

# Deep Reinforcement Learning - 1

Kalle Prorok, Umeå 2020

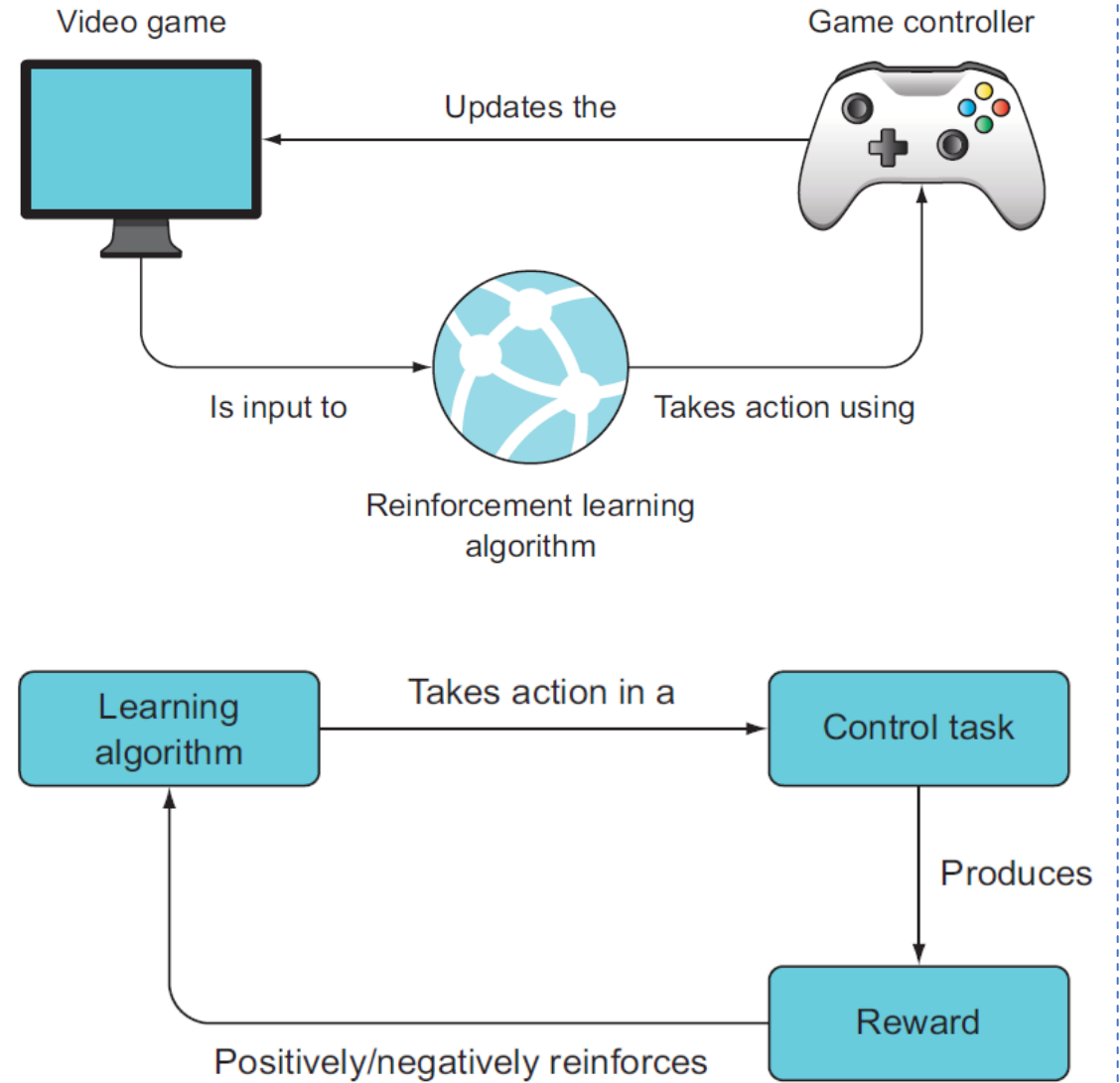
Parts based on the Manning book

Deep Reinforcement Learning in Action

By Alexander Zai and Brandon Brown

And the Sutton-Barto book

# *Reinforcement learning*



# Resources



5 Intro videos at course site



Sutton - Barto free online-book



the DRL book's GitHub  
repository: <http://mng.bz/JzKp>

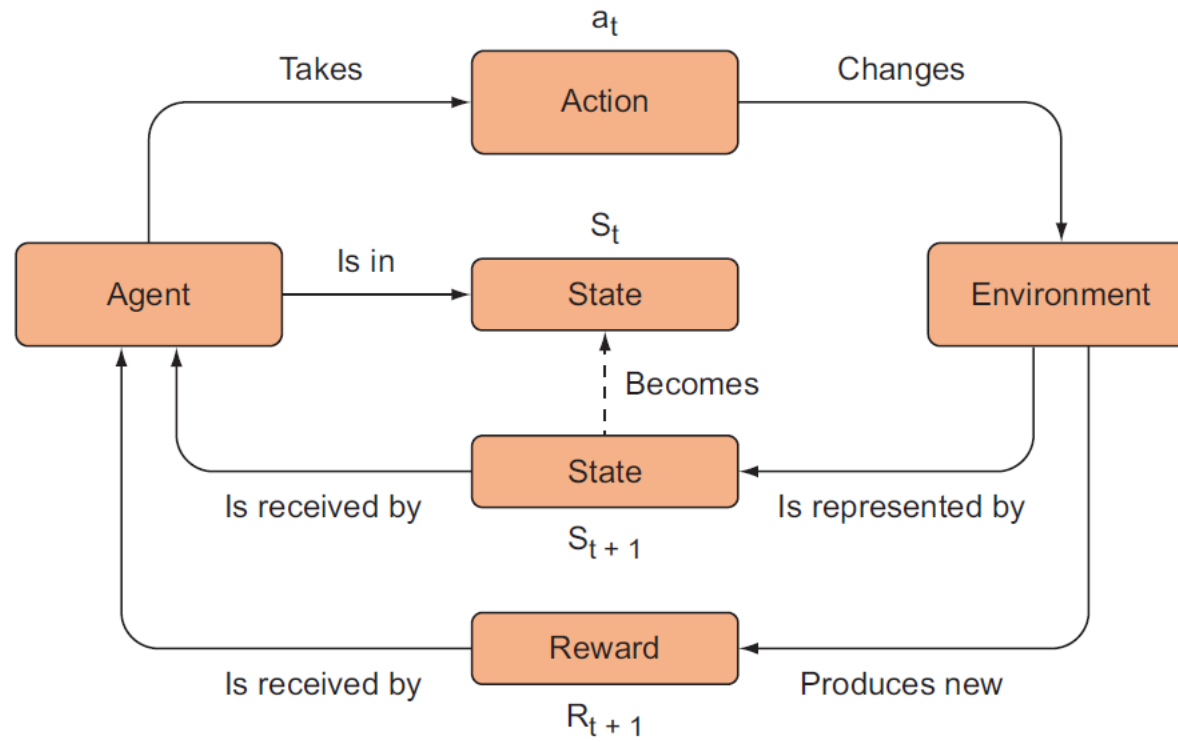
# *Dynamic programming*

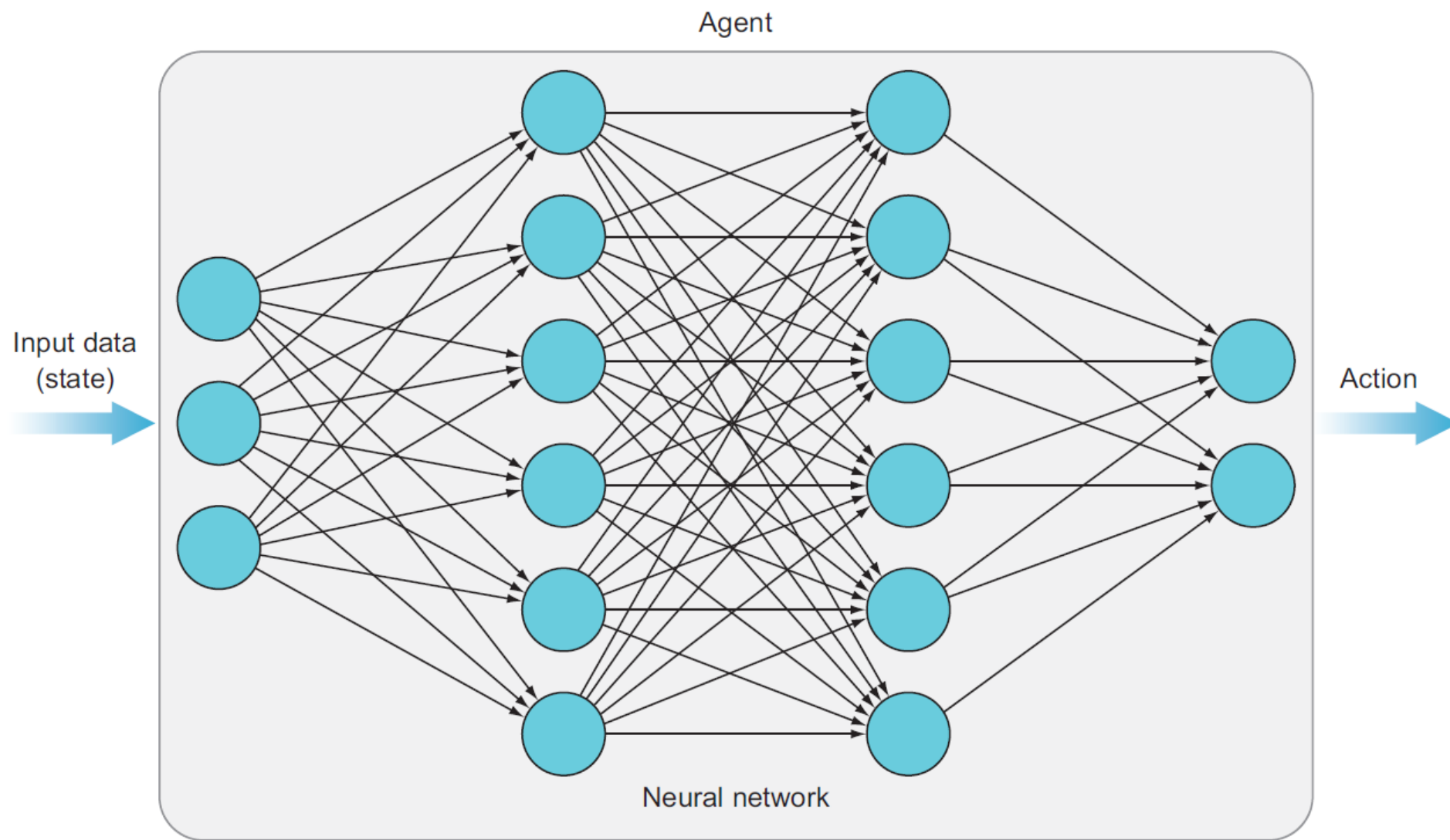
- *goal decomposition* as it solves complex high-level problems by decomposing them into smaller and smaller subproblems until it gets to a simple subproblem that can be solved
- train a robot vacuum to move from one room in a house to its dock
  - stay in/exit this room?
  - move toward the door” or “move away from the door
- You need to know your house extremely well

# *Monte Carlo*

- The trial and error strategy
- go to a party at a house that you've never been to before, you might have to look around until you find the bathroom on your own

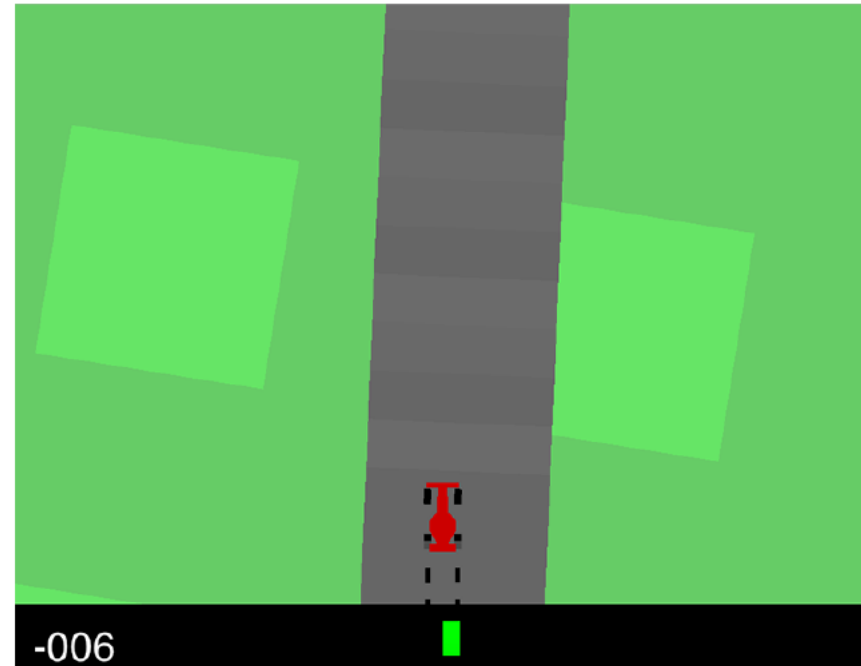
# Agent





# Open AI Gym

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

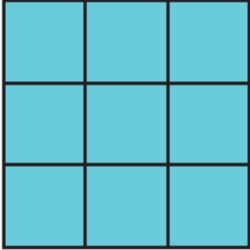
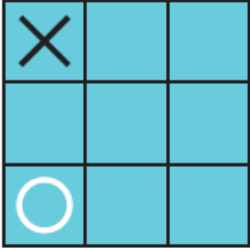
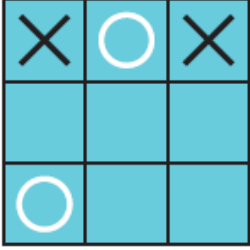




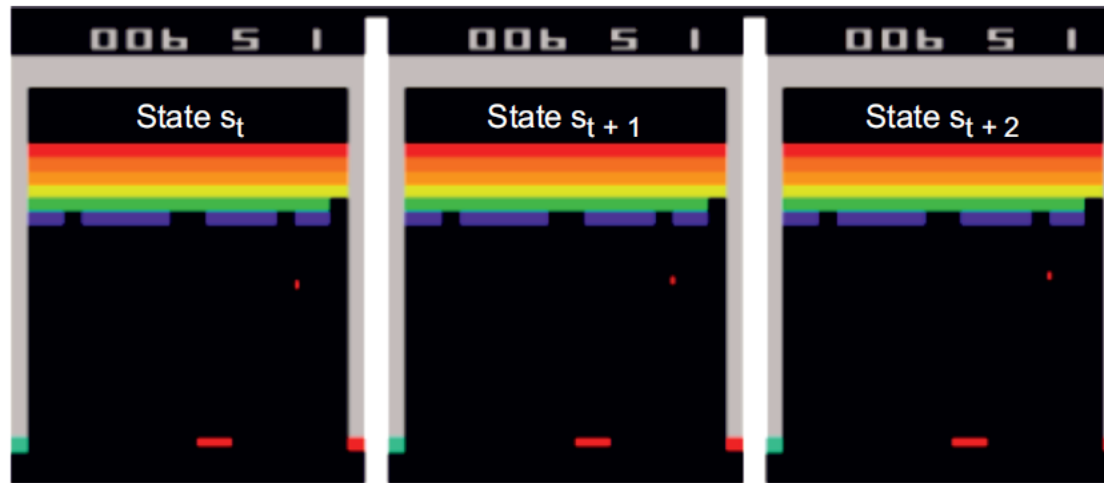
# An action lookup table for Tic-Tac-Toe

there are 255,168 valid board positions..

Game play lookup table

Key Current state	Value Action to take
	Place X in top left
	Place X in top right
	Place X in bottom right
...	

# DeepMind's DQN algorithm for Atari games

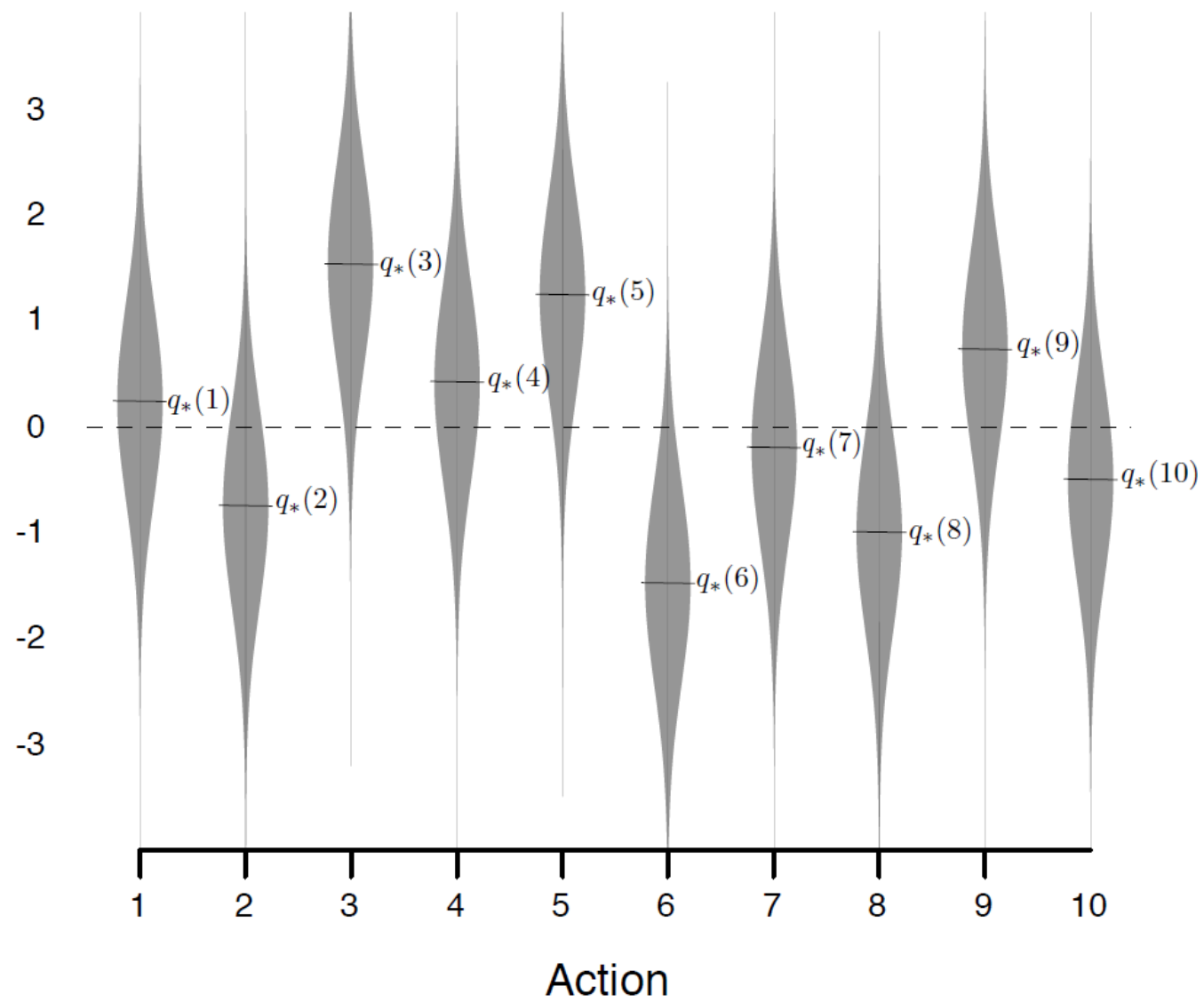


Four (three here)  $84 \times 84$  grayscale images at each step, which would lead to  $256^{28228}$  unique game states. DQN CNN have 1792 parameters

# *The multi-arm bandit problem*

- 10 slot machines with a sign that says “Play for free! Max payout \$10!
- each have a different average payout
  - the reward at each play is probabilistic
  - so try to figure out which one gives the most rewards on average
- ” play a few times, choosing different levers(a) and observing our rewards  $R$  for each action. Then we want to only choose the lever with the largest observed average reward.”

Reward  
distribution



expected reward at play k for taking an action ( $a$ )

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

```
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

# 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

Math	Pseudocode
$\forall a_i \in A_k$	def get_best_action(actions, history): exp_rewards = [exp_reward(action, history) for action in actions]
$a^* = \operatorname{argmax}_a Q_k(a_i)$	return argmax(exp_rewards)

The following listing shows it as legitimate Python 3 code.

### Listing 2.1 Finding the best actions given the expected rewards in Python 3

```
def get_best_action(actions):
    best_action = 0
    max_action_value = 0
    for i in range(len(actions)):
        cur_action_value = get_action_value(actions[i])
        if cur_action_value > max_action_value:
            best_action = i
            max_action_value = cur_action_value
    return best_action
```

←  
**Loops through all possible actions**

←  
**Gets the value of the current action**

a *greedy* (or exploitation) method!

# *Epsilon-greedy strategy*

- with a probability,  $\epsilon$ , we will choose an action,  $a$ , at random, and the rest of the time (probability  $1 - \epsilon$ ) 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

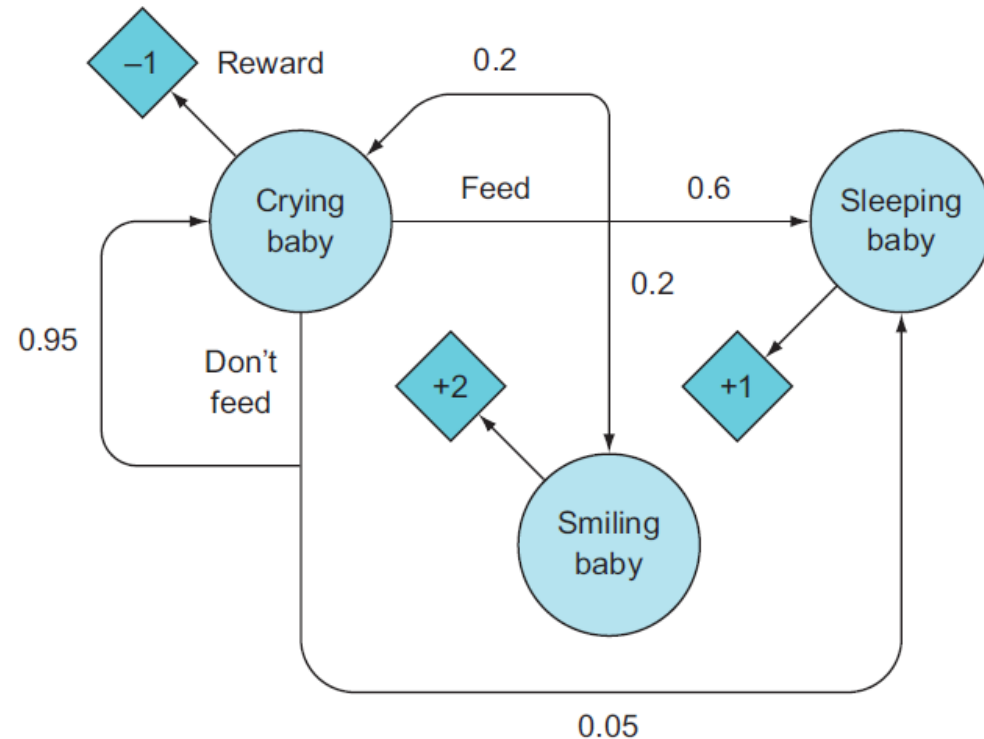
$$\Pr(A) = \frac{e^{Q_k(A)/\tau}}{\sum_{i=1}^n e^{Q_k(i)/\tau}}$$

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

$\tau$  is a parameter called *temperature* that scales the probability distribution of actions. Usually reduced over time, “cooling metal”.

# 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):



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

maximize the expected discounted return  $G_t$

# The Bellman equation for $v$

$$\begin{aligned}v_{\pi}(s) &\doteq \mathbb{E}_{\pi}[G_t \mid S_t = s] \\&= \mathbb{E}_{\pi}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \mid s, a) \left[ r + \gamma \mathbb{E}_{\pi}[G_{t+1} \mid S_{t+1} = s'] \right] \\&= \sum_a \pi(a|s) \sum_{s', r} p(s', r \mid s, a) \left[ r + \gamma v_{\pi}(s') \right], \quad \text{for all } s \in \mathcal{S},\end{aligned}$$

# *Contextual bandits*

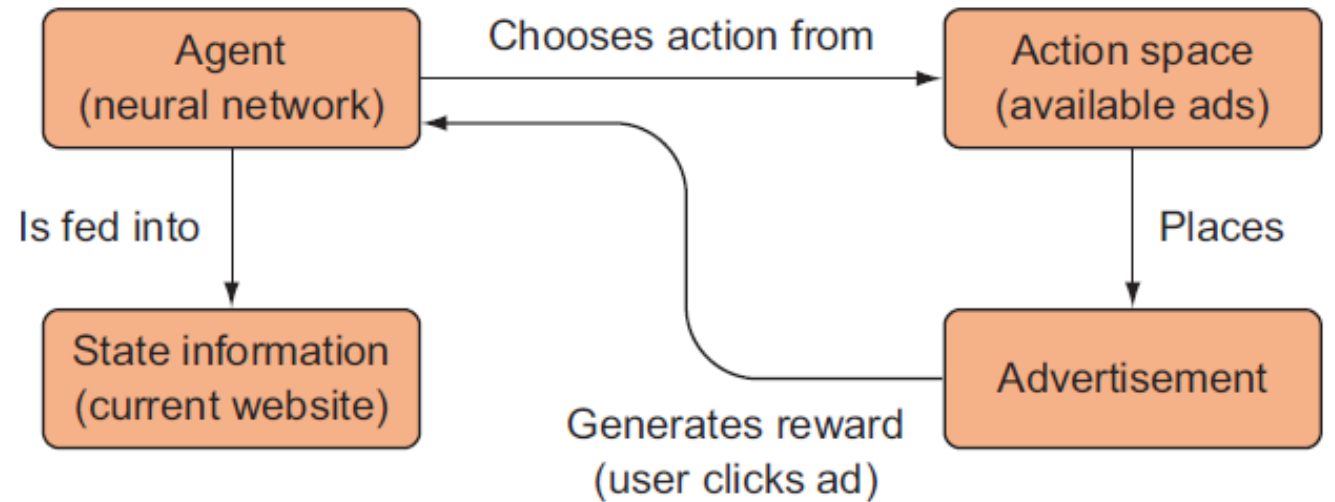
- Example: advertisement placement
  - maximize the probability that you will click ads depending on what site you visit
  - Let's say we manage 10 e-commerce websites, each focusing on selling a different broad category of retail items such as computers, shoes, jewelry, etc.

We want to increase sales by referring customers who shop on one of our sites to another site that they might be interested in. When a customer checks out on a particular site in our network, we will display an advertisement to one of our other sites in hopes they'll go there and buy something else.

Alternatively, we could place an ad for another product on the same site. Our problem is that we don't know which sites we should refer users to. We could try placing random ads, but we suspect a more targeted approach is possible.

## *state spaces*

- we know the user is buying something on a particular site, which may give us some information about that user's preferences and could help guide our decision about which ad to place



# *Building networks with PyTorch*

*A simple two-layer neural network with an Optimizer:*

```
model = torch.nn.Sequential(  
    torch.nn.Linear(10, 150),  
    torch.nn.ReLU(),  
    torch.nn.Linear(150, 4),  
    torch.nn.ReLU(),  
)
```

*A training loop:*

```
for step in range(100):  
    y_pred = model(x)  
    loss = loss_fn(y_pred, y_correct)  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()
```

*Our own Python class:*

```
from torch.nn import Module, Linear  
class MyNet(Module):  
    def __init__(self):  
        super(MyNet, self).__init__()  
        self.fc1 = Linear(784, 50)  
        self.fc2 = Linear(50, 10)  
    def forward(self, x):  
        x = F.relu(self.fc1(x))  
        x = F.relu(self.fc2(x))  
        return x  
model = MyNet()
```