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.

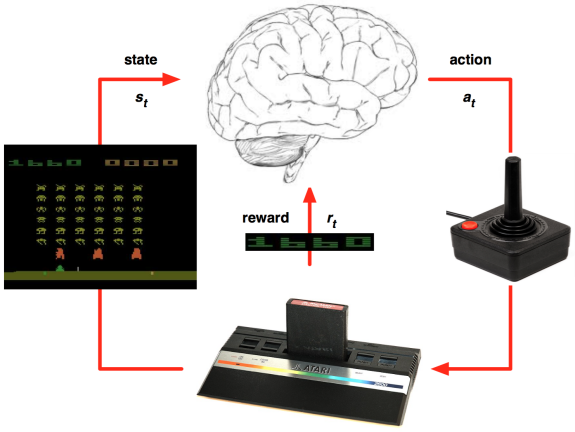

Figure 1: How an agent interacts with the environment.

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.

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:

```
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. The trained bot for the Coaster Racer flash game will determines whether it should turn and which way it turns.

Target function:

$Q^{\pi} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, where $\mathcal{S}$ is the set of *states* and $\mathcal{A}$ is the set of *actions* (turn left, forward or turn right), and $\mathbb{R}$ represents the value of being in a state $s \in \mathcal{S}$, applying a action $a \in \mathcal{A}$, and following policy π thereafter.

Target function representation:

Deep neural network.

Therefore, I seek to build a Q-learning agent trained via a deep convolutional neural network to approximate the optimal action-value function:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{1}$$

which is the maximum sum of rewards achievable by a behavior policy π.

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 Coaster Racer Environment

Coaster Racer Flash Game is a typical car racing game wrapped in OpenAI-Universe which is simply controlled by moving forward, left turn and right turn. With this simple setting, we only only an action space of three actions, namely, moving forward, left turn and right turn. The screen view as shown in Figure 2 is for the Gamebot. After preprocessing, the image as the bot see would be thrown into a Deep Convolutional Network and output an expected action. The Q function would compute a reward based on the current action. Through a long time learning or training, it would be expected that the Gamebot would grasp a reasonable driving behavior closing to human level.

Figure 2: A screenshot of Coaster Racer Game.

2.2 Algorithms and Techniques

To tackle the problem described in Section 1.2, 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.

2.2.1 Markov Decision Process

A *Markov decision process* (MDP) consist of four elements:

- $\mathcal{S}$ is the set of *states* (state space).
- $\mathcal{A}$ is the set of *actions* (action space). The set of actions that are available in some particular state $s_t \in \mathcal{S}$ is denoted $\mathcal{A}(s_t)$.
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the *transition function*, which is the probability given we are in state $s_t \in \mathcal{S}$, take action $a_t \in \mathcal{A}(s_t)$ and we will transition to state $s_{t+1} \in \mathcal{S}$.
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the *reward function*, which is the immediate reward received when in state $s_t \in \mathcal{S}$ action $a_t \in \mathcal{A}$ is taken and the MDP transitions to state $s_{t+1} \in \mathcal{S}$. However, it is also possible to define it either as $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ or $R : \mathcal{S} \to \mathbb{R}$. The first one gives rewards for performing an action a_t in a particular state s_t, and the second gives rewards when transitioning to state s_{t+1}.

2.2.2 Environment

In the reinforcement learning problem an agent does not have access to the dynamics (reward and transition functions) of the MDP. However, it interacts with an *environment* by way of three signals: a *state*, which describes the state of the environment, an *action*, which allows the agent to have some impact on the environment, and a *reward*, which provides the agent with feedback on its immediate performance.

2.2.3 Policy

In an MDP, the agent acts according to a policy π , which maps each state $s \in \mathcal{S}$ to an action $a \in \mathcal{A}(s)$. A policy that specifies a unique action to be performed is called a *deterministic* policy, and is defined as $\pi : \mathcal{S} \to \mathcal{A}$.

The interaction between the policy used by the agent and the environment works as follows. First, it starts at an *initial state* s_0. Then, the policy π selects an action $a_0 = \pi(s_0)$ from the set of available actions $\mathcal{A}(s_0)$, and the action is executed. The environmenttransitions to a new state s_1 based on the transition function T with probability $T(s_0, a_0, s_1)$, and a reward $r_0 = R(s_0, a_0, s_1)$ is received. This process continues, producing a *trajectory* of experience $s_0, a_0, s_1, r_1, a_1, s_2, r_2, a_2, \ldots$, and the process ends in a *terminal state* s_T and is restarted in the initial state.

We use three types of policies in this project:

- **Random.** Selects actions uniformly at random .
- **Greedy.** Selects the *max action*, which is the greedy action with the highest value,

$$\pi_{\text{greedy}}(s) = \underset{a \in \mathcal{A}(s)}{\operatorname{argmax}} Q(s, a) \tag{2}$$

- **ϵ-greedy.** Selects the best action for a proportion $1-\epsilon$ of the trials, and another action is randomly selected (with uniform probability) for a proportion,

$$\pi_\epsilon(s) = \begin{cases} \pi_{\text{rand}}(s, a) & \text{if } rand() < \epsilon \\ \pi_{\text{greedy}}(s, a) & \text{otherwise} \end{cases} \tag{3}$$

  where $\epsilon \in [0, 1]$ and $rand()$ returns a random number from a uniform distribution $\in [0, 1]$.

2.2.4 Value Functions

Most of the algorithms for solving MDPs (computing optimal policies) do it by learning a *value function*. A value function estimates what is good for an agent over the long run. It estimates the expected outcome from any given state, by summarizing the total amount of reward that an agent can expect to accumulate into a single number. Value functions are defined for particular policies.

The *state value function* (or V-function), is the expected return when starting in state s and following policy π thereafter (1),

$$V^\pi(s) = \mathbb{E}_\pi \left[R_t | s_t = s \right] \tag{4}$$

The *action value function* (or Q-function), is the expected return after selecting action a in state s and then following policy π,

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[R_t | s_t = s, a_t = a \right] \tag{5}$$

The *optimal value function* is the unique value function that maximises the value of every state, or state-action pair,

$$Q^*(s, a) \quad = \quad \max_\pi Q^\pi(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{6}$$

An *optimal policy* $\pi^*(s, a)$ is a policy that maximises the action value function from every state in the MDP,

$$\pi^*(s, a) = \underset{\pi}{\operatorname{argmax}} Q^\pi(s, a) \tag{7}$$

2.2.5 Q-learning

One of the most basic and popular methods to estimate action-value functions is the *Q-learning* algorithm. It is 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. Q-learning works by learning an action-value function that gives the expected utility of taking a given action in a given state and following a fixed policy thereafter. The update rule uses action-values and a built-in max-operator over the action-values of the next state in order to update $Q(s_t, a_t)$ as follows,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{8}$$

The agent makes a step in the environment from state s_t to s_{t+1} using action a_t while receiving reward r_t. The update takes place on the action-value a_t in the state s_t from which this action was executed. This version of Q-learning works well for tasks with a small a state-space, since it uses arrays or tables with one entry for each state-action pair.

2.2.6 Approximate Q-learning

In many cases in which there are far more states than could possibly be entries in a table we need to use function approximation. Approximate Q-learning consists in parameterizing an approximate action-value function, $Q(s, a; \theta_i) \approx Q(s, a)$, in which θ_i are the parameters (weights) of the action-value function at iteration i. Usually the number of parameters of a function approximator is much less than the state space, which means that a change in one parameter can change many Q-values, as opposed to just one as in the tabular case.

2.2.7 Experience Replay

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value function. One trick to make it work is to use *experience replay* , which consists in storing the experiences (s_t, a_t, r_t, s_{t+1}) at each time step t in a data set $D_t = \{e_1, \ldots, e_t\}$. During the training of approximate Q-learning random minibatches from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. The Q-learning update at iteration i uses the following loss function (2):

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a', \theta_i^-) - Q(s, a; \theta) \right)^2 \right] \tag{9}$$

where $(s, a, r, s') \sim U(D)$ is a sample minibatch of experience drawn uniformly at random from the memory pool of stores experiences.

2.2.8 Convolutional Neural Networks

Convolutional Neural Networks, or CNNs, are a special type of neural network that has a known grid-like topology. Like most other neural networks they are trained with a variant of the backpropagation algorithm. CNNs strengh is pattern recognition directly from pixels of images with minimal processing. We use a convolutional network as a function approximator for the game, since the actions are highly based on what would be seen as pixel matrix.

2.3 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

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:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{10}$$

3.1 Data Preprocessing

The original input is a 3 dimensional RGB image or matrix and the data preprocessing here is to downsize the matrix so the model can focus on the core features. The image is gray scaled and resized to as smaller size, [160, 120].

3.2 Implementation

Before getting started, we need to install Docker. Docker is a tool that lets you run virtual machines on the local computer. I created an image wrapping up everything necessary, like TensorFlow, OpenCV, Gym and Universe.

Either run the training code to train the model or the demo code to see it in action. The demo one loads the tensors from the TF checkpoint files and runs the model.

4 Results

4.1 Model Evaluation and Validation

The network was trained over 10,000 time steps and the behavior has been learned gradually. Compared to random bot, it has shown obvious better performance. For better or human level behavior, more training time steps are needed. Currently, I get something promising but not perfect enough.

4.2 Justification

5 Conclusion

5.1 Reflection

The most difficult aspect of this project was that is extremely hard to stabilize reinforcement learning with non-linear function approximators. There are plenty of tricks that can be used and hyperparameters that need to be tuned to get it to work, such as:

- **Q-learning**: exploration policy, discount factor, learning rate, number of episodes.
- **Experience Replay**: batch size, experience pool size.
- **Deep Neural Network**: convnets, number of layers, number of filters, number of kernels, optimizer (vanilla sgd, adam, rprop), learning rate, pooling, activation functions, etc.
- **ϵ-greedy**: fixed ϵ or decrease ϵ, initial value for ϵ, how many episodes to decrease it, final value.

All these techniques and parameters were selected by trial and error, and no systematic grid search was done due to the high computational cost. More than once it seemed that the implementation of the algorithms and techniques was incorrect, and it turned out that the wrong parameters were being used. A "simple" change such as decreasing ϵ, or changing the neural network optimizer made big changes in the performance of the value function.

5.2 Improvement

There are at least two aspects we can improve in the next stage,

1. **Better benchmarks:** In most of this project we used simple benchmarkers, such as playing against random players. While testing against random is probably the first thing to test against (if you can't beat a random player your learning algorithm is not working), it would be better to find a few heuristics and better players that can be used for testing.

2. **Incorporate other RL techniques:** The field of RL has been advancing fast in recent years. There are a few new and old techniques that I would like to try, such as asynchronous RL, double Q-learning, prioritized experience replay.

References

[1] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[2] V. Mni et al. Human-level control through deep reinforcement learning. *Nature*, pages 529–533, 2015.