Queen Mary, University of London
School of Electronic Engineering and Computer Science

ECS7002P: AI in Games

Coursework 2: Individual Assignment

Mughees Asif | 180288337

Due: 17 December 2021

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

An ϵ-greedy policy strikes the right balance between exploration (choosing different options) and exploitation (refining current option) of an action and the associated reward. During training, the Q-value is maximised to allow approximation of selecting a certain action in a given state. The action is then selected based on the eventual reward. The selection process produces two choices for the agent: select a random action, or select a previously stored action that maximised the Q-value. Instead of choosing one or the other, the ϵ-greedy policy allows the agent to leverage previously stored knowledge whilst exploring new options.

Algorithm 1 ϵ-greedy policy

Data: Q-table: state-action pairs, ϵ: positive integer, S: current state
Output: Optimal action

```
Function SELECT-ACTION(Q, ϵ, S):
    n ⟵ uniform random number b/w 0 & 1
    if n < ϵ then
        A ⟵ random action
    else
        A ⟵ max(S, ϵ)
    end
    return Selected Action A
end
```

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?

The domain of reinforcement learning is inherently unstable due to the sequential nature of trail-and-error problems. Performing gradient descent on states that are highly correlated e.g., snapshots of a video game on a milli-second basis, results in the model overfitting as the overall change in the state from one snapshot to another is marginal. This can cause divergence during gradient descent when a "nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q)" [Mnih et al., 2015]. Therefore, two modifications on the vanilla online Q-network allows the agents to choose the optimal action for a given state. Firstly, the addition of *all* the experiences to a replay buffer allows the maximisation of the data efficiency by consistently updating the weights. Secondly, a new neural network is defined to generate the targets y_i to offset the oscillations caused by the update of $Q(s_t, a_t)$ to the $Q(s_{t+1}, a_{t+1})$, hence also increasing the target values y_j [Mnih et al., 2015].

Q3: 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.

```
    # Baseline implementation
    updated_q_values = [rewards_sample + gamma * tf.reduce_max(future_rewards, axis=1)] *
    ↪  [(1-done_sample) - done_sample]

    # Correct implementation
    updated_q_values = rewards_sample + [(1-done_sample) * gamma *
    ↪  tf.reduce_max(future_rewards, axis=1)]

```

The baseline implementation utilises the immediate reward added to the discounted one-step estimated value, instead of using the cumulative sum of the discounted rewards. The aspect of using an estimation to calculate another estimation is referred to as **bootstrapping**, which produces a higher bias with a lower variance due to making estimations *during* an episode instead of waiting till the end. Additionally, the baseline implementation does not consider terminal states, which is incorrect as the model needed to be trained for `Atari` games, which in most instances have a terminal state.

Q4: Plot a moving average of the returns that you stored in the list episode reward history. Use a window of length 100.

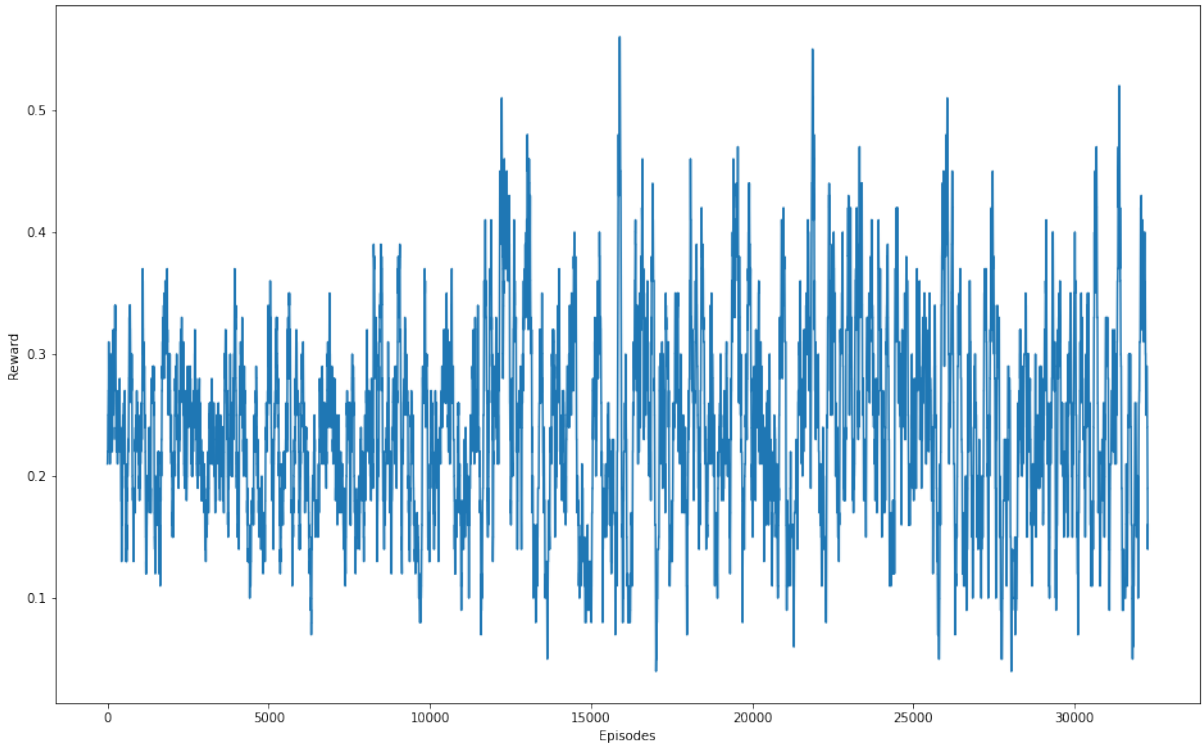

Figure 1: Moving average of the returns

Q5: 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`.

`MaxAndSkipEnv`[1]:

- Speeds up training of the neural network by returning an N^{nt} observation, thereby skipping the intermediate frames where the previously obtained optimal action is simply repeated.
- Takes the maximum of the last two frames as an observation to offset the flickering effect on some Atari games.

`EpisodicLifeEnv`[2]:

- Aids value estimation (estimation of future policies) by modifying the environment.
- Each time an agent loses a life, the episode is terminated, however the game is only reset once it is actually completed.

`WarpFrame`[2]:

- Helps in feature extraction.
- Converts all the images according to user-provided $N \times N$ dimensions, making it faster to locate the salient features.

`ScaledFloatFrame`[3]:

- Normalizes observations $\backsim [0, 1]$.
- Brings the pixel values down to a common scale.

`ClipRewardEnv`[3]:

- Clips the reward according to the sign.
- Normalizes the input and out to homogenise the gradient magnitudes during training.

`FrameStack`[2]:

- Helps in stacking the frames to monitor the motion of objects.
- Improves the reaction of agents as per the requirement of future states.

[1] Torres, J., 2020. Deep Q-Network (DQN). [online] Medium. Available at: <Link> [Accessed 12 December 2021].
[2] Clark, R., 2020. Mario's Gym Routine. [online] Medium. Available at: <Link> [Accessed 12 December 2021].
[3] OpenAI API Documentation.