Deep Reinforcement Learning - 2

Kalle Prorok, Umeå 2020

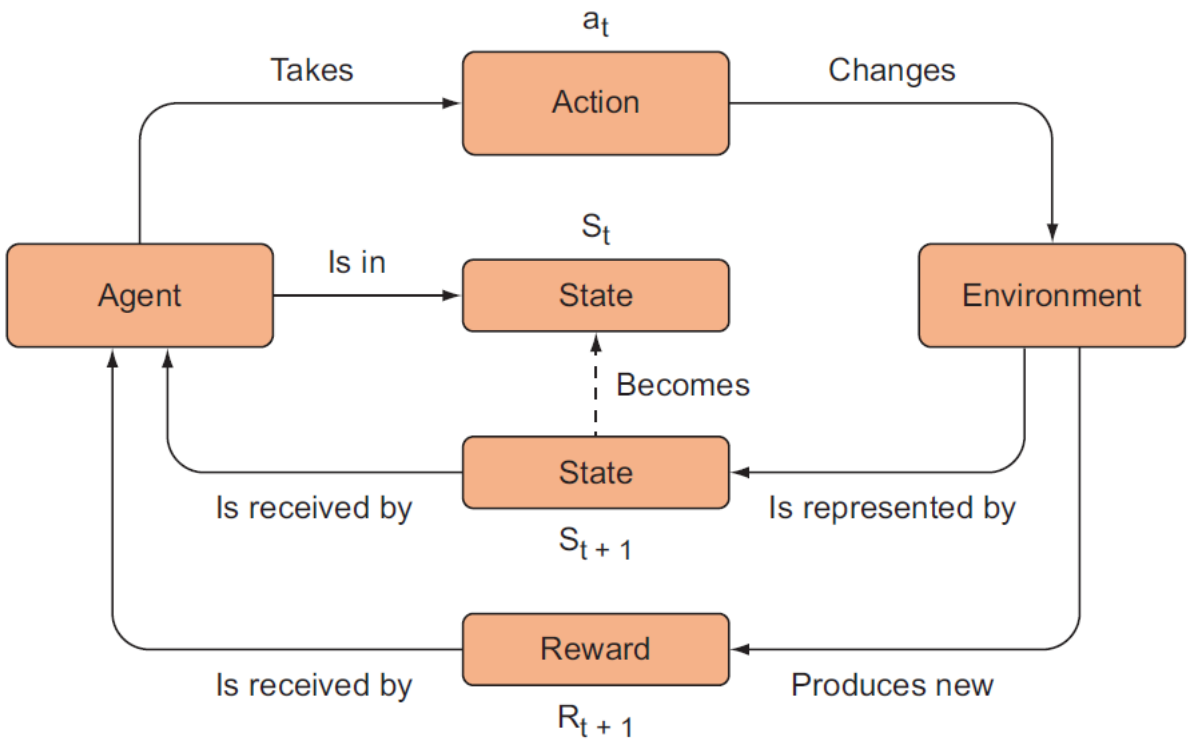
Parts based on the Manning book

Deep Reinforcement Learning in Action

By Alexander Zai and Brandon Brown

And the Sutton-Barto book

Agent


a_t
Takes
Action
Changes
S_t
Agent
Is in
State
Environment
Becomes
State
Is received by
Is represented by
S_{t+1}
Reward
Is received by
Produces new
R_{t+1}

Dilemma/Balance

Exploration

- Play the game and observe the rewards we get for the various machines
- To learn more
- Random selected action?

Exploitation

- Use our current knowledge about which machine seems to produce the most rewards, and keep playing that machine
- greedy

Epsilon-greedy strategy

- with a probability, ε, we will choose an action, *a*, at random, and the rest of the time (probability 1 – ε) we will choose the best lever based on what we currently know from past plays

```
eps = 0.2
rewards = [0]
for i in range(500):
    if random.random() > eps:
        choice = get_best_arm(record)

    else:
        choice = np.random.randint(10)
    r = get_reward(probs[choice])
```

Chooses the best action with 0.8 probability, or randomly otherwise

Computes the reward for choosing the arm

Softmax selection policy

- Treating patients with heart attacks, choose 1 treatment out of 10
  - doesn't know which one is the best yet
  - randomly choosing a treatment could result in patient death, not just losing some money. We really want to make sure we don't choose the worst treatment

$$\Pr(A) = \frac{e^{Q_k(A)/\tau}}{\sum_{i=1}^{n} e^{Q_k(i)/\tau}}$$

```
def softmax(av, tau=1.12):
    softm = np.exp(av / tau) / np.sum( np.exp(av / tau) )
    return softm
```

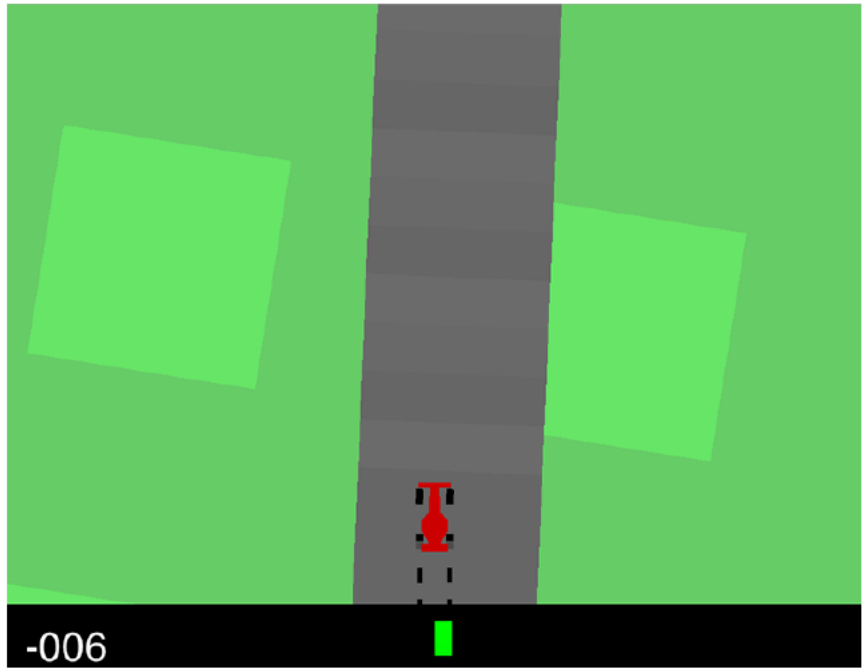
τ is a parameter called *temperature* that scales the probability distribution of actions. Usually reduced over time, "cooling metal".

Contextual (state-based) bandits

- Example: advertisement placement
  - maximize the probability that you will click ads depending on what site you visit
  - Let’s say we manage 10 e-commerce websites, each focusing on selling a different broad category of retail items such as computers, shoes, jewelry, etc.

Open AI Gym

```
import gym
env = gym.make('CarRacing-v0')
env.reset()
env.step(action)
env.render()
```



Expected reward at play k for taking an action (*a*)

$$Q_k(a) = \frac{R_1 + R_2 + \ldots + R_k}{k_a}$$

```
def exp_reward(action, history):
  rewards_for_action = history[action]
  return sum(rewards_for_action) /
     len(rewards_for_action)
```

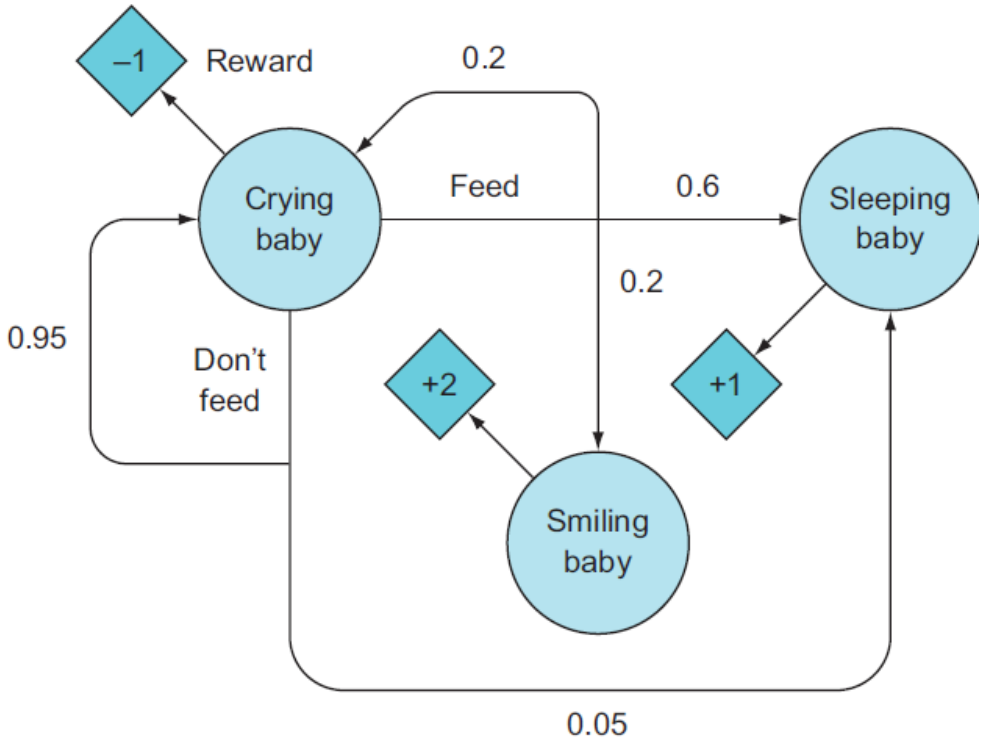
action-value function; the value of taking a particular action

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

maximize the expected discounted return Gt

The Markov property

- the current state alone contains enough information to choose optimal actions to maximize future rewards
- A game (or any other control task) that exhibits the Markov property is said to be a *Markov decision process* (MDP):



Markov property or not?

- Driving a car
- Deciding whether to invest in a stock or not
- Choosing a medical treatment for a patient
- Diagnosing a patient’s illness
- Predicting which team will win in a football game
- Choosing the shortest route (by distance) to some destination
- Aiming a gun to shoot a distant target

Policy functions

- How exactly do we use our current state information to decide what action to take?
- In words, a policy, π, is the strategy of an agent in some environment.
- A policy is a function that maps a state to a probability distribution over the set of possible actions in that state

Math	English
$\pi, s \rightarrow Pr(A \mid s)$, where $s \in S$	A policy, π, is a mapping from states to the (probabilistically) best actions for those states.

- The *optimal policy*—it's the strategy that maximizes rewards.

Math	English
$\pi^* = \text{argmax}\, E(R \mid \pi)$,	If we know the expected rewards for following any possible policy, π, the optimal policy, π^*, is a policy that, when followed, produces the maximum possible rewards.

How to choose the actions that lead to the maximal expected rewards?

Directly—We can teach the agent to learn what actions are best, given what state it is in.

Indirectly—We can teach the agent to learn which states are most valuable (value functions), and then to take actions that lead to the most valuable states.

Value functions

- *Value functions* are functions that map a state to the *expected value* (the expected reward) of being in some state

Math	English
$V_\pi: s \rightarrow E(R \mid s, \pi)$,	A value function, V_π, is a function that maps a state, s, to the expected rewards, given that we start in state s and follow some policy, π.

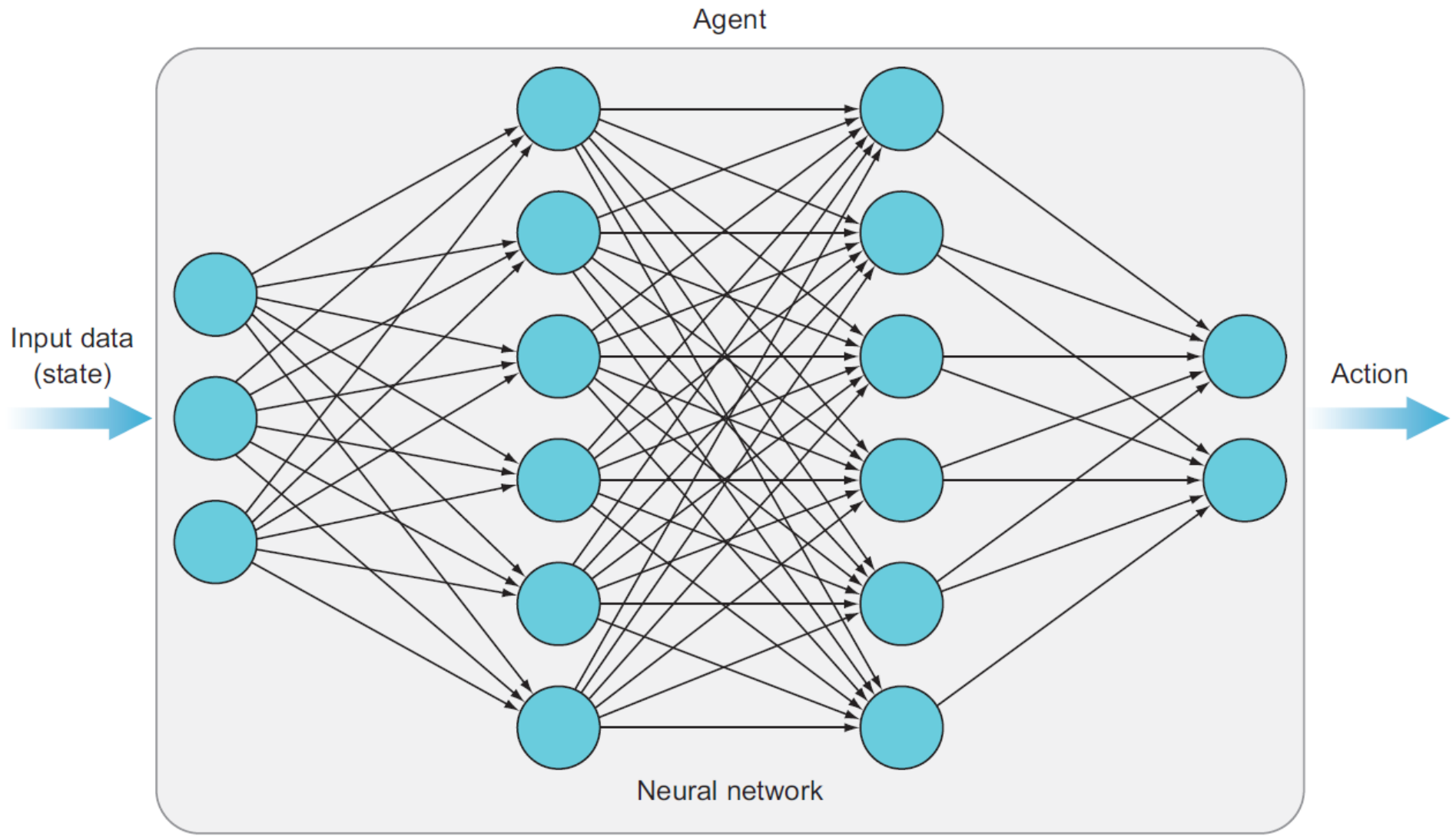
- Accepts a state, *s*, and returns the expected reward of starting in that state and taking actions according to our policy, π
- Actual values depends on the policy

Q function or Q value

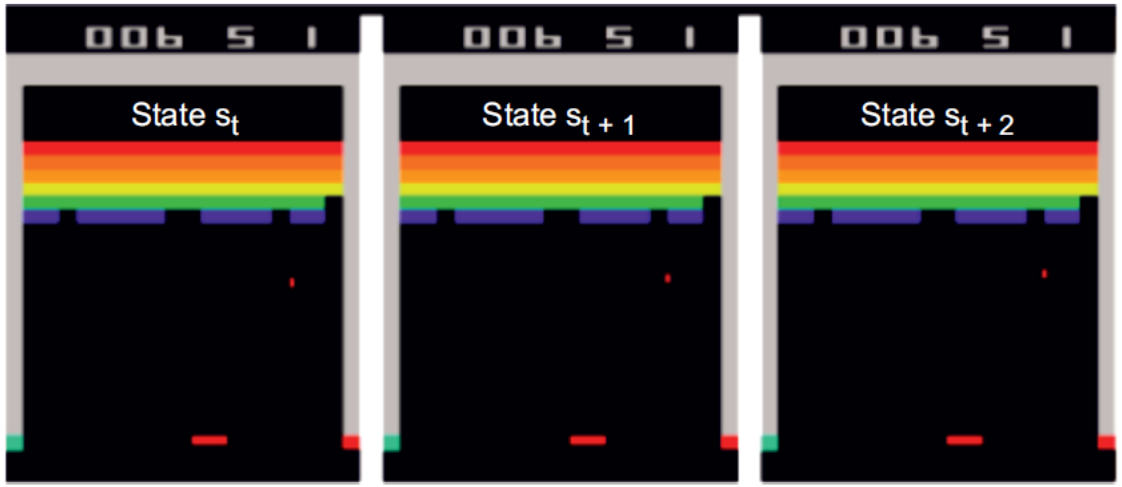
- Estimates of the expected reward for taking an action given a state

Math	English
$Q_\pi: (s \mid a) \rightarrow E(R \mid a,s,\pi)$,	Q_π is a function that maps a pair, (s, a), of a state, s, and an action, a, to the expected reward of taking action a in state s, given that we're using the policy (or "strategy") π.

- Q-learning (Watkins 1989, 1992)
- https://en.wikipedia.org/wiki/Q-learning


Agent
Input data
(state)
Action
Neural network

DeepMind’s DQN algorithm for Atari games



Four (three here) 84 × 84 grayscale images at each step, which would lead to 256^{28228} unique game states. DQN CNN have 1792 parameters