Project: Navigation

Project Details:

Project Description

For this project, we will train an agent to navigate (and collect bananas!) in a large, square world. The world contains both yellow and blue bananas.

Project Goal

The goal of the agent is to collect as many yellow bananas as possible while avoiding blue bananas. **The task is episodic, and in order to solve the environment, your agent must get an average score of +13 over 100 consecutive episodes.**

Reward

A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana.

State 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

Action

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 Environment

The details are taken from the Udacity's Deep Reinforcement Learning Nanodegree program. The environment is based on Unity ML-agents. Please read more about ML-Agents by perusing the GitHub repository.

The project environment is similar to, but not identical to the Banana Collector environment on the Unity ML-Agents GitHub page.

Follow the instructions below to explore the environment on your own machine! You will also learn how to use the Python API to control your agent.

Implementation

This navigation project implements a Value Based method called Deep Q-Networks

Deep Q-Networks

DeepMind leveraged a **Deep Q-Network (DQN)** to build the Deep Q-Learning algorithm that learned to play many Atari video games better than humans.

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. In this lesson, you'll learn all about the Deep Q-Learning algorithm, which addressed these instabilities by using **two key features**:

- **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 w in the

network q^to better approximate the action value corresponding to state $S$$S$ and action $A$$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.

Deep Q-Learning Algorithm

Please read the research paper that introduces the Deep Q-Learning algorithm.

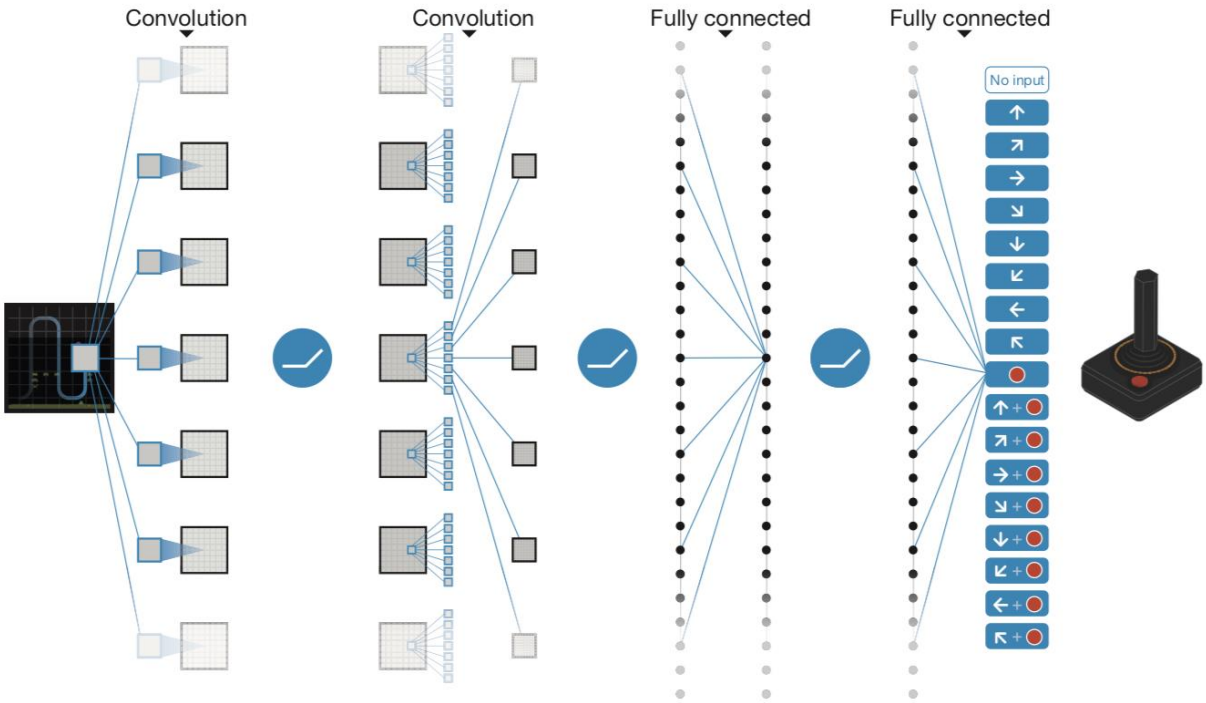


Illustration of DQN Architecture (Source)

Code / Implementation:

Github link: https://github.com/kishoregajjala/DeepRL-PRJ-Navigation

Readme.md provides more details of the project and environment.

The implementation of this project consists these files:

- **model.py**: This implements the Q-network class with regular fully connected Deep Neural Network using the PyTorch framework. The state_size parameter drives the size of the input layer. The action_size parameter drives the size of the output layer. This uses 2 hidden fully connected layers of 64 nodes

- **dqn_agent.py**: This implements the DQN agent and replay buffer classes

    - **DQN agent class** is implemented as described in the Deep Q-learning algorithm. It consists of these methods.
        - constructor() :
            - Initialize the replay memory buffer
            - Initialize the target and local network instances

        - step() :
            - Store experience (state, action, reward, next_state, done) in replay memory
            - If enough samples are available in memory, get random subset and learn – for every 4 steps, update the target network weights with the current weight values from the local network.
        - act() : Returns actions for given state as per current policy
        - learn(): Updates the neural network value parameters using given batch of experiences from the Replay buffer.
        - soft_update(): This is called by learn() to update the model parameters from the target neural network using the local network weights.

    - **The Replay Buffer class** implements fixed-size buffer to store experience tuples. The class consists of these methods.
        - add() : Adds a new experience to memory.
        - sample() : Randomly sample a batch of experiences from memory
        - len(): Return the current size of internal memory.

- **Navigation.ipynb:** This notebook creates coding environment to train your agent for the project.
    - **High level steps**:
        - Import the necessary packages and start the environment
        - Examine the state and action spaces
        - Take random actions in the environment
        - Train the agent using DQN

- Directly run the notebook within the online Workspace provided by Udacity Nanodegree for the Project #1 Navigation.
- The online workspace does not allow you to see the simulator of the environment; so, if you want to watch the agent while it is training, you should build your own local environment and make necessary adjustments to the path of the Unity environment and train locally.

  - Plot the scores

- **DQN Hyperparameters:**

  The DQN agent uses the following hyperparameter values as defined in dqn_agent.py file.

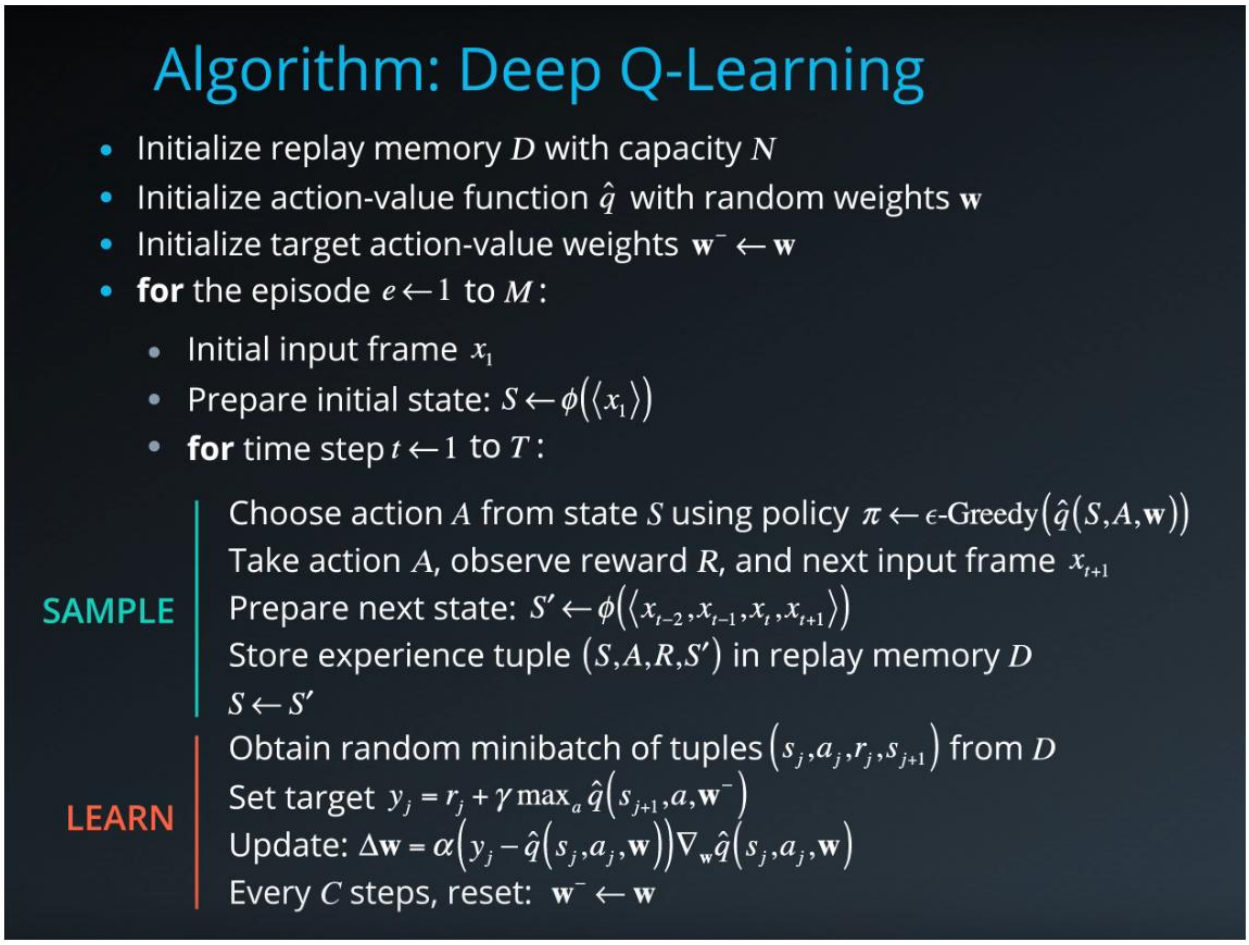
  ```
  BUFFER_SIZE = int(1e5)      # replay buffer size
  BATCH_SIZE = 64             # minibatch size
  GAMMA = 0.99                # discount factor
  TAU = 1e-3                  # for soft update of target parameters
  LR = 5e-4                   # learning rate
  UPDATE_EVERY = 4            # how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

- **The Architecture**

  Input nodes (37) → Fully Connected Layer (64 nodes, Relu activation) → Fully Connected Layer (64 nodes, Relu activation) → Output nodes (4)

  The Neural networks use Adam optimizer with a learning rate LR = 5e-4 and are trained using a batch size = 64.

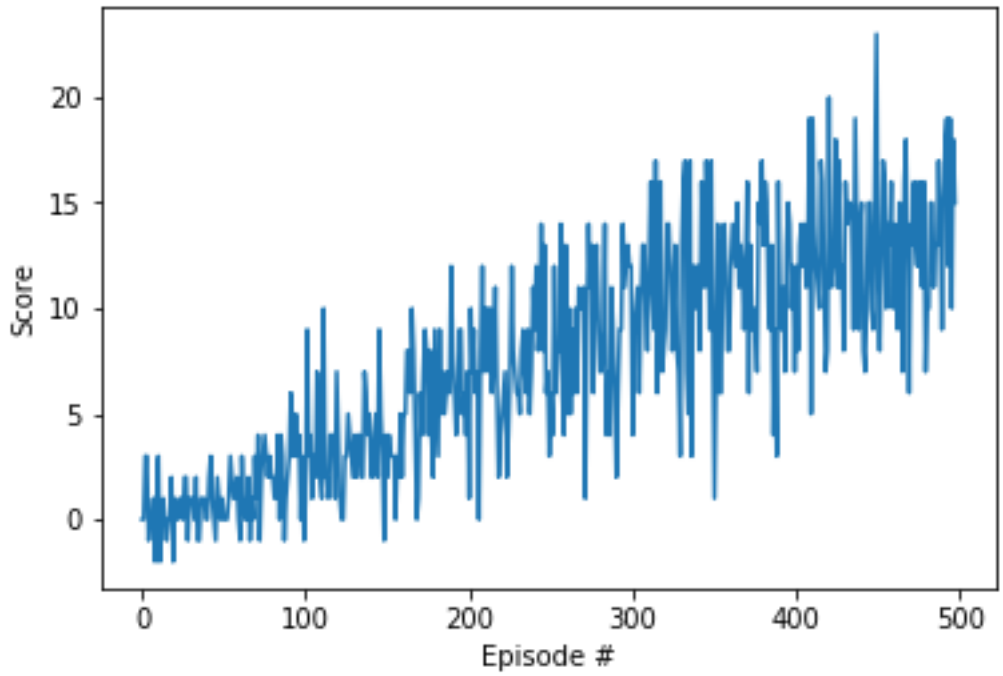
- **Learning Algorithm:**



Results and Plot of Rewards

```
Episode 100    Average Score: 1.14
Episode 200    Average Score: 4.45
Episode 300    Average Score: 8.07
Episode 400    Average Score: 10.85
Episode 498    Average Score: 13.00
Environment solved in 398 episodes!  Average Score: 13.00

 Agent total training time = 6.60 minutes
```



The agent solved this task in 398 episodes, and it received an average reward of at least +13 (over 100 episodes).

Ideas for Future Work

The submission has concrete future ideas for improving the agent's performance

- Train the agent by directly learning from raw pixels

  In the project, your agent learned from information such as its velocity, along with ray-based perception of objects around its forward direction. A more challenging task would be to learn directly from pixels!

  This environment is *almost* identical to the project environment, where the only difference is that the state is an 84 x 84 RGB image, corresponding to the agent's first-person view of the environment.

- Consider these several improvements to the original Deep Q-Learning algorithm
  - **Double DQN**
    Deep Q-Learning tends to overestimate action values. Double Q-Learning has been shown to work well in practice to help with this.

  - **Prioritized Experience Replay**

Deep Q-Learning samples experience transitions *uniformly* from a replay memory. Prioritized experienced replay is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability.

- ## Dueling DQN

  Currently, in order to determine which states are (or are not) valuable, we have to estimate the corresponding action values *for each action*. However, by replacing the traditional Deep Q-Network (DQN) architecture with a dueling architecture, we can assess the value of each state, without having to learn the effect of each action.