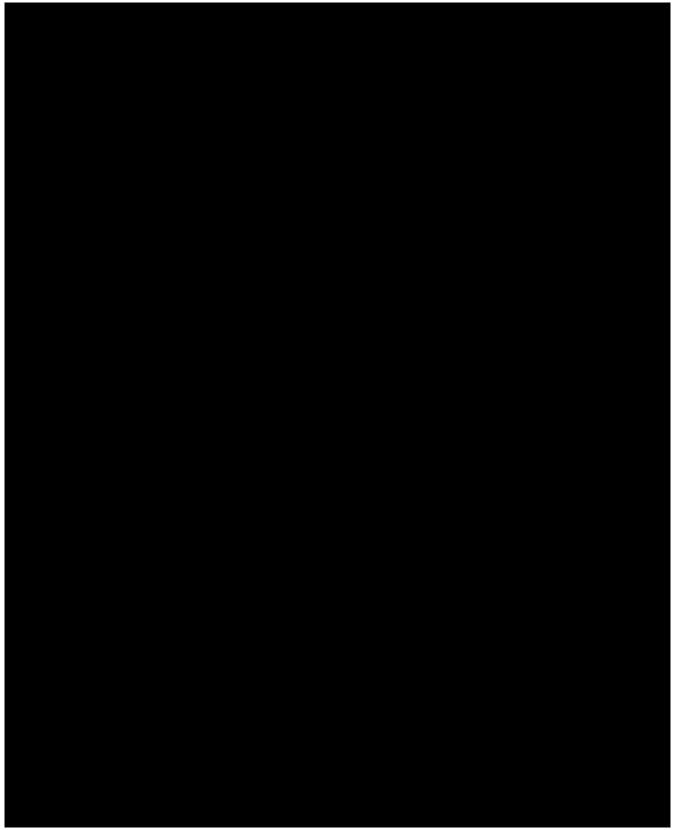
Hands-on Machine Learning

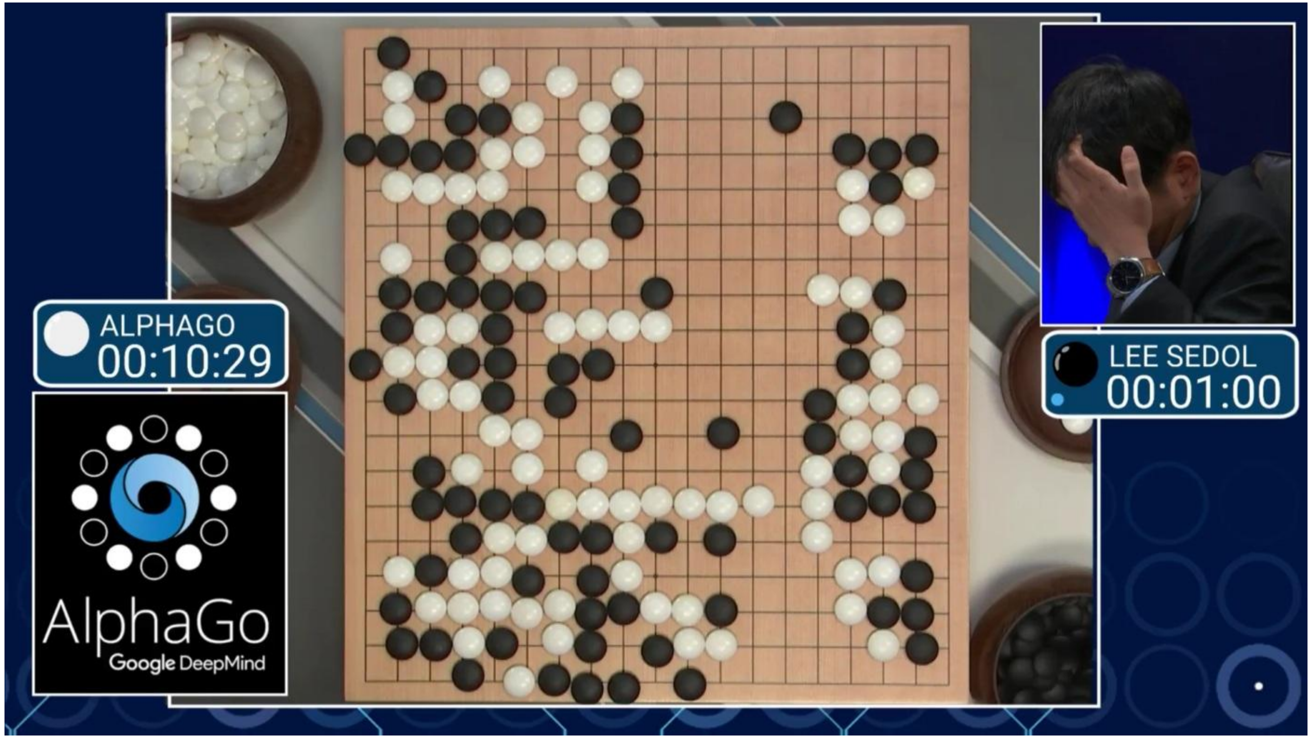
18. Reinforcement Learning

A Revolution in 2013

- DeepMind demonstrated a system that could learn to play any Atari game from scratch, and outperforming humans in most of them, using only raw pixels as inputs and without any prior knowledge of the rules of the games.



AlphaGo

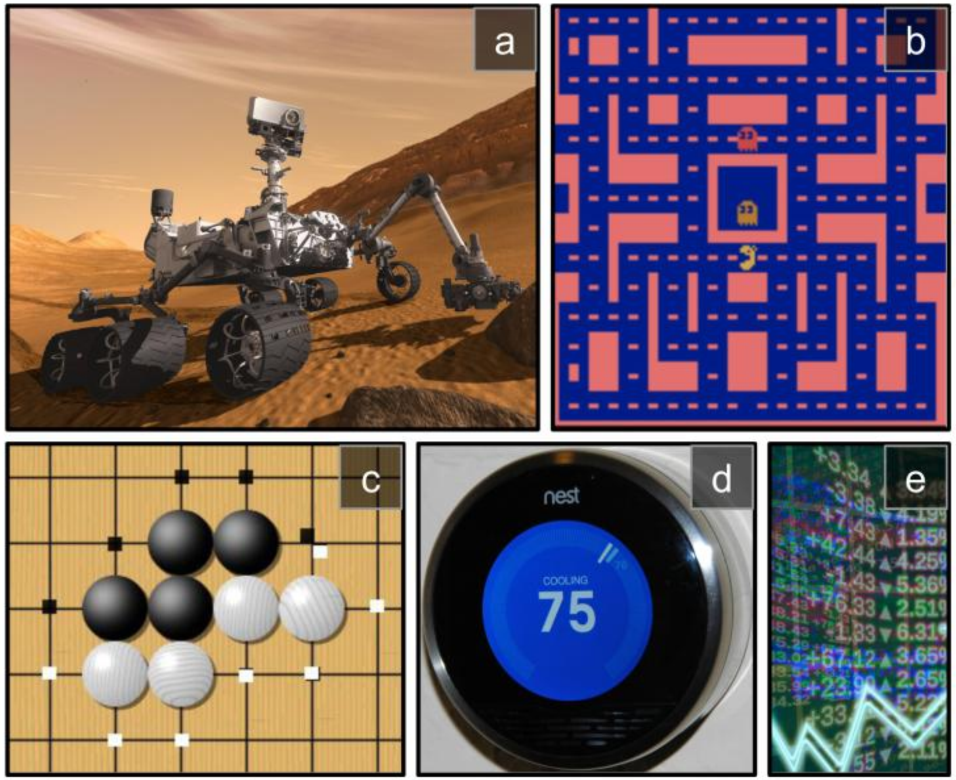


1.
Learning to Optimize Rewards

Reinforcement Learning

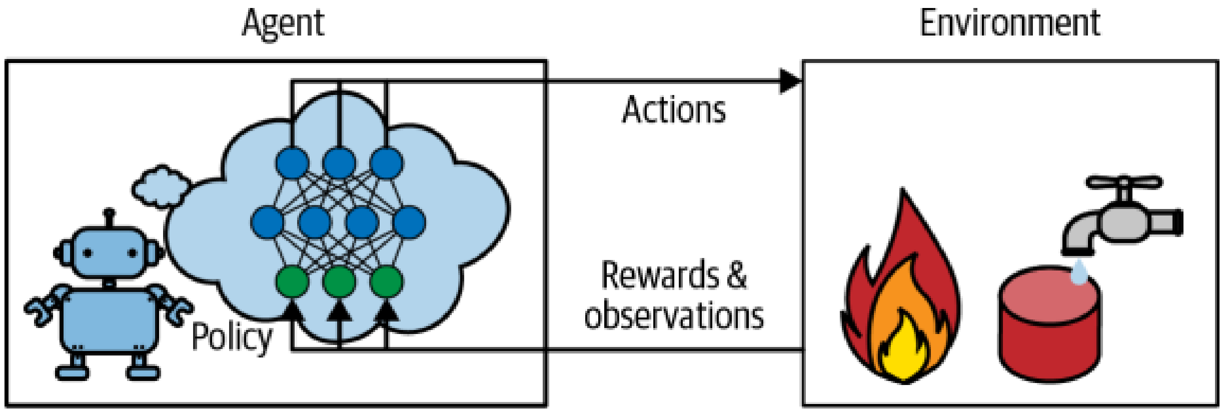
- In reinforcement learning (RL), a software *agent* makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards* from the environment.
- Its objective is to learn to act in a way that will maximize its expected rewards over time.
  - you can think of positive rewards as pleasure, and negative rewards as pain.
- The agent acts in the environment and learns by trial and error to maximize its pleasure and minimize its pain.

Reinforcement Learning Examples



Policy

- The algorithm a software agent uses to determine its actions is called its *policy*.
- The policy could be a neural network taking observations as inputs and outputting the action to take.

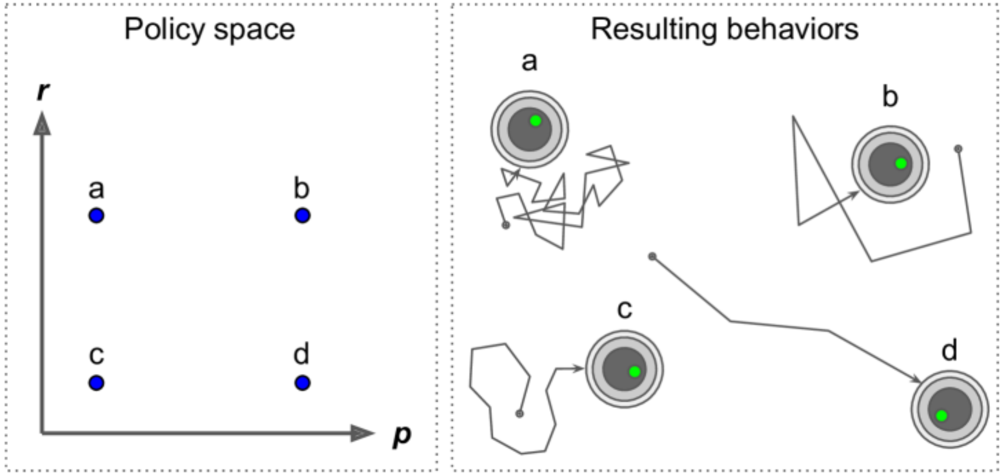


Stochastic Policy

- Consider a robotic vacuum cleaner whose reward is the amount of dust it picks up in 30 minutes.
- Its policy could be to move forward with some probability p every second, or randomly rotate left or right with probability $1-p$.
- The rotation angle would be a random angle between $-r$ and $+r$.
- Since the policy involves randomness, it is called a *stochastic policy*.
  - r and p are called *policy parameters*.
- The robot will have an irregular trajectory, which guarantees that it will eventually get to any place it can reach and pick up all the dust.
  - The question is, how much dust will it pick up in 30 minutes?

Policy Search

- One possible learning algorithm: try out many different values for policy parameters, and pick the combination that performs best.
  - This is an example of *policy search*, using a brute-force approach.
- When the *policy space* is too large, finding a good set of parameters this way is like searching for a needle in a gigantic haystack.



Policy Search via Genetic Algorithm

- To explore the policy space using *genetic algorithms*:
  - You could randomly create a first generation of 100 policies and try them out, then “kill” the 80 worst policies and make the 20 survivors produce 4 offspring each.
    - An offspring is a copy of its parent plus some random variation.
  - The surviving policies plus their offspring together constitute the second generation.
  - You can continue to iterate through generations this way until you find a good policy.

Policy Gradients

- *Policy Gradients* (PG):
  - Use optimization techniques, by evaluating the gradients of the rewards with regard to the policy parameters.
  - Tweaking these parameters by following the gradients toward higher rewards.
- Example:
  - For the vacuum cleaner robot, you could slightly increase p and evaluate whether doing so increases the amount of dust picked up by the robot in 30 minutes (reward).
  - If it does, then increase p some more, or else reduce p.

2.
Gymnasium

Gymnasium

- In order to train an agent, you first need to have a working environment.
- Training is hard and slow in the real world, so you generally need a *simulated environment* at least for bootstrap training.
- OpenAI Gym was a toolkit that provided a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, …), that you could use to train agents, compare them, or develop new RL algorithms.
- **Gymnasium** is a fork of OpenAI Gym that continues its development and maintenance after OpenAI stopped actively updating Gym.

CartPole

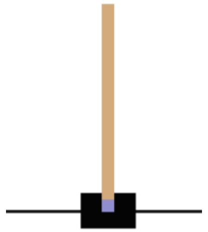
- ➢ Install the Gymnasium library, which provides environments for RL:

```
%pip install -q -U gymnasium swig
%pip install -q -U gymnasium[classic_control,box2d,atari,accept-rom-license]
```

- ➢ Import Gym and make a new CartPole environment:

```
import gymnasium as gym

env = gym.make("CartPole-v1", render_mode="rgb_array")
```



- ➢ CartPole is a game in which you try to balance the pole as long as possible.
  - ➢ The goal of this classic control task is to move the cart left and right so that the pole can stand (within a certain angle) as long as possible.

RL for CartPole

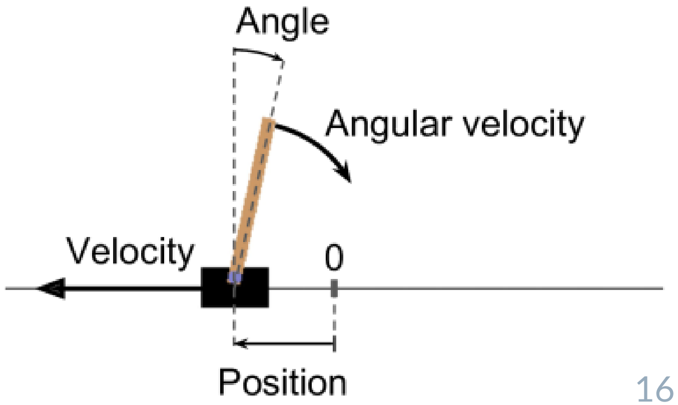
The CartPole Environment

- Initialize the environment using the `reset`() method:

```
obs, info = env.reset(seed=42)
obs
```

```
array([ 0.0273956 , -0.00611216,  0.03585979,  0.0197368 ], dtype=float32)
```

- For the CartPole environment, each observation is a 1D NumPy array containing four floats representing:
  - the cart’s horizontal position (0.0 = center)
  - its velocity (positive means right)
  - the angle of the pole (0.0 = vertical)
  - its angular velocity (positive means clockwise)

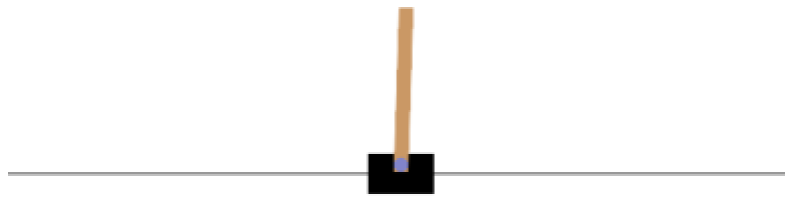


Render the Environment

- Use `render`() method to render this environment as an image.
- We set `render_mode="rgb_array"` when creating the environment, so the image will be returned as a NumPy array:

```
img = env.render()
img.shape  # height, width, channels (3 = Red, Green, Blue)
```

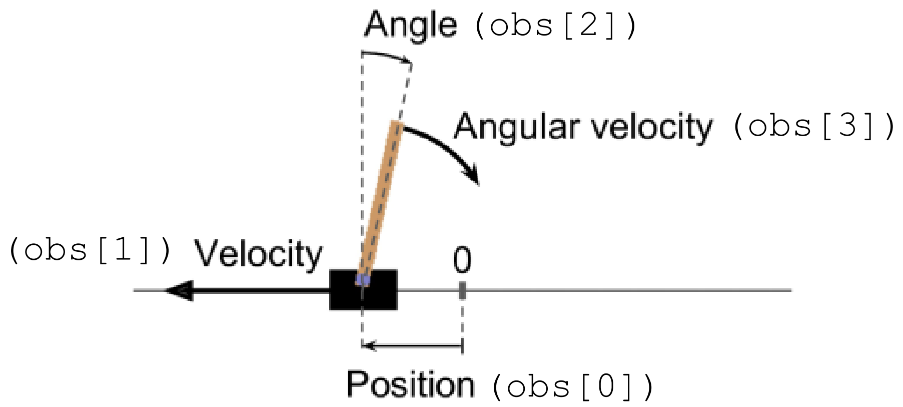
```
(400, 600, 3)
```



Action Space

```
env.action_space
```

```
Discrete(2)
```



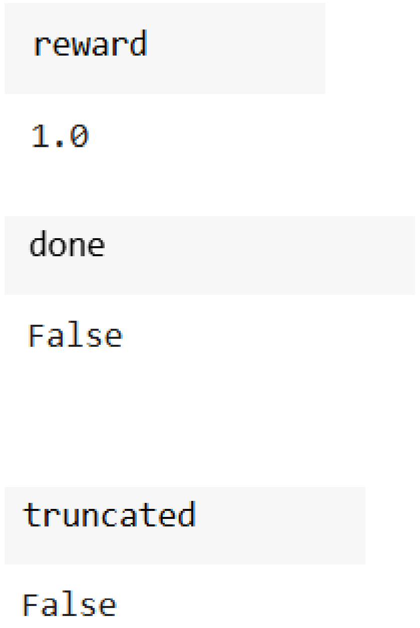
- Discrete(2) means that the possible actions are integers 0 and 1, which represent accelerating left or right.
- Since the pole is leaning toward the right (obs[2]>0), let’s accelerate the cart toward the right:

```
action = 1  # accelerate right
obs, reward, done, truncated, info = env.step(action)
obs
```

```
array([ 0.02727336,  0.18847767,  0.03625453, -0.26141977], dtype=float32)
```

Outputs of `step()` Method

- `reward`: in this environment, you get a reward of 1.0 at every step, no matter what you do, so the goal is to keep the episode running for as long as possible.
- `done`: `True` when the game is over. This will happen when the pole tilts too much, or goes off the screen, or after 200 steps (in this case, you have won).
- `truncated`: `True` when a game is interrupted early, for example by an environment wrapper that imposes a maximum number of steps per game.
  - Some RL algorithms treat truncated episodes differently from episodes finished normally (when `done` is `True`).



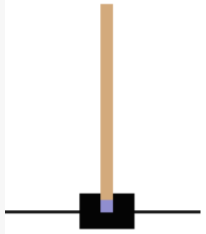
```
reward
```
```
1.0
```
```
done
```
```
False
```
```
truncated
```
```
False
```

Hardcoding a Policy

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs, info = env.reset(seed=episode)
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, truncated, info = env.step(action)
        episode_rewards += reward
        if done or truncated:
            break

    totals.append(episode_rewards)
```



```
np.mean(totals), np.std(totals), min(totals), max(totals)
```

```
(41.698, 8.389445512070509, 24.0, 63.0)
```