Udacity DRLND Project #1 - Navigation

Jing Zhao 9/25/2018

Introduction

In this report, we provide a description of the implementation details of a deep Q-network for solving the navigation (banana collection) problem in the Unity ML-Agents environment. We start off by introducing the environment. Then we talk about the learning algorithm – specifically, the deep double Q-learning algorithm. Next, we move on to displaying the results we got and discuss some interesting findings. We conclude our study by briefly explaining two ideas that may further improve our agent's performance.

Environment

The environment is provided by DRLND's course website. For this project, I am asked to 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 my agent is to collect as many yellow bananas as possible while avoiding blue bananas. Figure 1 is a screenshot of the environment.

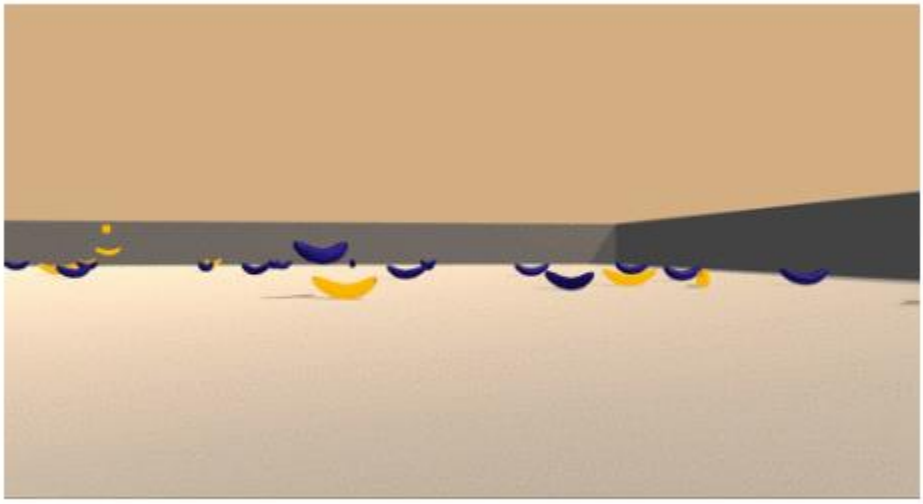

Figure 1. A screenshot of the Navigation (banana collection) environment.

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, my agent must get an average score of +13 over 100 consecutive episodes.

Learning Algorithm

We start off by using the basic deep Q-network introduced by Mnih in 2015 [1]. The two significant improvements of this algorithm over traditional online Q-learning are experience replay and fixed target network. A pseudocode from [1] is shown below for reference.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

The above deep Q-learning algorithm is then implemented in two python scripts: *dqn_agent_double.py* and *model.py*. In *dqn_agent_double.py*, we build two classes: Agent and ReplayBuffer. Agent defines interaction with and learning from the environment. ReplayBuffer is essentially a fixed-size buffer to store experience tuples. In *model.py*, we set up a neural network architecture to approximate state-action mapping that represents the optimal action-value function.

Double Q-learning algorithm has been shown to be effective in reducing overestimation in action values, thus resulting in improved performance on several Atari 2600 games [2]. The target used by a standard deep Q network (DQN) is

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^{-})$$

Note that the same values are used both to select and evaluate an action, which in turn may result in overoptimistic value estimates. To overcome this limitation, authors in [2] proposed a new target:

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\text{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-)$$

In doing so, the local (online) Q-network (θ_t) is used to select an action while the target network (θ_t^-) that is updated every τ steps is used to evaluate the Q-value. This change in Y_t is implemented in line 88-94 in *dqn_agent_double.py*.

We build a 3-layer feedforward neural network (NN) to approximate the optimal action value function. Table 1 shows the hyperparameters of this network along with some other parameters in the double Q-learning algorithm.

Table 1. Hyperparameters of Double DQN and other learning parameters

Parameter	Description	Value
State_size	size of NN input= dimension of the environment	37
Action_size	size of NN output= dimension of the action space	4
BUFFER_SIZE	Replay buffer size	100,000
BATCH_SIZE	Minibatch size	64
GAMMA	Discount factor	0.99
TAU	For soft update of target parameters	1e-3
LR	Learning rate	5e-4
UPDATE_EVERY	How often to update the target network	4
N_episodes	Total episodes in training	2000
Eps_start	Start value of epsilon decay	1
Eps_end	End value of epsilon decay	0.01
Eps_decay	(Exponential) decay factor of epsilon	0.995

Results

Once the double Q-learning algorithm is programmed in Python, we started training the AI agent and terminated simulation after 2000 episodes. Figure 2 below shows the score of each episode (in blue) as well as the moving-average across past 100 episodes (in red dash line). According to the project description, this environment is "solved" once the agent is able to receive an average reward (over 100 episodes) of at least +13. We achieved this goal in episode 486 meaning we solved the environment in episode 386. If we raised the bar from +13 to +15, the environment would be solved in 589 episodes. We saved the weights in a checkpoint file named "checkpoint_doubleQ_SOLVED-689.pth". Additionally, weights corresponding to episode 500, 100, 1500 and 2000 are also saved in the same directory. Interestingly, not much benefit has been gained through prolonged training. Performance of the agent fluctuates around 15.5 after episode 700. It's also worth noting that this environment would be solved in over 500 episodes if the standard DQN was used assuming all parameters stay the same. Therefore, we did see performance improvement by choosing a Double DQN over the standard DQN.

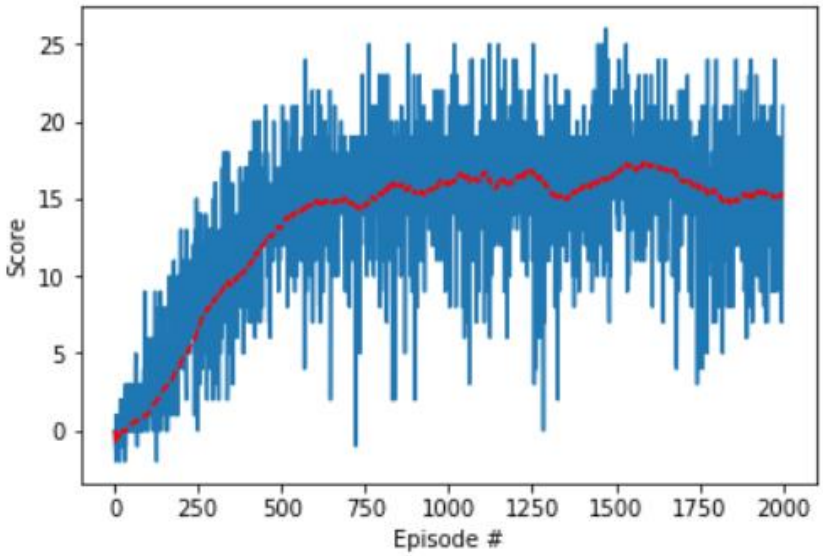

Figure 2. Scores (blue) and 100-point moving average (red) of training episodes using a Double Q-learning algorithm. The environment is solved (100 episode moving average > +13) in 386 episodes.

Ideas for Future Work

The course also introduced other two "tricks" that may improve agent's overall performance: Dueling DQN [3] and Prioritized Experience Replay [4]. The idea behind dueling DQN is that the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This would essentially focus the agent on learning from those valuable states. Prioritized experience replay provides similar "hints" to the agent by replaying important experiences more frequently than those unimportant ones. The "importance" is measured by the magnitude of temporal-difference of each learning update. I have not got time to implement these two ideas yet, but based on the course material, they are likely to enhance the agent's performance even further.

Reference

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. *Human-level control through deep reinforcement learning*. Nature, 518 (7540):529–533, 2015.

[2] van Hasselt, Hado, Guez, Arthur, and Silver, David. *Deep Reinforcement Learning with Double Q-learning*. In Proceedings of the 13th AAAI Conference on Artificial Intelligence, 2016.

[3] Wang, Z., Schaul T., Hessel, M., van Hasselt, H., Lanctot, M., de Freitas, N., *Dueling Network Architectures for Deep Reinforcement Learning*

[4] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. *Prioritized experience replay*. In ICLR, 2016.