##### Double Deep Q-Network

In this part, we will introduce how we implement DDQN algorithm, and to see if we conquer the supermario game.

##### AGENDA

First, we’ll have a brief overview of our implementation. Then, we will dive into the introduction of the environment and the agent. And final part is our result.

##### OVERVIEW

As we mention earlier, we use Double Deep Q-Networks as our main Algorithm. And then, we utilize two Convolutional Neural Networks, Q\_online and Q\_target. These networks independently approximate the optimal action-value function, helping our agent make decisions in the environment.

Next, we have two important values, TD Estimate and TD Target. These values are crucial for training our agent. The TD Estimate is the value predicted by our Q\_online network for a given state-action pair, while the TD Target is calculated using the Q\_target network and represents the maximum expected future rewards.

As for the action part, we employ an epsilon-greedy action selection strategy.

That’s all of our implementation, and later we will discuss it in detail

##### Enviroment

And Now, I would like to introduce our environment setup. Let's have a look!

First, we create the Super Mario Bros environment using the gym\_super\_mario\_bros.make function. This sets the stage for our agent to interact with the game.

And then, We utilize JoypadSpace to restrict only two action, walking right or jumping right. This greatly simplifies the decision-making process.

Since the actions taken by Mario in the game are not affected by the background color, we convert the background to grayscale. This means that each state array, shape [240, 256, 3] will become [240, 256, 1]. And then we use ResizeObservation function to further resize it into [84, 84, 1]. By using these methods, it significantly reduces the input data and computational load.

We can speed up training time by using skipFrame wrapper since the game is processed frame by frame and consecutive frames do not differ significantly. We can skip a certain number of frames (n) to accelerate the process without losing too much data. The rewards are accumulated during this skipping process.

Finally, we use FrameStack to combine four consecutive states into a single observation point. There are two main purposes for doing this: firstly, to provide more temporal information since consecutive frames can capture the dynamic changes in the game; secondly, to increase the dimensionality of the input data in order to train more complex models.

After this steps, our state array will look like the picture.

##### Agent

Next, we’ll start to talk about the agent part.

We have created a class named "Mario" that consists of three main components, act, memory and learn.

Let’s start to see how Mario take the Epsilon-greedy action.

## Act

For any given state, an agent can choose to do the most optimal action (exploit) or a random action (explore).

During the exploration phase, we randomly select an action.

On the other hand, during the exploitation phase, we aim to select the action with the highest expected reward.

To gradually reduce exploration as our agent learns, we decrease the exploration rate by multiplying it with an exploration rate decay factor. This helps the agent transition from a more exploratory behavior to a more exploitative one over time.

However, we still want to ensure that there is a minimum probability for exploration even as the exploration rate decreases. We set the exploration rate to the maximum value between the exploration rate minimum (self.exploration\_rate\_min) and the current exploration rate.

## Memory

In this section of the code, we are implementing the functionality to cache experiences and retrieve them for training our agent. Let's go through it step by step:

We receive the state, next\_state, action, reward, and done parameters representing the current experience. We convert them into Torch tensors and append these tensors as a tuple to our memory, which serves as a storage for experiences.

And we use mini batch experience replay to recall our info from memory.

## Learn

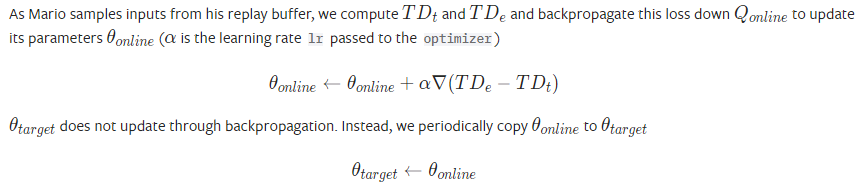
Finally, we are starting to talk the most important part of our agent, “learn”. There are two value involve in, TD Estimate and TD Target. After we get state and action from recall function, we can send the info to TD Estimate.

TD Estimate:

This function takes the state and action tensors as inputs and returns the estimated the optimal Q-value for the given state-action pair.

TD Target:

We compute the TD target value using the td\_target function. First, using epison-greedy action to find the next\_state action. Then using the reward, Qtarget which calculate from the input of next\_state and next action, and get the TD Target. Finally, passing TD Estimate and TD Target to get our loss function.



This equation represent how we update our loss function. And, we choose Adam as our optimizer. Learning rate = 0.00025. we also update target loss to ensure they will be same at the synchronous time.

Small conclusion ……

##### Result

As you can see in the plot, we get the higher reward when the time steps growing. So as the q value. And our loss values decrease compare to the earlier time steps. It’s time to see if Mario can finish the game.