Reward

- The goal of Agent is maximize Reward
- All goals is described as maximization of expected cumulative Reward
- Cumulative Reward at step t

$$G_t = R_{t+1} + R_{t+2} + \dots$$

R_t: Reward at step t

$$G_t = \sum_{k=0}^{t} R_{t+k+1}$$

- **Problem:** Rewards that come sooner (In the begin) are more problabe to happen, since they are more predictable than long term.
- As we need to **discount** the Reward as we go further from the current step.

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

$$= \gamma^0 \cdot R_{t+1} + \gamma^1 \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} \dots$$

- As the timestep increase the future Rewards get less and less.

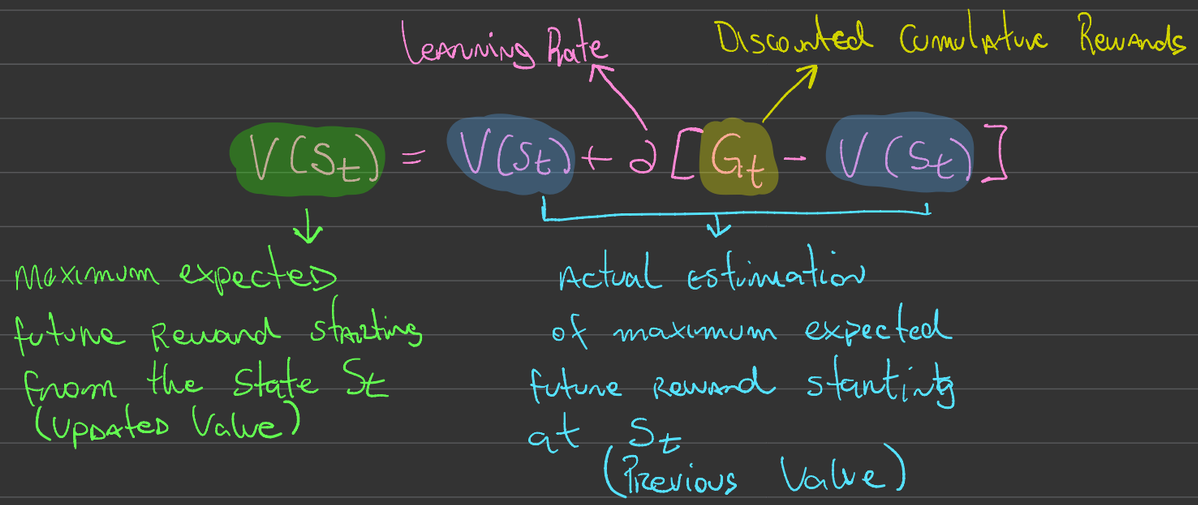
Episodic or Continuous Task

- **Episodic** have a start and a end point (terminal state)
- this create an episode
- **Example:** In Mario Bros, an episode begins at the launch of a new Mario; and end when you're killed or reach the end of the level.

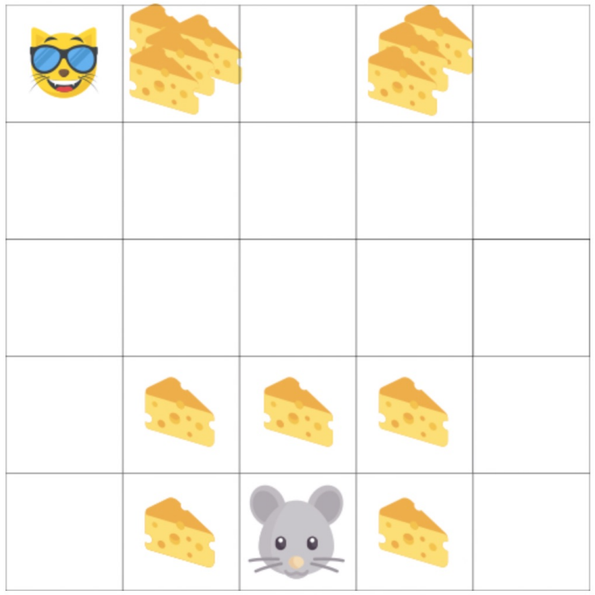
- Continuous are tasks that continues forever (no terminal state).
- In this case the Agent has to learn how to choose the best action and simultaneously interacts with the environment
- there'r not start/end point
- **Example:** Automated stock trading

Monte Carlo

- Move Carlo: collect Rewards at the end of the episode and then calculate the maximum Expected Future Reward.
- When an episode end (terminal State) the Agent looks at the total cumulative Reward to see how well it did.
- In this Approach, Rewards are only Received at end of the game
- then we start a new game with the added knowledge.



Example:

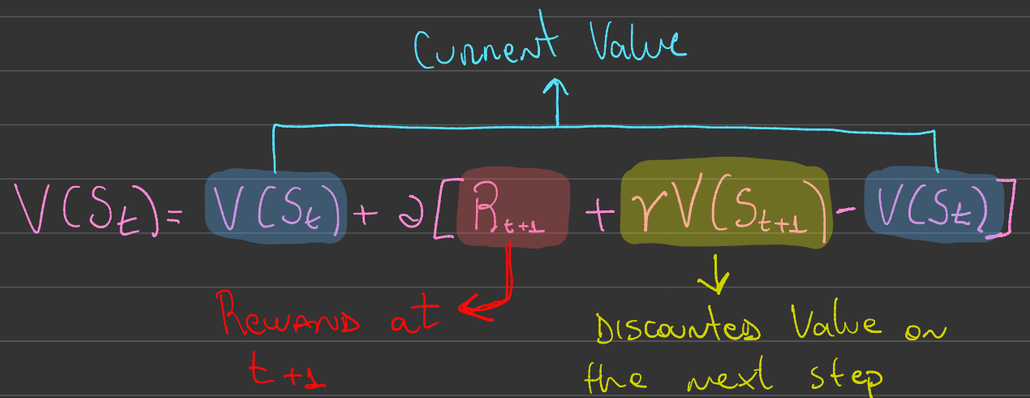


If we take the maze environment:

- We always start at the same starting point.
- We terminate the episode if the cat eats us or if we move > 20 steps.
- At the end of the episode, we have a list of State, Actions, Rewards, and New States.
- The agent will sum the total rewards Gt (to see how well it did).
- It will then update V(st) based on the formula above.
- Then start a new game with this new knowledge.

Temporal Difference TD

- Estimates the Expected Reward at each step, not waiting until the end of the episode to update the Maximum Expected Future Reward.
- It will update its value estimation V for the non-terminal state S_t occuring at the experience
- this method updates the Value Function after any individual steps.



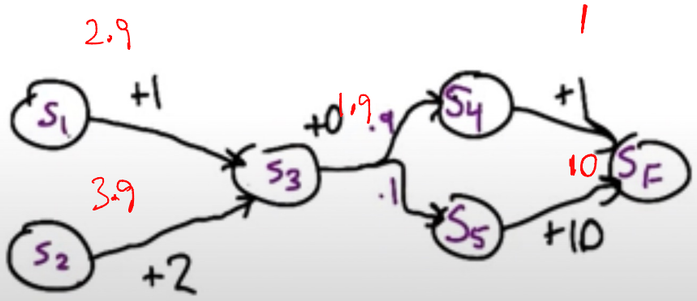
- At time t+1 they immediatly form a TD using the observed Reward R_{t+1} and the current estimated $V(S_{t+1})$

Exploration / Exploitation

- Exploration: find more information about the environment
- Exploitation: exploit know information to maximize Reward
- In the begining the Agent must explore to get knowledge of the environment.
- As times goes, the Agent start to exploit its youn knowledge to maximize the Reward.

Example

Learn $V(s) = \begin{cases} 0, & \text{if } s = S_F \\ E[r + \gamma V(s')], & \text{otherwise} \end{cases}$



- What's the value of each state? $\gamma = 1.0$

1) Start backward, as we know the value of $S_F = 0$

$V(S) = E[r + \gamma V(S)]$

$V(S_4) = E[1 + 1 . V(S_F)]$
$1 + 1.0 = \boxed{1}$

$V(S_5) = E[1 + 1 . V(S_F)]$
$V(S_5) = 10 + 0 = \boxed{10}$

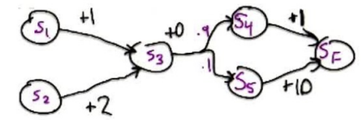
$V(S_3) = E[0 + 1 . \{ V(S_4) * 0.9 + V(S_5) * 0.1 \}]$
$0 + 1 . \{ .9 + 1.0 \} = \boxed{1.9}$

$V(S_2) = E[r + \gamma V(S_3)]$
$2 + 1 . 1.9 = \boxed{3.9}$

$V(S_1) = E[r + \gamma V(S_3)]$
$1 + 1 . 1.9 = \boxed{2.9}$

Estimating From Data

$\gamma = 1$



1. $S_1 \xrightarrow{+1} S_3 \xrightarrow{+0} S_4 \xrightarrow{+1} S_F$
2. $S_1 \xrightarrow{+1} S_3 \xrightarrow{+0} S_5 \xrightarrow{+10} S_F$
3. $S_1 \xrightarrow{+1} S_3 \xrightarrow{+0} S_4 \xrightarrow{+1} S_F$
4. $S_1 \xrightarrow{+1} S_3 \xrightarrow{+0} S_4 \xrightarrow{+1} S_F$
5. $S_2 \xrightarrow{+2} S_3 \xrightarrow{+0} S_5 \xrightarrow{+10} S_F$

QUIZ

What is an appropriate estimate for $V(S_1)$ after 3 episodes? 4?

- Now we want to calculate the $V(S_1)$ given those 5 samples

1) $S_1 \rightarrow S_3 \rightarrow S_4 \rightarrow SF$
$\boxed{2}$

2) $S_1 \rightarrow S_3 \rightarrow S_5 \rightarrow SF$
$\boxed{11}$

3) $S_1 - S_3 - S_4 - S_F$
$\boxed{2}$

4) $S_1 - S_3 - S_4 - SF$
$\boxed{2}$

$SF = 0$

$S_4 = 1 + 1.0 = \boxed{1}$
$S_3 = 0 + 1 . S_4 = \boxed{1}$
$S_1 = 1 + 1 . S_3 = 2$

$S_5 = 10 + 1 . S_F = 10$
$S_3 = 0 + 1 . S_5 = 10$
$S_1 = 1 + 1 . S_3 = 11$

$S_4 = 1 + 1 . S_F = 1$
$S_3 = 0 + 1 . S_4 = 1$
$S_1 = 1 + 1 . S_3 = 2$

Estimation = Average

- up to 3º Episode: $(2 + 11 + 2) / 3 = \boxed{5}$
- 4º Episode: $(2 + 11 + 2 + 2) / 4 = \boxed{4.25}$

Three Approachs of RL

Value Based

- The goal is to optimize the value function V(s)

- The V(s) tell us the maximum expected future Reward the Agent will get at each state

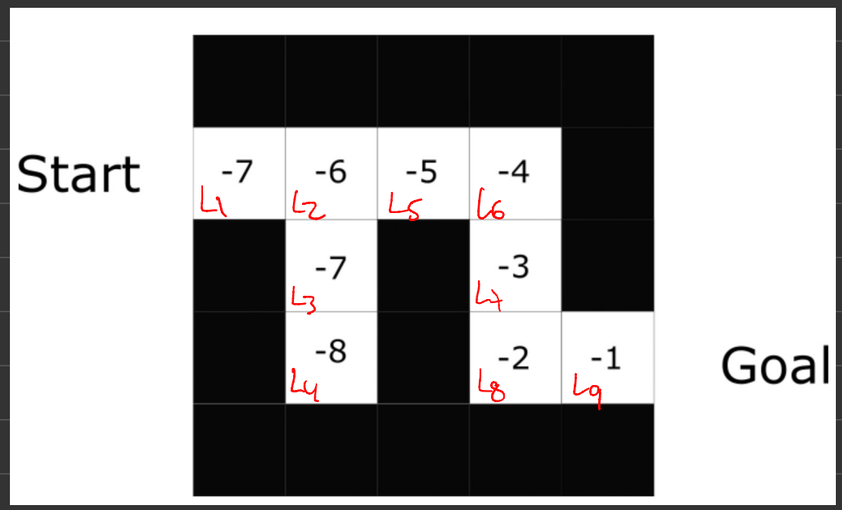
- the V(s) of each state is the total amount of Reward An agent can expect to Accumulate over the future, starting at that state.

$$V_\pi(s) = E_\pi\left[R_{t+1} + \underbrace{\gamma R_{t+2} + \gamma^2 R_{t+3} + \dots}_{\text{Discounted Reward}} \mid \underbrace{S_t = s}_{\text{given that State } s}\right]$$

expected

- The Agent will use this value function to select which state to choose at each time step. The agent takes the biggest value.

- In this example, at each time step, the Agent will take the biggest value: -7, -6, ...



L1: at this location only can choose L2
L2: check the V(s) of [L1, L3, L5]. Choose the higher

Policy Based

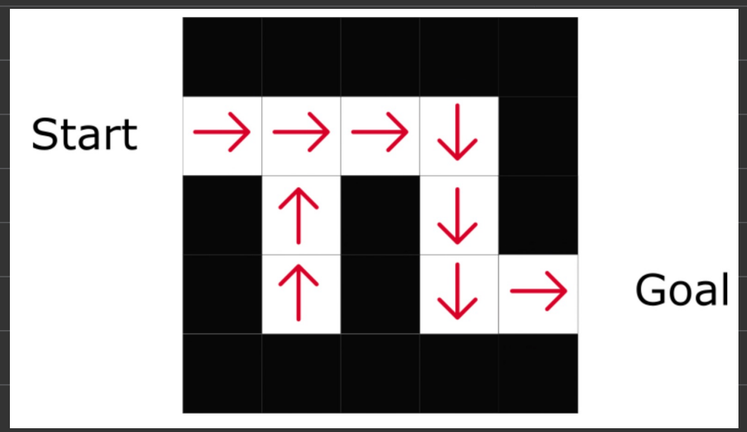
- the goal is to directly optime the policy function $\pi(s)$ without using a value function

$$a = \pi(s) \qquad \text{action} = \text{policy}(\text{state})$$

- We learn the policy. this let us map each state to the best corresponding action

- Deterministic: A policy at a given state will always Return the same Action
- Stochastic: output a prob. Distribution over Actions

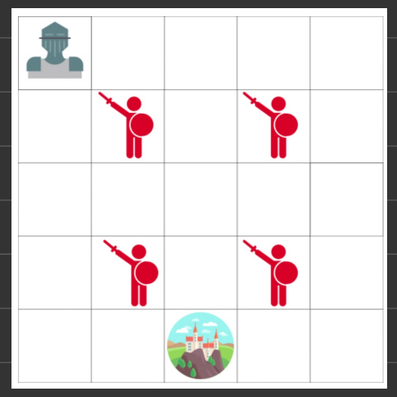
$$\pi(a|s) = P[A_t = a \mid S_t = s]$$



- In this example the policy directly indicate the best action to take for each steps.

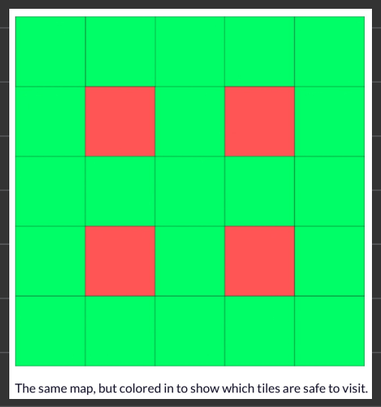
Q-LEARNING

The Problem:



- Move one tile each time
- lose -1 at each step (help to learn faster)
- If touch the enemy lose -100 and the episode finish
- If touch the castle, win 100 and episode finish

Strategy 1

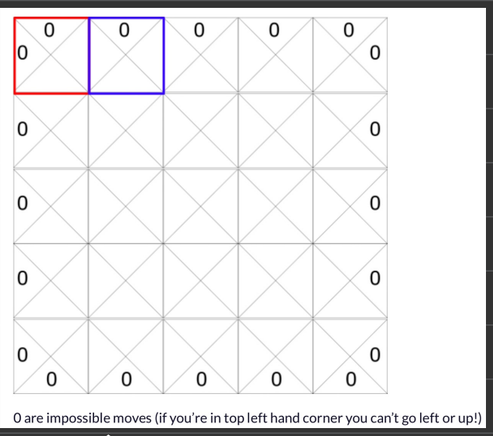


The same map, but colored in to show which tiles are safe to visit.

- Color each tile as green "safe" and Red "enemy"
- Tell agent to move only in green tiles.
- Question:
  - how to know the best path?
  - how avoid Infinite loop between adjacent green tiles?

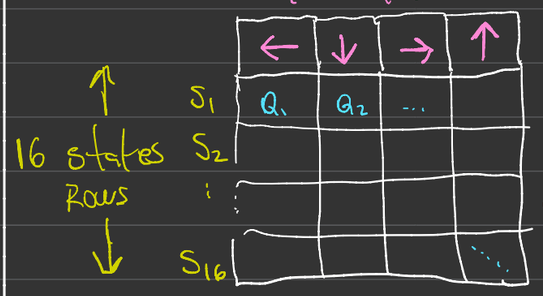
Strategy 2: Q-Table

- Create a table where we'll calculate the max Expected future Reward for each action at the state.



0 are impossible moves (if you're in top left hand corner you can't go left or up!)

- We transform this grid into a table where:
  - Rows: Represent all states
  - cols: Represent the actions
- In the castle example we have 4 actions (↑ ← ↓ →) and 16 states.



- each Q will Represent the Quality OR the Max Expected Future Reward for that given state AND action
- that means each value will be the Reward I'll get if I take that action at that state.
- Example: If I am at state S_1 AND take the action ↓ I'll receive the Q_2 as the Reward.

- How to calculate each Q-value of the table?

Using Q-learn Algorithm

↓

- The algorithm take two inputs: (state, Action)
- Return the expected future Reward of that action when you are at that state.

$$Q(S_t, a_t) = E[R_{t+1} + \gamma^1 R_{t+2} + \gamma^2 R_{t+3} \dots \mid S_t, a_t]$$

Q-Value for that state given that action | Expected discounted cumulative Reward | given that state and action

- We can see this Q-function as a table lookup where it find the row "state" and scan for the highest value in the col "Action"

- To construct the values it is necessary the Agent explore the environment. As the explore is being it start updating the values and after that it start to exploit the knowledge Acquired.

- To control the explore / exploit we use epsilon-greedy

```
if np.random.random() < epsilon:
    choose a random action  # EXPLORE
else:
    choose a action in memory  # EXPLOIT

reduce value of epsilon
```

epsilon [0, 1]

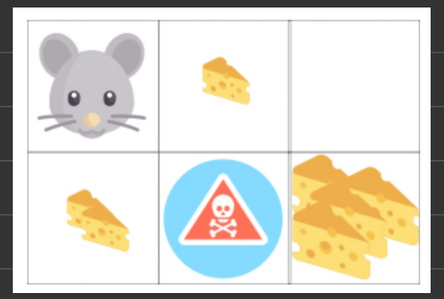
1. Initialize Q-values ($Q(s, a)$) arbitrarily for all state-action pairs.
2. For life or until learning is stopped...
3. Choose an action (a) in the current world state (s) based on current Q-value estimates ($Q(s, \cdot)$).
4. Take the action (a) and observe the the outcome state (s') and reward (r).
5. Update $Q(s, a) := Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$

Example:

- One cheese +1
- two cheese +2
- Big pile +10 (end episode)
- Rat poison -10 (end episode)

LR = 0.1

γ = 0.9



	←	→	↑	↓
Start	0	0	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

We move at random (for instance, right)

1. We are in start state and take a Random action →
2. We found a piece of cheese, so Reward +1
3. Update the Q-Value of being at start and doing action →.

Q(start, →)

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma max\, Q'(s',a') - Q(s,a)]$$

New Q value for that state and that action | Current Q value | Learning Rate | Reward for taking that action at that state | Discount rate | Maximum expected future reward given the new s' and all possible actions at that new state

0 | 0.1 | 1.0 | 0.9 | 0 | 0

$$NewQ(start, \rightarrow) = Q(start, \rightarrow) + \alpha\,[\Delta Q(start, \rightarrow)]$$

$$\Delta Q(start, \rightarrow) = R(start, \rightarrow) + \gamma \max Q'(1cheese, a') - Q(start, \rightarrow)$$

$$= 1.0 + 0.9 * \max[Q'(1cheese, \leftarrow), Q'(1cheese, \downarrow), Q'(1cheese, \rightarrow)] - Q(start, \rightarrow)$$

$$= 1.0 + 0.9 \times 0.0 - 0 = \boxed{1}$$

New Q(start, →) = 0 + 0.1 * 1.0 = 0.1

	←	→	↑	↓
Start	0	0.1	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

The updated Q-table

```python
def decrement_epsilon(self):
    self.epsilon = self.epsilon * self.eps_dec if self.epsilon > self.eps_end else self.eps_end

def learn(self, state, action, reward, state_):
    # get all possible actions given the new state_ the agent is now
    actions = np.array([self.Q[(state_, a)] for a in range(self.n_actions)])  # all actions given t

    # select the action that get the highest value
    a_max = np.argmax(actions)

    # calculate the discounted reward
    discounted_reward = reward + self.gamma * self.Q[(state_, a_max)] - self.Q[(state, action)]

    # update the previous state value with the new value
    self.Q[(state, action)] += self.lr * discounted_reward

    self.decrement_epsilon()
```

A recap...

- Q-learning is a value-based Reinforcement Learning algorithm that is used to find the optimal action-selection policy using a q function.
- It evaluates which action to take based on an action-value function that determines the value of being in a certain state and taking a certain action at that state.
- Goal: maximize the value function Q (expected future reward given a state and action).
- Q table helps us to find the best action for each state.
- To maximize the expected reward by selecting the best of all possible actions.
- The Q come from quality of a certain action in a certain state.
- Function Q(state, action) → returns expected future reward of that action at that state.
- This function can be estimated using Q-learning, which iteratively updates Q(s,a) using the Bellman Equation
- Before we explore the environment: Q table gives the same arbitrary fixed value → but as we explore the environment → Q gives us a better and better approximation.

Recap: (Tabular) Q-Learning

Algorithm:

Start with $Q_0(s, a)$ for all s, a.

Get initial state s

For k = 1, 2, … till convergence

Sample action a, get next state s'

If s' is terminal:

$$\text{target} = R(s, a, s')$$

Sample new initial state s'

else:

$$\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha) Q_k(s, a) + \alpha [\text{target}]$$

$$s \leftarrow s'$$

Deep Q-Learn

- High level IDEA: make Q-learning look like supervised learning
- Replace the Q-table by a function APROXIMATION that produces the Q-value
- This enable to interact with env. that has many states, that would be impossible to store in a table format
- Q-learning: take the current state AND do a table lookup to (ROWS) AND get the max value (COL) that MAXI the REWARD. This max will be the action the agent must PERFORM.
- DQN: Instead a table, take the current state AND use a Neural Net (function APROXIMATION) to predict what would be the next best ACTION to take. The table is Replaced by the Neural Net weights.

Problems

1) What would be the target value so that we can calculate loss AND backprop to upate weights?
2) How avoid correlation, that is, if the agent keep going to one direction, the network will "learn" to do just this ACTION

Recap: Approximate Q-Learning

Algorithm:

Start with $Q_0(s, a)$ for all s, a.

Get initial state s

For k = 1, 2, ... till convergence

Sample action a, get next state s'

If s' is terminal:

$\text{target} = R(s, a, s')$

Sample new initial state s'

else:

$\text{target} = R(s, a, s') + \gamma \max_{a'} Q_k(s', a')$

$\theta_{k+1} \leftarrow \theta_k - \alpha \nabla_\theta \mathbb{E}_{s' \sim P(s'|s,a)} \left[(Q_\theta(s, a) - \text{target}(s'))^2 \right] |_{\theta=\theta_k}$

$s \leftarrow s'$

Chasing a nonstationary target!

Updates are correlated within a trajectory!

- As the same network is producing the predictions AND targets, there will be to much instability, AS IN A supervised learning the targets ARE fixed given the same inputs.
- In RL, the targets ARE being updated all time AS the agent keep exploring the environment. This end up as the N.N chasing its own.
- To solve, this, use two identical N.N.
  1º used to predict AND being updated the weights
  2º the uses to produce the targets, so that the targets ARE A little fixed. This network do not backprop.
- After some timesteps C, copy the weights from 1 to 2, so that the targets get better through time AND the predictions of 1 get better.

- Use a replay memory with a capacity N to store

$(state_t, action_t, Reward_t, state_{t+1})$

the agent is in state S_0, perform the action A_0, get Reward R_0 and move to new state S_1

DQN Training Algorithm

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 $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

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

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

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

End For

End For