A Report on an attenpt to play Atari game [Pong] via DQN

Pengkun.Gu.

1/23/2022

Abstract

In this report, the authors mainly focus on the process they tried to design an AI for Atari game "Pong", using deep reinforcement learning method, to be explicit, the DQN (Deep-Q-Network). The result shows that the AI constructed by this frame can easily beat the bot inside this game.

Principles

The DQN is directly follows the Q-Learning procedure, that is, to decide the policy based on an approximately optimal Q-function. An iteration process is designed for the approximation of Bellman optimality equation, that is,

$$Q^{(t+1)}(s,a) = \mathbb{E}_{r,s'}[r + \gamma \cdot \sup_{a'} Q^{(k)}(s',a')]$$

If we can parameterize the Q-function by neural network, that is, $\mathcal{Q} = \{Q(\_,\_;\theta)|\theta \in \Theta\}$, the iteration can be substituted by gradient descent procedure. To approximately find θ^+ as the successor of θ satisfying $Q(\_,\_;\theta^+) \approx Q^+(s,a) = \mathbb{E}_{r,s'}[r + \gamma \cdot \underset{a'}{sup} Q(s',a';\theta)]$, (with MSE loss) it can be done by doing gradient descent from θ and converge to θ^+, with a proper learning rate. To approximately express the "exception" term appeared in the previous expression, we should do sampling from previous transitions, here we call them "memories".

There are many subtleties in choosing the parameters, such as the memory size, batch size, etc. It can lead to a huge amount of sorrows and tears in the training period.

And like a usual reinforcement learning process, we use ϵ-greedy algorithm with ϵ decrease since the starting state, until to a later state after which it holds on a relatively low level.

Algorithm

Following [1], we state the practical DQN-algorithm as follows. (Algorithm 1 in [1])

$ALGORITHMUS$
$for\ (episode\ in\ episodes):$
$\quad initialize$
$\quad while\ not\ done:$
$\quad\quad with\ probability\ \epsilon\ select\ a\ randomly$
$\quad\quad otherwise\ select\ a = \arg\max_a Q(s, a; \theta_{target})$
$\quad\quad observe\ (s', r, done?)\ after\ stepping\ a$
$\quad\quad store\ transition\ into\ memory\ \mathcal{M}$
$\quad\quad sample\ batch\ \mathcal{B}\ from\ \mathcal{M}$
$\quad\quad\quad if\ (done?), set\ better\ value\ y = r$
$\quad\quad\quad else\ set\ better\ value\ y = r + \gamma \cdot \max_{a'} Q(s', a'; \theta_{evaluat})$
$\quad\quad do\ batch\ gradient\ descent\ for\ \theta_{evaluat}\ according\ to\ MSE\ loss$
$\quad\quad update\ \theta_{target} = \theta_{evaluat}\ after\ several\ iterations$

It is worth noting that in works [2] and [3], 2 improved algorithms, respectively called "Double-DQN" and "Dueling-DQN", are proposed successively in recent years. However according to experiments in these 2 works, they do not provided too much enhancements for our game "Pong". The Double-DQN subtly changed the target function for preventing from the over-estimation phenomenon of Q-functions. The Dueling-DQN divided the Q-function as the sum of the value function and advantage functions, and separately estimated them, for reducing the variance in sampling in the actions. A hypothesis can be proposed that the Dueling-DQN doesn't work on game "Pong" because for "Pong", actions can control different states, that is, differences of values in different states is far smaller that differences of actions.

Details

First we defined the data structure of "memories" set as a certain class `Memory`, with its basic structure as a list of tuples of form (s, a, r, s'). It has 2 basic operations, called "push" and "sample", to add new transitions to the list and to sample batches of certain size from the list. Then we constructed a CNN, with 3 convolution layers and 2 full-connected layers, using package `torch`. Then we defined the stepping process (function `step`), to implement the policy π_Q, and the optimizing process(function `optimize`), to do gradient descent by `torch`. Before stepping process, we defined a function `obs2state` to transfer the observed image with 3 color channels to the processable *tensor* in `torch`. The environment we use is "PongNoFrameskip-v4" in package `gym`.

We first trained the networks (we use 2 networks to contain all information needed in iterations, called `evaluat_net` and `target_net`) for 400 episodes and it can then draw the bot in "Pong". And after that we trained them for several hundreds episodes and it finally completely beat the bot.

Fails

Several failed attempts (hundreds lines of codes) were contained in the `junk.py`.

One attempt was to do image pre-processing and stacking manually before inputting to the network. It failed because it ran too slow. The same curse of tardiness occurred on another attempt trying to stack several images together to input to the network.

And there was an excellent attempt to do the single-frame imputing, it nearly succeeded, for its fast processing speed, but it seemed converged to another bad policy. Maybe that's because of the network structure is improper or parameters are badly tuned, or something.

And there were some ideas ignited in the procedure of coding. For example, one can restrict the frequency of transitions being stored into memory, except those next to the terminal states. It was implemented in some preceding failed attempts. But because of the limited time before handing on this project, it is not implemented in this `main.py`.

References

[1]. Mnih V , Kavukcuoglu K , Silver D , et al. Playing Atari with Deep Reinforcement Learning[J]. Computer Science, 2013.

[2]. Van Hasselt, H., Guez, A. and Silver, D. 2016. Deep Reinforcement Learning with Double Q-Learning. Proceedings of the AAAI Conference on Artificial Intelligence. 30, 1 (Mar. 2016).

[3]. Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. & Freitas, N.. (2016). Dueling Network Architectures for Deep Reinforcement Learning. Proceedings of The 33rd International Conference on Machine Learning, in Proceedings of Machine Learning Research 48:1995-2003