Machine Learning Nanodegree
Capstone Project
A Game Bot trained with Deep Q-Learning

Peng Xu

February 14, 2016

1 Definition

1.1 Project Overview

Reinforcement Learning is a type of machine learning that allows you to create AI agents that learn from the environment by interacting with it. Just like how we learn to ride a bicycle, this kind of AI learns by trial and error. As seen in Figure 1, the brain represents the AI agent, which acts on the environment. After each action, the agent receives the feedback. The feedback consists of the reward and next state of the environment. The reward is usually defined by a human. If we use the analogy of the bicycle, we can define reward as the distance from the original starting point.

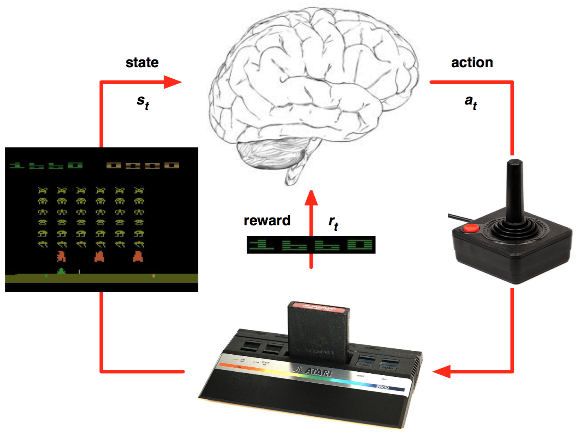

Figure 1: How an agent interacts with the environment.

Google's DeepMind published its famous paper Playing Atari with Deep Reinforcement Learning, in which they introduced a new algorithm called Deep Q Network (DQN for short) in 2013. It demonstrated how an AI agent can learn to play games by just observing the screen without any prior information about those games. The result turned out to be pretty impressive. This paper opened the era of what is called ?deep reinforcement learning?, a mix of deep learning and reinforcement learning.

1.2 Problem Statement

In this project, a deep reinforcement learning method, Deep Q Network, would be implemented and applied to play a Coast Racer game in OpenGym / Universe using TensorFlow.

Q-learning, a model-free online off-policy algorithm, whose main strength is that it is able to compare the expected utility of the available actions without requiring a model of the environment. In Q-Learning Algorithm, there is a function called Q Function, which is used to approximate the reward based on a state. We call it Q(s,a), where Q is a function which calculates the expected future value from state s and action a. Similarly in Deep Q Network algorithm, we use a neural network to approximate the reward based on the state. We will discuss how this works in detail.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{1}$$

OpenAI Gym is a toolkit for reinforcement learning research. It includes a growing collection of benchmark problems that expose a common interface, and a website where people can share their results and compare the performance of algorithms. Universe is a software platform for measuring and training an AI's general intelligence across the world's supply of games, websites and other applications. Universe allows an AI agent to use a computer like a human does: by looking at screen pixels and operating a virtual keyboard and mouse. We must train AI systems on the full range of tasks we expect them to solve, and Universe lets us train a single agent on any task a human can complete with a computer. With Universe, any program can be turned into a Gym environment. Universe works by automatically launching the program behind a VNC remote desktop. Hundreds of games have been translated into Gym environments and are ready for reinforcement learning, which mostly can be freely run with the universe Python library as follows:

```python
import gym
import universe # register Universe environments into Gym

env = gym.make('flashgames.DuskDrive-v0') # any Universe environment ID here
observation_n = env.reset()

while True:
  # agent which presses the Up arrow 60 times per second
  action_n = [[('KeyEvent', 'ArrowUp', True)] for _ in observation_n]
  observation_n, reward_n, done_n, info = env.step(action_n)
  env.render()
```

Among the several racing car games provided in Universe, the Coaster Racer flash game arose in front of me since it could be a typical simulation of Autonomous Driving, in which a vehicle is simply controlled by 3 inputs, left, right, forward. It is expected that the racing car can learn a smart driving behavior after a series of training steps leading to a maximal reward or namely score here.

1.3 Metrics

This metric consists in playing a high number of games (e.g. 100,000) against another agent (e.g. a random agent), and calculating the average of games won by the agent that uses the learned value function.

2 Analysis

2.1 Data Exploration

By using a Deep Q-Learning model, there is no need to use an external dataset since the data are captured along with training.

2.2 Exploratory Visualization

Skip.

2.3 Algorithms and Techniques

To tackle the problem described in Section **??**, we will use Reinforcement learning with Deep Learning to automatically learn evaluation functions by playing games by itself. Unlike other approaches that need a very large dataset, this approach will try to learn to play games without any domain knowledge (no dataset will be used). This is a promising approach for creating game-playing algorithms for playing other two-player games of perfect information.

2.4 Benchmark

This benchmark consists in playing against an agent that takes uniformly random moves. This is the most basic benchmark, but first we have to be sure that our learned evaluation function can play better than a random agent before moving into a harder benchmark. Also, this will help us to detect bugs in the code and algorithms: if a learned value function does not play significantly better than a random agent, is not learning. The idea is to test against this benchmark using Alpha-beta pruning at 1, 2 and 4-ply search.

3 Methodology

3.1 Data Preprocessing

3.2 Implementation

Train a Game Bot with Q Learning.
- Start Docker. $ source activate capstone

3.3 Refinement

4 Results

4.1 Model Evaluation and Validation

4.2 Justification

5 Conclusion

5.1 Free-Form Visualization

5.2 Reflection

5.3 Improvement