Project 1: Navigation

1. Introduction

The Navigation project has an agent which navigates an environment with yellow and blue bananas and eats yellow bananas for a reward while avoiding blue bananas (the agent gets penalized for eating a blue banana).

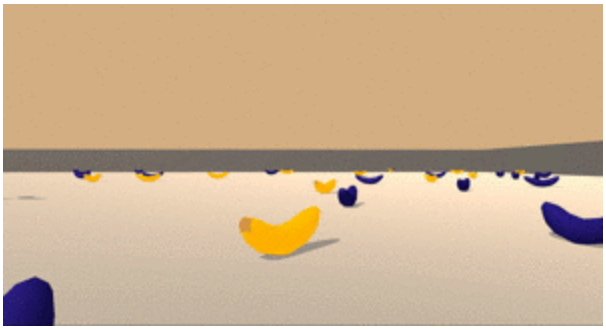

Figure 1 Banana World

We will investigate, implement and assess various Deep Q-Network (DQN) Algorithms that maximize rewards and minimize the number of training episodes:

- Deep Reinforcement Learning with Deep Q-Network (DQN);
- Deep Reinforcement Learning with Experience Replay and Prioritized Experience Replay;
- Deep Reinforcement Learning with Double Deep Q-Network (DDQN);
- Deep Reinforcement Learning with Dueling Deep Q-Network (Dueling DQN);

We are going to train an agent to navigate and collect yellow bananas in a large square world which has both yellow and blue bananas.

We have a reinforcement learning setup with states, actions, rewards between the agent and the environment.

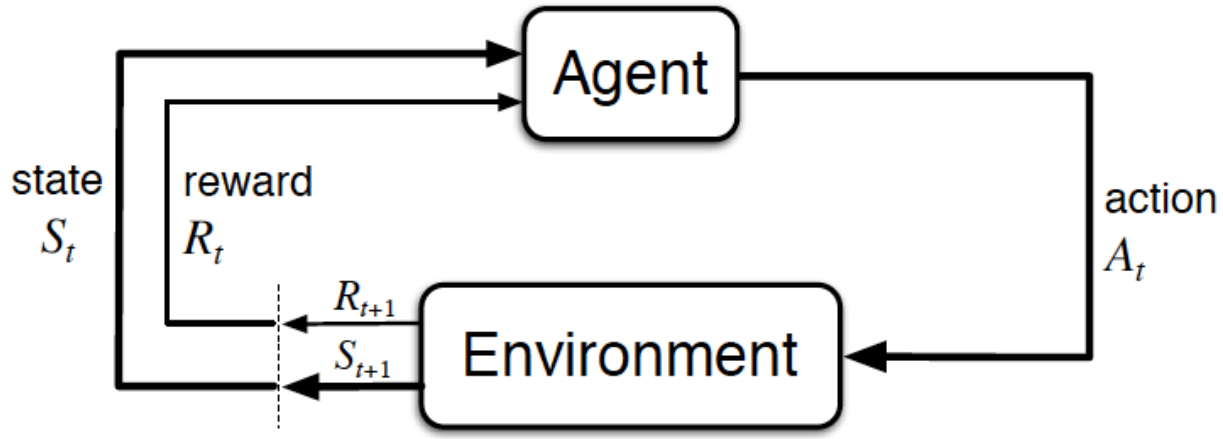

Figure 2 The agent-environment interaction in reinforcement learning. (Source: Sutton and Barton, 2017)

Rewards

A reward of +1 is awarded for collecting a yellow banana and reward of -1 is awarded for collecting a blue banana. Thus the goal of our agent is to collect as many yellow bananas as possible while avoiding blue bananas.

States

The state space has 37 dimensions and contains the agents velocity, along with ray-based perception of objects around agent's forward direction. The agent uses this information to learn how to best select actions.

Actions

The agent can select from four discrete actions corresponding to:

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

We finish this project by looking at potential extensions to the above algorithms.

2. Learning Algorithms

Deep Q-Network Algorithm

At the heart of DQN algorithms are the neural networks that map states to action values. The chosen architecture is a standard neural network with 3 fully-connected layers that maps states to action values:

```
QNetwork(
  (fc1): Linear(in_features=37, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=4, bias=True)
)
```

The Deep Q-Learning (here shown with memory replay) algorithm is a function approximator that works with a QNetwork to find the optimal actions that maximize rewards. The algorithm:

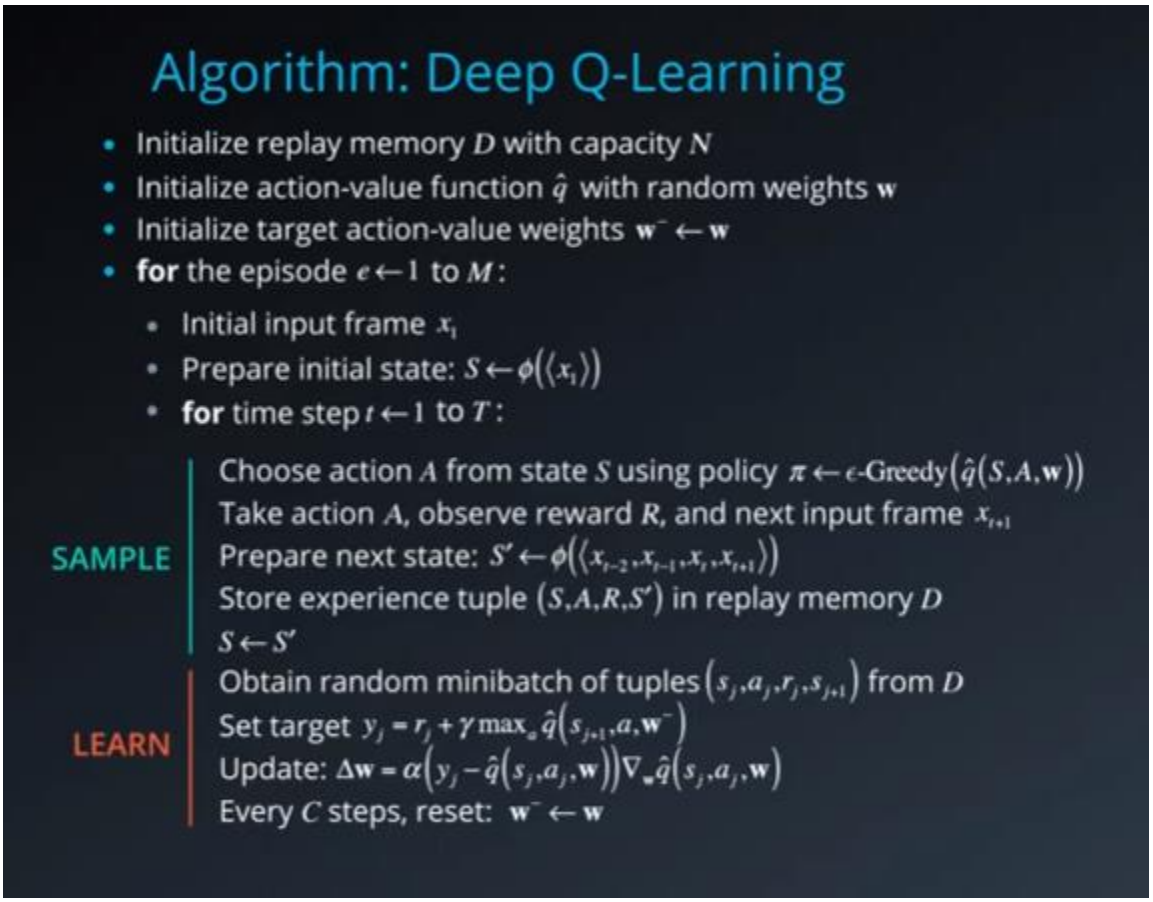

Figure 3 Deep Q-Learning Algorithm

We implement an Agent that implements the above algorithm. The agent has the following key functions:

- step
  Saves experiences in replay memory.
- act
  Determines the actions that should be taken in the current state given the current policy.
- learn

Updates value parameters using batch of experience tuples. Here our agent is modified to either use DQN or Double DQN. We also make adjustments for either using basic replay or prioritized replay.

- soft_update
  We soft update our model parameters.

Our algorithm is using the following parameters which can be tuned to improve various aspects of the model and training process:

```
{'buffer_size': 1000000,
 'batch_size': 64,
 'gamma': 0.99,
 'tau': 0.001,
 'learning_rate': 0.0005,
 'eps': 0.0,
 'update_every': 4,
 'device': device(type='cpu')}
```

Where buffer_size is the size of the memory buffer, batch size is the size of the batches being input into the model as we train, gamma is the discount factor, tau is the soft update parameter, learning_rate is the learning rate, eps is the epsilon greedy parameter, update_every tells us how frequent we must make updates and device tells us whether we are on a GPU or not.

Experience Replay and Prioritized Experience Replay

With prioritized experience we can better decide which memories to replay. Prioritized experience replay (Schaul et al. 2015) improves data efficiency, by replaying more often transitions from which there is more to learn (Hessel et al, 2017). Basic experience replay samples uniformly from experiences in the replay buffer while prioritized experience replay tries to bias towards experiences where there is more to learn and prioritizes them for selection during the learning process.

The four components that make up prioritized replay are:

1. Temporal Difference (TD) Error : This is the agent's transition error - it indicates how surprising or unexpected a transition is. The bigger the temporal difference error the more we expect to learn from it.

   $\delta_t = R_{t+1} + \gamma * max_a \hat{q}(S_{t+1}, a, w) - \hat{q}(S_t, A_t, w)$

2. Priority

   $p_t = |\delta_t| + \epsilon$

3. Sampling Probability: The probability of sampling transition i is given by

   $P(i) = \frac{P_i^\alpha}{\sum_k P_k^\alpha}$

4. Modified Update Rule: Correct the bias introduced by prioritized replay using importance-sampling weights as follows

   $\Delta w = \alpha \left(\frac{1}{N} * \frac{1}{P(i)} \right)^\beta * \delta_i * \nabla_w \hat{q}(S_i, A_i, w)$

The PrioritizedReplayBuffer is based on openai gym base class. It uses additional data structures to aid prioritization.

Double Deep Q-Network

Double DQN (DDQN, van Hasselt, Guez, and Silver 2016) addresses an overestimation bias of Q-learning by decoupling selection and evaluation of the bootstrap action (Rainbow: Combining Improvements in Deep Reinforcement Learning, Hessel et al, 2017).

Here we are selecting the best action using a different set of parameters **w** and evaluate those actions using a different set of parameters **w'**. Effecttively we have two function approximators that must agree on the best course of action. This prevents q-values from exploding in the early stages of learning and fluctuating later on. The only modification we are making to our update step to the DQN algorithm above is as follows:

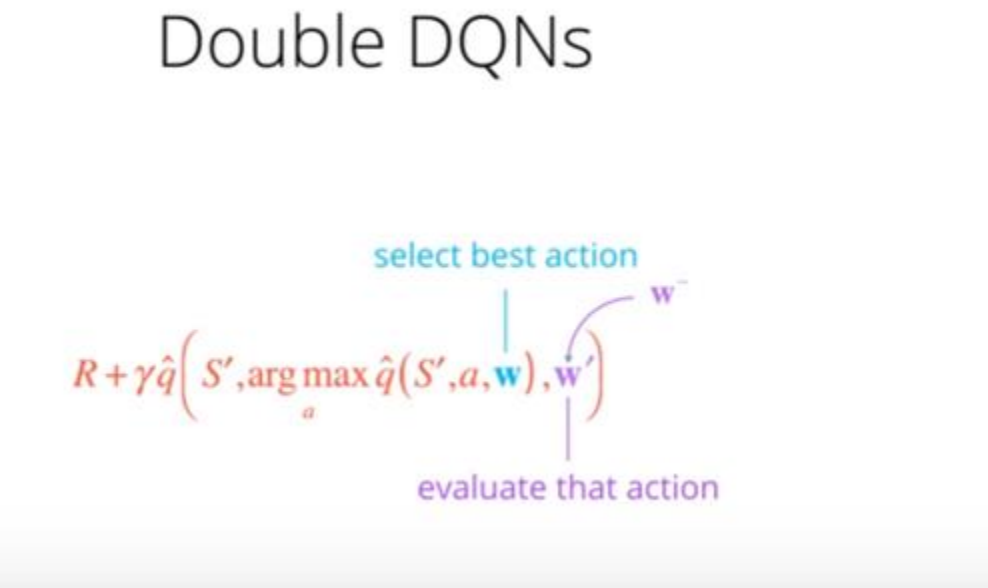

Figure 4 Double DQN Algorithm

Dueling Deep Q-Network

The Dueling DQN network architecture (Wang et al. 2016) helps to generalize across actions by separately representing state values and action advantages (Rainbow: Combining Improvements in Deep Reinforcement Learning, Hessel et al, 2017). Now we have two streams, one that estimates the state values and the other which estimates the advantage values, also called action values. The desired q-values are then obtain by summing up the results of the two streams. We implement this in the DNetwork network architecture:

```
DNetwork(
  (features): Linear(in_features=37, out_features=128, bias=True)
  (fc1_advantage): Linear(in_features=128, out_features=128, bias=True)
  (fc2_advantage): Linear(in_features=128, out_features=4, bias=True)
  (fc1_value): Linear(in_features=128, out_features=128, bias=True)
  (fc2_value): Linear(in_features=128, out_features=1, bias=True)
)
```

3. Implementation

Please see Navigation.ipynb

4. Ideas for future work

- Implement Rainbow

  In this project I have implemented the DQN, Double DQN, Double DQN with Prioritized learning and Dueling DQN. I would like to implement the Rainbow algorithm as proposed by the DeepMind team in their paper Rainbow: Combining Improvements in Deep Reinforcement Learning (2017).

  In this paper they propose combining DQN, DDQN, DDQN with Prioritized Learning, Dueling DQN, A3C, Distributional DQN and Noisy DQN into a single potent algorithm. They show that this algorithm significantly outperforms all the above algorithms individually. They argue that combining these algorithms is possible because the algorithms themselves are addressing different issues related to Deep Reinforcement Learning.

  To create Rainbow, in addition to the algorithms I have already implemented, I will first have to implement the A3C, Distributional DQN and Noisy DQN algorithms.

- Hyper-parameter tuning

I could tune the hyper-parameters more, make the network deeper, use drop out, change the architecture completely.