Using deep Q-learning with Unity ML-Agents toolkit

by Wes Brewer

December 27, 2018

Introduction

The goal of this project is to use deep reinforcement learning to develop an agent that will learn how to be able to score an average of at least +13 points over a window of 100 episodes. The environment used is the Unity ML-Agents Banana Collector environment. The goal of the game is to collect yellow bananas while avoiding blue bananas.

In this environment, there are four possible actions:

0 - walk forward
1 - walk backward
2 - turn left
3 - turn right

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. A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana (Udacity, 2018).

Methods

Note: that I started with the dqn_agent.py and model.py from https://github.com/udacity/deep-reinforcement-learning/tree/master/dqn/solution and adapted the training section, and "watch the trained agent" sections from the Jupyter notebook.

Neural Network Architecture. The method we use to solve this problem is to train a deep Q-network (DQN) that will learn optimal policies that map an input vector of 37 states to four possible actions (forward, backward, left, right). The neural network architecture that was chosen used two hidden fully connected layers, each containing 64 neurons as shown here:

Input: State (37)
FCN Layer (64)
ReLU activation
FCN Layer (64)
ReLU activation
Output: Action (4)

The loss function. PyTorch deep learning platform (pytorch.org) was used to implement the neural network. Neural networks are trained by minimizing an objective function, which is usually a functional difference between the target (predicted) value and the expected (ground truth) value.

We used mean squared error between the target and expected values. In this case, we estimate the action-value function by using the Bellman equation iteratively (Minh et al., 2015). So, the target value is computed as:

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a')\right]$$

Where *r* are the rewards, γ is the discount factor, *s'* and *a'* are the next state and action, and the maximum function here is used to select the action *a'* which maximizes the target $Q^*(s',a')$. The mean squared loss is used as the loss function, between the target and expected Q values:

$$L_i = \frac{1}{n}\sum_{i=1}^{n}(Q_i^* - Q_i)^2$$

Where the summation here is over a batch size of *n* training samples. These two can be combined to arrive at the following Bellman equation which is given in the seminal work of Mnih et al. (2015):

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

Experience Replay: A technique known as *experience replay* maintains a buffer of experiences (buffer size of 100,000) which may be reused to train the Q-network. Each step that is taken in the game is added to the replay buffer. A randomly sampled subset (of size 64 samples) are drawn from the replay buffer to train the DQN**.**

Exploration vs. Exploitation: To explore the exploration vs. exploitation trade-off space, an ε-greedy algorithm was used, which starts at a value of 1.0 (pure exploration), and then linearly reduces by 0.5% percent over each episode to a minimum value of 0.01 (almost pure exploitation).

Hyperparameters. Table 1 provides a list of all the hyperparameters that were used in the present study.

Table 1: DQN Hyperparameters

Parameter	**Value**
Replay buffer size	1e5
Batch size	64
Discount factor (γ)	0.99
Learning rate (α)	5e-4
Update frequency	4
FC1 units	64
FC2 units	64
Activation function	ReLU
Total number of episodes	2000
Max steps per episode	1000
ε start	1.0

ε end	0.01
ε decay rate	0.995

Results and Discussion

The model was trained using 2000 episodes. Figure 1 shows a history of the score over the entire training procedure, and the following shows the average score per 100 episodes:

```
Episode 100    Average Score: 1.41
Episode 200    Average Score: 4.98
Episode 300    Average Score: 10.00
Episode 400    Average Score: 11.86
Episode 500    Average Score: 12.58
Episode 600    Average Score: 14.57
Episode 700    Average Score: 14.89
Episode 800    Average Score: 15.39
Episode 900    Average Score: 15.94
Episode 1000   Average Score: 15.80
Episode 1100   Average Score: 15.94
Episode 1200   Average Score: 15.88
Episode 1300   Average Score: 16.29
Episode 1400   Average Score: 16.81
Episode 1500   Average Score: 15.80
Episode 1600   Average Score: 15.70
Episode 1700   Average Score: 16.33
Episode 1800   Average Score: 16.26
Episode 1900   Average Score: 16.57
Episode 2000   Average Score: 16.53
```

The project requirements required an average score greater than 13 over a period of 100 episodes. This was achieved by episode 600. A maximum average score of 16.81 was achieved by episode 1400. Over the course of the entire training span, the maximum single episode score achieved was 28.

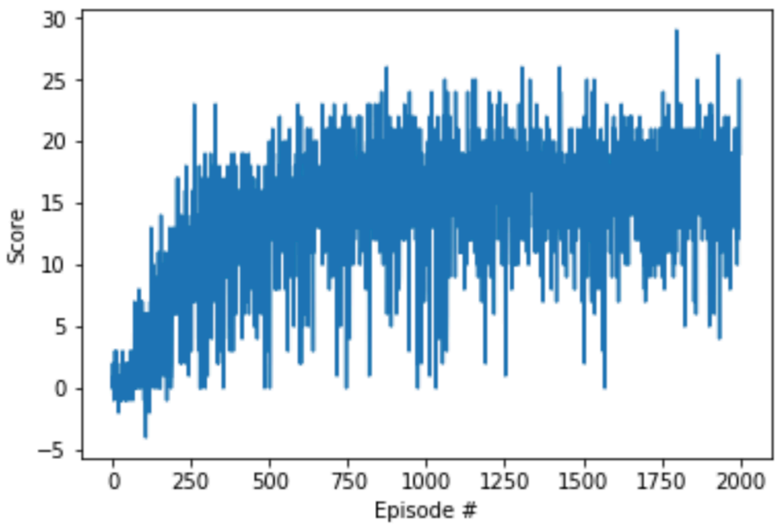

Figure 1: Plot of cumulative rewards / score over time.

A video of the final trained agent can be viewed at: https://youtu.be/m7aOodyDlkk

Future Work

This project represents very preliminary research into using deep Q-learning (DQN) for reinforcement learning. Future work would entail investigating implementing Double DQN, prioritized experience replay and Dueling DQN which are briefly described here.

One of the problems that Q-Learning suffers from is that it tends to overestimate action values (Van Hasselt et al., 2016). To alleviate this problem, Van Hasselt et al. (2016) have shown that using a technique called Double DQN reduces the overestimations.

In this work experiences were uniformly sampled from the replay buffer. *Prioritized* experience replay adds more weight to significant transitions so that they replay more frequently. The way this is implemented is by ranking the experiences according to their temporal-difference (TD) error, resulting in a more robust, performant, and efficient algorithm (Schaul, 2015). Experience replay helps to break the correlation between consecutive experiences and stabilizes the learning algorithm (Chakraborty, 2018).

Dueling DQNs separate the output of the DQN to separately estimate the state-value and the value advantage. The main advantage of using dueling DQNs is better generalization of learning across actions (Wang et al., 2015).

References

1. Udacity, "Navigation Project: the Environment – Introduction", Udacity's Deep Reinforcment Learning course, https://udacity.com (2018).
2. Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *Nature* 518.7540 (2015):529.
3. Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." *AAAI.* Vol. 2. 2016.
4. Schaul, Tom, et al. "Prioritized experience replay." *arXiv preprint arXiv:1511.05952* (2015).
5. Chakraborty, Arpan. "Prioritized Experience Replay V1", https://www.youtube.com/watch?v=cN8z-7Ze9L8 (2018).
6. Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." *arXiv preprint arXiv:1511.06581*(2015).