AI Project Report

Yuzhen Huang
519030910281
hyz1017@sjtu.edu.cn

Peng Tang
517020910038
tttppp@sjtu.edu.cn

Abstract

In this project, we use reinforcement learning to play a game named Snakes 3v3. Specially, we combine three popular DQN method together, including Double-DQN, Dueling-DQN and Prioritized-DQN. Meanwhile, we utilize some methods to improve the performance of our model, including vision transformation and ResNet. With these methods, our model has huge advantages over basic DQN model.

1 Introduction

This project is to play Snake 3V3, which is a multi-agent version of the traditional game. Our model is based on deep Q learning. There are many improved versions of DQN, such as Double-DQN, Dueling-DQN and Prioritized-replay-DQN. By referring to a model named Rainbow DQN, we realize that these methods do not conflict.(Hessel et al. [2017]) And to get better performance, we try to combine them together. The implementation details will be discussed in Chapter 2.

2 Algorithm Design

2.1 Pre-processing

In this environment, the type of observation is dictionary. To process it as an image, we need to convert the observation dictionary to an image. Meanwhile, to accelerate the learning process and simplify the neural network, we need to do some layering, cutting and rotation. The steps of pre-processing are shown below.

1. Layering: Assume that we are now control $snake_i$. According to the observation dictionary, we can get its observation obs_i. First, we need to create a zero tensor named $full\_map$ with shape of $(11, height, width)$, which has 11 layers. We use one-hot encoding to represent items. The information contained by each layer are:
   i. layer0: positions of all snakes.
   ii. layer1: head position of $snake_i$.
   iii. layer2: head positions of $snake_i$'s teammates.
   iv. layer3: head positions of $snake_i$'s enemies.
   v. layer4: positions of beams.
   vi. layer5: the complete positions of $snake_i$.
   vii. layer6-7: the complete positions of two teammates.
   viii. layer8-10: the complete positions of three enemies.
2. Cutting: In these game, when the snake head crosses the boundary, it can cross to a symmetrical position. To make the network to learn better, we cut $full\_map$ to a $view\_map$ with shape of $(11, 10, 10)$, which of center is the controlled snake's head and contains the information within 5 grids.

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

3. Rotation: When an illegal direction is entered, a legal direction is randomly selected. To avoid such situation, we need to rotate the image to keep the moving direction of the controlled snake is up all the time. Then we can limit the size of action space to 3. After the network choose an action, we use inverse transformation to get the real action and return it.

2.2 Network Design

As shown in Figure 1, our model includes six parts. We adopt the idea of image processing and take the observation of each snake as input. Then by using convolution network and ResNet block, we get the feature vector of the observation. Each ResNet block contain two convolution blocks, which includes convolution layer, normalization layer and activation layer. After fully connected layer process the the feature, Dueling-DQN part will predict the value of each action and output the result.

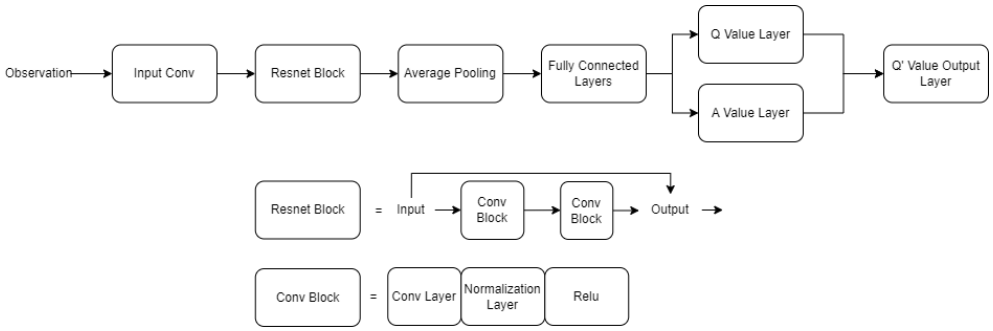

Figure 1: Network design

2.3 Basic Model

In Q-Learning algorithm, we need an array to store all Q values of corresponding states and actions. However, in this game, the number of states is too large to store. Thus, we choose DQN algorithm to replace the Q-Learning algorithm. In DQN, we can generate Q values through neural network.

In our model, we implement DQN by constructing two neural networks, target network and evolution network. The two networks are almost same except parameters. What's more, we need a replay buffer to store some transitions for training.

When training the evolution network, we firstly get a batch of transitions from the replay buffer randomly. Then we input the states of the batch to evolution network and get corresponding $Q(s, a)$ of the batch. To get the $Q^{Target}(s, a)$, we input the next-states of the batch to target network and get the $\max_{a'} Q(s', a')$. We can compute the $Q^{Target}(s, a)$ according to Q-learning algorithm as equation 1.

$$Q^{Target}(s, a) = r + \gamma \max_{a'} Q(s', a') \tag{1}$$

Then we can compute the loss function for evolution network according to equation 2.

$$Loss(w) = E[(Q^{Target}(s, a, w) - Q(s, a, w))^2] \tag{2}$$

With this loss function, we can train the evolution network. And we update the target network with the parameters of evolution network after some episodes. To illustrate the main idea of DQN in our model more clearly, we use Figure 2 to show it.

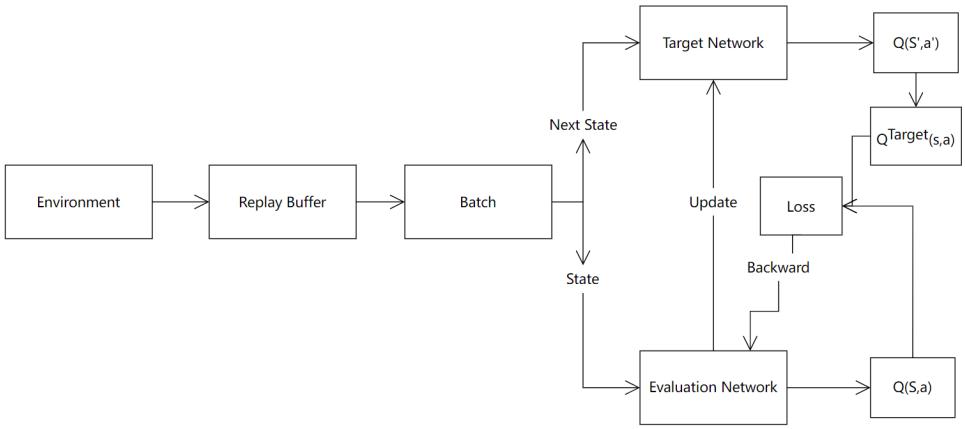

Figure 2: DQN training process

Therefore, we have our DQN algorithm as Algorithm 1.

Algorithm 1 DQN Algorithm

Initialize replay buffer D with size N, target network T and evolution network O with input dims and output dims and random weights.
for $episode = 1$ to $Max\_Episode$ **do**
  Get initial state s from game
  while $True$ **do**
    for $i = 1$ to $AgentNumber$ **do**
      Get observation b_i from s according to agent index i.
      Get feature $f_i = \phi(b_i)$, ϕ is the pre-processing function
      With probability ε to select a random virtual action a_i^v
      Otherwise select $a_i^v = \arg\max_a Q(f_i, a)$, $Q(f_i) = T(f_i)$
      Get real action $a_i^r = \varphi(a_i^v)$, φ is a mapping function which can map the network output to the real action of the agent
    end for
    Get next step information from game through actions a^r,$(s', r, done) = game(a^r)$, s' is next state and r is reward
    Get observation b_i' from s'
    Get feature $f_i' = \phi(b_i')$
    Check every snake whether it died after doing corresponding action by comparing the length of the snake before and after the action and record it in $Done$.
    if D is not filled **then**
      Store transition $(f_i, a_i^v, r_i, f_i', Done_i)$ in D for every snake i
    else
      Remove the oldest transition from D, then store transition $(f_i, a_i^v, r_i, f_i', Done_i)$ in D
    end if
    if D is filled **then**
      randomly get a mini-batch of transitions from D
      for each transition j in the mini-batch **do**
        $Q(f_j, a_j^v) = O(f_j, a_j^v)$
        $Q^{Target}(f_j, a_j^v) = r_j + (1 - Done_j)\gamma \max_{a^v} Q(f_j', a^v)$, $Q(f_j') = T(f_j')$
      end for
      Back-propagate with Loss function according to equation 2
      Update the target network T with evolution network O every fixed steps.
    end if
    $s = s'$
    if $done$ **then**
      break
    end if

34: **end while**
35: **end for**

2.4 Double DQN

The DQN algorithm easily over-estimate action values and these over-estimations usually have a bad impact on performance. According to the paper *Deep Reinforcement Learning with Double Q-learning*(van Hasselt et al. [2015]), we use Double-DQN algorithm to reduce these over-estimations and improve the performance of our agent.

The main difference between DQN and Double-DQN is the method of computing $Q^{Target}(s,a)$. The equation 3 shows the $Q^{Target}(s,a)$ computing of DQN and the equation 4 shows the $Q^{Target}(s,a)$ computing of Double-DQN.

$$Q^{Target}(s,a,w) = r + \gamma \max_{a'} Q(s',a',w) \tag{3}$$

$$Q^{Target}(s,a,w) = r + \gamma Q(s', \arg\max_{a'} Q(s',a',w'),w) \tag{4}$$

Where w represents the parameters of target network and w' represents the parameters of evolution network. From these equations, we can find that Double-DQN using the evolution network to get the action for s' rather than target network. In this way, Double-DQN use two networks' information to compute Q values.

Therefore, we can modify line 24 of Algorithm 1 to get Algorithm 2 easily.

Algorithm 2 Double-DQN Algorithm

1: ...
21: **if** D is filled **then**
22: randomly get a mini-batch of transitions from D
23: **for** each transition j in the mini-batch **do**
24: $Q(f_j, a_j^v) = O(f_j, a_j^v)$
25: $a_j^{v'} = \arg\max_{a_j^{v'}} Q(f'_j, a_j^{v'}), Q(f'_j) = O(f'_j)$
26: $Q^{Target}(f_j, a_j^v) = r_j + (1 - Done_j)\gamma Q(f'_j, a_j^{v'}), Q(f'_j) = T(f'_j)$
27: **end for**
28: Back-propagate with Loss function according to equation 2
29: **end if**
30: ...

2.5 Priority Replay Buffer

Replay buffer is used to store experiences from the past. In our previous model, we chose a batch of transitions from replay buffer randomly. However, the importance of different transitions should be different. These transitions with higher prediction loss should be more important and be more likely to be selected. In this way, the model can reach our goal with faster speed. In practice, we give every transition a corresponding priority. The higher the priority of a transition is, the more possible it will be sampled.

According to the paper *Prioritized Experience Replay*(Schaul et al. [2016]), We can define the priority of transitions by using the TD-error, which is the difference between the $Q^{Target}(s,a)$ and $Q(s,a)$. Thus, we define the priority of transition i as equation 5

$$p_i = \begin{cases} \max_{t<i} p_t, & case \quad 1 \\ (|\delta_i| + \epsilon)^\alpha, & case \quad 2 \end{cases} \tag{5}$$

In case 1, transition i is added to the replay buffer for the first time. In this case, because transition i is the newest, it should be important. Thus, we set its priority to maximum so that it can be selected with high probability.

In case 2, $\delta_i = Q^{Target}(s,a) - Q(s,a)$. ϵ is a small positive constant number that prevents the priority of transition i being zero. And $\alpha (0 \leq \alpha \leq 1)$ determines the importance of error. $\alpha = 0$ is the uniform case.

With the priority, we define the probability of sampling transition i as equation 6

$$P(i) = \frac{p_i}{\sum_k p_k} \tag{6}$$

What's more, with probability, we define importance-sampling(IS) weights(equation 7) to correct the bias caused by prioritized replay buffer.

$$w_j = \frac{(N * P(j))^{-\beta}}{\max_j w_j} \tag{7}$$

Where β linearly increase from a initial number β_0 to 1. With IS weights, we redefine loss function as equation 8

$$Loss = E[W * (Q^{Target} - Q)^2] \tag{8}$$

To improve the sampling efficiency, we use a binary tree data structure, sum tree, to store transitions. The value of the leaves in the sum tree is the priority of transitions and the parent's value is the sum of its children. Thus, the value of the root is the total sum of all transitions' priority. Using sum tree, we can sample, insert and update buffer with $O(\log n)$. We can construct sum tree by using an array. In this way, $LeftChild_{index} = 2 * Parent_{index} + 1$ and $RightChild_{index} = LeftChild_{index} + 1$. Therefore, we use a replay buffer D to store all transitions and a sum tree T to store corresponding priority. If the size of D is N, then the size of T is $2N - 1$ because we need N leaves in T. Thus, the corresponding priority of transition i in D is the value of $T[i + N - 1]$.

With above analysis, we can select a transition with Algorithm 3.

Algorithm 3 Get a transition from replay buffer D according to sum tree T

Input: priority number x
Output: transition t, its priority p and its index t_{index} in T

Set t_{index} to be 0
while $2 \times t_{index} + 1$ is smaller than the length of T **do**
  Left child index $l \Leftarrow 2 \times t_{index} + 1$
  Right child index $r \Leftarrow l + 1$
  if $x \leq T[l]$ **then**
    $t_{index} \Leftarrow l$
  else
    $x \Leftarrow x - T[l]$
    $t_{index} \Leftarrow r$
  end if
end while
Compute the index of transition t in D by setting $d_{index} \Leftarrow t_{index} - N + 1$, N is the size of D
$t \Leftarrow D[d_{index}]$
$p \Leftarrow T[t_{index}]$

Using Algorithm 3, we can sample a minibath with Algorithm 4.

Algorithm 4 Get a minibatch from replay buffer D according to sum tree T

Input: batch size k
Output: Batch B, its index B_{index} and its weight W

Initialize empty array B, B_{index}, and W with capacity k
Divide $[0, T[0]]$ into k ranges equally
for $i = 1$ to k **do**
  Randomly choose a number x_i from range i
  Call Algorithm 3 to get transition t_i, its priority p_i and its index $t_{i_{index}}$ by inputting x_i
  Compute probability P_i according to equation 6 with p_i
  Compute weight w_i according to equation 7 with P_i
  Store t_i in B, w_i in W and $t_{i_{index}}$ in B_{index}
end for

In order to update the sum tree when we change a leaf's priority, we define Algorithm 5.

Algorithm 5 Update sum tree T

Input: transition's index t_{index} in T and its new priority p

$c \Leftarrow p - T[t_{index}]$
$T[t_{index}] \Leftarrow p$
while $i \neq 0$ **do**
  $i \Leftarrow \frac{t_{index}-1}{2}$
  $T[t_{index}] \Leftarrow T[t_{index}] + c$
end while

With Algorithm 5, we define Algorithm 6 to store a new transition to the replay buffer D and its priority to the sum tree T.

Algorithm 6 Store a transition to replay buffer D and its priority to sum tree T.

Input: transition t

Set the priority p of transition t to be the max priority in T.
If p is zero, set it to be 1
Compute its index t_{index} in T according to the current pointer d_{index} of D. $t_{index} \Leftarrow d_{index} + N - 1$, N is the size of D.
$D[d_{index}] \Leftarrow t$
Call Algorithm 5 to update T by inputting t_{index} and p
$d_{index} \Leftarrow (d_{index} + 1) \mod N$

2.6 Dueling DQN

In Double-DQN and Prioritized-replay-DQN, we optimize the calculation of target Q value and sampling process respectively. In Dueling-DQN, we aim at optimizing the structure of neural network.(Wang et al. [2016]) Dueling-DQN divides the original Q network into two parts. First part, denoted as $V(S, w)$, is only related with current state. Second part, denoted as $A(S, A, w)$, is related with both current state and actions. And We can represent our Q value as follows:

$$Q(S, A) = V(S, w) + A(S, A, w)$$

This method is more efficient in using the sampled data. For example, when we sample one pair (S, A), we can also adjust $Q(S, A')$ by change $V(S, w)$. In addition, to prevent our model from degenerating into original model, we add some constrains on $A(S, A, w)$ to limit $\sum_a A(S, a, w)$ to 0. We get the final expression:

$$Q(S, A) = V(S, w) + (A(S, A, w) - \frac{1}{C} \sum_{a' \in \mathcal{A}} A(S, a', w))$$

where $\mathcal{A}$ is the action space and C is the size of action space $\mathcal{A}$.

2.7 Final Algorithm

Combining all algorithms defined before, we get our final model as Algorithm 7.

Algorithm 7 Final Algorithm

Initialize replay buffer D with size N, sum tree ST with size $2N - 1$, target dueling network T and evolution dueling network O introduced in **section 2.6** with input dims and output dims and random weights, and small positive number ϵ, exponents α and β and the increasing rate η of β
for $episode = 1$ to $Max\_Episode$ **do**
  Get initial state s from game
  while $True$ **do**
    for $i = 1$ to $AgentNumber$ **do**
      Get observation b_i from s according to agent index i.
      Get feature $f_i = \phi(b_i)$, ϕ is the pre-processing function
      With probability ε to select a random virtual action a_i^v

Otherwise select $a_i^v = \arg\max_a Q(f_i, a)$, $Q(f_i) = T(f_i)$
Get real action $a_i^r = \varphi(a_i^v)$, φ is a mapping function which can map the network output to the real action of the agent
end for
Get next step information from game through actions a^r,$(s', r, done) = game(a^r)$, s' is next state and r is reward
Get observation b_i' from s'
Get feature $f_i' = \phi(b_i')$
Check every snake whether it died after doing corresponding action by comparing the length of the snake before and after the action and record it in $Done$.
Store transition $(f_i, a_i^v, r_i, f_i', Done_i)$ in D for every snake i by calling Algorithm 6
if D is filled **then**
Randomly sample a mini-batch B with size k, its index B_{index} and its weight W from D by calling Algorithm 4
for each transition j in B **do**
$Q(f_j, a_j^v) = O(f_j, a_j^v)$
$a_j^{v'} = \arg\max_{a_j^{v'}} Q(f_j', a_j^{v'})$, $Q(f_j') = O(f_j')$
$Q^{Target}(f_j, a_j^v) = r_j + (1 - Done_j)\gamma Q(f_j', a_j^{v'})$, $Q(f_j') = T(f_j')$
end for
for $j = 1$ to k **do**
Compute the priority p_j of transition j according equation 5
Update sum tree ST by calling Algorithm 5 with inputting $B_{index}[j]$ and p_j
end for
Update β by setting $\beta = \min\{\beta + \eta, 1\}$
Back-propagate with Loss function according to equation 8 and W
Update the target network T with evolution network O every fixed steps.
end if
$s = s'$
if $done$ **then**
break
end if
end while
end for

3 Result

After finishing the model, we run the training process and plot the average length of snake against the number of iterations. After 2000 episodes, we get the following figure.

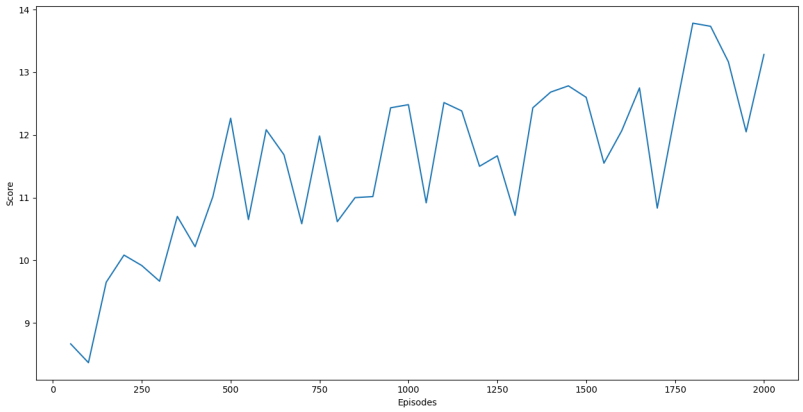

Figure 3: Average length of snake against the number of iterations

According to the figure, we can find that:

1. At the beginning, the score is disturbed and shows an upward trend overall. That is one of the characteristics of deep learning. Thus, we need to optimal our model to get more stable training process.
2. After the number of iterations is large, the score tends to stabilize. And We find that check points with maximum score during training process may not have the best performance in online competition.

4 Conclusion

In this project, our team successfully implement deep Q learning and make many improvements to our base model including Double-DQN, priority replay buffer and Dueling-DQN. Finally, our model achieve satisfactory performance on the game Snake3v3.

5 Roles of The Team

1. Yuzhen Huang: Implementation of basic model, Double-DQN, Dueling-DQN; Report writing.(50%)
2. Peng Tang: Implementation of Priority Replay Buffer; Report writing.(50%)

References

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.