Artificial Intelligence in Games – Reinforcement learning and Deep Q-network

Background:

Deep Q Network (DQN) is a neural network (and / or related tool) that uses Deep Q learning to provide models such as simulation of intelligent video games. Deep Q networks can consist of convolutional neural networks or other structures that use specific methods to learn about different processes, rather than specific names for specific neural network structures.

Q1: During training, why is it necessary to act according to an ε-greedy policy instead of a greedy policy (with respect to Q)

The "Exploration-Exploitation" issue in reinforcement learning must be balanced. Because our algorithm is model-free, it solves the learning problem by directly retrieving samples from the emulator. It also learns about the greedy policy (selecting an action with the highest Q value) while following a behaviour distribution that ensures that the state space is sufficiently explored. - A greedy policy with probability ensures that a random action is chosen with probability. This was discovered in the code marked "**Use epsilon-greedy for exploration.**" The agent chooses and performs actions in accordance with the -greedy policy based on Q.

Q2: How do the authors of the paper [Mnih et al., 2015] explain the need for a target Q-network in addition to an online Q-network?

When a non-linear function approximator, such as a neural network, is used to represent the action-value function which can be diverged by reinforcement learning. The basic idea behind reinforcement learning is to use the Bellman equation as an iterative update to estimate the action-value function. Where the preceding statement becomes impractical, Neural Networks step in by estimating the action-value function separately for each sequence without any generalisation. Instead, non-linear approximators such as neural networks are used. This, however, causes it to diverge. The reasons could be due to correlations in the sequence of observations; even minor changes to the action function can significantly alter the policy, and thus the data distribution and correlations between action-values and target values. One solution would be to iteratively update the action-values towards target values that are only updated on a periodic basis, reducing correlations with the target.

3. Explain why the one-step return for each state in a batch is computed incorrectly by the baseline implementation and compare it to the correct implementation.

One of the **env.step()** method's parameters is 'done'. **done** is a boolean type that indicates that the environment should be reset. This occurs when the episode has ended, which can happen in a variety of ways depending on the game. This is used to create the variable done sample. When the episode is reset, the updated q values variable (which denotes updating Q-Value) in the code that we commented becomes zero. It's incorrect to say it has zero. That's why we devised a new equation that keeps the rewards (obtained using the same **env.step() method**) regardless of the episode's state, and then applies the gamma to the future rewards based on the episode's state.

4. Plot a moving average of the returns that you stored in the list episode reward history. Use a window of length 100. Hint: np.convolve(episode reward history, np.ones(100)/100, mode='valid').

5. Several OpenAI Gym wrappers were employed to transform the original Atari Breakout environment. Briefly explain the role of each of the following wrappers: MaxAndSkipEnv, EpisodicLifeEnv, WarpFrame, ScaledFloatFrame, ClipRewardEnv, and FrameStack.

OpenAI Gym wrappers are very powerful features that allow you to add functionality to your environments, such as changing the rewards and inputs, or optimising computations, which our agent requires. We can also use subclasses of gym to change specific aspects of the environment. Wrapper that changes the way the environment handles observations, rewards, and actions. This functionality is provided by the following classes:

- gym.ObservationWrapper(self, observation):
  Used to modify the observations returned by the environment.
  Override observation() method

- gym.RewardWrapper(self, reward):
  Used to modify the reward returned by the environment.
  Override reward() method

- gym.ActionWrapper(self, action):
  Used to modify the actions passed to the environment
  Override action() method

We can add additional functionality to more complex applications of deep reinforcement learning. This can be achieved by creating a custom gym wrapper. Wrapper as a parameter. Data acquisition is a computationally intensive task, as most of the computation time is spent selecting actions. To address this situation, we prioritize the rapid collection of data based on your requirements. Define a step () function to achieve this type of situation, to speed things up, or to enable operations. The reset () feature is also basically useful for resetting the environment as needed.

- **MaxAndSkipEnv(gym.Wrapper):**

The gym wrapper returns actions, sums of rewards, and maximum rewards over the last observations. This wrapper has step() in conjunction with _ init _(self, env, skip=4) to only return actions and rewards every 'skip'-th frame.

- **EpisodicLifeEnv(gym.Wrapper):**

This gym wrapper that makes the End of Life equal to the episode life, but it only resets when the actual game is over. This is useful for viewing. This wrapper has two functions, step () and reset (). In this game, the step () function checks the current life, creates a loss terminal, and then updates and processes the life as a bonus, as the episode is considered from start to finish as a whole game. life. The reset () function resets the lifespan only when it has reached the end of its lifespan. In this way, life is temporary and helps learners to know this behind the scenes, but all conditions are still achievable.

- **WarpFrame(gym.ObservationWrapper):**

This wrapper is an observation wrapper that helps you change the observations returned by env. This wrapper helps to enclose the frame in a size of 84 x 84. If your environment uses dictionary observations, you can specify the observations / frames to enclose. If the image is grayscale, use OpenCV to convert the frame to COLOR_RGB2GRAY, resize it, and then increase the size of the frame.

• **ScaledFloatFrame(gym.ObservationWrapper):**
This wrapper is an observation wrapper that helps you change the observations returned by env. This wrapper helps optimize watch / frame memory requirements. Usually used only in small play buffers.

• **ClipRewardEnv(gym.RewardWrapper):**
This wrapper is a reward wrapper that helps you change the rewards returned by env. This wrapper helps you sort rewards by {+ 1, 0, 1} based on the sign of the reward you received.

• **FrameStack(gym.Wrapper):**
This wrapper is a gym wrapper that helps you optimize memory usage during processing. This wrapper stacks K frames and returns more memory-efficient lazy frames.
Lazy Frames: This object ensures that common frames between observations are saved only once. Its sole purpose is to optimize memory usage. This can be a huge amount for DQN's 2M frame playback buffer. This object only needs to be converted to a NumPy array before it is passed to the model

References:

https://www.techopedia.com/definition/34032/deep-q-networks