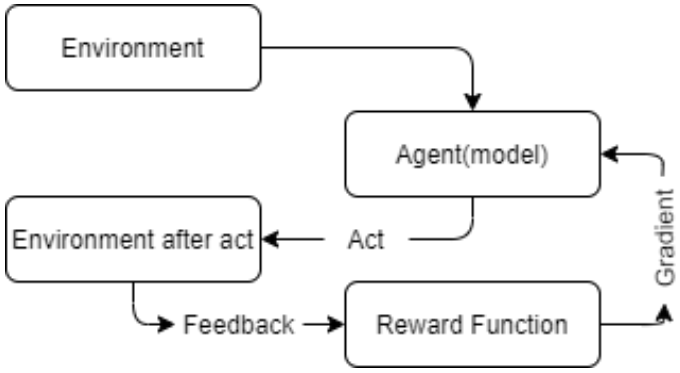
Level 2 Report

Basic Knowledge of Reinforcement Learning

Reinforcement Learning requires 3 main part: actor(agent), environment and reward function. For example, in our level one application:

- `Actor(agent)` : The thing actor need to do is control the bird fly up and down, which can control by touch the screen. To be specific, in the deep learning drived reinforcement learning, the actor is the model we trained, which is used to make the decision touch screen or keep away.
- `Environment` : The environment in the RL flappy bird is the game itself, it will give feedback each step you played. Any touch bewteen bird and tube or ground will cause the fail, any pass of tube will get score.
- `Reward Function` Determine the feedback from environment, return a reward value. In flappy bird, we return negative value when failed, positive value when get score.
    - The setting of reward function can be tricky, the setting can directly influence the performance of model. In some case if we just use a standard reward function which only provide fixed value for each feedback, we may face the issue that the gradient for training always positive or negative. In the other word, when the sampling of each action is not enough, it might sent same expectation to trainer, which can be improved by over sampling or minus a baseline value in the reward function, which makes the total reward of a action not good for the model at this environment to be negative, as a penalty. The technique is known as a simple Policy Gradient.

The structure of the three parties' workflow can be represent as the diagram:



There are two ways to train the agent: On-policy and off-policy

- `On-policy:` The agent learning from evironment and also interacte with environment. (That is what we used u=in flappy bird repo, it cost lots of time on sampling data, get a single sample then update parameter of the model for just one time.
- `Off-policy:` The agent learning from environment(it only looking at the environment, have not do any action on the environment. (Efficieny for training)

Q-Learning

We learning a critic, which determine the quality of the action. In that, we have a state value function $V^{\pi}(s)$, where π is the actor, s is the environment state. State value depend on both environment state and actor, it is possible to be differ when the environment is hte same but parameters in actor has been changed (well-trained actor vs bad actor).

State Value Function Estimation

Monte-Carlo based Approach

The method let actor interact with the environment, then we have states $s_1, s_2, s_3 \dots s_n$, though network(model) V^{π} , we can get cumulated rewards $G_1, G_2, G_3 \dots G_n$. To train the network, we consider it as a regression problem, minimize the distance between $V^{\pi}(s_n)$ and G_n , so that the network can be trained.

It takes long time to get the cumulated reward, since it need the scence from the start of the game to the end of the game. It will be less benefit when this is a long game. To solve this issue, we have TP approach:

Temporal-Difference Approach

As we know, there is a reward r_t bewteen the current cumulated reward $V^{\pi}(s_t)$ and the next cumulated reward $V^{\pi}(s_{t+1})$, so that we have a target function:

$V^{\pi}(s_t) - V^{\pi}(s_{t+1}) = r_t$

In the training progress, for every continuous two action, we estimate the cumulated reward $V^{\pi}(s_t)$ and $V^{\pi}(s_{t+1})$, then we can use regression method to minimize the distance between $V^{\pi}(s_t) - V^{\pi}(s_{t+1})$ and r_t .

The pro for this method is we can update the parameter for the network every action applied, which makes the update efficiently.

Compare MC and TD

TD consider more meticulous on each step while MC is more about considering global effect. But in the situation that if two steps in MC, one is positive another is negative, if they eliminate each other, we can not get a reasonable cumulated reward in the end. This makes the network missed to learn the possitive action in this game period. TD can consider each action's reward but cannot get the global effect, it can not learning the effect of the order for actions and the action's potential impact.

So we have a solution Q-function:

State-Action Value Function(Q-function)

$Q^{\pi}(s, a)$, input the action a at state s to the actor π to get a cumulated reward.

When we have a actor π , the π interact with the environment, then use TD or MC method we mentioned above to learn the Q function, which can evaluate the outcome of actor π and decide which action to take. Thus we can update the parameters and get a new π' . Here we have:

$\pi'(s) = \arg\max a \; Q^{\pi}(s, a)$

Here π' must be better than π, because we choosed the most reward action a by using actor π to interact at environment s , after update parameters by using the most optimal action, the new actor π must be better.

To train a deep Q network, we need a student network and a target network, these two networks are initialized as same. For every batch, we have:

$$Q^{\pi}(s_t, a_t) = r_t + Q^{\pi}(s_{t+1}, \pi(s_{t+1}))$$

Here, $Q^{\pi}(s_t, a_t)$ is the output of the student network while $r_t + Q^{\pi}(s_{t+1}, \pi(s_{t+1}))$ is the target network, consider it as a regression problem and minimize the distance between these two networks, update the updated student network to the target network, then continue training for next batch. The regression loss function we usual use is MSE loss, but considering the idea of training is very similar to knowledge distillation. So I think we may use Kullback–Leibler divergence loss to minimize the output distribution of two network, it probably works.

To avoid the network trains to the local optimal, we can use some random probability to obtain if the action decide by the network or a random voter. This is called `Epsilon Greedy`, we can set a value β , the value of β can be a specific number or a funtion to determin the value as epochs goes up. Then we have a change of $1 - \beta$ to place action by network and change of β to have a ramdom action.

Another useful tip for get sampling data is `Replay Buffer`, it is a instant memory to save the previous paris of action, environemnt and reward. To train the network, we can use the data from this buffer, which can be considered as the dataloader of normal deep learning training. Using of replay buffer can reduce the training time, because the training of reinforcement learning costs lots of time on sampling data. Also pick up the data from the buffer can make a batch more diverse since these experience in the buffer from differnct policy(network).

`DQN` is a combination of deep learning and q-learning, which uses a deep network as the Q function to decide the action.