Project 1: Navigation - Report

Introduction

In this project, I trained an agent to navigate and collect bananas in a large, square world.

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. 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.

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 the agent is to collect as many yellow bananas as possible while avoiding blue bananas.

Agent Implementation

Methods

The agent uses a Deep Q-Learning Algorithm. It is a Value-Based method, combining Q-Learning reinforcement learning (SARSAMAX) and a Deep Neural Network to update the Q-Table.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Big)}^{\text{temporal difference}}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

Bellman equation in Q-Learning

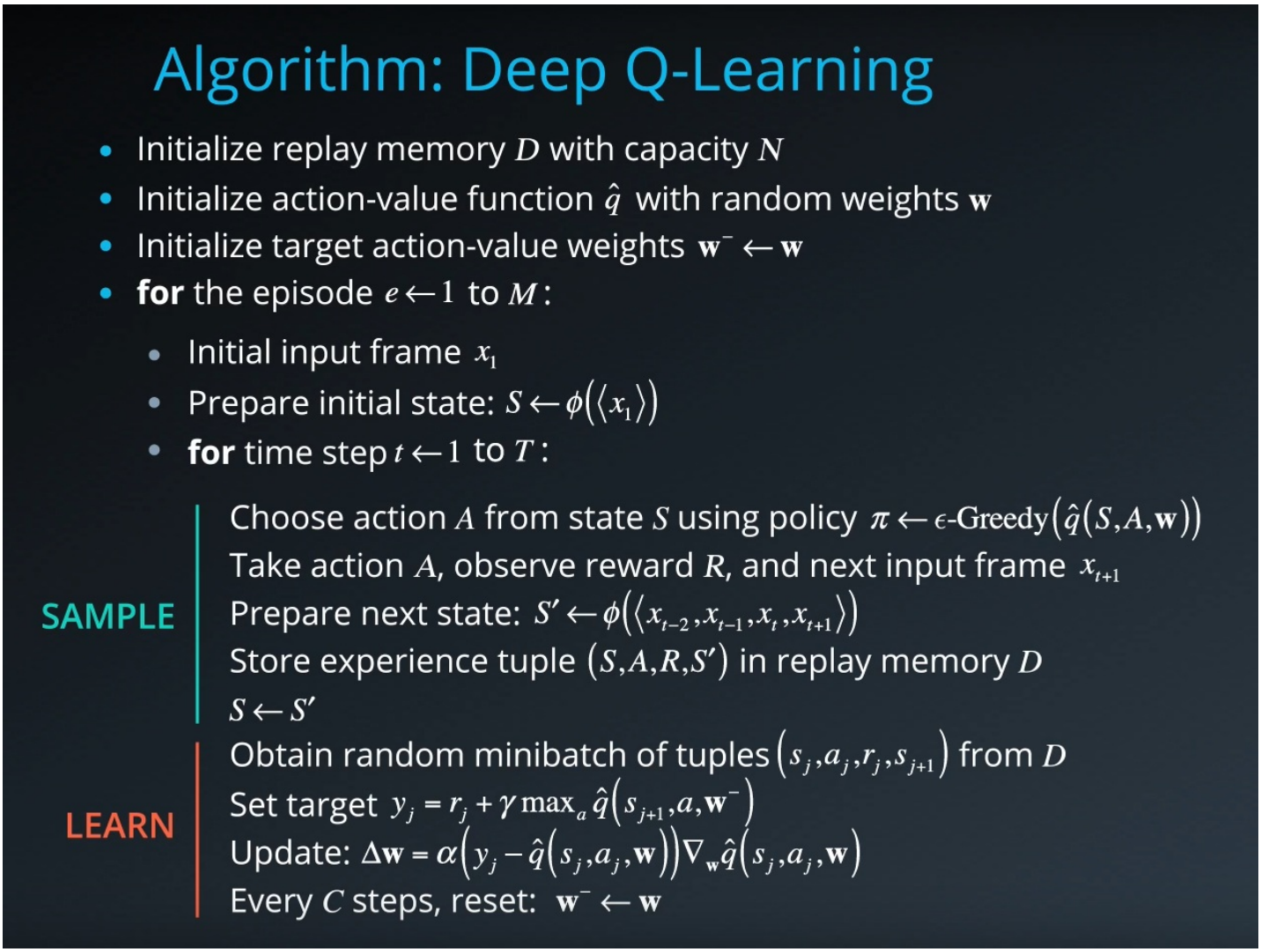
This implementation also includes 2 improvements over the original paper: - Experience Replay

> 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 (SS, AA, RR, S'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 ww in the network \hat{q}q^ to better approximate the action value corresponding to state SS and action AA with the following update rule: > here w^-w – are the weights of a separate target network that are not changed during the learning step, and (SS, AA, RR, S'S') is an experience tuple.

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

Algorithm



Taken from Udacity Deep Reinforcement Learning Nanodegree Course

Implementation

This code is a slightly modified implementation of the "Lunar Lander" tutorial from Udacity Nanodegree course. It is composed of the following elements:

- model.py: a PyTorch QNetwork class implementation. It initializes a Deep Neural Network composed of:
  - an input layer (size depending on the state_size parameter)
  - two hidden layers (in this example, ran with 64 cells each)
  - one activation layer (size depending on the action_size parameter)
- dqn_agent.py: DQN Agent Class and a Replay Buffer class implementation
  - DQN Agent implements the following methods:
  - constructor: initialization of the Replay Buffer and 2 instance of the Neural Network : the target network and the local network
  - step() allows to store a step taken by the agent (state, action, reward, next_state, done) in the Replay Buffer/Memory. Every 4 steps, it updates the target network weights with the current weight values from the local network
  - act() returns actions for the given state as per current policy using an Epsilon-greedy selection, providing a balance between exploration and exploitation for the Q Learning
  - learn() updates the Neural Network value parameters using batch of experiences from the Replay Buffer
  - soft_update() is called by learn() to softly updates the value from the target Neural Network from the local network weights
  - The ReplayBuffer class implements a fixed-size buffer to store experience tuples (state, action, reward, next_state, done)
  - add() adds an experience step to the memory
  - sample() randomly samples a batch of experience steps for the learning
- DQN_Banana_Navigation.ipynb :
  - Start the environment, train the agent, export the weights and visualize the results

Hyperparameters

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size (Initially 64)
GAMMA = 0.995           # discount factor (Initially 0.99)
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate (Initially 5e-4)
UPDATE_EVERY = 4        # how often to update the network
```

Training

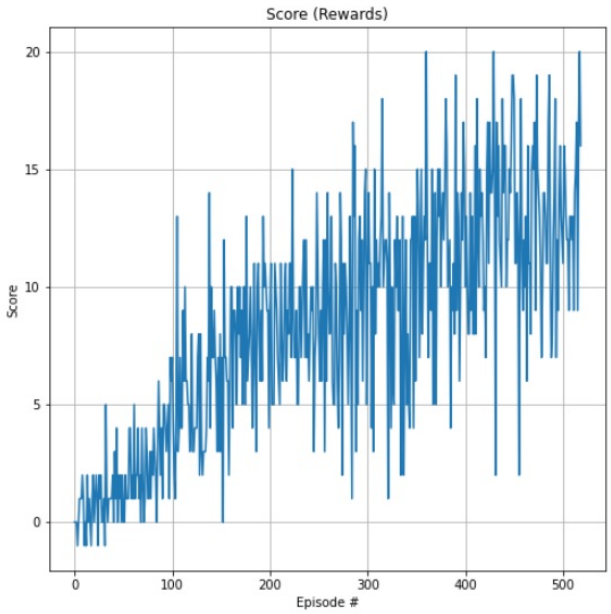
The agent was successfully trained in 419 episodes, achieving an average score of 13 for 100 episodes (between episodes 419 and 519 episodes). This successfully meet the objectives of the project instructions (achieving an average score of at least 13 over 100 episodes).

In [16]:
```
## Parameters

n_episodes = 5000
max_t = 2000
eps_start = 1.0
eps_end = 0.1
eps_decay = 0.995
```

In [17]:
```
scores = dqn(n_episodes, max_t, eps_start, eps_end, eps_decay)
```

```
Episode 100     Average Score: 1.71
Episode 200     Average Score: 6.59
Episode 300     Average Score: 8.73
Episode 400     Average Score: 10.57
Episode 500     Average Score: 12.63
Episode 519     Average Score: 13.00
Environment solved in 419 episodes!     Average Score: 13.00
```



Ideas for improvement

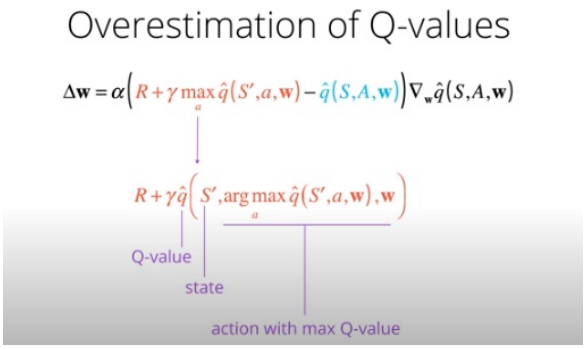
As stated in the Udacity Nanodegree course, a few improvements could be made:

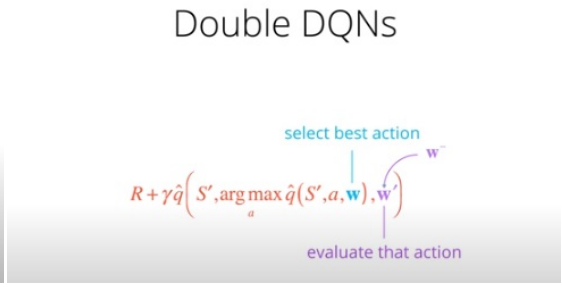
- Double Q-Learning

Abstract from the paper

> The popular Q-learning algorithm is known to overestimate action values under certain conditions. [...] It was not previously known whether, in practice, such overestimations are common, whether they harm performance, and whether they can generally be prevented. In particular, we first show that the recent DQN algorithm, which combines Q-learning with a deep neural network, suffers from substantial overestimations in some games in the Atari 2600 domain. We then show that the idea behind the Double Q-learning algorithm, which was introduced in a tabular setting, can be generalized to work with large-scale function approximation. We propose a specific adaptation to the DQN algorithm and show that the resulting algorithm not only reduces the observed overestimations, as hypothesized, but that this also leads to much better performance on several games.

From udacity Nanodegree course:

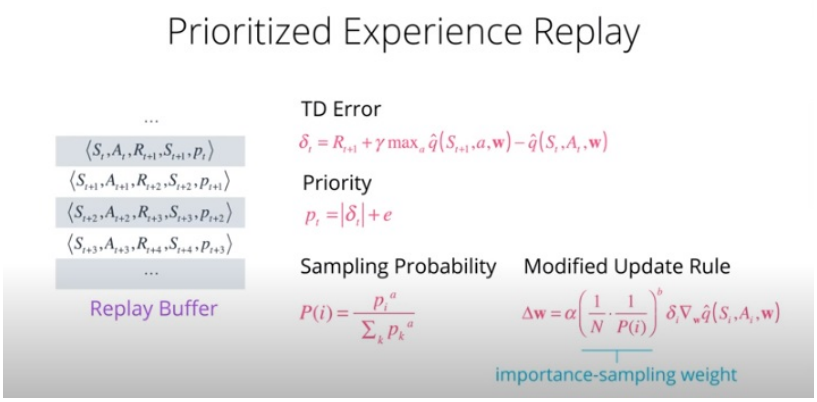




- Prioritized Experience Replay

Abstract from the paper

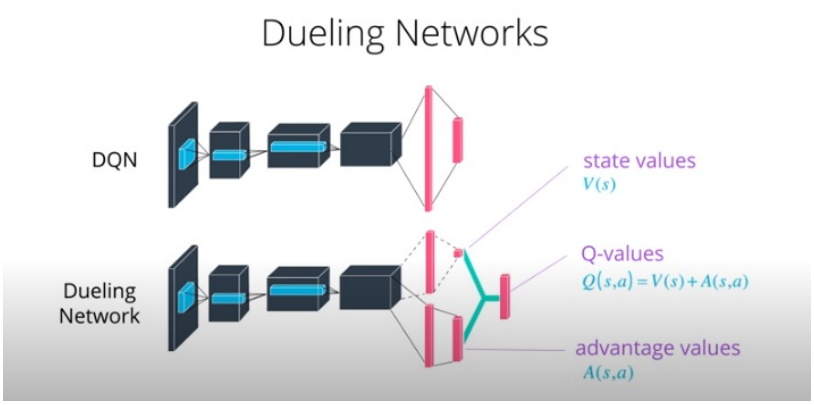
> Experience replay lets online reinforcement learning agents remember and reuse experiences from the past. In prior work, experience transitions were uniformly sampled from a replay memory. However, this approach simply replays transitions at the same frequency that they were originally experienced, regardless of their significance. In this paper we develop a framework for prioritizing experience, so as to replay important transitions more frequently, and therefore learn more efficiently. We use prioritized experience replay in Deep Q-Networks (DQN), a reinforcement learning algorithm that achieved human-level performance across many Atari games. DQN with prioritized experience replay achieves a new state-of-the-art, outperforming DQN with uniform replay on 41 out of 49 games.



- Dueling Q-Networks

Abstract from the paper

> In recent years there have been many successes of using deep representations in reinforcement learning. Still, many of these applications use conventional architectures, such as convolutional networks, LSTMs, or auto-encoders. In this paper, we present a new neural network architecture for model-free reinforcement learning. Our dueling network represents two separate estimators: one for the state value function and one for the state-dependent action advantage function. The main benefit of this factoring is to generalize learning across actions without imposing any change to the underlying reinforcement learning algorithm. Our results show that this architecture leads to better policy evaluation in the presence of many similar-valued actions. Moreover, the dueling architecture enables our RL agent to outperform the state-of-the-art on the Atari 2600 domain.



More improvements exist (Learning from multi-step bootstrap targets, Distributional DQN, Noisy DQN). By incorporating all 6 of them, the research team at Google DeepMind created the Rainbow algorithm, outperforming all the previous individual modifications.

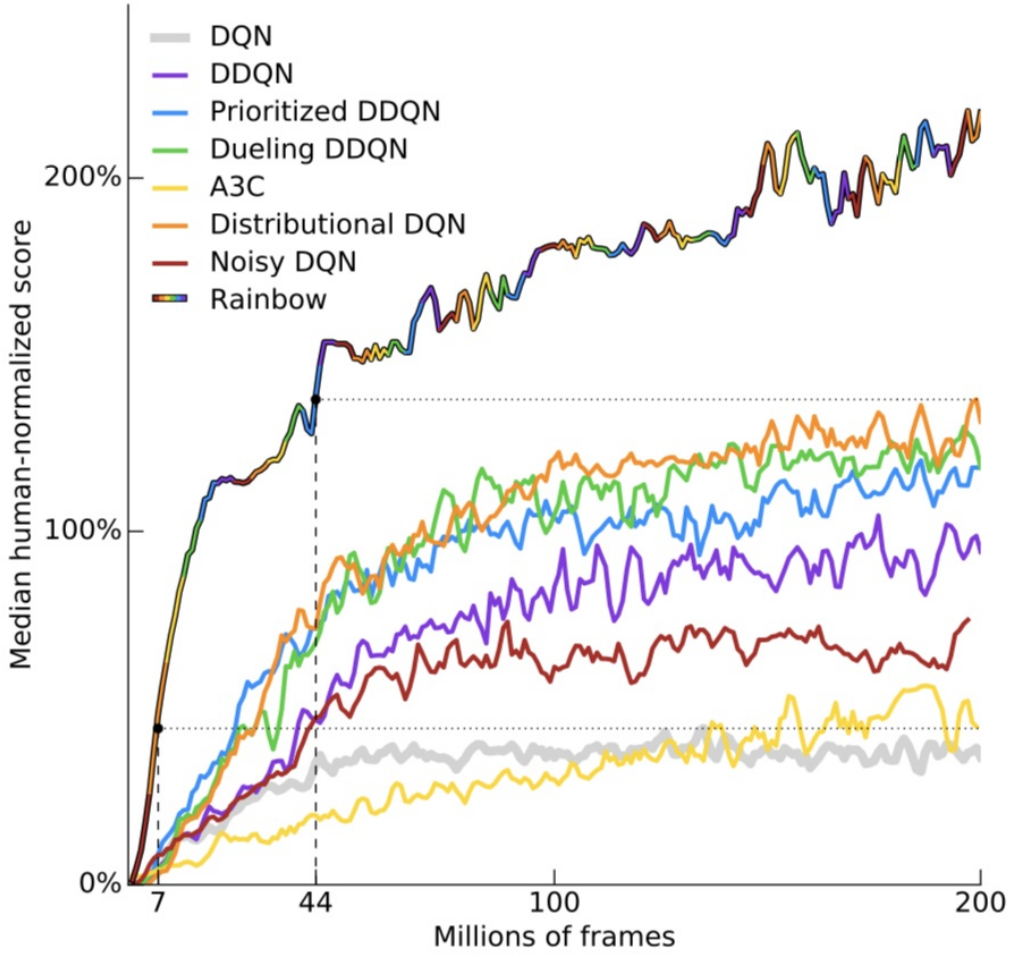


Figure 1: **Median human-normalized performance** across 57 Atari games. We compare our integrated agent (rainbow-colored) to DQN (grey) and six published baselines. Note that we match DQN's best performance after 7M frames, surpass any baseline within 44M frames, and reach substantially improved final performance. Curves are smoothed with a moving average over 5 points.