1.Algorithm

Q-learning algorithm

1) reward hypothesis
maximization of expected cumulative reward

2)action
for robot, the joints real force value is the action

3)state
for robot,like the position and velocities of joints, measurements of ground, contract sensor data

4)reward
negative is penalize value

5)goal of agent ()
maximization of expected cumulative reward,R(1)+...+R(t)is inthe past already decided,future is G(t)=R(t+1)+R(t+2)+..., R(t+1) is immediate reward

6) discount reward for continuous task

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \text{ where } \gamma \in [0,1]$$

gamma is discount rate

7)policy is mapping between action and state
deterministic policy S→A and stochastic policy SxA → [0,1]

8)state-value function v* and action-value function q*

9)Bellman Equation

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s].$$

10)Q-table
estimate the expected return from action and reward and the action function with same policy . if exist many same policy , the value is average.

11) convergence follows from the Law of Large Numbers

12)Monte_Carlo
about sample and Law of Large Numbers, every-visit MC is biased and low mean squared error (MSE). It is about to estimate Q-table

13)epsilon-greedy
use sectioned function to find action-value function estimate Q, epsilon value control which interaction is , explore or exploitation. Although, epsilon in math can not guarantee convergence but we can get better result in practice. a policy is pi, A(s) is the space number of action. it is about the final stochastic policy probability

$$\pi(a|s) \longleftarrow \begin{cases} 1-\epsilon+\epsilon/|\mathcal{A}(s)| & \text{if } a \text{ maximizes } Q(s,a) \\ \epsilon/|\mathcal{A}(s)| & \text{else} \end{cases}$$

14) Temporal Difference (TD,Sarsa)
methods will instead update the Q-table after every time step, and gamma is discount rate.

(From Temporal-Difference Control)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(\underbrace{R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}_{\text{alternative estimate}} - \underbrace{Q(S_t, A_t)}_{\text{current estimate}})$$

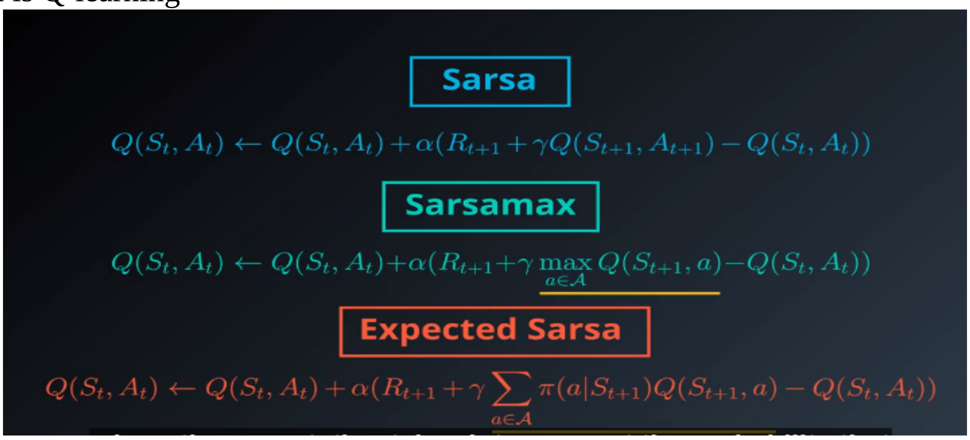
(From udacity)

Hyperparameters （wiki）
Learning rate (alpha)
The learning rate determines to what extent newly acquired information overrides old information. A factor of 0 will make the agent not learn anything, while a factor of 1 would make the agent consider only the most recent information.

Discount factor (gamma)
The discount factor determines the importance of future rewards. A factor of 0 makes the agent "opportunistic" by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward. If the discount factor meets or exceeds 1, the {\displaystyle Q} Q values may diverge.

Initial conditions (Q(s0, a0))
Since SARSA is an iterative algorithm, it implicitly assumes an initial condition before the first update occurs. A low (infinite) initial value, also known as "optimistic initial conditions",[4] can encourage exploration: no matter what action takes place, the update rule causes it to have higher values than the other alternative, thus increasing their choice probability. In 2013 it was suggested that the first reward r could be used to reset the initial conditions. According to this idea, the first time an action is taken the reward is used to set the value of Q. This allows immediate learning in case of fixed deterministic rewards. This resetting-of-initial-conditions (RIC) approach seems to be consistent with human behavior in repeated binary choice experiments.[5]

15)sarsamax is Q-learning



(from udacity)

On-policy TD control methods (like Expected Sarsa and Sarsa) have better online performance than off-policy TD control methods (like Sarsamax)

deep Q-learning

1) deep Q-learning include three cnn layers and two Full convolution layers for image feature abstraction , return action value

2)experience replay

replay buffer is tuples (S， A， R， S‘), experience replay is the act of sampling a small batch of tuples from the replay buffer and learn more from individual tuples multiple times

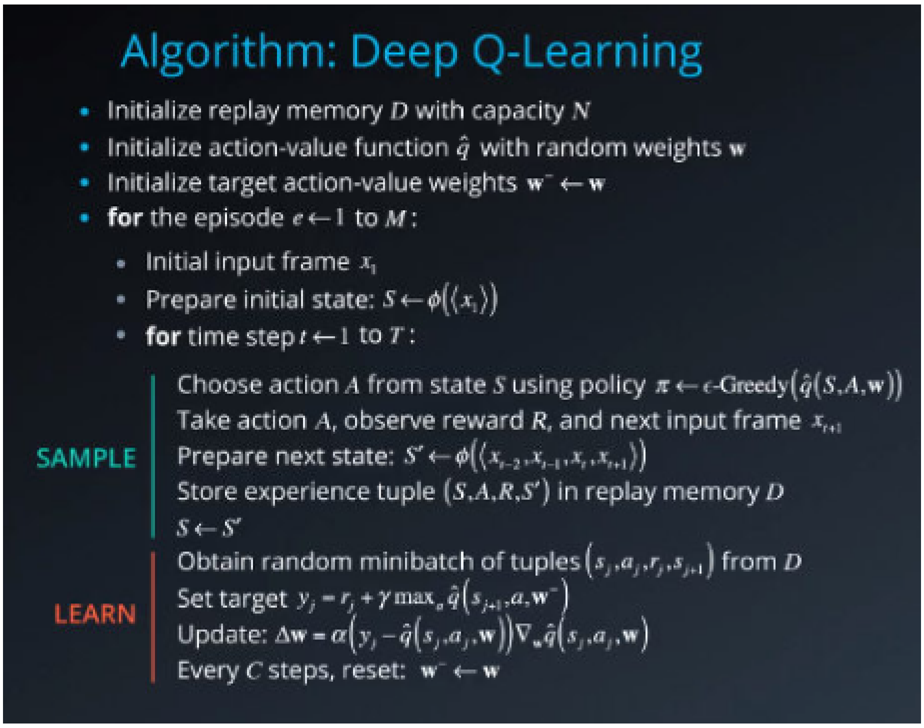
3)Fixed Q targets

$$\Delta w = \alpha \cdot \overbrace{(\underbrace{R + \gamma \max_a \hat{q}(S', a, w^-)}_{\text{TD target}} - \underbrace{\hat{q}(S, A, w)}_{\text{old value}})}^{\text{TD error}} \nabla_w \hat{q}(S, A, w)$$

(from udacity)

where w- are the weights of a separate target net work that are not changed during the learningstep and （S， A， R， S‘) is an experience tuple

4) specific algorithm

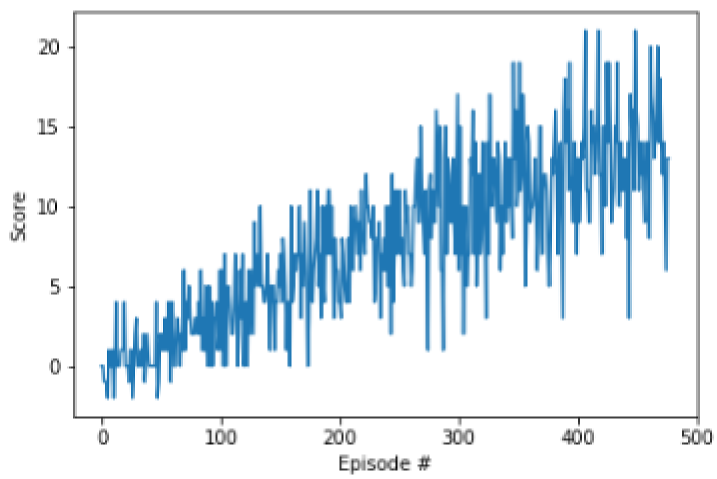


code

```
def dqn(n_episodes=2000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):
    """Deep Q-Learning.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
        eps_start (float): starting value of epsilon, for epsilon-greedy action selection
        eps_end (float): minimum value of epsilon
```

```
    eps_decay (float): multiplicative factor (per episode) for decreasing epsilon
    """
    scores = []                        # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    eps = eps_start                    # initialize epsilon

    # get the default brain
    brain_name = env.brain_names[0]
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]
        state = env_info.vector_observations[0]            # get the current state
        score = 0
        for t in range(max_t):


            action = agent.act(state, eps)

            env_info = env.step(action)[brain_name]        # send the action to the environment
            next_state = env_info.vector_observations[0]   # get the next state
            reward = env_info.rewards[0]                   # get the reward
            done = env_info.local_done[0]

            #next_state, reward, done, _ = env.step(action)
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break
        scores_window.append(score)       # save most recent score
        scores.append(score)              # save most recent score
        eps = max(eps_end, eps_decay*eps) # decrease epsilon
        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)),
end="")
        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
        if np.mean(scores_window)>=13.0:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode-100,
np.mean(scores_window)))
            torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
            break
    return scores
```

score- episode
change picture



```
def learn(self, experiences, gamma):
        """Update value parameters using given batch of experience tuples.

        Params
        ======
            experiences (Tuple[torch.Variable]): tuple of (s, a, r, s', done) tuples
            gamma (float): discount factor
        """
        states, actions, rewards, next_states, dones = experiences

        # Get max predicted Q values (for next states) from target model
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)
        # Compute Q targets for current states
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # Compute loss
        loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # ------------------- update target network ------------------- #
        self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)
```

Parameters

1） state_size
need consider the computer store, action representation by cnn

2） action_size
come from action types

Future
1) double DQN : add state by other FC

2) priorited experience replay : use probability to get experience

3) Dueling DQN :

4) multi-step bootstrap targets : multi-step reward

5) distributional DQN : pproximate the distribution of returns instead of the expected return

6) noisy DQN : a noisy linear layer that combines a deterministic and noisy stream

7) rainbow : all the aforementioned components into a single integrated agent