Micah Simmerman

Introduction to Artificial Intelligence (AI)

CSPB-3202

Fall 2023

**Q-Learning and the MountainCar-v0 OpenAI Gym Environment**

**Introduction**

OpenAI Gym is an open source pythonic library API that provides many useful class definitions for simulated training and performance testing of Artificial Intelligence (AI) learning programs (agents). This project essay summarizes the development, tuning, and performance evaluation of a model-free Q-learner agent designed to conquer the challenges of the OpenAI MountainCar-v0 Gym environment.

This MountainCar agent’s design is driven by a self-annealing epsilon-greedy q-policy, complete with offline database. The self-annealing search term driven stochasticity of this model, in conjunction with exploitation of a derived optimal q-policy, enables the agent to achieve a steady rate of gameplay performance with only 10,000 iterations of gameplay training. The models’ performance was monitored using side-by-side AUC curves. With appropriate selection of hyperparameters and sufficient training time, the MountainCar agent has been demonstrated to achieve a sustained 100%-episode win-rate using an average of 134+/-1 timesteps per episode.

**Methods and Tools:**

*Python Virtual Environments (venv):*

Some of the OpenAI library tools and dependencies involved in this project can be difficult to install and manage under some operating system environments. Managing the contents of OpenAI projects in a well-managed python virtual environment is therefore highly recommended. Managing OpenAI Gym projects in a python virtual environment dramatically increases the robustness and platform independence of your code, and may save time and frustration due to problems arising from OS incompatibilities and setting/resetting of globally defined PATH variables on your computer. It is also recommended to maintain your repository at the root of a clean HD partition with plenty of space for development activities. Multiple vendors exist for the creation of python virtual environments, including Virtualenv, Pipenv, and Conda.

*Library Dependencies:*

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

**Why Q-Learning?**

The primary benefits of q-learners include their high online availability, ease of hyperparameter adjustment, and relative training transparency compared to other algorithm techniques (e.g., neural networks). The Q-learner’s reliance on the Bellman equation for state valuations and policy iteration makes it a natural choice for MDP-model environments.

This reliance on the Bellman update ensures that an optimal play policy can always be achieved, provided that the agent is provided with a sufficient level of training and its’ hyperparameters have been appropriately selected.

Q-learning agents have many other advantages that make them an excellent choice for unsupervised learning models. Although many variations of the central q-learner model exist (e.g., “SARSA” on-policy methods), all q-learners contain a central update feature that relies on the Bellman Equation. Here are some more key benefits of q-learning algorithms:

* **Model-Free**: There is no need to construct complex abstractions of the agent’s environment to create an effective policy. Instead, Q-learners develop optimal policies by observing the rewards from each state-action pair taken during many rounds of gameplay. The best available choice corresponding to each state-action pair is memorized in an offline (i.e., pickle) database.
* **Guaranteed Optimality**: All Q-learning algorithms involve some natural extension of the Bellman Equation. A fundamental and important property of the Bellman equation is guaranteed convergence on an optimal policy.
* **No Abstract, Labeled, or Preprocessed Data Needed:** Q-learners do not require data labels or any complex pre-digestion of raw data to train. Instead, Q-learning decisions are derived solely from observed gameplay outcomes.
* **Tunable**: Q-learning hyperparameters can be adjusted easily, which increases the transparency of the agent model.
* **Stochastically Balanced**: Adding an element of “selective randomness” to the agent behavior in the form of an epsilon term allows the agent to achieve an optimum balance between state graph *exploration* and *exploitation* of the derived policy.
* **Extensible**:Q-learning is a sub-category of RL-based Machine Learning (ML) algorithms and offers a gentle introduction to an otherwise complex technical subject.
* **Used in Neural Network and Learning Ensembles:** Q-learning algorithms can be combined with neural networks and other high-order abstractions that maximize computational efficiency. Examples include Deep-Q Networks (DQNs) and Deep Neural Networks (DNNs). Q-learners are also a very common workhorse in many machine learning ensemble techniques (source: <https://arxiv.org/abs/2103.00445>).

For those in need of a general refresher on the Bellman equation, a discussion on the common forms of q-learners using the Bellman update equation can be found here: <https://www.cs.cmu.edu/~negrinho/assets/homework/deep_q_learning.pdf>.

*Enter, MountainCar-v0…*

The MountainCar-v0 environment is a continuous-state MDP, and a member of the OpenAI Gym’s Classic Control class of gaming environments. The objective of MountainCar-v0 is to incentivize the agent to reach the flagpole (goal state) located at the upper-right portion of the game space before a predetermined number of timestep iterations has passed. The state graph is initialized with the car (agent) placed at the bottom of a sinusoidal valley with zero velocity to start.

Each state within the continuous-valued GDP state graph is comprised of a position and velocity value pair that define the agents’ current state in the MDP graph. The q-learner is driven to explore the MDP state value graph by the “randomness” imposed by its’ epsilon term and driving effect of the for/while loop. The effects of these effects on each timestep iteration allows the agent to uncover an informed tabular policy based on the former relative utility of each move option with respect to the agents’ current state.

The agent’s (position, velocity) pair is updated during each timestep by the state values returned by “env.step(action)”. The state-value pair that is returned on each step depends on the agents’ sequence of action choices, which is at first quite random, but rapidly anneals to leverage an informed policy.

At each (state, velocity) intersection, the agent is presented with 3 options. Every action that does not encounter the goal state adds a -1-living penalty to the agents’ score. More details about the MountainCar-v0 continuous state MDP environment are shown below in Table 1.

|  |  |
| --- | --- |
| Action Space (discrete) | Discrete (3): {move left, no-op, move right} |
| Observation Shape (discrete) | array (2,): (position, velocity) |
| Observation High (continuous) | Range: [0.6, 0.07] |
| Observation Low (continuous) | Range: [-1.2, -0.07] |
| Import Statement | gym.make(“MountainCar-v0”) |

**Table 1.)** Description of the Mountain Car environment, collected from official Gymnasium documentation at: <https://www.gymlibrary.dev/environments/classic_control/mountain_car/>.

*Transition Dynamics of MountainCar-v0*

The MountainCar-v0 environment is governed by a set of Euler-Newton kinematic equations that act as a state-transition model for the MountainCar environment. The following equations control the state-transition dynamics of the MountainCar MDP environment according to <https://www.gymlibrary.dev/environments/classic_control/mountain_car/>.

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

**Equation 1.)** Kinematics equations governing the state-transition effects of the MountainCar-v0 environment. (“force” = 0.001 and “gravity” = 0.0025).

Wall collisions are defined as inelastic in MountainCar-v0 and occur at both ends of the terrain. Meaning that the car’s velocity term is set to zero upon contact with either boundary. The game (episode) is over when 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 hill (called “termination”); or the episode has exceeded some user-defined number of timesteps (often called “truncation”).

*Modeling the MDP State-Space*

The continuous MDP state-space model provided by MountainCar-v0 presents computational challenges owing to infinitesimal approximations and errors due to rounding and truncation of float-based values. To mitigate these effects, the continuous state space may be discretized (divided into approximate subregions, representative of the original state graph). “NUM\_BINS” is the primary variable used to control the level of discretization of the MDP state environment. This constant variable is therefore of primary importance when designing a discrete, yet optimized form of a continuous state space MDP. What level of discretization (i.e., binning) is necessary to compose an accurate discretized model of the MountainCar-v0 environment? An experiment involving a series of AUC curve evaluations was performed to answer this important question.

*Experimental Design*

A series of ten (10) 10,000-episode training runs were performed using various values of ‘NUM\_BINS’: 2, 6, 10, 14, 16, 18, 22, 26, 30, and 34. All other (previously optimized) variables were held constant over the course of the experiment. Variable settings: alpha=0.15, discount\_factor=0.95, epsilon=0.5, DIV\_FACTOR=26, is\_training=True}. Each experimental AUC curve was generated via 100-episode moving averages, collected over the course of agent training.

*Observations*

The outcomes of this experiment are shown below. At NUM\_BINS=2, the agent is unable to win the game at any point during the 10,000-episode training period. At NUM\_BINS=6, the agent is able to make some meaningful progress in completing the game successfully but demonstrates high variability that leads to a catastrophic failure that the agent cannot recover from. By NUM\_BINS=16, the agent has clearly hit some kind of stride in designing its’ policy and completes the game in c.a. 200 steps with only moderate variation. NUM\_BINS=26 is the clear winner of this experiment, achieving a steady win rate with near-zero variance by around 8,000 episodes.

A graph with a line

Description automatically generatedA graph with blue lines

Description automatically generated

A graph with blue lines

Description automatically generatedA graph with blue lines

Description automatically generated

A graph with blue lines

Description automatically generatedA graph with blue lines

Description automatically generated

A graph with blue lines

Description automatically generated A graph with blue line

Description automatically generated

A graph with blue lines

Description automatically generatedA graph with a line

Description automatically generated

**Figures 1-10.)** Side-by-side AUC graph comparison evaluating the optimal NUM\_BINS variable value for formulating a discrete approximation of the continuous MountainCar-v0 state space. NUM\_BINS values of 2 and 6 fail to train the model in any reliable sense. NUM\_BINS = 10, 14, and 16 show increasing accuracy defined in the q-learners’ model but exhibit a high level of variance, negatively impacting the end-of-training stability in these sessions. NUM\_BINS = 18 shows a substantial improvement in variance compared to NUM\_BINS=16, this is the first model capable of forming a stable training baseline. NUM\_BINS=26 is the clear winner, demonstrating high accuracy and performance by c.a. 7,000 iterations of gameplay. Variance appears to increase when NUM\_BINS extends past a range of ~26+/-2.

*Epsilon, Epsilon-Annealing, and Tuning for Optimal Policy Convergence*

An optimized value of epsilon was discovered in a similar manner as to the experiments involving “NUM\_BINS” (above), during these episodes, an epsilon value of 0.5 was discovered to be near optimal. Intermittent training failures were observed in epsilon values much lower than 0.5 and diminishing returns noted with epsilon values exceeding 0.5. Because epsilon is a self-annealing term, the agent’s behavior appears to more “intentional” with each iteration of gameplay. This is because the agent’s actions are progressively controlled by learned policy rather than stochasticity. Epsilon is initiated once at the start of every 10,000 round training session and diminishes by a value of epsilon\_decay\_value once per game until epsilon finally drops to zero.

*Choosing an Epsilon Decay Rate*

Epsilon controls the level of stochasticity (i.e., “randomness”) of the agent’s behavior through a comparison with a scaled random value generated at each time step. If the random value is less than epsilon (i.e., bounded by epsilon), the agent makes a random choice. If it is not, the agent makes the “greedy”, or policy-informed decision by exploiting the state-action pair associated with the largest value in the current q-table.

The choice of the epsilon annealing factor (AKA “epsilon\_decay\_rate”) is arguably of much greater importance than epsilon itself. This is because the parameter controls the extent and duration of the agent’s exploratory behavior, a prerequisite for the location of an optimal policy. [source]

One very useful technique for optimizing the epsilon\_decay\_rate term is to set it to a value that is proportional to some normalized fraction of epsilon itself. Interestingly, the performance of this model appears near optimal at an epsilon\_decay\_value equal to ~0.15% of epsilon (0.5\*0.0015 = 7.51E-4). Increasing this value further reduces model performance, setting the value of epsilon\_decay\_value lower than this results in diminishing returns.

Hill-Climbing Agent: Early Versions

More than one variation of the MountainCar agent was constructed over the course of this project. The early example was built for an earlier version of the MountainCar environment, which provided a single term state-value returned by env.step. This model is no longer supported by OpenAI Gym, however this early model played a significant role in learning about the dynamics of the MDP state environment. Results of early training runs using this model are shown in figures 12 and 13. See “q\_learner.py” in the project repository for more information.

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

A graph of different colored lines

Description automatically generated

Figure 12.) Early AUC training curves from the early predecessor of MountainCarAgent.py. This q-learning agent operates only in previous versions of the MountainCar environment that are not currently supported. The three most important variables in this q-learning model are “DISCRETE\_OBS\_SIZE”, “DIV\_FACTOR”, and “LEARNING\_RATE”, controlling state-space discretization, epsilon decay, and step size, respectively. See “q\_learner.py” for more information.

Back to MountainCarAgent.py…

**Defining the Agent to Act Based on Learned Policies**

MountainCarAgent.py is designed to exhibit alternative behaviors controlled by the parameters of its’ script function (i.e., **run(num\_episodes, is\_training, render)**). The game display monitor of MountainCarAgent can be turned on or off by setting the boolean ‘render’ variable to True or False.

The ‘num\_episodes’ variable sets the number of game episodes the agent will complete in the next training or test session. Setting ‘is\_training’ to True tells the agent to perform another round of q-policy training by overwriting the pickled q-table memoization file. An ‘is\_training’ value of False will tell the agent to refer to its’ current policy during all iterations of gameplay. This effectively “turns off” the agents’ exploratory behavior, telling it to invoke the learned policy instead of building a new one. Figure 13 shows the result of setting ‘is\_training’ to False after a successful round of training at 10,000 episodes of game training. The performance appears to stabilize somewhere around 135 steps per game, with a win-rate of 100%.

A graph showing a number of episodes

Description automatically generated with medium confidence

Figure 13.) AUC performance outcome for a pre-trained q-learning model in the MountainCar-v0 environment. The average number of timesteps needed for the agent to complete the course seems to hover around 136.

**Conclusion**

The development of the epsilon-greedy q-learning agent described in this article highlights many of the common considerations and difficulties in developing effective agents for OpenAI Gymnasium environments. Among these concerns are discretization choices that allow the agent to form an accurate approximation of continuous state graph environments in a computationally efficient manner. The results of these experiments have also shown the choice of an epsilon decay rate controlling the agent’s stochastic exploratory behaviors to be another consideration of primary importance.

The design considerations and parameter tuning completed during this project appear to suggest that this q-learner’s optimum performance in the MountainCar-v0 Gym environment lies somewhere near 130 time-steps per episode. The q-learning agent designed in this report may be extended to other MDP environments as well. An idea for future design improvement might involve adding new conditional behaviors to the run function that allow the agent to optimize its’ discrete approximation of a different Classic Control MDP environment, so that manual adjustment of parameters will no longer be necessary. The agent’s performance in alternative MDP environments could then be compared to other solving techniques (e.g., deep neural networks) based on overall performance and computational efficiency.

**Works Cited**

[1] Strategies for Decaying Epsilon in Epsilon Greedy (Policy Improvement)

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

[2] Mountain Car

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

[3] Reinforcement Learning algorithms – an intuitive overview

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

[4] Understanding Q-Learning: A Powerful Reinforcement Learning Technique

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

[5] Q-Learning Definition

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

[6] Ensemble Bootstrapping for Q-Learning

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