Udacity Deep Reinforcement Learning Nanodegree

Project 1: Navigation

Train an Reinforcement Learning Agent to Collect Bananas

Goal

In this project a reinforcement learning agent that navigates an environment that is similar to Unity's Banana Collector environment.

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 of our agent is to collect as many yellow bananas as possible while avoiding blue bananas. In order to solve the environment, our agent must achieve an average score of +13 over 100 consecutive episodes.

Approach

Here are the high-level steps taken in building an agent that solves this environment.

1. Evaluate the state and action space.
2. Establish baseline using a random action policy.
3. Implement learning algorithm.
4. Run experiments to measure agent performance.
5. Select best performing agent and capture video of it navigating the environment.

1. Evaluate State & Action Space

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:

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

2. Establish Baseline

Before building an agent that learns, I started by testing an agent that selects actions (uniformly) at random at each time step.

```
 env_info = env.reset(train_mode=False)[brain_name] # reset the environment
state = env_info.vector_observations[0]            # get the current state
score = 0                                          # initialize the score
while True:
    action = np.random.randint(action_size)        # select an action
    env_info = env.step(action)[brain_name]        # send the action to the environment
    next_state = env_info.vector_observations[0]   # get the next state
    reward = env_info.rewards[0]                   # get the reward
    done = env_info.local_done[0]                  # see if episode has finished
    score += reward                                # update the score
    state = next_state                             # roll over the state to next time step
    if done:                                       # exit loop if episode finished
        break

print("Score: {}".format(score))
```

Running this agent a few times resulted in scores from -2 to 2. Obviously, if the agent needs to achieve an average score of 13 over 100 consecutive episodes, then choosing actions at random won't work.

3. Implement Learning Algorithm

Agents use a policy to decide which actions to take within an environment. The primary objective of the learning algorithm is to find an optimal policy—i.e., a policy that maximizes the reward for the agent. Since the effects of possible actions aren't known in advance, the optimal policy must be discovered by interacting with the environment and recording observations. Therefore, the agent "learns" the policy through a process of trial-and-error that iteratively maps various environment states to the actions that yield the highest reward. This type of algorithm is called **Q-Learning**.

As for constructing the Q-Learning algorithm, the general approach is to implement a handful of different components, then run a series of tests to determine which combination of components and which hyperparameters yield the best results.

In the following sections, we'll describe each component of the algorithm in detail.

Q-Function

To discover an optimal policy, I setup a Q-function. The Q-function calculates the expected reward `R` for all possible actions `A` in all possible states `S`.

We can then define our optimal policy `π*` as the action that maximizes the Q-function for a given state across all possible states. The optimal Q-function `Q*(s,a)` maximizes the total expected reward for an agent starting in state `s` and choosing action `a`, then following the optimal policy for each subsequent state.

In order to discount returns at future time steps, the Q-function can be expanded to include the hyperparameter gamma `γ`.

Epsilon Greedy Algorithm

One challenge with the Q-function above is choosing which action to take while the agent is still learning the optimal policy. Should the agent choose an action based on the Q-values observed thus far? Or, should the agent try a new action in hopes of earning a higher reward? This is known as the **exploration vs. exploitation dilemma**.

To address this, I implemented an **𝛆-greedy algorithm**. This algorithm allows the agent to systematically manage the exploration vs. exploitation trade-off. The agent "explores" by picking a random action with some probability epsilon `𝛜`. However, the agent continues to "exploit" its knowledge of the environment by choosing actions based on the policy with probability (1-𝛜).

Furthermore, the value of epsilon is purposely decayed over time, so that the agent favors exploration during its initial interactions with the environment, but increasingly favors exploitation as it gains more experience. The starting and ending values for epsilon, and the rate at which it decays are three hyperparameters that are later tuned during experimentation.

You can find the 𝛆-greedy logic implemented as part of the `agent.act()` method here in `agent.py` of the source code.

Deep Q-Network (DQN)

With 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 used for this project can be found here in the `model.py` file of the source code. The network contains three fully connected layers with 64, 64, and 4 nodes respectively. Testing of bigger networks (more nodes) and deeper networks (more layers) did not produce better results.

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.

Each experience is stored in a replay buffer as the agent interacts with the environment. The replay buffer contains a collection of experience tuples with the state, action, reward, and next state `(s, a, r, s')`. The agent then samples from this buffer as part of the learning step. Experiences are sampled randomly, so that the data is uncorrelated. This prevents action values from oscillating or diverging catastrophically, since a naive Q-learning algorithm could otherwise become biased by correlations between sequential experience tuples.

Also, experience replay improves learning through repetition. By doing multiple passes over the data, our agent has multiple opportunities to learn from a single experience tuple. This is particularly useful for state-action pairs that occur infrequently within the environment.

The implementation of the replay buffer can be found here in the `agent.py` file of the source code.

Double Deep Q-Network (DDQN)

One issue with Deep Q-Networks is they can overestimate Q-values (see Thrun & Schwartz, 1993). The accuracy of the Q-values depends on which actions have been tried and which states have been explored. If the agent hasn't gathered enough experiences, the Q-function will end up selecting the maximum value from a noisy set of reward estimates. Early in the learning process, this can cause the algorithm to propagate incidentally high rewards that were obtained by chance (exploding Q-values). This could also result in fluctuating Q-values later in the process.

We can address this issue using Double Q-Learning, where one set of parameters `w` is used to select the best action, and another set of parameters `w'` is used to evaluate that action.

The DDQN implementation can be found here in the `agent.py` file of the source code.

Dueling Agents

Dueling networks utilize two streams: one that estimates the state value function `V(s)`, and another that estimates the advantage for each action `A(s,a)`. These two values are then combined to obtain the desired Q-values.

The reasoning behind this approach is that state values don't change much across actions, so it makes sense to estimate them directly. However, we still want to measure the impact that individual actions have in each state, hence the need for the advantage function.

The dueling agents are implemented within the fully connected layers here in the `model.py` file of the source code.

4. Run Experiments

Now that the various components of our algorithm are in place, it's time to measure the agent's performance within the Banana environment. Performance is measured by the fewest number of episodes required to solve the environment.

The table below shows the complete set of experiments. These experiments compare different combinations of the components and hyperparameters discussed above. However, note that all agents utilized a replay buffer.

5. Select best performing agent

The best performing agents were able to solve the environment in 200-250 episodes. While this set of agents included ones that utilized Double DQN and Dueling DQN, ultimately, the top performing agent was a simple DQN with replay buffer.

The complete set of results and steps can be found in this notebook.

Also, here is a video showing the agent's progress as it goes from randomly selecting actions to learning a policy that maximizes rewards.

Future Improvements

- **Test the replay buffer** — Implement a way to enable/disable the replay buffer. As mentioned before, all agents utilized the replay buffer. Therefore, the test results don't measure the impact the replay buffer has on performance.
- **Add *prioritized* experience replay** — Rather than selecting experience tuples randomly, prioritized replay selects experiences based on a priority value that is correlated with the magnitude of error. This can improve learning by increasing the probability that rare and important experience vectors are sampled.
- **Replace conventional exploration heuristics with Noisy DQN** — This approach is explained here in this research paper. The key takeaway is that parametric noise is added to the weights to induce stochasticity to the agent's policy, yielding more efficient exploration.