Project 2

GIX 李心成 2016211089
E.E. Dept. 郭开元 2015310502

Abstract

The theory of Reinforcement Learning (RL) is proven to be a good candidate to train an agent to interact with a certain environment. Previous work usually requires that the environment model is clearly built, which is not practical in applications, where sensor input cannot express the environment well. We choose the paper "Human-level control through deep reinforcement learning" [1], where this problem is addressed. Experiments on Atari 2600 games show that the proposed Deep Reinforcement Learning method achieves human level control on 29 of the 49 games. The experiments we do according to this paper gives out similar results in the finished training epochs.

1. Introduction

Reinforcement learning is proposed to solve the problem that how an agent optimizes its behavior within a certain environment. Usually, we model this problem with the following variables and functions:

- **State** s is used to represent the state of the environment.
- **Action** a denotes the action the agents take.
- **Reward** r denotes the target the agents is to maximize in the experiment.
- **Policy** $\pi(s)$ receives the current environment state s as input and returns the action that the agent should take.

RL is to solve the policy π given the environment and reward at each time step to maximize the final reward.

One of the algorithm to solve this problem is Q-learning, whose idea is also adopted in this paper. In Q-learning, we try to model the value of an action a given the environment state s, which is a function $Q(s, a)$ and the policy π adopts the action with the maximum value at each time step. In traditional Q-learning, this function is first initialized as a table for each (s, a) pair. Each entry in the table is updated iteratively during training.

For enumerable number of states, it is possible to use the above strategy. But for more complex situations like games or for human being, we can only get a high dimensional visual input as s. In these cases, it is impractical to enumerate all the possible states. This limits the the application domain of RL.

In this paper, a deep neural network is used to model the value function $Q(s, a)$. Recently, neural network, especially convolutional neural network (CNN) is showing great performance in image processing over traditional algorithms. Researches in neural science also show that the hierarchical structure of neural network is similar to what our brain uses to processes signal. The main challenge in using deep neural network in RL is the way to train the network. Directly using traditional RL framework leads to difficulties in training deep neural network for several reasons. First, there is high correlations in a sequence of observations. Second, update to Q affects the policy and thus affects data distribution. Third, there is correlation between the action value Q and the target value $r + \gamma Q(s', a')$.

To address these problems, this paper adopts two ideas, which are also the main contributions:

- An experience replay method is used to randomizes the data to reduce correlation between observations, hence reinforces the long term sight of the agent.
- An iterative update strategy is used to reduce the correlation between action value and target value.

The experimental result in the paper shows that the network can achieve human level control on 29 of 49 of the Atari 2600 games. We adopt the network structure and replay the experiment on two of the tasks, Breakout and Space Invaders, which gives out similar results on the already finished training epochs. While there is still difference between our result and that reported in the paper.

2. Methods

The main contribution of this paper is to solve the training problems when using deep neural networks as the Q function for RL. Two strategies are used in the training process: memory replay and iterative update of target Q function.

2.1. Memory Replay

In traditional Q learning, the optimization process of Q function is done through a process of interacting with the environment. This leads to high correlations between adjacent data used to update Q. The memory replay strategy stores a set of experience (s_t, a_t, r_t, s_{t+1}) as the memory, which denotes that with environment s_t at time t, the agent took action a_t, gets a reward r_t and transfers the environment to s_{t+1}. Each time to train the network, an experience is randomly selected from the memory to apply back propagation on the network.

Note that, to accelerate the speed of training, the experiences in memory are collected during training but not before training. This means the correlation of data is high at the beginning of training. But in the long run, the correlation will be small.

2.2. Iterative Update of Target Function

In general, the Q function measures the value of each action at a certain state s. The loss function can be expressed as:

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

If this loss is used to train Q it self, there will be correlation between the target and the network output. In this paper, a separate network $\hat{Q}$ is used to calculate the target. During the training process, the network for Q is updated each time. The parameters of Q is copied to $\hat{Q}$ every C cycles. In this way, $\hat{Q}$ converges to Q if the training of Q converges. But in each time of training, $\hat{Q}$ is different from Q and thus the correlation between target and network output becomes lower.

2.3. Training Flow

The overall training flow proposed in this paper is shown in Figure 1. Q and $\hat{Q}$ network is randomly initialized first. In each training episode, a new game is started with state s_0. Within each cycle in an episode, first, an action a_t is selected based on the function Q. This is done in an ϵ-greedy policy to enable the agent to try other possible actions. Then r_t and s_{t+1} is got from the game and (s_t, a_t, r_t, s_{t+1}) is stored to the memory. Next, s_{t+1} is used as the input to $\hat{Q}$ and the loss L is got by:

$$L = r_t + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

This loss is used to back propagate through Q to update the weights of Q. Every C cycles, the weights of Q is copied to Q'. An episode ends if the number of cycles reaches an upper bound or the game ends.

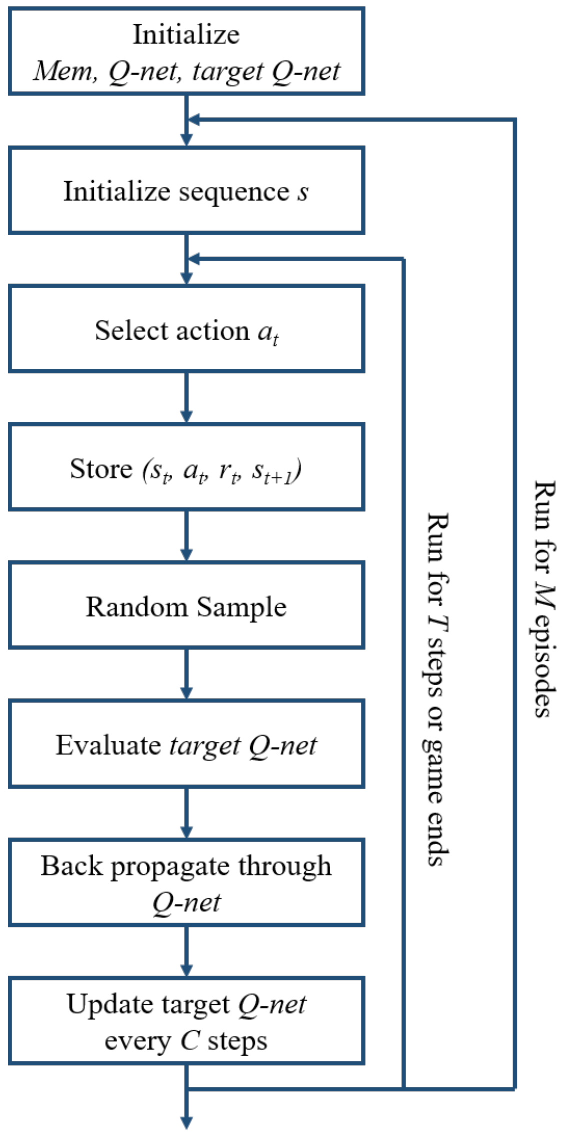


Figure 1. Training flow graph

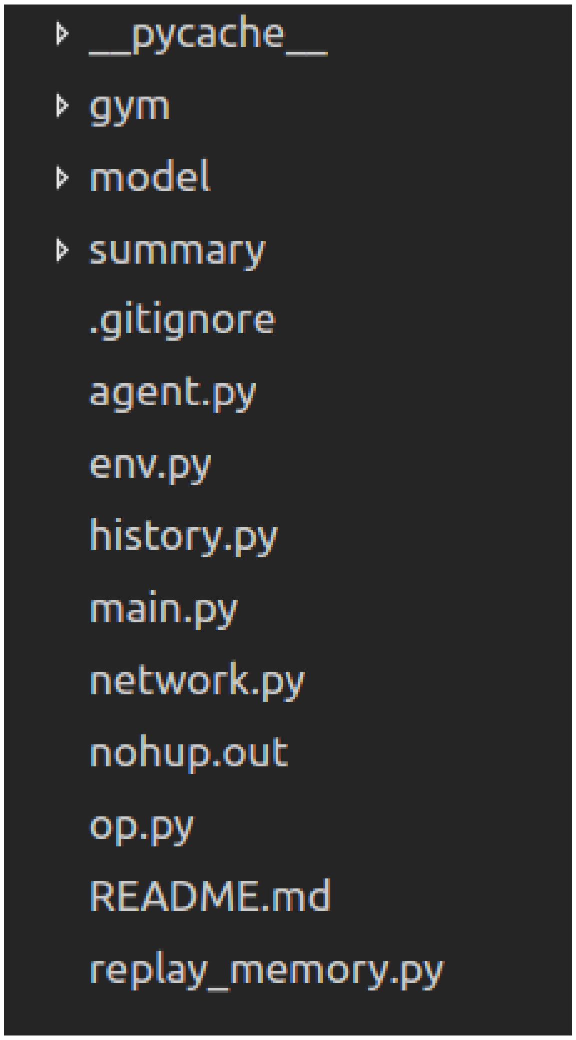
3. Implementation

We use Tensorflow 0.12 [1] and OpenAI Gym [2] framework for our implementation. Tensorflow is an open source software library for numerical computation using data flow graphs, while OpenAI Gym is an open source interface to reinforcement learning tasks, including the Atari 2600 games.

Figure 2 is the code architecture of our implementation. main.py contains the hyper parameters and some running options of our code.

agent.py implements how to train the agent and how the

[1]Tensorflow `https://www.tensorflow.org/`
[2]OpenAI Gym `https://gym.openai.com/`



```
▷ __pycache__
▷ gym
▷ model
▷ summary
  .gitignore
  agent.py
  env.py
  history.py
  main.py
  network.py
  nohup.out
  op.py
  README.md
  replay_memory.py
```

Figure 2. code architecture

agent play the Atari 2600 games using Deep Q-Network learning algorithm. It uses other modules.

env.py module is a wrapper for the OpenAI Gym Atari 2600 game environment, and takes the preprocessing works described in the paper.

history.py module implements the function ϕ that stack m most recent frames as input of the Q-function.

network.py module implements the Q function, which is a CNN network. It uses some op from op.py module such as convolution op, liner op and clip error op, which can let the network more stable according to the paper.

relay_memory.py module implements the experience replay described in paper. The paper give a memory size of 1,000,000, which is really huge, consider of the complex structure of one memory cell. To make it more space efficiency so that can fits to our servers, we did some optimization. Original structure of memory cell

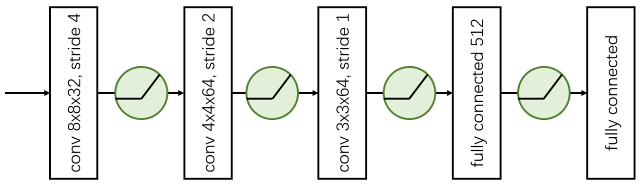


Figure 3. Structure of the CNN used for Q function

is $(\phi_t, a_t, r_t, \phi_{t+1})$, in which ϕ is the m most recently frames $[s_{t-m+1}, s_{t-m+2}, ..., s_{t-1}, s_t]$. But we can only store (a_t, r_t, s_{t+1}) as a memory cell without the redundancy and restore the whole $(\phi_t, a_t, r_t, \phi_{t+1})$ from most recent frames we saved. In that way we lower the memory demand to $\frac{1}{2m}$ of original one.

Change the flag parameters in main.py and run

```
python main.py
```

will start training or playing. Run

```
tensorboard --logdir="./summary"
```

can see the interactivity graph in your browser.

4. Experiments

4.1. Hyper Parameters and Tricks in Training

Network structure is fixed for different games except for the last layer because the size of action set varies. The network receives 84×84 4 channel images as input, where the 4 channels are consecutive 4 frames. The network structure is shown in Figure 3.

The Y channel of every frame of the game is extracted to train the network. The reward at each time step is clipped to 1 or -1 or 0 to normalize the error for different games. Thus the learning rate is kept same. The agent works in an ϵ-greedy way where the ϵ anneals linearly from 1 to 0.1 in the first million frames and fixed at 0.1 then. 50 million frames are used for training the network. The memory size is set to a million.

4.2. Result

The experiments we do are with 2 games, Breakout and Space Invaders. Limited by the computation resource available, the training process of Breakout is in 96% process and that of Space Invaders is in 63% process. Figure 5 and Figure 4 shows the log of the training process.

It is clear from the result that for both of the games, the average q value, which is the estimated best value at each time step is increasing and the average reward in each episode is raising during the episode. This shows the validity of training. Figure 6 is from the paper which shows

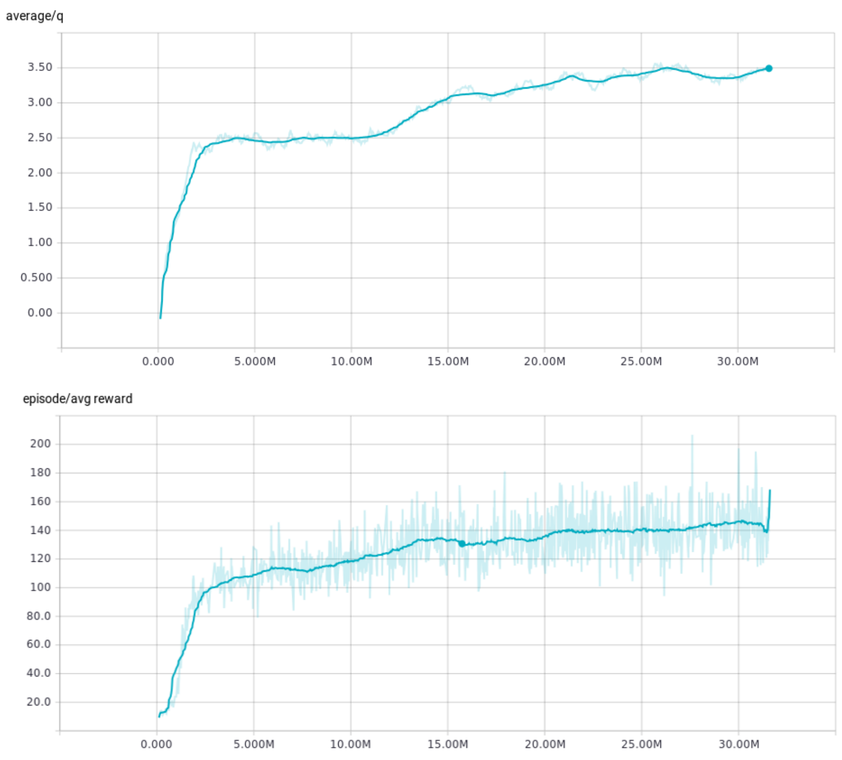


Figure 4. Space Invaders training log of the average q value and the average clipped reward gain in each episode.

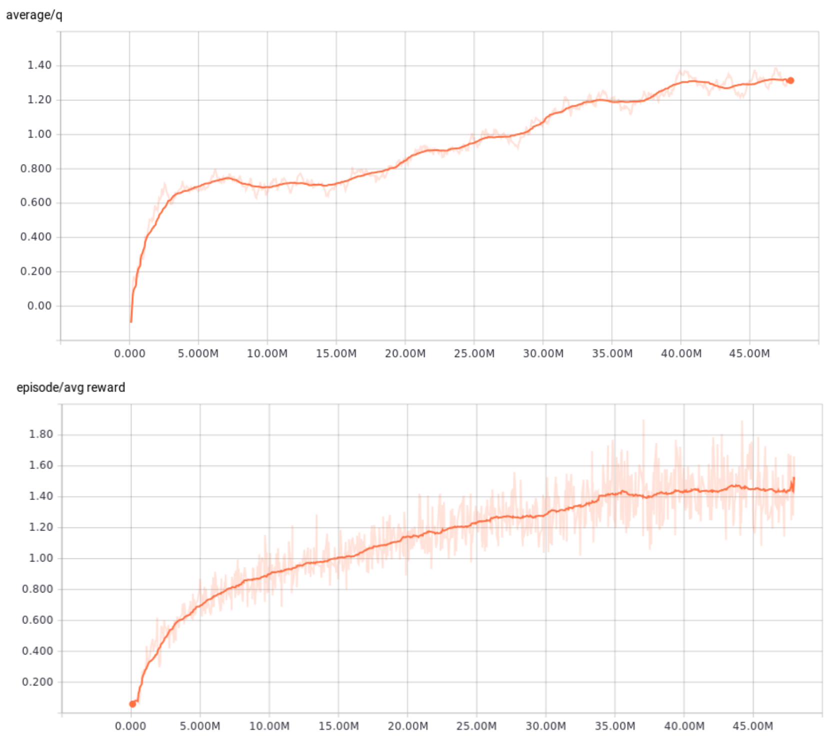
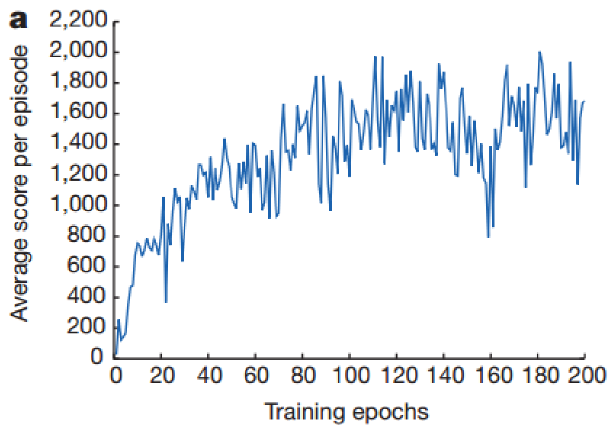


Figure 5. Breakout training log of the average q value and the average clipped reward gain in each episode.



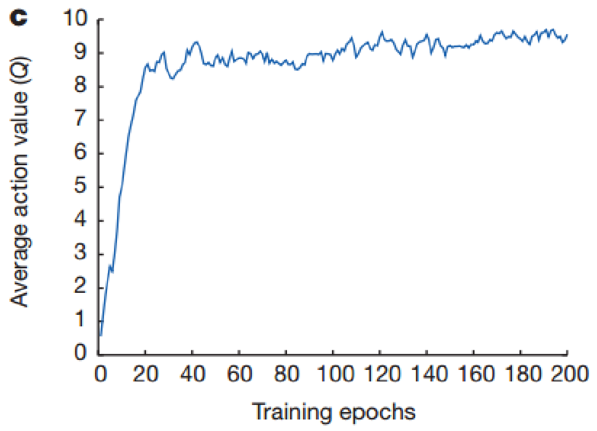


Figure 6. Space Invaders training log of the the average score and the average action value in each episode.[1]

the training result on game Space Invader. There is still difference between our result and that reported in the paper. Further work should be done after training to checkout the details.

But even with the proposed training process in the paper, the performance of the agent is still showing great fluctuation. Moreover, the fluctuation is increasing during the training process. This may be a topic in future research.

5. Discussion

While the methods and tricks described in the paper achieved a huge success, there are still some spaces for improvement. There are two simple but effective methods that can be apply to Deep Q-Network for further performance improvement.

First is Double Q-learning [2]. Original Deep Q-Network often tend to overestimate Q values of some actions through observations, which give agent a hard time to learn the best policy. The idea behind Double Q-learning is that change the way we estimate target Q value, from original

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

to

$$targetQ = r + \gamma Q(s', \operatorname{argmax} Q(s', a))$$

. As we can see, instead of taking the max over Q values, it use Q network to chose an action, and target Q network to generate the target Q value for that action. By decoupling the action choice from the target Q value generation, it reduces the observed overestimation, and leads to much better performance in some situations.

Then is Dueling DQN [3]. The idea behind Dueling is quite simple: to give Q value a more fine degree definition. That is turn Q value to the sum of reward of state V(s) and reward of action A(s).

$$Q(s, a) = V(s) + A(s)$$

Tune their parameters separately at the final layer and combine them to Q value finally. Decouple V(s) and A(s) actually give the agent ability to think of state and action separately, which makes more sense at some situations, hence leads to more robust estimation of policy and better performance.

There are still more design spaces of DQN that can be explored, such as the combination with policy network and so on. RL is an interesting area and deserve our efforts to solve it.

References

[1] 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*, 518(7540):529–533, 2015.

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

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