Navigation Project

Introduction:

The navigation project is about constructing a reinforcement-learning agent capable to navigate in a large 3D space and collects banana (see Figure 1). The agent gets a reward of +1 for collecting yellow banana and a reward of -1 for collecting blue banana. The goal of the agent is to accumulate as much as possible of positive rewards by collecting yellow bananas and avoiding blue bananas.

The state space has 37 dimensions and contains the agent's velocity, along with a ray-based perception of objects around the agent's forward direction. The agent has to learn how to best selects actions that maximize its rewards. There are four discrete actions available to the agent: 0, 1, 2, and 3 which correspond to move forward, move backward, turn left and turn right, respectively.

The navigation project has episodic task, and the agent must get a score of +13 over 100 consecutive episodes.

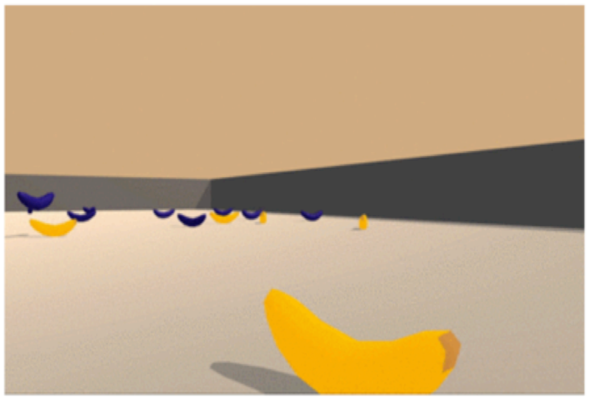

Figure 1. Banana environment.

Learning algorithm:

The agent adopted to solve the navigation project is the DQN agent. A deep Q-network is trained to optimize the action policy and so to learn how selecting successful policies using end-to-end reinforcement learning. The first Q-network model implemented in this project is a very simple one, which maps the state space to scalar action values. It is composed of three fully connected (fc) layers; the first two connected layers have 64 units followed by a ReLU activation function; the last fc layer is the output layer which is composed of four

units, the outputs are the scalar estimates of the q-values corresponding to the four actions. The activation function at the output layer is a linear function.
The Q-network is trained using the experience replay mechanism and an iterative update that adjusts the action-values towards target values, which are periodically updated. That means there two Q-networks, local and target. The local network is updated each iteration, while the target network updates occur at every given period of iterations.

Implementation:

The algorithm is implemented using two loops: the first loop runs over a pre-defined number of episodes, and the second (inner) loop runs over a pre-defined number of iterations where the Q-learning updates are applied to samples of experience drawn at random from the stored experiences in the memory.

Implementation steps:

For episode = 1, n_episodes **do**

- Reset the environment with a valid train mode *"env_info = env.reset(train_mode=True)[brain_name]"*
- Select a initial state from the state space observations "*state = env_info.vector_observations[0]*"
- **For** t = 1, max_t **do**
    1. with probability ε select a random action or take the action corresponding to the maximum Q-value "*action = agent.act(state, eps)*"
    2. execute action in environment "*env_info = env.step(action)[brain_name]"*
    3. get reward, next state and an indicator of termination *"reward =env_info.rewards[0], next state =env_info.vector_observations[0], done = env_info.local_done[0]"*
    4. save the experience (action, reward, next state, done) in the replay memory "*agent.step(state, action, reward, next_state, done)*"
    5. every pre-defined period of steps, sample a random batch of experiences from the memory and learn the agent by performing gradient descent step "*agent.step(state, action, reward, next_state, done)*"
    6. update target network parameters every pre-defined period of steps "*agent.step(state, action, reward, next_state, done)*"
    7. update current state by next state value *"state = next_state"*
    8. update scores *"score += reward"*

The initial training parameters are set as follows:

- number of episodes = 2000,
- maximum iterations: max_t = 1000,
- initial value of epsilon: start_eps = 1.0,
- epsilon decay: eps_decay = 0.995,
- fixed epsilon limit: end_eps: 0.01

Results:

Based on the implementations and the first agent model used for navigation, the algorithm achieved the scores shown in Figure 2.

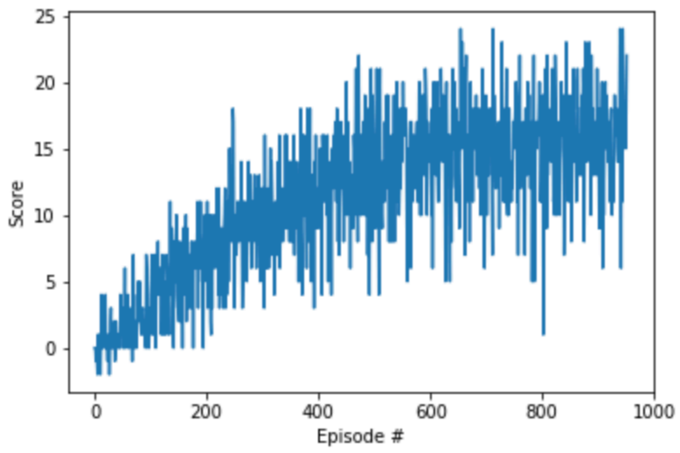

Figure 2. First DQN scores

The DQN agent was able to solve the navigation problem in fewer than 950 episodes.

Future work:

The DQN used in this first implementation is indeed a very shallow network with fully connected layers, and the input is a state observation vector defined by the velocity values. We notice that the scores exhibit high variances during iterations, which could be related to the model. Direct improvements are to tweak the parameters of the network and hyper-parameters to analyze how the agent's scores evolve. An extension that could improve the results is to train the agent in an environment with visual features like images. As in the deep Q-Network paper trained on Atari games, the state space can be defined by image observations and the agent learns directly from pixels. With this representation of the state space, the deep convolutional neural networks CNN can be used to build the Q-Network. Further enhancements are to use Double DQN or Deuling DQN.