# Application of Reinforcement Learning to Unsupervised Control of Complex Systems

Google DeepMind's AI beats world's best Go player (May 2017)

Courtesy: newscientist.com

AlphaZero annihilated AlphaGo 100-0.

AlphaGo was trained on games played by humans, whereas AlphaZero just taught itself how to play.

# Lecture Materials

- https://github.com/earlaleluya/RL-CrashCourse

# Learning Outcomes

By the end of this lecture,

1. You can describe the core concepts of reinforcement learning (agent, environment, action, reward, state, and policy).
2. You can apply reinforcement learning on a simple environment (i.e. cart pole balancing problem)
3. You can adapt reinforcement learning to your research problems by defining states, actions, and rewards.

# Outline

- Paradigms of Machine Learning (Supervised, Unsupervised, and Reinforcement Learning)
- Core Concepts (agent, environment, action, reward, state, and policy)
- Cart Pole Balancing Simulation
  - Exploration and Exploitation
  - Q-learning algorithm
  - SARSA algorithm
- Open Forum

# Supervised Learning

Hello baby! This is an apple.

Hi again baby! This is also an apple.

After several days ...

What is this fruit?

"Uses **labeled data** to make predictions"

# Unsupervised Learning

Hey baby! Can you sort these Lego blocks?

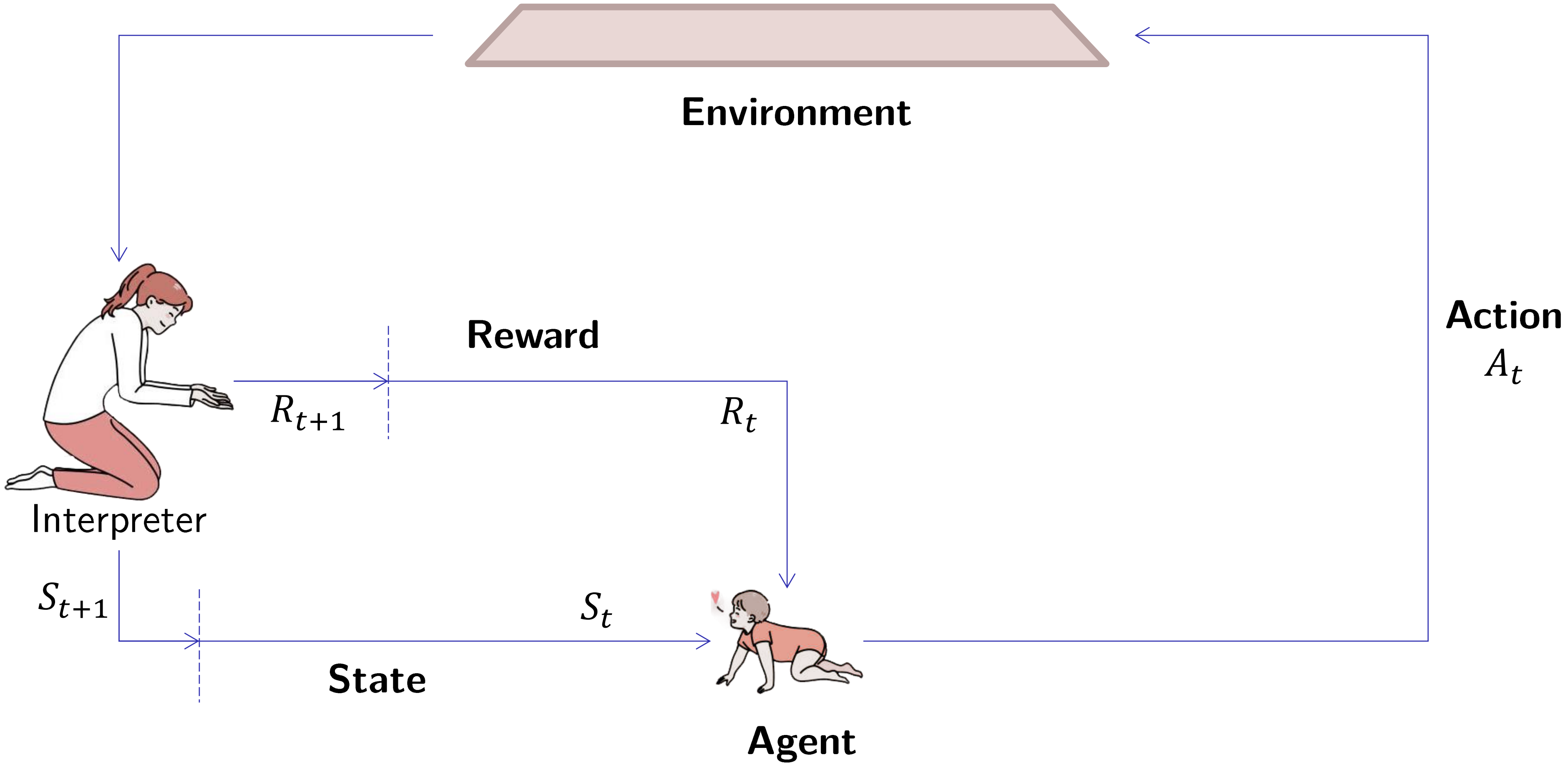
"Finds hidden patterns and structures in **unlabeled data**"

# Reinforcement Learning

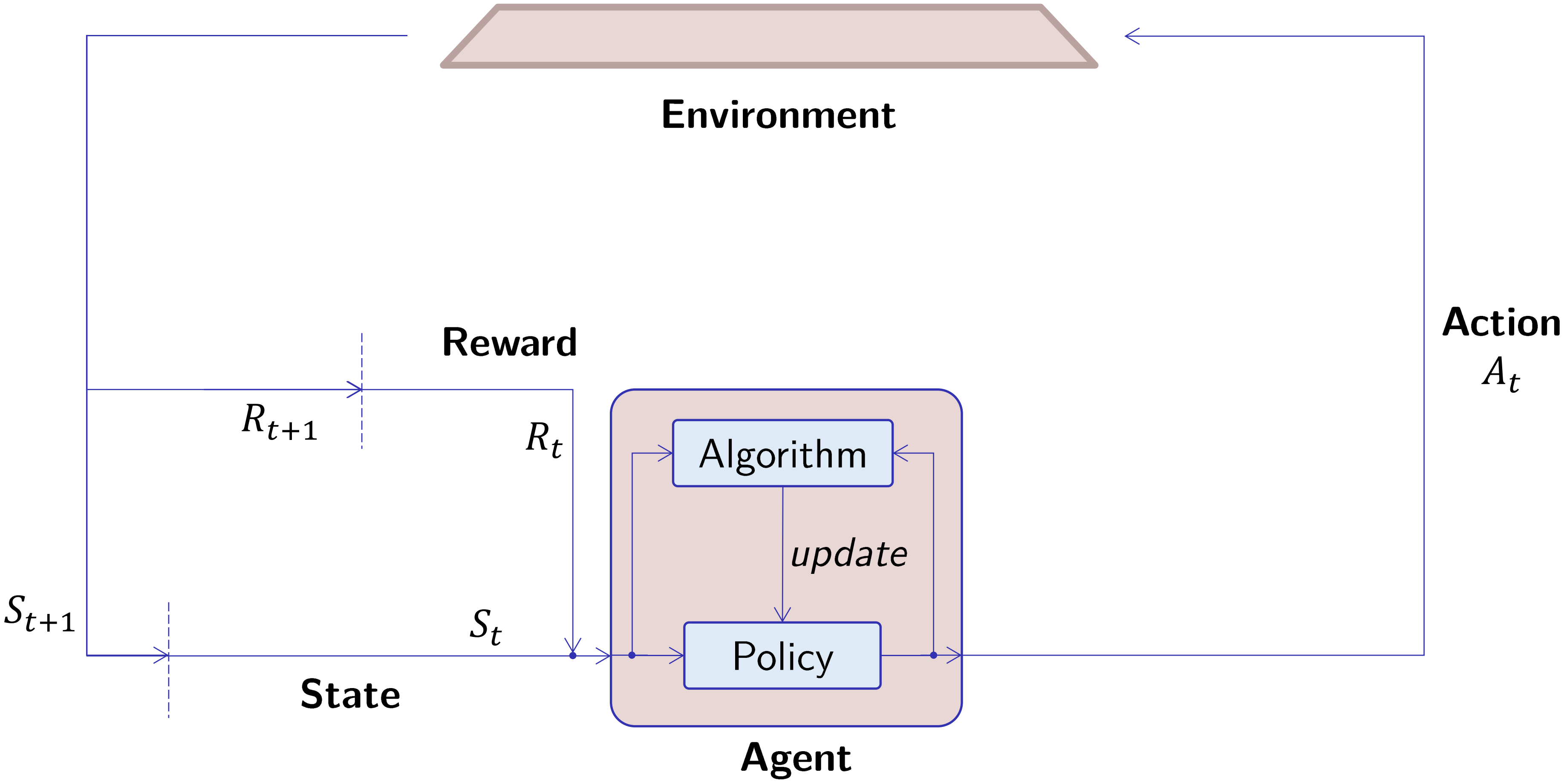


"An agent learns through **trial and error** to make decisions in a dynamic environment to **maximize long-term rewards**."

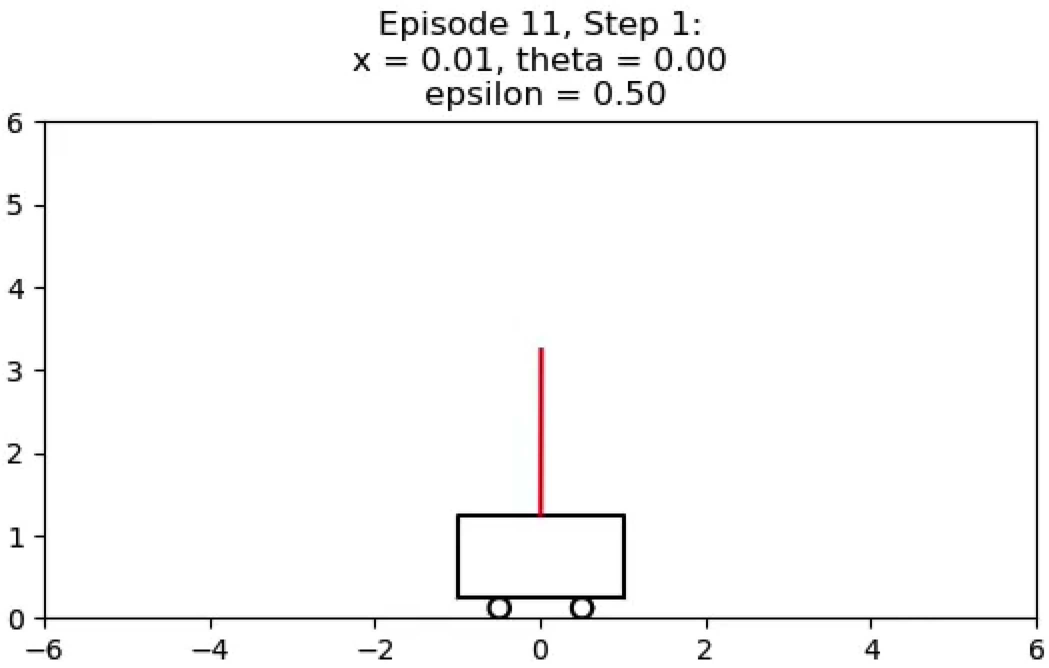
# Core Concept



# Core Concept



# Cart Pole Balancing Simulation




Discussion Focus:

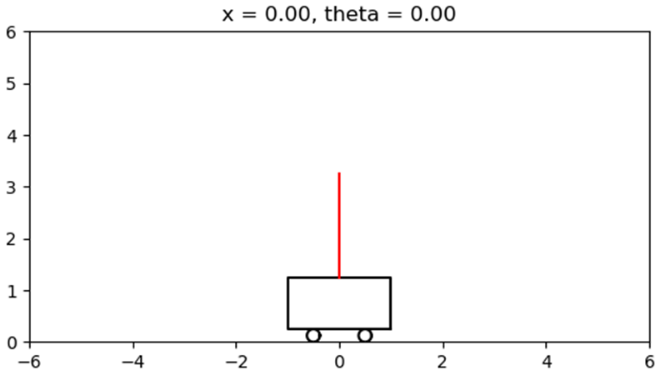
1. More emphasis on how the algorithms (i.e., Q-learning and SARSA) work, rather than on preparing the simulation's environment and codebase.
2. More discussion on how each component functions in the RL workflow, rather than on emphasizing the practicality and sophistication; (to appreciate RL from a simple example)

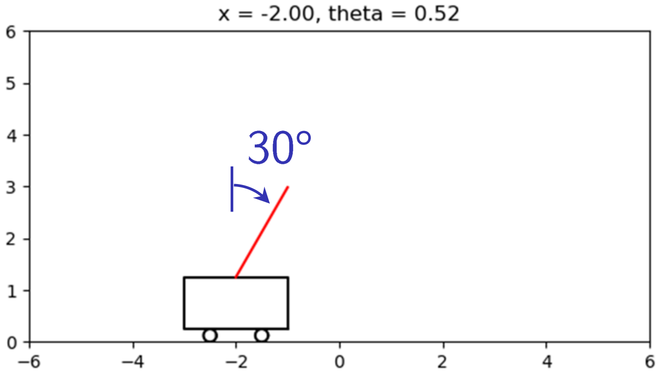
# Environment

Required Parameters:

$x$: cart position in meters

$\theta$: pole angle in radians




examples\python\environment.py

```python
class Environment:

    def draw_plot(self, x, theta, pole_length=2, episode=0, step=0, epsilon=0):
        self.ax.clear()

        # Ground robot body
        pxg = [x+1, x-1, x-1, x+1, x+1]
        pyg = [0.25, 0.25, 1.25, 1.25, 0.25]

        # Ground Robot wheels
        pxw1, pyw1 = self.plot_circle(x-0.5, 0.125, 0.125, 0.125)
        pxw2, pyw2 = self.plot_circle(x+0.5, 0.125, 0.125, 0.125)

        # Pole
        pxp = [x, x + pole_length * np.sin(theta)]
        pyp = [1.25, 1.25 + pole_length * np.cos(theta)]

        # Plot
        self.ax.plot(pxg, pyg, 'k-')        # ground robot body
        self.ax.plot(pxw1, pyw1, 'k')       # circle 1
        self.ax.plot(pxw2, pyw2, 'k')       # circle 2
        self.ax.plot(pxp, pyp, 'r-')        # pole

        # Display x and theta values
        self.ax.set_title(f"Episode {episode}, Step {step}:\nx = {x:.2f}, theta = {theta:.2f}\n epsilon = {epsilon:.2f}")
        self.ax.axis([-6, 6, 0, 6])
        self.ax.set_aspect('equal', adjustable='box')  # keep proportions
        self.fig.canvas.draw()
        plt.pause(0.001)
```

Task 1: Try replacing the value of x and theta

```python
env.draw_plot(x=-2, theta=0.52)
```

then,

```python
env.draw_plot(x=2, theta=-0.52)
```

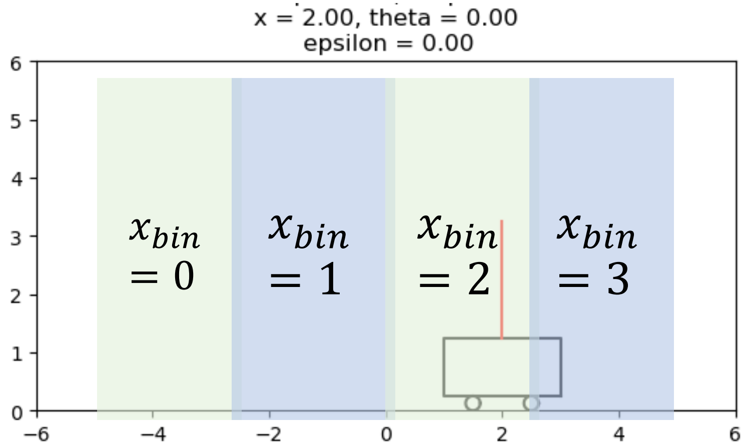
# State

$x$: cart position in meters
$\dot{x}$: cart velocity in meters per $\tau$ second
$\theta$: pole angle in radians
$\dot{\theta}$: pole angular velocity in radians per $\tau$ second

State value (integer),

$$|s| = x_{bin} \cdot n_{bins}^3 + \dot{x}_{bin} \cdot n_{bins}^2 + \theta_{bin} \cdot n_{bins} + \dot{\theta}_{bins}$$

Example, assume $n_{bins} = 4$ and $x_{bounds} = [-5, 5]$




**examples\python\state.py**

```python
class State:
    x = 0       # cart position in meters
    x_dot = 0   # cart velocity
    theta = 0   # pole angle in radians
    theta_dot = 0 # pole angular velocity per 'tau' second

    def __init__(self, n_states, noise=0.01):
        self.n_states = n_states
        self.noise = noise
        self.reset()
        self.bounds = {
            'x': [-5.0, 5.0],
            'x_dot': [-1.0, 1.0],
            'theta': [np.deg2rad(-12), np.deg2rad(12)],
            'theta_dot': [-1.0, 1.0]    # 1 rad/tau = 57.3 deg/tau
        }
        self.n_bins = int(round(self.n_states ** 0.25))  # 4th root of n_states

    def compute_state_value(self):
        if self.is_fail():
            return -1
        x_bin = self.discretize(self.x, 'x')
        x_dot_bin = self.discretize(self.x_dot, 'x_dot')
        theta_bin = self.discretize(self.theta, 'theta')
        theta_dot_bin = self.discretize(self.theta_dot, 'theta_dot')
        return (x_bin * self.n_bins**3) + (x_dot_bin * self.n_bins**2) + (theta_bin * self.n_bins) +
theta_dot_bin


    def is_fail(self):
        robot_exceeds_left = (self.x < self.bounds['x'][0])
        robot_exceeds_right = (self.x > self.bounds['x'][1])
        pole_falls_at_left = (self.theta < self.bounds['theta'][0])
        pole_falls_at_right = (self.theta > self.bounds['theta'][1])
        return (robot_exceeds_left or robot_exceeds_right or pole_falls_at_left or pole_falls_at_right)
```

Task 2: Try replacing the values of any of the state parameters (x, x_dot, theta, theta_dot)

Example:

```python
state.x = 0
state.x_dot = 0
state.theta = 0
state.theta_dot = 0
```

Question: What do we represent the state with a singular scalar integer?

# Reward

**examples\python\reward.py**

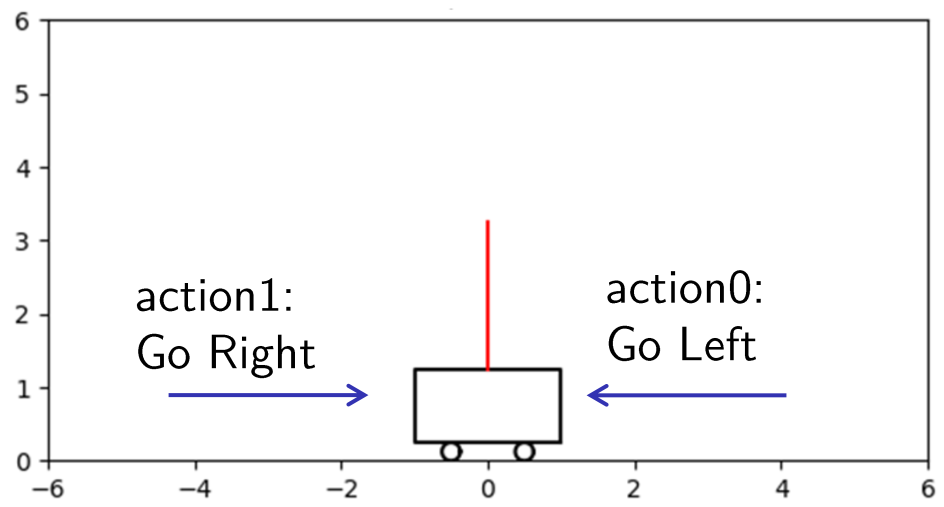
```
class Reward:

    def compute_reward(self, state_value): # R_{t+1}
        if state_value > 0:    # no fail
            reward = 1.0
        else:    # fail
            reward = 0.0
        return reward
```

Task 3: Try replacing the state parameter values. Note: `1.0` means no-fail, while `0.0` means fail.

Example:
`state.x = 5.1`
`state.x_dot = 0`
`state.theta = 0`
`state.theta_dot = 0`

Question: What if we modify the reward system where `0.0` for every time step the pole remains balanced, while `-1.0` when the pole falls?

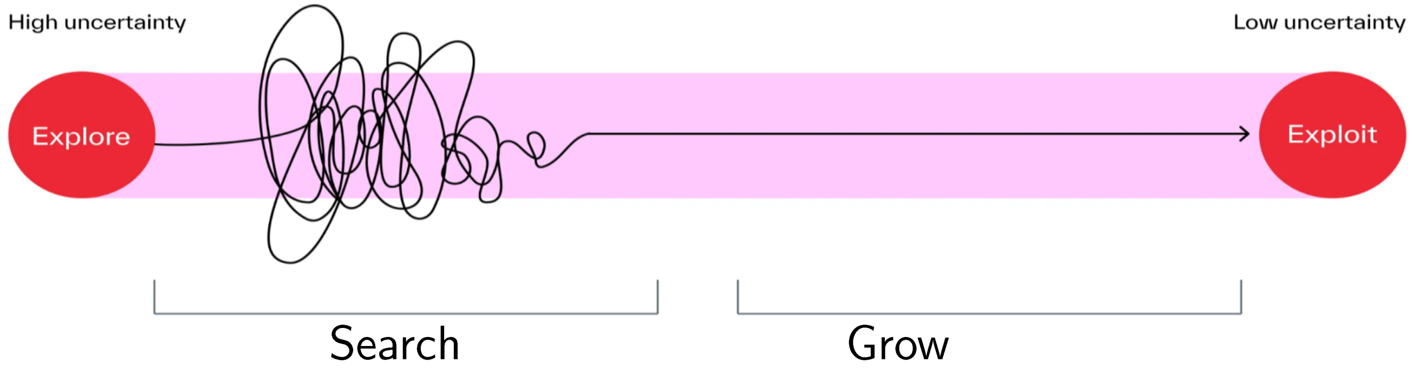
# Action




examples\python\action.py

```
class Action:
    def __init__(self):
        self.actions = ['left', 'right']

    def decide(self, epsilon, agent, state):
        if np.random.rand() < epsilon:  # implements epsilon-greedy
            self.action = self.explore()
        else:
            self.action = self.exploit(agent, state)
        return self.action


    def explore(self):
        return random.choice(self.actions)

    def exploit(self, agent, state):
        return self.actions[agent.best_action_idx(state)]
```

**What is exploration and exploitation tradeoff?**



# Agent (Q-learning algorithm)

examples\python\q.py

```
class Q:

    def __init__(self, n_states, n_actions, alpha, gamma, data_path=None, save_path=None):
        self.Q = np.zeros((n_states, n_actions)) if data_path is None else self.load_csv(data_path)
        self.alpha = alpha
        self.gamma = gamma

    def update(self, state, action, next_state, next_reward):
        if next_state < 0:    # fail
            self.Q[state,action] = ((1-self.alpha)*(self.Q[state,action])) + (self.alpha*(next_reward+
(self.gamma*0)))
        else:   # no fail
            self.Q[state,action] = ((1-self.alpha)*(self.Q[state,action])) + (self.alpha*(next_reward+
(self.gamma*np.max(self.Q[next_state]))))
```

$$Q^{new}(s_t, a_t) = \underbrace{(1-\alpha)Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha\left[r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a)\right]}_{\text{learned value}}$$

Where,

- $\alpha$ is the learning rate
- $r_{t+1}$ is the reward
- $\gamma$ is the discount rate
- $\max_a Q(s_{t+1}, a)$ is the maximum expected future reward

# Agent (SARSA algorithm)

examples\python\sarsa.py

```python
class SARSA:

    def update(self, state, action, next_state, next_action, next_reward):
        if next_state < 0:     # fail
            target = next_reward  # No future reward if failed
        else:    # no fail
            target = next_reward + self.gamma * self.Q[next_state, next_action]
        self.Q[state, action] = (1 - self.alpha) * self.Q[state, action] + self.alpha * target
```

$$Q^{new}(s_t, a_t) = \underbrace{(1-\alpha)Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha[\,r_{t+1} + \gamma \cdot Q(s_{t+1}, a_{t+1})]}_{\text{learned value}}$$

Where,

- $\alpha$ is the learning rate
- $r_{t+1}$ is the reward
- $\gamma$ is the discount rate

# Q-learning algorithm

This is the big picture! To generate an optimal Q table.

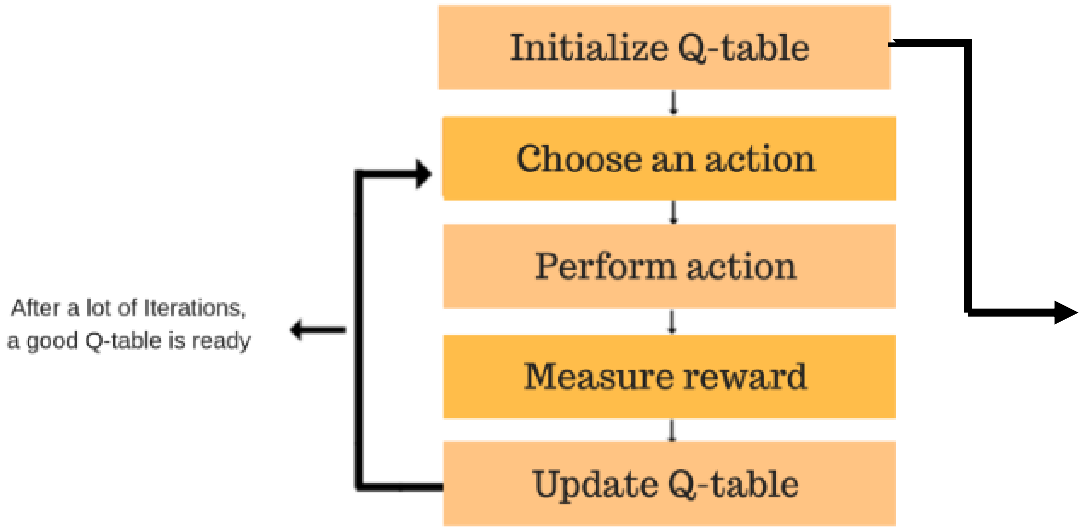
| | action0 | action1 |
|---|---|---|
| state0 | 0 | 0 |
| state1 | 0 | 0 |
| state2 | 0 | 0 |
| state3 | 0 | 0 |
| state4 | 0 | 0 |
| ⋮ | 0 | 0 |
| | 0 | 0 |
| state255 | 0 | 0 |

Initial Q table

| | action0 | action1 |
|---|---|---|
| state0 | 0.501232 | 0.766755 |
| state1 | 0.510511 | 0.0502303 |
| state2 | 0 | 0.000648321 |
| state3 | 0.513589 | 0.846939 |
| state4 | 0.0930823 | 0.795683 |
| ⋮ | 0.311592 | 0.545304 |
| | 0.648021 | 0.982138 |
| state255 | 0.568814 | 0.874563 |

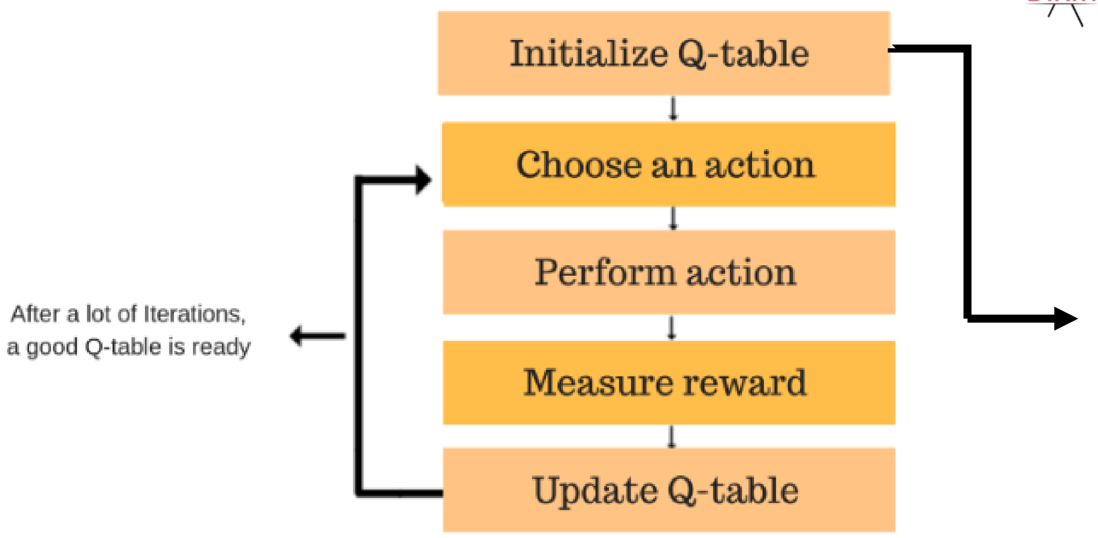
Optimal Q table

# Q-learning algorithm process




| | Left | Right |
|---|---|---|
| state0 | 0 | 0 |
| state1 | 0 | 0 |
| state2 | 0 | 0 |
| state3 | 0 | 0 |
| state4 | 0 | 0 |
| state5 | 0 | 0 |
| state6 | 0 | 0 |
| state7 | 0 | 0 |

# Q-learning algorithm process

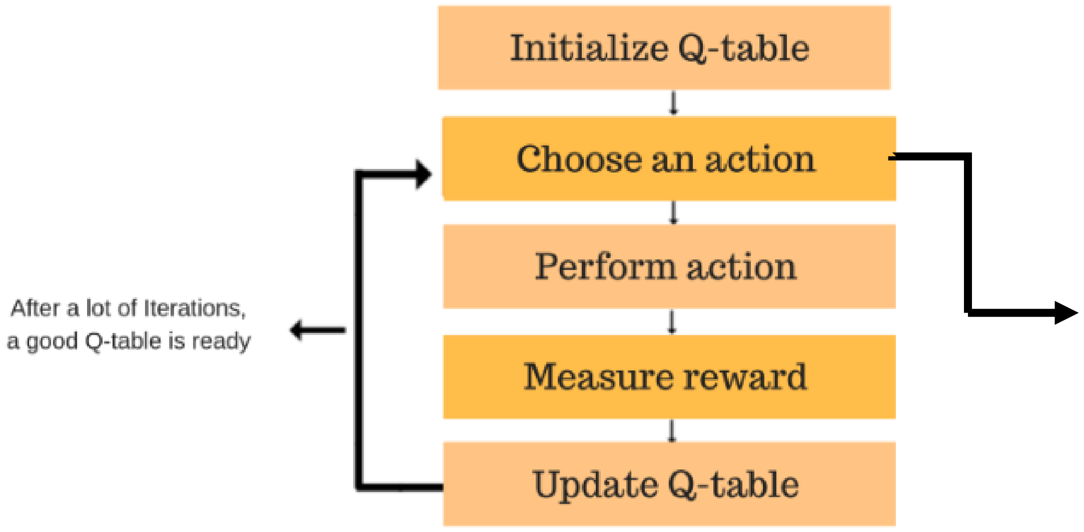



START

| | Left | Right |
|---|---|---|
| state0 | 0 | 0 |
| state1 | 0 | 0 |
| state2 | 0 | 0 |
| state3 | 0 | 0 |
| state4 | 0 | 0 |
| state5 | 0 | 0 |
| state6 | 0 | 0 |
| state7 | 0 | 0 |

Initially, you choose an action randomly.

# Q-learning algorithm process

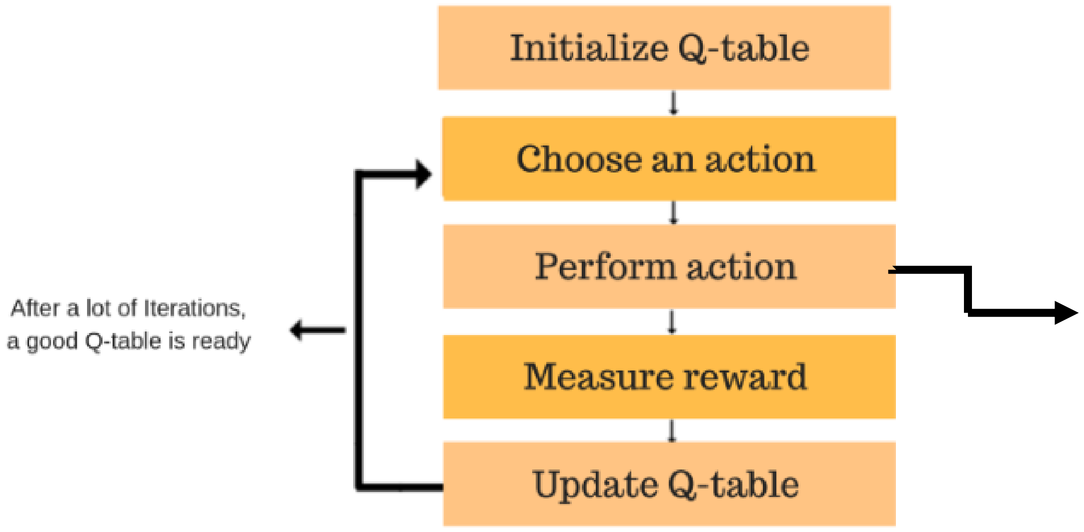


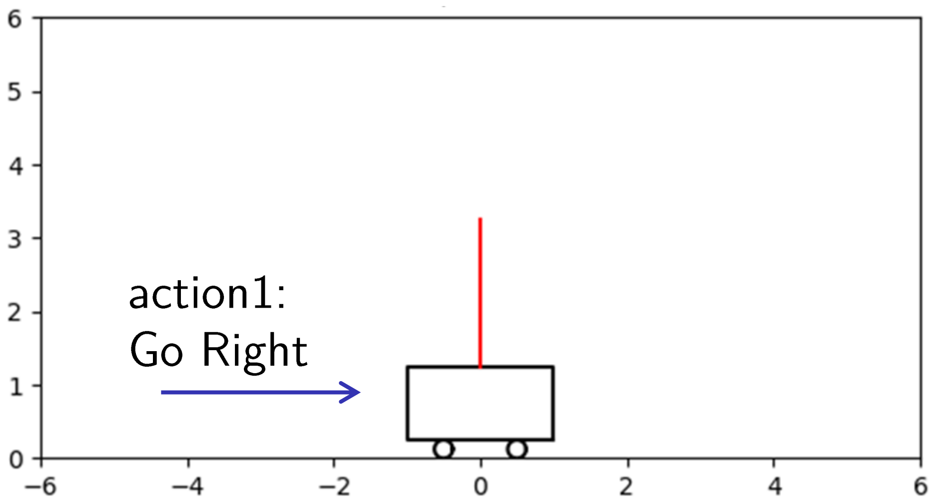

| | Left | Right |
|---|---|---|
| state0 | 0 | 0 |
| state1 | 0 | 0 |
| state2 | 0 | 0 |
| state3 | 0 | 0 |
| state4 | 0 | 0 |
| state5 | 0 | 0 |
| state6 | 0 | 0 |
| state7 | 0 | 0 |

At some time $t$

Question: At some time $t$, how will you choose an action?

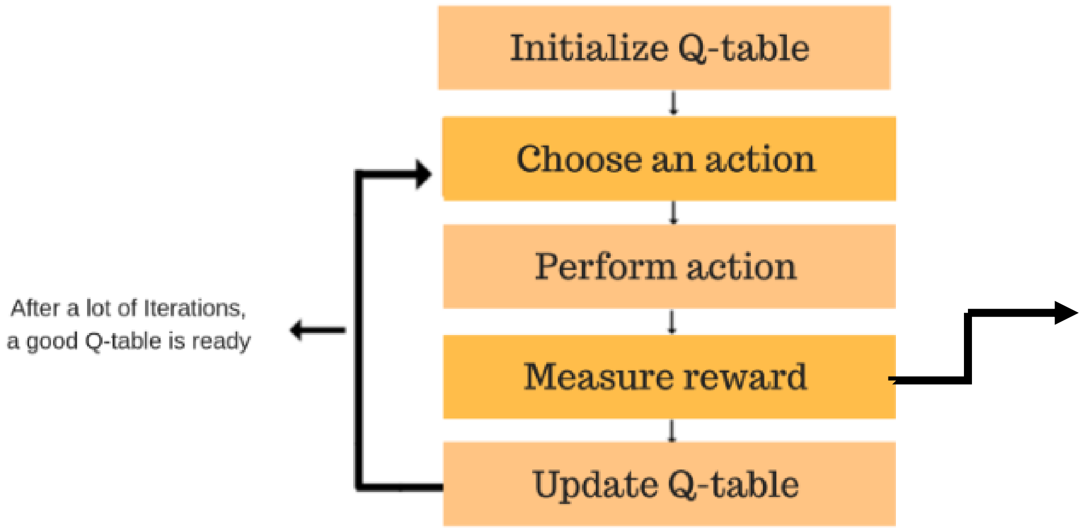
# Q-learning algorithm process




Apply force F towards what direction?

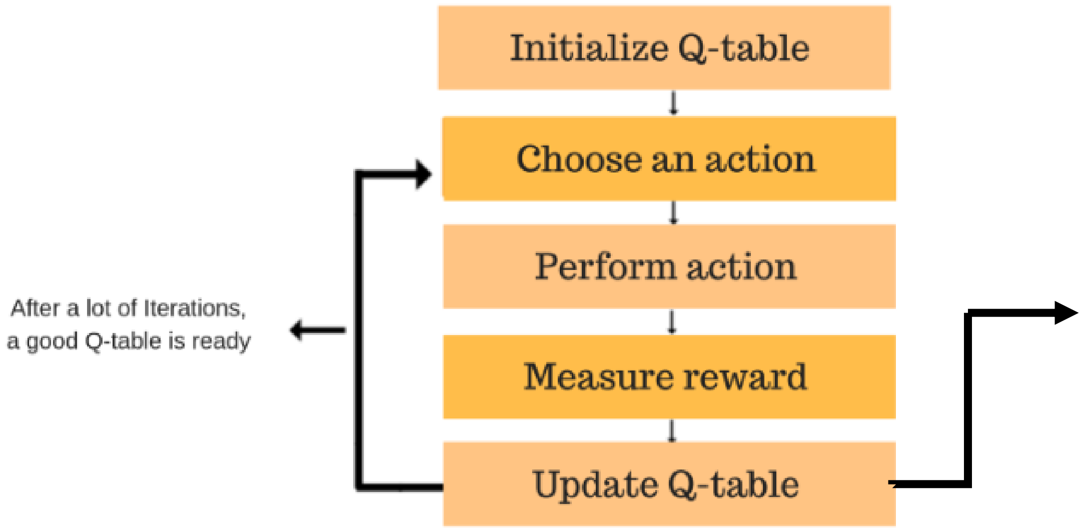
# Q-learning algorithm process




$r_{t+1} = 1.0$ , if pole stays balanced
$r_{t+1} = 0.0$ , if pole falls

How will you impose a reward system?

# Q-learning algorithm process

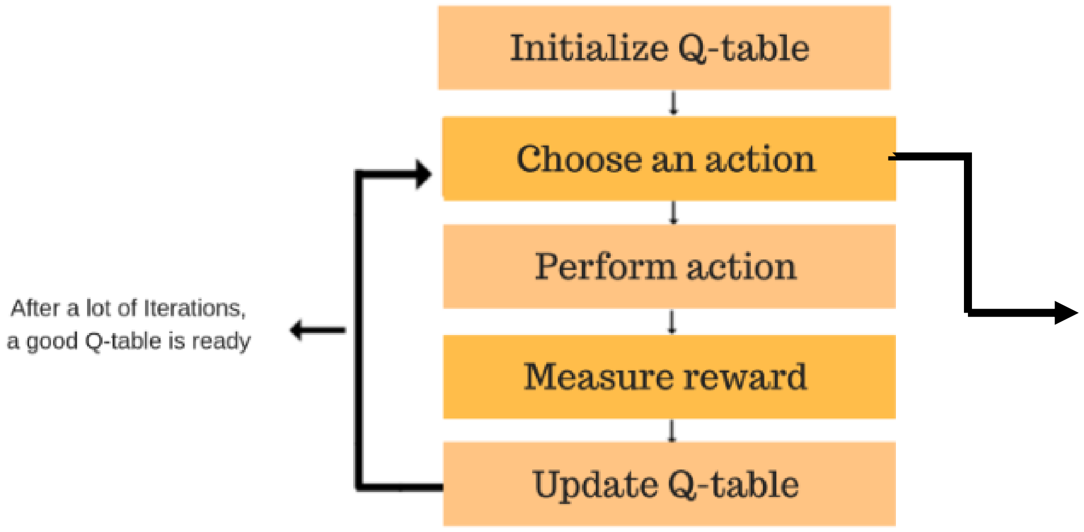



| | Left | Right |
|---|---|---|
| state0 | 0 | 0 |
| state1 | 0 | 0 |
| state2 | 0 | 0.846939 |
| state3 | 0 | 0 |
| state4 | 0 | 0 |
| state5 | 0 | 0 |
| state6 | 0 | 0 |
| state7 | 0 | 0 |

$Q(s_t, a_t)$ is updated per step.

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

# Q-learning algorithm process

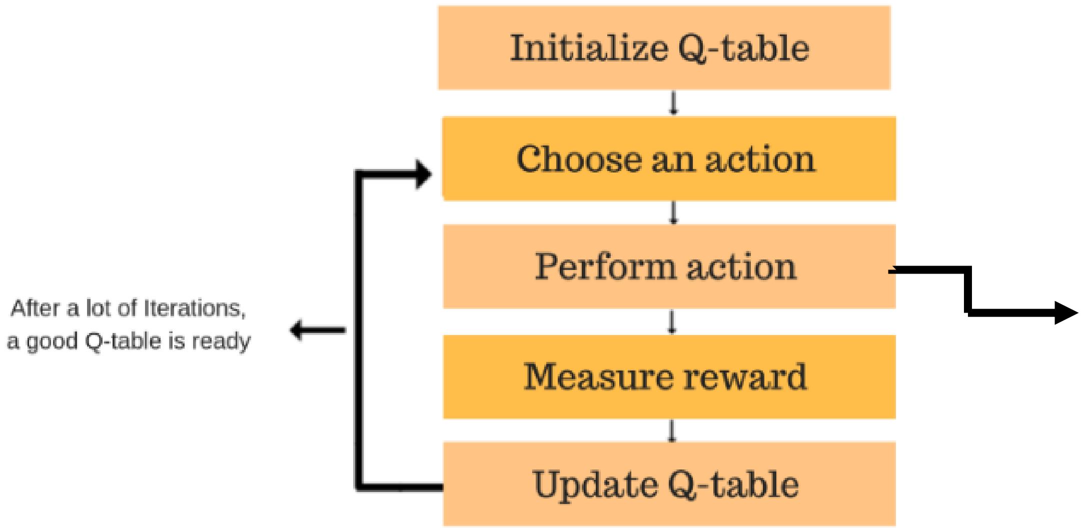


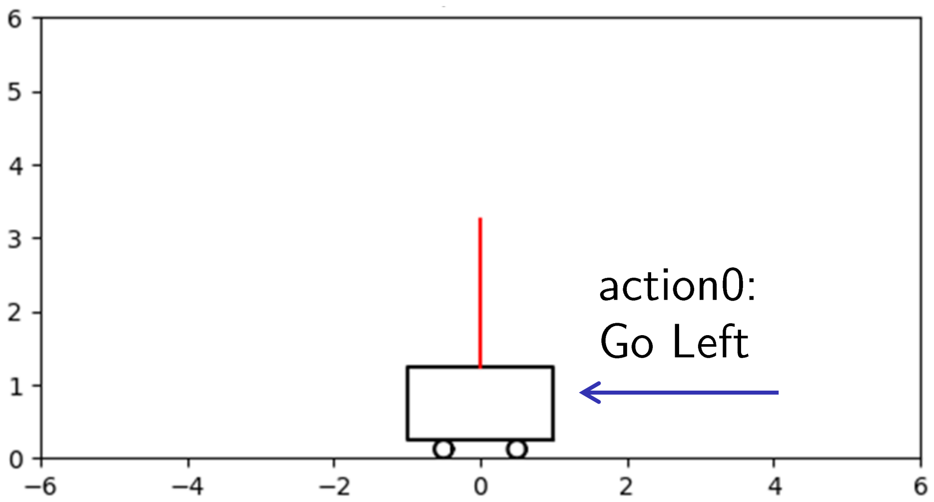

| | Left | Right |
|---|---|---|
| state0 | 0 | 0 |
| state1 | 0 | 0 |
| state2 | 0 | 0.846939 |
| state3 | 0 | 0 |
| state4 | 0 | 0 |
| state5 | 0 | 0 |
| state6 | 0 | 0 |
| state7 | 0 | 0 |

At some time $t$

Let us go to the next step.

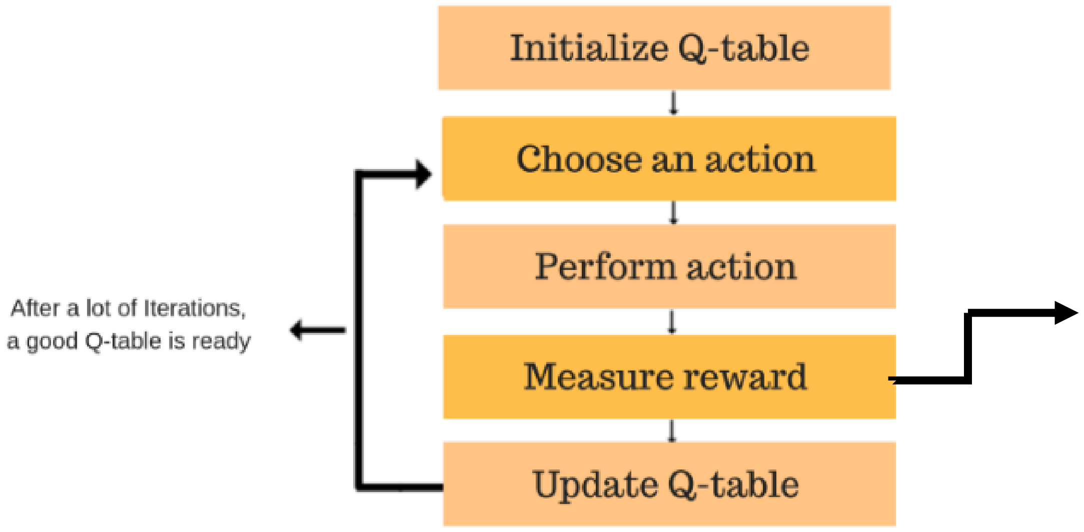
# Q-learning algorithm process




Apply force F towards what direction?

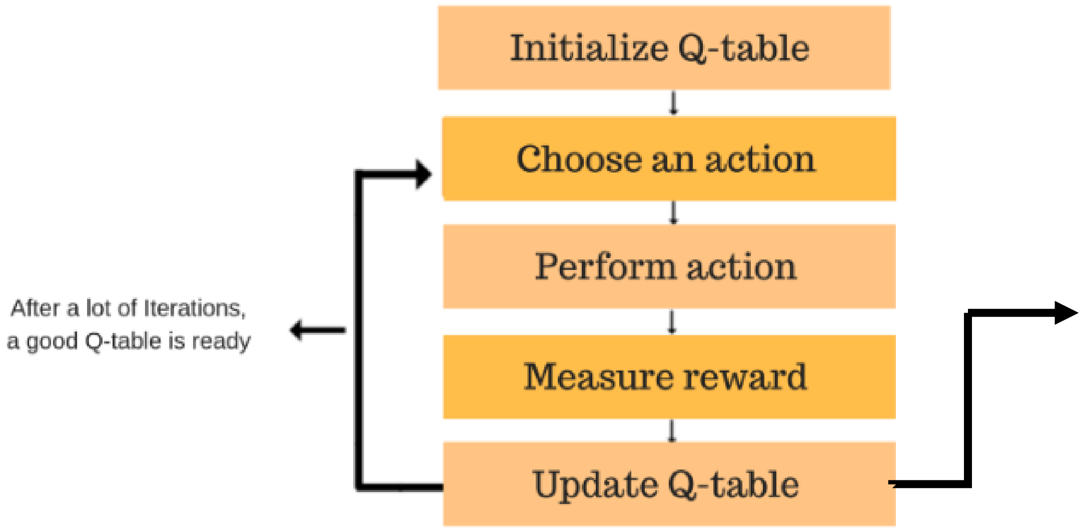
# Q-learning algorithm process




$$r_{t+1} = 1.0 \text{ , if pole stays balanced}$$
$$r_{t+1} = 0.0 \text{ , if pole falls}$$

Applying the same reward system...

# Q-learning algorithm process




| | Left | Right |
|---|---|---|
| state0 | 0 | 0 |
| state1 | 0 | 0 |
| state2 | 0 | 0.846939 |
| state3 | 0 | 0 |
| state4 | 0 | 0 |
| state5 | 0.340343 | 0 |
| state6 | 0 | 0 |
| state7 | 0 | 0 |

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

# Explore new states and algorithms

- We explore different **states** for the cartpole simulation.
  - $S = \{s_{bin}\}$
  - $S = \{x, \dot{x}, \theta, \dot{\theta}\}$
  - $S = I^{H \times W \times C}$
- Baseline RL algorithms
- Where do we go from here? You write your code using the template uploaded in GitHub.

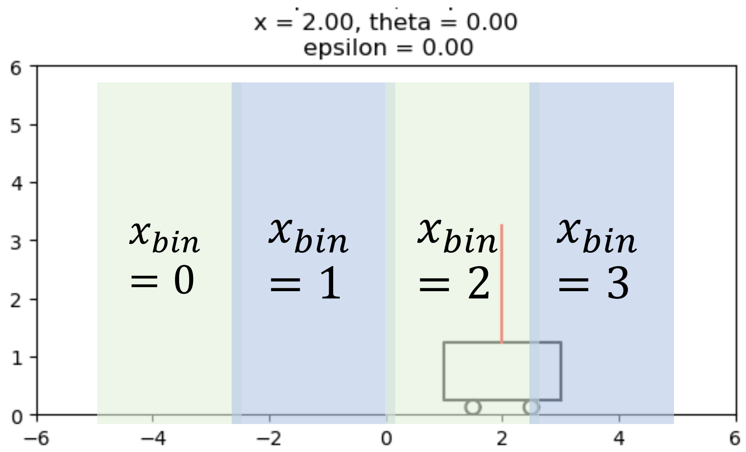
# First, the State is:

Represented the four variables $(x, \dot{x}, \theta, \dot{\theta})$ with a single scalar value,

$$S = \{s_{bin}\}$$

where,

$$s_{bin} = x_{bin} \cdot n_{bins}^3 + \dot{x}_{bin} \cdot n_{bins}^2 + \theta_{bin} \cdot n_{bins} + \dot{\theta}_{bins}$$




| | action0 | action1 |
|---|---|---|
| state0 | 0 | 0 |
| state1 | 0 | 0 |
| state2 | 0 | 0 |
| state3 | 0 | 0 |
| state4 | 0 | 0 |
| . | 0 | 0 |
| . | 0 | 0 |
| . | 0 | 0 |
| state$_{N-1}$ | 0 | 0 |

# Next, using $S = \{x, \dot{x}, \theta, \dot{\theta}\}$

Instead of a single value, we define state based on the actual parameters.

- *Discretized* bins: $S = \{s_{bin}\}$
- *Continuous* state representation: $S = \{x, \dot{x}, \theta, \dot{\theta}\}$

The state space is $S \subset \mathbb{R}^4$, meaning there are infinitely many possible states.

- ⚠️ A Q-table is impossible here because you need infinite rows.
- Instead, we replace it with a function approximator, can be Linear Function or Neural Networks; formally:

$$Q(s, a) \approx f_\vartheta(s, a)$$

where $f_\vartheta$ is parameterized function with weights $\vartheta$.

# Lastly, using $S = I^{H \times W \times C}$

- Raw visual input is the only available observation.
- The state space is $S \subset \{0,1,\cdots,255\}^{H \times W \times C}$. Thus, we use function approximator:

$$Q(s,a) \approx f_{\vartheta}(s,a)$$

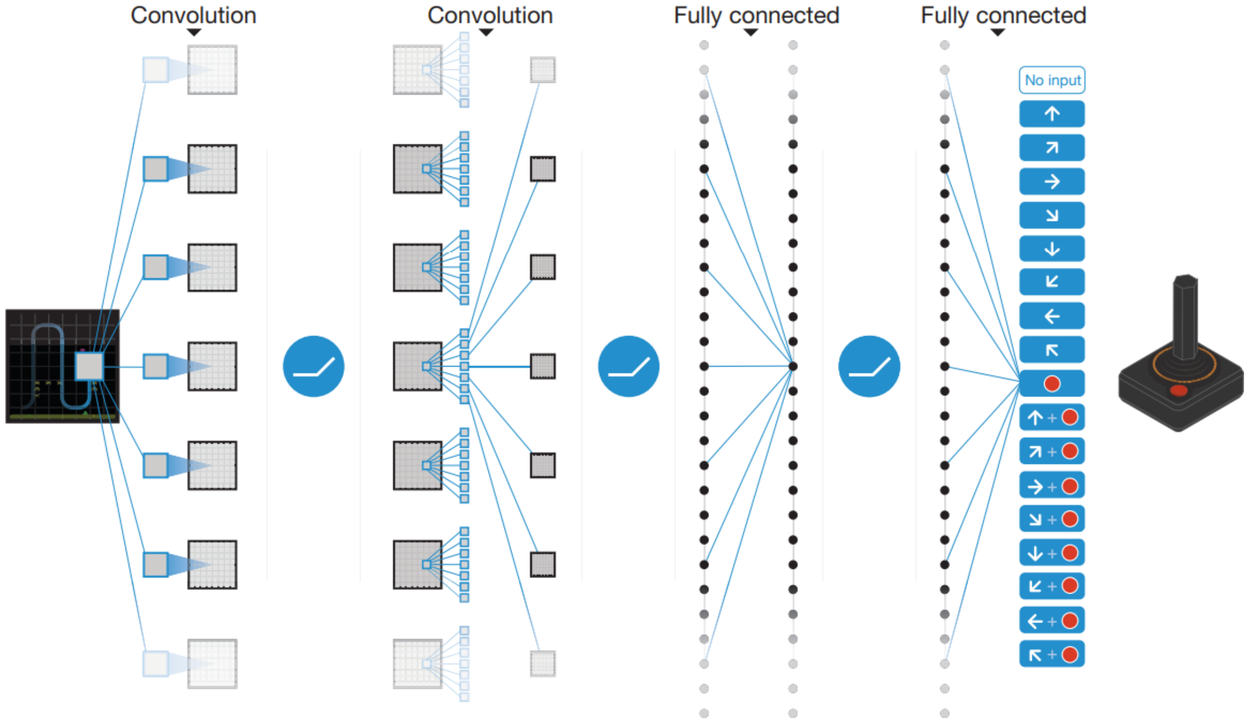
where $f_{\vartheta}$ is parameterized function with weights $\vartheta$.

# Deep Q Network

**Algorithm 1** Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  **for** $t = 1, T$ **do**
    With probability $\epsilon$ select a random action $a_t$
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    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 $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  **end for**
**end for**

Source: [Playing Atari with Deep Reinforcement Learning]




Source: [Human-level control through deep reinforcement learning]

# Baseline RL Algorithms

- Stable-baseline3 is a set of reliable implementations of RL algorithms in PyTorch.

| Name | Box | Discrete | MultiDiscrete | MultiBinary | Multi Processing |
|---|---|---|---|---|---|
| ARS [1] | ✔ | ✔ | ❌ | ❌ | ✔ |
| A2C | ✔ | ✔ | ✔ | ✔ | ✔ |
| CrossQ [1] | ✔ | ❌ | ❌ | ❌ | ✔ |
| DDPG | ✔ | ❌ | ❌ | ❌ | ✔ |
| DQN | ❌ | ✔ | ❌ | ❌ | ✔ |
| HER | ✔ | ✔ | ❌ | ❌ | ✔ |
| PPO | ✔ | ✔ | ✔ | ✔ | ✔ |
| QR-DQN [1] | ❌ | ✔ | ❌ | ❌ | ✔ |
| RecurrentPPO [1] | ✔ | ✔ | ✔ | ✔ | ✔ |
| SAC | ✔ | ❌ | ❌ | ❌ | ✔ |
| TD3 | ✔ | ❌ | ❌ | ❌ | ✔ |
| TQC [1] | ✔ | ❌ | ❌ | ❌ | ✔ |
| TRPO [1] | ✔ | ✔ | ✔ | ✔ | ✔ |
| Maskable PPO [1] | ❌ | ✔ | ✔ | ✔ | ✔ |