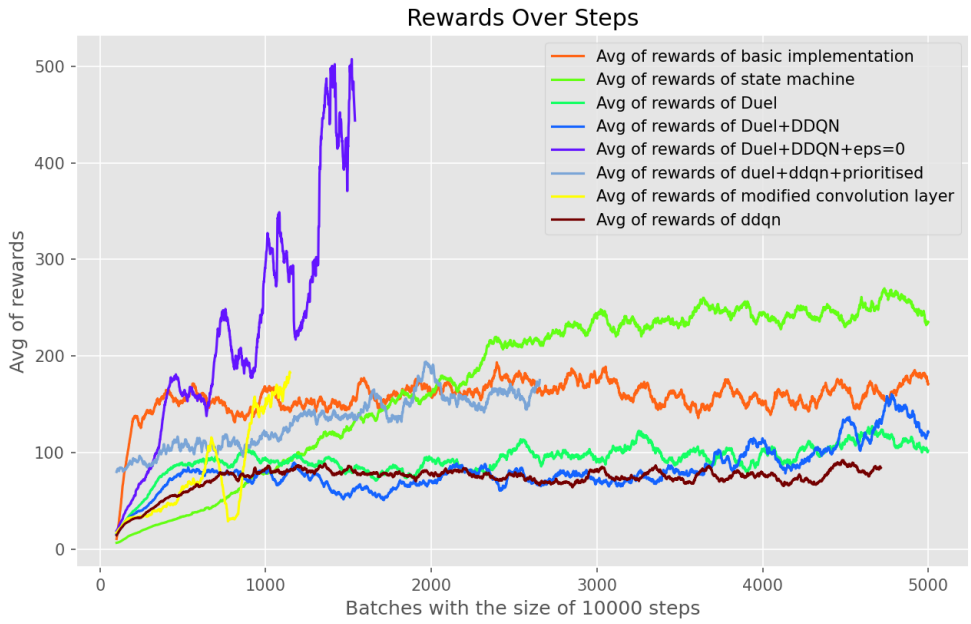
Exploration of playing Atari Breakout using DQN

1 A quick preview of our work



In this project, we finished these following requirements

- **Boost the training speed using Prioritized Experience Replay✓**
- **Improve the performance using Dueling DQN✓**
- **Stabilize the movement of the paddle✓**(The model training is very successful and converges quickly)

You can check our work via this link `https://github.com/yanying8592/DQN-breakout`
In addition, we have made innovations in other aspects

- We added ddqn model and integrated ddqn model with dueling dqn model
- We explore the method of fast training model, which can quickly train convergence in a very short number of training times
- During the training, we found several interesting agent models. Their strategy is to save life rather than choose to pass the customs. Therefore, such models will never lose the game and continue the game forever while getting high scores.

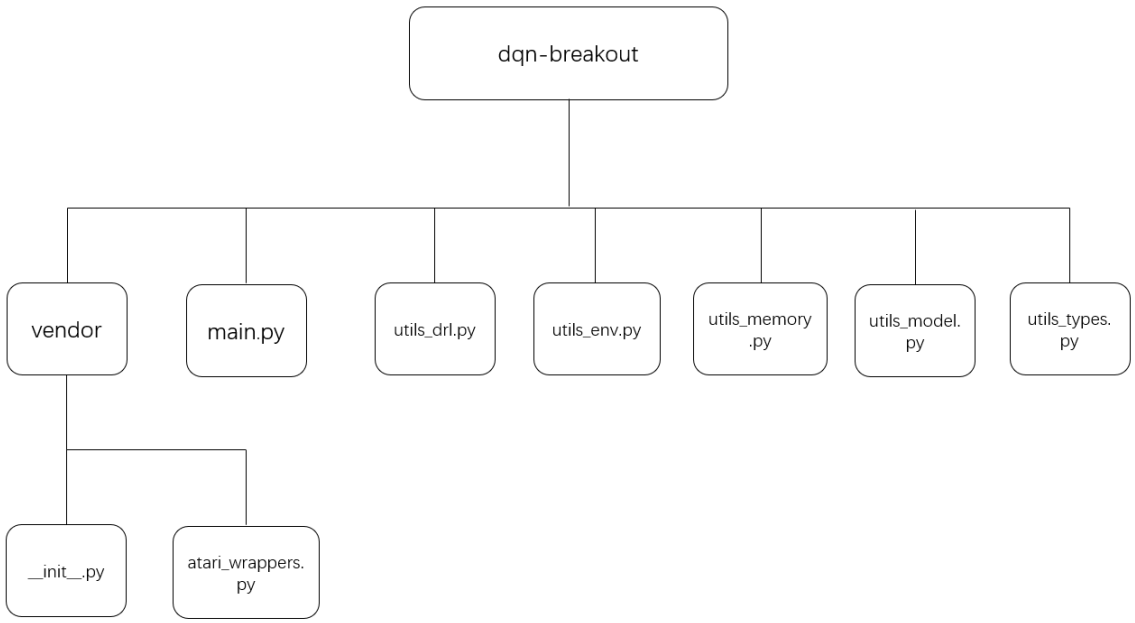
Member	Ideas (%)	Coding (%)	Writing (%)
Jianing Zheng (19335281)	50%	50%	50%
Zhuojie Yang (19335240)	50%	50%	50%

2 Introduction

In this report, we will divide the mid-term project into three parts: In the first part, we will explain in detail that what each component is responsible for, and how the components are connected together in the given implementation. In the second, third and fifth parts, we will try to use different methods to improve the original model and explain the reasons for using them and their structure. These methods include Priority Experience Replay, Dueling-DQN and Double-DQN. In the fourth part, we try to stabilize the movement of paddle by using the idea of state machine, which increases the stability of paddle to a certain extent. Finally, in the sixth part, we will compare and analyze the experimental results obtained by using the improved method mentioned above.

3 Code analysis

Before starting, It's necessary to know that what the components of the project are and where they are located. Just look at the figure below:



In this project, dqn-breakout is a parent folder that contains several python files: main.py, utils_drl.py, utils_env.py, utils_memory.py, utils_model.py, utils_types.py, and a subfolder vendor,

which contains two files: __ init __.py, atari_wrappers.py. In order to make the report as concise as possible, we will not introduce each file, but explain the important components around the whole DQN process, and finally connect them together.

3.1 agent

In utils_drl.py, there is a class Agent. It provides an agent, which is the main character of dqn-breakout. It can choose the corresponding action according to the strategy and learn according to the rewards.

```
class Agent(object):
...
```

The agent will take ε-greedy policy to select the next action according to the current state.

```
def run(self, state: TensorStack4, training: bool = False) -> int:
    """run suggests an action for the given state."""
    if training:
    self.__eps -= \
    (self.__eps_start - self.__eps_final) / self.__eps_decay
    self.__eps = max(self.__eps, self.__eps_final)

    if self.__r.random() > self.__eps:
    with torch.no_grad():
    return self.__policy(state).max(1).indices.item()
    return self.__r.randint(0, self.__action_dim - 1)
```

The above run function will return the action which the agent will select. If the random number is greater than eps, the agent will randomly select an action; Otherwise, the agent will select the action with the greatest action value among the existing actions. It should be noticed that ε decays according to the exp curve until it is equal to eps_ final, because in the initial stage of training, agents should explore more and then stabilize slowly.

Next, let's talk about how agents learn.

```
def learn(self, memory: ReplayMemory, batch_size: int) -> float:
    """learn trains the value network via TD-learning."""
    state_batch, action_batch, reward_batch, next_batch, done_batch = \
    memory.sample(batch_size)
    values = self.__policy(state_batch.float()).gather(1, action_batch)
    values_next = self.__target(next_batch.float()).max(1).values.detach()
    expected = (self.__gamma * values_next.unsqueeze(1)) * \
    (1. - done_batch) + reward_batch
    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()
```

The learning process of agent makes use of TD learning strategy. Each time, the agent gets a sample of batch_size size from the memory, which contains states, actions, rewards and so on. Then the agent uses the current state to calculate the value according to the existing policy. When estimating the value corresponding to the next state, the agent adopts the Q learning strategy and directly selects the maximum action reward.

After getting the current value and the expected value, we use smooth_ l1_ Loss method to calculate the loss. The calculation formula is:

$$loss(x, y) = \frac{1}{n} \sum_i z_i$$

where z¡sub¿i¡/sub¿ is given by:

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if —}x_i \text{ - } y_i\text{— ¡ 1} \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$$

The loss will be used for backward to update the weights in the neural network through gradient descent. Considering that gradient explosion may occur during training, we add gradient truncation: param.grad.data.clamp_ (min,max)

3.2 memory

In utils_memory.py, there is a class ReplayMemory. It is a container that records and updates the states, actions, rewards, etc. corresponding to each pos, and provides the agent with batch_size samples.

```
    class ReplayMemory(object):
    ...
    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

    self.__pos = (self.__pos + 1) % self.__capacity
    self.__size = max(self.__size, self.__pos)

    def sample(self, batch_size: int) -> Tuple[
    BatchState,
    BatchAction,
```

```
        BatchReward,
        BatchNext,
        BatchDone,
        ]:
        indices = torch.randint(0, high=self.__size, size=(batch_size,))
        b_state = self.__m_states[indices, :16].to(self.__device).float()
        b_next = self.__m_states[indices, 1:].to(self.__device).float()
        b_action = self.__m_actions[indices].to(self.__device)
        b_reward = self.__m_rewards[indices].to(self.__device).float()
        b_done = self.__m_dones[indices].to(self.__device).float()
        return b_state, b_action, b_reward, b_next, b_done
```

The value of capacity is the capacity of the memory. We record the latest sample in the memory through the push function. If the number of records exceeds the capacity of the memory, the oldest records will be overwritten.

When the memory is asked to provided batch_size samples to the agent, it would generate batch_size random numbers firstly, and these numbers are the indices of the samples to be taken. Then we take the corresponding states, actions, rewards and so on from each array according to these indices and return them to the agent.

3.3 env

utils_model.py may be a little complicated, and it imports some function from atari_wrappers.py, which is in vendor folder. It provides the environment of dqn-breakout, and it can interact with the agent.

```
        class MyEnv(object):
        ...
```

Env provides many functions, but here we will only explain the more important ones.

```
        def step(self, action: int) -> Tuple[TensorObs, int, bool]:
        action = action + 1 if not action == 0 else 0
        obs, reward, done, _ = self.__env.step(action)
        return self.to_tensor(obs), reward, done
```

The function step forwards an action to the environment and returns the newest observation, the reward, and an bool value indicating whether the episode is terminated. It should be noticed that in this function, env.step function is called and it is not equal to step. Actually, env is return by the function warp_deepmind, which is in atari_wrappers.py.

```
        def reset(
        self,
        render: bool = False,
        ) -> Tuple[List[TensorObs], float, List[GymImg]]:
        """reset resets and initializes the underlying gym environment."""
        self.__env.reset()
        init_reward = 0.
        observations = []
```

```
        frames = []
        for _ in range(5): # no-op
        obs, reward, done = self.step(0)
        observations.append(obs)
        init_reward += reward
        if done:
        return self.reset(render)
        if render:
        frames.append(self.get_frame())

        return observations, init_reward, frames
```

If the game is end or it is terminated, the function reset will reset and initialize the gym environment, then it will return the original observations and reward.

```
        def make_state(obs_queue: deque) -> TensorStack4:
        return torch.cat(list(obs_queue)[1:]).unsqueeze(0)
```

The function make_state makes up a state given an obs queue.

```
        def evaluate(
        self,
        obs_queue: deque,
        agent: Agent,
        num_episode: int = 3,
        render: bool = False,
        ) -> Tuple[
        float,
        List[GymImg],
        ]:
        self.__env = self.__env_eval
        ep_rewards = []
        frames = []
        for _ in range(self.get_eval_lives() * num_episode):
        observations, ep_reward, _frames = self.reset(render=render)
        for obs in observations:
        obs_queue.append(obs)
        if render:
        frames.extend(_frames)
        done = False

        while not done:
        state = self.make_state(obs_queue).to(self.__device).float()
        action = agent.run(state)
        obs, reward, done = self.step(action)

        ep_reward += reward
        obs_queue.append(obs)
        if render:
        frames.append(self.get_frame())
```

```
31
32          ep_rewards.append(ep_reward)
33
34          self.__env = self.__env_train
35          return np.sum(ep_rewards) / num_episode, frames
```

The function evaluate uses the given agent to run the game for a few episodes and returns the average reward and the captured frames.

3.4 model

In utils_model.py, there is a class DQN. It provides a neural network framework, which is the model for calculating the value of each action, for the project, and initializes the weight of each neuron. Of course, it also defines the forward process.

```
1           class DQN(nn.module):
2           ...
```

First, let's look at the structure of neural networks.

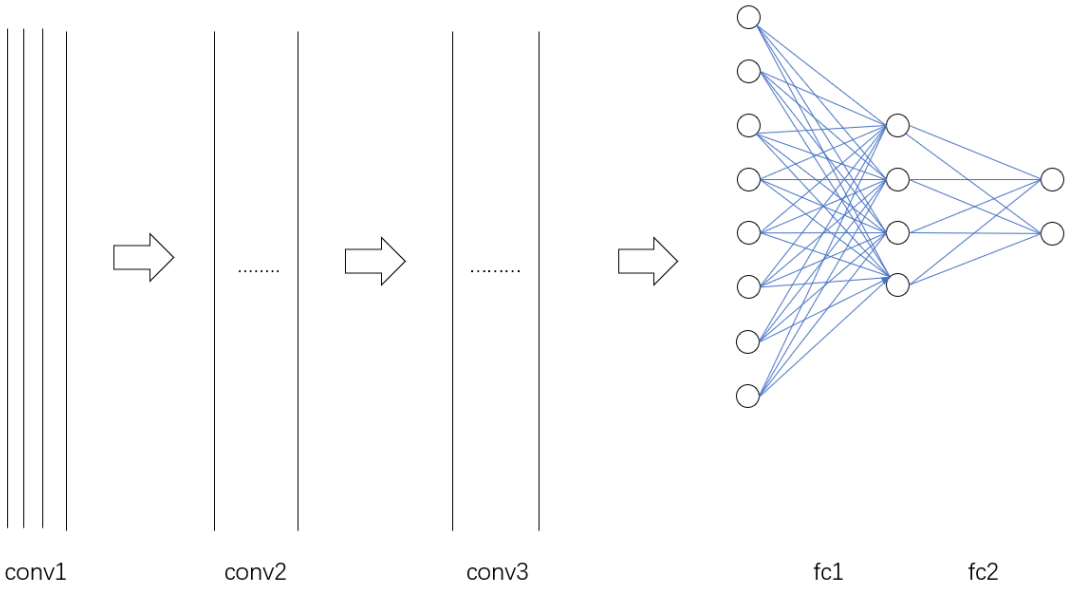
```
1           def __init__(self, action_dim, device):
2           super(DQN, self).__init__()
3           self.__conv1 = nn.Conv2d(4, 32, kernel_size=8, stride=4, bias=False)
4           self.__conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2, bias=False)
5           self.__conv3 = nn.Conv2d(64, 64,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

From the above code, it is not difficult to see that the neural network has three convolution layers and two full connection layers. For the size obtained after convolution, there is such a formula.

$$conv = \frac{X - kernel + 2 * padding}{stride} + 1$$

Where conv is the size after convolution in a certain dimension, and X is the size before convolution. In addition, because the neural network should calculate the value of each action, so the output dimensions of the second full connection layer fc2 should be equal to action space's dimensions.

The neural network may look like this:



In order to improve the generalization ability of neural network, we should add an activation layer to the two adjacent layers of neural network. As shown below, we use RELU function as the activation layer.

```
1       def forward(self, x):
2       x = x / 255.
3       x = F.relu(self.__conv1(x))
4       x = F.relu(self.__conv2(x))
5       x = F.relu(self.__conv3(x))
6       x = F.relu(self.__fc1(x.view(x.size(0), -1)))
7       return self.__fc2(x)
```

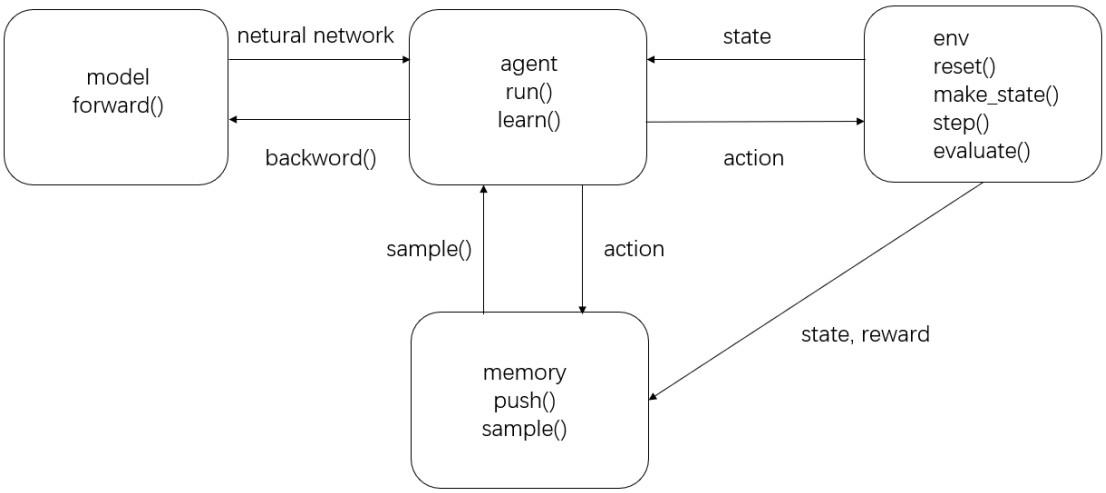
The last is the initialization of weights. Here, we choose to use Kaiming initialization method and make the bias of convolution layer zero.

```
1       def init_weights(module):
2       if isinstance(module, nn.Linear):
3       torch.nn.init.kaiming_normal_(module.weight, nonlinearity="relu")
4       module.bias.data.fill_(0.0)
5       elif isinstance(module, nn.Conv2d):
6       torch.nn.init.kaiming_normal_(module.weight, nonlinearity="relu")
```

3.5 connect

If we compare the four upper parts to the limbs, then main.py is the body connecting the limbs. Here, agent, memory and env will be instantiated. It should be clear that the model is instantiated as policy neural network and target neural network in the agent.

The figure below may help understanding how the components are connected together:



For each step in the project:

- If the game is end or it is terminated, the env will call reset function, and the obs_queue will add original observations at the end.
- The current state will be obtained after the env calls make_state function, then the agent selects an action according to the state. Next, the env forwards the action to the environment and returns the newest observation and the reward. Observation will be added to the obs_queue,and next state, action and reward will be pushed to memory.
- If it's time to update policy, the agent will call the learn function.
- If it's time to update target, the agent will call the sync function.
- If it's time to evaluate the current result, the env will call the evaluate function to get avg_reward, and the value of step, avg_reward will be written into a file rewards.txt.

4 Prioritized Experience Replay: Turbo the speed of convergence

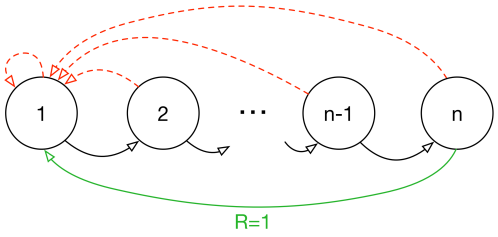
4.1 Why use Prioritized Experience Replay?

The training process of DQN can be roughly summarized as: input the current state into model, and obtain q values of all actions that may correspond to the current state of model. Then use Epsilon Greedy to select any random action or action that maximizes the q value. Apply the current action to the game simulation environment to obtain new state and reward and store them in the experience pool.

The role of experience pool is to train agents. When training agents, the traditional DQN training method is to randomly sample experience samples from the experience pool, use these samples to train the model and obtain the loss function, and then update agents according to the loss function. Although there is no problem with such a training mode – the agent is trained by using the previously collected experience so that it can learn the experience in the

experience pool, we believe that experience can be classified into important and unimportant. A sample taken from the experience pool (in Breakout's case) is a frame token from the gamer's screen .Some frames may just be the ball flying in the air without touching the bricks or falling to the ground, so the agent can not learn any experience from it, and even worse. But some frames are important, like the ball hitting a brick, or the ball hitting the board and the ball falling off the screen.

Obviously, these experiences are more useful, so learning these experiences is more effective for training agents. For example:



For this game, starting from State 1, a unique reward can only be obtained by walking along each state one by one to State N, with a probability of $\frac{1}{2^n}$. If a completely random strategy is used, the probability is extremely low, so the agent may never know that the reason why he did not get the reward is that he did not move to the right, so the training will never converge. If useful experiences can be selected from the experience pool, the agent may likely to summarize its own theorie of playing the game from these experiences in the long training process.

4.2 How to implement Priority Experience Replay?

How to determine which experience is more useful? Obviously, it needs to be used in combination with value function. We know that the state value function is usually used to represent the quality of the state, and the Q value function is used to represent the quality of the action in the policy, because the model we want to train can actually be understood as a huge policy decision-making unit, which makes a series of analysis and decisions through input, and finally gives the possible Q value function in the current state. Therefore, we can find that if we want to better train the model, the best way is to start with the output data of the model. Therefore, the author here takes TD error as a standard to judge the quality of experience.

In the previous study, we know that TD learning is a short-sighted way to quickly obtain the reward of the next action. TD error simply makes a difference between the Q value that may be obtained in the next state and the Q value in the current state. The author believes that if the difference between the two is large, it indicates that the current decision-making using model will have a great impact on the environment. It is likely that the ball falls or breaks a brick, resulting in a large positive or negative rewards.

TD error can be written in the following form, where Q target is the output of the model

we are training, and the other Q we use as the benchmark is the previously locked model.

$$\delta = r + \max_{a'} \gamma Q_{\text{target}}\ (s', a') - Q(s, a)$$

Obviously, we can directly use δ as the standard to judge whether experience is important or not. If the absolute value of delta or the value after square is relatively large, we can choose this empirical learning. This sounds reasonable, but the author found that it often makes Delta's relatively small experience rank last and can never be accessed. However, because the agent is constantly learning, the agent may find that this relatively small delta is actually very useful after several rounds of learning. However, at this time, because the sampling method of the experience pool is to select the largest Delta for learning, it is difficult to change this potentially important δ.

There are two new schemes designed by the author

- proportional prioritization $p_i = |\delta_i| + \epsilon$
- rank-based prioritization $p_i = \frac{1}{rank(i)}$

Because proportional prioritization uses random numbers, it is not a very robust scheme. Therefore, most priority experience replay implementations choose rank based prioritization. However, the sampling probabilities of both are based on

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

However, if the prioritized experience replay is used, the distribution of samples will be changed, which may lead the convergence of our model to different values. Therefore, the author proposed a concept called importance sampling and using a new weight to achieve it.

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

In the weight, β is a super parameter used to determine how much to offset the influence of prioritized experience replay on the convergence results. N represents the number of samples sampled. When beta is equal to 1, the product of weight and sampling probability is 1, which is equivalent to completely remove the function of prioritized experience replay.

Therefore, the loss function of prioritized experience replay can be expressed in the following form

$$\nabla_{\theta_i} L\left(\theta_i\right) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta_i\right) - Q\left(s, a; \theta_i\right)\right) \nabla_{\theta_i} Q\left(s, a; \theta_i\right)\right]$$

$$\nabla_\theta L(\theta) = w\delta\nabla_\theta Q(s, a)$$

4.3 Code implementation of prioritized experience replay

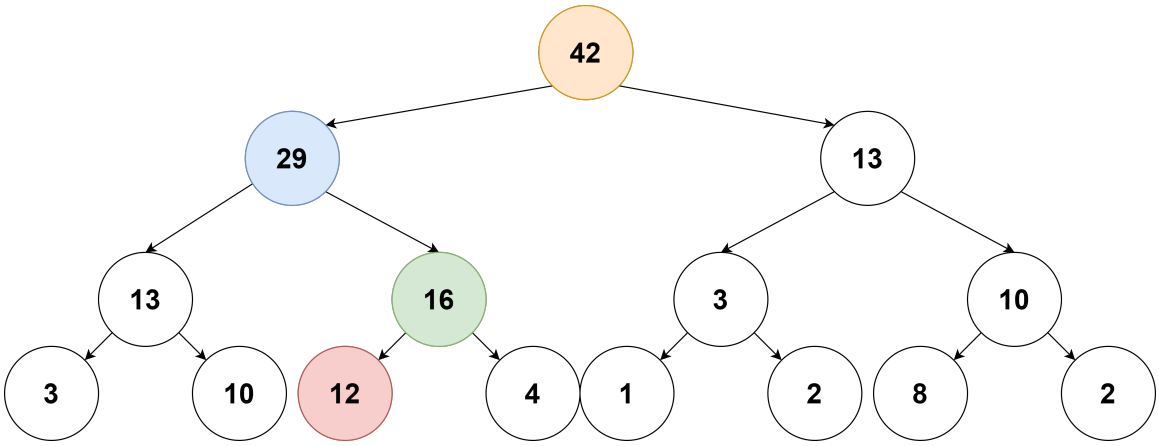
Code implementation referenced Mo fan's GitHub repo and tutorial

https://morvanzhou.github.io/tutorials/machine-learning/reinforcement-learning/4-6-prioritized-replay/

https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow/tree/master/contents/5.2_Prioritized_Replay_DQN

When we specific to the code implementation, a special data structure called sum tree is used. The parent node of each layer is the sum of its child nodes. At this time, the sampling process divides the value of the root node of the tree by the number of samples, which is divided into sampling number intervals, and then randomly selects a value in each interval. According to this value, search from the root node in sum tree until the value of the leaf node closest to the random value is found as the value to be sampled.

For example, if you want to divide 42 into six sampling intervals, and the length of the interval is [0-7], [7-14], [14-21], [21-28], [28-35], [35-42], if you want to randomly select 24 in the interval [21-28], it is found that 24 is smaller than the value of child node 29, so move to the node of 29, and then find that 24 is larger than child node 13, so go to the node of 16. At this time, the value to be searched becomes 24-13 = 11. It is found that 11 is smaller than child node 12, so we go to node 12. Because this node is a leaf node, 12 is the sampling index number closest to 24. Therefore, the data accessed from this node is obtained as a sample of this interval range.



Based on this data structure, the code is implemented as follows

First, define a tree structure and use an array to store the nodes of the tree.

```
class SumTree(object):
    data_pointer = 0

    def __init__(self, capacity):
        self.capacity = capacity # for all priority values
        self.tree = np.zeros(2 * capacity - 1)
```

Then, the initialization operation sets the size of the tree and initializes the nodes in the tree

```
    def __init__(self, capacity):
        self.capacity = capacity # for all priority values
        self.tree = np.zeros(2 * capacity - 1)
```

The add operation is used to update the sum of the values of the two child nodes maintained by the internal node

```
    def add(self, p, data):
        tree_idx = self.data_pointer + self.capacity - 1
        self.data[self.data_pointer] = data # update data_frame
        self.update(tree_idx, p) # update tree_frame

        self.data_pointer += 1
        if self.data_pointer >= self.capacity: # replace when exceed the capacity
            self.data_pointer = 0
```

The update operation is similar to the push down operation of the heap. It pushes the value of the newly updated tree node to its parent node to the root node to ensure that all nodes keep the data up-to-date

```
    def update(self, tree_idx, p):
        change = p - self.tree[tree_idx]
        self.tree[tree_idx] = p
        # then propagate the change through tree
        while tree_idx != 0:  # this method is faster than the recursive loop in
            ↪ the reference code
            tree_idx = (tree_idx - 1) // 2
            self.tree[tree_idx] += change
```

Finally, the most important function is the get_leaf operation, which is the same as the process we described above

```
    def get_leaf(self, v):
        parent_idx = 0
        while True:    # the while loop is faster than the method in the reference
            ↪ code
            cl_idx = 2 * parent_idx + 1      # this leaf's left and right kids
            cr_idx = cl_idx + 1
            if cl_idx >= len(self.tree):     # reach bottom, end search
                leaf_idx = parent_idx
                break
            else:      # downward search, always search for a higher priority node
                if v <= self.tree[cl_idx]:
                    parent_idx = cl_idx
                else:
                    v -= self.tree[cl_idx]
                    parent_idx = cr_idx

        data_idx = leaf_idx - self.capacity + 1
```

```
        return leaf_idx, self.tree[leaf_idx], self.data[data_idx]
```

In addition to replacing the original memory data structure, we also need to replace the memory sampling code. First, divide the interval to the number of values to be sampled, then use the for loop to randomly select a value from these n intervals, and find the most appropriate leaf node through sum tree to update the sampled value.

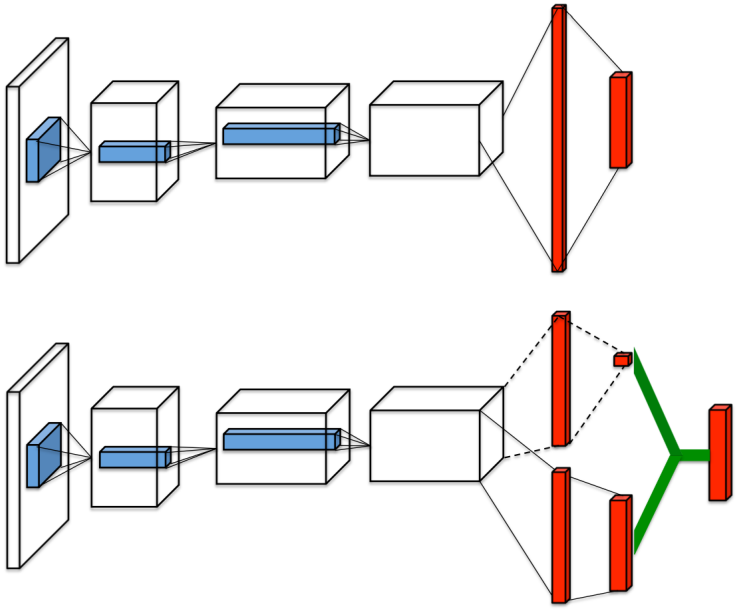
```
    def sample(self, n):
        ...
        pri_seg = self.tree.total_p / n
        self.beta = np.min([1., self.beta + self.beta_increment_per_sampling])
        min_prob = self.tree.get_min() / self.tree.total_p

        for i in range(n):
            a, b = pri_seg * i, pri_seg * (i + 1)
            v = np.random.uniform(a, b)
            idx, p, data = self.tree.get_leaf(v)
            prob = p / self.tree.total_p
            b_idx[i] = idx
            b_states[i] = data[0][0][0:4]
            b_actions[i] = data[1]
            b_rewards[i] = data[2]
            b_next[i] = data[0][0][1:]
            b_dones[i] = data[3]
            ISWeights[i, 0] = np.power(self.__capacity * prob, -self.beta)
        ...
```

Finally, because we also introduced the concept of weight, we need to introduce weight when calculating the loss function

```
    loss = (ISWeights * F.smooth_l1_loss(values, expected, reduction = 'none')).
        ↪ mean()
```

5 Dueling-DQN: An advanced model for policy training

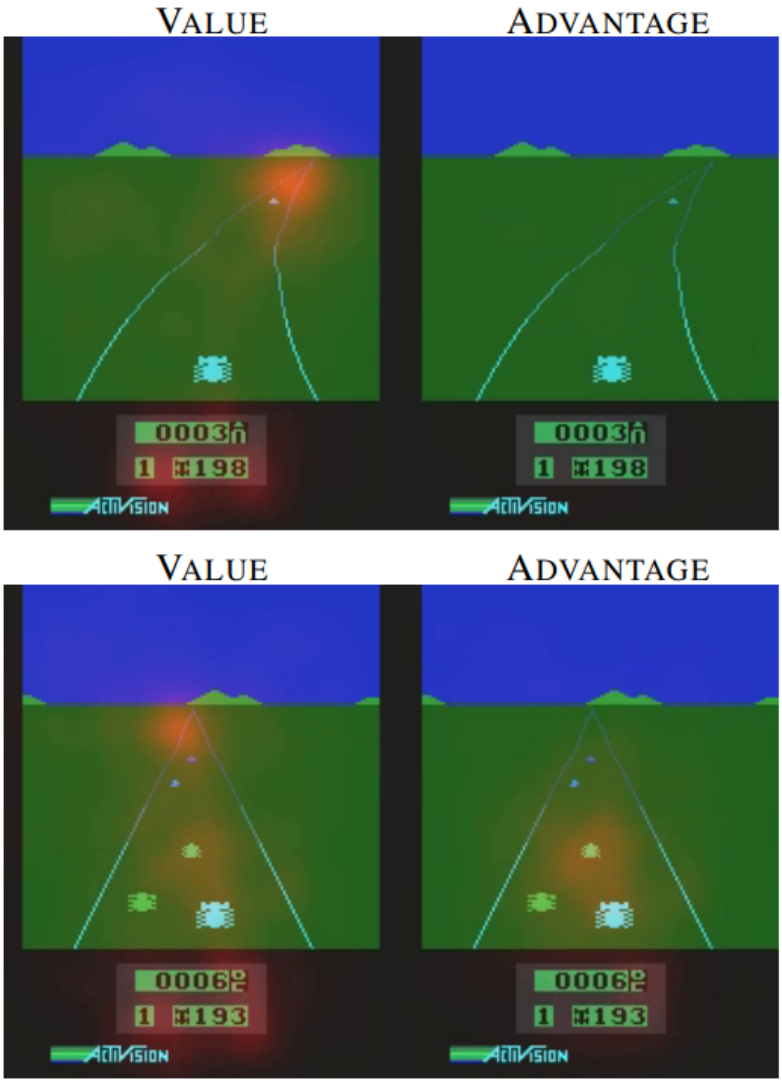
5.1 Structure of Dueling-DQN



Structure on the top is the basic structure of DQN with four convolotion layers and two full connection layers implemented by paper Playing Atari with Deep Reinforcement Learnin, which is the opening work of DQN.

Different from the original model, Dueling DQN changed the model by adding another pair of full connection pairs at the tail of convolutional layer. Two pairs of full connection layers are responsible for different optimization jobs: one is trying to find the optimal state the game might move to while the other one is responsible for evaluating the advatages of taking different actions from the current state. The latter is actually doing the same job as the full connection layer in the original DQN model while the former is new in Dueling Network. We then combine these two parts of outputs by padding the smaller output size produced by optimal state predictor to the same size produced by action advantage predictor whose size is the same as the dimention of actions to control the game.

Actually, the author who created Dueling-DQN had provided an appropriate picture to show the importance of estimating the optimal state.



The game Atari Enduro is to drive a car on the road without crashing with the car coming from behind and the aim is to drive as far as possible. The annotation of "value" is the same meaning as "state" we mentioned above because a state can be evaluated by its value function. To simplify the name of calling "the predictor of optimal state", "the evaluator of the advantage of diffenrent actions", we call them "value stream" and "advantage stream".

What we can see from the picture above is 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.

Having a long-term vision is extremly useful for agent to reach a higher score because the aim is to go as far as possible and it is the same as the game Breakout we play because although the most important thing is to avoid the ball falling off the board, the main target of the game is to reach a higher score by hitting all the bricks.

We can make a bold conclusion that when facing a decision process, to gain the long term

reward, the agent who make the decision need to have the long-term vision. Which is the job the value stream is doing.

5.2 The formal derivation of Dueling-DQN

Stata value function and q value function behaving according to stochastic policyπ can be defined as follows:

$$Q^{\pi}(s,a) = \mathbb{E}\left[R_t \mid s_t = s, a_t = a, \pi\right], \text{ and}$$
$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi(s)}\left[Q^{\pi}(s,a)\right].$$

We know that q value function shows the reward of choosing specific action under the current state and state value function calculates the exspectation of q value function, showing the possible future reward the state might get. So it's obvious for us to define the expression bellow:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

Where function A is what we called as the "advantage stream". We can directly get from this expression that function A shows the bias between the expectation of rewards(state value function) and the actual reward when action was taken under the guide of policy.

According to Bellman optimal equation, only when the difference between q value function and state value function is close to zero under the same policy, can we say an optimal policy is found.

$$A(s,a^*) = 0$$

$$a^* = \arg\max_{a\in\mathcal{A}} Q(s,a)$$

So we rewrite the equation as the form below:

$$Q^{\pi}(s,a) = V^{\pi}(s) + A^{\pi}(s,a)$$

where Q function is the output of model providing all the possible q values under all the policies(the model itself if the collection of policies) and let loss function to use it.

the orinal model only have one item in the equation, but Dueling-DQN introduced two: V function and A function, which can lead to a possible situation that the value of V function had been trained close to zero, so the value of A function is nealy the same as the value of Q function.

Although in this situation, the loss function can still be calculated properly and can perform gradient descent as usual, the V function has lost its use and what we introduced into the new model is actually V function(or we can call"value stream).

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s,a,r,s'}\left[\left(y_i^{DQN} - Q\left(s,a;\theta_i\right)\right)\nabla_{\theta_i} Q\left(s,a;\theta_i\right)\right]$$

How to avoid it? Remember that we expect the value of A function is zero and we can

achieve that by finding the max value of Q value function. So we rewrite the equation as below:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \max_{a' \in |\mathcal{A}|} A(s,a';\theta,\alpha))$$

In the situation we want to avoid, Q value function and A value function have the same value, so findint the maximum Q function is the same as finding the maximum A function. So we can see in the parentheses, we replace Q function as A function.

The author also provide another form to achieve the same goal, which is easier to perform in the code

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

We use this form instead in our code.

5.3 Implementation of Dueling-DQN

Actually there are only a few changes we need to perform on the original DQN model. The value stream only output the best state, so the output dimension is 1 and the advantage stream output different bias between reward under actual action and the expectation of rewards.

So we replace the original full connection layer with two pairs of newly defined layers

```
1    self.fc1_adv = nn.Linear(in_features=7*7*64, out_features=512)
2    self.fc1_val = nn.Linear(in_features=7*7*64, out_features=512)
3
4    self.fc2_adv = nn.Linear(in_features=512, out_features=num_actions)
5    self.fc2_val = nn.Linear(in_features=512, out_features=1)
```

In the forward process, because the output size must be the same as action dimension, so we expand the output size value stream to the same as the size of the output of advantage stream. And we combine these two outputs according to the expression

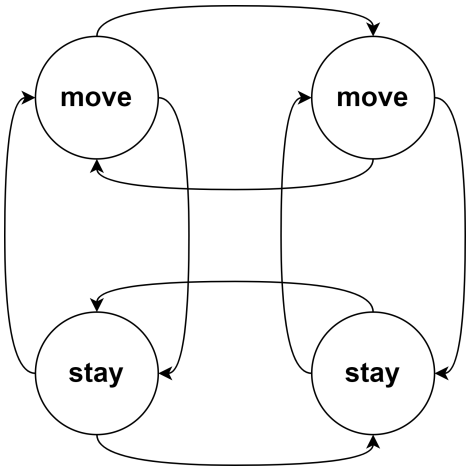
```
1    self.fc1_adv = nn.Linear(in_features=7*7*64, out_features=512)
2    self.fc1_val = nn.Linear(in_features=7*7*64, out_features=512)
3
4    self.fc2_adv = nn.Linear(in_features=512, out_features=num_actions)
5    self.fc2_val = nn.Linear(in_features=512, out_features=1)
```

Moreover, we initialize Dueling-DQN model instead of original DQN model

```
1    self.__policy = Dueling_DQN(action_dim, device).to(device)
2    self.__target = Dueling_DQN(action_dim, device).to(device)
```

6 Stablelize the movement of the paddle

In order to stabilize the movement of the pad, we propose a method based on state machine. Although it is very difficult to converge in the end, it does stabilize the pad.




```
    if done:
        reward -= 100
    if action == last_action and last_action == last_last_action:
        if action == 0:
            reward += 1
        reward += 1
    if action == last_action and last_action != last_last_action:
        reward += 0.5
    if action != last_action and last_action == last_last_action:
        reward -= 1
    if action != last_action and last_action != last_last_action:
        reward -= 2
```

The working principle of this state machine is that the agent will execute the same action only after repeating the same action twice.

For example, if the last action is move, and the last action is move, then if this action is also move, it conforms to the operation mode of the state machine, so it is encouraged to a certain extent. Because we want the paddle to be as stable as possible, if the first two actions are move and the action selected by the agent this time is also move, then reward is set to 1;

If the previous two times are stay, and the agent selection is also stay this time, we encourage the agent to select the stay operation, so as to ensure that the paddle stays in the position reward is set to 2.

If the last two actions are the same, but the last operation is different from the last two, it means that the agent has explored a possible effective scheme, so the last two actions will be the same. Therefore, we will encourage this behavior by setting rewad to 0.5.

If the previous two actions are the same and the current operation is different, this operation may not be the best choice. We want the agent to continue the previous action according to a certain inertia. Therefore, in this case, we will give the agent some punishment. For example, if the first two actions are stay and the current action is move, the movement of the pad will be stable if you continue to stay. Therefore, in order to punish such actions, we set reward to - 1.

However, there may be a more chaotic situation, that is, the three actions are completely different. In this case, the movement of the pad must be chaotic. In this case, we set the maximum penalty, that is, the reward is -2.

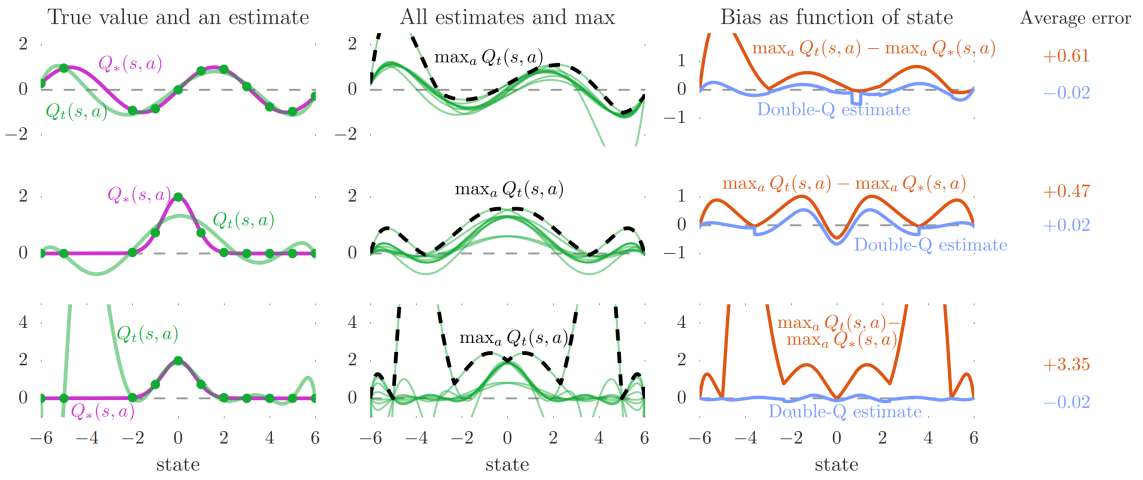
The rationality of such a scheme is that if the actions obtained by the agent from the previous two frames are left or right, the agent will quickly turn left or right under the drive of the state-machine until it moves to the target position, and if the agent think it needs to stop(twice), then the agent will stop immediately under the drive of the state-machine.

This state machine model actually refers to the branch prediction model of CPU design. Based on this branch prediction model, the prediction accuracy of CPU for cycle can be as high as 99%. We hope that applying this state machine to agent training can better improve the prediction ability and performance of agent.

7 Introducing Double-DQN into our model

7.1 Why use Double-DQN?

When reading the paper, I found that the model which makes the breakout score reach the highest is usually the combination of ddqn and dueling dqn. Because dueling dqn and ddqn are different improvements: dueling-dqn improves the model architecture, while ddqn improves the calculation method of loss function. Therefore, based on dueling dqn, the loss function can be modified to realize ddqn.



As can be seen from this figure, the optimal Q value function is closest to the value of the real Q value function (obviously, the purpose of estimating the optimal Q value function is to fit the Q value function). And it can be seen from the middle figure that if the maximum Q value function is directly estimated, it is difficult for this Q value function to fit the real Q value function. The last figure shows the deviation between the optimal Q value function and the real Q value function and the max Q value function The deviation from the real Q value function is found to be much greater than the former, up to 160 times. Therefore, we can see

that the effect of using optimal Q value function is better.

In the traditional dqn training, including dueling dqn, the method of calculating target is

$$Y_t^{\mathrm{DQN}} \equiv R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a; \boldsymbol{\theta}_t^-\right)$$

Obviously, the Max used in the formulamax$_a\, Q\left(S_{t+1}, a; \boldsymbol{\theta}_t^-\right)$ will cause great deviation. How to fit the optimal Q value function? This is what ddqn had done. The target calculation method proposed by ddqn is modified to the following form (note that the operator used here is argmax operator instead of Max operator.):

$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q\left(S_{t+1}, \operatorname*{argmax}_a Q\left(S_{t+1}, a; \boldsymbol{\theta}_t\right); \boldsymbol{\theta}_t\right)$$

Two Q models are used here. The internal q model is the training model, and the external is the locked target q model. The way to calculate the target in ddqn is to find the action that maximizes the current Q value from the trained q model, and input this action into the locked target q model to obtain the final fitted Q value function.

The reason for this is that the target q model we finally use is considered to be optimal. Therefore, finding the appropriate action input target q model can obtain the theoretically optimal fitting function.

7.2 Implementation of DDQN

In the original training process, the target (expected) is calculated as follows

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

Now the target of ddqn is calculated as follows:

```
values = self.__policy(state_batch.float()).gather(1, action_batch)
target_value = self.__target(next_batch.float()).detach()
max_action = self.__policy(next_batch.float()).max(1).unsqueeze(0)
expected = (self.__gamma * target_value.gather(1, max_action)) * (1. -
    ↪ done_batch) + reward_batch
```

First, find the Q value of all samples in batch in target q model and store it in target_Value variable. Then find the maximum value from the output of the currently trained q model as the optimal action.

Because the input to the model is a batch, we need to use unsqueeze() to upgrade the dimension of the array. For example, the original array is [0,1,2]. Because the index type received by the gather function in pytorch is [[0,1,2]], we need to upgrade the dimension of the outermost dimension and save it as max_action。

Finally, use target_value.gather(1, max_action) according to the index given by max_action to index the the corresponding optimal Q value function in the target_value array.Then the

expected value is found.

8 The experimental results

8.1 The models we trained

- Basic implement of DQN
- Using Dueling-DQN
- Using Double-DQN
- Using DDQN+Duel
- Using DDQN+Duel+Prioritised Experience Replay
- Using DDQN+Duel with eps setting to 0(no greedy)
- Using DDQN+Duel+PR with state machine

Both of the models achieve good result especially using DDQN+Duel+PR with state machine and using DDQN+Duel with eps setting to 0(no greedy)

8.2 Basic implement of DQN

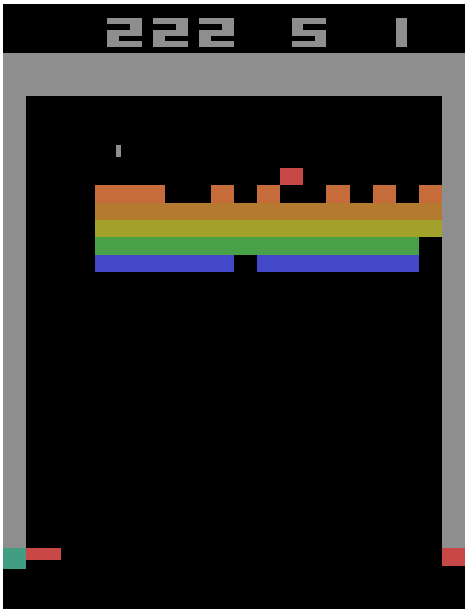
In the beginning, We trained the original implementation to use it as a basic control group. We trained it 50 million times to make sure it was trained as well as possible. Because it was trained locally and the training speed was not fast, the whole training process took us more than a week.

In the initial stage of training, the reward obtained by the agent is very low, generally no more than ten. However, after about 800000 training, its reward began to increase rapidly, from single digits to ten digits or even hundreds. But this is only the beginning.

In the following training, the agent is still improving on the whole, but it does not improve the score fast. Although it was trained for a large number of times, according to the training results above, it is still not convergent. If we observe the training results more carefully, we will find that there are still large oscillations and uncertainties in each reward. For example, it gets a reward of more than three hundred in one time, but it may get less than one hundred in the next time. This problem can occur even at the end of the training process.

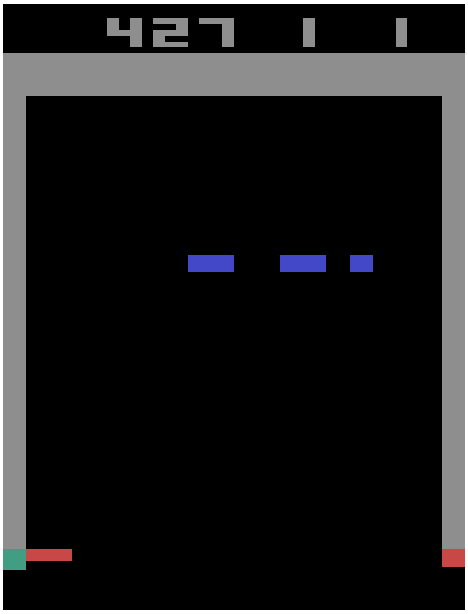
Based on the above training results, we infer the reasons for these situations:

At the beginning of training, because the agent always can't receive the ball after hitting a few bricks, the reward is also very low. However, after training about 800000 times, the agent successfully hit the ball into the inside of the bricks. The ball keeps hitting bricks inside, and it doesn't need the agent to catch the ball to get a lot of points. Just like the figure below.



However, when the number of bricks becomes less, it becomes very difficult to hit the remaining bricks. Because the agent have many ways to hit the ball, but they may not be able to hit the bricks. The agent can only try to hit the remaining bricks by exploring and constantly changing the trajectory of the ball. The problem is that the amount of training required to get more bricks increases almost exponentially. So although we trained 50 million times, the training results still don't converge, and the agent can't hit all the bricks.

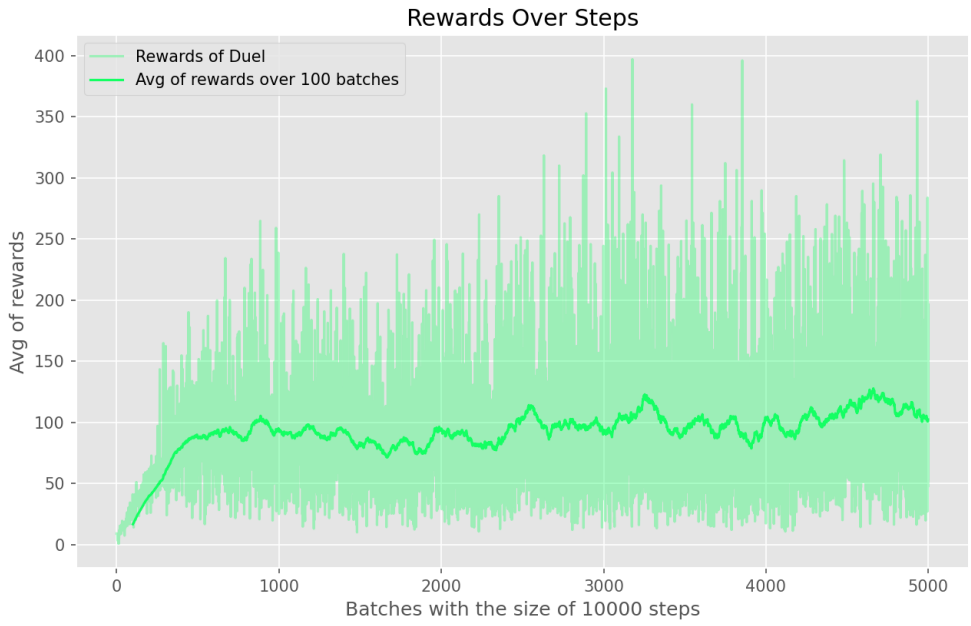
In the experiment, we found that the scores obtained by putting the trained model into the simulation environment are not close to the average scores, or even very different. We thought we should get the highest score at the end of the training, so we tried the last 500 models, and the highest score was 427. This is only a short distance from hitting all the bricks.



Through observation, we also found that the score of the upper brick is higher than that of the lower brick, so the agent is more inclined to hit the upper brick, which is why the remaining bricks in the above results are below.

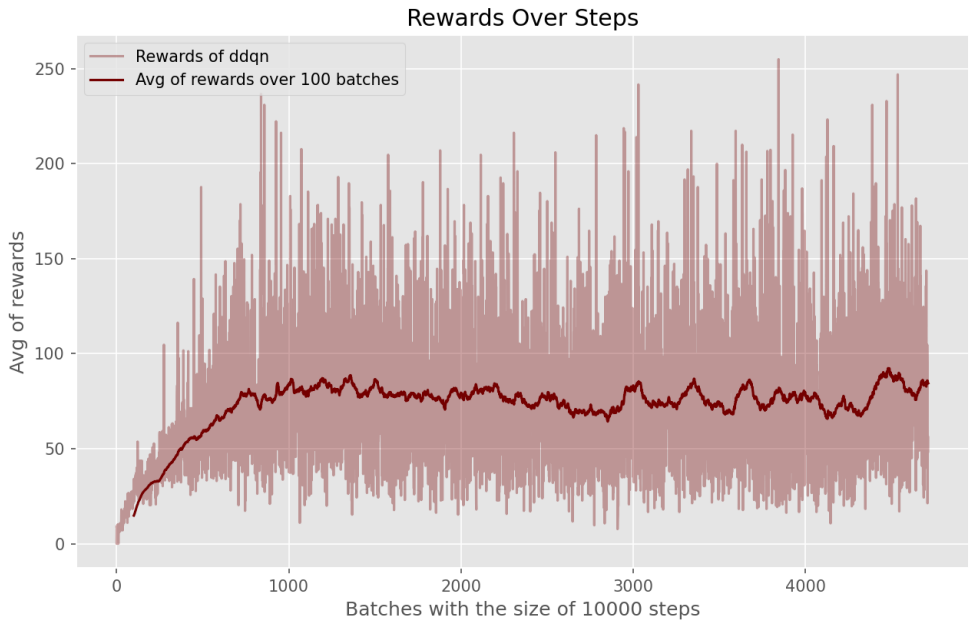
8.3 Using Dueling-DQN

The convergence rate and average reward of Dueling-DQN alone are not excellent.



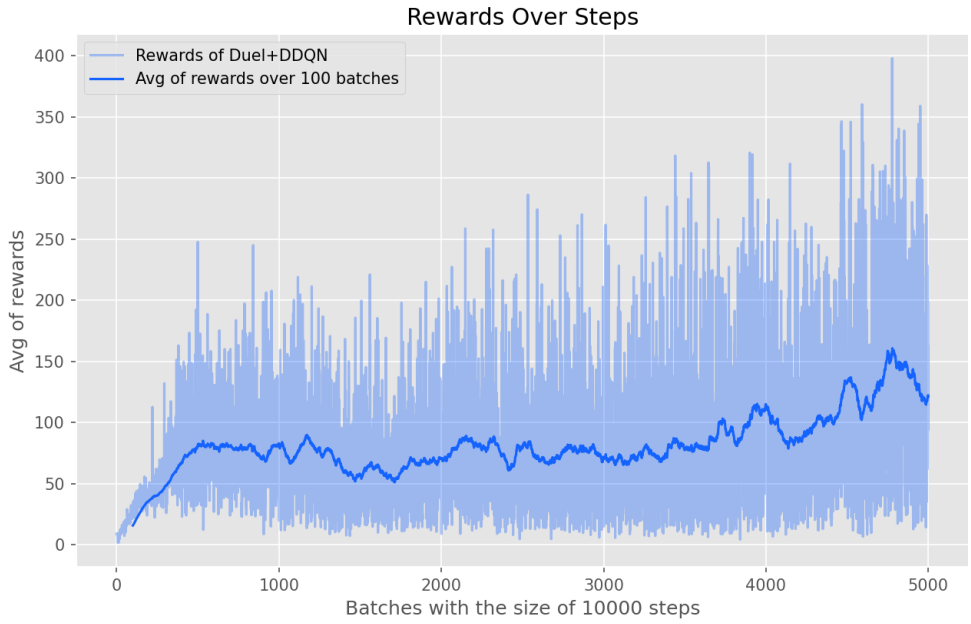
8.4 Using Double-DQN

The convergence rate and average reward of Double-DQN alone are not excellent.



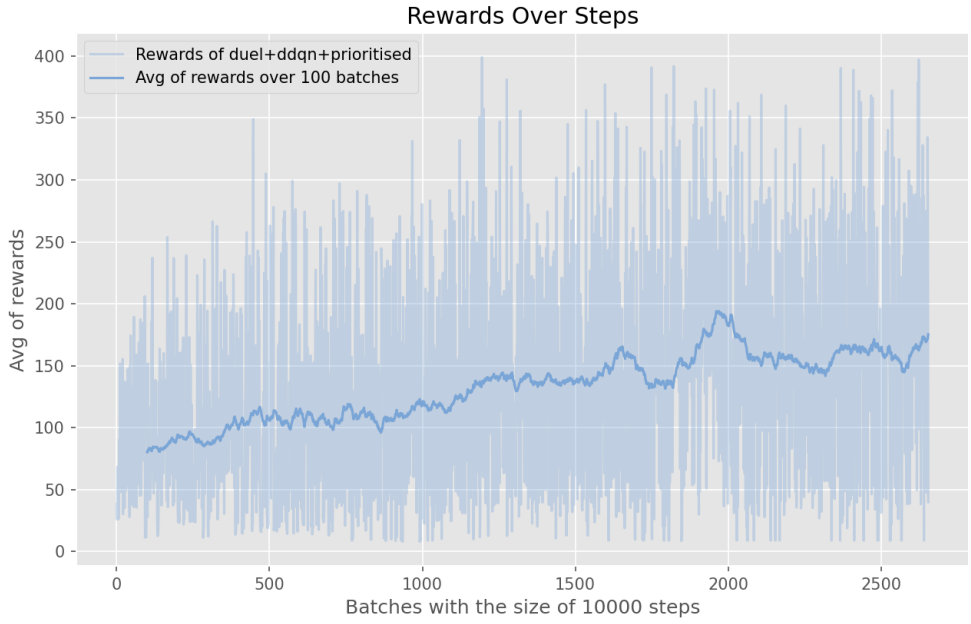
8.5 Using DDQN+Duel

The convergence rate and average reward of Double-DQN and Dueling-DQN alone are not excellent.

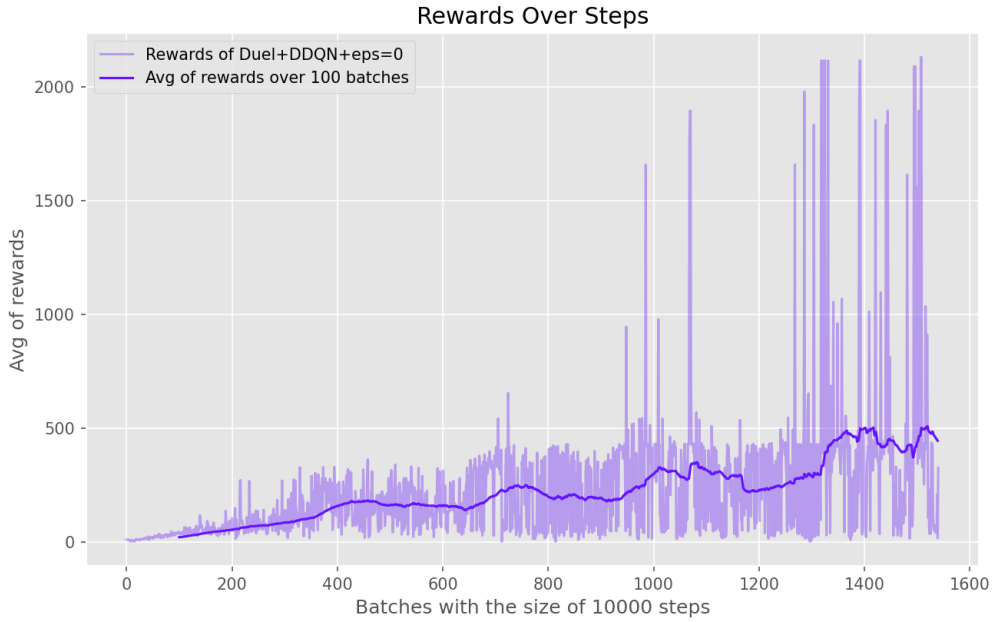


8.6 Using DDQN+Duel+Prioritised Experience Replay

When combining the Double-DQN, Dueling-DQN and Prioritised Experience Replay togegher, the convergence rate is a bit larger and the average reward grows.



8.7 Using DDQN+Duel with eps setting to 0(no greedy)

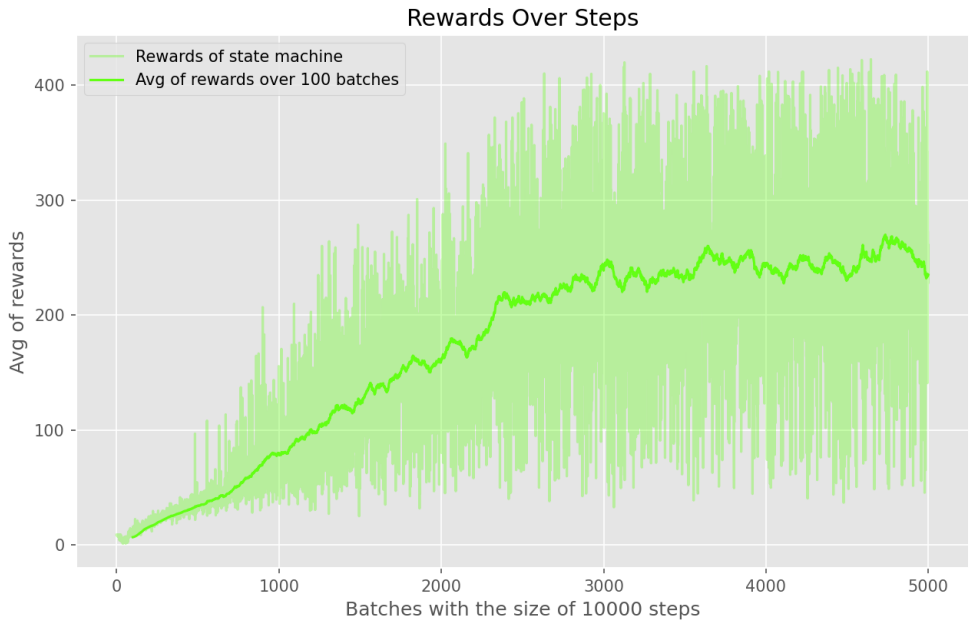


We found a way to make the model converge super fast: set EPS to 0.

After the agent slowly uses greedy strategy to train for a period of time to find the possible direction, eps is set to 0, so that the agent will firmly explore and learn along the previously found path, so that after only 10 million iterations, the model can steadily score 400 points.

Later, there was even a surprisingly high score of 2,000 points (we're not sure where this score came from), perhaps because the model eventually learned a life-saving trick of saving the ball first, rather than knocking down all the bricks. In this case, we can punish the agent not to focus on catching the ball. But because there is no time left for us, we can't have another try.

8.8 Using DDQN+Duel+PR with state machine



The final model is our most satisfing model. We creatively introduce the state-machine to maintain the stability of the paddle. The results show that this choice makes the movement of the pad very stable and decisive.

Because the movement of the paddle is very stable, it is not easy to lose the ball, which indirectly ensures that the convergence speed is very fast and the average score is very high

You can notice that the average score curve of this model is continuously rising, while other models will tend to be stable after rising to a certain extent. Therefore, this is a model full of potential. If we are given more time, we can train this model better.

9 Reference

1. Ziyu Wang and Tom Schaul and Matteo Hessel and Hado van Hasselt and Marc Lanctot and Nando de Freitas,**Dueling Network Architectures for Deep Reinforcement Learning**
2. Shervin Halat and Mohammad Mehdi Ebadzadeh, **Modified Double DQN addressing stability**
3. Tom Schaul and John Quan and Ioannis Antonoglou and David Silver, **Prioritized Experience Replay**
4. https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow
5. https://github.com/noctrog/Breakout_DDQN