Carnegie Mellon

School of Computer Science

Deep Reinforcement Learning and Control

Evolutionary Methods

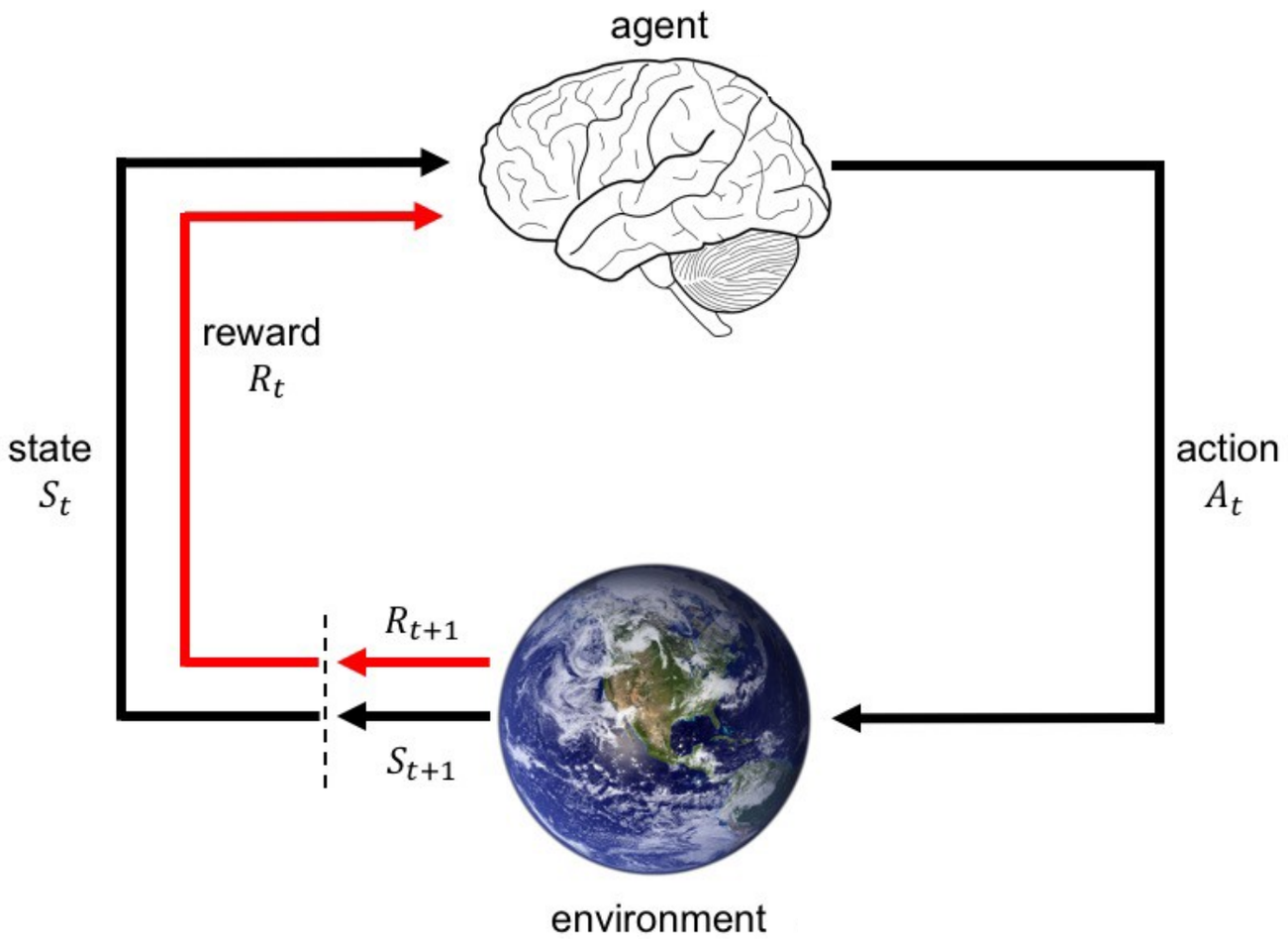
Fall 2019, CMU 10-703

Katerina Fragkiadaki

Part of the slides borrowed by Xi Chen, Pieter Abbeel, John Schulman

Reinforcement Learning

Learning behaviours from rewards while interacting with the environment



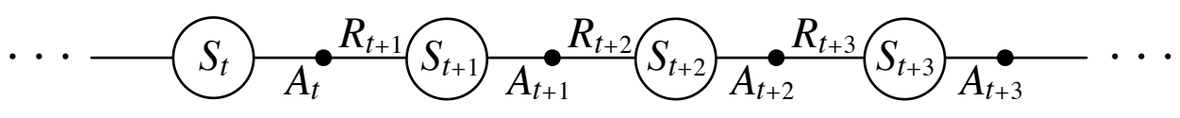
Agent and environment interact at discrete time steps: $t = 0, 1, 2, 3, \ldots$

Agent observes state at step t: $S_t \in \mathcal{S}$

produces action at step t : $A_t \in \mathcal{A}(S_t)$

gets resulting reward: $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$

and resulting next state: $S_{t+1} \in \mathcal{S}^+$



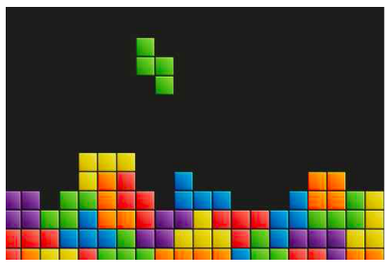
A concrete example: Playing Tetris

- states: the board configuration and the falling piece (lots of states ~ 2^200)
- actions: translations and rotations of the piece
- rewards: score of the game
- Our goal is to learn a policy that maximizes the the score of the game *in expectation.*
  - *Q:* what does in expectation mean?
  - A: we will play the game multiple times and the average score across those runs should be high.

- Imagine a very straightforward training process (search for best actions to take at each state) by playing the game for **multiple days** and simply asking an expert player what to do at each state we encounter, and learn using supervised learning.
- Q1: During the training period, could someone visit all states in Tertis, to figure out what is the best action

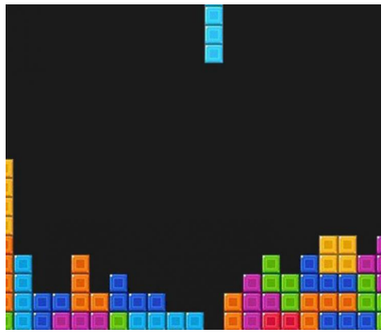
A concrete example: Playing Tetris

- Policy: a mapping from states to actions



Go right

Turn clockwise

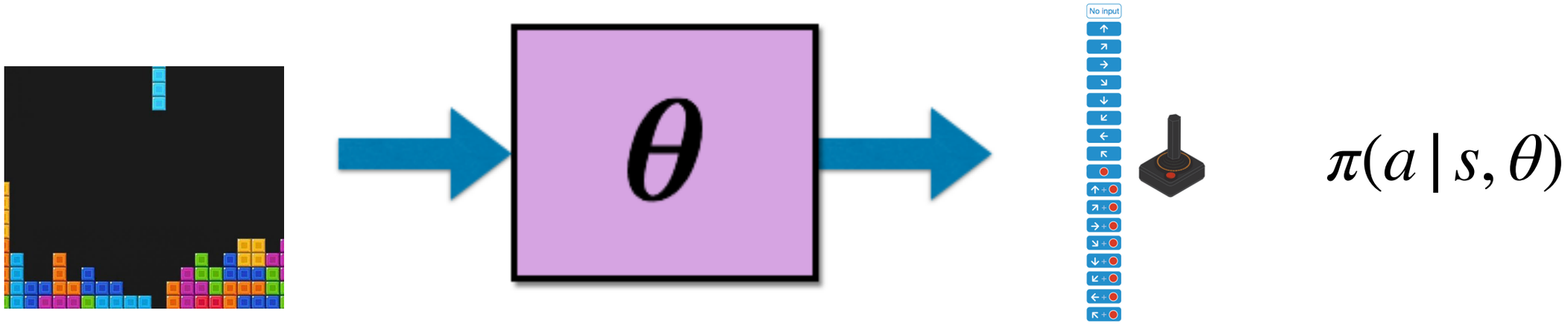


Do nothing

- If the number of state space was small, we could have an exhaustive enumeration of states paired with the optimal action(s) to take.
- For Tetris, and the real world, it is not small.. There will always be states at test time that we have not visited at training time, i.e., and we will not know what to do.
- Q: any solutions?

A concrete example: Playing Tetris

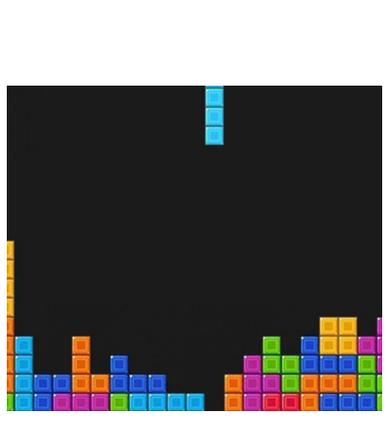
- Policy: a mapping from states to actions

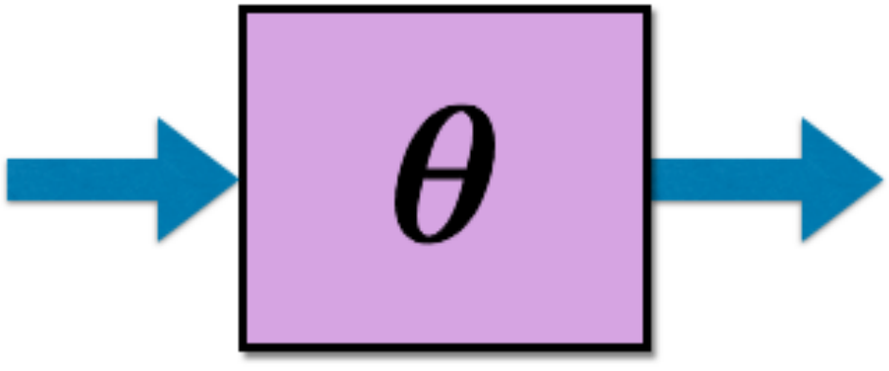


- A function parameterized by parameters \theta. In principal, we can represent actions to take for all states.
- Q1: Who is larger: the number of parameters or the number of states?
- Q2: Do we know how to act now on states that we didn't see during training?
- Q3: What are some properties that our function should have to be able to **generalize from seen to unseen states**, i.e., suggest reasonanable actions to take?

A concrete example: Playing Tetris

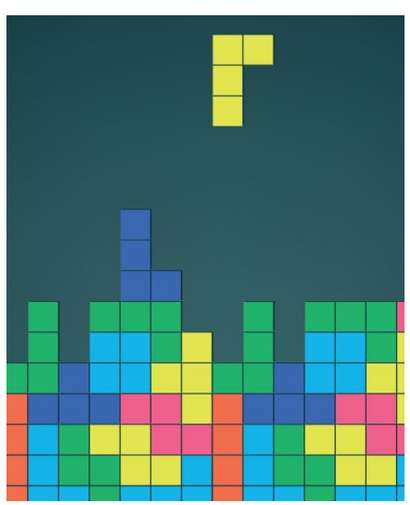
- Policy: a mapping from states to actions



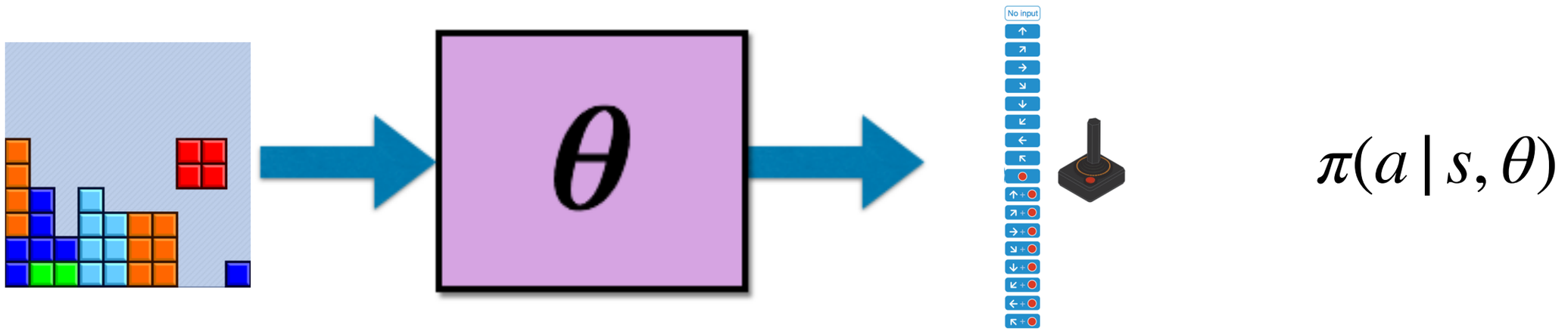


$$\pi(a \mid s, \theta)$$

- Our function should learn features that make *two distinct states (in pixel space) to be close in feature space when they share the same optimal action*. The action prediction will be based on such discovered features. E.g., in our case, the color of the blocks is irrelevant, as well as they location on the whole board.



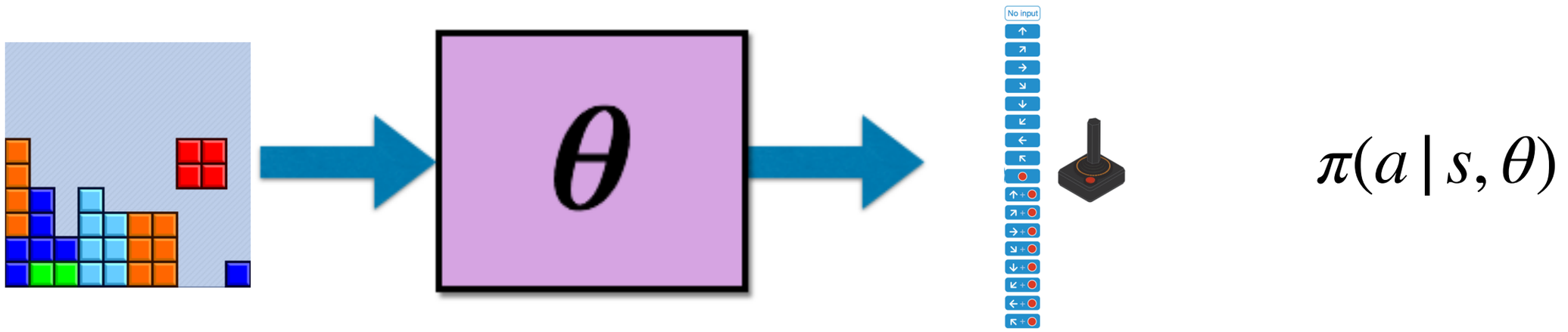
Who will provide the features?



Two choices:

1. **The engineer will manually define** a set of features to capture the state (board configuration). Then the model will just map those features to a distribution over actions, e.g., using a linear model.
2. **The model will learn** the features by
   - imitating expert actions, when an expert is available
   - trial-and-error, while playing the game

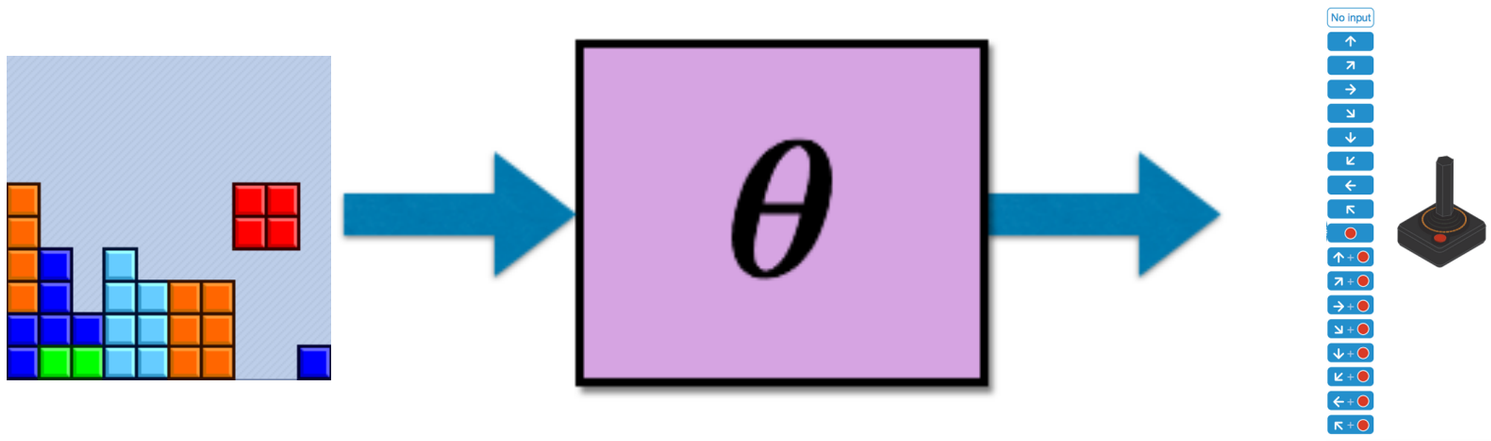
Q: How can we learn the parameters of our policy?



An encoding for the state. Two choices:

1. **The engineer** will manually define a set of features to capture the state (board configuration). Then the model will just map those features to a distribution over actions, e.g., using a linear model.
2. **The learning model** will learn the features by
   - imitating expert actions, when an expert is available
   - trial-and-error, while playing the game

Q: How can we learn the weights?

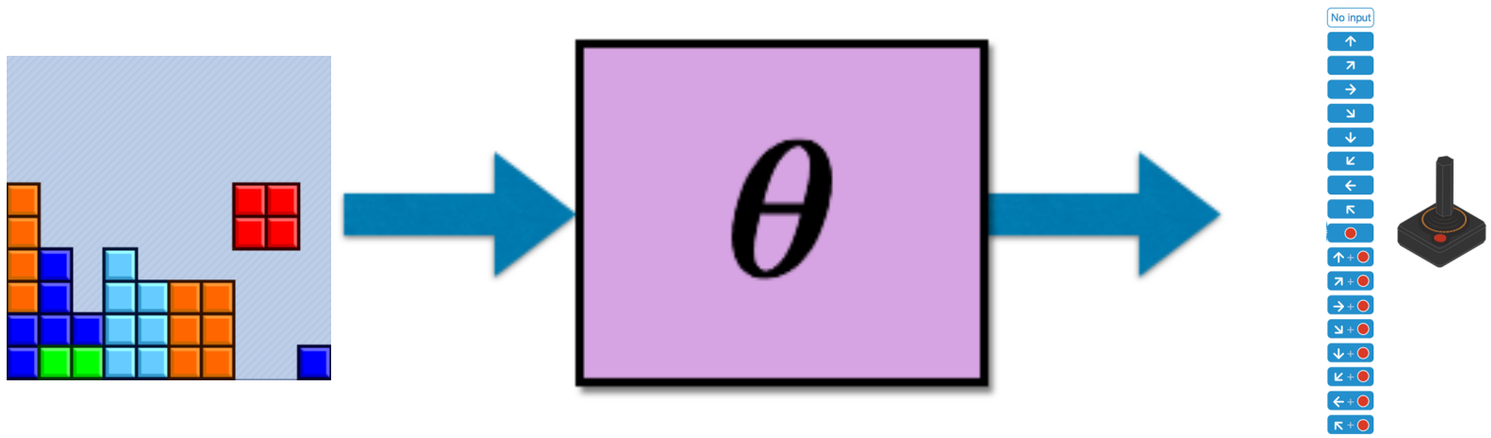


$$\pi(a \mid s, \theta)$$

$$\max_{\theta} . \quad U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \mid \pi_\theta, \mu_0(s_0) \right]$$

τ : trajectory, a game fragment or a full game

Q: How can we learn the weights?



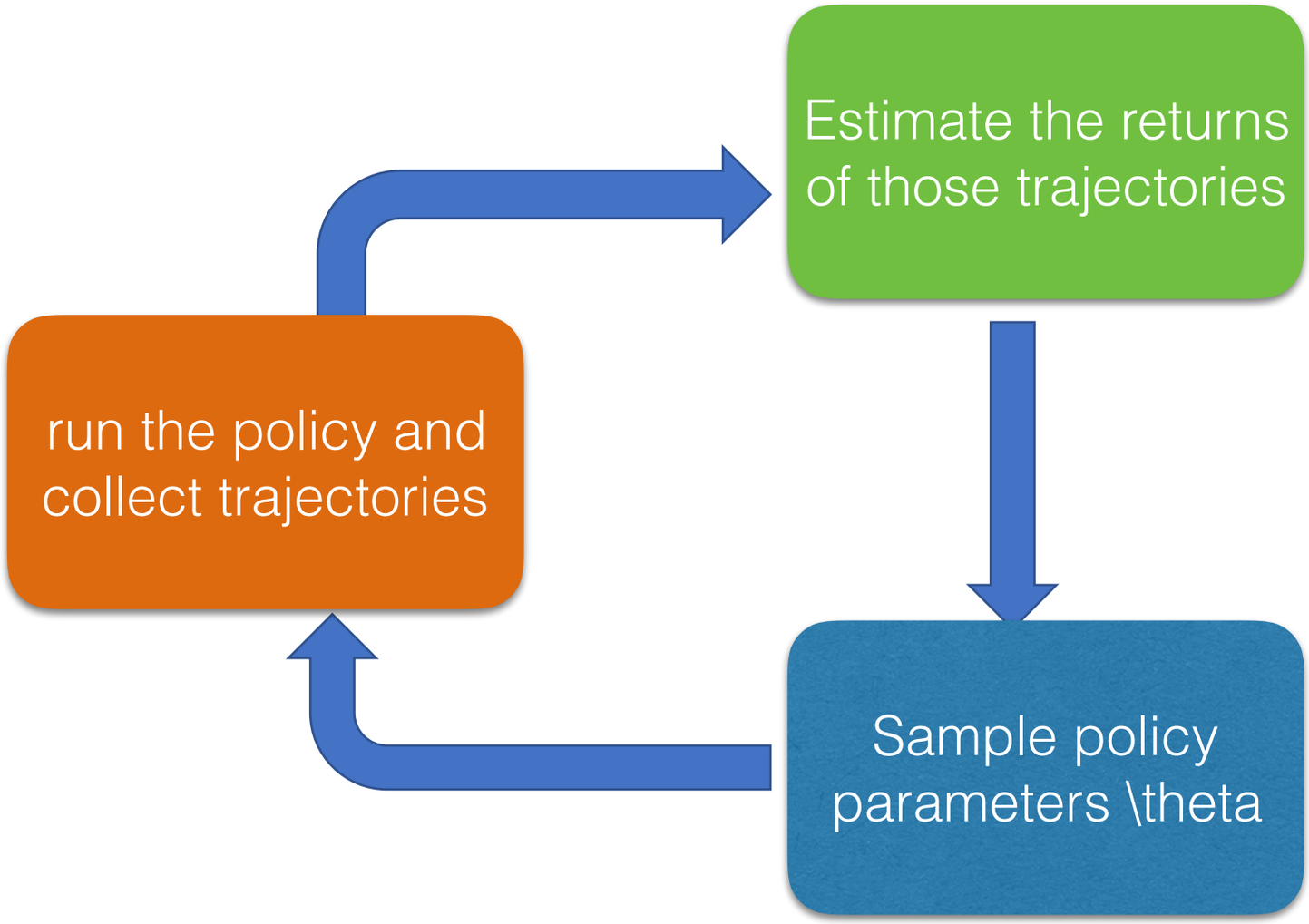
$$\pi(a \mid s, \theta)$$

$$\max_{\theta} . \quad U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \mid \pi_\theta \right]$$

τ : trajectory, a game fragment or a full game

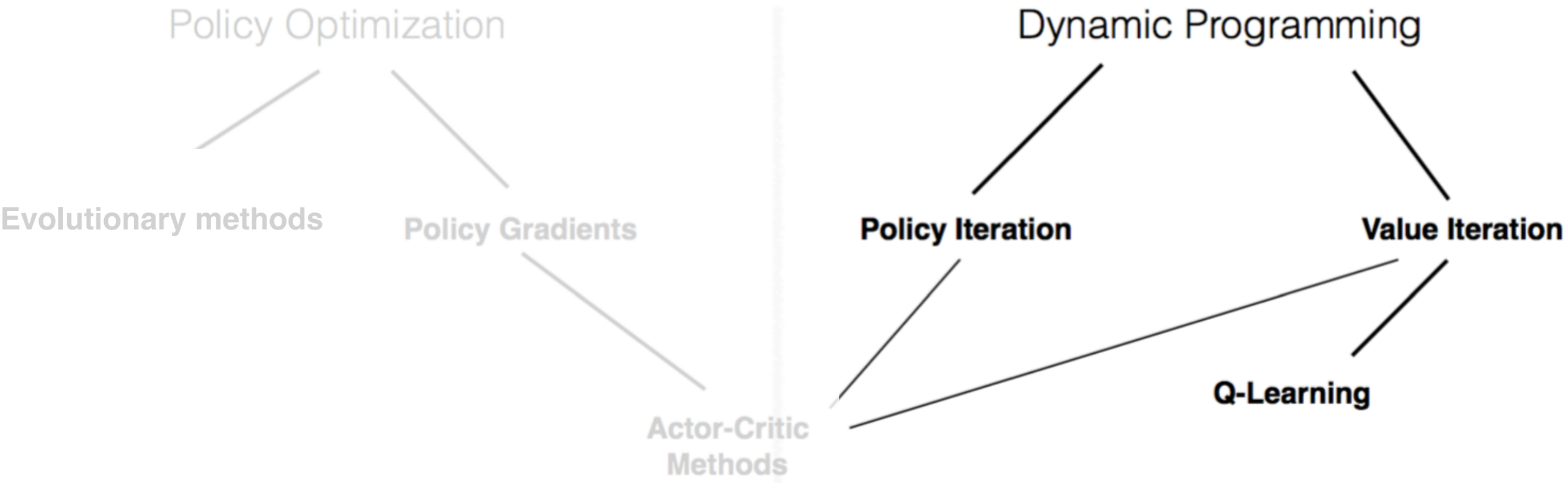
How are we going to solve this optimization?

Simple choice: Black box optimization



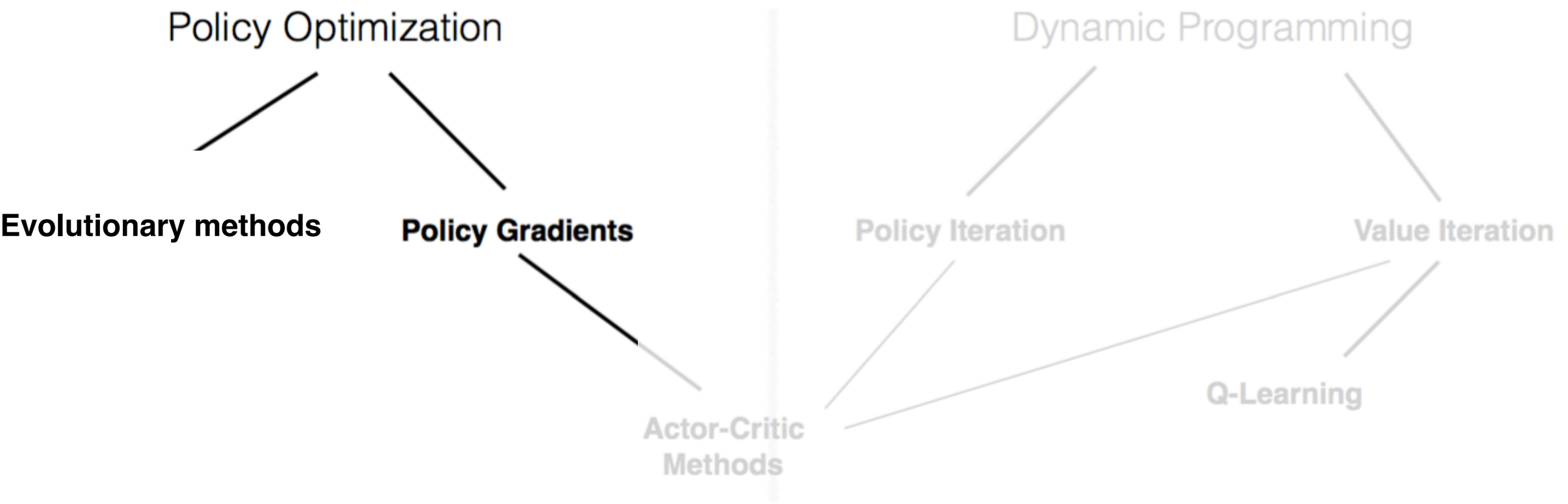
- Perturb randomly policy parameters, collect trajectories, evaluate the trajectories, keep the parameters that gave the largest improvement, repeat
- Black-box optimization: No information regarding the structure of the reward, that it is additive over states, that states are interconnected in a particular way, etc..

$$\max_{\theta} . \quad U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \,|\, \pi_\theta, \mu_0(s_0) \right]$$



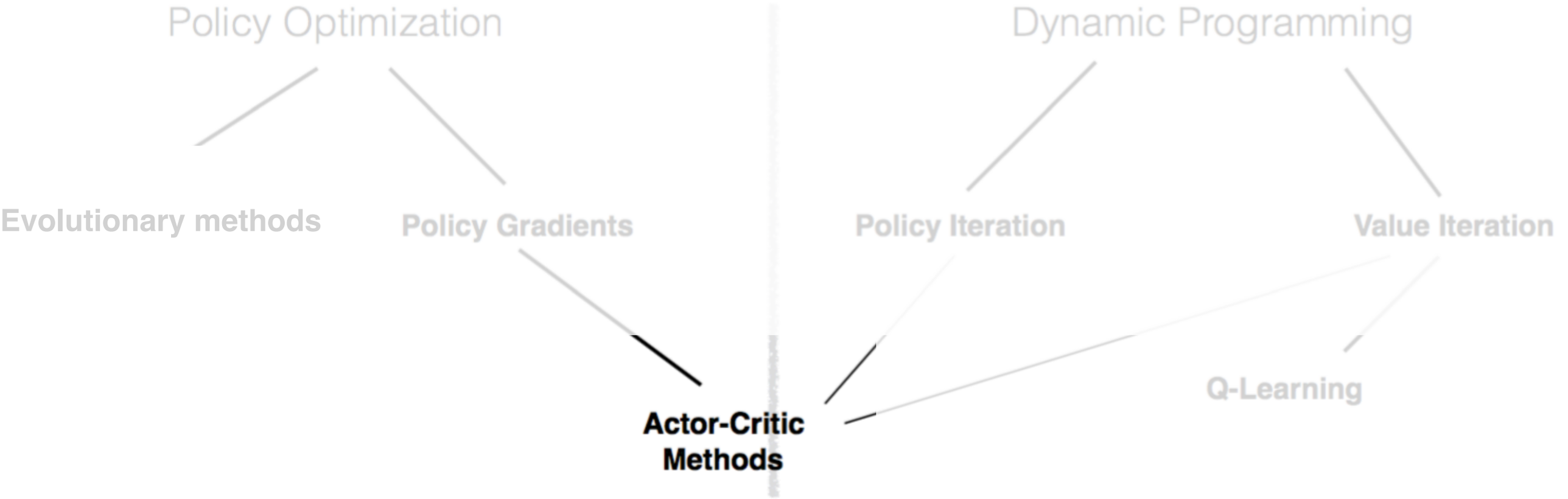
Dynamic programming methods compute values for states: try to estimate which states are good and which are bad for a given policy.They use recursive equations/updates that encode how one state results from another: the value of a state will update the value of the states that are after it and before it.

$$\max_{\theta} . \quad U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \,|\, \pi_\theta, \mu_0(s_0) \right]$$



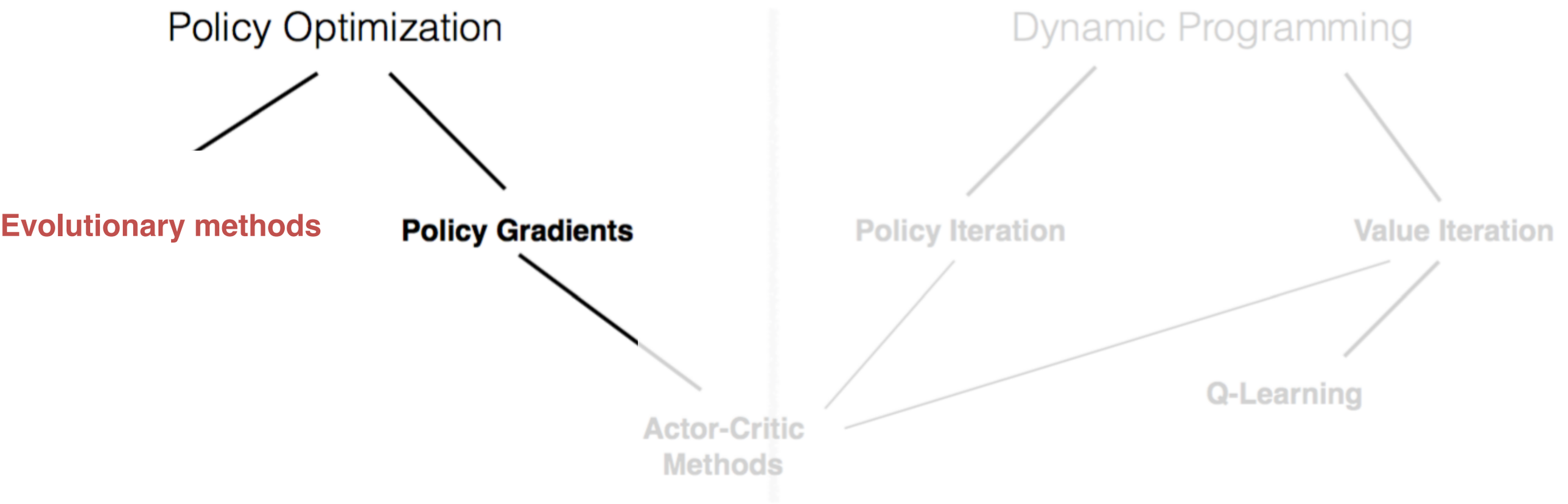
Policy optimization just searches over policy parameters without computing values for states: it just looks are the raw rewards obtained as we search over the parameters. It does not exploit the structure of the states

$$\max_{\theta} . \quad U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \,|\, \pi_\theta, \mu_0(s_0) \right]$$



Actor critic method both search over policies and over state values, and use the state values, as opposed to the raw rewards to guide policy search. Estimate of value function are usually way less noisy than raw rewards

$$\max_{\theta} . \quad U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \,|\, \pi_\theta, \mu_0(s_0) \right]$$



Policy optimization just searches over policy parameters without computing values for states: it just looks are the raw rewards obtained as we search over the parameters. It does not exploit the structure of the states

Evolutionary methods

$$\max_{\theta} . \quad U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[R(\tau) \,|\, \pi_\theta, \mu_0(s_0) \right]$$

General algorithm:

Initialize a population of parameter vectors (genotypes)

1. Make random perturbations (mutations) to each parameter vector
2. Evaluate the perturbed parameter vector (fitness)
3. Keep the perturbed vector if the result improves (selection)
4. GOTO 1

Biologically plausible…

Cross-entropy method

Parameters to be sampled from a multivariate Gaussian with diagonal covariance. We will evolve this Gaussian towards parameter samples that have highest fitness

Input: parameter space Θ, number of parameter vectors n, proportion $\rho \leq 1$, noise η
Initialize: Set the parameter $\mu = \bar{0}$ and $\sigma^2 = 100I$ (I is the identity matrix)
for $k = 1, 2, \ldots$ **do**
Generate a random sample of n parameter vectors $\{\theta_i\}_{i=1}^{n} \sim \mathcal{N}(\mu, \sigma^2 I)$
For each θ_i, play L games and calculate the average number of rows removed (score) by the controller
Select $\lfloor \rho n \rfloor$ parameters with the highest score $\theta'_1, \ldots, \theta'_{\lfloor \rho n \rfloor}$
Update μ and σ: $\mu(j) = \frac{1}{\lfloor \rho n \rfloor} \sum_{i=1}^{\lfloor \rho n \rfloor} \theta'_i(j)$ and $\sigma^2(j) = \frac{1}{\lfloor \rho n \rfloor} \sum_{i=1}^{\lfloor \rho n \rfloor} [\theta'_i(j) - \mu(j)]^2 + \eta$

Works embarrassingly well in low-dimensions, e.g., the 22 *Bertsekas features*.

Cross-entropy method

Work embarrassingly well in low-dimensions

Method	Mean Score	Reference
Nonreinforcement learning		
Hand-coded	631,167	Dellacherie (Fahey, 2003)
Genetic algorithm	586,103	(Böhm et al., 2004)
Reinforcement learning		
Relational reinforcement learning+kernel-based regression	≈50	Ramon and Driessens (2004)
Policy iteration	3183	Bertsekas and Tsitsiklis (1996)
Least squares policy iteration	<3000	Lagoudakis, Parr, and Littman (2002)
Linear programming + Bootstrap	4274	Farias and van Roy (2006)
Natural policy gradient	≈6800	Kakade (2001)
CE+RL	21,252	
CE+RL, constant noise	72,705	
CE+RL, decreasing noise	348,895	

István Szita and András Lörincz. "Learning Tetris using the noisy cross-entropy method". In: *Neural computation* 18.12 (2006), pp. 2936–2941

$\mu \in \mathbb{R}^{22}$

Much later:

Approximate Dynamic Programming Finally Performs Well in the Game of Tetris

Victor Gabillon
INRIA Lille - Nord Europe, Team SequeL, FRANCE
victor.gabillon@inria.fr

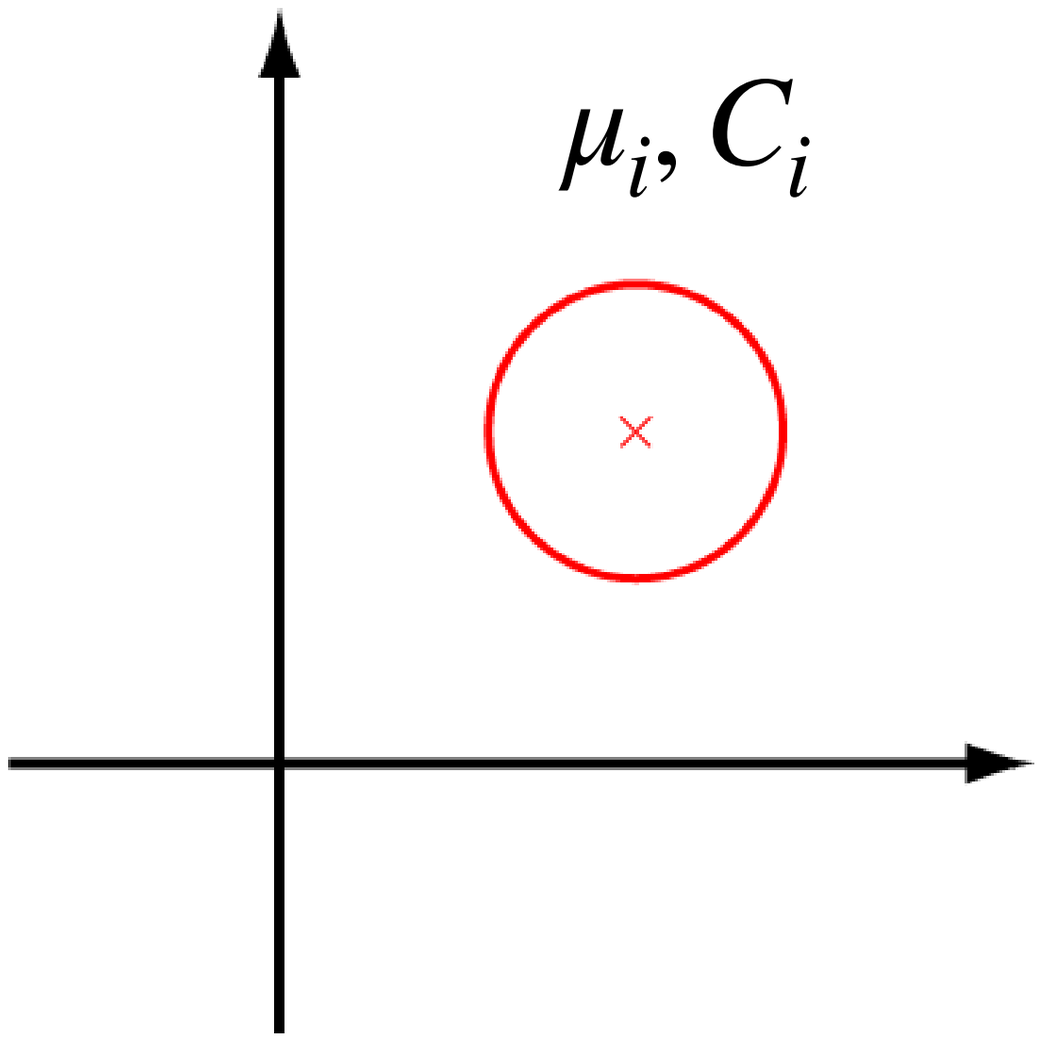
Mohammad Ghavamzadeh*
INRIA Lille - Team SequeL & Adobe Research
mohammad.ghavamzadeh@inria.fr

Bruno Scherrer
INRIA Nancy - Grand Est, Team Maia, FRANCE
bruno.scherrer@inria.fr

[NIPS 2013]

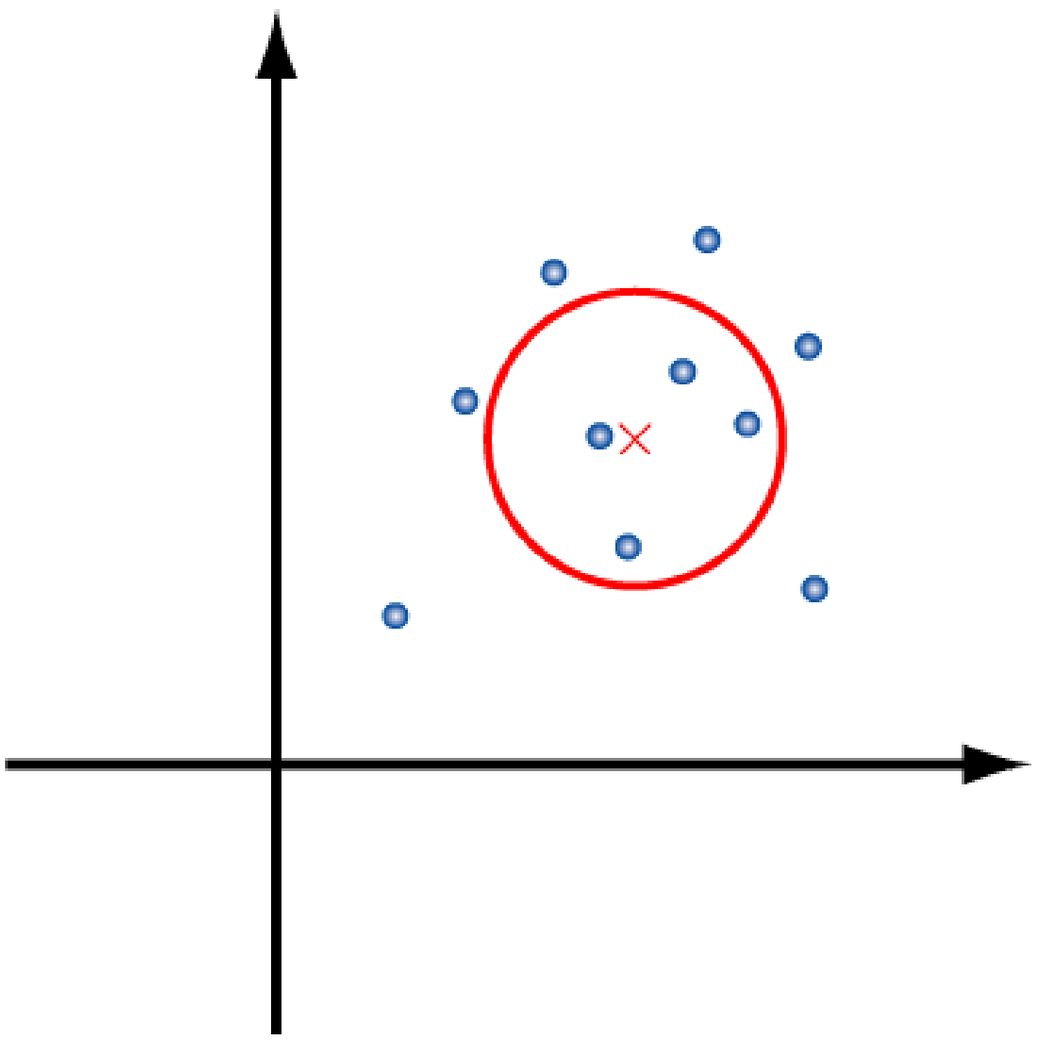
Covariance Matrix Adaptation

We can also consider a full covariance matrix



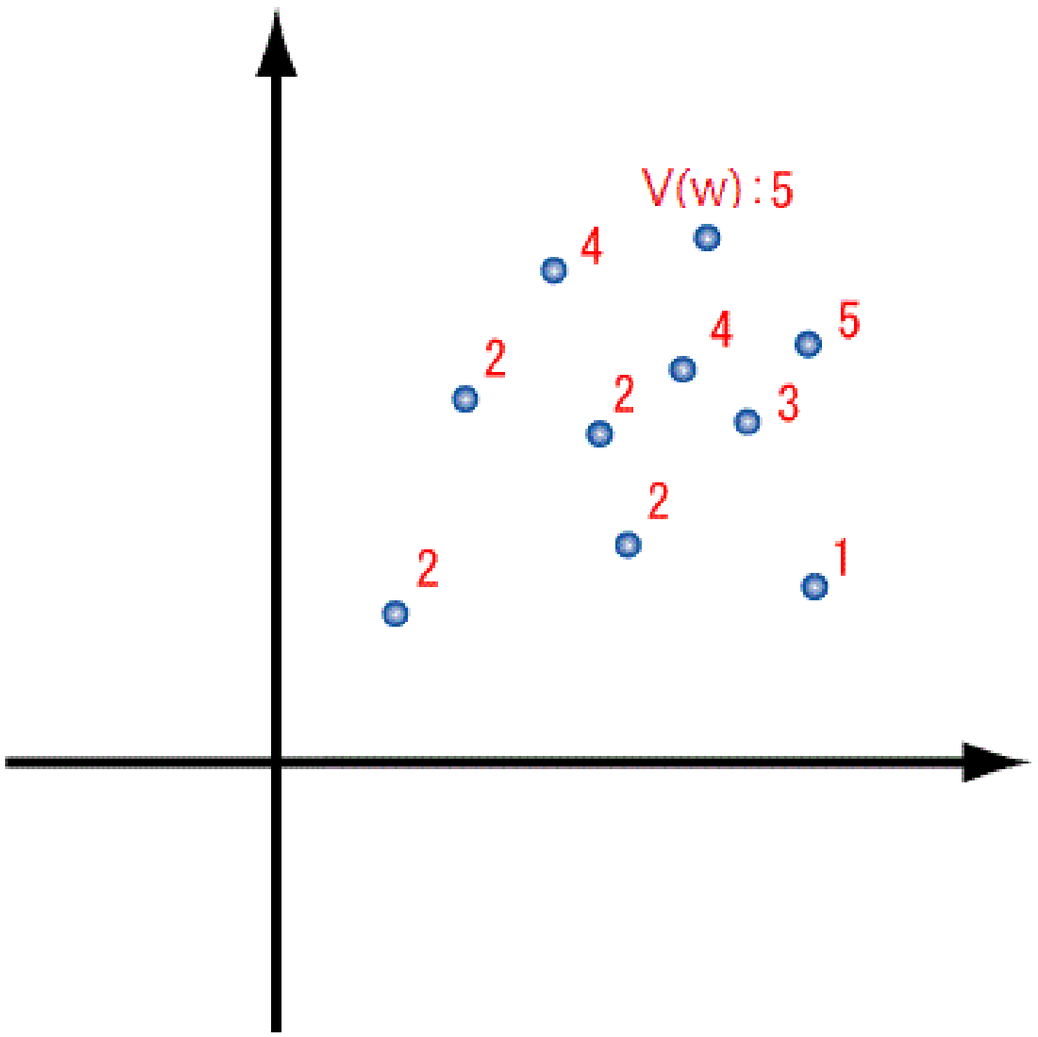
- Sample
- Select elites
- Update mean
- Update covariance
- iterate

Covariance Matrix Adaptation



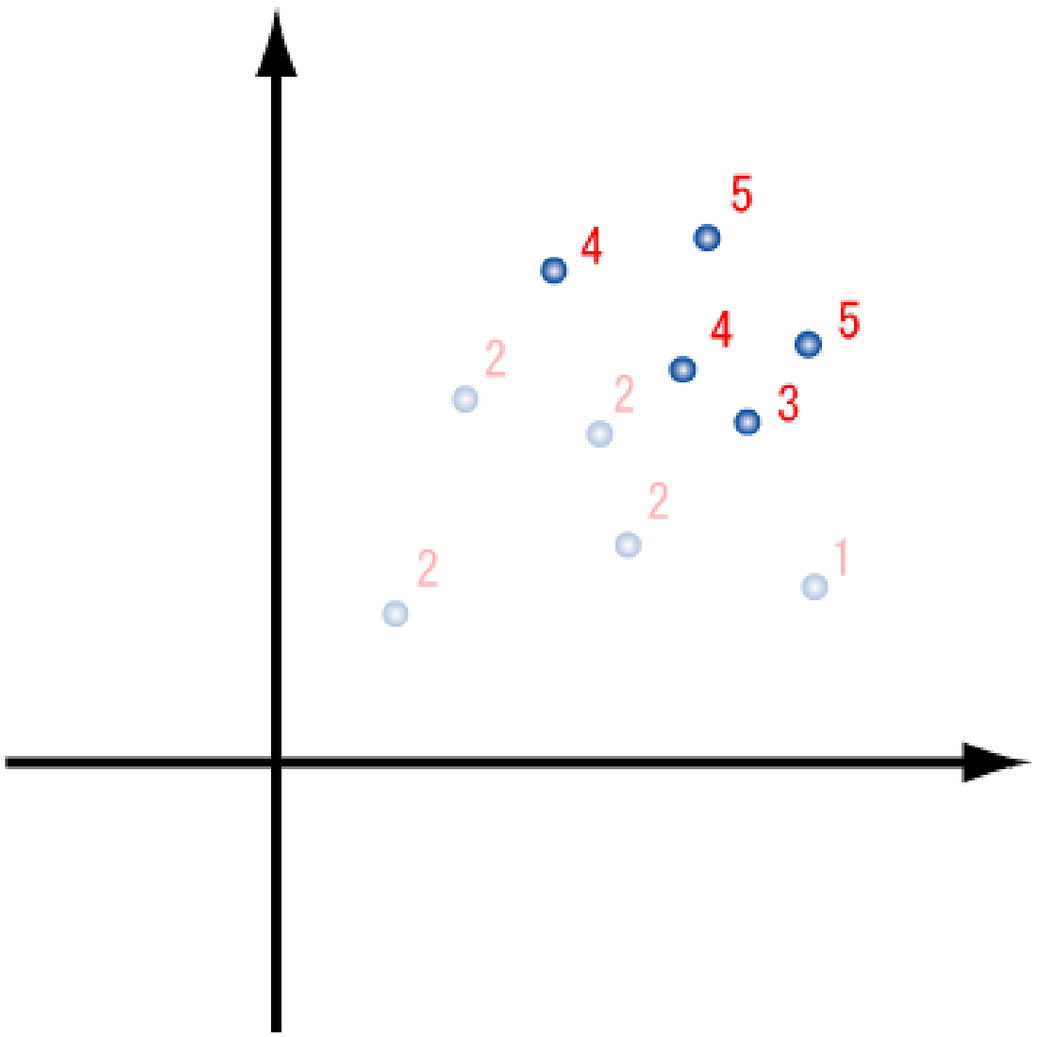
- Sample
- Select elites
- Update mean
- Update covariance
- iterate

Covariance Matrix Adaptation



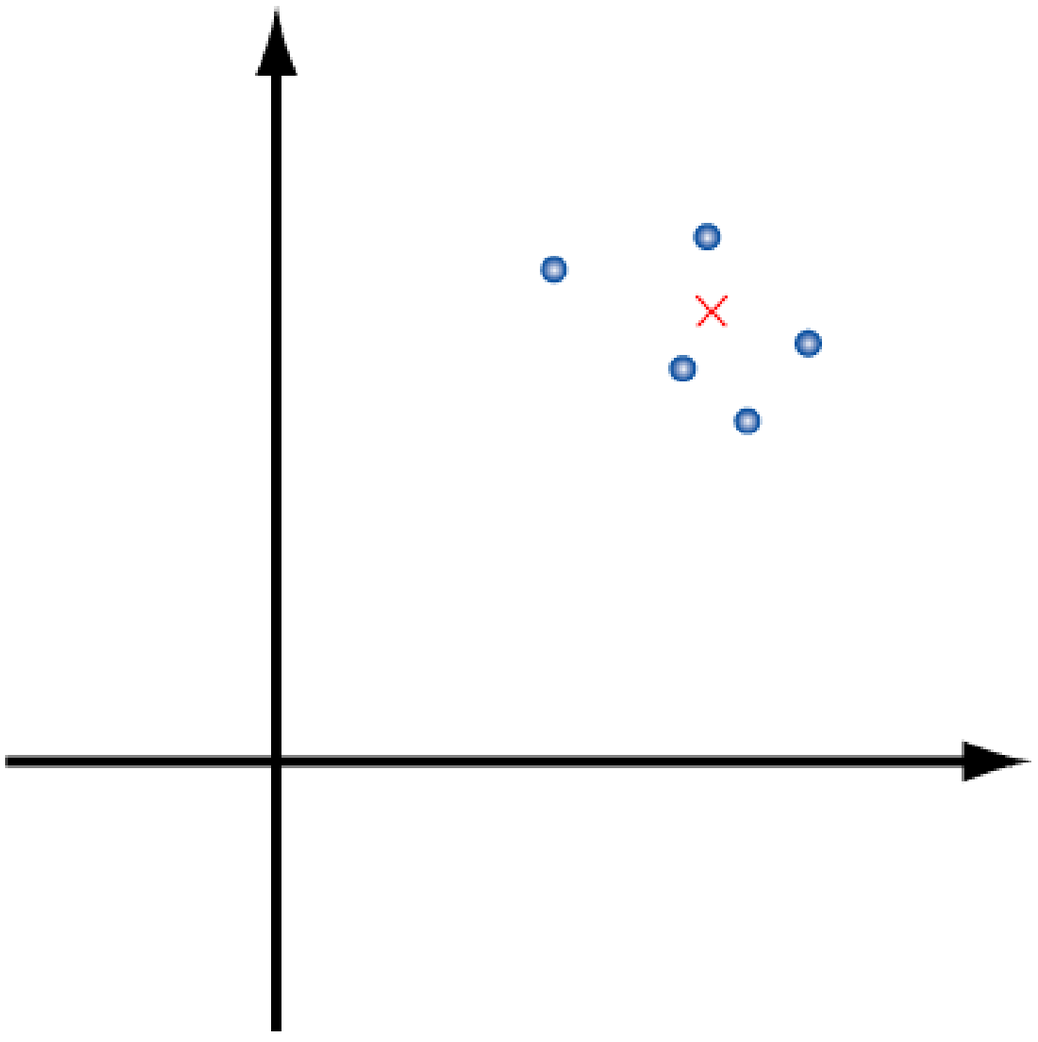
- Sample
- Select elites
- Update mean
- Update covariance
- iterate

Covariance Matrix Adaptation



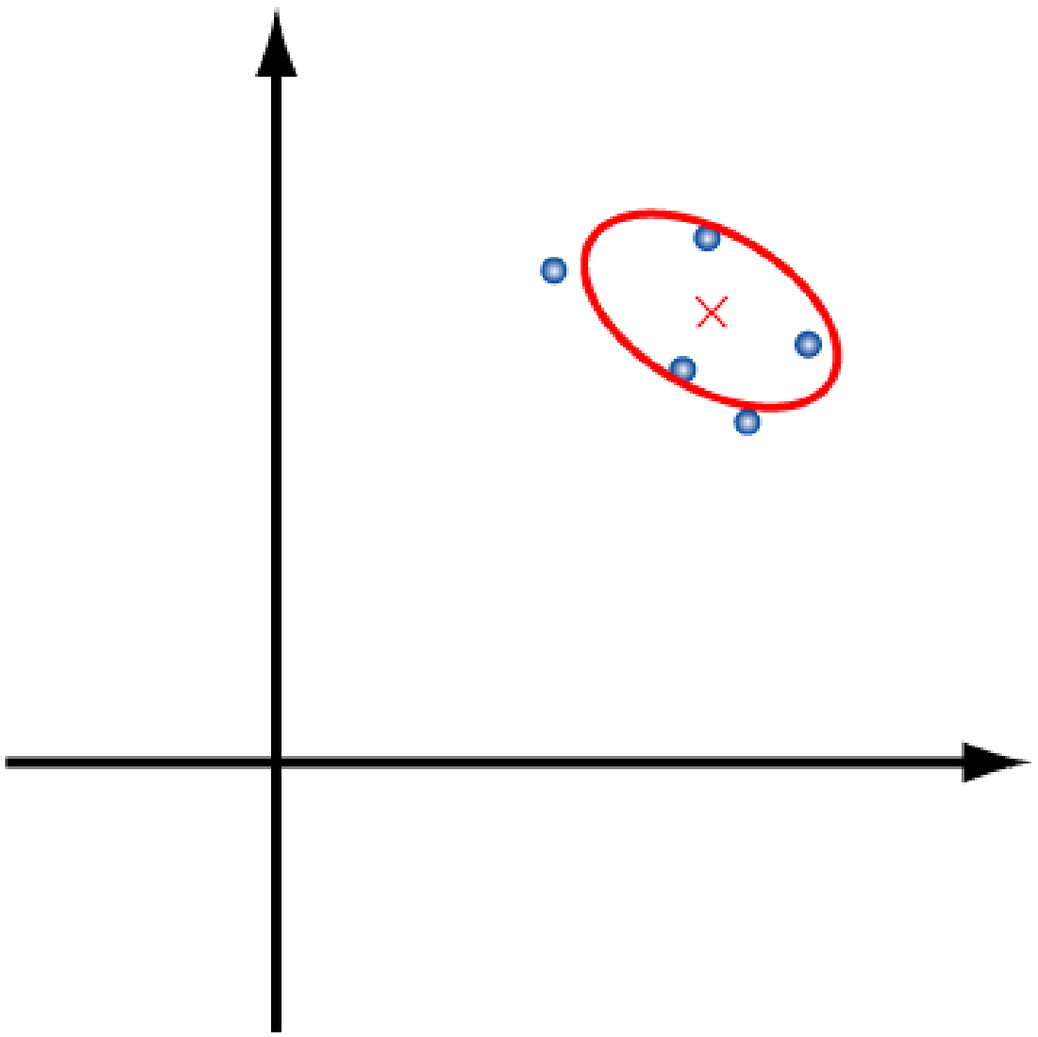
- Sample
- Select elites
- Update mean
- Update covariance
- iterate

Covariance Matrix Adaptation



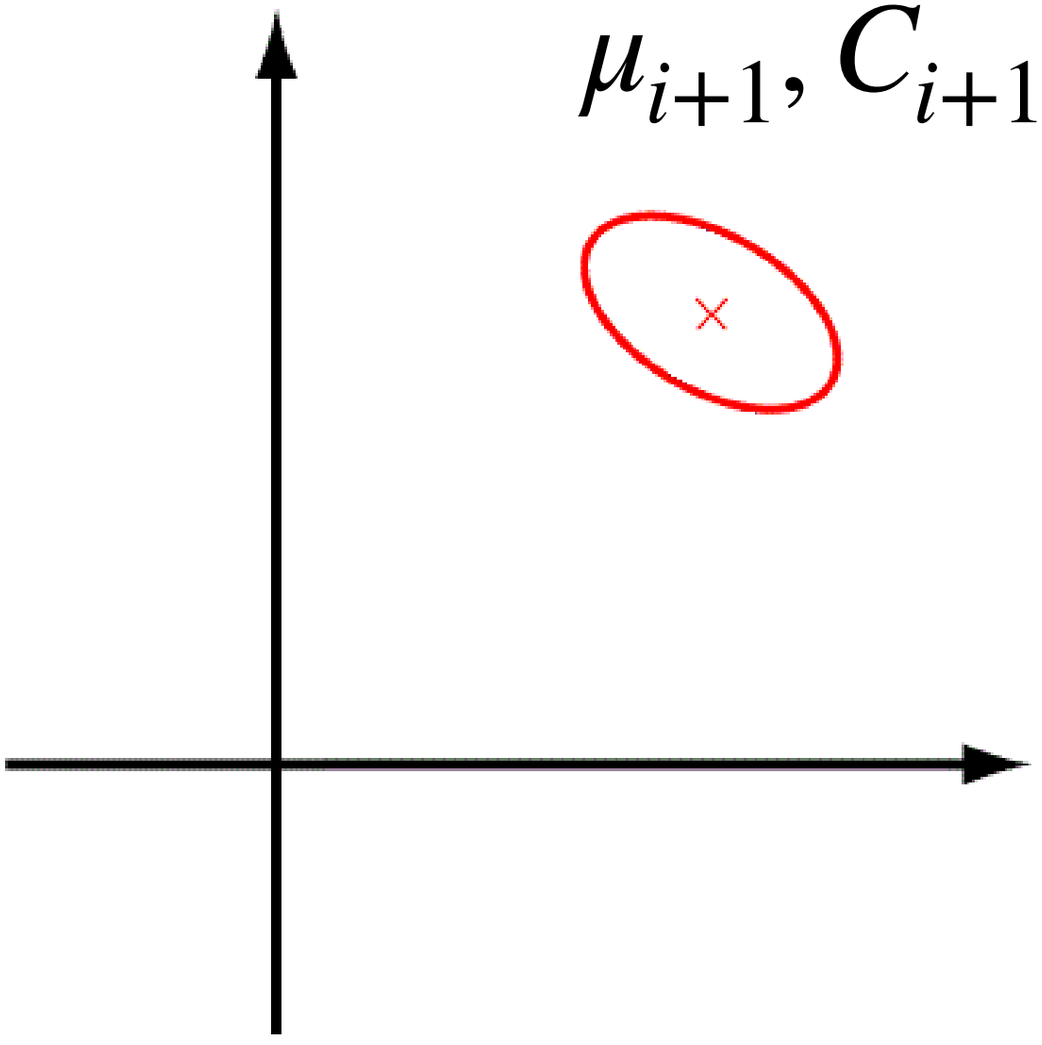
- Sample
- Select elites
- Update mean
- Update covariance
- iterate

Covariance Matrix Adaptation



- Sample
- Select elites
- Update mean
- Update covariance
- iterate

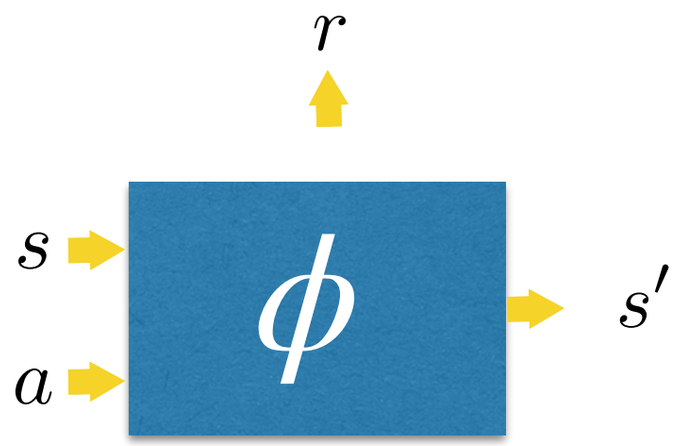
Covariance Matrix Adaptation



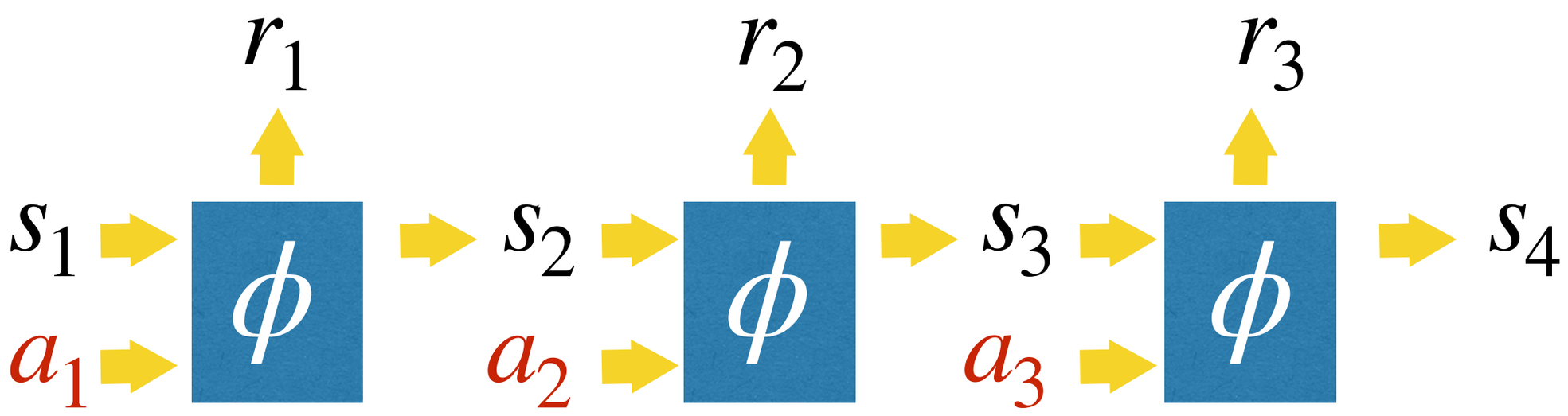
- Sample
- Select elites
- Update mean
- Update covariance
- iterate

Evolution for Action Selection

- A learned model:

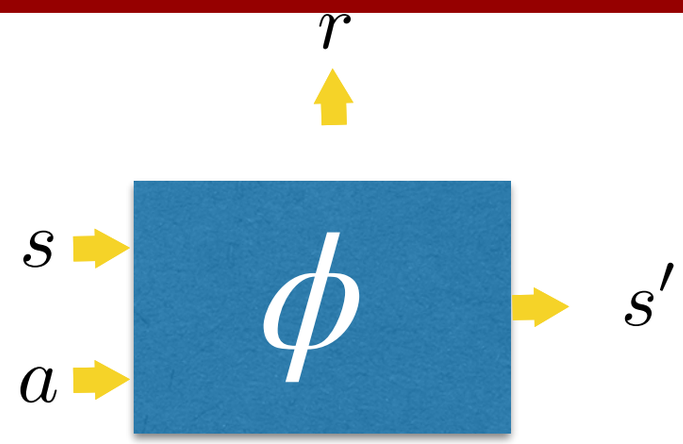


- Unrolling the model forward:

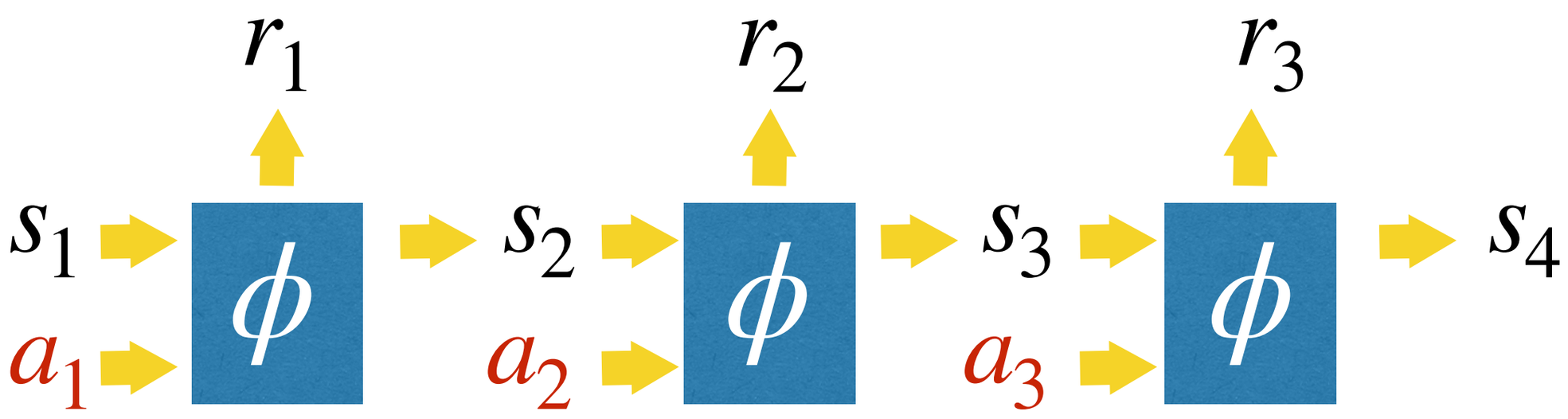


Evolution for Action Selection

- A learned model:

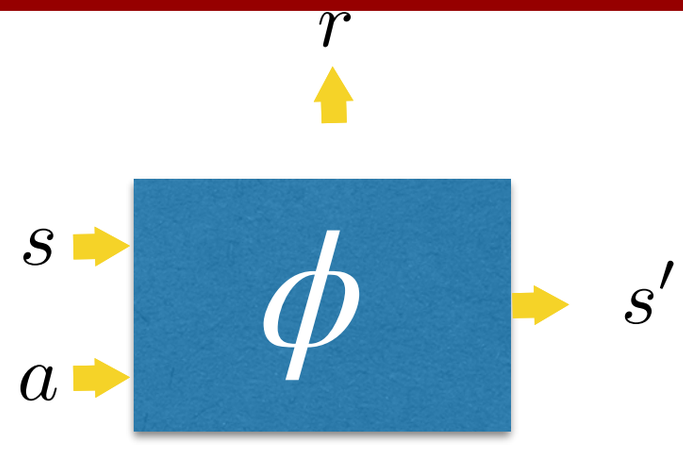


- Objective: maximizing sum of rewards

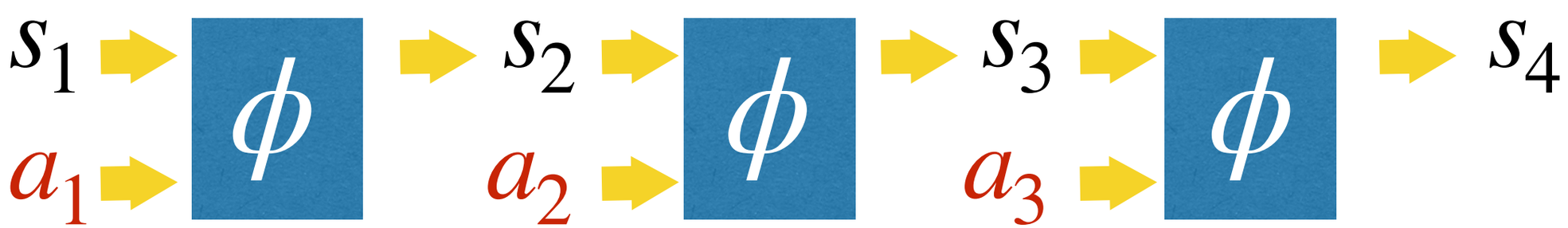


Evolution for Action Selection

- A learned model:

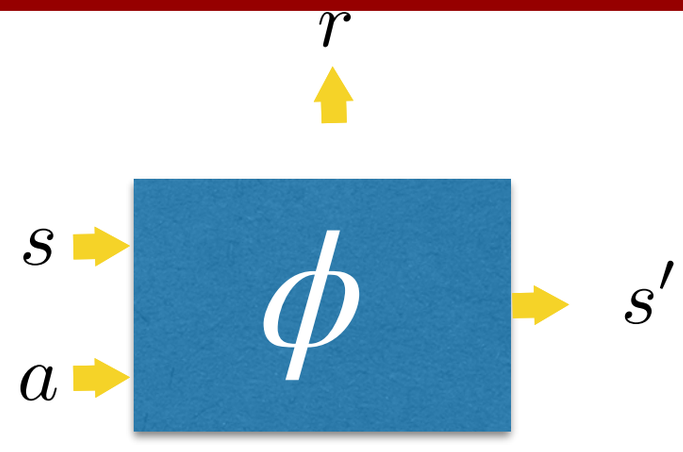


- Objective: reaching a desired state: $\|s_4 - s_*\|$

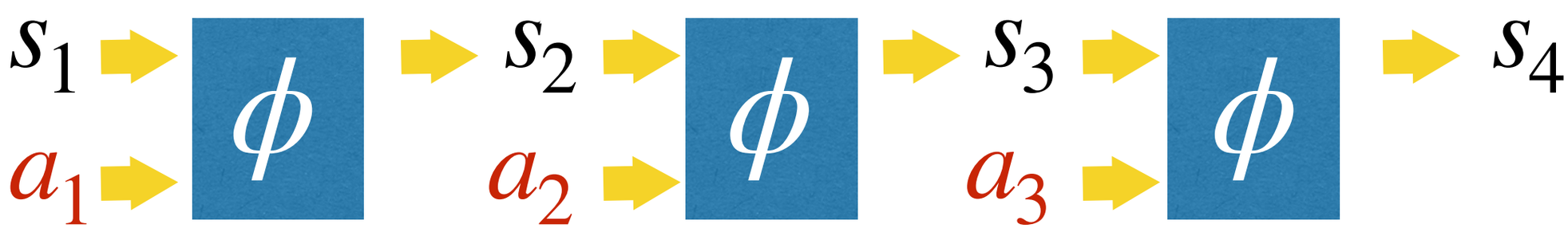


Evolution for Action Selection

- A learned model:



- Objective: reaching a desired state: $\|s_4 - s_*\|$



- Search over the action sequence to apply using CEM

Natural Evolutionary Strategies

- So far, we have been evolving Gaussian parameter distributions, and we have been selecting the best parameter offsprings
- NES considers every offspring.
- We will start the presentation using a general distribution for our parameters, whose parameters we are trying to evolve so that samples can generate high utility values

Natural Evolutionary Strategies

Consider a distribution of policy parameters: $\theta \sim P_\mu(\theta)$

Goal: maximize $\mathbb{E}_{\theta \sim P_\mu} \mathrm{F}(\theta)$, where fitness score $\mathrm{F}(\theta) = \mathbb{E}_{\tau \sim \pi_\theta, s_0 \sim \mu_0(s)} R(\tau)$

Optimize by stochastic gradient ascent:

Natural Evolutionary Strategies

Consider a distribution of policy parameters: $\theta \sim P_\mu(\theta)$

Goal: maximize $\mathbb{E}_{\theta \sim P_\mu} \mathrm{F}(\theta)$, where fitness score $\mathrm{F}(\theta) = \mathbb{E}_{\tau \sim \pi_\theta, s_0 \sim \mu_0(s)} R(\tau)$

Optimize by stochastic gradient ascent:

$$\nabla_\mu \mathbb{E}_{\theta \sim P_\mu(\theta)} \left[\mathrm{F}(\theta) \right] = \nabla_\mu \int P_\mu(\theta) \mathrm{F}(\theta) d\theta$$

Natural Evolutionary Strategies

Consider a distribution of policy parameters: $\theta \sim P_\mu(\theta)$

Goal: maximize $\mathbb{E}_{\theta \sim P_\mu} \mathrm{F}(\theta)$, where fitness score $\mathrm{F}(\theta) = \mathbb{E}_{\tau \sim \pi_\theta, s_0 \sim \mu_0(s)} R(\tau)$

Optimize by stochastic gradient ascent:

$$\begin{aligned} \nabla_\mu \mathbb{E}_{\theta \sim P_\mu(\theta)} \left[\mathrm{F}(\theta) \right] &= \nabla_\mu \int P_\mu(\theta) \mathrm{F}(\theta) d\theta \\ &= \int \nabla_\mu P_\mu(\theta) \mathrm{F}(\theta) d\theta \end{aligned}$$

Natural Evolutionary Strategies

Consider a distribution of policy parameters: $\theta \sim P_\mu(\theta)$

Goal: maximize $\mathbb{E}_{\theta \sim P_\mu} \mathrm{F}(\theta)$, where fitness score $\mathrm{F}(\theta) = \mathbb{E}_{\tau \sim \pi_\theta, s_0 \sim \mu_0(s)} R(\tau)$

Optimize by stochastic gradient ascent:

$$\begin{aligned}
\nabla_\mu \mathbb{E}_{\theta \sim P_\mu(\theta)} \left[\mathrm{F}(\theta) \right] &= \nabla_\mu \int P_\mu(\theta) \mathrm{F}(\theta) d\theta \\
&= \int \nabla_\mu P_\mu(\theta) \mathrm{F}(\theta) d\theta \\
&= \int P_\mu(\theta) \frac{\nabla_\mu P_\mu(\theta)}{P_\mu(\theta)} \mathrm{F}(\theta) d\theta
\end{aligned}$$

Natural Evolutionary Strategies

Consider a distribution of policy parameters: $\theta \sim P_\mu(\theta)$

Goal: maximize $\mathbb{E}_{\theta \sim P_\mu} \mathrm{F}(\theta)$, where fitness score $\mathrm{F}(\theta) = \mathbb{E}_{\tau \sim \pi_\theta, s_0 \sim \mu_0(s)} R(\tau)$

Optimize by stochastic gradient ascent:

$$
\begin{aligned}
\nabla_\mu \mathbb{E}_{\theta \sim P_\mu(\theta)} \left[\mathrm{F}(\theta) \right] &= \nabla_\mu \int P_\mu(\theta) \mathrm{F}(\theta) d\theta \\
&= \int \nabla_\mu P_\mu(\theta) \mathrm{F}(\theta) d\theta \\
&= \int P_\mu(\theta) \frac{\nabla_\mu P_\mu(\theta)}{P_\mu(\theta)} \mathrm{F}(\theta) d\theta \\
&= \int P_\mu(\theta) \nabla_\mu \log P_\mu(\theta) \mathrm{F}(\theta) d\theta
\end{aligned}
$$

Natural Evolutionary Strategies

Consider a distribution of policy parameters: $\theta \sim P_\mu(\theta)$

Goal: maximize $\mathbb{E}_{\theta \sim P_\mu} \mathrm{F}(\theta)$, where fitness score $\mathrm{F}(\theta) = \mathbb{E}_{\tau \sim \pi_\theta, s_0 \sim \mu_0(s)} R(\tau)$

Optimize by stochastic gradient ascent:

$$
\begin{aligned}
\nabla_\mu \mathbb{E}_{\theta \sim P_\mu(\theta)} \left[\mathrm{F}(\theta) \right] &= \nabla_\mu \int P_\mu(\theta) \mathrm{F}(\theta) d\theta \\
&= \int \nabla_\mu P_\mu(\theta) \mathrm{F}(\theta) d\theta \\
&= \int P_\mu(\theta) \frac{\nabla_\mu P_\mu(\theta)}{P_\mu(\theta)} \mathrm{F}(\theta) d\theta \\
&= \int P_\mu(\theta) \nabla_\mu \log P_\mu(\theta) \mathrm{F}(\theta) d\theta \\
&= \mathbb{E}_{\theta \sim P_\mu(\theta)} \left[\nabla_\mu \log P_\mu(\theta) \mathrm{F}(\theta) \right]
\end{aligned}
$$

A concrete example

- Suppose $\theta \sim P_\mu(\theta)$ is a Gaussian distribution with mean μ, and covariance matrix $\sigma^2 I$

$$\log P_\mu(\theta) = -\frac{||\theta - \mu||^2}{2\sigma^2} + \text{const}$$

$$\nabla_\mu \log P_\mu(\theta) = \frac{\theta - \mu}{\sigma^2}$$

Remember we need to aggregate such vectors for multiple samples to approximate the expectation:

$$\mathbb{E}_{\theta \sim P_\mu(\theta)} \left[\nabla_\mu \log P_\mu(\theta) \mathrm{F}(\theta) \right]$$

Sampling from a multivariate Gaussian

- Suppose $\theta \sim P_\mu(\theta)$ is a Gaussian distribution with mean μ, and covariance matrix $\sigma^2 I$

Imagine we have access to random vectors $\epsilon \sim \mathcal{N}(0,I)$

$$\theta_1 = \mu + \sigma * \epsilon_1,\ \epsilon_1 \sim \mathcal{N}(0,I)$$

$$\theta_2 = \mu + \sigma * \epsilon_2,\ \epsilon_2 \sim \mathcal{N}(0,I)$$

The theta samples have the desired mean and variance

A concrete example

- Suppose $\theta \sim P_\mu(\theta)$ is a Gaussian distribution with mean μ, and covariance matrix $\sigma^2 I$

$$\log P_\mu(\theta) = -\frac{||\theta - \mu||^2}{2\sigma^2} + \text{const}$$

$$\nabla_\mu \log P_\mu(\theta) = \frac{\theta - \mu}{\sigma^2}$$

A concrete example

- Suppose $\theta \sim P_\mu(\theta)$ is a Gaussian distribution with mean μ, and covariance matrix $\sigma^2 I$

$$\log P_\mu(\theta) = -\frac{||\theta - \mu||^2}{2\sigma^2} + \text{const}$$

$$\nabla_\mu \log P_\mu(\theta) = \frac{\theta - \mu}{\sigma^2}$$

$$\theta_1 = \mu + \sigma * \epsilon_1,\ \epsilon_1 \sim \mathcal{N}(0,I)$$

$$\theta_2 = \mu + \sigma * \epsilon_2,\ \epsilon_2 \sim \mathcal{N}(0,I)$$

- If we draw two parameter samples θ_1, θ_2, and obtain two trajectories τ_1, τ_2:

$$\mathbb{E}_{\theta \sim P_\mu(\theta)} \left[\nabla_\mu \log P_\mu(\theta) \mathrm{F}(\theta) \right]$$

$$\approx \frac{1}{2} \left[\mathrm{F}(\theta_1) \frac{\theta_1 - \mu}{\sigma^2} + \mathrm{F}(\theta_2) \frac{\theta_2 - \mu}{\sigma^2} \right], \text{ where } \mathrm{F}(\theta_1) = R(\tau_1), \mathrm{F}(\theta_2) = R(\tau_2)$$

$$= \frac{1}{2\sigma} \left[\mathrm{F}(\theta_1)\epsilon_1 + \mathrm{F}(\theta_1)\epsilon_2 \right]$$

Natural Evolutionary Strategies

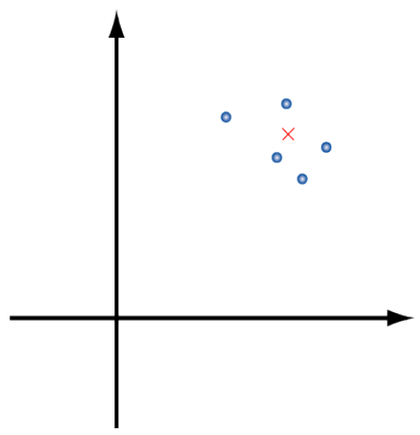
Algorithm 1 Evolution Strategies

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0
for $t = 0, 1, 2, \ldots$ **do**
  Sample $\epsilon_1, \ldots \epsilon_n \sim \mathcal{N}(0, I)$
  Compute returns $F_i = F(\mu_t + \sigma\epsilon_i)$ for $i = 1, \ldots, n$
  Set $\mu_{t+1} \leftarrow \mu_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
end for

Compare the two update rules for the mean

$$\mu_{t+1} = \mu_t + \frac{\alpha}{n\sigma} \sum_{i=1}^{n} \mathrm{F}(\theta_i)\epsilon_i$$

$$\mu_{t+1} = \sum_{i=1}^{n_{elit}} \theta_i^{elit,t}$$



- Sample
- Select elites
- Update mean
- Update covariance
- iterate

Connection to Finite Differences

- Antithetic sampling
  - Sample a pair of policies with mirror noise $(\theta_+ = \mu + \sigma\epsilon, \theta_- = \mu - \sigma\epsilon)$

Connection to Finite Differences

- Antithetic sampling
  - Sample a pair of policies with mirror noise $(\theta_+ = \mu + \sigma\epsilon, \theta_- = \mu - \sigma\epsilon)$
  - Get a pair of rollouts from environment (τ_+, τ_-)

Connection to Finite Differences

- Antithetic sampling
  - Sample a pair of policies with mirror noise $(\theta_+ = \mu + \sigma\epsilon, \theta_- = \mu - \sigma\epsilon)$
  - Get a pair of rollouts from environment (τ_+, τ_-)
  - SPSA: Finite Difference with random direction

$$\nabla_\mu \mathbb{E}\left[R(\tau)\right] \approx \frac{1}{2}\left[R(\tau_+)\frac{\theta_+ - \mu}{\sigma^2} + R(\tau_-)\frac{\theta_- - \mu}{\sigma^2}\right]$$
$$= \frac{1}{2}\left[R(\tau_+)\frac{\sigma\epsilon}{\sigma^2} + R(\tau_-)\frac{-\sigma\epsilon}{\sigma^2}\right]$$
$$= \frac{\epsilon}{2\sigma}\left[R(\tau_+) - R(\tau_-)\right]$$

$$\nabla_\mu \mathbb{E}_{\theta \sim P_\mu(\theta)}\left[\mathrm{F}(\theta)\right] = \nabla_\mu \int P_\mu(\theta)\mathrm{F}(\theta)d\theta$$

(before we had: $\frac{1}{2\sigma}\left[R(\tau_1)\epsilon_1 + R(\tau_2)\epsilon_2\right]$)

Connection to Finite Differences

- Antithetic sampling
  - Sample a pair of policies with mirror noise $(\theta_+ = \mu + \sigma\epsilon, \theta_- = \mu - \sigma\epsilon)$
  - Get a pair of rollouts from environment (τ_+, τ_-)
  - SPSA: Finite Difference with random direction

$$\nabla_\mu \mathbb{E}_{\theta \sim P_\mu(\theta)} \left[\mathrm{F}(\theta) \right] \approx \frac{1}{2} \left[\mathrm{F}(\theta_+) \frac{\theta_+ - \mu}{\sigma^2} + \mathrm{F}(\theta_-) \frac{\theta_- - \mu}{\sigma^2} \right], \text{ where } \mathrm{F}(\theta_+) = R(\tau_+), \mathrm{F}(\theta_-) = R(\tau_-)$$

$$= \frac{1}{2} \left[\mathrm{F}(\theta_+) \frac{\sigma\epsilon}{\sigma^2} + \mathrm{F}(\theta_-) \frac{\sigma\epsilon}{\sigma^2} \right]$$

$$= \frac{\epsilon}{2\sigma} \left[\mathrm{F}(\theta_+) - \mathrm{F}(\theta_-) \right]$$

Finite Difference

$$\frac{\partial U}{\partial \theta_j}(\theta) = \frac{U(\theta + \epsilon e_j) - U(\theta - \epsilon e_j)}{2\epsilon}$$

(before we had: $\frac{1}{2\sigma} \left[R(\tau_1)\epsilon_1 + R(\tau_2)\epsilon_2 \right]$)

$$e_j = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \leftarrow j\text{'th entry}$$

Question

- Evolutionary methods work well on relatively low-dimensional problems
- Can they be used to optimize deep network policies?

Evolution Strategies as a Scalable Alternative to Reinforcement Learning

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

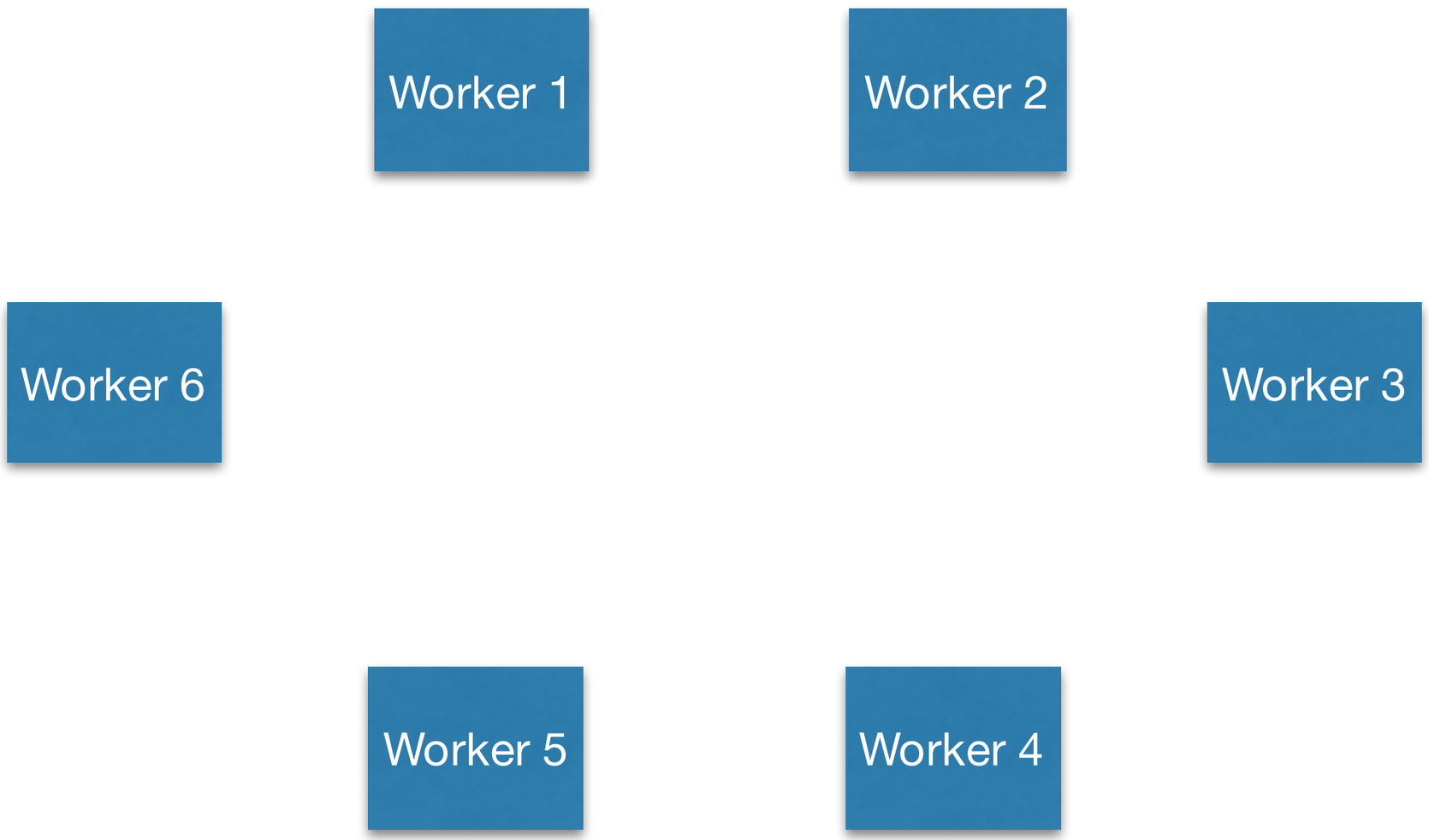
Algorithm 1 Evolution Strategies

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0
for $t = 0, 1, 2, \ldots$ **do**
  Sample $\epsilon_1, \ldots \epsilon_n \sim \mathcal{N}(0, I)$
  Compute returns $F_i = F(\mu_t + \sigma\epsilon_i)$ for $i = 1, \ldots, n$
  Set $\mu_{t+1} \leftarrow \mu_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
end for

Main contribution:

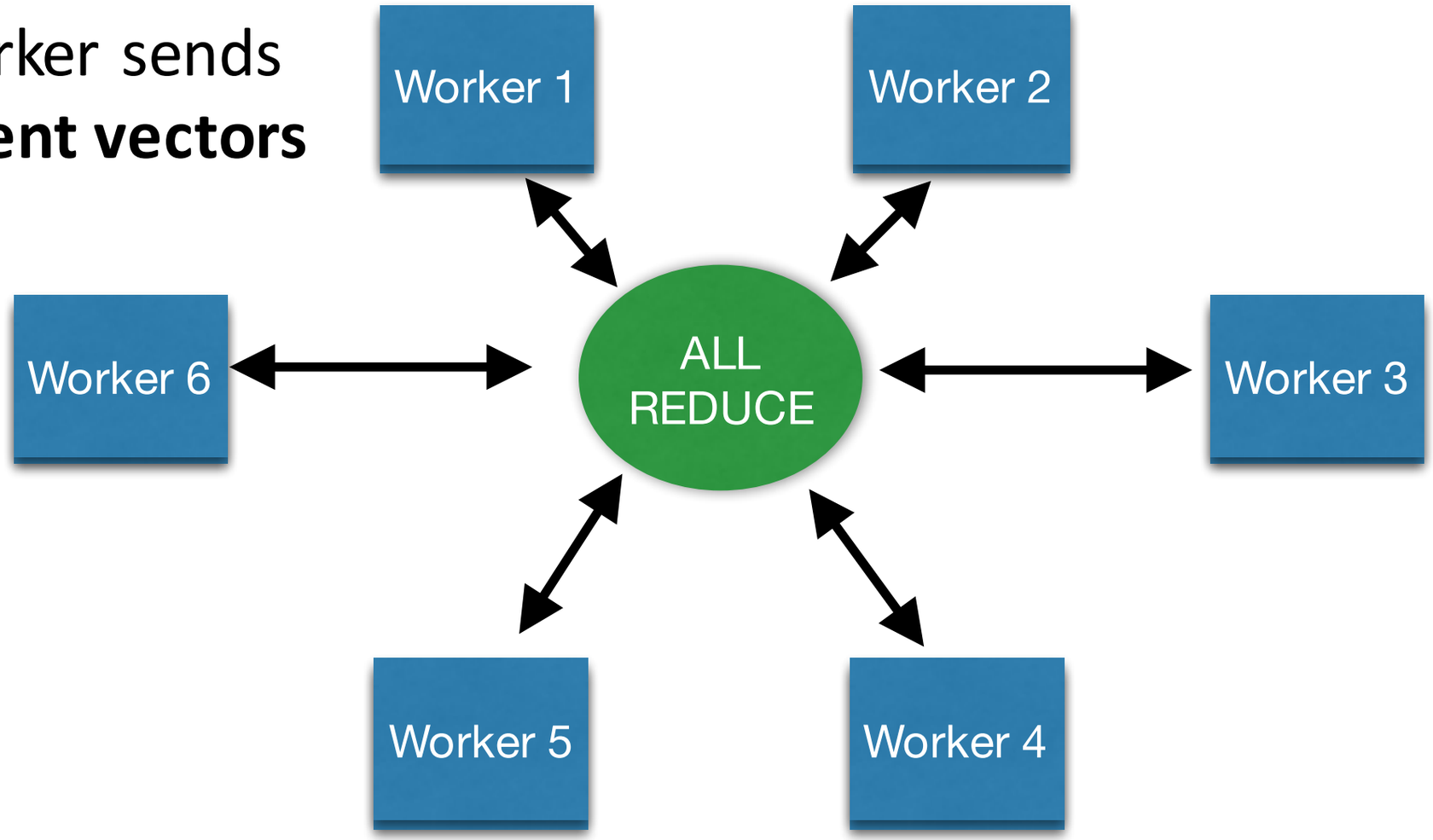
- Parallelization with a need for tiny only cross-worker communication

Distributed Deep Learning

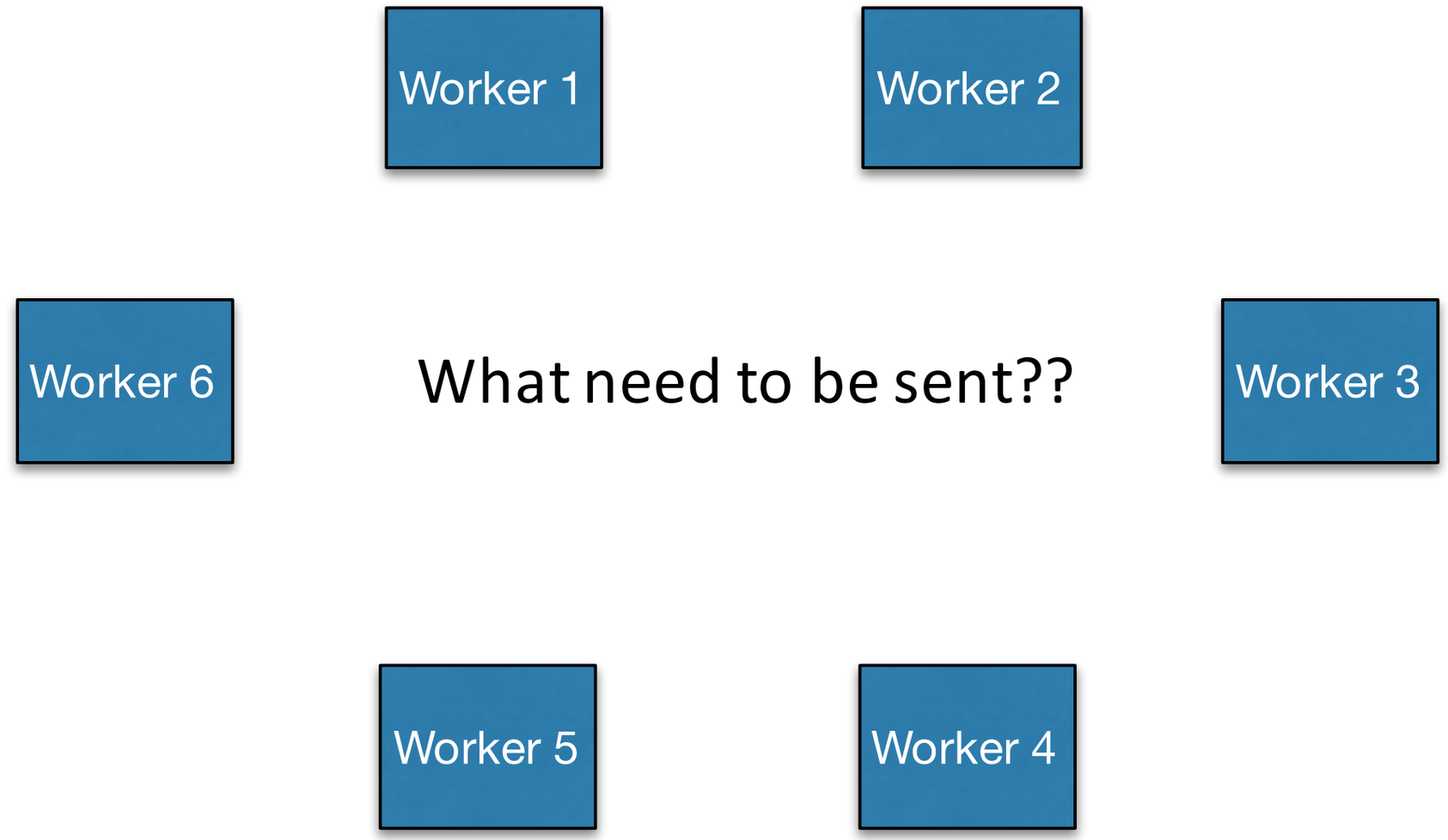


Distributed Deep Learning

Each worker sends
big gradient vectors



Distributed Evolution



Distributed Evolution

Worker 1

Worker 2

Algorithm 1 Evolution Strategies

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0
for $t = 0, 1, 2, \ldots$ **do**
  Sample $\epsilon_1, \ldots \epsilon_n \sim \mathcal{N}(0, I)$
  Compute returns $F_i = F(\mu_t + \sigma\epsilon_i)$ for $i = 1, \ldots, n$
  Set $\mu_{t+1} \leftarrow \mu_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} F_i \epsilon_i$
end for

Worker 5

Worker 4

Distributed Evolution

Worker 1

Worker 2

Worker 6

θ and R(τ)?

Worker 3

θ is big!

but $\theta = \mu + \sigma\epsilon$

Same for all workers

Only need seed of random number generator!

Distributed Evolution

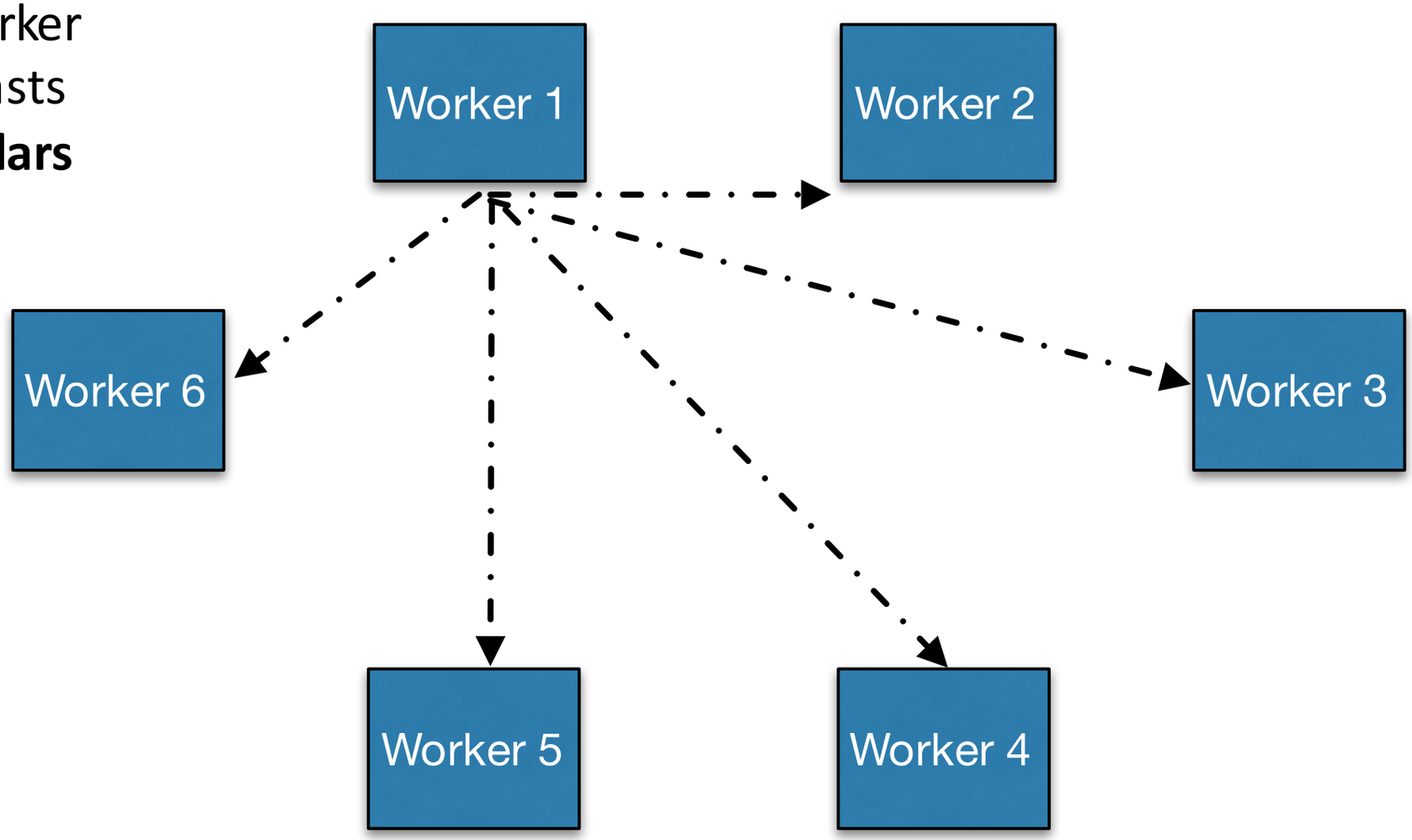
Algorithm 2 Parallelized Evolution Strategies

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0
Initialize: n workers with known random seeds, and initial parameters θ_0
for $t = 0, 1, 2, \ldots$ **do**
  for each worker $i = 1, \ldots, n$ **do**
    Sample $\epsilon_i \sim \mathcal{N}(0, I)$
    Compute returns $F_i = F(\mu_t \vdash \sigma\epsilon_i)$
  end for
  Send all scalar returns F_i from each worker to every other worker
  for each worker $i = 1, \ldots, n$ **do**
    Reconstruct all perturbations ϵ_j for $j = 1, \ldots, n$
    Set $\mu_{t+1} \leftarrow \mu_t + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} F_j \epsilon_j$
  end for
end for

[Salimans, Ho, Chen, Sutskever, 2017]

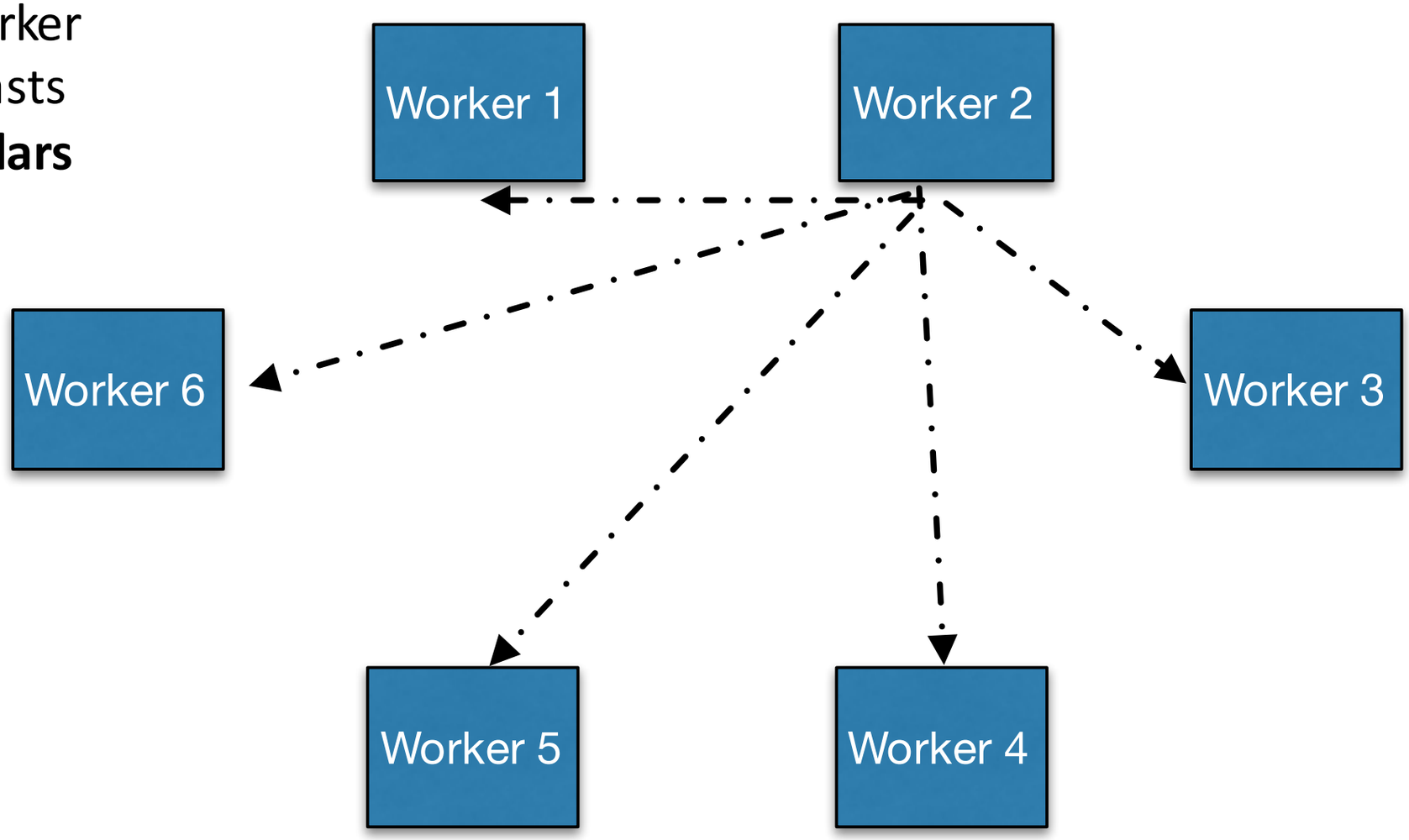
Distributed Evolution

Each worker broadcasts **tiny scalars**



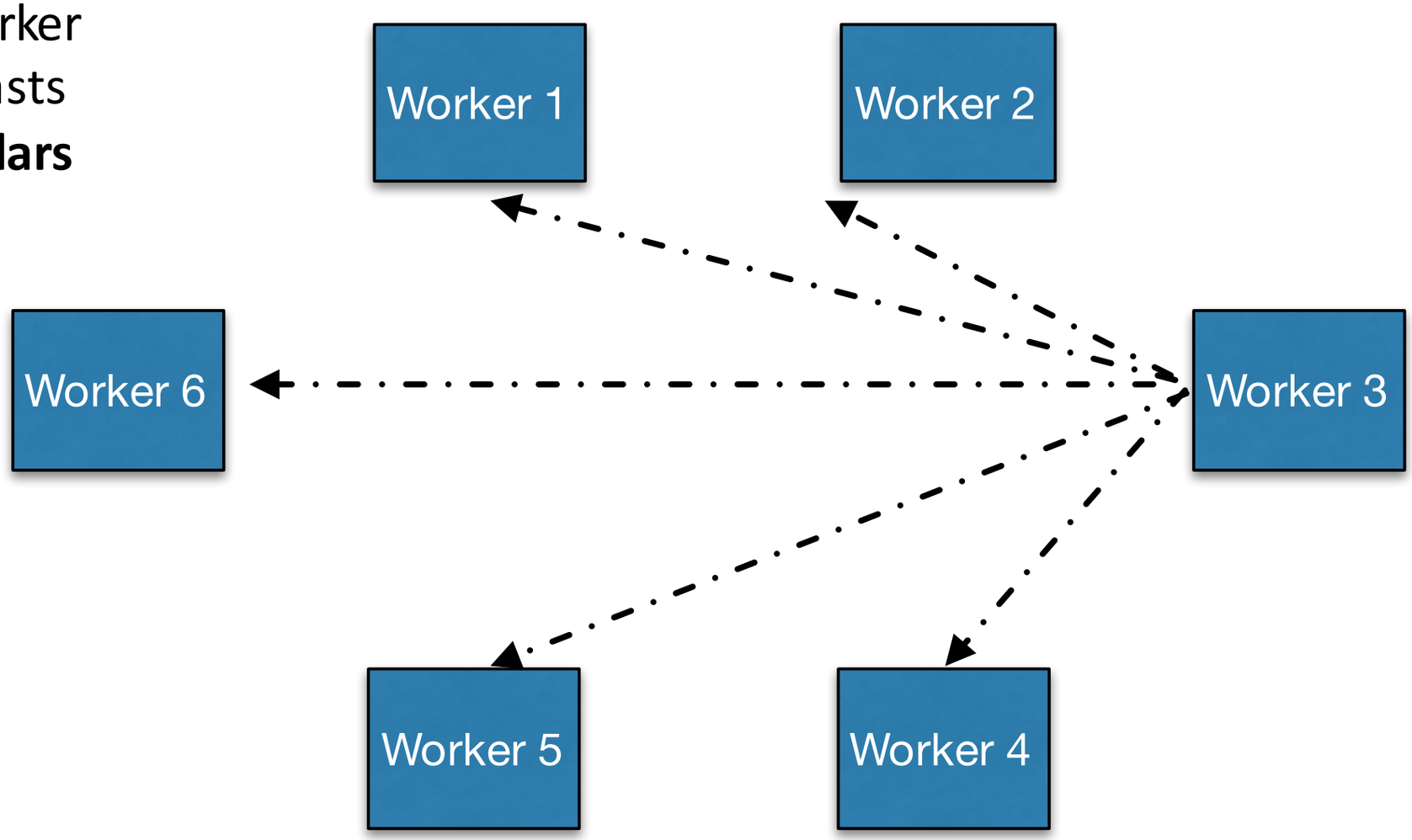
Distributed Evolution

Each worker broadcasts **tiny scalars**



Distributed Evolution

Each worker broadcasts **tiny scalars**



Distributed Evolution Scales Very Well :-)

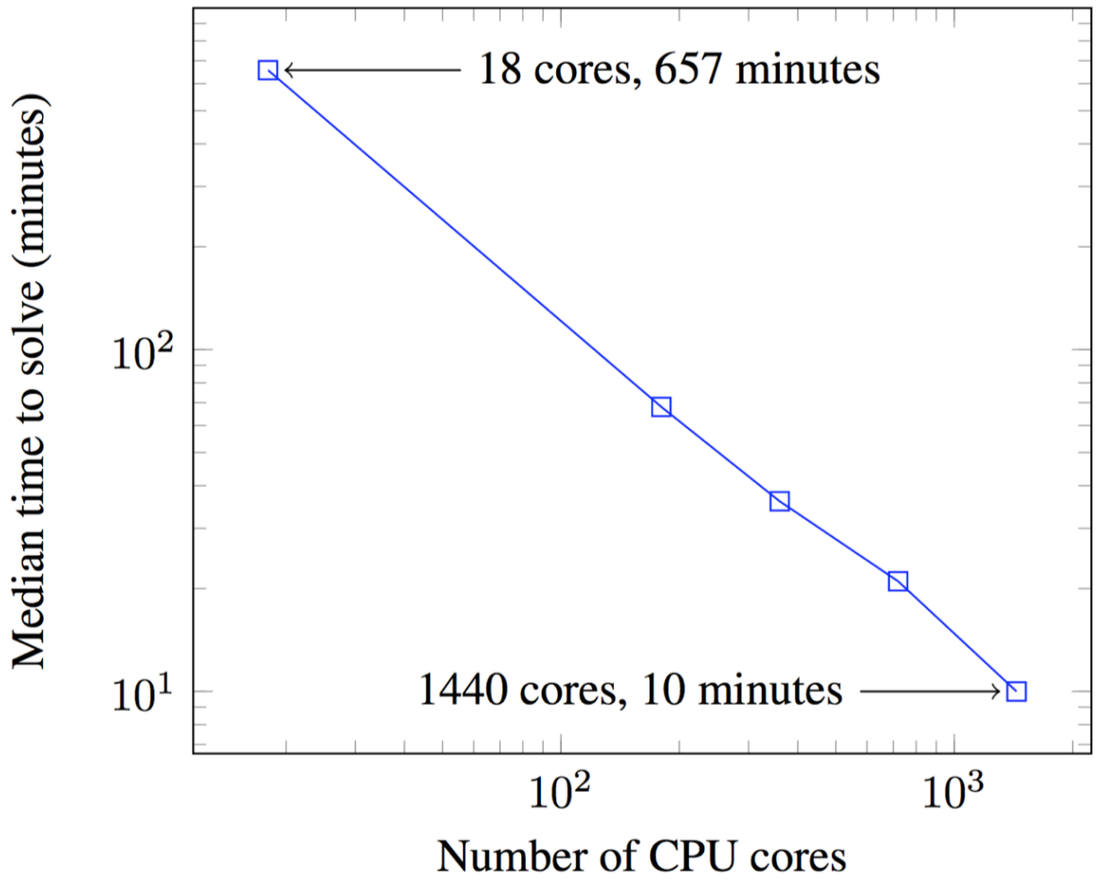
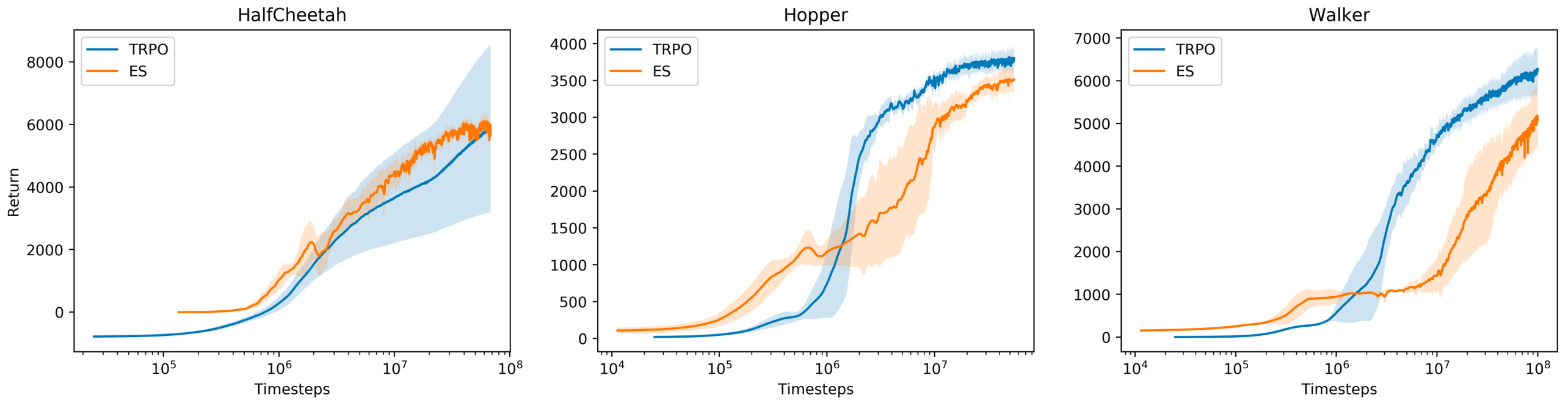


Figure 1. Time to reach a score of 6000 on 3D Humanoid with different number of CPU cores. Experiments are repeated 7 times and median time is reported.

[Salimans, Ho, Chen, Sutskever, 2017]

Distributed Evolution Requires More Samples :-(



[Salimans, Ho, Chen, Sutskever, 2017]

Natural Evolution beyond RL

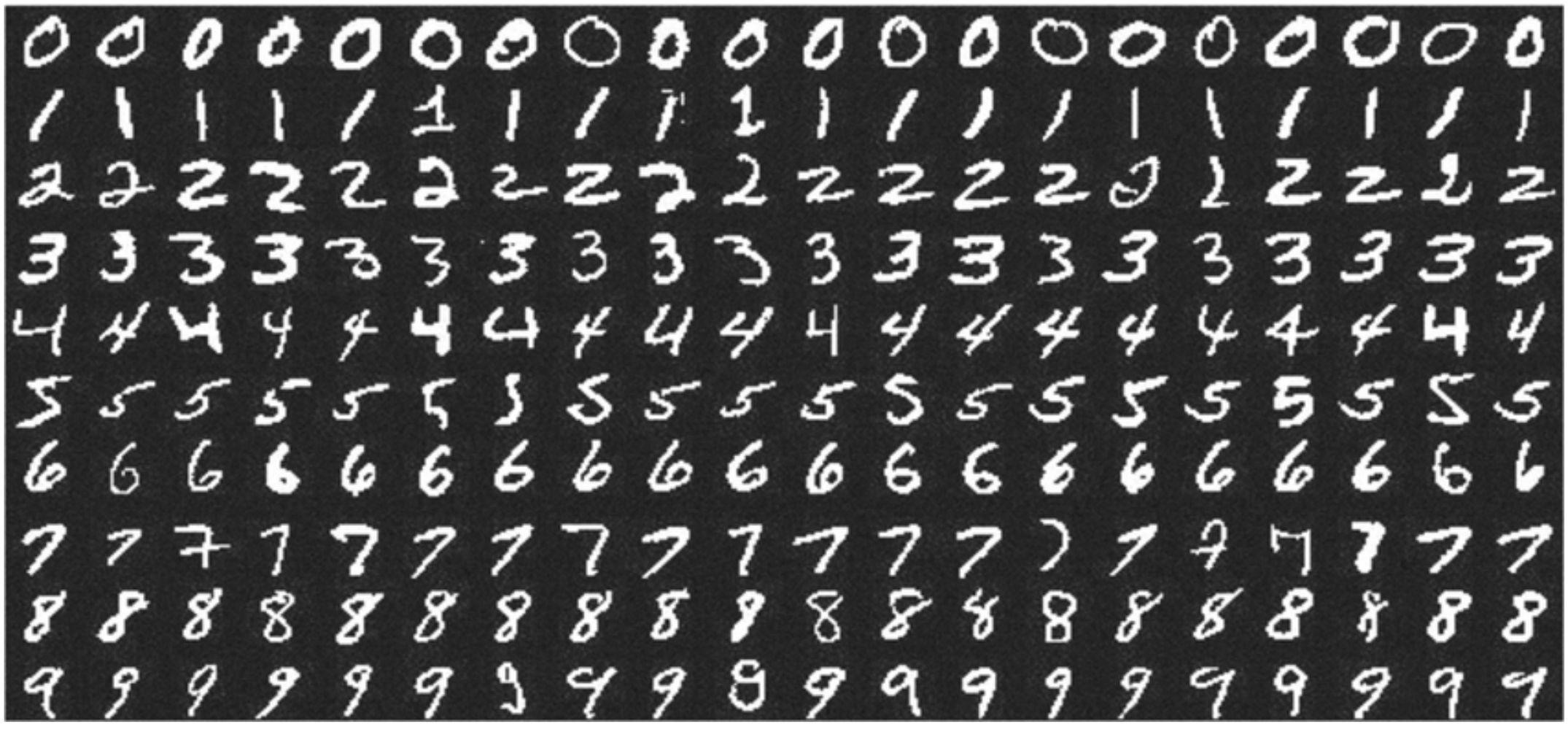
- Q: Why we do not use evolutionary methods for training image classifiers?
- A: In this case, the gradient is available: we have a direction to take a step in our parameter space, (in high dimensionas SGD is not easily trapped in local minima). In low dim non-convex problems, potentially such methods would also make sense, despite the fact we can differentiate.

Comparing ES with SGD in MNIST

On the Relationship Between the OpenAI Evolution Strategy and Stochastic Gradient Descent

By: Xingwen Zhang, Jeff Clune, Kenneth O. Stanley (Uber AI labs)

Slides copied from William Saunders, UoT



Parameter Update for ES

- Generate n pseudo-offspring by adding perturbation v_i sampled from uniform Gaussian with mean 0, covariance I $\theta + \sigma v_i$
- Compute rewards for each offspring (r_i)
- Estimate gradient

$$g^{ES} = \frac{1}{n\sigma} \sum_{i=1}^{n} v_i r_i$$

- Update parameters using gradient estimate with any optimizer (ie. ADAM)

Comparing ES with SGD in MNIST

- We can use ES gradient as an inefficient way to train a classification network on MNIST, to compare it's performance to the SGD gradient
- We can compute the correlation between the SGD gradient and ES gradient for each minibatch
- Initially achieved 96.99% validation accuracy with 10,000 pseudo-offspring, 2000 iterations on a network with 3,274,634 parameters

Gradient Correlation

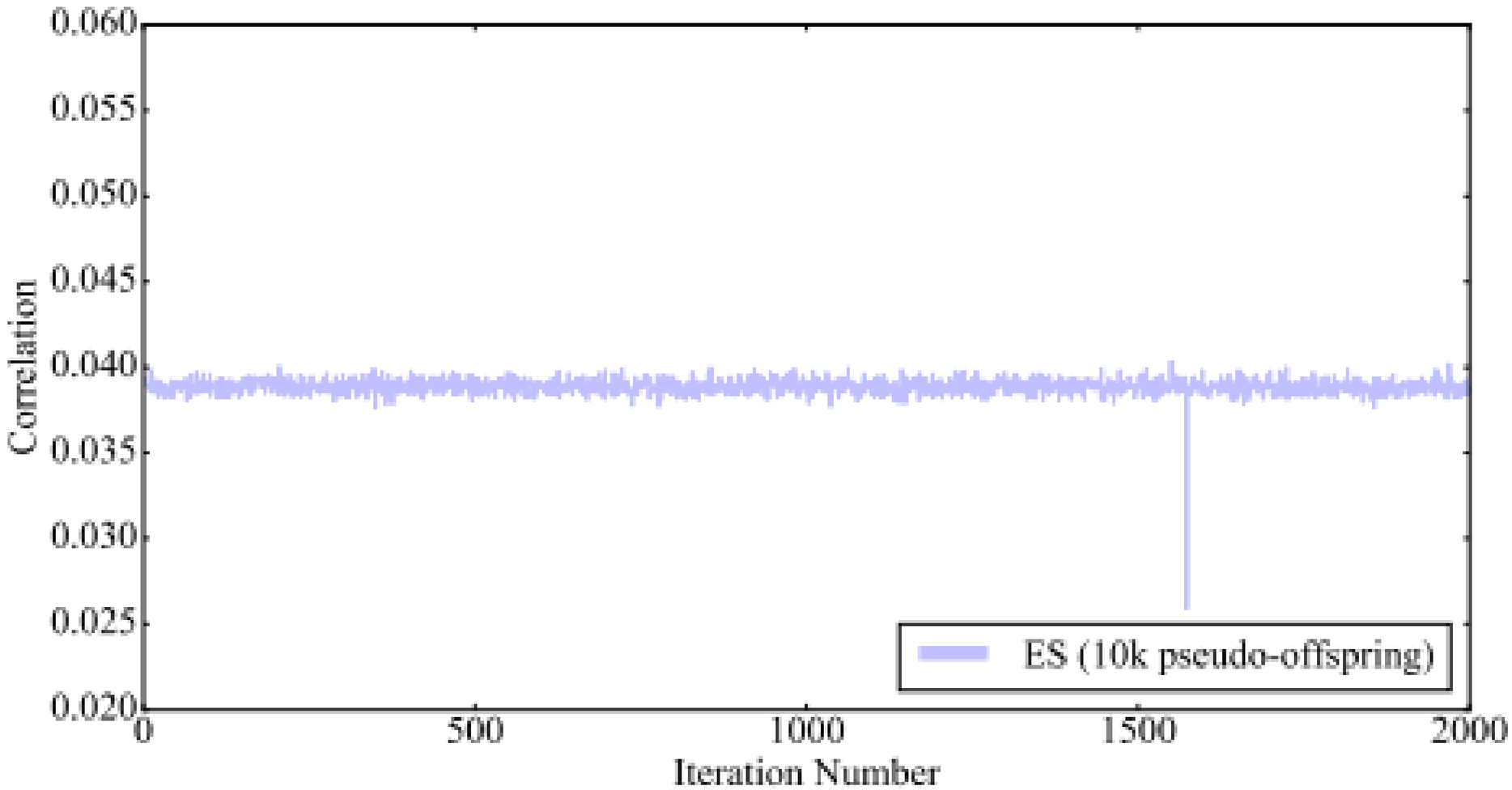


Figure 1: **Correlation of the gradients estimated by ES and the analytic gradients for the same sequence of mini-batches**. The correlation between ES and gradients used by SGD is remarkably stable.

Noisy SGD is a Good Proxy for Natural Evolution

- It's remarkable that the 3.9% correlation allows for validation accuracy within 1.7% of SGD accuracy
- The correlation between ES and SGD gradient is stable over time
- We can simulate ES gradient by adding uniform noise to SGD gradient to achieve the same correlation to the SGD gradient
- It's more efficient to run experiments on algorithm changes with this SGD + noise proxy

Noisy SGD is a Good Proxy for Natural Evolution

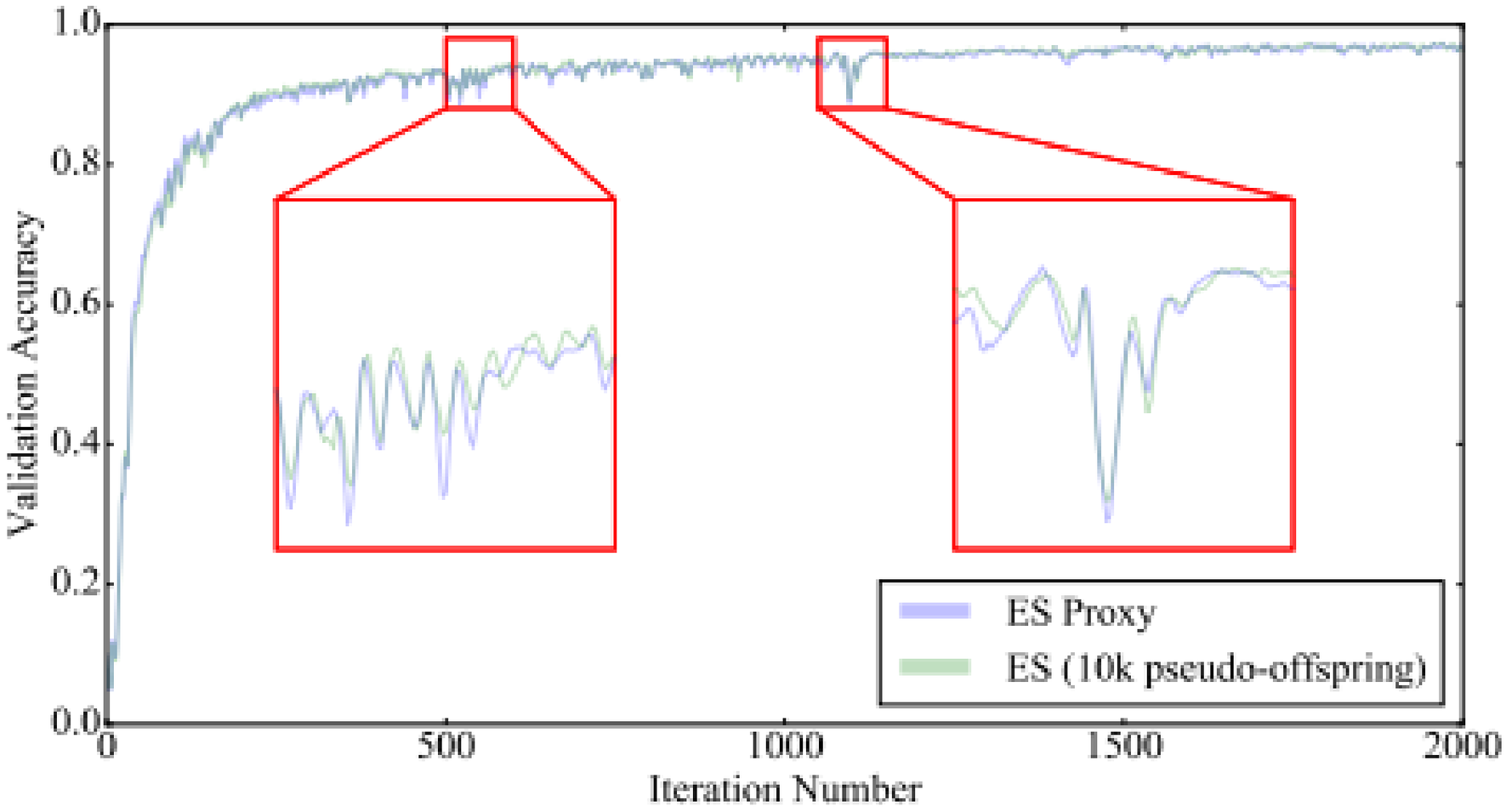


Figure 2: **Validation accuracy of ES and the ES proxy for the same sequence of mini-batches.** Notice how the fluctuations in performance by ES and its proxy are nearly identical throughout the run despite the randomness in the SGD proxy and the ES pseudo-offspring. Insets show local areas of the curves close up. The implication is that mini-batches are the primary driver of noise in the search, impacting ES and SGD in the same way. Recall that validation accuracy over iterations in this figure and throughout this paper is reported on the entire MNIST test set.

MNIST with Neuro-Evolution

- Limited the number of dimensions that are perturbed for each pseudo-offspring, improving performance but requiring more pseudo-offspring
- Achieve 99% accuracy on MNIST using 50,000 pseudo-offspring, 10,000 training batches
- But used a smaller network (2 layers, 28938 parameters)

Conclusions

- Evolutionary strategies are competitive to SGD since the gradient is in any case noisy in RL problems
- When optimizing differentiable objectives with deep nets, they do not have much to affer: much more expensive and more noisy than the true gradient.
- ES performance much depends on the number of workers for high dimensional problems.
- Smart trick to avoid a lot of communication between workers by sending scalar rewards and sharing the random seeds. Easier to scale up.
- We should always make sure our method beats this obvious baseline: NES with reasonable (not outrageous) resource allocation.