Navigation with Deep-Q Learning

Xuehua Zhang

I. Introduction

We can solve many reinforcement learning problems, using solution methods that represent the action values in a small table. We referred to this table as a Q-table.

We can using neural networks to expand the size of the problems that we can solve with reinforcement learning. The Deep Q-Learning algorithm [1][2] represents the optimal action-value function q_* as a neural network (instead of a table).

Unfortunately, reinforcement learning is notoriously unstable when neural networks are used to represent the action values. The Deep Q-Learning algorithm addressed these instabilities by using two key features:

A. Experience Replay

When the agent interacts with the environment, the sequence of experience tuples can be highly correlated. The naive Q-learning algorithm that learns from each of these experience tuples in sequential order runs the risk of getting swayed by the effects of this correlation. By instead keeping track of a replay buffer and using experience replay to sample from the buffer at random, we can prevent action values from oscillating or diverging catastrophically.

The replay buffer contains a collection of experience tuples $(S, A, R, S\prime)$. The tuples are gradually added to the buffer as we are interacting with the environment.

The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

B. Fixed Q-Targets

In Q-Learning, we update a guess with a guess, and this can potentially lead to harmful correlations. To avoid this, we can update the parameters ω in the network $\widehat{q}$ to better approximate the action value corresponding to state S and action A with the following update rule:

$$\nabla\omega = \alpha \cdot \overbrace{\underbrace{(R + \gamma \max \widehat{q}(S', a, \omega^{-1})}_{\text{TD target}} - \underbrace{\widehat{q}(S, A, \omega))}_{\text{old value}}}^{\text{TD error}} \nabla_w \widehat{q}(S, A, \omega),$$

where ω^{-1} are the weight of a separate target network that are not changed during the learning step, and (S, A, R, S') is an experience tuple.

The remainder of the report is organized as follows. The deep Q-learning algorthim is presented in Section II. In Section III, we describe the enviroment, and present implementation in Section IV. The simulation results are provided in Section V. Finally, we conclude the report in Section VI.

II. Deep Q-Learning Algorithm

A. Architechure

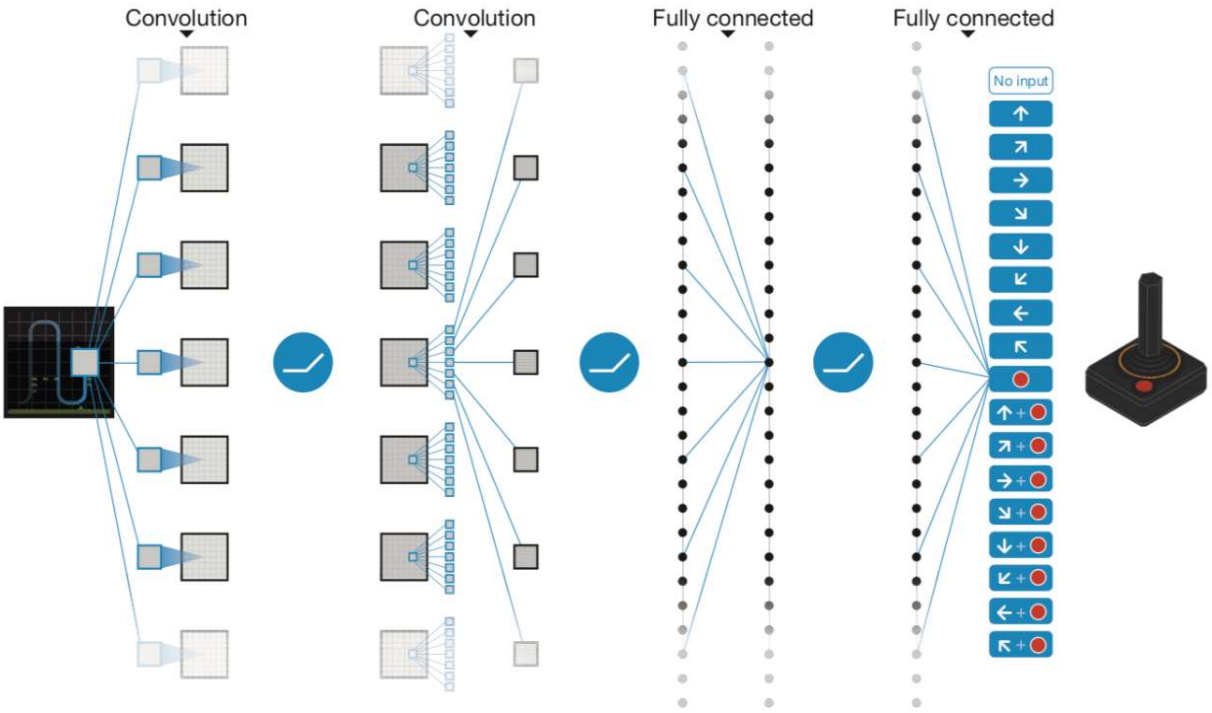

Fig. 1. Illustration of DQN Architecture

The input to the neural network consists of an $84 \times 84 \times 4$ image produced by the preprocessing map ϕ, followed by three convolutional layers and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is, $max(0, x)$).

B. Algorithm: Deep Q-Learning

1) INITIALIZE:

a) Initialize replay memory D with capacity N:

b) Initialize action-value function $\widehat{q}$ with random weights w. :

c) Initialize target action-value weights $w^{-1} \leftarrow w$:

d) for the episode $\epsilon \leftarrow 1$ to M::

e) Initial input frame x_1:

f) Prepare initial state: $S \leftarrow \phi(< x_1 >)$:

g) for time step $t \leftarrow 1$ to T::

2) SAMPLE:

a) Choose action A from state S using policy $\pi \leftarrow \epsilon-$Greedy$(\widehat{q}(S, A, w))$:

b) Take action A, observe reward R, and next input frame x_{t+1}:

c) Prepare next step: $S' \leftarrow \phi(< x_{t-2}, x_{t-1}, x_t, x_{t+1} >)$:

d) Store experience tuple (S, A, R, S') in replay memory D:

e) $S \leftarrow S'$:

3) LEARN:

a) Obtain random minibatch of tuples $(S_j, A_j, R_j, S'_{j+1})$ from D:

b) Set target $y_j = r_j + \gamma$max$_a \widehat{q}(S_{j+1}, a, w^{-1})$:

c) Update: $\nabla w = \alpha(y_j - \widehat{q}(S_j, A_j, w))\nabla_w \widehat{q}(S_j, A_j, w)$:

d) Every C steps. reset $w^{-1} \leftarrow w$:

III. The Enviroment

A. Unity ML-Agents

Unity Machine Learning Agents (ML-Agents) is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents.

For game developers, these trained agents can be used for multiple purposes, including controlling NPC (Non-player character) behavior (in a variety of settings such as multi-agent and

adversarial), automated testing of game builds and evaluating different game design decisions pre-release.

In this work, we use Unity's rich environments to design, train, and evaluate the deep-Q learning algorithm. You can read more about ML-Agents by perusing the GitHub repository in the following link: https://github.com/Unity-Technologies/ml-agents.

B. Description of the Enviroment

In this work, we train an agent to navigate (and collect bananas!) in a large, square world.

A reward of $+1$ is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of your agent is to collect as many yellow bananas as possible while avoiding blue bananas.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to:

0 - move forward.

1 - move backward.

2 - turn left.

3 - turn right.

The task is episodic, and in order to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes.

IV. Implementation

There are total three files: model.py, agent.py and Navigation.ipynb. model.py defines the neural network. The network consists of two hidden layers layers. The input of the neural network is state_size and the output is action_size. For the hidden layer, I set fc1_units=120, fc2_units=80 and fc3_units=58. Each hidden layer is followed by a Relu activation function. Agent.py define the Agent class and ReplayBuffer class. The main program is in Navigation.ipynb where to train an agent to navigate a large world and collect yellow bananas, while avoiding blue bananas.

V. Simulation Results

The following plot shows how the agent was able to learn and improve it's score through the episodes. The agent took 365 episodes to get an average of score of more than 13.

```
Episode 100     Average Score: 0.92
Episode 200     Average Score: 4.46
Episode 300     Average Score: 7.08
Episode 400     Average Score: 10.82
Episode 465     Average Score: 13.01
Environment solved in 365 episodes!     Average Score: 13.01
```

Fig. 2. The record of trainning

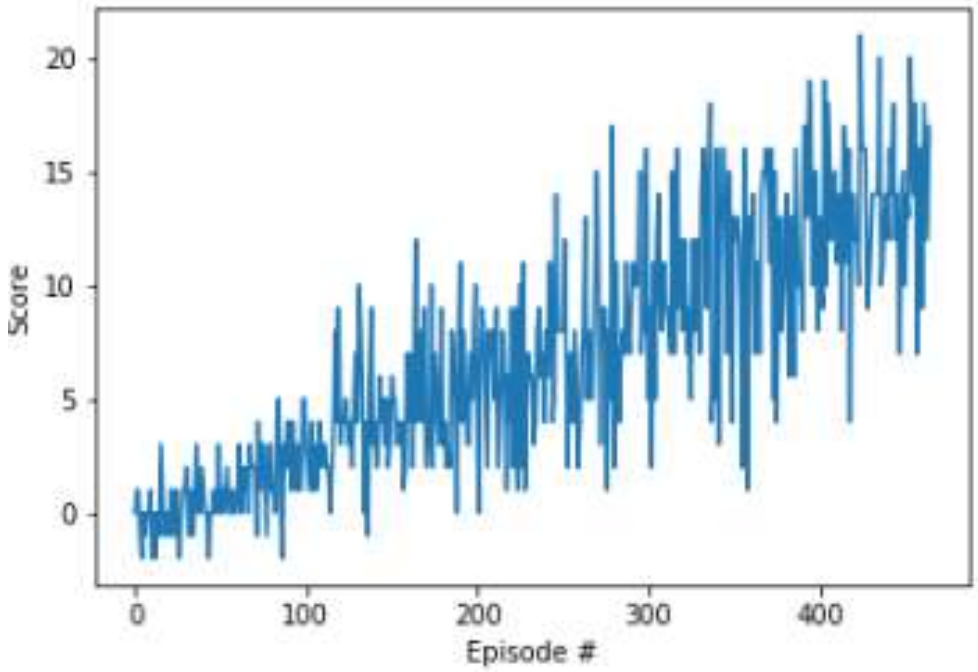

Fig. 3. Plotted Rewards

VI. Future Work

There are two things I would like to try as the future work. Firstly, as we know, several improvements to the original Deep Q-Learning algorithm have been suggested. They are Double DQN, Prioritized Experience Replay and Dueling DQN. I would like to try their performance and compare them to the original Deep Q-Learning algorithm. Secondly, I would like to try the more challenging task where the agent learn directly from pixels.

REFERENCES

[1] Riedmiller, Martin, "Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method," *European Conference on Machine Learning*. Springer, Berlin, Heidelberg, 2005.

[2] Mnih, Volodymyr, et al, "Human-level control through deep reinforcement learning," Nature518.7540 (2015): 529.