

Modeling Markets, Pandemics, and Peace: The Mathematics of Multi-Agent Systems

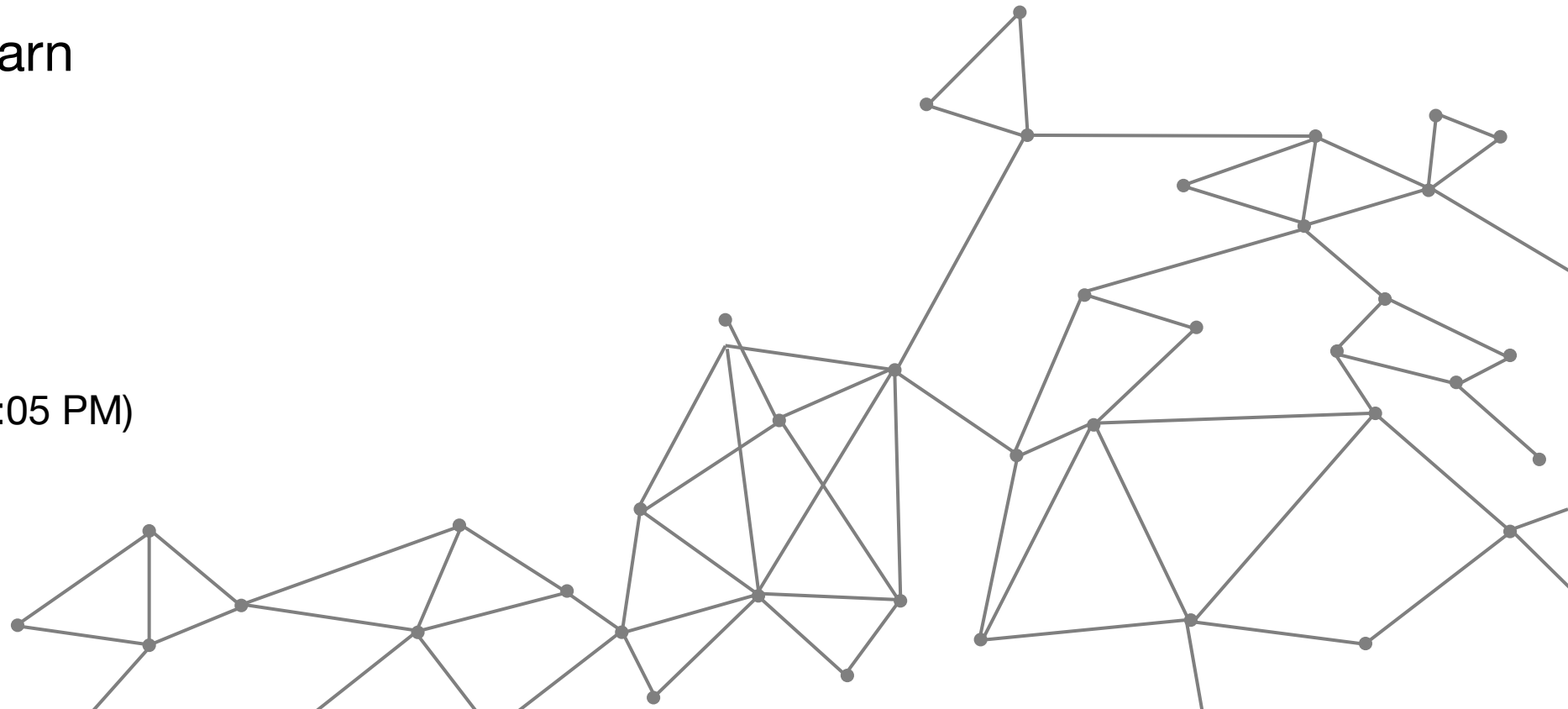


Lecture 2

How computers learn

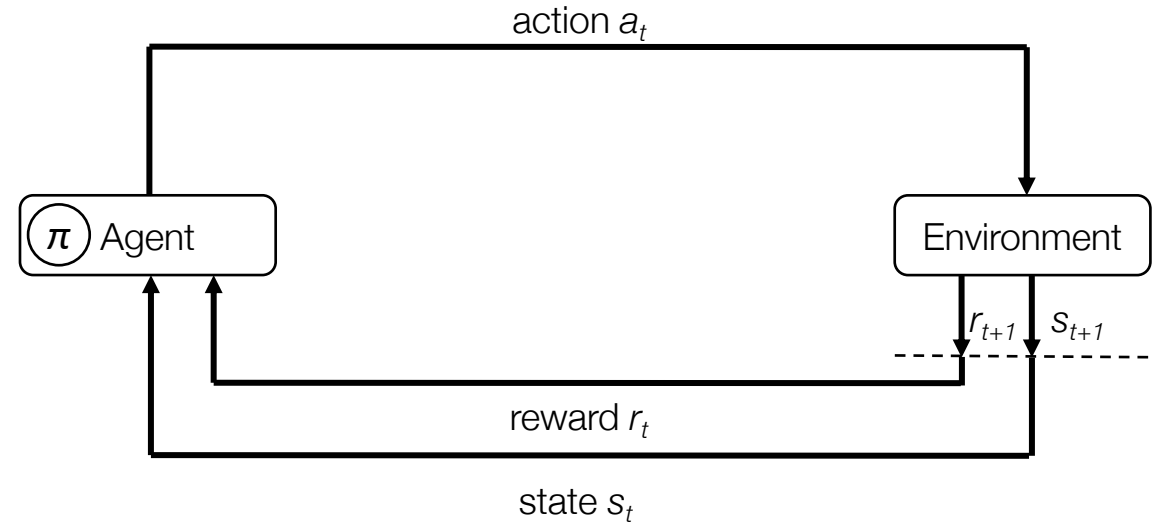
MIT HSSP

July 9th, 2022 (Starting 1:05 PM)



Recap of RL

- Policy $\pi: S \rightarrow A$ such that $a_t = \pi(s_t)$
- Agent's goal = find policy that maximizes total reward

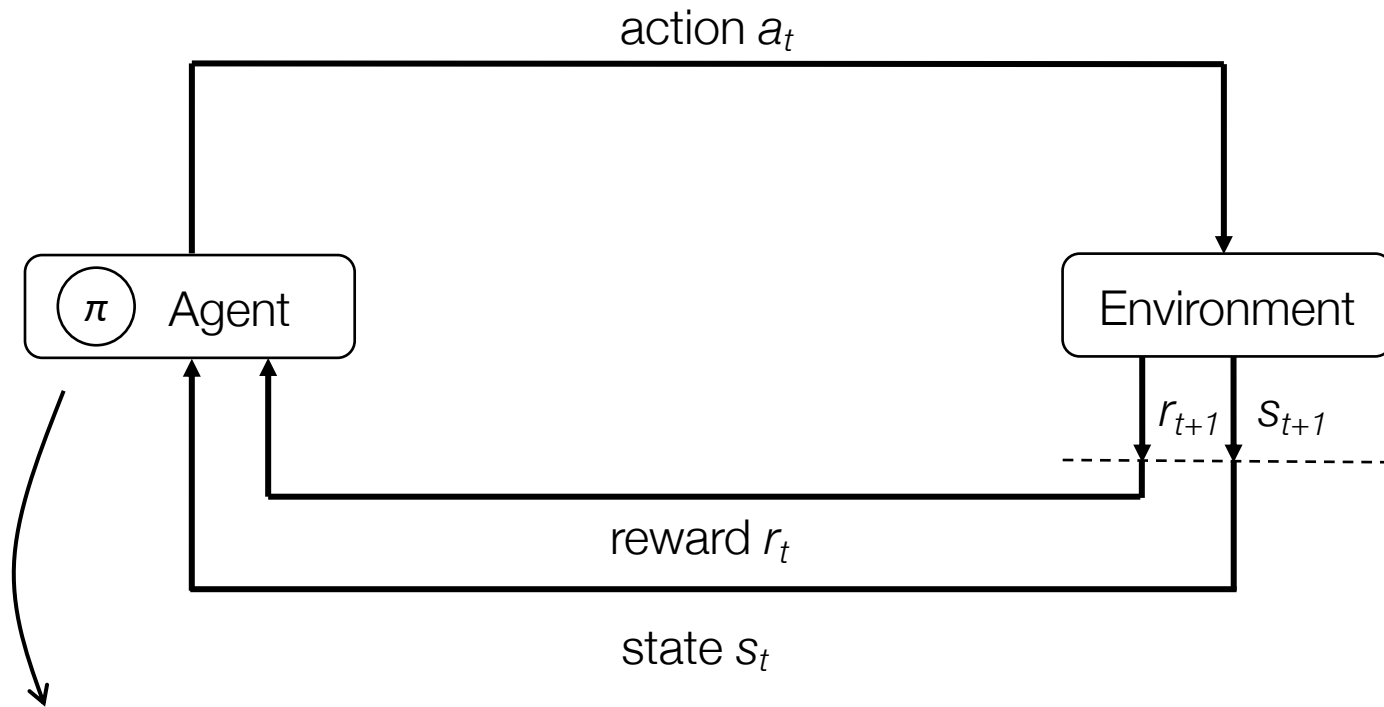


- Our life is easier when states are **Markovian** (the future depends only on the current state and not the past)

- Bellman's equation:
$$v_{\pi}(s_t) = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} R_{t+k+1} | S = s_t \right] = r(s_t, a_t) + \mathbb{E}[v_{\pi}(s_{t+1})]$$

- Optimal value function:
$$v_*(s) = \text{maximize}_{\pi}(v_{\pi}(s))$$

Today

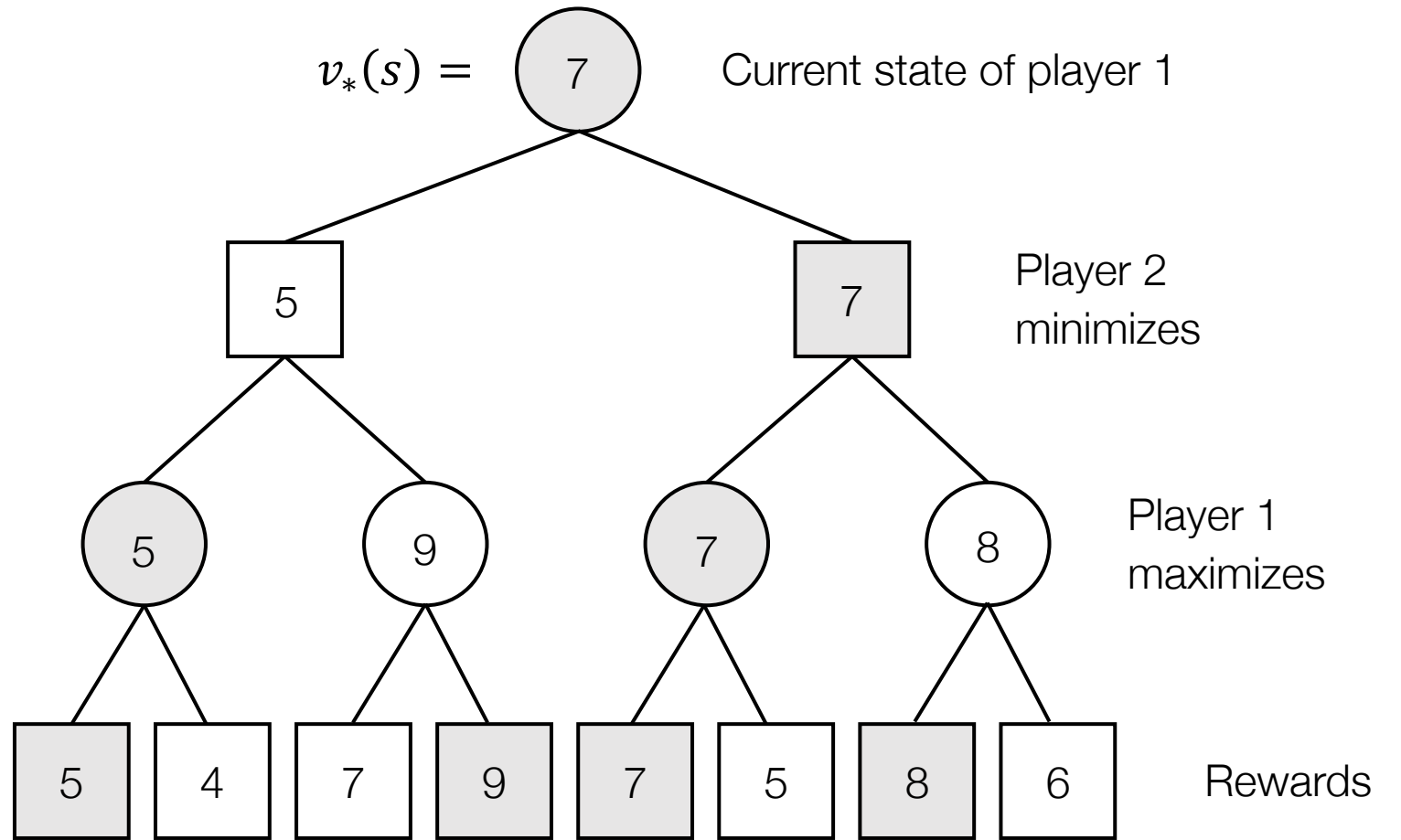


1. How do we represent and learn π using a computer?

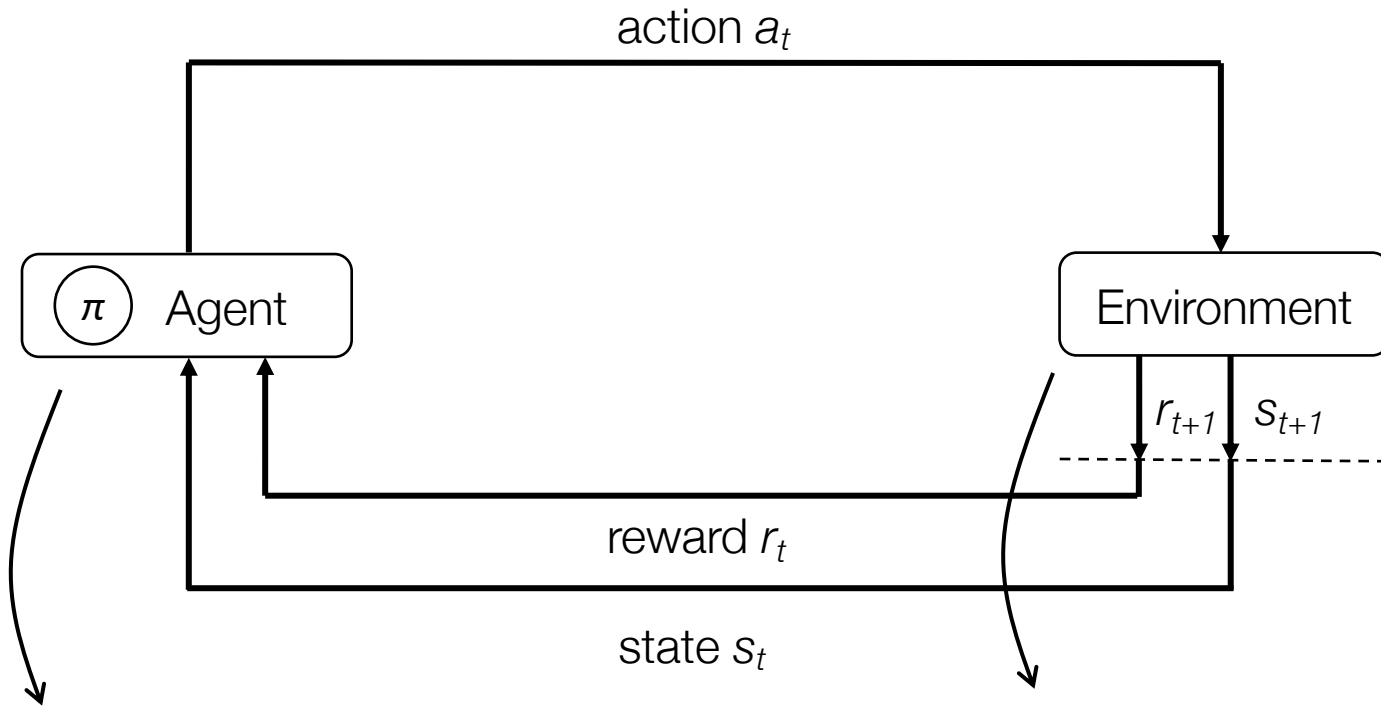
Estimating the optimal value function

Bellman's equation

“If I know the shortest path from Boston to DC runs through New York, then once I get to New York, I should just follow the shortest path from New York to DC.”



Today



1. How do we represent and learn π using a computer?

2. How do we figure out the behavior of the environment if it is not given to us?

$$v_{\pi}(s_t) = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} R_{t+k+1} | S = s_t \right]$$

$$v_*(s) = \text{maximize}_{\pi}(v_{\pi}(s))$$

There are two dominant methods in reinforcement learning:

(a) Learn π directly, trying out different policies to maximize reward (policy gradient method)

(b) Learn the state-action value function Q (Q-learning)

What is machine learning?



Low-complexity organism

```
pi_net = nn.Sequential(  
    nn.Linear(obs_dim, 64),  
    nn.Tanh(),  
    nn.Linear(64, 64),  
    nn.Tanh(),  
    nn.Linear(64, act_dim)  
)
```

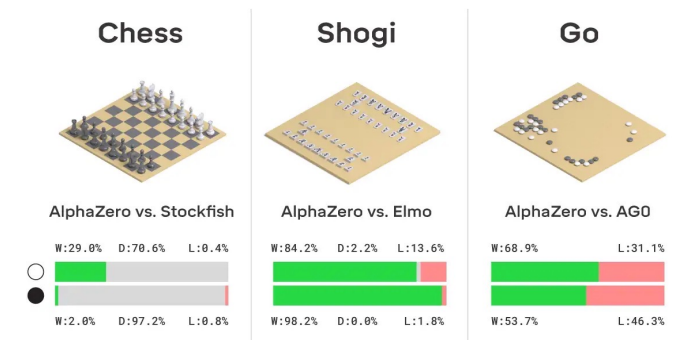
Low-complexity program

3.5 billion years of
natural selection



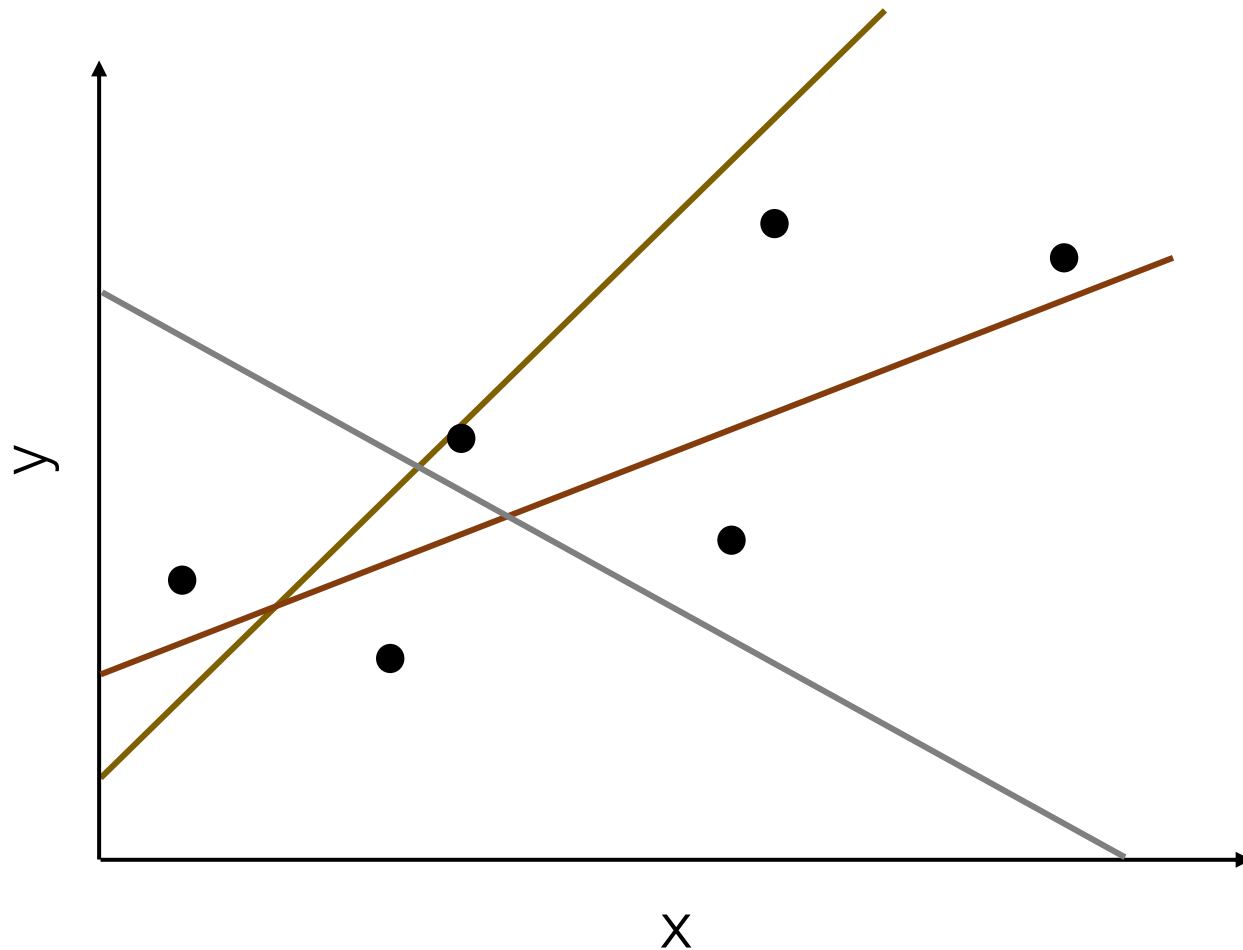
High-complexity organism

Machine learning



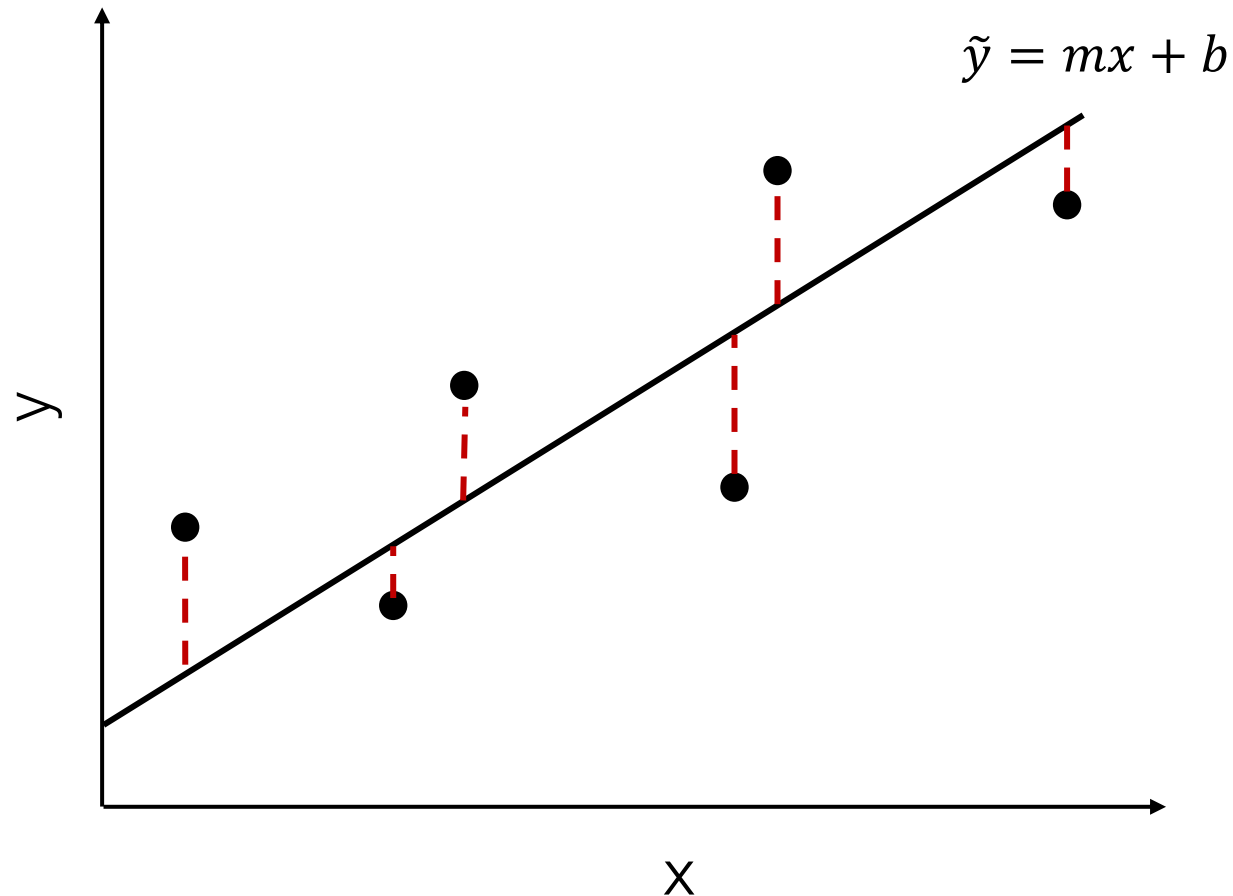
High-complexity program

The simplest example of machine learning



Which is the
line of best fit?

The simplest example of machine learning



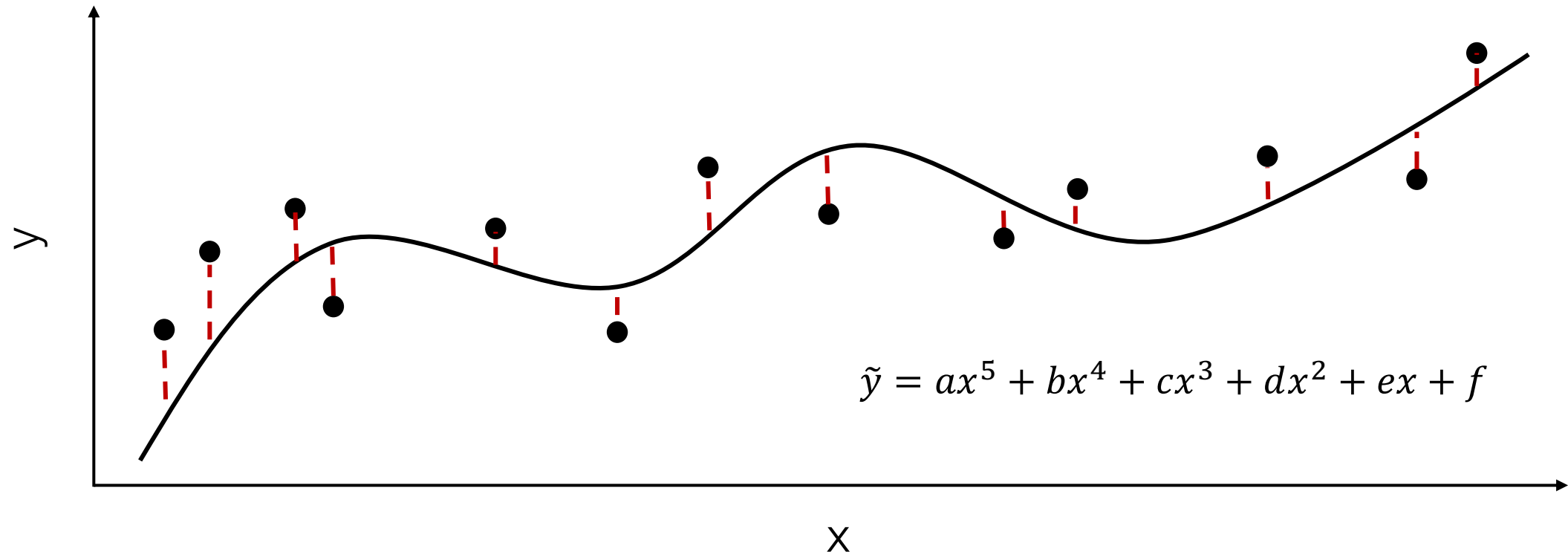
We want to find a line, defined by m and b , that “best fit” our data.

Define “best fit” as the line that **minimizes** the **average squared length of the dotted lines**:

$$L = \frac{1}{n} \sum_{i=1}^n (y_i - \tilde{y}_i)^2 = \frac{1}{n} \sum_{i=1}^n (y_i - mx_i - b)^2$$

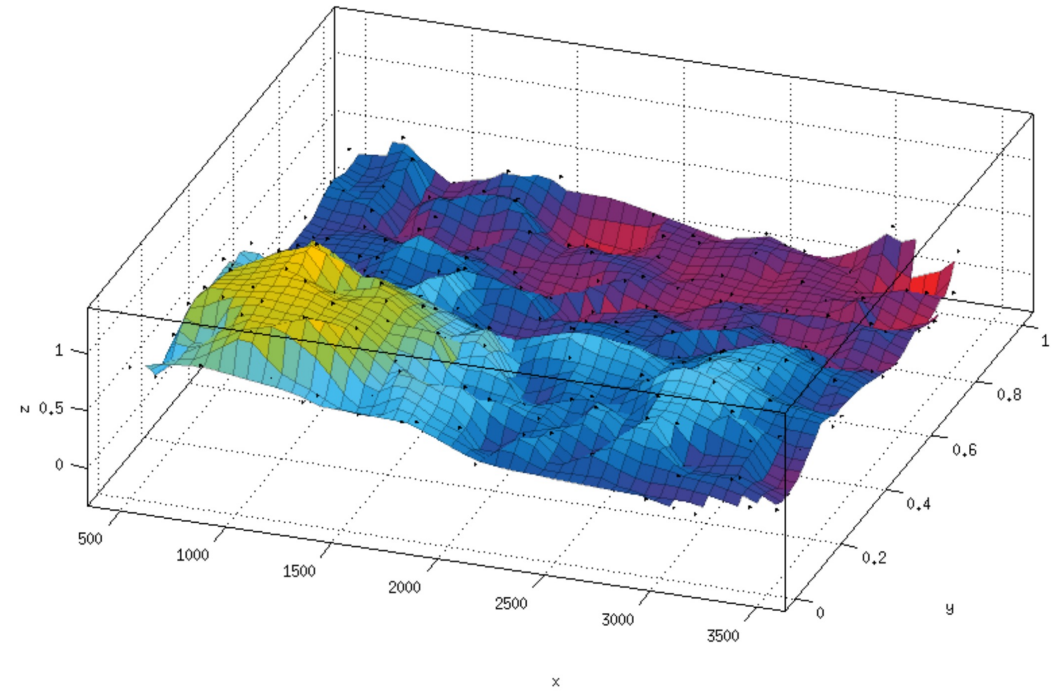
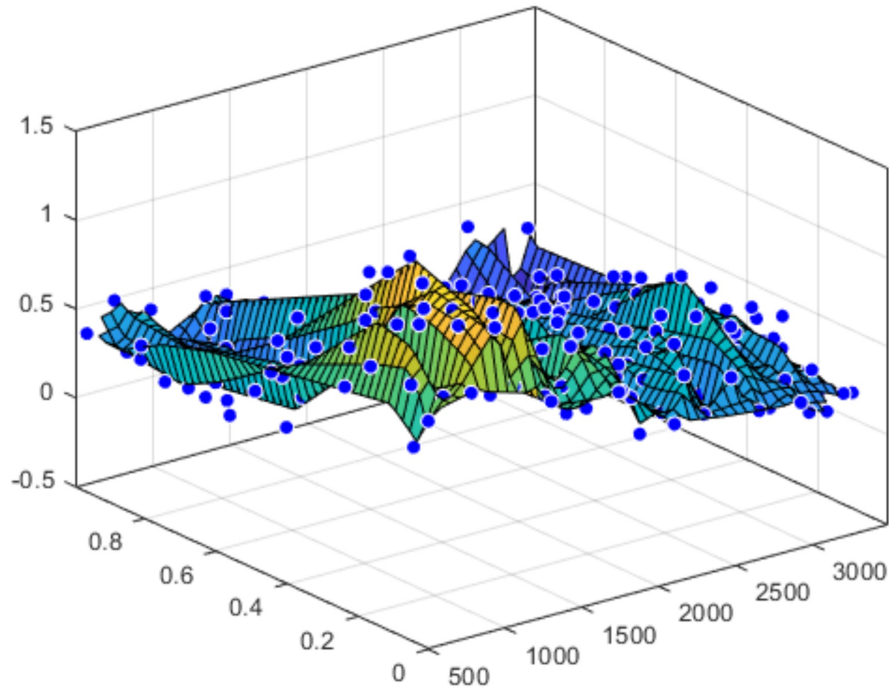
“Loss”

Fitting more complicated curves



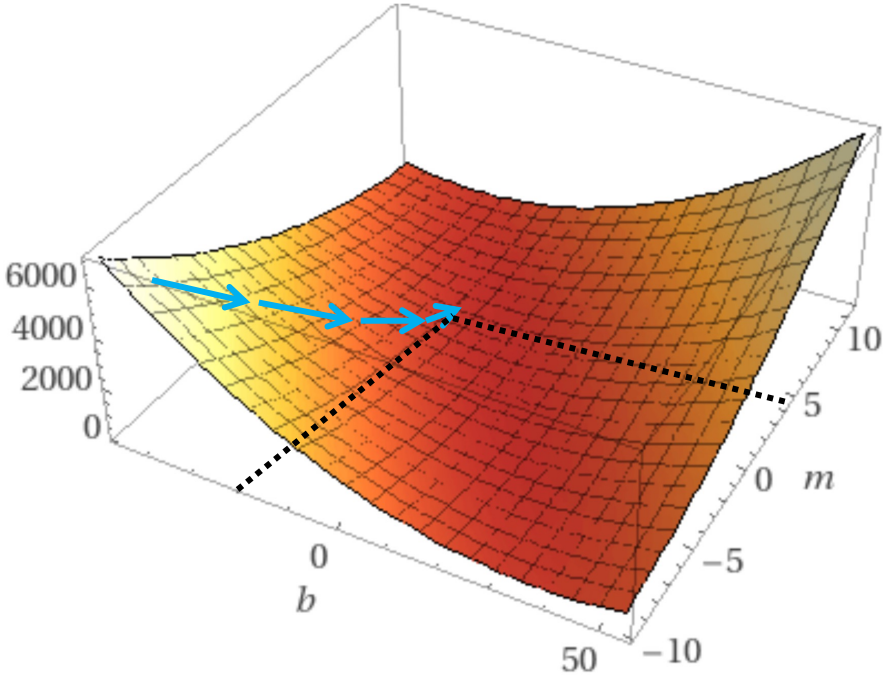
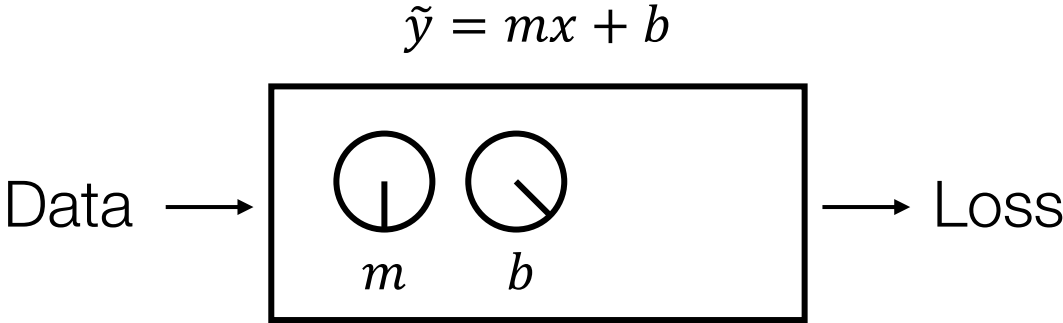
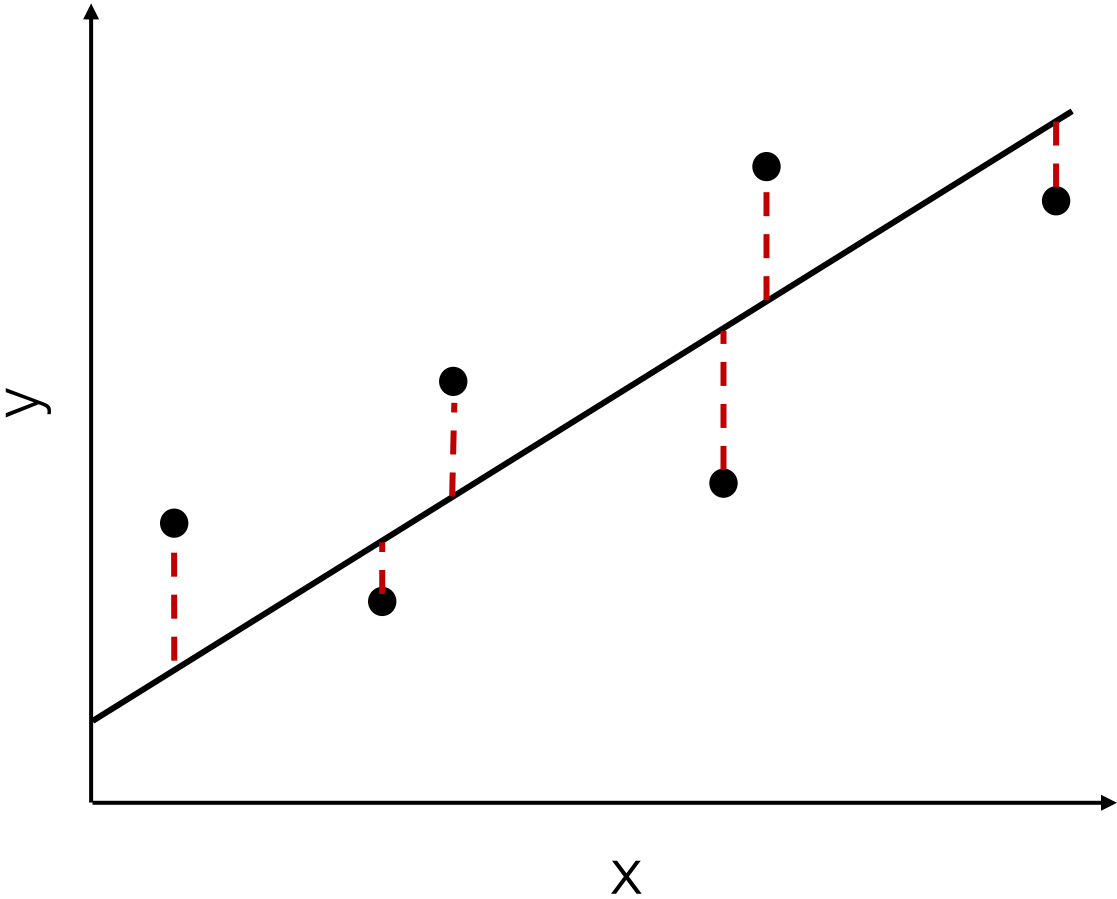
Find parameters $\{a, b, c, d, e, f\}$ such that $L = \frac{1}{n} \sum_{i=1}^n (y_i - \tilde{y}_i)^2$ is minimized

Fitting more complicated curves

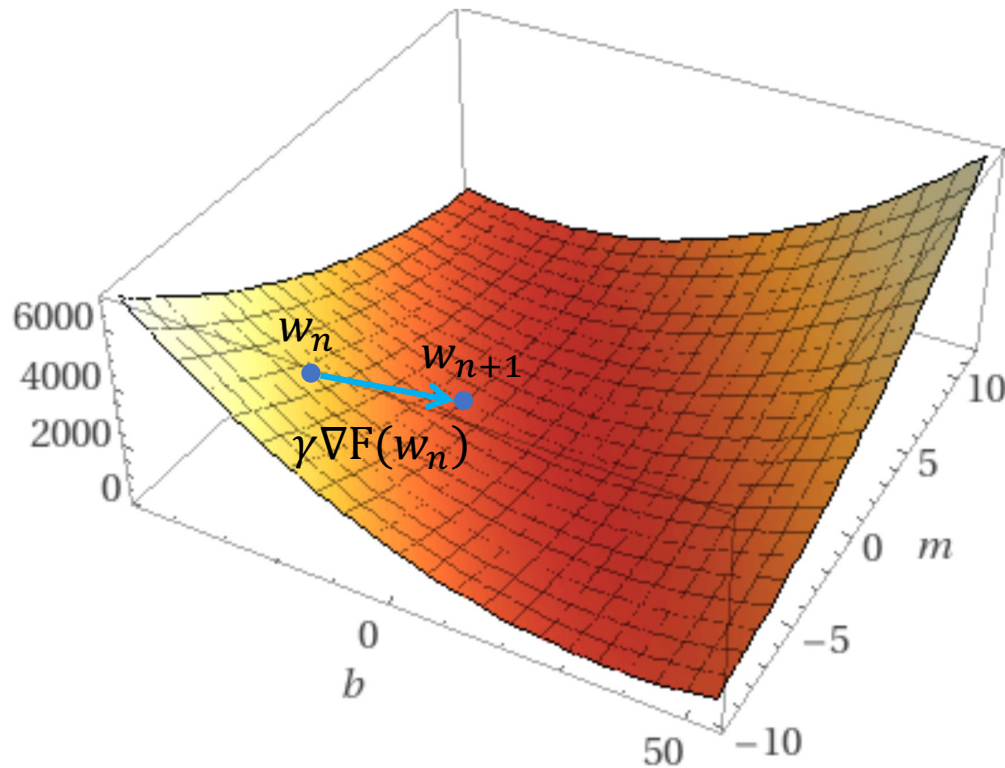


Find parameters $\{w_1, w_2, w_3, \dots, w_n\}$ such that $L = \frac{1}{n} \sum_{i=1}^n (y_i - \tilde{y}_i)^2$ is minimized

How do we find the optimal parameters?



The gradient descent algorithm



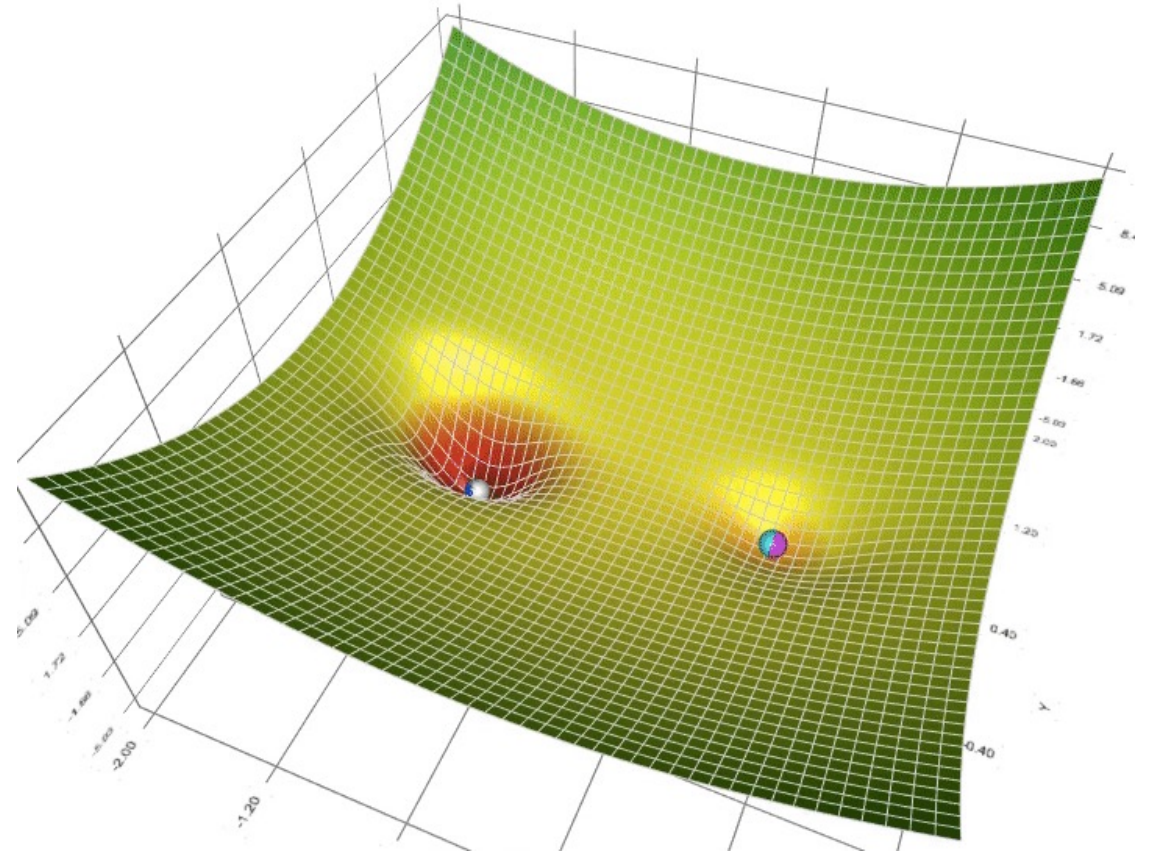
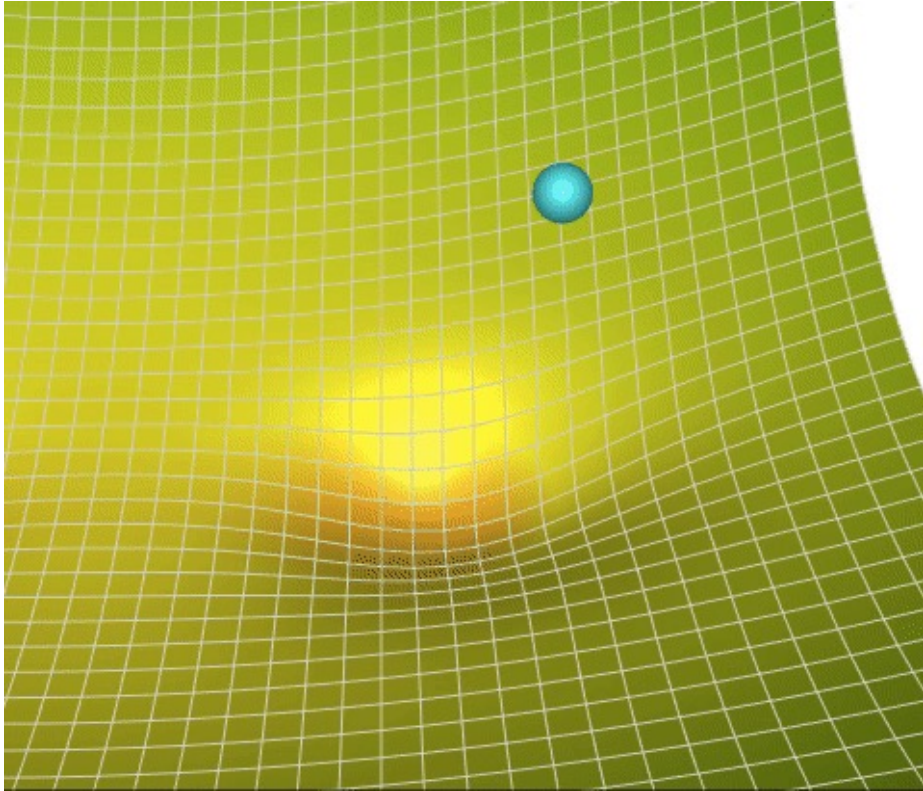
New configurations Old knob configurations

$$w_{n+1} = w_n - \underbrace{\gamma \nabla L(w_n)}_{\text{Direction of steepest descent}}$$

Learning rate

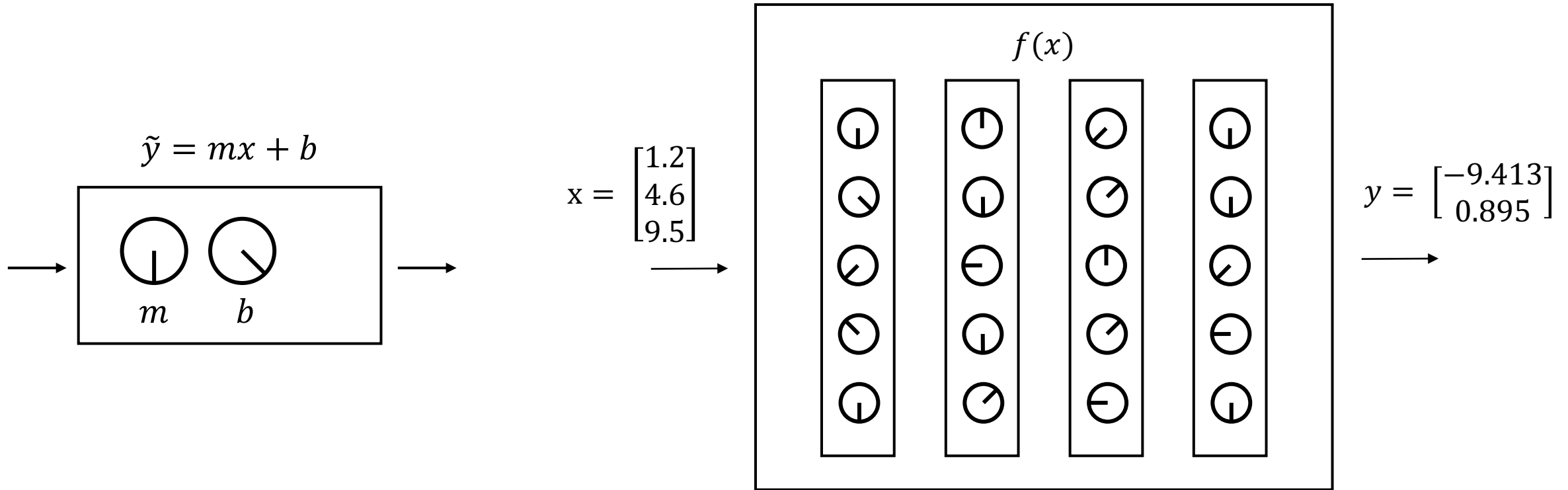
Direction of steepest descent

The gradient descent algorithm



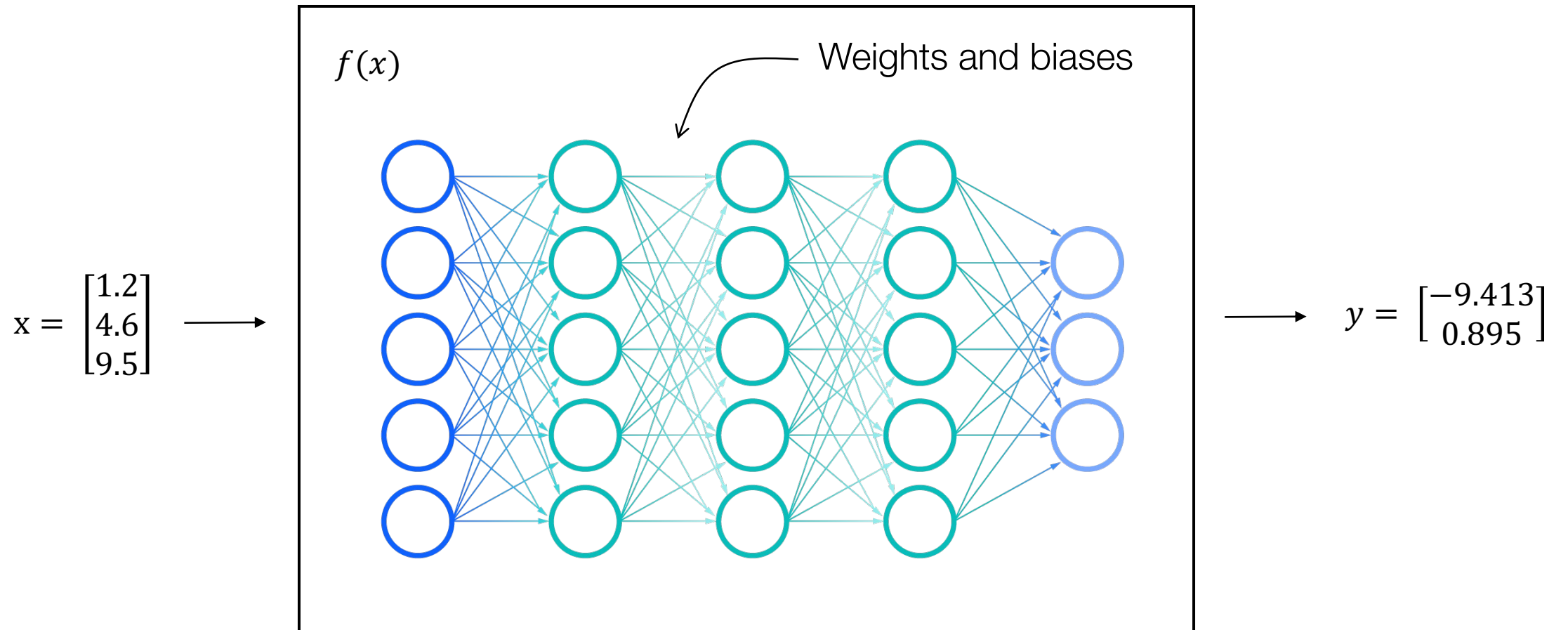
Animations courtesy of [Lili Jiang](#)

Time for an upgrade

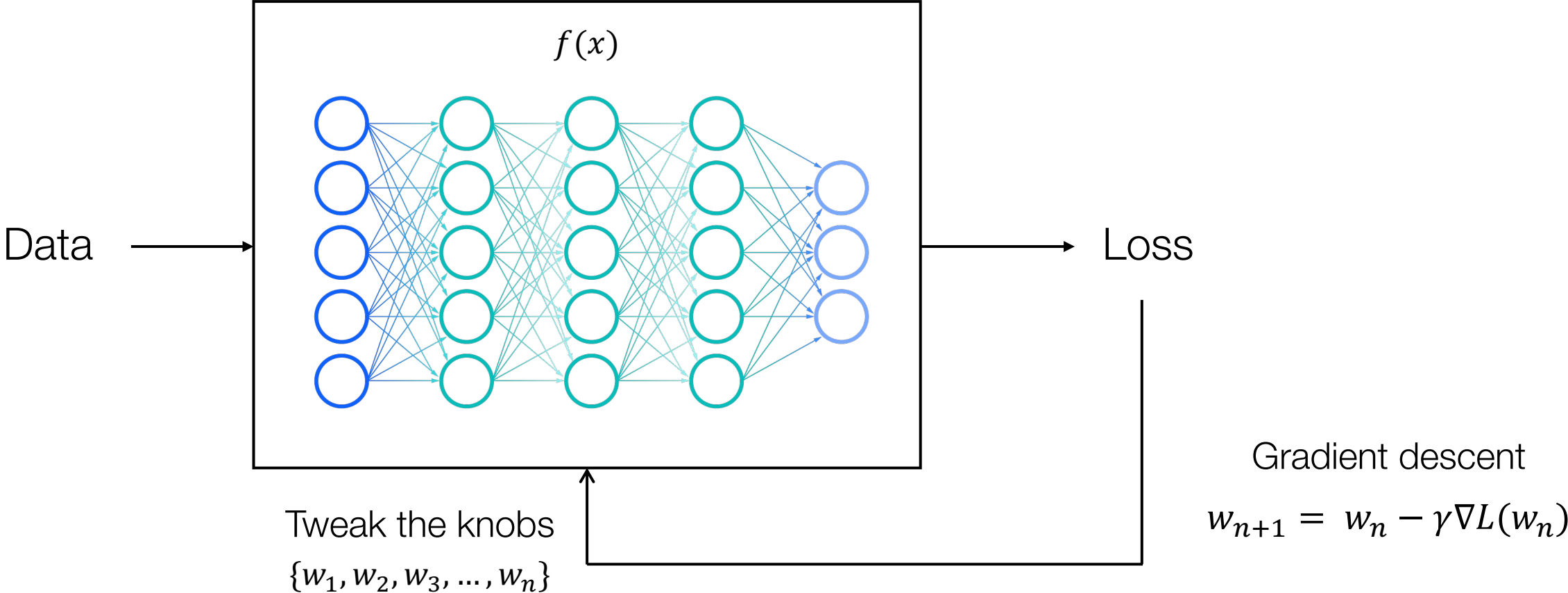


Neural networks are large, arbitrary function approximators

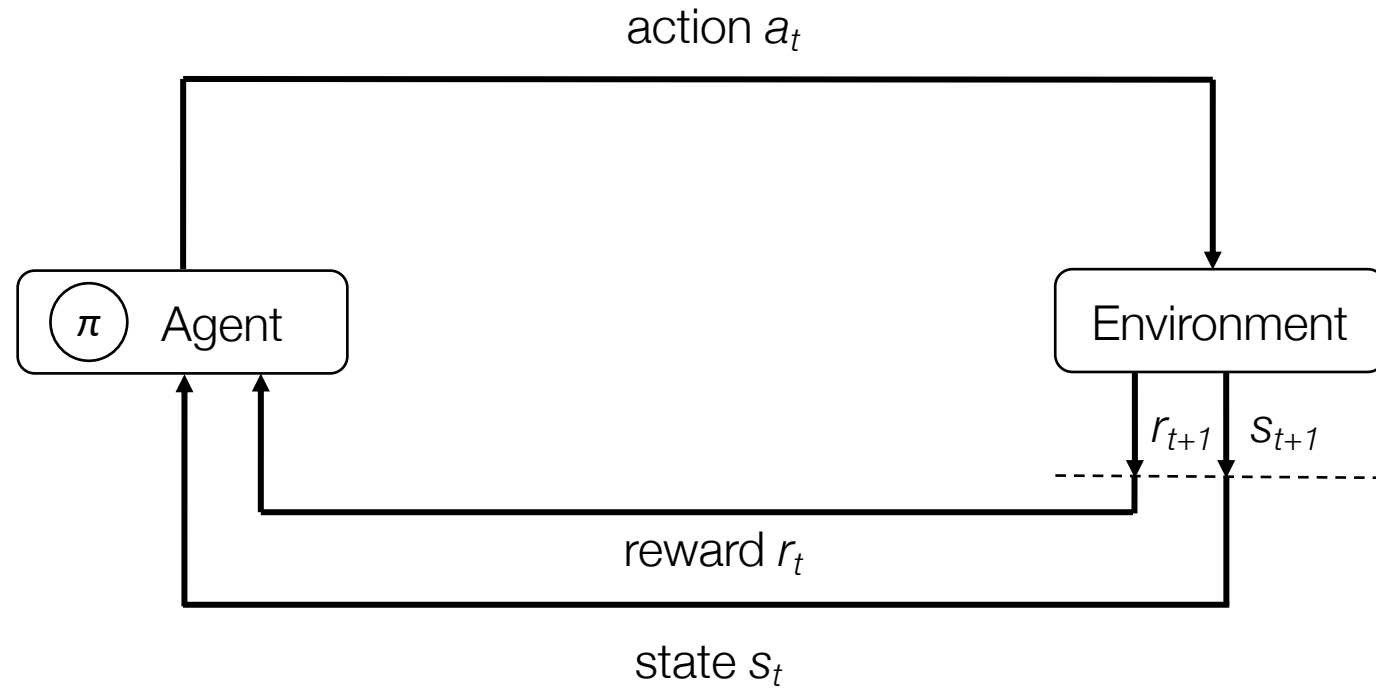
Neural networks are large, arbitrary function approximators



How machines learn

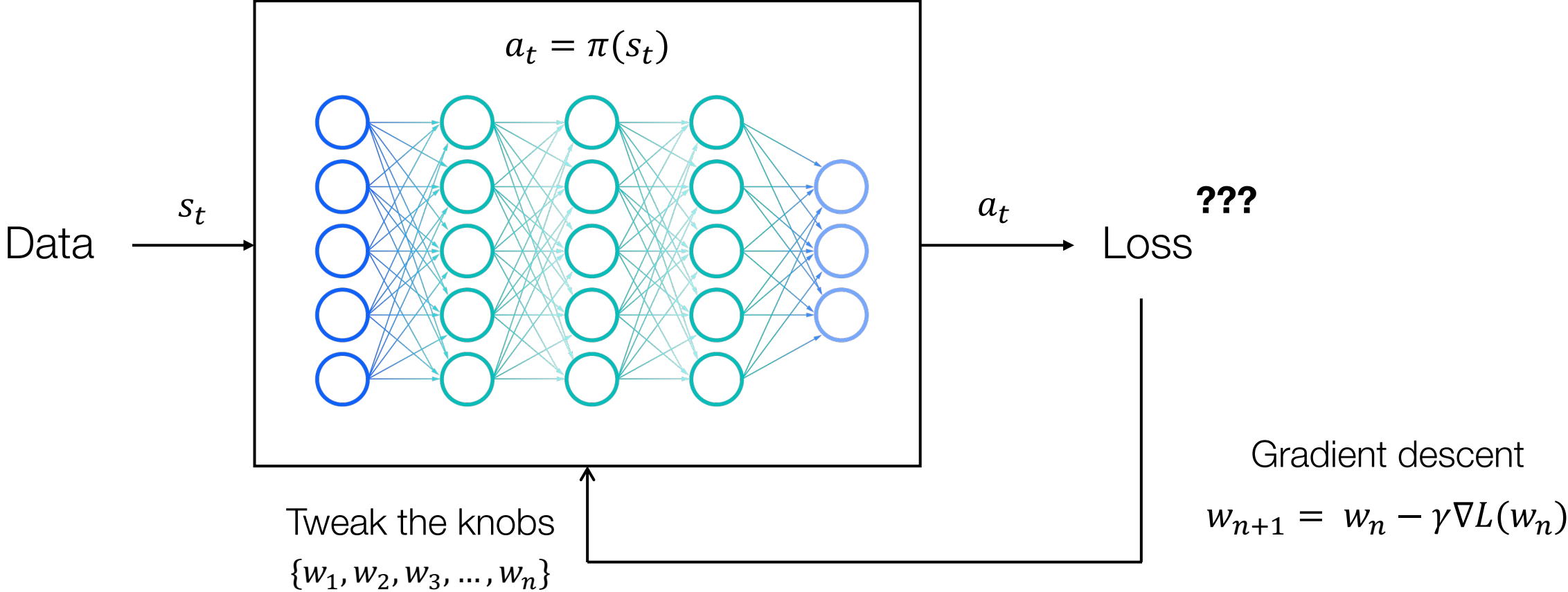


Representing the policy function

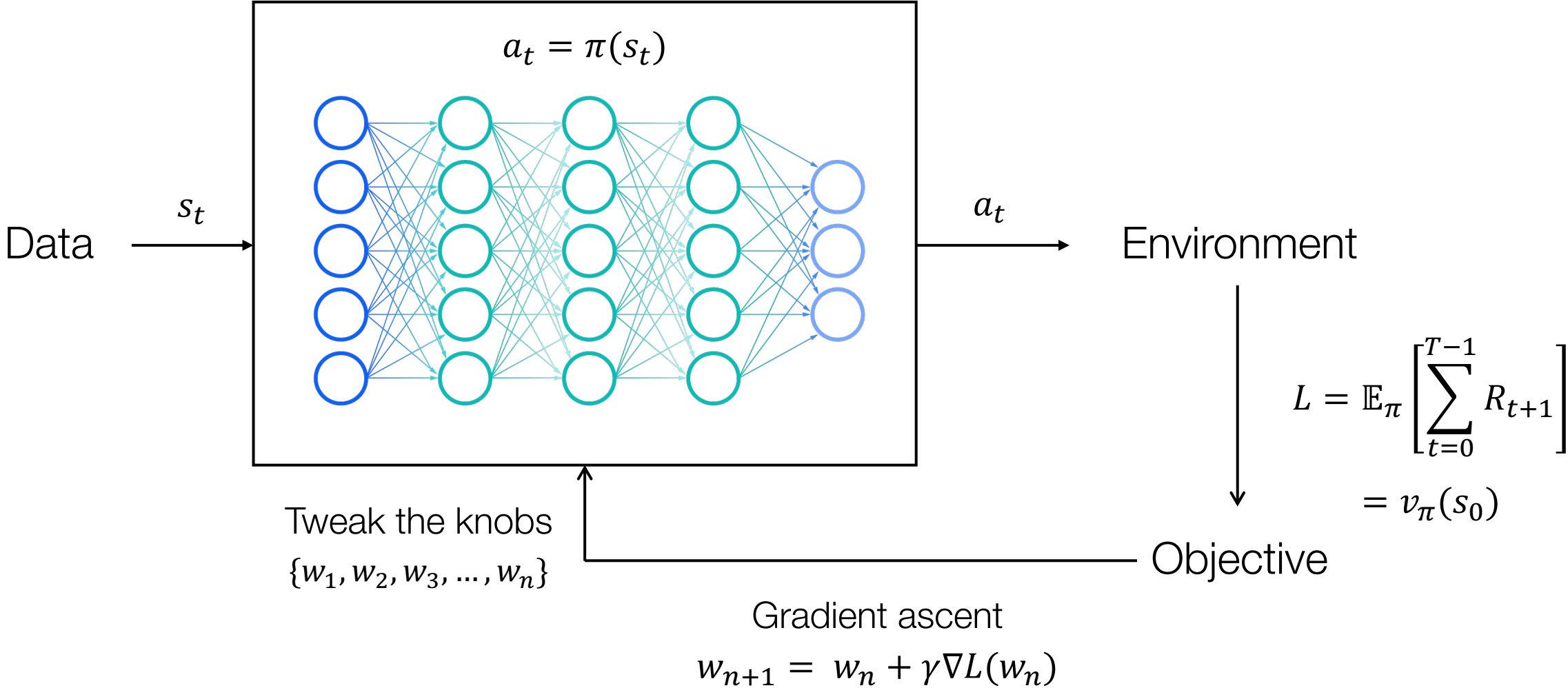


Policy $\pi: S \rightarrow A$ such that $a_t = \pi(s_t)$

Representing the policy function



Representing the policy function





Math ahead!
(feel free to ignore for one slide)


The policy gradient algorithm

Tweak the knobs on the neural network using $w_{n+1} = w_n + \gamma \nabla L(w_n)$, $L = \mathbb{E} \left[\sum_{t=0}^{\infty} R_{t+1} | \pi_w \right]$

$$\nabla_w L(w_n) = \nabla_w \mathbb{E} \left[\sum_{t=0}^{T-1} R_{t+1} | \pi_w \right] = \sum_{t=0}^{T-1} \nabla_w P(s_t, a_t | \tau) r_{t+1} = \sum_{t=0}^{T-1} P(s_t, a_t | \tau) \frac{\nabla_w P(s_t, a_t | \tau)}{P(s_t, a_t | \tau)} r_{t+1}$$

$$= \sum_{t=0}^{T-1} P(s_t, a_t | \tau) \log \nabla_w P(s_t, a_t | \tau) r_{t+1} \sim \sum_{t=0}^{T-1} \log \nabla_w P(s_t, a_t | \tau) r_{t+1} \quad \dots$$

Using $\frac{d}{dx} \log f(x) = \frac{f'(x)}{f(x)}$



Approximating one trial during training



$$= \sum_{t=0}^{T-1} \nabla_w \log \pi_w(s_t, a_t) \sum_{t'=t+1}^T R_{t'}$$

(we now have a gradient on π ! Presenting... the REINFORCE algorithm)

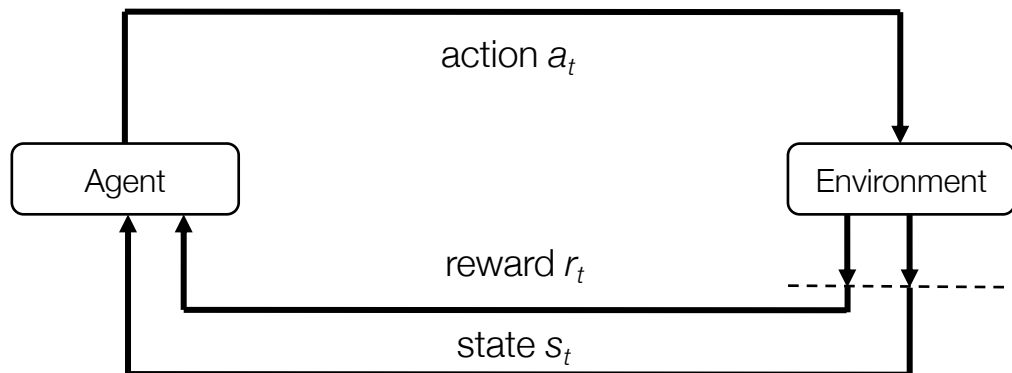
The policy gradient algorithm (REINFORCE)

$$w_{n+1} = w_n + \gamma \underbrace{\sum_{t=0}^{T-1} \nabla_w \log \pi_w (s_t, a_t)}_{\text{By going in the direction of steepest ascent in the logarithm of the policy probability}} \underbrace{\sum_{t'=t+1}^T R_{t'}}_{\text{Weighted by the total reward obtained in that trajectory}}$$

Tweak the knobs on our policy network

By going in the direction of steepest ascent in the logarithm of the policy probability

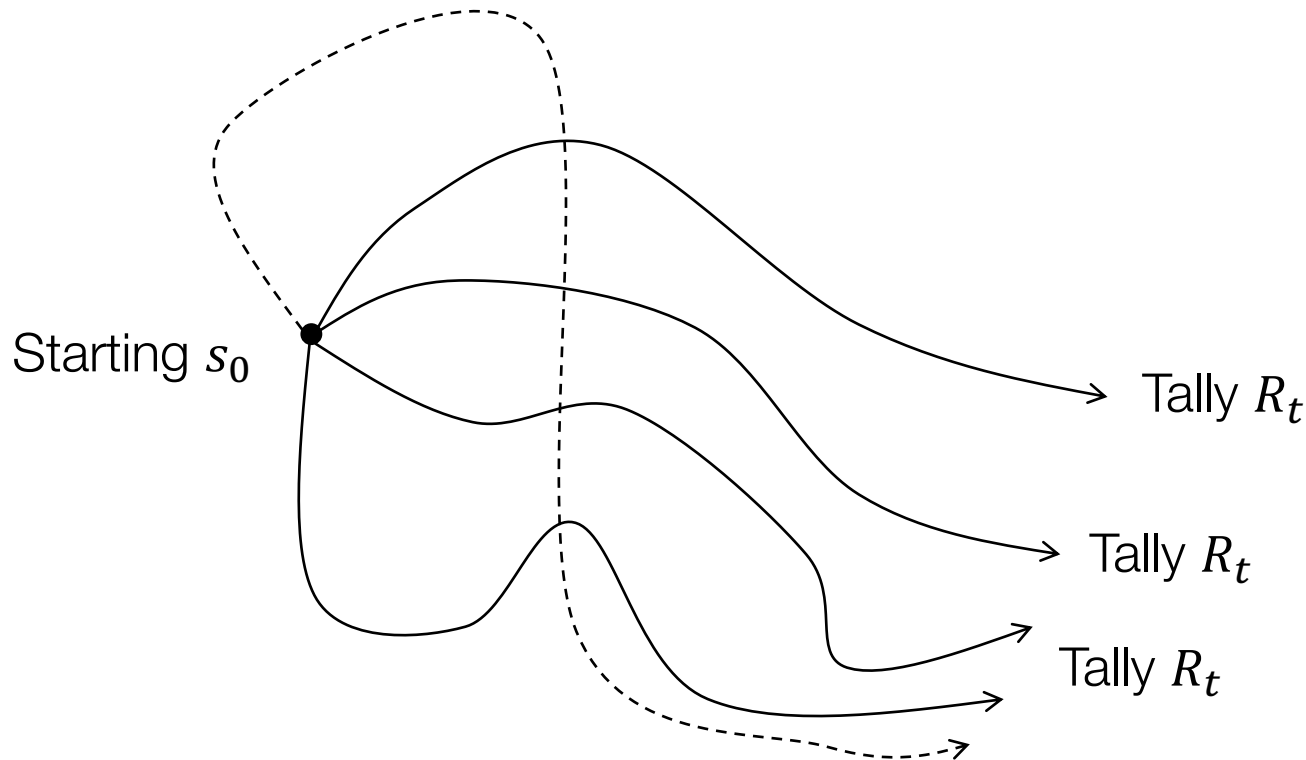
Weighted by the total reward obtained in that trajectory



- 1) Use a neural network to represent the policy π
- 2) For every trajectory (trial), try out that policy and record the rewards you get
- 3) Tweak the knobs on π using REINFORCE

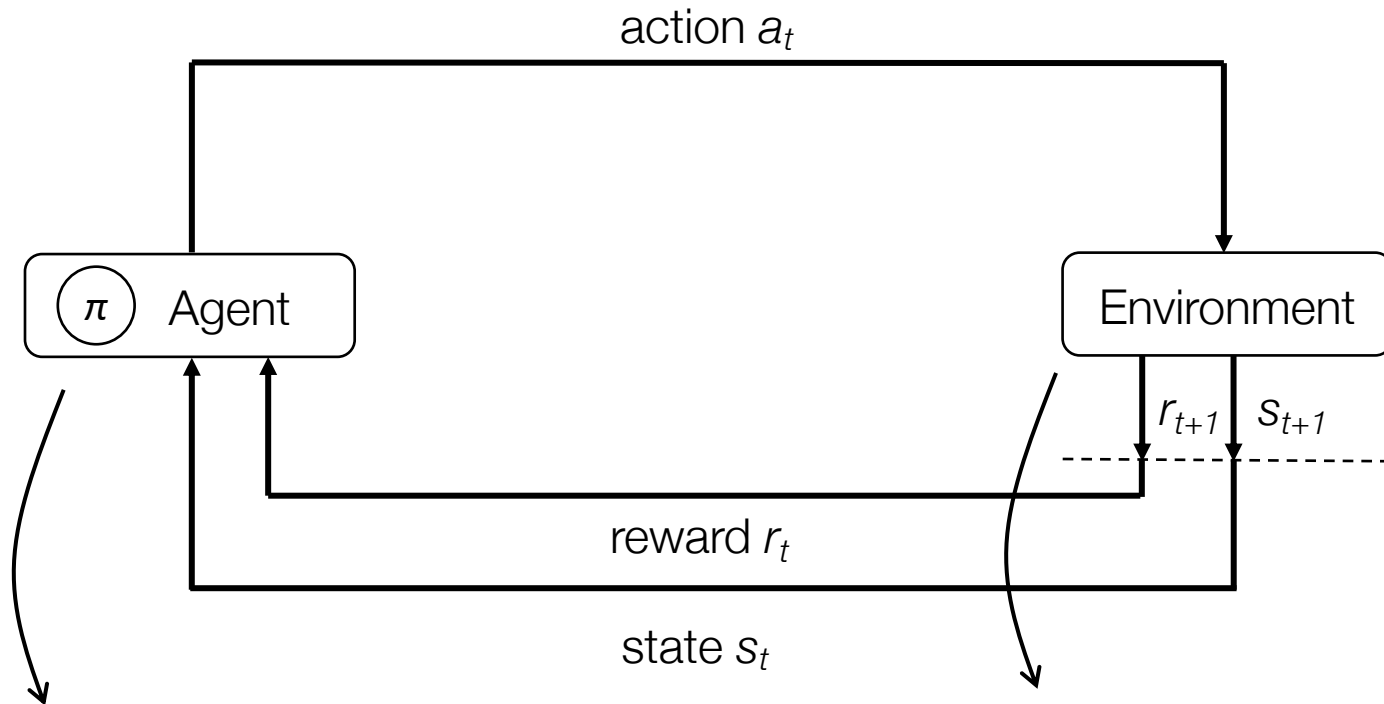
But there are problems!

$$\mathbb{E} \left[w_{n+1} = w_n + \gamma \sum_{t=0}^{T-1} \nabla_w \log \pi_w (s_t, a_t) \sum_{t'=t+1}^T R_{t'} \right]$$



- 1) Each trajectory can differ greatly from another, resulting in highly variable policies
- 2) We aren't necessarily sampling the whole space of trajectories, causing the distribution to skew towards a non-optimal direction
- 3) On-policy learning can be expensive! We would like to re-use old experiences.

Back to big picture



1. How do we represent and learn π using a computer?

2. How do we figure out the behavior of the environment if it is not given to us?

$$v_{\pi}(s_t) = \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} R_{t+k+1} | S = s_t \right]$$

$$v_*(s) = \text{maximize}_{\pi}(v_{\pi}(s))$$

There are two dominant methods in reinforcement learning:

(a) Learn π directly, trying out different policies to maximize reward (policy gradient method)

(b) Learn the action-value function (Q-learning)

Action-value function

The state-value function assigns a value to each state:

$$\begin{aligned}v_{\pi}(s_t) &= \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} R_{t+k+1} | S = s_t \right] \\ &= r(s_t, a_t) + \mathbb{E}[v_{\pi}(s_{t+1})]\end{aligned}$$

$$v_*(s) = \max_{\pi}(v_{\pi}(s))$$

Similarly, we can define the action-value function:

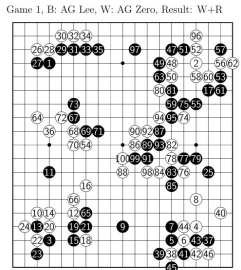
$$\begin{aligned}q_{\pi}(s_t, a_t) &= \mathbb{E}_{\pi} \left[\sum_{k=0}^{\infty} R_{t+k+1} | S = s_t, A = a_t \right] \\ &= r(s_t, a_t) + \mathbb{E}[q_{\pi}(s_{t+1}, a_{t+1})]\end{aligned}$$

$$q_*(s, a) = \max_{\pi}(q_{\pi}(s, a))$$

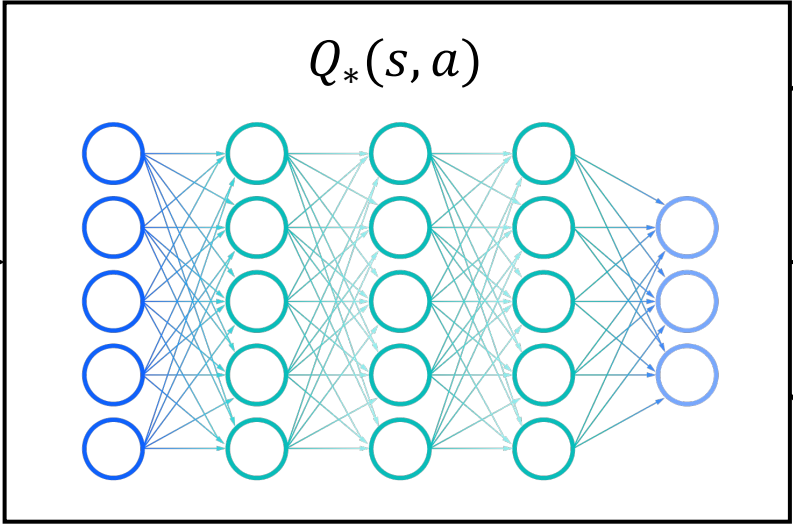


Idea: learn this instead of $\pi_*(s)$

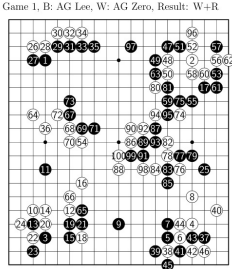
Deep Q-network (DQN)



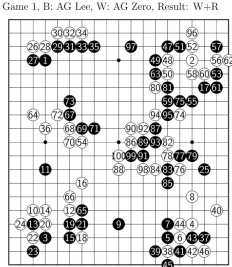
s



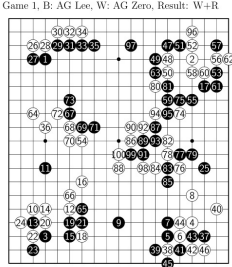
Compute action-values for each possible action from state s



$Q_*(s, a^1)$



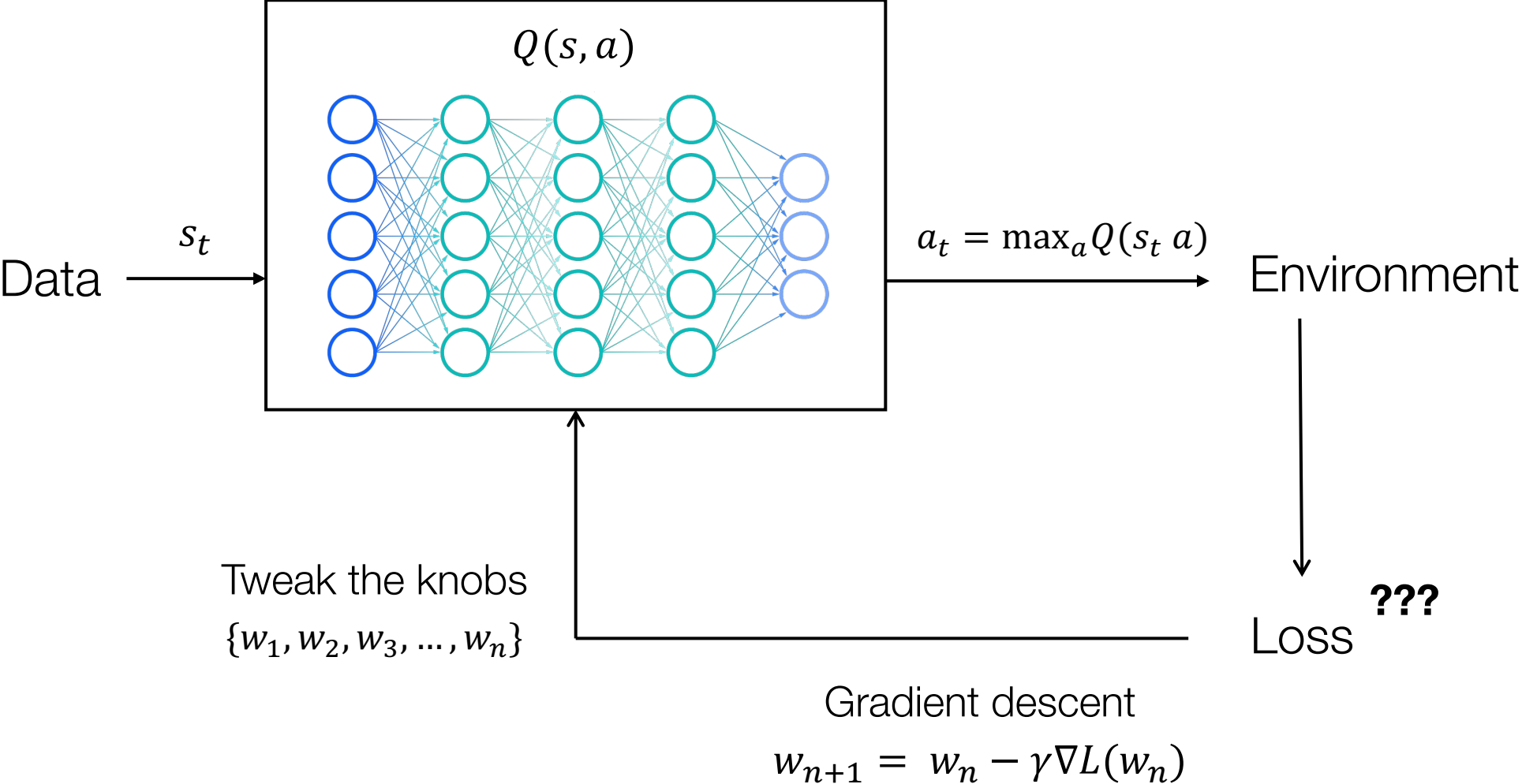
$Q_*(s, a^2)$



$Q_*(s, a^3)$

Agent picks action
 $a = \max_a Q_*(s_t a)$

Deep Q-learning



Deep Q-learning loss

Recall that one popular loss objective is to minimize the “average of the squared errors”

$$L = \frac{1}{n} \sum_{i=1}^n (y_i - \tilde{y}_i)^2$$

“Actual” “Estimate”

In Q-learning,

$$L = \mathbb{E}[(Q_*(s_t, a_t) - Q(s_t, a_t))^2]$$

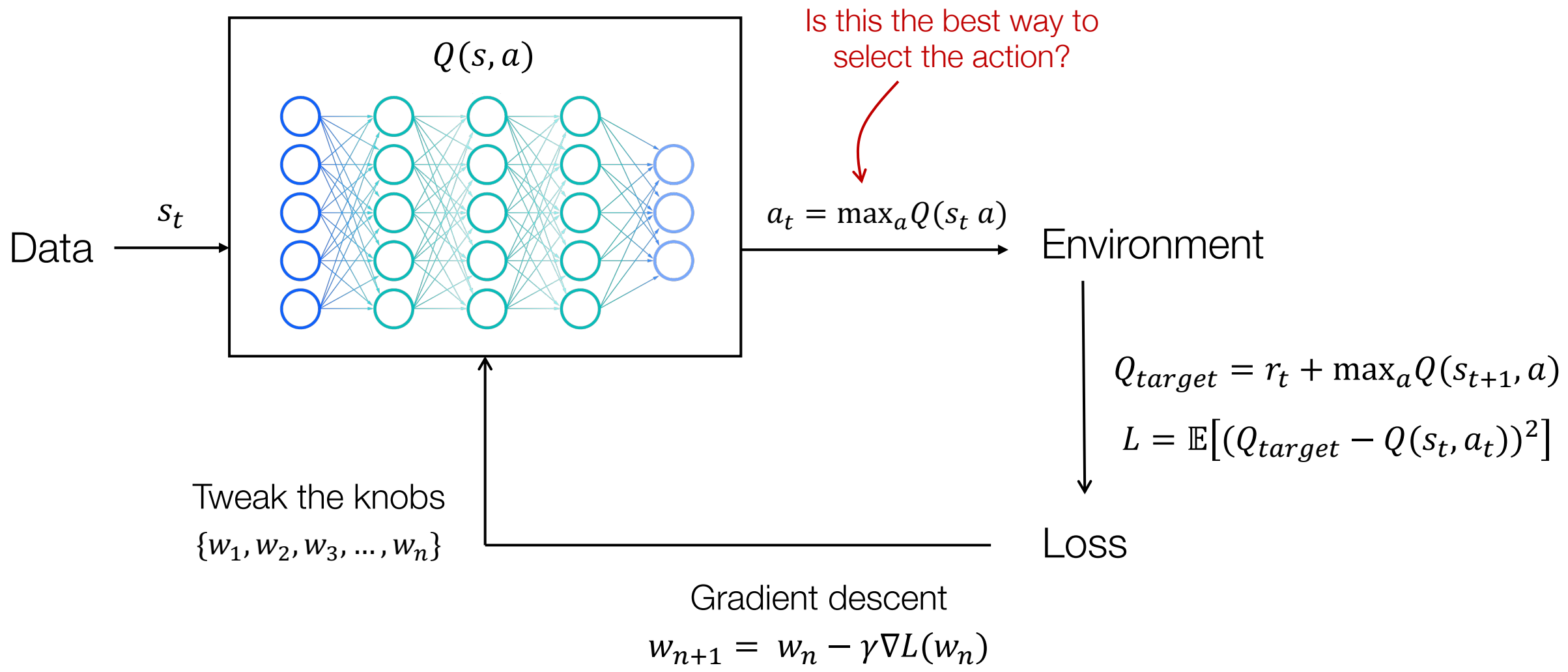
But we don't know actual $Q_*(s_t, a_t)$!

Instead, we **assume** $Q \sim Q_*$ and compute target value using Bellman equation:

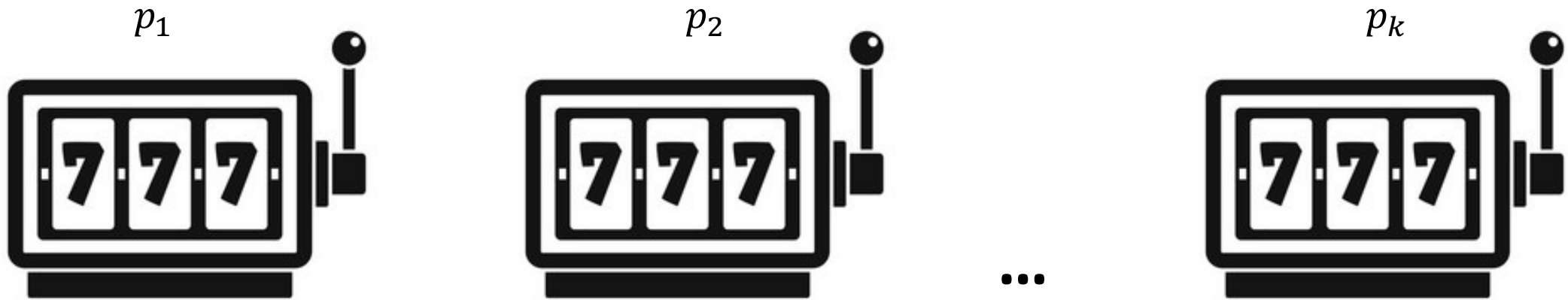
$$Q_{target} = r_t + \max_a Q(s_{t+1}, a)$$

$$L = \mathbb{E}[(Q_{target} - Q(s_t, a_t))^2]$$

Deep Q-learning



k-armed bandit problem



For machine i , you get \$1 with probability p_i and \$0 with probability $1 - p_i$.

The probabilities are *hidden*.

You get to play 100 times and want to win as much \$\$ as possible.

What do you do?

Exploration vs. exploitation trade-off

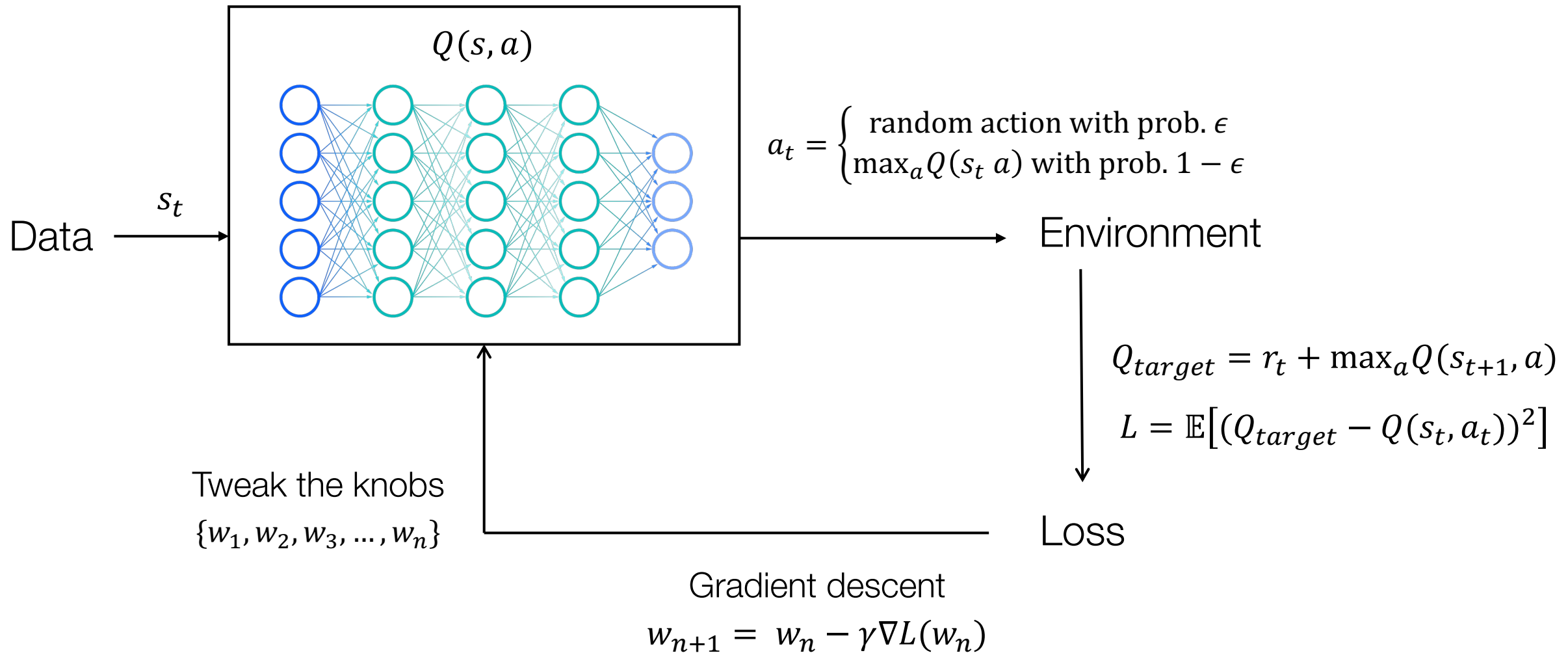
The RL agent needs to decide whether to exploit what they know or explore a random action

Too much exploitation (i.e. always select best known action $a_t = \max_a Q(s_t, a)$)
⇒ can get stuck in greedy suboptimal policy

Too much exploration (i.e. always select random action a_t)
⇒ not utilizing prior information

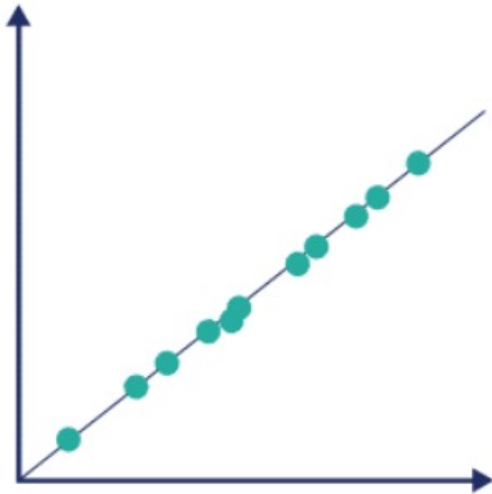
Simple fix: use ϵ -greedy selection: $a_t = \begin{cases} \text{random action with prob. } \epsilon \\ \max_a Q(s_t, a) \text{ with prob. } 1 - \epsilon \end{cases}$

ϵ -greedy deep Q-learning

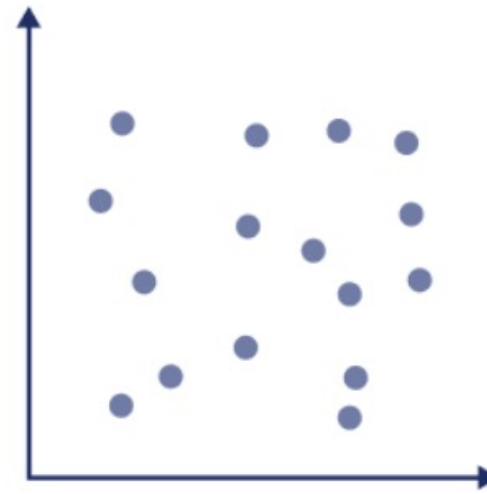


DQN convergence problem

Gradient descent theory requires that training data is “independent and identically distributed” (i.i.d.) for it to converge



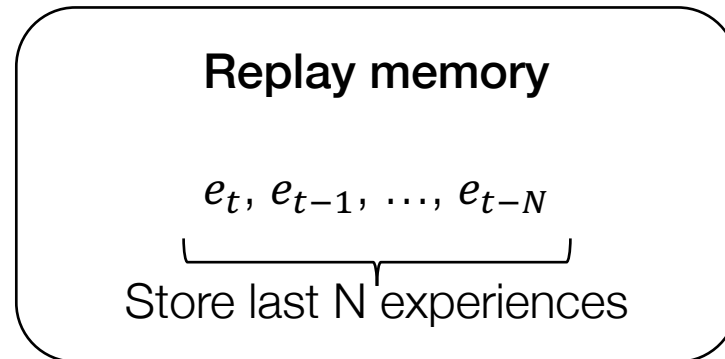
But sequential experiences are often highly correlated



We can break this correlation by introducing random sampling

Experience replay

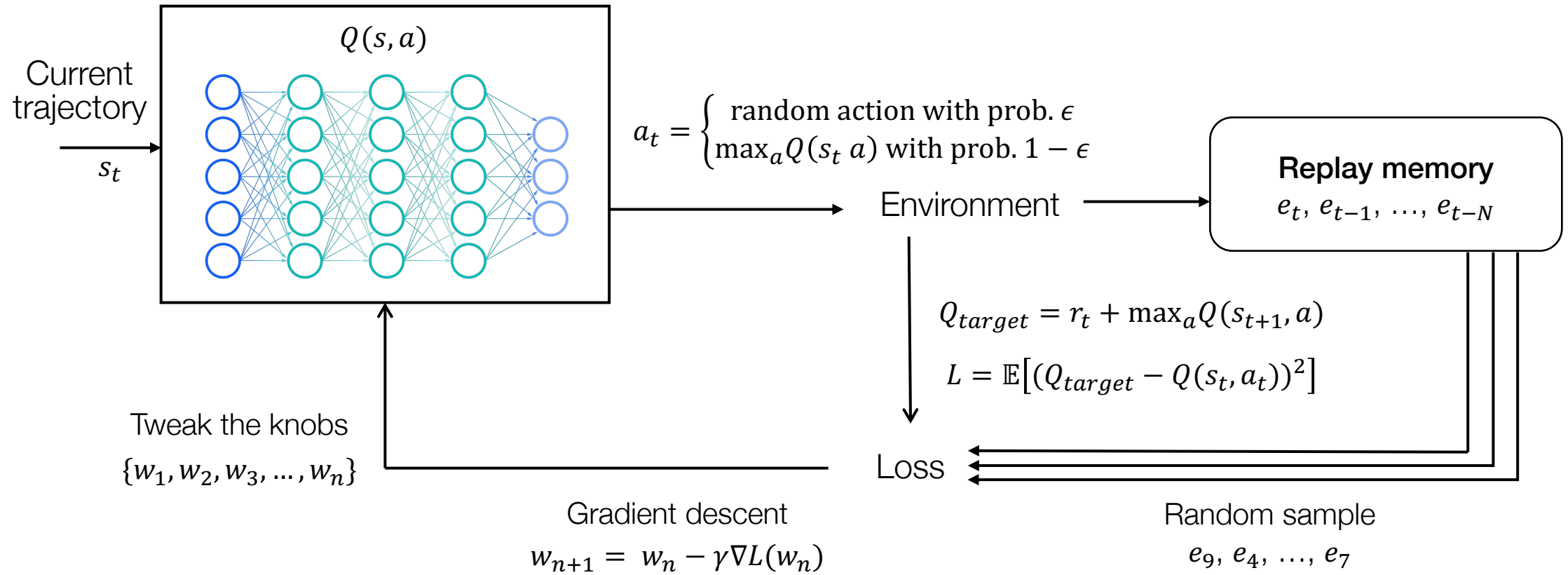
Define the agent's experience at time t by $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$



For $Q(s, a)$ training:

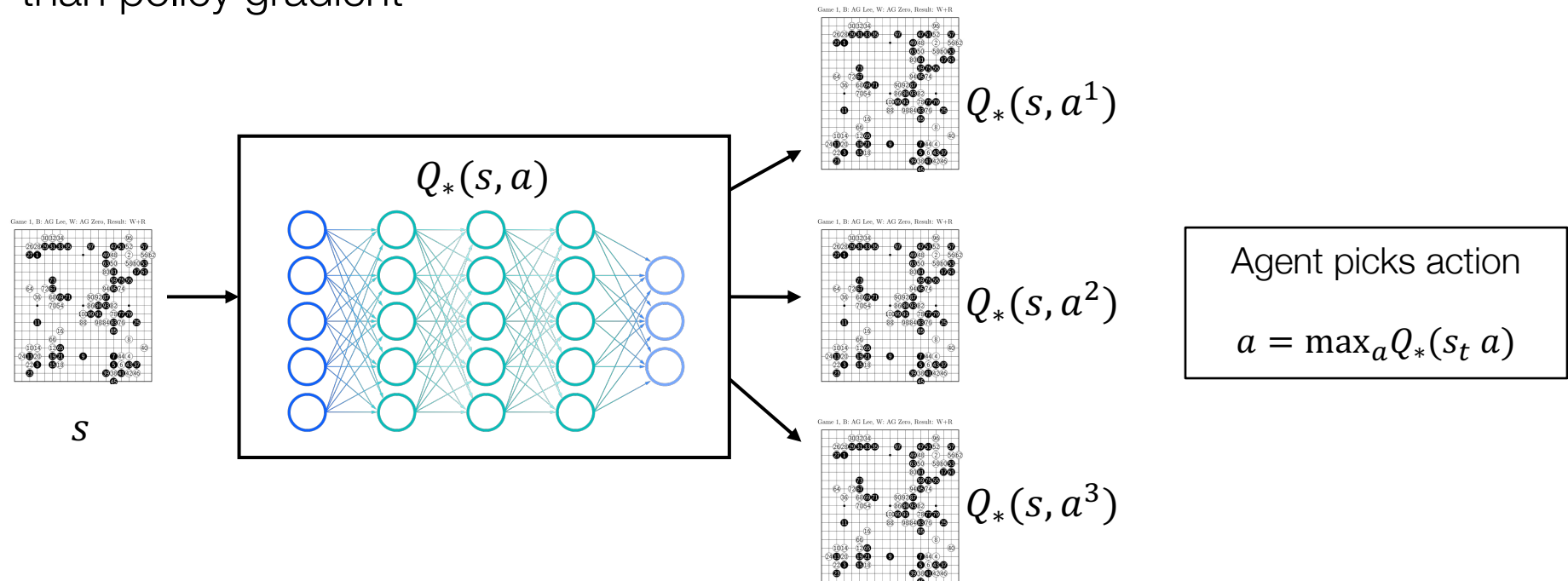
- 1) Randomly sample batch of m experiences from replay memory
- 2) Update $Q(s, a)$ via gradient descent
- 3) Repeat 1-2 until convergence

ϵ -greedy deep Q-learning with experience replay



But there are other problems (again)!

- 1) Can't learn stochastic policies (i.e. probability distribution over actions)
- 2) Hard to deal with continuous action spaces
- 3) Usually slower than policy gradient



Can we... maybe... combine the two? 🤔 🙅🏻

Policy gradient + Q-learning = Actor-critic

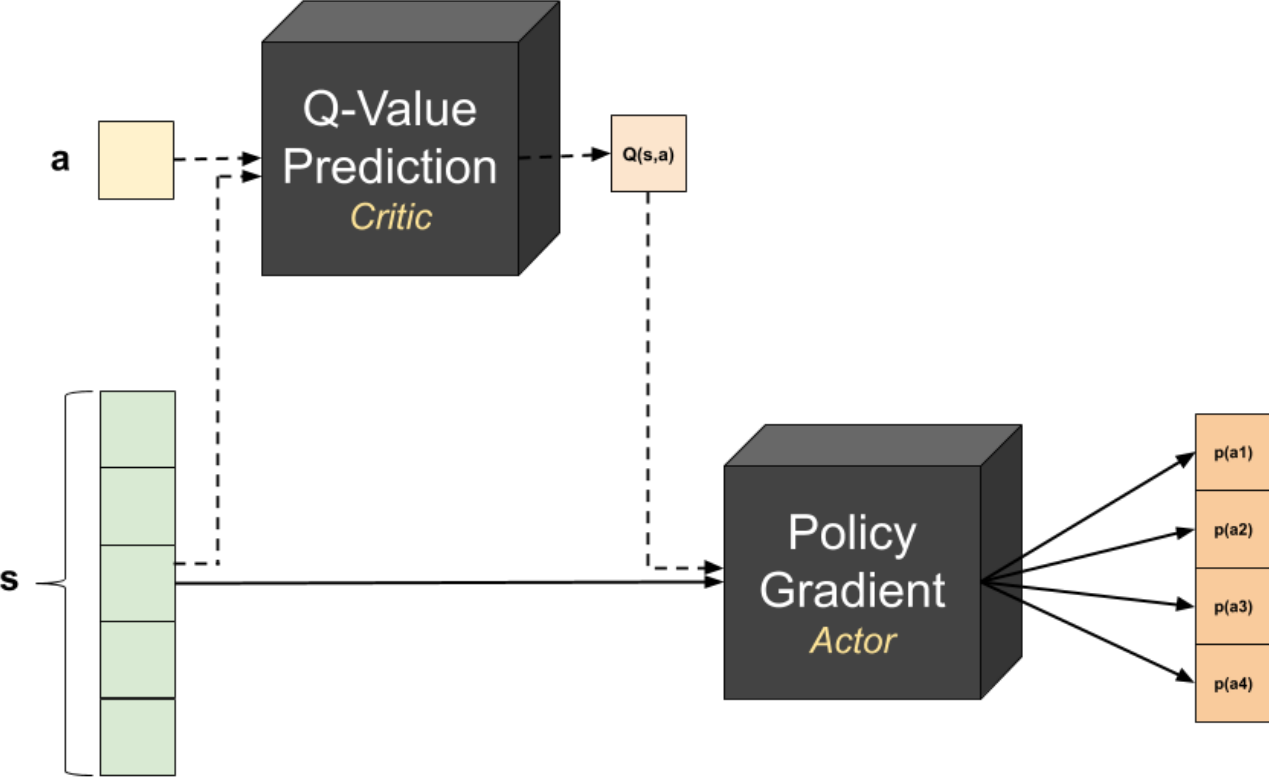
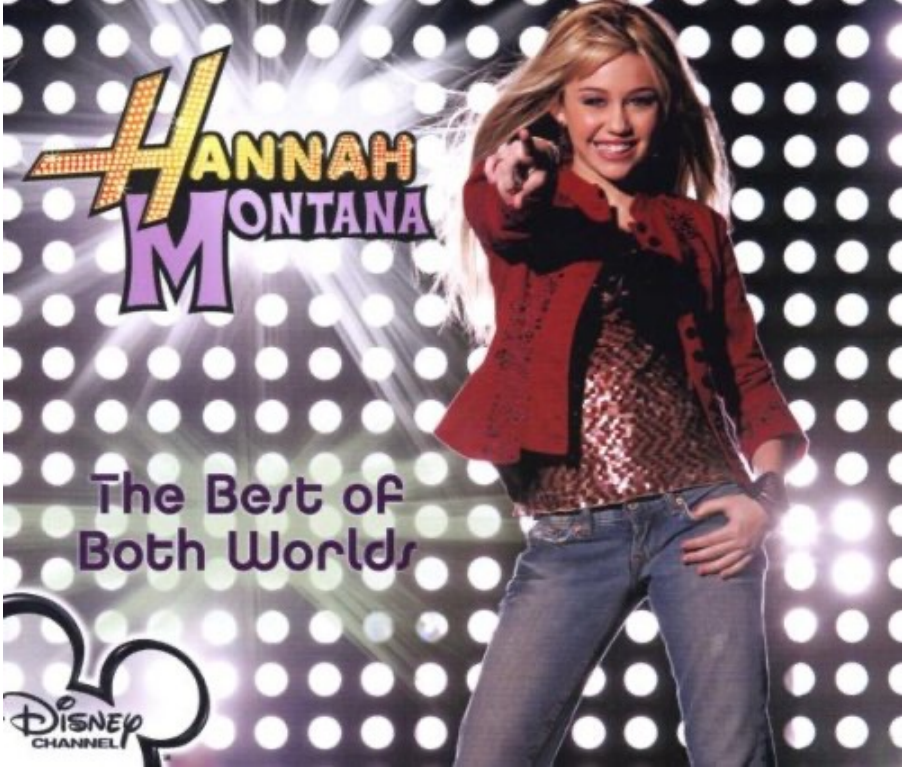


Figure credits: [Shaked Zychlinski](#)



Improving policy gradient

$$w_{n+1} = w_n + \gamma \sum_{t=0}^{T-1} \underbrace{\nabla_w \log \pi_w (s_t, a_t)}_{\text{By going in the direction of steepest ascent in the logarithm of the policy probability}} \underbrace{\sum_{t'=t+1}^T R_{t'}}_{\text{Weighted by the total reward obtained in that trajectory}} \quad Q(s_t, a_t)$$

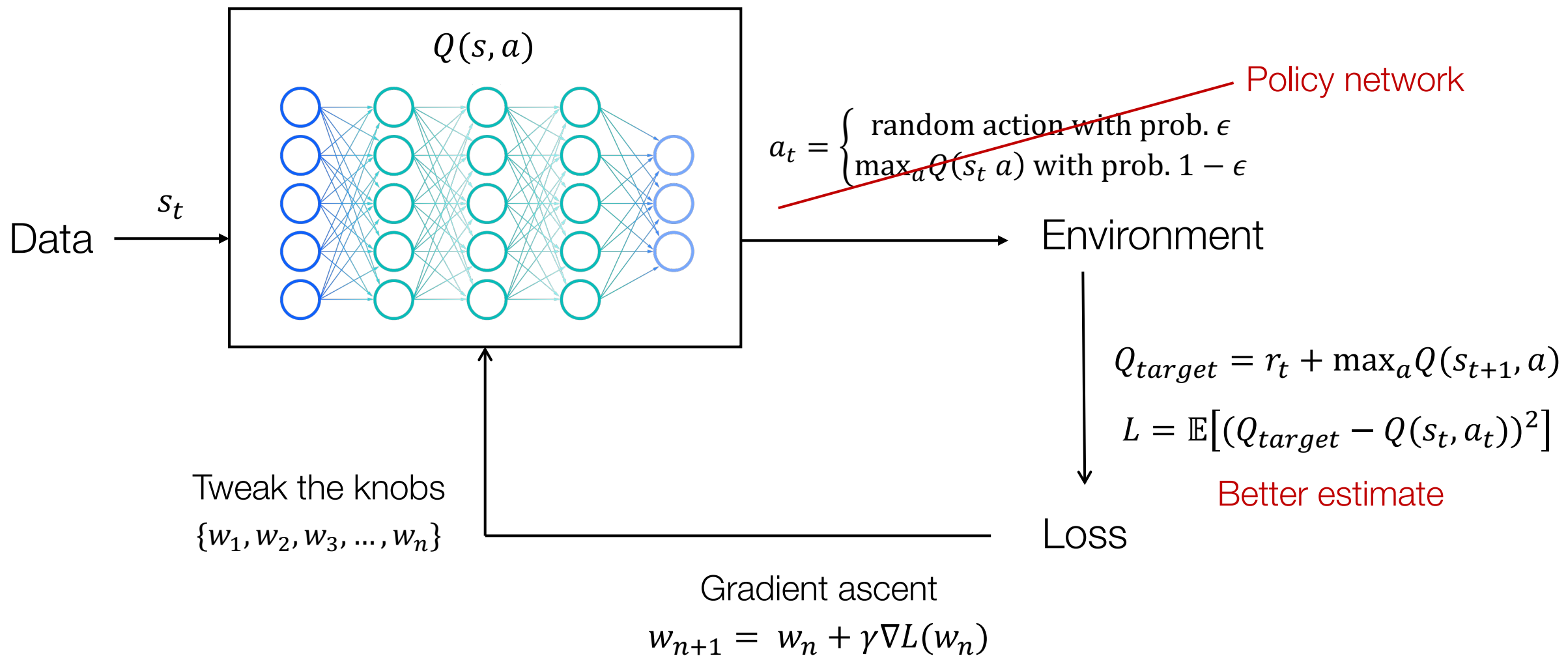
Tweak the knobs on our policy network

By going in the direction of steepest ascent in the logarithm of the policy probability

Weighted by the total reward obtained in that trajectory

- Lower variability in training (random events in the world don't affect our review of what happened)
- More efficient use of data: instead of re-running each trial, we can use experience captured by Q
- As our evaluation of Q gets better, our policy gradient gets better!

Improving deep Q-learning



The best of both worlds

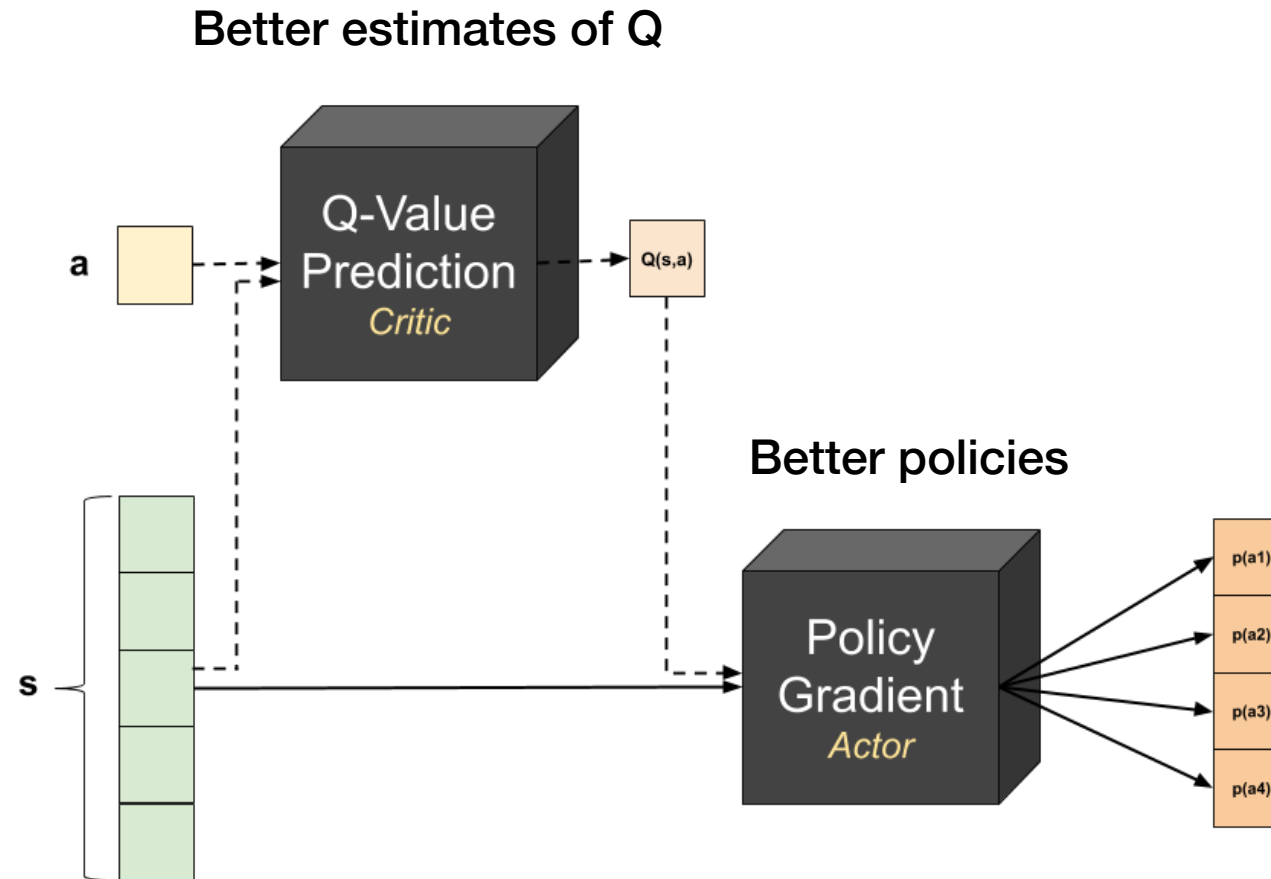


Figure credits: [Shaked Zychlinski](#)

Next time

- Humans as RL agents
- Utility theory
- Behavioral economics

Lecture 1 Introduction and the RL problem	Lecture 2 How computers learn	Lecture 3 How people learn	Lecture 4 Multi-agent systems	Lecture 5 Interactions on graphs	Lecture 6 Complex systems science
--	--	--------------------------------------	--	---	--

References and additional resources

- [Reinforcement Learning: An Introduction](#) by Richard S. Sutton and Andrew G. Barto
- [Deep Learning video series](#) by 3Blue1Brown
- [RL video series](#) by deeplizard (specifically videos on Q-learning & experience replay)
- Medium article [Qrash Course II: From Q-Learning to Gradient Policy & Actor-Critic in 12 Minutes](#)