Mid-term Assignments report

Cui Canming, Deng Xiaojun

November 28, 2022

1 Introduction

In our Mid-term Assignments, we have accomplished the following:

- Read through the implementation and explain in detail.
- Boost the training speed using Prioritized Experience Replay.
- Using Double DQN.
- Using Dueling DQN.
- Stabilize the movement of the paddle.
- Using Federated learning to accelerate the training speed.
- Completed three bonus.

We have also open-sourced our code, which you can find on https://github.com/91Mrcui/SYSU_RL_Mid_term_DQN-breakout.git

In this report, we will present our work in three parts. The first part is a detailed explanation of the original DQN-breakout. The second part is our improvement work, including theoretical and physical significance and implementation methods. The third part is the presentation, comparison and analysis of the experimental results. The last part is a short summary.

You can take a quick look at the results of our experiments in Figure 1.

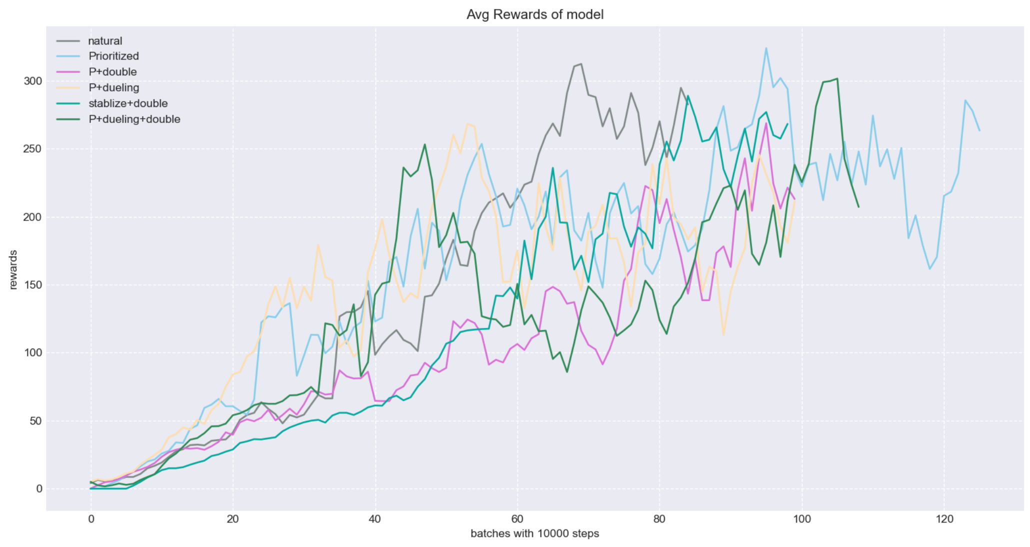

Figure 1: A quick look of experiments.

This is our authorship matrix:

Student ID	Member	Ideas(%)	Coding(%)	Writing(%)
20337025	Canming Cui	50	50	50
20337027	Xiaojun Deng	50	50	50

2 Analysis the origin implementation

Before we start our work, we need to have a thorough understanding of the original DQN-breakout project, in particular how each part of the DQN-breakout (agent, environment, etc.) is implemented and in which files it is contained. The architecture of the DQN-breakout project is shown in Figure 2.We will briefly describe the important elements in each files.

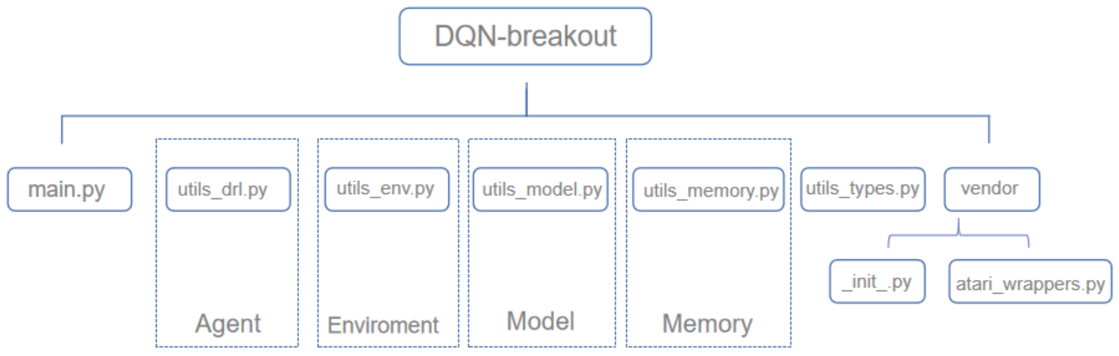

Figure 2: The architecture of the DQN-breakout project.

2.1 main.py

The main.py file is the entry point to the program, where we can see how the program is executed. It begins by defining a series of variables.We parse these variables in the code comments.

```
GAMMA = 0.99     #the discount (attenuation) factor
GLOBAL_SEED = 0      # seed
MEM_SIZE = 100_000  #capacity of Memory pool
RENDER = False   #if RENDER = TRUE, the game process will be rendered during each
                                            evaluation
SAVE_PREFIX = "./models"    #Where to save the model
STACK_SIZE = 4   #the channels in ReplayMemory

EPS_START = 1.   #epsilon initial value
EPS_END = 0.1    #epsilon final values
EPS_DECAY = 1000000 #Number of epsilon decays

BATCH_SIZE = 32 #the number of samples sampling from ReplayMemory
POLICY_UPDATE = 4   #Policy network update cycle
TARGET_UPDATE = 10_000  ##Target network update cycle
WARM_STEPS = 50_000 #wait until there are enough records in ReplayMemory
MAX_STEPS = 50_000_000  #the max number of training steps
EVALUATE_FREQ = 100_000 #Model evaluation cycle
```

Next, main.py initializes the random number, computing device, environment, agent and ReplayMemory. Then start implementing the DQN algorithm. Since this part of the code is long, we will not show it here, but introduce the execution process. In each step of the loop:

1. First, judge whether a round has ended. If it is finished, reset the environment state and put the observation data into the queue for storage.
2. Judge whether to start learning according to the size of the memory pool.

3. Observe the current state, and select actions according to the state, and then obtain the observed new information obs, rewards and whether the game is over.

4. Put the observation results into the queue, and record experience(current status, actions, reward and whether the game is over) in MemoryReplay.

5. If it is time to update the Policy network, update the Policy network.

6. If it is time to update the Target network, update the Target network network by modifying the parameters of the current network to be the same as the Policy network.

7. If it is time to evaluate the network, evaluate the current network, save the average reward and the Policy network(model), and end the game. If RENDER == True, rendered the frames.

2.2 utils_drl.py(Agent)

The Agent class is implemented in utils_drl.py, which acts as an agent in the DQN. It can choose actions according to policy and learn according to rewards.

The agent selects actions in the run() function through $\epsilon - greed$ policy. There is a $1 - \epsilon$ probability to select the action with the largest reward, and there is an ϵ probability to select the random action. It is important that the ϵ will gradually decrease, because the strategy selection should gradually stabilize as the training progresses:

```
def run(self, state: TensorStack4, training: bool = False, testing: bool = False) ->
                                           int:
        """run suggests an action for the given state."""
        # Decreasing epsilon
        if training:
            self.__eps -= \
                (self.__eps_start - self.__eps_final) / self.__eps_decay
            self.__eps = max(self.__eps, self.__eps_final)
        # according to policy
        if testing or self.__r.random() > self.__eps:
            with torch.no_grad():
                return self.__policy(state).max(1).indices.item()
        # Randomly select
        return self.__r.randint(0, self.__action_dim - 1)
```

In the learn() function, the agent learns through TD's learning strategy, updating the parameters of the neural network. For each learning, the agent will randomly sample the batch size from the memory, including status, operation, reward, etc. Then, the agent calculates the value of the current state according to the current policy, uses the target network to calculate the value of the next state, and calculates the expected value of the current state according to the updating formula.

After getting the current value and the expected value, agent will calculate loss function using smooth_l1_Loss. Then the loss will be used for backward to update the weights in the neural network through gradient descent.

```
def learn(self, memory: ReplayMemory, batch_size: int) -> float:
        """learn trains the value network via TD-learning."""
        #Learning by sampling from memory
        state_batch, action_batch, reward_batch, next_batch, done_batch = \
            memory.sample(batch_size)

        #Neural network learning according to the update formula
        values = self.__policy(state_batch.float()).gather(1, action_batch) #Q(s,a)
        values_next = self.__target(next_batch.float()).max(1).values.detach() #Q(s',a
                                                                        ')
        expected = (self.__gamma * values_next.unsqueeze(1)) * \
            (1. - done_batch) + reward_batch    #r + gamma * Q(s',a') * (1 - done),
                                                        Discussion on the
                                                        classification of done

        #Calculating loss functions and updating gradients
        loss = F.smooth_l1_loss(values, expected)
        self.__optimizer.zero_grad()
        loss.backward()
        for param in self.__policy.parameters():
            param.grad.data.clamp_(-1, 1)
        self.__optimizer.step()

        return loss.item()
```

Finally, The sync() function synchronizes the weight from the policy network to the target network. The save() function saves the state dict of the policy network.

2.3 utils_env.py(Environment)

utils_env.py imports some function from atari_wrappers.py, and configuring the breaak-out game environment. It can interact with agent. There are some functions in MyEnv class:

- reset(): If the game ends or terminates, the reset function resets and initializes the gym environment, and then returns to the original observation and reward.
- step(): Forward an action to the environment, and return the latest observation results, rewards and a bool value indicating whether the event is terminated.
- get_frame(): renders the current game frame.
- to_tensor(): converts an observation to a torch tensor.
- get_action_dim(): returns the reduced number of actions.
- get_action_meanings(): returns the actual meanings of the reduced actions.
- get_eval_lives(): returns the number of lives to consume in an evaluation round.
- make_state(): makes up a state given an obs queue.
- make_folded_state(): makes up an n_state given an obs queue.
- show_video(): creates an HTML element to display the given mp4 video in IPython.
- evaluate(): uses the given agent to run the game for a few episodes and returns the average reward and the captured frames.

2.4 utils_model.py(model)

The utils_model.py file defines the neural network model for the DQN and also defines the forward propagation process. The neural network architecture is shown in Table 1.

Conv Name or Linear Name	In Size	Out size
Conv1	84x84	20x20
Conv2	20x20	9x9
Conv3	9x9	7x7
fc_v1	64*7*7	512
fc_v2	512	3

Table 1: neural network architecture

We can see the structure of forward propagation from forward().The adjacent two-layer neural network contains an active layer.:

```
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 = F.relu(self.__fc1(x.view(x.size(0), -1)))
        return self.__fc2(x)
```

2.5 utils_memory.py

The memory pool is implemented in the file utils_memory.py, which mainly realize data storage and random sampling. There are two important functions: the push() function inserts experience into the queue, and the sample() function is used to sample in the queue.

3 Our experiment

3.1 Boosting the speed with Prioritized Experience Replay

3.1.1 The theoretical and physical significance

Not all transitions in memory can provide information that can enable the neural network to learn more efficiently. In the state of many transitions, the agent didn't make an efficient action, which leads to fewer reward and smaller TD-error. So a method to speedup the training is to make good use of the transitions with larger TD-error.

We found Prioritized Replay Experience DQN[SQAS15] is a DQN variant for this question. It introduces a stochastic sampling method that interpolates between greedy prioritization and uniform random sampling, which ensures that the probability of being sampled is proportional to the transition's TD-error, and guarantees a non-zero probability even for the low priority transition. The possibility of being sampled for transition i is

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

where p_i is the priority for transition i being sampled, and α determines how much prioritization is used.

The author also introduces two variant for implementing this method, and we choose the first one, proportional prioritization, where $p_i = |\delta| + \epsilon$, and ϵ is a small positive constant that prevents the edge-case of transitions not being revisited once their error is zero. In practice, we implement a Sum-Tree to help manage the priority of each transitions. Its implementation in detail is included in the code detail part.

But the estimation of the expected value with stochastic updates relies on those updates corresponding to the same distribution as its expectation. Prioritized replay introduces bias because it changes this distribution in an uncontrolled fashion, and therefore changes the solution that the estimates will converge to (even if the policy and state distribution are fixed). Fortunately this bias can be corrected by introducing a importance-sampling (IS) weights

$$w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^\beta$$

where N is the size of replay memory, and β is a exponent determines how much IS weight compensate for the bias. If $\beta = 1$, the IS weight fully compensate for non-uniform probabilities $P(i)$. In the Q-learning part, the weights will be folded into Q-learning updates by replacing δ with $w_i\delta$.

According to the paper, in typical reinforcement learning scenarios, the unbiased nature of the updates is most important near convergence at the end of training, because the process is highly non-stationary anyway, due to changing policies, state distributions and bootstrap targets. It is useful to exploit the flexibility of annealing the amount of importance-sampling correction over time, by defining a schedule on the exponent β that reaches 1 only at the end of learning. In both author's practice and ours, β can be linear annealed to from its initial value to 1.

3.1.2 Advancement implementation

Firstly is the implementation of Sum-tree structure, which is used for managing the priorities of the transitions stored in the memory. Observing that it is a complete binary tree, we implement it with an array, which is the same as a HEAP structure. The array implementation may not so visual as a recursion implementation, but it has better performance in reducing time for getting and updating.

```
class Sumtree(object) :
    def __init__(
        self ,
        capacity : int
    ) -> None:
        self.__arr = np.zeros(2 * capacity)
        #[1,capacity - 1] for interval and [capacity, 2 * capacity - 1] for leaf
        self.__capacity = capacity

    #updating from leaf nodes to interval nodes
    #if a leaf node changes, its ancestors should be updated
    def update(
        self,
        val : float,
        position : int
    ) -> None  :
        index = position + self.__capacity
        change = val - self.__arr[index]
        self.__arr[index] = val
        while True :
            parent = index // 2
            self.__arr[parent] += change
            if parent == 1 :
                break
            index = parent

    #getting the total sum of all priorities
    def get_root(self) -> float :
        return self.__arr[1]

    #getting a leaf node with a input value
    def get_leaf(
        self,
        val : float
    ) -> Tuple[int, float] :
        index = 1
        while True :
            lchild = index * 2                    #left child
            rchild = index * 2 + 1                #right child
            if index >= self.__capacity :
                break         #loop until a leaf node
            else :
                #if ge left child, go into the left child
                if self.__arr[lchild] >= val :
                    index = lchild
                #if lt left child, go into the right child and substract left child
                                                                value
                else :
                    val -= self.__arr[lchild]
                    index = rchild
        return index - self.__capacity, self.__arr[index]

    #getting the max priority of all transitions
    def get_p_max(self) -> float :
        return np.max(self.__arr[-self.__capacity:])

    #getting the min priority of all transitions
    def get_p_min(self) -> float :
        return np.min(self.__arr[-self.__capacity:])
    #getting a copy of the priorities. Used for computing IS weight
    def get_p_all(self):
        return np.copy(self.__arr[-self.__capacity:])
```

Next is modification of memory pushing, sampling and updating part. A new transition's priority is assigned the maximum of all priorities. While sampling, the total sum of priorities is divided into k ranges. We randomly choose a value in every range, and use it to get the sample index from the Sum-tree. Then, we use the sample index to fetch the transition from the replay buffer, and compute the importance-sampling weight. After the agent learning the samples, the sampled transitions' priority is updated to the new TD-error. But in practice, we clip the TD-error of the transition in order to limit some priorities to be too large.

```
    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
        self.__m_rewards[self.__pos, 0] = reward
        self.__m_dones[self.__pos, 0] = done

        p = self.__tree.get_p_max()
        if p == 0 :
            p = ABS_ERR_UPPER    #setting the init priority
        self.__tree.update(p, self.__pos)    #update Sum-tree
        self.__pos += 1
        self.__size = max(self.__size, self.__pos)
        self.__pos %= self.__capacity

    def sample(self, batch_size: int) -> Tuple[
            BatchIndex,
            BatchISweight,
            BatchState,
            BatchAction,
            BatchReward,
            BatchNext,
            BatchDone,
    ]:
        b_index = [0] * batch_size
        seg = self.__tree.get_root() / batch_size          #divide into segments
        probs = np.power(self.__tree.get_p_all(), ALPHA)
        sumProb = np.sum(probs)
        probs = probs / sumProb
        isweights = torch.Tensor(np.power(self.__capacity * probs, -self.__beta)).to(
                                                  self.__device)
        self.__beta = np.min([1.0, self.__beta + BETA_INC])     #annealing BETA
        #get a sample index from each range
        for j in range(batch_size) :
            a = j * seg
            b = (j + 1) * seg
            val = np.random.uniform(a, b)                       #get a random value
            index, p = self.__tree.get_leaf(val)                #get the sample's
                                                      index
            b_index[j] = index


        #computing IS weight and sampling
        b_index = torch.LongTensor(tuple(b_index))
        b_isweight = isweights[b_index]
        b_isweight /= torch.max(b_isweight)
        b_isweight = b_isweight.to(self.__device).float()
        b_state = self.__m_states[b_index, :4].to(self.__device).float()   #state
        b_next = self.__m_states[b_index, 1:].to(self.__device).float()    #next_state
        b_action = self.__m_actions[b_index].to(self.__device)
        b_reward = self.__m_rewards[b_index].to(self.__device).float()
        b_done = self.__m_dones[b_index].to(self.__device).float()

        return b_index, b_isweight, b_state, b_action, b_reward, b_next, b_done

    def batch_update(
        self,
        batch_index : BatchIndex,
        abs_err : BatchErr,
    ) -> None :
        batch_index = batch_index.cpu().data.numpy()
        abs_err = abs_err.cpu().data.numpy()
        abs_err += EPSILON                              #avoid priority being 0
        clip_err = np.minimum(abs_err, ABS_ERR_UPPER)    #avoid priority lt 1.0
        for i in range(len(batch_index)) :
            self.__tree.update(clip_err[i], batch_index[i])  #update Sum-tree
```

We also have to fold IS weights into TD-error while computing loss.

```
    weight_expected = expected.mul(isweight_batch)
    weight_values = values.mul(isweight_batch)
    loss = F.smooth_l1_loss(weight_values, weight_expected)
```

3.2 Using Dueling-DQN

3.2.1 The theoretical and physical significance

Duel[WSH+16] use already published algorithms, but it has different network architecture named dueling architecture. It is proposed on the basis of observing that not all states are valuable enough. Dueling architecture pay more attention to learn which state is valuable (or not valuable), without having to learn the effect of each action for each state.

In natural network, the states are the input stream, which pass a convolutional feature learning module and later a fully-connected value learning module. The output stream is the values of each action for states. In this neural network, there is just a single streams throughout this process. But in dueling architecture, it will be seperated into two streams that represents the value and advantage functions, while sharing a common convolutional feature learning module. The two streams are combined via a special aggregating layer to produce an estimate of the state-action value function Q. The difference between two architechures is shown as Figure 3.

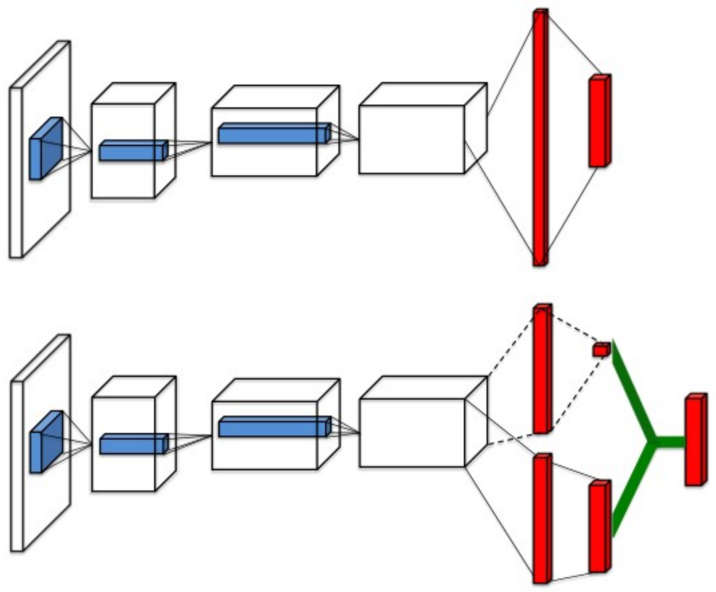

Figure 3: A popular single stream Q-network **(top)** and the duel-ing Q-network **(bottom)**. The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation to combine them. Both networks output Q-values for each action.

In dueling architecture, the action value Q can be represented as

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

where s is state, a is action, θ is parameters of convolutional layers and β and α are parameters of value and advantage fully-connected layers respectively.

3.2.2 Advancement implementation

The implementation of Dueling DQN is easy and simple. We have to separate the stream just after it passing feature learning mudule into V stream and A stream.

```
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)
        #v stream
        self.__fc_v1 = nn.Linear(64*7*7, 256)
        self.__fc_v2 = nn.Linear(256, 1)
        #a stream
        self.__fc_a1 = nn.Linear(64*7*7, 256)
        self.__fc_a2 = nn.Linear(256, action_dim)
        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)
        #v stream
        v = F.relu(self.__fc_v1(x))
        v = self.__fc_v2(v)
        a = F.relu(self.__fc_a1(x))
        a = self.__fc_a2(a)
        #a stream
        out = v.expand_as(a) + (a - a.mean().expand_as(a))   #aggregating layer
        return out

    @staticmethod
    def init_weights(module):
        if isinstance(module, nn.Linear):
            torch.nn.init.kaiming_normal_(module.weight, nonlinearity="relu")
            module.bias.data.fill_(0.0)
        elif isinstance(module, nn.Conv2d):
            torch.nn.init.kaiming_normal_(module.weight, nonlinearity="relu")
```

3.3 Using Double-DQN

3.3.1 The theoretical and physical significance

Double DQN[VHGS16] is a DQN variant for reducing action value overestimation and thus enhance the performance of the agent. Q-learning is known to learn unrealistically high action values because it contains a maximization step over estimated action values, which tends to make high overestimation. High overestimation have been proved to affect the stabability of training. Fortunately, Double Q-learning is a effective method for solving this question, by decoupling the selection from evaluation. Its idea is introduced to Double DQN. The selection of the action for next state's max action value requires another new neural network, but in fact, the existing policy network is just useful for this.

In natural DQN, the target of action values in an update process is

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

where S_{t+1} is the next state, a is action for next state, and θ_t^- the parameter of policy. But in Double DQN, the major difference is the target value is replaced by

$$Y_t^{(DoubleDQN)} = R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a, \theta_t), \theta_t^-)$$

. Intuitively, we use the policy network to estimate a maximum action value of $Q(S_{t+1}, a, \theta_t^-)$, and use this action to estimate the target value.

3.3.2 Advancement implementation

The implementation of Double DQN is also easy. We just estimate the action with max value of each next state in policy network and use it to estimate the next state's target value.

```
values = self.__policy(state_batch.float()).gather(1, action_batch)
values_next = self.__target(next_batch.float()).detach()
max_act = self.__policy(next_batch.float()).max(1).indices.unsqueeze(-1)
expected = (self.__gamma * values_next.gather(1, max_act)) * \
    (1. - done_batch) + reward_batch
```

3.4 Stabilize the movement of the paddle

3.4.1 The theoretical and physical significance

About how to stabilize the movement of the paddle (avoid high-freq paddle shaking effects) so that the agent plays more like a human player, we didn't find any relevant work on the Internet, so we put forward an idea that we could give more rewards to those stable actions that are more like people, and encourage agents to strengthen these actions.

But what kind of actions are more like human actions? We invited several students to play the game and observed their operations. Finally, we came to the conclusion that:

1. When the ball bounces upward, the paddle controlled by human players tends to stay still rather than shake violently.
2. The paddle controlled by human players will not change direction many times in a very short time when moving, and will always move in one direction.

We encourage agents to do the above behaviors by giving additional rewards. Consider three consecutive actions a_{t-2}, a_{t-1}, a_t. If $a_{t-2} = a_{t-1} = a_t$, we should give the agent the highest additional reward. If $a_{t-2} \neq a_{t-1}, a_{t-1} = a_t$, we should give the agent the second highest additional reward. If $a_{t-2} = a_{t-1}, a_{t-1} \neq a_t$, it means that this behavior breaks the continuous action and should be punished. If the three actions are different, the lowest additional reward should be given, because we do not want to see this situation.

a_{t-2}	a_{t_1}	a_t	additional reward
0	0	0	6
1	1	1	4
2	2	2	4
1	0	0	3
2	0	0	3
1	1	0	-1.5
1	1	2	-1.5
2	2	0	-1.5
2	2	1	-1.5
2	2	0	-1.5
0	0	1	-1.5
0	0	2	-1.5
others	...	...	-3

Table 2: additional reward table

In addition, different types of actions will also bring different additional rewards. Generally, the additional reward of "stay" is higher than the other two.

The additional reward settings for each specific case are shown in the Table 2 (which we will subsequently adjust based on the results of the experiment).

3.4.2 Advancement implementation

To implement the above mechanism, we need to add the following code in the loop of mian.py file:

```
    if action == old_action and old_action == older_action:
        if action == 0:
            reward += 6
        else:
            reward += 4
    if action == old_action and old_action != older_action:
        reward += 3
    if action != old_action and old_action == older_action:
        reward -= 1.5
    if action != old_action and old_action != older_action:
        reward -= 3
    if done:
        reward -= 100
    #update the record
    older_action=old_action
    old_action = action
```

We trained the stabilize DQN-breakout and found that although the addition of stable operation will slow down the speed of convergence of DQN, it can make the racket more stable, that is, make the agent plays more like a human player. You can see the corresponding video in our attachment, where the agent looks like a human player.

3.5 Accelerate the training speed with Federated Learning

This is just a simple attempt on our part, and more experimentation is needed.

3.5.1 The theoretical and physical significance

Firstly, we have different interpretations of "Accelerate the training speed". We consider that this is a method to accelerate the training speed if an agent can train fewer times to achieve a model that is as good as other agents that has been trained more times. So we propose the idea of using Federated Learning[LSTS20] to improve the performance of reinforcement learning models.

Federated learning is a distributed machine learning technique whose core idea is to build global models based on virtual fused data by training distributed models across multiple data sources with local data, without exchanging local individual or sample data, but only by exchanging model parameters or intermediate results.

We trained different agents in different environments for various rounds for each experiment. In each round, the server aggregates those information and comes up with a global model and we have a global model at last.

3.5.2 Advancement implementation

Since it was more difficult to implement on the framework of this project, we referred to this project in our implementation and validated our Federated-Learning-DQN ideas on their framework.

We ran two experiments on the breakout game for comparison, a natural DQN experiment and a Federated natural DQN experiment. In the Federated Learning experiment, we used 5 clients and set 2 clients to be trained in each round. In both experiments, we set the total number of training sessions to 250,000 and evaluated the model once every 10,000 training sessions and recorded the reward for this evaluation, so that the model would be evaluated a total of 25 times.

We record the rewards of each evaluation model and plot the line graph of the reward transformations. A comparison of the two experiments is shown in Figure 4, where the evaluation curve for the natural DQN is in orange and the evaluation curve for the Federated Learning DQN is in blue, and it can be seen that the agent participated in Federated Learning obtained a higher score for the model when trained for the same number of rounds. In other words, the Federated Learning DQN took less time in achieving the same excellent model.

This part is only a tentative attempt and will be studied in depth when the opportunity arises.

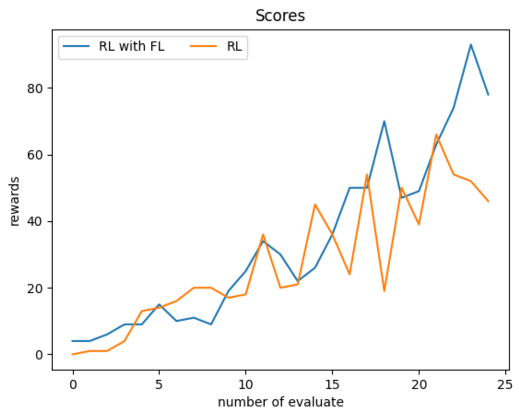

Figure 4: The comparison of the DQN and Federated Learning DQN.

4 Experimental results

We trained several models, and the corresponding rewards images were drawn. Due to time and resource constraints, we set MAX_STEPS = 10_000_000 for each model training, i.e. evaluate the model 100 times, which allows the training results to be displayed more quickly.

The rewards for natural DQN change as follows.

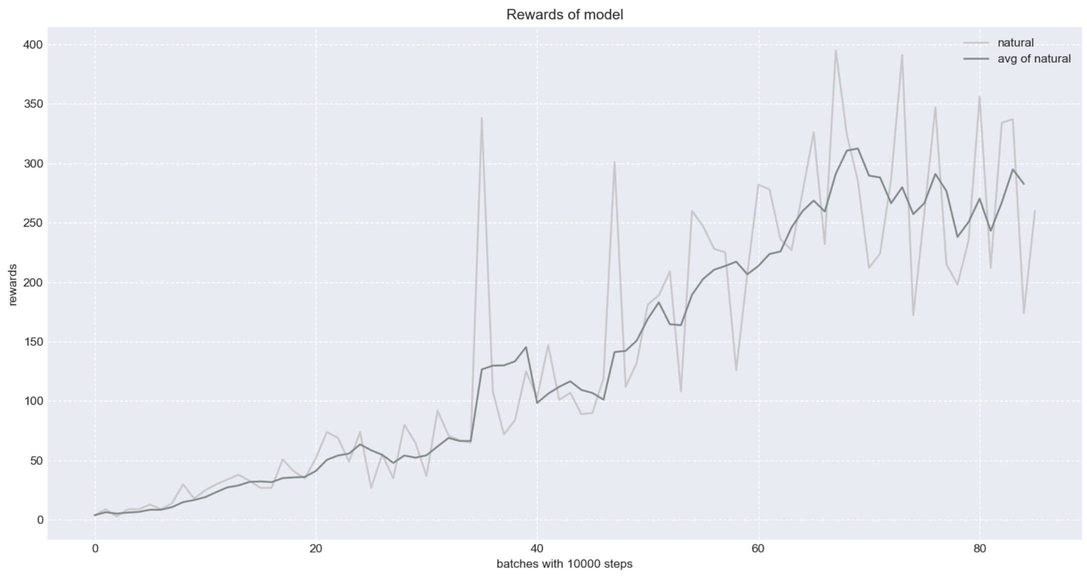

Figure 5: Natural DQN

During the experiment, we found that the upper bricks scored more than the lower bricks. Therefore, the agent will learn to hit the ball into the space above as much as possible, which will make some evaluation scores rise rapidly.

4.1 Prioritized Experience Replay

We can see that the prioritized experience replay has played a role in improving performance. The rewards have grown faster, and the optimal performance is better than natural DQN.

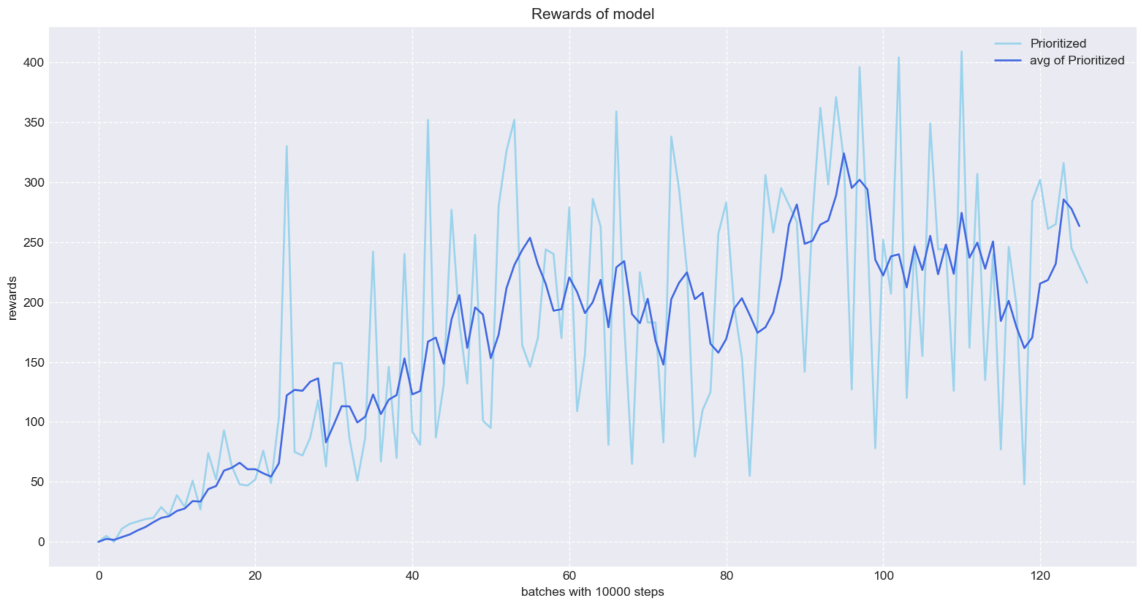

Figure 6: DQN with Prioritized Experience Replay

4.2 Prioritized Experience Replay + Double DQN

The performance of Double DQN with Prioritized Experience Replay is not very good. We consider this is caused by insufficient training time, because it is still rising at a relatively high speed.

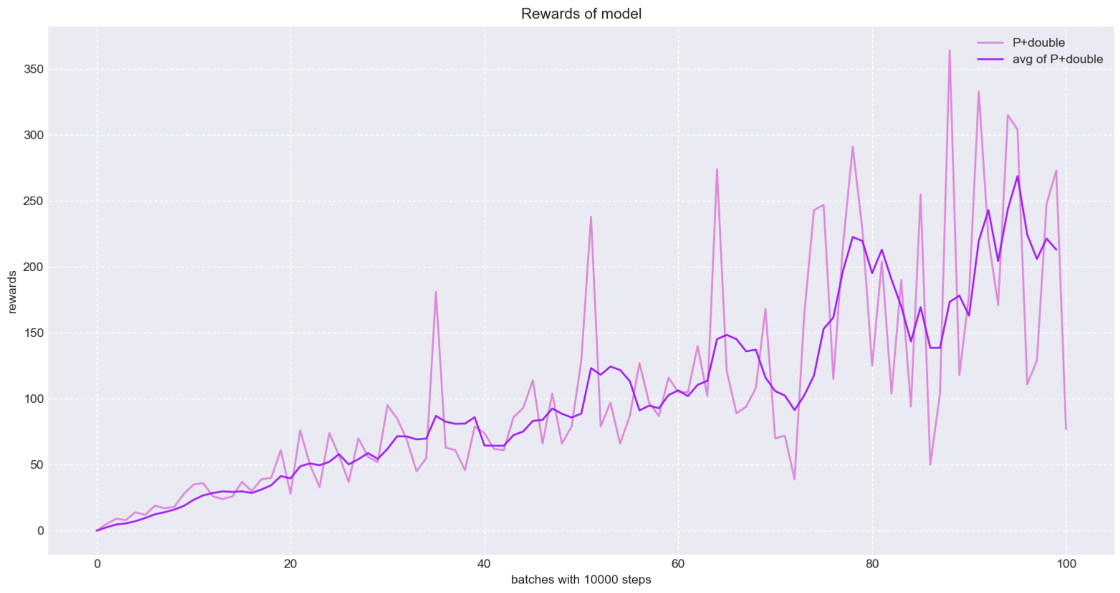

Figure 7: Double DQN with Prioritized Experience Replay

4.3 Prioritized Experience Replay + Dueling DQN

The convergence rate and rewards of Dueling DQN alone are not excellent. Dueling DQN seems not suitable for the game of break out, which was also mentioned in its original paper.

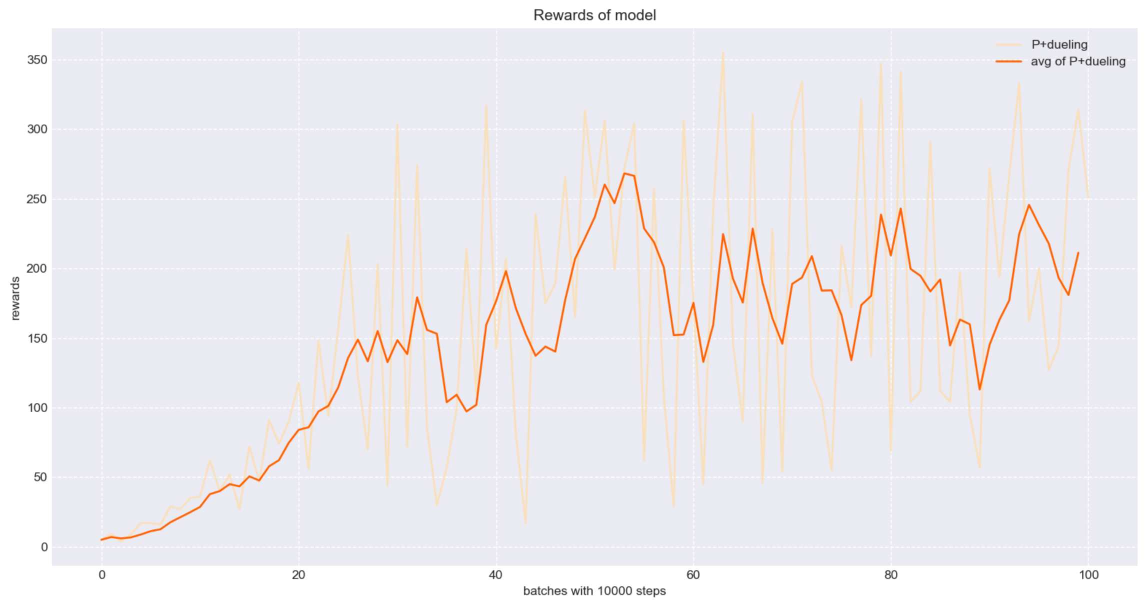

Figure 8: Dueling DQN with Prioritized Experience Replay

4.4 Prioritized Experience Replay + Dueling + Double DQN

When we use the three together, we have not achieved excellent results.

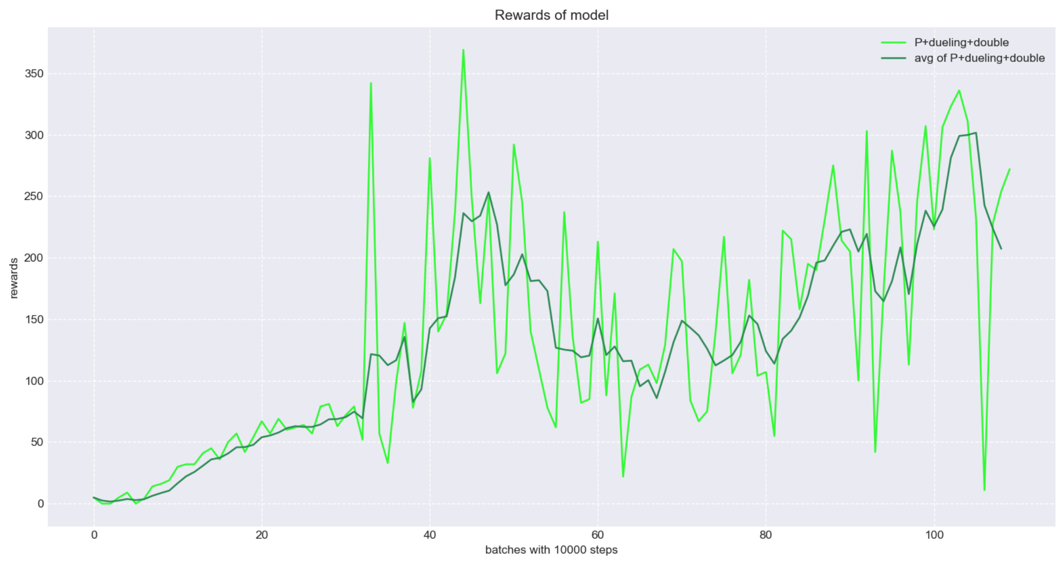

Figure 9: Prioritized Experience Replay + Dueling + Double DQN

4.5 stabilize the movement+Double DQN

At the end of the experiment, we try to add the stabilization technology to Double DQN, and its convergence rate and rewards are as follows.

The change of the image is not obvious, but in our demonstration video, you can see that the agent is really like a human player.

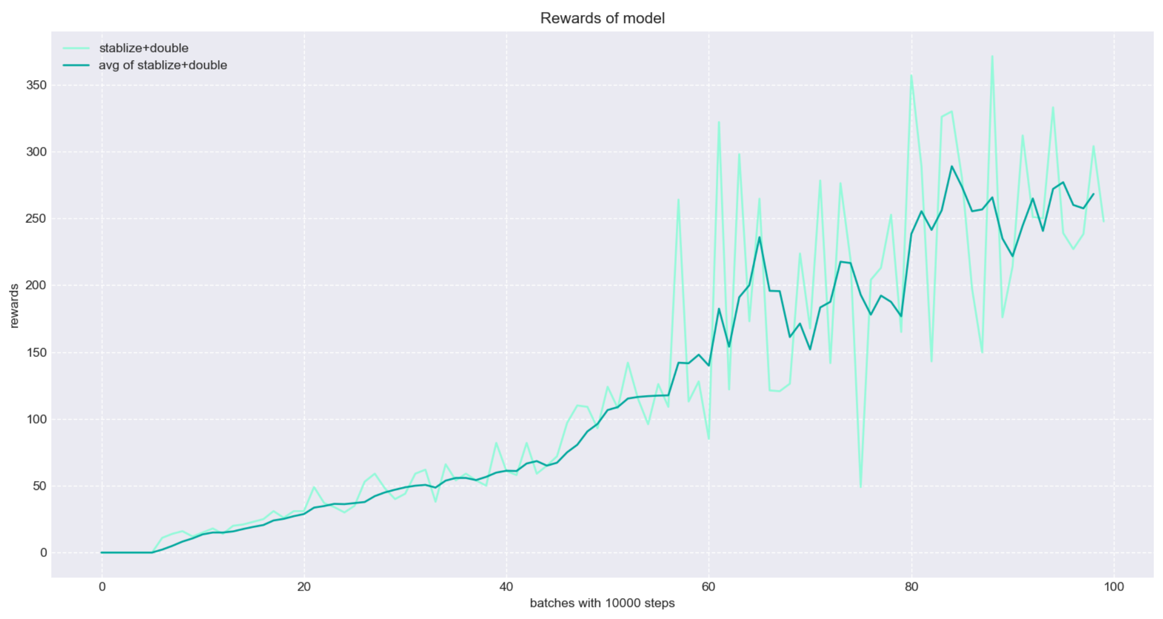

Figure 10: stabilize the movement+Double DQN

We have placed relevant codes of all models and two demonstration videos in the attachment, one of which is stable-Double-DQN and the other is the model without stable method. You can also find relevant documents in the github's repositories we mentioned at the beginning.

5 Conclusion

This mid-term project showed us the basic application of Nature-DQN, and then we make certain optimizations on this basis. We have tried various combinations of optimisation: Prioritized Experience Replay, Duling DQN, Double DQN, stabilize the movement and Federated Learning, and have also analysed the corresponding experimental results.We look forward to the opportunity to investigate the DQN-breakout problem in more depth in the future.

Finally,, we would like to give our thoughts on completing this Mid-term Assignment. During this Mid-term Assignment, we worked together and divided up the work to complete this challenging task. In the process of completing the project, we reviewed what we had learnt in class and learnt a lot of new knowledge and gained a lot of valuable experience. In addition, also gained a lot from this process of encountering problems - thinking about them - solving them. We have developed the good habit of thinking and analysing independently first when I encounter difficulties in experimental difficulties, and after thinking and trying and then search for relevant information, so that our work efficiency will be higher and we will be able to improve ourselves more.

References

[LSTS20] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[VHGS16] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[WSH$^+$16] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.