##### AGENDA

Now, we will start by giving an overview of the implementation. Then, we will dive into the introduction of the environment and the agent. Finally, we will discuss the results of our DDQN training.

##### OVERVIEW

First, we utilize two Convolutional Neural Networks, called 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 in play: the TD Estimate and the TD Target. These values are crucial for training our agent effectively. 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. And we will introduce it more detail later.

##### Enviroment

And Now, I would like to introduce you to the powerful environment setup for our Super Mario Bros DDQN project. We've utilized a series of transformations to enhance the learning process. Let's dive in!

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 the action to either walking right or jumping right.

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]. The consistent size of frame must be ensured for efficient computation.

To address the issue of high frame rates, we employ the SkipFrame wrapper which is inherited from gym.Wrapper. It is primarily used to speed up computation during the step execution. Consecutive frames do not differ significantly, so 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. Each state array is transformed into [4, 84, 84, 1].

##### 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 us start to see how Luigi 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 to try out. We generate a random number between 0 and 1, and if it falls below our exploration rate, we choose a random action index from the available actions.

On the other hand, during the exploitation phase, we aim to select the action with the highest expected reward. We pass the current state through our neural network, self.net, to obtain the predicted action values. We then select the action index with the maximum value using torch.argmax along the action axis

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 move them to the appropriate device (e.g., GPU) if necessary. Then, we append these tensors as a tuple to our memory, which serves as a storage for experiences.

To train our agent, we randomly sample a batch of experiences from the memory. We use the random.sample function to select a batch of experiences of size self.batch\_size from the memory.

Next, we unzip the sampled batch using the zip(\*batch) function and apply torch.stack to each element, effectively converting the batch of experiences into tensors. This results in state, next\_state, action, reward, and done tensors that contain the respective values for each experience in the batch.

## Learn

In this section of the code, we perform the necessary steps to update our Q-network based on the experiences sampled from memory. Let's break it down:

Sample from Memory:

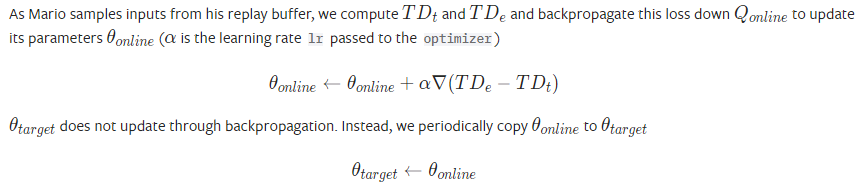
We call the recall function to sample a batch of experiences from the memory. This retrieves the state, next\_state, action, reward, and done tensors representing the sampled experiences.

TD Estimate:

We calculate the TD (Temporal Difference) estimate using the td\_estimate function. 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.

##### Result