An Introduction to (Multi-Agent) Reinforcement Learning for Complex Systems

Introduction to the Course
Overview of RL Basics

Mirco Musolesi

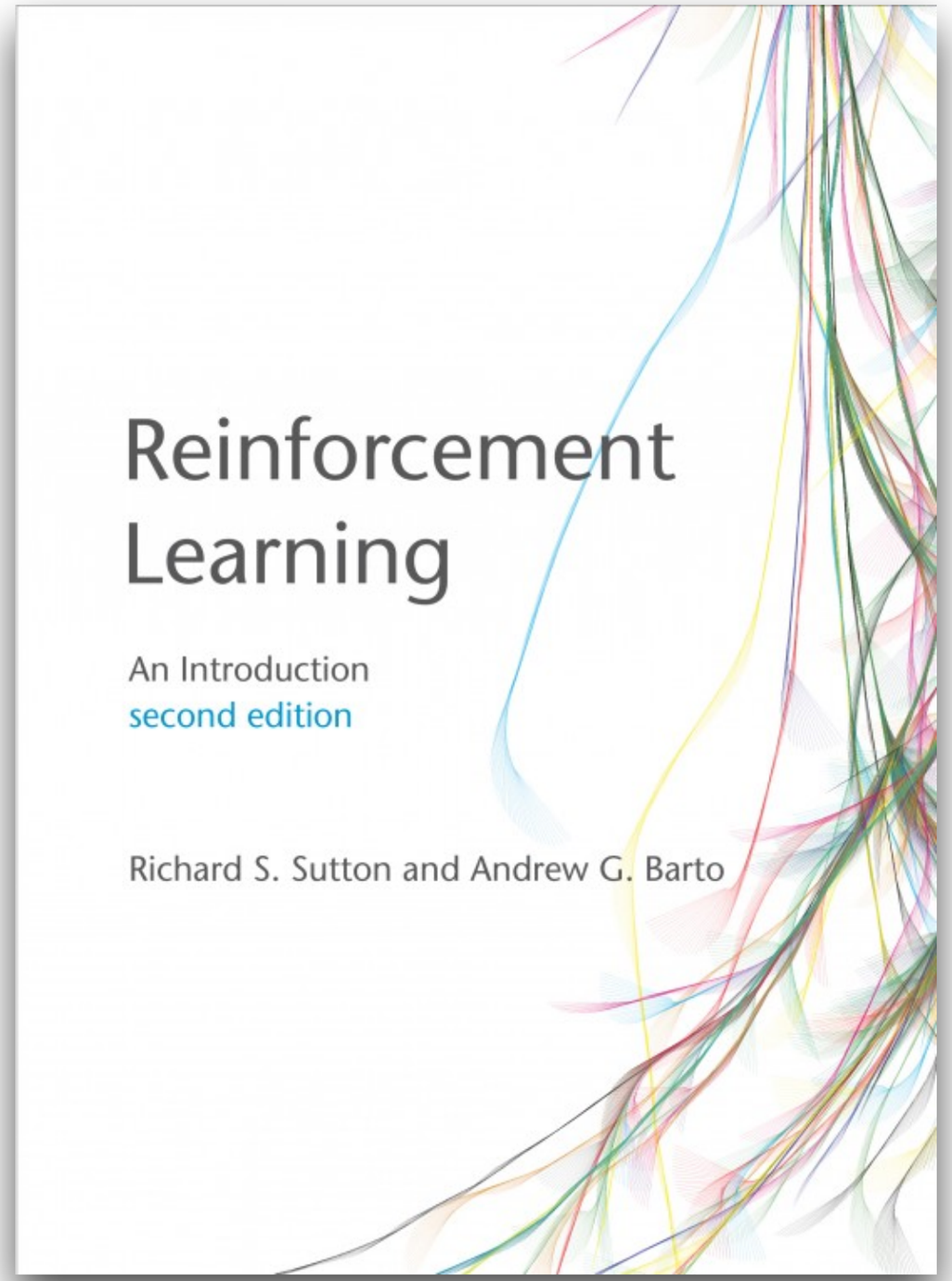
mircomusolesi@acm.org

An Introduction to (Multi-Agent) Reinforcement Learning for Complex Systems

- In this module we will cover core topics in Reinforcement Learning.
- I will provide with an overview of both theoretical foundations and applications. We will discuss key recent papers in this area and we will outline the open challenges in this field.

An Introduction to (Multi-Agent) Reinforcement Learning for Complex Systems

- Mirco Musolesi
- W: https://www.mircomusolesi.org
- E: mircomusolesi@acm.org



http://incompleteideas.net/book/the-book.html

Introduction to Reinforcement Learning

- Key idea: a natural way of thinking about learning is learning through interaction with the external world.

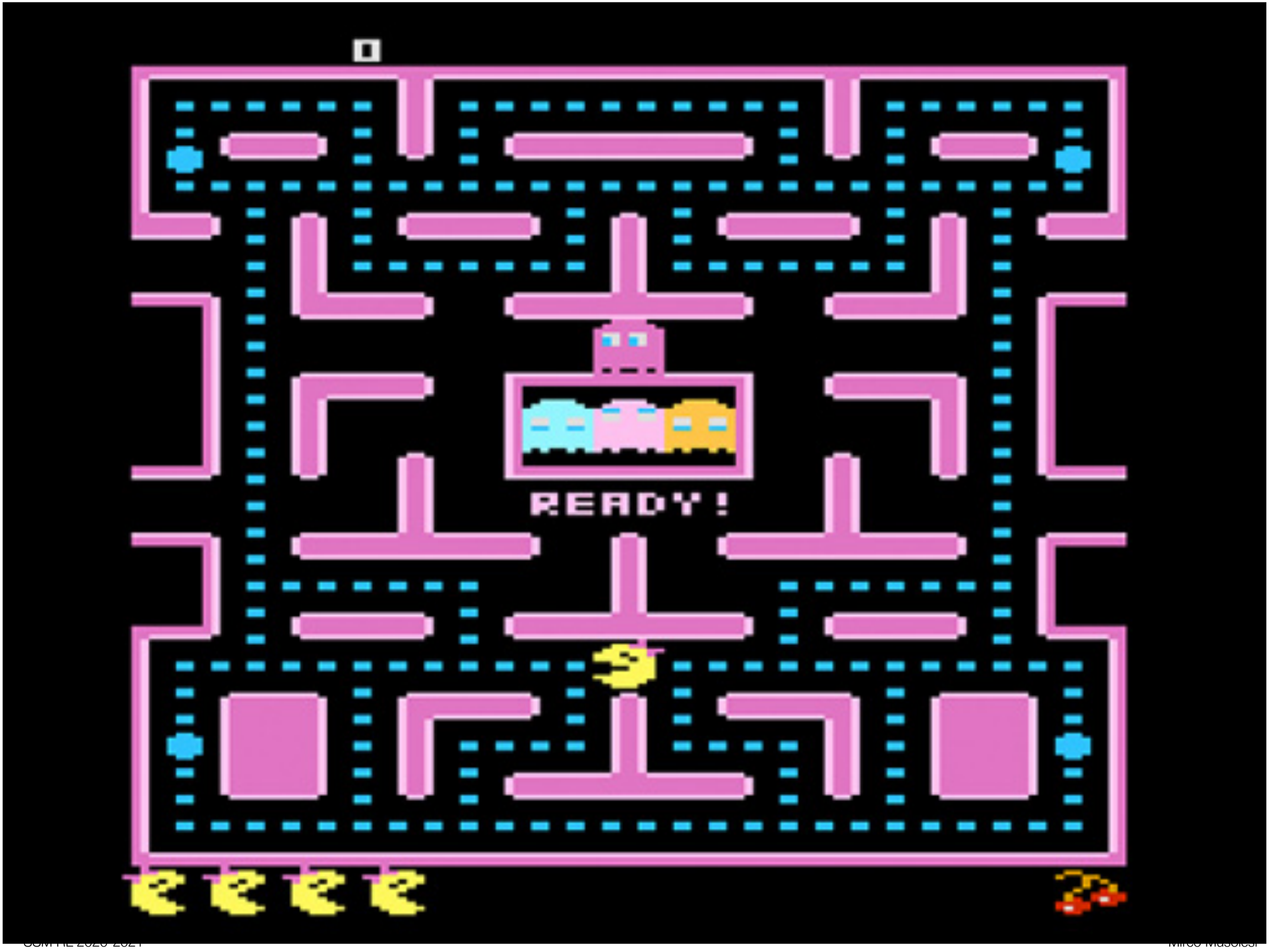
- Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence.

- Reinforcement learning is learning what to do - how to map situations to actions - so as to maximise a numerical reward.

  - *Goal-directed* learning from interaction.

- The learner is not told which actions to take, but instead it must discover which actions yield the most reward by trying them.


WAYVE


READY!

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**

Daan Wierstra **Martin Riedmiller**

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future

ARTICLE

doi:10.1038/nature16961

Mastering the game of Go with deep neural networks and tree search

David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. Here we introduce a new approach to computer Go that uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of state-

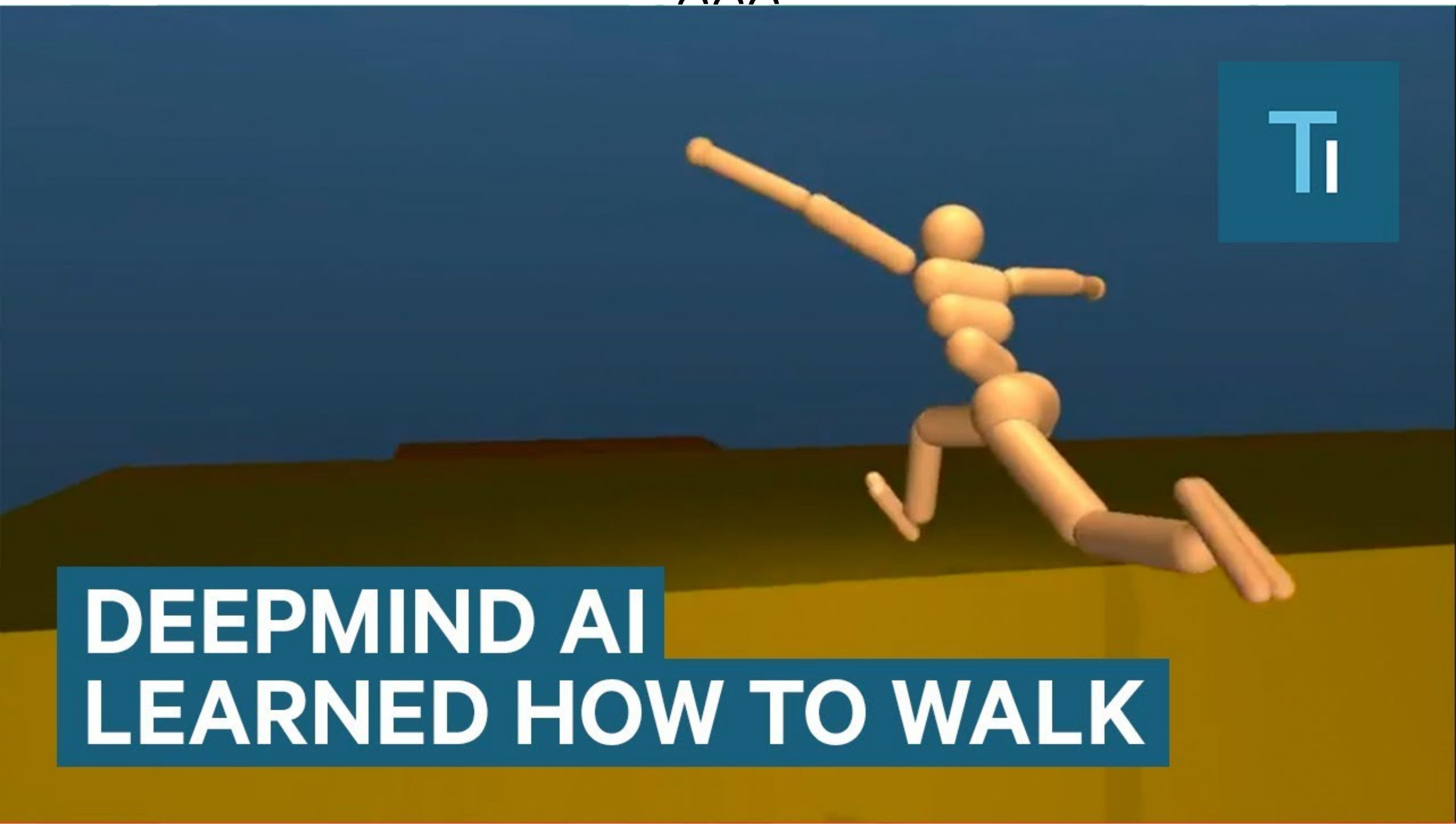
ARTICLE

doi:10.1038/nature24270

Mastering the game of Go without human knowledge

David Silver[1]*, Julian Schrittwieser[1]*, Karen Simonyan[1]*, Ioannis Antonoglou[1], Aja Huang[1], Arthur Guez[1], Thomas Hubert[1], Lucas Baker[1], Matthew Lai[1], Adrian Bolton[1], Yutian Chen[1], Timothy Lillicrap[1], Fan Hui[1], Laurent Sifre[1], George van den Driessche[1], Thore Graepel[1] & Demis Hassabis[1]

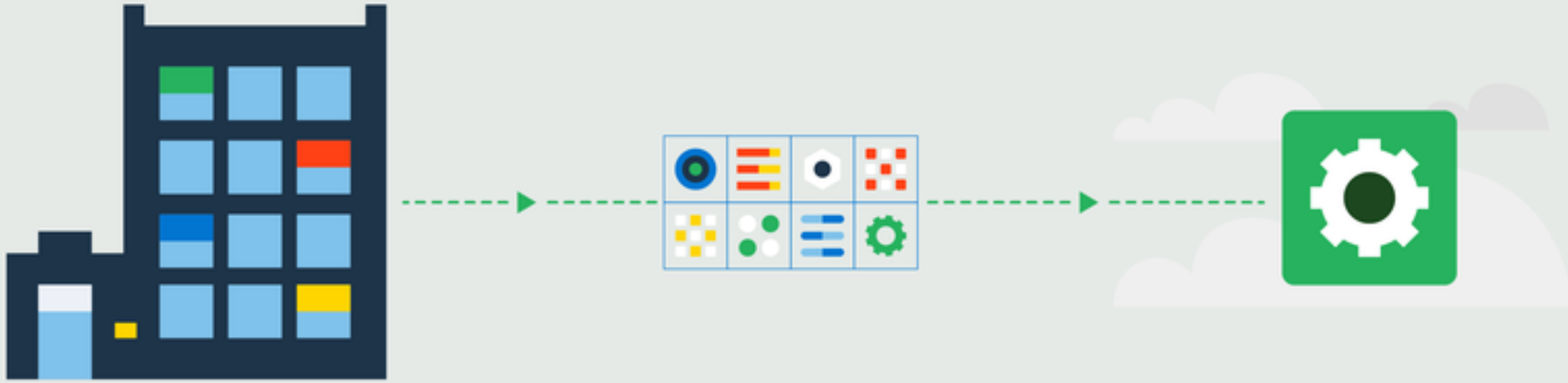
A long-standing goal of artificial intelligence is an algorithm that learns, *tabula rasa*, superhuman proficiency in challenging domains. Recently, AlphaGo became the first program to defeat a world champion in the game of Go. The tree search in AlphaGo evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play. Here we introduce an algorithm based solely on reinforcement learning, without human data, guidance or domain knowledge beyond game rules. AlphaGo becomes its own teacher: a neural network is trained to predict AlphaGo's own move selections and also the winner of AlphaGo's games. This neural network improves the strength of the tree search, resulting in higher quality


DEEPMIND AI
LEARNED HOW TO WALK

Every five minutes, our cloud-based AI pulls a snapshot of the data centre cooling system as represented by thousands of physical sensors.

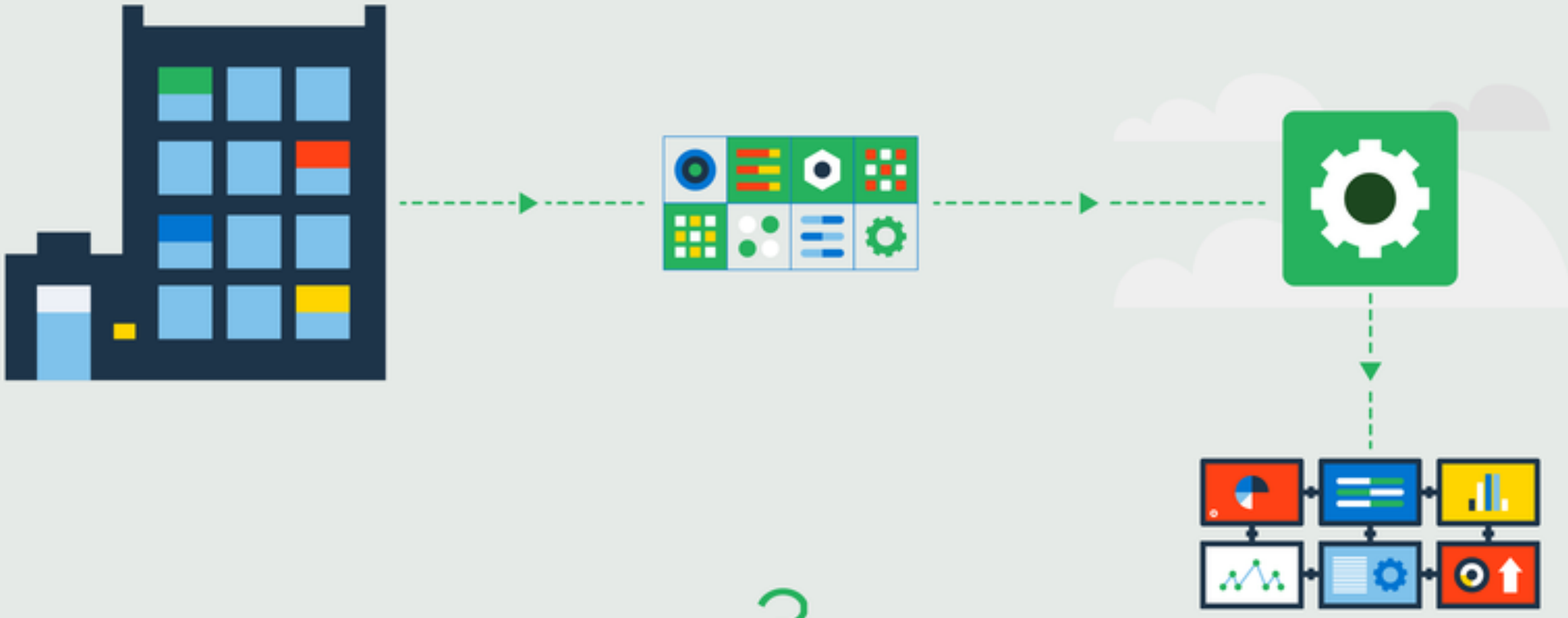
1

The information is fed into our deep neural networks, which predict the future energy efficiency and temperature based on proposed actions.



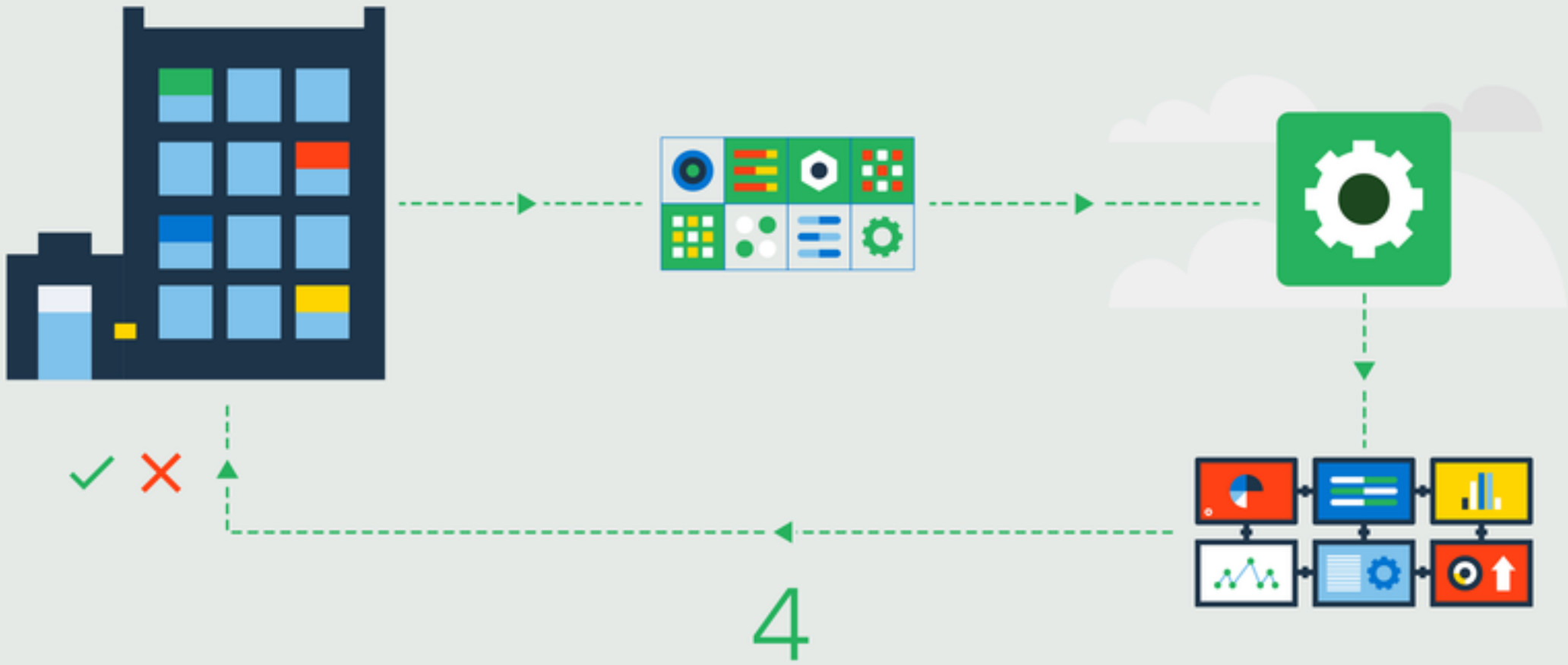
2

The AI selects actions that satisfy safety constraints and minimise future energy consumption.

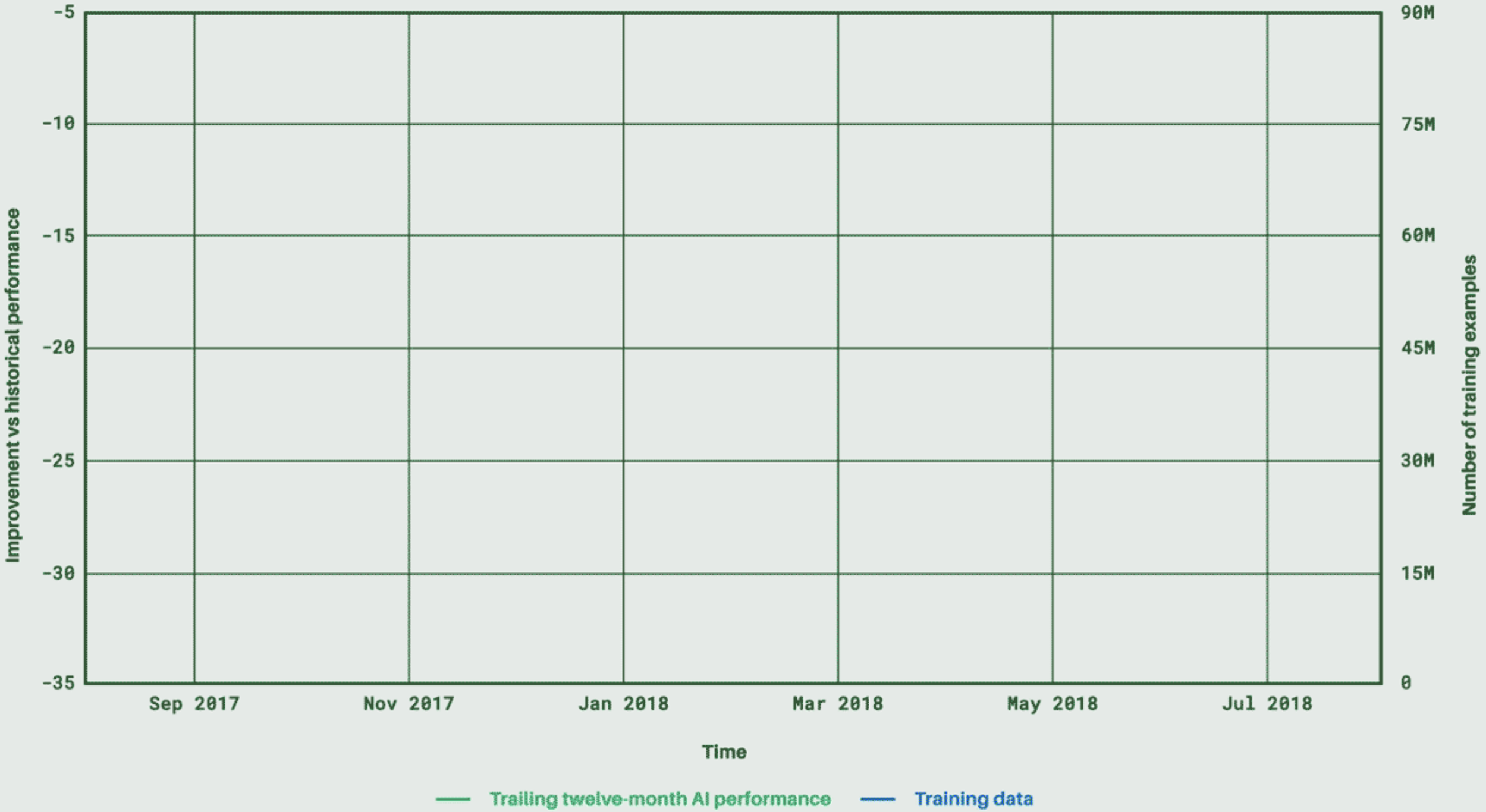


3

Optimal actions are sent back to the data centre, where the local system verifies them against its own safety constraints before implementation.

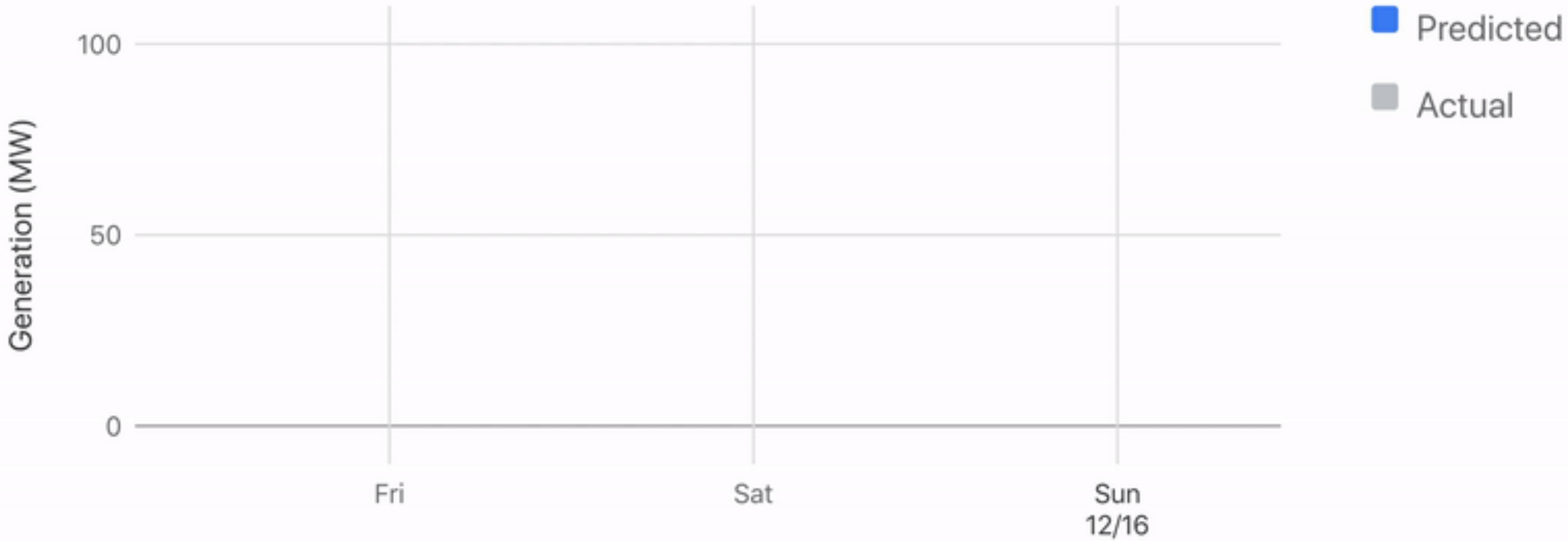


4



Source: https://deepmind.com/blog/article/safety-first-ai-autonomous-data-centre-cooling-and-industrial-control

The DeepMind system predicts wind power output 36 hours ahead...



Source: https://deepmind.com/blog/article/machine-learning-can-boost-value-wind-energy

RL in Android

- RL used in Android for:
  - Adaptive battery:
    - It is used to learn and anticipate future battery use
  - Adaptive brightness of the video:
    - Algorithm learns preferences in terms of brightness from the user

Finite Markov Decision Processes

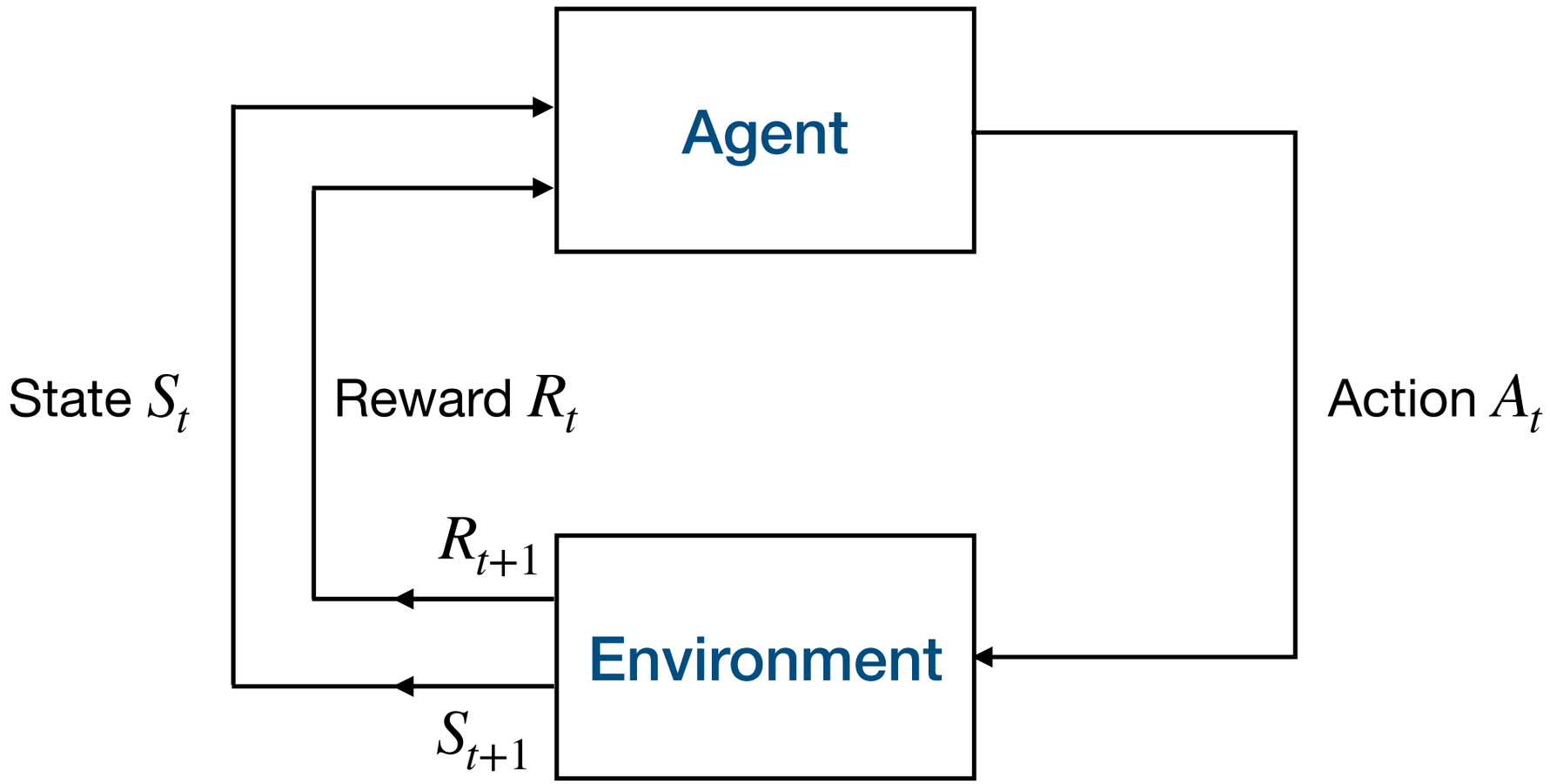
- Markov Decision Processes (MDPs) are a mathematically idealised formulation of Reinforcement Learning for which precise theoretical statements can be made.

- Tension between breadth of applicability and mathematical tractability.

- MDPs provide a way for framing the problem of learning from experience, and, more specifically, from interacting with an environment.

Markov Decision Processes: Definitions

- Two entities:
  - **Agent**: learner and decision maker.
  - **Environment**: everything else outside the agent.
- The agent interacts with the environment selecting **actions**.
- The environment changes following actions of the agent.


Agent
State S_t
Reward R_t
Action A_t
R_{t+1}
Environment
S_{t+1}

Markov Decision Processes: Definitions

- The agent and the environment interact at each of a sequence of discrete time steps $t = 0,1,2,3,...$

- At each time step t, the agent receives some representation of the environment **state** $S_t \in \mathcal{S}$ where $\mathcal{S}$ is the set of the states.

- On that basis, an agent selects an **action** $A_t \in \mathcal{A}(S_t)$ where $\mathcal{A}(S_t)$ is the set of the actions that can be taken in state S_t.

- At time $t + 1$ as a consequence of its action the agent receives a **reward** $R_{t+1} \in \mathcal{R}$, where $\mathcal{R}$ is the set of rewards (expressed as real numbers).

Goals and Rewards

- The goal of the agent is formalised in terms of the reward it receives.
- At each time step, the reward is a simple number $R_t \in \mathbb{R}$.
- Informally, the agent's goal is to maximise the total amount it receives.
- The agent should not maximise the immediate reward, but the *cumulative reward*.

The “Reward Hypothesis”

- We can formalise the goal of an agent by stating the “reward hypothesis”:

All of what we mean by goals and purposes can be well thought of as the maximisation of the expected value of the cumulative sum of a received scalar signal (reward).

Expected Returns

- We will now try to conceptualise the idea of **cumulative rewards** more formally.

- An agent receives a sequence of rewards $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$

- In order to define cumulative rewards, we introduce the concept of **expected return** G_t, which is a function of the reward sequence.

Episodic Tasks and Continuing Tasks

- Typically, we identify two cases: episodic tasks and continuing tasks.

- An **episodic task** is one in which we can identify a final step of the sequence of rewards, i.e., in which the interaction between the agent and the environment can be broken into sub-sequences that we call **episodes** (such a play of a game, repeated tasks, etc.).

  - Each episode ends in terminal state after T steps, followed by a reset to a standard starting state or to a sample of a distribution of starting states.

  - The next episode is completely independent from the previous one.

- A **continuing task** is one in which it is not possible to identify a final state (e.g., on-going process control or robots with a long-lifespan).

Expected Return for Episodic Tasks and Continuing Tasks

- In the case of *episodic tasks* the expected return associated to the selection of an action A_t is the sum of rewards defined as follows:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T$$

- In the case of *continuing tasks* the expected return associated to the selection of an action A_t is defined as follows:

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

where γ is the discount rate, with $0 \leq \gamma \leq 1$.

Why Discounting?

- The definition of expected return that we used for episodic tasks would be problematic for continuing tasks: the expected return of time of termination T would be equal to ∞ in some cases, such as when the reward is equal to 1 at each time step.

- The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth γ^{k-1} what it would be worth if it were received immediately.

Relation between Returns at Successive Time Steps

- Returns at successive time steps are related to each others as follows:

$$
\begin{aligned}
G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \\
&= R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \\
&= R_{t+1} + \gamma G_{t+1}
\end{aligned}
$$

Policies and Value Functions

- Almost all reinforcement learning algorithms involve estimating value functions, i.e., functions of states (or of state-action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state).

- A policy is used to to model how the behaviour of the agent based on the previous experience and the rewards (and consequently the expected returns) an agent received in the past.

Definition of Policy

- Formally, a *policy* is a mapping from states to probabilities of each possible action.

- If the agent is following policy π at time t, then $\pi(a \mid s)$ is the probability that $A_t = a$ if $S_t = s$.

Definition of State-Value Function

- The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter.

- For MDPs, we can define the *state-value function* v_π for policy π formally as:

$$v_s \doteq E_\pi[G_t \,|\, S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|\, S_t = s, A_t = a]$$

for all $s \in \mathcal{S}$

where $E_\pi[\,.\,]$ denotes the expected value of a random variable given that the agent follows π and t is any time step. The value of the terminal state is 0.

Definition of Action-Value Function

- Similarly, we define the *action-value function*, i.e., the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s, a) \doteq E_\pi[G_t \,|\, S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|\, S_t = s, A_t = a]$$

Choosing the Rewards

- When we model a real system as a Reinforcement Learning problem, the hardest problem is to select the right rewards.

- Typically, we use negative values for actions that do not help us in reaching our goal and positive if they do (and sometimes we set the values to 0 if they do not help us in reaching the goal).

- An alternative is to set the values of rewards to a negative number until we reach our goal (and we set the value to 0 when we reach our goal).

Choosing the Rewards

- It is very important to keep in mind that we should not “reward” the intermediate steps or the single actions.

- We are not “teaching” the agent how to execute an intermediate step, but how to reach the final goal. If we do so, the agent will learn how to reach the intermediate step, e.g., how to execute a sub-task.

- The reward should tell the agent if the current step is a step forward towards the final goal or not.

Example of Rewards

Credit: Shutterstock

Maze -> Rewards: -1 for no exit 0 for exit

Examples of Rewards

Chess -> Rewards: 1 for victory, -1 for defeat

Choosing the Rewards

- Sometimes it's not possible to know the reward until the end of an episode. The typical example is a board game (chess, go, etc.).

- This is usually called *credit assignment problem,* i.e., the problem of assigning a reward to each step.

- In that case the reward might be assigned at the end of a Montecarlo rollout for example (stochastic estimate of the reward).

- For example if the game is successful we can use +1 as reward for all the steps that leads to the victory (or -1 otherwise).

Example of Rewards

- In Go or Chess, the reward will be 1 for winning or -1 losing for the terminal state (i.e., the state at time T), but we will know the result of the game only at the end.

- Therefore, the reward can be assigned only at the end of an episode.

- In Go or Chess, we can for example assign 1 or -1 to each step in case of victory or loss at the end of the episode after a Montecarlo playout/rollout.

How to Estimate the State-Value (Action-Value) Functions

- If the behaviour of the MDP is known (i.e., the transitions probabilities between all the states are known), the state function or the action-state function can be estimated by considering all the possible moves.

- This is not possible when:

  - The transitions probabilities are not know.

  - The system is very complex (for example a board game has a very large number of potential game configurations).

How to Estimate the State-Value (Action-Value) Functions: Monte-Carlo Methods

- Alternatively, the state-value function v_π and the action-value function q_π can be estimated through experience.

- One possibility is to keep average values of the actual returns that have followed a certain state (or a certain action) while following a policy π. These values will converge to the actual state-value function v_π and the action-value function q_π asymptotically.

- These methods based on averaging sample returns are referred to as *Monte Carlo methods*.

How to Estimate the State-Value (Action-Value) Functions: Monte-Carlo Methods

- Monte Carlo methods are not appropriate in case the number of states is very large.

- In this case, it is not practical to keep separate averages for each state individually.

- Instead, v_π and q_π are maintained as parametrised functions with the number of parameters << number of states.

- Various function approximators of different complexity are possible.

  - Artificial neural networks are a possible option as function approximators -> Deep Reinforcement Learning

Optimal Policies and Optimal Value Functions

- Solving a reinforcement learning is roughly equivalent to finding a policy that maximises the amount of reward over the long run.

- In finite MDPs there is always at least one policy that is better or equal to all the other policies: this is called the *optimal policy*.

- Although there may be more than one, we denote all the optimal policies with π_*. They are characterised by the same value function v_* defined as

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

for all $s \in \mathcal{S}$.

Optimal Policies and Optimal Value Functions

- Optimal policies also shares the same optimal action-value function q_*, which is defined as

$$q_* \doteq \max_\pi q_\pi(s, a)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.

- We can write q_* in terms of v_* as follows:

$$q_*(s, a) = E[R_{t+1} + \gamma v_*(S_{t+1}) \,|\, S_t = s, A_t = a].$$

Difference between Reinforcement Learning and Supervised Learning

- Supervised learning is learning from a set of labeled examples.

- In interactive problems, it is hard to obtain labels in the first place.

- In “unknown” situations, agent have to learn from their experience. In these situations, reinforcement learning is most beneficial.

Difference between Reinforcement Learning and Unsupervised Learning

- Unsupervised learning is learning from datasets containing unlabelled data.

- You might think that reinforcement learning is a type of unsupervised learning, because it does not rely on examples (labels) of correct behaviour and instead explores and learns it. However, in reinforcement learning the goal is to maximise a reward signal instead of trying to find a hidden structure.

- For this reason, reinforcement learning is usually considered a third paradigm in addition to supervised and unsupervised learning.

Temporal-Difference Learning

- Temporal-difference (TD) methods like Monte Carlo methods can learn directly from experience.

- Unlike Monte Carlo methods, TD method update estimates based in part on other learned estimates, without waiting for the final outcome (we say that they *bootstrap*).

- We will first consider the problem of prediction (TD prediction) first (i.e., we fix a policy π and we try to estimate the value v_π for that given policy).

- Then we will consider the problem of finding an optimal policy (TD control).

Three Key Problems

- Recall: we consider three key RL problems:
  - The *prediction problem*: the estimation of v_π and q_π for a fixed policy π.
  - The *policy improvement problem*: the estimation of v_π and q_π while trying at the same time to improve the policy π.
  - The *control problem*: the estimation of an optimal policy π_*.

TD Prediction

- Both TD and Monte Carlo methods for the prediction problem are based on experience.

- Roughly speaking, Monte Carlo methods wait until the return following the visit is known, then use that return as a target for $V(S(t))$.

- An every-visit Monte Carlo method suitable for non-stationary environment is: $V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$.

- where G_t is the actual return following time t and α is a constant size parameter. This is not based on the average values but on a weighted average (you can get the average if you consider instead $\frac{1}{n}$ as step-size parameter).

TD Prediction

- Monte Carlo methods must wait until the end of the episode to determine the increment to $V(S(t))$, because only at that point it is possible to calculate $G(t)$.

- TD methods instead need to wait only until the next step.

- At time $t+1$ they immediately from a target make a useful update using the observed reward R_{t+1} and the estimate $V(S_{t+1})$.

TD(0)

- The TD(0) method is based on the following update:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

on transition to S_{t+1} and receiving R_{t+1}.

This method is also called 1-step TD.

- Essentially, the target for the Monte Carlo update is G_t, whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$.

TD(0)

Input: the policy π to be evaluated

Algorithm parameter: step size $\alpha \in (0,1]$

Initialise $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:

Initialise S

Loop for each step of episode:

$A \leftarrow$ action given by π for S

Take action A, observe R, S'

$$V(S) \leftarrow V(S) + \alpha[R + \alpha V(S') - V(S)]$$

$$S \leftarrow S'$$

until S is terminal

TD(0)

- Note that the quantity in brackets in the TD(0) update is a sort of error, measuring the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$.

- The *TD error* is defined as:

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

- The TD error at each time is the error in the estimate *made at that time*.

- It is interesting to note that, since the TD error depends on the next state and next reward, it is not actually available until one time step later.

  - In other words, δ_t is the error in $V(S_t)$ available at a time $t + 1$.

Advantages of TD Prediction Methods

- Compared to Dynamic Programming methods, TD methods do not require a model of the environment, of its reward and next-state probability distributions.
- Compared to Monte Carlo methods, TD methods are implemented in an online, fully incremental fashion.
  - With Monte Carlo methods, one must wait until the end of an episode (i.e., when the return is known), instead with TD methods, we need to wait only one time step.
  - Why does this matter?
    - Some applications have very long episodes.
    - Some applications are actually continuing tasks.

Theoretical Basis of TD(0)

- Even if the learning process happens step-by-step, we have convergence guarantees supporting the methods presented in the lecture (see Sections 6.2 and 9.4 of Barto and Sutton 2018).

- More precisely, for any fixed policy π, TD(0) has been proved to converge to v_π, in the mean for a constant step-size parameter if it is sufficiently small, and with probability 1 if the step-size decreases given stochastic approximation conditions (see Section 2.7 of Barto and Sutton 2018).

- But what is the faster in terms of convergence between dynamic programming, Monte Carlo and TD?

  - It's still an open question, in case you are looking for a research topic!

On-Policy and Off-Policy Control

- Recall the difference between on-policy and off-policy control:
  - *On-policy control*: exploration of the states *following* the policy.
  - *Off-policy control*: exploration of the states *not following* the current policy (for example with a stochastic policy).

Sarsa: On-Policy TD Control

- We now consider the use of TD prediction methods for the control problem.
- For an on-policy method, we must estimate $q_\pi(s, a)$ for the current behaviour policy π and for all the states s and actions a.
- Above, we consider the transitions from state to state and we learned the values of states.
- Now we consider the transitions from state-action pair to state-action pair and learn the values of state-action pair.

Sarsa: On-policy TD Control

- Formally, these cases are identical: they are both Markov chain with a reward process. The theorems assuring the converge of state values under TD(0) also apply to the corresponding algorithm for action values:
- $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$
- This update is done after every transition from a non-terminal state S_t.
- If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1}) \leftarrow 0$.

Sarsa: Online TD(0) Control

- The update of Sarsa uses all the elements of the quintuple: $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$.
  - This indicates a transition from a state to the next.
  - This quintuple gives rise to the name *Sarsa*.
- We can design an on-policy control algorithm based on the Sarsa prediction method.
  - As in all on-policy methods, we continually estimate q_π for the behaviour policy π.
  - At the same time, we assume a greedy policy using Q values.
    - By doing so we will have a convergence of the Q values to q_*.

Q-learning: Off-policy TD Control

- Q-learning is one of the classic RL algorithms.
- Q-learning is an off-policy TD control algorithm, defined by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

- In this case, the learned action-value function Q directly approximates q_*, the optimal action-value function, independent of the policy being followed.
  - Policy still matters since it determines which state-action pairs are visited/updated. However, only requirement for convergence is that all pairs continue to be updated.
  - Early convergence proofs.

Learning from Delayed Rewards

Christopher John Cornish Hellaby Watkins

King's College

Thesis Submitted for Ph.D.

May, 1989

Machine Learning, 8, 279–292 (1992)
© 1992 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Technical Note

Q-Learning

CHRISTOPHER J.C.H. WATKINS
25b Framfield Road, Highbury, London N5 1UU, England

PETER DAYAN
Centre for Cognitive Science, University of Edinburgh, 2 Buccleuch Place, Edinburgh EH8 9EH, Scotland

Abstract. Q-learning (Watkins, 1989) is a simple way for agents to learn how to act optimally in controlled Markovian domains. It amounts to an incremental method for dynamic programming which imposes limited computational demands. It works by successively improving its evaluations of the quality of particular actions at particular states.

This paper presents and proves in detail a convergence theorem for Q-learning based on that outlined in Watkins (1989). We show that Q-learning converges to the optimum action-values with probability 1 so long as all actions are repeatedly sampled in all states and the action-values are represented discretely. We also sketch extensions to the cases of non-discounted, but absorbing, Markov environments, and where many Q values can be changed each iteration, rather than just one.

Summary

- The methods that we discussed are among the most-used methods in RL.

- These methods are usually referred to as tabular methods, since the state-action space can fit in a table.

  - Table with 1 row per state-action entry.

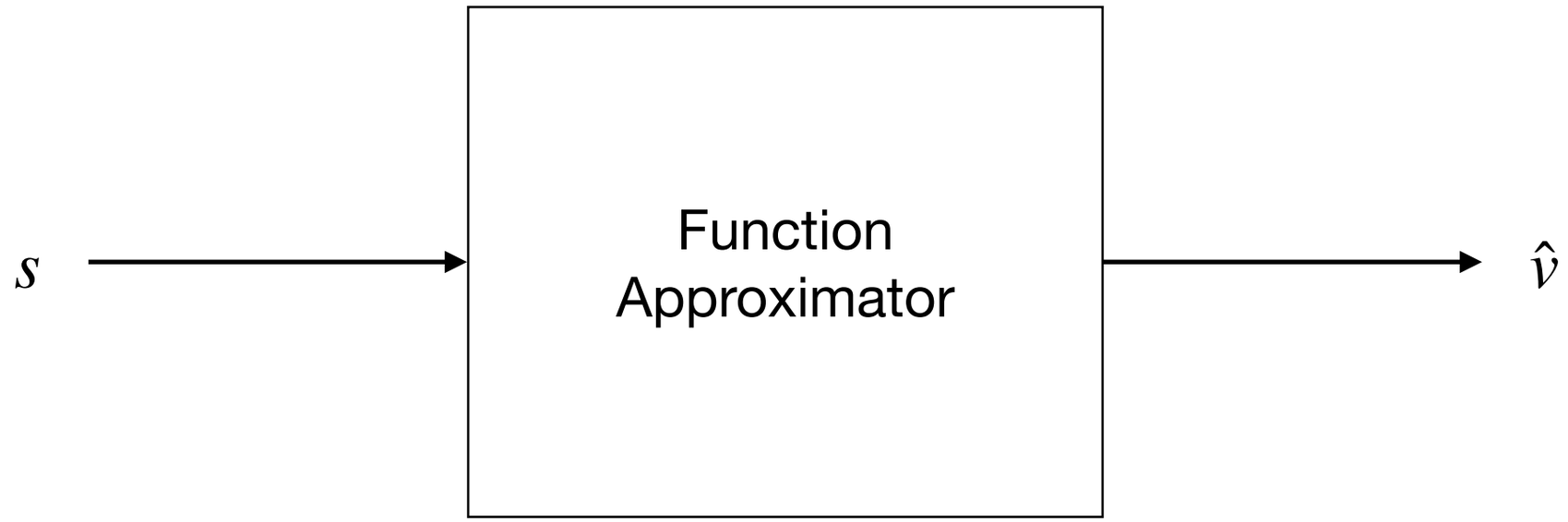
  - What happens if you can't fit all the state-action entry in a table?

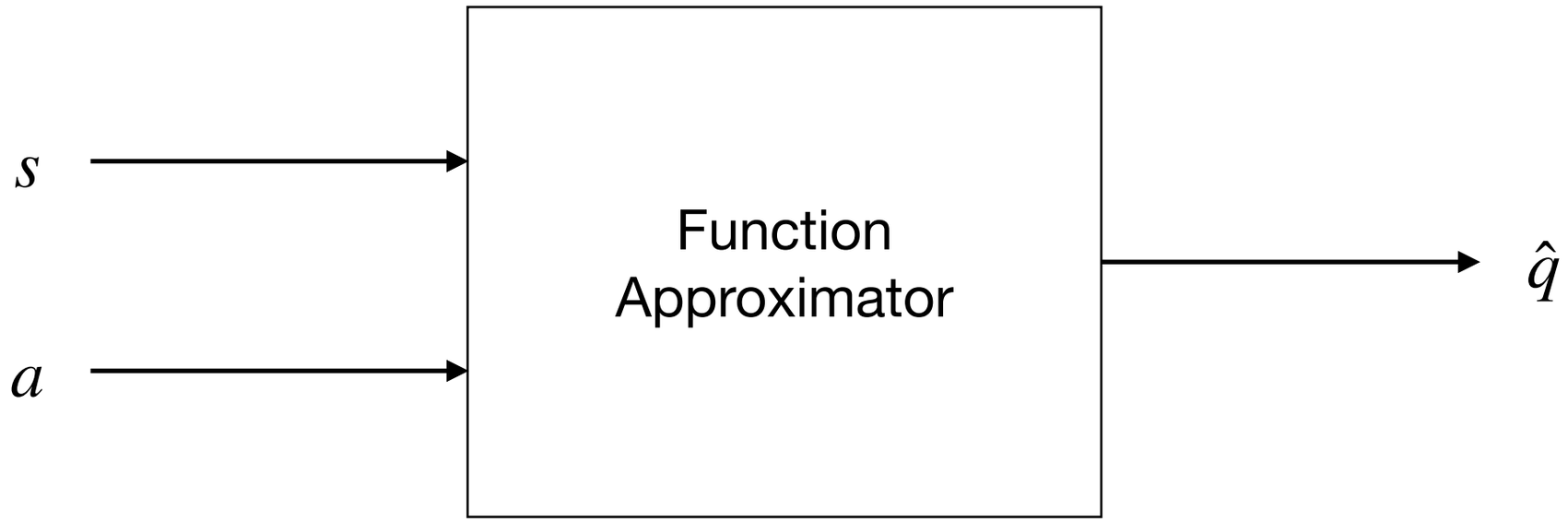
    - We need *function approximation* rather than tables.

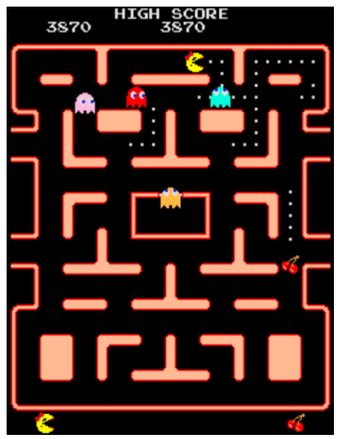
Summary

- Function Approximation will provide a mapping between a state or state-action to a value function.

- More precisely, a value-function approximation is a function with in input the state (or the state and action), which gives in output the value for the state (or the state and action).


Function
Approximator
s
$\hat{v}$


s
a
Function
Approximator
$\hat{q}$



Function Approximator

$\hat{q}$