UNIVERSITY OF TECHNOLOGY
IN THE EUROPEAN CAPITAL OF CULTURE
CHEMNITZ

Deep Reinforcement Learning

Advantage actor-critic (A2C, A3C)

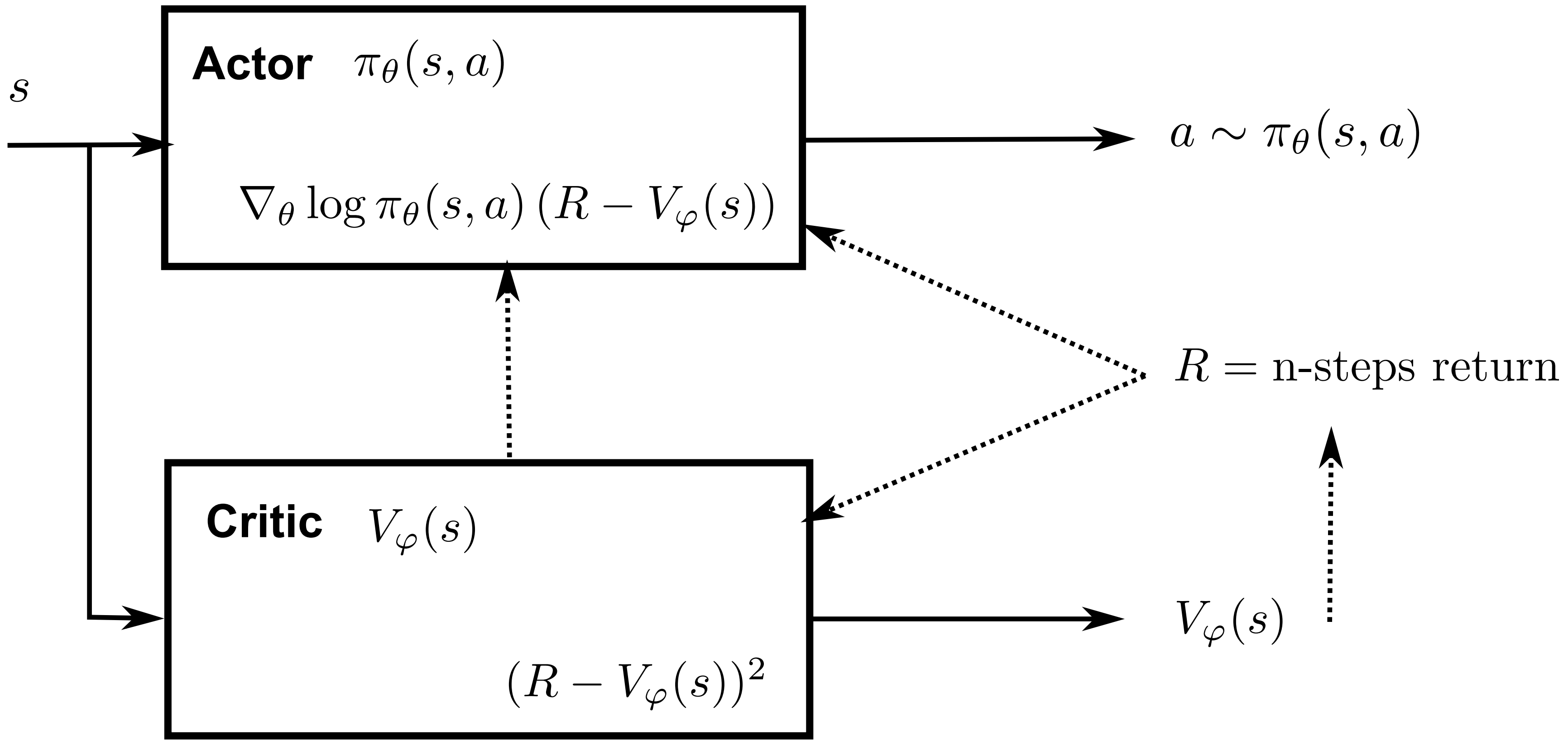
Julien Vitay
Professur für Künstliche Intelligenz - Fakultät für Informatik

1 - Distributed RL

Advantage actor-critic

- Let's consider an **n-step advantage actor-critic**:

$$
\begin{aligned}
A_t^n &= R_t^n - V_\varphi(s_t) \\
&= \sum_{k=0}^{n-1} \gamma^k \, r_{t+k+1} + \gamma^n \, V_\varphi(s_{t+n}) - V_\varphi(s_t)
\end{aligned}
$$


s
Actor $\pi_\theta(s, a)$
$\nabla_\theta \log \pi_\theta(s, a) \, (R - V_\varphi(s))$
$a \sim \pi_\theta(s, a)$
$R =$ n-steps return
Critic $V_\varphi(s)$
$(R - V_\varphi(s))^2$
$V_\varphi(s)$

- The **actor** $\pi_\theta(s, a)$ uses the PG theorem to learn the policy:

$$\nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{s_t \sim \rho_\theta, a_t \sim \pi_\theta} [\nabla_\theta \log \pi_\theta(s_t, a_t) \, (R_t^n - V_\varphi(s_t))]$$

- The **critic** $V_\varphi(s)$ approximates the value of each state:

$$\mathcal{L}(\varphi) = \mathbb{E}_{s_t \sim \rho_\theta, a_t \sim \pi_\theta} [(R_t^n - V_\varphi(s_t))^2]$$

Advantage actor-critic

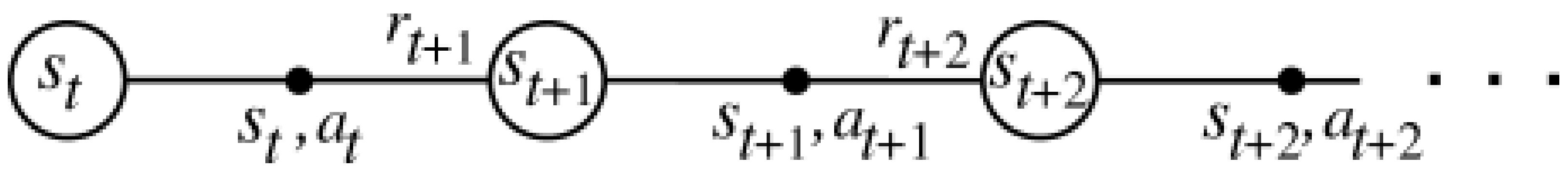
- The advantage actor-critic is strictly **on-policy**:
  - The critic **must** evaluate actions selected the current version of the actor π_θ, not an old version or another policy.
  - The actor must learn from the current value function $V^{\pi_\theta} \approx V_\varphi$.

$$\begin{cases} \nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{s_t \sim \rho_\theta, a_t \sim \pi_\theta}[\nabla_\theta \log \pi_\theta (s_t, a_t) \, (R^n_t - V_\varphi(s_t))] \\ \\ \mathcal{L}(\varphi) = \mathbb{E}_{s_t \sim \rho_\theta, a_t \sim \pi_\theta}[(R^n_t - V_\varphi(s_t))^2] \end{cases}$$

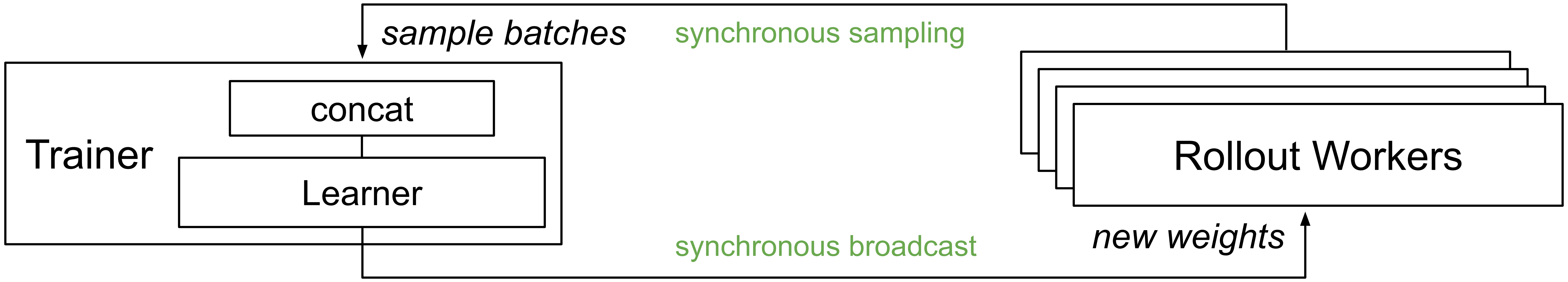
- We cannot use an **experience replay memory** to deal with the correlated inputs, as it is only for off-policy methods.

Distributed RL

- We cannot get an uncorrelated batch of transitions by acting **sequentially** with a single agent.



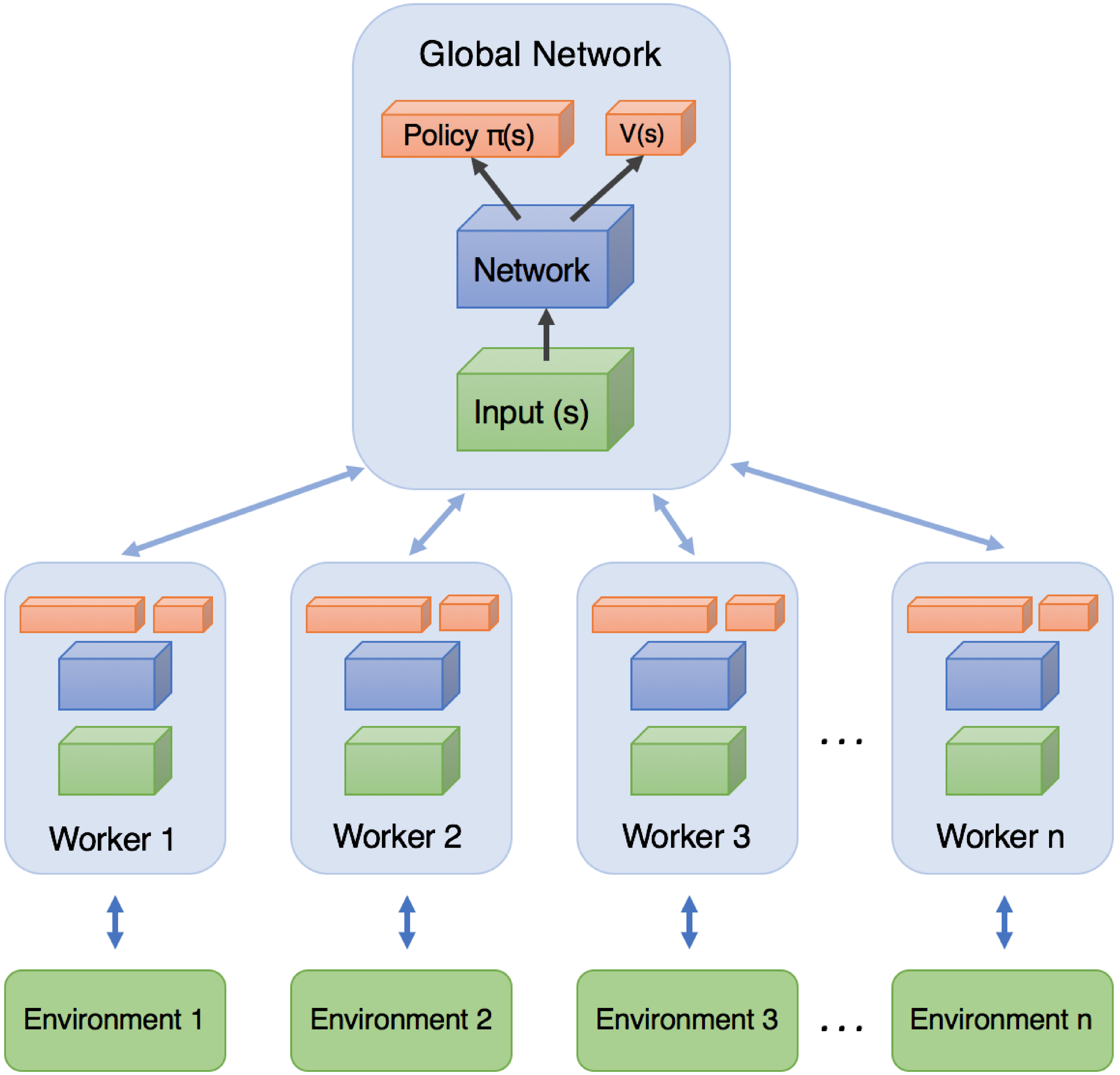
- A simple solution is to have **multiple actors** with the same weights θ interacting **in parallel** with different copies of the environment.



Source: https://ray.readthedocs.io/en/latest/rllib.html

- Each **rollout worker** (actor) starts an episode in a different state: at any point of time, the workers will be in **uncorrelated states**.
- From time to time, the workers all send their experienced transitions to the **learner** which updates the policy using a **batch of uncorrelated transitions**.
- After the update, the workers use the new policy.

Distributed RL



Distributed RL

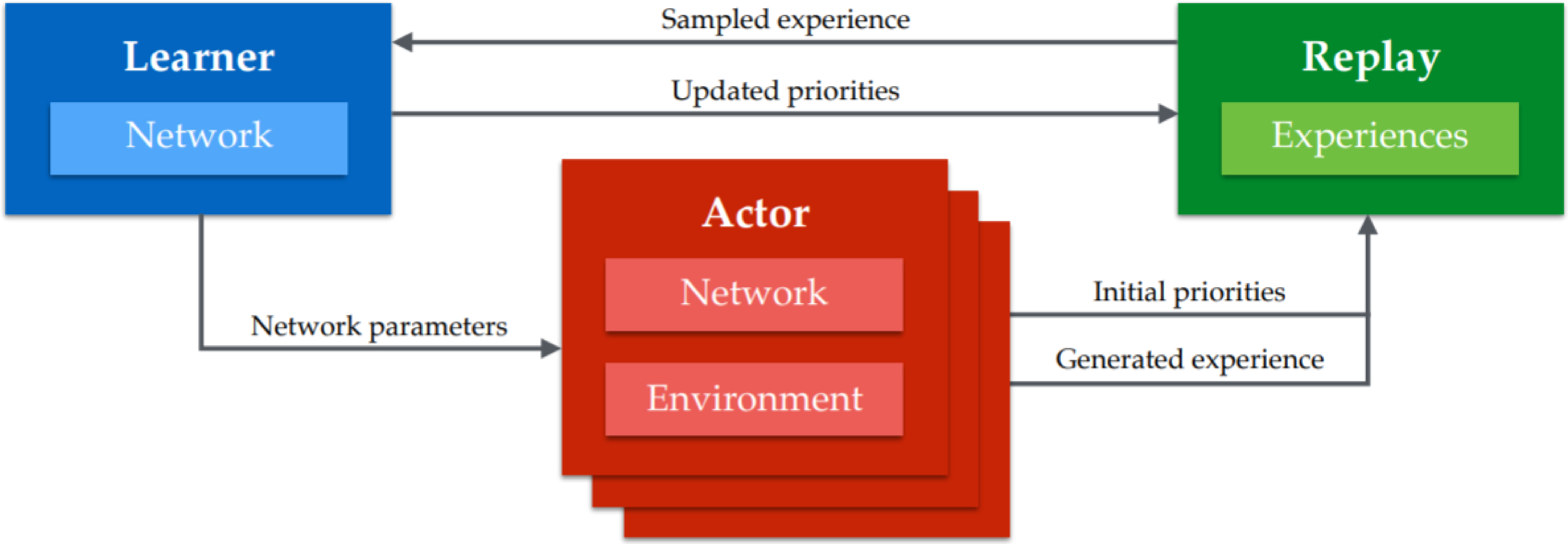
- Initialize global policy or value network θ.
- Initialize N copies of the environment in different states.
- **while** True:
    - **for** each worker in parallel:
        - Copy the global network parameters θ to each worker:

          $$\theta_k \leftarrow \theta$$

        - Initialize an empty transition buffer $\mathcal{D}_k$.
        - Perform d steps with the worker on its copy of the environment.
        - Append each transition (s, a, r, s') to the transition buffer.
    - join(): wait for each worker to terminate.
    - Gather the N transition buffers into a single buffer $\mathcal{D}$.
    - Update the global network on $\mathcal{D}$ to obtain new weights θ.

Distributed RL for value-based networks (DQN variants)

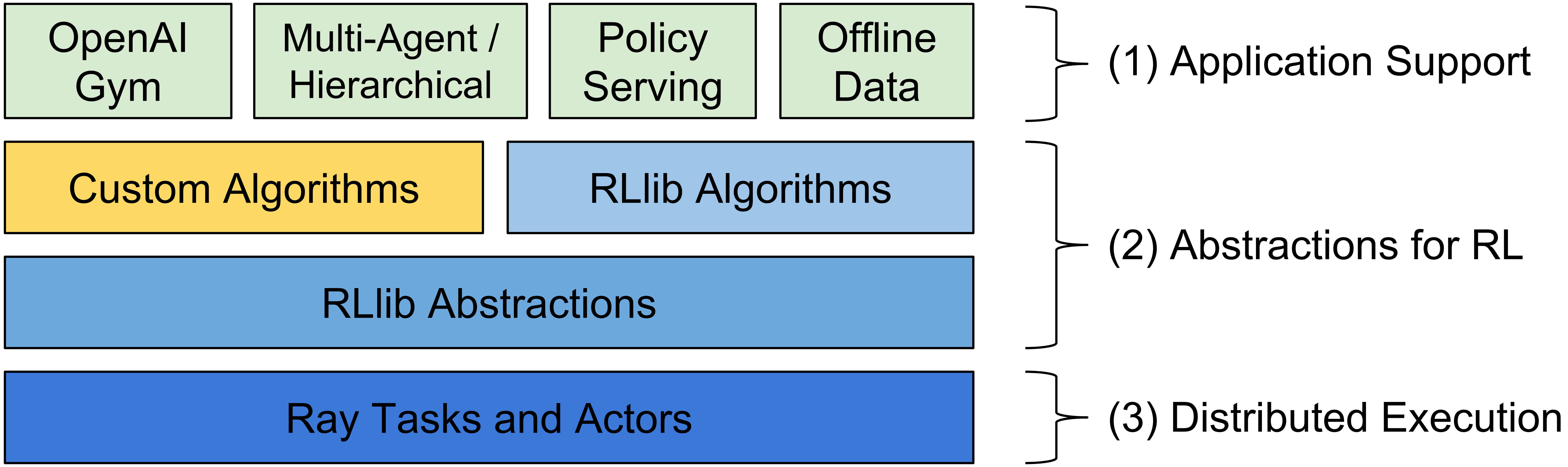
- Distributed learning can be used for any deep RL algorithm, including DQN variants.
- Distributed DQN variants include GORILA, IMPALA, APE-X, R2D2.
- “All” you need is one (or more) GPU for training the global network and N CPU cores for the workers.
- The workers fill the ERM much more quickly.



Horgan et al. (2018) Distributed Prioritized Experience Replay. arXiv:1803.00933.

Distributed RL

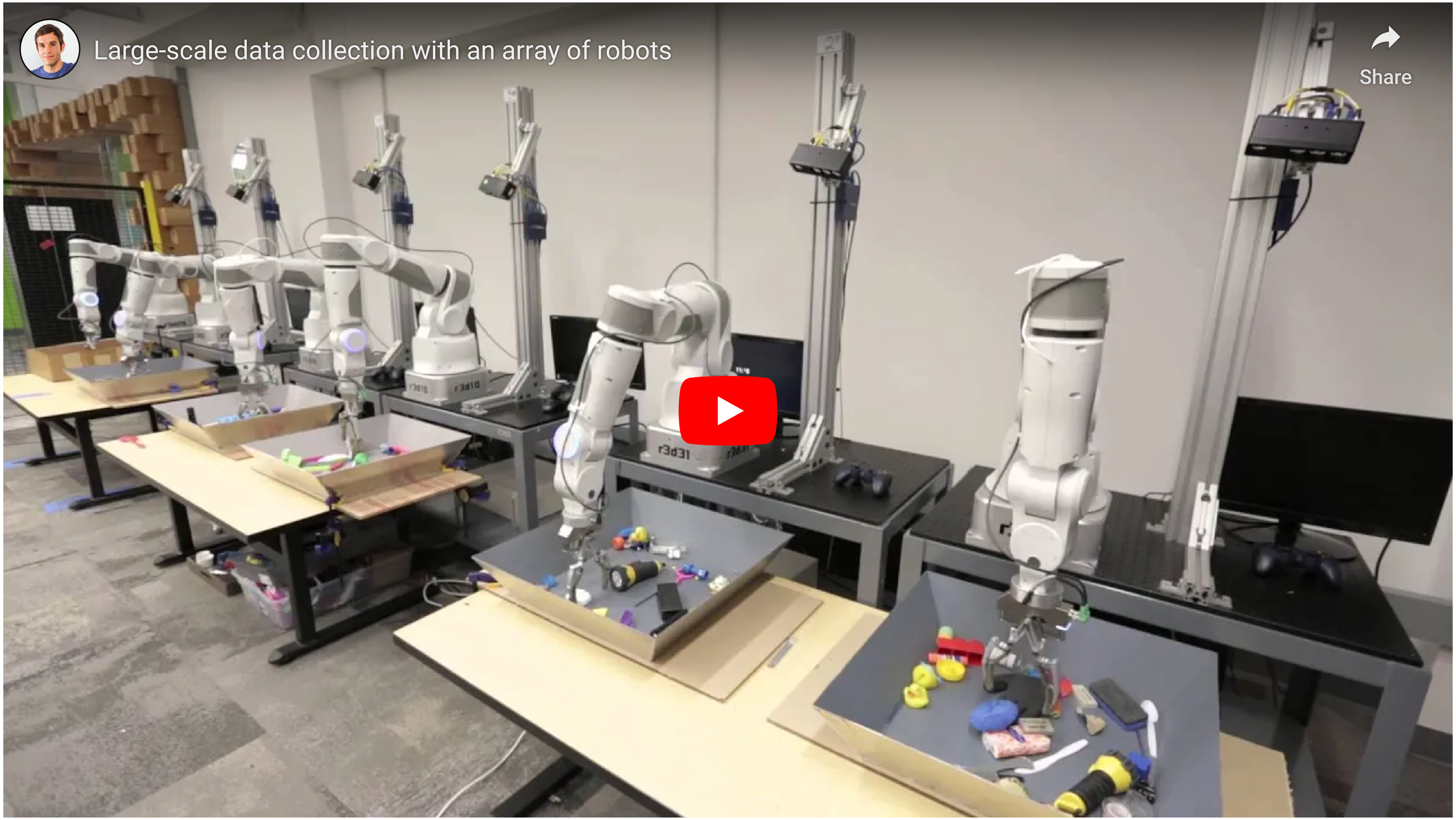
- In practice, managing the communication between the workers and the global network through processes can be quite painful.
- There are some **frameworks** abstracting the dirty work, such as **RLlib**.



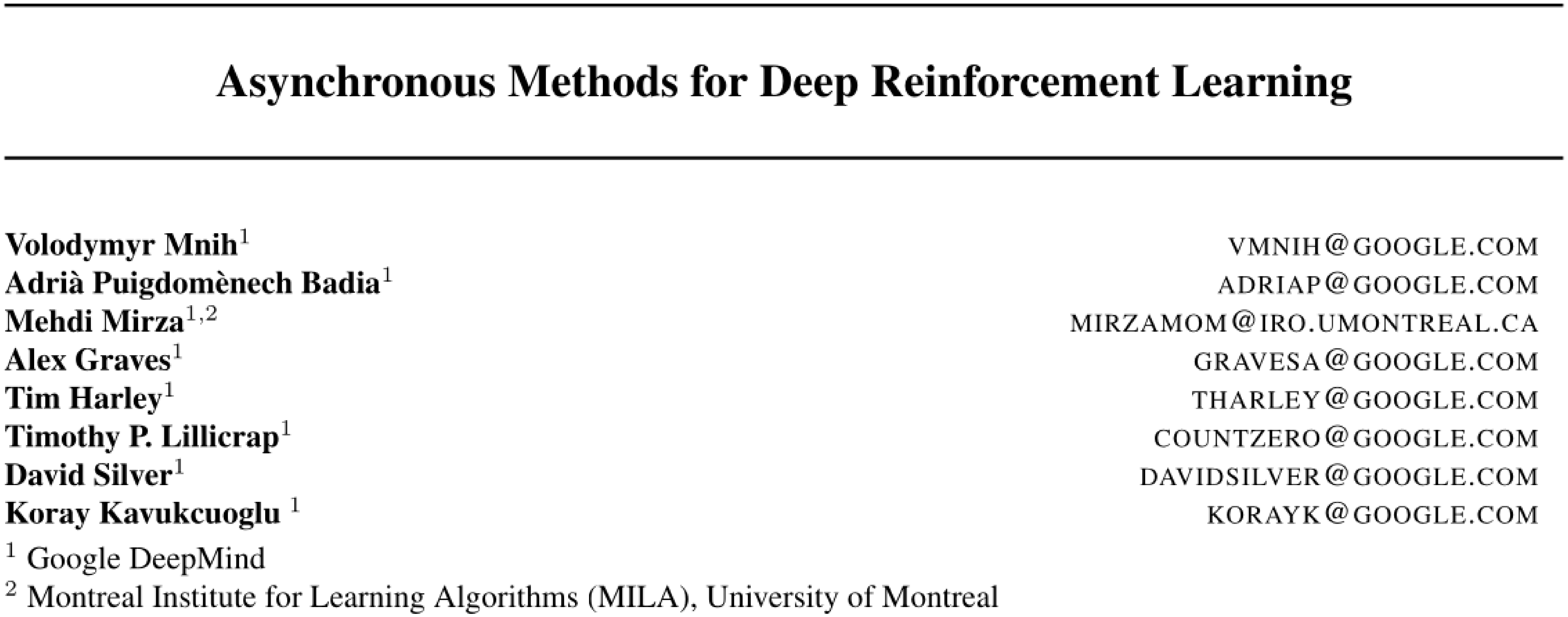
Source: https://ray.readthedocs.io/en/latest/rllib.html

Distributed RL

- Having multiple workers interacting with different environments is easy in simulation (Atari games).
- With physical environments, working in real time, it requires lots of money...



2 - A3C: Asynchronous advantage actor-critic



Asynchronous Methods for Deep Reinforcement Learning

Volodymyr Mnih[1]	VMNIH@GOOGLE.COM
Adrià Puigdomènech Badia[1]	ADRIAP@GOOGLE.COM
Mehdi Mirza[1,2]	MIRZAMOM@IRO.UMONTREAL.CA
Alex Graves[1]	GRAVESA@GOOGLE.COM
Tim Harley[1]	THARLEY@GOOGLE.COM
Timothy P. Lillicrap[1]	COUNTZERO@GOOGLE.COM
David Silver[1]	DAVIDSILVER@GOOGLE.COM
Koray Kavukcuoglu [1]	KORAYK@GOOGLE.COM

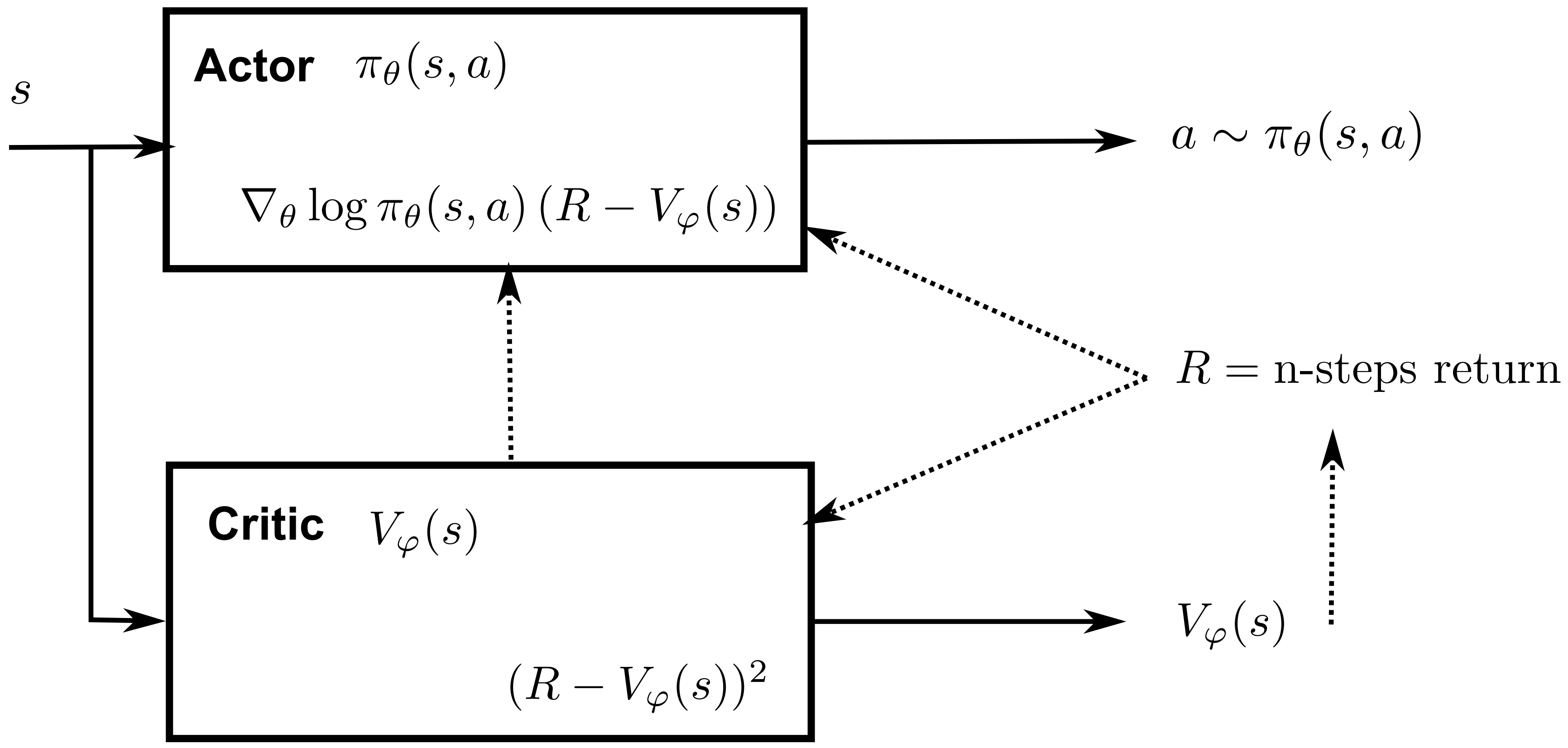
[1] Google DeepMind
[2] Montreal Institute for Learning Algorithms (MILA), University of Montreal

Mnih et al. (2016) Asynchronous Methods for Deep Reinforcement Learning. ICML. arXiv:1602.01783

A3C: Asynchronous advantage actor-critic

- Mnih et al. (2016) proposed the **A3C** algorithm (asynchronous advantage actor-critic).
- The stochastic policy π_θ is produced by the **actor** with weights θ and learned using :

$$\nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{s_t \sim \rho_\theta, a_t \sim \pi_\theta}[\nabla_\theta \log \pi_\theta (s_t, a_t) \, (R_t^n - V_\varphi(s_t))]$$



- The value of a state $V_\varphi(s)$ is produced by the **critic** with weights φ, which minimizes the mse with the **n-step return**:

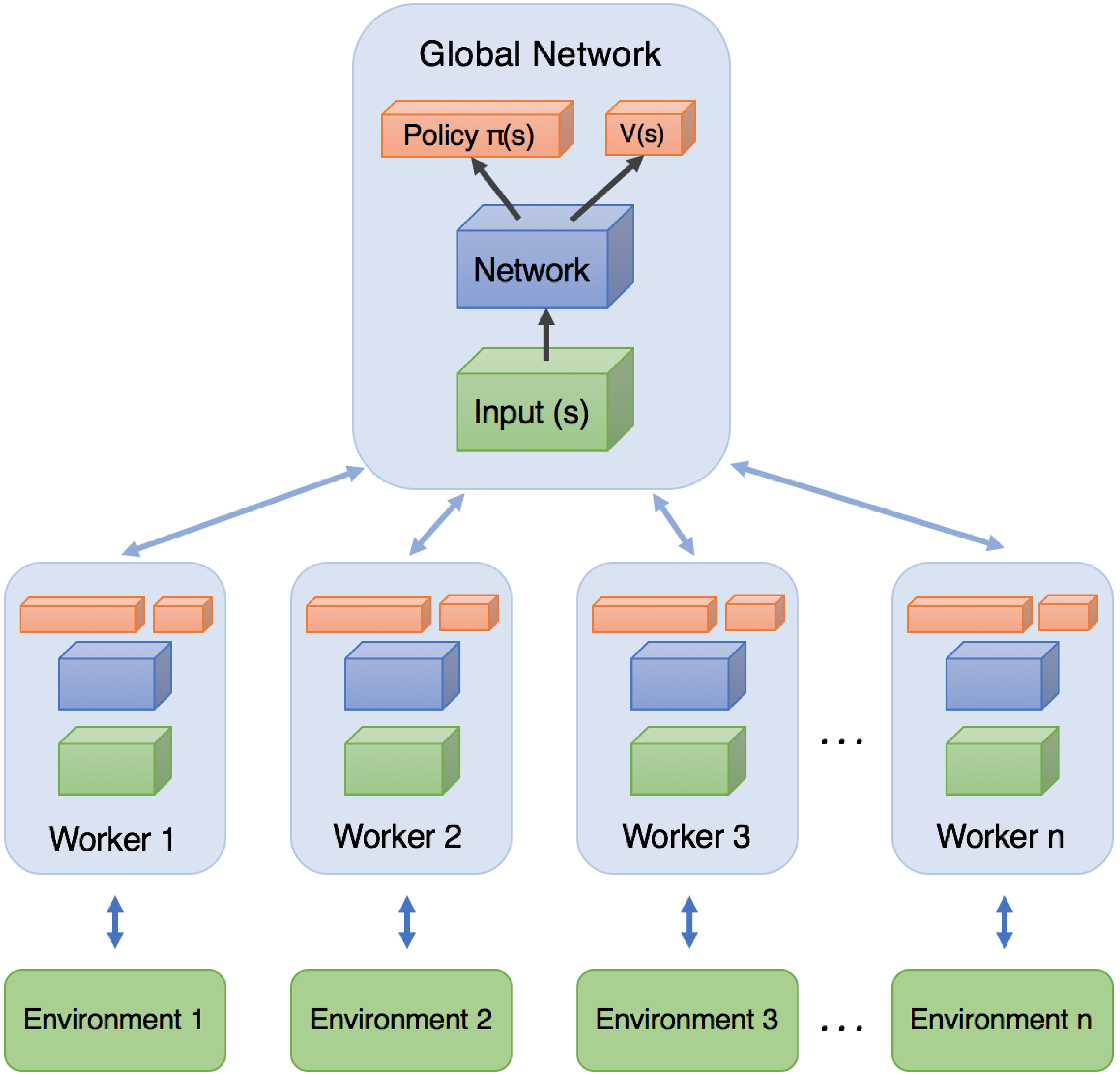
$$\mathcal{L}(\varphi) = \mathbb{E}_{s_t \sim \rho_\theta, a_t \sim \pi_\theta}[(R^n_t - V_\varphi(s_t))^2]$$

$$R^n_t = \sum_{k=0}^{n-1} \gamma^{k} \, r_{t+k+1} + \gamma^n \, V_\varphi(s_{t+n})$$

- Both the actor and the critic are trained on batches of transitions collected using **parallel workers**.
- Two things are different from the general distributed approach: workers compute **partial gradients** and updates are **asynchronous**.

Mnih et al. (2016) Asynchronous Methods for Deep Reinforcement Learning. ICML. arXiv:1602.01783

A3C: Asynchronous advantage actor-critic



- **def** worker(θ, φ):
    - Initialize empty transition buffer $\mathcal{D}$. Initialize the environment to the **last** state visited by this worker.
    - **for** n steps:
        - Select an action using π_θ, store the transition in the transition buffer.
    - **for** each transition in $\mathcal{D}$:
        - Compute the **n-step return** in each state $R_t^n = \displaystyle\sum_{k=0}^{n-1} \gamma^k \, r_{t+k+1} + \gamma^n \, V_\varphi(s_{t+n})$
    - Compute **policy gradient** for the actor on the transition buffer:

$$d\theta = \nabla_\theta \mathcal{J}(\theta) = \frac{1}{n} \sum_{t=1}^n \nabla_\theta \log \pi_\theta(s_t, a_t) \, (R_t^n - V_\varphi(s_t))$$

    - Compute **value gradient** for the critic on the transition buffer:

$$d\varphi = \nabla_\varphi \mathcal{L}(\varphi) = -\frac{1}{n} \sum_{t=1}^n (R_t^n - V_\varphi(s_t)) \, \nabla_\varphi V_\varphi(s_t)$$

    - **return** $d\theta$, $d\varphi$

A2C: global networks

- Initialize actor θ and critic φ.
- Initialize K workers with a copy of the environment.
- **for** $t \in [0, T_\text{total}]$:
    - **for** K workers **in parallel**:
        - $d\theta_k, d\varphi_k$ = worker(θ, φ)
    - join()
    - Merge all gradients:

$$d\theta = \frac{1}{K} \sum_{i=1}^K d\theta_k \; ; \; d\varphi = \frac{1}{K} \sum_{i=1}^K d\varphi_k$$

    - Update the actor and critic using gradient ascent/descent:

$$\theta \leftarrow \theta + \eta \, d\theta \; ; \; \varphi \leftarrow \varphi - \eta \, d\varphi$$

A3C: Asynchronous advantage actor-critic

- The previous slide depicts **A2C**, the synchronous version of A3C.
- A2C synchronizes the workers (threads), i.e. it waits for the K workers to finish their job before merging the gradients and updating the global networks.
- A3C is **asynchronous**:
  - the partial gradients are applied to the global networks **as soon as** they are available.
  - No need to wait for all workers to finish their job.
- As the workers are not synchronized, this means that one worker could be copying the global networks θ and φ **while** another worker is writing them.
- This is called a **Hogwild!** update: no locks, no semaphores. Many workers can read/write the same data.
- It turns out NN are robust enough for this kind of updates.

A3C: asynchronous updates

- Initialize actor θ and critic φ.
- Initialize K workers with a copy of the environment.
- **for** K workers **in parallel**:
    - **for** $t \in [0, T_\text{total}]$:
        - Copy the global networks θ and φ.
        - Compute partial gradients:

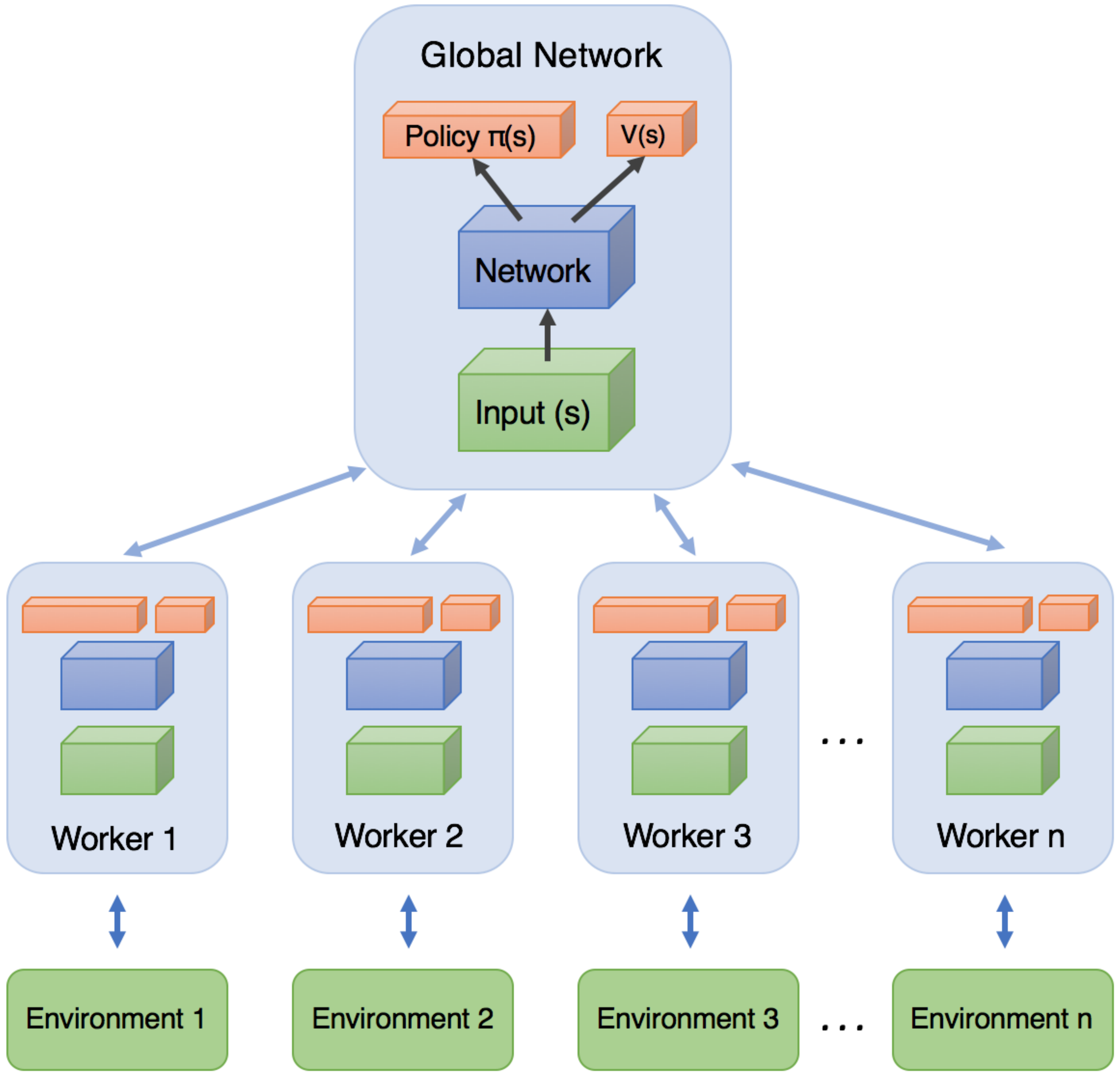
$$d\theta_k, d\varphi_k = \text{worker}(\theta, \varphi)$$

        - Update the **global** actor and critic using the **partial gradients**:

$$\theta \leftarrow \theta + \eta \, d\theta_k$$

$$\varphi \leftarrow \varphi - \eta \, d\varphi_k$$

A3C: Asynchronous advantage actor-critic



- A3C does not use an *experience replay memory*, but relies on **multiple parallel workers** to distribute learning.
- Each worker has a copy of the actor and critic networks, as well as an instance of the environment.
- Weight updates are synchronized regularly though a **master network** using Hogwild!-style updates (every $n = 5$ steps!).
- Because the workers learn different parts of the state-action space, the weight updates are not very correlated.

- It works best on shared-memory systems (multi-core) as communication costs between GPUs are huge.
- As an actor-critic method, it can deal with **continuous action spaces**.

Mnih et al. (2016) Asynchronous Methods for Deep Reinforcement Learning. ICML. arXiv:1602.01783

A3C : results

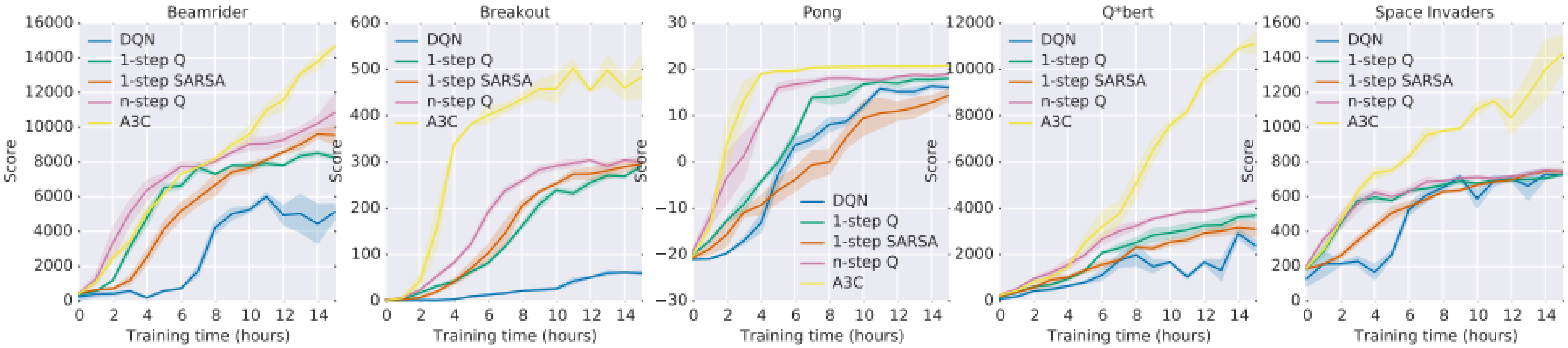
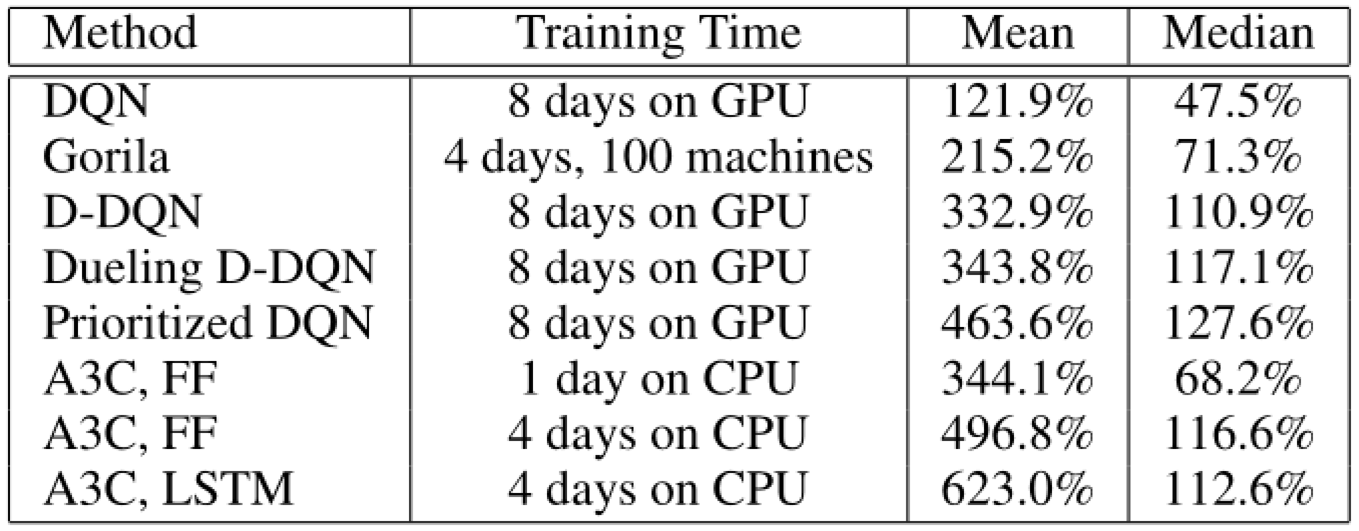


Figure 1. Learning speed comparison for DQN and the new asynchronous algorithms on five Atari 2600 games. DQN was trained on a single Nvidia K40 GPU while the asynchronous methods were trained using 16 CPU cores. The plots are averaged over 5 runs. In the case of DQN the runs were for different seeds with fixed hyperparameters. For asynchronous methods we average over the best 5 models from 50 experiments with learning rates sampled from $LogUniform(10^{-4}, 10^{-2})$ and all other hyperparameters fixed.



Method	Training Time	Mean	Median
DQN	8 days on GPU	121.9%	47.5%
Gorila	4 days, 100 machines	215.2%	71.3%
D-DQN	8 days on GPU	332.9%	110.9%
Dueling D-DQN	8 days on GPU	343.8%	117.1%
Prioritized DQN	8 days on GPU	463.6%	127.6%
A3C, FF	1 day on CPU	344.1%	68.2%
A3C, FF	4 days on CPU	496.8%	116.6%
A3C, LSTM	4 days on CPU	623.0%	112.6%

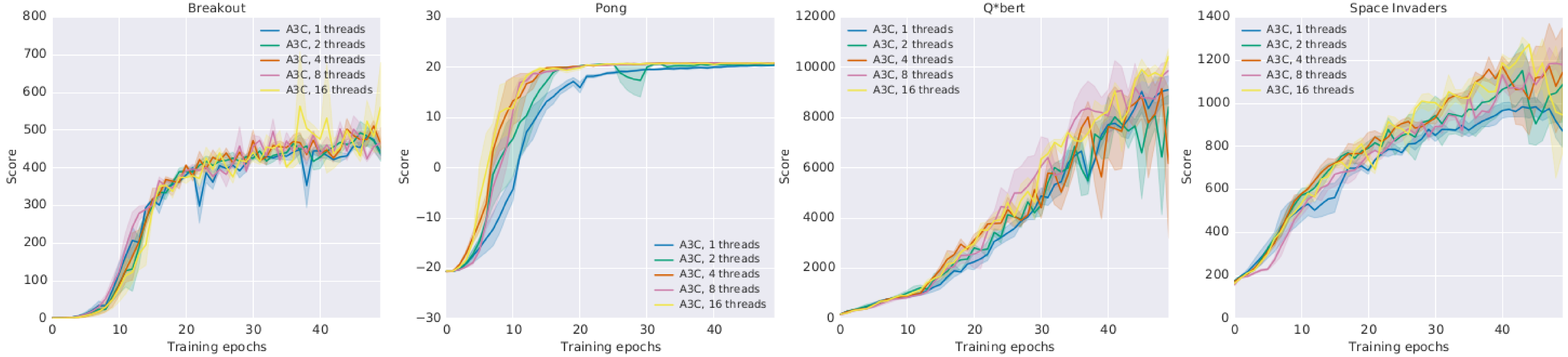
Table 1. Mean and median human-normalized scores on 57 Atari games using the human starts evaluation metric. Supplementary Table SS3 shows the raw scores for all games.

- A3C set a new record for Atari games in 2016.
- The main advantage is that the workers gather experience in parallel: training is much faster than with DQN.
- LSTMs can be used to improve the performance.

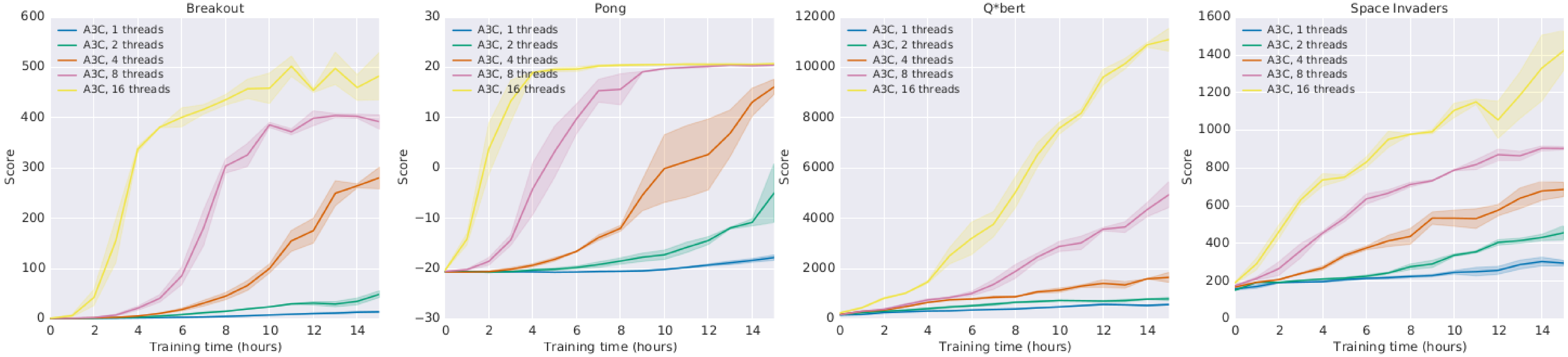
Mnih et al. (2016) Asynchronous Methods for Deep Reinforcement Learning. ICML. arXiv:1602.01783

A3C : results

- Learning is only marginally better with more threads:



but much faster!



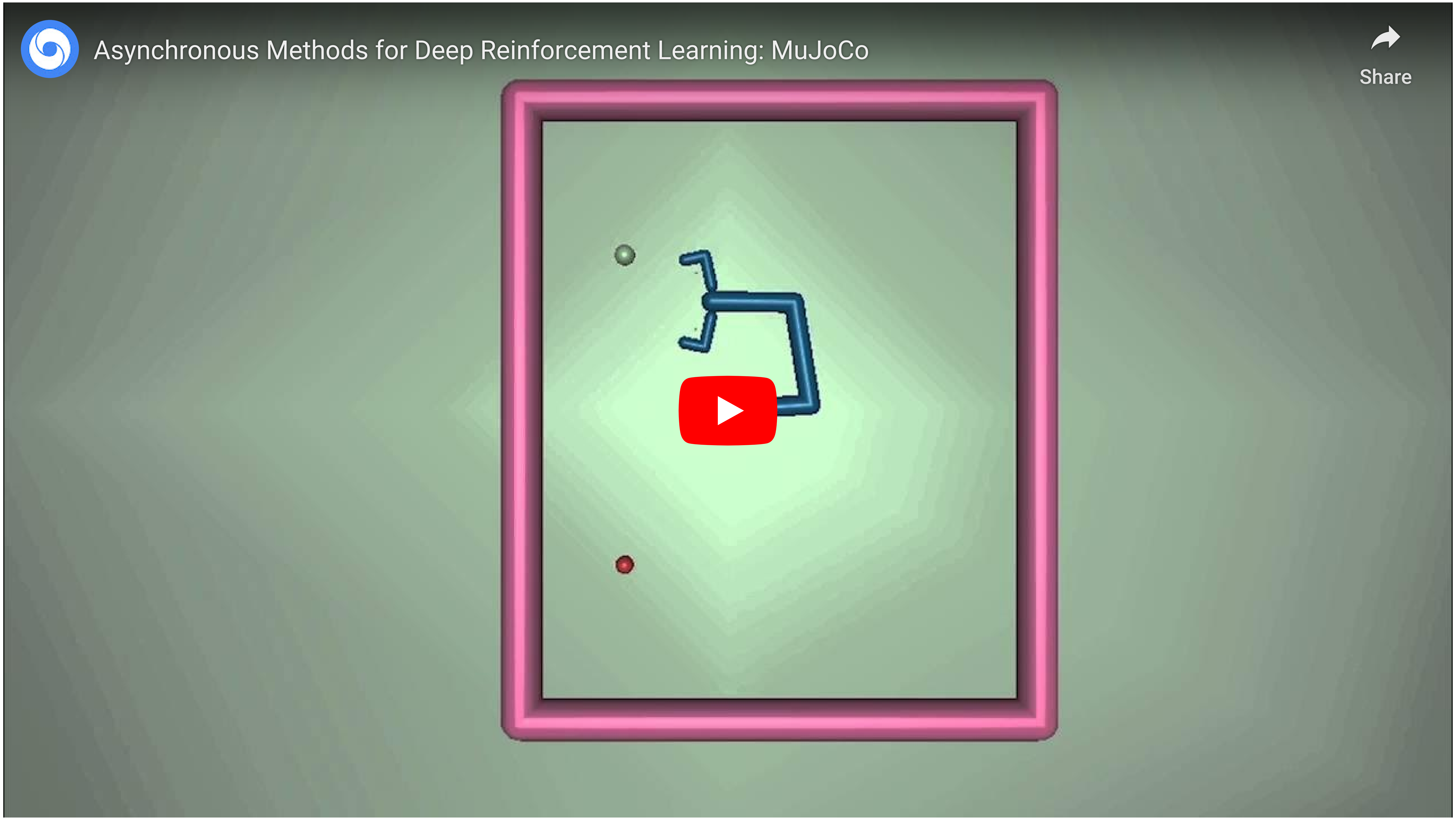
Mnih et al. (2016) Asynchronous Methods for Deep Reinforcement Learning. ICML. arXiv:1602.01783

A3C: TORCS simulator

A3C: Labyrinth

Asynchronous Methods for Deep Reinforcement Learning: Labyrinth

A3C: continuous control problems


Asynchronous Methods for Deep Reinforcement Learning: MuJoCo
Share

Comparison with DQN

- A3C came up in 2016. A lot of things happened since then…

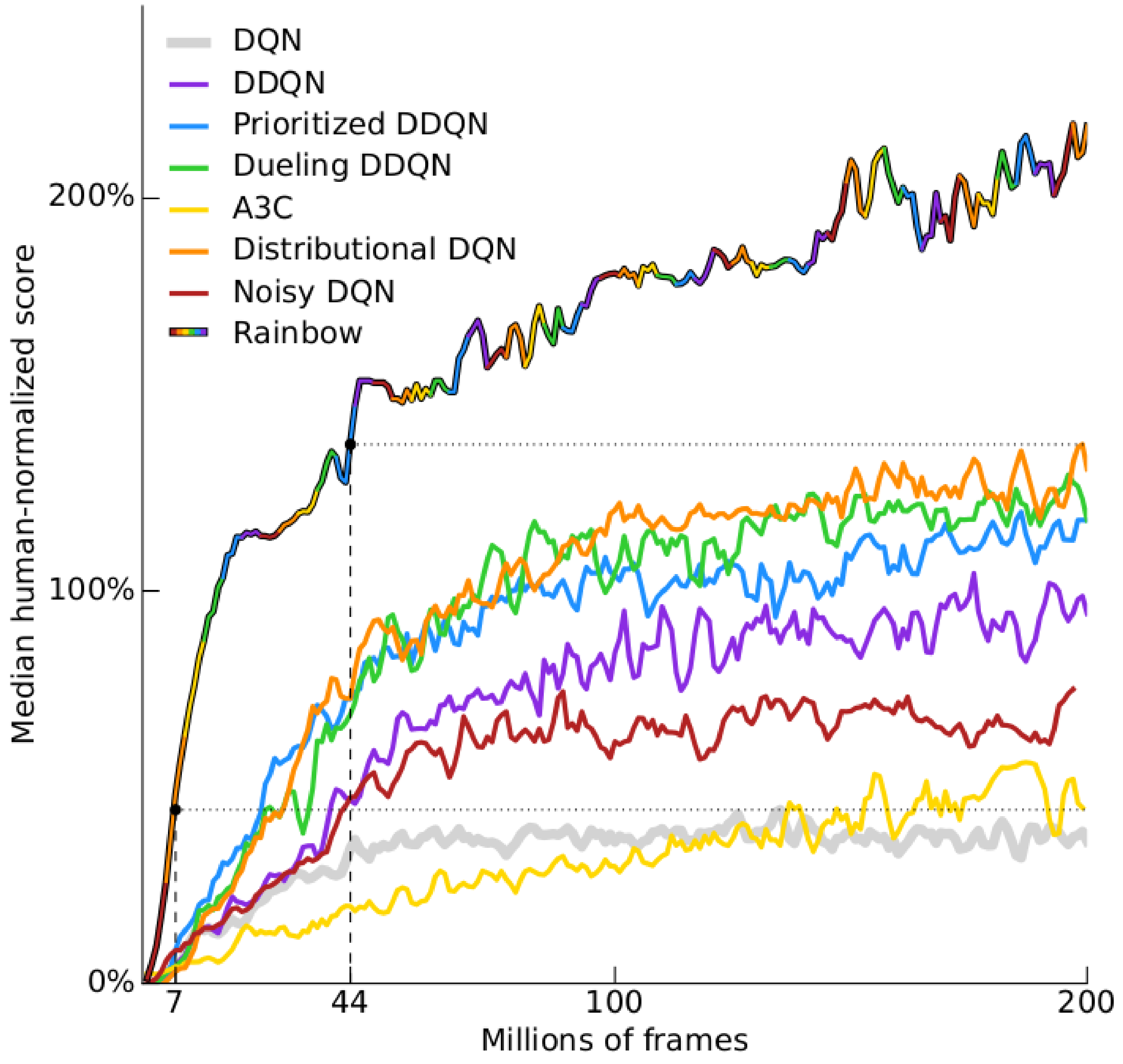


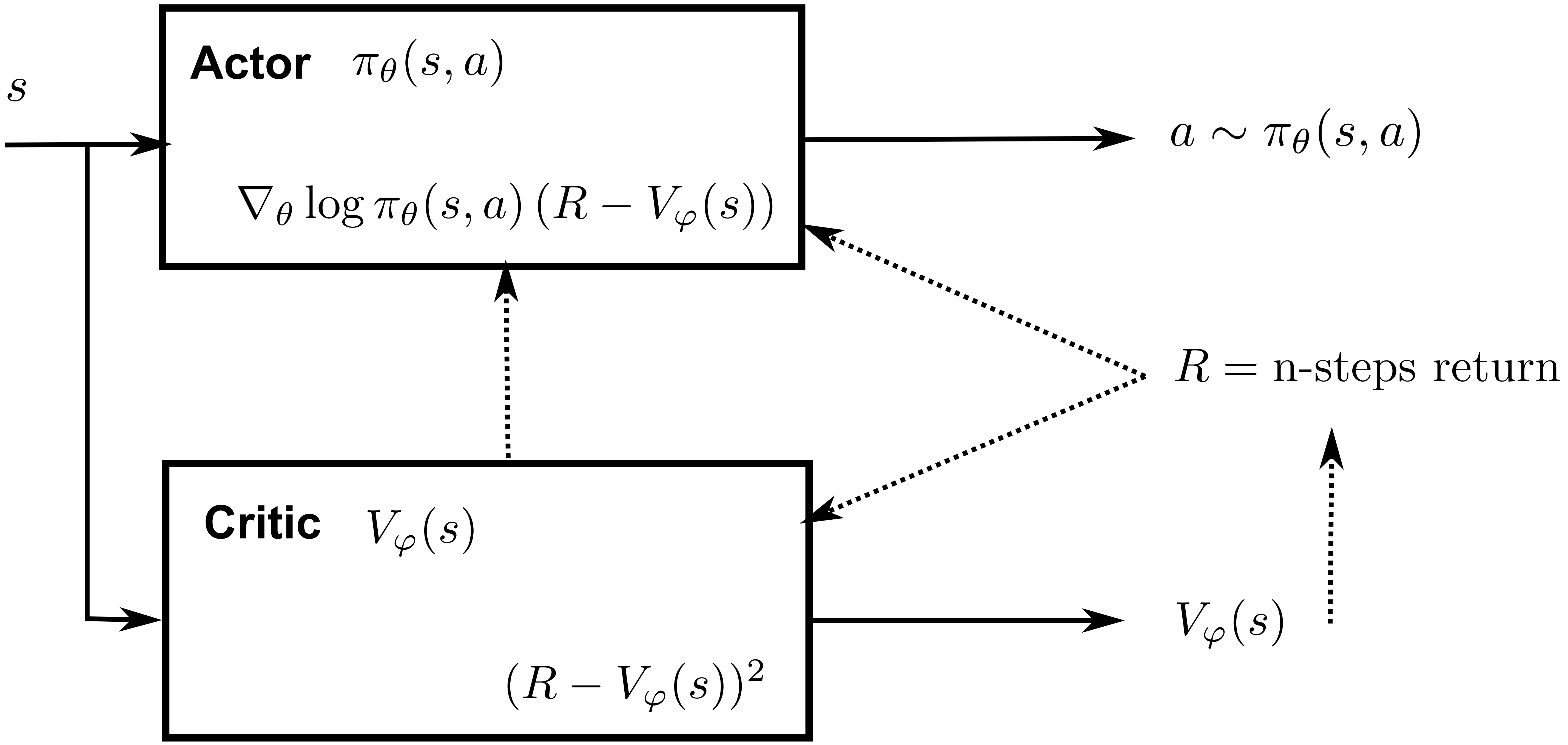
Figure 1: **Median human-normalized performance** across 57 Atari games. We compare our integrated agent (rainbow-colored) to DQN (grey) and six published baselines. Note that we match DQN's best performance after 7M frames, surpass any baseline within 44M frames, and reach substantially improved final performance. Curves are smoothed with a moving average over 5 points.

Hessel et al. (2017) Rainbow: Combining Improvements in Deep Reinforcement Learning. arXiv:1710.02298

3 - Actor-critic neural architectures

Actor-critic neural architectures

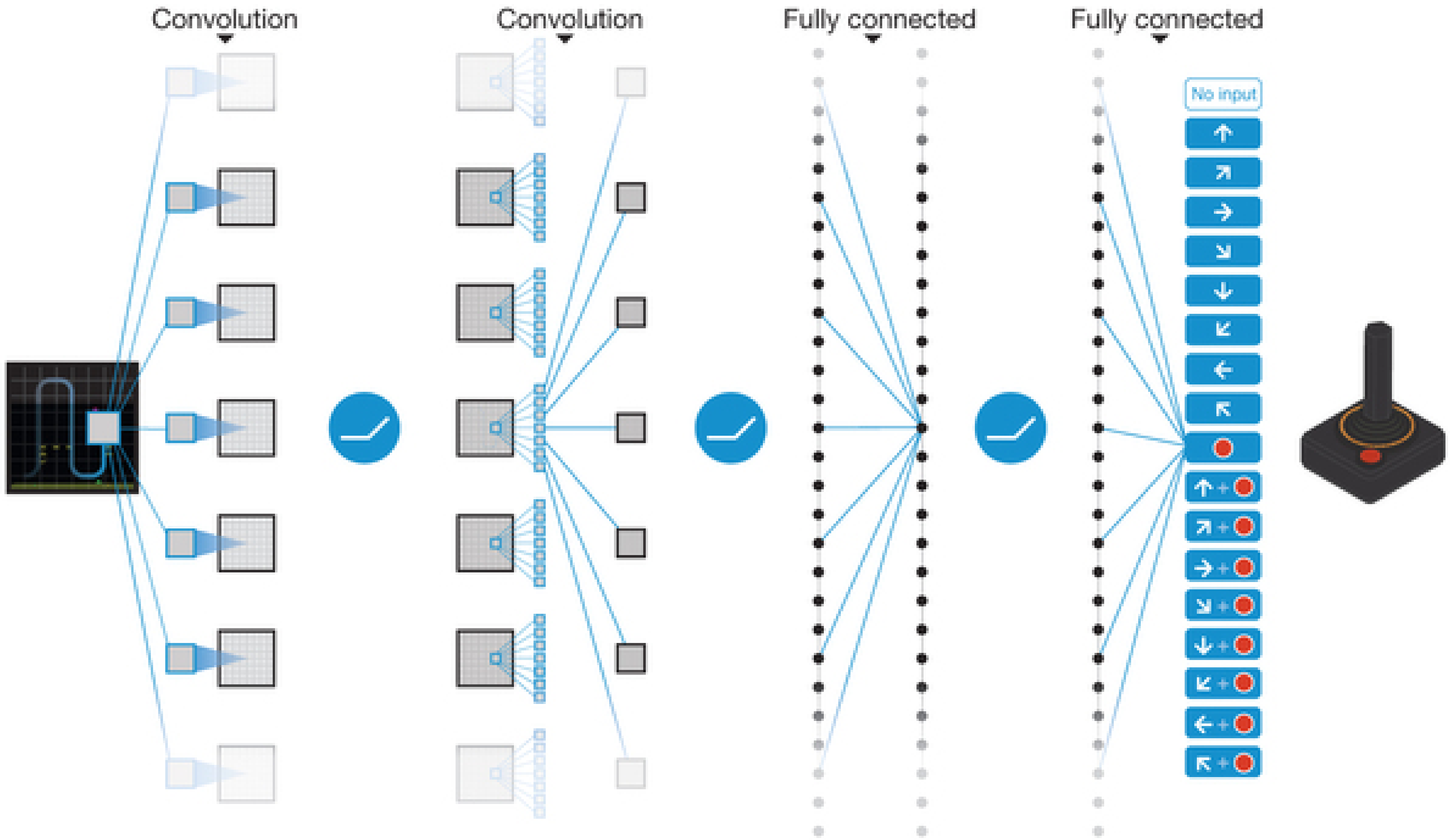
- We have considered that actor-critic architectures consist of two separate neural networks, both taking the state s (or observation o) as an input.


s
Actor $\pi_\theta(s, a)$
$\nabla_\theta \log \pi_\theta(s, a) \, (R - V_\varphi(s))$
$a \sim \pi_\theta(s, a)$
$R = \text{n-steps return}$
Critic $V_\varphi(s)$
$(R - V_\varphi(s))^2$
$V_\varphi(s)$

- Each of these networks have their own loss function. They share nothing except the “data”.
- Is it really the best option?

Early visual features

- When working on images, the first few layers of the CNNs are likely to learn the same visual features (edges, contours).
- It would be more efficient to **share** some of the extracted features.



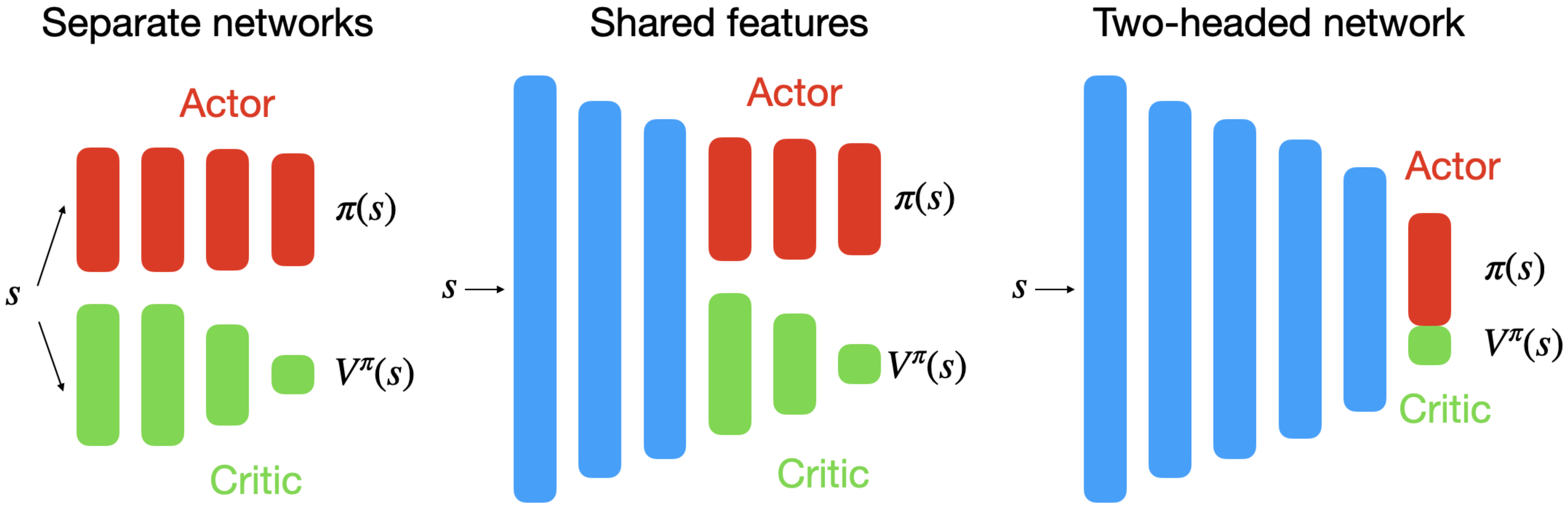
Mnih et al. (2013) Playing Atari with Deep Reinforcement Learning. http://arxiv.org/abs/1312.5602

Shared architectures

- Actor-critic architectures can share layers between the actor and the critic, sometimes up to the output layer.
- A compound loss sums the losses for the actor and the critic. Tensorflow/pytorch know which parameters influence which part of the loss.

$$\mathcal{L}(\theta) = \mathcal{L}_\text{actor}(\theta) + \mathcal{L}_\text{critic}(\theta)$$

- For pixel-based environments (Atari), the networks often share the convolutional layers.
- For continuous environments (Mujoco), separate networks sometimes work better than two-headed networks.



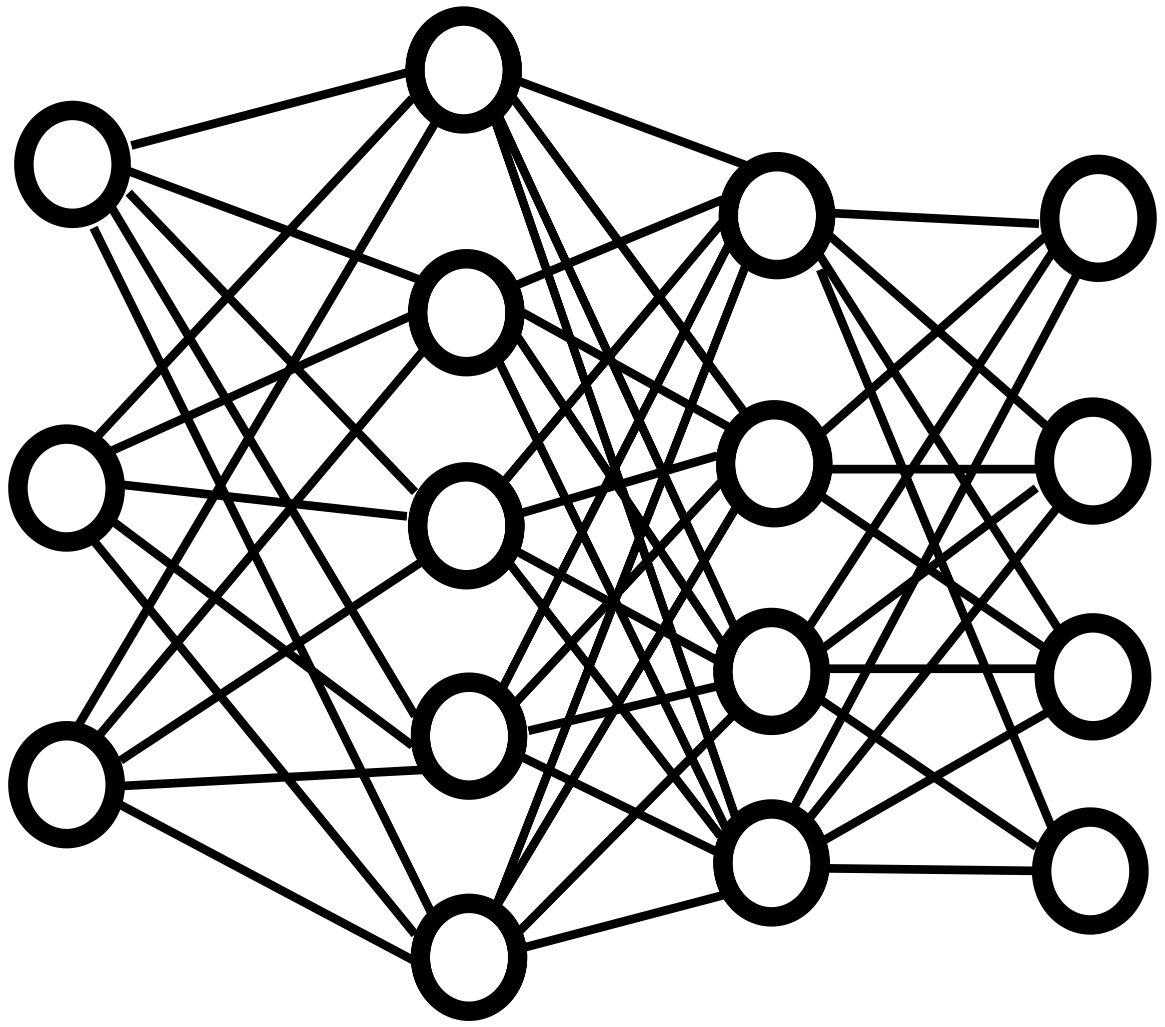
4 - Continuous stochastic policies

Continuous action spaces

- One of the main advantages of actor-critic / PG methods over value-based methods is that they can deal with **continuous action-spaces**.
- Suppose that we want to control a robotic arm with n degrees of freedom.
- An action $\mathbf{a}$ could be a vector of joint displacements:

$$\mathbf{a} = \begin{bmatrix} \Delta \theta_1 & \Delta \theta_2 & \ldots \, \Delta \theta_n \end{bmatrix}^T$$

- The output layer of the policy network can very well represent this vector, but how would we implement **exploration**?
- ϵ-greedy and softmax action selection would not work, as all neurons are useful.
- The most common solution is to use a stochastic **Gaussian policy**.

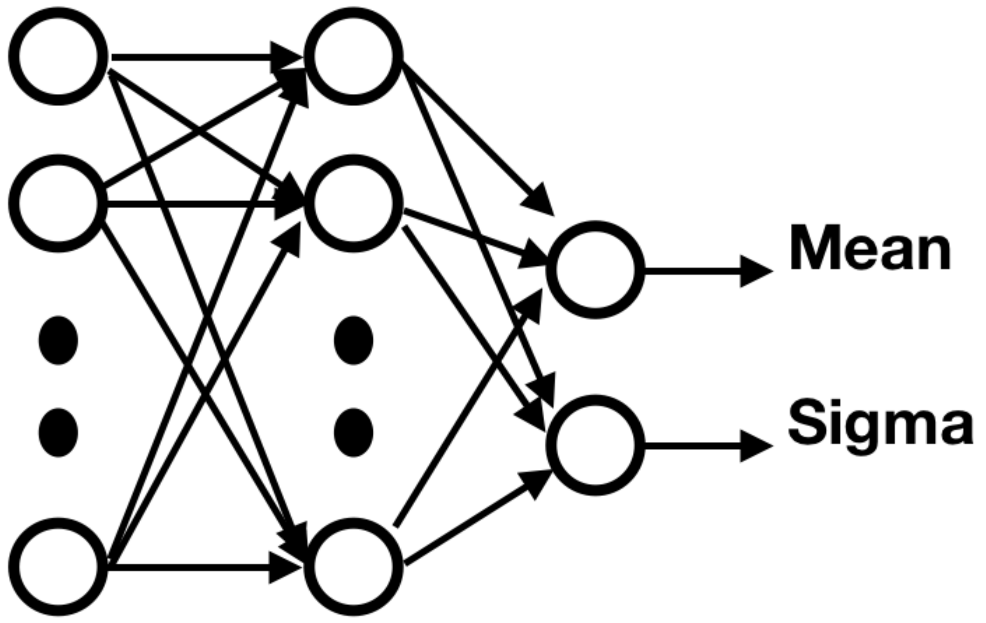


Gaussian policies

- A Gaussian policy considers the vector $\mathbf{a}$ to be sampled from the **normal distribution** $\mathcal{N}(\mu_\theta(s), \sigma_\theta(s))$.
- The mean $\mu_\theta(s)$ and standard deviation $\sigma_\theta(s)$ are vectors that can be the output of the **actor** neural network with parameters θ.
- If you have n degrees of freedom, you just need $2n$ output neurons.
- **Sampling** an action from the normal distribution is done through the **reparameterization trick**:

$$\mathbf{a} = \mu_\theta(s) + \sigma_\theta(s) \, \xi$$

where $\xi \sim \mathcal{N}(0, I)$ comes from the standard normal distribution.



https://medium.com/@vittoriolabarbera/continuous-control-with-a2c-and-gaussian-policies-mujoco-pytorch-and-c-4221ec8ba024

Gaussian policies

- The good thing with the normal distribution is that we know its pdf:

$$\pi_\theta(s, a) = \frac{1}{\sqrt{2\pi\sigma^2_\theta(s)}} \, \exp -\frac{(a - \mu_\theta(s))^2}{2\sigma^2_\theta(s)}$$

- The log-likelihood $\log \pi_\theta(s, a)$ is a simple differentiable function of $\mu_\theta(s)$ and $\sigma_\theta(s)$:

$$\log \pi_\theta(s, a) = -\frac{(a - \mu_\theta(s))^2}{2\sigma^2_\theta(s)} - \frac{1}{2} \, \log 2\pi\sigma^2_\theta(s)$$

so we can easily compute its gradient w.r.t θ and apply backpropagation:

$$\begin{cases} \nabla_{\mu_\theta(s)} \log \pi_\theta(s, a) = \dfrac{a - \mu_\theta(s)}{\sigma_\theta(s)^2} \\ \\ \nabla_{\sigma_\theta(s)} \log \pi_\theta(s, a) = \dfrac{(a - \mu_\theta(s))^2}{\sigma_\theta(s)^3} - \dfrac{1}{\sigma_\theta(s)} \end{cases}$$

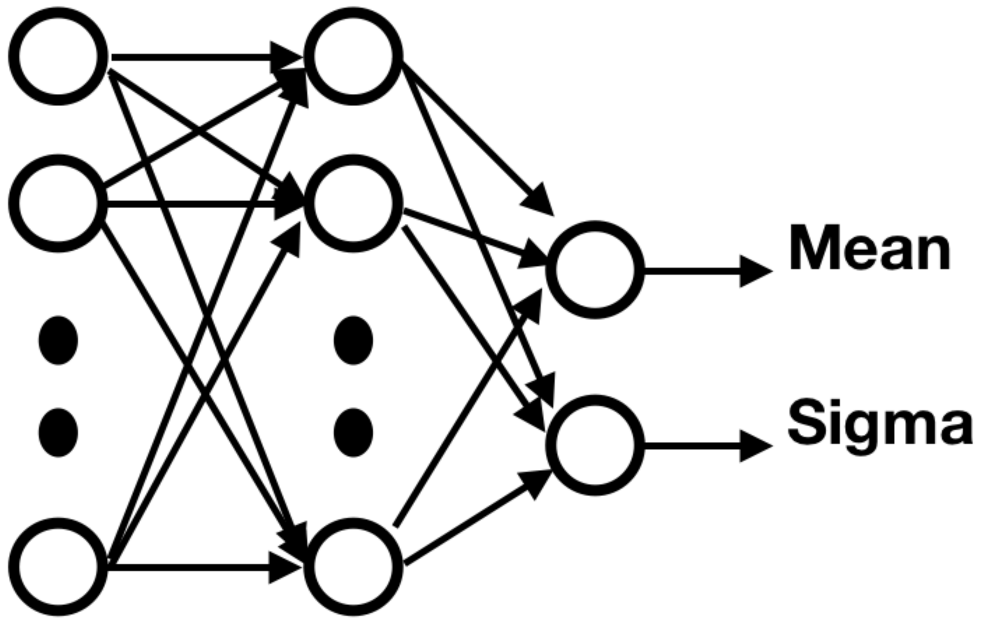
Gaussian policies

- A Gaussian policy samples actions from the **normal distribution** $\mathcal{N}(\mu_\theta(s), \sigma_\theta(s))$, with $\mu_\theta(s)$ and $\sigma_\theta(s)$ being the output of the actor.

- The score $\nabla_\theta \log \pi_\theta(s, a)$ can be obtained easily using the output of the actor:

$$\mathbf{a} = \mu_\theta(s) + \sigma_\theta(s)\, \xi$$

$$\begin{cases} \nabla_{\mu_\theta(s)} \log \pi_\theta(s, a) = \dfrac{a - \mu_\theta(s)}{\sigma_\theta(s)^2} \\ \\ \nabla_{\sigma_\theta(s)} \log \pi_\theta(s, a) = \dfrac{(a - \mu_\theta(s))^2}{\sigma_\theta(s)^3} - \dfrac{1}{\sigma_\theta(s)} \end{cases}$$



- The rest of the score ($\nabla_\theta \mu_\theta(s)$ and $\nabla_\theta \sigma_\theta(s)$) is the problem of tensorflow/pytorch.

Source: https://medium.com/@vittoriolabarbera/continuous-control-with-a2c-and-gaussian-policies-mujoco-pytorch-and-c-4221ec8ba024

- This is the same **reparametrization trick** used in variational autoencoders to allow backpropagation to work through a sampling operation.
- **Beta** distributions are an even better choice to parameterize stochastic policies (Chou et al., 2017).

Chou et al. (2017) Improving Stochastic Policy Gradients in Continuous Control with Deep Reinforcement Learning using the Beta Distribution. ICML.

References

Chou, P.-W., Maturana, D., and Scherer, S. (2017). Improving Stochastic Policy Gradients in Continuous Control with Deep Reinforcement Learning using the Beta Distribution. in *International Conference on Machine Learning* http://proceedings.mlr.press/v70/chou17a/chou17a.pdf.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., et al. (2017). Rainbow: Combining Improvements in Deep Reinforcement Learning. http://arxiv.org/abs/1710.02298.

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., et al. (2018). Distributed Prioritized Experience Replay. http://arxiv.org/abs/1803.00933.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., et al. (2016). Asynchronous Methods for Deep Reinforcement Learning. in *Proc. ICML* http://arxiv.org/abs/1602.01783.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., et al. (2013). Playing Atari with Deep Reinforcement Learning. http://arxiv.org/abs/1312.5602.