Micah Simmerman

Introduction to Artificial Intelligence (AI)

CSPB-3202

Fall 2023

**Q-Learning RL in 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.

**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:*

* Linux or Mac based operating system, or a WSL Linux Distribution (Windows)

*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. Many of the OpenAI Gym environment packages depend heavily on UX/UI graphic libraries such as Box2D and Retro. These can be difficult to install on Windows operating systems. The most common solution for this incompatibility is to install and configure a WSL-based Linux distribution (WSL1 or WSL2).
2. TensorFlow works best under a GPU-based computational abstraction that is afforded by installing the correct GPU drivers. According to techpowerup.com, the CUDA family of GPU drivers may be used for precisely this purpose. The interested reader should evaluate the specifications of the available GPU (i.e., CPU-GPU serial bus bandwidth, GPU power consumption, quantity of GPU cores, etc.) before investing large amounts of time attempting to abstract his or her GPU.

***(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 other methods (e.g., ).
* **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*

The MountainCar-v0 Environment is a Classic Control MDP world where the objective is to incentivize an agent to accelerate the car with enough motion to reach the flagpole goal state.

**Action Space:** Composed of three (3) discrete deterministic actions {0: accelerate left, 1: no acceleration, 2: accelerate right}.

**Observation Space:** An array with shape (2,). The elements are: {0: current x-axis position, 1: velocity}.

**Transition Dynamics:** MountainCar-v0 is a pure **Markov Decision Process (MDP)** that provides the agent with a set of reliable transition dynamics, which are kinematic equations governing the position and velocity of the car, given a series of state-action pairs that lead the agent to the current state. The kinematic equations that govern the transition dynamics of the MountainCar-v0 world, according to <https://www.gymlibrary.dev/environments/classic_control/mountain_car/> are:

* **velocityt+1 = velocityt + (action - 1) \* force - cos(3 \* positiont) \* gravity**
* **positiont+1 = positiont + velocityt+1**

Equation 1.) “force” = 0.001 and “gravity” = 0.0025.

Wall collisions are defined to be inelastic at either end of the terrain, and the car’s velocity is instantaneously set to zero after a collision with either boundary.

**Reward:** the agent receives a living reward of -1 at each timestep.

**Starting State:** The car is assigned to a random position between -0.6 and -0.4, and a starting velocity of 0.0 upon game initiation (.reset()).

**Terminating Conditions:** An episode ends when one of two conditions is met: 1.) the position of the car is greater than or equal to 0.5, indicating that the car has reached the flagpole’s position at the top of the right hill (“termination”); or 2.) the episode’s length of the episode has reached 200 timesteps (“truncation”).

*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 upon an Optimal Policy*

The hill-climbing agent is controlled by a self-annealing epsilon term, that guides its’ policy iteration and improvement algorithm. The agent’s policy is defined with a q-table matrix data structure that stores and updates q-values associated with every available state-action pair that the agent sees.

This technique iteratively refines q-table values that correspond to the available state-action pairs provided by the agents’ environment in the current state. Epsilon controls the level of stochasticity (i.e., “randomness”) of the agent’s behavior by comparison to a scaled random value: If the random value is less than epsilon, the agent makes the “random choice”. If not, the agent makes the “greedy” (i.e., “policy informed”) decision by exploiting the state-action pair that is associated with the largest q value in the table. Because epsilon is an annealing term, the agent’s behavior becomes more “intentional” with each round of gameplay as its’ actions are progressively controlled by its “learned” policy rather than stochasticity. The epsilon term is set once at the start of a training epoch and is diminished during every step of gameplay.

Recall that a gameplay episode is composed of multiple timesteps, and that there can be numerous episodes in a single training session (e.g., 10,000+ episodes of gameplay is quite common for an online training run). But why do we want the agent to act randomly at all? Can’t we let the agent exploit the optimal policy at each timestep? Wouldn’t that be better?

The reason is that the agent needs space to explore all options before settling into a so-called “optimal policy”. This is to ensure that the agent has information about the best starting options to base its’ gameplay strategy on. This is critical to the agent’s performance, since each state-action pair is a function of the sequence of state-action pairs leading to the current position, and because creating an optimal policy means taking the absolute best action at every time step. If the agent does not explore its’ options before designing a policy, this policy will very likely be misinformed.

*Choosing the Optimal Epsilon Decay Rate*

The choice of “*epsilon\_decay\_value*” is a subject of great interest in epsilon-greedy q-learners since the annealing rate of epsilon effectively determines **if** and **how quickly** the agent converges on an optimal policy.

In the experience of this author, the best design choices for an epsilon-greedy decay factor will be directly proportional to the initialized value of epsilon itself. Numerous formulas exist for controlled decay of simulated annealing the epsilon control term. [CITATION] This choice of hyperparameter control depends heavily on the complexity of the problem space, but the most robust choices for an epsilon annealing rate are likely to be linear or polynomial in form. The figure below illustrates some of the epsilon-greedy design choices implemented during this project.

(MENTION BELLMAN UPDATE EQUATION HERE?)

LEARNING\_RATE = 0.15  #

DISCOUNT = 0.95  #

EPISODES = 10000  #

SHOW\_EVERY = 1000  #

DIV\_FACTOR = 30 #larger DIV\_FACTOR 🡺 larger delta\_epsilon 🡺 faster annealing

END\_EPSILON\_DECAYING = EPISODES // DIV\_FACTOR

# normalized epsilon decay value

epsilon\_decay\_value = epsilon/(END\_EPSILON\_DECAYING - START\_EPSILON\_DECAYING)

Figure 1.) Optimized choice of LEARNING\_RATE (alpha), DISCOUNT (gamma), EPISODES (rounds of gameplay during online training), SHOW\_EVERY (periodicity factor determining )

# 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.

**Training and Test Results:**

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-rate threshold appear to be controlled by 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.

**Conclusions**

**Proposals for Future Work**

**Works Cited**

Strategies for Decaying Epsilon in Epsilon Greedy (Policy Improvement)

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

Training Deep Q Learning Networks (DQN) Intro and Agent-Reinforcement Learning

<https://pythonprogramming.net/training-deep-q-learning-dqn-reinforcement-learning-python-tutorial/>

Train a Deep Q Network with TF-Agents

<https://www.tensorflow.org/agents/tutorials/1_dqn_tutorial#environment>

Mountain Car

<https://www.gymlibrary.dev/environments/classic_control/mountain_car/>

Atari

<https://www.gymlibrary.dev/environments/atari/>

Reinforcement Learning algorithms – an intuitive overview

<https://smartlabai.medium.com/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc>

Understanding Q-Learning: A Powerful Reinforcement Learning Technique

<https://medium.com/@navneetsingh_95791/understanding-q-learning-a-powerful-reinforcement-learning-technique-29a3da36f611>

Introduction to Reinforcement Learning. Part 3: Q-Learning with Neural Networks, Algorithm DQN

<https://markelsanz14.medium.com/introduction-to-reinforcement-learning-part-3-q-learning-with-neural-networks-algorithm-dqn-1e22ee928ecd>

Q-Learning Definition

<https://www.techtarget.com/searchenterpriseai/definition/Q-learning#:~:text=Q%2Dlearning%20is%20a%20machine,a%20type%20of%20reinforcement%20learning>.

Ensemble Bootstrapping for Q-Learning

<https://arxiv.org/abs/2103.00445>

Papers

# “A deep reinforcement learning assisted simulated annealing algorithm for a maintenance planning problem”