Udacity Deep Reinforcement Learning Nanodegree

Project 1: Navigation

Introduction

This project repository contains Nishi Sood's work for the Udacity's Deep Reinforcement Learning Nanodegree Project 1: Navigation. For this project, I have trained an agent to navigate (and collect bananas!) in a large, square world.

Project's goal

- The environment is square world filled with yellow and blue bananas.
- A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana.
- The goal is to pick up the yellow bananas and avoid the blue ones.
- The task is episodic, and the end goal is to reach an average score of +13 over 100 consecutive episodes.

Project's Environment

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

- 0 - move forward.
- 1 - move backward.
- 2 - turn left.
- 3 - turn right.

Project Environment Setup/ Configuring Dependencies:

Step 1: Clone the DRLND Repository and installing dependencies

1. Set Up for the python environment to run the code in this repository

Environment

```
conda create --name drlnd python=3.6
source activate drlnd
```

2. Install of OpenAI gym packages

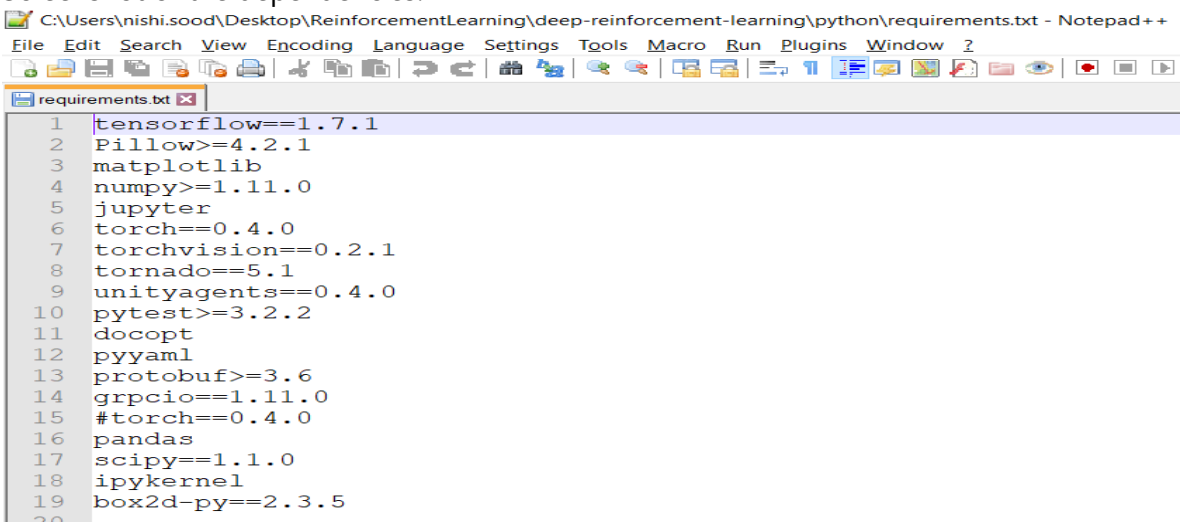
OpenAI gym

git clone https://github.com/openai/gym.git
cd gym
pip install -e .
pip install -e .[classic_control]
conda install swig
pip install -e .[box2d]

3. Other Dependencies

git clone https://github.com/udacity/deep-reinforcement-learning.git
cd deep-reinforcement-learning/python
pip install .

Screenshot of the dependencies:



NOTE: List of the installed dependencies can be found in "requirements.txt" attached with the project.

Trouble shooting:

If you get the error for installing torch library:

On Windows:

1. Open deep-reinforcement-learning\python\requirements.txt
2. Remove the line torch==0.4.0

On Anaconda:

3. conda install pytorch=0.4.0 -c pytorch

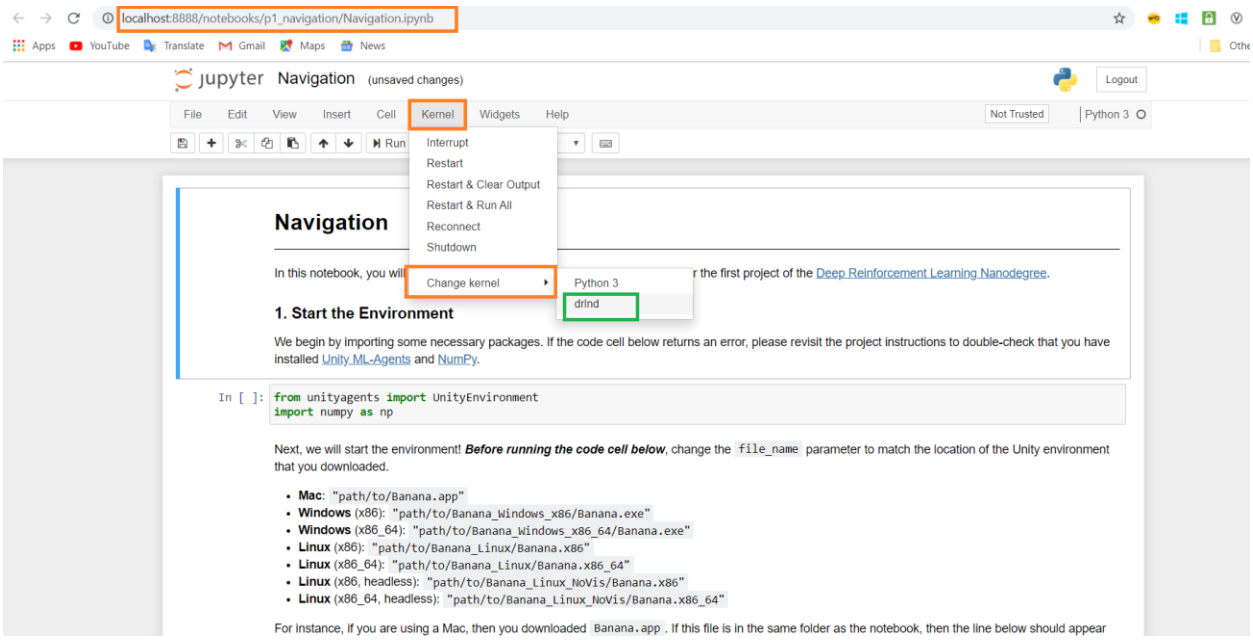
4. cd deep-reinforcement-learning/python

pip install .
conda install pytorch=0.4.0 -c pytorch

4. Create an IPython kernel for the drlnd environment

Kernel
python -m ipykernel install --user --name drlnd --display-name "drlnd"

5. Before running code in a notebook, change the kernel to match the drlnd environment by using the drop-down Kernel menu.



Step 2: Download the Unity Environment

1. Download the environment from one of the links below. You need only select the environment that matches your operating system:

   - Linux: click here
   - Mac OSX: click here
   - Windows (32-bit): click here
   - Windows (64-bit): click here

2. Place the file in the DRLND GitHub repository, in the p1_navigation/ folder, and unzip (or decompress) the file.

Description

- **`dqn_agent.py`:** code for the agent used in the environment
- **`model.py`:** code containing the Q-Network used as the function approximator by the agent
- **`Navigation_EnvironmentExploration.ipynb`:** explore the unity environment and verify the setup
- **`Navigation_Solution.ipynb`:** notebook containing the solution (Eeach cell to be executed to train the agent)
- **`Navigation_Pixels.ipynb`:** notebook containing the code for the pixel-action problem
- `dqn.pth`: saved model weights for the original DQN model
- `ddqn.pth`: saved model weights for the Double DQN model
- `Duelingddqn.pth`: saved model weights for the Dueling Double DQN model
- **Report.md:** The submission includes a file in the root of the GitHub repository that provides a description of the implementation.

Steps to Run the Project

1. Ensure that he Banana.exe is at the same level as the Navigation.ipynb

2. Use jupyter to run the Navigation.ipynb notebook: jupyter notebook Navigation.ipynb

3. Follow the instructions in Navigation.ipynb to get started with training your own agent.

4. To train the agent run the cells in order. They will initialize the environment and train until it reaches the goal condition of +13.

5. A graph of the scores during training will be displayed after training.

6. To watch a trained smart agent, follow the instructions below:

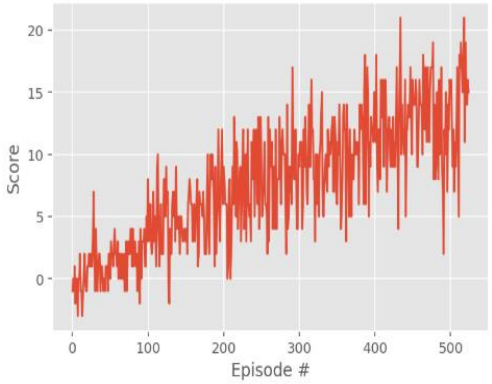
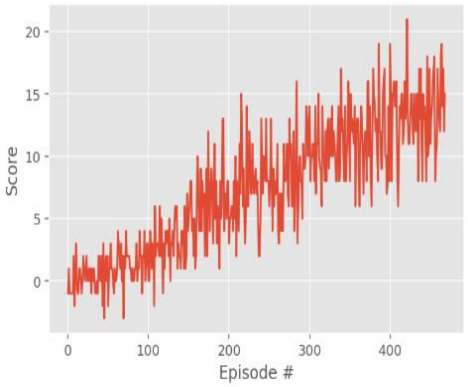
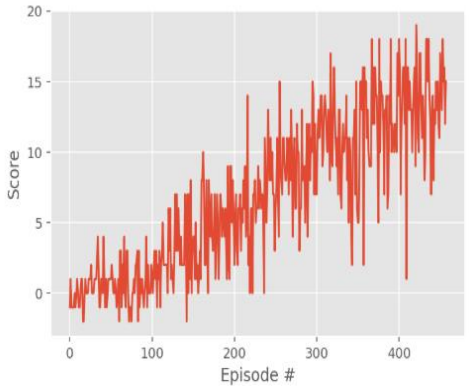
- **DQN**: If you want to run the original DQN algorithm, use the checkpoint `dqn.pth` for loading the trained model. Also, choose the parameter `qnetwork` as `QNetwork` while defining the agent and the parameter `update_type` as `dqn`.
- **Double DQN**: If you want to run the Double DQN algorithm, use the checkpoint `ddqn.pth` for loading the trained model. Also, choose the parameter `qnetwork` as `QNetwork` while defining the agent and the parameter `update_type` as `double_dqn`.
- **Dueling Double DQN**: If you want to run the Dueling Double DQN algorithm, use the checkpoint `dddqn.pth` for loading the trained model. Also, choose the

parameter `qnetwork` as `DuelingQNetwork` while defining the agent and the parameter `update_type` as `double_dqn`.

Results

Plot showing the score per episode over all the episodes.

The best performance was achieved by **Dueling DQN** where the reward of +13 was achieved in **358** episodes.

Double DQN (Episodes:425)	DQN(Episodes:369)	Dueling DQN(Episodes:358)
		

Report.md

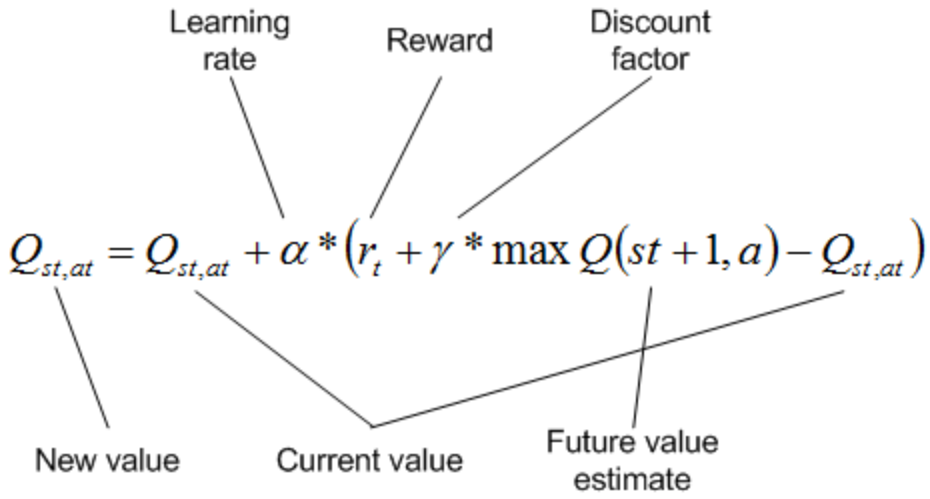
Deep Reinforcement Learning : Project 1:Navigation Report

Author : Nishi Sood

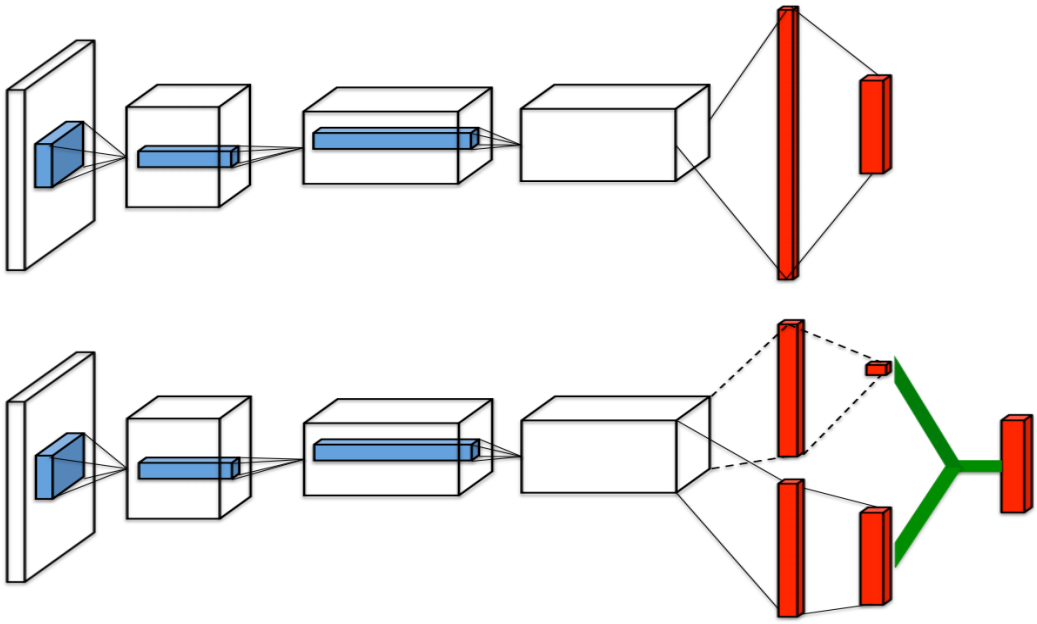
About Deep Reinforcement Learning Terminologies

- Reinforcement learning refers to goal-oriented algorithms, which learn how to attain a complex objective (goal) or maximize along a particular dimension over many steps; for example, maximize the points won in a game over many moves.
- They can start from a blank slate, and under the right conditions they achieve superhuman performance.

- Like a child incentivized by spankings and candy, these algorithms are penalized when they make the wrong decisions and rewarded when they make the right ones – this is reinforcement.
- At the heart of the learning algorithm is the , Deep Q-learning, which surpassed human-level performance in Atari games.
- To step back for a bit, the idea of Q-learning is to learn the action-value function, often denoted as `Q(s, a)` , where `s` represents the current state and `a` represents the action being evaluated.



- **Q-learning** is a form of Temporal-Difference learning (TD-learning), where unlike Monte-Carlo methods, we can learn from each step rather than waiting for an episode to complete. The idea is that once we take an action and are thrust into a new state, we use the current Q-value of that state as the estimate for future rewards.
- Double DQN
  - The popular Q-learning algorithm is known to overestimate action values under certain conditions.
  - DQNs are known to overestimate the value function because of the `max` operator.
  - The idea of Double DQN is to disentangle the calculation of the Q-targets into finding the best action and then calculating the Q-value for that action in the given state. The trick then is to use one network to choose the best action and the other to evaluate that action.



- Dueling DQN

  - Normally, DQNs have a single output stream with the number of output nodes equal to the number of actions. But this could lead to unnecessarily estimating the value of all the actions for states for states which are clearly bad and where, choosing any action won't matter that much.
  - One stream outputs a single scalar value denoting the value function for that state, `V(s)` while the other stream outputs the advantage function for each action in that state `A(a, s)`. The advantage function accounts for the advantage achieved for choosing action `a` . They are combined together using a special aggregrate layer:

  $$ Q (s, a) = V(s) + (A(s, a) - 1/A * mean_a (A (s, a))$$

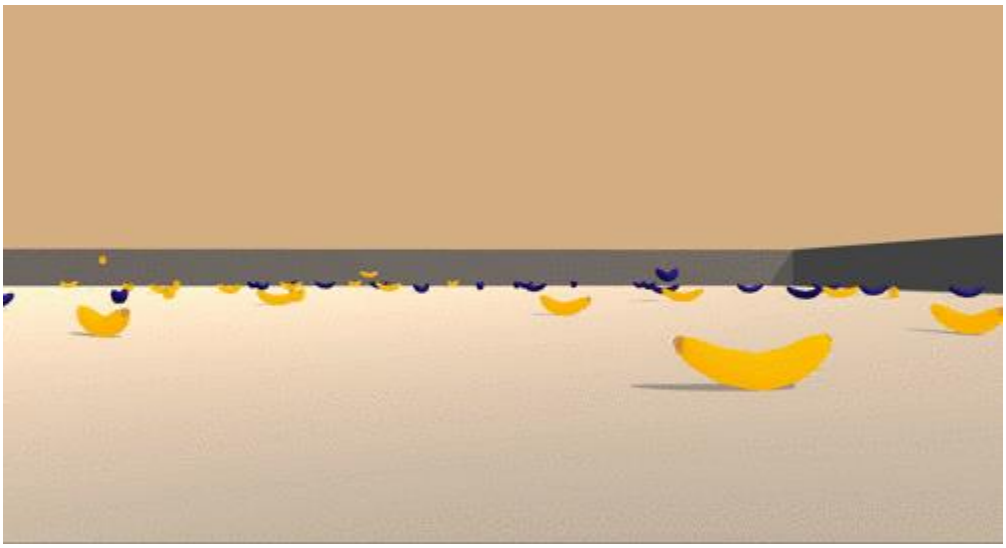
- Prioritized experience replay

  - Experience replay lets online reinforcement learning agents remember and reuse experiences from the past.
  - This is the other important technique used for stabilizing training. If we keep learning from experiences as they come, then we are basically observed a sequence of observations each of which are linked to each other.
  - In prior work, experience transitions were uniformly sampled from a replay memory. However, this approach simply replays transitions at the same

frequency that they were originally experienced, regardless of their significance.

Project's goal

In this project, **the goal is to train an agent to navigate a virtual world and collect as many yellow bananas as possible while avoiding blue bananas**



Environment details

The environment is based on Unity ML-agents

Note: The project environment provided by Udacity is like, but not identical to the Banana Collector environment on the Unity ML-Agents GitHub page.

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents. Agents can be trained using reinforcement learning, imitation learning, neuroevolution, or other machine learning methods through a simple-to-use Python API.

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 the 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.**

Learning Algorithm

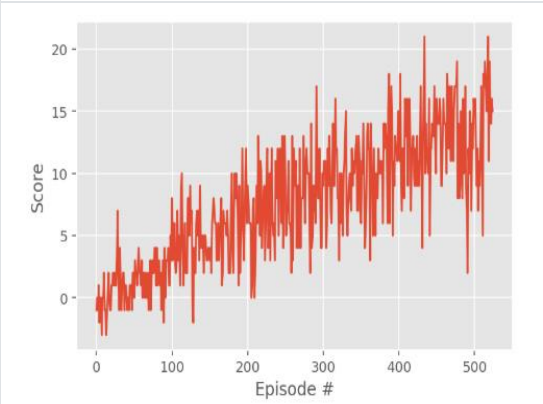
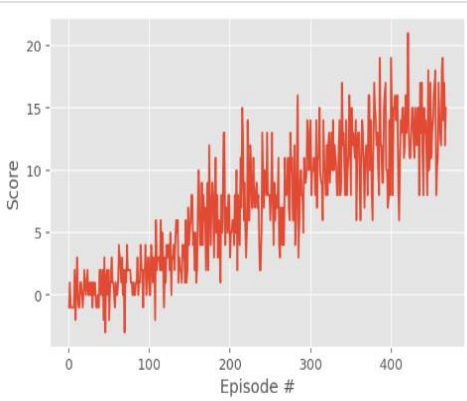
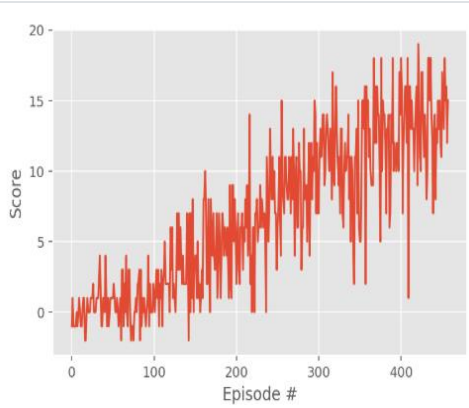
To watch a trained smart agent, follow the instructions below:

- **DQN**: If you want to run the original DQN algorithm, use the checkpoint `dqn.pth` for loading the trained model. Also, choose the parameter `qnetwork` as `QNetwork` while defining the agent and the parameter `update_type` as `dqn`.
- **Double DQN**: If you want to run the Double DQN algorithm, use the checkpoint `ddqn.pth` for loading the trained model. Also, choose the parameter `qnetwork` as `QNetwork` while defining the agent and the parameter `update_type` as `double_dqn`.
- **Dueling Double DQN**: If you want to run the Dueling Double DQN algorithm, use the checkpoint `dddqn.pth` for loading the trained model. Also, choose the parameter `qnetwork` as `DuelingQNetwork` while defining the agent and the parameter `update_type` as `double_dqn`.

Results/ Plot of Rewards

Plot showing the score per episode over all the episodes.

The best performance was achieved by **Dueling DQN** where the reward of +13 was achieved in **358** episodes.

Double DQN (Episodes:425)	DQN(Episodes:369)	Dueling DQN(Episodes:358)
		

Ideas for future work

- This model was a simple DQN network, we even tried double q-learning and, switch to a dueling dqn architecture.
- we can obviously implement DQN improved algorithms such as Prioritized Experience Replay, Dueling, Rainbow to get better result.
- Also, a systematic exploration of the hyperparameters, several Neural Networks architectures, could help us to build an agent that would learn faster.
- We can see if the Google Deepmind ConvNet architecture generalizes to this environment.