Syed Bilal Mehdi
Project: Navigation
Deep Reinforcement Learning

Learning Algorithm (DQN)

The learning algorithm used in the project is the standard DQN algorithm. For the sake of completion, a high-level short review DQN within the context of Q-learning is summarized below:

(Vanilla) Q-learning:

Q-learning, one of the most well-known reinforcement learning algorithm, is an off-policy method that learns a Q-function using the Bellman equation as:

$$Q(s,a) = r + \gamma \max_a Q(s',a') \qquad (1)$$

Unfortunately, for large (or continuous) state spaces, the above equation can be intractable. In such cases it is common to use a function approximator, such as a neural network, to represent the Q-function. In such cases, Equation (1) is replaced by an optimization process that minimizes the loss function:

$$L = norm\left(Q(s,a) - \left(r + \gamma \max_a Q(s',a')\right)\right).$$

As may be clear from the above loss function, the purpose of this optimization is to imitate application of Equation (1) for the case when function approximation is at play.

The features that differentiate DQN from vanilla Q-learning algorithm are (i) experience replay and (ii) fixed Q-targets.

Experience Replay:

Experience replay involves a couple of steps. First, as the agent interacts with the environment, tuples of *state*, *action*, *reward*, and *next_state* are stored in a chunk of memory. Second, after every few steps, a number of stored tuples are pulled from the memory for the agent to learn from.

This provides two primary advantages for the algorithm. (i) A standard assumption in any learning algorithm is the independence of samples used for optimization. If the tuples pulled for training are in the same order as experienced by the agent (for example, in standard Q-learning), this assumption is not met. However, by pulling tuples randomly from the storage, experience reply breaks this correlation increasing robustness of the algorithm. (ii) Experience replay allows the algorithm to learn from an experience multiple times, thereby making better use of – often expensive to collect – data.

Target Network:

Q-learning can be very unstable when a non-linear function approximator (such as neural network) is used. This can be attributed to the fact that function approximation makes Q(s, a) to be correlated to Q(s', a') thus making the update of Eq. (1) feedback on itself.

DQN breaks this direct feedback by using two different networks namely Q_{local} and Q_{target}. The update to the Q function in then performed as $Q_{local}(s,a) = r + \gamma \max_a Q_{target}(s',a')$. Here Q_{target} is updated (comparatively) infrequently keeping the Q function update more stable.

Results

The primary exploration performed in this project was the effect of different hyperparameters on the learning.

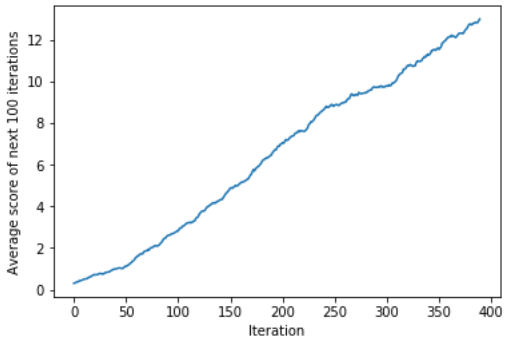
Initial Settings

As a starting point, code and parameters from an exercise earlier in the course were used. The environment within these settings was solved very quickly. The details follow:

Parameters: $\gamma = 0.99, \tau =$ 1e-03, number of hidden layers $=$ 2, units in each layer: 64, 64.

Number of episodes to solve environment: 390

Average score over 100 iterations is shown in the figure below.



It can be seen that the environment was solved fairly easily by the learning algorithm.

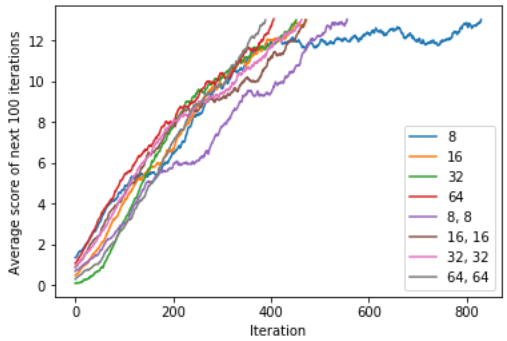
Neural Network Size

Since the environment was solved quickly with the initial settings, smaller sizes of the networks (both target and local) were tried. The environment was solved by networks as small as 1 hidden layer with 8 units only. The details follow:

Parameters: $\gamma = 0.99, \tau =$ 1e-03, number of hidden layers $=$ 1 *or* 2, units in each layer: 8, 16, 32 or 64.

Number of episodes to solve environment: around 400 (for 2 hidden layers with 64 units each) to up to 831 (1 hidden layer with 8 units)

Average score over 100 iterations for all variations tried are shown below.



It can be seen in the figure above that larger networks have performed better for the most part. However, Q networks with only one hidden layer and 64 units performed almost as good as the networks with two hidden layers and same number of units in each layer.

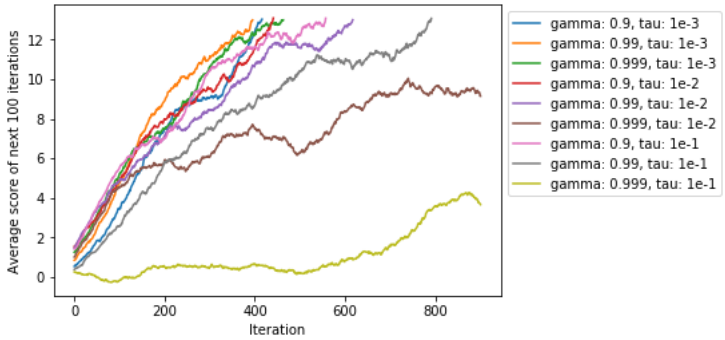
γ and τ

Next, the network size was frozen to one hidden layer with 64 units and different values of γ and τ were tried. Some values explored failed to solve the environment. The details follow:

Parameters: $\gamma = 0.9, 0.99 \text{ or } 0.999, \tau = 1\text{e-}1, 1\text{e}{-2}, \text{or } 1\text{e-}3, \text{number of hidden layers } = 1, \text{units in each layer: } 64.$

Number of episodes to solve environment: less than 400 ($\gamma = 0.99, \tau = 1\text{e} - 3$; same as the setting from before) to more than 900 at which point it was declared that the setting failed to solve the environment

Average score over 100 iterations for all variations tried are shown below.



It can be seen that small values of τ combined with large values γ made the learning the slowest. This could be because slow updating of target network may lead to old (and inaccurate) Q values spread throughout the local network messing the whole system up.

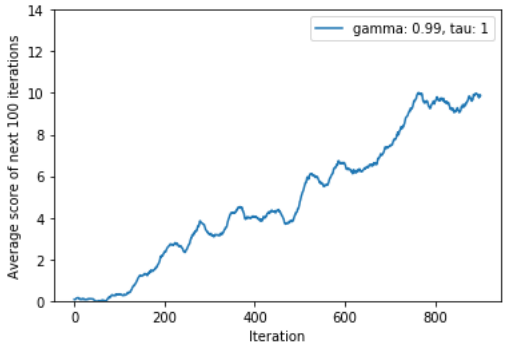
Without Separate Target Network

Lastly, learning was explored without having a separate target network. This was achieved by simply setting $\tau = 1$. This makes sure that the target network is always the same as the local network. The details follow:

Parameters: $\gamma = 0.99, \tau = 1.0, \text{number of hidden layers } = 1, \text{units in each layer: } 64.$

Number of episodes to solve environment: more than 900 at which point it was declared that the setting failed to solve the environment

Average score over 100 iterations for all variations tried are shown below.



It can be seen that learning is very slow in this setting and may never succeed to solve the environment.

Future Work:

Although this version of the environment could be solved quickly with decent ranges of parameters, different versions of the environment can be explored for a tougher challenge – such as using pixel data only. A more difficult variation would also justify use of more sophisticated methods such as double DQN, prioritized replay and even Rainbow. Finally, the effects of not having replay experience replay could be explored by making the buffer size to one and updating the network after every step.