Report for Mid-term Assignments

members: 马浩铭20337221, 刘昱名20337219

whole project is open source at https://github.com/ma-h-m/DQN-Breakout

Project description

Given a fully implemented Nature-DQN agent(which can be found here) to play the game Breakout, explain in details the function of each part of the code, and try to do some improvements, such as boosting the training speed, improving the performance, stabilizing the movement of the paddle by accomplishing some specified methods.

Authorship

Member	Ideas	Coding	Writing
马浩铭	40%	60%	50%
刘昱名	60%	40%	50%

Part 1: explain the function of each part

utils_types.py

According to the comments at the beginning, all the names below are used as hints to help the programmer to code. From this file, we can know the shapes of these types. However, it can NOT help the program to check the vars are legal or not.

utils_memory.py

This part is used as the memory pool for the agent to learn. In DQN algorithm, each stage, the agent interacts with the environment to generate the experiences. Then the experiences are teared into pieces to eliminate the association between each samples, due to one of the basic assumptions of gradient descent method, each sample must be independent to each other, and stored in the memory pool. Often, one piece of experiences includes the record of current state, action, reward, next state. When learning, the agent takes one of the pieces from the memory pool randomly, and try to learn.

In the `__init__` function, we can see that the sequence of the experiences is not distorted in the memory. The last piece shares the next state with the next piece's last state. The whole experience can be interpreted as `states 0 -- action 1 -- reward 1 -- done 1 -- states 1 -- action 2 -- reward 2 -- done 2 -- states 3 ...` `dones` represent the whole game is end or not.

The `push` function can push one piece of experiences into the memory pool. The pool store the pieces using a circular queue, which make it easier to clear the whole pool.

The `sample` function can get a batch of pieces of experiences randomly. Each piece includes the record of current state, action, reward, next state and whether the game ends at this point. In this function, the `ReplayMemory` class works as samples in other machine learning methods. The return type can refer to the `utils_type.py`.

utils_model.py

In this file, DQN net is defined. We can see how the DQN network in constructed, and how to generate the new action.

In in the `__init__` function, the DQN class inherits its super class, using the pytorch structure to easily create a neural network. It uses three convolutional layers to extract features, and two fully connected layers to generate actions basing on the features

extracted. When our group realizing the dueling DQN, we change the fully connected layers into an advantage network to evaluate each actions' advantage over others and an value network to evaluate the current states' value. We will explain it in details later.

The `forward` function shows us the forward propagation of the network. In other words, this function is to generate action when receiving the current state. The input is a picture in three channels, and each pixel has a value between 0 and 255 in each channel. So the first step is to convert them into a float between 0 and 1. Then, there is relu function after each convolutional layer to extract the features, followed by the fully connected layers to 'make decisions' according to those features. The returned value is a vector, indicating the values of each action predicted by the network. And those actions actually matches the buttons on the console.

The `init_weights` function is to initialize the weight of the parameters in the net, and is encapsulated in the pytorch's function.

utils_drl.py

In this file, the agent is defined. The DQN algorithm uses a trick called 'Fixed Targets' when learning. In a short period, the DQN net to compute the target value for the sampled tuple is fixed. It will not be upgraded immediately after the gradient is calculated each time. Instead, the fixed network will be synchronized a few times later. This trick is to stabilize the training procedure. And we can see how it is realized in this file.

The `__init__` function declares all the items in the class. Among them, the important ones includes `__eps_decay`, which is used to decline the learning rate when each epoch ends, `__policy` and `__target`, which are important for the Fixed Target trick, `__optimizer`, which can choose the learning rate and optimizer for training.

The `run` function can generate an action when given a state. At the beginning, it will directly return some random actions because the eps is too big. As time passes by, the eps decays and the `__policy` network starts to work and generate the action.

The `learn` function shows how the agent can learn. Thanks to the well implemented memory class and the help of pytorch, all we need to do in this function is to get a batch of samples from the memory pool, get the predicted values by TD learning, and use optimizer from pytorch to upgrade the parameters. And when computing the TD error, according to Fixed Target tricks, we use target network to calculate the target value, and divide the predicting value according to our policy network.

The `sync` function is to update the `__target` network. Just load all the parameters from the `__policy` network.

The `save` function can save the models' parameters. We can use the saved models to continue training some time later or use them to visualize how well the models can play the game.

utils_env.py

This file define a class to make it more convenient to manipulate the environment.

First of all, this file creates a class called `MyEnv`, which provides environments for the propose of both training and evaluating.

In the function `__init__`, we use a temporary variable called `env_raw` to make two environments. The two environments are totally the same at the start of our program, and the reason why we separate them is that only then can we prevent the evaluating process from messing up training process. Then, in order to reuse our code, a variable call `self.__env` is created, which make our code easier to maintain.

The `reset` function will reset and initialize the environment assigned to `self.__env`. Considering there are two environments exiting in the class simultaneously, we can simply regard the environment assigned to `self.__env` as the very environment to be reset. On the contrary of a common belief that we should do nothing and remain the environment not changed, this function will play 5 step without any movement. After that, an changed environment along with initial reward and observations will be returned to the caller. As we can see, there is a possibility that the game will end in these 5 steps, and in this case, we will reset again.

The `step` function will be quite confusing without explanation. It accepts an integer as action, and the integer will remain 0 if it is originally 0, otherwise it will be added by 1. After that, the integer will be passed to the activated environment which is assigned

to `self.__env`. To realize why should do that, we can simply recall the main idea of our agent, which is aimed to play various kinds of games. The usage of action is as follows.

action	effect
0	noop
1	fire
2	right
3	left

So, 1 is useless in this game, but our input will be 0,1 or 2. So, we add them by 1 to avoid the occurence of 1.

The `get_frame` is a function that renders current state. It will ask the activated environment to provide a RGB array representing current state.

Function `to_tensor` is a static method which convert a numpy array to a pytorch tensor, with size $(1,84,84)$.

Function `get_action_dim` is a static method always returns 3. This number represents the possible number of actions. As the table above, we can go a step further and confirm that action 1 is not a suitable choice for the game.

The function `get_action_meanings` will return a list `["NOOP", "RIGHT", "LEFT"]`, actually it has not been called in the whole program, and its meaning is to remind human readers. The reason why "FIRE" is missing is that action number 1 will be changed to 2 in our function `step`. So as a piece of encapsulated code, there is no need to show users the details.

The `get_eval_life` returns 5 all the time. It represents that you have 5 lives in the game.

The two function `make_state` and `make_folded_state` are quite alike. The first one will discard the oldest frame while the second one won't. The underlying reason of it is that our agent will use last 4 frames to predict the best action, so that there will be an error if we pass 5 frames as parameter. Meanwhile, on the terms of storing experience in `memory`, we need last 5 frames, and that is in which `make_folded_state` plays its role.

The `show_video` will use the photos got in our testing process to make a video, which make human easy to understand the behavior pattern of agent.

The last but not the least, `evaluate` is of vital importance in terms of assessing whether an agent is satisfying enough. The function will ask the agent to run a few episodes and return average reward. Before each life of each episode, the board will stay still for 5 steps to provide enough frames for our agent.

Till now we have discussed all the components in the original version of the code and how they are connected with each other. Now we are going to explain the improvement we made for the algorithm and why they can work.

main.py

In file main.py, we firstly define some hyperparameters. After that we instantiate the classes we defined before. Eventually, the most significant process, training, takes place.

We initial a deque to contain the returned frames of environment. And for each step:

- If last episode has done, we reset the environment and record observations.
- If enough warming-up data are collected, we will make agent step forward using policy network. Otherwise we will just simply take a random step to enlarge our memory. Although both two procedure are based on `agent.run`, the parameter `training` will control the behavior of the function. After getting a action, we can interact with the environment, and observe new state as well as receive new reward. The state, action and reward information will be stored in memory, for the purpose of further usage.

- Then, we will make our agent learn from history every 4 steps, which means that between two learning processes, 4 paths are collected. The `learn` function will extract some samples from the memory and calculate the state-action value $Q(s,a)$ using policy network. And it will also calculate the max value of next state among all the actions$max_{a\in A}{Q(s',a)}$ with target network. Next, the TD error is available to calculate. For a single sample, the label is $\gamma * max_{a\in A}{Q(s',a)} + R $. But considering we should take those terminated states into consideration, we should multiply the expected value by $1.0 - doneBatch$.
- And every 10000 steps, we will update the target network using the parameters of policy network, so that the two network will become identical. This procedure not only make our network more stable which leads to the convenience of training, but also betters the performance of target network so that it can be competent to assess the policy network. At the start of training, the accuracy of policy network is as poor as random value, and the target network is also no better than policy network. So as the label of the training process, the output of target network is also far from satisfactory in the beginning. But after several steps, the policy network will have a better performance than random values, and the target network will also perform better after updating. The improved target network will provide better label, as a consequence of which the policy network will also be trained better. We just do such process over and over again, and our two network will become better and better.
- Eventually, if `EVALUATE_FREQ` is reached, we will write our reward and parameters as well as the frames.

Part 2: Improvement we made for DQN and why it works

Dueling DQN

The net of dueling DQN is similar to DQN. After using convolution layers to extract features of the input frames, instead of using a series of fully connected layers to fit the function Q*(s,a), the dueling DQN use two different fully connected layers to fit the function V*(s) and A*(s,a). Here the Q* and V* function share the same meaning as in the DQN. Q* shows the total value when the agent is in state s and choose action a. And V* shows the total value when the agent is in state s. And A* shows when the agent is in state s, how good is action a. It is defined as `A*(s,a) = Q*(s,a) - V*(s)` The picture below illustrates the structure difference between DQN and dueling DQN.

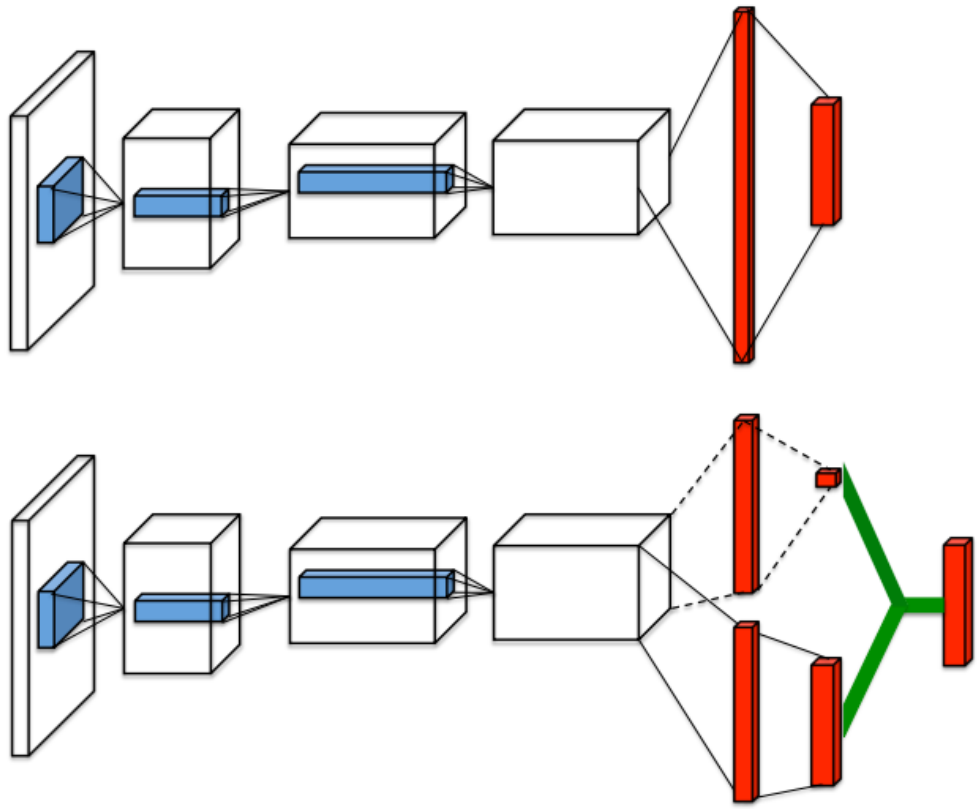


Figure 1. A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (9) to combine them. Both networks output Q-values for each action.

> referring to Dueling Network Architectures for Deep Reinforcement Learning

So, why dueling DQN works? Well, theoretically, the dueling DQN can directly learn the value of each state, therefore, the agent can avoid some risky, or bad, states instead of entering those states and try to choose an action to avoid losing point, and the advantage net can focus on avoiding bad actions. One example is shown in the picture below. The red area indicates the attention of each net. We can see that in the first graph, when the barrier is far away from the agent, the value network focuses on the horizon area, because this is the place where the barriers would appear. At the same time, the advantage net is 'absent-minded', because it knows that whatever it does, the final reward would not be affected. However, in the second graph, when one barrier is close, the advantage net would focus on the barrier and try to make decisions to avoid it. In conclusion, dueling DQN extract the V and A function, using to predict the value of states and the value of actions, which makes the net's performance better.

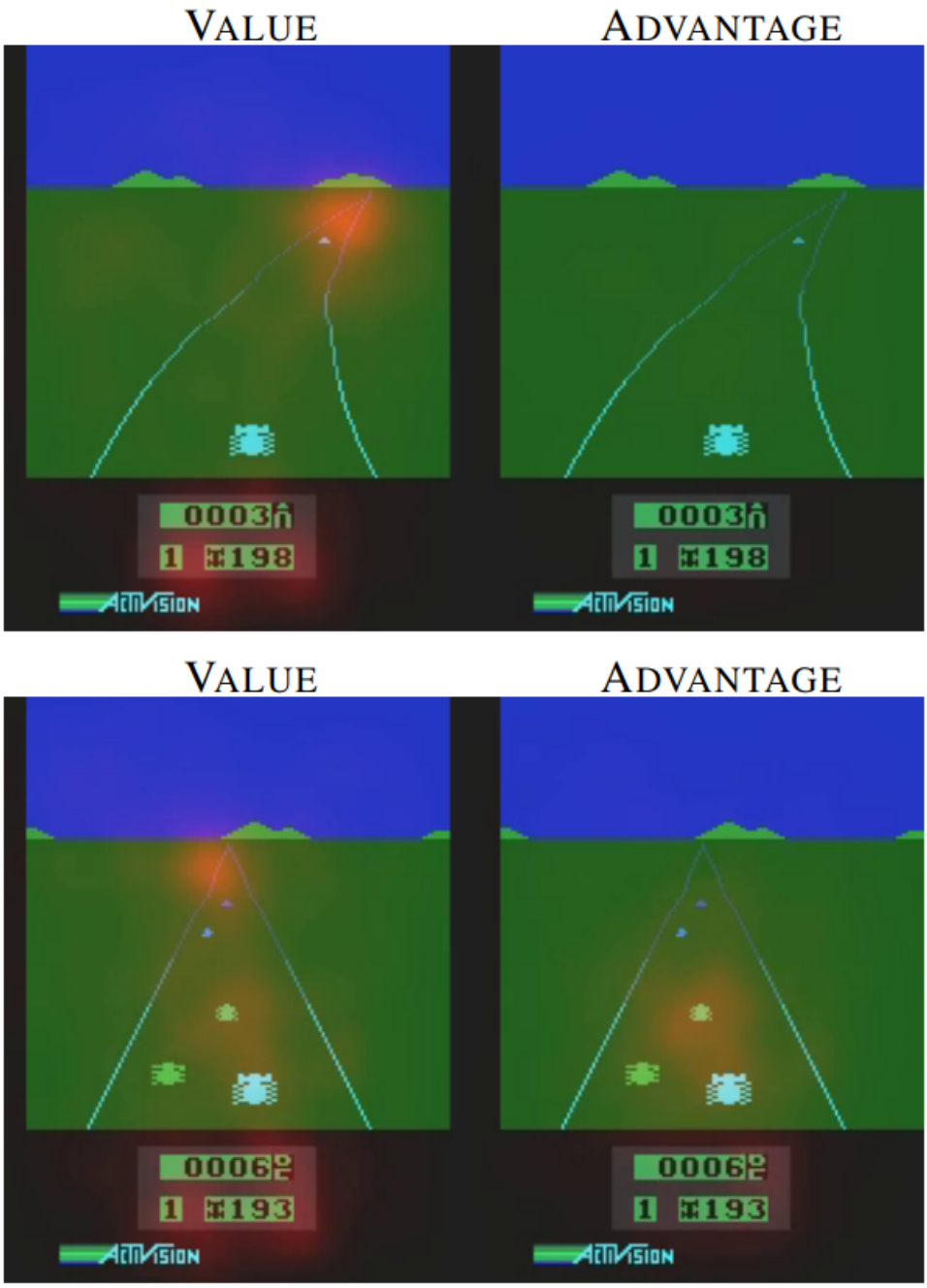


Figure 2. See, attend and drive: Value and advantage saliency maps (red-tinted overlay) on the Atari game Enduro, for a trained dueling architecture. The value stream learns to pay attention to the road. The advantage stream learns to pay attention only when there are cars immediately in front, so as to avoid collisions.

> referring to Dueling Network Architectures for Deep Reinforcement Learning

And when it comes to implementation, we will face some problems. According to the description above, we can get:

```
Q(s, a; theta, alpha, beta) = V(s; theta, beta) + A(s, a; theta, alpha)
```

theta is stands for parameters in convolutional layers, and alpha and beta stands for parameters in net A and V respectively. But this formula has a kind of problem called unidentifiable problem. When learning, the agent get some rewards from the environment, we cannot determine how much goes to the V function, and how much goes to the A function. In other words, the we can plus c, a constant, to V function, and minus c to A function, but get the same outcome Q. This will lead to fluctuate in the training period, and nonconvergence in the end.

In the paper Dueling Network Architectures for Deep Reinforcement Learning, the author proposed that set the maximum of A action to 0. And Q function will change into the following form:

```
Q(s, a; theta, alpha, beta) = V(s; theta, beta) + (A(s, a; theta, alpha) - max A(s,a';theta, alpha)))
```

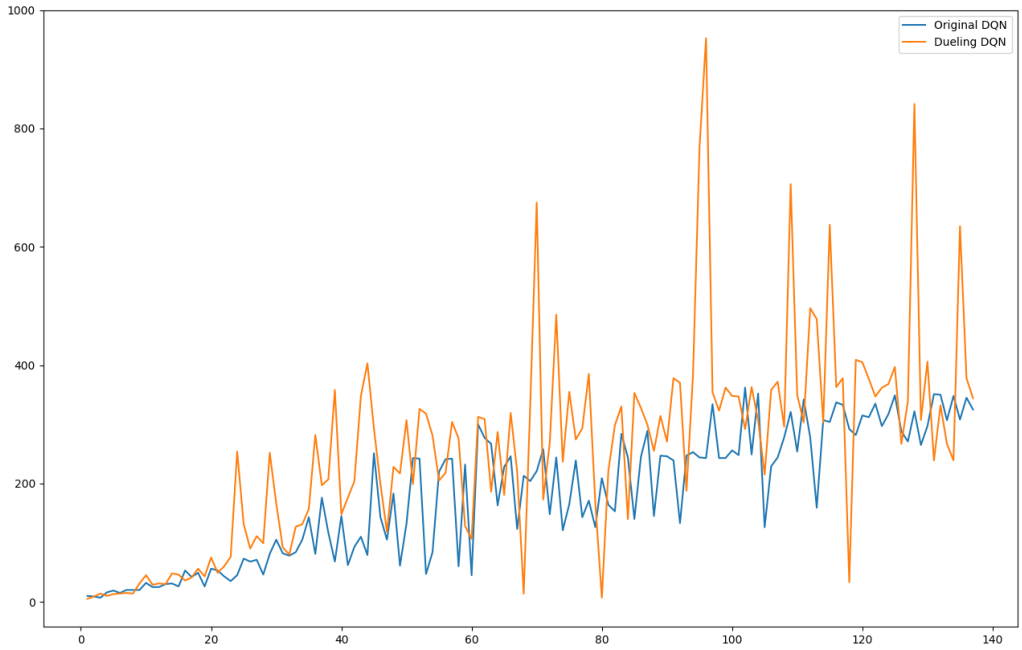
Let a* = argmax $_a$ Q(a,a'; theta, alpha, beta) = argmax $_a$ A(s,a'; theta, alpha), we can use the equation above to derive `Q(s, a*; theta, alpha, beta) = V(s; theta, beta)`. As a result, the problem is solved.

In practice, we use mean, instead of maximum, to solve the problem. This is a conclusion from practice, and performs better that maximum.

Thanks to pytorch, we just have to change the definition and the forward propagation functions, and do not have to adjust the back propagation progress. The code change is just as follows:

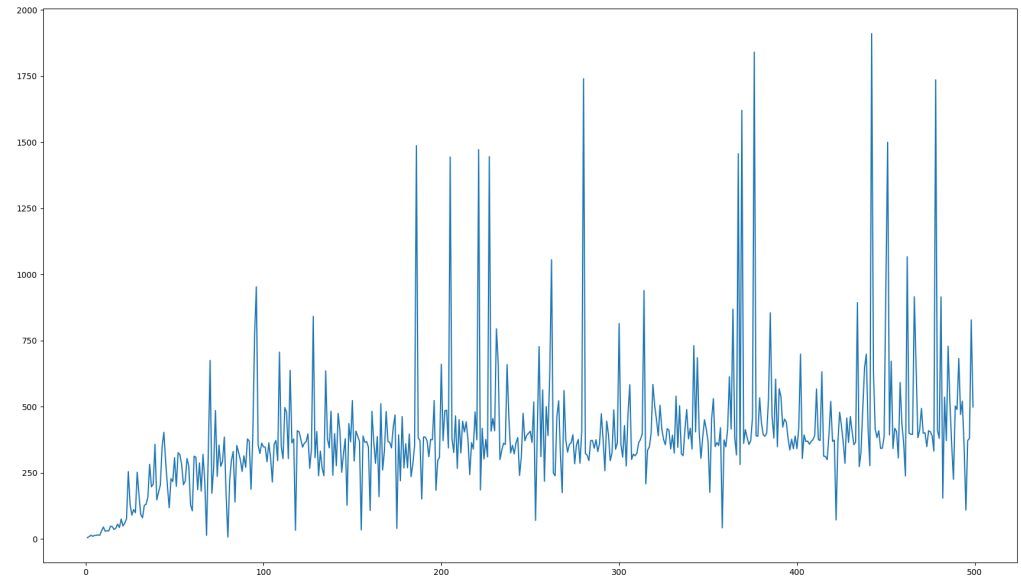
```
## in the utils_model.py
class DQN(nn.Module):

   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.advantage = nn.Sequential(
           nn.Linear(64*7*7, 512),
           nn.ReLU(),
           nn.Linear(512, action_dim)
       )
       self.value = nn.Sequential(
           nn.Linear(64*7*7,512),
           nn.ReLU(),
           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))
       x = x.view(x.size(0),-1)
       # x = F.relu(self.__fc1(x.view(x.size(0), -1)))
       advantage = self.advantage(x)
       value = self.value(x)
       return value + advantage - advantage.mean()
```

Due to the lack of time and computing power, we just compare the result of the first 137 epochs of training. The reward is shown as follows.



As we can see, the dueling DQN perform better than DQN most time. The reason for this improvement is discussed above.

We have trained the dueling DQN for 500 epochs and the rewards are shown below.We can see that there is an upward trend for the reward, but due to the lack of time and computing power, we cannot adjusting the superparameters or train the net more epochs to let the outcome convergent.



Stabilize the movement of the paddle

The main idea of how to stabilize the movement of the paddle is to encourage our agent to stay still when not necessary. And the method is that giving a slightly higher reward if our agent decides not to move (action 0). There is a dilemma that if we give our

agent a larger benefit from staying still, it will just does nothing all along the game. On the contrary, if we give our agent a smaller benefit, it will be hard for our agent to learn our aim, that is, the agent will perform no more stable than before. In short, it is hard to decide how to set the extra reward given to our agent due to staying still.

One possible idea is that we should not give the extra bonus at the beginning of the training process. The reason why we should avoid to do so is that at the beginning of the training, the agent has no idea how to catch the ball, as a result, it may learn to just stay still regardless the position of the ball. Although this strategy is obviously more stable than making random steps and can receive higher rewards, but we are not looking forward to an agent just staying still. So, if we can make our agent train for a while without the extra bonus for staying still, it can learn how to catch the ball properly. In that case, it will be suitable to add extra bonus back.

In order to realize the new feature, I modify the file utils_memory.py, adding a extra bonus to those unmoved steps.

```
def push(
       self,
       folded_state: TensorStack5,
       action: int,
       reward: int,
       done: bool,
) -> None:
   self.__m_states[self.__pos] = folded_state
   self.__m_actions[self.__pos, 0] = action

   # It is this sentence that has changed !!
   self.__m_rewards[self.__pos, 0] = reward + 0.01 * (action == 0)

   self.__m_dones[self.__pos, 0] = done
   self.__pos += 1
   self.__size = max(self.__size, self.__pos)
   self.__pos %= self.__capacity
```

Actually, the reward was an integer in the past, and so as the data type of `self.__m_rewards` . So, we should change the data type otherwise the fraction will be ignored.

```
self.__m_rewards = sink(torch.zeros((capacity, 1), dtype=torch.float16))
```

Stabilize the movement of the paddle

The main idea of how to stabilize the movement of the paddle is to encourage our agent to stay still when not necessary. And the method is that giving a slightly higher reward if our agent decides not to move (action 0). There is a dilemma that if we give our agent a larger benefit from staying still, it will just does nothing all along the game. On the contrary, if we give our agent a smaller benifit, it will be hard for our agent to learn our aim, that is, the agent will perform no more stable than before. In short, it is hard to decide how to set the extra reward given to our agent due to staying still.

One possibile idea is that we should not give the extra bonus at the beginning of the training process. The reason why we should avoid to do so is that at the begining of the training, the agent has no idea how to catch the ball, as a result, it may learn to just stay still regardless the position of the ball. Although this strategy is obviously more stable than making random steps and can receive higher rewards, but we are not looking forward to an agent just staying still. So, if we can make our agent train for a while without the extra bonus for staying still, it can learn how to catch the ball properly. In that case, it will be suitable to add extra bonus back.

In order to realize the new feature, I modify the file utils_memory.py, adding a extra bonus to those unmoved steps.

```
def push(
       self,
       folded_state: TensorStack5,
       action: int,
       reward: int,
```

```
        done: bool,
    ) -> None:
        self.__m_states[self.__pos] = folded_state
        self.__m_actions[self.__pos, 0] = action

        # It is this sentence that has changed !!
        self.__m_rewards[self.__pos, 0] = reward + 0.01 * (action == 0)

        self.__m_dones[self.__pos, 0] = done
        self.__pos += 1
        self.__size = max(self.__size, self.__pos)
        self.__pos %= self.__capacity
```

Actually, the reward was an integer in the past, and so as the data type of `self.__m_rewards` . So, we should change the data type otherwise the fraction will be ignored.

```
self.__m_rewards = sink(torch.zeros((capacity, 1), dtype=torch.float16))
```

In terms of the outcome of this modification, we can observe that the paddle becomes more stable, but flips still exits. I conclude that the extra bonus is not significant enough to totally eliminate the flips, but it can obviously reduce unnecessary movement.

In order to promote the performance of the agent, we should carefully set the extra bonus of staying still. It is hard to find a suitable value for the extra bonus for staying still, neither too large to discourage any movement nor too small so that the flips are not totally eliminated. Only by attempting a large number of different values can we have the ability to determine the best hyperparameter.