Deep Reinforcement Learning– Project 1: Navigation

Abhranil Chandra

This is a technical description of my solution to the Navigation project which is the first of the three compulsory projects in Deep Reinforcement Learning Nanodegree from Udacity.

1. Summary: Banana Hungry Agent

The project tries to train an agent to collect the maximum number of yellow bananas(positive rewards) while avoiding the blue ones(negative rewards). The agent learns by **trial and error** to collect the correct bananas and to maximize the cumulative reward(**delayed reward** problem). Thus the problem is well suited to be framed as a **reinforcement learning(RL) problem**.

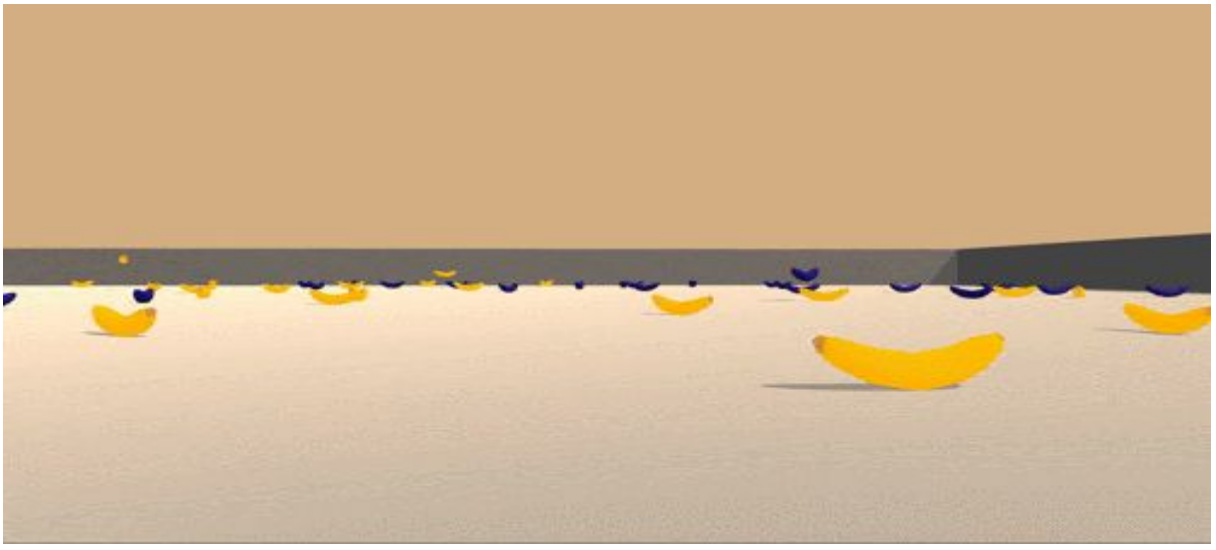

Figure 1: Overview of the environment

2. Why transition from RL to Deep RL?

We can choose from either **Monte Carlo** methods or **Temporal-Difference** (TD) methods.
The **Monte-Carlo** approach is **not viable** as it requires us to run the agent for the whole episode before making any decisions and continuous tasks do not have any terminal state. Even in the case of episodic tasks**,** it leads to bad performance if we for the terminal state before making any decisions in the environment's episode.
Temporal-Difference (TD) Methods do not wait for the final outcome and update estimates based in part on other learned estimates. TD methods update the **Q-table after every time step**. The Q-table **approximates the action-value function q_π**. An example is demonstrated below

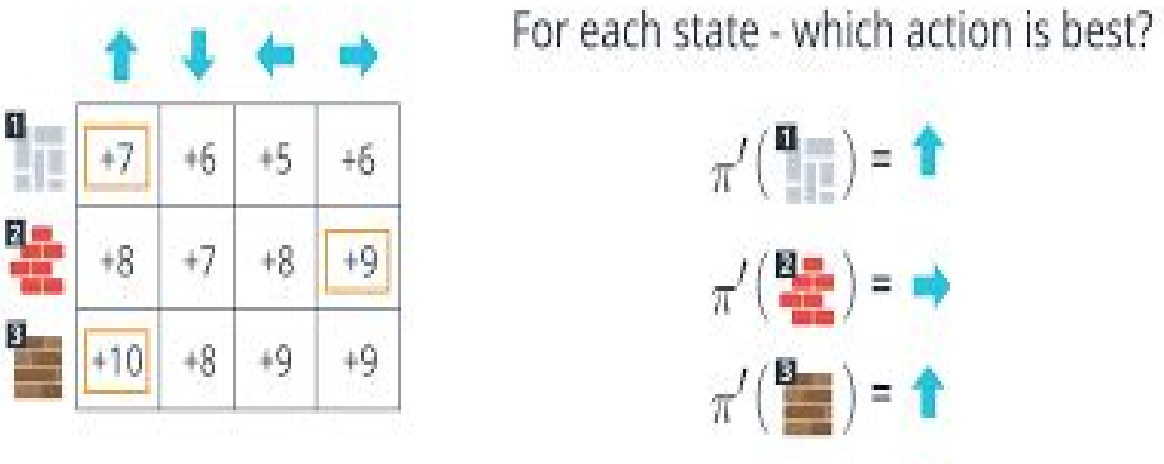

Figure 2: Q-table Example

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha R_{t+1} + \gamma \max_{a \in A} Q(S_{t+1}, a) - Q(S_t, A_t)$$

But the major drawback of Q-table is that it can handle only **discrete** states and actions. In general, most actions are continuous in nature, moreover, the states can also be continuous. To address this issue, the **function approximations** were done using **neural networks**. In our problem setting, we have 37 continuous states with 4 discrete actions. Thus a **fully-connected neural network** is appropriate for the **optimal-value function q_*** approximation.

3. Agents Implementation(Algorithm)

I have used a **model-free value-based** method called **Deep Q-Networks**.

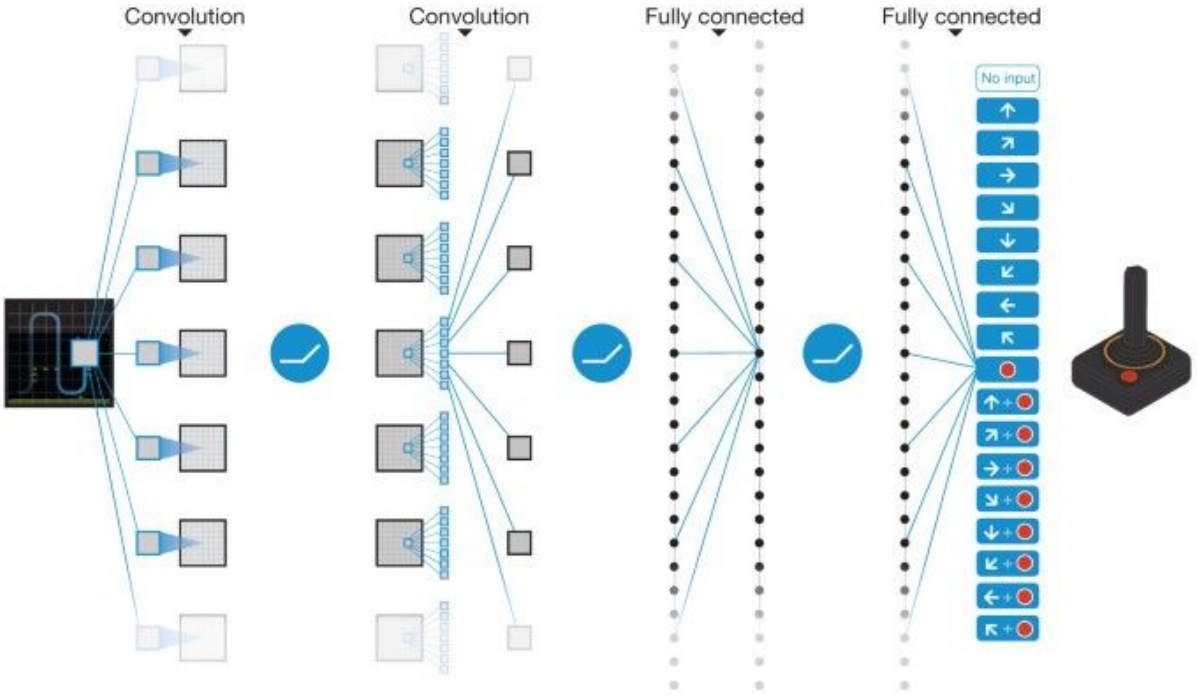

Figure 3: Deep Q-Network

Algorithm: Deep Q-Learning

- Initialize replay memory D with capacity N
- Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
- Initialize target action-value weights $\mathbf{w}^- \leftarrow \mathbf{w}$
- **for** the episode $e \leftarrow 1$ to M:
  - Initial input frame x_1
  - Prepare initial state: $S \leftarrow \phi(\langle x_1 \rangle)$
  - **for** time step $t \leftarrow 1$ to T:

SAMPLE

Choose action A from state S using policy $\pi \leftarrow \epsilon\text{-Greedy}(\hat{q}(S,A,\mathbf{w}))$
Take action A, observe reward R, and next input frame x_{t+1}
Prepare next state: $S' \leftarrow \phi(\langle x_{t-2}, x_{t-1}, x_t, x_{t+1} \rangle)$
Store experience tuple (S,A,R,S') in replay memory D
$S \leftarrow S'$

LEARN

Obtain random minibatch of tuples (s_j, a_j, r_j, s_{j+1}) from D
Set target $y_j = r_j + \gamma \max_a \hat{q}(s_{j+1}, a, \mathbf{w}^-)$
Update: $\Delta\mathbf{w} = \alpha(y_j - \hat{q}(s_j, a_j, \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s_j, a_j, \mathbf{w})$
Every C steps, reset: $\mathbf{w}^- \leftarrow \mathbf{w}$

Figure 4: Deep Q-Learning Algorithm

This algorithm uses a deep neural network to approximate the Q-table(action-value pairs) using a reinforcement learning method called Q-Learning or SARSA max.

However, the RL algorithm in its raw form is **highly unstable** and so two techniques used to stabilize the training were invented and described by Deepmind scientists in their Nature publication "Human-level control through deep reinforcement learning (2015)" :

- **Experience Replay**

  The sequence of **experience tuples (S, A, R, S')** can be **highly correlated** so it breaks the assumption of sample independence. The naive Q-Learning algorithm having these experience tuples as input may get swayed by the effects of this correlation. By instead keeping track of a **replay buffer** and using experience replay to sample randomly from the buffer thus preventing action values from oscillating or diverging catastrophically.

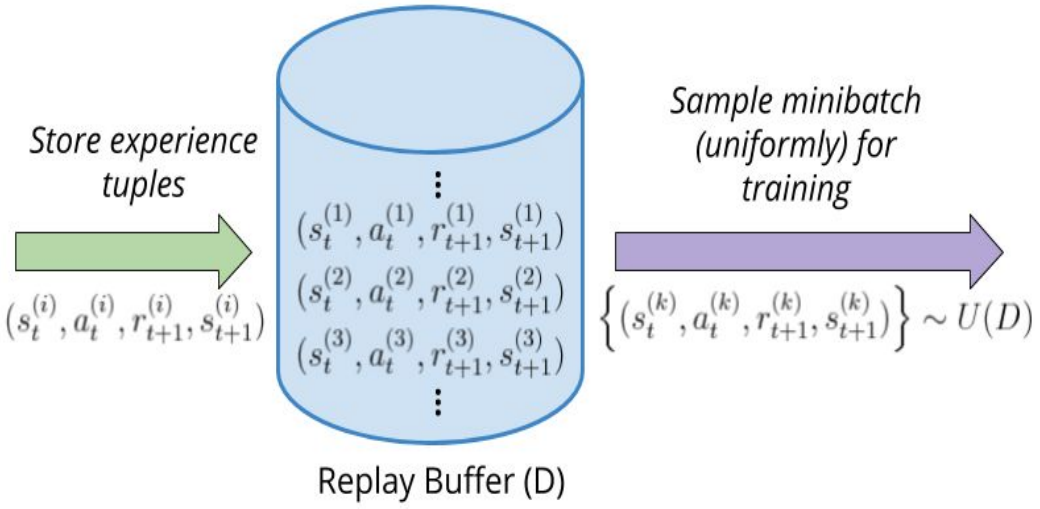

Figure 5: Replay Buffer

- **Fixed Q Targets**

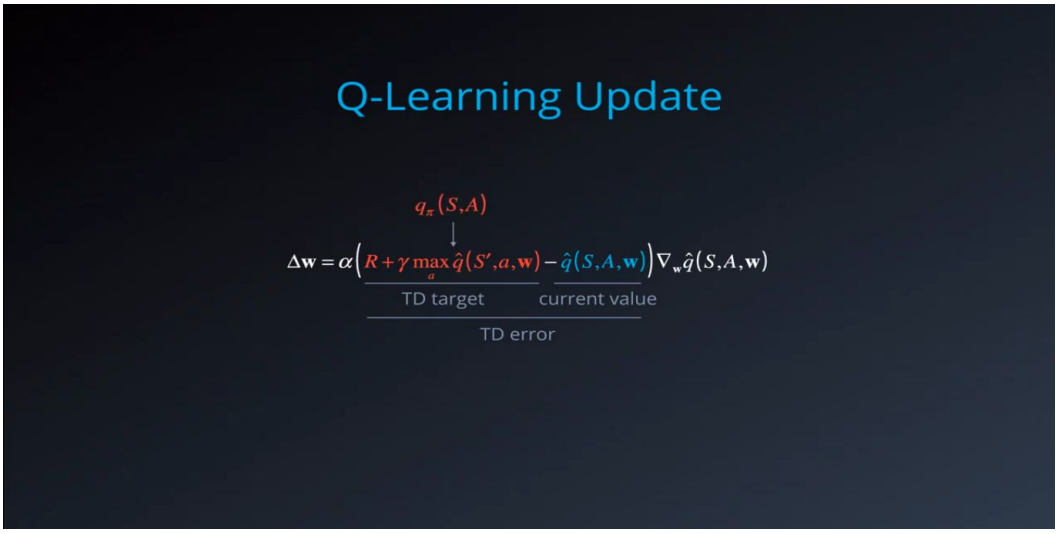

Figure 6: Update rule of Q-Learning

As is visible from the equation the target is itself dependent on w, the updated parameter. Thus at every step of training, **our Q-values shift but so do the target values** and so the results oscillate in the training phase. We use a separate target network to **update the target Q-Network's weights less often than the primary Q-Network.** In my implementation, the target Q-Network's weights are updated every **4 time-steps**.

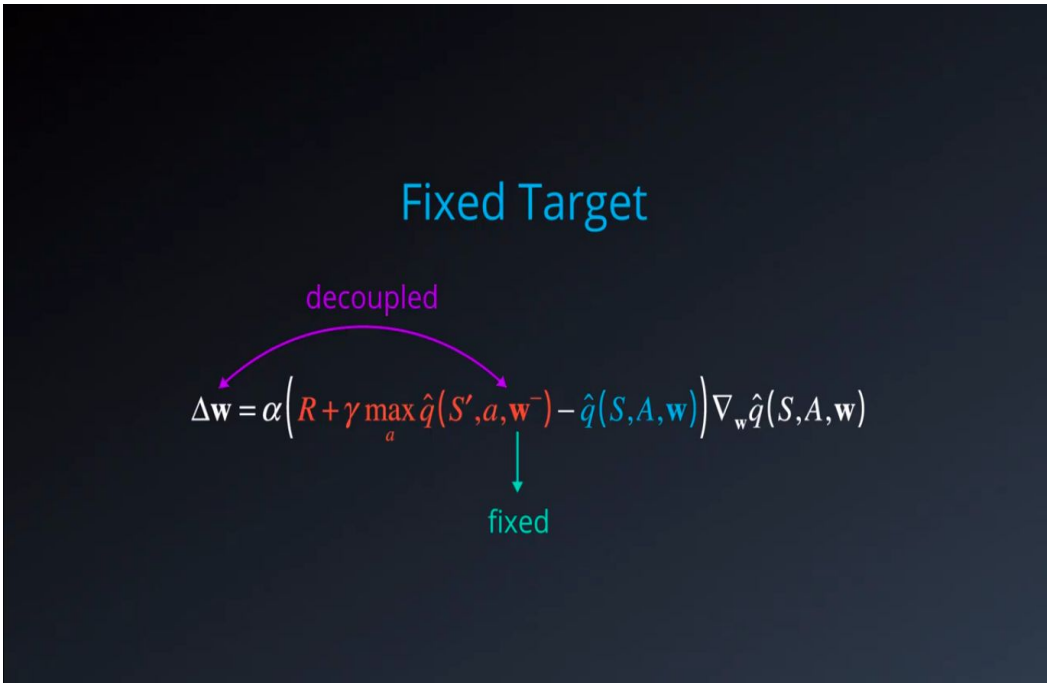

Figure 7: Update rule of Deep Q-Learning

4. Agents Implementation(Code)

The basic code is from the "Lunar Lander" tutorial from the Deep Reinforcement Learning Nanodegree and has been adjusted for being used with the banana environment.

The code consists of :

- **model.py**: Q-Network class is implemented which is used as a function approximator to predict the action to perform depending on the environment observed states. This Neural Network consists of-
    I. the input layer of the size of the state_size(here 37) parameter passed in the constructor
    II. 2 fully connected hidden layers of 1024 nodes each
    III. the output layer of the size of the action_size(here 4) parameter passed in the constructor
- **dqn_agent.py**: Code for the agent used in the environment
    I. The DQN agent class is implemented, as described in the Deep Q-Learning algorithm: *Replay Buffer* initialized, the *target* network and the *local* network are initialized, step() stores steps taken by the agent in Replay Buffer and updates *target* network weights every 4 steps, act() returns actions for the given state as per current policy, learn() updates the Neural Network value parameters using a given batch of experiences from the Replay Buffer.
    II. The ReplayBuffer class implements a fixed-size buffer to store experience tuples (state, action, reward, next_state, done): add() allows to add an experience step to the memory, sample() allows to randomly sample a batch of experience steps for the learning
- **Navigation.ipynb**: This Jupyter notebook allows the agent to train:
    I. Train the agent using DQN
    II. Plot the scores and graph

5. DQN Parameters

Hyperparameters used in dqn_agent.py

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

The Neural-Network use the following architecture :

Input nodes (37) -> Fully Connected Layer (1024 nodes, Relu activation) -> Fully Connected Layer (1024 nodes, Relu activation) -> Output nodes (4)

The Neural Network uses Adam optimizer.

6. Results

With the chosen architecture and parameters mentioned above, my **results** were :

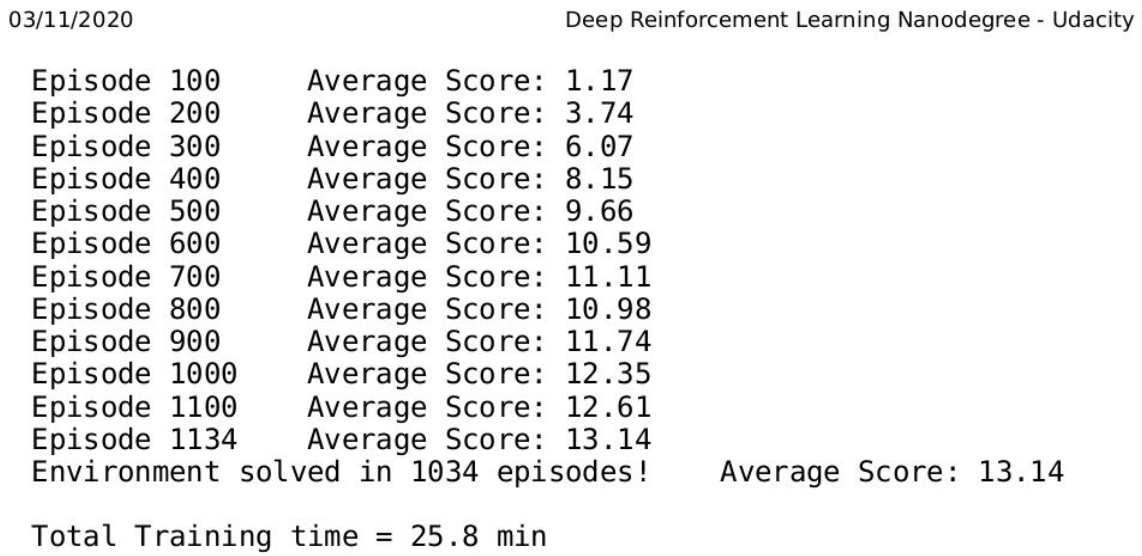

Figure 8: Training-Logs

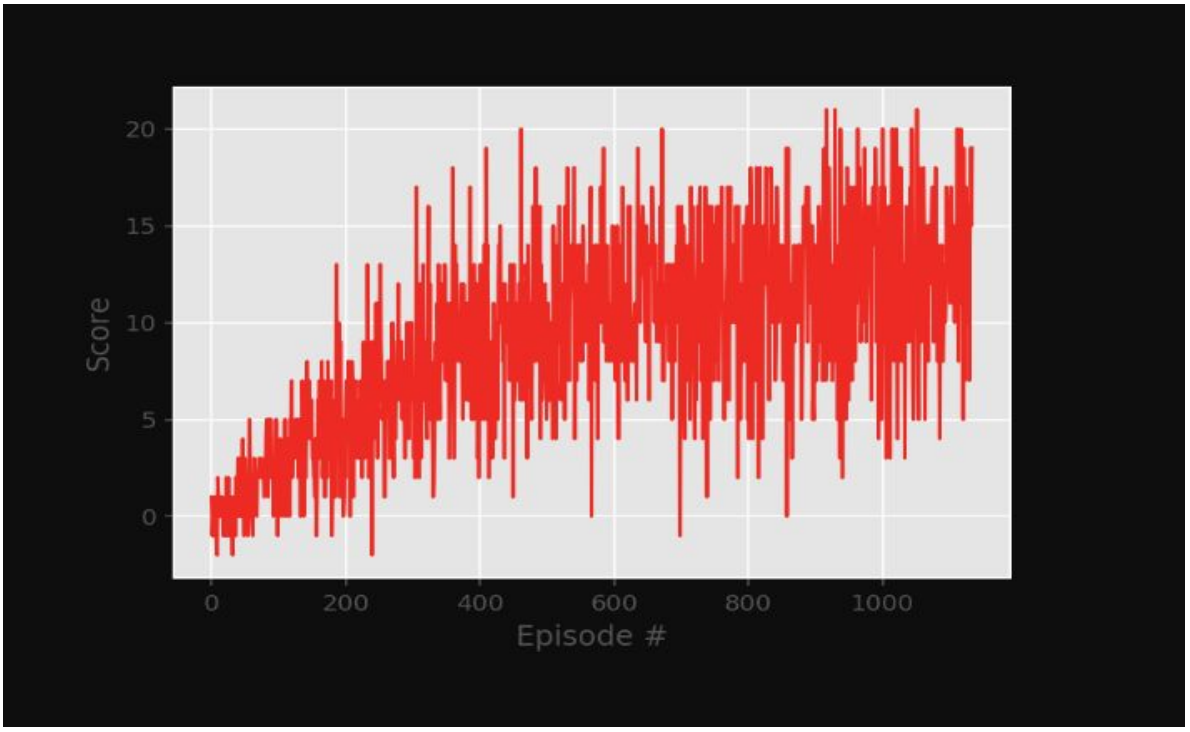

Figure 9: Score-Plot

The agent is able to receive an average reward, over 100 episodes of at least +13(+13.14 to be precise), and in 1034 episodes only (In comparison, according to Udacity's solution code for the project, their agent was benchmarked to be able to solve the project in fewer than 1800 episodes)

7. Ideas for future work

As discussed in the Udacity Course, in the future I would try to train the agent directly from the environment's observed raw pixels instead of using the environment's internal states (37 dimensions) using a CNN at the input of the network in order to process the raw pixels values (after some little preprocessing like rescaling the image size, converting RGB to grayscale, ...). A very good resource to explore this idea is in one of Andrej Kaparthy's blogs- **Deep Reinforcement Learning: Pong from Pixels**

Other enhancements might also be implemented to increase the performance of the agent:

- Using Double DQNs
- Using Dueling DQNs

- Using Prioritized Replay