Report of SJTU MVIG Training Section II

Zelin Ye
Shanghai Jiao Tong University
542782758@qq.com

Abstract

Recently, Deep Reinforcement Learning (DRL) has becoming more and more popular. During this period of time, I have also learned some related knowledges and implemented some algorithms referring to some great projects and papers like [9, 7, 16].

In this report, I not only introduce some basic theories of Reinforcement Learning according to [10, 11, 15](Because it can be regarded as the foundation of DRL in some way, some algorithms are based on it), but also illustrate three classic DRL algorithms, DQN, Double DQN and Dueling network.

To the three algorithms, except for introducing their basic theories and architecture, I also show some implement details of them.

At the end of this report, I conclude testing results and analysis of three models. Then I summarize this training stage, both in knowledges and practices. Ultimately, I express my wishes to the future.

1. Introduction

After half a month studying on Deep Reinforcement Learning(DRL), I have gone through many related materials, papers, codes and projects, which do help me master the basic concepts, theories and some classical algorithms of DRL.

In addition to learning theoretical knowledges, I also put these theories into practice. In the beginning, I implement the Deep Q-learning algorithm(DQN) referring to [4, 5](Basically, I use most of their codes) and apply it to an Atari game called *breakout*. However, there exist some drawbacks in DQN algorithm, like overestimations. Therefore, I use Double DQN algorithm and Dueling network to improve or replace the original DQN algorithm. I train models in *breakout* game with three different algorithms, and some differences do appear, which are a little bit different from what I have imagined. After that, I try to change the architecture of DQN network to find something different.

The following sections are about some basic concepts of Reinforcement Learning(RL), the three algorithms, some details of my implementations and corresponding experiment results.

In this paper, I will try my best to use my own words to introduce the basic concepts instead of just copying related papers, there may be some inaccuracies or even errors in my statements, please just point them out and let me know in time.

2. Basic Theories

2.1. Reinforcement Learning

Fundamentally, RL can be considered as a kind of algorithm. What problem RL tries to solve is how an agent can finish a certain task greatly just by interacting with environment.

In this process, there are some vital concepts: action, state and reward. These concepts will be discussed later. In short, the process of RL can be described like this:

An agent takes an action, then it comes to a new state, may be better or worse. In the meantime, it will receive a reward. According to this reward, the agent will adjust its policy to approach the best one.

The so-called policy, is a mapping from states to actions, that is, the agent in a state $\boldsymbol{s}$ will use its policy to decide a certain action $\boldsymbol{a}$. Therefore, in mathematics, the policy can be described as

$$a = \pi(s)$$

Such kind of algorithm is called Reinforcement Learning (RL).

2.2. Markov Decision Process

Markov Decision Process(MDP) is the foundation of RL, we can transfer RL into a MDP model. However, MDP has its own condition. A state s_t can be considered as Markov iff the next state depends only on the current state and action.

To describe the concept in a mathematical way, there have a simple equation (*P* represents probability):

$$P\left(s_{t+1}|s_t\right) = P\left(s_{t+1}|s_t, s_{t-1},..., s_1, s_0\right)$$

As is illustrated before, RL can be considered as a MDP model. A MDP can be represented as a tuple (*S*,*A*,*P*), namely states, actions and state transition probability (the probability that the agent comes to state s_{t+1} according to current state s_t and action a_t). If we have *P*, we can calculate future state.

Besides, we also need a value to describe a state, checking whether the state is good or not. The value of state st can be defined as ***Return***:

$$G_t = R_{t+1} + \lambda\, R_{t+2} + ... = \sum_{k=1}^{\infty} \lambda^k\, R_{t+k+1}$$

R is reward, λ is discount factor, which means the recent feedbacks are more important, while the old feedbacks are less important.

Neveras, we can not calculate ***Return*** unless the process ends. So we use a value function $v(s)$ to represent potential value of a state. Value function can be considered as the expectation of rewards:

$$v(s) = \mathbb{E}\left[\, G_t \mid S_t = s\right]$$

More details about value function can refer to the next section.

2.3. Value Function

Since we have defined value function, the next step is to calculate it.

We merge the equations of value function and ***Return***, then we can get a new equation, called **Bellman Equation**:

$$v(s) = \mathbb{E}\left[\, R_{t+1} + \lambda\, v(\, S_{t+1}\,) \mid S_t = s\right]$$

From the above eqation, we should notice that the value of current state is related to the value of next state and reward. Therefore, value function can be calculated by iteration.

2.3.1 Action-Value Function

The above value function represents the value of a certain state, however, there are many optional actions in each state.

We need a value to discribe a state and a action, not just discribe a state, because we care both state and action. So we define Aciton-Value function

$$Q^{\pi}(s, a).$$

π represents the Action-Value is under the policy π.

We also use rewards to measure Action-Value. But the rewards here are the ones received after an action, while the former rewards are expection value. With these definitions, we can derive its expression:

$$Q^{\pi}(s, a) = \mathbb{E}_{\,s'}\left[r + \lambda\, Q^{\pi}(s^{'}, a^{'})\, |s, a\right]$$

In practice, we will widely use Action-Value function instead of value function.

2.3.2 Optimal Value Function

The goal of RL is to find the best policy, which can be transferred to find optimal value function.

After defining value function, with the equation in the previous section 2.3.1, we can derive the following equation:

$$Q^{*}(s, a) = \mathbb{E}_{\,s'}\left[r + \lambda \max_{a\prime} Q^{*}(s^{'}, a^{'})\, |s, a\right]$$

There are two basic algorithms to solve **Bellman Equation**, policy iteration and value iteration, here I focus on value iteration, for it may be more important to the following content.

2.3.3 Policy Iteration

Policy iteration is a way to make policy converge to the optimal one. We can use **Bellman Equation** to derive policy iteration equation:

$$v_{k+1}(s) = \sum_{a} \pi\,(a|s) \sum_{s^{'},r} p(s^{'}, r|s, a)[r + \gamma\, v_k(s^{'})]$$

Generally speaking, policy iteration has two steps. One is called **Policy Evaluation**, used to update value function. Another is **Policy Improvement**, which will generate new samples for policy evaluation by greedy policy. The relation between the two steps can refer to figure 1.

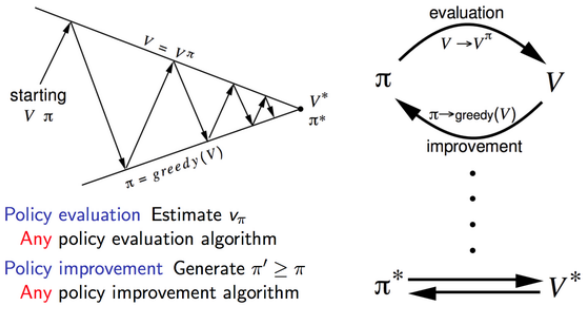

Figure 1. The Relationship Between Two Steps of Policy Iteration

In short, the whole process is using current policy to generate new samples, which will be used to evaluate the values of this policy, then the policy will be updated by such values. Repeating this process, and finally it can converge to optimal policy.

For more details, please refer to its algorithm:

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      $a \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
      If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return V and π; else go to 2

Figure 2. The Algorithm of Policy Iteration

Typically, the θ is the end point of this algorithm, because we can not do infinite iteration, so we define a signal θ to represent the end of iteration.

2.3.4 Value Iteration

We can change the original value function into the iteration type:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma\, v_k(s')]$$

The value iteration algorithm is as follow:

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
      $v \leftarrow V(s)$
      $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
   $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Figure 3. The Algorithm of Value Iteration

This algorithm is very straight-forward, but it depends on state transition function $\boldsymbol{p}$. Besides, in this form, it needs to ergodic all states, which is inefficient. Therefore, we generally use Action-Value function in practice, the corresponding equation is similar to its definition.

$$Q_{i+1}(s, a) = \mathbb{E}_{s'}\left[r + \lambda \max_{a'} Q_i(s', a')\,|s, a\right]$$

2.4. Q-Learning

The core idea of Q-Learning is from value iteration. However, different from value iteration, Q-Learning only use limited samples. Because actually, we can not ergodic all states and actions. In this case, Q-Learning has its own way to update Q value:

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

Q-Learning approaches the target Q by using a way like gradient descent. The step length depends on α, if we choice a proper α value, we can greatly reduce the impact from error. We can prove this way can make Q converge to the optimal value.

The detailed algorithm is as follow (I think this Chinese version makes sense and is easier to understand than some English versions, at least for me):

初始化$Q(s,a), \forall s \in S, a \in A(s)$,任意的数值，并且$Q(terminal-state,\cdot) = 0$
重复（对每一节episode）：
   初始化 状态S
   重复（对episode中的每一步）：
      使用某一个policy比如（$\epsilon - greedy$)根据状态S选取一个动作执行
      执行完动作后，观察reward和新的状态S'
      $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \lambda \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$
      $S \leftarrow S'$
   循环直到S终止

Figure 4. The Algorithm of Q-Learning

3. Network Architecture

3.1. Deep Q-learning(DQN)

Compared with some Deep Learning network, the network of DQN is simple. In original DQN, we use 4 consecutive 84*84 images as input, and then we have 2 convolution layers and 2 full-connected layers.

After these input images go through the original DQN network, we will receive a vector, which contains Q-values of actions.

DQN network is a typical convolution neural network structure. And we do not use pooling layers in this network, because they will decrease the network's sensibility to the location of objects in the images. Such feature may be useful in Deep Learning field, but the location information is vital to DRL. Therefore, this network give up pooling layers.

To get more details of DQN network architecture, please refer to the following figure 5 and 6.

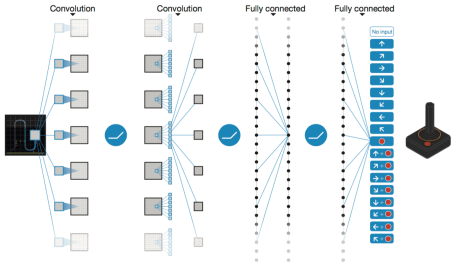

Figure 5. The Network Architecture of DQN Algorithm

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Figure 6. The Detailed Network Architecture of DQN Algorithm

3.2. Double DQN

Although the DQN may work in some DRL projects, it is also known for some drawbacks, especially overestimations. The Double Q-learning algorithm(Double DQN) can solve this problem to a certain extent.

Basically, there are no big changes in network architecture between DQN and Double DQN. They just use different data processing methods. DQN uses same values to select and evaluate an action, which is the primary cause of overestimations. In Double DQN, two value functions are learned by assigning each experience randomly to update one of the two value functions, such that there are two sets of weights, θ and $\theta^{'}$. Then the corresponding formulas in DQN should be rewritten as

$$Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{argmax}\, Q(S_{t+1}, a; \theta_t); \theta_t)$$

$$Y_t^{DoubleQ} = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{argmax}\, Q(S_{t+1}, a; \theta_t); \theta_t^{'})$$

3.3. Dueling Network

The Dueling network architecture can refer to figure 7.

The top network in the figure is a popular single stream Q-network, and the Dueling Q-network is the buttom one. The Dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation to combine them. Both networks output Q-values for each action.

The key insight behind Dueling network architecture, is that for many states, it is unnecessary to estimate the value of each action choice. In some states, it is of paramount importance to know which action to take, but in many other states, the choice of action has no repercussion on what happens.

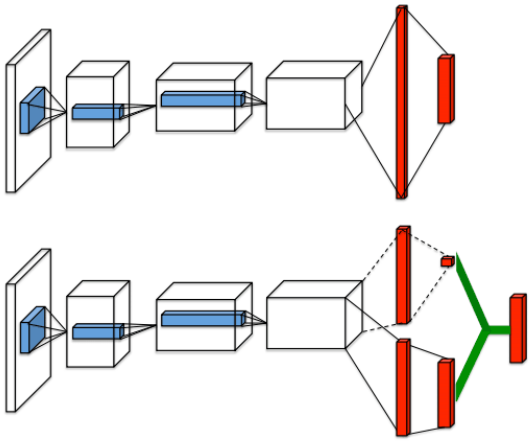

Figure 7. The Dueling Network Architecture

The lower layers of the dueling network are convolutional as in the original DQNs (refer to). However, instead of following the convolutional layers with a single sequence of fully connected layers, Dueling network uses two sequences (or streams) of fully connected layers. The streams are constructed such that they have the capability of providing separate estimates of the value and advantage functions. Finally, the two streams are combined to produce a single output Q function. As in (DQN paper), the output of the network is a set of Q values, one for each action.

4. Data Processing

Different from Deep Learning, DRL does not require a large number of fixed datasets, we just need to capture the game screen and use it as raw input. Hence, I use some interfaces to get game screen, extract states information and input them into network.

4.1. The Arcade Learning Environment(ALE)

ALE is a simple object-oriented framework that allows researchers and hobbyists to develop AI agents for Atari 2600 games. It is built on top of the Atari 2600 emulator Stella and separates the details of emulation from agent design.

ALE provides the Python interface, so I can import it in my python programs and use ALE as a lib.

With this interface, I can easily get valid actions, start a new game, check whether current game is over, and even show the game in my screen with **cv** lib.

Indeed, although there are some other interfaces, which may do better than ALE, like OpenAI Gym. I think ALE is specialized in Atari 2600, in Atari games, ALE has its own advantages.

4.2. OpenAI Gym

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Go.

It makes no assumptions about the structure of our agent, and is compatible with any numerical computation library, such as tensorFlow or Theano. It provides the Python interface, we can use it from Python code, which can be connected with famous frameworks like tensorflow.

The core Gym interface is Env, which is the unified environment interface. There is no interface for agents, that part is left to programmers. The following are some of the Env methods used frequently:

- **reset(self)**:Reset the environment's state. Returns observation.
- **step(self, action)**: Step the environment by one timestep. Returns observation, reward, done, info.
- **render(self, mode='human', close=False)**: Render one frame of the environment. The default mode will do something human friendly, such as pop up a window. Passing the close flag signals the renderer to close any such windows.

Basically, after installing Gym, I just need to write *import gym* in my Python programs, then I can make full use of the interfaces in Gym.

Besides, such interface can help me construct my model without taking concrete game into consideration. That is to say, my model can be easily applied to any game supported by Gym.

4.3. Relevant Data

Fundamentally speaking, there are two basic concepts in DRL: the environment (namely, the outside world) and the agent (namely, the algorithm we are using). The agent sends actions to the environment, and the environment replies with states and rewards (that is, a score or some other things like that).

The information we can directly get from the game are just screen images and score (if the game have it). However, we can extract following elements from the original information, which can be used in our network, either with my own codes or some interfaces like Gym or ALE.

- **States**: For *breakout* game, the states would be the whole game screen consisting of many pixel points. With the help of specific interface like Gym and **cv**, I can easily get states and transfer it into a proper format, which can be received by corresponding network.
- **Actions**: We can use corresponding interfaces to get all valid actions of a game, such as moving forward, turning left and etc, then store them in a matrix or some other data structures. Then the agent will select actions from these storages.
- **Rewards**: After the agent taking a certain action, such action will contribute to a certain result (get one more point, lose one life or make game over). According to some mapping relations, we can transfer those results into rewards.

Indeed, we can use integers or float numbers to represent rewards.

5. Experiments

5.1. Trained Models

During this period of time, I have trained three models with DQN, Double DQN and Dueling network. To Double DQN and Dueling network models, I train them about 5 millions episodes (Before announcing the report is delayed, I just train them 1 million episodes). As for DQN model, I have trained it for several days, and up to 5 millions episodes. The corresponding test results of these models can refer to table2.

5.2. Basical Stages

Based on convention, I divide the whole training process into three stages: observe, explore and train.

- **Observe**: In this stage, the agent just choose action randomly to do some observe and store corresponding information in a matrix. I do not use network in this stage.
- **Explore**: In this stage, I start to use network, and make *epsilon* (the probability that the agent takes action randomly) decrease linearly. If the episodes value are small, the network will not get enough train, which will directly affect the performance in the following stage.
- **Train**: In this stage, the agent takes some actions according to the results of previous stages and do some adjustments in its policy by the rewards.

5.3. Significant Parameters

In my models, there are some significant parameters which should be declared.

- **epsilon(ep)**: As is illustrated above, it represents the probability that the agent takes action randomly. On a regular basis, I set the start epsilon as 1 and the end epsilon as 0.01.

- **t_ep_end**: It is the time when *epsilon* reach ep_end (*= 10000), according with practice, I let the epsilon decrease linearly, after *t_ep_end* time, it will reach the end *epsilon*.

- **t_learn_start**: It is the time when the agent begins to train (*= 10000), before training, the agent needs to observe and explore, after certain episodes, it can start training.

- **gamma**: It represents the discount factor of returning Y value.

6. Results

6.1. Comparisons Between Three Models

I train three models with DQN, Double DQN and Dueling network. Due to limited time and hardware, I just set the three models' training episodes as 5 millions.

In this case, I find DQN performs best on average, while the rest two models perform as they have not been trained in the beginning 1 million episodes, then they have some improvement after 3 millions episodes.

In most time of fisrt 1 million episodes, the latter two models can not hit even one brick, sometimes they may get one or two points, but in this case, they always hit the same location to get such low scores. After 3 millions episodes, they begin to make progress stably, while the DQN model has little progress and the results fluctuate greatly.

From my point of view, perhaps the training time is too limited, so that the DQN's overestimation help its model get higher score in a short time, but it is hard for DQN model to keep the high training speed and stability, there is little progress after 5 millions episodes. Therefore, I think the rest two models will perform better than DQN as long as I have enough training time(such as ten days or 50 millions episodes).

The detailed comparsion after 5 millions episodes can be seen in the table 1.

Table 1. Testing Results of Three Algorithms

Algorithm	aveg_score	max_score	min_score
DQN	6	15	3
Double DQN	4	11	0
Dueling network	5	11	1

6.2. The Results of DQN Model

I concentrate on training DQN model, and I list its testing results alone. In the end, I have trained 5 millions episodes, the results are showed in table 2.

Table 2. Testing Results of DQN Algorithms

Steps/million	aveg_score	max_score	min_score
1	3	5	0
2	3	7	0
3	5	10	2
4	5	11	3
5	6	15	3

6.3. Results of Original Version

Due to the limited time and poor hardware, I can not complete training process of the three algorithms. Here I show the results of original codes (I modify the codes slightly to build my models) in figure 8.

Figure 8. The Result of Original Codes

In the above figure, DQN is purple line, DDQN is red and Dueling DDQN is blue, the green line is Dueling DQN, which I do not train.

7. Conclusion

In this report, I not only introduce and analyze some the basic theories and concepts of RL and DRL, but also show the test results of three DRL algorithms, DQN, Double DQN and Dueling network.

According to the test results of three models, I find that DQN model can have a palpale effect in a short training time. However, as the training proceeds, the effect will be decreasing, and the performance of DQN model is lack of stability. I think the primary cause is the DQN's overestimations, which can make it obtain rapid progress in the beginning but limit the pace and stability of training in the same time.

On the contrary, the other two models do not perform well in the beginning. In the first 1 million episodes, they hardly get even one point, even if they get several points, these points all come from the same locations. After 3

millions episodes, they begin to perform better and make progress stably. Hence, I analyze that their training effect will have a gradual or even sudden rise when they have enough training, and their final performance will definitely be better than DQN model.

In this training section, I also met multiple difficulies in theories and practices. I got warm help from mentors and teammates, here I would like to extend my sincere thanks to them.

During this period of time, I do harvest a great deal. I have not only learned the knowledges I interested in, but also trained my own models.

Although the future learning would be harder, and time will less, I believe that I have prepared for the next training section and all coming difficulites.

References

[1] Learning reinforcement learning (with code, exercises and solutions). http://www.wildml.com/2016/10/learning-reinforcement-learning/.

[2] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 06 2013.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[4] carpedm20. deep-rl-tensorflow. https://github.com/carpedm20/deep-rl-tensorflow.

[5] devsisters. Dqn-tensorflow. https://github.com/devsisters/DQN-tensorflow.

[6] d.silver. Ucl course on rl. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html.

[7] D. S. Hado van Hasselt, Arthur Guez. Deep reinforcement learning with double q-learning, 2016.

[8] Kaixhin. Atari. https://github.com/Kaixhin/Atari.

[9] V. Mnih1, K. Kavukcuoglu1, and D. S. et al. Human-level control through deep reinforcement learning, 2015.

[10] F. Sung. Dqn from entry to give up. https://zhuanlan.zhihu.com/p/21421729.

[11] F. Sung. Intelligentunit. https://zhuanlan.zhihu.com/intelligentunit.

[12] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction. https://webdocs.cs.ualberta.ca/~sutton/book/bookdraft2016sep.pdf.

[13] D. S. e. a. Volodymyr Mnih, Koray Kavukcuoglu. Playing atari with deep reinforcement learning, 2013.

[14] M. M. e. a. Volodymyr Mnih, Adri Puigdomnech Badia. Asynchronous methods for deep reinforcement learning, 2016.

[15] J. Wang. Demystifying deep reinforcement learning. http://blog.sina.com.cn/s/blog_44befaf60102wh1p.html.

[16] M. H. e. a. Ziyu Wang, Tom Schaul. Dueling network architectures for deep reinforcement learning, 2015.