Navigation Project Report

Implementation

The agent is trained using a Deep Q-Network algorithm (DQN).

The agent selects and executes actions according to an ϵ-greedy policy based on Q.

The DQN algorithm modifies standard online *Q-learning* in **two ways** to make it suitable for training large neural networks without diverging:

First, Experience Replay

By building a database of experience tuples (S, A, R, S'), called a **replay buffer,** and sampling a small batch of experience tuples from the buffer *at random*, which prevents action values from oscillating or diverging catastrophically.

The tuples are gradually added to the buffer as the agent interacts with the environment.

Second, Fixed Q-Targets

In this modification, we use a separate network for updating the parameters w in the network $\hat{q}$ to better approximate the action value corresponding to state S and action 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.

This modification makes the algorithm more stable compared to standard online Q-learning, where an update possibly leads to oscillations or divergence of the policy.

DQN Model Architecture

At the heart of the agent, there is a Deep Neural Network that acts as a function approximator.

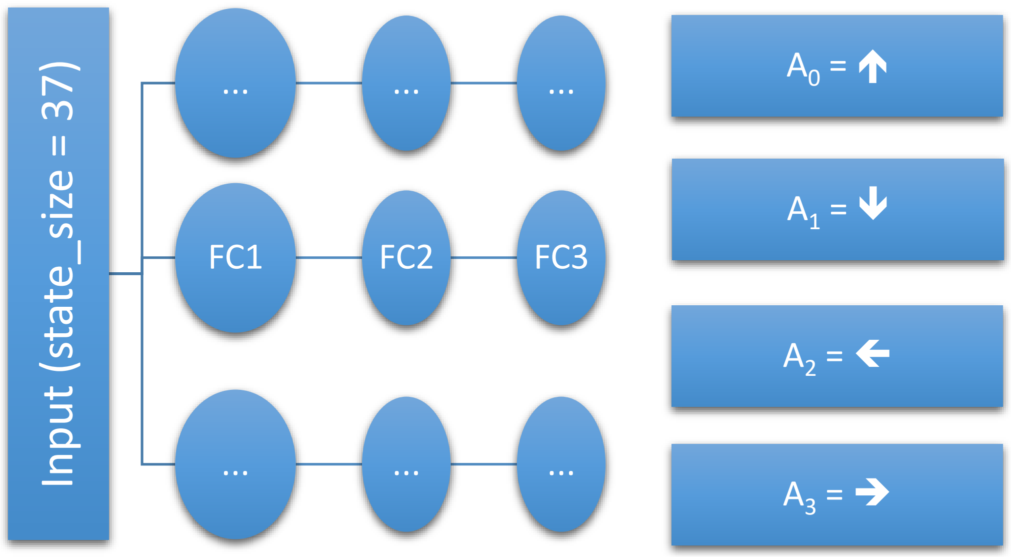
The used DQN model consists of **three fully connected layers**.

The first hidden layer takes the size of the state space (*state_size = 37*) as input, and outputs *64 units*, which are passed as input to the second hidden layer.

The second hidden layer outputs *64 units*, which are passed as input to the third and final output layer.

The final output from the third output layer is the size of the action space (*action_size = 4*), corresponding to the four available discrete actions:

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



Learning Algorithm

- Initialize replay memory (*ReplayBuffer*) with capacity (*BUFFER_SIZE = int(1e5)*).
- Initialize action-value function (*qnetwork_local*) with random weights *w*.
- Initialize target action-value weights (*qnetwork_target*) w^- ← *w*.

```
• for episode (i_episode ← 1 to 2000):
    o Prepare initial state: (state = env_info.vector_observations[0])
    o Initialize score ← 0
    o for time step t ← 1 to 1000:
        ▪ Choose action A from state S using policy $\pi \leftarrow \epsilon$-Greedy($\hat{q}(S, A, w)$)
        ▪ Take action A, observe reward R
        ▪ Prepare next state: (next_state = env_info.vector_observations[0])
        ▪ Store experience tuple (S,A,R,S`) in replay memory (ReplayBuffer)
        ▪ state ← next_state
        ▪ Add reward to score
        ▪ Obtain random mini-batch of experience tuples (states, actions,
          rewards, next_states, dones) from memory
        ▪ Set target Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
        ▪ Update network weights (self.optimizer.step())
        ▪ Every 4 steps, reset: $w^{-} \leftarrow w$ (self.soft_update(self.qnetwork_local,
          self.qnetwork_target, TAU))
```

Results

```
Episode 100	Average Score: 1.63
Episode 200	Average Score: 6.53
Episode 300	Average Score: 8.78
Episode 400	Average Score: 11.32
Episode 448	Average Score: 13.03
Environment solved in 348 episodes!	Average Score: 13.03
```

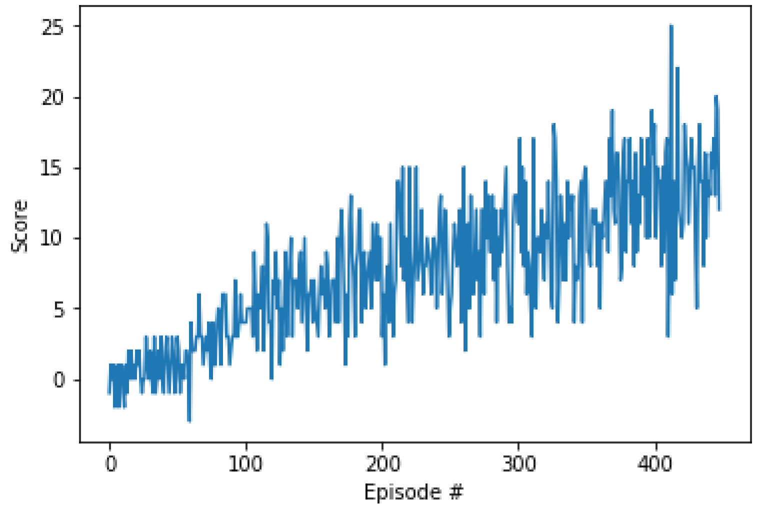


Figure 1 Plot of Rewards

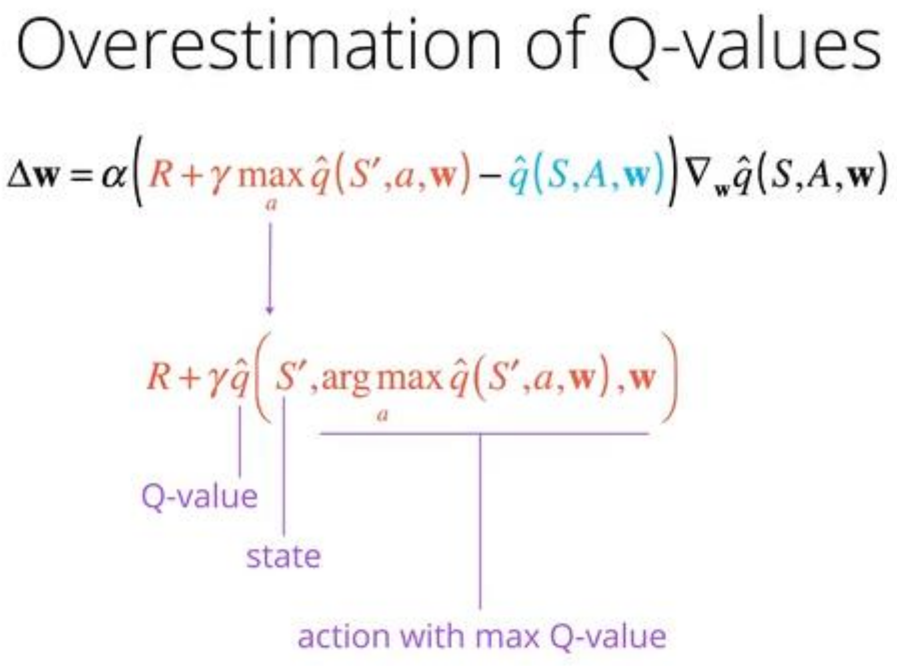
The agent was able to receive an average reward (over 100 episodes) of **13.03** in **348 episodes**.

Future Improvements

Several improvements could be implemented to enhance the agent's performance, including:

1) Double DQN

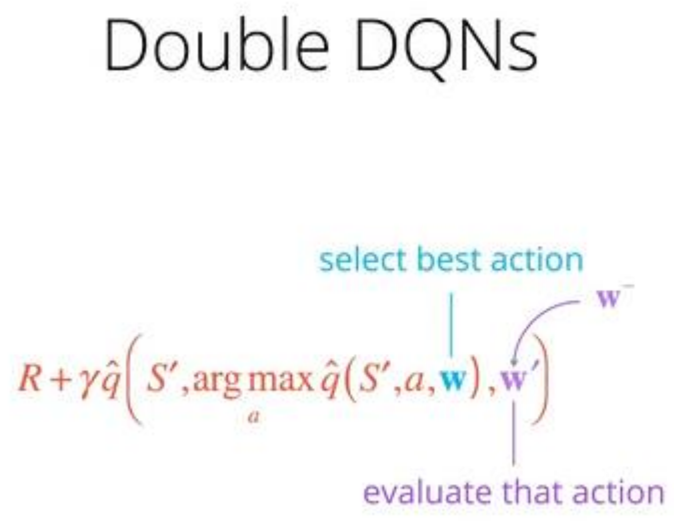
The max operator in standard Q-learning and DQN uses the **same values** (w) both to select and to evaluate an action. This makes it more likely to select overestimated values, resulting in overoptimistic value estimates.


Overestimation of Q-values
$\Delta \mathbf{w} = \alpha\left(R + \gamma \max_a \hat{q}(S',a,\mathbf{w}) - \hat{q}(S,A,\mathbf{w})\right)\nabla_{\mathbf{w}}\hat{q}(S,A,\mathbf{w})$
$R + \gamma\hat{q}\left(S', \arg\max_a \hat{q}(S',a,\mathbf{w}), \mathbf{w}\right)$
Q-value
state
action with max Q-value

To prevent this, we can **decouple** the *selection* from the *evaluation*, using Double Q-learning

The idea of **Double Q-learning** is to reduce overestimations in the original DQN by decomposing the max operation in the target into action selection and action evaluation.

We will evaluate the greedy policy according to the online network, but use the target network to estimate its value.



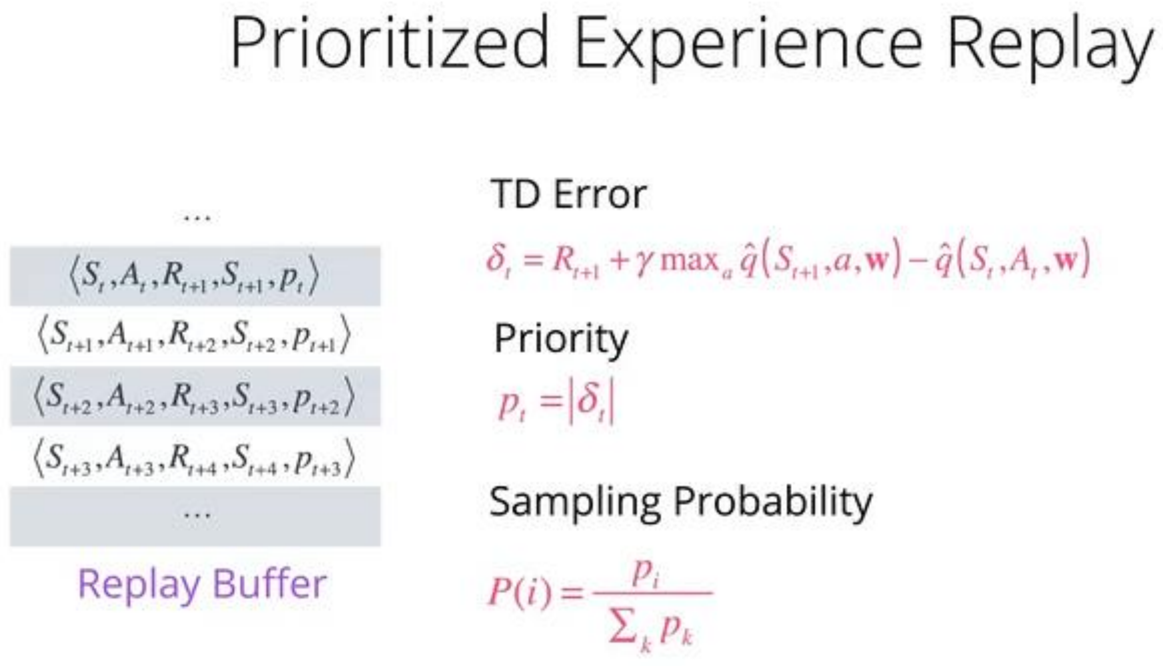
Double DQNs

$$R+\gamma\hat{q}\Big(S',\underbrace{\arg\max_{a}\hat{q}(S',a,\mathbf{w})}_{\text{select best action}},\underbrace{\mathbf{w}'}_{\text{evaluate that action}}\Big)$$

2) Prioritized Experience Replay

Regular *Experience replay* approach simply replays transitions at the **same frequency** that they were originally experienced, regardless of their significance.

Prioritized Experience Replay replays **important** transitions *more frequently*, and therefore learns more efficiently.



Prioritized Experience Replay

…

$\langle S_t, A_t, R_{t+1}, S_{t+1}, p_t \rangle$

$\langle S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, p_{t+1} \rangle$

$\langle S_{t+2}, A_{t+2}, R_{t+3}, S_{t+3}, p_{t+2} \rangle$

$\langle S_{t+3}, A_{t+3}, R_{t+4}, S_{t+4}, p_{t+3} \rangle$

…

Replay Buffer

TD Error

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

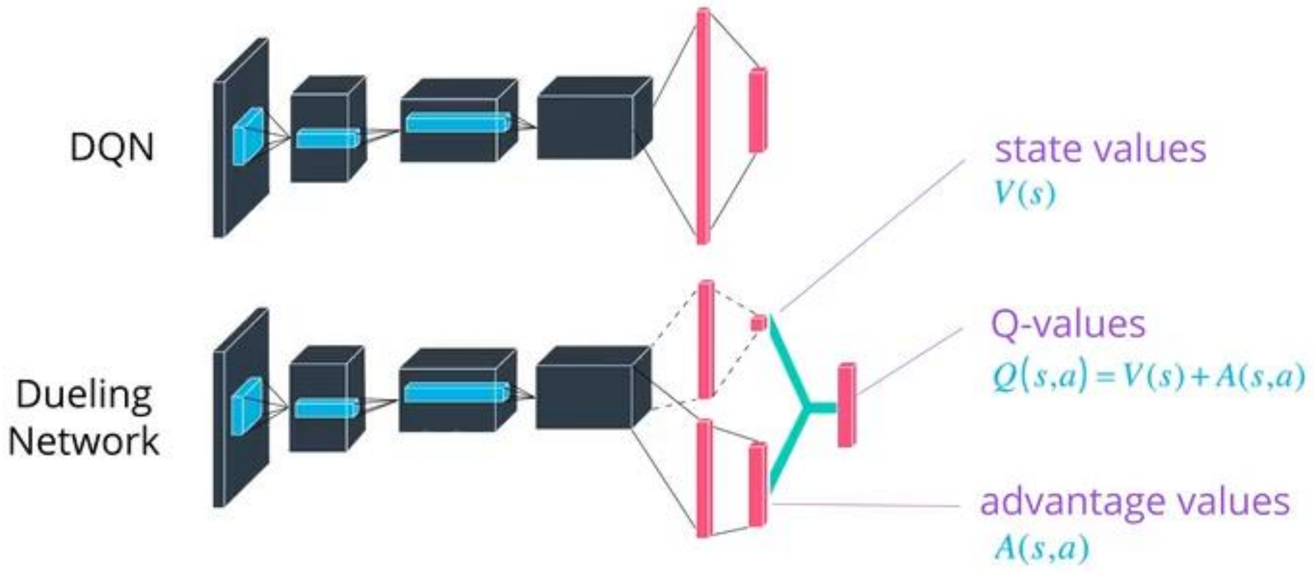
Priority

$$p_t = |\delta_t|$$

Sampling Probability

$$P(i) = \frac{p_i}{\sum_k p_k}$$

3) Dueling DQN



Use **two streams** of neural networks instead of one; one stream will estimate the **state value function**, and the other estimates the **advantage values** for each **action**.

These streams may share some layers in the beginnings, such as convolutional layers, and then branch off with their fully connected layers.

Finally, the desired Q-values are obtained by combining the state and advantage values.

Hence, we can assess the value of each state directly, without having to learn the effect of each action.

4) Learning from Pixels

Add a convolutional neural network (CNN) layer to the DQN architecture, in order to learn directly from pixels.

References

1) Human-level control through deep reinforcement learning

https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf

2) Lesson 2.2: Deep Q-Networks (Udacity's Deep Reinforcement Learning Nanodegree)
3) DQN LunarLander Exercise: Udacity's Deep Reinforcement Learning Nanodegree GitHub Repo

https://github.com/udacity/deep-reinforcement-learning/tree/master/dqn

4) Deep Reinforcement Learning with Double Q-learning

https://arxiv.org/abs/1509.06461

5) Prioritized Experience Replay

https://arxiv.org/abs/1511.05952

6) Dueling Network Architectures for Deep Reinforcement Learning

https://arxiv.org/abs/1511.06581