Micah Simmerman

Introduction to Artificial Intelligence (AI), CSPB-3202

**Q-Learning and DQN RL in Atari OpenAI Gym Game Environments**

**Introduction**

OpenAI Gym is a Pythonic library API package that provides numerous class definitions for simulated training and testing of reinforcement learning (RL) agents. This paper begins with an at-length description of the development and hyperparameter adjustment of a model-free Q-learning agent, which can achieve win ratios greater than 90% (including a stabilized response) in under 10,000 iterations of game play (i.e., episodes). The development of this Q-learning agent involved careful discretization of the state-space as well as intensive tuning of hyperparameters as part of an annealing epsilon-greedy policy improvement method.

After we have covered the fundamentals of Agent and Environment creation in OpenAI Gym, we extend our discussion to the topic of Deep Q-Networks (or “DQN” for short), which forms a natural extension of the Q-learning algorithm, extended over a Deep Convolutional Neural Network (cDNN) topology.

**Body (Break into Sections):**

**Methods and Tools**

*Hardware Used in this Project:*

* 1TB SSD HD
* 16GB 3200 MHz RAM
* 4GB CUDA enabled GPU
* Linux or PowerShell Bash Terminal and basic familiarity with directory management
* (Optional): fork or clone of this project repository

*Operating System and Development Environment:*

* Modern operating system, (Gym seems to prefer Linux and Mac based OS.
* VSCode or another IDE (suggested)

*Library Dependencies:*

While the bulk of this project proceeded as planned, several hardware and dependency issues were encountered, mainly concerning the library compatibility of my local Windows 10 Operating System(s). These findings are presented in the list below:

1. Atari OpenAI Gym apparently depends heavily on UX/UI graphic library packages such as Box2D and Retro. These libraries can be VERY difficult to install and configure on a Windows10 operating system. The most referenced solution for this is to install and configure an Ubuntu subsystem using WSL1 or WSL2.
2. TensorFlow works best under a GPU based processing abstraction that is enabled by installing the appropriate GPU drivers for the system in question. According to techpowerup.com, the CUDA family of GPU drivers can be used to program many modern GPUs for this purpose. The details of this configuration process typically require the operation of a bash terminal using pip.

***(RESOLUTION: TBD) install WSL2 interpreter and layer Ubuntu on the Win10 python lib developer env.***

To see a short list of the dependencies that were installed to complete this project under a local development environment, please visit the GitHub repository listed for this project (INCLUDE GITHUB LINK HERE).

**Problem 1: Generalized Q-Learning Agent in the MountainCar-v0 (Classic Control)**

I began this project in earnest with a script-based agent, controlled with a self-annealing epsilon-control value/policy iteration and improvement algorithm at its’ core. This technique employs the use of a Q-table, which is akin to a 2-dimensional memoization table that registers assigned q-values associated with each state action pair at the agents’ current state.

The purpose of the epsilon term is to control the level of stochasticity in the agent’s immediate choice of action: starting with a freshly initiated game environment, the agent selects an action randomly from it’s current state roughly 50% of the time.

After spending a great deal of time installing and configuring the dependencies required to implement this q-learning agent; I was able to test, train, and monitor the performance of the hill-climbing bot by monitoring the program response inside of a Windows10 PowerShell terminal window

By splitting the agents’ behavior between epsilon-greedy Q-policy improvement, and completely randomized behavior,

This MountainCar Q-learning agent utilizes an episodically annealing epsilon stochasticity control term that is decremented by a fixed ratio of epsilon’s initial value (called ‘*epsilon\_decay\_value’* in figure 1, shown below). The epsilon

# define the epsilon control parameter

epsilon = 0.5  # controls agent stochasticity, decreases by 'epsilon\_decay\_value' for the first 'END\_EPSILON\_DECAYING' episodes

START\_EPSILON\_DECAYING = 1

DIV\_FACTOR = 30  # try 2,3,4,...  larger value == larger delta\_epsilon "chunks"

END\_EPSILON\_DECAYING = EPISODES // DIV\_FACTOR  # larger divisor => larger delta\_epsilon => more exploitation

epsilon\_decay\_value = epsilon/(END\_EPSILON\_DECAYING - START\_EPSILON\_DECAYING)  # normalized epsilon decay value

Figure 1.) Important variables for tuning the epsilon hyperparameter’s that control the agent’s annealing process. ‘DIV\_FACTOR’ forms the primary control variable here because it alone controls the fraction of epsilon that is removed during each step of game play. Stochasticity control using epsilon is shown later.

The annealing rate of the epsilon stochasticity term is delicately controlled by ‘epsilon\_decay\_value’ (shown above)

are the hyperparameters used to control the agent’s exploration/exploitation trade-off on each iteration of a single round of gameplay.

Interestingly, the choice of epsilon itself (i.e., 0.5 as shown above) does not afford much control over the agent’s behavior. Instead ‘DIV\_FACTOR’ (which in turn modifies ‘epsilon\_decay\_value’) is the primary variable that controls the rate of annealing, and therefore the time required to reach a stable and efficient performance baseline generated by a globally optimal policy.

which (after a series of other parameter adjustments) carefully tunes the level of stochasticity in the agents’ behavior when locating the optimal policy.

it under the MountainCar-v0 Gym environment.

**Results:**

LEARNING\_RATE = 0.15  # 0.1

DISCOUNT = 0.95  # 0.95

EPISODES = 10000  #

SHOW\_EVERY = 1000  #

DIV\_FACTOR = 30  # try 2,3,4,...  larger value == larger delta\_epsilon "chunks"

END\_EPSILON\_DECAYING = EPISODES // DIV\_FACTOR  # larger divisor => larger delta\_epsilon => more exploitation

A screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

91.95% win rate over just 10,000 episodes. That’s a NEW RECORD!! Let’s try that again, since part of the agent q-learning model is stochastic by definition.

A screenshot of a computer

Description automatically generated

A graph of different colored lines

Description automatically generated

That time the epoch was EVEN better. After a lot of fine-tuning, the three variables that appear to push this q-learning model above the 90% win threshold appear to be 1.) the optional dimension composing DISCRETE\_OBS\_SIZE parameter, 2.) DIV\_FACTOR, which is inversely proportional to the size of the epsilon decay value, and 3.) LEARNING\_RATE, which appears rather stubbornly planted between 0.10 and 0.15, with preference for the later.

**Sources Cited**

<https://medium.com/@CalebMBowyer/strategies-for-decaying-epsilon-in-epsilon-greedy-9b500ad9171d>