King Abdullah University of Science and Technology

KAUST

Deep Reinforcement Learning

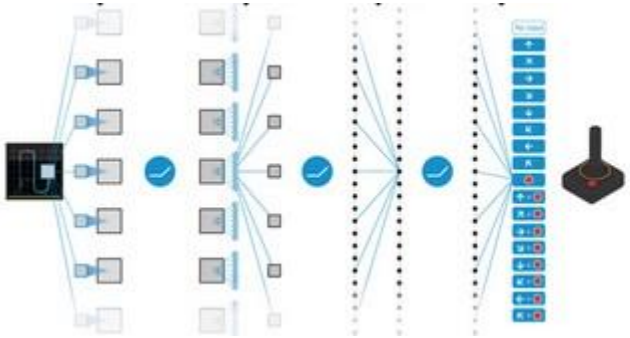
GROUP MEETING – 20.10.2016

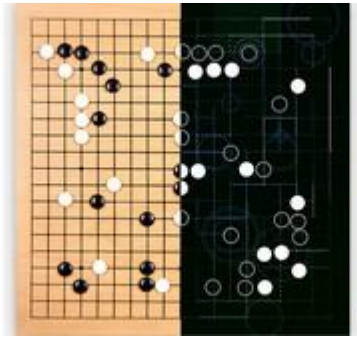
Matthias Mueller

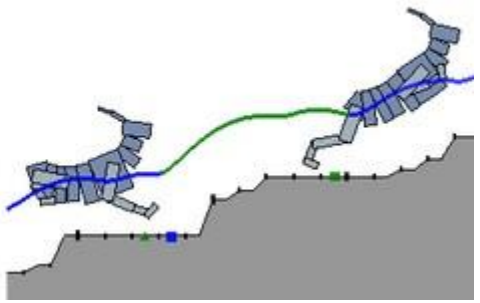
*Slides stolen from David Silver, John Schulmann, Xianling Zhang

Motivation – RL is hot!

- Automatically learn to play ATARI games (from raw game pixels!)
- Beat world champions at Go
- Robots are learning how to perform complex manipulation tasks that defy explicit programming
- Simulated quadrupeds are learning to run and leap



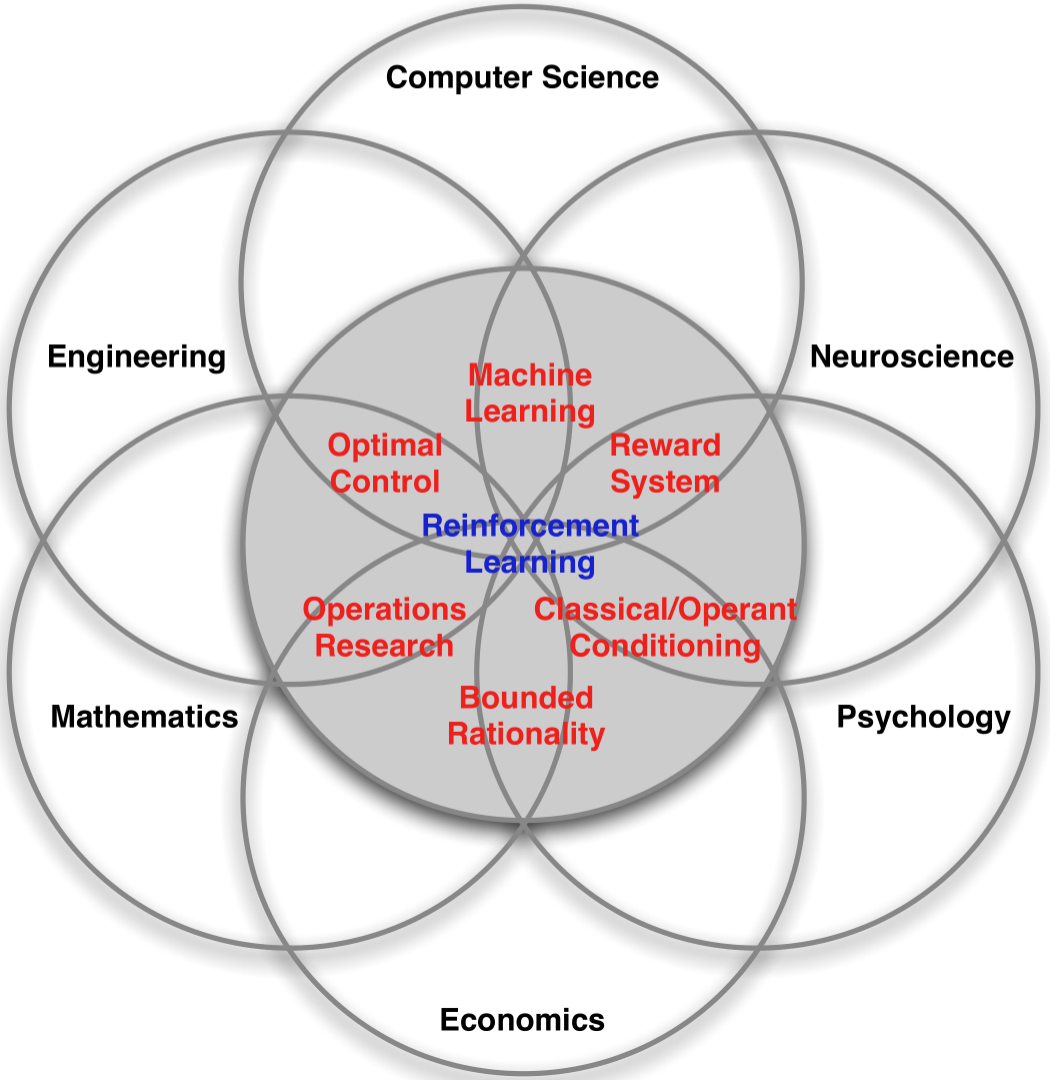




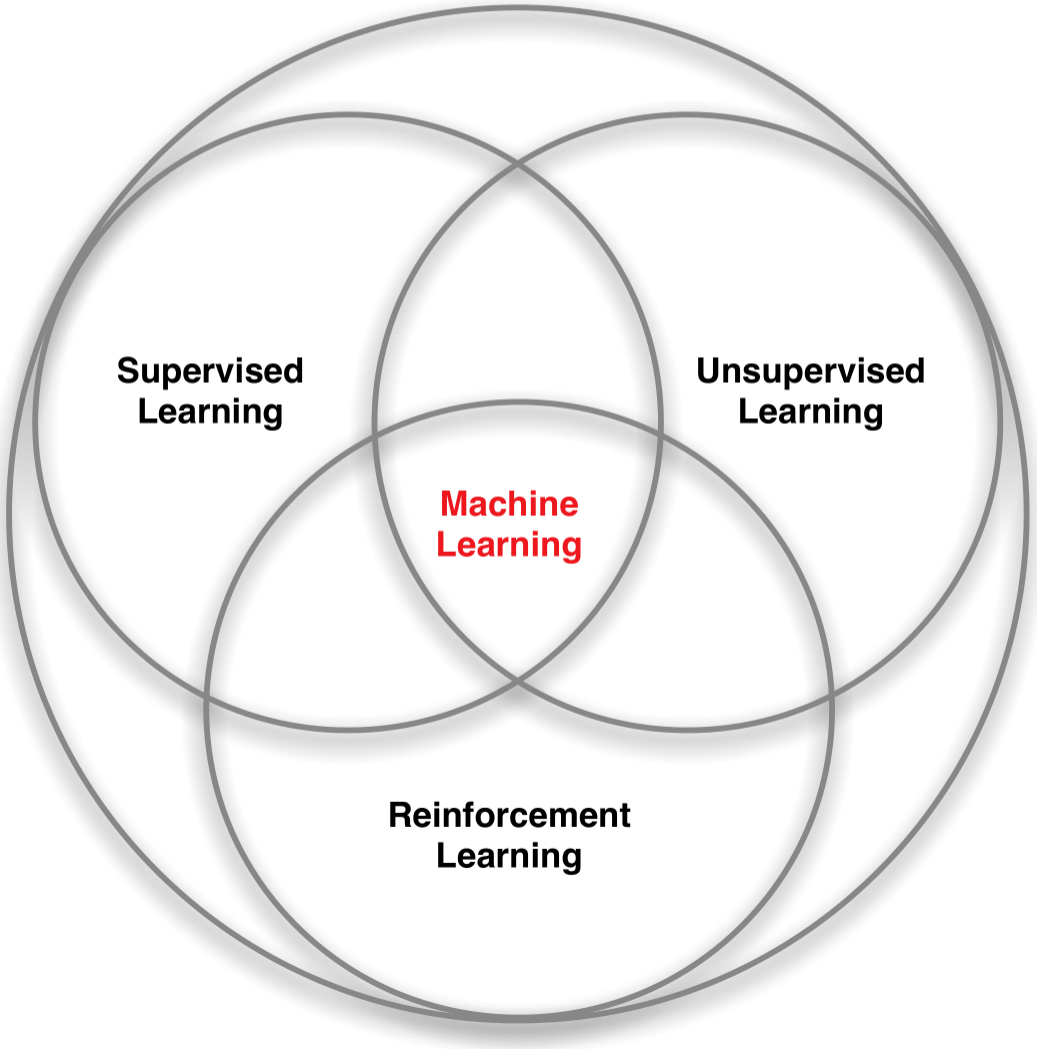
Overview

- **Introduction to Reinforcement Learning**
  - What is RL?
  - Examples of RL
  - Elements of RL
  - Q-Learning
- Deep Reinforcement Learning
  - Deep Q-Network (DQN)
  - Policy Gradients (PG)

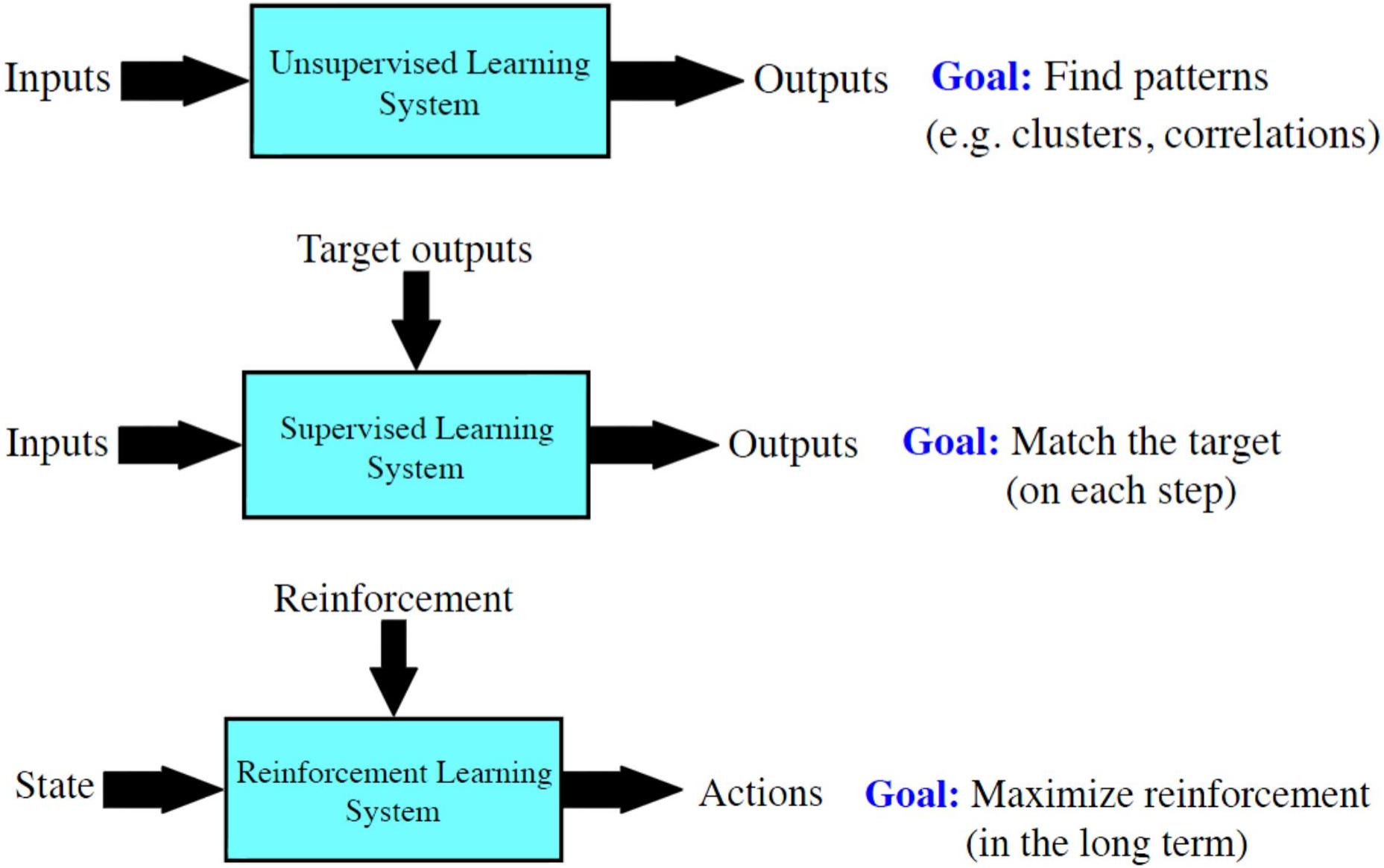
Many Faces of Reinforcement Learning



Branches of Machine Learning



3 Types of Learning



Overview

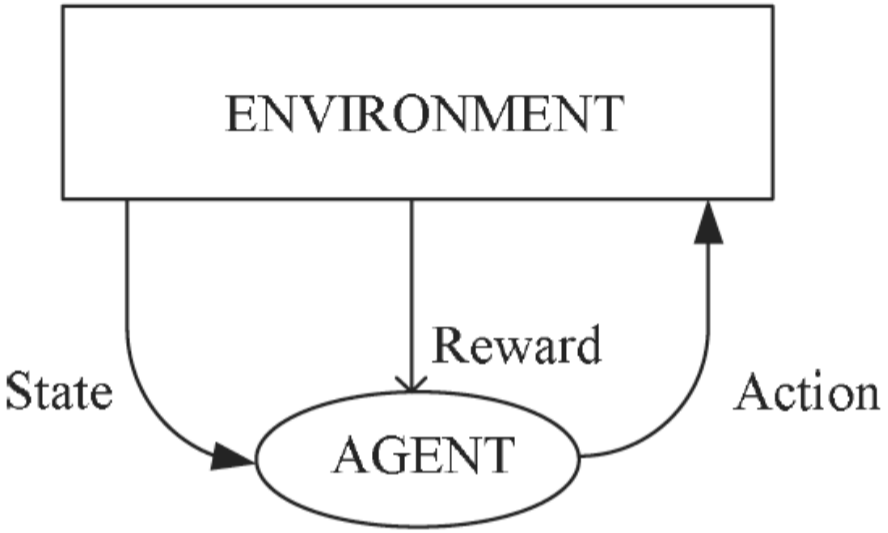
- Introduction to Reinforcement Learning
  - **What is RL?**
  - Examples of RL
  - Elements of RL
  - Q-Learning
- Deep Reinforcement Learning
  - Deep Q-Network (DQN)
  - Policy Gradients (PG)

Definition of RL

In reinforcement learning, the learner is a decision-making agent that takes actions in an environment and receives reward (or penalty) for its actions in trying to solve a problem. After a set of trial-and-error runs, it should learn the best policy, which is the sequence of actions that maximize the total reward.

Chapter 18, Introduction to Machine Learning.
Ethem Alpaydın 2009

RL diagram



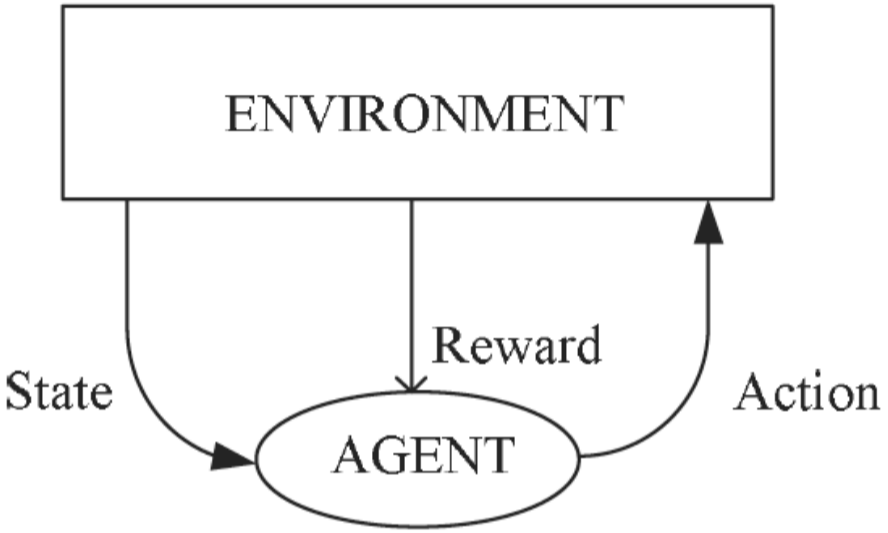
Agent: decision maker

State: one of the possible states in environment

Action: agent output to affect the environment

Reward: how good the action is

Objective of RL



Learning what to do —how to map states to actions—so as to maximize a numerical reward signal

Solution to a task: the best sequence of actions (with the maximum cumulative reward)

RL vs Supervised Learning

Reinforcement Learning	Supervised Learning
Learning with a critic	Learning with a teacher
How well we have been doing in the past	What's good or bad
Learn to generate an internal value for the intermediate states or actions (how good they are)	Learn to minimize general risk on predicting
Exploration and Exploitation	Over-fitting and Under-fitting

Characteristics of Reinforcement Learning

What makes reinforcement learning different from other machine learning paradigms?

- There is no supervisor, only a *reward* signal
- Feedback is delayed, not instantaneous
- Time really matters (sequential, non i.i.d data)
- Agent's actions affect the subsequent data it receives

A Simple Example: k-armed bandit

Target: earn money AMAP

Task: decide which lever to pull

Agent: you

State: single state (one slot machine)

Action: choose one lever to pull

Reward: money given by the machine after one action (immediate reword after a single action)

Another Example: a machine to play chess

Target: win the game

Task: decide a sequence of moves

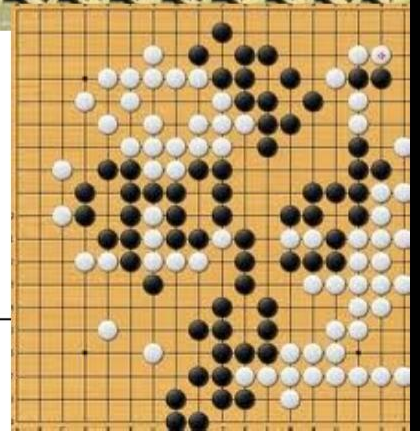
Agent: player

State: state of the board(environment)

Action: decide a legal move

Reward: win or lose

(when game is over)



Is there a supervised learner who can tell you how to move?

Examples of Reinforcement Learning

- Fly stunt manoeuvres in a helicopter
- Defeat the world champion at Backgammon
- Manage an investment portfolio
- Control a power station
- Make a humanoid robot walk
- Play many different Atari games better than humans

Rewards

- A reward R_t is a scalar feedback signal
- Indicates how well agent is doing at step t
- The agent's job is to maximise cumulative reward

Reinforcement learning is based on the reward hypothesis

Definition (Reward Hypothesis)

All goals can be described by the maximisation of expected cumulative reward

Do you agree with this statement?

Examples of Rewards

- Fly stunt manoeuvres in a helicopter
  - +ve reward for following desired trajectory
  - −ve reward for crashing
- Defeat the world champion at Backgammon
  - +/−ve reward for winning/losing a game
- Manage an investment portfolio
  - +ve reward for each $ in bank
- Control a power station
  - +ve reward for producing power
  - −ve reward for exceeding safety thresholds
- Make a humanoid robot walk
  - +ve reward for forward motion
  - −ve reward for falling over
- Play many different Atari games better than humans
  - +/−ve reward for increasing/decreasing score

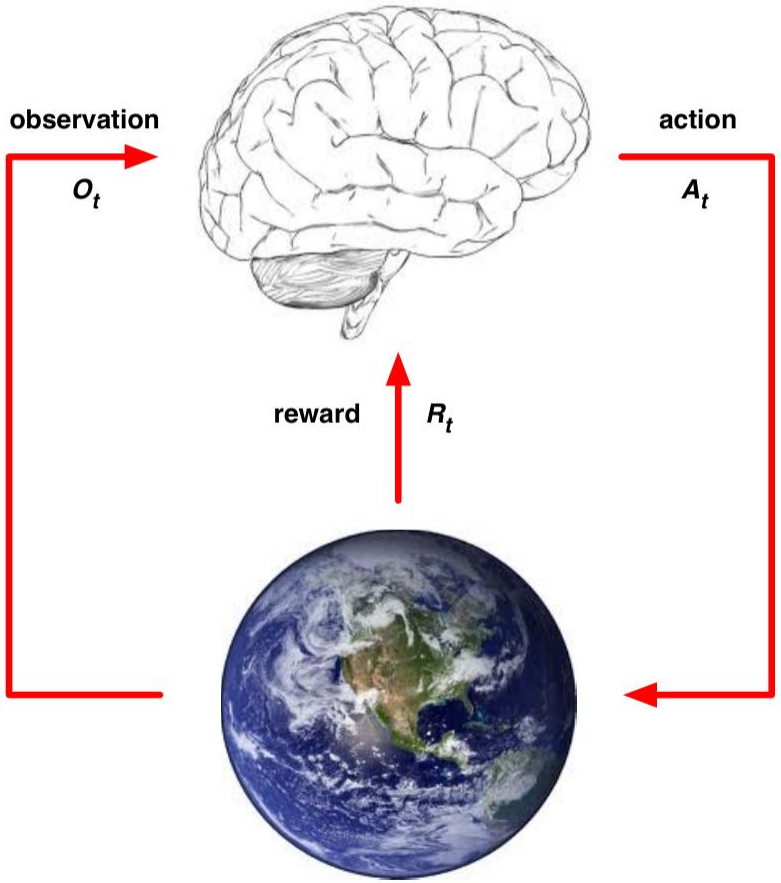
Sequential Decision Making

- Goal: *select actions to maximise total future reward*
- Actions may have long term consequences
- Reward may be delayed
- It may be better to sacrifice immediate reward to gain more long-term reward
- Examples:
  - A financial investment (may take months to mature)
  - Refuelling a helicopter (might prevent a crash in several hours)
  - Blocking opponent moves (might help winning chances many moves from now)

Overview

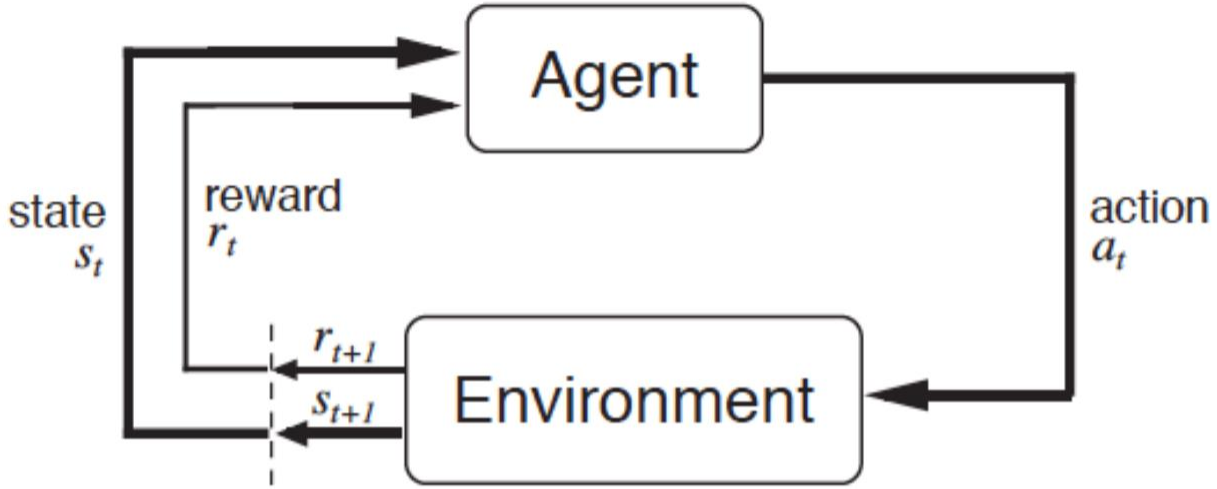
- Introduction to Reinforcement Learning
  - What is RL?
  - **Elements of RL**
  - Q-Learning
- Deep Reinforcement Learning
  - Deep Q-Network (DQN)
  - Policy Gradients (PG)

Agent and Environment



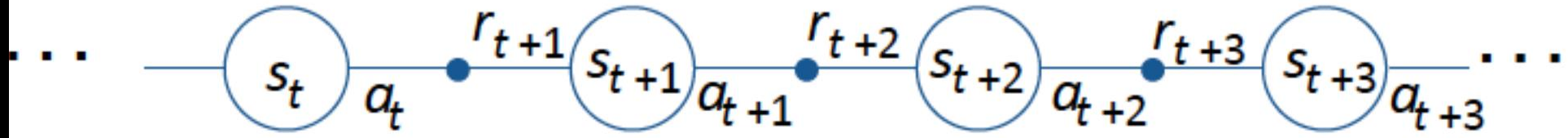
- At each step t the agent:
  - Executes action A_t
  - Receives observation O_t
  - Receives scalar reward R_t
- The environment:
  - Receives action A_t
  - Emits observation O_{t+1}
  - Emits scalar reward R_{t+1}
- t increments at env. step

The Agent-Environment Interface

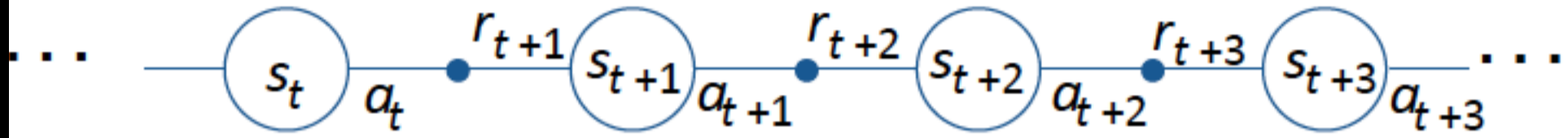


Agent and environment interact at discrete time steps: $t = 0,1,2,\ldots$

- Agent observes state at step t: $s_t \in S$
- produces action at step t: $a_t \in A(s_t)$
- gets resulting reward: $r_{t+1} \in \Re$
- and resulting next state: s_{t+1}



State, Action and Reward



Modeled using a *Markov Decision Process* (MDP)

- Reward and next state are sampled from their respective probability distributions

$$p(r_{t+1} \mid s_t, a_t) \quad \text{and} \quad p(s_{t+1} \mid s_t, a_t)$$

- r_{t+1} and s_{t+1} can be
  - deterministic (only one possible value and state)
  - stochastic (different reward value and next state each time when choosing the same action) – reward value is defined by a probability distribution $p(r|a)$ or $p(r|s,a)$

The Agent Learns a Policy

Policy at step t , π_t **:**

- A mapping from states to probabilities of selecting each possible action

$$\pi_t(s, a) = \textbf{probability that } a_t = a \textbf{ when } s_t = s$$

- Reinforcement learning methods specify how the agent changes its policy as a result of its experience.
- Roughly, the agent's goal is to maximize the total amount of reward it receives over the long run.

Elements of RL

- A ***policy***
  - a mapping from states to probabilities of selecting each possible action
  - defines the learning agent's way of behaving at a given time
  - a simple function, or lookup table, or involve a search process
  - the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior
  - can be stochastic or deterministic
- A ***Reward function***
- A ***Value function***

Elements of RL

- A ***policy***
  - A mapping from states to probabilities of selecting each possible action
- A ***Reward function***
  - maps each perceived state (or state-action pair) of the environment to a single number, a **reward**, indicating the intrinsic desirability of that state.
  - A RL agent's sole objective is to maximize the total reward it receives in the long run.
  - defines what are the good and bad events for the agent in an immediate sense.
- A ***Value function***

Elements of RL

- A ***policy***
  - A mapping from states to probabilities of selecting each possible action
- A ***Reward function***
  - defines what are the good and bad events for the agent in an immediate sense.
- A ***Value function***
  - specifies what is good in the long run
  - the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state
  - values indicate the long-term desirability of states after taking into account the states that are likely to follow, and the rewards available in those states

Value Functions

- The **value of a state** ***s*** under a policy **π**, denoted $V^{\pi}(\boldsymbol{s})$, is the expected return when starting in ***s*** and following **π** thereafter

State-value function for policy π :

$$V^{\pi}(s) = E_{\pi}\left\{R_t \,\middle|\, s_t = s\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right\}$$

- The **value of taking action** ***a*** in state ***s*** under a policy **π**, denoted $V^{\pi}(\boldsymbol{s},\boldsymbol{a})$, as the expected return starting from ***s***, taking the action ***a***, and thereafter following policy **π**:

Action-value function for policy π :

$$Q^{\pi}(s,a) = E_{\pi}\left\{R_t \,\middle|\, s_t = s, a_t = a\right\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s, a_t = a\right\}$$

Bellman Equation for a Policy π

The basic idea:

$$
\begin{aligned}
R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} \cdots \\
&= r_{t+1} + \gamma\left(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} \cdots \right) \\
&= r_{t+1} + \gamma R_{t+1}
\end{aligned}
$$

Therefore:

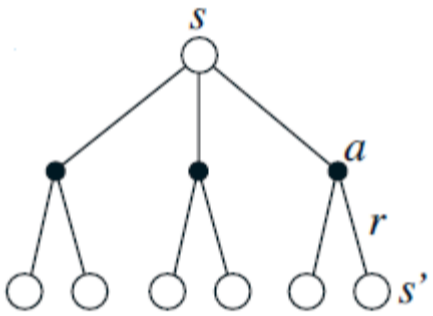
$$
\begin{aligned}
V^{\pi}(s) &= E_{\pi}\{R_t \mid s_t = s\} \\
&= E_{\pi}\{r_{t+1} + \gamma R_{t+1} \mid s_t = s\}
\end{aligned}
$$

Bellman Equation for a Policy π (2)

Without the expectation operator:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} [R^{a}_{ss'} + \gamma V^{\pi}(s')]$$

Backup diagrams



for V^{π}

where $P^{a}_{ss'}$ is **transition probabilities**:

$$P^{a}_{ss'} = \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\} \quad \textbf{for all } s, s' \in S, a \in A(s)$$

and $R^{a}_{ss'}$ is **reward probabilities:**

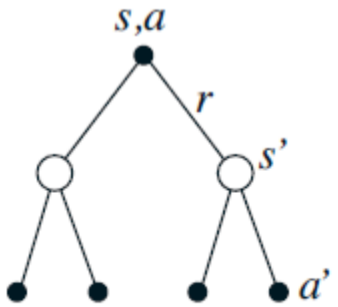
$$R^{a}_{ss'} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\} \quad \textbf{for all } s, s' \in S, a \in A(s)$$

This is a set of equations (in fact, linear), one for each state.
The value function for π is its unique solution.

Bellman Equation for a Policy π (3)

$$
\begin{aligned}
Q^{\pi}(s,a) &= E_{\pi}\{R_t \mid s_t = s, a_t = a\} \\
&= E_{\pi}\{r_{t+1} + \gamma R_{t+1} \mid s_t = s, a_t = a\} \\
&= \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma V^{\pi}(s')]
\end{aligned}
$$

Backup diagrams

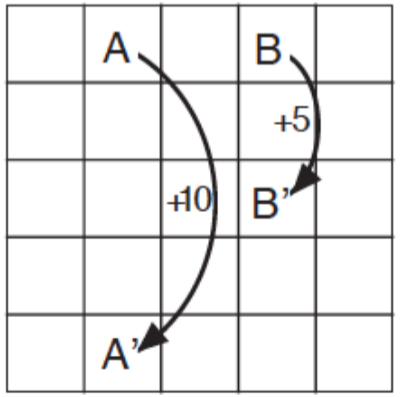


for Q^{π}

Example: Gridworld

- deterministic actions: north, south, east, and west
- reward of -1: actions that would take the agent off the grid
- reward of +10: all actions taking A ->A'
- reward of +5: all actions taking B->B;
- reward of 0: others

A is the best state
But expected return is less than 10 (its immediate reward)



(a)

Actions

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

(b)

$V^{\pi}(s)$

State-value function for equiprobable random policy;

$\gamma = 0.9$

high probability of hitting the edge of the grid

Optimal Value Functions

Solving a reinforcement learning task = finding a policy that achieves a lot of reward over the long run

- For finite MDPs, policies can be **partially ordered**:

$$\pi >= \pi' \text{ if and only if } V^{\pi}(s) >= V^{\pi'}(s) \text{ for all } s \in S$$

- There are always one or more policies that are better than or equal to all the others. These are the **optimal policies**, π^*.
- Optimal policies share the same **optimal state-value function**:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \quad \text{for all } s \in S$$

- Optimal policies share the same **optimal action-value function**:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \quad \text{for all } s \in S \text{ and } a \in A(s)$$

For the state-action pair (s,a), the expected return for taking action a in state s and thereafter following an optimal policy.

Bellman Optimality Equation for V*

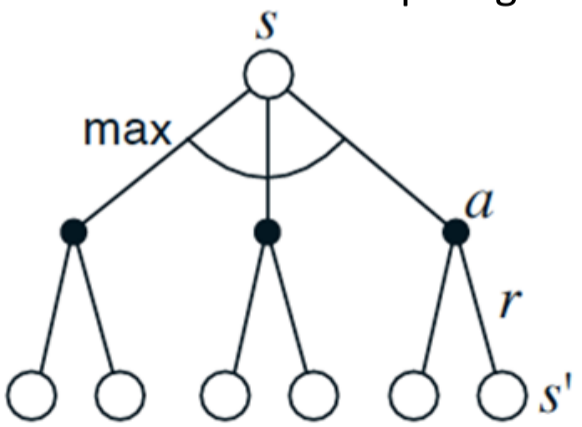
The **value of a state** under an optimal policy must equal the expected return for the best action from that state:

$$
\begin{aligned}
V^*(s) &= \max_{a \in A(s)} Q^*(s,a) \\
&= \max_{a \in A(s)} E_{\pi^*}\{r_{t+1} + \gamma R_{t+1} \mid s_t = s, a_t = a\} \\
&= \max_{a \in A(s)} E\{r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\} \\
&= \max_{a \in A(s)} \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')]
\end{aligned}
$$

- When the environment model parameters, $P^a_{ss'}$ and $R^a_{ss'}$ are known.

V* is the unique solution of this system of nonlinear equations.

Backup diagram

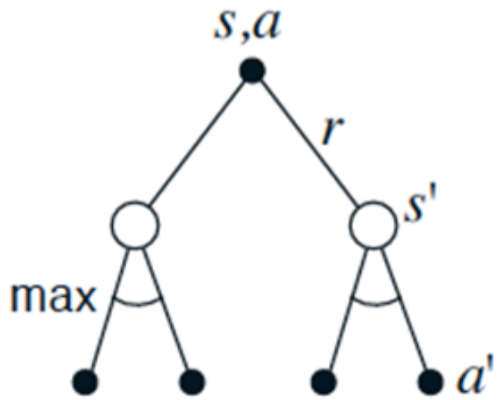


Bellman Optimality Equation for Q*

The Bellman optimality equation for Q* is

$$
\begin{aligned}
Q^*(s,a) &= E\{r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\} \\
&= E\{r_{t+1} + \gamma \max_{a' \in A(s_{t+1})} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a\} \\
&= \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma \max_{a' \in A(s')} Q^*(s', a')]
\end{aligned}
$$

Backup diagram



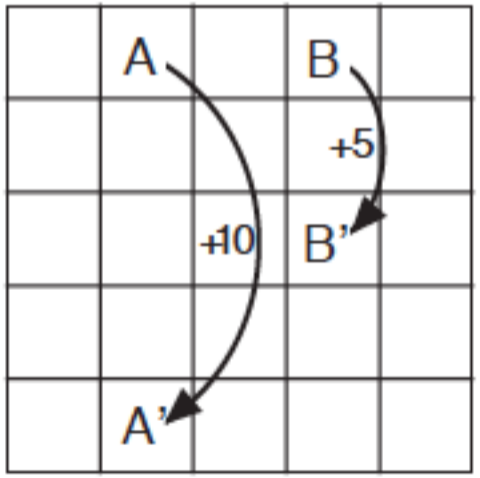
Q* is the unique solution of this system of nonlinear equations.

Why Optimality State-Value Functions are useful

Any policy that is greedy w.r.t. V* is an optimal policy

Therefore, given V*, one-step-ahead search produces the long-term optimal actions.

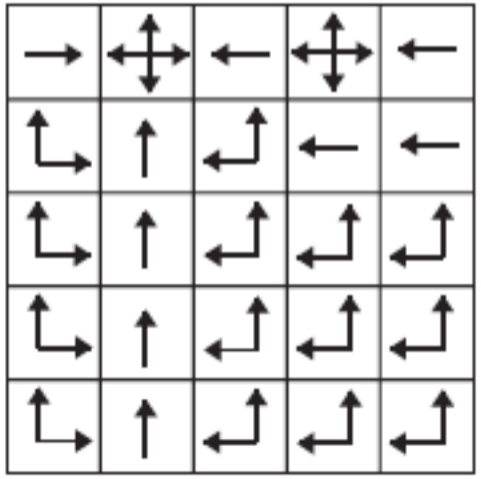
- E.g., back to the example of gridworld:



a) gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

b) V^*



c) π^*

Find Policy from Value

Given V*, the agent does a greedy one-step search:

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')]$$

Given Q*, the agent does not even to do a one-step search:

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} Q^*(s,a)$$

→ Q is more interesting to have

Solving the Bellman Optimality Equation

- Finding an optimal policy by solving the Bellman Optimality Equation requires the following:
  - accurate knowledge of environment dynamics;
  - we have enough space and time to do the computation;
  - the Markov Property
- How much space and time do we need?
  - polynomial in number of states (via dynamic programming methods),
  - BUT, number of states is often huge
- We usually have to settle for approximations.
- Many RL methods can be understood as approximately solving the Bellman Optimality Equation.

Temporal Difference (TD) Learning

- Model is **not** available
- Explore the environment to query the model
  - Deterministic case
  - Non-deterministic case
- ***Temporal Difference*** (TD) algorithm
  - Explore to see the value of next state and reward
  - Update the value of current state by checking the difference between

Current estimate of the value of a state (or a state-action pair) and The value of the next state and the reward received.

Exploration vs Exploitation

Explore in the environment: with Q, how can we choose an action?

Exploration	Exploitation
Discover new possibilities	Refining current procedure
e.g., find another lever that probably gives a higher reward	e.g., keep choosing a lever once it gives immediate reward
Long-term process	Short-term process
A tradeoff between Exploration and Exploitation	

e-greedy method

- Near-greedy action selection rule: ***e*-greedy** methods
  - behave greedily most of the time
  - with small probability *e*, select an action at random, uniformly, independently of the action-value estimates
- **Advantage**: as the number of plays increases, every action will be sampled an infinite number of times, thus ensuring that all the Q(s,a) converge to Q*(s,a).
- Also implies that the probability of selecting the optimal action converges to greater than 1-*e* (to near certainty).

e=0.01, 0.1 …

Softmax Action Selection

- Drawback of e-greedy action selection:

  exploring by choosing equally among all actions.
- **Solution**: vary the action probabilities as a graded function of estimated value
- *Softmax* action selection: choose action *a* on the *t*-th play with probability

$$\frac{\exp(Q_t(a)/\tau)}{\sum_{b=1}^{n}\exp(Q_t(b)/\tau)}$$

where $\tau > 0$, called temperature parameter

  - High temperatures: actions to be all equiprobable
  - Low temperatures: greater difference in selection probability for actions

Overview

- Introduction to Reinforcement Learning
  - What is RL?
  - Elements of RL
  - **Q-Learning**
- Deep Reinforcement Learning
  - Deep Q-Network (DQN)
  - Policy Gradients (PG)

Q Learning algorithm (in deterministic worlds)

In **deterministic** worlds, for each (s,a), there is a single reward and next state possible.

$$Q^*(s,a) = \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \max_{a' \in A(s')} Q^*(s',a')]$$

is thus reduced to

$$Q*(s,a) = r_{t+1} + \gamma \max_{a_{t+1}} Q*(s_{t+1}, a_{t+1})$$

backed-up estimate

which can be used to update Q(s,a) in learning process

Q Learning algorithm (in deterministic worlds)

For each (s,a), initialize Q(s,a) arbitrarily, e.g., $\hat{Q}(s,a) := 0$

Repeat (for each episode):

Initialize *s*

Repeat (for each step of episode):

- Select an action a_{t+1} using policy derived from *Q*, e.g., e-greedy, and execute it
- Observe immediate reward r_{t+1} and new state s_{t+1}
- Update table entry as follows:

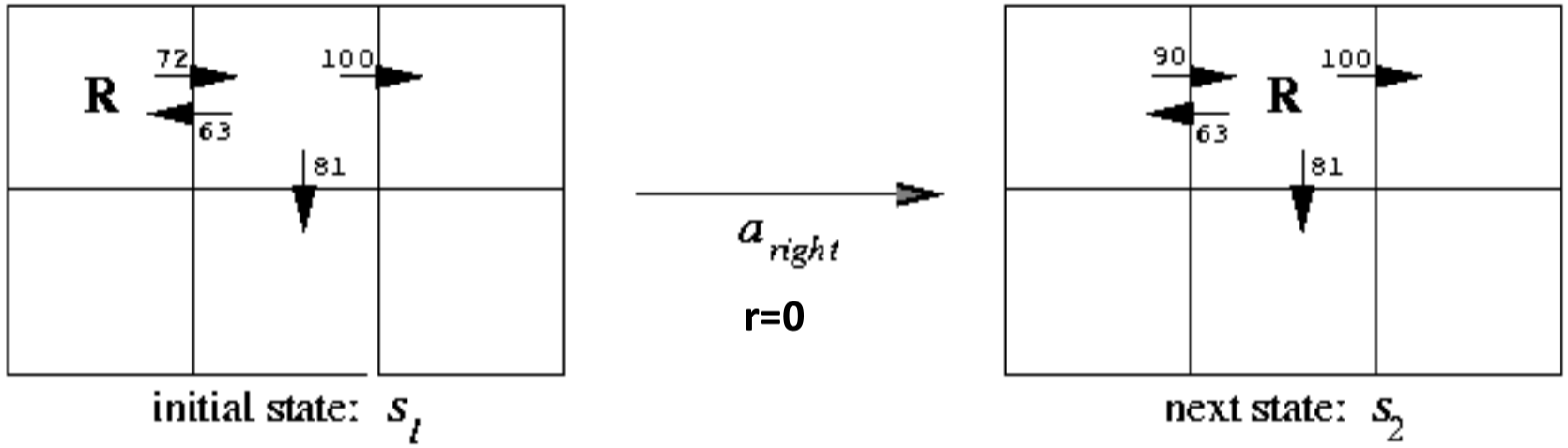
$$\hat{Q}(s_t, a_t) := r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1})$$

- $s := s_{t+1}$

Until *s* is terminal

Example of updating Q

- Given the Q values from a previous iteration on the arrows



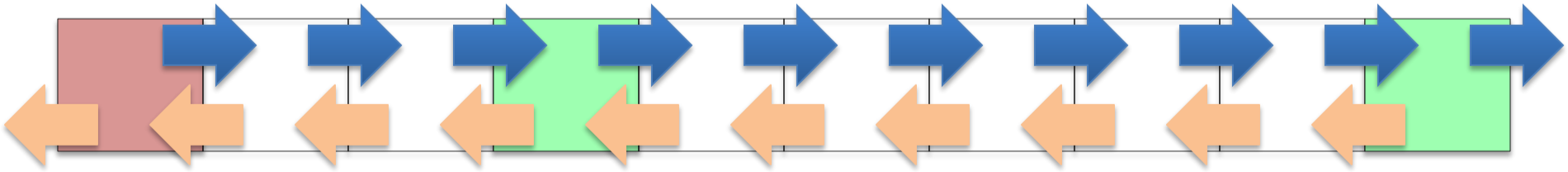
$$\hat{Q}(s_1, right) := r + \gamma \max_{a_{t+1}} \hat{Q}(s_2, a_{t+1})$$

$$= 0 + 0.9 \max(63, 81, 100)$$

$$= 90$$

Example of Q Learning

- 10 states
- Reward +1 entering green squares, -1 entering red, 0 otherwise
- 2 actions: Left and Right (trying to move off the end goes to the other end)
- e=0.25 (e-greedy), 25% probability of moving opposite of the chosen action
- Discount factor $\gamma = 0.9$



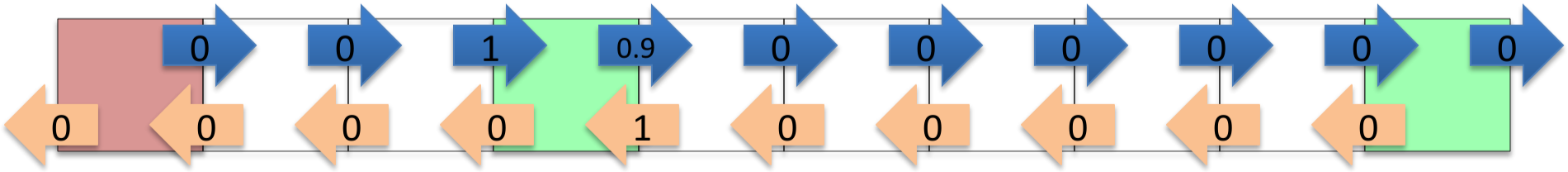
Example of Q Learning

- Update Q(s,a) by

$$\hat{Q}(s_t, a_t) := r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1})$$

in Q Learning algorithm

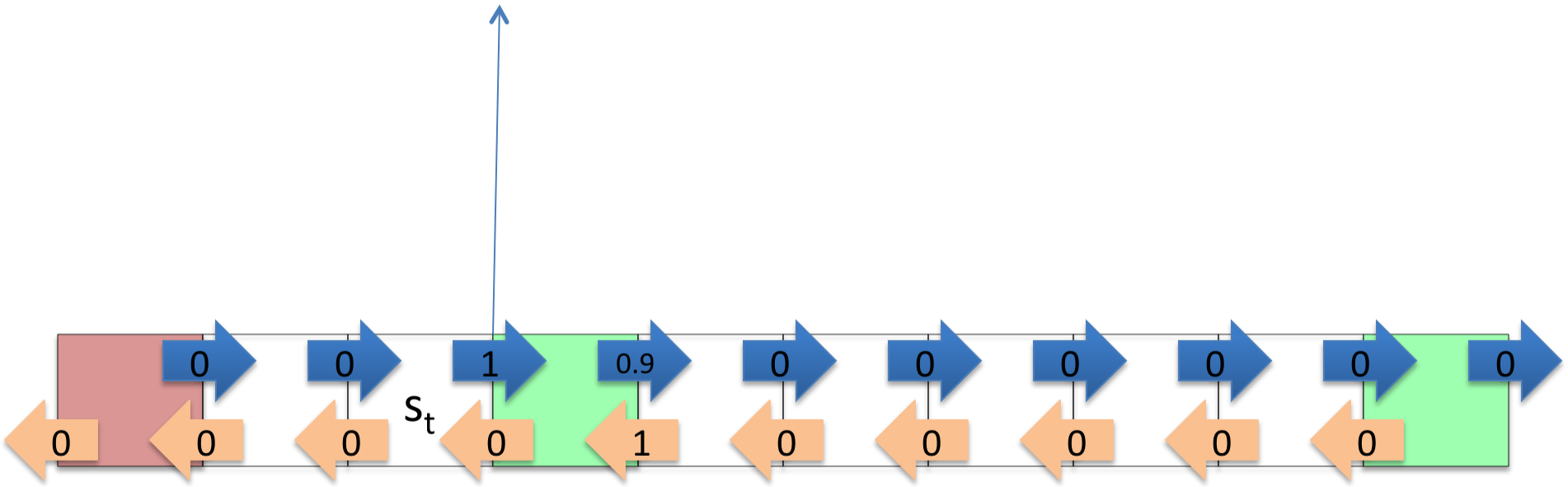
- After a while, what we have is:



Example of Q Learning

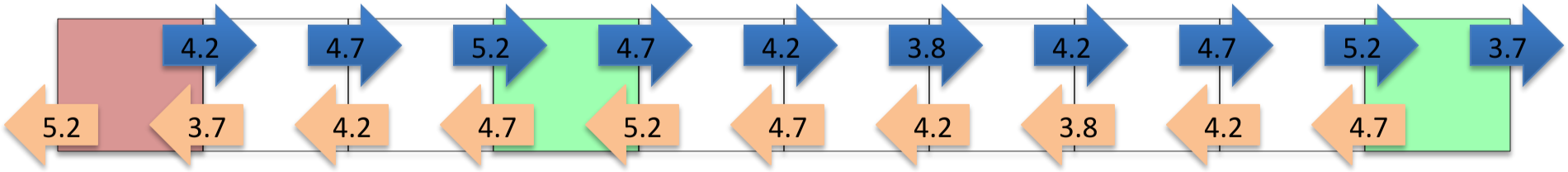
- How to update $Q(s_t, \text{right})$?

$$
\begin{aligned}
Q(s_t, right) &= r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \\
&= 1 + 0.9 * \max(0.9, 0) \\
&= 1.81
\end{aligned}
$$



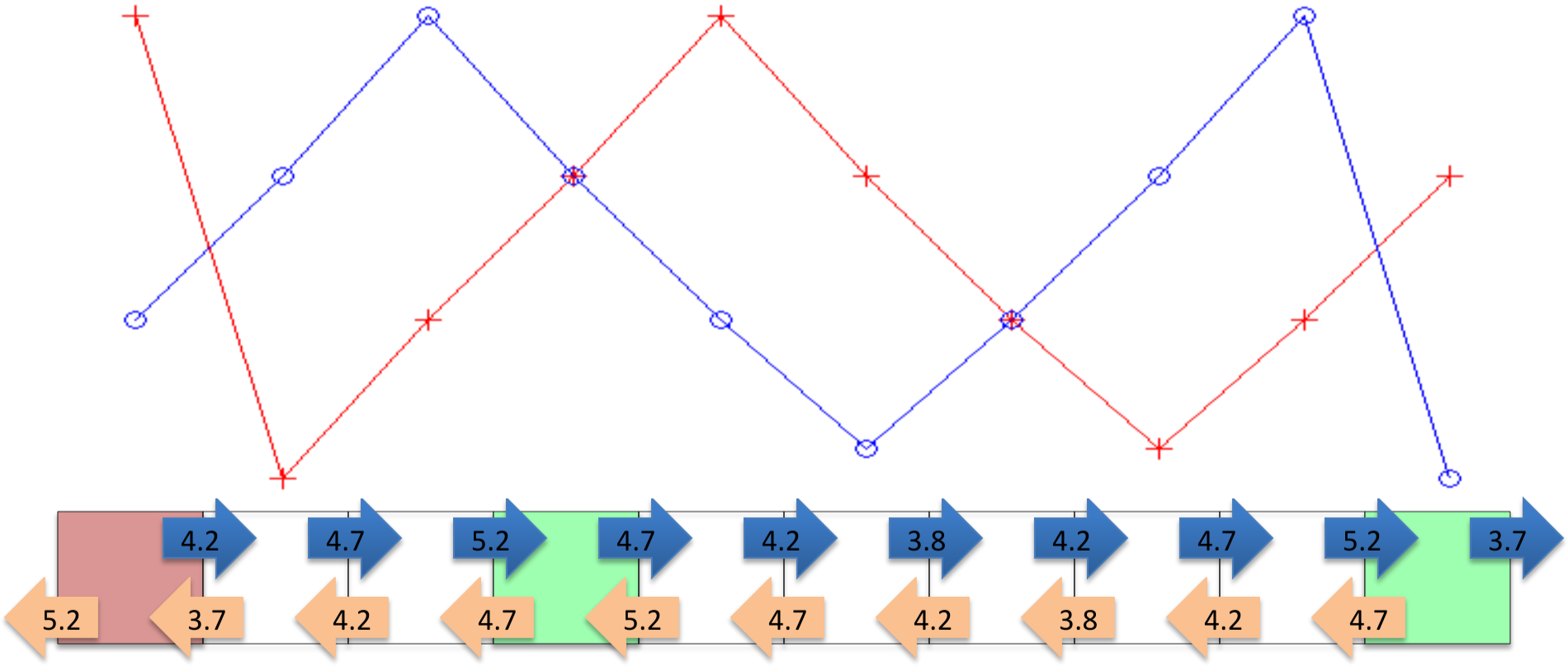
Example of Q Learning

- Finally, get the optimal Q values



Example of Q Learning

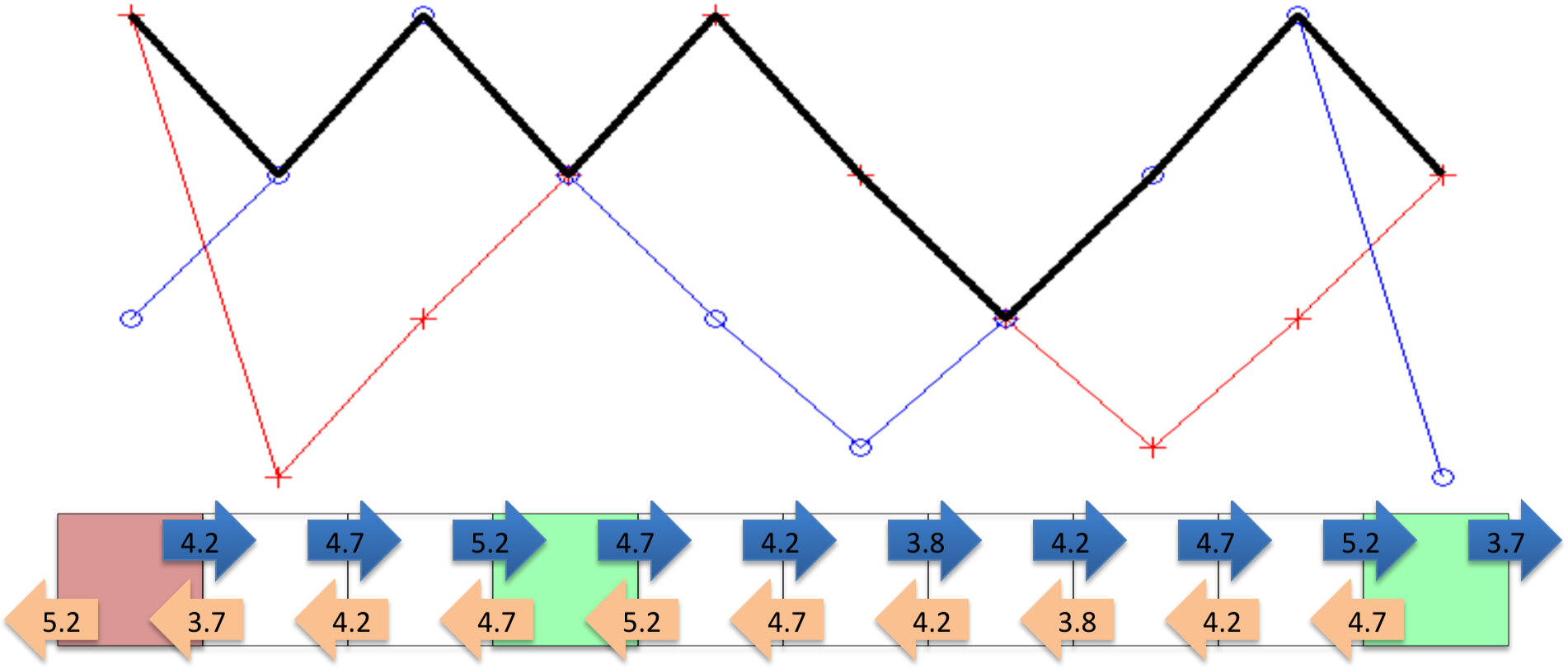
- Plot the Q values, with BLUE for right and RIGHT for left



Example of Q Learning

- What is **the optimal value** in each state? V*(s)=?

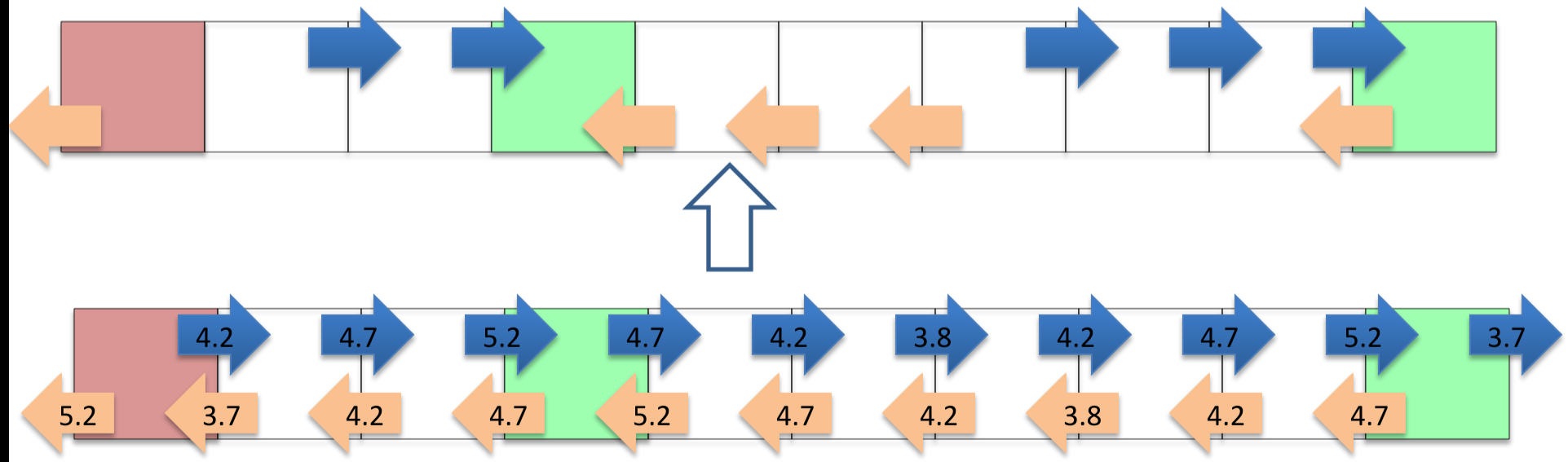
  The MAX shown in BLACK



Example of Q Learning

- What is the **optimal policy** in each state? $\pi^*(s)=?$

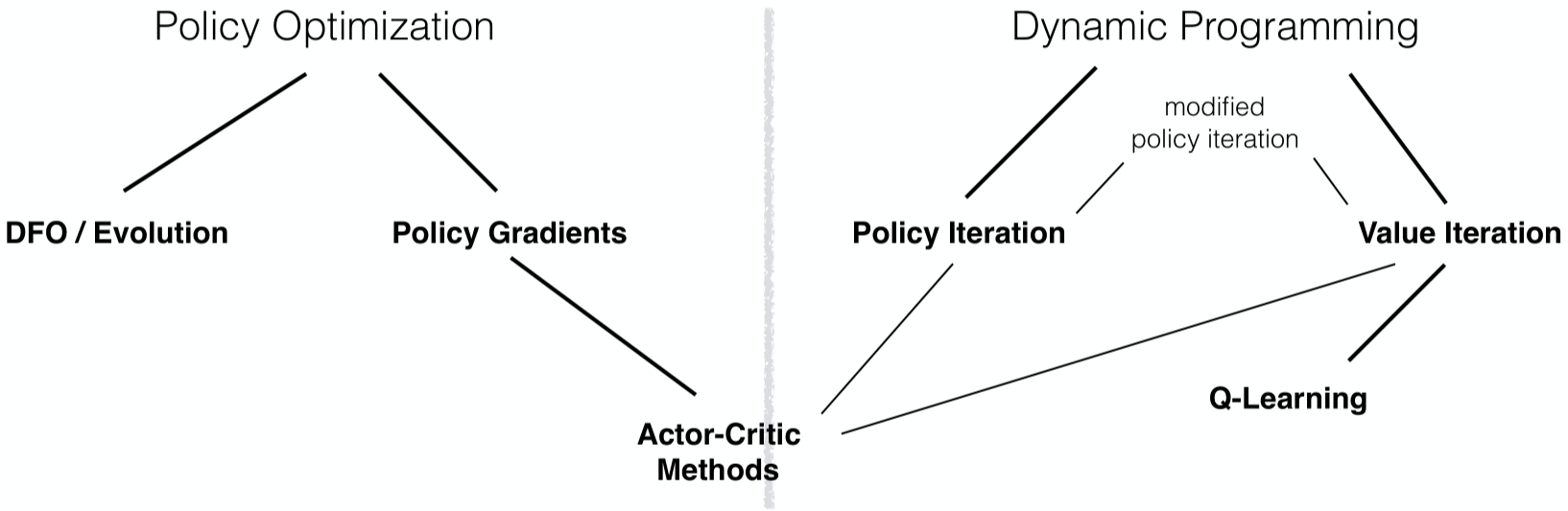
  The action that gives MAX



Overview

- Introduction to Reinforcement Learning
  - What is RL?
  - Elements of RL
  - Q-Learning
- **Deep Reinforcement Learning**
  - Deep Q-Network (DQN)
  - Policy Gradients (PG)

Approaches to RL



What is Deep RL?

- RL using nonlinear function approximators
- Usually, updating parameters with stochastic gradient descent

Overview

- Introduction to Reinforcement Learning
  - What is RL?
  - Elements of RL
  - Q-Learning
- Deep Reinforcement Learning
  - **Deep Q-Network (DQN)**
  - Policy Gradients (PG)

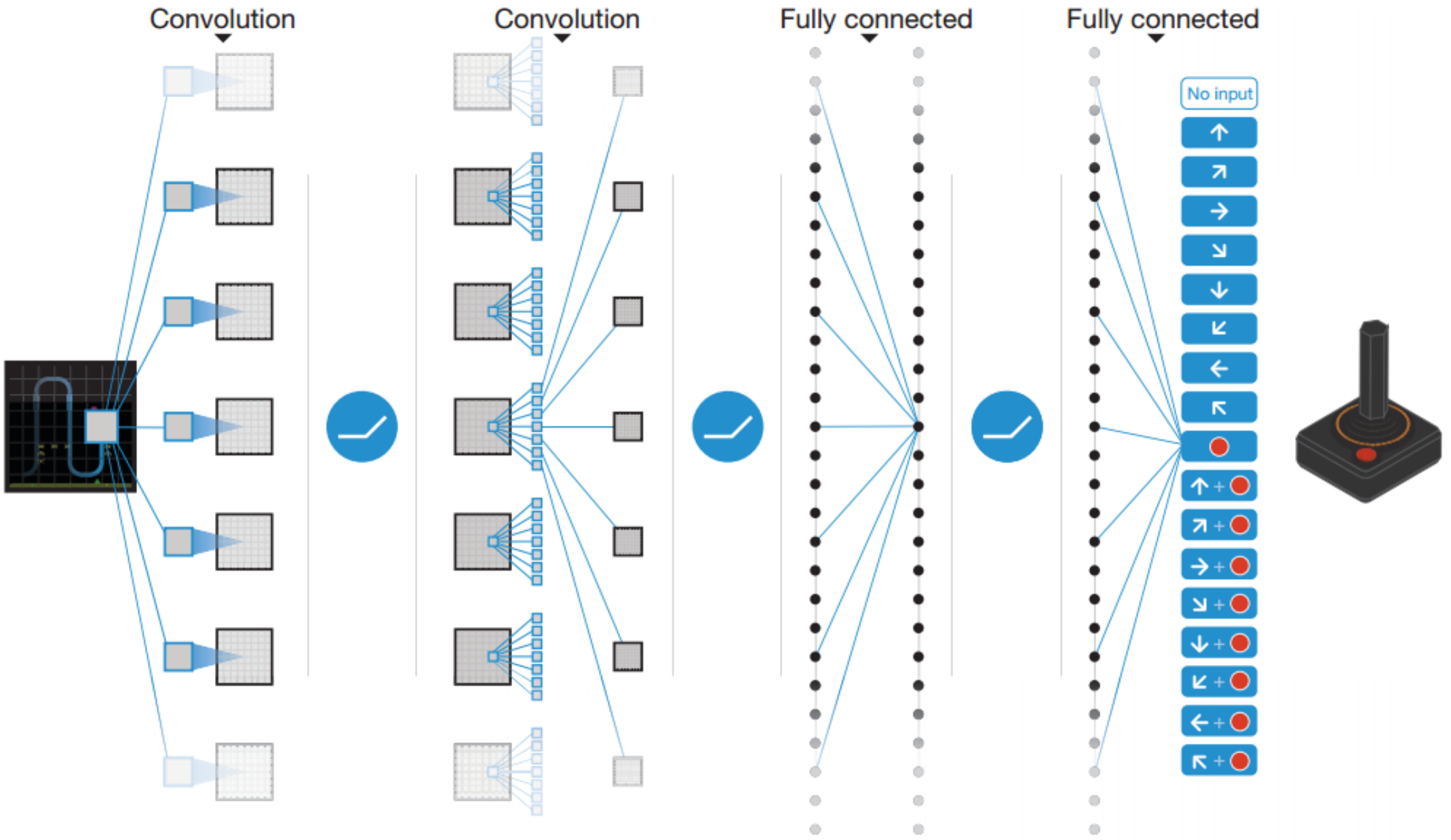
Deep Q-Network

- Use deep neural network to approximate

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \middle| s_t = s,\ a_t = a,\ \pi\right]$$

- The Q-learning update at iteration i uses the following loss function

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim \mathrm{U}(D)} \left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i) \right)^2 \right]$$


Convolution
Convolution
Fully connected
Fully connected
No input

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For t = 1,T **do**

With probability ε select a random action a_t

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

Execute action a_t in emulator and observe reward r_t and image x_{t+1}

Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

Policy Gradients (PG)

- Recently, most people prefer to use Policy Gradients, including the authors of the original DQN
- PG is preferred because it is end-to-end
- There's an explicit policy and a principled approach that directly optimizes the expected reward
- Example: learn to play an ATARI game (Pong!) with PG, from scratch, from pixels, with a deep neural network

Policy Gradient Methods: Overview

Problem:

$$\text{maximize } E[R \mid \pi_\theta]$$

Intuitions: collect a bunch of trajectories, and ...

1. Make the good trajectories more probable
2. Make the good actions more probable (actor-critic, GAE)
3. Push the actions towards good actions (DPG, SVG)

Score Function Gradient Estimator

- Consider an expectation $E_{x \sim p(x \mid \theta)}[f(x)]$. Want to compute gradient wrt θ

$$\begin{aligned}
\nabla_\theta E_x[f(x)] &= \nabla_\theta \int \mathrm{d}x \; p(x \mid \theta) f(x) \\
&= \int \mathrm{d}x \; \nabla_\theta p(x \mid \theta) f(x) \\
&= \int \mathrm{d}x \; p(x \mid \theta) \frac{\nabla_\theta p(x \mid \theta)}{p(x \mid \theta)} f(x) \\
&= \int \mathrm{d}x \; p(x \mid \theta) \nabla_\theta \log p(x \mid \theta) f(x) \\
&= E_x[f(x) \nabla_\theta \log p(x \mid \theta)].
\end{aligned}$$

- Last expression gives us an unbiased gradient estimator. Just sample $x_i \sim p(x \mid \theta)$, and compute $\hat{g}_i = f(x_i) \nabla_\theta \log p(x_i \mid \theta)$.
- Need to be able to compute and differentiate density $p(x \mid \theta)$ wrt θ

Derivation via Importance Sampling

Alternate Derivation Using Importance Sampling

$$\mathbb{E}_{x\sim\theta}\left[f(x)\right] = \mathbb{E}_{x\sim\theta_{\text{old}}}\left[\frac{p(x\mid\theta)}{p(x\mid\theta_{\text{old}})}f(x)\right]$$

$$\nabla_\theta\mathbb{E}_{x\sim\theta}\left[f(x)\right] = \mathbb{E}_{x\sim\theta_{\text{old}}}\left[\frac{\nabla_\theta p(x\mid\theta)}{p(x\mid\theta_{\text{old}})}f(x)\right]$$

$$\nabla_\theta\mathbb{E}_{x\sim\theta}\left[f(x)\right]\big|_{\theta=\theta_{\text{old}}} = \mathbb{E}_{x\sim\theta_{\text{old}}}\left[\frac{\nabla_\theta p(x\mid\theta)\big|_{\theta=\theta_{\text{old}}}}{p(x\mid\theta_{\text{old}})}f(x)\right]$$

$$= \mathbb{E}_{x\sim\theta_{\text{old}}}\left[\nabla_\theta \log p(x\mid\theta)\big|_{\theta=\theta_{\text{old}}}f(x)\right]$$

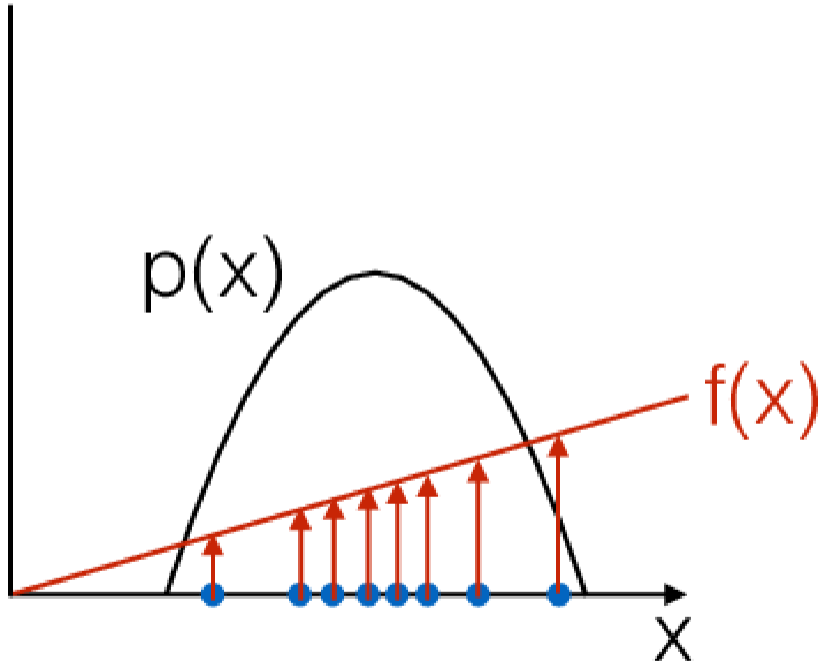
Score Function Gradient Estimator: Intuition

$$\hat{g}_i = f(x_i) \nabla_\theta \log p(x_i \mid \theta)$$

- Let's say that $f(x)$ measures how good the sample x is.
- Moving in the direction $\hat{g}_i$ pushes up the logprob of the sample, in proportion to how good it is
- *Valid even if $f(x)$ is discontinuous, and unknown, or sample space (containing x) is a discrete set*

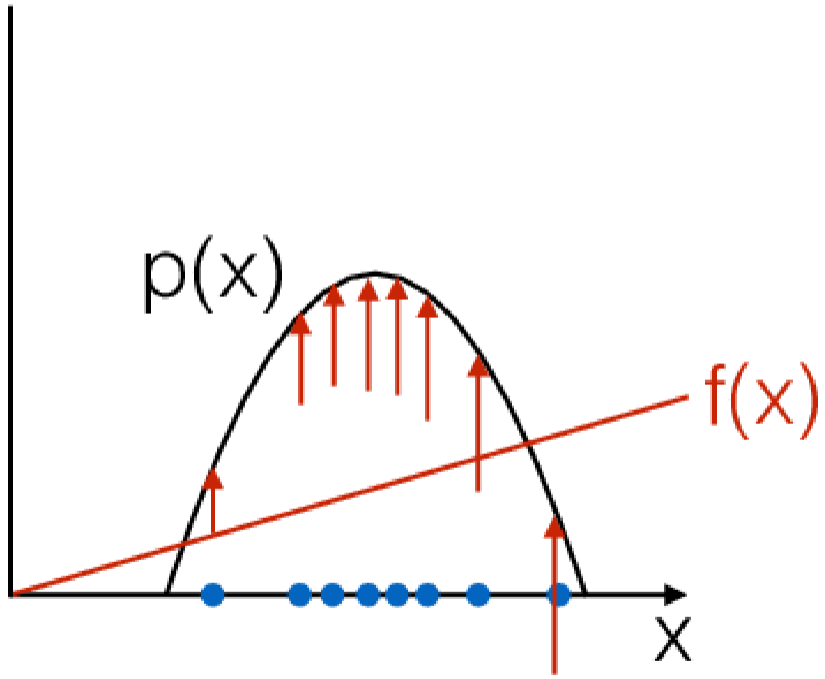
Score Function Gradient Estimator: Intuition

$$\hat{g}_i = f(x_i)\nabla_\theta \log p(x_i \mid \theta)$$



Score Function Gradient Estimator: Intuition

$$\hat{g}_i = f(x_i)\nabla_\theta \log p(x_i \mid \theta)$$



Score Function Gradient Estimator for Policies

- Now random variable x is a whole trajectory $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$

$$\nabla_\theta E_\tau[R(\tau)] = E_\tau[\nabla_\theta \log p(\tau \mid \theta) R(\tau)]$$

- Just need to write out $p(\tau \mid \theta)$:

$$p(\tau \mid \theta) = \mu(s_0) \prod_{t=0}^{T-1} [\pi(a_t \mid s_t, \theta) P(s_{t+1}, r_t \mid s_t, a_t)]$$

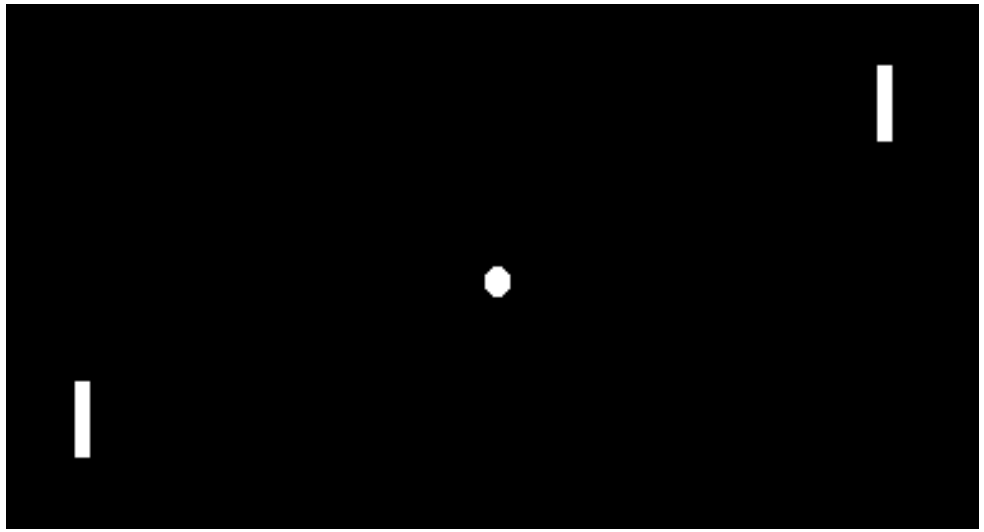
$$\log p(\tau \mid \theta) = \log \mu(s_0) + \sum_{t=0}^{T-1} [\log \pi(a_t \mid s_t, \theta) + \log P(s_{t+1}, r_t \mid s_t, a_t)]$$

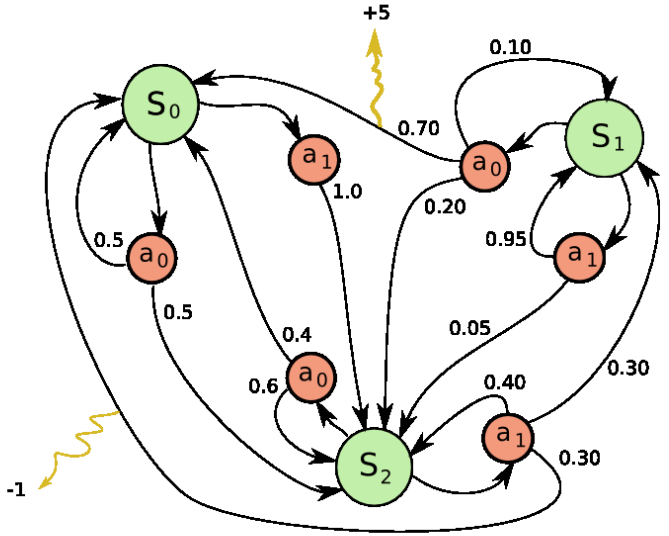
$$\nabla_\theta \log p(\tau \mid \theta) = \nabla_\theta \sum_{t=0}^{T-1} \log \pi(a_t \mid s_t, \theta)$$

$$\nabla_\theta \mathbb{E}_\tau [R] = \mathbb{E}_\tau \left[R \nabla_\theta \sum_{t=0}^{T-1} \log \pi(a_t \mid s_t, \theta) \right]$$

- Interpretation: using good trajectories (high R) as supervised examples in classification / regression

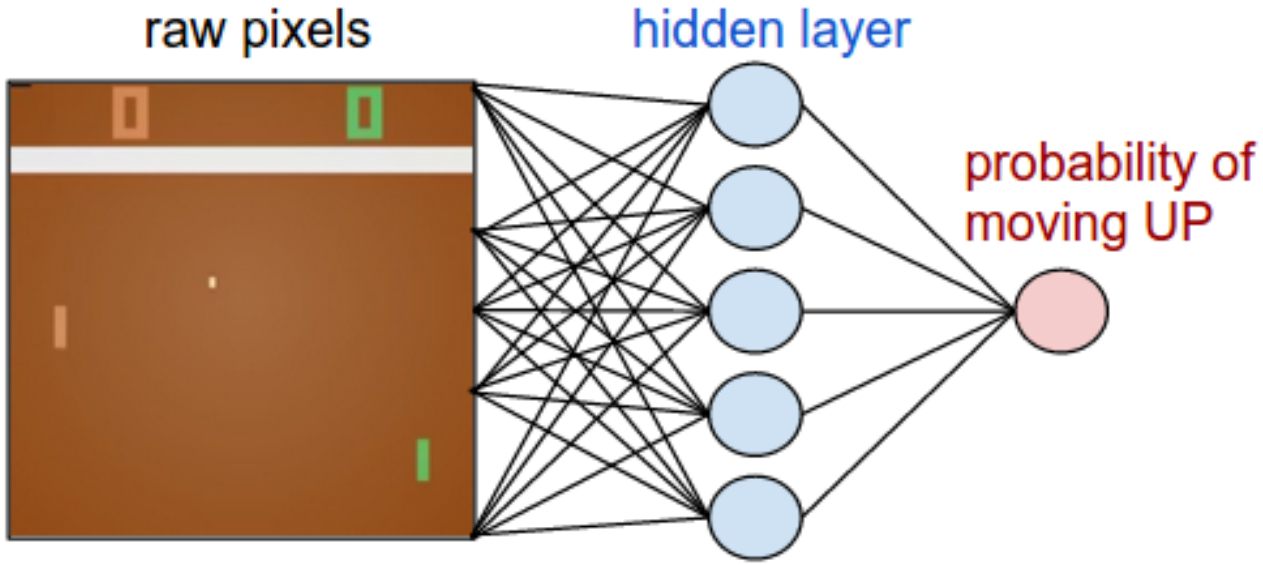
Example - Pong from pixels





Left: The game of Pong. **Right:** Pong is a special case of a Markov Decision Process (MDP): A graph where each node is a particular game state and each edge is a possible (in general probabilistic) transition. Each edge also gives a reward, and the goal is to compute the optimal way of acting in any state to maximize rewards.

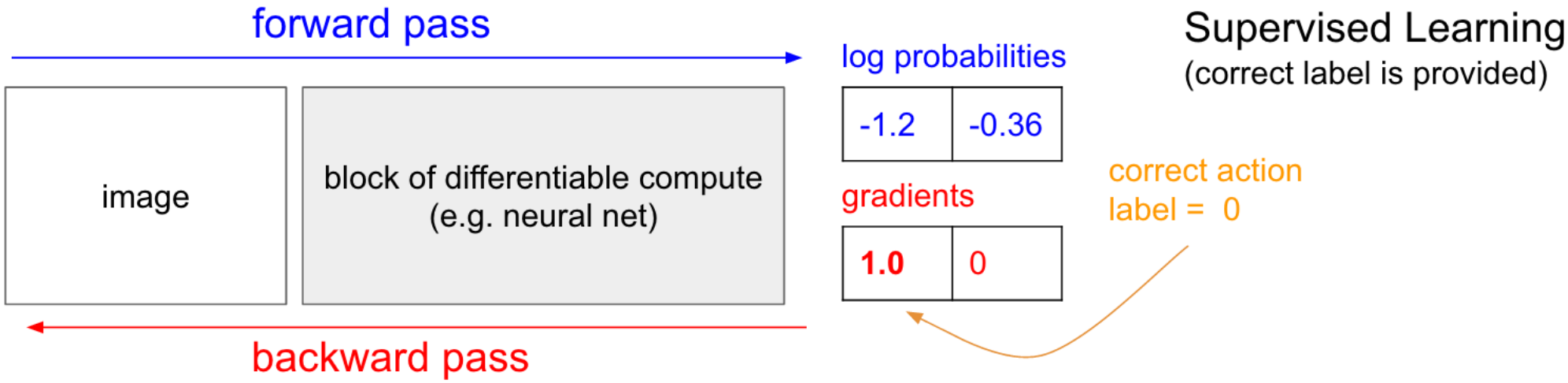
Policy Network




```
h = np.dot(W1, x) # compute hidden layer neuron activations
h[h<0] = 0 # ReLU nonlinearity: threshold at zero
logp = np.dot(W2, h) # compute log probability of going up
p = 1.0 / (1.0 + np.exp(-logp)) # sigmoid function (gives probability of going up)
```

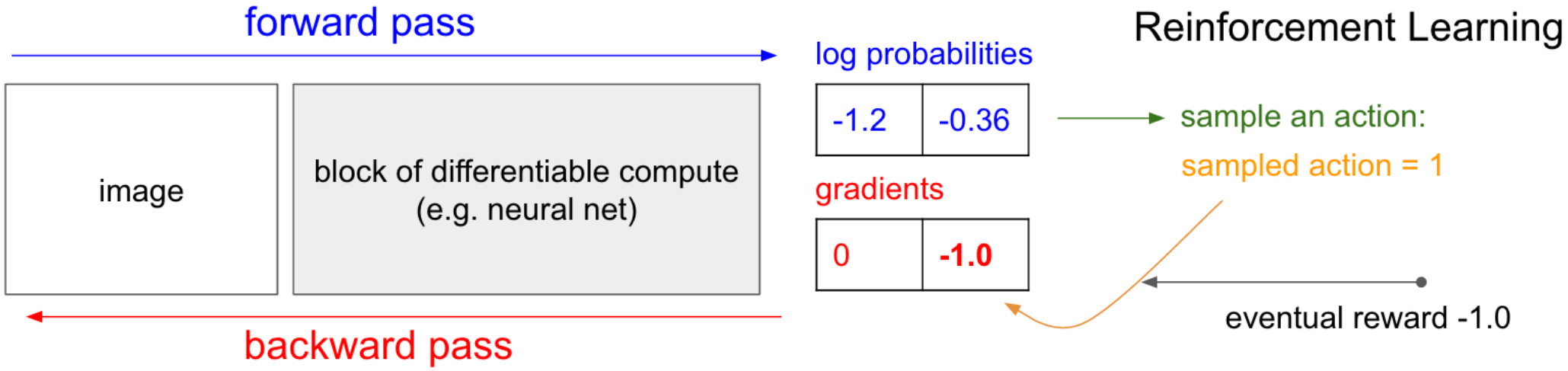
Supervised Learning

- Feed image to the network and get probabilities (e.g. 30% up, 70% down)
- Suppose the label indicates that the correct action is to go UP
- Enter gradient of 1.0 on the log probability of UP and run backprop
- Update parameters with gradient vector $\nabla_w \log p\,(y=\text{UP} \mid x)$ –> Network slightly more likely to predict UP when it sees a very similar image in the future



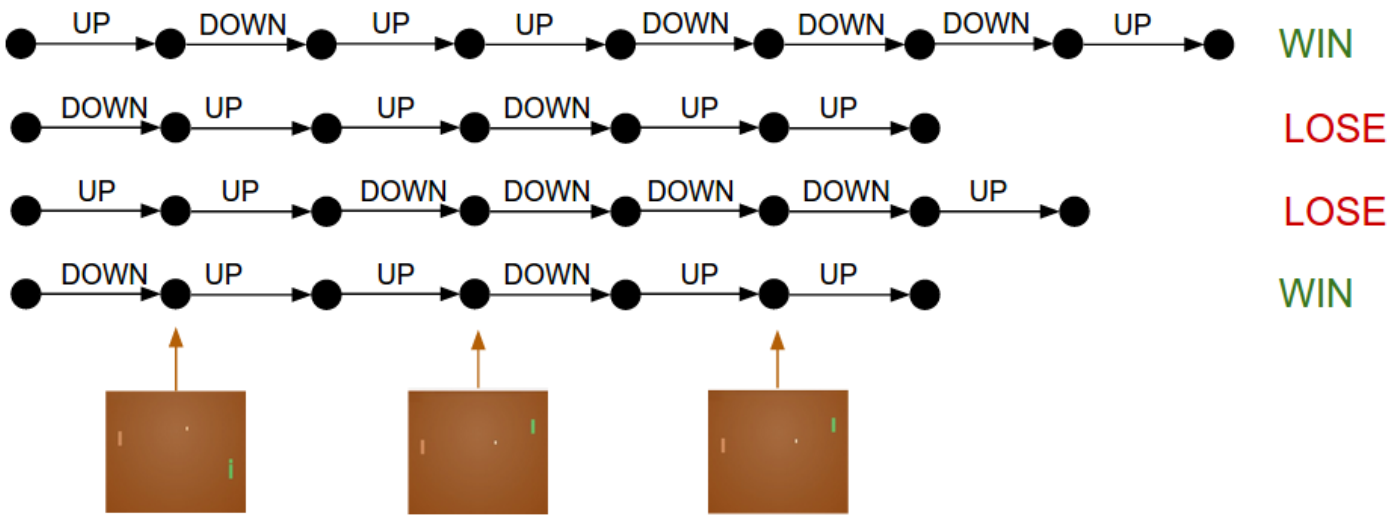
Policy Gradients

- Sample an action from this distribution; E.g. suppose we sample DOWN
- Wait until the end of the game, then take the reward we get (+1 if we won or -1 if we lost)
- Enter that scalar as the gradient for the action we have taken (DOWN in this case)
- Backprop we will find a gradient that *discourages* the network to take the DOWN action for that input in the future



Training protocol

- Initialize the policy network with some W1, W2 and play 100 games of Pong (“rollouts”)
- Suppose we won 12 games and lost 88
- Take all 200*12 = 2400 decisions of the winning games and do a positive update
- Take the other 200*88 = 17600 decisions of the losing games and do a negative update



- Each black circle is a game state
- Each arrow is a transition, annotated with sampled action
- Slightly encourage every single action in the two games we won
- Slightly discourage every single action in the two games we lost

Discounted Reward

- In a more general RL setting we would receive some reward r_t at every time step
- Common choice is to use a discounted reward, so the "eventual reward" is:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$ (later rewards are exponentially less important)

- Important trick: normalize returns (e.g. subtract mean, divide by standard deviation) before doing backprop

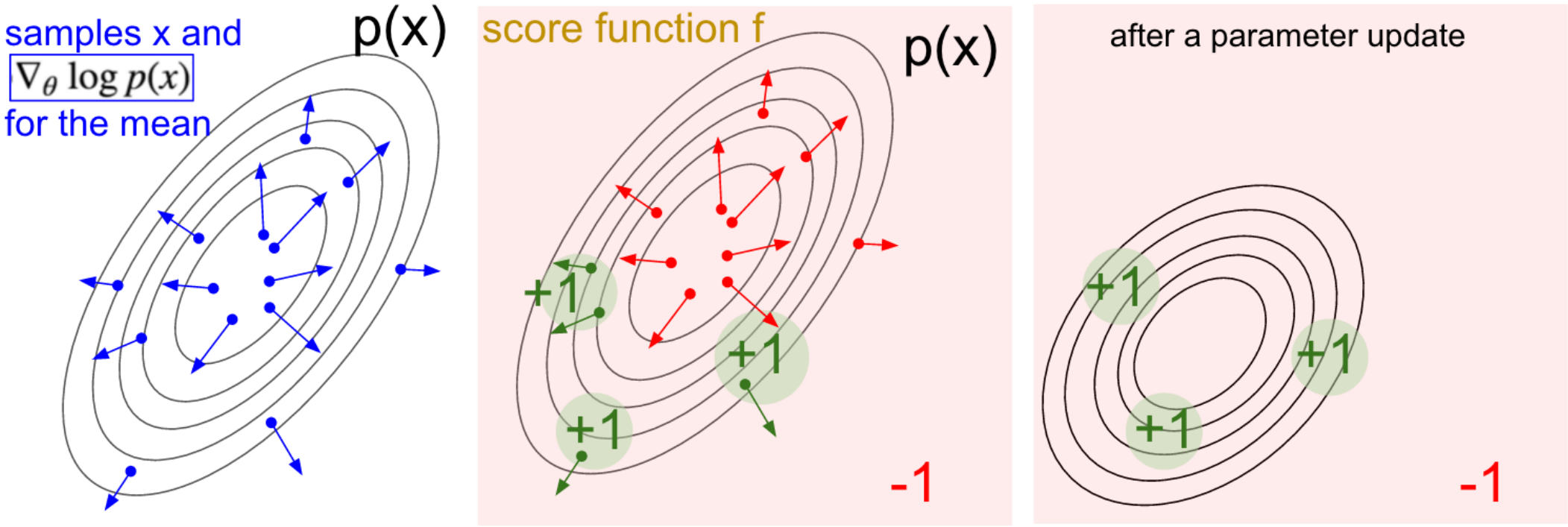
→ encourage and discourage roughly half of the performed actions

→ mathematically this can be interpreted as a way of controlling the variance of the policy gradient estimator

Deriving Policy Gradients

$$\begin{aligned}
\nabla_\theta E_x[f(x)] &= \nabla_\theta \sum_x p(x) f(x) && \text{definition of expectation} \\
&= \sum_x \nabla_\theta p(x) f(x) && \text{swap sum and gradient} \\
&= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) && \text{both multiply and divide by } p(x) \\
&= \sum_x p(x) \nabla_\theta \log p(x) f(x) && \text{use the fact that } \nabla_\theta \log(z) = \frac{1}{z} \nabla_\theta z \\
&= E_x[f(x) \nabla_\theta \log p(x)] && \text{definition of expectation}
\end{aligned}$$

Visualizing Policy Gradients

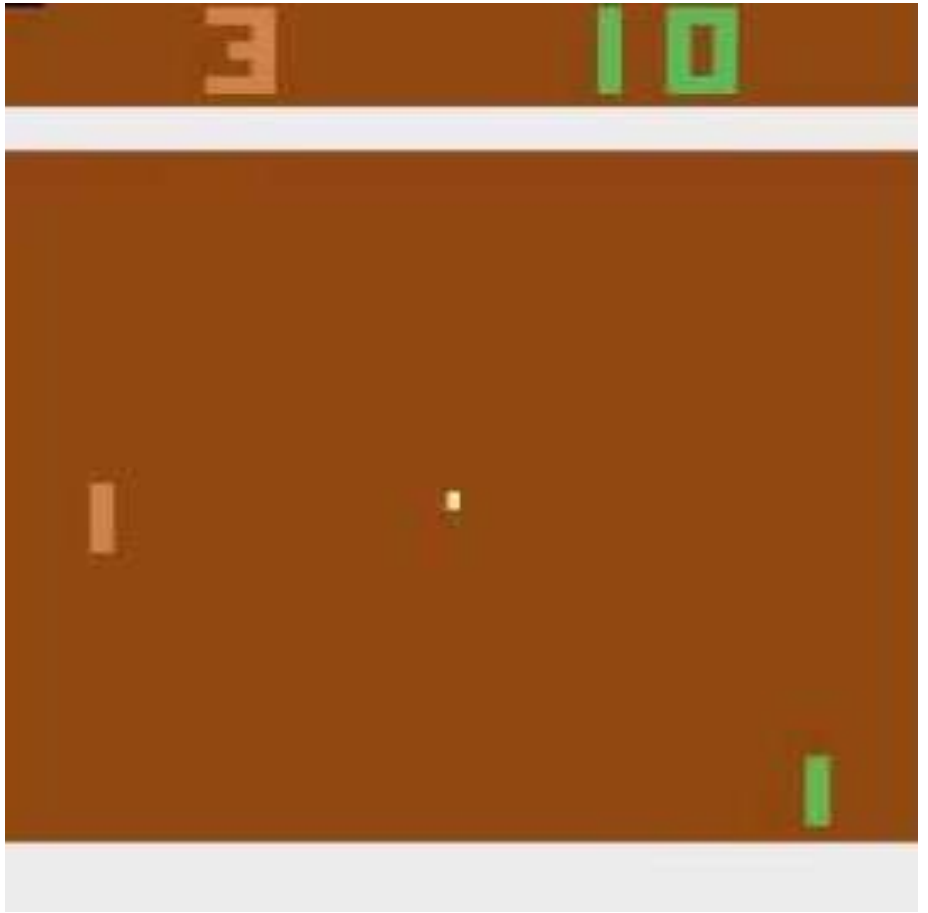


Learning

- Train 2-layer policy network with 200 hidden layer units with batches of 10 episodes (play pong until one player reaches 21 points)
- The total number of episodes was approximately 8,000, so the algorithm played roughly 200,000 Pong games and made a total of ~800 updates

→ Policy that is slightly better than the AI player

→ If you train on GPU with ConvNets and also optimize hyperparameters properly you can consistently dominate the AI player

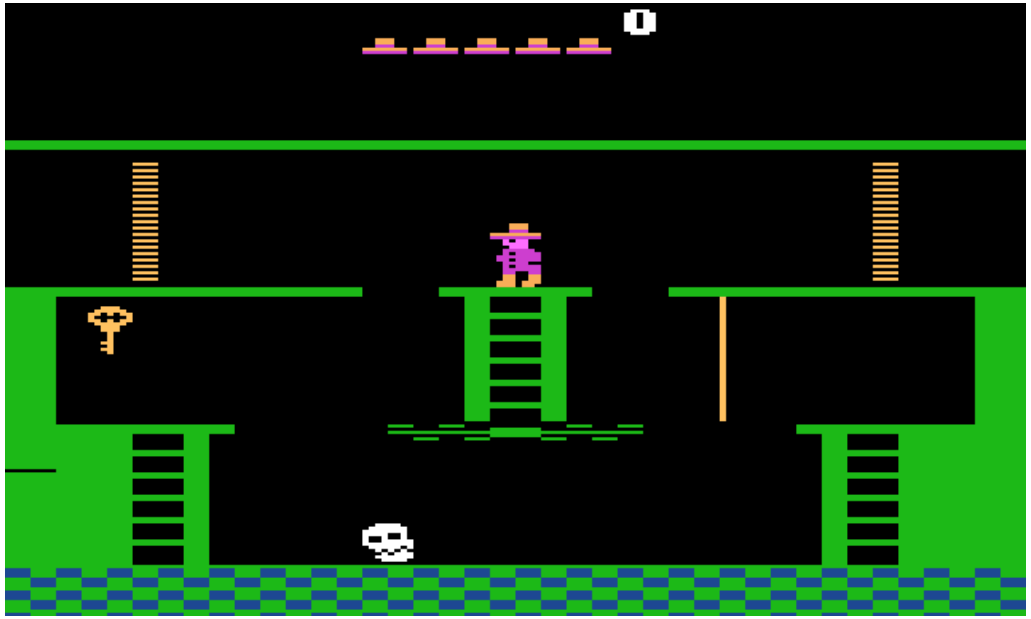


Human vs. Machine

- Human understands the objective of the game and infers the reward.
- Human brings in a huge amount of prior knowledge, such as intuitive physics and intuitive psychology.
- Humans build a rich, abstract model and plan within it. Humans can figure out what is likely to give rewards without ever actually experiencing the rewarding or unrewarding transition.
- RL machine assumes an arbitrary reward function and updates it through environment interactions.
- RL machine starts from scratch and policy gradients are a brute force solution, where the correct actions are eventually discovered and internalized into a policy.
- RL machine actually has to experience a positive reward, and experience it very often in order to eventually and slowly shift the policy parameters towards repeating moves that give high rewards

What are RL machines good at?

- RL machines excel in games with frequent reward signals, that requires precise play, fast reflexes, and not too much long-term planning
- RL machines fail in games where a rich, abstract model of the game is necessary



References & Resources

- Playing Atari with Deep Reinforcement Learning (Archive, 2013)
- High-Dimensional Continuous Control Using Generalized Advantage Estimation (Archive, 2015)
- Human-level control through deep reinforcement learning (Nature, 2015)
- Gradient Estimation Using Stochastic Computation Graphs (NIPS, 2015)
- Asynchronous Methods for Deep Reinforcement Learning (Archive, 2016)
- http://rll.berkeley.edu/deeprlcourse/
- http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html
- https://www.nervanasys.com/demystifying-deep-reinforcement-learning/
- http://karpathy.github.io/2016/05/31/rl/