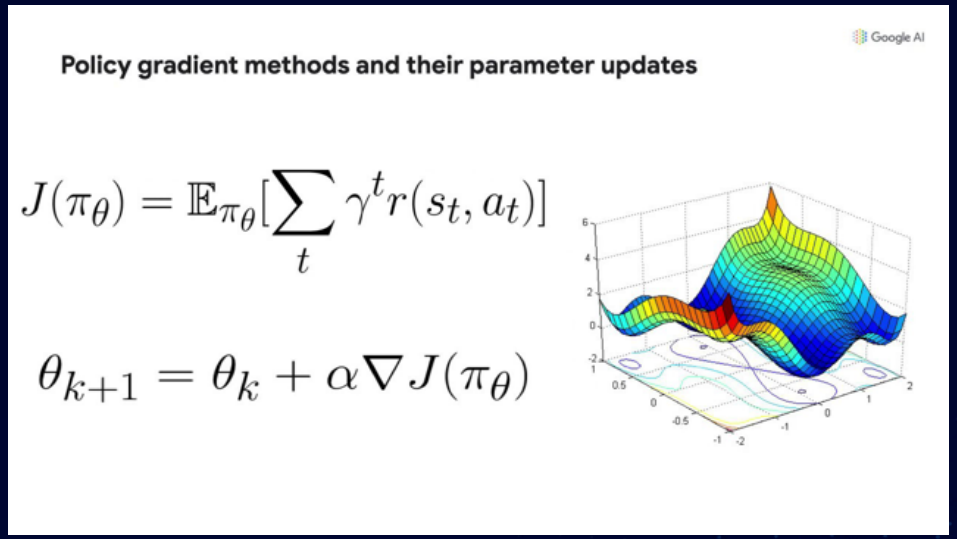


Policy Gradient Methods

Slides from or based on the 2nd edition of the book by Sutton and Barto

http://www.incompleteideas.net/book/the-book.html
See also Chapter 23 of the AIMA textbook
and
- Chapter 18 of Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (2nd edition)
By Aurélien Géron

Last modified on 2022/05/21
by f.maire@qut.edu.au

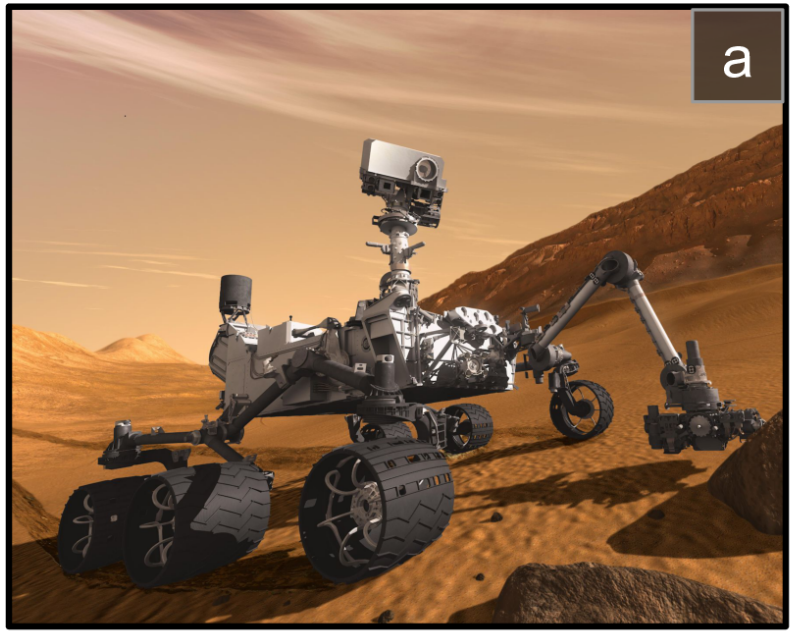
Outline

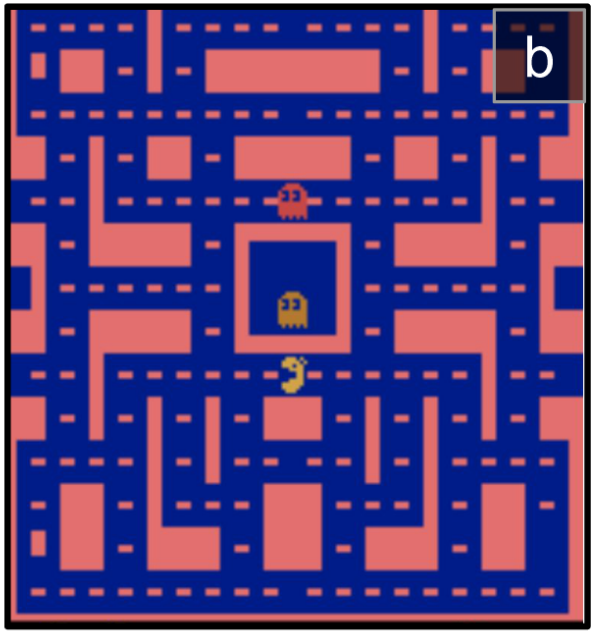
- RL refresher
- OpenAI gym
- Policy Gradient
- REINFORCE

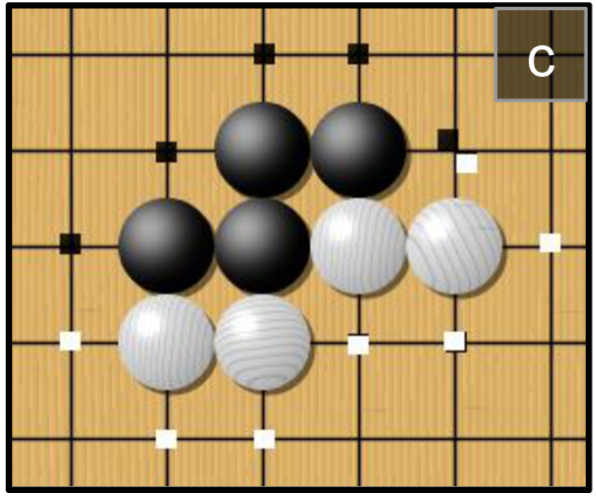
Learning to Optimize Rewards

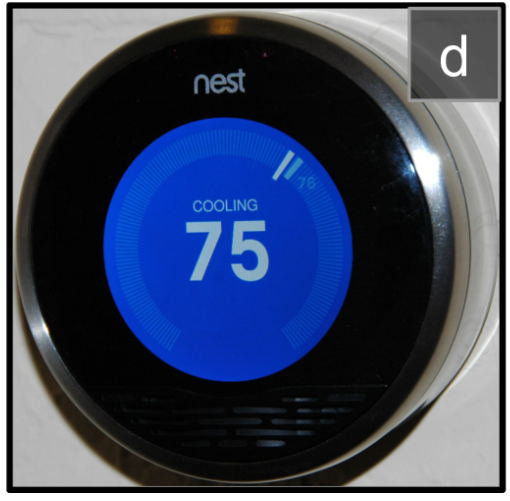
- In Reinforcement Learning, a software agent makes **observations** and takes **actions** within an **environment**, and in return it receives rewards.
- Its objective is to learn to act in a way that will **maximize** its **expected rewards** over time.

Refresher from last lecture









[a] controlling a robot

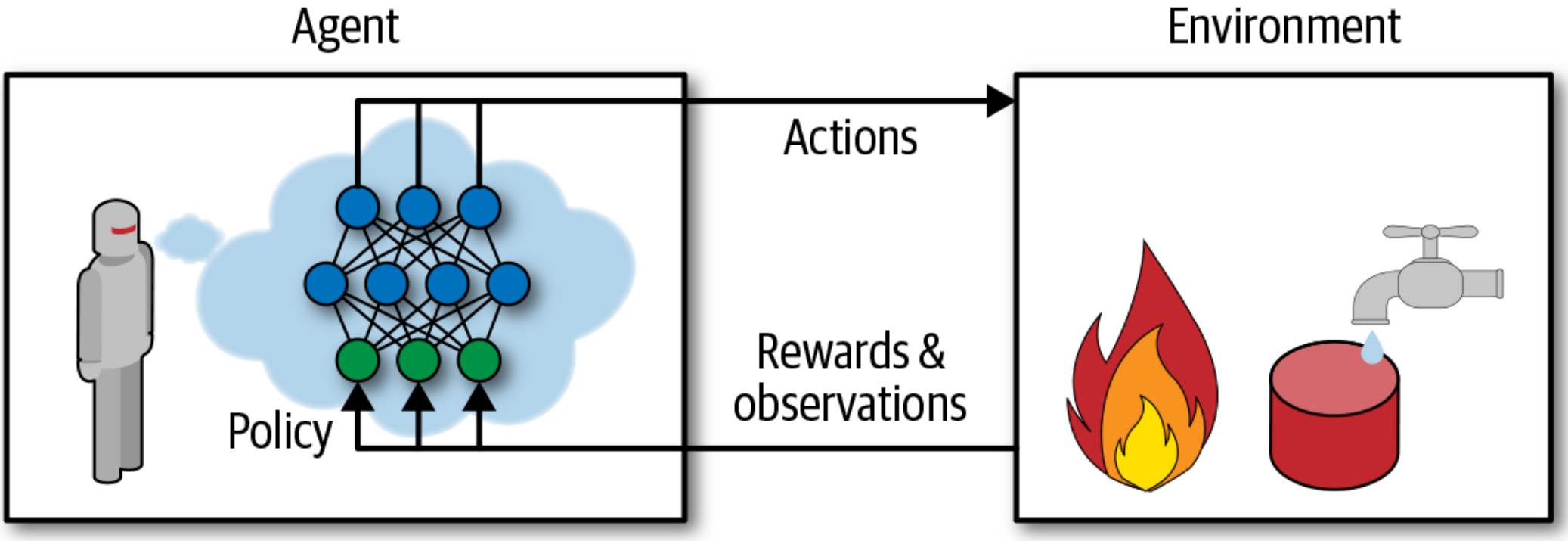
[b] controlling Ms. Pac-Man

[c] playing a board game

[d] smart thermostat

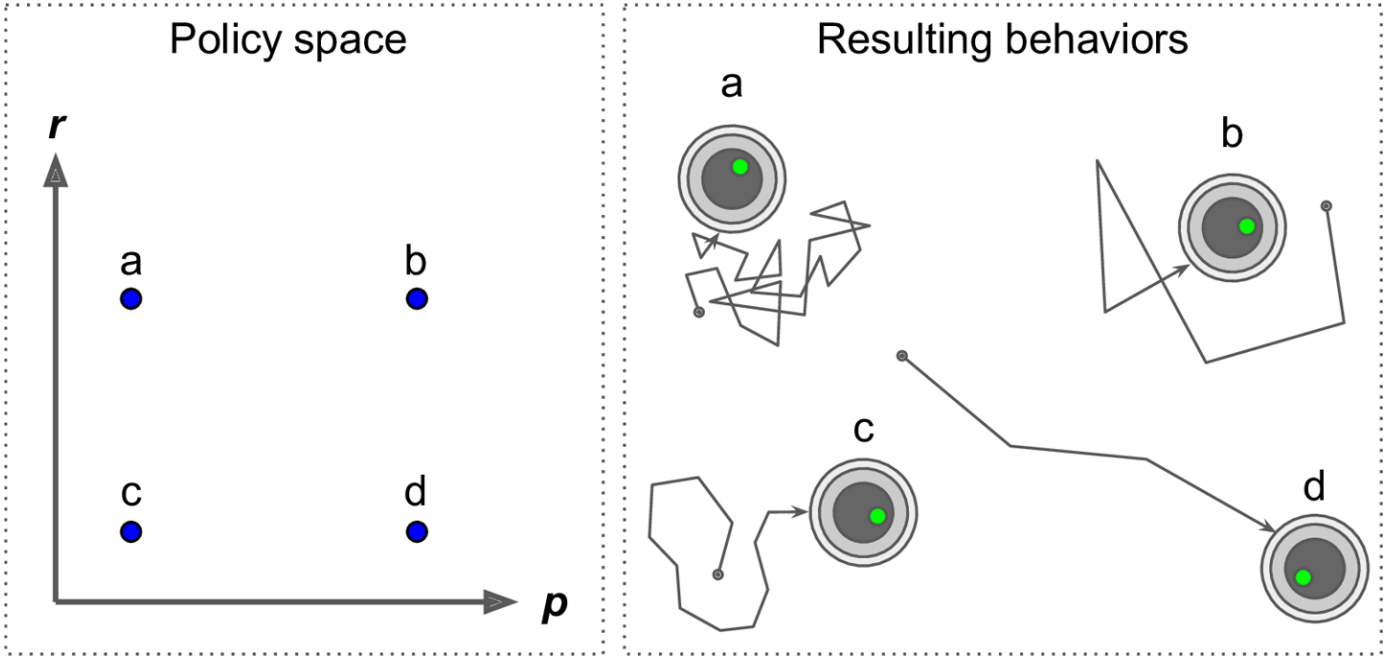
[e] stock market

Policy Search



The policy can be any algorithm you can think of, and it does not have to be deterministic. In fact, in some cases it does not even have to observe the environment!

Policy Search



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.

OpenAI Gym

- *OpenAI Gym* is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), so you can train agents, compare them, or develop new RL algorithms.

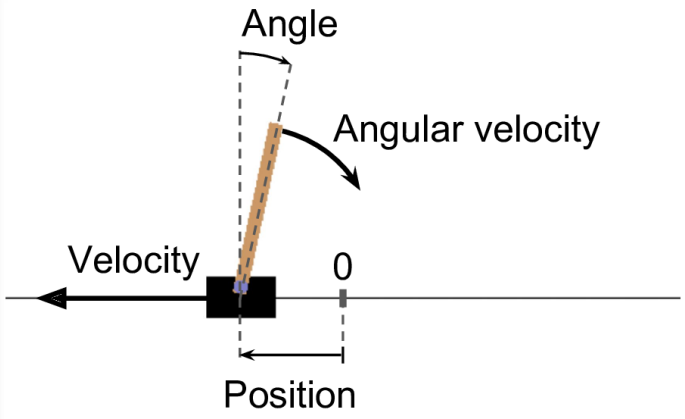
To install OpenAI Gym

$ python3 -m pip install -U gym

CartPole environment

```
frederic@frederic-HP-ZBook-17: ~

(base) frederic@frederic-HP-ZBook-17:~$ python
Python 3.7.4 (default, Aug 13 2019, 20:35:49)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import gym
>>> env = gym.make("CartPole-v1")
>>> obs = env.reset()
>>> obs
array([ 0.00547344,  0.04872556, -0.02911279, -0.00855014])
>>>
```



"CartPole-v1"is a 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it.
The observation vector 'obs' has 4 entries:

- the cart's horizontal position (0.0 = center),
- its velocity (positive means right),
- the angle of the pole (0.0 = vertical), and
- its angular velocity (positive means clockwise).

CartPole environment

- We can display this environment by calling its render() method
- The attribute action_space tells us what actions are possible:

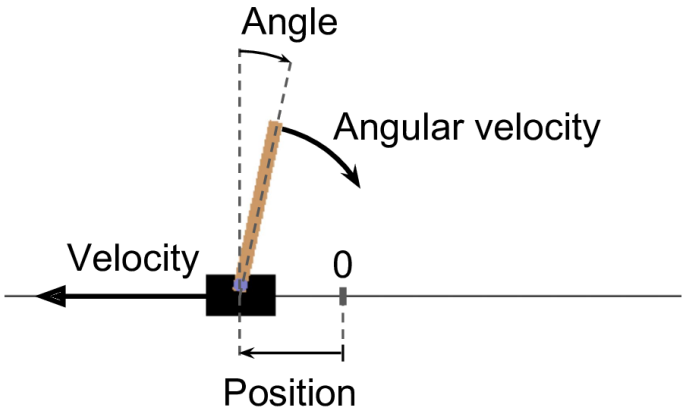
```
>>> env.action_space
Discrete(2)
```

- *Discrete(2)* means that the possible actions are integers 0 and 1, which represent accelerating left (0) or right (1)

CartPole environment

- Since the pole is leaning toward the right (obs[2] > 0), let's accelerate the cart toward the right

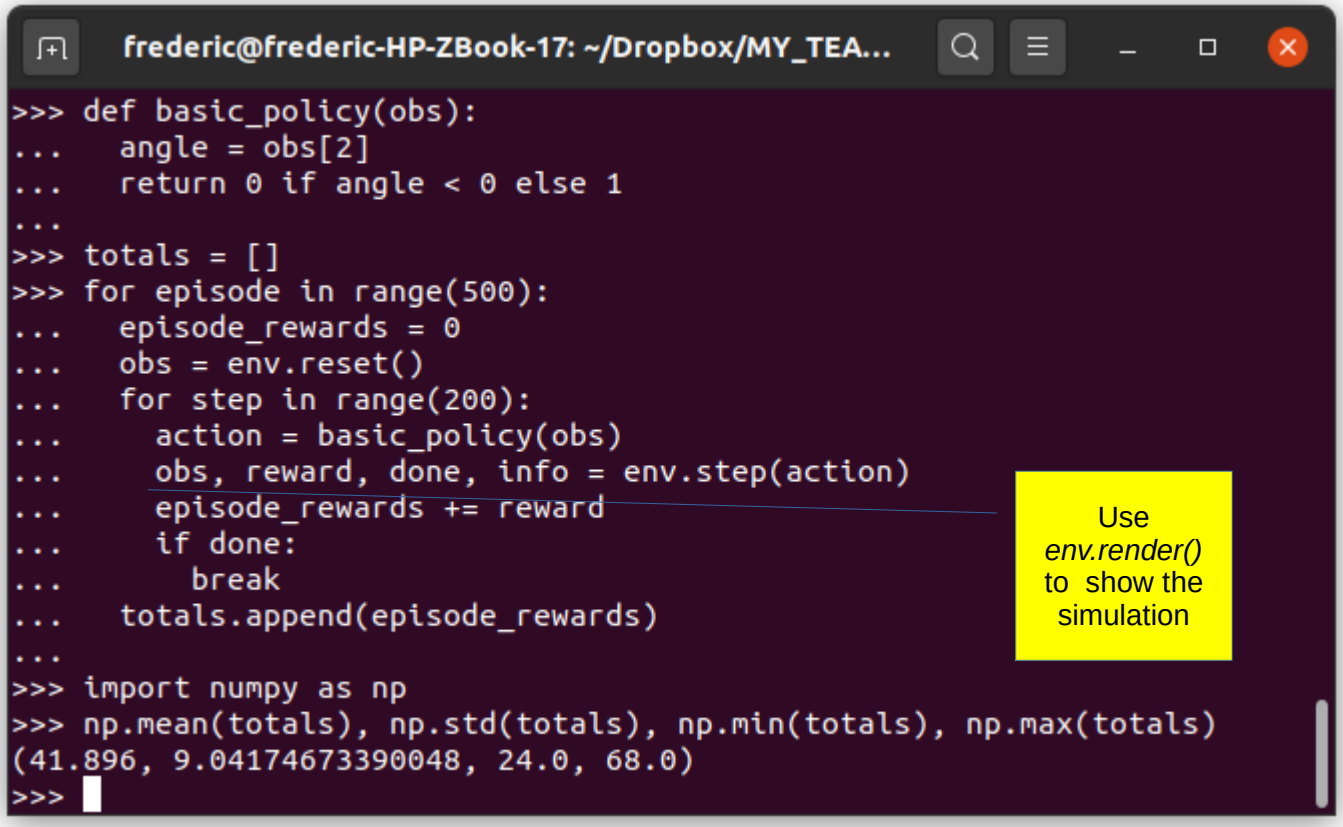
```
frederic@frederic-HP-ZBook-17: ~/Dropbox/MY_T...
>>>
>>> env.action_space
Discrete(2)
>>> action = 1  # accelerate right
>>> obs, reward, done, info = env.step(action)
>>> obs
array([ 0.03661636,  0.19827995,  0.02833881, -0.32907916])
>>> reward
1.0
>>> done
False
>>> info
{}
>>>
```



Once you have finished using an environment, you should call its close() method to free resources.

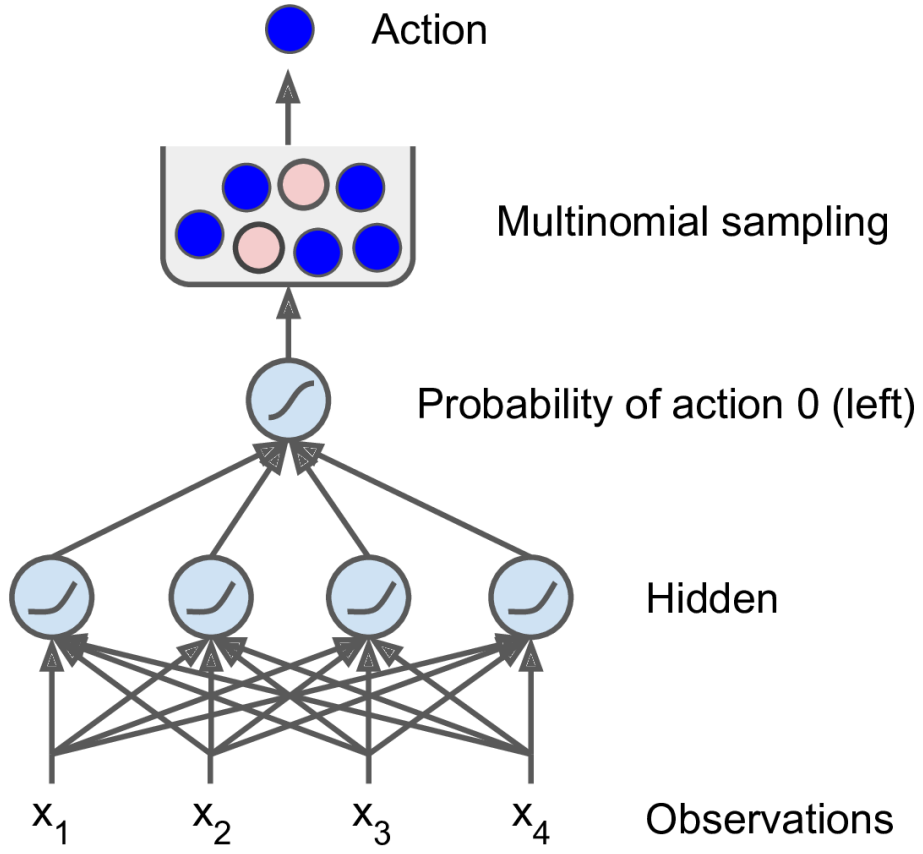
A simple policy

- Let's hardcode a simple policy that
  - accelerates left when the pole is leaning toward the left and
  - accelerates right when the pole is leaning toward the right.

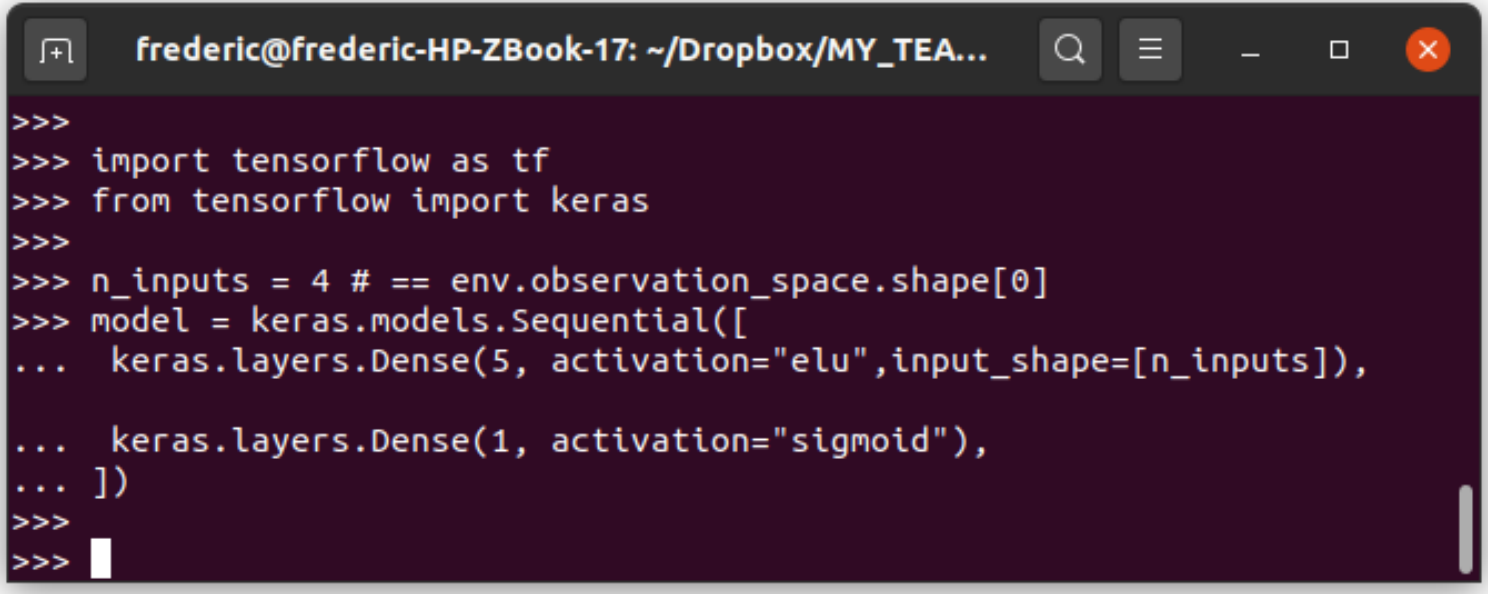


A Neural Network Policy

- In the case of the CartPole environment, there are just two possible actions (left or right), so we only need one output neuron.
- It will output the probability p of action 0 (left), and of course the probability of action 1 (right) will be 1 – p. For example, if it outputs 0.7, then we will pick action 0 with 70% probability, or action 1 with 30% probability.



A Neural Network Policy using tf.keras



Policy Gradients (PG)

- PG algorithms optimize the parameters of a policy by following the gradients toward higher rewards
- One popular class of PG algorithms, called REINFORCE algorithms, was introduced back in 1992 by Ronald Williams

Policy Gradients

We can simplify the notation without losing any meaningful generality by assuming that every episode starts in some particular (non-random) state s_0 . Then, in the episodic case we define performance as

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0),$$

Here, we consider methods for learning the policy parameter based on the gradient of some scalar performance measure *J* with respect to the policy parameter.

These methods seek to maximize performance, so their updates approximate gradient ascent in *J*

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)},$$

Policy Gradients

In policy gradient methods, the policy can be parameterized in any way, as long as $\pi(a|s,\boldsymbol{\theta})$ is differentiable with respect to its parameters, that is, as long as $\nabla\pi(a|s,\boldsymbol{\theta})$ (the column vector of partial derivatives of $\pi(a|s,\boldsymbol{\theta})$ with respect to the components of $\boldsymbol{\theta}$) exists and is finite for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, and $\boldsymbol{\theta} \in \mathbb{R}^{d'}$.

If the action space is discrete and not too large, then a natural and common kind of parameterization is to form parameterized numerical preferences $h(s,a,\boldsymbol{\theta}) \in \mathbb{R}$ for each state–action pair. The actions with the highest preferences in each state are given the highest probabilities of being selected, for example, according to an exponential soft-max distribution:

$$\pi(a|s,\boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}},$$

Policy Gradients

The action preferences themselves can be parameterized arbitrarily. For example, they might be computed by a deep artificial neural network (ANN). Or the preferences could simply be linear in features,

$$h(s, a, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}(s, a), \qquad \text{using feature vectors } \mathbf{x}(s, a) \in \mathbb{R}^{d'}$$

The Policy Gradient Theorem

- The policy gradient theorem for the episodic case establishes that

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}),$$

The distribution μ is the state distribution under π

The Policy Gradient Theorem (proof)

$$\nabla v_\pi(s) = \nabla \left[\sum_a \pi(a|s) q_\pi(s,a) \right], \quad \text{for all } s \in \mathcal{S}$$

$$= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a) \Big] \qquad \text{(product rule of calculus)}$$

$$= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s',r} p(s',r|s,a) \big(r + v_\pi(s') \big) \Big]$$

$$= \sum_a \Big[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \nabla v_\pi(s') \Big]$$

The Policy Gradient Theorem (proof)

$$= \sum_a \left[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \nabla v_\pi(s') \right]$$

Last equation of the previous slide

$$= \sum_a \Bigg[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \qquad \text{(unrolling)}$$

$$\sum_{a'} \left[\nabla \pi(a'|s') q_\pi(s',a') + \pi(a'|s') \sum_{s''} p(s''|s',a') \nabla v_\pi(s'') \right] \Bigg]$$

$$= \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \Pr(s \to x, k, \pi) \sum_a \nabla \pi(a|x) q_\pi(x,a),$$

after repeated unrolling, where $\Pr(s \to x, k, \pi)$ is the probability of transitioning from state s to state x in k steps under policy π.

The Policy Gradient Theorem (proof)

$\eta(s)$ denote the number of time steps spent, on average, in state s in a single episode.

$$
\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \nabla v_\pi(s_0) \\
&= \sum_s \left(\sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi) \right) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) \\
&\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a)
\end{aligned}
$$

REINFORCE

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta})$$

$$= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta}) \right] .$$

REINFORCE

$$\nabla J(\boldsymbol{\theta}) \propto \mathbb{E}_\pi \left[\sum_a \pi(a|S_t, \boldsymbol{\theta}) q_\pi(S_t, a) \frac{\nabla \pi(a|S_t, \boldsymbol{\theta})}{\pi(a|S_t, \boldsymbol{\theta})} \right]$$

$$= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right] \qquad \text{(replacing } a \text{ by the sample } A_t \sim \pi\text{)}$$

$$= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \right], \qquad \text{(because } \mathbb{E}_\pi[G_t|S_t, A_t] = q_\pi(S_t, A_t)\text{)}$$

REINFORCE update

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}.$$

REINFORCE (pseudo code)

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Summary - 1

- Last time, we focused on ***action-value*** methods. That is, methods that learn action values and then use them to determine action selections.
- Today, we considered methods that learn a parameterized policy that enables actions to be taken without consulting action-value estimates.

Summary - 2

- If the **state-value** function is also used to assess or criticize the policy's action selections, then the value function is called a ***critic*** and the policy is called an ***actor***
- The overall method is then called an ***actor–critic*** method