An Introduction to Deep Reinforcement Learning

Part 2 – Policy Gradient, Actor-Critic, Evolutionary Strategies

Karim Bouyarmane

Recap of Part I:

Definition of **Markov Decision Processes** (**states $\boldsymbol{s}$**, **actions $\boldsymbol{a}$**, **rewards $\boldsymbol{r}$**/objective, **return $\boldsymbol{R}$** as collected sum of discounted rewards along a trajectory – change of notation from Part 1)

Definition of a **policy** π (mapping from state to action, feedback law) and **optimal policy** π^* (best action to take at every state)

executing a policy = following a trajectory τ and collecting rewards r_t along the trajectory, that we accumulate in the return

$$R(\tau) = \sum_{t=0}^{+\infty} \gamma^t r_t$$

Definition of **state value** $V(s)$ (expected return for a trajectory that **starts from s**, starting with 0 return)

> When executing a trajectory that **doesn't start from s** and that arrives at some point at s, the expected return becomes the actual return accumulated so far + the (discounted) value of state s, $\gamma^t V(s)$

Definition of **Q-value** or **action value** $Q(s,a)$ (expected return for a trajectory that **starts from s with action a**, starting with 0 return)

Fundamental relationships that link π^*, V, Q together:

$$V(s) = \max_a Q(s,a)$$

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

$$V(s) = Q(s, \pi^*(s))$$

Bellman equation (V at a state defined by V at neighboring states, Q of action at state defined by Q of actions at neighboring states) allows to define a supervised learning problem on Q or V, with the labels being themselves defined with Q or V.
Bellman equation allows to generate labeled data for the learning problem

Exact methods/dynamic programming methods (e.g. *value iteration*: starting from random V values, then iteratively enforcing Bellman equation for V, other example: *policy iteration*)

Q-learning:

- Initialize random policy, i.e. random values for Q
- start from initial state s_0
- pick action a_0
- simulate action a_0 on state s_0
- observe new state s_1 and reward r_1
- apply "supervised" learning on $Q(s_0, a_0)$ with the label being $r_1 + \gamma \max_a Q(s_1, a)$

  i.e. we have collected a pair of labelled data $\langle X, Y \rangle$ for Q with $X = (s_0, a_0)$ and $Y = r_1 + \gamma \max_a Q(s_1, a)$

- now agent is in state s_1
- pick action a_1
- etc…

Q-learning:

- Initialize random policy, i.e. random values for Q
- start from initial state s_0
- pick action a_0 (exploitation of current policy $\text{argmax}_a\, Q(s_0, a)$, or exploration with random action with probability ϵ)
- simulate action a_0 on state s_0
- observe new state s_1 and reward r_1
- apply "supervised" learning on $Q(s_0, a_0)$ with the label being $r_1 + \gamma \max_a Q(s_1, a)$

  i.e. we have collected a pair of labelled data $\langle X, Y \rangle$ for Q with $X = (s_0, a_0)$ and $Y = r_1 + \gamma \max_a Q(s_1, a)$

- now agent is in state s_1
- pick action a_1 (exploitation of current policy $\text{argmax}_a\, Q(s_1, a)$, or exploration with random action with probability ϵ)
- etc…

Q-learning = start with random Q/policy, execute policy, collect experience tuples $\langle s, a, s', r \rangle$ along trajectory, enforce Bellman equation for this experience tuple and update Q/policy

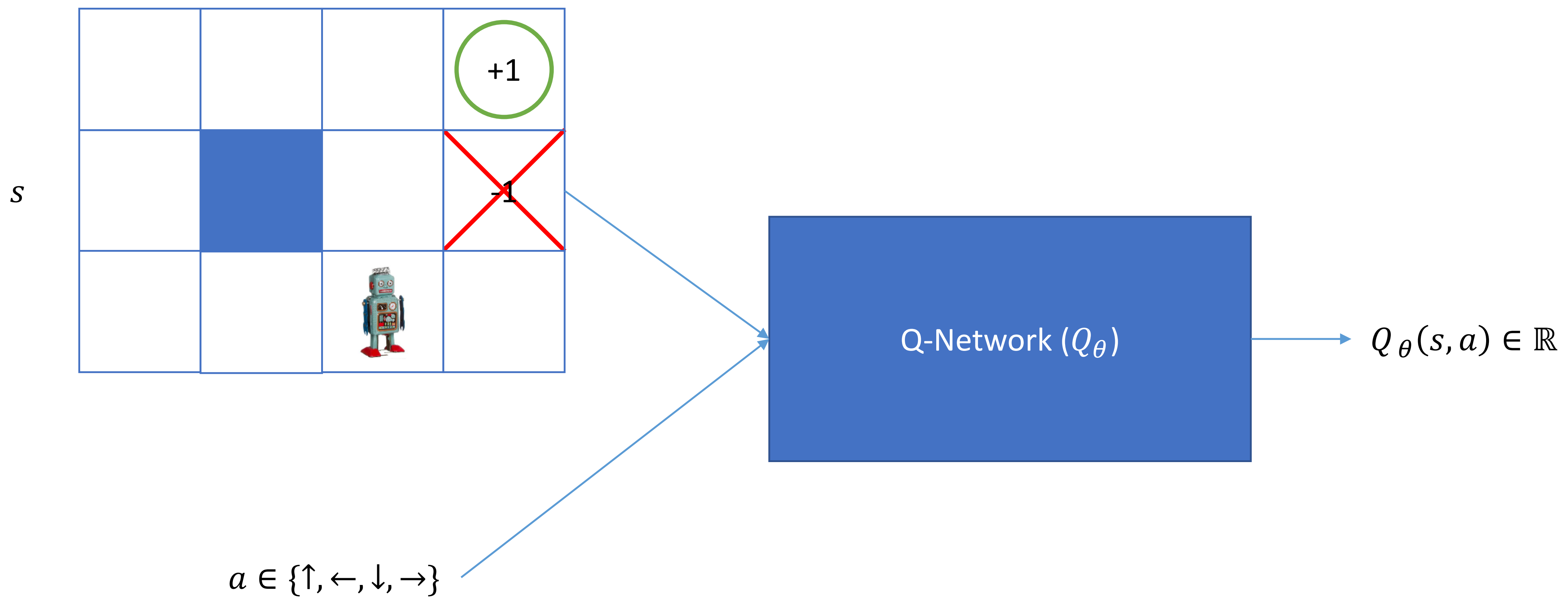
$$Q(s,a) = \alpha Q(s,a) + (1-\alpha)\left(r + \gamma \max_{a'} Q(a', s')\right)$$

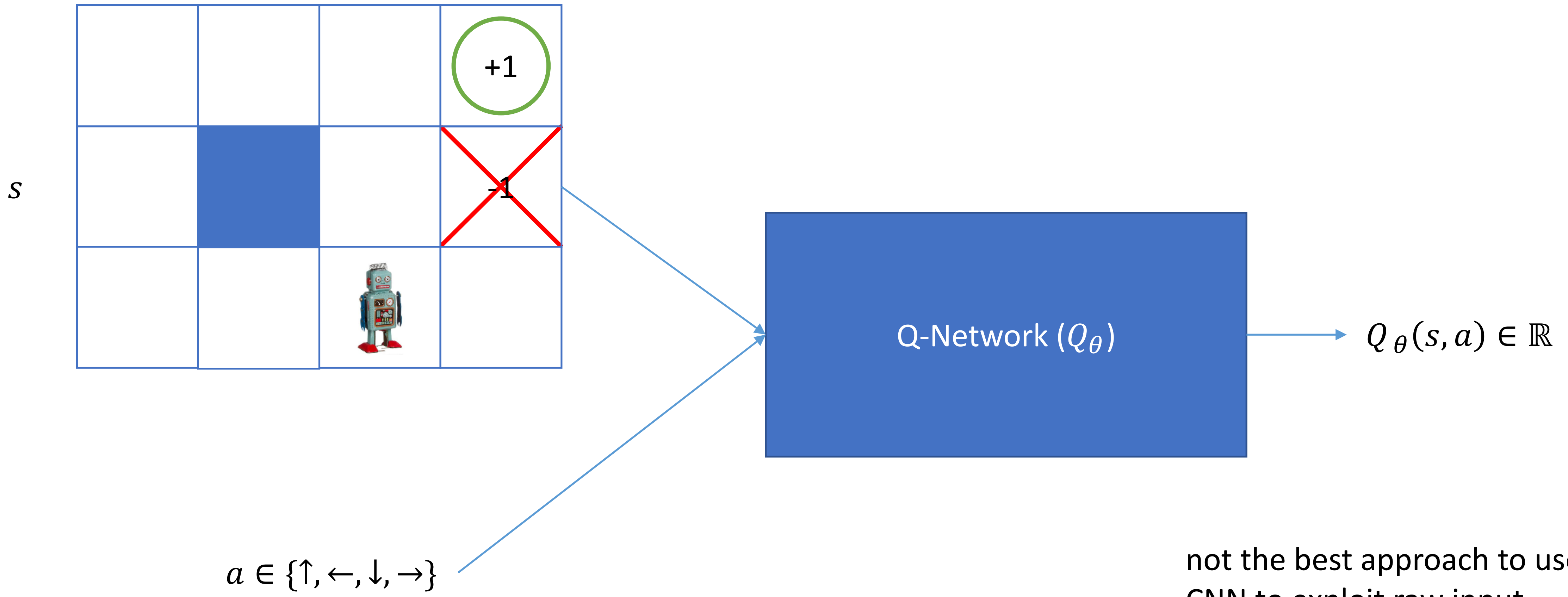
Approximate Q-learning (e.g. with deep neural net, called Q-network) = parameterize Q_θ as a neural net with weights θ

$$\theta = \theta - \alpha \nabla_\theta \left(r + \gamma \max_{a'}\ Q_\theta(a', s') - Q_\theta\ (s,a)\right)^2$$

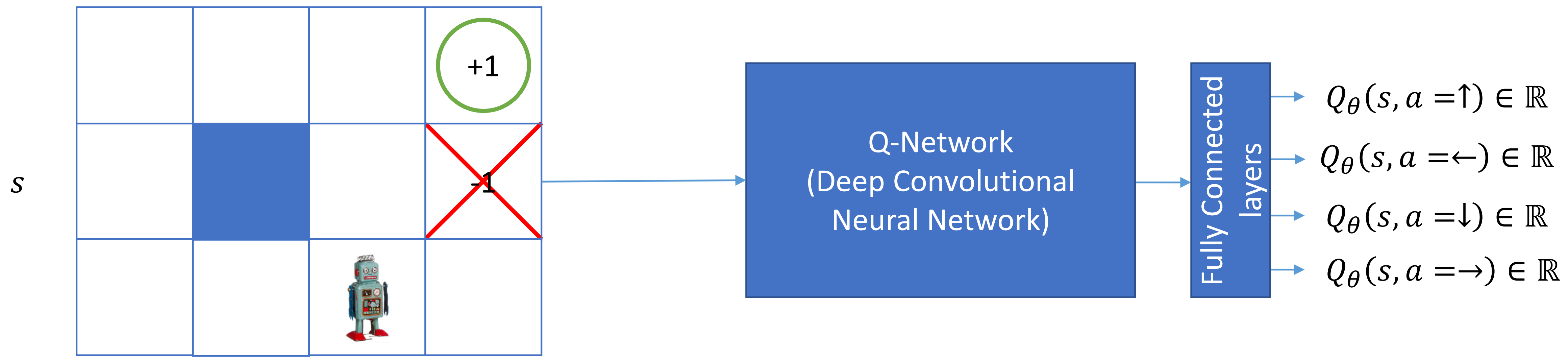
Deepmind's DQN use a separate network θ' to compute target of Bellman equation, update target network with Q-Network periodically, apply Bellman equation SGD step with batch of experience tuples sampled from previous experiences (memory buffer) rather using current experience tuple, store current experience tuple in memory buffer for later sampling

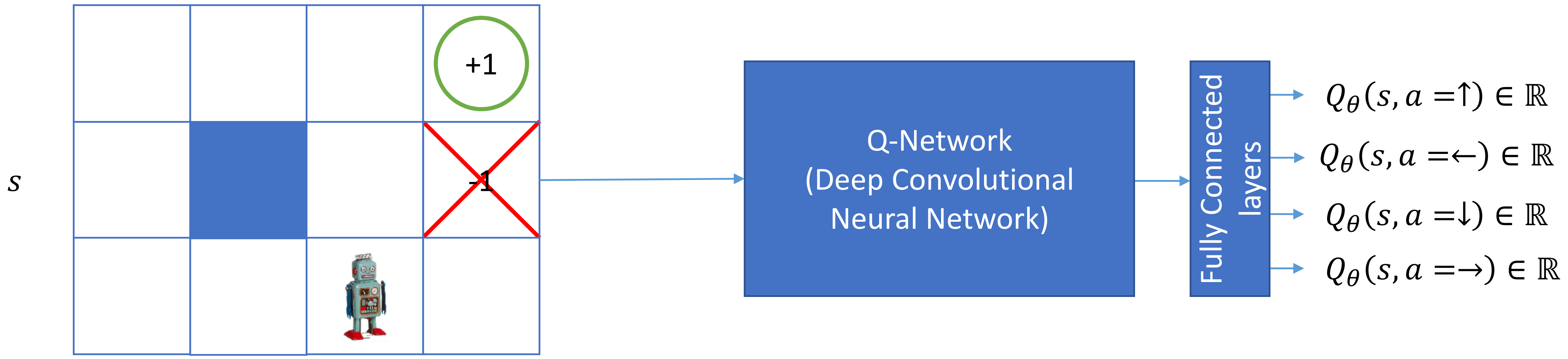
$$\theta = \theta - \alpha \nabla_\theta \left(r + \gamma \max_{a'}\ Q_{\theta'}(a', s') - Q_\theta\ (s,a)\right)^2$$


s
+1
-1
$a \in \{\uparrow, \leftarrow, \downarrow, \rightarrow\}$
Q-Network (Q_θ)
$Q_\theta(s, a) \in \mathbb{R}$

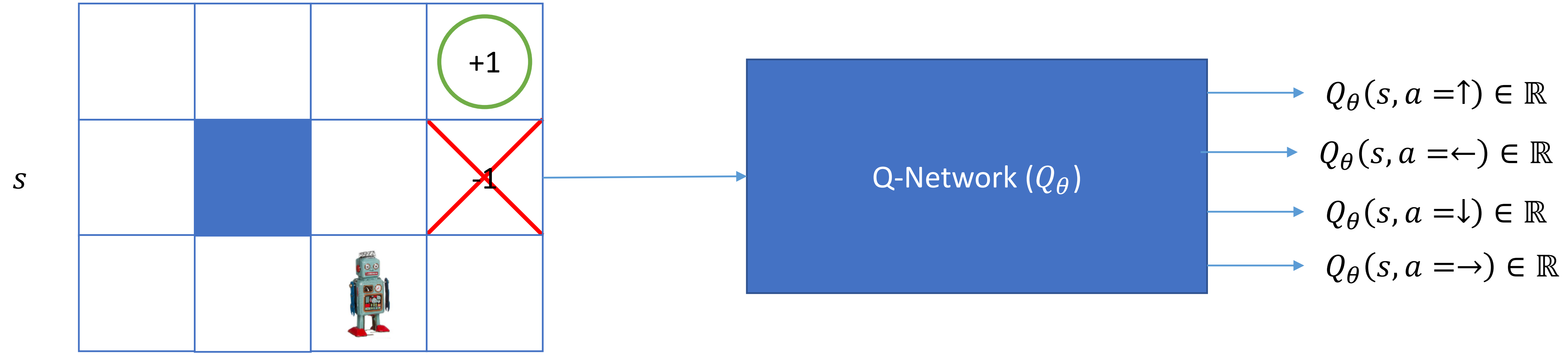

s
+1
-1
Q-Network (Q_θ)
$Q_\theta(s,a) \in \mathbb{R}$
$a \in \{\uparrow, \leftarrow, \downarrow, \rightarrow\}$

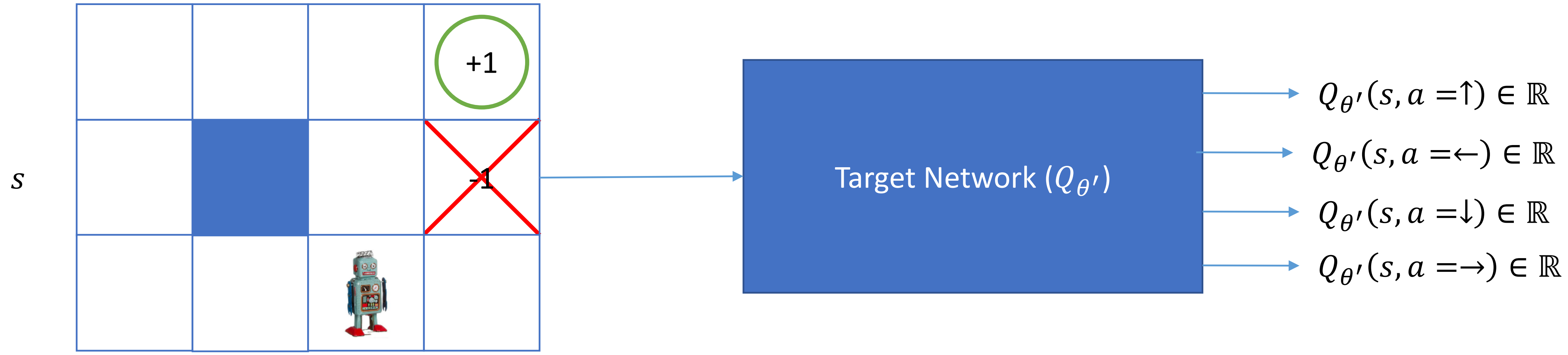
not the best approach to use CNN to exploit raw input images as states

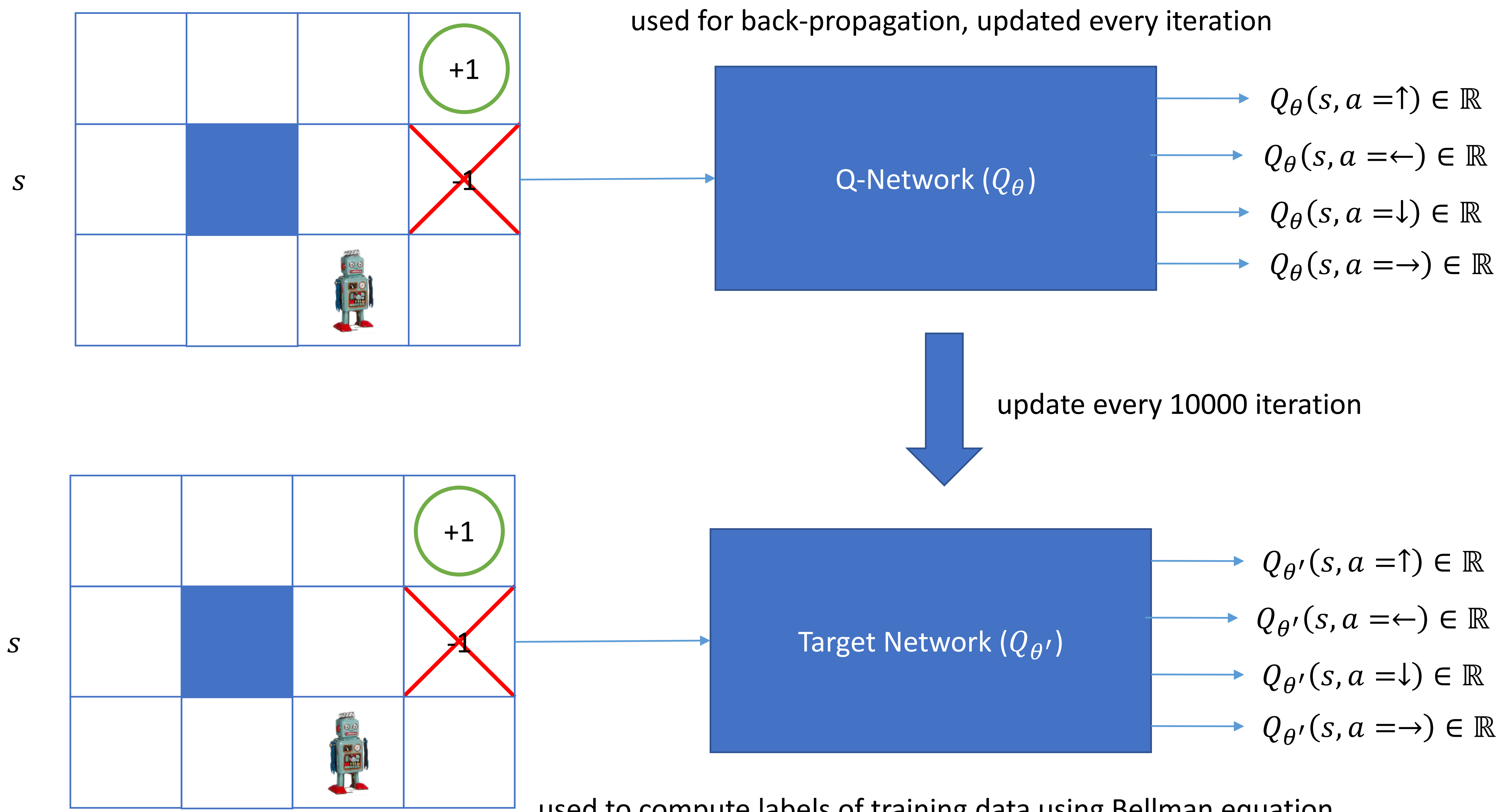

s
+1
-1
Q-Network
(Deep Convolutional
Neural Network)
Fully Connected
layers
$Q_\theta(s, a = \uparrow) \in \mathbb{R}$
$Q_\theta(s, a = \leftarrow) \in \mathbb{R}$
$Q_\theta(s, a = \downarrow) \in \mathbb{R}$
$Q_\theta(s, a = \rightarrow) \in \mathbb{R}$


s
+1
-1
Q-Network
(Deep Convolutional
Neural Network)
Fully Connected
layers
$Q_\theta(s, a = \uparrow) \in \mathbb{R}$
$Q_\theta(s, a = \leftarrow) \in \mathbb{R}$
$Q_\theta(s, a = \downarrow) \in \mathbb{R}$
$Q_\theta(s, a = \rightarrow) \in \mathbb{R}$

better approach: multi-task network


s
+1
-1
Q-Network (Q_θ)
$Q_\theta(s, a = \uparrow) \in \mathbb{R}$
$Q_\theta(s, a = \leftarrow) \in \mathbb{R}$
$Q_\theta(s, a = \downarrow) \in \mathbb{R}$
$Q_\theta(s, a = \rightarrow) \in \mathbb{R}$

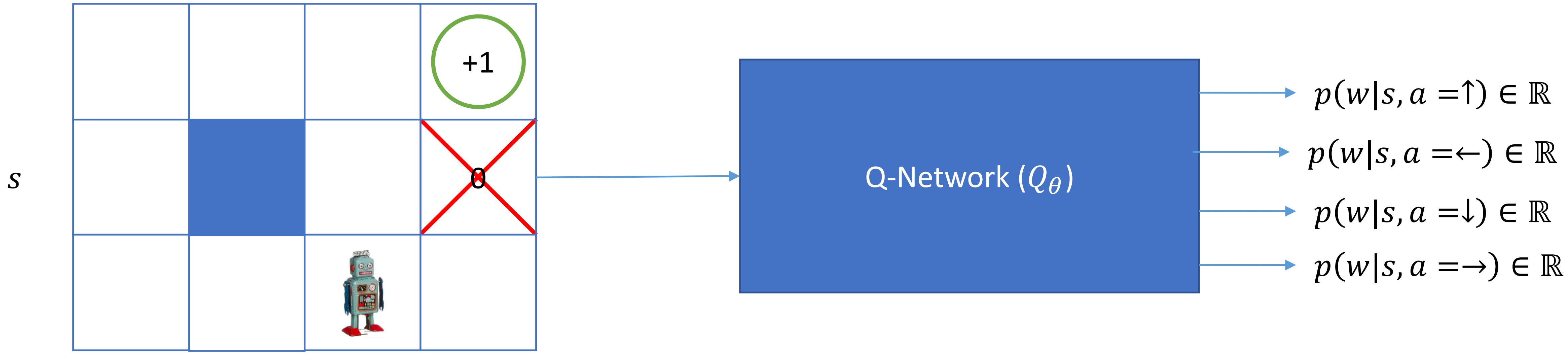

s
+1
-1
Target Network ($Q_{\theta'}$)
$Q_{\theta'}(s, a = \uparrow) \in \mathbb{R}$
$Q_{\theta'}(s, a = \leftarrow) \in \mathbb{R}$
$Q_{\theta'}(s, a = \downarrow) \in \mathbb{R}$
$Q_{\theta'}(s, a = \rightarrow) \in \mathbb{R}$


s
+1
-1
Q-Network (Q_θ)
$Q_\theta(s, a = \uparrow) \in \mathbb{R}$
$Q_\theta(s, a = \leftarrow) \in \mathbb{R}$
$Q_\theta(s, a = \downarrow) \in \mathbb{R}$
$Q_\theta(s, a = \rightarrow) \in \mathbb{R}$
update every 10000 iteration
s
+1
-1
Target Network ($Q_{\theta'}$)
$Q_{\theta'}(s, a = \uparrow) \in \mathbb{R}$
$Q_{\theta'}(s, a = \leftarrow) \in \mathbb{R}$
$Q_{\theta'}(s, a = \downarrow) \in \mathbb{R}$
$Q_{\theta'}(s, a = \rightarrow) \in \mathbb{R}$

used for back-propagation, updated every iteration

used to compute labels of training data using Bellman equation
“frozen” version of Q-network for 10000 iteration

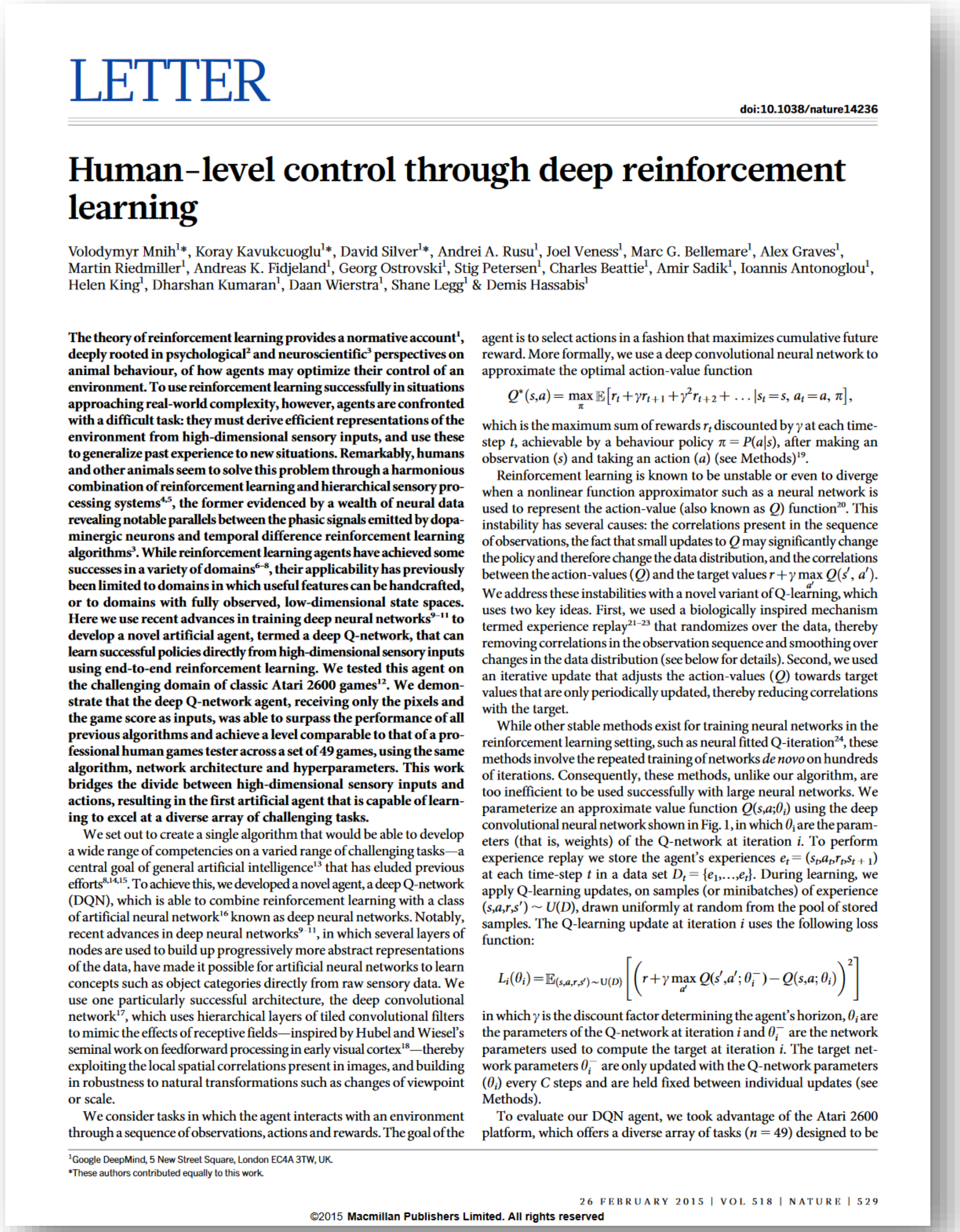
Note: in a win/lose process (only one reward at the end, +1 if win and 0 if lose, with no intermediate rewards), value and action value have particular interpretation: the probability of winning



s
+1
0
Q-Network (Q_θ)
$p(w|s, a = \uparrow) \in \mathbb{R}$
$p(w|s, a = \leftarrow) \in \mathbb{R}$
$p(w|s, a = \downarrow) \in \mathbb{R}$
$p(w|s, a = \rightarrow) \in \mathbb{R}$

$$Q(s, a) = p(w|s, a)$$
$$V(s) = p(w|s)$$

INNOVATIONS IN
The microbiome
nature
THE INTERNATIONAL WEEKLY JOURNAL OF SCIENCE
LEARNING CURVE
Self-taught AI software attains human-level performance in video games
PAGES 486 & 529
EPIDEMIOLOGY
SHARE DATA IN OUTBREAKS
Forge open access to sequences and more
PAGE 477
COSMOLOGY
A GIANT IN THE EARLY UNIVERSE
A supermassive black hole at a redshift of 6.3
PAGES 490 & 512
QUANTUM PHYSICS
TELEPORTATION FOR TWO
Transferring two properties of a single photon
PAGES 491 & 516
NATURE.COM/NATURE
26 February 2015


LETTER

doi:10.1038/nature14236

Human-level control through deep reinforcement learning

Volodymyr Mnih[1]*, Koray Kavukcuoglu[1]*, David Silver[1]*, Andrei A. Rusu[1], Joel Veness[1], Marc G. Bellemare[1], Alex Graves[1], Martin Riedmiller[1], Andreas K. Fidjeland[1], Georg Ostrovski[1], Stig Petersen[1], Charles Beattie[1], Amir Sadik[1], Ioannis Antonoglou[1], Helen King[1], Dharshan Kumaran[1], Daan Wierstra[1], Shane Legg[1] & Demis Hassabis[1]

The theory of reinforcement learning provides a normative account[1], deeply rooted in psychological[2] and neuroscientific[3] perspectives on animal behaviour, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new situations. Remarkably, humans and other animals seem to solve this problem through a harmonious combination of reinforcement learning and hierarchical sensory processing systems[4,5], the former evidenced by a wealth of neural data revealing notable parallels between the phasic signals emitted by dopaminergic neurons and temporal difference reinforcement learning algorithms[3]. While reinforcement learning agents have achieved some successes in a variety of domains[6–8], their applicability has previously been limited to domains in which useful features can be handcrafted, or to domains with fully observed, low-dimensional state spaces. Here we use recent advances in training deep neural networks[9–11] to develop a novel artificial agent, termed a deep Q-network, that can learn successful policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. We tested this agent on the challenging domain of classic Atari 2600 games[12]. We demonstrate that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks.

We set out to create a single algorithm that would be able to develop a wide range of competencies on a varied range of challenging tasks—a central goal of general artificial intelligence[13] that has eluded previous efforts[8,14,15]. To achieve this, we developed a novel agent, a deep Q-network (DQN), which is able to combine reinforcement learning with a class of artificial neural network[16] known as deep neural networks. Notably, recent advances in deep neural networks[9–11], in which several layers of nodes are used to build up progressively more abstract representations of the data, have made it possible for artificial neural networks to learn concepts such as object categories directly from raw sensory data. We use one particularly successful architecture, the deep convolutional network[17], which uses hierarchical layers of tiled convolutional filters to mimic the effects of receptive fields—inspired by Hubel and Wiesel's seminal work on feedforward processing in early visual cortex[18]—thereby exploiting the local spatial correlations present in images, and building in robustness to natural transformations such as changes of viewpoint or scale.

We consider tasks in which the agent interacts with an environment through a sequence of observations, actions and rewards. The goal of the agent is to select actions in a fashion that maximizes cumulative future reward. More formally, we use a deep convolutional neural network to approximate the optimal action-value function

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

which is the maximum sum of rewards r_t discounted by γ at each time-step t, achievable by a behaviour policy $\pi = P(a|s)$, after making an observation (s) and taking an action (a) (see Methods)[19].

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function[20]. This instability has several causes: the correlations present in the sequence of observations, the fact that small updates to Q may significantly change the policy and therefore change the data distribution, and the correlations between the action-values (Q) and the target values $r + \gamma \max_{a'} Q(s', a')$. We address these instabilities with a novel variant of Q-learning, which uses two key ideas. First, we used a biologically inspired mechanism termed experience replay[21–23] that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution (see below for details). Second, we used an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target.

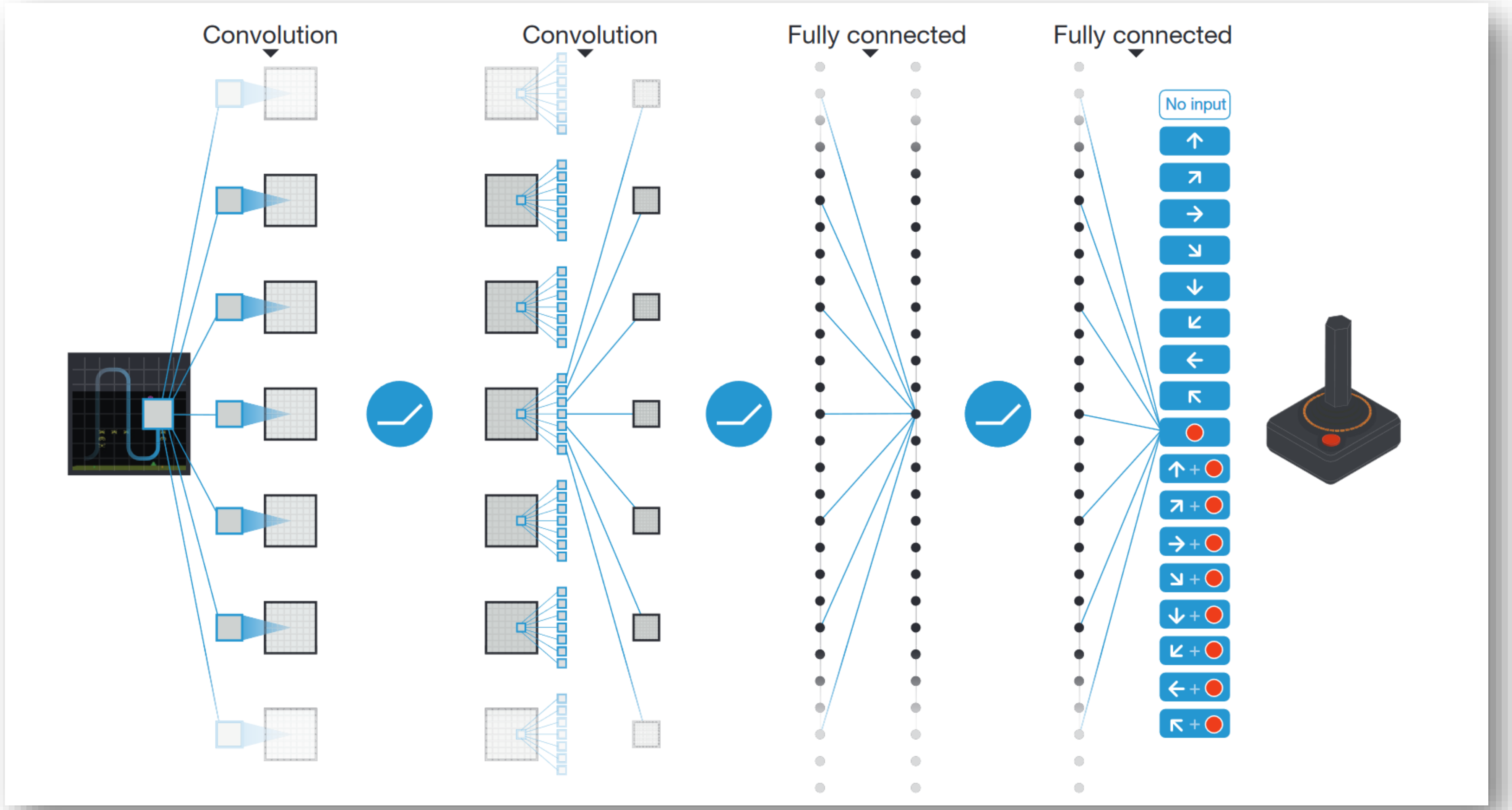
While other stable methods exist for training neural networks in the reinforcement learning setting, such as neural fitted Q-iteration[24], these methods involve the repeated training of networks *de novo* on hundreds of iterations. Consequently, these methods, unlike our algorithm, are too inefficient to be used successfully with large neural networks. We parameterize an approximate value function $Q(s,a;\theta_i)$ using the deep convolutional neural network shown in Fig. 1, in which θ_i are the parameters (that is, weights) of the Q-network at iteration i. To perform experience replay we store the agent's experiences $e_t = (s_t,a_t,r_t,s_{t+1})$ at each time-step t in a data set $D_t = \{e_1,\ldots,e_t\}$. During learning, we apply Q-learning updates, on samples (or minibatches) of experience $(s,a,r,s') \sim U(D)$, drawn uniformly at random from the pool of stored samples. The Q-learning update at iteration i uses the following loss function:

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

in which γ is the discount factor determining the agent's horizon, θ_i are the parameters of the Q-network at iteration i and θ_i^- are the network parameters used to compute the target at iteration i. The target network parameters θ_i^- are only updated with the Q-network parameters (θ_i) every C steps and are held fixed between individual updates (see Methods).

To evaluate our DQN agent, we took advantage of the Atari 2600 platform, which offers a diverse array of tasks ($n = 49$) designed to be

[1]Google DeepMind, 5 New Street Square, London EC4A 3TW, UK.
*These authors contributed equally to this work.


Convolution
Convolution
Fully connected
Fully connected
No input

Part 2

In this part we will cover ***direct policy optimization,*** with two big families of methods

- **Policy gradient methods**, Actor-Critic, A2C, A3C, PPO, TRPO

- **Evolutionary Strategies** (ES), namely Cross-Entropy Method (CEM) and Covariance Matrix Adaptation (CMA-ES)

In policy gradient methods, we consider **stochastic policies**

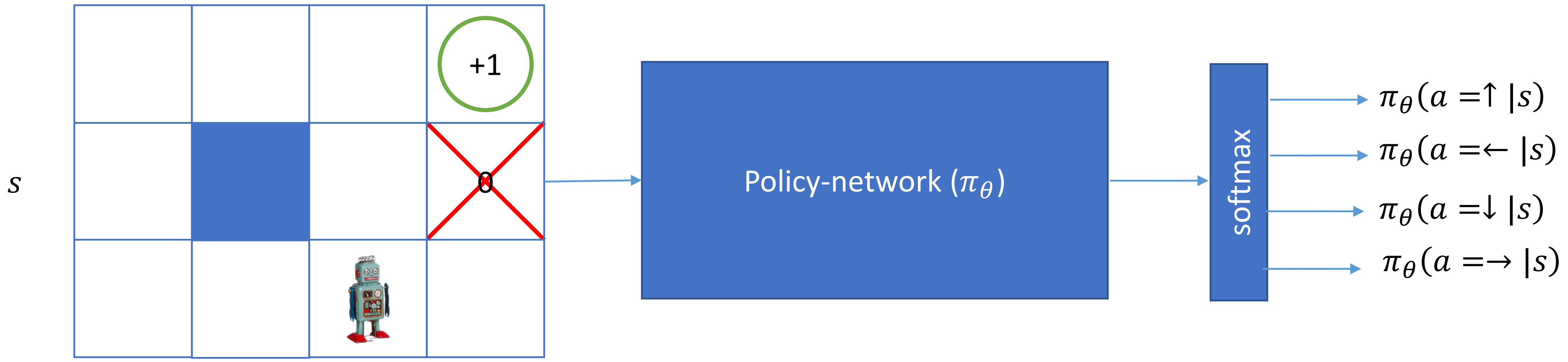
$$\pi(a|s)$$

(in Q-learning we learned deterministic policies $a = \pi(s)$)

More specifically, we consider **parameterized** stochastic policies (e.g. neural nets)

$$\pi_{\theta}(a|s)$$

π_{θ} is called the **policy network**


s
+1
0
Policy-network (π_θ)
softmax
$\pi_\theta(a = \uparrow | s)$
$\pi_\theta(a = \leftarrow | s)$
$\pi_\theta(a = \downarrow | s)$
$\pi_\theta(a = \rightarrow | s)$

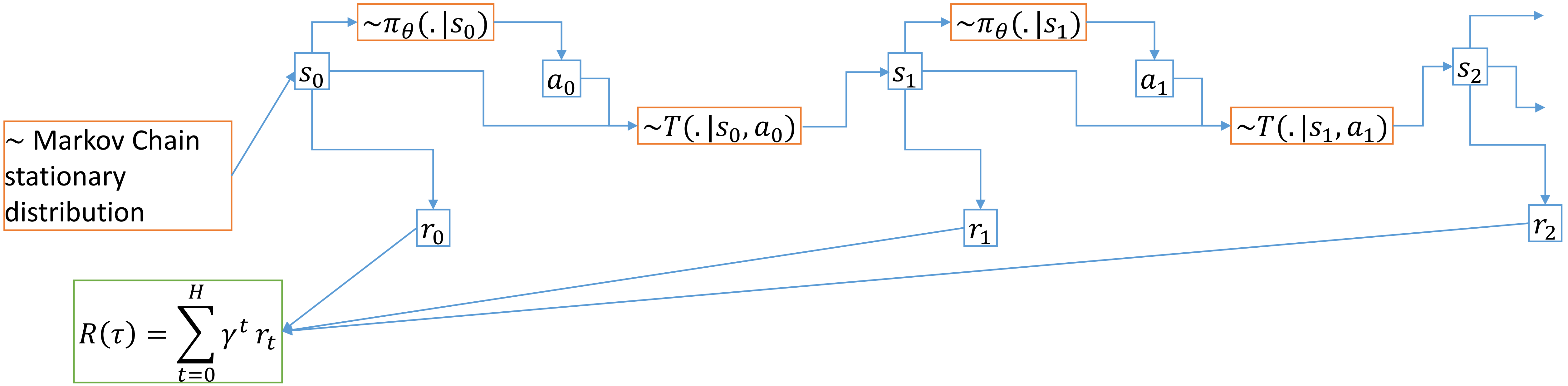
The objective of **policy gradient** is to solve

$$\underset{\theta}{\operatorname{argmax}}\, J(\theta) = \underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{\tau\sim P}[R(\tau)|\pi_\theta]$$

where τ is a **trajectory** (also called a **rollout**) that follows a probability distribution P

$$\tau = s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots$$

$P(\tau)$ is the probability to sample trajectory when sampling actions a_i from the policy π_θ and states s_i from the transition model T



In order to solve

$$\underset{\theta}{\operatorname{argmax}} J(\theta) = \mathbb{E}_{\tau \sim P}[R(\tau)|\pi_\theta]$$

We perform a **gradient ascent** by computing the **policy gradient**

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim P}[R(\tau)|\pi_\theta]$$

We use the **log likelihood trick** to compute the policy gradient

$$\nabla_\theta \mathbb{E}_{\tau\sim P}[R(\tau)|\pi_\theta] = \mathbb{E}_{\tau\sim P}[\nabla_\theta \log P(\tau|\pi_\theta) R(\tau)]$$

We decompose the probability of trajectory τ under policy π_θ

$$\begin{aligned}
\nabla_\theta \log P(\tau|\pi_\theta) &= \nabla_\theta \log \prod_{t=0}^{H} \pi_\theta(a_t|s_t) T(s_{t+1}|s_t, a_t) \\
&= \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t) + \sum_{t=0}^{H} \nabla_\theta \log T(s_{t+1}|s_t, a_t) \\
&= \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t)
\end{aligned}$$

We finally get the **policy gradient expression**

$$\nabla_\theta \mathbb{E}_{\tau \sim P}[R(\tau) | \pi_\theta] = \mathbb{E}_{\tau \sim P}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t | s_t)\, R(\tau)\right]$$

We finally get the **policy gradient expression**

$$\nabla_\theta \mathbb{E}_{\tau\sim P}[R(\tau)|\pi_\theta] = \mathbb{E}_{\tau\sim P}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)\right]$$

and use a basic **Monte-Carlo estimator** for the expectation, by sampling trajectories (rollouts) from the current policy

Monte-Carlo estimator is unbiased, but very noisy (high variance)

To reduce variance, we can consider only **future return** at every step of the summation

$$\nabla_\theta \mathbb{E}_{\tau\sim P}[R(\tau)|\pi_\theta] = \mathbb{E}_{\tau\sim P}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R_{t}(\tau)\right]$$

where R_t denotes the **future return starting from time-step *t***

$$R(\tau) = \sum_{t'=0}^{+\infty} \gamma^{t'} r_{t'}$$

$$R_{t}(\tau) = \sum_{t'=t}^{+\infty} \gamma^{t'} r_{t'}$$

To further reduce variance, we can subtract a **baseline**

$$\nabla_\theta \mathbb{E}_{\tau\sim P}[R(\tau)|\pi_\theta] = \mathbb{E}_{\tau\sim P}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t)\,(R_t(\tau) - b)\right]$$

The estimator is still unbiased, *if b doesn’t depend on the actions*

$$\nabla_\theta \mathbb{E}_{\tau\sim P}[R(\tau)|\pi_\theta] = \mathbb{E}_{\tau\sim P}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t)\, A_t\right]$$

$A_t = R_t(\tau) - b$ is called the **Advantage**

There are many possible choices for baselines *b* (constant baselines, time-dependent baselines, state-dependent baselines), it can be shown that a good choice for the baseline (in terms of variance reduction), is the **value of the state** $b = V(s_t)$

$$\nabla_\theta \mathbb{E}_{\tau\sim P}[R(\tau)|\pi_\theta] = \mathbb{E}_{\tau\sim P}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t)\,(R_t(\tau) - V(s_t))\right]$$

However, we don't know the value function, so we estimate it (learn it) along the policy by **parameterizing** it with parameter vector w (e.g. neural network V_w)

$$\nabla_\theta \mathbb{E}_{\tau \sim P}[R(\tau) | \pi_\theta] = \mathbb{E}_{\tau \sim P}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t | s_t) \left(R_t(\tau) - V_{w}(s_t)\right)\right]$$

We can also use the following expressions

$$\begin{aligned}\nabla_\theta \mathbb{E}_{\tau\sim P}[R(\tau)|\pi_\theta] &= \mathbb{E}_{\tau\sim P}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t)\,(Q(s_t,a_t) - V_w(s_t))\right] \\ &= \mathbb{E}_{\tau\sim P}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t)\,(r_t + \gamma V_w(s_{t+1}) - V_w(s_t))\right] \\ &= \mathbb{E}_{\tau\sim P}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t)\,(r_t + \gamma r_{t+1} + \gamma^2 V_w(s_{t+2}) - V_w(s_t))\right]\end{aligned}$$

This is the **policy gradient theorem**

Actor-Critic algorithm

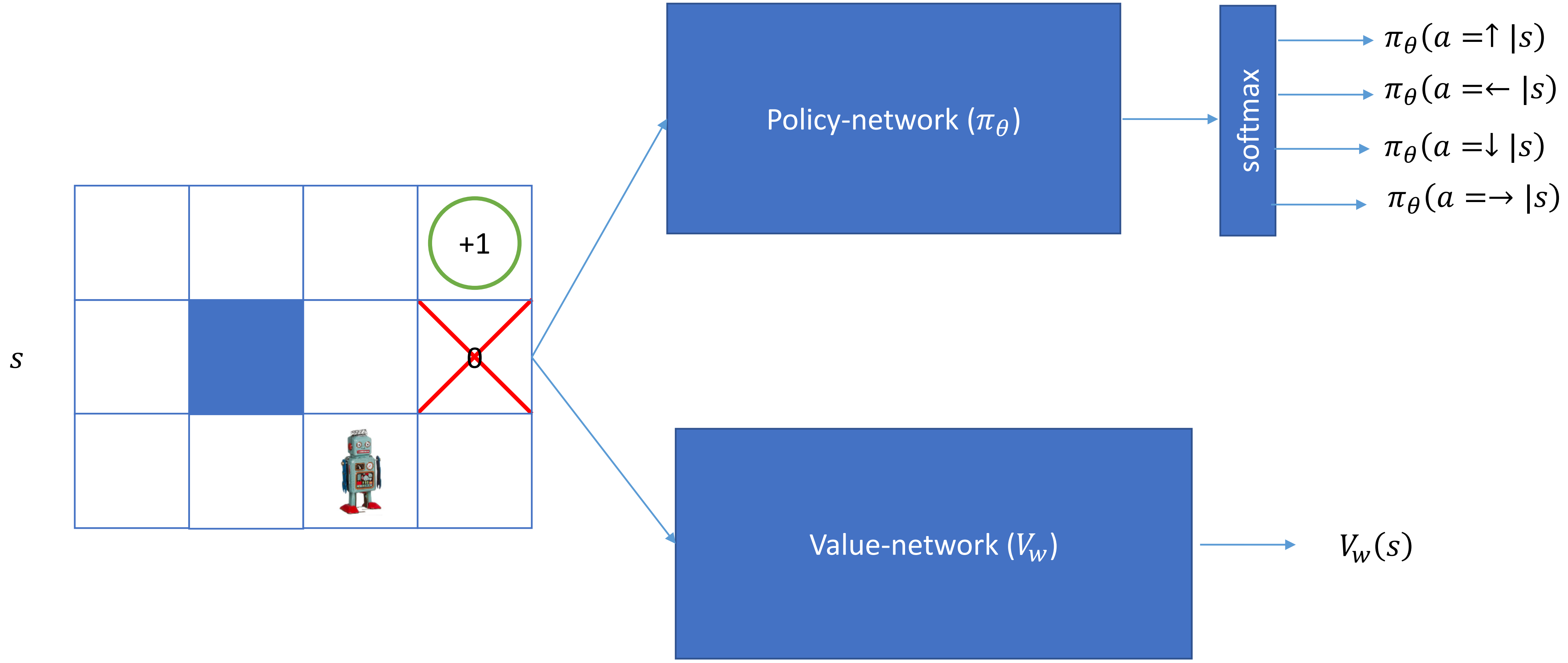
- Initialize random weights θ for the policy-network (actor network) and random weights w for the Value-network (critic network)
- For N iterations until policy converges:
    - Sample M trajectories τ from current policy π_θ
    - For each timestep t along each trajectory, compute the advantage $A_t = R_t(\tau) - V_w(s_t)$
    - Estimate the policy gradient using MC estimation on

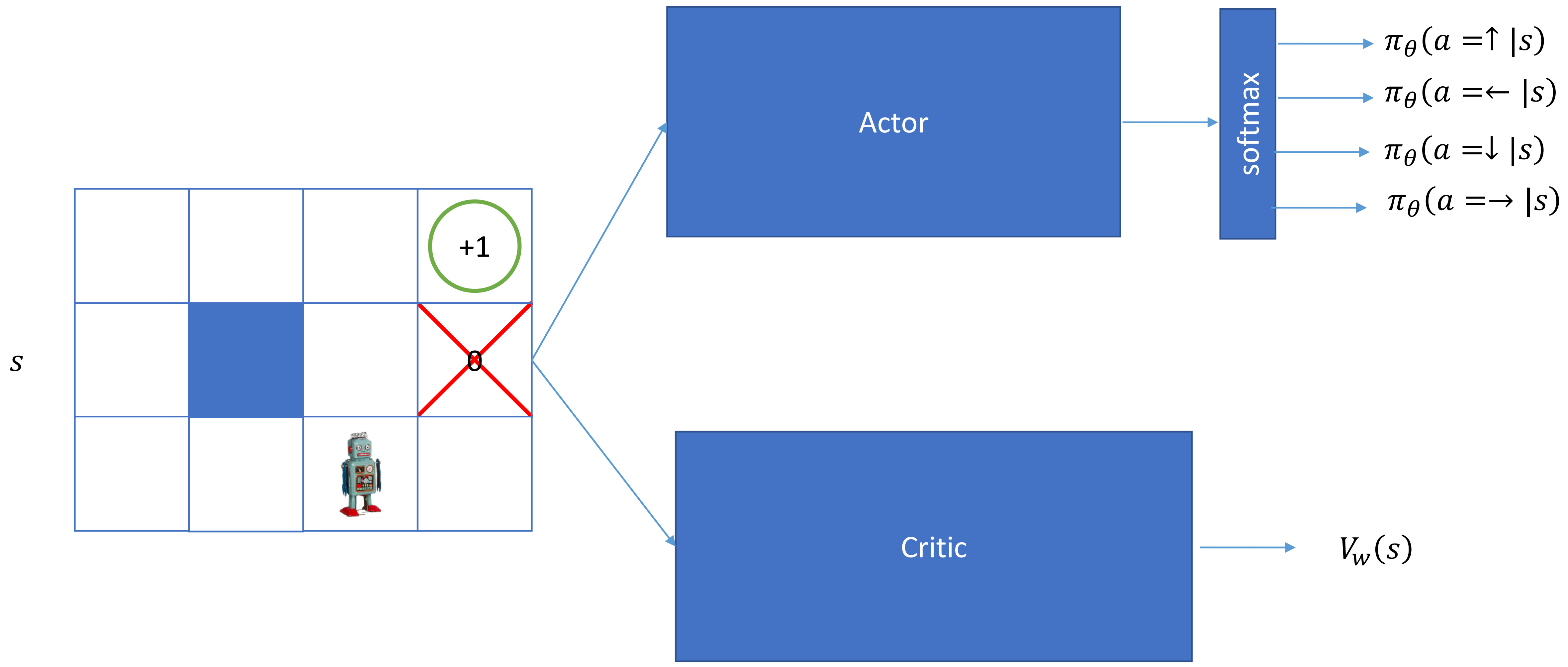
$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim P}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t)\, A_t\right]$$

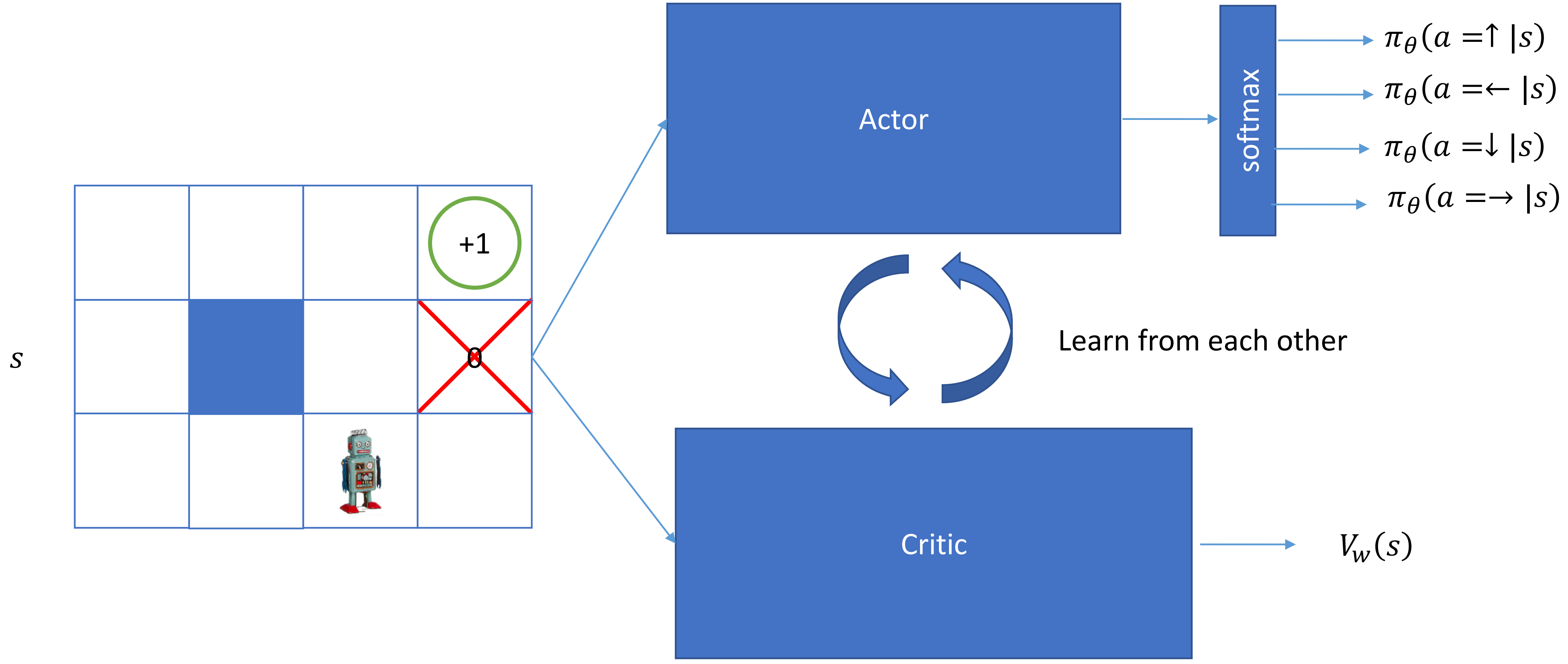
And perform gradient ascent step along $\nabla_\theta J(\theta)$

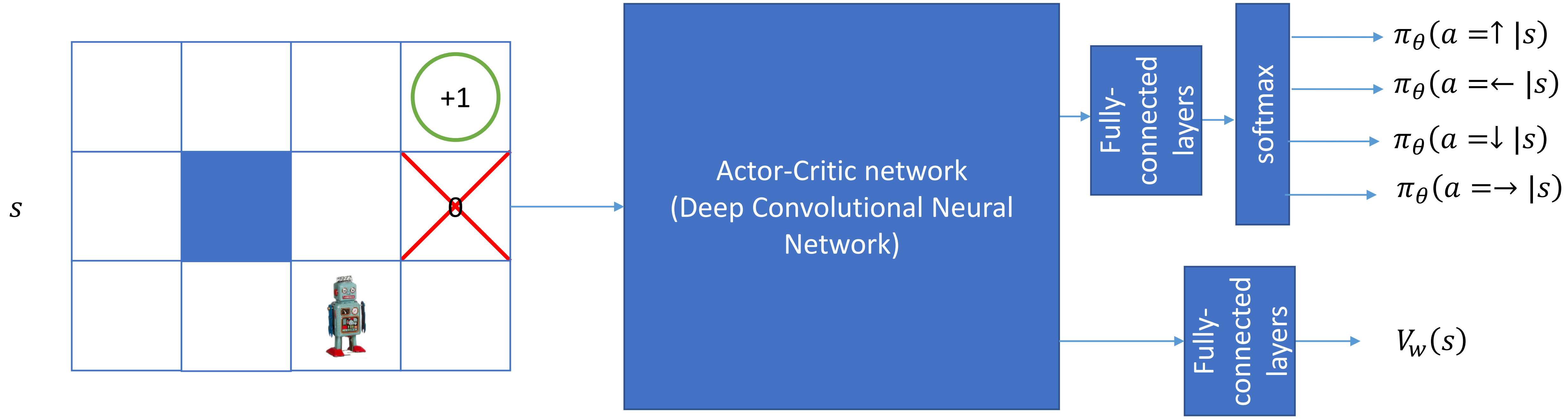
- Refit the baseline

$$w = \underset{w}{\operatorname{argmin}} \sum \|V_w(s_t) - R_t(\tau)\|^2$$

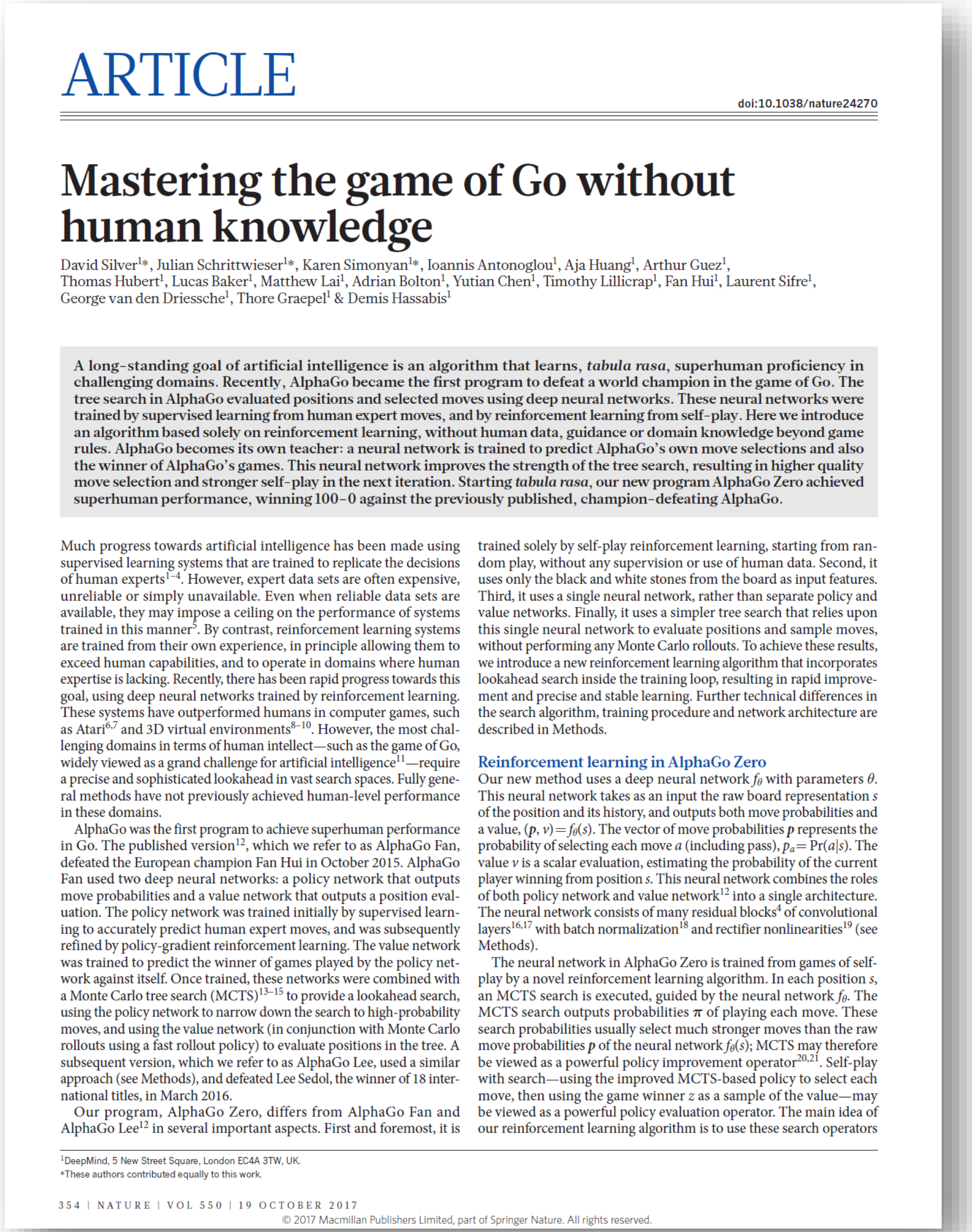

s
+1
0
Policy-network (π_θ)
softmax
$\pi_\theta(a = \uparrow | s)$
$\pi_\theta(a = \leftarrow | s)$
$\pi_\theta(a = \downarrow | s)$
$\pi_\theta(a = \rightarrow | s)$
Value-network (V_w)
$V_w(s)$


s
+1
0
Actor
softmax
$\pi_\theta(a = \uparrow | s)$
$\pi_\theta(a = \leftarrow | s)$
$\pi_\theta(a = \downarrow | s)$
$\pi_\theta(a = \rightarrow | s)$
Critic
$V_w(s)$


s
+1
0
Actor
softmax
$\pi_\theta(a = \uparrow | s)$
$\pi_\theta(a = \leftarrow | s)$
$\pi_\theta(a = \downarrow | s)$
$\pi_\theta(a = \rightarrow | s)$
Learn from each other
Critic
$V_w(s)$


s
+1
0
Actor-Critic network
(Deep Convolutional Neural Network)
Fully-connected layers
softmax
$\pi_\theta(a = \uparrow | s)$
$\pi_\theta(a = \leftarrow | s)$
$\pi_\theta(a = \downarrow | s)$
$\pi_\theta(a = \rightarrow | s)$
Fully-connected layers
$V_w(s)$

nature
THE INTERNATIONAL WEEKLY JOURNAL OF SCIENCE
At last — a computer program that can beat a champion Go player PAGE 484
ALL SYSTEMS GO
CONSERVATION
SONGBIRDS À LA CARTE
Illegal harvest of millions of Mediterranean birds
PAGE 452
RESEARCH ETHICS
SAFEGUARD TRANSPARENCY
Don't let openness backfire on individuals
PAGE 459
POPULAR SCIENCE
WHEN GENES GOT 'SELFISH'
Dawkins's calling card 40 years on
PAGE 462
NATUREASIA.COM
28 January 2016
Vol. 529, No. 7587



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 move selection and stronger self-play in the next iteration. Starting *tabula rasa*, our new program AlphaGo Zero achieved superhuman performance, winning 100–0 against the previously published, champion-defeating AlphaGo.

Much progress towards artificial intelligence has been made using supervised learning systems that are trained to replicate the decisions of human experts[1–4]. However, expert data sets are often expensive, unreliable or simply unavailable. Even when reliable data sets are available, they may impose a ceiling on the performance of systems trained in this manner[5]. By contrast, reinforcement learning systems are trained from their own experience, in principle allowing them to exceed human capabilities, and to operate in domains where human expertise is lacking. Recently, there has been rapid progress towards this goal, using deep neural networks trained by reinforcement learning. These systems have outperformed humans in computer games, such as Atari[6,7] and 3D virtual environments[8–10]. However, the most challenging domains in terms of human intellect—such as the game of Go, widely viewed as a grand challenge for artificial intelligence[11]—require a precise and sophisticated lookahead in vast search spaces. Fully general methods have not previously achieved human-level performance in these domains.

AlphaGo was the first program to achieve superhuman performance in Go. The published version[12], which we refer to as AlphaGo Fan, defeated the European champion Fan Hui in October 2015. AlphaGo Fan used two deep neural networks: a policy network that outputs move probabilities and a value network that outputs a position evaluation. The policy network was trained initially by supervised learning to accurately predict human expert moves, and was subsequently refined by policy-gradient reinforcement learning. The value network was trained to predict the winner of games played by the policy network against itself. Once trained, these networks were combined with a Monte Carlo tree search (MCTS)[13–15] to provide a lookahead search, using the policy network to narrow down the search to high-probability moves, and using the value network (in conjunction with Monte Carlo rollouts using a fast rollout policy) to evaluate positions in the tree. A subsequent version, which we refer to as AlphaGo Lee, used a similar approach (see Methods), and defeated Lee Sedol, the winner of 18 international titles, in March 2016.

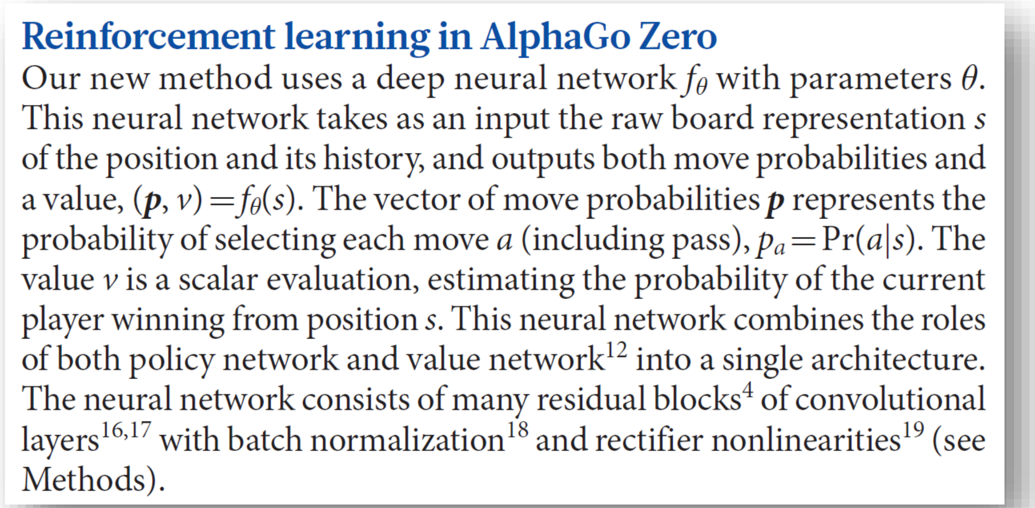
Our program, AlphaGo Zero, differs from AlphaGo Fan and AlphaGo Lee[12] in several important aspects. First and foremost, it is trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data. Second, it uses only the black and white stones from the board as input features. Third, it uses a single neural network, rather than separate policy and value networks. Finally, it uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte Carlo rollouts. To achieve these results, we introduce a new reinforcement learning algorithm that incorporates lookahead search inside the training loop, resulting in rapid improvement and precise and stable learning. Further technical differences in the search algorithm, training procedure and network architecture are described in Methods.

Reinforcement learning in AlphaGo Zero

Our new method uses a deep neural network f_θ with parameters θ. This neural network takes as an input the raw board representation s of the position and its history, and outputs both move probabilities and a value, $(\boldsymbol{p}, v) = f_\theta(s)$. The vector of move probabilities $\boldsymbol{p}$ represents the probability of selecting each move a (including pass), $p_a = \Pr(a|s)$. The value v is a scalar evaluation, estimating the probability of the current player winning from position s. This neural network combines the roles of both policy network and value network[12] into a single architecture. The neural network consists of many residual blocks[4] of convolutional layers[16,17] with batch normalization[18] and rectifier nonlinearities[19] (see Methods).

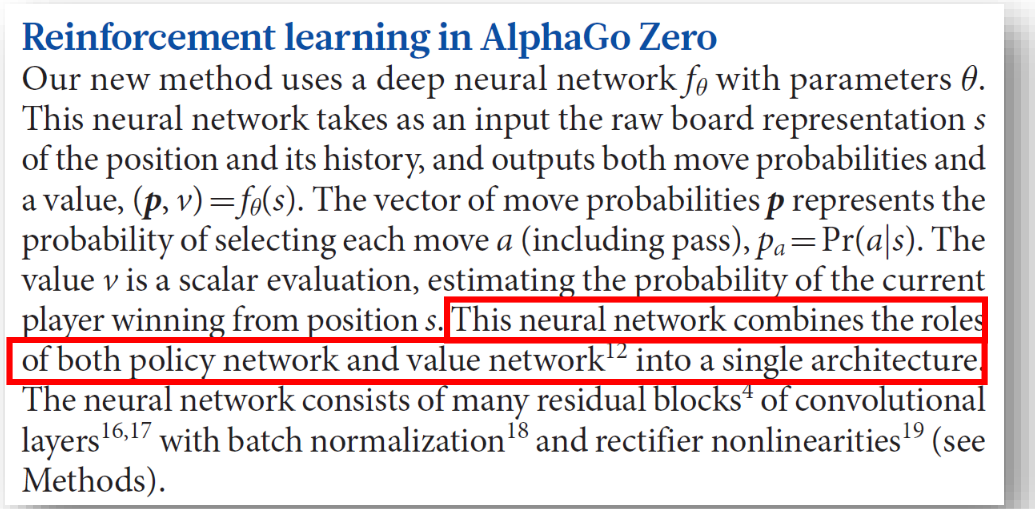
The neural network in AlphaGo Zero is trained from games of self-play by a novel reinforcement learning algorithm. In each position s, an MCTS search is executed, guided by the neural network f_θ. The MCTS search outputs probabilities $\boldsymbol{\pi}$ of playing each move. These search probabilities usually select much stronger moves than the raw move probabilities $\boldsymbol{p}$ of the neural network $f_\theta(s)$; MCTS may therefore be viewed as a powerful policy improvement operator[20,21]. Self-play with search—using the improved MCTS-based policy to select each move, then using the game winner z as a sample of the value—may be viewed as a powerful policy evaluation operator. The main idea of our reinforcement learning algorithm is to use these search operators

[1]DeepMind, 5 New Street Square, London EC4A 3TW, UK.
*These authors contributed equally to this work.



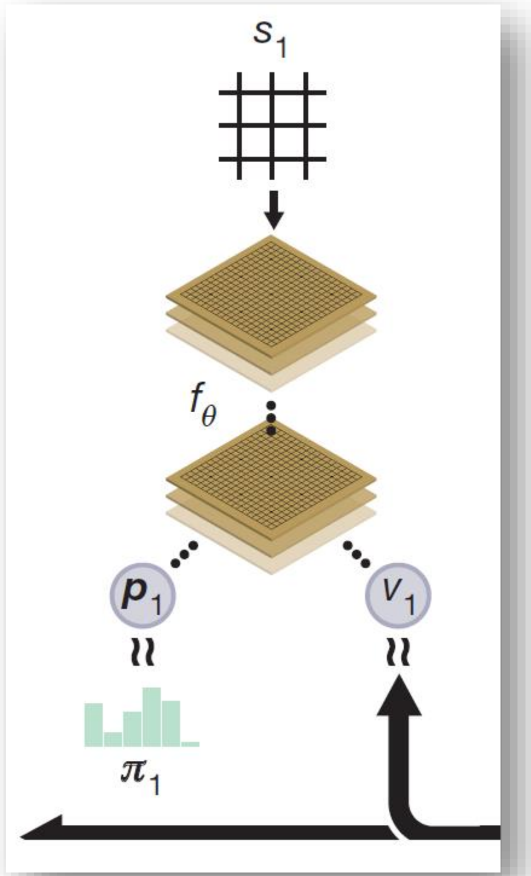
Reinforcement learning in AlphaGo Zero

Our new method uses a deep neural network f_θ with parameters θ. This neural network takes as an input the raw board representation s of the position and its history, and outputs both move probabilities and a value, $(\boldsymbol{p}, v) = f_\theta(s)$. The vector of move probabilities $\boldsymbol{p}$ represents the probability of selecting each move a (including pass), $p_a = \Pr(a|s)$. The value v is a scalar evaluation, estimating the probability of the current player winning from position s. This neural network combines the roles of both policy network and value network[12] into a single architecture. The neural network consists of many residual blocks[4] of convolutional layers[16,17] with batch normalization[18] and rectifier nonlinearities[19] (see Methods).



Reinforcement learning in AlphaGo Zero

Our new method uses a deep neural network f_θ with parameters θ. This neural network takes as an input the raw board representation s of the position and its history, and outputs both move probabilities and a value, $(\boldsymbol{p}, v) = f_\theta(s)$. The vector of move probabilities $\boldsymbol{p}$ represents the probability of selecting each move a (including pass), $p_a = \Pr(a|s)$. The value v is a scalar evaluation, estimating the probability of the current player winning from position s. This neural network combines the roles of both policy network and value network[12] into a single architecture. The neural network consists of many residual blocks[4] of convolutional layers[16,17] with batch normalization[18] and rectifier nonlinearities[19] (see Methods).


s_1
f_θ
$\boldsymbol{p}_1$
v_1
≈
≈
$\boldsymbol{\pi}_1$

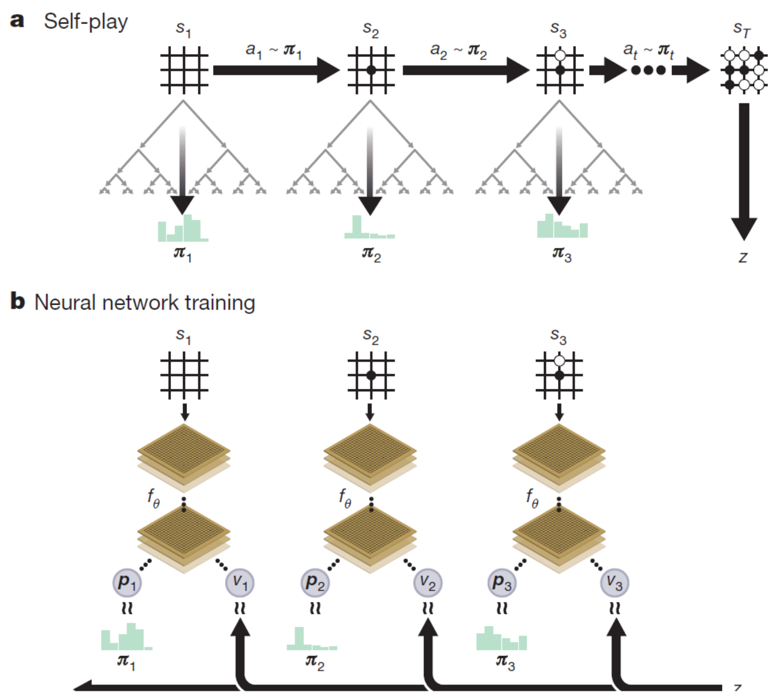


Figure 1 | Self-play reinforcement learning in AlphaGo Zero. a, The program plays a game $s_1, ..., s_T$ against itself. In each position s_t, an MCTS α_θ is executed (see Fig. 2) using the latest neural network f_θ. Moves are selected according to the search probabilities computed by the MCTS, $a_t \sim \boldsymbol{\pi}_t$. The terminal position s_T is scored according to the rules of the game to compute the game winner z. **b**, Neural network training in AlphaGo Zero. The neural network takes the raw board position s_t as its input, passes it through many convolutional layers with parameters θ, and outputs both a vector $\boldsymbol{p}_t$, representing a probability distribution over moves, and a scalar value v_t, representing the probability of the current player winning in position s_t. The neural network parameters θ are updated to maximize the similarity of the policy vector $\boldsymbol{p}_t$ to the search probabilities $\boldsymbol{\pi}_t$, and to minimize the error between the predicted winner v_t and the game winner z (see equation (1)). The new parameters are used in the next iteration of self-play as in **a**.

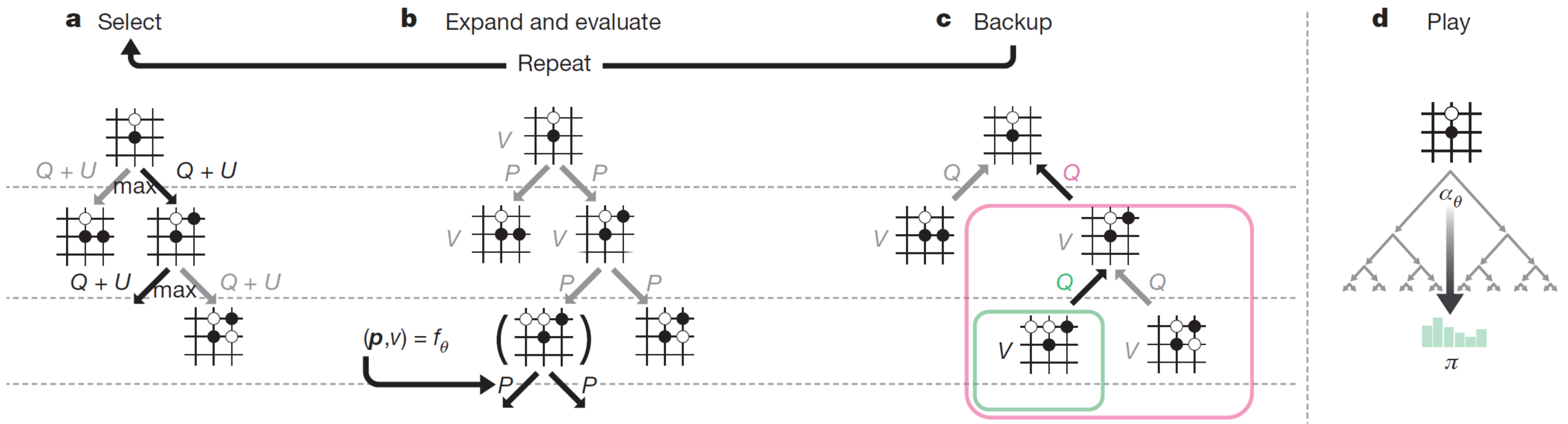


Figure 2 | MCTS in AlphaGo Zero. a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus an upper confidence bound U that depends on a stored prior probability P and visit count N for that edge (which is incremented once traversed). **b**, The leaf node is expanded and the associated position s is evaluated by the neural network $(P(s, \cdot), V(s)) = f_\theta(s)$; the vector of P values are stored in the outgoing edges from s. **c**, Action value Q is updated to track the mean of all evaluations V in the subtree below that action. **d**, Once the search is complete, search probabilities $\boldsymbol{\pi}$ are returned, proportional to $N^{1/\tau}$, where N is the visit count of each move from the root state and τ is a parameter controlling temperature.

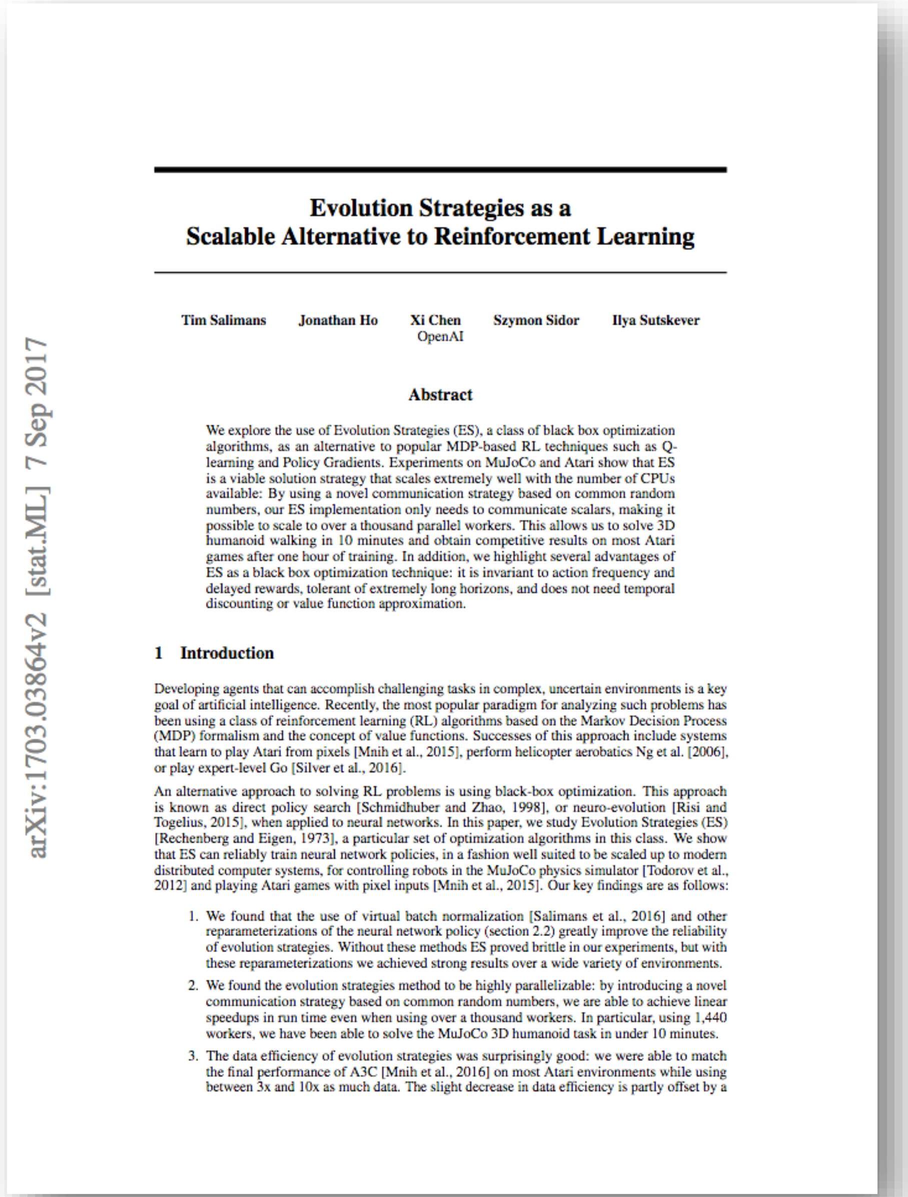
Evolution Strategies

Evolution Strategies as a Scalable Alternative to Reinforcement Learning

We've discovered that **evolution strategies (ES)**, an optimization technique that's been known for decades, rivals the performance of standard **reinforcement learning (RL)** techniques on modern RL benchmarks (e.g. Atari/MuJoCo), while overcoming many of RL's inconveniences.

MARCH 24, 2017
12 MINUTE READ

In particular, ES is simpler to implement (there is no need for backpropagation), it is easier to scale in a distributed setting, it does not suffer in settings with sparse rewards, and has fewer hyperparameters. This outcome is surprising because ES resembles simple hill-climbing in a high-dimensional space based only on finite differences along a few random directions at each step.



Evolution Strategies as a Scalable Alternative to Reinforcement Learning

Tim Salimans **Jonathan Ho** **Xi Chen** **Szymon Sidor** **Ilya Sutskever**
OpenAI

Abstract

We explore the use of Evolution Strategies (ES), a class of black box optimization algorithms, as an alternative to popular MDP-based RL techniques such as Q-learning and Policy Gradients. Experiments on MuJoCo and Atari show that ES is a viable solution strategy that scales extremely well with the number of CPUs available: By using a novel communication strategy based on common random numbers, our ES implementation only needs to communicate scalars, making it possible to scale to over a thousand parallel workers. This allows us to solve 3D humanoid walking in 10 minutes and obtain competitive results on most Atari games after one hour of training. In addition, we highlight several advantages of ES as a black box optimization technique: it is invariant to action frequency and delayed rewards, tolerant of extremely long horizons, and does not need temporal discounting or value function approximation.

1 Introduction

Developing agents that can accomplish challenging tasks in complex, uncertain environments is a key goal of artificial intelligence. Recently, the most popular paradigm for analyzing such problems has been using a class of reinforcement learning (RL) algorithms based on the Markov Decision Process (MDP) formalism and the concept of value functions. Successes of this approach include systems that learn to play Atari from pixels [Mnih et al., 2015], perform helicopter aerobatics Ng et al. [2006], or play expert-level Go [Silver et al., 2016].

An alternative approach to solving RL problems is using black-box optimization. This approach is known as direct policy search [Schmidhuber and Zhao, 1998], or neuro-evolution [Risi and Togelius, 2015], when applied to neural networks. In this paper, we study Evolution Strategies (ES) [Rechenberg and Eigen, 1973], a particular set of optimization algorithms in this class. We show that ES can reliably train neural network policies, in a fashion well suited to be scaled up to modern distributed computer systems, for controlling robots in the MuJoCo physics simulator [Todorov et al., 2012] and playing Atari games with pixel inputs [Mnih et al., 2015]. Our key findings are as follows:

1. We found that the use of virtual batch normalization [Salimans et al., 2016] and other reparameterizations of the neural network policy (section 2.2) greatly improve the reliability of evolution strategies. Without these methods ES proved brittle in our experiments, but with these reparameterizations we achieved strong results over a wide variety of environments.
2. We found the evolution strategies method to be highly parallelizable: by introducing a novel communication strategy based on common random numbers, we are able to achieve linear speedups in run time even when using over a thousand workers. In particular, using 1,440 workers, we have been able to solve the MuJoCo 3D humanoid task in under 10 minutes.
3. The data efficiency of evolution strategies was surprisingly good: we were able to match the final performance of A3C [Mnih et al., 2016] on most Atari environments while using between 3x and 10x as much data. The slight decrease in data efficiency is partly offset by a

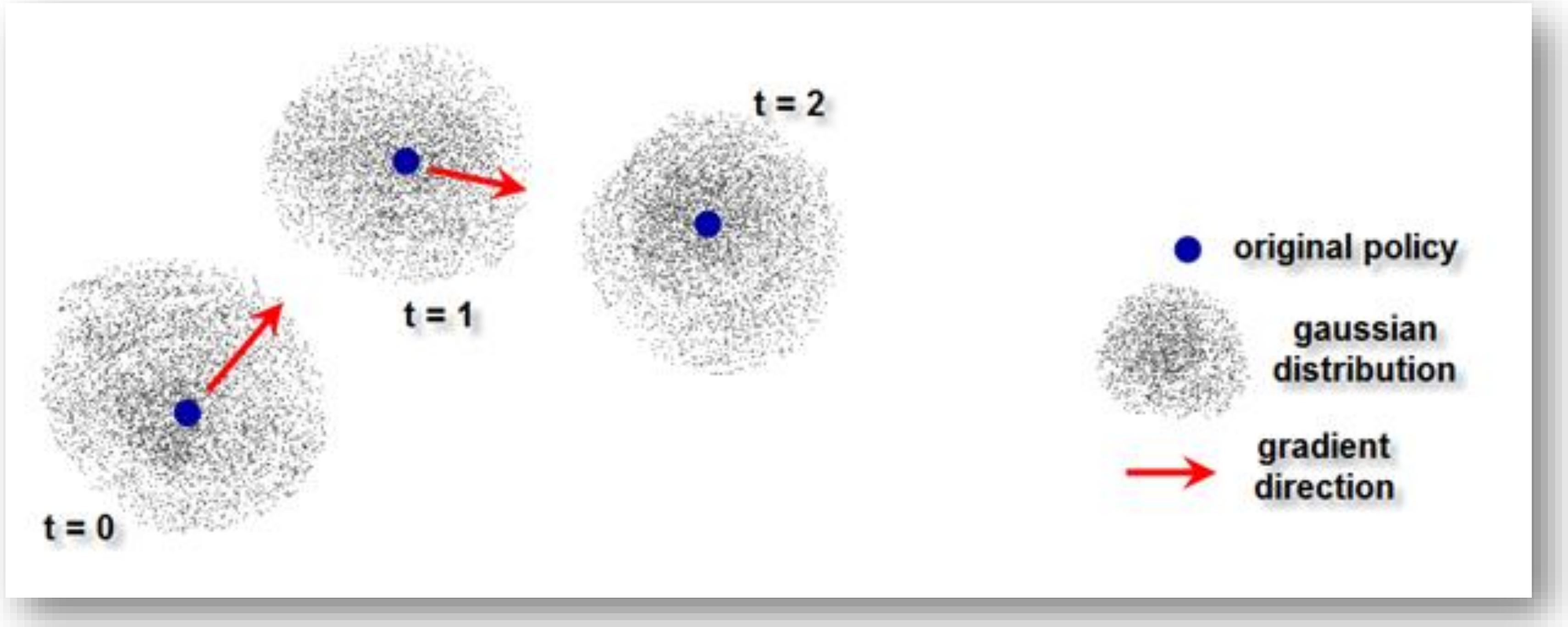
Example of a basic evolution strategy: the **Cross-Entropy Method** (CEM-ES)

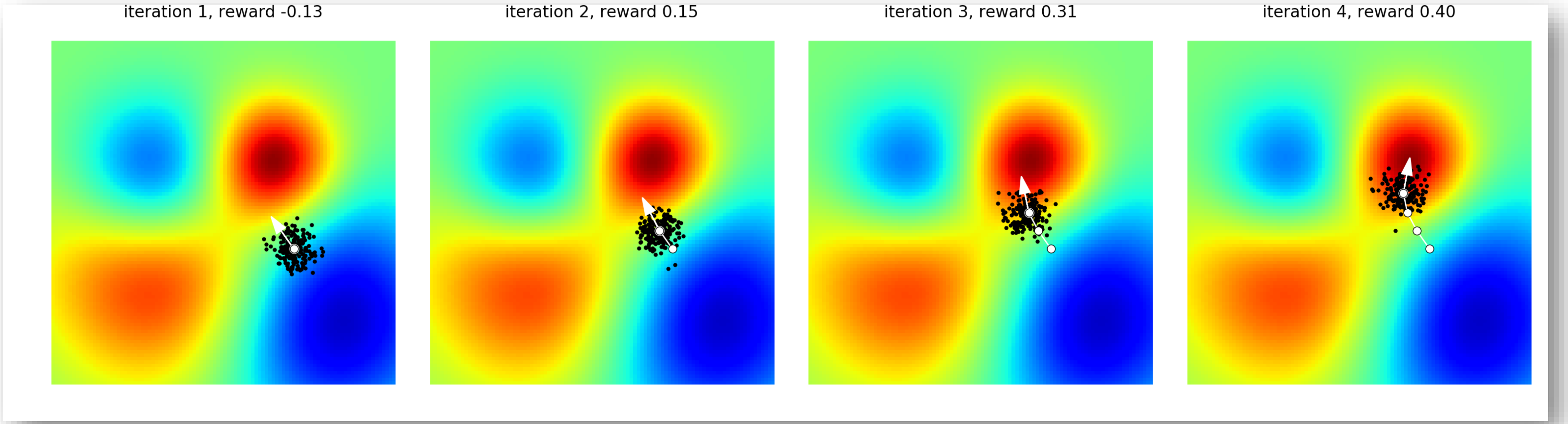
Work with parameterized policies π_θ

Assume that $\theta \sim \mathcal{N}(\mu, \sigma I)$

For N iterations (**generations**) until convergence:

- Sample n candidates $\theta_i \sim \mathcal{N}(\mu, \sigma I)$
- Executes rollouts (trajectories) τ_i following each policy π_{θ_i} and evaluate $\mathbb{E}[R(\tau_i) | \pi_{\theta_i}]$ with Monte Carlo
- Select $p\%$ best policies θ_i from these simulations (called the **elite set**)
- Refit μ and σ to the elite set (e.g. with Maximum Likelihood Estimation) for the next generation


t = 2
t = 1
t = 0
original policy
gaussian distribution
gradient direction


iteration 1, reward -0.13
iteration 2, reward 0.15
iteration 3, reward 0.31
iteration 4, reward 0.40

Example of a basic evolution strategy: the **Cross-Entropy Method** (CEM-ES)

Work with parameterized policies π_θ

Assume that $\theta \sim \mathcal{N}(\mu, \sigma I)$

For N iterations (**generations**) until convergence:

- Sample n candidates $\theta_i \sim \mathcal{N}(\mu, \sigma I)$
- Executes rollouts (trajectories) τ_i following each policy π_{θ_i} and evaluate $\mathbb{E}[R(\tau_i) | \pi_{\theta_i}]$ with Monte Carlo
- Select $p\%$ best policies θ_i from these simulations (called the **elite set**)
- Refit μ and σ to the elite set (e.g. with Maximum Likelihood Estimation) for the next generation

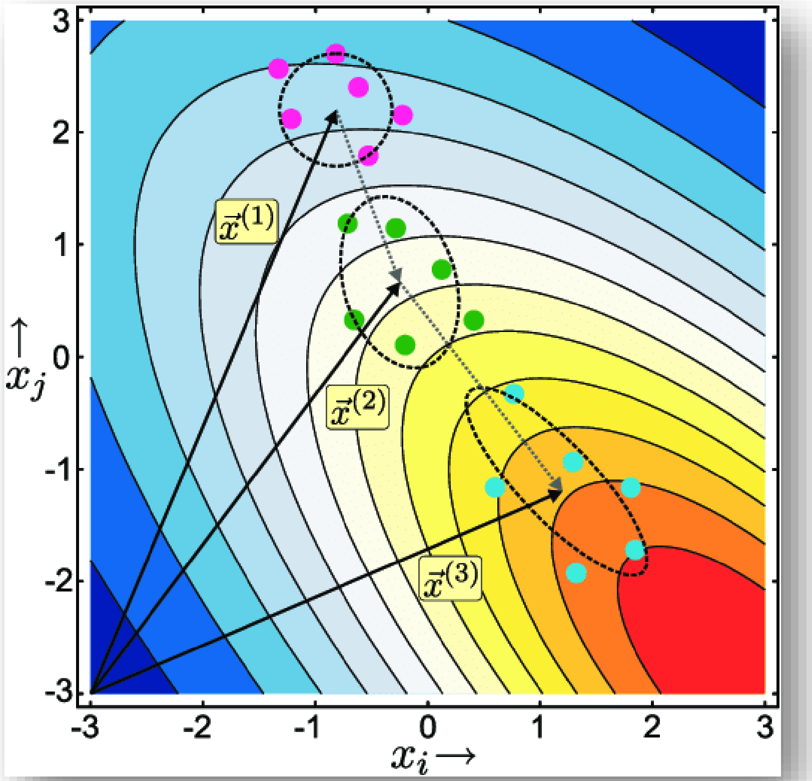
More advanced evolution strategy: using **Covariance Matrix Adaptation** (**CMA-ES**)

Work with parameterized policies π_θ

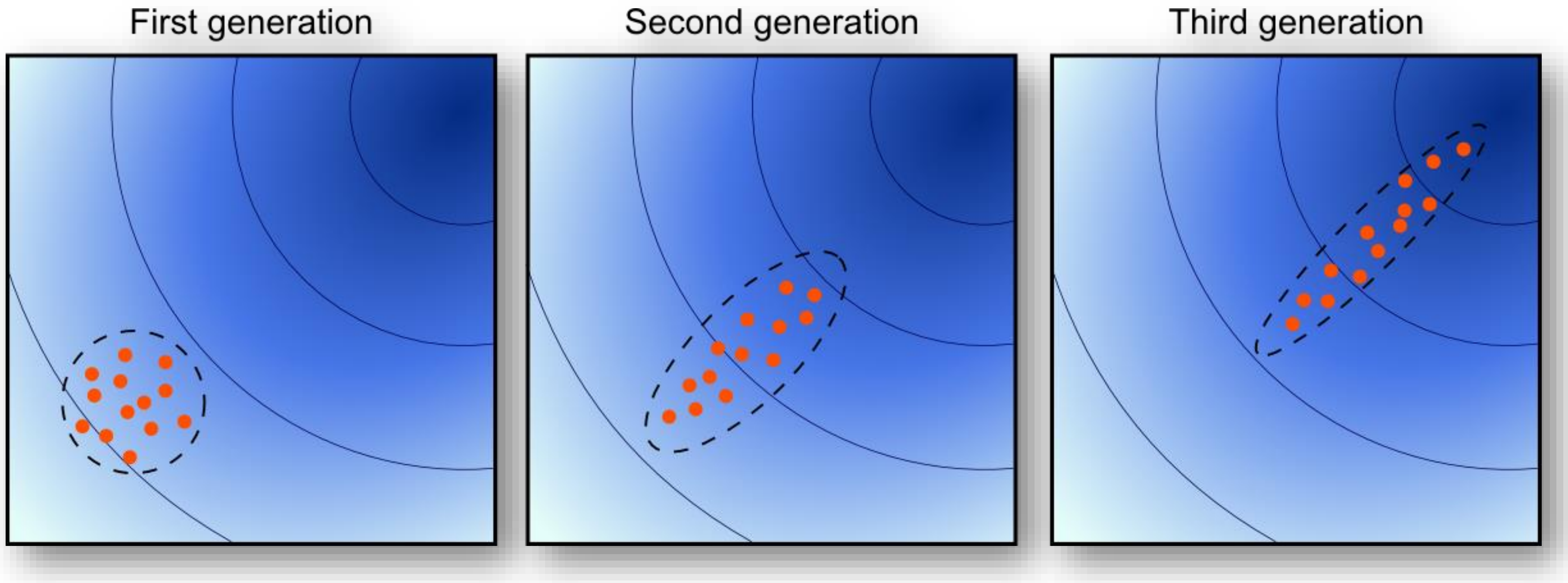
Assume that $\theta \sim \mathcal{N}(\mu, C)$

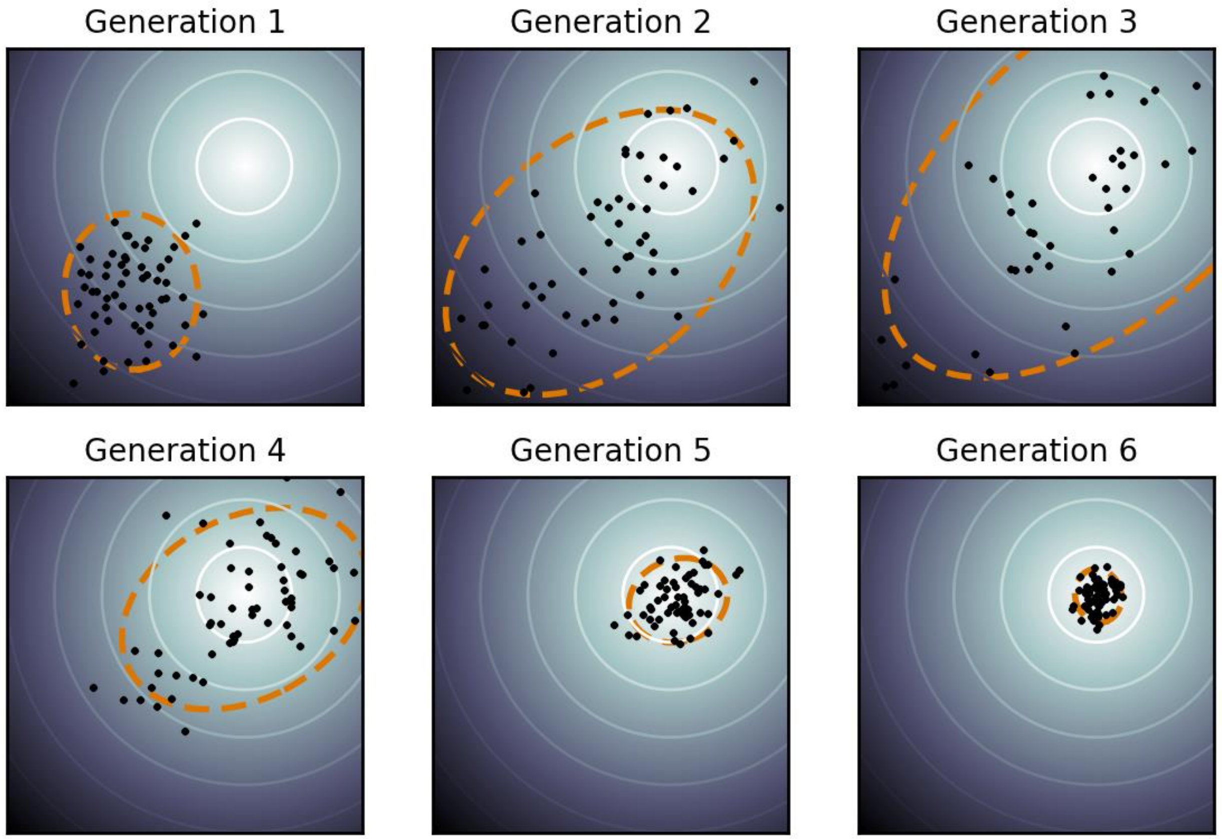
For N iterations (**generations**) until convergence:

- Sample n candidates $\theta_i \sim \mathcal{N}(\mu, C)$
- Executes rollouts (trajectories) τ_i following each policy π_{θ_i} and evaluate $\mathbb{E}[R(\tau_i)|\pi_{\theta_i}]$ with Monte Carlo
- Select $p\%$ best policies θ_i from these simulations (called the **elite set**)
- Refit μ (e.g. with Maximum Likelihood Estimation)
- adapt C as $C = \alpha C + (1 - \alpha) y y^T$, where $y = (\mu_{new} - \mu_{old}) / ||\mu_{new} - \mu_{old}||$ (direction of the search)



$\vec{x}^{(1)}$
$\vec{x}^{(2)}$
$\vec{x}^{(3)}$
$x_i \rightarrow$
$\uparrow x_j$
-3
-2
-1
0
1
2
3


First generation
Second generation
Third generation


Generation 1
Generation 2
Generation 3
Generation 4
Generation 5
Generation 6