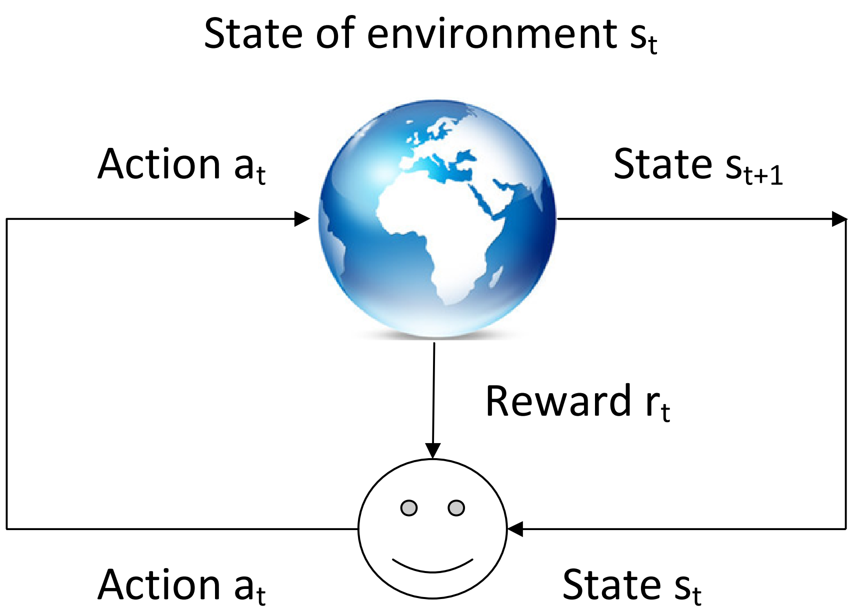
Обучение с подкреплением



s_t - состояние среды в момент t

r_t – награда получаемая агентом в момент t

после действия a_t

Среда переходит в состояние s_{t+1}

p(a|s) policy function

q(a,s) Q-value function

v(s) – value function

Общая постановка задачи обучения с подкреплением

Дан Марковский процесс принятия решений и соответственно функции вероятности перехода $P(s, a, s') = P(s'|a, s)$ из состояния s в состояние s$'$, a – действие.

Политика $\pi(a, s) = P(a|s)$. Ставится задача поиска политики максимизирующей функционал:

$$J(\pi) = E\left[\sum_{t=0}^{\infty} \gamma^t R_t \,|\pi\right]$$

Функция ценности состояния (value function)

$$v(s_{t'}) = E\left[\sum_{t=t'}^{\infty} \gamma^t R_t \,|\pi, s_{t'}\right]$$

Функция ценности действия из состояния (q-value function)

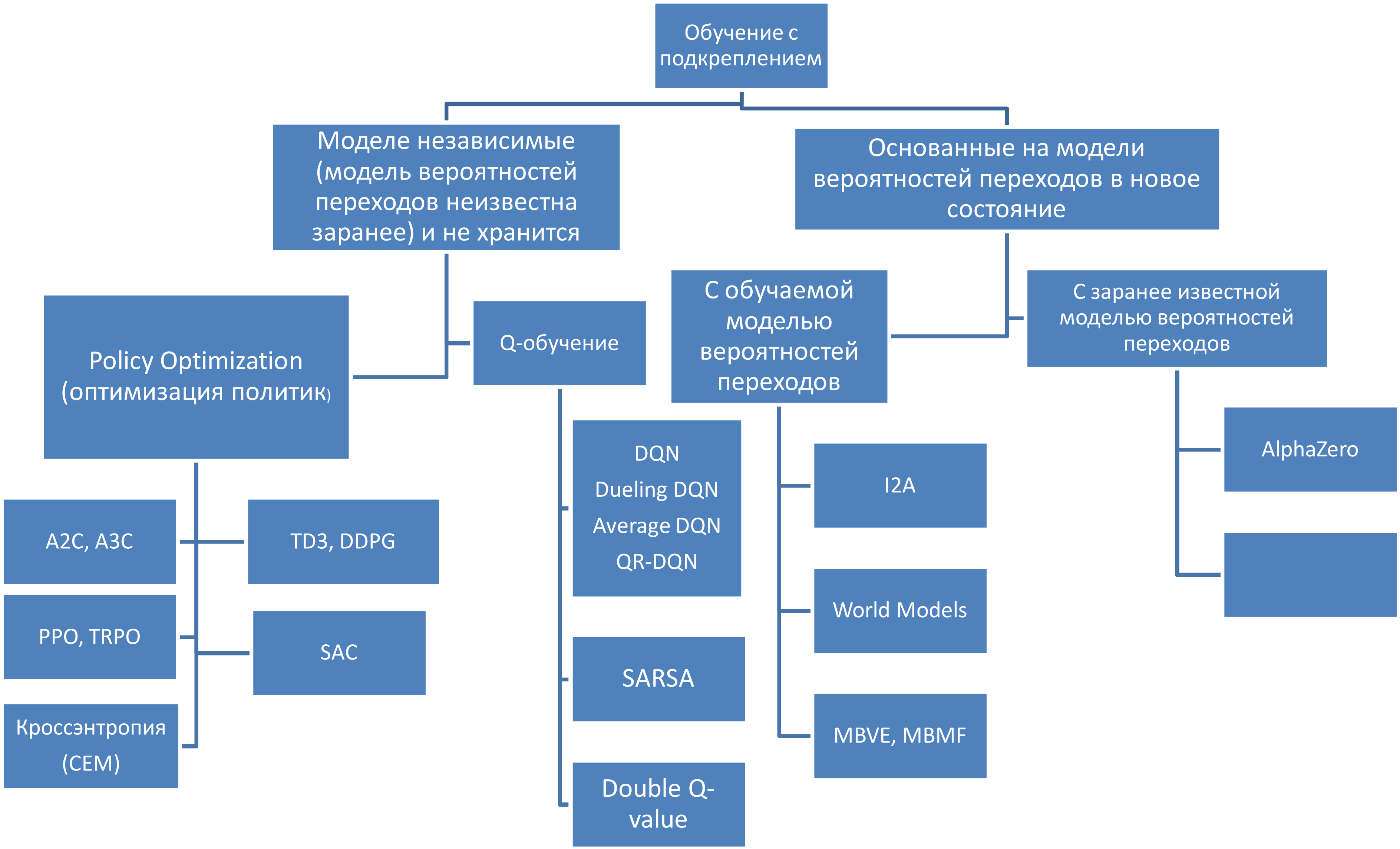
$$q(a_{t'}, s_{t'}) = E\left[\sum_{t=t'}^{\infty} \gamma^t R_t \,|\pi, s_{t'} a_{t'}\right]$$

Функции оптимальности Беллмана для ценности состояния

$$v(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a) \cdot \left(R(a,s) + \gamma \cdot v(s')\right)$$

Функции оптимальности Беллмана для ценности действия в состоянии

$$q(a,s) = \sum_{s' \in S} P(s'|s,a) \cdot \left(R(a,s) + \gamma \cdot \max_{\mathrm{a}' \in \mathrm{A}} q(a',s')\right)$$


Обучение с подкреплением
Моделе независимые (модель вероятностей переходов неизвестна заранее) и не хранится
Основанные на модели вероятностей переходов в новое состояние
Policy Optimization (оптимизация политик)
Q-обучение
С обучаемой моделью вероятностей переходов
С заранее известной моделью вероятностей переходов
A2C, A3C
TD3, DDPG
PPO, TRPO
SAC
Кроссэнтропия (CEM)
DQN
Dueling DQN
Average DQN
QR-DQN
SARSA
Double Q-value
I2A
World Models
MBVE, MBMF
AlphaZero

[2] A2C / A3C (Asynchronous Advantage Actor-Critic): Mnih et al, 2016

[3] PPO (Proximal Policy Optimization): Schulman et al, 2017

[4] TRPO (Trust Region Policy Optimization): Schulman et al, 2015

[5] DDPG (Deep Deterministic Policy Gradient): Lillicrap et al, 2015

[6] TD3 (Twin Delayed DDPG): Fujimoto et al, 2018 (*ищут оптимальную стратегию как решение уравнения оптимальности Беллмана*)

[7] SAC (Soft Actor-Critic): Haarnoja et al, 2018

[8] DQN (Deep Q-Networks): Mnih et al, 2013

[9] C51 (Categorical 51-Atom DQN): Bellemare et al, 2017

[10] QR-DQN (Quantile Regression DQN): Dabney et al, 2017

[11] HER (Hindsight Experience Replay): Andrychowicz et al, 2017

[12] World Models: Ha and Schmidhuber, 2018

[13] I2A (Imagination-Augmented Agents): Weber et al, 2017

[14] MBMF (Model-Based RL with Model-Free Fine-Tuning): Nagabandi et al, 2017

[15] MBVE (Model-Based Value Expansion): Feinberg et al, 2018

[16] AlphaZero: Silver et al, 2017

CEM (Кроссэнтропийный метод)

Инициализировать политику

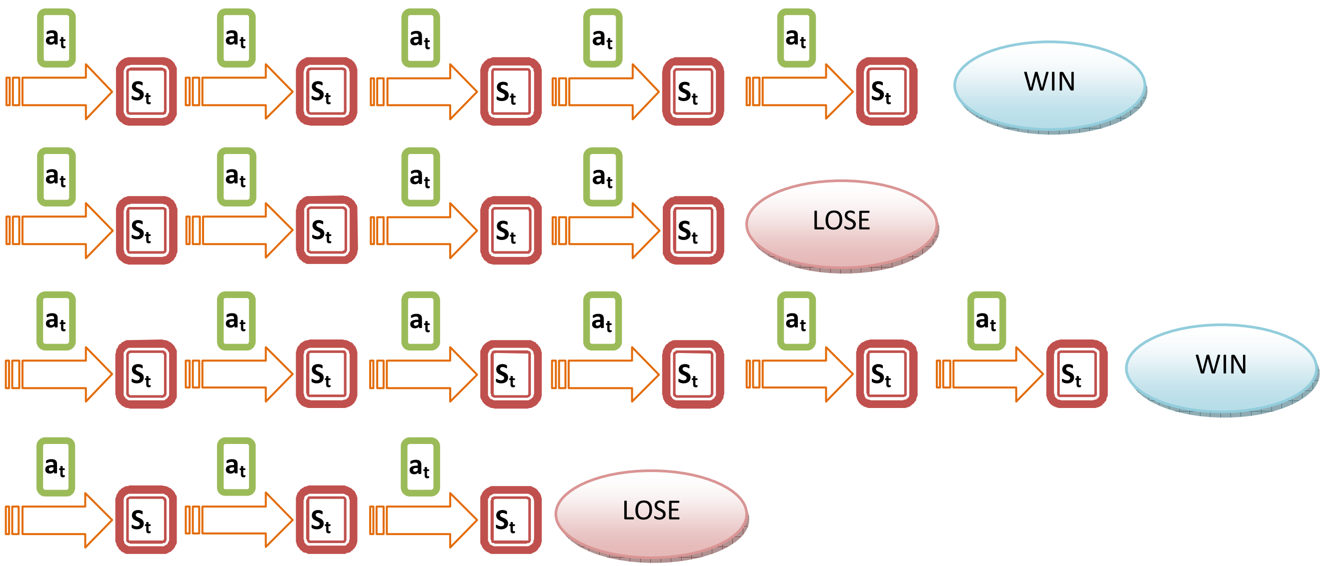
Повторить:

Сэмплировать 100 сессий

Взять 25 лучших сессий (элитных)

Изменить политику по лучшим сессиям, использовав обучение с учителем, на примерах state, actions.

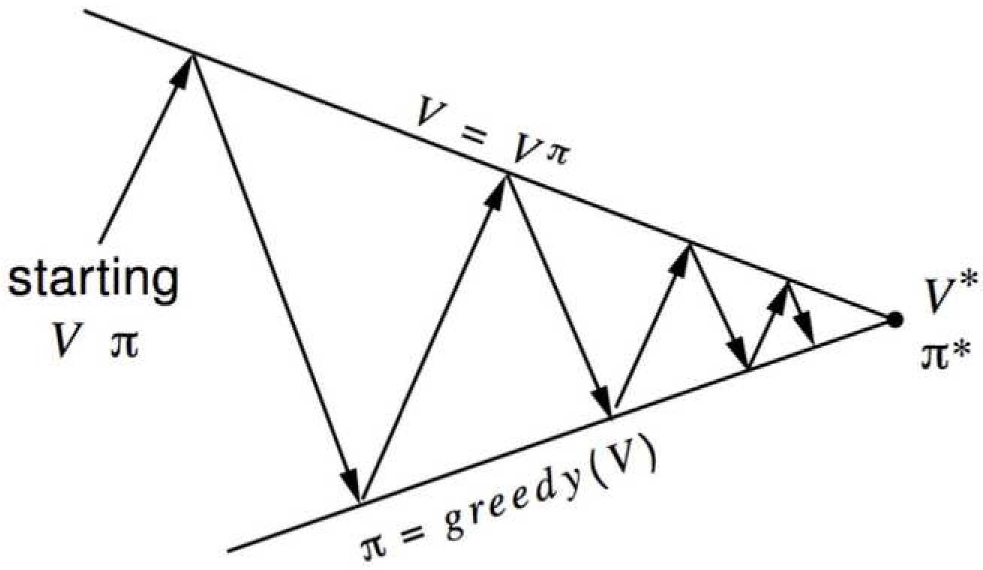
В качестве политики можно выбрать нейронную сеть с выходным слоем softmax, функцией потерь бинарная кроссэнтропия, на входе состояния, на выходе сделанное действие с вероятностью 1.



Интуитивное объяснение обучения с подкреплением методов без знания модели

Итерация политики (Policy iteration). Основанный на модели метод. Model-Based.

Итерация политики запускает цикл между оценкой и улучшением политики.

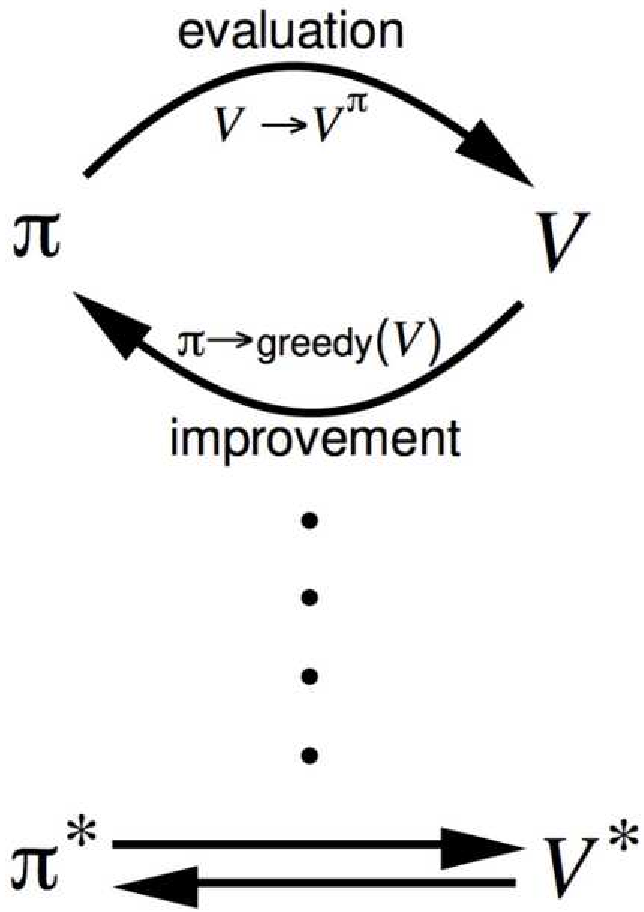


Policy evaluation Estimate v_π

Any policy evaluation algorithm

Policy improvement Generate $\pi' \geq \pi$

Any policy improvement algorithm



1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\quad \Delta \leftarrow 0$
   $\quad$ For each $s \in \mathcal{S}$:
   $\quad\quad v \leftarrow V(s)$
   $\quad\quad V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\big[r + \gamma V(s')\big]$
   $\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $\quad a \leftarrow \pi(s)$
   $\quad \pi(s) \leftarrow \operatorname{arg\,max}_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
   $\quad$ If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return V and π; else go to 2

Итерация политики

Оценка политики оценивает функцию стоимости V с помощью жадной политики, полученной в результате последнего улучшения политики. Улучшение политики, с другой стороны, обновляет политику с помощью действия, которое максимизирует V для каждого состояния. Уравнения обновления основаны на уравнении Беллмана. Итерация продолжается до сходимости.

Итерация значения (функции ценности) (Value iteration)

Значение Iteration содержит только один компонент. Он обновляет функцию значения V на основе оптимального уравнения Беллмана.

$$
\begin{aligned}
v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r | s, a) \Big[r + \gamma v_*(s') \Big]
\end{aligned}
$$

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Q-learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}^{\text{temporal difference}}$$

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q
    Take action a, observe r, s'
    Update
      $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$;
  Until s is terminal

Double Q – learning

$$Q^A_{t+1}(s_t,a_t) = Q^A_t(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_t + \gamma Q^B_t\left(s_{t+1}, \arg\max_a Q^A_t(s_{t+1},a)\right) - Q^A_t(s_t,a_t)\right), \text{ and}$$

$$Q^B_{t+1}(s_t,a_t) = Q^B_t(s_t,a_t) + \alpha_t(s_t,a_t)\left(r_t + \gamma Q^A_t\left(s_{t+1}, \arg\max_a Q^B_t(s_{t+1},a)\right) - Q^B_t(s_t,a_t)\right).$$

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

Sarsa

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

САРСА очень напоминает Q-learning. Ключевое различие между SARSA и Q-learning заключается в том, что SARSA - это алгоритм на основе политики. Это означает, что SARSA изучает значение Q на основе действия, выполняемого текущей политикой вместо жадной политики.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

DQN

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1$,T **do**

With probability ε select a random action a_t

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

Execute action a_t in emulator and observe reward r_t and image x_{t+1}

Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

```
def learn(self):
        size_block = 100
        arr = numpy.random.randint(0,self.max_size-self.nmem,size_block)
        states = []
        states_last = []
        for i in range(self.nmem):
                states.append(self.states[arr+i+1])
                states_last.append(self.states[arr+i])
        res = self.target.predict_on_batch(states)
        res_last = self.model.predict_on_batch(states_last)
        acts = self.acts[arr+1]
        rews = self.rews[arr+1]
        dones = self.dones[arr+1]
        outs = res_last.copy()
        nums = numpy.expand_dims(acts.max(axis=1),axis=1)
        mnums = acts >= nums
        numnet = numpy.expand_dims(res.max(axis=1),axis=1)
        mnumnet = res >= numnet
        try:
                outs[mnums] = rews + self.gamma*res[mnumnet]*(1-dones)
        except:
                pass
        self.model.train_on_batch(states_last, outs)
        if(self.index%5000==0):
                self.target.set_weights(self.model.get_weights())
                self.target.save_weights("out/1.file")
                print(self.index)
```