DeepQ Learning in Atari Games

RAFAEL GALDEANO BARSOTTI

Rio de Janeiro
2018
DeepQ Learning in Atari Games

"Declaro ser o único autor da presente monografia, requisito parcial para a obtenção do grau de Bacharel em Matemática Aplicada, e ressalto que não recorri a qualquer forma de colaboração ou auxílio de terceiros para realizá-lo a não ser nos casos e para os fins autorizados pelo professor orientador".

______________________________
Rafael Galdeano Barsotti

Orientador: Renato Rocha Souza

Rio de Janeiro
2018
Rafael Galdeano Barsotti

DeepQ Learning in Atari Games

“Projeto de Monografia apresentado à Escola de Matemática Aplicada – FGV/EMAp como requisito parcial para continuidade ao trabalho de monografia.”

Aprovado em ____ de __________ de _____.
Grau atribuído ao Projeto de Monografia: _______.

Professor Orientador: Renato Rocha Souza
Escola de Matemática Aplicada - FGV/EMAp
Fundação Getulio Vargas
Abstract

The aim of this paper is to develop an AI agent with self-learning capabilities that is able to play classical Atari console games without human intervention and achieve next to human level performance. In order to achieve our goal we will use the OpenAI Gym library that will provide us with a simulated atari game environment where we are able to collect important information regarding the agent and it’s environment. Initially our AI agent is expected to randomly explore the environment through the use of Monte Carlo Tree Search and build a Q-Table that will allow us to train a neural network to generalize past experiences and help the agent take the best possible action given the current state of the game.
Index

1 Introduction 5
2 Theoretical Background 6
3 Development and Results 8
  3.1 Frozen Lake ............................................................... 8
  3.2 Cart Pole ................................................................. 10
  3.3 Breakout Game ......................................................... 11
4 Conclusion 12
5 References 13
6 Appendix: Neural Networks 14
1 Introduction

Reinforcement learning techniques seek to develop general methods for situations where there is one or more agents interacting with each other and their environment, and through this interaction the agents try to achieve a certain objective. The importance of this kind of problem becomes self-evident when one considers the vast amount and diversity of situations where a self-learning agent capable of learning a task and automating it could be used. Although many diverse techniques have been developed in recent years an specific one named "DeepQ Learning" has been gaining ground and showing promising results. This technique consists of taking advantage of supervised learning methods like neural networks combined with brute force simulation.

In order to build our AI agent we will initially break down the duration of the game in regular time intervals, where at each instant the agent will be provided with a set of possible actions to choose. We will also create an empty Q table containing states, actions and their respective rewards initially populated with the results of our agent randomly exploring its environment. Then we will train our policy function using a deep neural network that takes as input a state and an action, and outputs its expected reward.
2 Theoretical Background

Figure 1: General Reinforcement Learning scheme

\[ S := \text{Set of All Possible States} \]

\[ A := \text{Set of All Possible Actions} \]

It is important to take into account the time which the agent made an action and whether the reward was instantaneous or delayed.

\[ \gamma := \text{Temporal Discount Factor} \]

We will now define \( Q \) as our action-value function:

\[ Q : S \times A \rightarrow \mathbb{R} \]

To simplify notation we concentrate on a single action. Let \( R_i \) now denote the reward received after the \( i \)th selection of this action, and let \( Q_n \) denote the estimate of its action value after it has been selected \( n \) times, which we can now write simply as:

\[ Q_n = \frac{R_1 + R_2 + \ldots + R_{n-1}}{n-1} \]

For computational efficiency purposes:
\[ Q_{n+1} = \frac{1}{n} \sum_{i=1}^{n} R_i \]
\[ = \frac{1}{n} (R_n + \sum_{i=1}^{n-1} R_i) \]
\[ = \frac{1}{n} (R_n + (n-1) \frac{1}{n-1} \sum_{i=1}^{n-1} R_i) \]
\[ = \frac{1}{n} (R_n + (n-1)Q_n) \]
\[ = \frac{1}{n} (R_n + nQ_n - Q_n) \]
\[ = Q_n + \frac{1}{n} [R_n - Q_n] \tag{1} \]

\[ Q_{n+1} = Q_n + \alpha [R_n - Q_n] \]
\[ = \alpha R_n + (1 - \alpha)Q_n \]
\[ = \alpha R_n + (1 - \alpha)\left[ \alpha R_{n-1} + (1 - \alpha)Q_{n-1} \right] \]
\[ = (a - \alpha)^n Q_1 + \sum_{i=1}^{n} \alpha (1 - \alpha)^{n-1} R_i \tag{2} \]

Choose action that maximizes discounted return:

\[ G_t = R_{t+1} + \gamma R_{t+2} + \gamma R_t + 3... = \sum_{i=1}^{\infty} \gamma^k R_{t+k+1} \]

Bellman Equation for tabular Q Values Update
\[ Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha (r_t + \gamma max Q(s_{t+1}, a)) \]
3 Development and Results

3.1 Frozen Lake

Figure 2: Frozen Lake Environment

The first environment that we are going to use to test our AI agent will be the Frozen Lake environment. The main goal of this game is to arrive at "G" starting from "S" without falling into the a hole "H". At each step our agent chooses to go left, right, up or down and receives a reward of 0 at each "F", −1 if it falls into a hole and +1 when it arrives at "G".

Figure 3: Q Value Table

This problem can be modeled as a \((S, A, R, S)\) Markov Decision Process where we initially create a 0 matrix of \(S \times A\) dimension and through random iteration update the Q values using the Bellman Equation.
```python
import gym
import numpy as np

# Load Environment
env = gym.make('FrozenLake8x8-v0')
Q = np.zeros([env.observation_space.n, env.action_space.n])

# Set Q Learning Parameters
eta = .628
gamma = .9
episodes = 5000
reward_list = []

# Q Learning Algorithm
for i in range(episodes):
    # Reset environment
    s = env.reset()
    r_total = 0
    done = False
    j = 0
    while j < 100:
        env.render()
        j += 1
        a = np.argmax(Q[s, :] + np.random.randn(1, env.action_space.n) * (1. / (i + 1)))
        s1, r, done, _ = env.step(a)
        Q[s, a] = Q[s, a] + eta * (r + gamma * np.max(Q[s1, :]) - Q[s, a])
        r_total += r
        s = s1
        if done:
            break
    reward_list.append(r_total)
    env.render()
print("Reward Sum on all episodes ", str(sum(reward_list) / episodes))
print("Final Values Q-Table")
print(Q)
```
3.2 Cart Pole

The second problem we are going to try to solve is of a more continuous nature. The objective of this game is to try and hold the pole as straight as possible while the car is receiving random shocks.

Figure 4: Cart Pole Environment
3.3 Breakout Game

The third and last game that we are going to try is the famous Breakout game.

Figure 5: Cart Pole Environment
4 Conclusion
5 References


6 Appendix: Neural Networks