Introduction to deep reinforcement learning

DEEP REINFORCEMENT LEARNING IN PYTHON

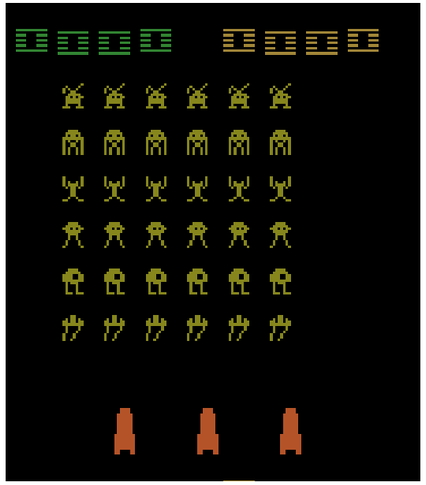
Timothée Carayol
Principal Machine Learning Engineer, Komment

datacamp

Why Deep Reinforcement Learning

- Traditional RL is suitable for low-dimensional tasks

- Many applications have high-dimensional state and/or action space



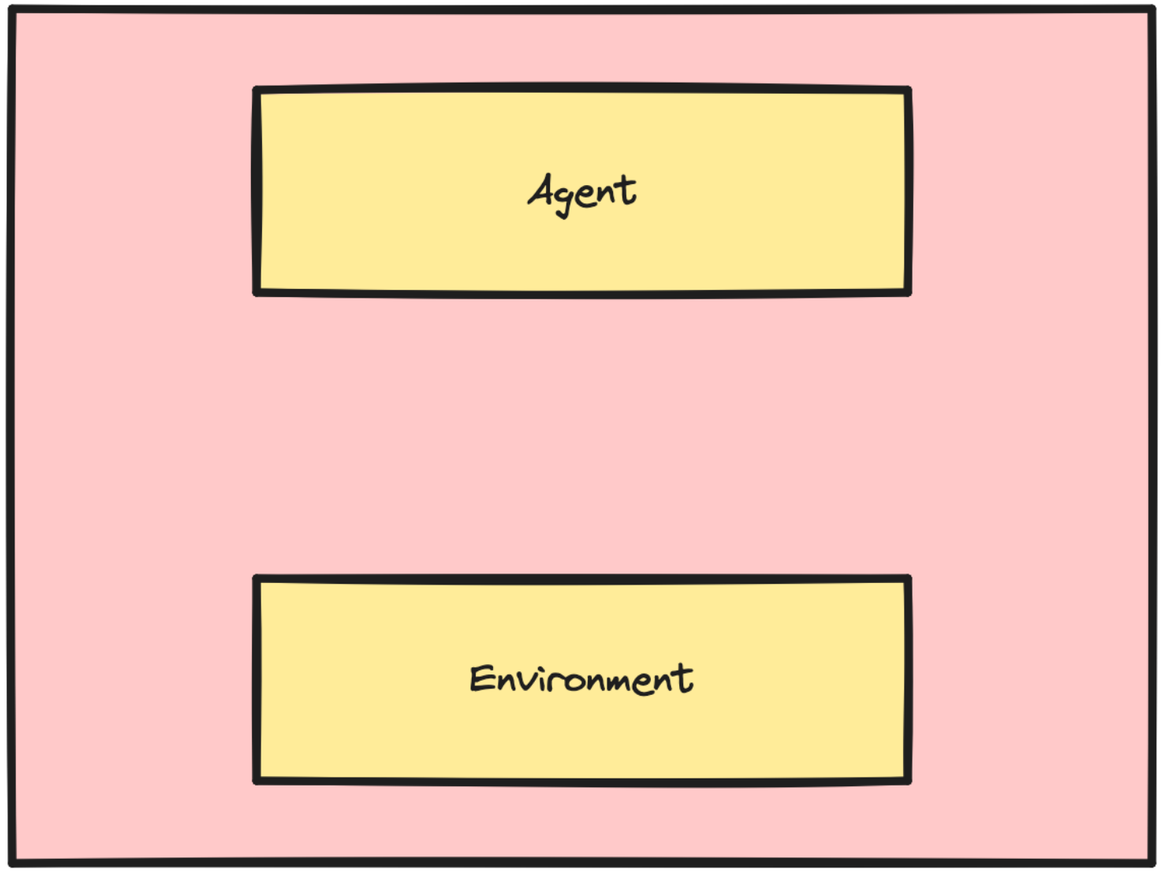
The ingredients of DRL

1. Reinforcement Learning concepts
2. Deep Learning and `PyTorch`

- DRL uses these concepts with deep neural networks

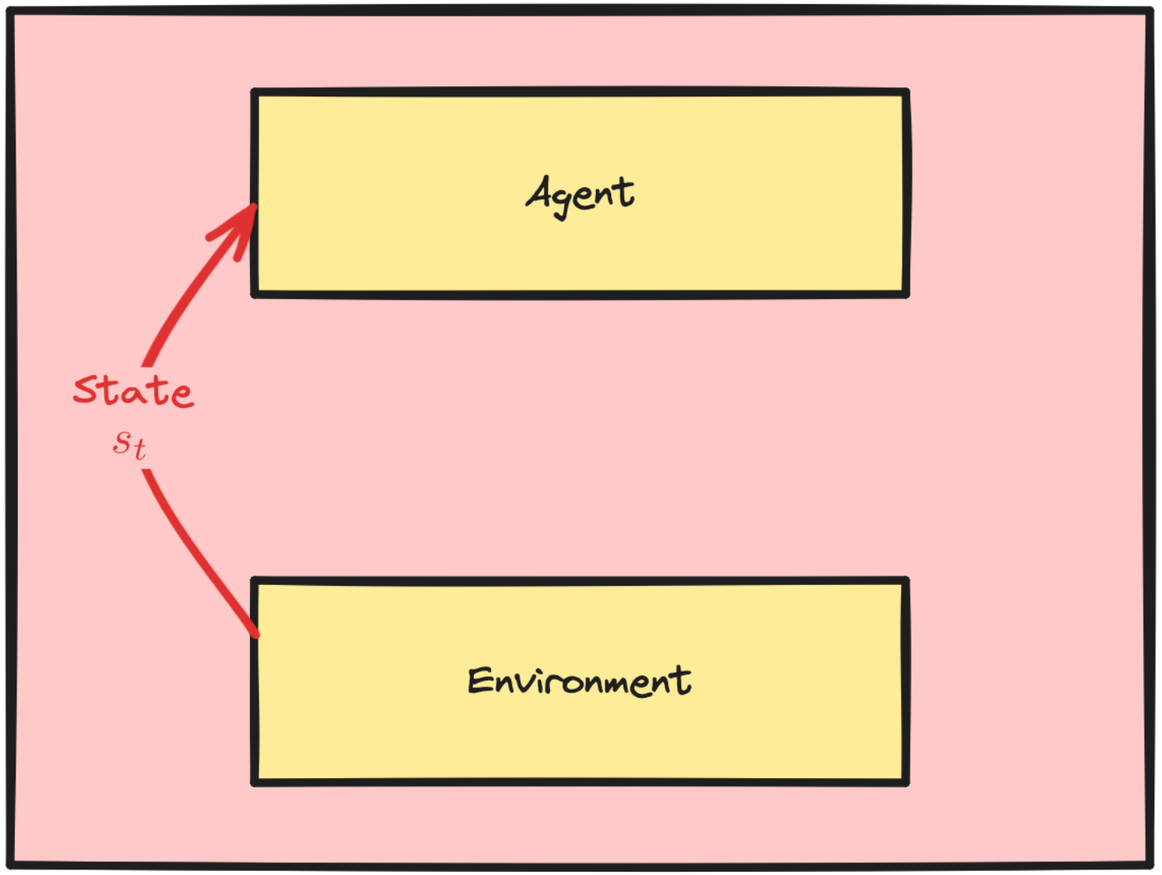
The RL framework

- Step t:



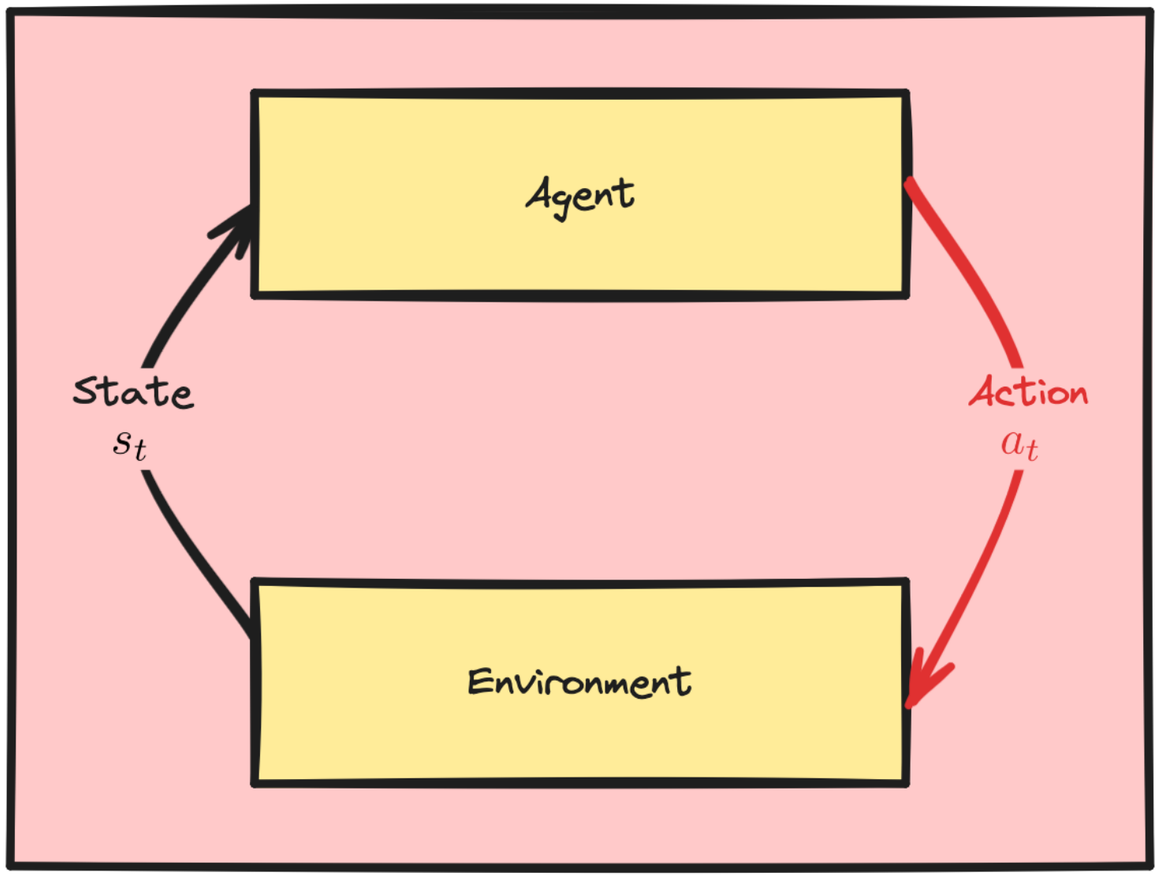
The RL framework

- Step t:
  - Agent observes state s_t



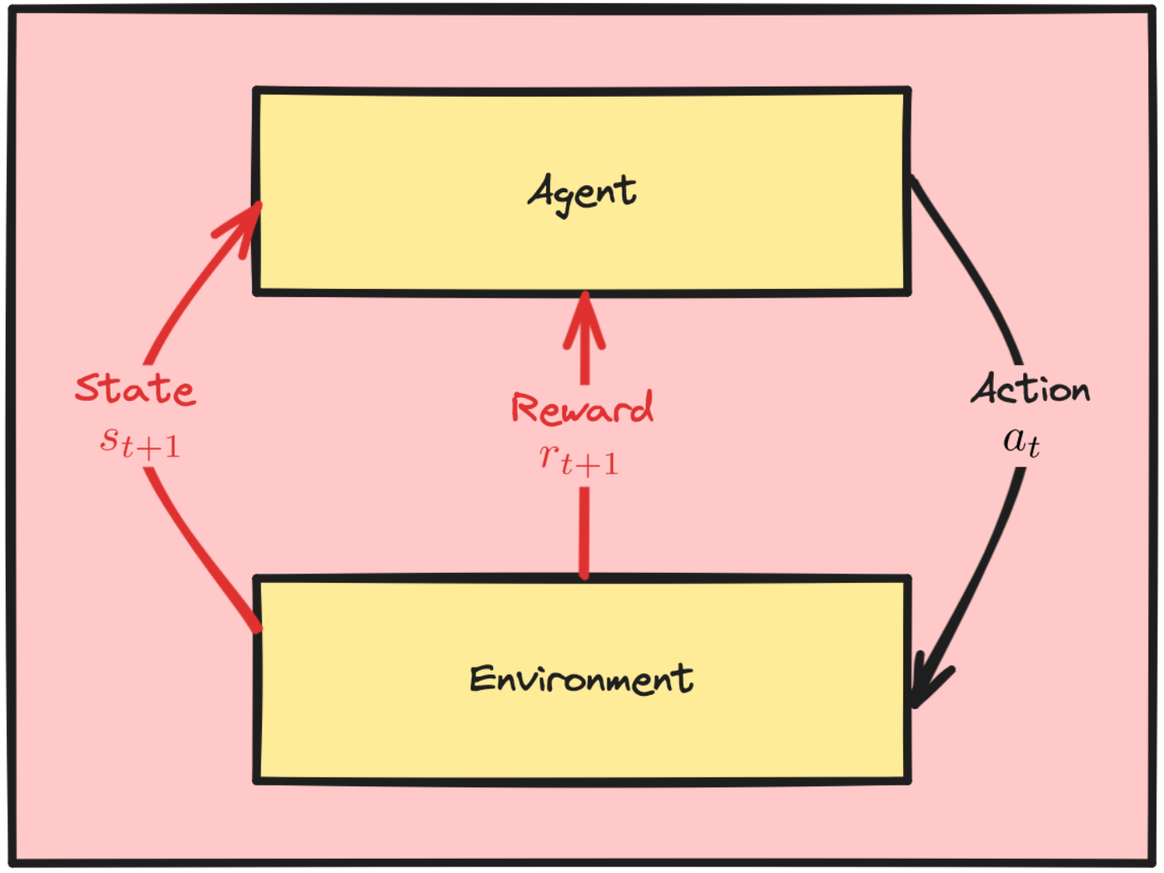
The RL framework

- Step t:
  - Agent observes state s_t
  - Agent takes action a_t



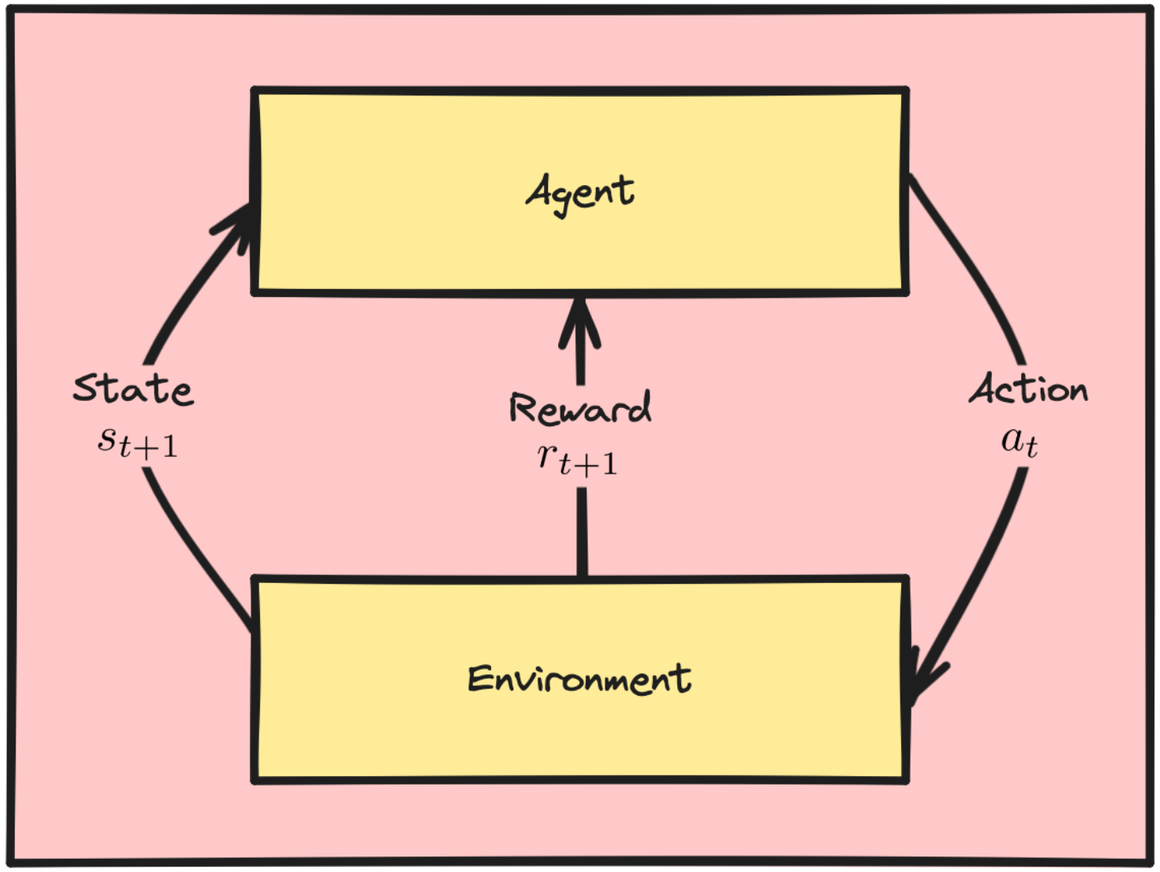
The RL framework

- Step t:
  - Agent observes state s_t
  - Agent takes action a_t
- Step t+1:
  - Environment gives reward r_t
  - State evolves to s_{t+1}



The RL framework

- Step t:
  - Agent observes state s_t
  - Agent takes action a_t
- Step t+1:
  - Environment gives reward r_t
  - State evolves to s_{t+1}
- Repeat until episode is complete



Policy $\pi(s_t)$

- Mapping from state to action, describing how the agent behaves in a given state s_t

- **Deterministic:**
  - Returns the selected action
- **Stochastic:**
  - Returns a distribution over actions
  - Policy is a probability distribution over possible actions

Trajectory and episode return

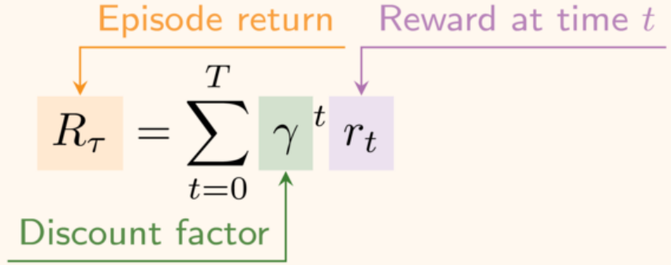
Trajectory τ

Sequence of all states and actions in an episode

$$\tau = \Big((s_0, a_0), (s_1, a_1), ..., (s_T, a_T) \Big)$$

Episode return R_τ

Total (discounted) rewards accumulated along trajectory τ.



Setting up the environment

```
env = gym.make("ALE/SpaceInvaders-v5")
# Define neural network architecture
class Network(nn.Module):
    def __init__(self, dim_inputs, dim_outputs):
        super(Network, self).__init__()
        self.linear = nn.Linear(dim_inputs, dim_outputs)
    def forward(self, x):
        return self.linear(x)
# Instantiate network
network = Network(dim_inputs, dim_outputs)
# Instantiate optimizer
optimizer = optim.Adam(network.parameters(), lr=0.0001)
```

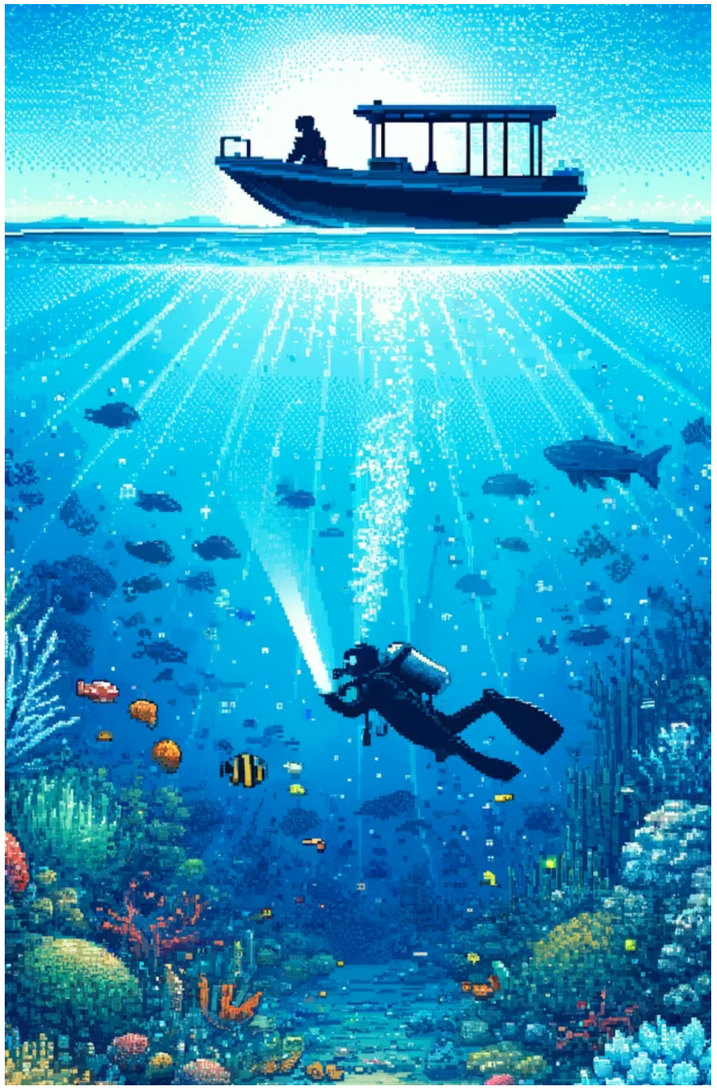
The basic loop

```
for episode in range(1000):
  state, info = env.reset()
  done = False
  while not done:
    action = select_action(network, state)
    next_state, reward, terminated, truncated, _ = (
                                      env.step(action))
    done = terminated or truncated

    loss = calculate_loss(network, state, action,
          next_state, reward, done)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    state = next_state
```

- Outer loop: iterate through episodes
- Inner loop: iterate through steps
  - Select an action
  - Observe new state and reward
  - Calculate the loss and update the network
  - Update the state
- (Loss?)

Coming next

- DRL is powerful!
- Value-based and policy-based approaches
- DQN and refinements
- Policy gradient methods



Let's practice!

DEEP REINFORCEMENT LEARNING IN PYTHON

Introduction to deep Q learning

DEEP REINFORCEMENT LEARNING IN PYTHON

Timothée Carayol

Principal Machine Learning Engineer, Komment

datacamp

What is Deep Q Learning?

Q-Learning refresher

Action Value function $Q_\pi(s, a)$

Sum of future rewards if action a is taken in state s, assuming that the policy π is followed afterwards.

Action Value function

Sum of future rewards

$$Q_\pi(s, a) = E_{\tau \sim \pi}\left[R_\tau \middle| s_t = s, a_t = a\right]$$

Expected value over future trajectories given policy π is followed

- Knowledge of Q would enable optimal policy:

$$\pi(s_t) = \arg\max_a Q(s_t, a)$$

- Goal of Q-learning: learn Q over time

Q-Learning refresher

Bellman Equation (in Q-learning)

In a deterministic environment:

Action Value function

Discount rate

$$Q_\pi(s_t, a_t) = r_{t+1} + \gamma \max_{a_{t+1}} Q_\pi(s_{t+1}, a_{t+1})$$

Next reward

Q-value of the best next action

Temporal Difference target

(A.k.a. TD-target, Q-target, or target Q-value.)
Refers to the right side of the Bellman equation, used as target value for the Q-learning update rule.

$$r_{t+1} + \gamma \max_{a_{t+1}} Q_\pi(s_{t+1}, a_{t+1})$$

- Bellman equation: recursive formula for Q
- Right side of Bellman Equation: "TD-target"
- Use TD-target from Bellman Equation to update $\hat{Q}$ after each step

Q-learning update rule

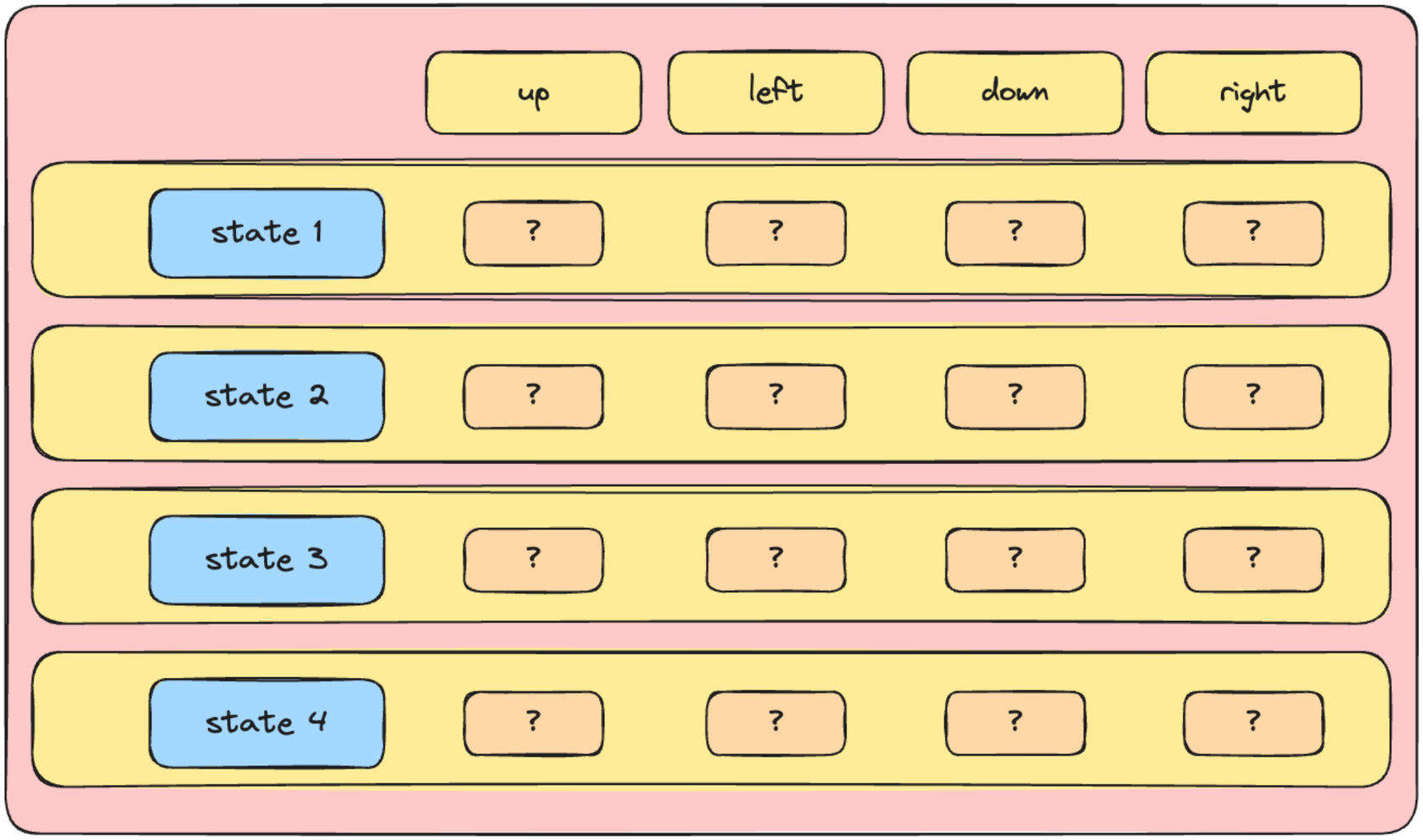
Learning rate α

$$\hat{Q}_{new}(s_t, a_t) = (1-\alpha)\ \hat{Q}_{old}(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}_{old}(s_{t+1}, a_{t+1}) \right)$$

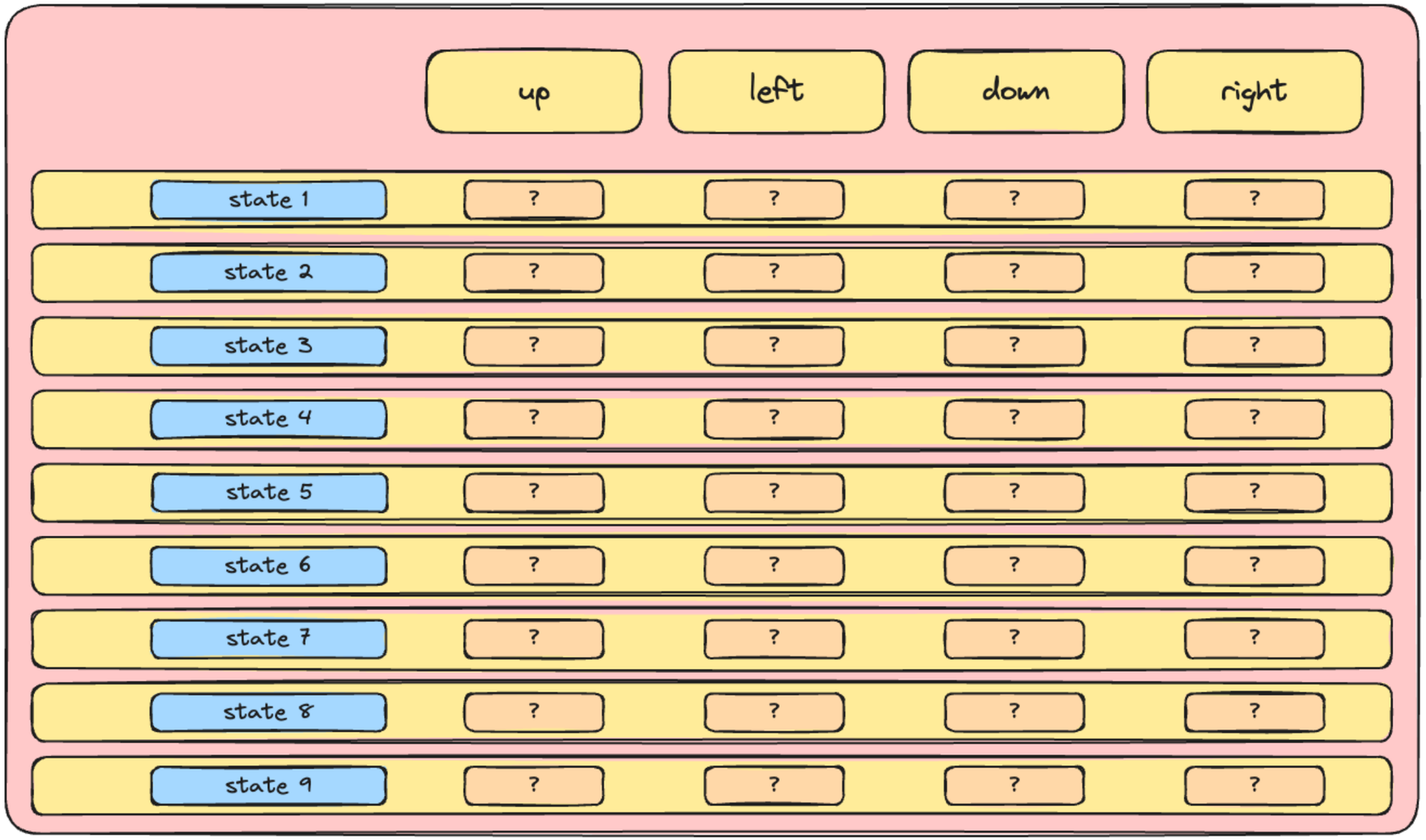
Updated Q-value

Previous Q-value

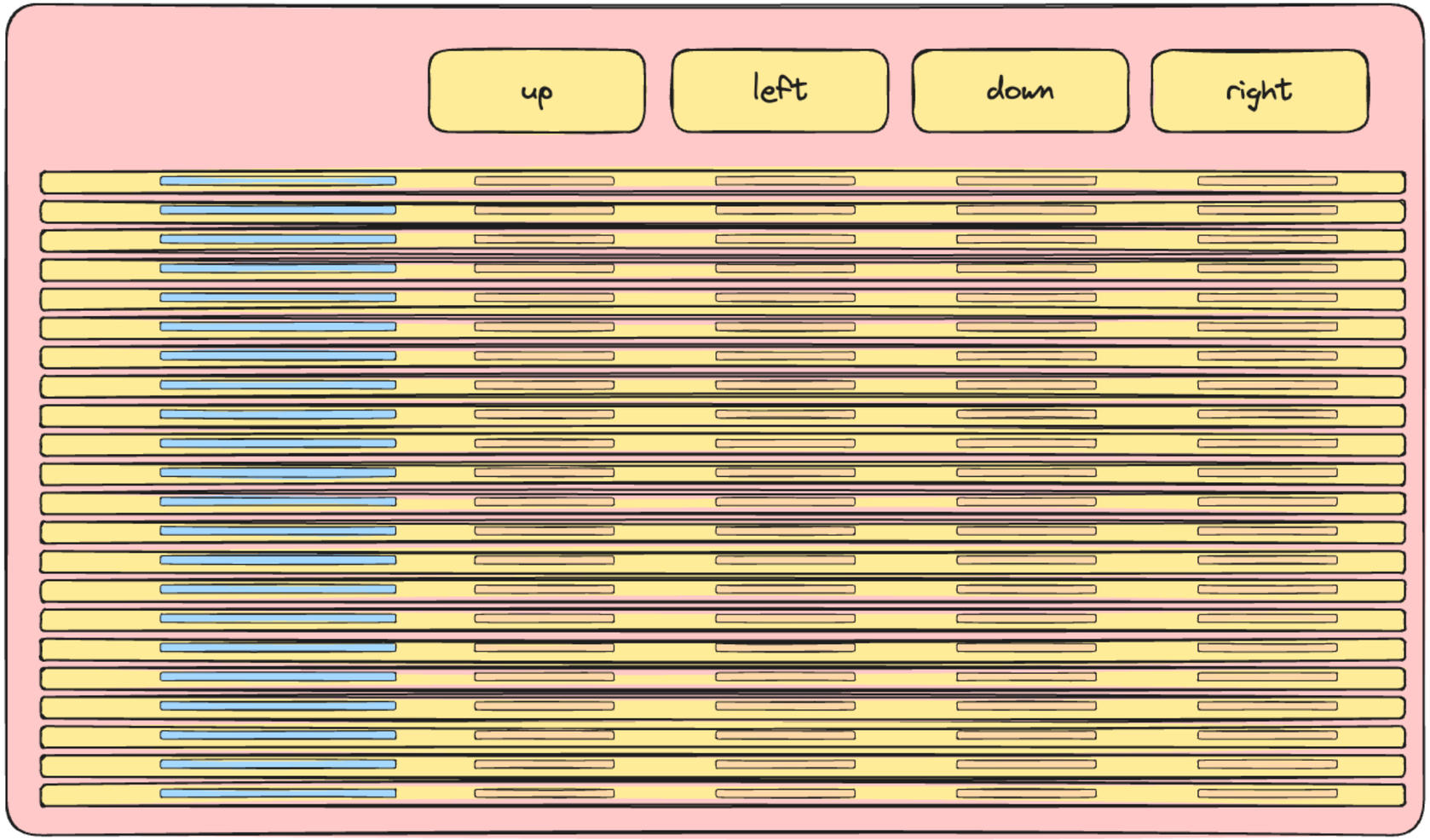
The Q-Network



The Q-Network

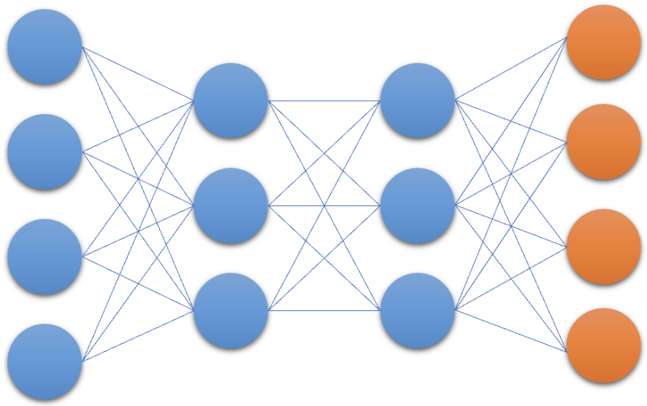


The Q-Network



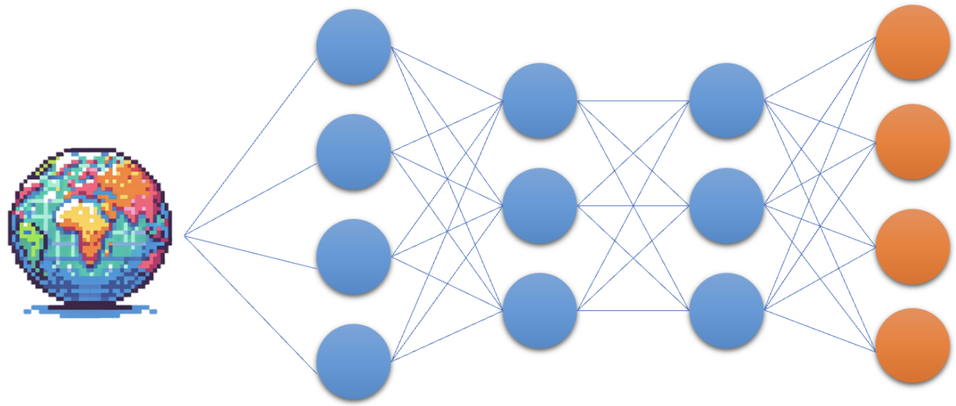
The Q-Network

- At the heart of Deep Q Learning: a neural network



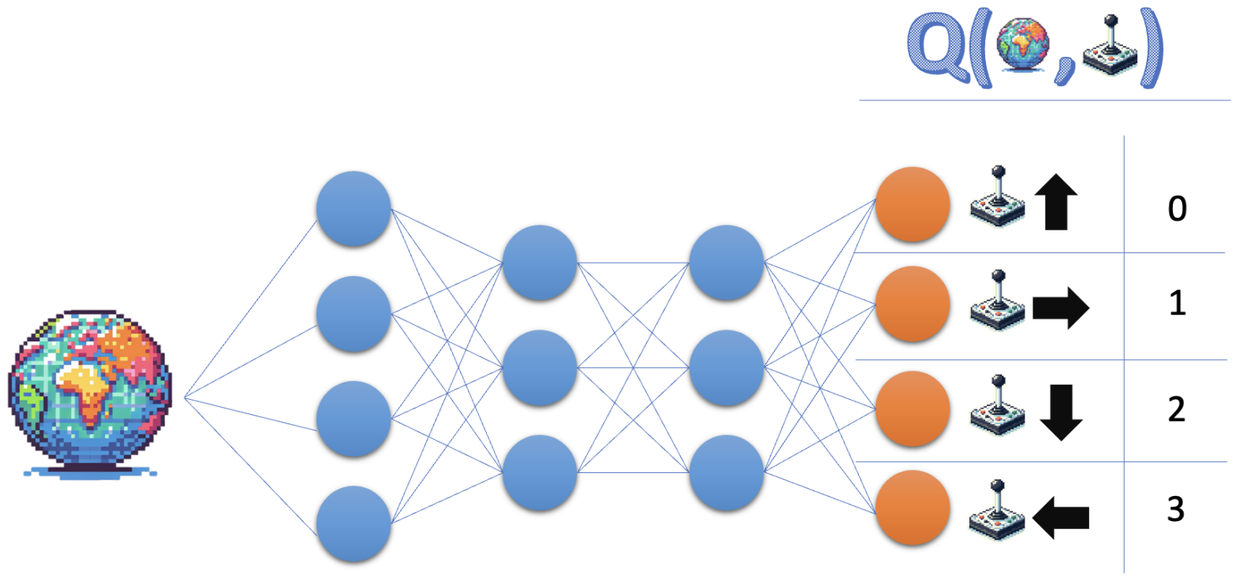
The Q-Network

- At the heart of Deep Q Learning: a neural network



The Q-Network

- At the heart of Deep Q Learning: a neural network mapping state to Q-values



- A network approximating the action-value function is called 'Q-network'
- Q-networks are commonly used in Deep Q Learning algorithms, such as DQN.

Implementing the Q-network

```
class QNetwork(nn.Module):

  def __init__(self, state_size, action_size):
    super(QNetwork, self).__init__()
    self.fc1 = nn.Linear(state_size, 64)
    self.fc2 = nn.Linear(64, 64)
    self.fc3 = nn.Linear(64, action_size)

  def forward(self, state):
    x = torch.relu(self.fc1(torch.tensor(state)))
    x = torch.relu(self.fc2(x))
    return self.fc3(x)

q_network = QNetwork(8, 4)
optimizer = optim.Adam(q_network.parameters(),
                       lr=0.0001)
```

- Input dimension determined by state
- Output dimension determined by number of possible actions
- In this example:
  - 2 hidden layers with 64 nodes each
  - ReLu activation function

Let's practice!

DEEP REINFORCEMENT LEARNING IN PYTHON

The barebone DQN algorithm

DEEP REINFORCEMENT LEARNING IN PYTHON

Timothée Carayol

Principal Machine Learning Engineer, Komment

datacamp

The Barebone DQN

- Our first step to the full DQN algorithm
- Features:
  - Generic DRL training loop
  - A Q-network
  - Principles of Q-learning

```
for episode in range(1000):
  state, info = env.reset()
  done = False
  while not done:
    # Action selection
    action = select_action(network, state)
    next_state, reward, terminated, truncated, _ = (
                                        env.step(action))
    done = terminated or truncated
    # Loss calculation
    loss = calculate_loss(network, state, action,
                          next_state, reward, done)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    state = next_state
```

The Barebone DQN action selection

```
def select_action(q_network, state):
    # Feed state to network to obtain Q-values
    q_values = q_network(state)
    # Obtain index of action with highest Q-value
    action = torch.argmax(q_values).item()
    return action
```

```
Q-values: [-0.01, 0.08, 0.12, -0.07]
Action selected: 2, with q-value 0.12
```

- Policy: select action with highest Q-value
  - $a_t = \arg\max_a Q(S_t, a)$
  - Here: action 2, with Q-value `0.12`

The Barebone DQN loss function

- Action-Value function satisfies Bellman Equation
- Idea: minimize the difference between both sides a.k.a. TD-error or Bellman error
- Use Squared Bellman Error as loss function:

DQN loss (Squared Bellman Error)

$$\mathcal{L}(\theta) = \left[\left(r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}_\theta(s_{t+1}, a_{t+1}) \right) - \hat{Q}_\theta(s_t, a_t) \right]^2$$

TD target

Current Q value estimate

Bellman Equation (in Q-learning)

$$Q_\pi(s_t, a_t) = r_{t+1} + \gamma \max_{a_{t+1}} Q_\pi(s_{t+1}, a_{t+1})$$

Action Value function

Discount rate

Next reward

Q-value of the best next action

Bellman Error a.k.a. TD-error

$$\left(r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}) \right) - \hat{Q}(s_t, a_t)$$

TD target

Current Q value estimate

The Barebone DQN loss function

```
def calculate_loss(
      q_network, state, action,
      next_state, reward, done):
    q_values = q_network(state)
    current_state_q_value = q_values[action]
    next_state_q_value = q_network(next_state).max()
    target_q_value = reward +
                gamma * next_state_q_value * (1-done)
    loss = nn.MSELoss()(
              current_state_q_value, target_q_value)
    return loss
```

- **Current state Q-value:**

$$Q(s_t, a_t)$$

- **Next state Q-value:**

$$\max_a Q(s_{t+1}, a)$$

- **Target Q-value:**

$$r_{t+1} + \gamma \max_a Q(s_{t+1}, a)$$

- **DQN Loss:**

$$\left(Q(s_t, a_t) - (r_{t+1} + \gamma \max_a Q(s_{t+1}, a)\right)^2$$

Describing the episodes

```
describe_episode(episode, reward, episode_reward, step)
```

```
| Episode    1 | Duration:   84 steps | Return: -871.38 | Crashed  |
| Episode    2 | Duration:   53 steps | Return: -452.68 | Crashed  |
| Episode    3 | Duration:   57 steps | Return: -414.22 | Crashed  |
| Episode    4 | Duration:   54 steps | Return: -475.09 | Crashed  |
| Episode    5 | Duration:   67 steps | Return: -532.31 | Crashed  |
| Episode    6 | Duration:   53 steps | Return: -407.00 | Crashed  |
| Episode    7 | Duration:   52 steps | Return: -380.45 | Crashed  |
| Episode    8 | Duration:   55 steps | Return: -380.75 | Crashed  |
| Episode    9 | Duration:   88 steps | Return: -688.68 | Crashed  |
| Episode   10 | Duration:   76 steps | Return: -338.06 | Crashed  |
```

Let's practice!

DEEP REINFORCEMENT LEARNING IN PYTHON