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.

The primary benefits of q-learning include its’ high online availability, ease of hyperparameter tuning, and relative transparency compared to other RL learning algorithms, and its’ effectiveness in identifying globally optimized policies. This project will investigate the performance of a self-annealing epsilon-greedy q-learning policy by critiquing its’ performance in the OpenAI Gym “MountainCar-v0” puzzle environment.

**Body (Break into Sections):**

**Methods and Tools Used**

*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). Many of these findings are presented in the list below:

1. Atari OpenAI Gym depends heavily on UX/UI graphic library packages such as Box2D and Retro, which can be VERY difficult to install and configure within Windows operating systems.
2. The most referenced solution for solving the incompatibility of Gym with Windows operating systems is to install and configure an Ubuntu subsystem, enabled under WSL1 or WSL2. The author has determined this to be a very error-prone process.
3. TensorFlow works best under a GPU-based computational abstraction afforded by the correct GPU drivers. According to techpowerup.com, the CUDA family of GPU drivers can be used for precisely this purpose. It is suggested that the reader ensure he or she has access to a well-configured, Gym-compatible bash terminal before getting started.

***(RESOLUTION: TBD) install WSL2 interpreter and layer Ubuntu on the Win10 python lib developer env? Something is unstable about the current local configuration.***

To see a full list of the dependencies used to implement the q-learner developed in this project, please visit the GitHub repository located at the following URL: <https://github.com/Micah614/AI_SemesterProject>.

**Generalized Q-Learning and the MountainCar-v0 (Classic Control) Gym Environment**

*Why Q-Learning?*

Q-learning agents have multiple advantages that make them an excellent choice for unsupervised agent learning models. Some of these benefits include:

* **Model-Free**. There is no need to construct complex abstractions of the environment to create a functional policy.
* **Mathematically Simple**. Q-learning algorithms are mathematically simple at their core. All Q-learning algorithms can be viewed as a natural extension of the Bellman Equation. A fundamental and vital property of the Bellman equation is that it ensures convergence on an optimal policy.
* **Unsupervised**. Q-learners train themselves to make optimal decisions based on real-time policy adjustment and varying levels of stochasticity. They do not rely on labels or complex pre-digestion of raw data. Decisions are derived directly from the data.
* **Adjustable**. Q-learning hyperparameters are easily adjusted. The effects of these hyperparameters on agent training are relatively simple to comprehend compared to \_\_\_.
* **Stochastically Balanced**. Adding an element of “selective randomness” to the agent allows it to discover an optimal balance between state space exploration and policy exploitation which assists q-learners in locating an optimal policy.
* **Extensible**.Q-learning is a sub-category of RL-based Machine Learning (ML) algorithms, and therefore offers a gentle introduction to an otherwise complex and highly technical field.
* **Used in Neural Network and Learning Ensembles (i.e., higher abstractions).** Q-learning algorithms are frequently included in neural network architectures, including Deep-Q Network (DQN), Deep Neural Network (DNNs) topologies. Q-learners are also a common workhorse in many machine learning ensemble methods (source: <https://arxiv.org/abs/2103.00445>).

For those in need of a general refresher on the math involved, particularly the central Bellman operator, a discussion about the common variations of q-learning algorithms may be found here: <https://www.cs.cmu.edu/~negrinho/assets/homework/deep_q_learning.pdf>.

*MountainCar-v0 Environment*

(TODO: Describe the “MountainCar-v0” OpenAI Gym environment, documentation, feature abstractions, etc.)

*Monitoring the Agents’ Progress*

After spending a great deal of time installing and configuring the various Gym/Gymnasium library dependencies required to implement my q-learner, I was able to test, train, and monitor the performance of the hill climbing agent by monitoring its real-time progress in a Windows10 PowerShell.

(TODO: continue this section)

*Modeling the State-Space Abstractions*

(TODO: Explain “DISCRETE\_OBS\_SIZE”)

*Epsilon-Annealing, and Convergence on the Optimal Policy*

The hill-climber agent is controlled by a self-annealing, epsilon guided policy iteration and improvement algorithm that relies heavily on a q-table data structure. This technique iteratively refines q-table values that correspond to the available state-action pairs provided by the agents’ environment in the current state. The epsilon term controls the level of stochasticity (i.e., “randomness”) of the agent’s behavior at any game state by comparing it with a proportionally scaled, randomly selected value.

Because epsilon is a self-annealing term, the agent chooses an action at random when the comparison term is bounded by epsilon and chooses a policy-informed action resulting in maximum q-value, otherwise. In this way, the agent begins each episode in an “exploring starts” mindset but converges on an optimal solution as epsilon is degraded at a linear rate following each time step.

The choice of the “epsilon\_decay\_value” hyperparameter is a subject of great interest, since the annealing rate of epsilon effectively determines if and how quickly the agent converges upon an optimal policy (as indicated by a mini batch win rate of 100% with zero level variance).

proportional to initiated value of epsilon itself (see “q\_learner.py” for more information on this).

This attitude of exploring starts is critical if the agent hopes to uncover a truly optimal policy during game play.

Step-by-Step:

1. Starting with a freshly initiated environment, the agent selects a random action approximately 50% of the time.
2. The agent

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>