Double Deep Q-Learning algorithms for Atari games with PyTorch

Saeid Amirhaftehran
Autonomous and Adaptive Systems
University of Bologna
email : saeid.amirhaftehran@studio.unibo.it

Abstract

Reinforcement learning is an area of Machine learning in which an agent tries to solve a particular task by exploring an environment and receiving a reward signal. In this article, I want to implement Double Deep Q-Learning (DDQN) to play an Atari game. One of the most valuable libraries to implement such an algorithm is PyTorch.

1 Introduction

Machine learning to play a video game is one of the most popular areas of Reinforcement Learning. Deep reinforcement learning is essential for making an AI play video games. This research uses <Playing Atari with Deep Reinforcement Learning> and <Deep Reinforcement Learning with Double Q-learning> from Google Deep Mind as the main sources. We will implement the algorithm in PyTorch using Python.

2 Problem Definition

The main purpose of this research is trying to implement an algorithm to solve the Breakout-v0 environment in the Open AI gym kit using the Double DQN. The environment simulates a game where the agent tries to clear all the obstacles in the upper right of the game with a ball while preventing the ball from falling. The game's primary goal is to direct the agent to win the game(clear all the obstacles with the lowest number of losing games – which means the ball gets past the paddle). This article's main environment is Breakout, but the algorithm can be used for most Atari 2600 games.

3 Break-Out Environment

3.1 Framework

The framework used for the problem is the GYM, a toolkit made by Open AI for developing and comparing reinforcement learning algorithms. For these purposes, we use the GYM version of 0.19 to implement the Break-Out environment. This environment is part of the Atari environment. In this game, you move a paddle and hit the ball in a brick wall at the top of the screen. Your goal is to destroy the brick wall. You can try to break through the wall and let the ball wreak havoc on the other side, all on its own! You have five lives.

Submitted to NeurIPS 2021 AI for Science Workshop.

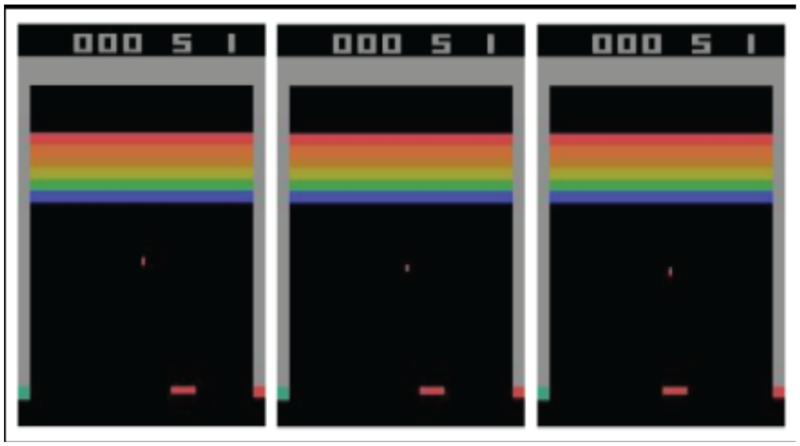

Figure 1: Break-Out environment

3.2 Observations and state space

By default, the environment returns the RGB image that is displayed to human players as an observation.

3.3 Action space

Four discrete actions are available: do nothing, move left, move right, and Fire.

3.4 Reward

The amount of reward achieved by the action. The reward here is the number of obstacles that the agent destroyed.

4 Data Preparation

In this article, we use breakout V0. You can see the environment in **??**. In this mode, the frame skip parameter in the Atari environment is set to 2 to 5. This parameter indicates the number of frames (steps) in the one action is repeated. This is called the frame-skipping technique, which allows us to play more games without significantly increasing the run time. In this research, we set the frame skip parameters into 4.

Another critical factor that can make our computation faster is the screen itself. The RGB images do not provide more information than grey-scale images. Therefore, it is necessary to keep only helpful information by cropping frame images and converting them to a grey scale. We use the provided method inside the GYM.wrappers library to convert the real RGB image of the game. First, we make this environment using $gym.make$ method. Then followed by the $GrayScaleObservation$ method of wrappers to convert the image to Gray Scale. In figure 2, you can see the result of this transforming an RGB game frame into grey.

In the next step, we will use a technique named Frame Stack. We will stack 4 RGB frames altogether. A stack of 4 frames is simply to catch information like the velocity of objects in this game movement of the ball. We use the $FrameStack$ method for stacking images in the $Gym.wrappers$ library. The method will stack the last seen 4 observations.

As you can see in 1, the top region of the screen is not related to the game. It contains the current score and number of lives agents have, so that we will be cropping this part. We will crop this region using the $crop$ method available in the PyTorch Transformers library. You can see the final result of

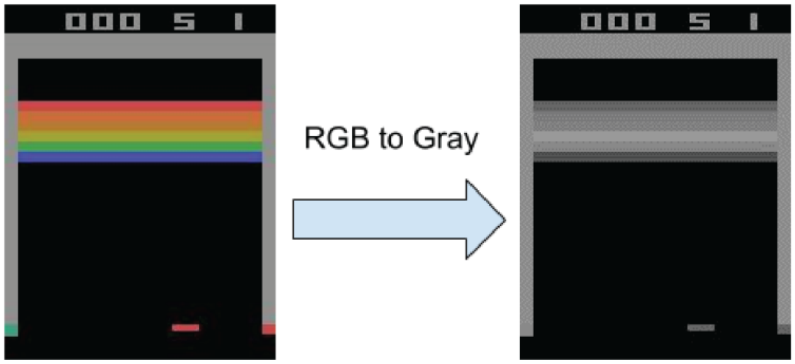

Figure 2: Transform RGB frame to grey-scale

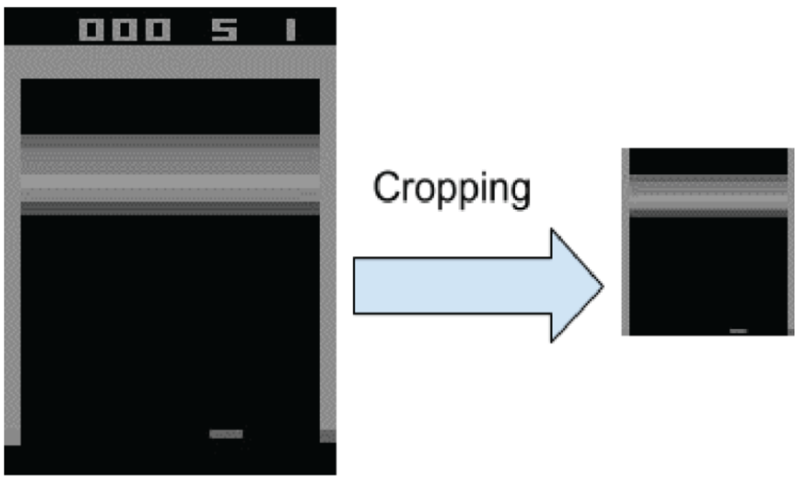

Figure 3: Cropping grey scale frame

transforming the original frame of the game and cropping out the unimportant part of the screen in figure 3.

This game's original observation size is 210 rows and 160 columns. After cropping the image again using the $resize$ method available in the PyTorch Transformers library, we will resize the observation to 84 rows and columns.

The last thing that we consider is clipping the reward between -1 and 1.

5 Double Deep Q-Learning

5.1 Background

Here we are going to design the brain of our agent. Lets S be a state, a be an action, $R(S, a)$ be the reward function, and $Q(S, a)$ be the value function. Under a given policy π The true value of action a in state s is:

$$Q_\pi(s, a) \equiv E[R_1 + \gamma R2 + ...|S_0 = s, A_0 = a, \pi] \tag{1}$$

Where $\gamma \in [0, 1]$ is a discount factor that trades off the importance of immediate and later reward.

Estimates for the optimal action values can be learned using Q-learning. The most interesting problems are too large to learn all state action values separately. Instead, we can learn a parameterized value function:

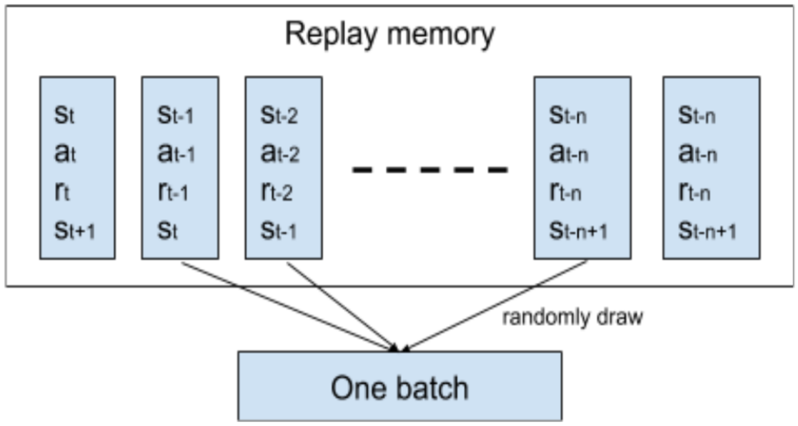

Figure 4: Experience Replay

$$\theta_{t+1} = \theta_t + \alpha(Y_t - Q(S_t, A_t; \theta_t))\nabla_{\theta t} Q(S_t, A_t; \theta_t) \tag{2}$$

Where α is a scalar step size, and the target Y_t is defined as:

$$Y_t^{DQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, \alpha, a; \theta_) \tag{3}$$

5.2 Deep Q Network

A deep Q network (DQN) is a neural network for a given state s outputs a vector of the action values. Two important ingredients of DQN propose using a Target Network with parameters θ^{-} and a reply memory.

In DQN, the Target network is as same as Online Network. There are different ways to update the target network. The most proposed way is that the Target Network copies its weights every τ step from the online network.

For the experience replay, we use memory to store observed transitions for some time and sample uniformly from this memory bank to update the network 4. Until now, our agent has taken the screen images at each step, which are the last four frames (84*84*4). Then feed it into Q-network and come up with the action. Then our Game class took this action and returned the following images, which are S_{t+1}, and reward R_t. The quadruplet (S_t, a_t, r_t, S_{t+1}) will be stored in memory and taken as a sample for training.

5.3 Double DQN

The max operator in standard DQN, in (3), uses the same value for both to select and evaluate an action. This makes it more likely to select overestimated values, resulting in over-optimistic value estimates. To prevent this, we can decouple the selection from the evaluation. This is the idea behind Double Q-learning. So in Double DQN, For each update, one set of weights is used to determine the greedy policy (θ) and the other to determine its value (θ').

So everything is just like the DQN except that the target will be computed using the target network:

$$Y_t^{DDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, argmaxQ(S_{t+1}, a; \theta_t); \theta'_t) \tag{4}$$

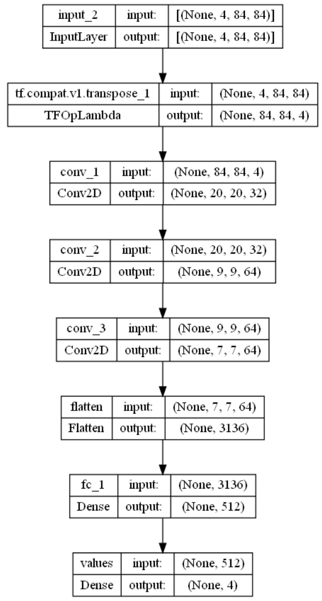

Figure 5: Architect of Q-Network

6 Network Architect

In this section I will explain the architect of my network. I use PyTorch 1.13.1 with Cuda version 11.7 to implement my network on Python version 3.9. I use the same architect propose by Google Deep Mind for playing Atari games using Gray Scale Observation 5.

7 Conclusion

The Double deep Q-learning algorithm has made an essential step toward general artificial intelligence. The purpose of this article was to implement the basic Double Deep Q-learning algorithm with PyTorch.