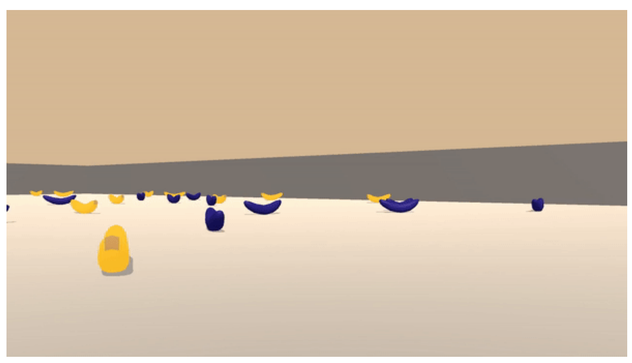
README.md

# Udacity Deep Reinforcement Learning Nanodegree - Project 1: Navigation



## Introduction

In this project, I built an reinforcement learning (RL) agent that navigates an environment similar to Unity's Banana environment.

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

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.
```

## Getting Started

### Install Dependencies

```
cd /python
pip install .
```

### Instructions

- Run `python3 navigation_watch_trained_agent.py` to see the trained agent in action.
- Run `python3 navigation_watch_random_agent.py` to see an untrained agent performing random actions.
- Run `python3 navigation_train_agent.py` to re-train the agent.
- `model.py` defines the Deep Q-Networks's architecture.
- `agent.py` defines the DQN agent class.
- `checkpoint.pth` are the saved network weights after training.

## Deep Q-Network (DQN) Learning Algorithm

The following sections describe the main components of the implemented DQN algoirthm.

### Q-Function

The Q-function calculated the expected reward `R` for all possible actions `A` in all possible states `S`. The optimal policy `π*` then is the action that maximizes the Q-function for a given state across all states. The optimal Q-fucntion `Q*(s,a)` maximizes the total expected reward for an agent following the optimal policy for each subsequent state. Furthermore, the Q-function can be expanded to discount returns at future time steps, setting an hyperparameter gamma `γ`.

### Epsilon Greedy Algorithm

For systematically managing the exploration vs. exploitation trade-off, I implemented an ε-greedy algorithm (see `agent.act()`), where the agent "explores" by picking a random action with some probability epsilon \epsilon, but continues to "exploit" its knowledge of the environment by choosing actions based on the policy with probability (1-ε).The value of ε is decaying over time, so that the agent favors exploration during its initial interactions with the environment, but increasingly favors exploitation as it gains more experience.

### Network Architecture

In Deep Q-Learning, a deep neural network is used to approximate the Q-function. Given a network F, finding an optimal policy is a matter of finding the best weights w such that F(s,a,w) ≈ Q(s,a).

The neural network architecture (`model.py`) is implemented via PyTorch and contains three fully connected layers with 64, 64, and 4 nodes respectively.

As for the network inputs, rather than feeding-in sequential batches of experience tuples, I randomly sample from a history of experiences using an approach called Experience Replay.

### Experience Replay

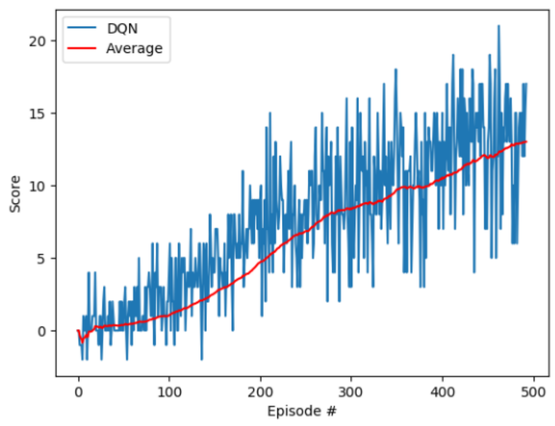
Experience replay allows the RL agent to learn from past experience, by storing filling a replay buffer as the agent interacts with the environment (see `agent.step()`) The replay buffer contains a collection of experience tuples with the state, action, reward, and next state (s, a, r, s').

## Results

The implemented RL agent is able to solve the environment in <400 episodes:

```
Episode 50      Average Score:  0.386
Episode 100     Average Score:  1.00
Episode 150     Average Score:  2.53
Episode 200     Average Score:  4.66
Episode 250     Average Score:  6.81
Episode 300     Average Score:  8.42
Episode 350     Average Score:  9.64
Episode 400     Average Score:  10.42
Episode 450     Average Score:  11.90
Episode 493     Average Score: 13.01

Environment solved in 393 episodes!     Average Score: 13.01
```




## Ideas for Future Improvements

- Implement DQN improvements (e.g prioritized experience replay, dueling DQN etc.)
- Directly learn from pixels instead of ray-based perception