That's because consecutive video games frames are very similar to each other and so network will overfit this small subset of possible experiences. DeepMind solved this with experience replay. Instead of learning on our last frame, at each step we sample multiple experiences uniformly from last N experiences and perform mini-batch update. Every new experience gets added to this replay memory of size N, while the oldest one gets deleted.

Another issue is that the update that increases $Q(s_t, a_t)$, often also increases $Q(s_{t+1}, a)$ for all actions, meaning it also increases Q-value targets, which can lead to divergence. To ease this problem and hence improve the stability of learning, separate neural network called target network is used. It's only used for calculating target Q-values (equation 1). We obtain it by copying our main network every C updates. This approach creates delay between the time that update to some Q-value is made and the time this update affects calculation of target Q-values. We're going to refer to the function that the target network is approximating as $\hat{Q}$.

We'd also like to talk a bit about preprocessing of video game frames. Just like in DeepMind paper we convert each video game frame to grayscale and cut upper portion of a frame to make the image square. This roughly represents the playing field of all Atari games. Images were also downscaled to 84x84 size.

Another thing that needs to be accounted for is movement in a video game. In Breakout for example we'd like to also take the direction of the ball into account not just it's position. That's why multiple consecutive (usually 4) frames are used as a single state descriptor. This new dimension is represented as a depth of input image to convolutional network (remember we're using grayscale so our depth was 1 before).

C. DQN Improvements

After the initial release of DQN paper, three improvements were published by DeepMind. We use two of those in our implementation. First one [5] is generalization of Double Q-learning that works with neural network function approximation. That makes our network double deep Q-network or DDQN. Double Q-learning eliminates possible overestimation of action values that occurs due to maximization bias. It turns out that DQN makes significant overestimations with certain Atari games. Tabular Double Q-learning requires two independently learned Q-functions. With DQN we can use the fact that we already have 2 networks: Q and $\hat{Q}$. With these two functions we can rewrite the equation 1 that calculates target Q-value using Double Q-learning.

$$Q_t(s, a) = r + \gamma \hat{Q}(s', \max_{a'} Q(s', a'))$$

DDQN has been shown to improve performance on certain Atari games.

Second improvement [6] that we're using is prioritized experience replay. We mentioned in previous section that for each DQN update we sample multiple experiences uniformly from last N experiences. This approach doesn't account for the fact that we can learn more from some experiences than others. We'd like to select important experiences more often than the others. Important experiences are defined to be as those that don't fit well with our current estimate of Q. With that definition, calculating priority for a certain experience is easy.

$$priority = |Q(s, a) - Q_t(s, a)|$$

We update priority for an experience every time it is selected. We can then sample experiences based on their priorities.

D. Implementation

We implemented DDQN with prioritized experience replay in Python. We used Keras [10] as deep learning library for our convolutional networks and implemented most of the other functionalities ourselves. Implementation can be found on Github[1]. For training we mostly used the same parameters that DeepMind used in their papers [6] but we modified replay memory size due to hardware limitations. They fitted these parameters over a small subset of Atari games. The values we used can be observed in tables I and II.

[1]https://github.com/simejanko/ddqn-ml-assignment

TABLE I
DDQN PARAMETERS

Parameter	Value
replay memory size (N)	850000
agent history length (constitutive frames per experience)	4
target network update frequency (C)	10000
discount factor (γ)	0.99
optimizer	RMSProp
learning rate (α)	0.00025
minibatch size	32
initial exploration (ϵ_{start})	1
final exploration (ϵ_{end})	0.1
final exploration frame (when should ϵ decay to ϵ_{end})	1000000
number of warmup frames	50000

TABLE II
CONVOLUTIONAL NEURAL NETWORK ARCHITECTURE

Layer	Filter size	Stride	Number of filters	Activation
conv.	8x8	4	32	ReLU
conv.	4x4	2	64	ReLU
conv.	3x3	1	64	ReLU
dense			512	ReLU
dense			$\|\|action\_space\|\|$	Linear

IV. RESULTS

To test the algorithm's performance we used Atari Breakout and Atari Pong. The implementation of these environments in OpenAI Gym [11] was used. We did have to modify when the end of the episode is denoted, to resemble behavior that DeepMind used [12]. Episode ends on first life loss instead of when life counter reaches 0.

During training phase we used ϵ-greedy policy with ϵ decaying from 1 to 0.1 over first million frames. We trained Breakout for 4 million frames, while Pong was trained for about 2 million frames in total, which amounted to about 9 days of training. To monitor our progress we used running average reward and also a more stable average action Q-value.

From figure 2 we can see that the DDQN is indeed slowly converging for Breakout, but hasn't converged yet. We would have to continue running the alghorithm to achieve it's final performance. Figure 3 shows us that the algorithm is also converging for Pong. In Pong a game is actually played until us, or the built-in AI player achieves the result of 21. Cumulative reward of -21 means we lost all games, while 21 is the optimal cumulative reward. We can see that that by the end of training run, we are consistently beating AI opponent and are closing in on the optimal reward. It may look like the algorithm has stopped converging, but the average action Q-value is still increasing at this point (figure 4), meaning the opposite may be true.

For testing phase we ran 100 episodes with our learned Q-network using ϵ-greedy policy with ϵ set to 0.05. Similiarly to Deepmind, we didn't end the episode when a single life

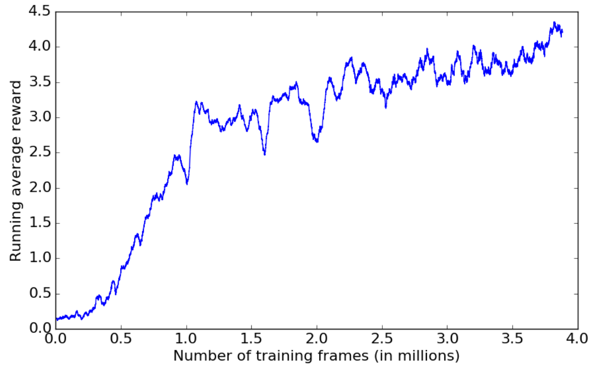

Fig. 2. Running average reward of 500 episodes of Atari Breakout training.

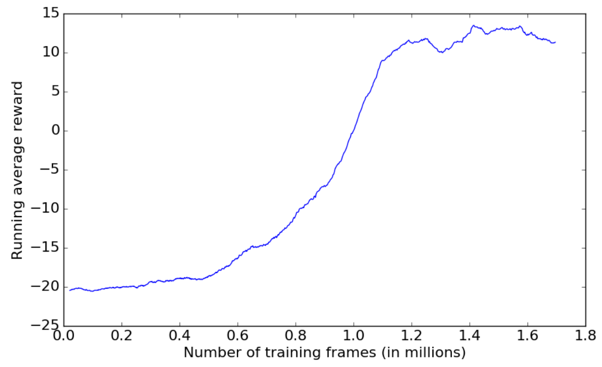

Fig. 3. Running average reward of 50 episodes of Atari Pong training.

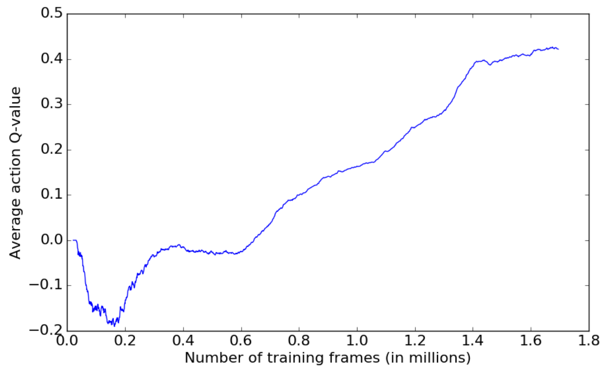

Fig. 4. Running average Q-value of 50 episodes of Atari Pong training.

TABLE III
ATARI BREAKOUT AND ATARI PONG TESTING PHASE RESULTS.

method	avg. reward (Pong)	avg. reward (Breakout)
random	-18	1.6
human	15.5	27.9
DDQN	17.39	12.7
DDQN (DeepMind)	18.9	371.6

is lost during testing phase. Average reward was used as an evaluation metric. We compared the results with random agent and human performance as measured in DeepMind's paper, as well as their DDQN with prioritized experience replay [6]. Note that they trained their DDQN for 200 million frames and based on their graphs the actual convergence for the game of Breakout was achieved at around 50 million frames which is significantly more than we could afford to run. The results are shown in table III.

We can see that our implementation is performing significantly better than a random agent in both games, but worse than DDQN as measured by DeepMind. As stated before, to get an actual comparison we'd have to run the DDQN training for 200 million frames, which we couldn't afford.

For a game of Pong we came very close to DeepMind's performance, while our Breakout agent is worse than human as measured by DeepMind. Possible explanation of why we achieved better result in a game of Pong in less training frames is game's complexity. Breakout and Pong may seem like very similar games but that's not necessarily true. Every time we reach new rows of tiles in a game of Breakout (by getting better), those tiles are disappearing for the first time since the beginning of training. Since we're learning from pixels that means training cases that are noticeably different from anything that the convolution network has seen so far. This adds an extra dimension to the problem.

V. Conclusion

We've shown that DQN with same parameters can be used to play the Atari Breakout and Atari Pong games significantly better than the random agent, but we couldn't get algorithm to run long enough to achieve optimal performance. We did get very close with Pong. To get better sense of the algorithm's performance we'd also have to test it on more (including non-Atari) games but couldn't due to hardware limitations. DeepMind reports some Atari games needing 200 million frames till convergence [6].

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[4] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[5] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR, abs/1509.06461*, 2015.

[6] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[7] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581*, 2015.

[8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[9] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[10] F. Chollet, "Keras," https://github.com/fchollet/keras, 2015.

[11] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.