Atari Breakout with Deep Q Learning

Milo Marchetti
Department of Computer Engineering
University of Bologna
milo.marchetti@studio.unibo.it

Abstract

The purpose of this project is to implement an autonomous agent capable of playing Atari Breakout through Deep Reinforcement Learning. The algorithms chosen to build the agent are Deep Q-Learning and its improvements, which are Double DQN, Duelling DQN and Double DQN but with prioritized experience replay buffer. Finally a performance comparison of these different approaches is provided. However, is not easy to define which is the best approach because of the complexity of the environment and the massive amount of computational resources needed to execute a *state of the art* training session.

1 Introduction

1.1 Q-learning

Q-learning is one of the early breakthroughs in reinforcement learning, it is an off-policy TD control algorithm defined by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)$$

In this way, the learned action-value function Q, directly approximates q_* (the optimal action-value function) independent of the policy being followed [1].

1.2 Deep Q-learning

Deep Q-learning is the evolution of Q-learning introduced to solve the limitation of this latter. Even though Q-learning is a powerful algorithm it relies on tables to keep the action-value entries (these methods are usually referred to as *tabular methods*). This is a big limitation when the algorithm is used to solve complex problems with a high number of states. Deep Q-learning menages to deal with complex environments combining Q-learning with a deep multi-layer network, used as function approximator for the action-value function. This algorithm is called DQN (Deep Q-network).

1.3 Experience Replay

Experience replay is an improvement made by the authors of DQN in order to deal with the convergence problem of Q-learning. In fact, in Q-learning the updates are highly correlated and this leads to slow the convergence time. To avoid this problem a replay memory to collect experiences is introduced. Each time the agent takes an action A_t in a state S_t and obtains a reward R_{t+1} the environment changes into S_{t+1}. All those information are used to fill the replay memory with tuples characterized by the following structure $(S_t, A_t, R_{t+1}, S_{t+1})$ .The random selection of uncorrelated experiences from the replay memory reduces the variance of the update. This allows Q-learning to converge more quickly.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

2 DQN algorithms

2.1 DQN

DQN algorithm deploys neural networks to approximate the optimal action value function $Q^*(s, a)$. The replay memory is filled with agent's experiences $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ which are stored at each time-step t in a data set $D_t = \{e_1, \ldots, e_t\}$. During learning, Q-learning updates are applied on samples of experiences drawn uniformly at random from the pool of stored samples. To improve the stability of this method with neural networks a good practise is to use a separate network for generating the targets y_j in the Q-learning update. More precisely, every C updates the network Q is cloned to obtain a target network $\hat{Q}$ and use $\hat{Q}$ for generating the Q-learning y_j for the following C updates to Q [2].

2.2 Double DQN

The popular Q-learning algorithm is known to overestimate action values under certain conditions. The max operator in DQN uses the same values both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates. To prevent this, we can decouple the selection from the evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. The greedy policy is evaluated according to the online network, but the target network is used to estimate its value [4].

2.3 Duelling DQN

Duelling DQN is inspired by Double DQN but the neural network structure is modified in order to obtain better results. The architecture explicitly separates the representation of state value and (state-dependent) action advantages. Intuitively, the duelling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way. The duelling architecture consists of two streams that represent the value and advantage functions, while sharing a common convolutional feature learning module. The two streams are combined via a special aggregating layer to produce an estimate of the state-action value function Q as follow [5]:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha) \right)$$

2.4 Double DQN with prioritized experience replay

Experience replay lets online reinforcement learning agents remember and reuse experiences from the past. This improvement of the Experience Replay introduces a way for prioritizing the experiences so as to replay important transitions more frequently, and therefore learn more efficiently. Concretely, the probability of sampling transition i is defined as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_i^\alpha}$$

where $p_i > 0$ is the priority of transition i. The exponent α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case. Prioritized replay introduces bias because it changes this distribution in an uncontrolled fashion, and therefore changes the solution that the estimates will converge to. It is possible to correct this bias using importance-sampling weights:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)$$

that fully compensates for the non-uniform probabilities $P(i)$ if $\beta = 1$. These weights can be folded into the Q-learning update by using $w_i \cdot \delta_i$ instead of δ_i[3].

Figure 1: OpenAI Gym Breakout

3 Implementation

All the algorithms have been trained for 50 epochs, where an epoch is composed of 10000 updates of the non-target network. Considering that the network is updated every 4 frames the overall number of frames of a training session is 2 million. The behaviour policy during training was ϵ-greedy with ϵ annealed linearly from 1.0 to 0.1 over the first million frames and fixed at 0.1 thereafter. The replay memory size is reduced with respect to the size indicated by the paper form 1 million to 50000. The reason for this choice is that an average PC cannot manage 1 million replay buffer without facing memory issues. Finally, to speed up the time required for the training, the project was ported in Google Colaboratory [7] (to do this is enough to change the relative path for saving the model) to exploit the GPU services. Details about the environment and the implementation of the algorithms are shown below.

3.1 Environment and Preprocessing

The environment used to test all the algorithms is provided by OpenAI Gym [6] which is a toolkit for developing and comparing reinforcement learning algorithms. The environment chosen is the old popular game Breakout Fig.1. The goal for the player is to destroy the maximum number of bricks by moving the platform, to hit the bouncing ball, performing one of the following action:

- NOOP
- FIRE
- RIGHT
- LEFT

In this environment, the observation is an RGB image of the screen, which is an array of shape (210, 160, 3). Working directly with frames which are 210×160 pixels can be demanding in terms of computation and memory requirements. To avoid these problems is crucial to apply a basic preprocessing step aimed at reducing the input dimensionality and dealing with some artefacts of the Atari 2600 emulator. At first the RGB frame are converted in greyscale and rescaled to 84×84. Then the 4 most recent preprocessed frames are stacked together to produce the input for the network. Moreover, the agent sees and selects actions on every k^{th} frame instead of every frame, and its last action is repeated on skipped frames (the suggested number of skipped frames in Breakout is 4). This technique allows the agent to play roughly k times more games without significantly increasing the runtime.

3.2 DQN and Double DQN

The basic DQN and the Double DQN algorithm are implemented following the guidelines of the papers [2] and [3]. The networks employed for these algorithms are multi-layer convolutional neural networks described in Table 1.

Table 1: DQN network

Layer (type)	Output shape	Params #
conv2d (Conv2D)	(None, 20, 20, 32)	8224
conv2d_1 (Conv2D)	(None, 9, 9, 64)	32832
conv2d_2 (Conv2D)	(None, 7, 7, 64)	36928
flatten (Flatten)	(None, 3136)	0
dense (Dense)	(None, 512)	1606144
dense_1 (Dense)	(None, 4)	2052

At the beginning the basic DQN model was trained using the optimizer described in the paper, RMSprop, with the given parameters. The result obtained using this optimizer is good but after some tries it turned out that the newer optimizer Adam, with the same learning rate, lead to a significant reduction of the training time to achieve the same result. For this reason, double DQN and the others algorithm improvements are trained using Adam instead of RMSprop.

3.3 Duelling DQN

Duelling DQN has the same structure of the Double DQN algorithm, but the neural network is changed as shown in Table 2.

Table 2: Duelling DQN network

Layer (type)	Output shape	Params #
Initial common Section		
conv2d (Conv2D)	(None, 20, 20, 32)	8224
conv2d_1 (Conv2D)	(None, 9, 9, 64)	32832
conv2d_2 (Conv2D)	(None, 7, 7, 64)	36928
flatten (Flatten)	(None, 3136)	0
Value Stream		
dense (Dense)	(None, 512)	1606144
dense_1 (Dense)	(None, 1)	513
Advantage Stream		
dense (Dense)	(None, 512)	1606144
dense_1 (Dense)	(None, 4)	2052
Final common Section		
lambda (Lambda)	(None, 4)	9

3.4 Double DQN with prioritized experience replay

Double DQN with prioritized experience replay is implemented extending the basic replay buffer in a way of saving the additional information needed to sample the experiences according to the proportional prioritization. The basic replay buffer is realized using a deque to collect the information, thus prioritized replay buffer is not very efficient because of the access complexity of this latter. To reduce the training time, the effective sample according to the prioritization is not done every update but periodically. A more efficient way for sampling from distribution is using a different data structure suggested in the paper called Sum Tree. The rest of the algorithm is equal to double DQN expect for the network's parameters update that is weighted according to the importance sampling weights.

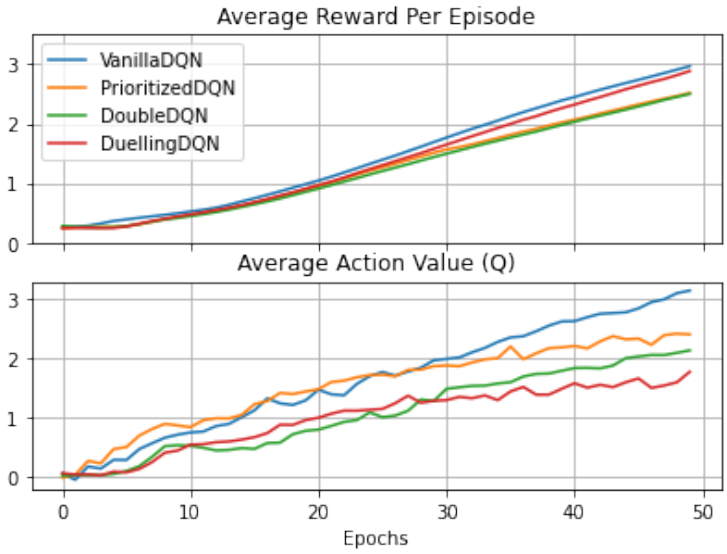

Figure 2: Training Comparison

4 Results

The results obtained by these implementations are far from the *state of the art* results reported in the papers. There are mainly two reasons, the first one is the size reduction of the experience replay buffer due to the limited resources of the machines (Google Colaboratory services and my own machine) used to train the models. The second is the training time that is significantly reduces with respect to the one suggested by the papers, that in some cases lasted for days or weeks.

4.1 Training

The data collected during the training, expressed in terms of Average Reward per Episode and Average Action Value over the training epochs, are shown in Fig.2. As we can see the data obtained are similar for each approach. Probably with longer training session it is possible to see the differences between the algorithms even in the training data.

4.2 Testing

The testing results are obtained by collecting the Average Reward per Episode of each algorithm over 30 games with a ϵ-greedy action selection policy, where ϵ is fixed to 0.05 Fig.3.

By looking at Fig.3 and spectating games played by the agents, is possible to notice that during the first life the agent manages to obtain higher score. While the following lives, before the game reset, are characterized by lower scores. This happens because hitting a higher brick cause the increment of the ball's speed and the width reduction of the platform. Considering the fact that this situation is met only after the agent has learnt enough to reach it, the key to increase the agent's score is probably just a longer training session. Anyway, good results are obtained by the Duelling DQN model with an average reward per episode around 15/18. Vanilla and Double DQN share similar results, reflecting what is written in the Double DQN paper [4]: Double DQN leads to improvements in many games, but the result obtained with Breakout are more or less the same of Vanilla DQN. Finally, similar results are obtained also by Double DQN with prioritized experience replay. The lack of a visible performance increment is due to the inefficient implementation of the replay buffer. The absence of Sum Tree as data structure significantly increased the training time thus, in order to achieve 50 epochs, the training session was split in 2 parts. The replay memory reset caused by the split and the periodic sampling from the buffer, probably influenced the algorithm performances.

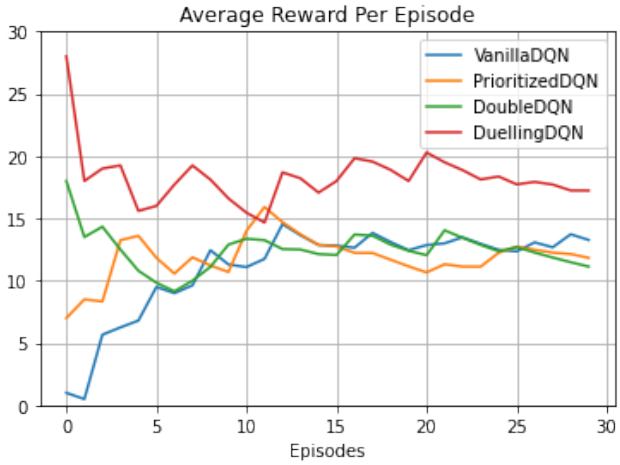

Figure 3: Evaluation Comparison

References

[1] Richard S. Sutton & Andrew G. Barto (2018) Reinforcement Learning an introduction. MIT Press, Cambridge, MA

[2] Mnih, V, Kavukcuoglu, K., Silver, D. et al. (2015) Human-level control through deep reinforcement learning. Nature 518, 529–533. https://doi.org/10.1038/nature14236

[3] Schaul, J. Quan, I. Antonoglou, and D. Silver (2015) Prioritized experience replay. arXiv preprint arXiv:1511.05952, 2015.

[4] H. Van Hasselt, A. Guez, and D. Silver (2015) Deep rein-forcement learning with double q-learning. arXiv preprintarXiv:1509.06461, 2015.

[5] Wang, Z., Schaul, T., Hessel, M., Hasselt, H. V., Lanctot, M., and Freitas, N. D. Dueling networkarchitectures for deep reinforcement learning. ArXiv, abs/1511.06581, 2016.

[6] OpenAI Gym https://gym.openai.com/

[7] Google Colaboratory https://colab.research.google.com/notebooks/intro.ipynb?utm_source=scs-index