Carnegie Mellon
School of Computer Science

Deep Reinforcement Learning and Control

Episodic Memory and RL
Fast and Slow RL

Katerina Fragkiadaki

Fast and Slow learning

By training a neural network (NN) to emulate the observation-label pairs in a dataset, e.g., pairs of states with their state values, or state/actions with their Q values, learning is necessarily slow: it requires many iterations for a NN to learn the mapping. We cannot increase the learning rate, else we won't be able to either converge or we will overwrite previously learnt data.

- Conclusion: **Deep learning of policies and value functions is necessarily slow**.
- Consequence: DeepRL is sample inefficient, and policies do not quickly adapt to novel conditions: **it takes time from the interaction experienced to the knowledge acquired from the interaction to be reflected in the NN weights**. It is hard to balance interaction collection with gradient updates.

Fast and Slow learning

In contrast, **humans can learn fast from their experiences as well as adapt fast their strategies based on their experiences.**

Examples:

- When you touch a hot stove, you immediately learn (and remember) that you should not touch that stove again.
- When a new iPhone comes out, you use your past experience with other iPhones to accelerate your learning to use the new one (you can figure it out after few clicks).
- When your ankle breaks during soccer, you quickly figure out how to walk to go to the side of the field.

Naive Baseline: Policy adaptation via Finetuning

- When a new iPhone comes out, you use your past experience with other iPhones to accelerate your learning to use the new one (you can figure it out after few clicks).
- Given a pertained generalized policy on operating iPhone, i fine-tune the policy and Q function using few experience trajectories collected on the new i-Phone.
- This usually does not work: *we overfit on the recent interactions, forget old data*. Fine-tune weights would require lots of interactions, and would usually result in forgetting knowledge from previous interactions. (except if we carefully augment the neural architecture, e.g., ``progressive networks", but still we would need way more interaction data).

Fast and Slow learning

To emulate such fast human learning from experience, we would potentially need computational **memory structures beyond neural weights**, or we would need to find better ways to **adapt/finetune neural weights with little data**.

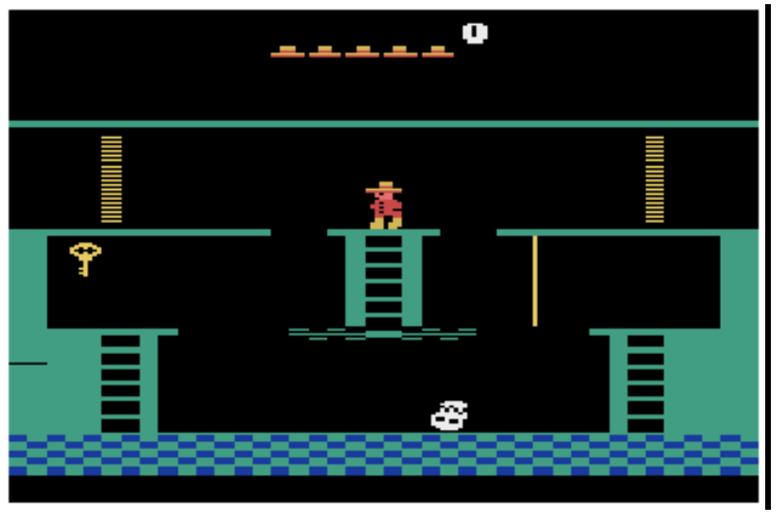
Such fast learning will help us

1. explore better
2. solve the task faster.

Episodic Memory and RL

Under sparse rewards, if K precise actions are needed to discover the first reward, finding that first reward requires time exponential to the number of actions needed:

$$p(\text{get first key}) = p(\text{get down ladder 1}) * p(\text{get down rope}) * p(\text{get down ladder 2}) * p(\text{jump over skull}) * p(\text{get up ladder 3}).$$



Learning Montezuma's Revenge from a Single Demonstration, Salimans and Chen

Intrinsic motivation methods seek not rewards but novel states or transitions, and make more progress in sparse reward setups.

However, **lack of a structured memory of states and their connectivity does not permit organized way of exploring the environment**. Such a cognitive map would permit fast assimilating the results of exploration in guiding future exploration.

Solutions

Episodic memory structures: non-parametric book-keeping of states, their (Monte Carlo or bootstrapped) rewards, state-to-state connectivity trajectories.

Recurrent policies that learn from a series of attempts/episodes.

Neural weight initialization that permits updating weights with very few gradient steps.

Mixture of policies / map of policies (skills): fast pick the right one with few trials.

Solutions

Episodic memory / cognitive mapping structures: non-parametric book-keeping of states, their future cumulative rewards, state-to-state connectivity structure, and so on.

Recurrent policies that learn from a series of attempts/episodes.

Neural weight initialization that permits updating weights with very few gradient steps.

Mixture of policies / map of policies (skills): fast pick the right one with few trials.

Episodic Memory

Episodic memory is the memory of autobiographical events (times, places, associated emotions, and other contextual who, what, when, where, why knowledge) that can be explicitly stated or conjured. It is the collection of past personal experiences that occurred at a particular time and place. For example, if one remembers the party on their 7th birthday, this is an episodic memory. They allow an individual to figuratively travel back in time to remember the event that took place at that particular time and place

Events that are recorded into episodic memory may trigger episodic learning, i.e. a change in behavior that occurs as a result of an event. For example, a fear of dogs after being bitten by a dog is a result of episodic learning.

From Wikipedia

Cognitive Map

A **cognitive map** (sometimes called a mental map or mental model) is a type of mental representation which serves an individual to acquire, code, store, recall, and decode information about the relative locations and attributes of phenomena in their everyday or metaphorical spatial environment. The concept was introduced by Edward Tolman in 1948. The concept was used to explain the behavior of rats that appeared to learn the spatial layout of a maze, and subsequently the concept was applied to other animals, including humans.

From Wikipedia

Neural Episodic Control

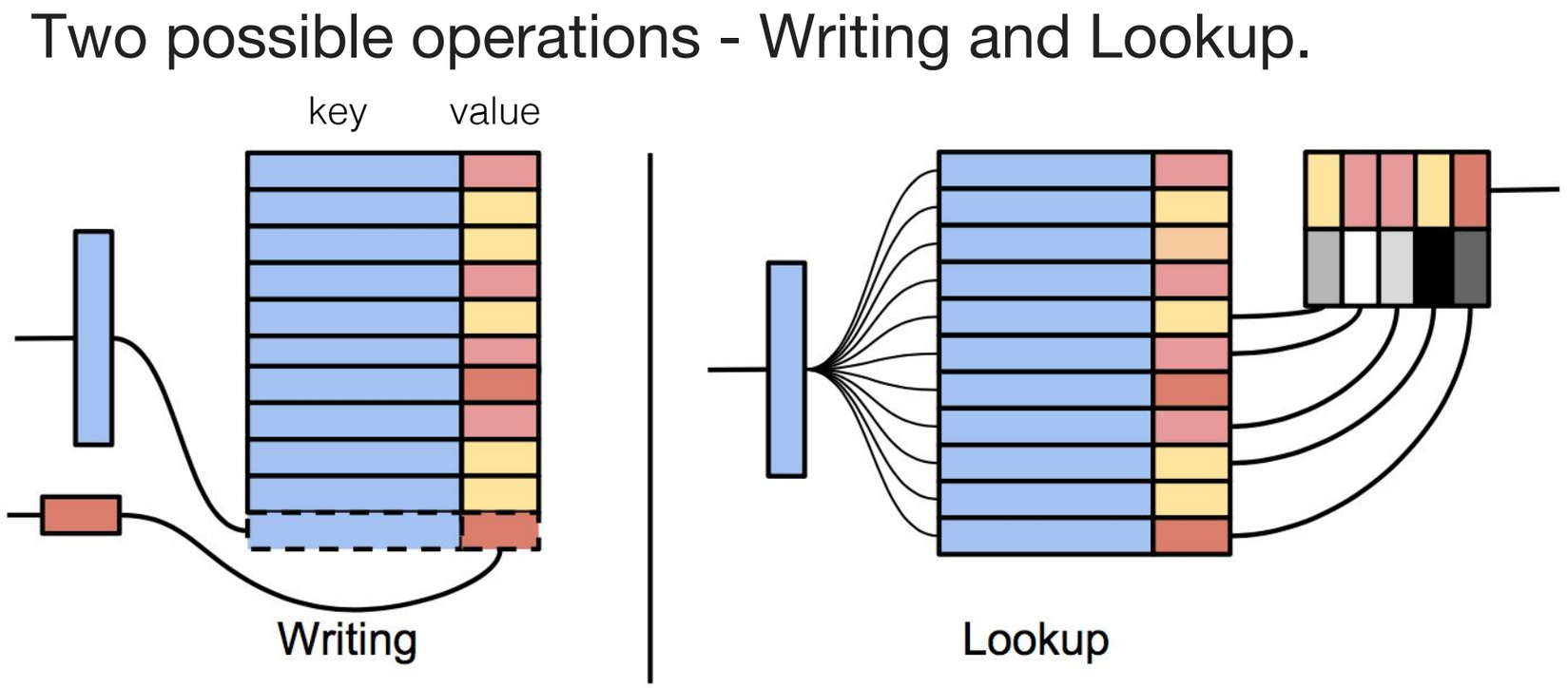
Alexander Pritzel APRITZEL@GOOGLE.COM
Benigno Uria BURIA@GOOGLE.COM
Sriram Srinivasan SRSRINIVASAN@GOOGLE.COM
Adrià Puigdomènech ADRIAP@GOOGLE.COM
Oriol Vinyals VINYALS@GOOGLE.COM
Demis Hassabis DEMISHASSABIS@GOOGLE.COM
Daan Wierstra WIERSTRA@GOOGLE.COM
Charles Blundell CBLUNDELL@GOOGLE.COM
DeepMind, London UK

Episodic memory of state/actions and their Q values that learns to play Atari

Differentiable Neural Dictionary

I have one DND per action (discrete actions)

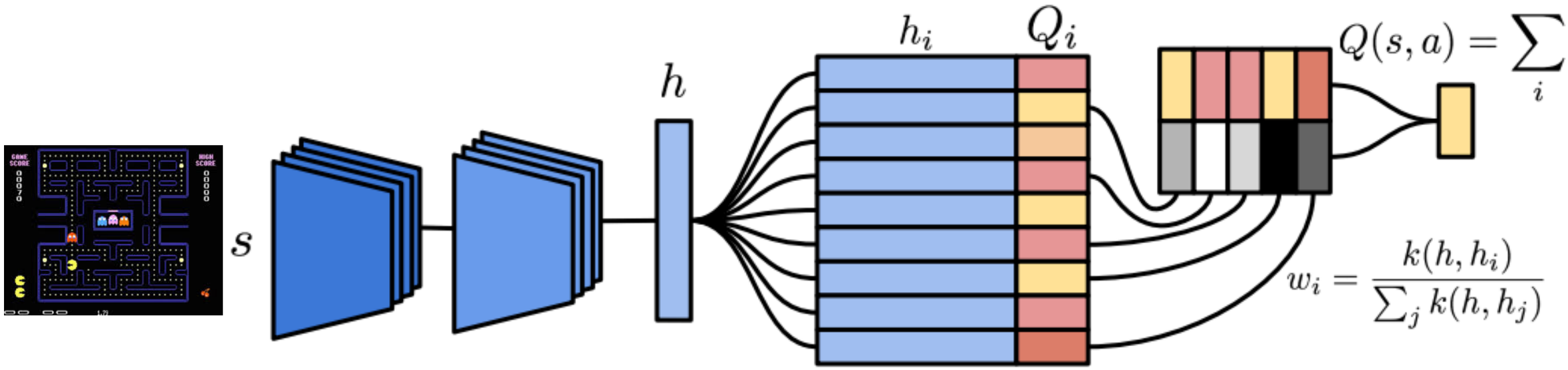
Two possible operations - Writing and Lookup.



Reading from DND

Input: a state s, output: its Q estimate

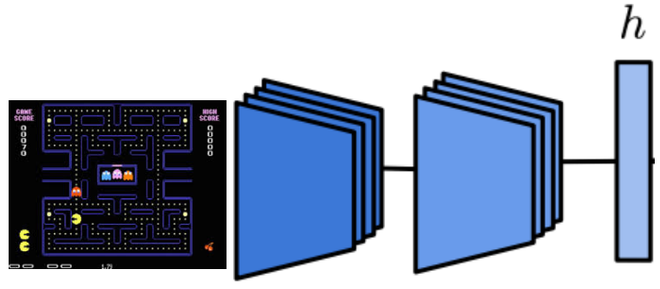
A nearest neighbor look up in the DND.

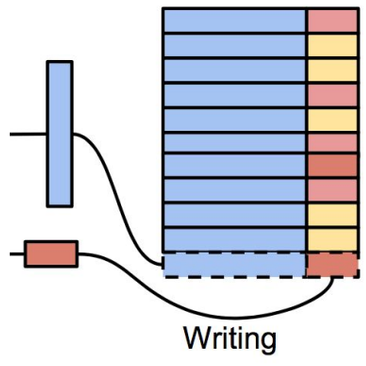


$k(h_i, h_j)$: a similarity kernel

Writing at DND

Embed the state:





Get its Q estimate from the DNDs and instantaneous rewards:

$$Q^{(N)}(s_t, a) = \sum_{j=0}^{N-1} \gamma^j r_{t+j} + \gamma^N \max_{a'} Q(s_{t+N}, a')$$

how is this computed?

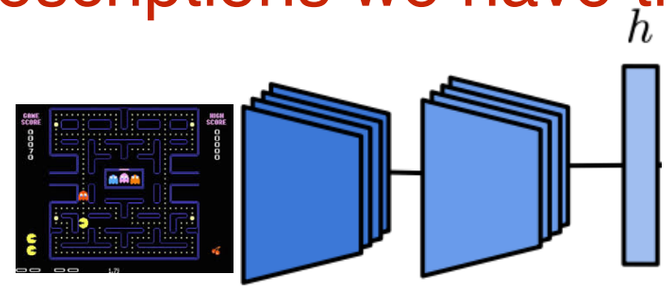
If *identical* key h present, compute: $Q_i \leftarrow Q_i + \alpha(Q^{(N)}(s,a) - Q_i)$

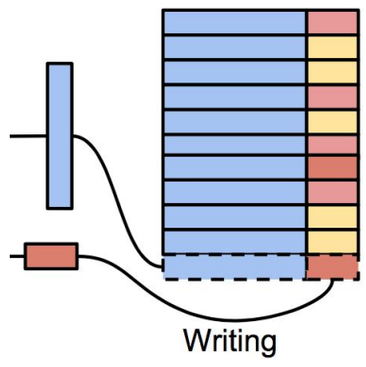
Else add row $\left(h(\phi), Q^N(s,a)\right)$ to the DND

Writing at DND

This reminds tabular methods: we have a long table of states and actions, but instead of raw state descriptions we have their *learnt* embeddings

Embed the state:





Get its Q estimate from the DNDs and instantaneous rewards:

$$Q^{(N)}(s_t, a) = \sum_{j=0}^{N-1} \gamma^j r_{t+j} + \gamma^N \max_{a'} Q(s_{t+N}, a')$$

how is this computed?

If *identical* key h present, compute: $Q_i \leftarrow Q_i + \alpha(Q^{(N)}(s,a) - Q_i)$

Else add row $\left(h(\phi), Q^N(s,a)\right)$ to the DND

Learning the state embeddings

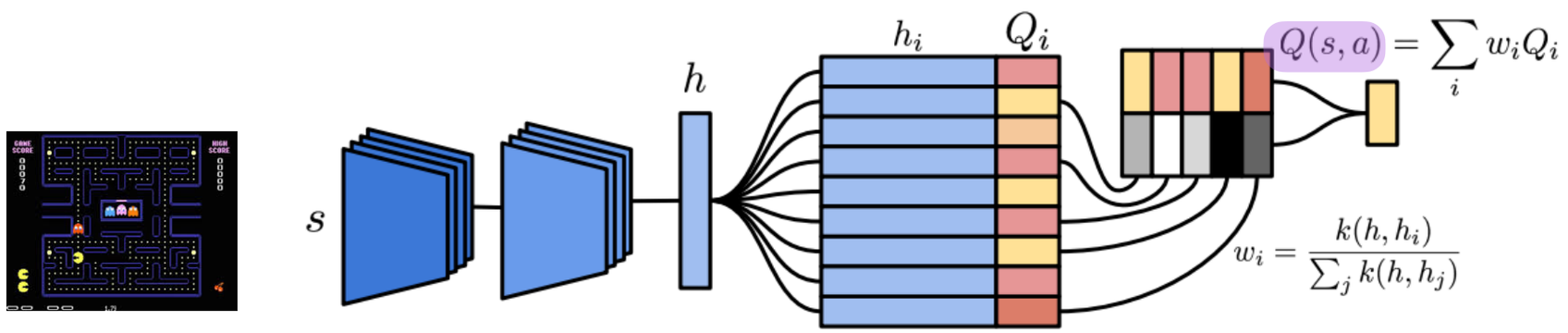
Algorithm 1 Neural Episodic Control

$\mathcal{D}$: replay memory.
M_a: a DND for each action a.
N: horizon for N-step Q estimate.
for each episode **do**
  for $t = 1, 2, \ldots, T$ **do**
    Receive observation s_t from environment with embedding h.
    Estimate $Q(s_t, a)$ for each action a via (1) from M_a
    $a_t \leftarrow \epsilon$-greedy policy based on $Q(s_t, a)$
    Take action a_t, receive reward r_{t+1}
    Append $(h, Q^{(N)}(s_t, a_t))$ to M_{a_t}.
    Append $(s_t, a_t, Q^{(N)}(s_t, a_t))$ to $\mathcal{D}$.
    Train on a random minibatch from $\mathcal{D}$.
  end for
end for

the Q target

loss: $\mathscr{L}(\phi) = \sum_s \left(Q^N(s,a) - Q(s,a)\right)^2$

$$Q^{(N)}(s_t, a) = \sum_{j=0}^{N-1} \gamma^j r_{t+j} + \gamma^N \max_{a'} Q(s_{t+N}, a')$$



Results

We learn faster

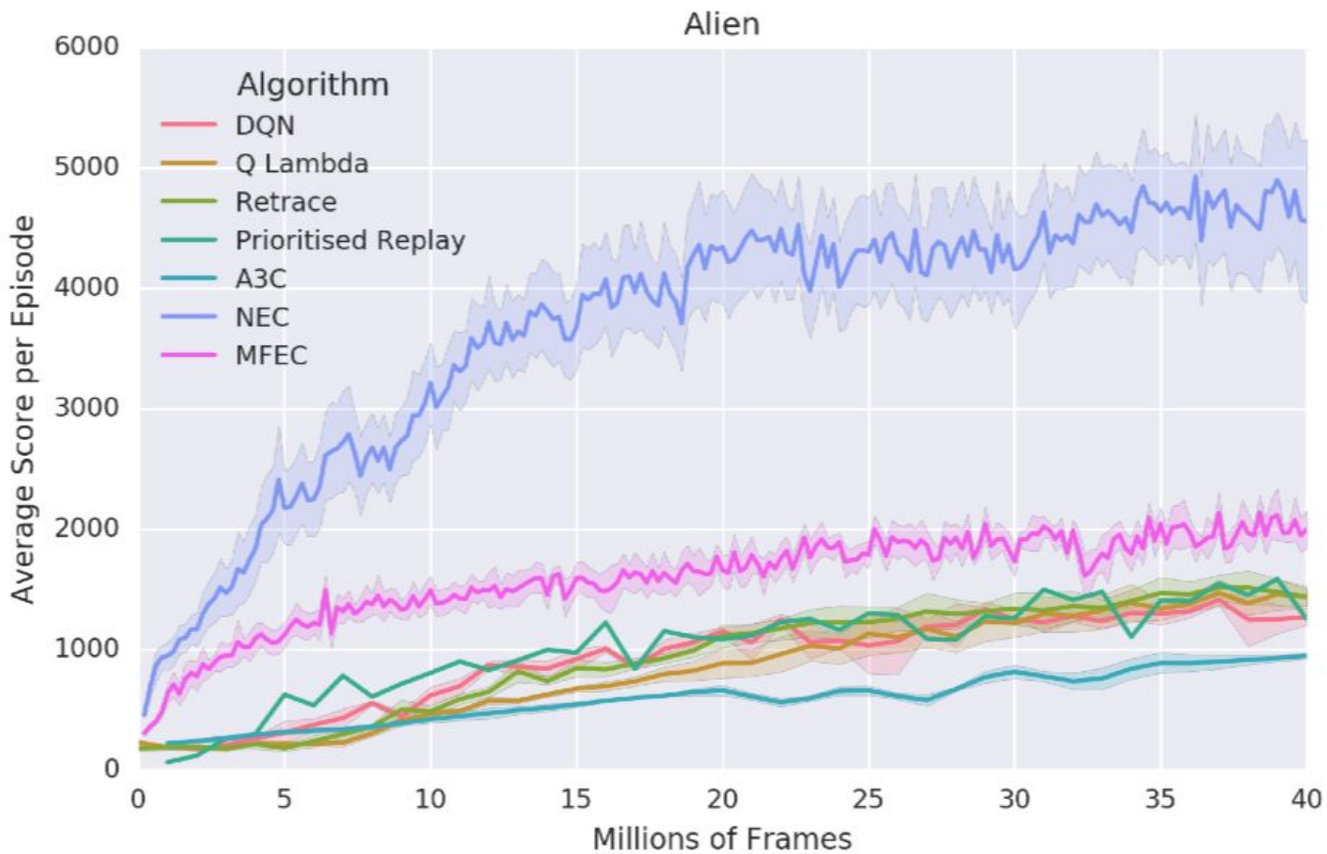
Frames	Nature DQN	Q*(λ)	Retrace(λ)	Prioritised Replay	A3C	NEC	MFEC
1M	-0.7%	-0.8%	-0.4%	-2.4%	0.4%	**16.7%**	12.8%
2M	0.0%	0.1%	0.2%	0.0%	0.9%	**27.8%**	16.7%
4M	2.4%	1.8%	3.3%	2.7%	1.9%	**36.0%**	26.6%
10M	15.7%	13.0%	17.3%	22.4%	3.6%	**54.6%**	45.4%
20M	26.8%	26.9%	30.4%	38.6%	7.9%	**72.0%**	55.9%
40M	52.7%	59.6%	60.5%	**89.0%**	18.4%	83.3%	61.9%

Table 1. Median across games of human-normalised scores for several algorithms at different points in training

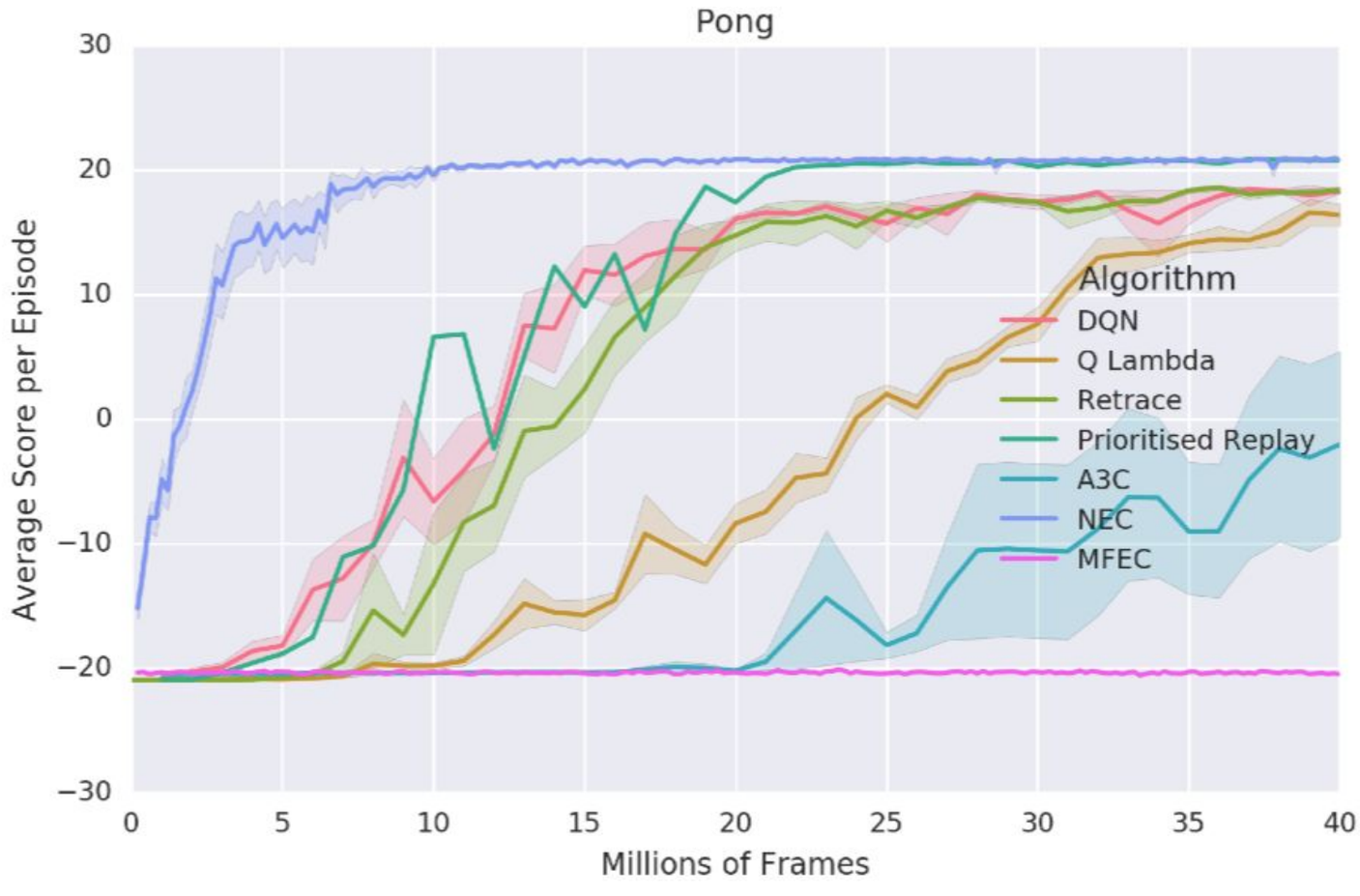
Frames	Nature DQN	Q*(λ)	Retrace(λ)	Prioritised Replay	A3C	NEC	MFEC
1M	-10.5%	-11.7%	-10.5%	-14.4%	5.2%	**45.6%**	28.4%
2M	-5.8%	-7.5%	-5.4%	-5.4%	8.0%	**58.3%**	39.4%
4M	8.8%	6.2%	6.2%	10.2%	11.8%	**73.3%**	53.4%
10M	51.3%	46.3%	52.7%	71.5%	22.3%	**99.8%**	85.0%
20M	94.5%	135.4%	**273.7%**	165.2%	59.7%	121.5%	113.6%
40M	151.2%	**440.9%**	386.5%	332.3%	255.4%	144.8%	142.2%

Table 2. Mean human-normalised scores for several algorithms at different points in training

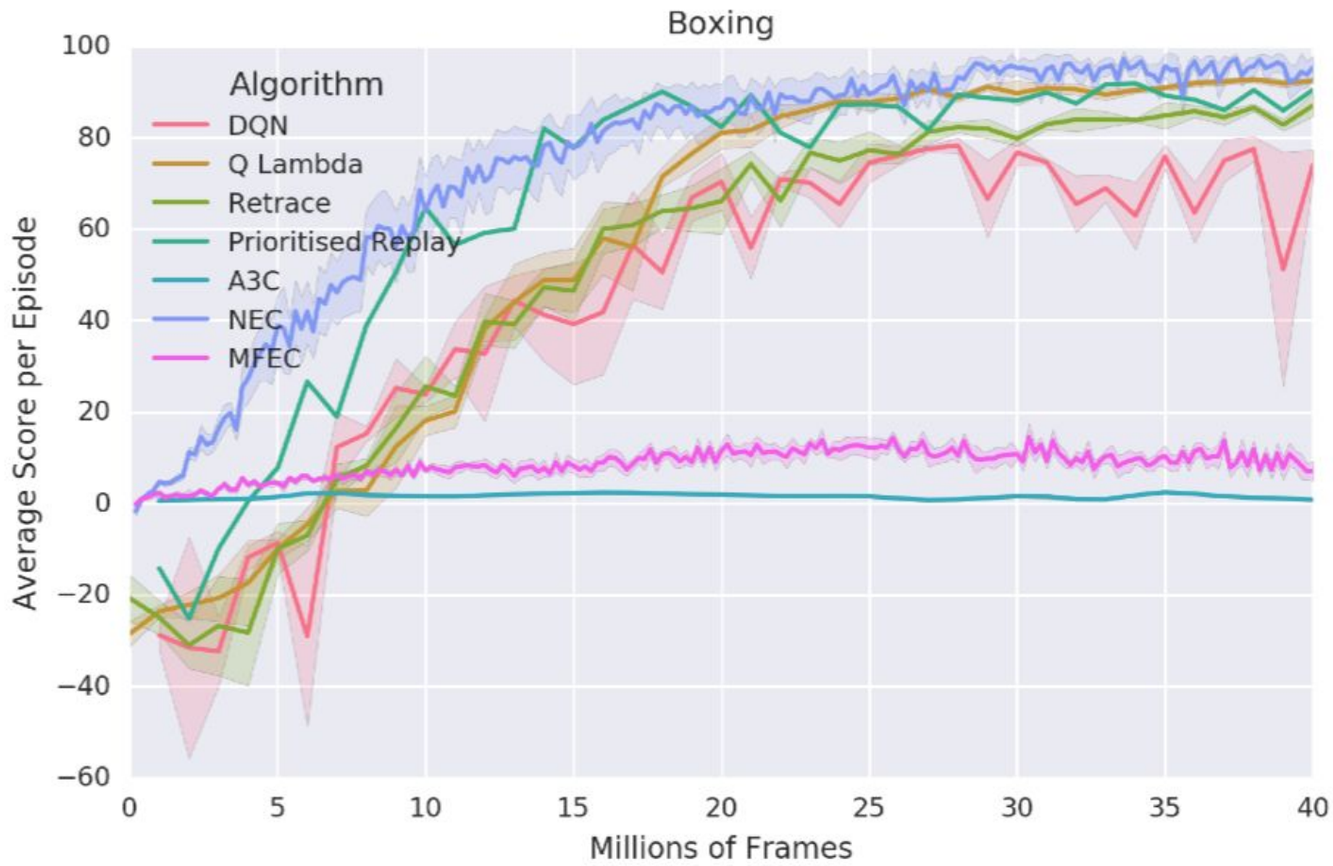
NEC - results


Alien
Algorithm
DQN
Q Lambda
Retrace
Prioritised Replay
A3C
NEC
MFEC
Average Score per Episode
0
1000
2000
3000
4000
5000
6000
0
5
10
15
20
25
30
35
40
Millions of Frames

NEC - results


Pong
30
20
10
0
−10
−20
−30
0
5
10
15
20
25
30
35
40
Average Score per Episode
Millions of Frames
Algorithm
DQN
Q Lambda
Retrace
Prioritised Replay
A3C
NEC
MFEC

NEC - results


Boxing
Algorithm
DQN
Q Lambda
Retrace
Prioritised Replay
A3C
NEC
MFEC
Average Score per Episode
100
80
60
40
20
0
−20
−40
−60
0
5
10
15
20
25
30
35
40
Millions of Frames

Go-Explore: a New Approach for Hard-Exploration Problems

Adrien Ecoffet **Joost Huizinga** **Joel Lehman** **Kenneth O. Stanley*** **Jeff Clune***
Uber AI Labs
San Francisco, CA 94103
adrienle,jhuizinga,joel.lehman,kstanley,jeffclune@uber.com
*Co-senior authors

Structured memory as state connectivity map

Go-Explore: a New Approach for Hard-Exploration Problems

Adrien Ecoffet **Joost Huizinga** **Joel Lehman** **Kenneth O. Stanley*** **Jeff Clune***
Uber AI Labs
San Francisco, CA 94103
adrienle,jhuizinga,joel.lehman,kstanley,jeffclune@uber.com
*Co-senior authors

Book-keep an archive of trajectories that—accidentally through exploration—reached a particular state in the world and the state reached.

We will be updating the archive with:

- New states reached and the corresponding trajectories
- Better (shorter) trajectories to reach already archived states
- We will be resetting ourselves to such states and continue exploring updating our state connectivity map.

Then, we will turn such single trajectories into robust policies.

Learning from trajectories and rewards

Algorithm 1 Demonstration-Initialized Rollout Worker

Input: a human demonstration $\{(\tilde{s}_t, \tilde{a}_t, \tilde{r}_t, \tilde{s}_{t+1}, \tilde{d}_t)\}_{t=0}^{T}$, number of starting points D, effective RNN memory length K, batch rollout length L.
Initialize starting point τ^* by sampling uniformly from $\{\tau - D, \ldots, \tau\}$
Initialize environment to demonstration state $\tilde{s}_{\tau^*}$
Initialize time counter $i = \tau^* - K$
while TRUE **do**
    Get latest policy $\pi(\theta)$ from optimizer
    Get latest reset point τ from optimizer
    Initialize success counter $W = 0$
    Initialize batch $\mathcal{D} = \{\}$
    for step in $0, \ldots, L-1$ **do**
        if $i \geq \tau^*$ **then**
            Sample action $a_i \sim \pi(s_i, \theta)$
            Take action a_i in the environment
            Receive reward r_i, next state s_{i+1} and done signal d_{i+1}
            $m_i =$ TRUE ▷ We can train on this data
        else ▷ Replay demonstration to initialize RNN state of policy
            Copy data from demonstration $a_i = \tilde{a}_i, r_i = \tilde{r}_i, s_{i+1} = \tilde{s}_{i+1}, d_i = d_i$.
            $m_i =$ FALSE ▷ We should mask out this transition in training
        end if
        Add data $\{s_i, a_i, r_i, s_{i+1}, d_i, m_i\}$ to batch $\mathcal{D}$
        Increment time counter $i \leftarrow i + 1$
        if $d_i =$ TRUE **then**
            if $\sum_{t \geq \tau^*} r_t \geq \sum_{t \geq \tau^*} \tilde{r}_t$ **then** ▷ As good as demo
                $W \leftarrow W + 1$
            end if
            Sample next starting starting point τ^* uniformly from $\{\tau - D, ..., \tau\}$
            Set time counter $i \leftarrow \tau^* - K$
            Reset environment to state $\tilde{s}_{\tau^*}$
        end if
    end for
    Send batch $\mathcal{D}$ and counter W to optimizer
end while

Algorithm 2 Optimizer

Input: number of parallel agents M, starting point shift size Δ, success threshold ρ, initial parameters θ_0, demonstration length T, learning algorithm $\mathcal{A}$ (e.g. PPO, A3C, Impala, etc.)
Set the reset point $\tau = T$ to the end of the demonstration
Start rollout workers $i = 0, \ldots, M-1$
while $\tau > 0$ **do**
    Gather data $\mathcal{D} = \{\mathcal{D}_0, \ldots, \mathcal{D}_{M-1}\}$ from rollout workers
    if $\sum_{\mathcal{D}}[W_i] / \sum_{\mathcal{D}}[d_{i,t}] \geq \rho$ **then** ▷ The workers are successful sufficiently often
        $\tau \leftarrow \tau - \Delta$
    end if
    $\theta \leftarrow \mathcal{A}(\theta, \mathcal{D})$ ▷ Make sure to mask out demo transitions
    Broadcast θ, τ to rollout workers
end while

Reset time point

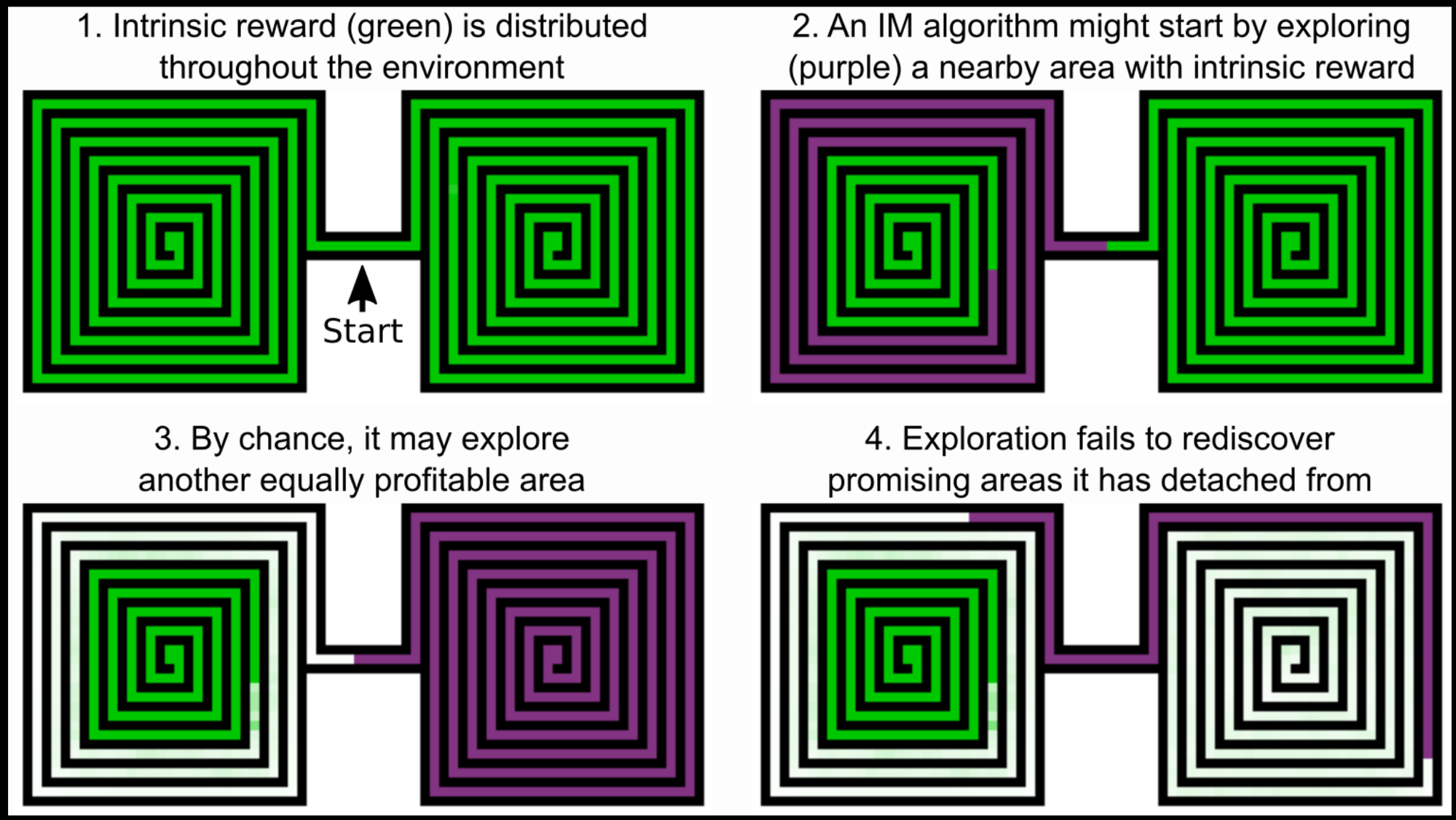
Solve the subtask from τ till T

Learning Montezuma's Revenge from a Single Demonstration, Salimans and Chen

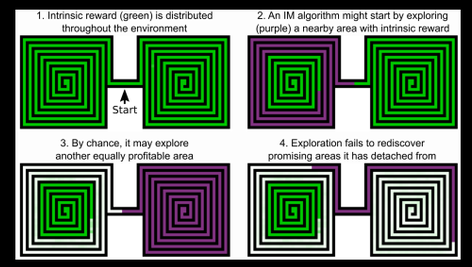
Intrinsic Motivation

- Helps, but why doesn’t it work better?

“Detachment”


1. Intrinsic reward (green) is distributed throughout the environment
Start
2. An IM algorithm might start by exploring (purple) a nearby area with intrinsic reward
3. By chance, it may explore another equally profitable area
4. Exploration fails to rediscover promising areas it has detached from

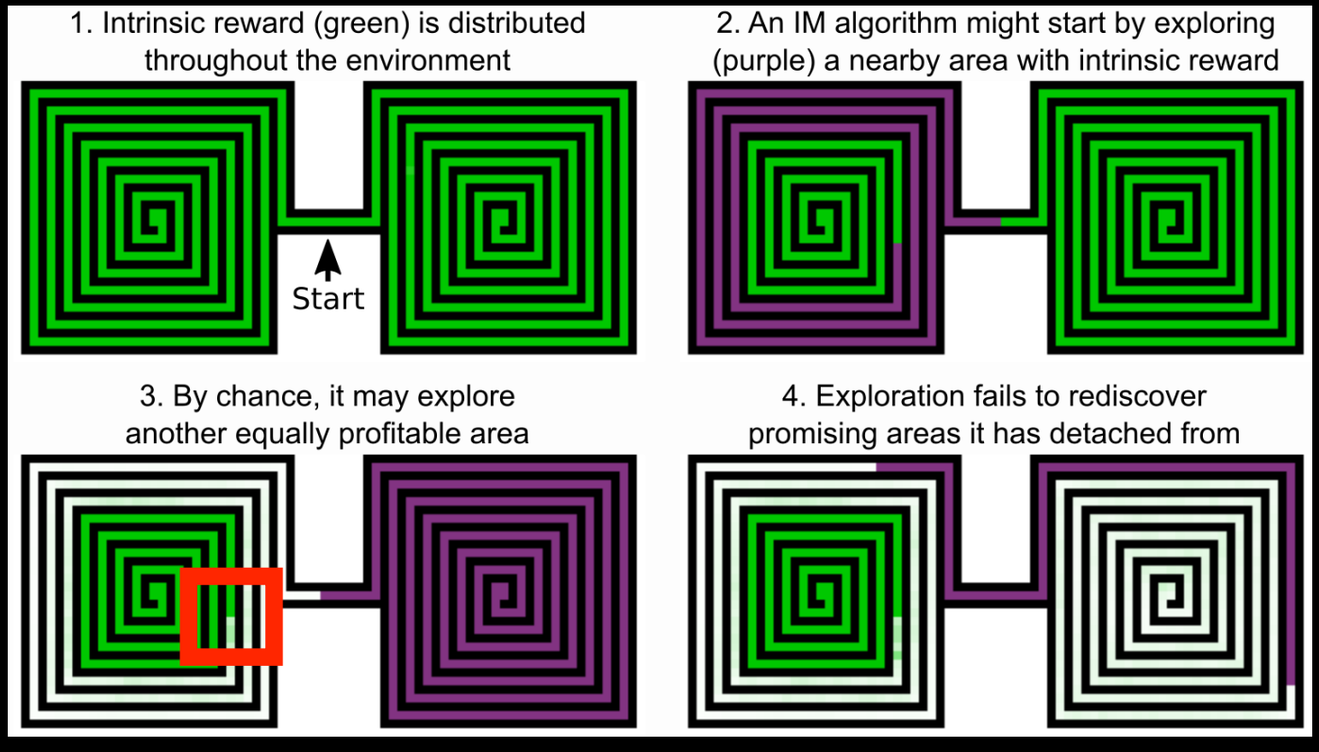
Detachment



- Replay buffers
  - Should remember in theory, but forget/fail in practice
    - replay buffer size must be very large
    - but that causes optimization/stability issues

Detachment

- Proposal: explicitly remember
  - where promising locations are
  - how to get back to them



“Derailment”

- Most RL algorithms:
  - take promising policy, perturb it, hope it explores further
  - most likely breaks policy!
    - especially as length, complexity, & precision of sequence increases

Derailment

- Insight: First return, then explore

Derailment

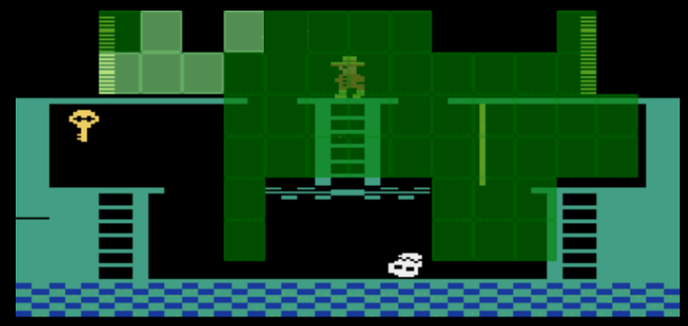
- Insight: First return, then explore
  - counter: hurt robustness?

Go-Explore Strategy

- Phase 1: First solve
- Phase 2: Then robustify
  - pay the cost to robustify only once you know *what* needs to be robustified

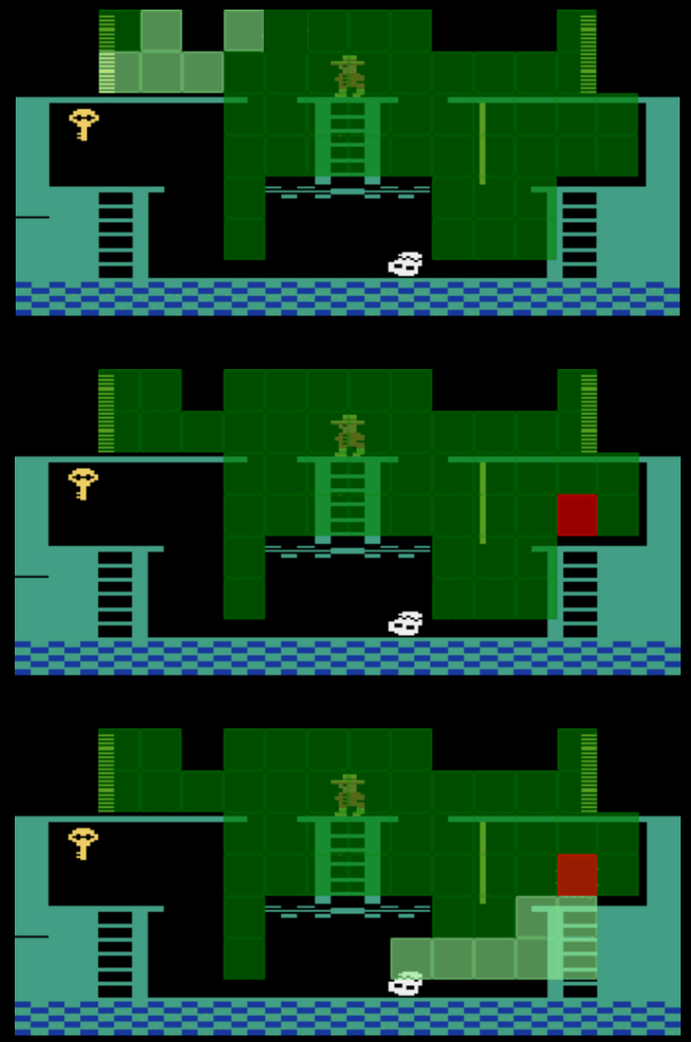
Go-Explore: Phase 1

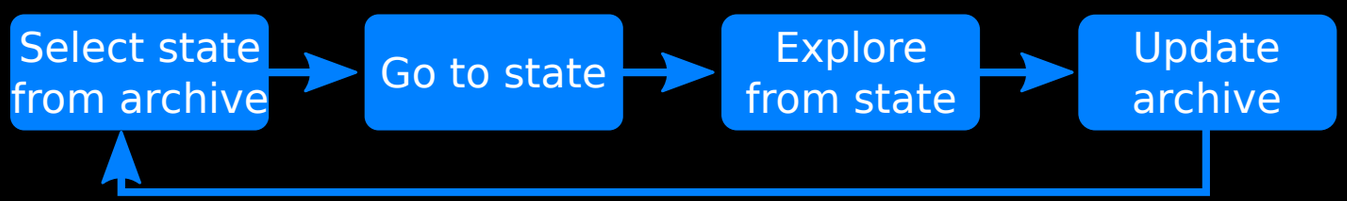
- initialization:
  - take random actions, store cells visited



Go-Explore: Phase 1

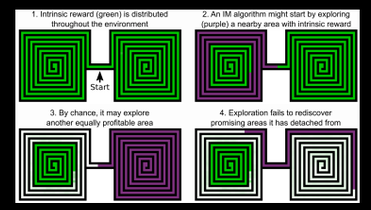
- Phase 1: explore until solved
  A. choose a cell from archive
  B. Go back to it
  C. Explore from it
  D. add newly found cells to archive
    - if better, replace old way of reaching cell



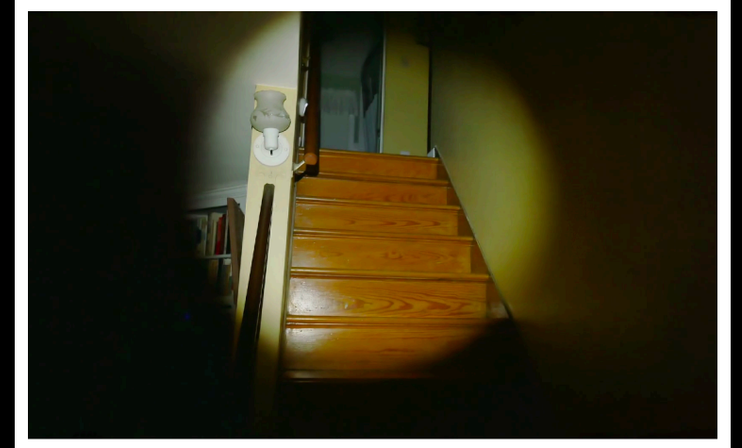


Avoids Detachment

By Remembering Promising Exploration Stepping Stones



- Intrinsic motivation:
  - narrow beam mining intrinsic motivation and moving on

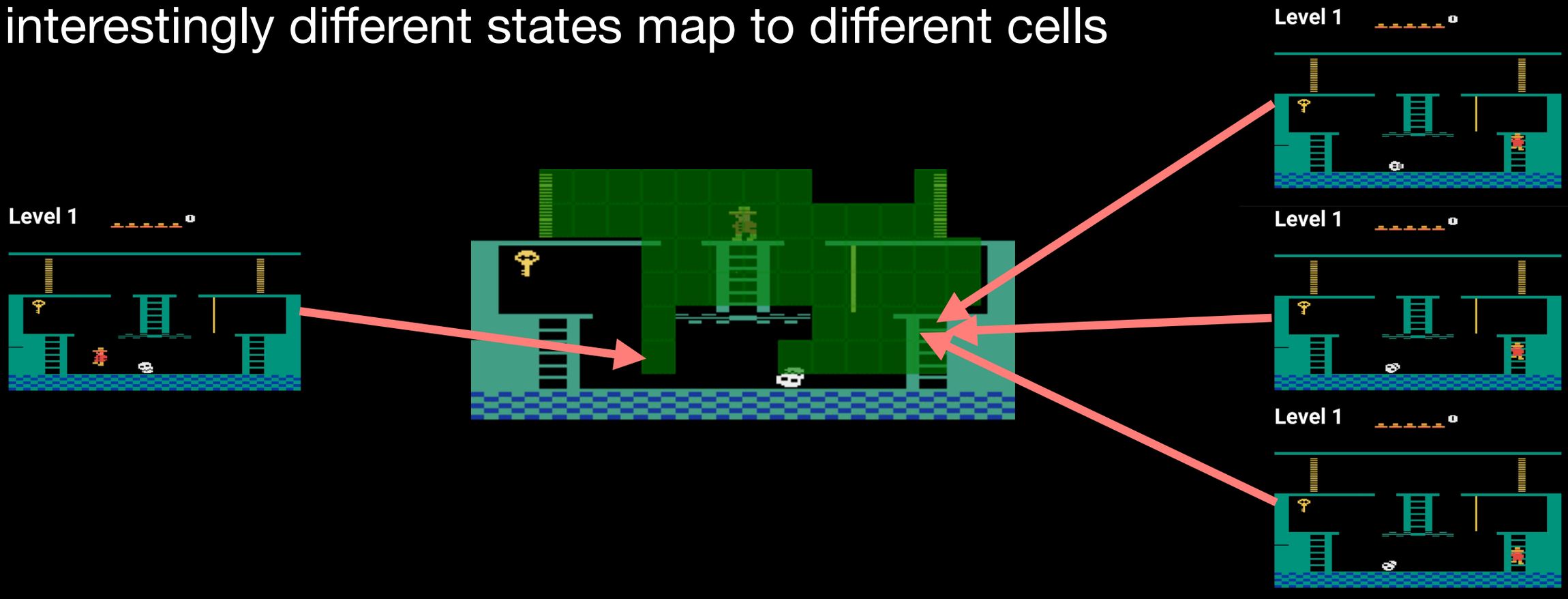


- Go-Explore
  - continuously expands sphere of knowledge

We are building a room connectivity map of the house!

Cell Representations

- For large state spaces (e.g. Atari), need conflation
  - similar states map to same cell
  - interestingly different states map to different cells

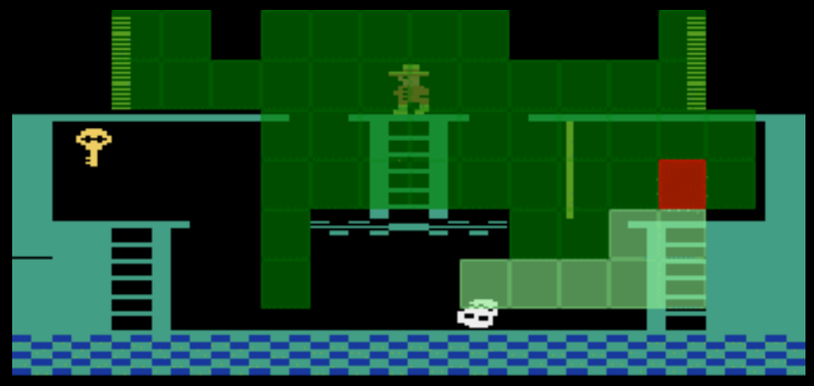


Cell Representations

- First attempt: downsampling images
  - dumb
  - fast
  - no game-specific domain knowledge

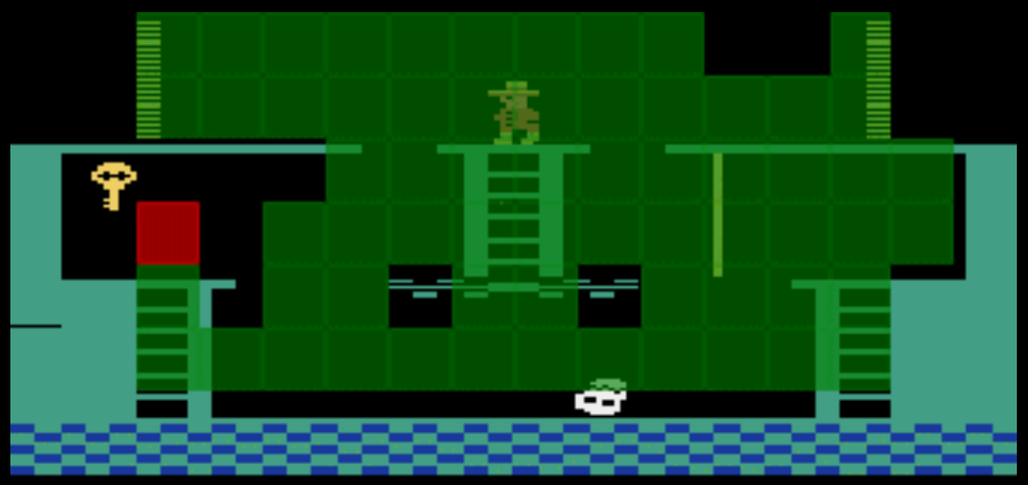
Choosing Cells

- Prefer cells that are
  - led to new states (productive)
  - less tested (newer)



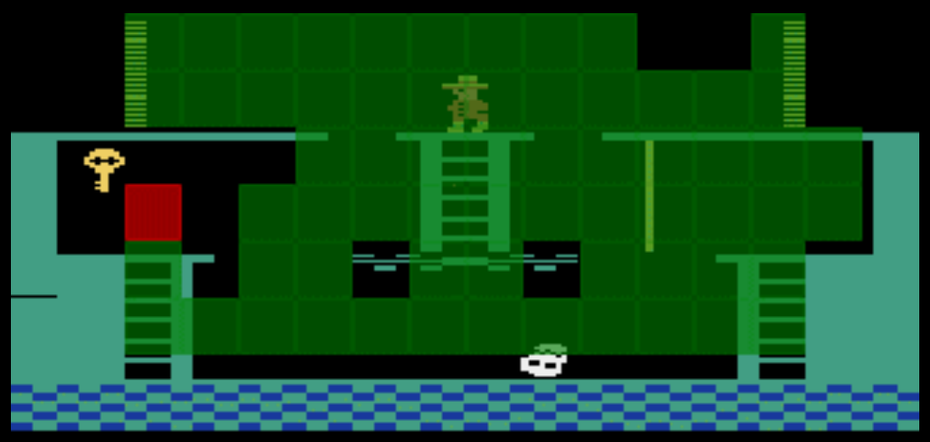
Returning to Cells

- resettable & deterministic: reset state (or replay actions)
- stochastic environment:
  - goal-conditioned policy
    - e.g. UVFA (Schaul et al. 2015), HER (Andrychowicz et al. 2017)
  - other ideas



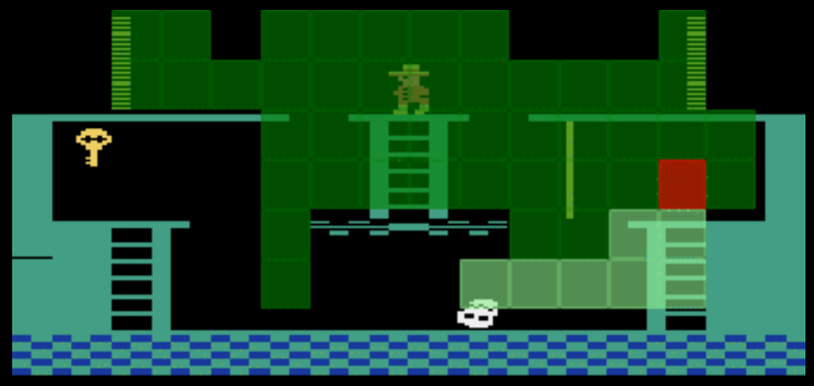
Returning to Cells

- save action-sequence trajectories to cells
  - open loop
  - no neural network!

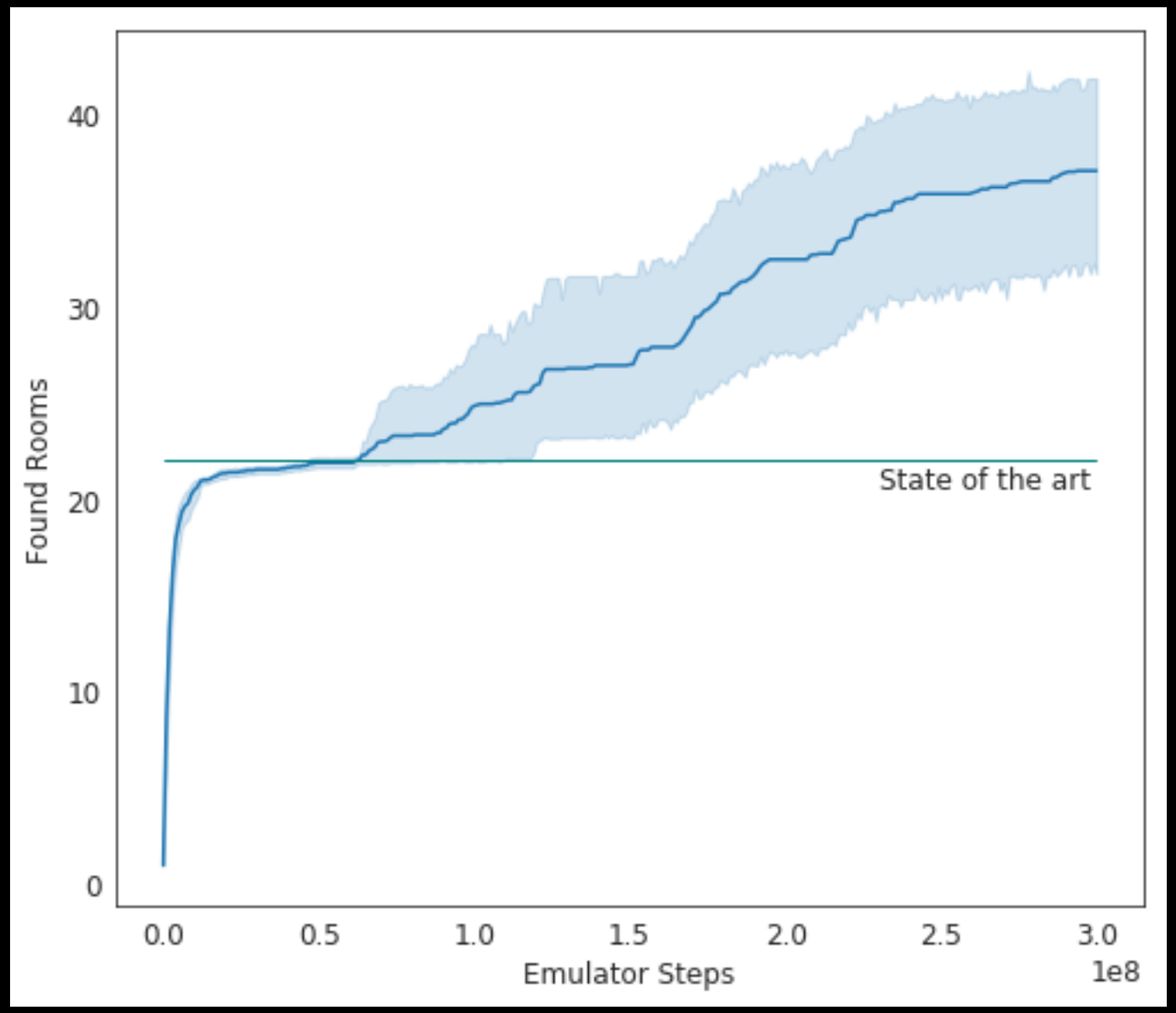


Exploration

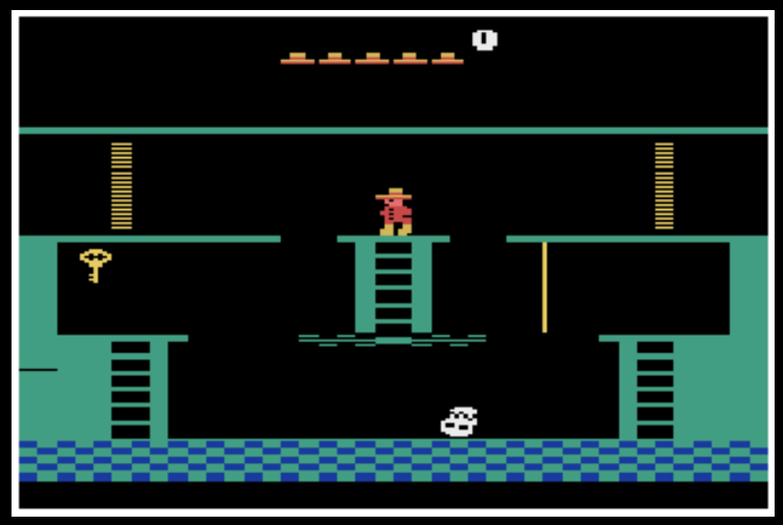
- after returning to a cell
- take random actions (k=100)



Montezuma's Revenge Results: Phase 1



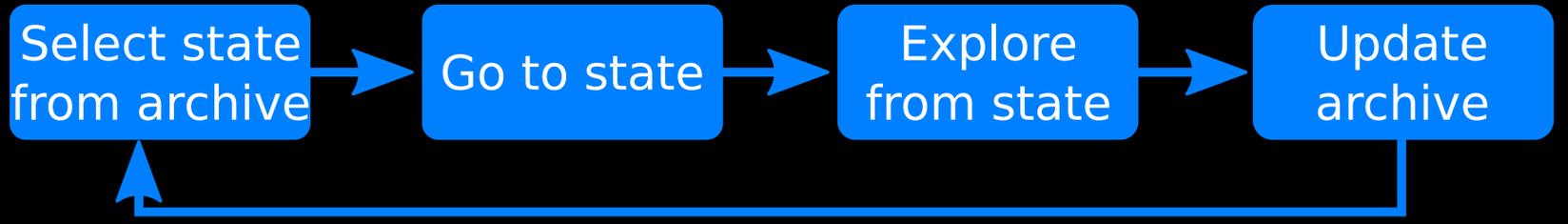
- Solves level 1 65% of runs



Go-Explore

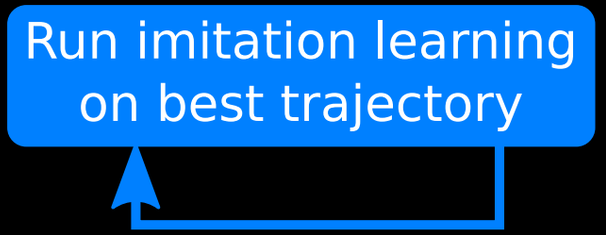
Separates learning a solution into two phases

Phase 1: Explore Until Solved

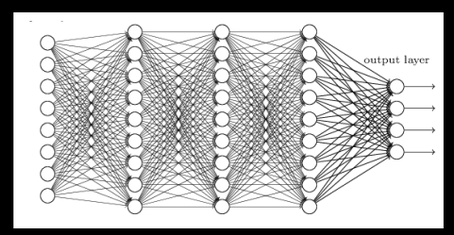


no neural networks
(in current work)

Phase 2: Robustify
(if necessary)



produces neural network
robust to stochasticity



Phase 2: Robustify

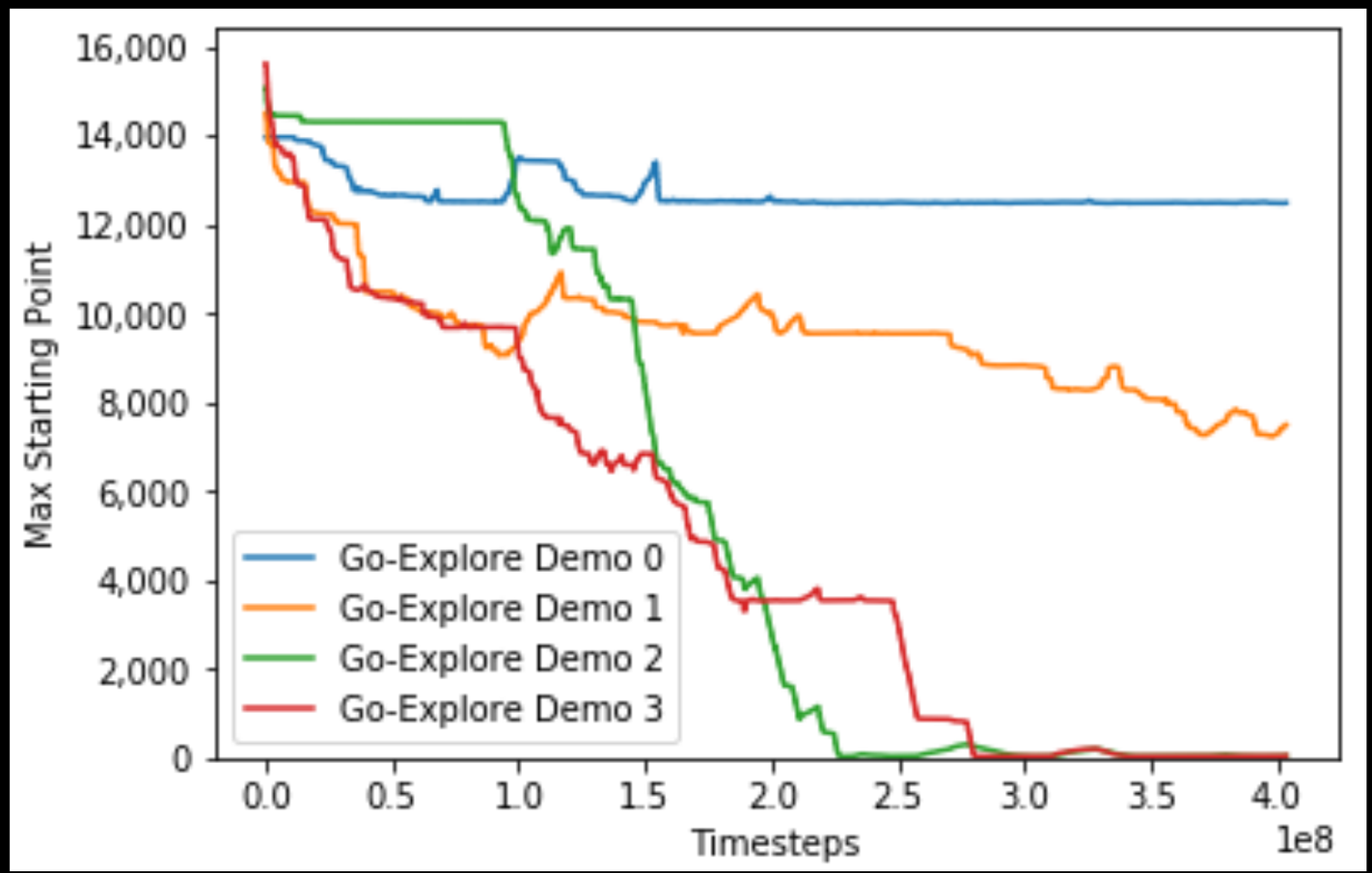
- Imitation learning can work well with human demonstrations
  - including on Montezuma's Revenge & Pitfall (e.g. Aytar 2018)
- Go-Explore Phase 1 generates solution demonstrations
  - automatically
  - quickly
  - cheaply
  - as many as you want

Phase 2: Robustify

- Most imitation learning algorithms should work
- We chose the “backward algorithm”
  - from Salimans & Chen 2018 (this workshop!)
  - similar: Backplay from Resnick et al. 2018

Many Demonstrations

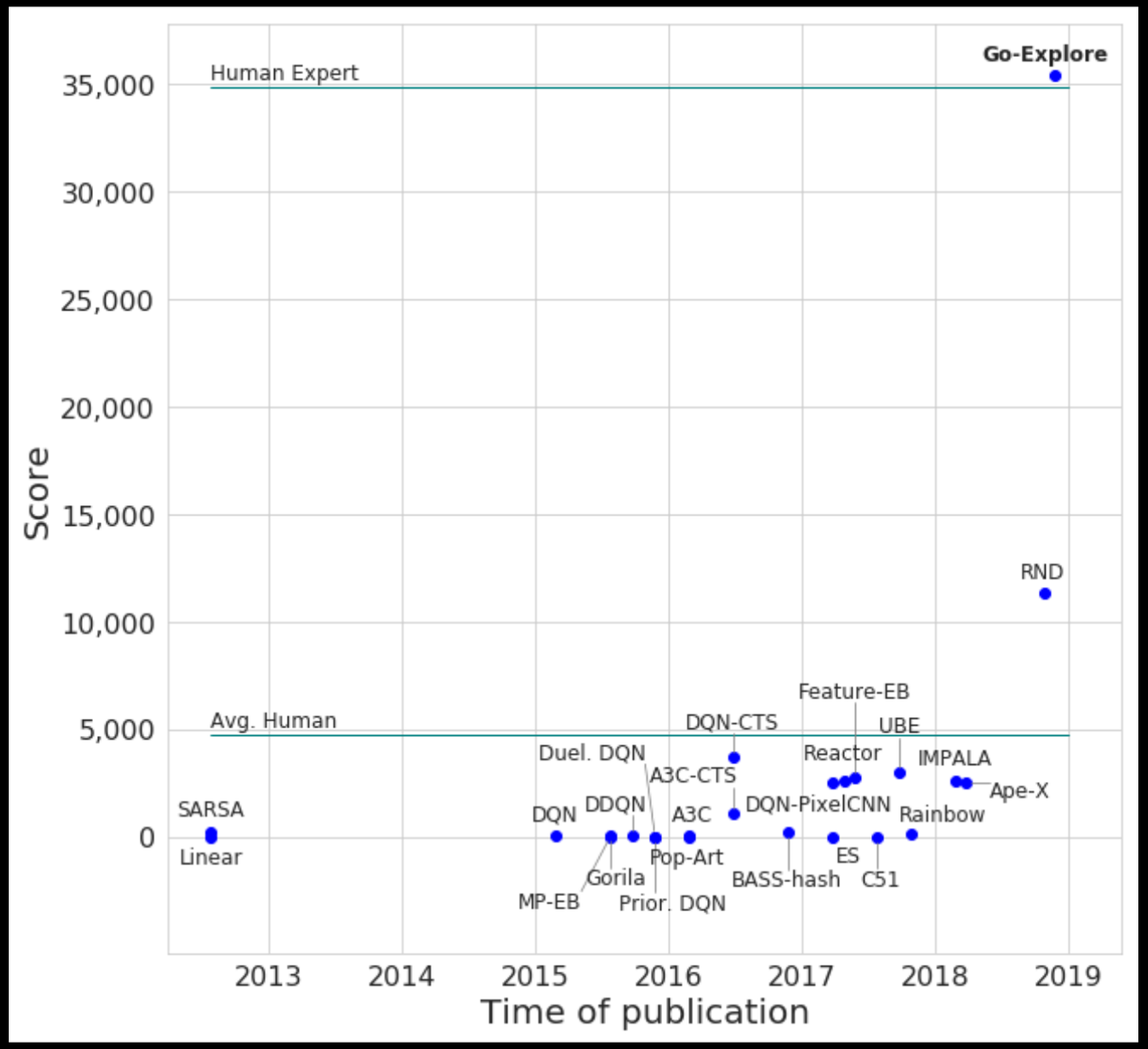
- Backwards Imitation somewhat unreliable from one demonstration
- We modified it to learn from many
  - here, 4
- Add no-ops at beginning
- Success rate: 100%



Example Successful Attempt

Results with Robust Deep Neural Networks

- 3x previous state of the art!
- Robust to stochasticity
  - random # no-ops up to 30
- No game-specific domain knowledge

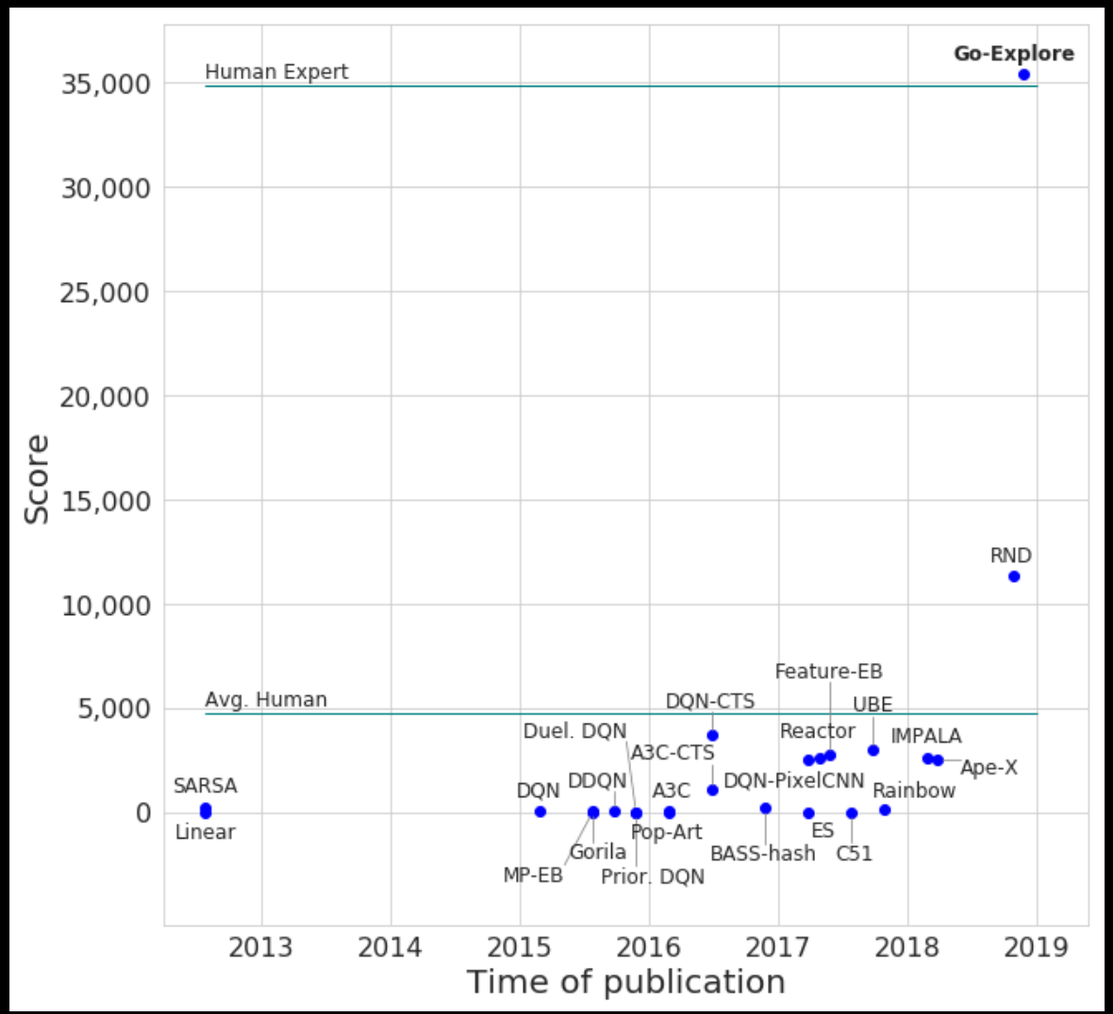


Go-Explore created a healthy debate

- When should we require stochasticity?
  - test time (only)?
  - training time too?

Stochasticity at Test Time

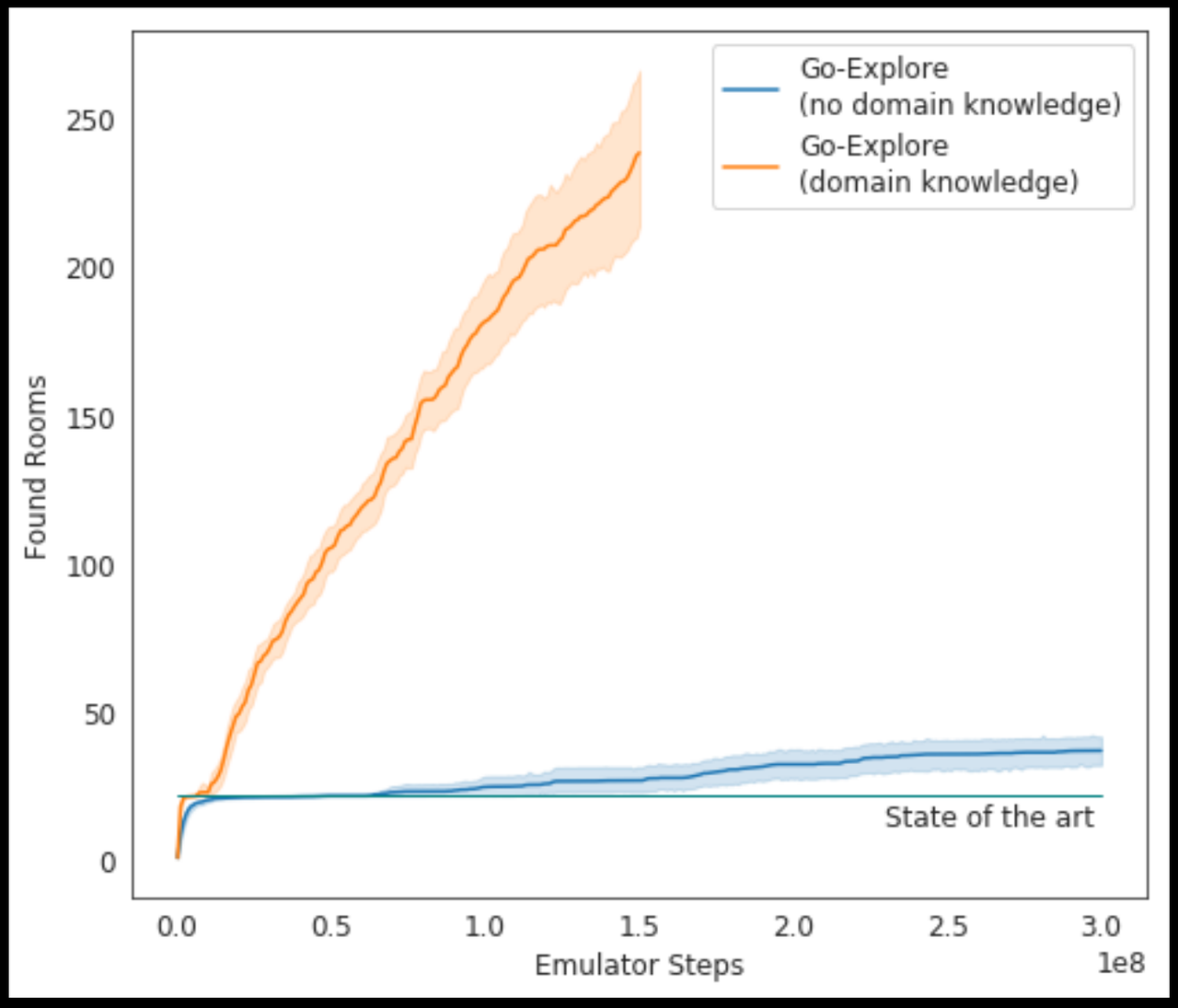
- Classic way: random number up to 30 no-ops



Adding Domain Knowledge

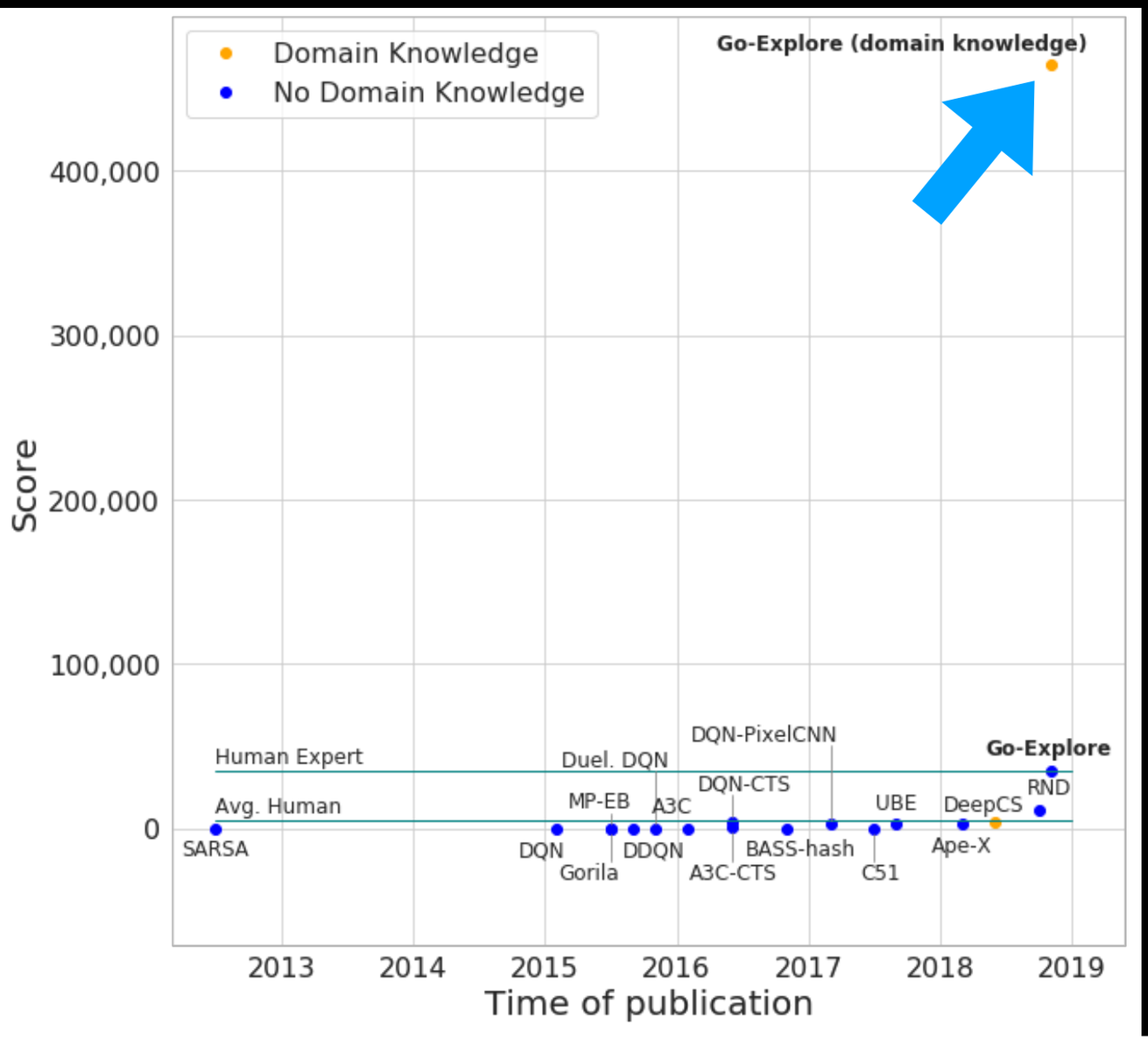
- Go-Explore can add it via cell representation
  - Important notes:
    - final post-robustification policies still play from pixels only
      - do not consume domain knowledge at eval time
      - wrote simple code to extract info from pixels (not emulator)
  - Montezuma's Revenge
    - unique combinations of: x, y location, room, level, numKeysHeld
  - Pitfall
    - <x, y> location, room

Montezuma's Results with Domain Knowledge
Phase 1: Exploration (deterministic eval)



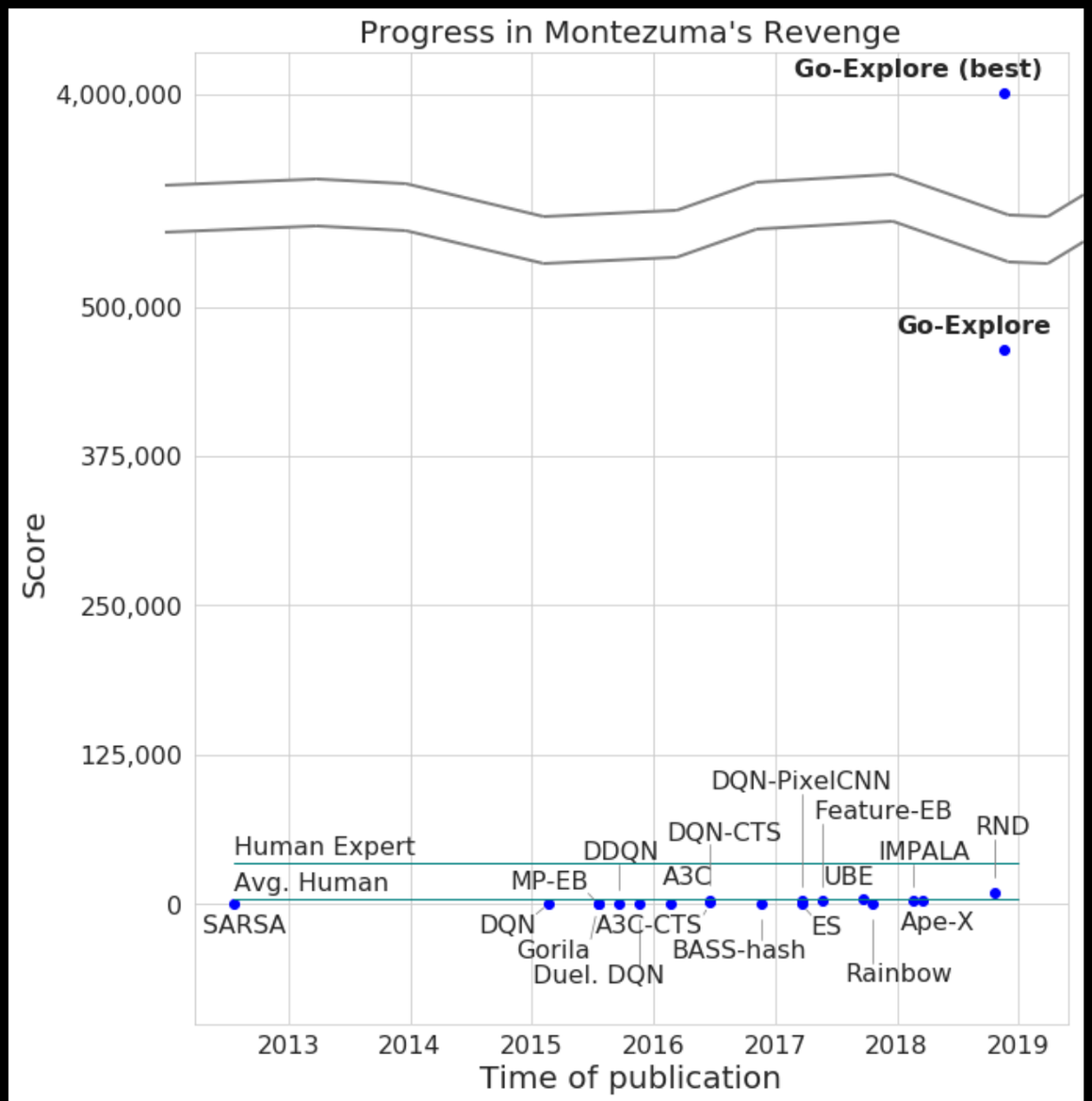
- Solves 9 levels on average
- in half the time

Results with Robust Deep Neural Networks



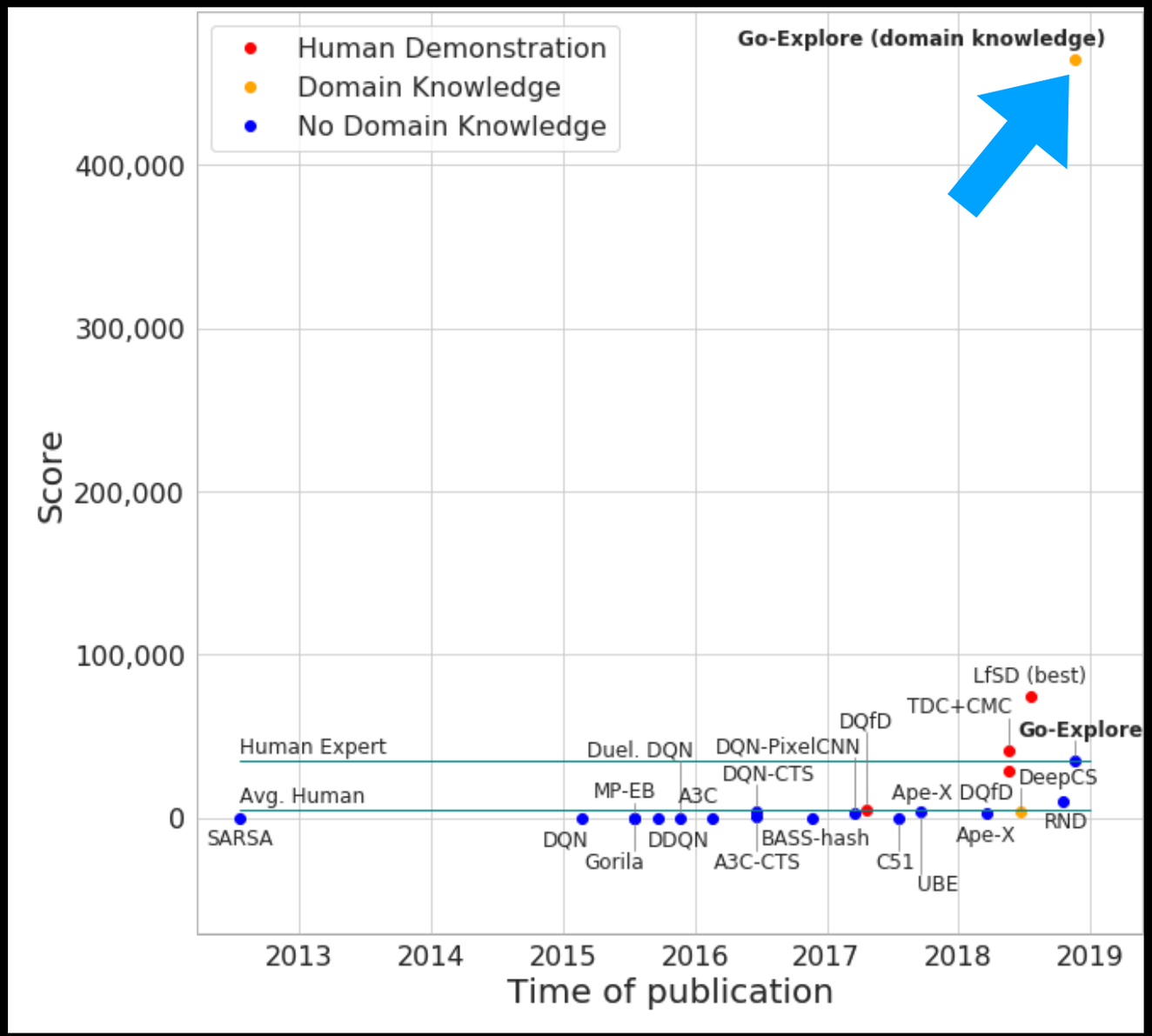
- Solves all 3 unique levels
- Levels 3+ nearly identical
  - slight timing differences
  - score changes state/inputs
- On average
  - scores over 469,209!
  - solves 29 levels!
- Sticky action eval:
  - scores 281,264
  - level 18

Results with Robust Deep Neural Networks



- Increased Gym's time limits
- Best neural network
  - scores over 4 million
  - reaches level 295
- Beats human world record
  - 1,219,200
  - achieves strictest definition of "super-human"

Results with Robust Deep Neural Networks



- Even beats previous work from human demonstrations
  - better demos
  - more demos

Not Expensive!

- Solving Level 1 of Montezuma's Revenge
  - Phase 1: Exploration
    - ~1 hour!
    - single machine (22 CPUs, 50GB RAM)
    - runs produce ~4GB of data
  - Phase 2: Robustification
    - ~1 day
      - 16 GPUS
- All told: ~1 day on modest hardware
  - compare to billions of frames, thousands of workers

Similar to graph search? (e.g. breadth-first search)

- Yes, but

1. Go-Explore adds Phase 2: Robustification to handle noise
2. Such algorithms are intractable in raw high-D state space

Similar to graph search? (e.g. breadth-first search)

- Yes, but

1. Go-Explore adds Phase 2: Robustification to handle noise
2. Such algorithms are intractable in raw high-D state space
   - Go-Explore insight: run these great graph search algorithms in low-D conflated spaces!
   - adds many challenges
     - which cells can you reach from current cell? how (have to search)? can't replace subpaths. etc.
   - requires many algorithmic innovations
   - new research area: porting classic graph algorithms to high-D (conflated representations): BFS, DFS, Dijkstra's, A-Star, etc.

Episodic memory / structured memory

- Explicit memory of state space structure critical for intelligent exploration.
- Without it you cannot explore a museum in a systematic way.
- Tons of biological inspiration from cognitive maps and so on.

Solutions

Episodic memory structures: non parametric book-keeping of states, their (Monte Carlo or bootstrapped) rewards, state to state connectivity trajectories.

Mixture of policies / map of policies (skills): fast pick the right one with few trials.

Recurrent policies that learn from a series of attempts.

Neural weight initialization that permits updating weights with very few gradient steps.

Solutions

Episodic memory structures: non-parametric book-keeping of states, their (Monte Carlo or bootstrapped) rewards, state-to-state connectivity trajectories.

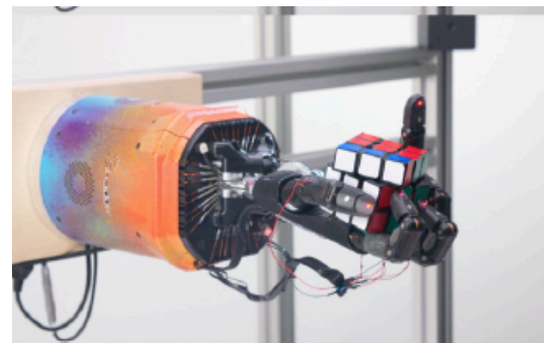
Recurrent policies that learn from a series of attempts/episodes.

Neural weight initialization that permits updating weights with very few gradient steps.

Mixture of policies / map of policies (skills): fast pick the right one with few trials.

Recurrent task policies

- Classic RL formulation: learn to solve a task or a set of tasks given M episodes of interactions relevant to the task(s).
  - Examples: rotate the rubic cube to a desired orientation from a randomly sampled initial pose.
- Adaptive (meta) RL: learn to solve a task K consecutive times
  - Examples: rotate a rubic cube to desired orientations K consecutive times. This permits my early attempts to help me improve towards and be better at the later ones

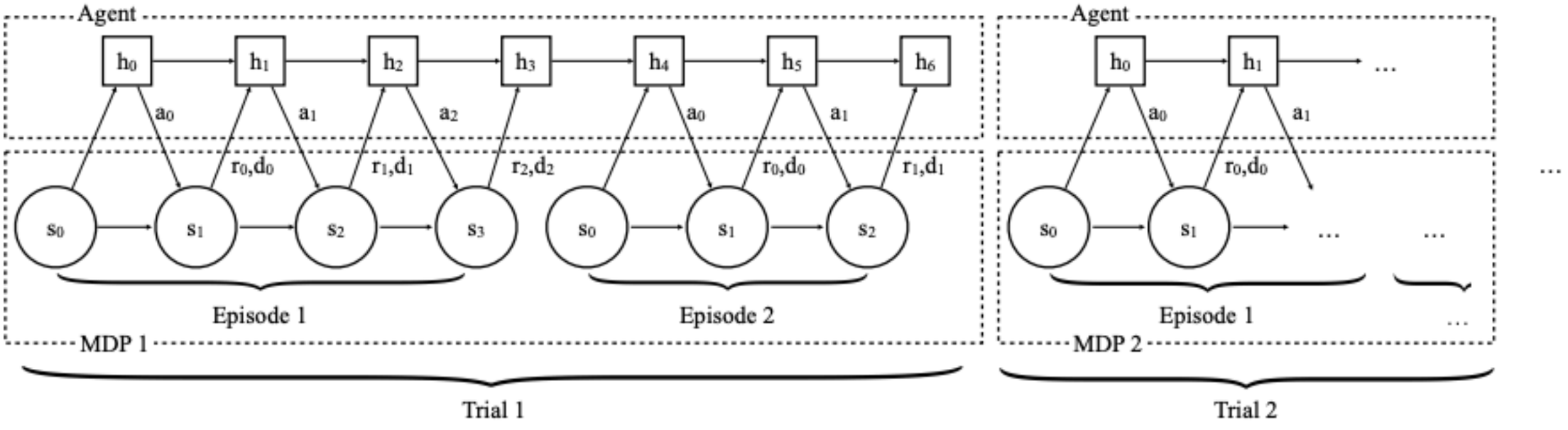


Recurrent task policies

First timestep: agent is placed in a random task/world. E.g., in a random rubric cube orientation and target re-orientation.

New observation= (old observation, reward, ``done" signal)

New-task episode (*trial*) = K old task episodes



Memory is erased at the end of a trial, instead of the end of an episode.

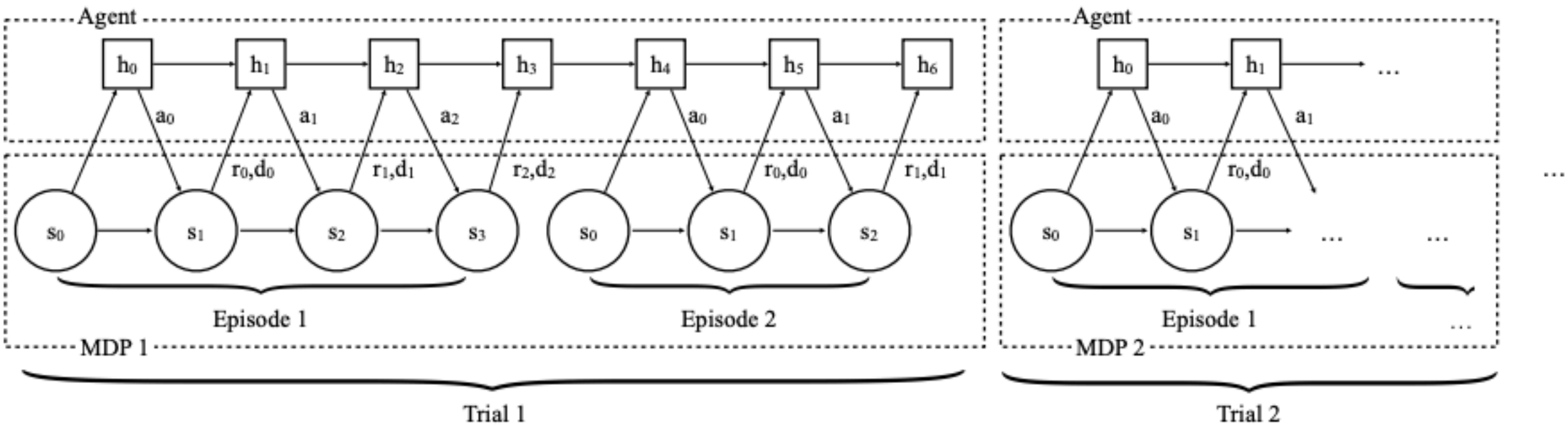
This permits learning to learn from failures of previous episodes within each trial.

Recurrent task policies

First timestep: agent is placed in a random task/world. E.g., in a random rubric cube orientation and target re-orientation.

New observation= (old observation, reward, ``done" signal)

New-task episode (*trial*) = K old task episodes



Fast RL: the fast update of the latent state h

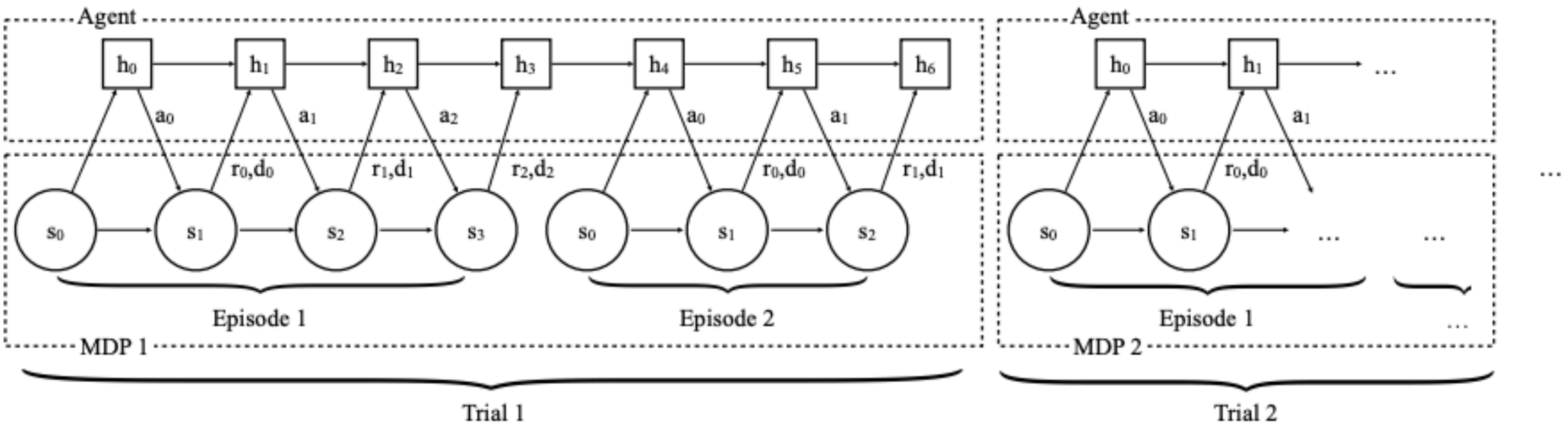
Slow RL: the slow weight updates of the recurrent neural net weights

Recurrent task policies

First timestep: agent is placed in a random task/world. E.g., in a random rubric cube orientation and target re-orientation.

New observation= (old observation, reward, ``done" signal)

New-task episode (*trial*) = K old task episodes



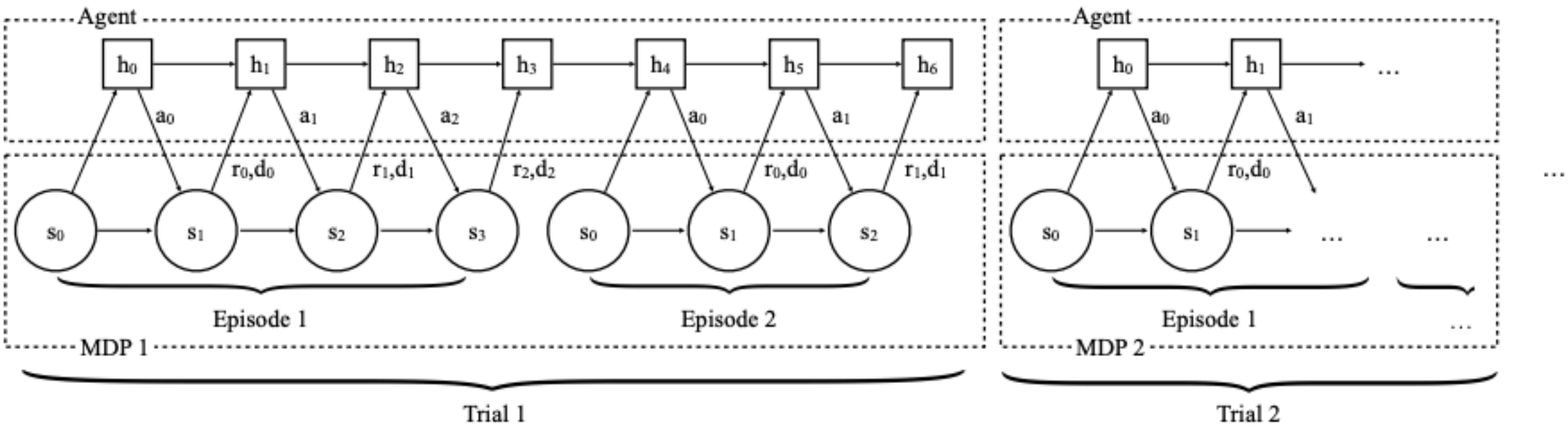
Generalization: For each trial a separate MDP is drawn. For example, for the cube orientation task, we draw MDPs with different arm dynamics, or cube physical properties. In this way, our LSTM policy learns to adapt to the new environment after K interactions

Recurrent task policies

First timestep: agent is placed in a random task/world. E.g., in a random rubric cube orientation and target re-orientation.

New observation= (old observation, reward, ``done" signal)

New-task episode (*trial*) = K old task episodes



Performance metric: Maximize the expected total discounted reward accumulated during a single trial rather than a single episode.

Recurrent task policies - Bernoulli bandits

We are given L arms. The reward from each arm is binary and governed by a Bernoulli distribution of parameter p. p is unknown. We want to maximize the total reward we obtain during K number of interactions (K pulls). In other words, minimize the cumulative regret.

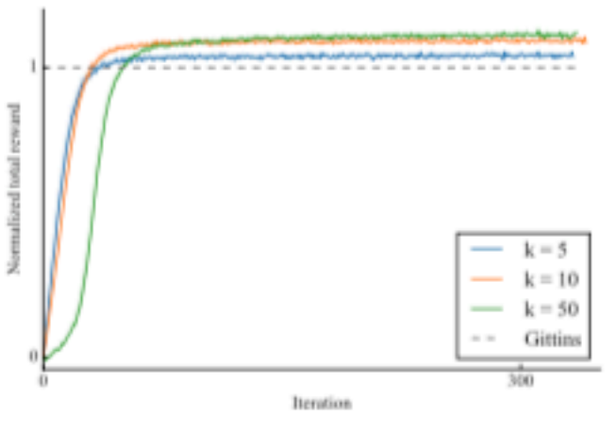
We can use:

- epsilon-greedy
- UCB
- Thompson sampling
- …

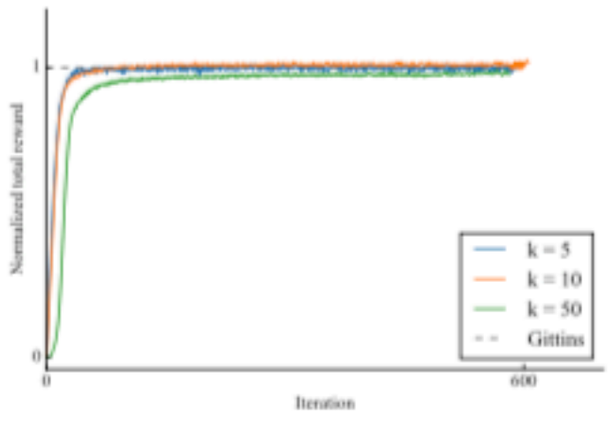
Recurrent task policies - Bernoulli bandits

k : number of bandits, n : number of interactions

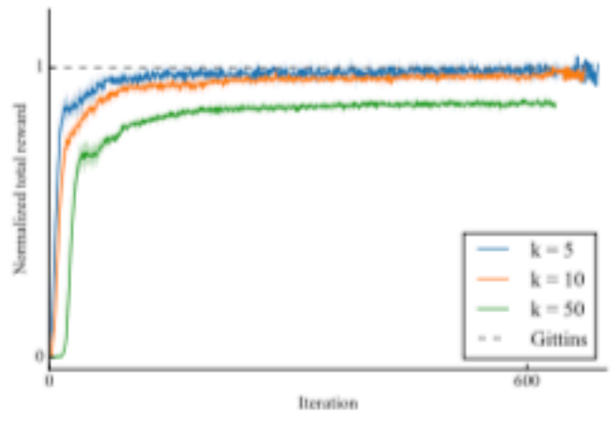
Setup	Random	Gittins	TS	OTS	UCB1	ϵ-Greedy	Greedy	RL^2
$n = 10, k = 5$	5.0	**6.6**	5.7	6.5	**6.7**	**6.6**	**6.6**	**6.7**
$n = 10, k = 10$	5.0	**6.6**	5.5	6.2	**6.7**	**6.6**	**6.6**	**6.7**
$n = 10, k = 50$	5.1	6.5	5.2	5.5	**6.6**	6.5	6.5	**6.8**
$n = 100, k = 5$	49.9	**78.3**	74.7	**77.9**	**78.0**	75.4	74.8	**78.7**
$n = 100, k = 10$	49.9	**82.8**	76.7	81.4	82.4	77.4	77.1	**83.5**
$n = 100, k = 50$	49.8	**85.2**	64.5	67.7	84.3	78.3	78.0	**84.9**
$n = 500, k = 5$	249.8	**405.8**	**402.0**	**406.7**	**405.8**	388.2	380.6	**401.6**
$n = 500, k = 10$	249.0	**437.8**	429.5	**438.9**	**437.1**	408.0	395.0	432.5
$n = 500, k = 50$	249.6	**463.7**	427.2	437.6	457.6	413.6	402.8	438.9



(a) $n = 10$



(b) $n = 100$



(c) $n = 500$

Solutions

Episodic memory structures: non parametric book-keeping of states, their (Monte Carlo or bootstrapped) rewards, state to state connectivity trajectories.

Mixture of policies / map of policies (skills): fast pick the right one with few trials.

Recurrent policies that learn from a series of attempts.

Neural weight initialization that permits updating weights with very few gradient steps.

Solutions

Episodic memory structures: non parametric book-keeping of states, their (Monte Carlo or bootstrapped) rewards, state to state connectivity trajectories.

Mixture of policies / map of policies (skills): fast pick the right one with few trials.

Recurrent policies that learn from a series of attempts.

Neural weight initialization that permits updating weights with very few gradient steps.

Learning multiple policies for solving a task

- What if we do not keep track of only the single best performing policy for our objective or set of objectives?
- What if we keep track (archive) a SET of policies, each of which does well on a different set of conditions (different environments, different trade-offs of objectives, and so on)
- What if such jointly optimization of policies for a set of related objectives/conditions would help us optimize each one better?

Damage Recovery

Animals

- Have intuitions about different ways to move
- Conduct a few, intelligent tests
- Pick a behavior that works despite injury



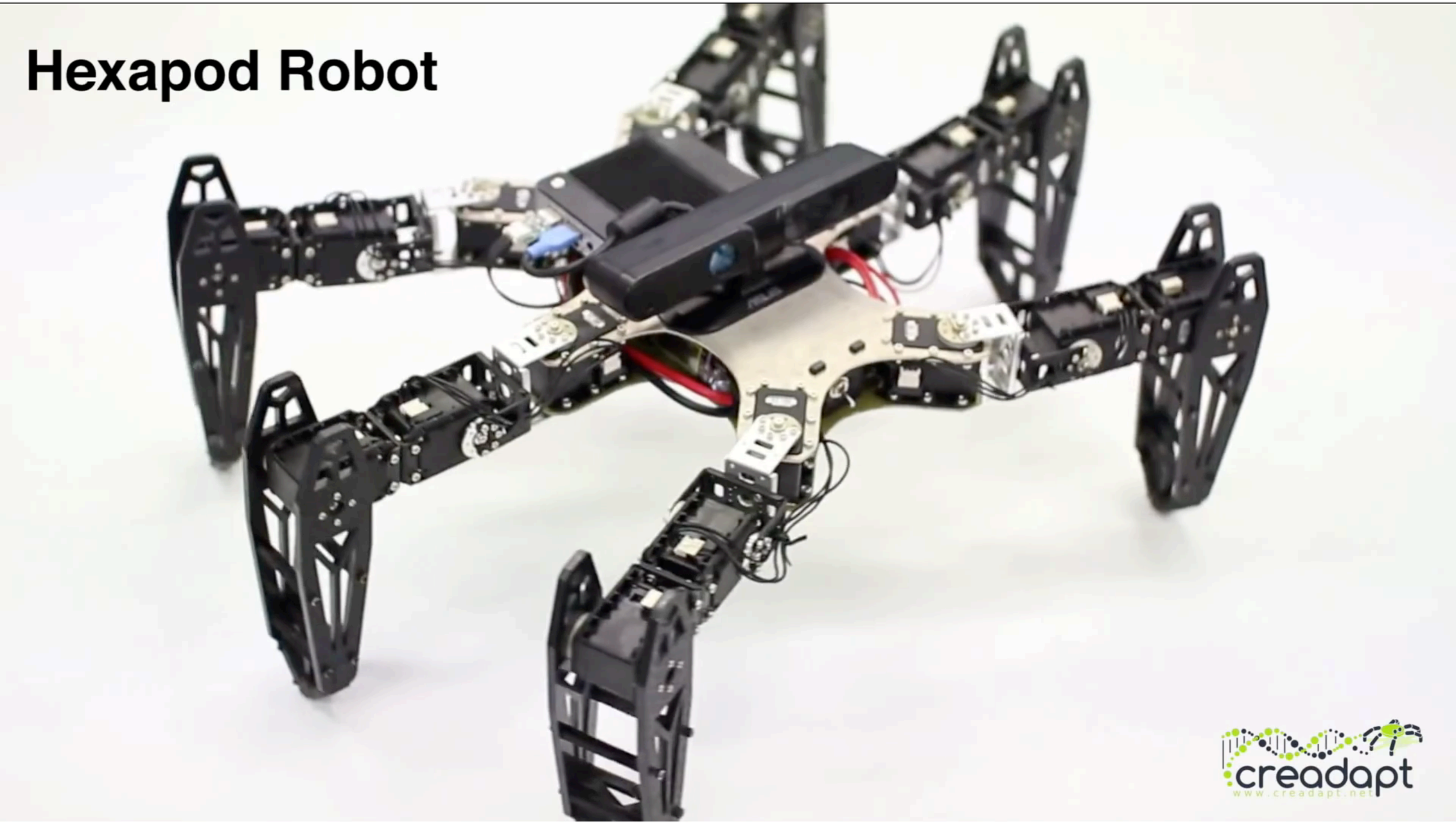
Robots that Adapt Like Animals

- Have intuitions about different ways to move
- Conduct a few, intelligent tests
- Pick a behavior that works despite injury

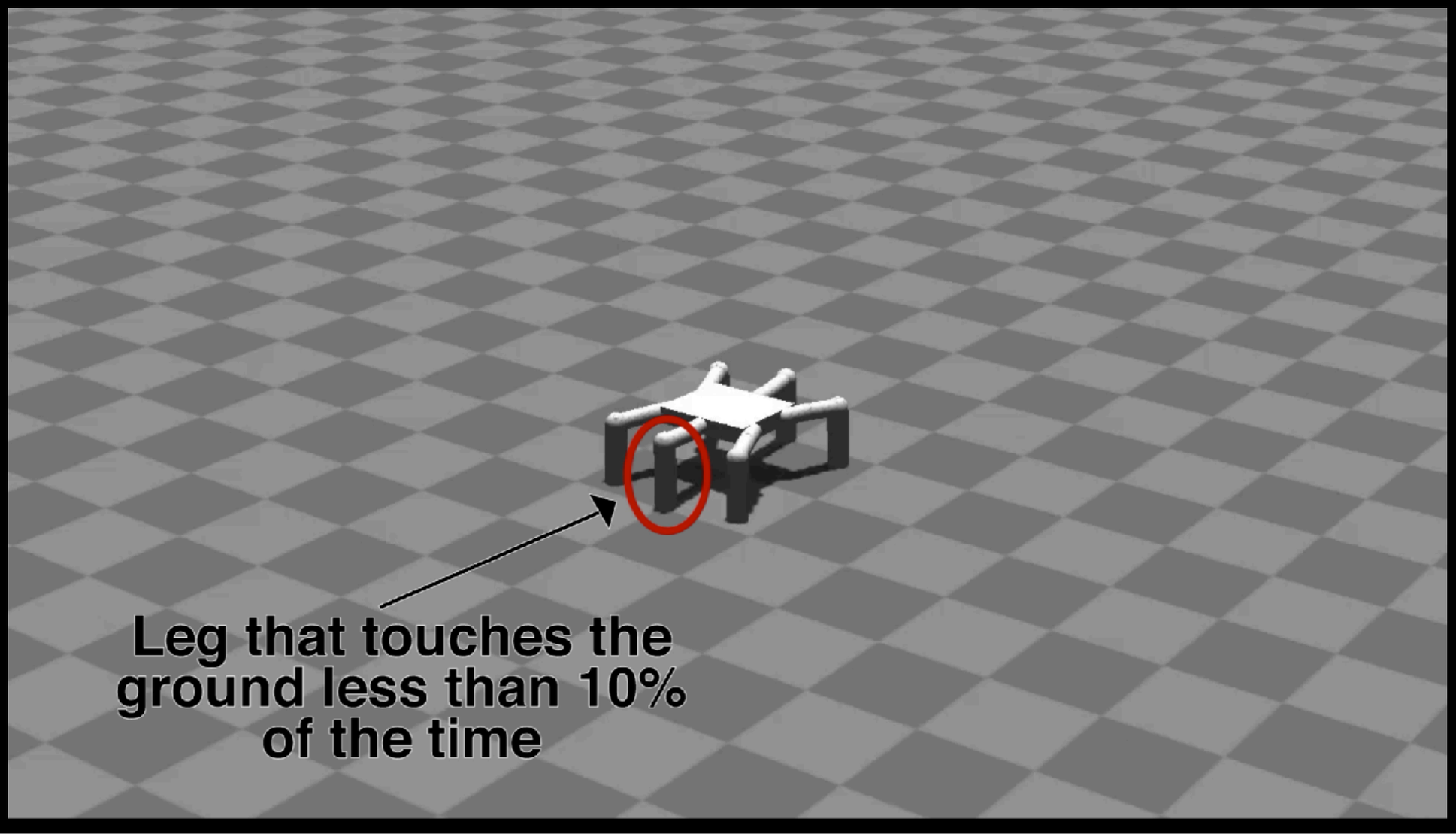
intuitions about
different ways to move

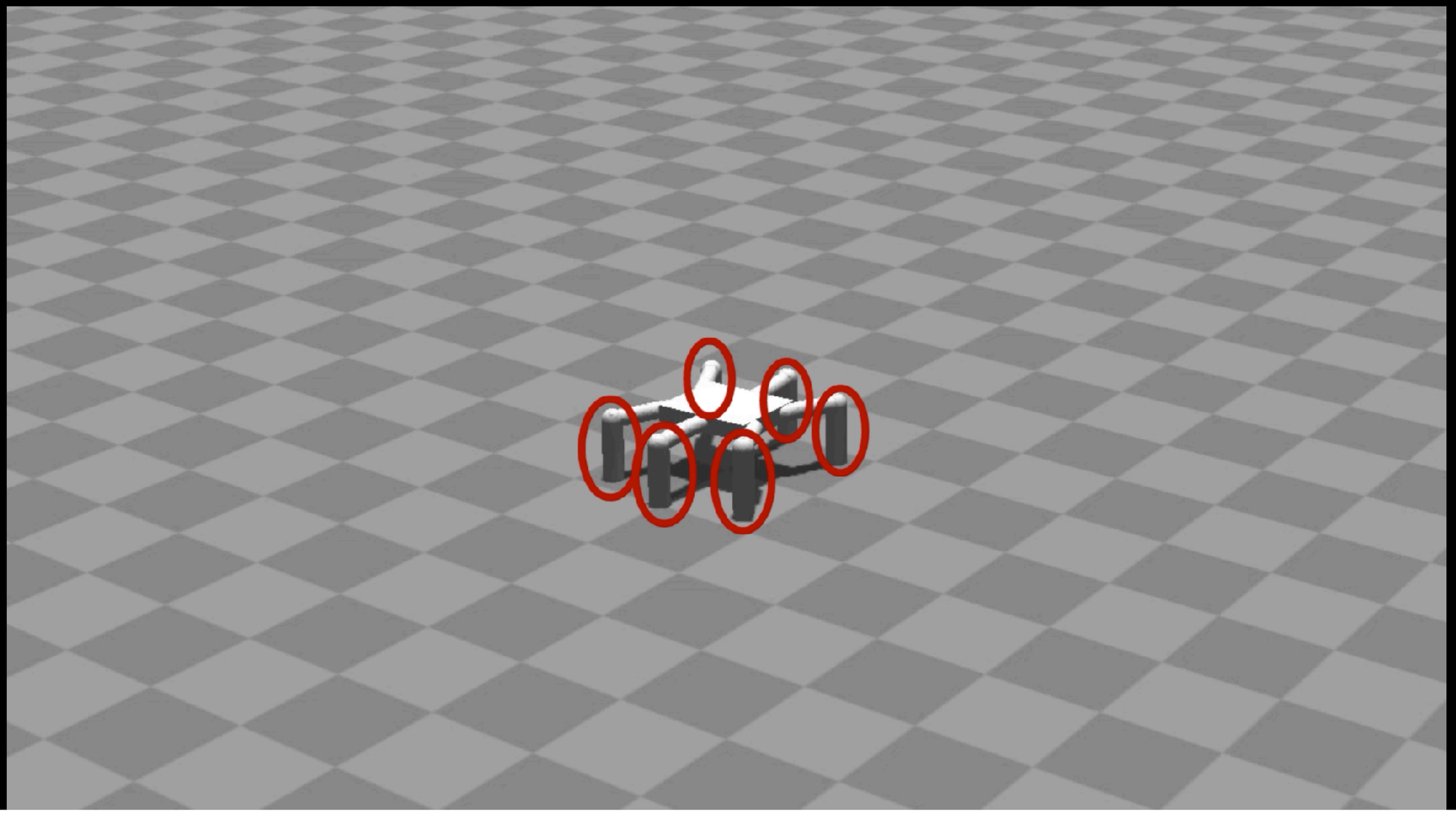
- Traditional machine learning methods produce little diversity
- Need an algorithm to produce diverse set of high-quality solutions
- Quality Diversity algorithms (MAP-Elites)

Hexapod Robot



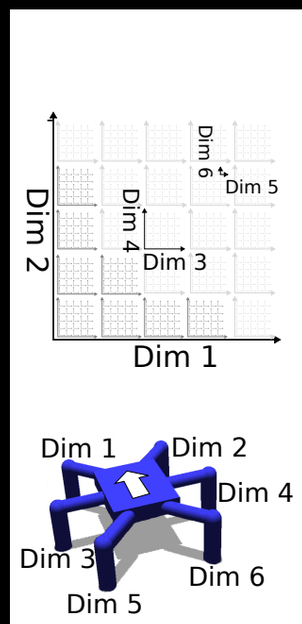
intuitions about
different ways to move

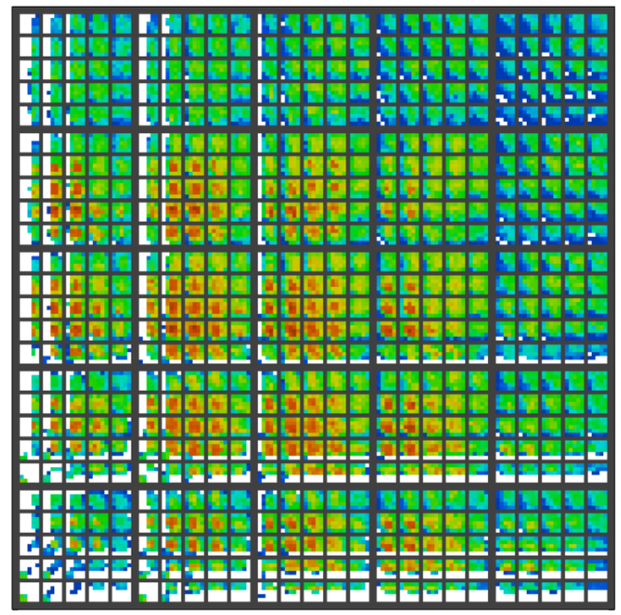




intuitions about
different ways to move

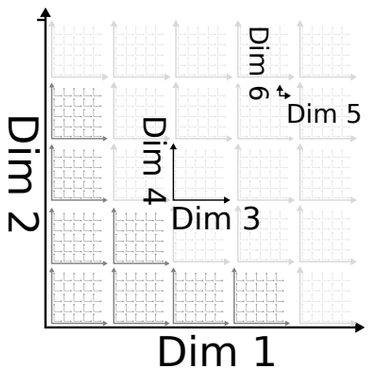
- MAP-Elites
- Behavioral characterization
  - % of time each leg touches the ground (6-dimensional)
- Massive search space
- MAP-Elites map has ~13,000 diverse, high-performing gaits

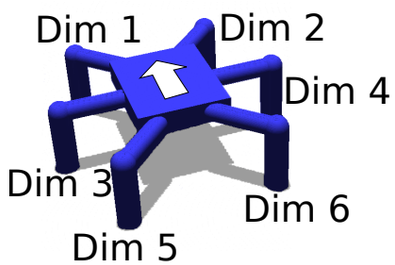


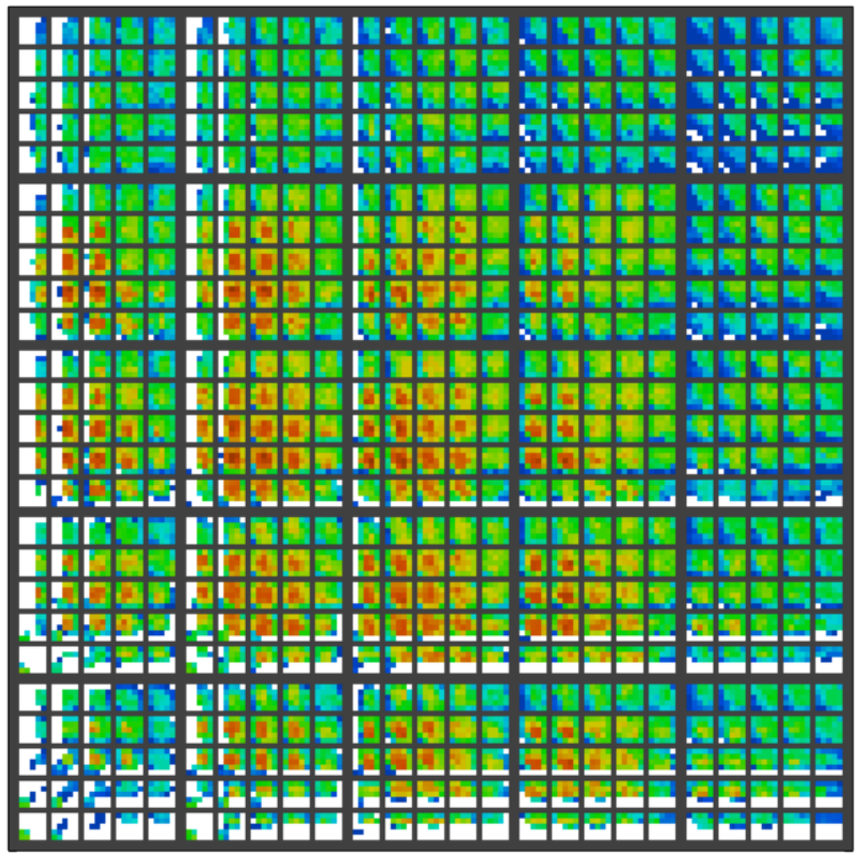


Initial Map

intuitions about
different ways to move







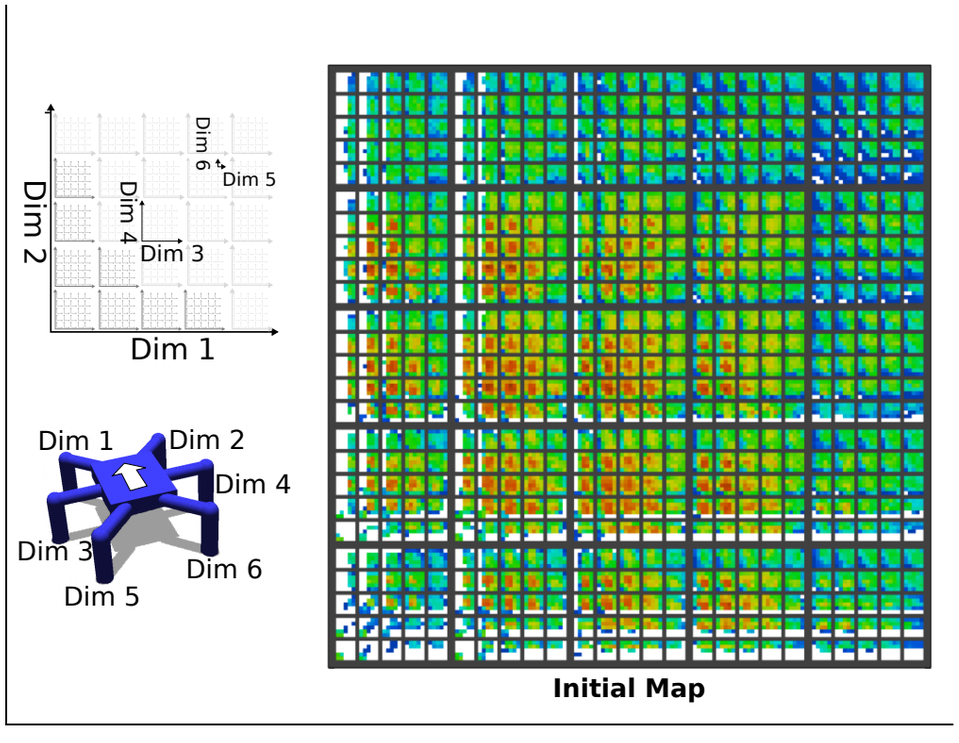
Initial Map

On the undamaged,
simulated robot

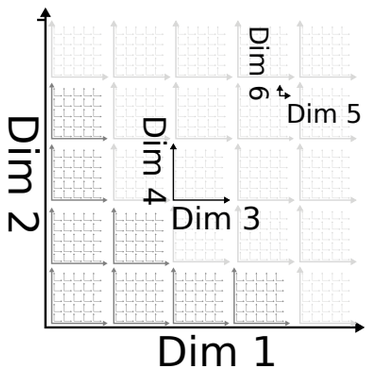
Filling in the map by MAP-elites

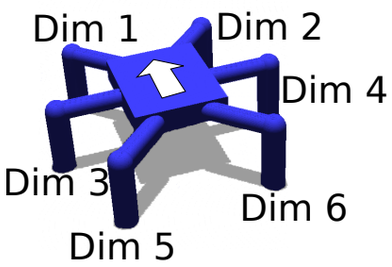
procedure MAP-ELITES ALGORITHM (SIMPLE, DEFAULT VERSION)
$(\mathcal{P} \leftarrow \emptyset, \mathcal{X} \leftarrow \emptyset)$ ▷ *Create an empty, N-dimensional map of elites: {solutions $\mathcal{X}$ and their performances $\mathcal{P}$}*
for iter $= 1 \rightarrow I$ **do** ▷ *Repeat for I iterations.*
if iter $< G$ **then** ▷ *Initialize by generating G random solutions*
$\mathbf{x}' \leftarrow$ random_solution()
else ▷ *All subsequent solutions are generated from elites in the map*
$\mathbf{x} \leftarrow$ random_selection($\mathcal{X}$) ▷ *Randomly select an elite x from the map $\mathcal{X}$*
$\mathbf{x}' \leftarrow$ random_variation($\mathbf{x}$) ▷ *Create x', a randomly modified copy of x (via mutation and/or crossover)*
$\mathbf{b}' \leftarrow$feature_descriptor($\mathbf{x}'$) ▷ *Simulate the candidate solution x' and record its feature descriptor $\mathbf{b}'$*
$p' \leftarrow$performance($\mathbf{x}'$) ▷ *Record the performance p' of x'*
if $\mathcal{P}(\mathbf{b}') = \emptyset$ or $\mathcal{P}(\mathbf{b}') < p'$ **then** ▷ *If the appropriate cell is empty or its occupants's performance is $\leq p'$, then*
$\mathcal{P}(\mathbf{b}') \leftarrow p'$ ▷ *store the performance of x' in the map of elites according to its feature descriptor $\mathbf{b}'$*
$\mathcal{X}(\mathbf{b}') \leftarrow \mathbf{x}'$ ▷ *store the solution x' in the map of elites according to its feature descriptor $\mathbf{b}'$*
return feature-performance map ($\mathcal{P}$ and $\mathcal{X}$)

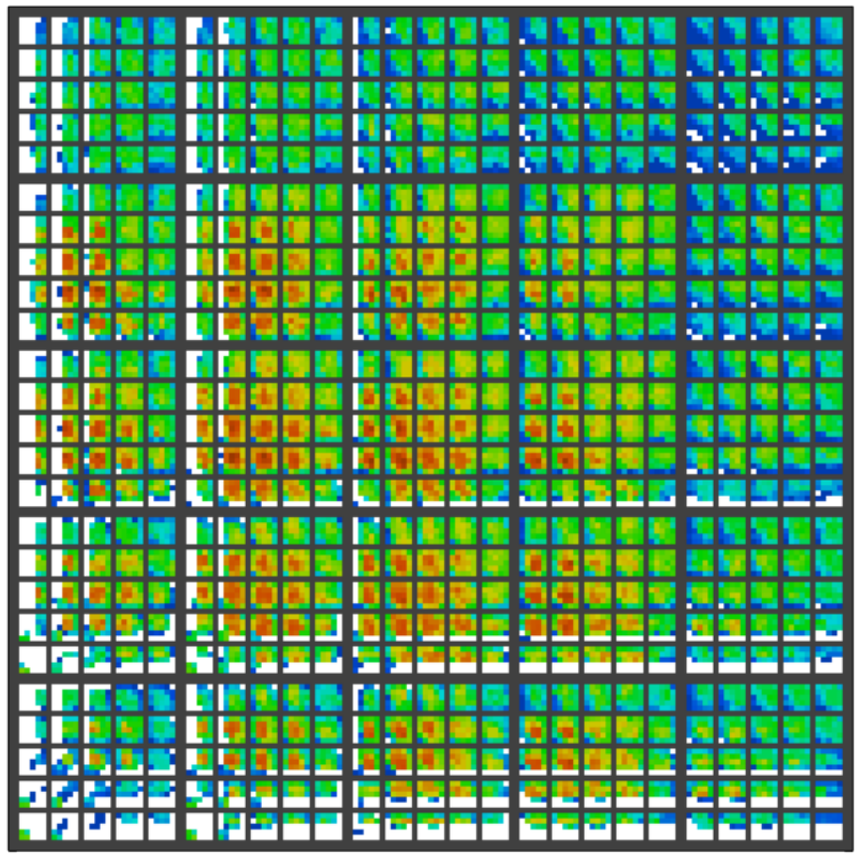
Q:How is MAP-elites better than evolutionary strategies discussed in the first lecture?



intuitions about different ways to move





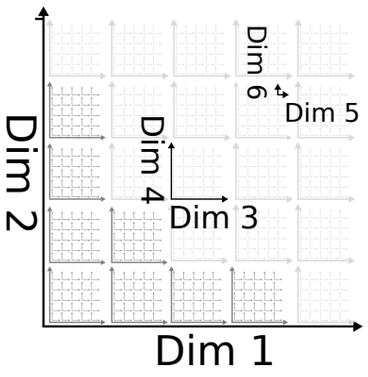


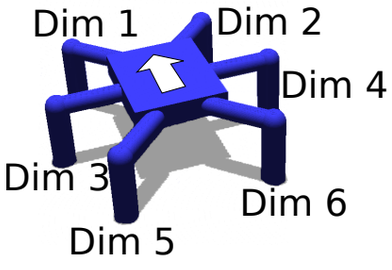
Initial Map

few, intelligent tests

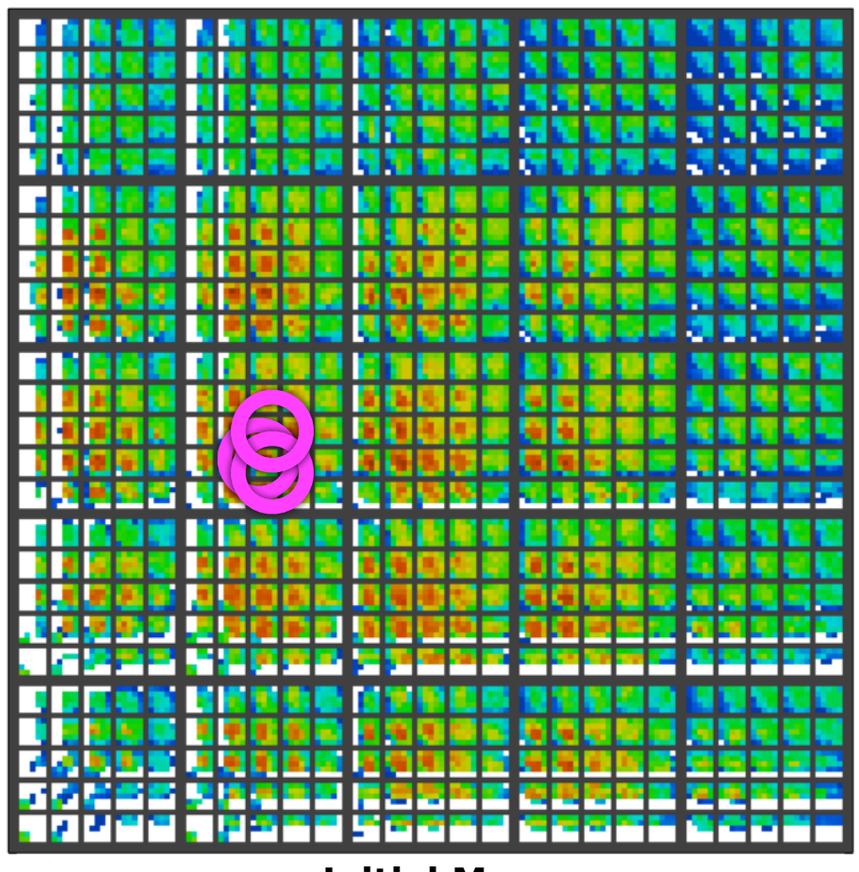
Which behaviors should we test?

intuitions about different ways to move





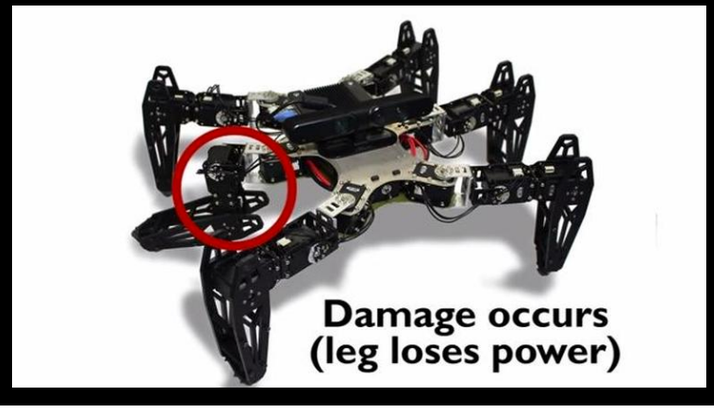
few, intelligent tests



Initial Map

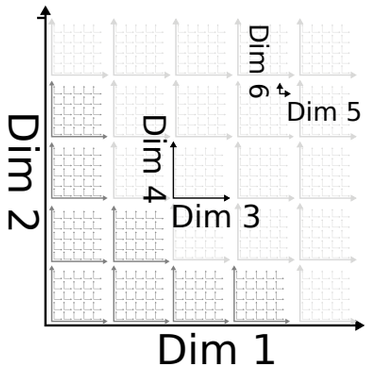
Could try top N:

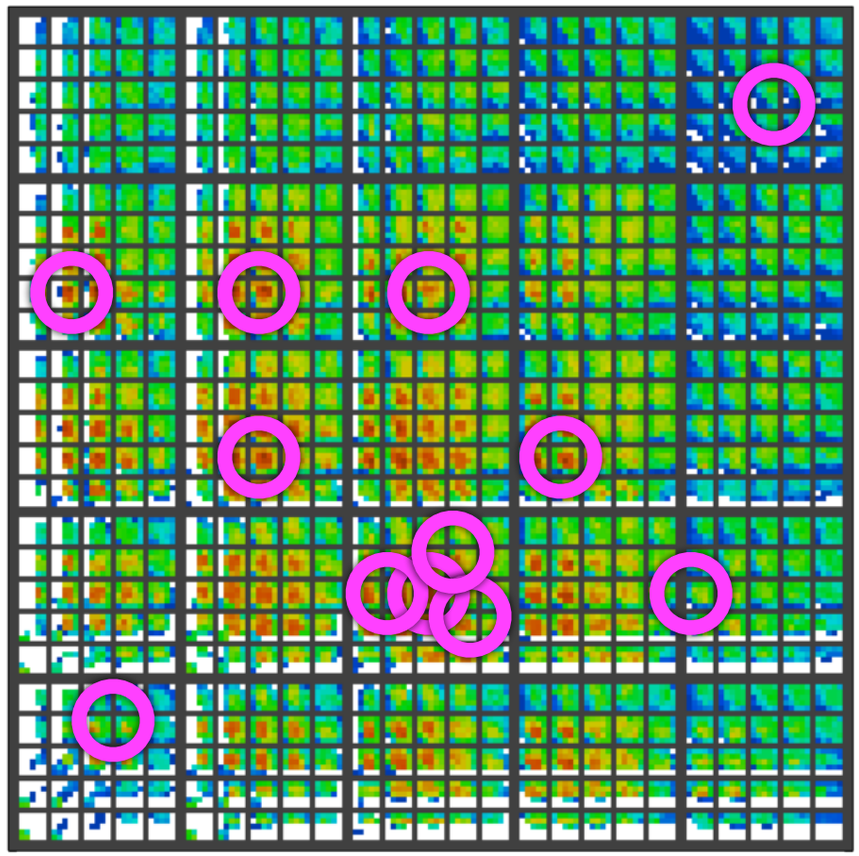
But they are likely very similar.



intuitions about different ways to move

few, intelligent tests





Initial Map

Bayesian Optimization:

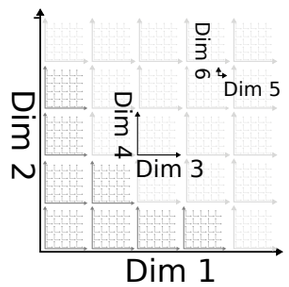
Tries different types solutions

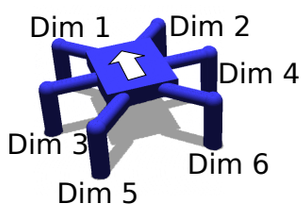
Bayesian Optimization

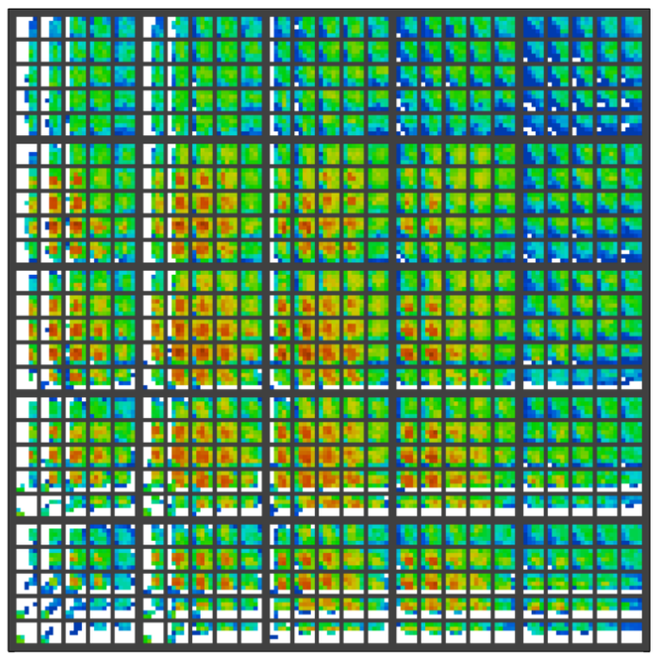
Prior:
MAP-Elites Map

Posterior:
Map updated after real-world tests

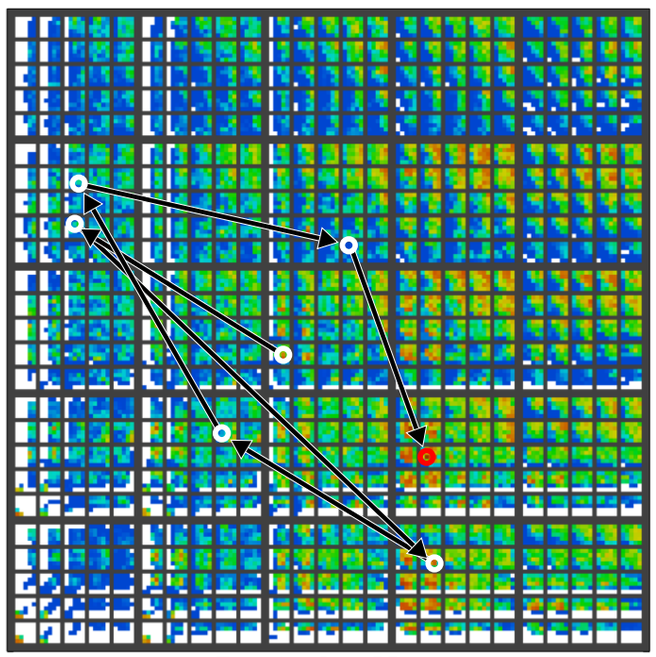
Stop when:
A real-world behavior is >90% of best untested point





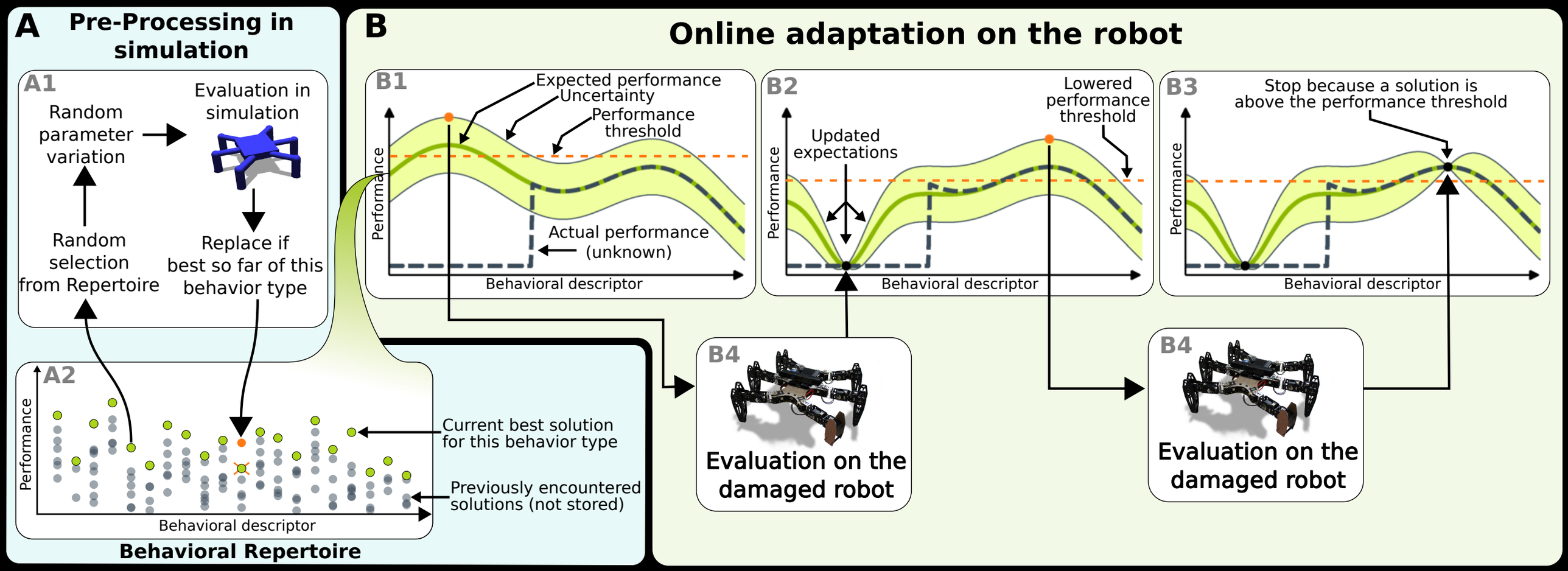


Initial Map



Posterior Map

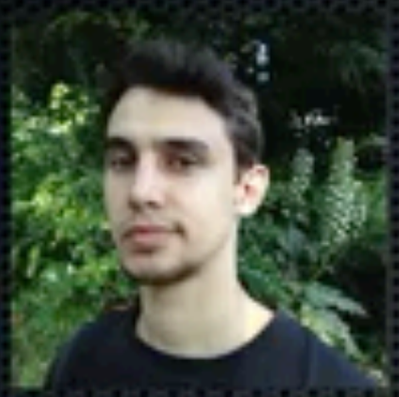
One-dimensional Example



Robots that can adapt like animals

Nature, 2015

which describes damage recovery via Intelligent Trial and Error



Antoine Cully
UPMC/CNRS
(France)

Jeff Clune
University of Wyoming
(USA)

Danesh Tarapore
UPMC/CNRS
(France)

Jean-Baptiste Mouret
UPMC/CNRS/Inria/UL
(France)

Revolutionary idea

What if we do not keep track of the single environment that we care about, but rather, evolve environments as we evolve solutions (policies)?

What if this joint search over policies and challenges results in better search in the policy space, by allowing mutations to happen across policies of different environments?

Paired Open-Ended Trailblazer (POET)- Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions

1. generating new environments $E(\cdot)$ from those currently active,
2. optimizing paired agents within their respective environments, and
3. attempting to transfer current agents θ from one environment to another.

Algorithm 2 POET Main Loop

Input: initial environment $E^{\text{init}}(\cdot)$, its paired agent denoted by policy parameter vector θ^{init}, learning rate α, noise standard deviation σ, iterations T, mutation interval N^{mutate}, transfer interval N^{transfer}
Initialize: Set `EA_list` empty
Add $(E^{\text{init}}(\cdot), \theta^{\text{init}})$ to `EA_list`
for $t = 0$ **to** $T - 1$ **do**
  if $t > 0$ **and** $t \mod N^{\text{mutate}} = 0$ **then**
    `EA_list` = MUTATE_ENVS(`EA_list`) **# new environments created by mutation**
  end if
  M = len(`EA_list`)
  for $m = 1$ **to** M **do**
    $E^m(\cdot), \theta_t^m$ = `EA_list`[m]
    $\theta_{t+1}^m = \theta_t^m +$ ES_STEP$(\theta_t^m, E^m(\cdot), \alpha, \sigma)$ **# each agent independently optimized**
  end for
  for $m = 1$ **to** M **do**
    if $M > 1$ **and** $t \mod N^{\text{transfer}} = 0$ **then**
      $\theta^{\text{top}} =$ EVALUATE_CANDIDATES$(\theta_{t+1}^1, \ldots, \theta_{t+1}^{m-1}, \theta_{t+1}^{m+1}, \ldots, \theta_{t+1}^M, E^m(\cdot), \alpha, \sigma)$
      if $E^m(\theta^{\text{top}}) > E^m(\theta_{t+1}^m)$ **then**
        $\theta_{t+1}^m = \theta^{\text{top}}$ **# transfer attempts**
      end if
    end if
    `EA_list`$[m] = (E^m(\cdot), \theta_{t+1}^m)$
  end for
end for

1. generating new environments $E(\cdot)$ from those currently active,
2. optimizing paired agents within their respective environments, and
3. attempting to transfer current agents θ from one environment to another.

Algorithm 2 POET Main Loop

Input: initial environment $E^{\text{init}}(\cdot)$, its paired agent denoted by policy parameter vector θ^{init}, learning rate α, noise standard deviation σ, iterations T, mutation interval N^{mutate}, transfer interval N^{transfer}
Initialize: Set `EA_list` empty
Add $(E^{\text{init}}(\cdot), \theta^{\text{init}})$ to `EA_list`
for $t = 0$ **to** $T - 1$ **do**
  if $t > 0$ **and** $t \mod N^{\text{mutate}} = 0$ **then**
    `EA_list` = MUTATE_ENVS(`EA_list`) **# new environments created by mutation**
  end if
  M = len(`EA_list`)
  for $m = 1$ **to** M **do**
    $E^m(\cdot), \theta_t^m$ = `EA_list`[m]
    $\theta_{t+1}^m = \theta_t^m + \text{ES\_STEP}(\theta_t^m, E^m(\cdot), \alpha, \sigma)$ **# each agent independently optimized**
  end for
  for $m = 1$ **to** M **do**

Algorithm 1 ES_STEP

Input: an agent denoted by its policy parameter vector θ, an environment $E(\cdot)$, learning rate α, noise standard deviation σ
Sample $\epsilon_1, \epsilon_2, \ldots, \epsilon_n \sim \mathcal{N}(0, I)$
Compute $E_i = E(\theta + \sigma\epsilon_i)$ for $i = 1, \ldots, n$
Return: $\alpha \frac{1}{n\sigma} \sum_{i=1}^{n} E_i \epsilon_i$

1. generating new environments $E(\cdot)$ from those currently active,
2. optimizing paired agents within their respective environments, and
3. attempting to transfer current agents θ from one environment to another.

Algorithm 2 POET Main Loop

Input: initial environment $E^{\text{init}}(\cdot)$, its paired agent denoted by policy parameter vector θ^{init}, learning rate α, noise standard deviation σ, iterations T, mutation interval N^{mutate}, transfer interval N^{transfer}
Initialize: Set `EA_list` empty
Add $(E^{\text{init}}(\cdot), \theta^{\text{init}})$ to `EA_list`
for $t = 0$ **to** $T - 1$ **do**
  if $t > 0$ **and** $t \mod N^{\text{mutate}} = 0$ **then**
    `EA_list` = MUTATE_ENVS(`EA_list`)
  end if
  M = len(`EA_list`)
  for $m = 1$ **to** M **do**
    $E^m(\cdot), \theta_t^m$ = `EA_list`[m]
    $\theta_{t+1}^m = \theta_t^m +$ ES_STEP$(\theta_t^m, E^m(\cdot), \alpha, \sigma)$ **# each agent independently optimized**
  end for
  for $m = 1$ **to** M **do**
    if $M > 1$ **and** $t \mod N^{\text{transfer}} = 0$ **then**
      $\theta^{\text{top}} =$ EVALUATE_CANDIDATES$(\theta_{t+1}^1, \ldots, \theta_{t+1}^{m-1}, \theta_{t+1}^{m+1}, \ldots, \theta_{t+1}^M, E^m(\cdot), \alpha, \sigma)$
      if $E^m(\theta^{\text{top}}) > E^m(\theta_{t+1}^m)$ **then**
        $\theta_{t+1}^m = \theta^{\text{top}}$ **# transfer attempts**
      end if
    end if
    `EA_list`$[m] = (E^m(\cdot), \theta_{t+1}^m)$
  end for
end for

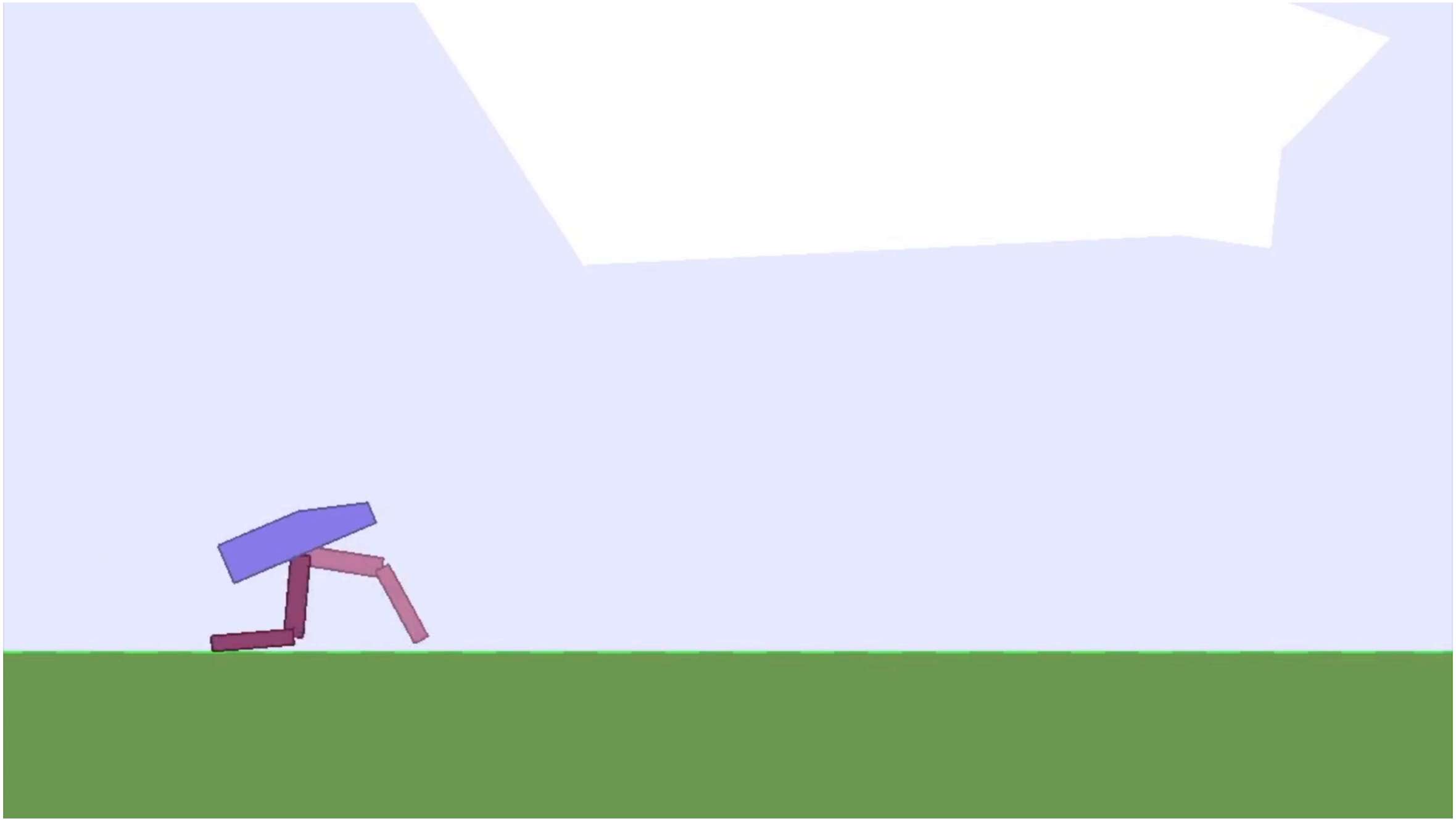
For each policy parameters, we take a policy update step in the new environment, and then we evaluate.

1. generating new environments $E(\cdot)$ from those currently active,
2. optimizing paired agents within their respective environments, and
3. attempting to transfer current agents θ from one environment to another.

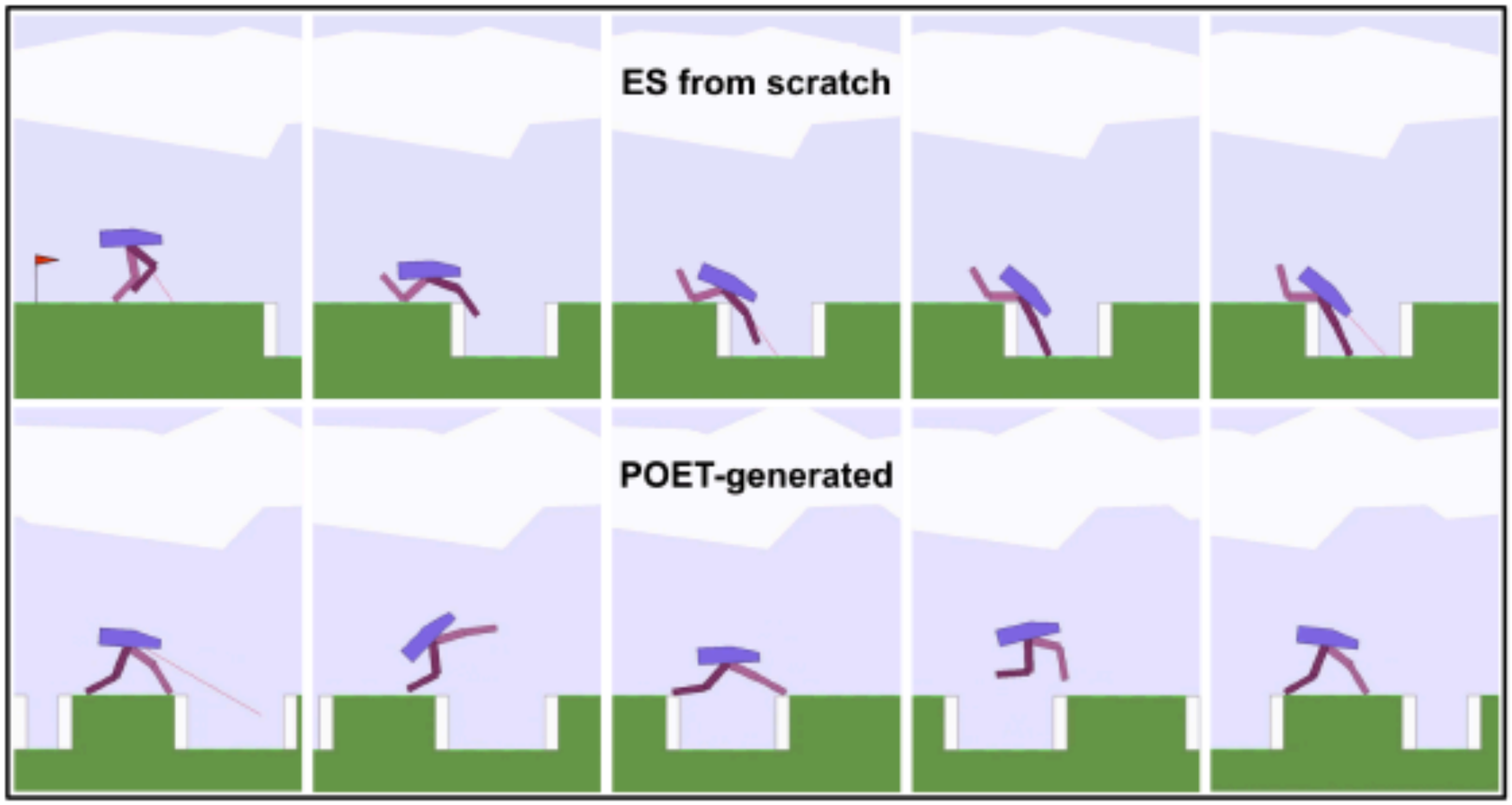
Algorithm 2 POET Main Loop

Input: initial environment $E^{\text{init}}(\cdot)$, its paired agent denoted by policy parameter vector θ^{init}, learning rate α, noise standard deviation σ, iterations T, mutation interval N^{mutate}, transfer interval N^{transfer}
Initialize: Set `EA_list` empty
Add $(E^{\text{init}}(\cdot), \theta^{\text{init}})$ to `EA_list`
for $t = 0$ **to** $T - 1$ **do**
  if $t > 0$ **and** $t \mod N^{\text{mutate}} = 0$ **then**
    `EA_list` = MUTATE_ENVS(`EA_list`) **# new environments created by mutation**
  end if
  M = len(`EA_list`)
  for $m = 1$ **to** M **do**
    $E^m(\cdot), \theta_t^m$ = `EA_list`[m]
    $\theta_{t+1}^m = \theta_t^m +$ ES_STEP$(\theta_t^m, E^m(\cdot), \alpha, \sigma)$ **# eacl**
  end for
  for $m = 1$ **to** M **do**
    if $M > 1$ **and** $t \mod N^{\text{transfer}} = 0$ **then**
      $\theta^{\text{top}} =$ EVALUATE_CANDIDATES$(\theta_{t+1}^1, \ldots, \theta_{t+1}^{m-1}, \theta_{t+1}^{m+1}, \ldots, \theta_{t+1}^M, E^m(\cdot), \alpha, \sigma)$
      if $E^m(\theta^{\text{top}}) > E^m(\theta_{t+1}^m)$ **then**
        $\theta_{t+1}^m = \theta^{\text{top}}$ **# transfer attempts**
      end if
    end if
    `EA_list`$[m] = (E^m(\cdot), \theta_{t+1}^m)$
  end for
end for

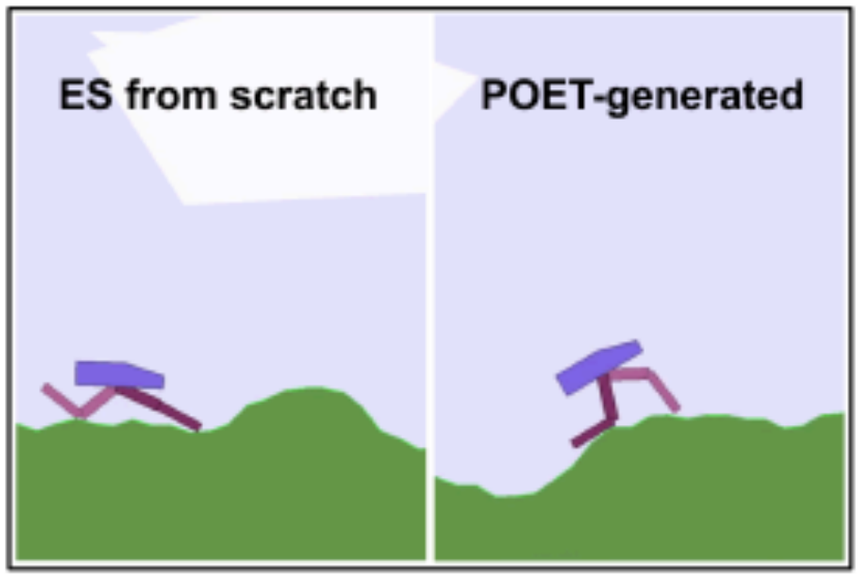
We ensure the new environments are not too hard and not too easy. How?



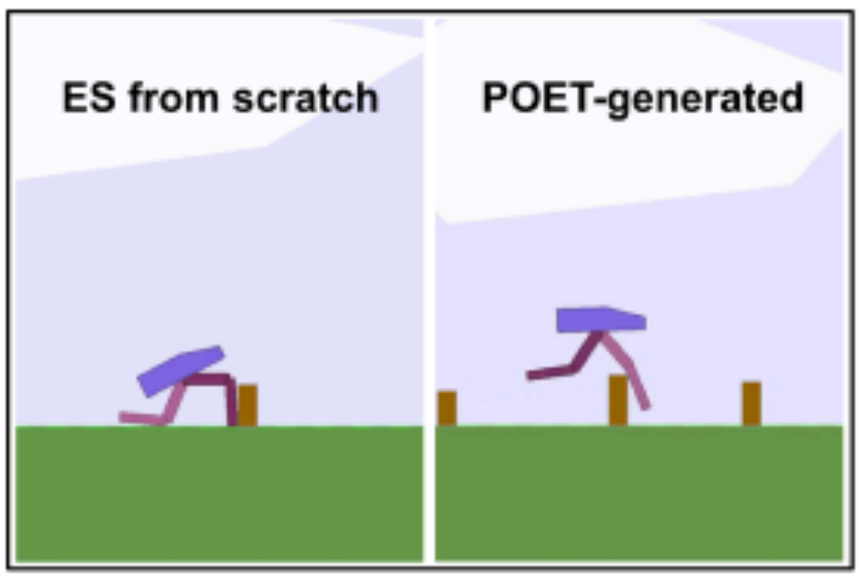
Joint search over policies and environments results in better policies than a similar search only across policies in a single environment.



(a) ES-/POET-generated agents attempting gaps

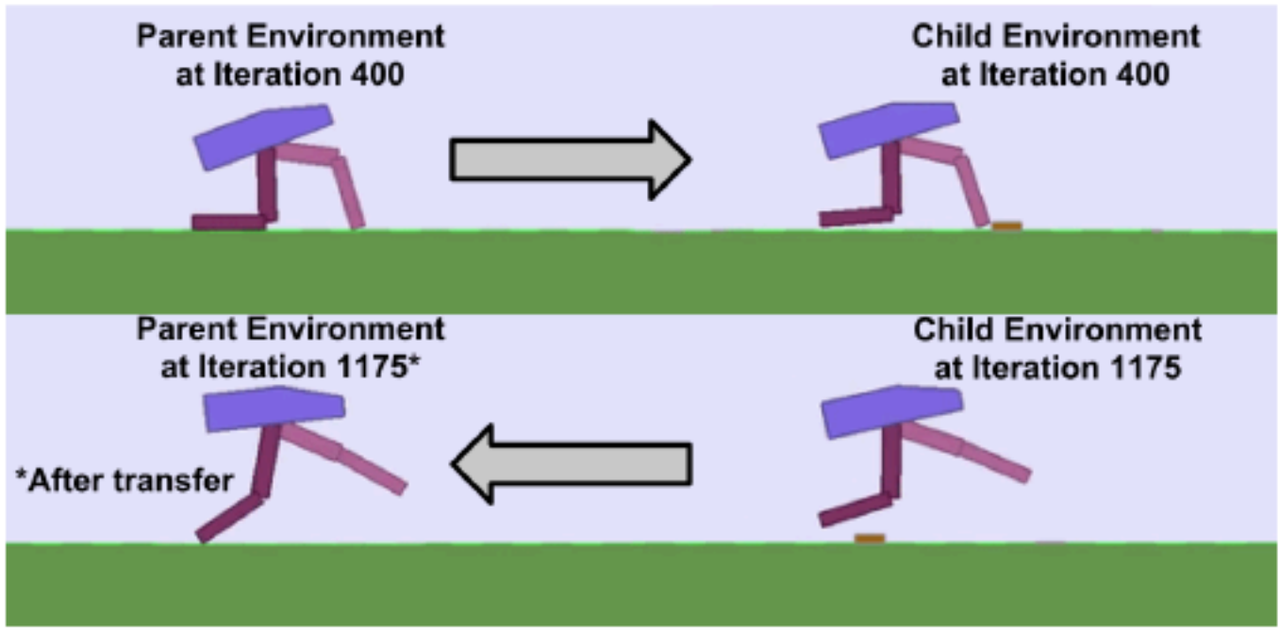


(b) ES-/POET-generated agents on rough surfaces

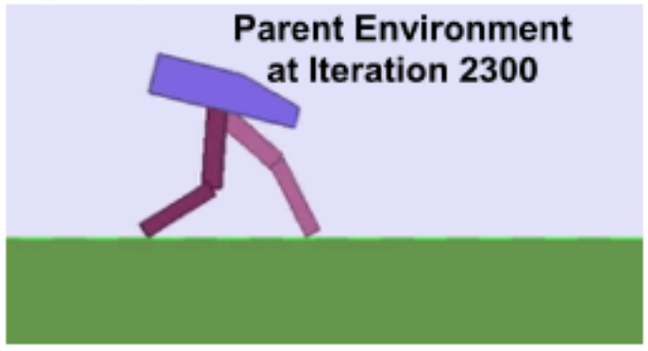


(c) ES-/POET-generated agents attempting stumps

Joint search over policies and environments results in better policies than a similar search only across policies in a single environment.



(a) Transfer from agent in parent environment to child environment and vice versa



(b) The walking gait of agent in parent environment at Iteration 2,300

Figure 7: **Synergistic two-way transfer between parent and child environments.** At iteration 400, a transfer from parent environment yields a child agent now learning in a stumpy environment, shown in the top row of (a). The child agent eventually learns to stand up and jump over the stumps, and at iteration 1,175 that skill is transferred *back* to the parent environment, depicted in the bottom row of (a). This transfer from the child environment back to the parent helps the parent agent learn a more optimal walking gait compared to its original one. Given the same amount of computation, the agent with the original walking gaits reaches a score of 309, while the one with the more optimal walking gait as illustrated in (b) reaches a score of 349.

Solutions

Episodic memory structures: non parametric book-keeping of states, their (Monte Carlo or bootstrapped) rewards, state to state connectivity trajectories.

Mixture of policies / map of policies (skills): fast pick the right one with few trials.

Recurrent policies that learn from a series of attempts.

Neural weight initialization that permits updating weights with very few gradient steps.

Solutions

Episodic memory structures: non parametric book-keeping of states, their (Monte Carlo or bootstrapped) rewards, state to state connectivity trajectories.

Mixture of policies / map of policies (skills): fast "pick the right one" with few trials.

Recurrent policies that learn from a series of attempts.

Neural weight initialization that permits updating weights with very few gradient steps.

Learning to Adapt: Meta-Learning for Model-Based Control

Ignasi Clavera [1 *] Anusha Nagabandi [1 *] Ronald S. Fearing [1] Pieter Abbeel [1] Sergey Levine [1] Chelsea Finn [1]

Finding neural weights that can be optimally adapted with a SMALL set of test interactions. Using robots in the real world does not permit a lot of interactions to adapt from anyways.

We consider a model based RL scenario. We wish to learn a model of the state transition function: $f(s, a; \theta) \rightarrow s'$ that can be *updated* using access to a small set of interactions D_{target}in a target (test) world, so that the resulting model has best performance in the target world.

Naive finetuning solution: minimize the training error in D_{target}. Unless D_target is large this does not generalize to D_{target}^{test}.

Learning to Adapt: Meta-Learning for Model-Based Control

Ignasi Clavera [1 *] Anusha Nagabandi [1 *] Ronald S. Fearing [1] Pieter Abbeel [1] Sergey Levine [1] Chelsea Finn [1]

Finding neural weights that can be optimally adapted with a SMALL set of test interactions. Using robots in the real world does not permit a lot of interactions to adapt from anyways.

We consider a model based RL scenario. We wish to learn a model of the state transition function: $f(s, a; \theta) \rightarrow s'$ that can be *updated* using access to a small set of interactions D_{target}in a target (test) world, so that the resulting model has best performance in the target world.

Meta-learning Idea: optimize for initial parameters \theta so that after the update rule using the small set of interactions D_target the resulting \theta' is high performing in the target world.

Adaptation rule for policies or dynamics

The adaptation rule denoted as $u(s_E^{t-M:t}, a_E^{t-M:t}, \theta)$ takes in the past M states and actions and a prior dynamics neural parameters or policy θ, and outputs an updated dynamics parameter vector or policy θ' such that the resulting dynamics function or policy are more accurate predictor for the dynamics / effective policy at the current time step.

Earlier we saw recurrence as the update rule that takes initial hidden state h to updated hidden state h'. Here we will be using gradient descent steps as our update rule, so normal finetuning. Is just we will be seeking for neural weights \theta for which fine-tuning works.

Adaptation rule for policies or dynamics

The adaptation rule denoted as $u(s_E^{t-M:t}, a_E^{t-M:t}, \theta)$ takes in the past M states and actions and a prior dynamics neural parameters or policy θ, and outputs an updated dynamics parameter vector or policy θ' such that the resulting dynamics function or policy are more accurate predictor for the dynamics / effective policy at the current time step.

Algorithm 1 Meta-training a prior for online adaptation

Require: $\rho_{\mathcal{E}}$ distribution over tasks

Require: Learning rate $\beta \in \mathbb{R}^+$

Require: Dataset $\mathcal{D}$

1: Randomly initialize θ

2: **while** not done **do**

3: Sample a batch of E tasks $\mathcal{E}_i \sim \rho(\mathcal{E}_i)$

4: **for all** $\mathcal{E}_i$ **do**

5: $s_{\mathcal{E}_i}^{t-M:t}, a_{\mathcal{E}_i}^{t-M:t-1}, s_{\mathcal{E}_i}^{t:t+K}, a_{\mathcal{E}_i}^{t:t+K} \sim \mathcal{D}$

6: $\theta'_{\mathcal{E}_i} \leftarrow u(s_{\mathcal{E}_i}^{t-M:t-1}, a_{\mathcal{E}_i}^{t-M:t-1}, \theta)$

7: Compute $\mathcal{L}_{\mathcal{E}_i}(f_{\theta'_{\mathcal{E}_i}}, s_{\mathcal{E}_i}^{t:t+K+1}, a_{\mathcal{E}_i}^{t:t+K})$

8: **end for**

9: $\theta \leftarrow \theta - \beta \nabla_\theta \frac{1}{E} \sum \mathcal{L}_{\mathcal{E}_i}(f_{\theta'_{\mathcal{E}_i}}, s_{\mathcal{E}_i}^{t:t+K+1}, a_{\mathcal{E}_i}^{t:t+K})$

10: $u \leftarrow u - \beta \nabla_u \frac{1}{E} \sum \mathcal{L}_{\mathcal{E}_i}(f_{\theta'_{\mathcal{E}_i}}, s_{\mathcal{E}_i}^{t:t+K+1}, a_{\mathcal{E}_i}^{t:t+K})$

11: **end while**

12: Return parameters θ and/or update rule u

e.g., environments with similar observations s and actions a but different physics parameters

sample a few interactions in the environment E_i

adaptation rule

the loss of the updated net

gradient descent

$$u(s^{t-M:t}, a^{t-M:t-1}, \theta) = \theta - \alpha \nabla_\theta \left(\frac{1}{M} \sum_{m=1}^{M} ||f_\theta(s^{t-m}, a^{t-m}) - \Delta s^{t-m}||_2^2 \right).$$

Algorithm 2 Online Model Adaptive Control

Require: Dynamics model prior parameters θ
Require: Update rule u
Require: Recent experience $s^{t-M:t-1}, a^{t-M:t-1}$
Require: Number of samples n_{cand}, horizon H

$f_{\theta'} \leftarrow u(s^{t-M:t}, a^{t-M:t-1}, \theta)$
Sample sequences of actions $\{A_1^t, ..., A_{n_{cand}}^t\}$
Initialize total returns $R_i = 0$
$\hat{s}_i{}^t \leftarrow s_i^t$ for $i = 1...n_{cand}$
for $h = 0...H-1$ **do**
  for $i = 1...n_{cand}$ **do**
    $R_i \leftarrow R_i + \gamma^h r(\hat{s}_i^{t+h}, a_i^{t+h})$
    $\hat{s}_i^{t+h+1} \leftarrow f_{\theta'}(\hat{s}_i^{t+h}, a_i^{t+h})$
  end for
end for
$i^* \leftarrow \arg\max_i R_i$
Return $A_{i^*}^t$


Learning to Adapt in Dynamic, Real-World
Environments through Meta-RL