AMRITA NEOGI 05/08/2023

INFO - 550

Final Project

Using Q-Learning to solve the CartPole balancing problem.

GitHub Link: AmritaNeogi/AI_term_project.io: This is an AI project (github.com)

Introduction

Being a “model-free learning” algorithm, Q-Learning is regarded as a straightforward example of a reinforcement learning algorithm. Model-free algorithms enable agents to learn policies directly, as opposed to model-based algorithms, which depend on agents having prior environmental knowledge.

Q-Learning doesn't require a model or a challenging operational framework. Instead, it employs previously discovered “states” to take into account potential future moves and then stores this data in a “Q-Table”. For every action taken from a state, the policy table or the Q table, has to include a positive or negative reward.

The agent can make decisions based on the values assigned by Q-learning, which updates action-value functions for each state-action combination. Q-learning guarantees a convergence to the answer as long as exploration and value updates go on by continually updating these values.

The formula for this algorithm is shown below and can be easily understood by reading it from left to right.

$$Q^{new}(s_t, a_t) \leftarrow (1 - \underbrace{\alpha}_{\text{learning rate}}) \cdot \underbrace{Q(s_t, a_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \underbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}_{\text{new value (temporal difference target)}}$$

In this project I have built a neural network that can learn to play games through reinforcement learning. More specifically, we'll use Q-learning to train an agent to play a game called Cart-Pole. In this game, a freely swinging pole is attached to a cart. The cart can move to the left and right, and the goal is to keep the pole upright as long as possible.

This project comprises of the following components: Open AI Gym Environment setup, Random Baseline Strategy, Implementation of Deep Q Learning, Implementation of Deep Q Learning with Replay Memory, Q Learning Agent.

We have simulated our code using OpenAI Gym, Tensorflow and PyTorch.

Environment

CartPole-v1 is one of OpenAI's environments that are open source. In this environment, there exists a pole that runs over a frictionless track and is managed by a force of +1 or -1 applied to the cart. The pendulum starts upright with the intention of keeping it from falling over. Cart location, cart velocity, pole angle, and pole tip velocity are the four values that specify the state space. We can either go left or right in the action space. Each time step that the pole stays erect results in a reward of +1. When the cart moves more than 2.4 units from the centre or when the pole angle varies by more than 15 degrees from vertical, the episode is over**.**

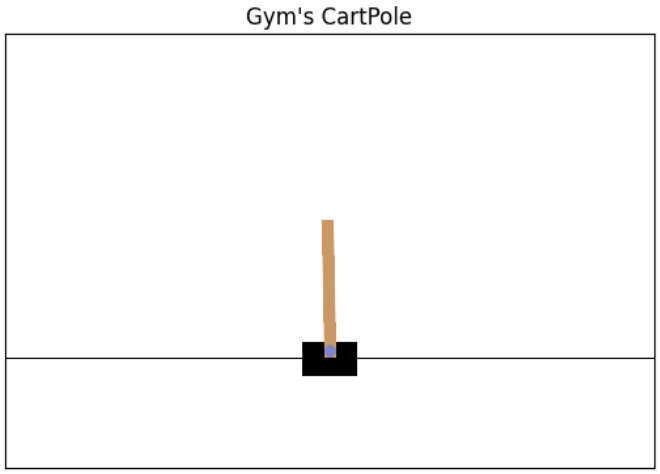
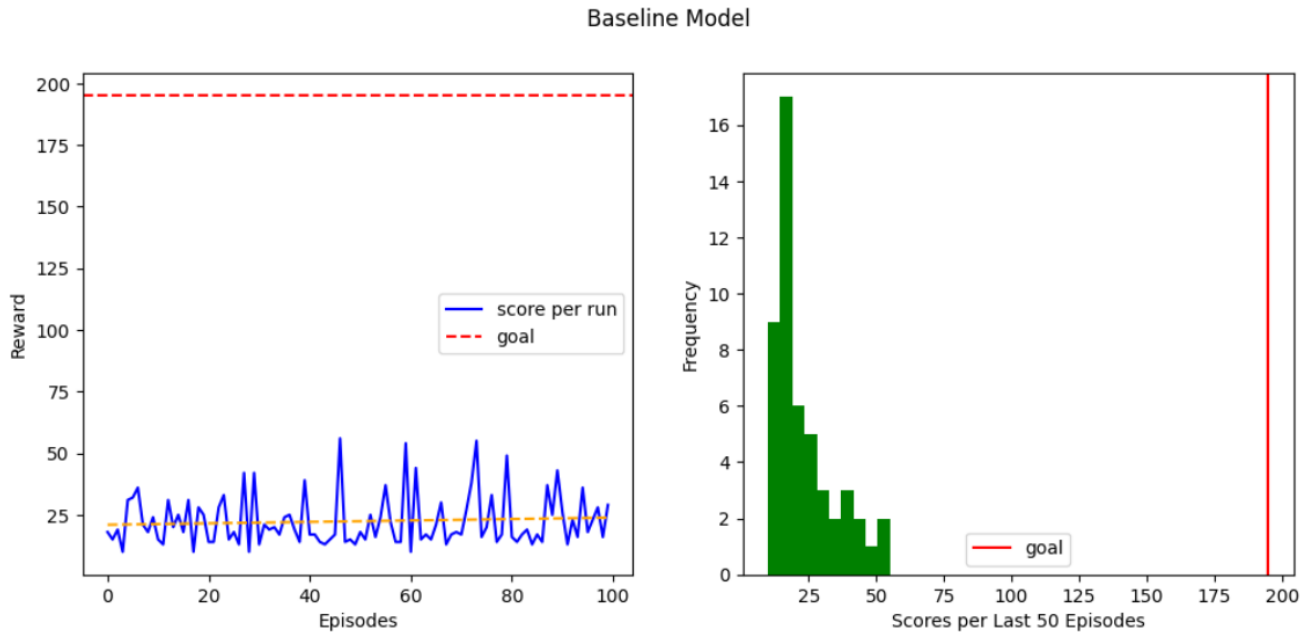


Fig: Cartploe-v1

Implementation and Results

To begin with, I created a straightforward strategy or a Baseline Model where the action is randomly chosen from the action space before putting any deep learning techniques into practice. This method will act as a starting point for additional approaches and make it simpler to comprehend how to interact with the agent in the Open AI Gym environment.

Several values, including the *next_state*, the *reward*, and if the simulation is *complete*, are returned by one environment step. The prize total across 150 episodes (simulation runs) is depicted in the plot below.



As expected, this strategy fails to address the environment. The agent is failing to gain knowledge from their mistakes. Despite occasionally being fortunate (receiving a reward of almost 75), their performance is only 10 steps above average.

The key concept underlying Q-learning is that we have a function called $Q: State \times Action \rightarrow R$ that can inform the agent of which actions will yield which rewards. It is possible to create a policy that maximizes rewards if we know the value of Q.

Since we don't have access to complete knowledge in the real world, we must devise methods for estimating Q. The values of Q can be changed after each action the agent takes using a lookup table, which is a common technique. However, this method is cumbersome and cannot handle vast action and state spaces. For this reason, I have trained a neural network that can approximate Q because they can be used to approximate any function.

I implemented a neural network used in the *DQL class* that uses PyTorch to execute its two primary methods: *update* and *predict*. The agent's state is provided to the network as an input, and it outputs the Q values for each action. The agent decides to take the next action by choosing the highest Q value.

The main loop for all subsequent algorithms is the *q_learning* function.

It has several parameters, including:

- The environment, called "CartPole," that we want to solve is represented by *env* in the Open Ai Gym.

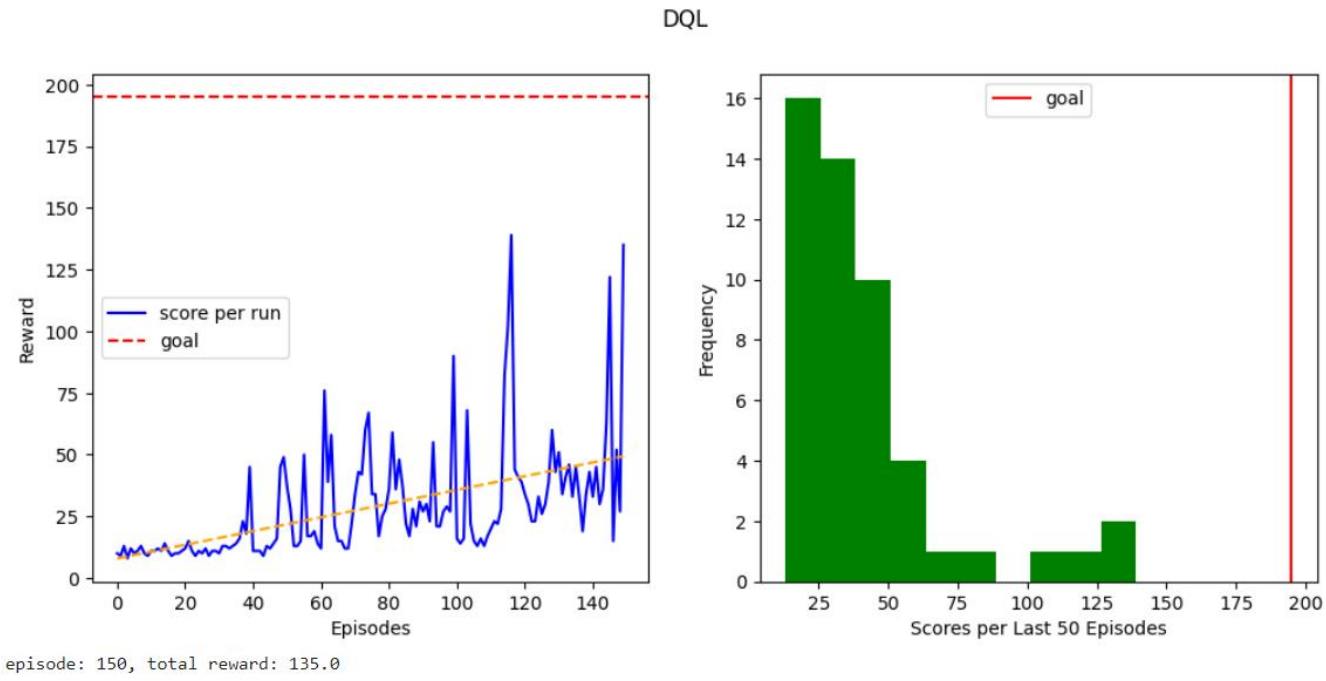
- *episodes* refer to the number of games we want to play.

- *gamma* is a factor used for discounting, which reduces the impact of future rewards on the agent. It makes rewards that are further in the future less valuable than those that are closer.

- *epsilon* is the proportion of random actions relative to the actions taken based on the accumulated knowledge of the agent. This strategy is called "Greedy Search Policy." Initially, the agent has no experience, so it is common to set epsilon to higher values and then gradually decrease it as it learns.

- *eps_decay* indicates how quickly epsilon decreases as the agent learns. The value of 0.99 is taken from the original DQN paper.

The simplest agent alters its Q-values in accordance with its most recent observation. It has no memory, but it learns by first investigating its surroundings and then gradually lowering the value of its epsilon so that it can make wise decisions.



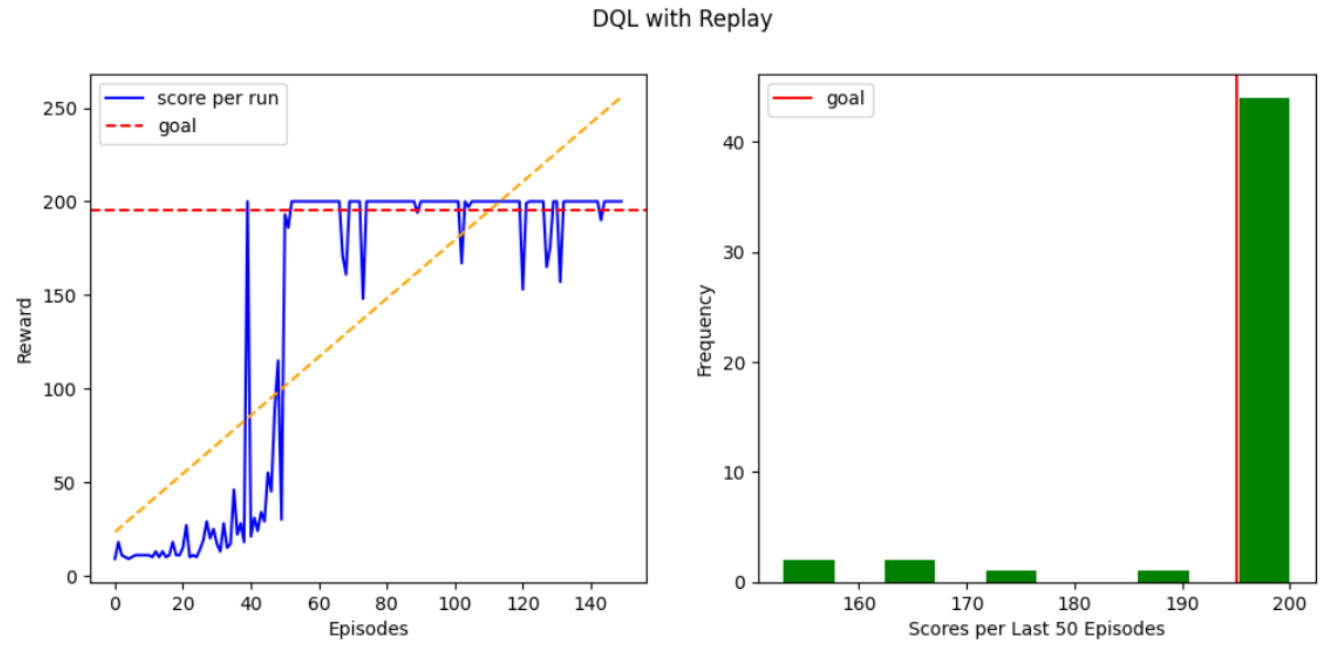
The graph up top demonstrates how much the agent's performance has advanced. It reached around 140 steps, which is impossibly many for a random agent, as we've already observed. We can see that

the performance improves over time because the trend line is likewise positive. However, the agent failed to cross the goal line after 150 attempts, and its average performance is still only about 15 steps, indicating that there is still potential for development.

Relay Memory

Attempting to approximate Q by utilizing a single sample at a time is not particularly successful. A good example of that is the graph above. Comparing the network's performance to a random agent, it was noticeably better. It was unable to cross the 195-step threshold, though. I put experience replay into place to increase network stability and make sure that prior experiences are not thrown out but rather used in training.

The experiences of the agent are retained in *memory* by experience replay. The neural network is trained using batches of experiences that are randomly selected from memory. Gaining experience and updating the model are the two phases of this type of learning. The amount of the replay that is used to update the network is determined by its size. Memory is an array that records the state, reward, and action of the agent as well as whether or not the action resulted in the game's completion and the next state.



As anticipated, compared to its counterpart that just recalls the most recent activity, the neural network with replay appears to be far more reliable and intelligent. The agent was able to cross the winning barrier and

maintain this level after roughly 60 episodes. Additionally, it succeeded in receiving the maximum award — 200.

Challenges and Solutions

There are few challenges when working with Q learning approaches. Firstly, this type of learning works well when there are few moves or the environment is simple, as the agent can easily remember past moves and repeat them. The Q-Table, however, can quickly fill up in more complicated contexts with numerous states, extending training times. The problem with Q-Learning is that it focuses more on certainties than on predictions; for each condition, it either knows the proper action to take or, in the case of an unfamiliar state, chooses at random.

Secondly, the environments with a large state-space provide a challenge for this technique because it depends on updating a function for each existing pair of state and action. This is due to the fact that as we visit a state-action pair more frequently, we become better at approximating its real value. It takes significantly longer to converge to the true values when we have a lot of states or actions to complete, so we distribute our visits among more pairs.

The CartPole environment describes the state in terms of continuous variables, such as cart position, velocity, angle, and pole velocity. To solve this problem, the states need to be *discretized*. Each variable is divided into multiple "buckets" to treat them as comparable states. The agent uses three buckets for cart position and velocity and six buckets each for the pole angle and velocity at the tip.

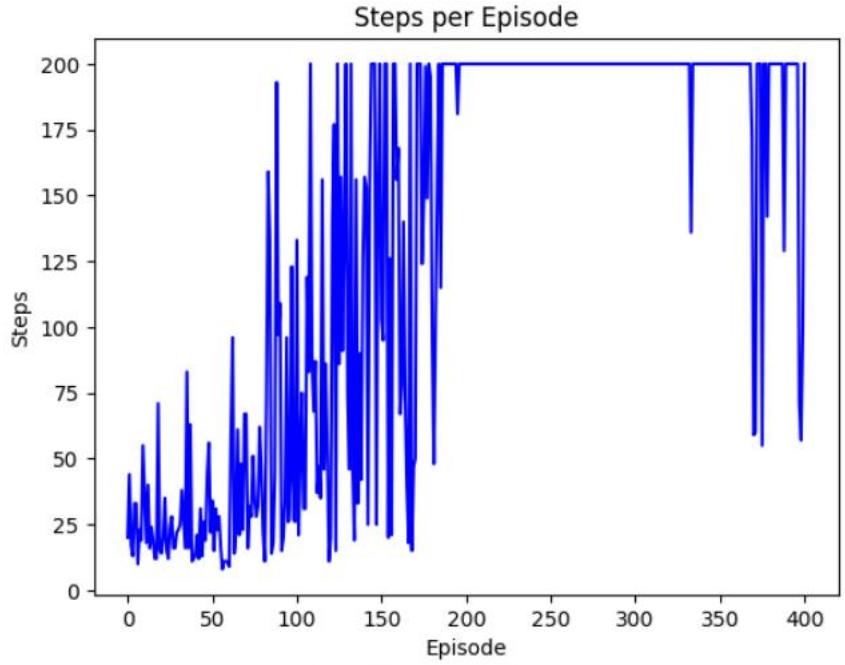
```
def discretize_state(self, obs):
    discretized = list()
    for i in range(len(obs)):
        scaling = ((obs[i] + abs(self.lower_bounds[i]))
                   / (self.upper_bounds[i] - self.lower_bounds[i]))
        new_obs = int(round((self.buckets[i] - 1) * scaling))
        new_obs = min(self.buckets[i] - 1, max(0, new_obs))
        discretized.append(new_obs)
    return tuple(discretized)
```

I developed a pseudocode for *CartPoleAgent()* and incorporated a function called *discretize_state()* into it. Using the *discretize_state(obs)* function, an observation from the outside world is converted into something

easier to handle. We have condensed the state space, made the Q-table smaller and simpler to fill by merging very similar states and treating them as the same.

Final Result

This instance of our agent was able to finish 314 episodes during training, which translates to more than 230 steps without losing the pole. It is clear that in the initial episodes, the incentives are kept modest as the agent learns the values for each state-action combination and continues to explore the state-space.




```
Number of successful episodes: 314
```

However, the agent's performance keeps getting better as we finish more episodes, and more episodes are finished successfully.

Conclusion and Discussion

The performance of a Deep Q Learning agent in the Open AI CartPole environment has been greatly enhanced by the introduction of the experience replay and the target network. To enhance the performance of the agent, other modifications could be made, such as Dueling Network Architectures (Wang et al., 2015).

In the second case, even though the agent appears to be operating well already, it is still possible to enhance its performance. We are compelled to feed our agent less information when we discretize the states, which is one of the drawbacks of this strategy, as was previously discussed. Our agent's performance could be affected by a variety of factors, not all of which are related to chance. We can modify the agent's parameters to significantly alter its performance, including the number of buckets for each variable defining the state, the rate at which the learning rate and epsilon decay, and the lowest value that epsilon and the learning rate can reach.

There are numerous additional methods that can also be used to solve these issues, each with unique advantages and disadvantages. In future we can use a SARSA and REINFORCE implementation for this problem. This could also be considered as a first step for Ensemble learning. Once we are able to comprehend how these algorithms are different from one another, we can use this technique to integrate various learning models to create a single learning that is more potent.

References:

[1] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., & De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581.

[2] Reinforcement Q-Learning from Scratch in Python with OpenAI Gym. (2019). Learndatasci.com. Retrieved 9 December 2019, from https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym

[3] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., … & Wierstra, D. (2015). Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971.

[4] Choudhary, A. (n.d.). [DQN Formula]. Retrieved September 15, 2020, from https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

[5] Using Q-Learning for OpenAI's CartPole-v1 | by Ali Fakhry | The Startup | Medium

[6] Wikipedia contributors. (2020, October 20). Q-learning. In Wikipedia, The Free Encyclopedia. Retrieved 00:03, October 29, 2020, from https://en.wikipedia.org/w/index.php?title=Q-learning&oldid=984486286

Disclaimer: The grammar and language in this article were corrected with the assistance of AI tools including Grammarly and Quillbot.