Report for Udacity DRLND Project 1

Navigation (Banana)

Introduction

This report describes the Navigation collecting bananas project results using the Unity environment. This report demonstrates an implementation of the DQN algorithm which is able to successfully complete the project requirements.

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 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.

RL to Deep RL (Q table to DQN)

Reinforcement Learning

In traditional Reinforcement Learning (RL), there are two methods to solving the problem: Monte Carlo and Temporal-Difference (TD) methods.

While Monte-Carlo approaches requires we run the agent for the whole episode before making any decisions. This solution is no longer viable with continuous tasks that does not have any terminal state, as well as episodic tasks for cases when we do not want to wait for the terminal state before making any decisions in the environment's episode.

This is where Temporal-Difference (TD) Control Methods step in. The agent updates estimates based in part on other learned estimates, without waiting for the final outcome. TD estimate methods will update the Q-table after every time steps.

The Q-table is used to approximate the action-value function q(π) for the policy π. The Q-Learning is one effective TD method that uses an update rule that attempts to approximate the optimal value function at every time step:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \lambda \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

Deep Reinforcement Learning

The shortcoming of Q-Learning is that the Q-table assumes that agent works with only discrete states and actions. Actually most actions that need to take place in a physical environment are continuous in nature. Moreover, the states can also be continuous, for instance sensors' information as well as visual images taking from the environment.

To address this issue, the use of function approximation created a major breakthrough, since this is exactly the purpose of neural networks to create such approximations.

In the standard problem of Navigation, we are given 37, continuous states, 4 discrete actions so a fully-connected neural network is appropriate for approximate optimal-value function q* .

Stabilizing Deep Reinforcement Learning

The Deep Q-Learning algorithm represents the optimal action-value function q* as a neural network (instead of a table).

Unfortunately, reinforcement learning is notoriously unstable when neural networks are used to represent the action values. The Deep Q-Learning algorithm addresses these instabilities by using two key features:

- Experience Replay
- Fixed Q-Targets

Experience Replay

When the agent interacts with the environment, the sequence of experience tuples can be highly correlated. The naive Q-learning algorithm that learns from each of these experience tuples in sequential order runs the risk of getting swayed by the effects of this correlation. By instead keeping track of a replay buffer and using experience replay to sample from the buffer at random, we can prevent action values from oscillating or diverging catastrophically.

The replay buffer contains a collection of experience tuples (S, A, R, S′). The tuples are gradually added to the buffer as we are interacting with the environment.

The act of sampling a small batch of tuples from the replay buffer in order to learn is known as experience replay. In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience.

Fixed Q-Targets

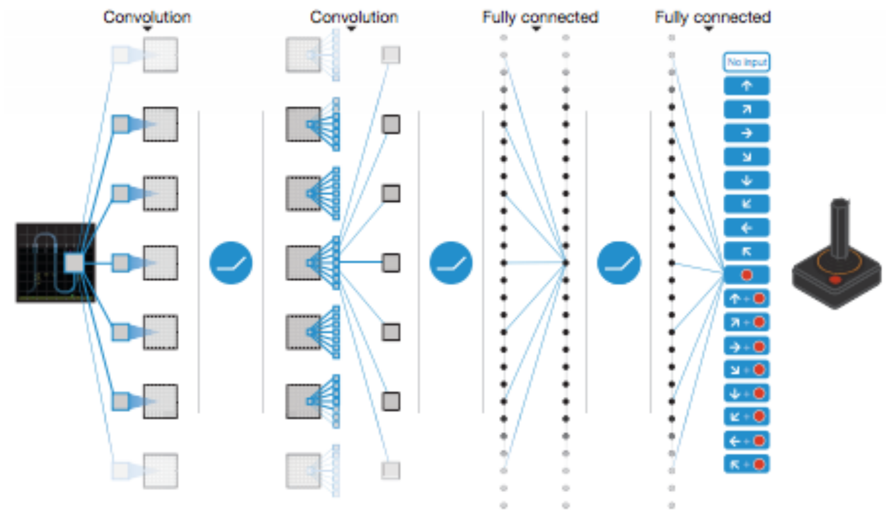
In Q-Learning, we update a guess with a guess, and this can potentially lead to harmful correlations. To avoid this, we can update the parameters delta w in the network q^ to better approximate the action value corresponding to state S and action A with the following update rule:

$$\Delta w = \alpha \cdot \overbrace{(\underbrace{R + \gamma \max_a \hat{q}(S', a, w^-)}_{\text{TD target}} - \underbrace{\hat{q}(S, A, w)}_{\text{old value}})}^{\text{TD error}} \nabla_w \hat{q}(S, A, w)$$

where w− are the weights of a separate target network that are not changed during the learning step, and (S, A, R, S′) is an experience tuple.
Note: How the example in the video would look in real life?
See: Carrot Stick Riding.

Implementation

Here is DQN architecture and algorithm defined in the paper *Human-level control through deep reinforcement learning*



In the implementation, we update the target Q-Network’s weights every 4 time steps. As for the other parameters involved in the update, α or the learning rate is initialized at 5e − 4 and is learned thanks to the usage of Adam optimizer.
The discount rate γ is set to 0.99 which is very close to 1, meaning that the return objective takes future rewards into account more strongly and the agent becomes more farsighted.

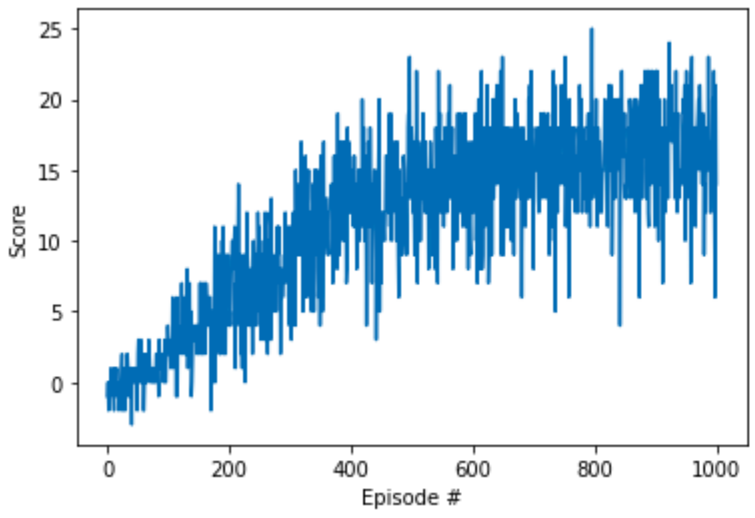
Policy

The learning policy is Greedy in the Limit with Infinite Exploration (GLIE). This strategy initially favors exploration versus exploitation, and then gradually prefers exploitation over exploration.

As such, I train the model using an ε-greedy policy with ε annealed exponentially from 1.0 to 0.01 with a decay of 0.995 every episode.

Results

Then agent solved situation (Score of 13.0, averaged over last 100 episodes) in 478 episodes and the best average score of 16.49 is achieved in 1000 episodes.



Future ideas DQN Improvements

The following techniques are worth pursuing for faster convergence or higher peak performance:

Hyperparameter search

The learning rates selected may not be optimal and may result in slower convergence.

Architecture search

Is it possible that a deeper model or adding / removing units in each of the hidden layers may improve performance, or combined with Double DQN or Dueling DQN?

Prioritized experience replay

Prioritized replay of states and actions seems to be helpful to convergence and definitely will improve performance.