Udacity Deep Reinforcement Learning Nano Degree

Report of Project 1: Navigation

Pongsasit Thongpramoon

Section 1: Learning Algorithm

Section 1.1: Deep Q-learning, algorithm overview

Section 1.2: Models and Agent Code explanation

Section 2: Plot of Rewards

Section 2.1: Training code explanation

Section 2.2: Plot of Rewards

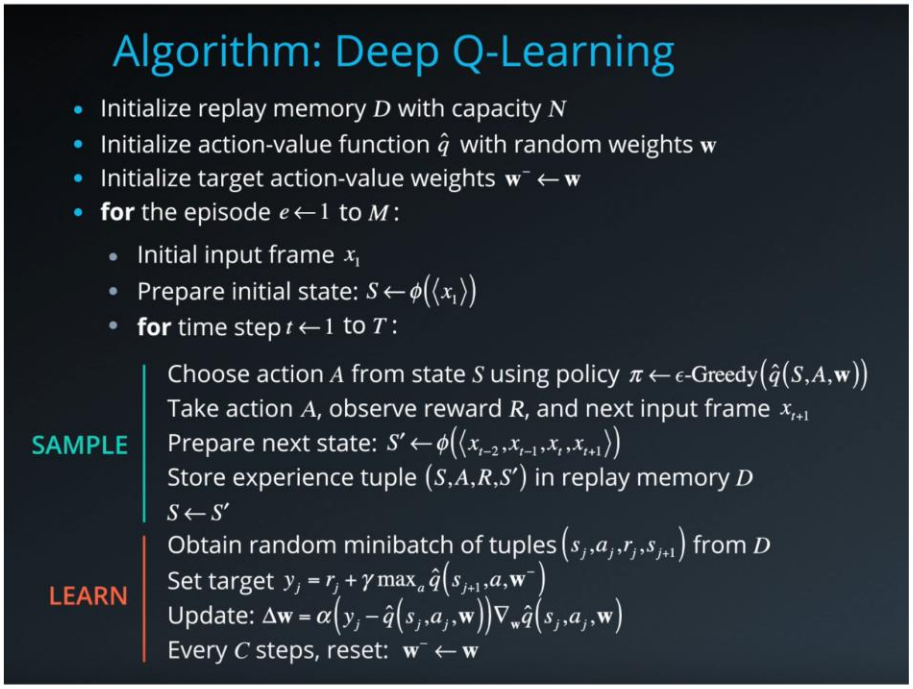
Section 3: Ideas for Future Work

Section 3.1: Improvement of Model Architecture

Section 3.2: Improvement of Algorithm

Section 1: Learning Algorithm

Section 1.1: Deep Q-learning, algorithm overview



Ref: https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893

This Deep Q-Learning algorithm combined Q-learning (Sarsa Max in Reinforcement Learning) And Deep Learning (to learn the Q table). To explain above pseudo-code to more generic term, we need to know

1. Replay memory: this memory is the space to save an experience tuple (S, A, R, S') of an agent. This saved experience tuple will use for model to learn and update weight.
2. Q targets: A straightforward way to make target values more stationary is to have a separate network called the target network. Our targets are fixed for as many steps as we fix our target network. This improves our chances of convergence.

Section 1.2: Model and Agent Code explanation

1. model.py (Q Network)
   3 Fully connected Neural network (without any dropout and batch normalization).
   First layer: input size =37 (state size of the environment), output size =128
   Second layer: input size =128, output size =128
   Third layer: input size =128, output size =4 (Forward, Backward, Left, Right)
2. dqn_agent.py (Agent)
   BUFFER_SIZE = int(1e5) , BATCH_SIZE = 64, GAMMA = 0.99, TAU = 1e-3 , LR = 5e-4, UPDATE_EVERY = 4

   Class Agent:

   Initialize 2 instances of the local network and the target network.

   Initialize the memory buffer (Replay Buffer).

   function step():

   Allows to store a step taken by the agent (state, action, reward, next_state, done) in the replay buffer every 4 steps. And if their are enough samples available in the Replay Buffer, update the target network weights with the current weight values from the local network.

   function act():

   Use epsilon greedy selection to select the action. And returns actions for the given state as per current policy.

   function learn():

   Update the Neural Network value parameters using given batch of experiences from the Replay Buffer.

   function soft_update():

   Softly updates the value from the target Neural Network from the local network weights.

   Class ReplayBuffer:

   function add(): allows to add an experience step to the memory.

   function sample(): allows to randomly sample a batch of experience steps for the learning.

Section 2: Plot of Rewards

Section 2.1: Training code explanation

Navigation.ipynb

- Import the Necessary Packages.
- Examine the State and Action Spaces.
- Take Random Actions in the Environment (No display).
- Train an agent using DQN

  for each episode.
  reset an environment, get state, initialize score to zero.

  In each episode run over all of the steps until the episode done.
  (Actor)
  Let an actor act.

  (Environment)
  Get an action from the actor to the environment.
  Get next state.
  Get reward.
  Get done status.

  (Actor)
  Store a step taken by the agent.
  Update the target network weights with the current weight values from the local network.

  Update by add reward to score that initialized per episode.

  check if Done then break out the steps loop.
  Update scores of all episodes (list) by append the score (float).
  Decrease epsilon decay.

  And save model if the model can reach the score we want

  Return the train function with scores (list of score of every episodes).

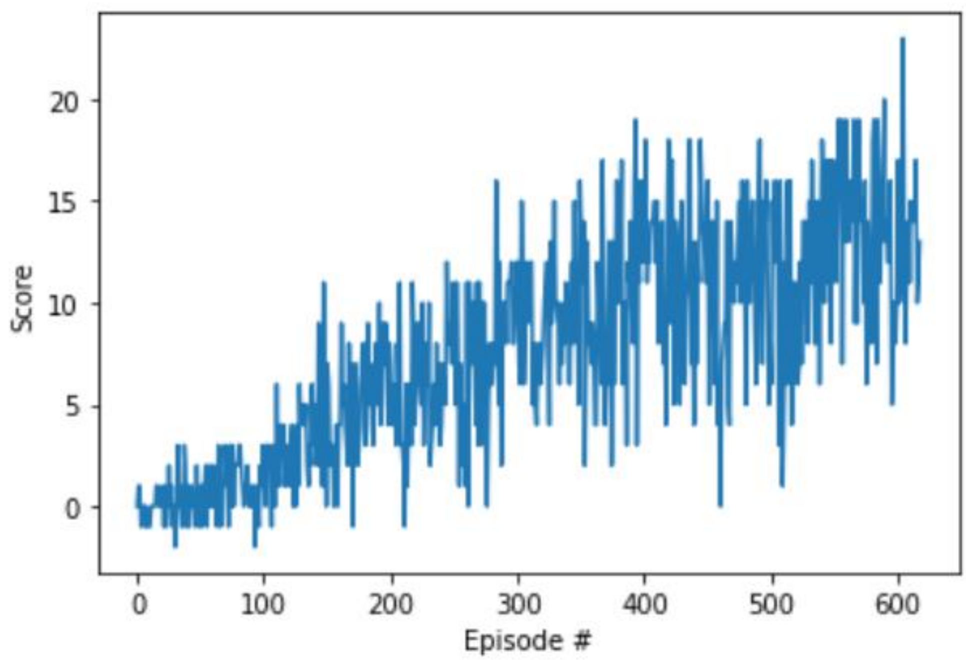
- Plot the scores

Section 2.2: Plot of Rewards

In [15]:
```
scores = dqn(agent)

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.show()
```

```
Episode 100	Average Score: 0.56
Episode 200	Average Score: 3.97
Episode 300	Average Score: 6.76
Episode 400	Average Score: 9.65
Episode 500	Average Score: 11.35
Episode 600	Average Score: 12.46
Episode 618	Average Score: 13.02
```



At the episode 618th, the system can get average score more than 13.

Section 3: Ideas for Future Work

Section 3.1: Improvement of Model Architecture

- Including a drop out layer to reduce the chance of overfitting.
- Use L2 regularization to reduce the chance of overfitting.

Section 3.2: Improvement of Algorithm

- Double DQN (DDQN)

Algorithm 1 Double Q-learning

Initialize Q^A,Q^B,s
repeat
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s',a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s',a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

Pseudo-code Source: "Double Q-learning" (Hasselt, 2010)

The original Double Q-learning algorithm uses two independent estimates Q local action and Q target action.
With a 0.5 probability,
We use estimate Q local action to determine the maximizing action, but use it to update Q target actor.
We use Q target actor to determine the maximizing actor, but use it to update Q local action.
By doing so, we obtain an unbiased estimator Q local actor (state, argmax Q{next state, action) for the expected Q value and inhibit bias.