School of Data and Computer Science, Sun Yat-sen University

DQN Breakout

18340047 Xiaolong Guo, 18308045 Zhengyang Gu

November 28, 2020

Contents

1 Breakout

1.1 Introduction

In a Breakout game:

- A player is given a paddle that it can move horizontally
- At the beginning of each turn, a ball drops down automatically from somewhere in the screen
- The paddle can be used to bounce back the ball
- There are layers of bricks in the upper part of the screen
- The player is awarded to destroy as many bricks as possible by hitting the bricks with the bouncy ball
- The player is given 5 turns in each game

2 Explanation of the original implementation

We made some comments in the codes, which were our understanding of each part of the codes.

2.1 utils_types.py

There is no original static type checking in python, which will brought us some unpredictable errors. However, it can be acheived through the *typing* module, as shown below.

```
def __init__(self, device: TorchDevice) -> None:
  pass
```

This example specifies the type of parameters and return. However, if a variable is specified as the *Any* type, python won't perform a static type checking on it.

The function of this file is as the comments said. It's not for static type checking, but it acts as hints for human only.

```
"""
Aliases created in this module are useless for static type
    checking, instead,
they act as hints for human only
"""
```

2.2 utils_model.py

This file defines the *DQN* class which is the network used by the dqn algorithm.

- The *forward* method: The input is the state whose length is 4. And the output is the evaluation of the state calculated by the forward algorithm. This method can be called in the form of *model_var(x)* which is equavalent to *model_var.forward(x)*.
- The *init_weights* method: The function of this method is to initiate the weights of the network.

2.3 utils_model.py

The difference between *state* and *folded_state* needs to be clarified at first.

- The *state* is a variable of length 4, while the *folded_state* is a variable of length 5.
- The *state* denotes s single state, while the *folded_state* represents the folded form of two continuous states, of which [:4] is the current state and [1:] is the next state.

This file defines the *ReplayMemory* class whcih is the replay memory used by dqn algorithm.

- The *push* method: The input is *(folded_state, action, reward, done)* in which *done* is a boolean that denotes whether the game is over. The method stores these variables in the memory.
- The *sample* method: The input is the size of batch. And the method randomly samples a batch in which *folded_state* is restored to *state* and *next*.

2.4 utils_drl.py

This file implements the dqn algorithm through defining the *Agent* class.

- The *run* method: This method chooses an action by applying ϵ-greedy algorithm.
- The *learn* method: This method trains the policy network using nature dqn algorithm.
- The *sync* method: This method copies the weights from policy network to target network.
- The *save* method: This method saves the state dict of the policy network.

2.5 utils_env.py

This file implements the breakout's environment *MyEnv* using atari environment provided by deepmind.

- The *reset* method: This method resets and initializes the underlying gym environment. It takes *render* as the input, which is used to decide whether the environment renders images.
- The *step* method This method forwards an action to the environment and returns the newest observation, the reward, and an bool value indicating whether the episode is terminated.
- The *evaluate* method: This method uses the given agent to run the game for a few episodes and returns the average reward and the captured frames. It takes *(obs_queue, agent, num_episode, render)* as the input. The *obs_queue* is an obervation queue of length 5, which can be converted into *folded_state*. The *num_episode* specifies the number of episodes that the game will be run for.

2.6 main.py

This file is the entrance of all of the codes. It defines some hyperparameters of the model and combines all codes together.

- *MAX_STEPS*: This hyperparameter denotes the max steps the environment runs. Every step the *run* method is called once to get the action generated by the model. And the *step* method is called once and takes the action as input. And the *push* method is called once to update the memory.
- *WARM_STEPS* This hyperparameter denotes the steps that the environment runs before training.
- *POLICY_UPDATE* This hyperparameter denotes the frequency of the update of policy network. Every time the policy network updates, the *learn* method is called once.
- *TARGET_UPDATE* This hyperparameter denotes the frequency of the update of target network. Every time the target network updates, the *sync* method is called once.

- *EVALUATE_FREQ* This hyperparameter denotes the frequency of the evaluation. Every time the the evaluation is being done, the *evaluate* method is called once to get the average reward, and to get frames depending on whether *RENDER* is True. And those data will be stored in disk as the form of files.

3 research

3.1 Abstract

We tried to improve the given implementation [1] of Nature-DQN for Breakout game in four ways: 1. change the Nature-DQN to a Double-DQN 2. use the Priortized Experience Replay 3. change the model to a Dueling DQN 4. add stable reward to the network to stablize the movement. We successfully implement the four methods and their combinations, which were open-sourced on github [2]. With ablation studies, we conclude that both our improvement 3 and 4 are useful for the breakout game, while 1 and 2 are not, which is similar to the result of Rainbow DQN research [7]

3.2 Related Works

3.2.1 Double DQN

In Nature DQN, non-terminated states' s Q value can be calculated by

$$y_j = R_j + \gamma \max_{a'} Q'(\phi(S'_j), A'_j, w')$$

while in Double DQN, the value is calculated via

$$y_j = R_j + \gamma Q'(\phi(S'_j), arg \max_{a'} Q(\phi(S'_j), a, w), w')$$

.The difference between them is that the Nature DQN directly uses the max Q value of next state estimated by target network, while the Double DQN first find the best action using the policy network, and then use the target network to calculate the Q value of next state after doing that action. This helps to avoid overfitting[3].

3.2.2 Prioritized Experience Replay

Prioritized DQN [4] adopts PER(Prioritized Experience Replay) to speed up the training process. The PER introduce a kind of priority mechanism, which prefers those that have bigger TD-error since the bigger TD-error means the bigger gap between estimated Q value and goal. In addition, a sumtree needs to be implemented to store the priority to have a better perfomance.

3.2.3 Dueling DQN

Dueling DQN [6] devides the model into two parts, one of which is state function which estimates the value of states, the other of which is advantage function which estimates the value of actions.

$$Q(s, a, w, \alpha, \beta) = V(s, w, \alpha) + A(s, a, w, \beta)$$

Additionally, centralization has to be done to ensure that the sum of the Q value with the same states and differnent actions has nothing to do with the second term of the formula, but it just depends on the state function.

$$Q(s, a, w, \alpha, \beta) = V(s, w, \alpha) + (A(s, a, w, \beta) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s, a', w, \beta))$$

3.3 Our Implementation

3.3.1 Double DQN

We simply change the network form Nature DQN to Double DQN as follow.

```
values_next = self.__target(next_batch.float()).gather(
            1, self.__policy(next_batch.float()).max(1).indices.
 unsqueeze(1)).detach()

```

3.3.2 Boost the training speed

We use PER to speed up the training. In our implementation, we use an open source implementation of PER. With DDQN implemented and PER peovided, we fix our code according to the pseudocode provided by [4], which is shown in figure 1.

First, we adjust the experience in PER as the form we already have. Second, we replace the origin memory class' instantiation and method calls with PER's. Finally, we add priority update and importance sampling like line 11 to 13 in figure 1.

Thea final step is shown bellow, please refer to our code for more details.

```
    td_error = (expected - values).detach()
    memory.update_priority(rank_e_id, td_error.cpu().numpy())

    values = values.mul(w)
    expected = expected.mul(w)
    loss = F.smooth_l1_loss(values, expected)

```

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

Figure 1: Prioritized experience replay DQN pseudocode from [4]

3.3.3 Improve the performance

We use Dueling DQN to improve the performance. What we need to do is changing the network model for value function for approximation as follow, which can be visualized as figure 2

```
class DQN(nn.Module):  # duel dqn

```

```
    def __init__(self, action_dim, device):
        super(DQN, self).__init__()
        self.__conv1 = nn.Conv2d(4, 32, kernel_size=8, stride=4, bias=False)
        self.__conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2, bias=False)
        self.__conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1, bias=False)
        self.__fc1 = nn.Linear(64 * 7 * 7, 512)
        self.__fc2 = nn.Linear(512, action_dim)
        self.__fc1a = nn.Linear(64 * 7 * 7, 512)
        self.__fc2a = nn.Linear(512, 1)
        self.__device = device

    def forward(self, x):
        x = x / 255.
        x = F.relu(self.__conv1(x))
        x = F.relu(self.__conv2(x))
        x = F.relu(self.__conv3(x))
        advantagex = F.relu(self.__fc1(x.view(x.size(0), -1)))
        advantage = self.__fc2(advantagex)
        valuex = F.relu(self.__fc1a(x.view(x.size(0), -1)))
        value = self.__fc2a(valuex)
        return value + (advantage - advantage.mean(1, keepdim=True))

```

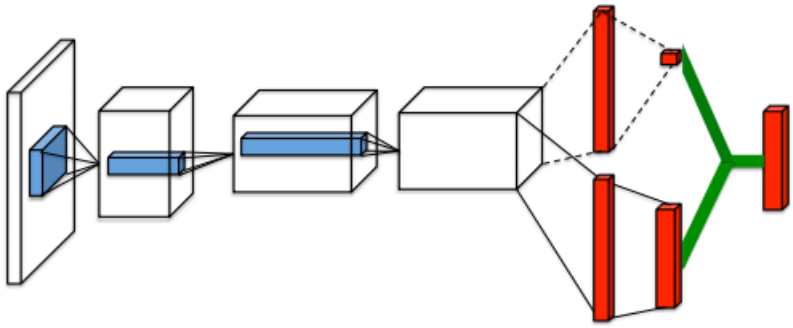

Figure 2: Dueling network architecture from [6].

3.3.4 Stablize the movement

To satble the movement of paddle in the Breakout game, we have two ideas, one is policy based and the other is model based:

- policy-based: since the policy is ϵ-greedy, we can give a much larger probability for actions that can stablized the movement, like no-op actions or actions that keep a certain moving direction.
- model-based: simply give a extra reward to actions that help the stable.

It's much easier to implement the model-based method, so we chose it and implemented it as follow.

```
reward_batch[action_batch == 0] += 0.1  # stable reward

```

The method may destroy the model-free quality of DRL, for it requires the network to konw what actions are no-ops. But it's still intuitive and has the ability to represent the cost of action in real word or energy cost for human players.

3.4 Experiment

3.4.1 Priortized Dueling Double DQN

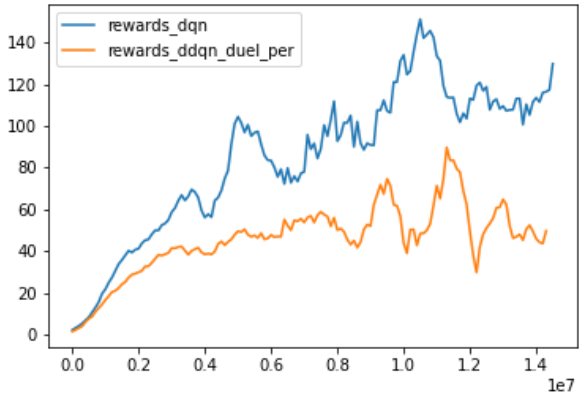

Figure 3: the reward curve of PDD-DQN and Nature DQN baseline in Breakout game. Every curve is smoothed with a moving average of 10 to improve readability

We compose all the improvement above except the stable component together as a Priortized Dueling Double DQN(PDD-DQN), and gain performance shown by figure 3. Our PDD-DQN appears to work even woser than Nature DQN in the Breakout game. So we need to do ablation studies to find out what's the reason.

3.4.2 ablation studies

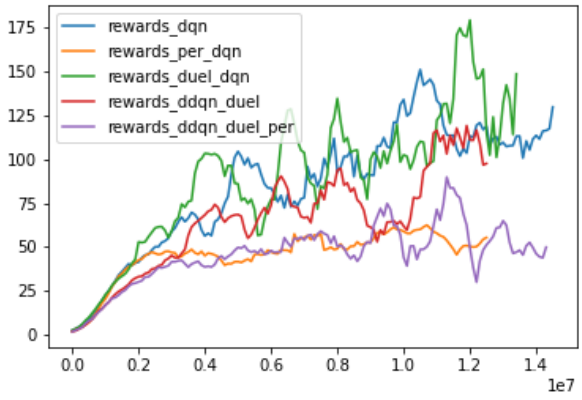

Figure 4: ablation studies on components of SPDD-DQN. With only the Dueling component, it works better than baseline. Once adding other components like Double DQN and Prioritized DQN, it's performance decrease lower than baseline. Every curve is smoothed with a moving average of 10.

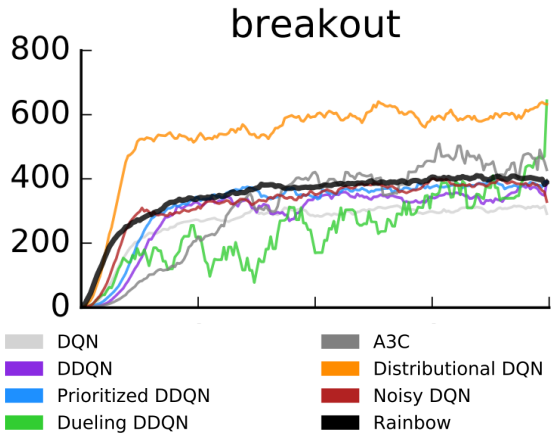

Figure 5: results on Breakout game of the Rainbow DQN research [7]. Even Rainbow DQN doesn't work much better than Nature DQN on this game. Double DQN and Prioritized DQN are shown to have lower performance than baseline before the reward reaches 200.

Figure 6: learning curve of Stable Dueling DQN, with Dueling DQN and baseline compared. With a stable component, Stable Dueling DQN keeps and improves the advantage of Dueling DQN to the baseline.

The result of ablation studies was show in figure 4. We found that Dueling DQN has higher performance than Nature DQN in Breakout game, while Double DQN and Prioritized DQN have lower.

3.4.3 compared with others

To validate our result, we compare it with Rainbow DQN research [7], whose result in Breakout game is shown in figure 5. We found that in Breakout game, what the research of Rainbow DQN gains is similar to ours, where the performance of DQN is better than Double DQN and Prioritized DQN before the reward reaches 200.

3.4.4 stable component

We plot the learning curve of Stable Dueling DQN, with baseline and Dueling DQN compared, which is shown in figure 6. We found that the stable component didn't destroy the advantage of Dueling DQN, but improve it a little.

To see how much the Stable Dueling DQN stablize the paddle, we pick the models of same epoch in it and Dueling DQN and compare the playing video generated by the models, which can be watch in attachment. We found the the stable component can stablize the movement, to a certain extent.

3.5 Conclusions

We implemented a Stable Prioritized Dueling Double DQN(SPDD-DQN) for the Breakout game and found the stable component and dueling component of the network can effectively improve the performance, while the Prioritized component and Double DQN component do not work well, which is similar to the result on Breakout game of the Rainbow DQN research [7].

Future work may include the experiment with more running epochs and the experiment on other games.

3.6 Acknowledgments

Thanks to the authors of open source code [5].
Thanks to the baseline result shared by 18340043.
Thanks to the HPC provided by SYSU.
Authorship matrix is as follow.

Member	Ideas(%)	Coding(%)	Writing(%)
Xiaolong Guo	60	60	40
Zhengyang Gu	40	40	60

References

[1] https://github.com/lukeluocn/dqn-breakout

[2] https://github.com/guzy0324/dqn-breakout

[3] van Hasselt, H.; Guez, A.; and Silver, D. 2016. Deep reinforcement learning with double Q-learning. In Proc. of AAAI, 2094–2100.

[4] Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2015. Prioritized experience replay. In Proc. of ICLR.

[5] https://github.com/Damcy/prioritized-experience-replay

[6] Wang, Z.; Schaul, T.; Hessel, M.; van Hasselt, H.; Lanctot, M.; and de Freitas, N. 2016. Dueling network architectures for deep reinforcement learning. In Proceedings of The 33rd International Conference on Machine Learning, 1995–2003.

[7] van Hasselt, H.; Hessel, M.; and Silver, D. 2017. Rainbow: Combining Improvements in Deep Reinforcement Learning.