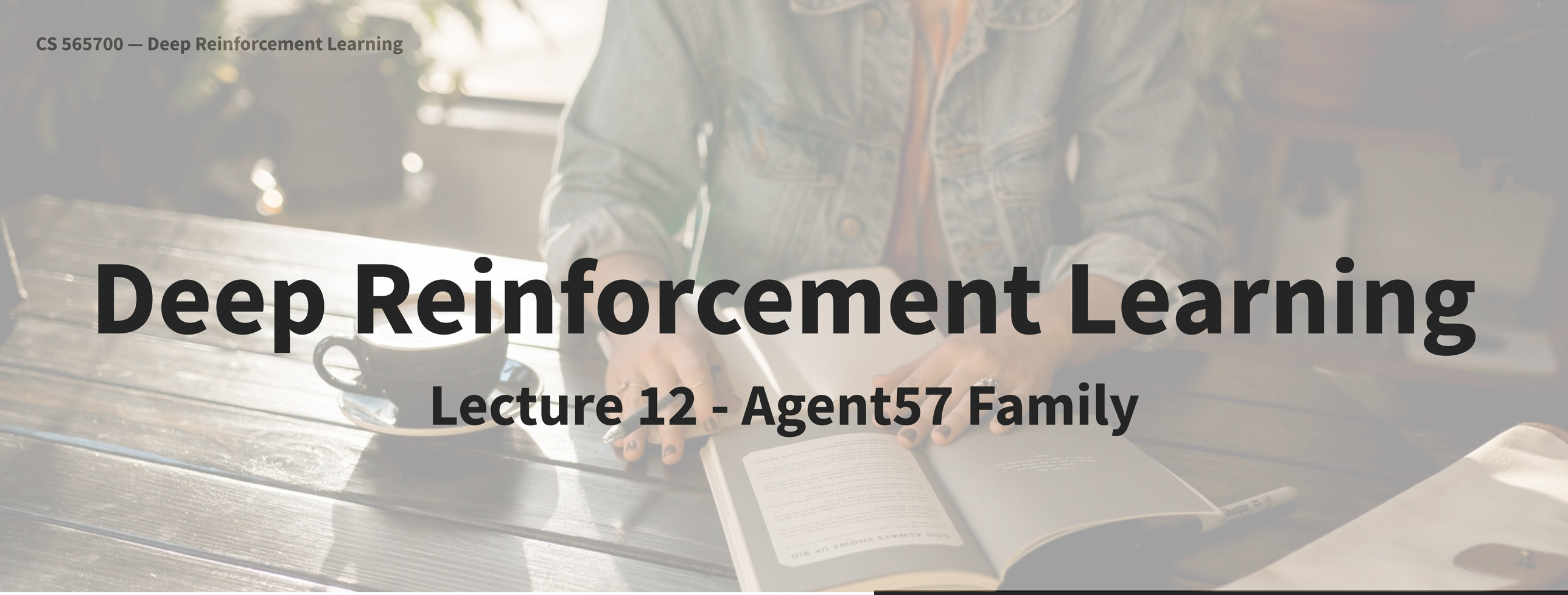


Deep Reinforcement Learning

Lecture 12 - Agent57 Family

國立清華大學
NATIONAL TSING HUA UNIVERSITY

National Tsing Hua University
Department of Computer Science

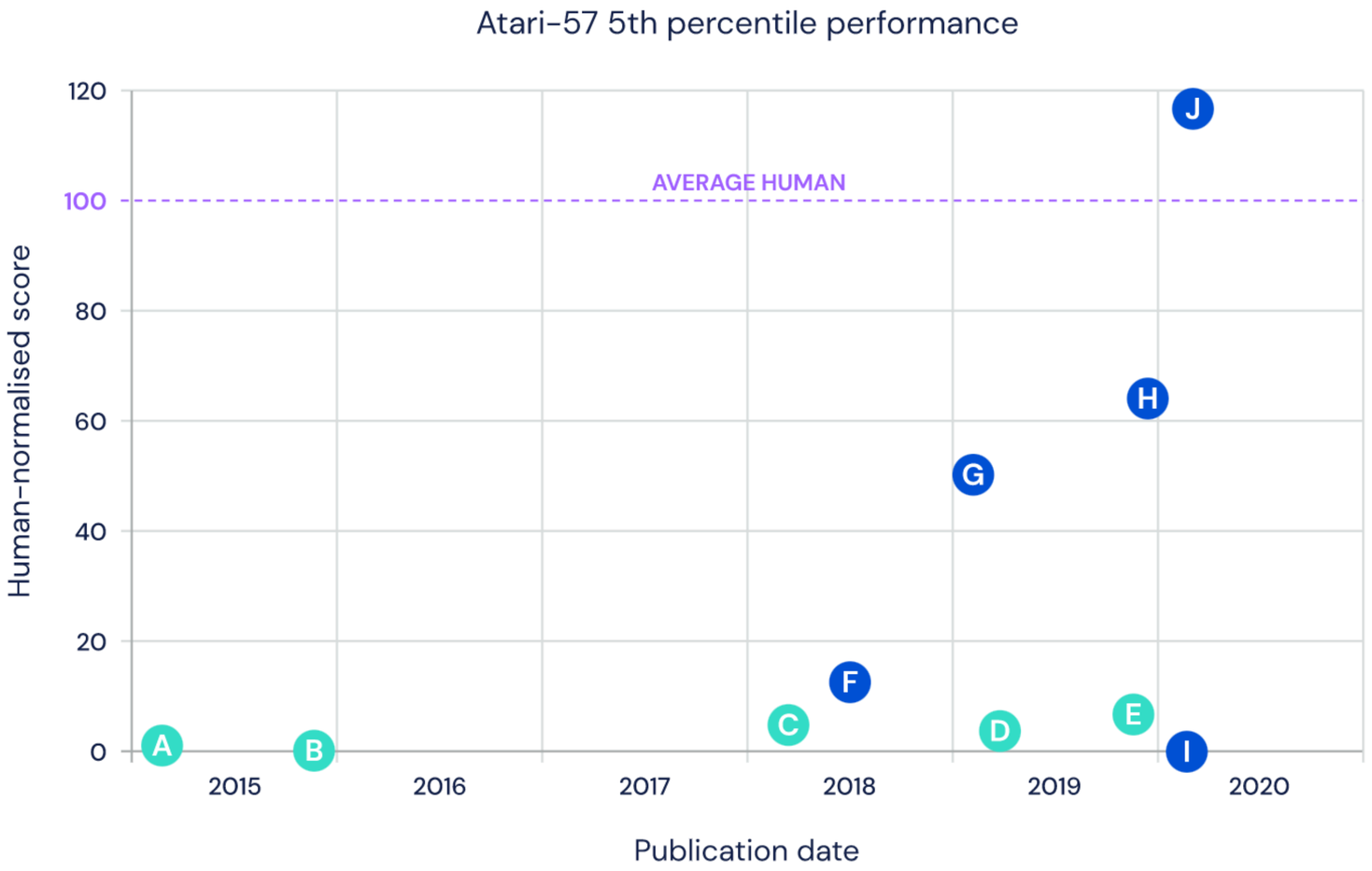
Prof. Chun-Yi Lee

Outline

- **Agent57 Family Overview**
- DQN Series
- R2D2
- Never Give Up
- Multi-Armed Bandit
- Agent57

Performance Comparison on Atari Games

A comparison of different RL methods on Atari Environments



Agents that use a distributed setup are **blue**, whereas single-actor agents are **teal**. The 5th percentile analysis shows that state of the art algorithms such as MuZero and R2D2 perform dramatically below the human benchmark (purple dotted line), whereas Agent57 performs better than humans on the hardest Atari games.

Single-actor agents

- A DQN
- B Prioritised Dueling
- C Rainbow
- D C51-IDS
- E FQF

Distributed agents

- F ApeX
- G R2D2
- H NGU
- I MuZero
- J Agent57

The Metric for Comparing RL Agents

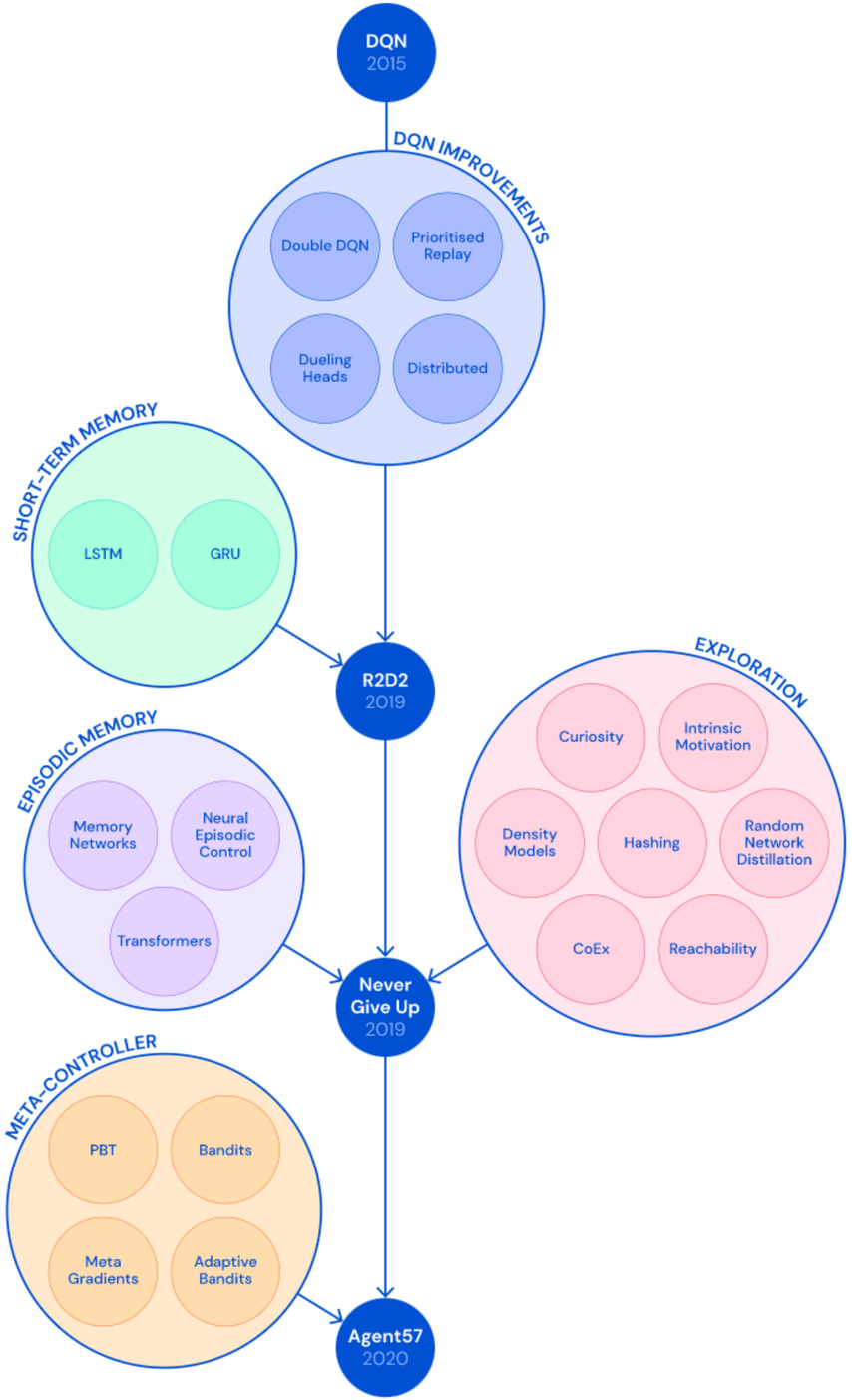
The performance of an agent is compared across different tasks

- The performance of agents are usually measured against human beings
  - This is accomplished by normalized to the average human scores
- The performance of an RL method is evaluated across multiple tasks
  - An RL method may perform very well on some tasks, while failing on others (e.g., **Agent A**)
  - An RL method may perform equally nice for all tasks, including easy and hard ones (e.g., **Agent B**)
- Which one is better?
  - Can we design an RL method that has outstanding performance on all tasks?

Agent57 Ancestry

Evolution of RL Methods for Atari Games

- The figure on the right-hand side shows the conceptual advancements to **DQN** that have resulted in the development of more generally intelligent agents.
- Since 2012 (**DQN**), the research community has developed many extensions and alternatives to **DQN**.
- Despite these advancements, however, all deep reinforcement learning agents have consistently failed to score in four games: **Montezuma's Revenge**, **Pitfall**, **Solaris** and **Skiing**.



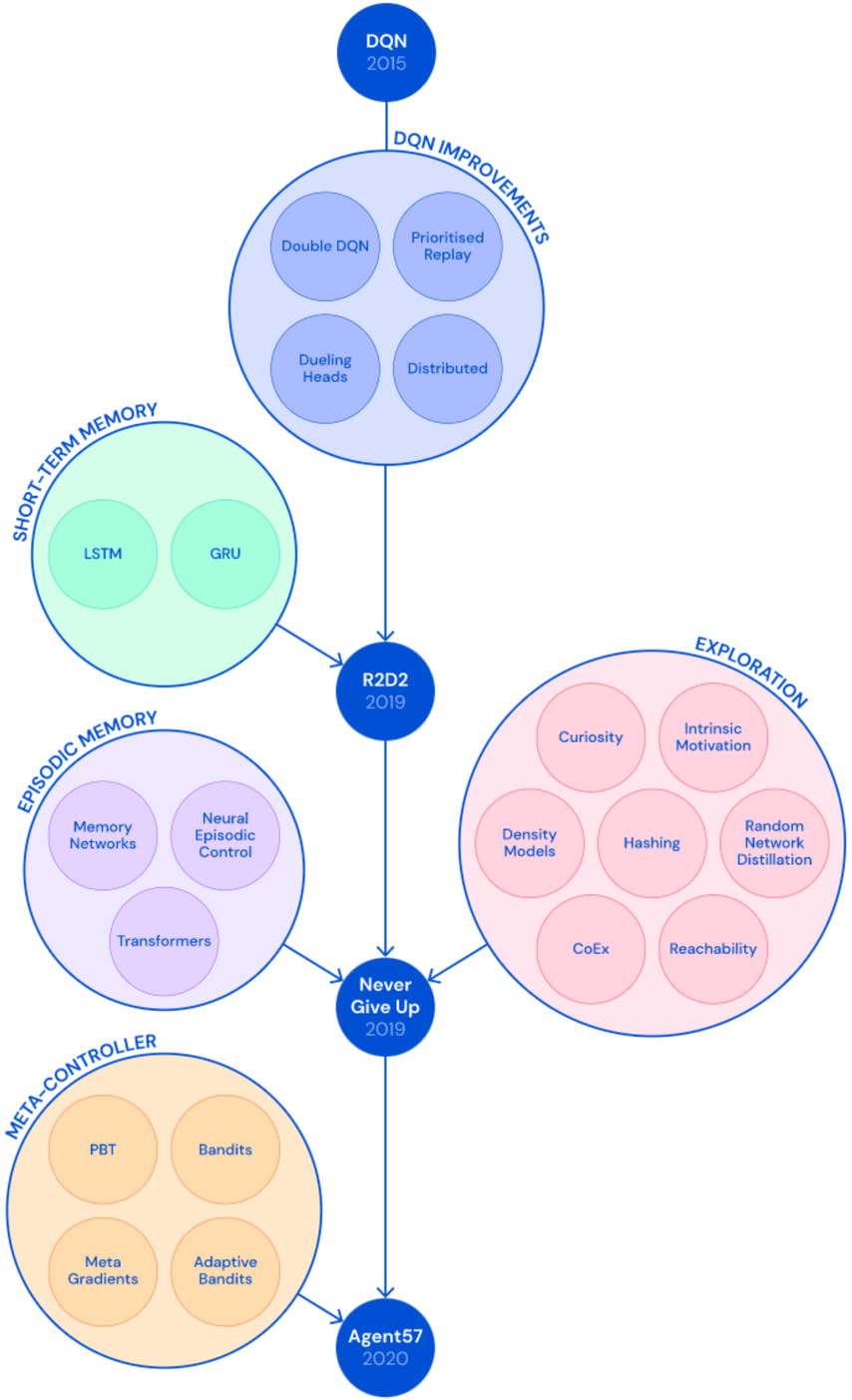
Agent57 Ancestry

Evolution of RL Methods for Atari Games

- **Montezuma's Revenge** and **Pitfall** require extensive exploration to obtain good performance
    - Exploration-exploitation problem

- **Solaris** and **Skiing** are long-term credit assignment problems
    - It's challenging to match the consequences of an agents' actions to the rewards it receives
    - Agents must collect information over long time scales to get the feedback necessary to learn
    - Example: Agent is able to play Solaris well (Link)

- For Agent57 to tackle these four challenging games in addition to the other Atari57 games, several changes to DQN were necessary.



Outline

- Agent57 Family Overview
- DQN Series
- R2D2
- Never Give Up
- Multi-Armed Bandit
- Agent57

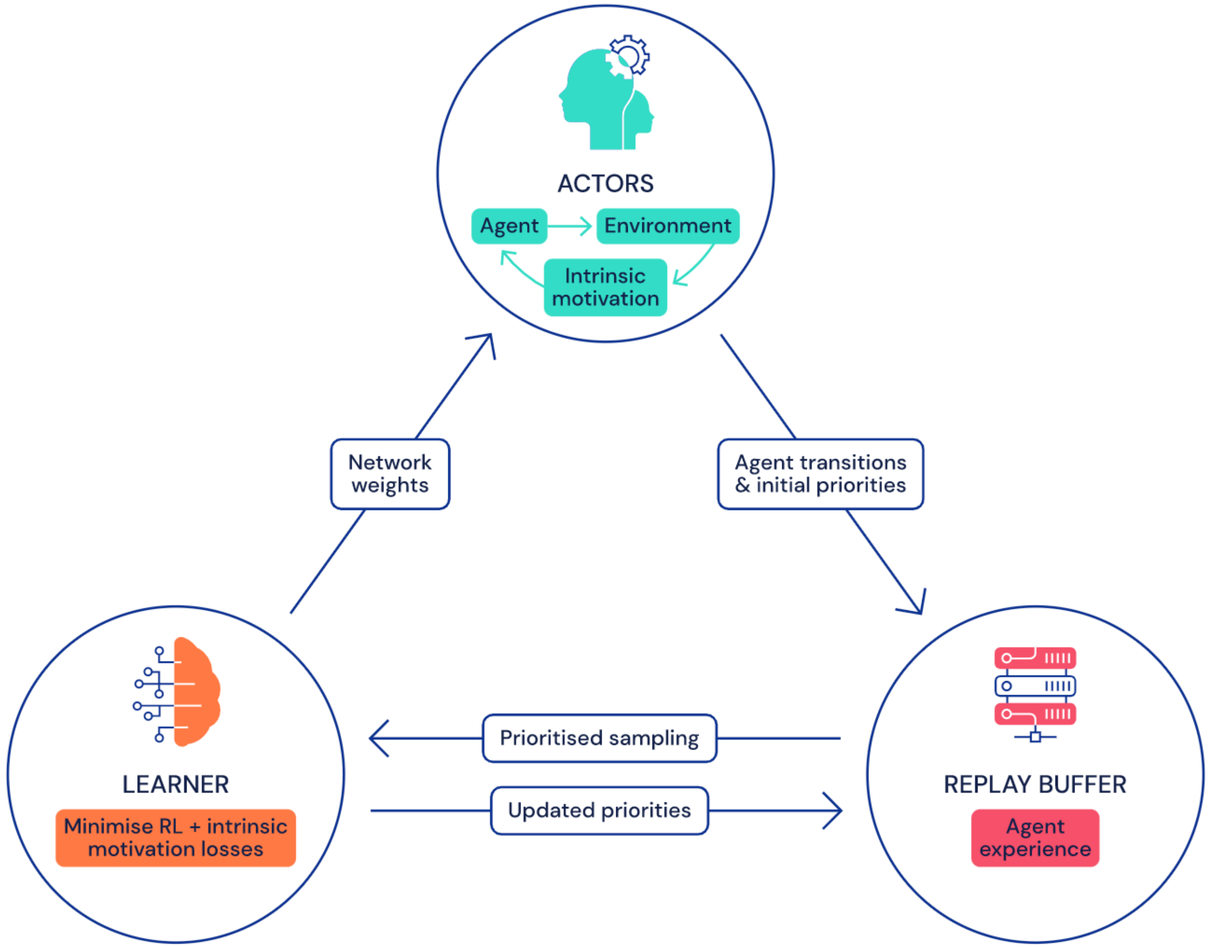
DQN Series and the Improvements

Single-Actor DQN Methods

- There have been many famous single actor based DQN methods proposed in the previous years
  - **Double DQN**
  - **Dueling DQN**
  - **Prioritized Experience Replay for DQN**
  - **Deep Recurrent Q-Network (DRQN)**
  - **Rainbow DQN**

DQN Series and the Improvements

Distributed Agents Based DQN Methods

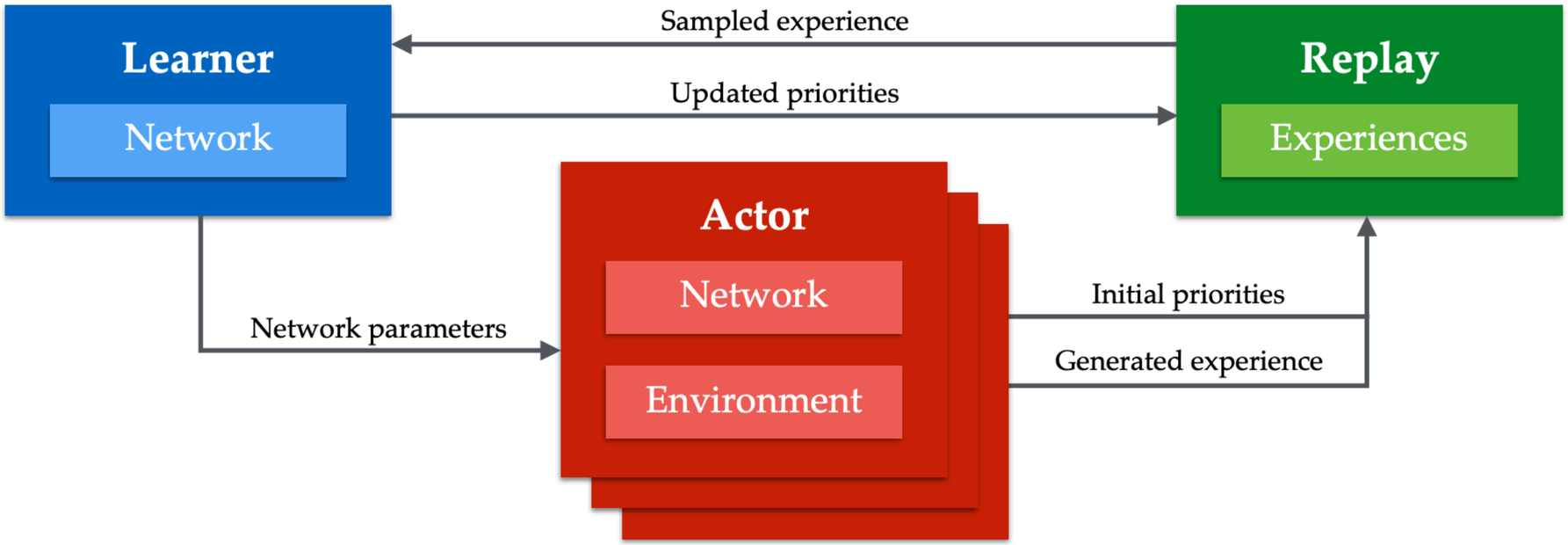


- There have been several distributed variants of DQN introduced previously
  - **Gorila DQN**
  - **ApeX**
  - **IMPALA**
- There have also been many other distributed methods discussed before
  - **A3C and A2C**
- Distributed agents could be run on many computers simultaneously
  - This allowed agents to acquire and learn from experience more quickly
  - This also enables researchers to rapidly iterate on ideas

Ape-X

Distributed Prioritized Experience Replay

- A distributed RL + special prioritized replay buffer
- It can be used on the algorithms with replay buffer (DQN, DDPG …)



Ape-X

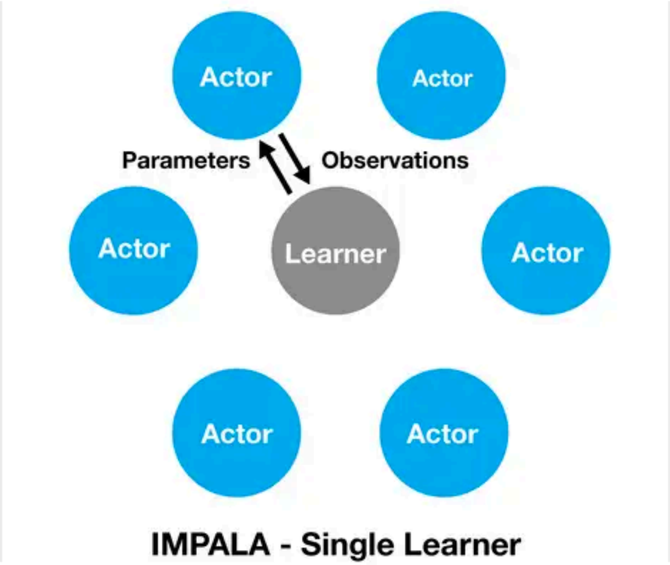
The tricks used in Ape-X

- Distributed replay with prioritized sampling
- N-step return targets
- Double Q-learning
- Dueling DQN Network architecture
- Four-frame stacking

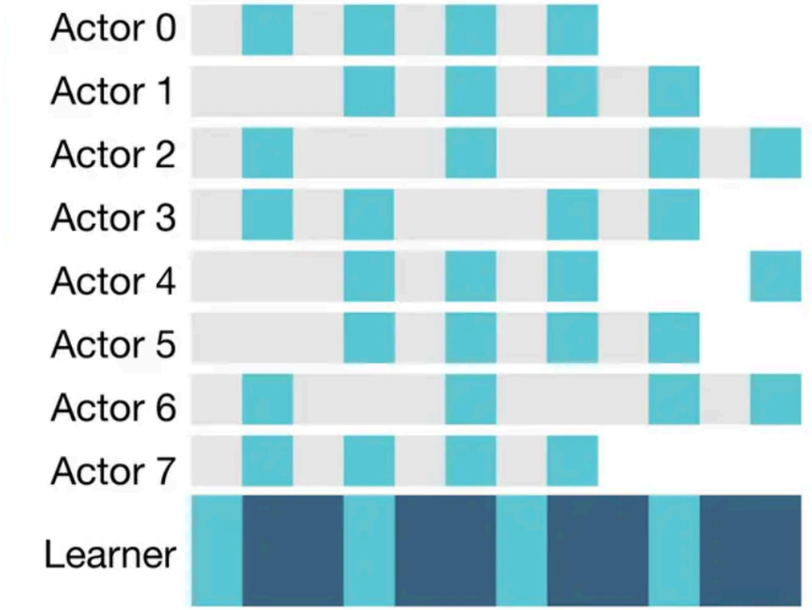
IMPALA

IMPALA — Single Learner Scheme

- For each actor, they do not need to wait for each other or the learner
- For the learner, it can always update itself without waiting for the actors
- More efficient in CPU utilization



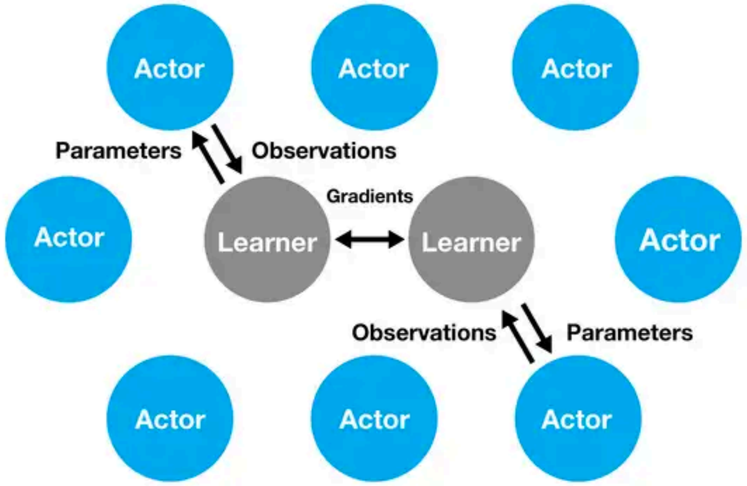
IMPALA - Single Learner



IMPALA

IMPALA — Multi Learner Scheme

- Since IMPALA decouples the learning and the acting phase, it could have multi learners
- Each learner could maintain its own policy and share gradients to each other

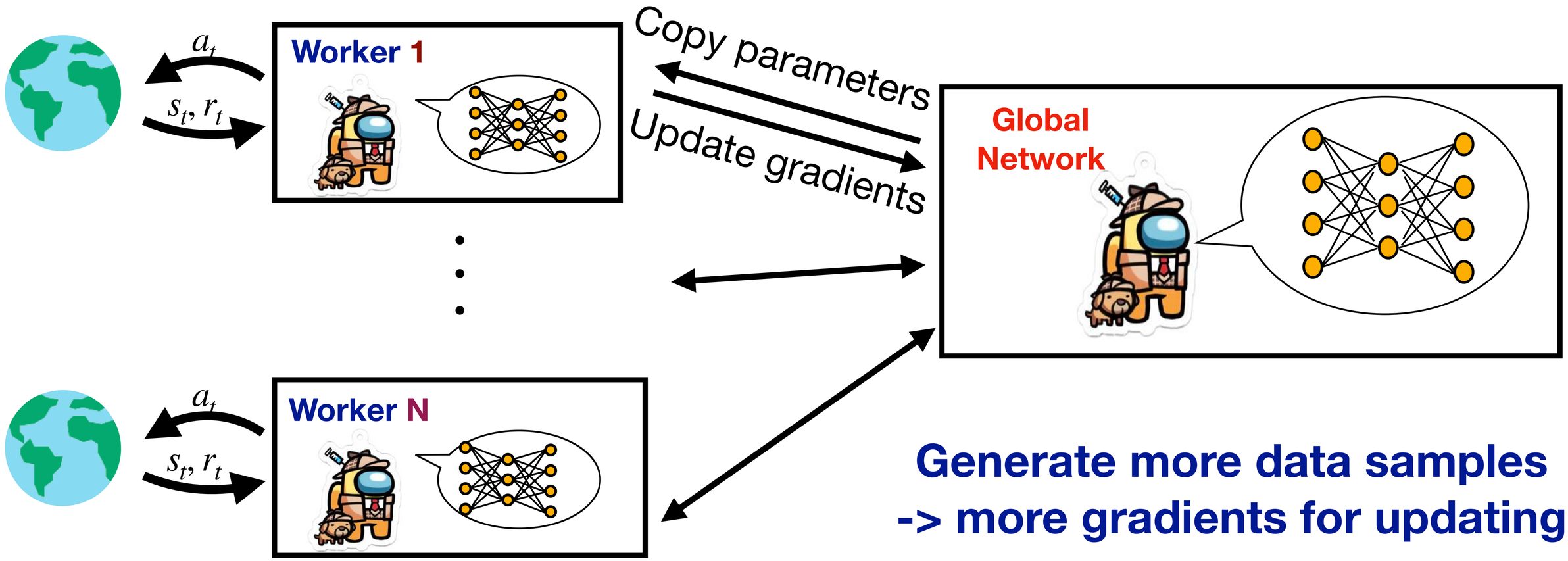


IMPALA - Multiple Learners

Asynchronous Advantage Actor Critic

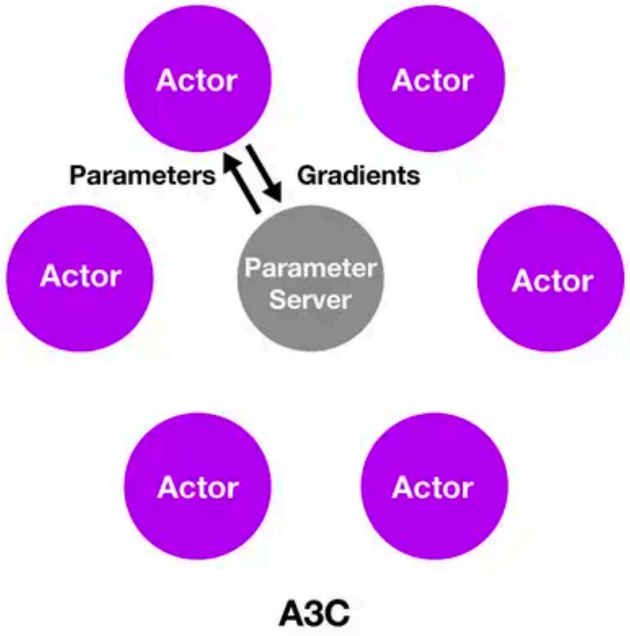
The A3C Framework

- A3C

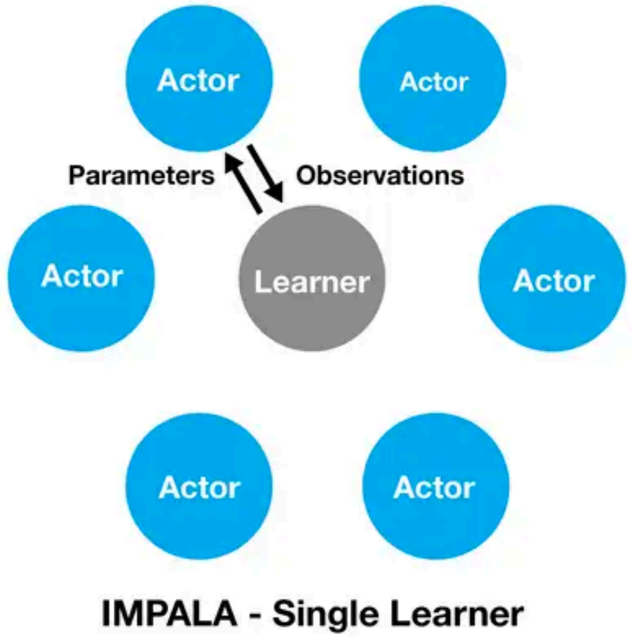


IMPALA

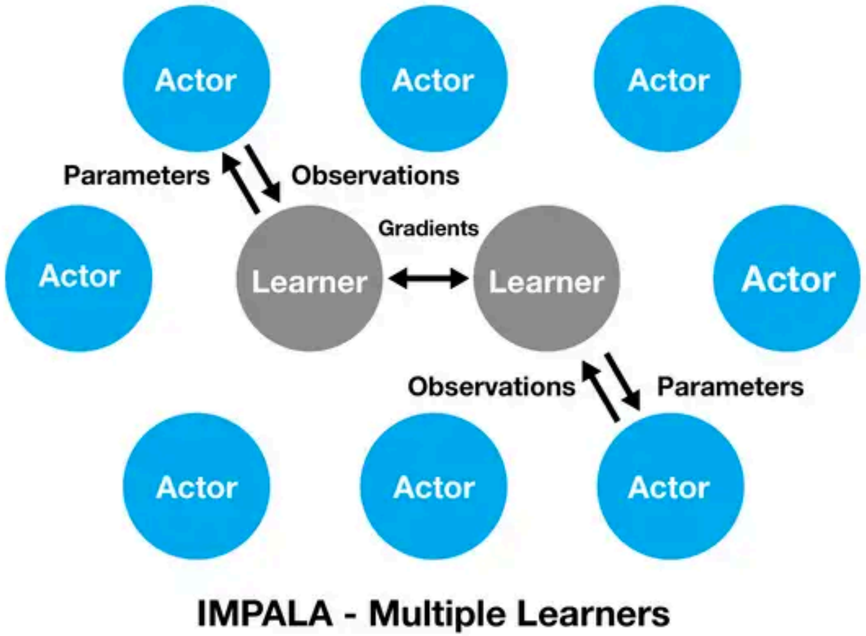
A Comparison of Different Distributed RL Schemes



A3C



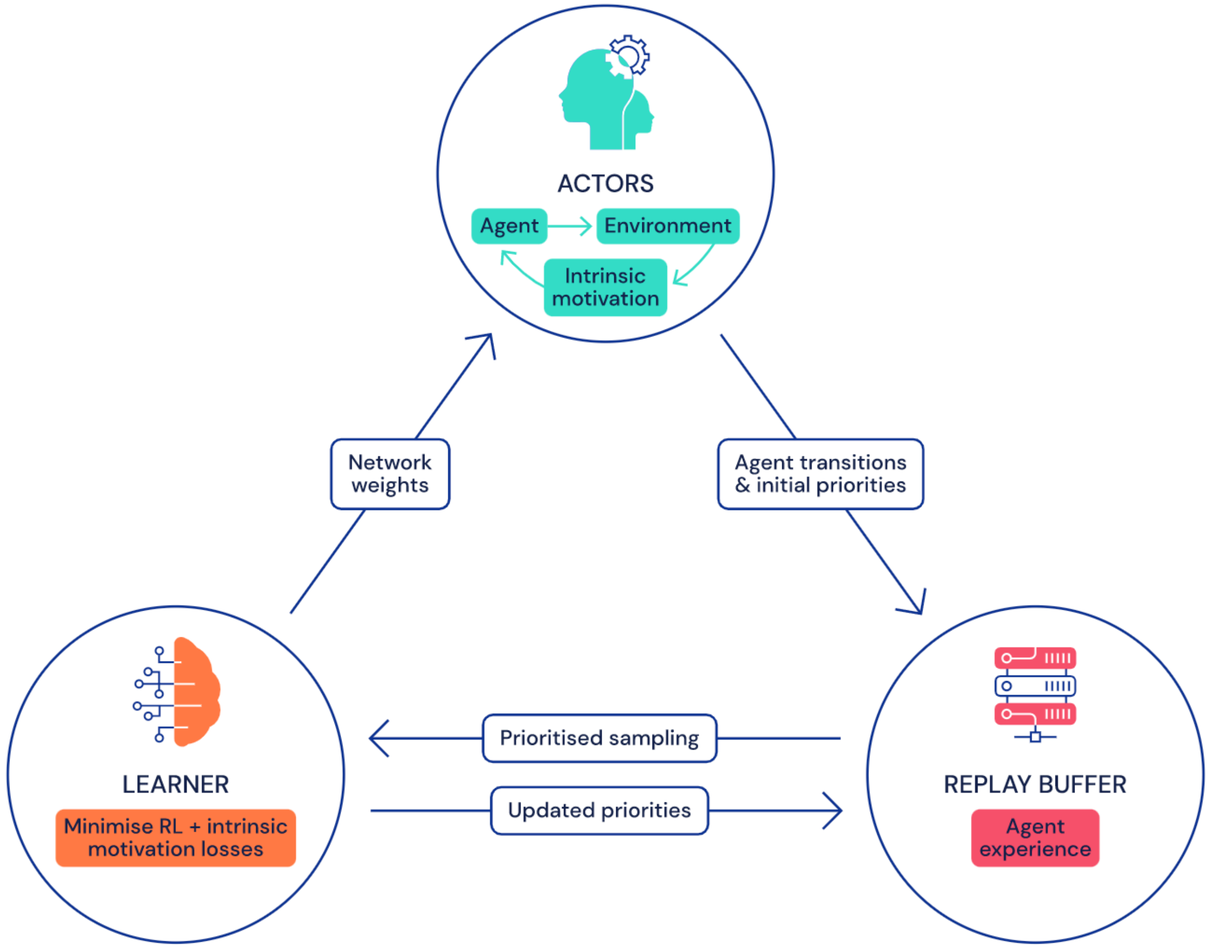
IMPALA - Single Learner



IMPALA - Multiple Learners

DQN Series and the Improvements

Distributed Agents Based DQN Methods



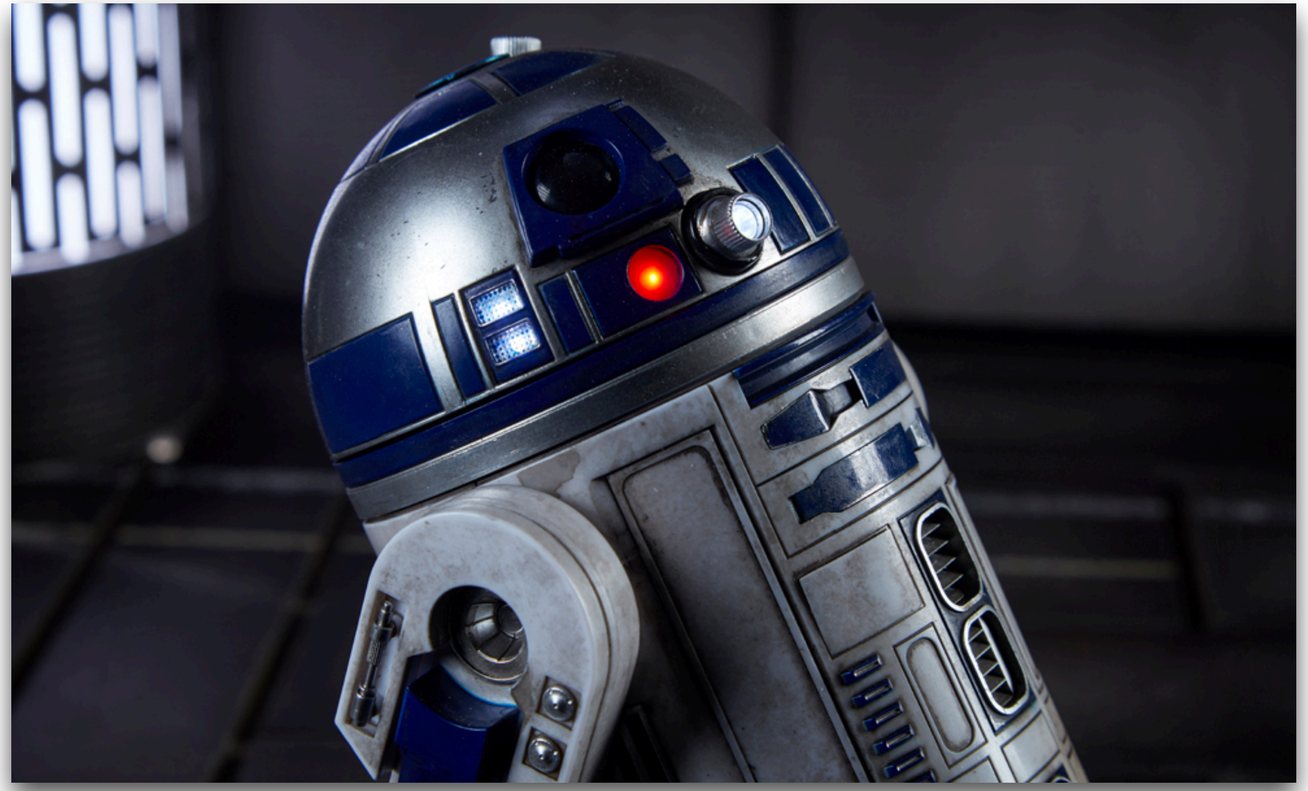
- Agent57 is also a distributed RL method that decouples **the data collection** and **the learning processes**
- Many actors interact with independent copies of the environment, feeding data to a central **replay buffer** in the form of a **prioritized experience replay**
- A learner then samples training data from this **replay buffer**
- It then updates the parameters of its neural network by minimizing losses
- Each actor shares the same network architecture as the learner, but with **its own copy of the weights**
- **The learner weights are sent to the actors frequently**, allowing them to **update their own weights** in a manner determined by their individual priorities

Outline

- Agent57 Family Overview
- DQN Series
- **R2D2**
- Never Give Up
- Multi-Armed Bandit
- Agent57

Recurrent Experience Replay in Distributed Reinforcement Learning

The method proposed is also named “R2D2”



- Published at **ICLR 2019**
- The proposed technique is Recurrent Replay Distributed DQN (**R2D2**)
- The paper investigates the training of RNN-based RL agents from distributed prioritized experience replay
  - Demonstrated the effect of experience replay on parameter lag, which may cause
    - Representational drift
    - Recurrent state staleness
    - Diminished training stability and performance
  - Performed an empirical study for mitigating the above issues

Recurrent Experience Replay in Distributed Reinforcement Learning

The techniques used in R2D2 (Similar to Ape-X)

- Distributed replay with prioritized sampling
  - The number of actors up to 256
- N-step return targets
  - **n** is set to **five** in R2D2
- Double Q-learning
- Dueling DQN Network architecture
- Use four frame-stacking on Atari, and single frame on DMLab
- **Introducing an LSTM layer after the convolution stack**

Recurrent Experience Replay in Distributed Reinforcement Learning

N-step target of Q-value function

- No reward clipping, but instead use invertible value function rescaling of the form:

$$h(x) = \text{sign}(x)(\sqrt{|x| + 1} - 1) + \epsilon x$$

  - **The concept of invertible value function was used in Ape-X as well**

- This results in the following n-step targets for the Q-value function.

$$\hat{y}_t = h\left(\sum_{k=0}^{n-1} r_{t+k}\gamma^k + \gamma^n h^{-1}\left(Q(s_{t+n}, a^*; \theta^-)\right)\right), \quad a^* = \arg\max_a Q(s_{t+n}, a; \theta).$$

Recurrent Experience Replay in Distributed Reinforcement Learning

Problem in RNN/LSTM based training strategies

- R2D2 compares two strategies of training an LSTM from replay experience
  - **Using a zero start state to initialize the network at the beginning of sampled sequences**
    - Simple
    - Atypical initial recurrent state (‘initial recurrent state mismatch’), which may limit its ability to fully rely on its recurrent state and learn to exploit long temporal correlations
  - **Replaying whole episode trajectories**
    - Avoids the problem of finding a suitable initial state
    - Practical, computational, and algorithmic issues due to varying and potentially environment-dependent sequence length

Recurrent Experience Replay in Distributed Reinforcement Learning

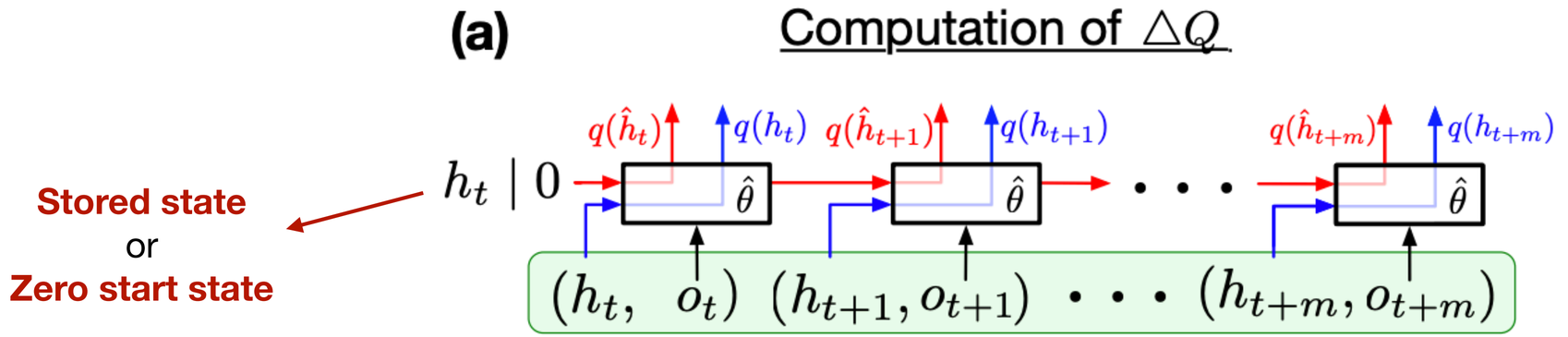
New strategies for training RNN from random sampled replay sequences

- **Stored state**
  - Store the recurrent states in replay and use them to initialize the network
  - Remedies the weakness of the zero start state strategy
  - However, it may suffer from ‘representational drift’ leading to ‘recurrent state staleness’ (out-dated)
    - This is because the stored recurrent state generated by a sufficiently old network could differ significantly from a typical state produced by a more recent version

- **Burn-in**
  - Use a portion of the replay sequence only for unrolling the network and producing a start state
  - Update the network only on the remaining part of sequence.
    - This allows the network to partially recover from a poor start state (zero, or stored but stale)

Recurrent Experience Replay in Distributed Reinforcement Learning

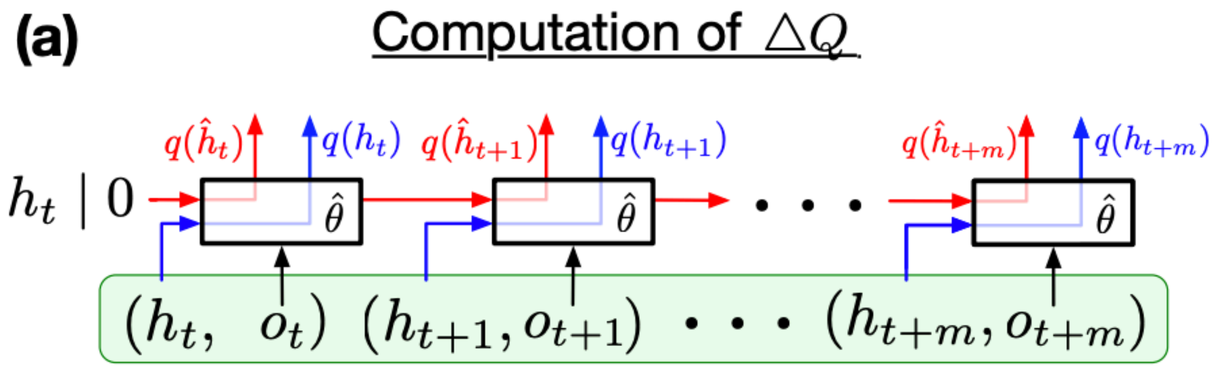
Q-value discrepancy ΔQ as a measure for recurrent state staleness.



- $o_t, \ldots, o_{t+m}$ and $h_t, \ldots, h_{t+m}$ denote the replay sequence of observations and stored current states, where m is the length of replay
- $h_{t+1} = h(o_t, h_t, \theta)$ and $q(h_t, \theta)$ denote the recurrent state output and Q-value output generated by the RNN
- $\hat{h}_t$ is the hidden state and is initialized under two strategies $\hat{h}_t = 0$ or $\hat{h}_t = h_t$
- $\hat{h}_{t+i} = h(o_{t+i-1}, \hat{h}_{t+i-1}, \hat{\theta})$ is computed by unrolling the network with parameter $\hat{\theta}$ on the sequence $o_t, \ldots, o_{t+l+m-1}$, where l is burn-in prefix
- **Red line** — Q-value estimated by hidden state $\hat{h}_t$ generated by two strategies, parameterized by $\hat{\theta}$
- **Blue line** — Q-value estimated by recurrent state h_t restored in the replay, parameterized by $\hat{\theta}$

Recurrent Experience Replay in Distributed Reinforcement Learning

Q-value discrepancy ΔQ as a measure for recurrent state staleness.



Q-value discrepancy

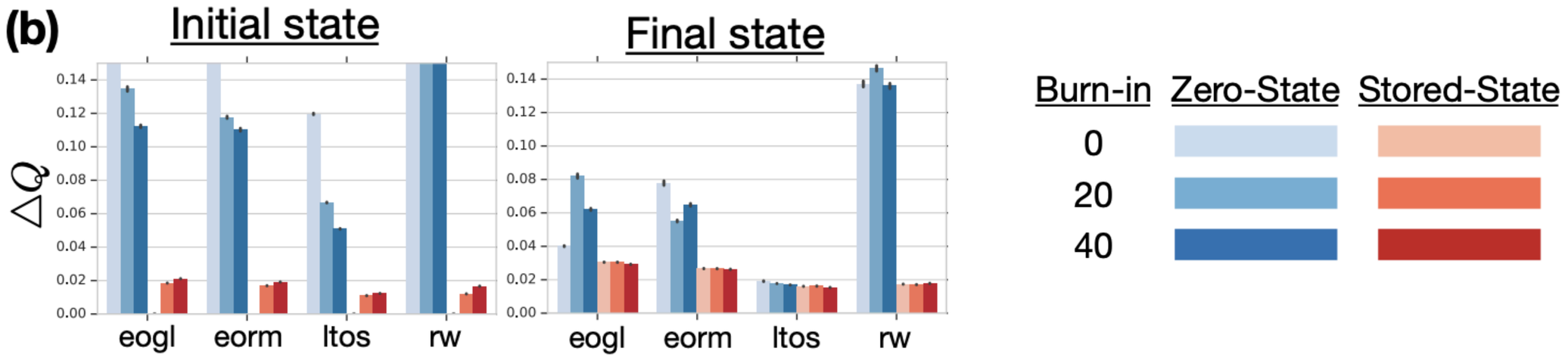
$$\Delta Q = \frac{\|q(\hat{h}_{t+i}; \hat{\theta}) - q(h_{t+i}; \hat{\theta})\|_2}{|\max_{a,j}(q(\hat{h}_{t+j}; \hat{\theta}))_a|}$$

Normalized

- Estimate Q-values for the first $(i = l)$ and the last $(i = l + m - 1)$ states of the non-burn-in part
- Not directly comparing the Q-values produced by acting and training time, i.e., $q(h_t; \theta)$ and $q(\hat{h}, \hat{\theta})$, as these can naturally be distinct as the agent being trained.
- Focus on the difference that results from applying the same network (parameterized by $\hat{\theta}$) to the distinct recurrent state.

Recurrent Experience Replay in Distributed Reinforcement Learning

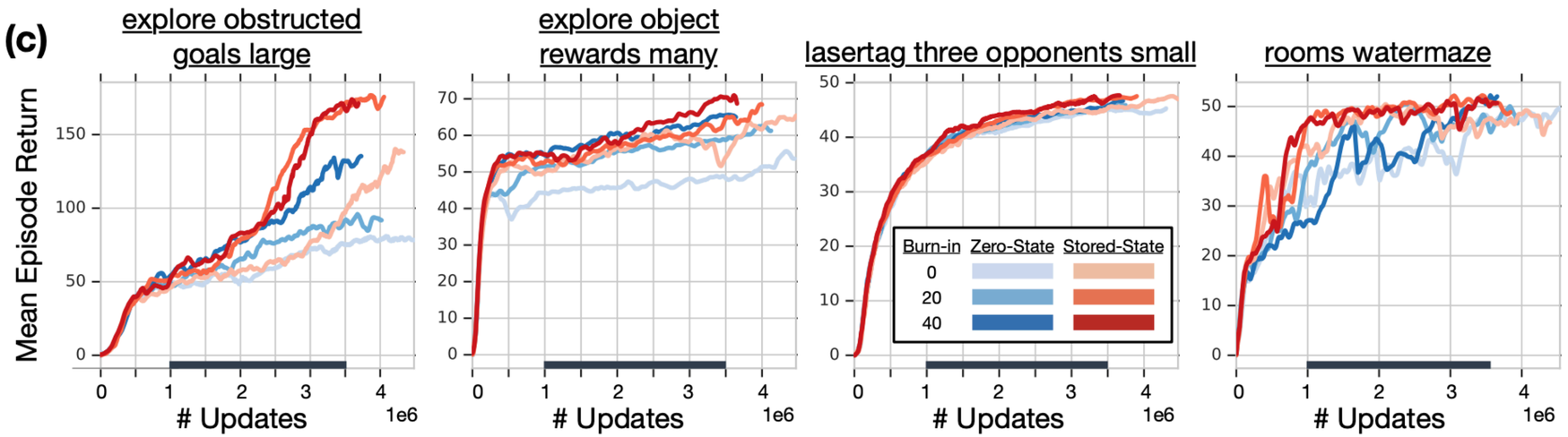
Q-value discrepancy ΔQ as a measure for recurrent state staleness.



- Zero start state results in more severe effect of state staleness
- The effect is greatly reduced for the last sequence state compared to the first ones
  - RNN has had time to recover from the atypical start state
  - But the effect of staleness is still substantially worse here for the zero state than the stored state strategy

Recurrent Experience Replay in Distributed Reinforcement Learning

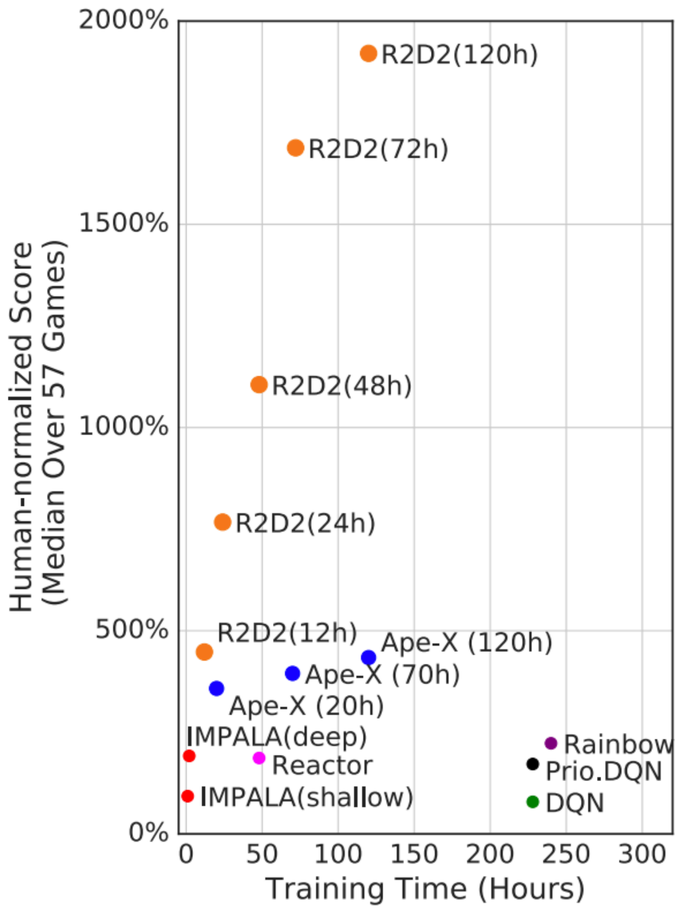
Q-value discrepancy ΔQ as a measure for recurrent state staleness.

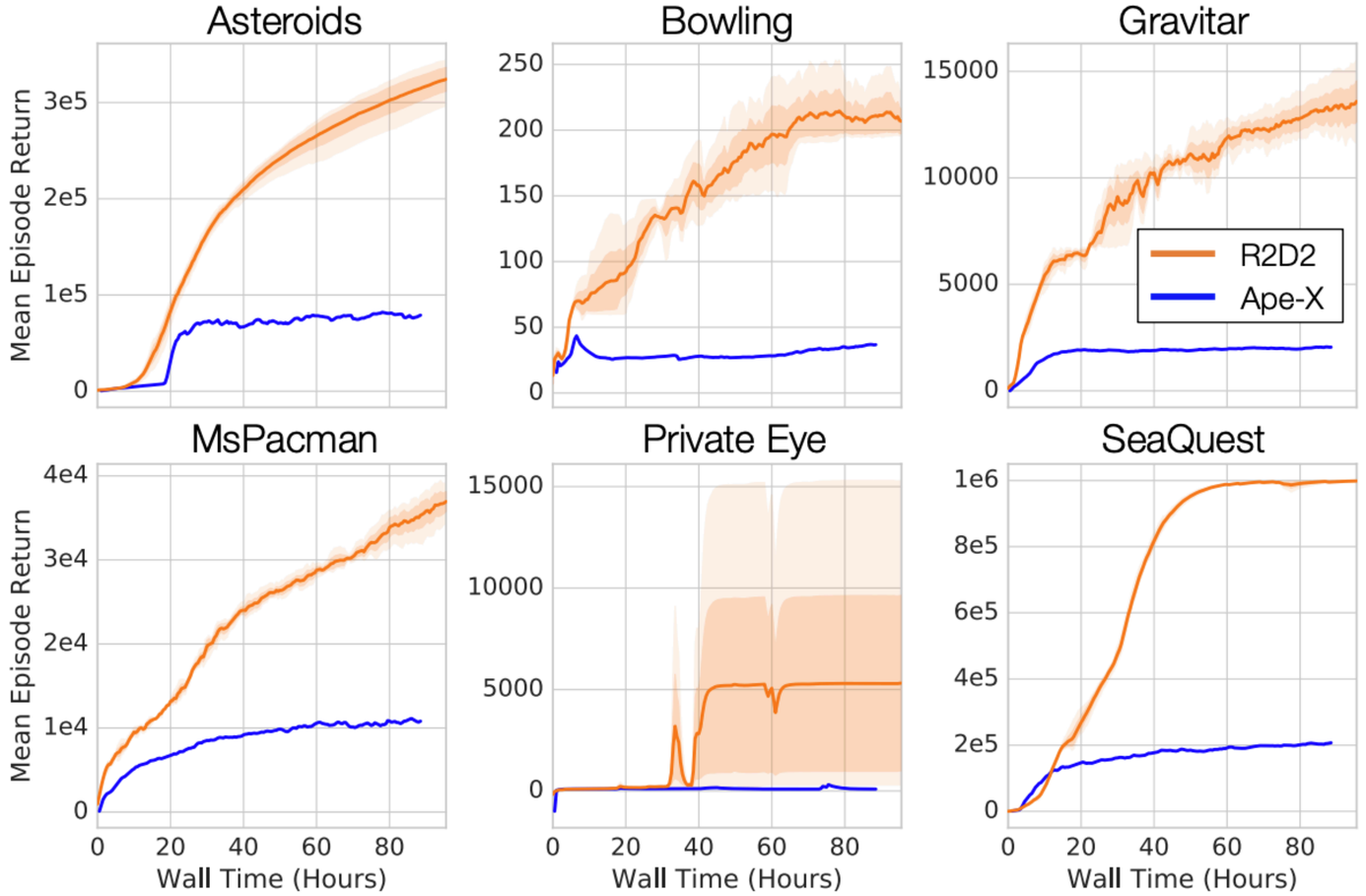


- Burn-in strategy does not show a significant effect on the Q-value discrepancy for later sequence states, however, it lead a noticeable performance improvement.
- The zero state strategy without burn-in performs worse overall
  - Zero state strategy with burn-in means it unrolls the network over a prefix of states on which the network does not receive updates.
  - Burn-in can prevent ‘destructive updates’ to the RNN parameters resulting from the highly inaccurate initial outputs s on the first few time steps after a zero state initialization.

Recurrent Experience Replay in Distributed Reinforcement Learning

Experimental Results





Outline

- Agent57 Family Overview
- DQN Series
- R2D2
- Never Give Up
- Multi-Armed Bandit
- Agent57

Never Give Up: Learning Directed Exploration Strategies

- A paper introduced by DeepMind at **ICLR 2020**.

Never Give Up: Learning Directed Exploration Strategies

Adopting Episodic Memory in the RL Method

- **Never Give Up (NGU)** was designed to augment R2D2 with another of memory: episodic memory
- Episodic memory enables **NGU** to detect when new parts of a game are encountered, so the agent can explore these newer parts of the game
- This makes the agent's behavior (**exploration**) deviate significantly from the policy the agent is trying to learn (**obtaining a high score in the game**)
    - Thus, off-policy learning again plays a critical role
- **NGU** was the first agent to obtain positive rewards, without domain knowledge, on **Pitfall**
    - No RL method had scored any points since the introduction of the Atari57 benchmark
- Unfortunately, **NGU** sacrifices performance on what have historically been the "easier" games and so, on average, underperforms relative to **R2D2**.

Never Give Up: Learning Directed Exploration Strategies

Highlights

- First reinforcement learning agent able to solve hard exploration games by **learning a range of directed exploratory policies**
- First algorithm to achieve non-zero rewards (with a mean score of 8,400) in the game of **Pitfall!** without using demonstrations or hand-crafted features
- **NGU** o jointly learns a family of policies, with various degrees of exploratory behavior
    - The learning of the exploratory policies can be thought of as a **set of auxiliary tasks** that can help build a shared architecture that continues to develop even in the absence of extrinsic rewards

Never Give Up: Learning Directed Exploration Strategies

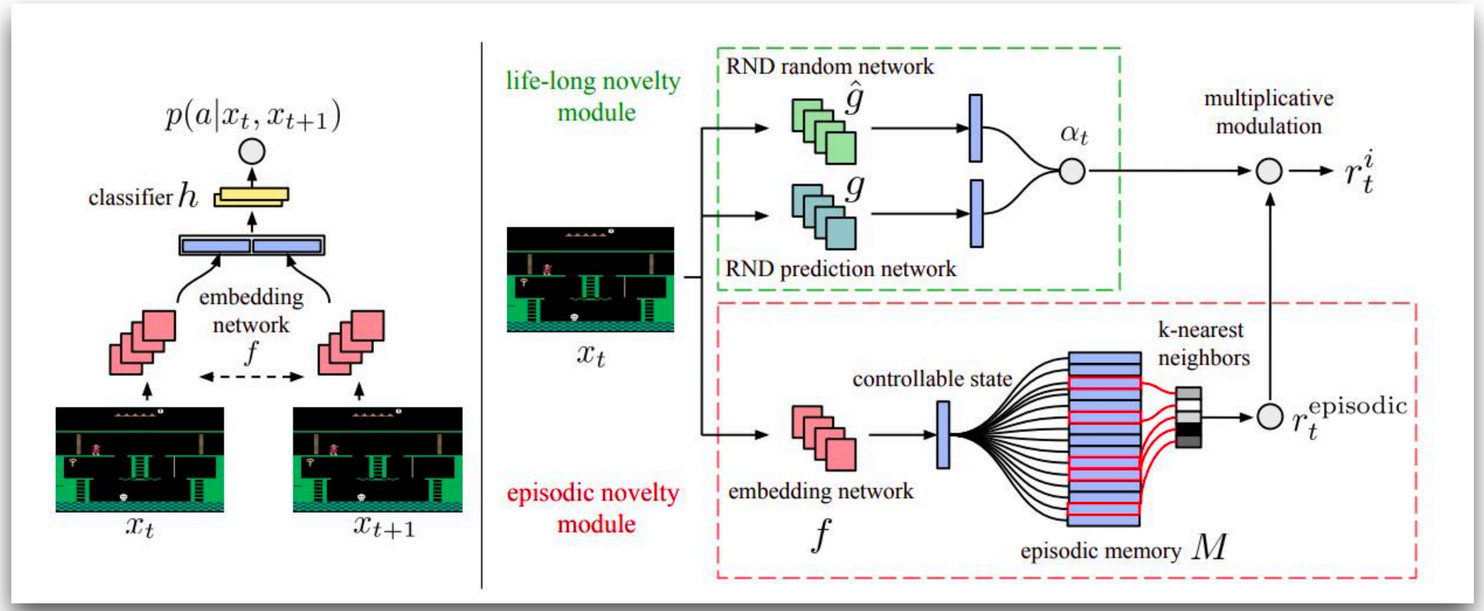
Contributions

- The main contributions of **NGU** consists of the following items:
- An exploration bonus combining ***life-long*** and ***episodic*** novelty to learn exploratory strategies that can **maintain exploration** throughout the agent's training process (to **never give up**)
    - Developing intrinsic motivation rewards that encourage an agent to explore and visit as many states as possible by providing more dense "internal" rewards for novelty-seeking behaviors
    - Long-term ***life-long*** novelty rewards encourage visiting many states throughout training, across many episodes
    - Short-term ***episodic*** novelty rewards encourage visiting many states over a short span of time (e.g., within a single episode of a game)

- To learn a family of policies that separate exploration and exploitation using a conditional architecture with shared weights

Never Give Up: Learning Directed Exploration Strategies

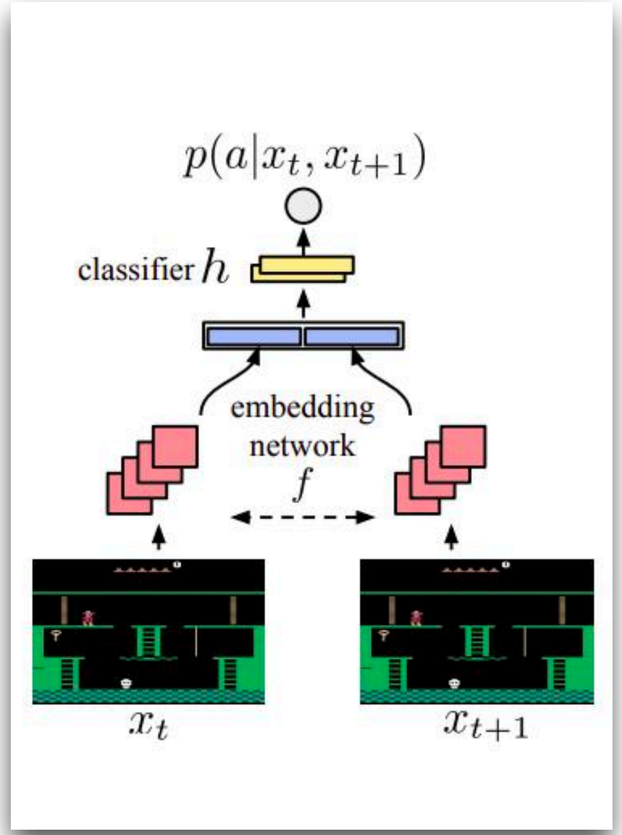
The never-give-up intrinsic reward generation architecture



- The network is trained based on the augmented reward $r_t = r_t^e + \beta r_t^i$
- The intrinsic reward r_t^i satisfies three properties:
  - It rapidly discourages revisiting the same state within the same episode
  - It slowly discourages visits to states visited many times across episodes
  - The notion of state ignores aspects of an environment that are **not influenced by an agent's actions**

Never Give Up: Learning Directed Exploration Strategies

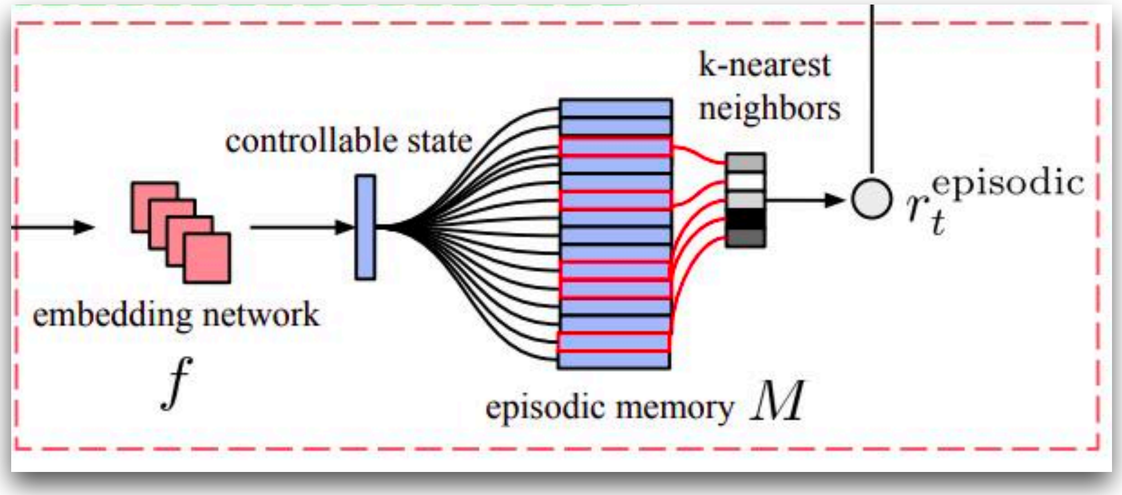
The never-give-up intrinsic reward — embedding network



- The embedding network can be thought of as a siamese network by maximizing $p(a \mid x_t, x_{t+1})$
- When used to embed states, the classifier and x_{t+1} are discarded.
- The network projects the states into a space where as much useless information as possible has been discarded.

Never Give Up: Learning Directed Exploration Strategies

The never-give-up intrinsic reward — episodic novelty module (1/3)



The embedding states are then saved in a buffer and clustered using k-NN so as to get a pseudo-count of the states.

Thus $r_t^{episodic}$ is defined by,

$$r_t^{episodic} = \frac{1}{\sqrt{n(f(x_t)}} \approx \frac{1}{\sqrt{\sum_{f_i \in N_k} K(f(x_t), f_i)} + c}$$

Never Give Up: Learning Directed Exploration Strategies

The never-give-up intrinsic reward — episodic novelty module (2/3)

$$r_t^{episodic} = \frac{1}{\sqrt{n(f(x_t)}} \approx \frac{1}{\sqrt{\sum_{f_i \in N_k} K(f(x_t), f_i) + c}}$$

- $n(f(x_t))$ is the counts for the visits to the abstract state $f(x_t)$
- Approximate the count $n(f(x_t))$ as the sum of the similarities given by a kernel function $K : \mathbb{R}^p \times \mathbb{R}^p \to \mathbb{R}$, over the content of episodic memory M

Never Give Up: Learning Directed Exploration Strategies

The never-give-up intrinsic reward — episodic novelty module (3/3)

$$r_t^{episodic} = \frac{1}{\sqrt{n(f(x_t)}} \approx \frac{1}{\sqrt{\sum_{f_i \in N_k} K(f(x_t), f_i)} + c}$$

- In practice, the pseudo-counts are computed using k-nearest neighbors of $f(x_t)$ in the memory M, denoted as $N_k = \{f_i\}_{i=1}^{k}$
- Use inverse kernel for K, where $K(x, y) = \frac{\epsilon}{\frac{d^2(x,y)}{d_m^2} + \epsilon}$, d is the Euclidean distance and d_m is the moving average of d. (d_m is used to make the kernel more robust)

Never Give Up: Learning Directed Exploration Strategies

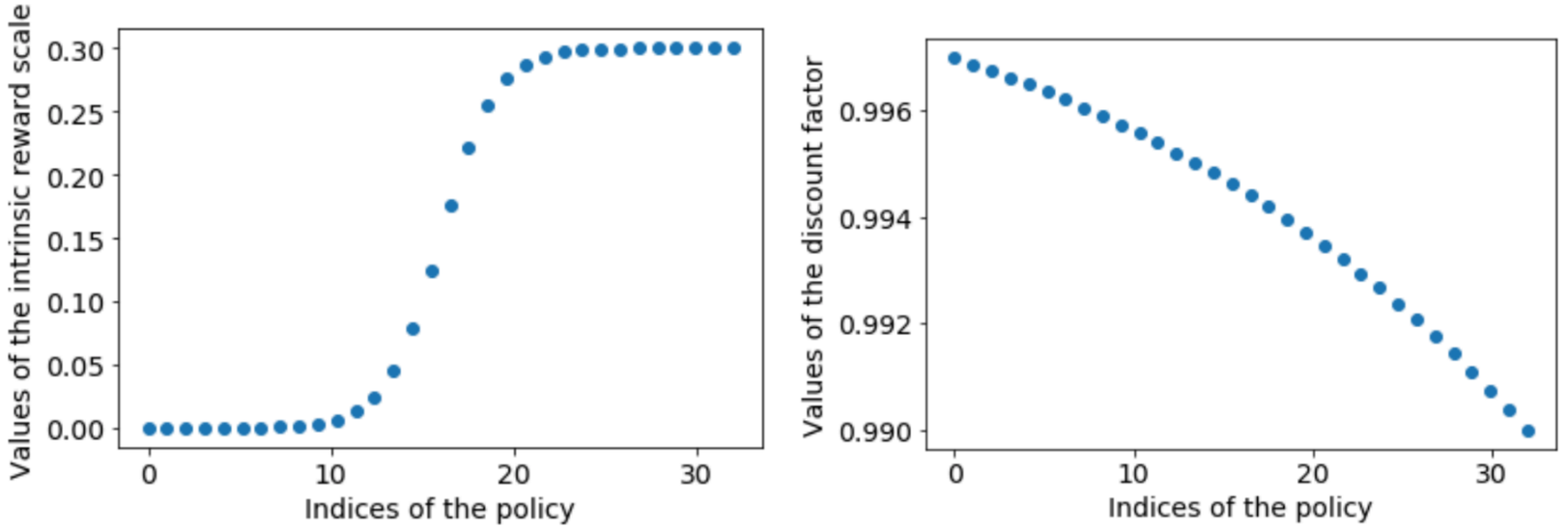
The NGU agent

- Using intrinsic rewards as a means of exploration subtly changes the underlying Markov Decision Process (MDP) being solved
  - If the augmented reward $r_t = r_t^e + \beta r_t^i$ varies in ways unpredictable from the action and states, then the decision process may no longer be a MDP, but instead be a Partially Observed MDP (POMDP)
  - Solving PODMPs can be much harder than solving MDPs
- **NGU** uses **R2D2** as the basis of the DRL agent
  - NGU adopts a family of augmented rewards of the form $r_t = r_t^e + \beta r_t^i$
  - A discrete number N of values $\{\beta_j\}_{j=0}^{N-1}$ are available for selection
  - At the beginning of each episode and in each actor, **NGU** uniformly selects a pair (β_j, γ_j)

Never Give Up: Learning Directed Exploration Strategies

Distributed Training

- **NGU** adopts different values of $\{\beta_j\}_{j=0}^{N-1}$ and $\{\gamma_j\}_{j=0}^{N-1}$, allowing the actors to behave differently (the values of each pair (β_j, γ_j) are derived by a heuristic function)



(a) Values taken by the $\{\beta_i\}_{i=0}^{N-1}$

(b) Values taken by the $\{\gamma_i\}_{i=0}^{N-1}$

Figure 7: Values taken by the $\{\beta_i\}_{i=0}^{N-1}$ and the $\{\gamma_i\}_{i=0}^{N-1}$ for $N = 32$ and $\beta = 0.3$.

Never Give Up: Learning Directed Exploration Strategies

State-action value function learning

- Given N different pairs of (β_j, γ_j), the corresponding state-action value function varies
- **NGU** aims to learn the N different associated optimal state-action value functions $Q^*_{r_j}$ associated with each reward function $r_{j,t} = r^e_t + \beta_j r^i_t$
  - The exploration rate β_j controls the degree of exploration. Higher values will encourage exploratory policies and smaller values will encourage exploitative policies
  - for purposes of learning long-term credit assignment, each $Q^*_{r_j}$ has its own associated discount factor γ_j
- To learn the state-action value function $Q^*_{r_j}$, **NGU** trains a recurrent neural network $Q(x, a, j; \theta)$
  - j is a one-hot vector indexing one of N pairs of (β_j, γ_j), x is the current observation, a is the action, and θ are the parameters for the network

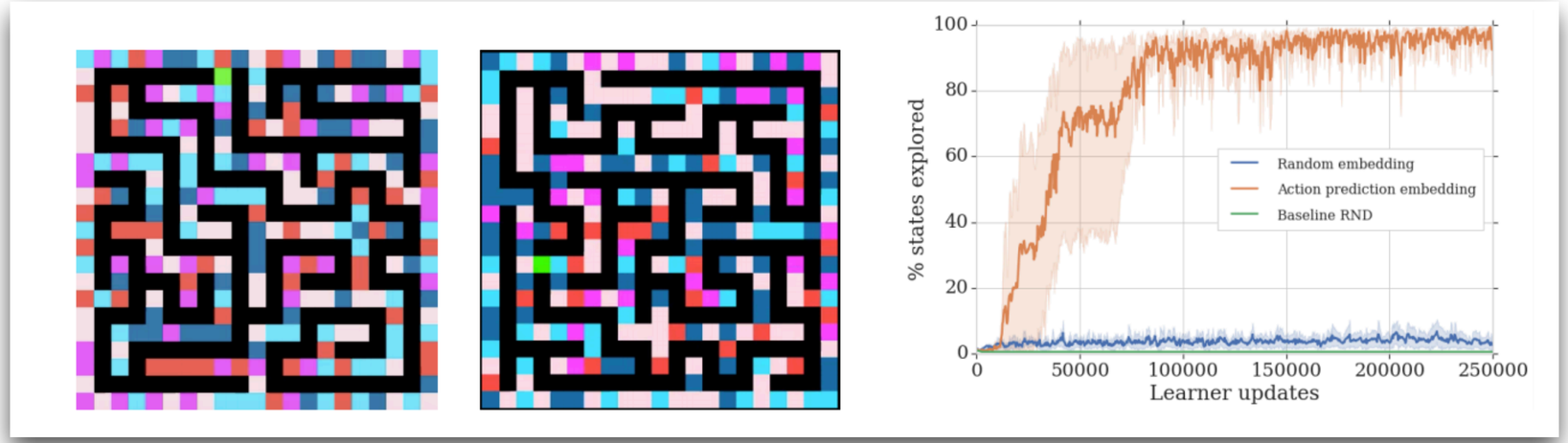
Never Give Up: Learning Directed Exploration Strategies

Training procedure

- **NGU** decouples the data collection and the learning processes by having many actors feed data to a central prioritized replay buffer
- **A learner** can sample training data from the replay buffer that contains sequences of transitions
    - These sequences come from actor processes that interact with independent copies of the environment, and they are prioritized based on TD errors
    - The priorities are initialized by the actors and updated by the learner with the updated state-action value function $Q(x, a, j; \theta)$
    - The learner samples transitions based on the priorities, and updates the parameters by minimizing the estimation loss
- Each **actor** shares the same network architecture as the learner but with different weights
    - The learner weights θ are sent to the actor frequently
    - Each actor uses different values of ϵ
    - At the beginning of each episode and in each actor, **NGU** uniformly selects a pair (β_j, γ_j)

Never Give Up: Learning Directed Exploration Strategies

Experimental results



- Design an environment called ***Random Disco Maze***
  - The agent in green; the pathway in black. The colors of the wall change at every time step
  - There is no extrinsic reward
- The results show the importance of estimating the exploration bonus using a controllable state representation

Never Give Up: Learning Directed Exploration Strategies

Experimental results on hard exploration games

Algorithm	Gravitar	MR	Pitfall!	PrivateEye	Solaris	Venture
Human	3.4k	4.8k	6.5k	69.6k	**12.3k**	1.2k
Best baseline	**15.7k**	**11.6k**	0.0	11k	5.5k	2.0k
RND	3.9k	10.1k	-3	8.7k	3.3k	1.9k
R2D2+RND	15.6k±0.6k	10.4k±1.2k	-0.5±0.3	19.5k±3.5k	4.3k±0.6k	**2.7k±0.0k**
R2D2(Retrace)	13.3k±0.6k	2.3k±0.4k	-3.5±1.2	32.5k±4.7k	6.0k±1.1k	2.0k±0.0k
NGU(N=1)-RND	12.4k±0.8k	3.0k±0.0k	**15.2k±9.4k**	40.6k±0.0k	5.7k±1.8k	46.4±37.9
NGU(N=1)	11.0k±0.7k	8.7k±1.2k	9.4k±2.2k	60.6k±16.3k	5.9k±1.6k	876.3±114.5
NGU(N=32)	14.1k±0.5k	10.4k±1.6k	8.4k±4.5k	**100.0k±0.4k**	4.9k±0.3k	1.7k±0.1k

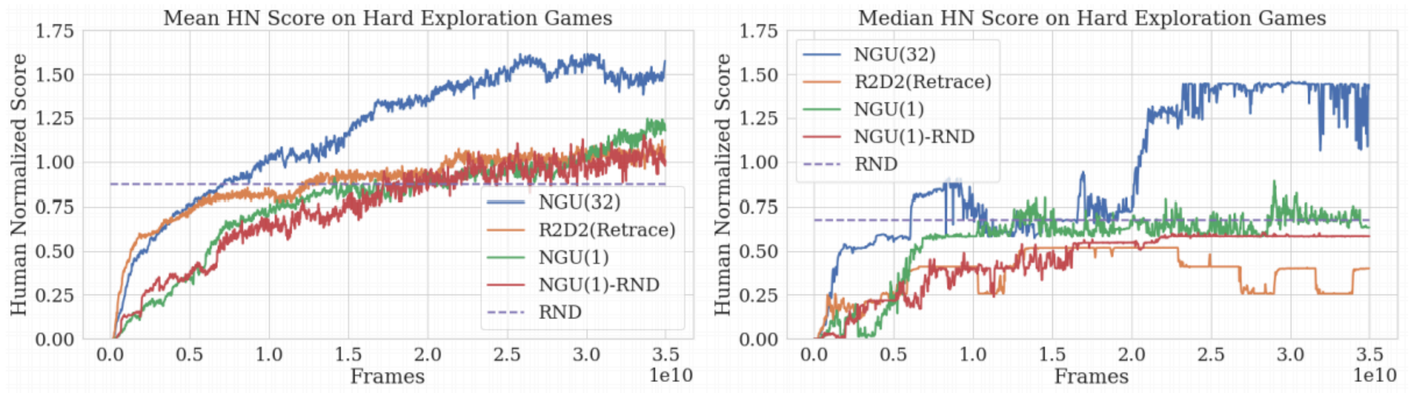
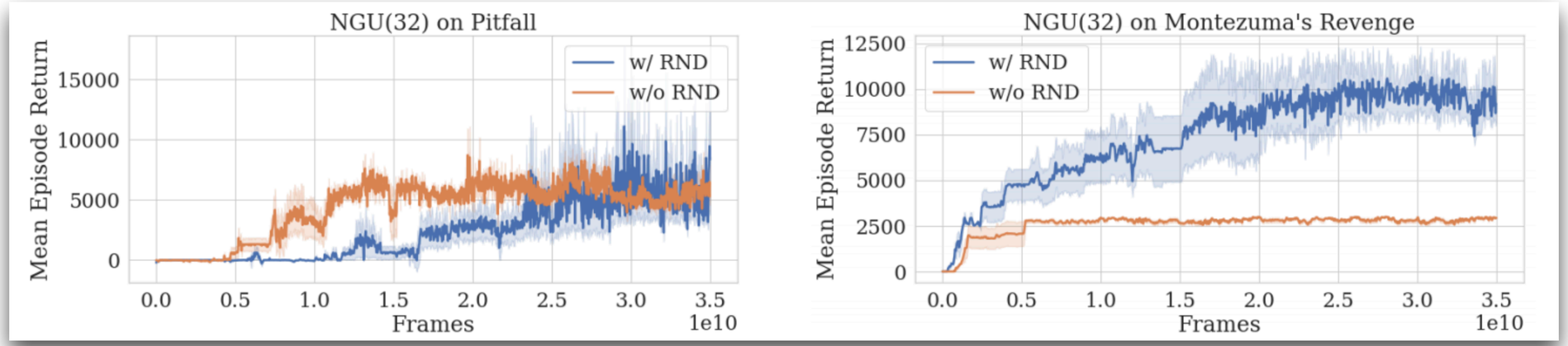


Figure 4: Human Normalized Scores on the 6 hard exploration games.

Never Give Up: Learning Directed Exploration Strategies

Experimental results on hard exploration games



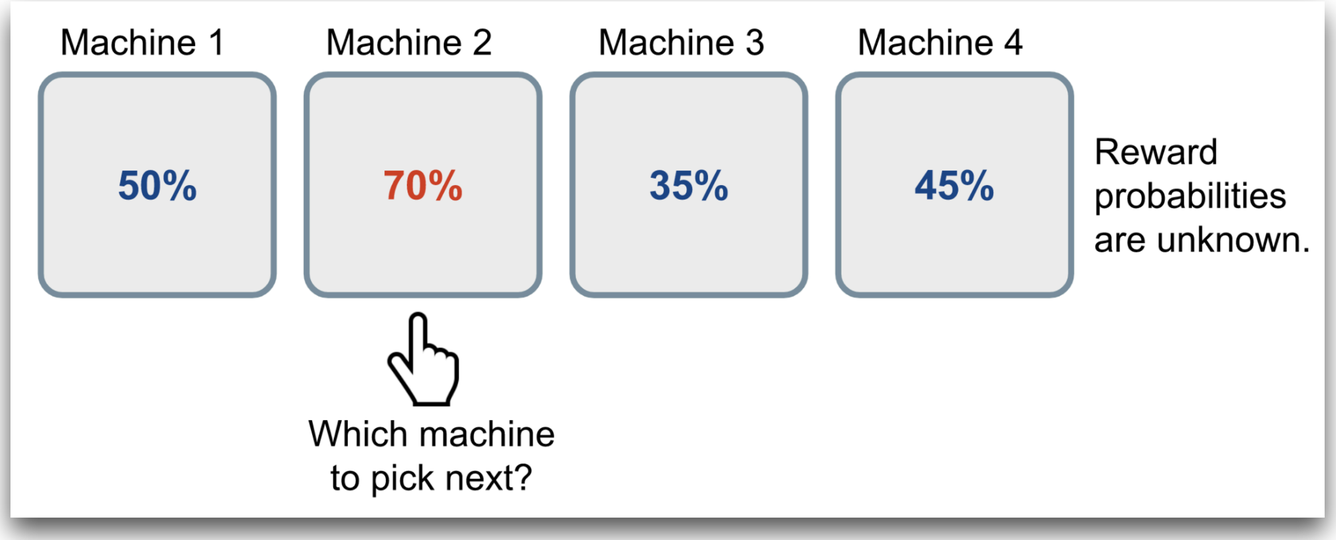
Algorithm	Pong	QBert	Breakout	Space Invaders	Beam Rider
Human	14.6	13.4k	30.5	1.6k	16.9k
R2D2	**21.0**	408.8k	837.7	43.2k	**188.2k**
R2D2+RND	20.7±0.0	353.5k±41.0k	815.8±5.3	**54.5k±2.8k**	85.7k±9.0k
R2D2(Retrace)	20.9±0.0	415.6k±55.8k	838.3±7.0	35.0k±13.0k	111.1k±5.0k
NGU(N=1)-RND	-8.1±1.7	647.1k±50.5k	**864.0±0.0**	45.3k±4.9k	166.5k±8.6k
NGU(N=1)	-9.4±2.6	**684.7k±8.8k**	**864.0±0.0**	43.0k±3.9k	114.6k±2.3k
NGU(N=32)	19.6±0.1	465.8k±84.9k	532.8±16.5	44.6k±1.2k	68.7k±11.1k

Outline

- Agent57 Family Overview
- DQN Series
- R2D2
- Never Give Up
- **Multi-Armed Bandit**
- Agent57

Multi-Armed Bandit

What is multi-armed bandit?



- An illustration of how a *Bernoulli* multi-armed bandit works. The reward probabilities are **unknown** to the player

- The multi-armed bandit problem is a classic problem that well demonstrates the exploration vs exploitation dilemma
- Imagine you are in a casino facing multiple slot machines and each is configured with an unknown probability of how likely you can get a reward at one play
- The question is: **What is the best strategy to achieve highest long-term rewards?**

Multi-Armed Bandit

The definition of a Bernoulli multi-armed bandit

- A Bernoulli multi-armed bandit can be described as a tuple of $\langle A, R \rangle$, where:
  - We have K machines with reward probabilities, $\{\theta_1, \ldots, \theta_K\}$
  - At each time step t, we take an action a on one slot machine and receive a reward r.
  - A is a set of actions, each referring to the interaction with one slot machine.
    - The value of action a is the expected reward $Q(a) = \mathbb{E}[r \mid a] = \theta$
    - If action a at the time step t is on the I-th machine, then $Q(a_t) = \theta_i$
  - R is the reward function. In the case of Bernoulli bandit, we observe a reward r in a stochastic fashion
    - At the time step t, $r_t = R(a_t)$ may return reward 1 with probability $Q(a_t)$ or 0 otherwise

Multi-Armed Bandit

The Optimization of a Bernoulli multi-armed bandit

- The goal is to maximize the cumulative reward $\sum_{t=1}^{T} r_t$
- The optimal reward probability θ^* of the optimal action a^* is represented as follows:

$$\theta^* = Q(a^*) = \max_{a \in A} Q(a) = \max_{1 \leq i \leq K} \theta_i$$

- Our loss function is the total regret we might have by not selecting the optimal action up to the time step T:

$$L_T = \mathbb{E}[\sum_{t=1}^{T} (\theta^* - Q(a_t))]$$

Multi-Armed Bandit

The ϵ-Greedy Algorithm for dealing with multi-armed bandit

- The ϵ-greedy algorithm takes the best action most of the time, but does random exploration occasionally
- The action value is estimated according to the past experience by averaging the rewards associated with the target action a that we have observed so far (up to the current time step t):

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{\tau=1}^{t} r_\tau \mathbf{1}[\mathbf{a}_\tau = \mathbf{a}]$$

  where $\mathbf{1}$ is a binary indicator function and $N_t(a) = \sum_{\tau=1}^{t} \mathbf{1}[\mathbf{a}_\tau = \mathbf{a}]$ is how many times the action a has been selected so far
- According to the ϵ-greedy algorithm, with a small probability ϵ we take a random action, but otherwise (which should be the most of the time, probability 1-ϵ) we pick the best action that we have learnt so far

$$\hat{a}_t^* = \underset{a \in A}{\text{argmax}} \hat{A}_t(a)$$

Multi-Armed Bandit

Optimistic exploration

- **Assumption**
  - New states = Good states
- **Methodology**
  - Add bonus to rewards
  - Requires estimating state visitation frequencies or novelty
  - Typically realized by means of exploration bonuses

Multi-Armed Bandit

The upper confidence bound (UCB) exploration

- UCB :

$$A_t = \arg\max_a [Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}}]$$

- c : A confidence value that controls the level of exploration
- $N(a)$: the number of times that action '***a***' has been selected, prior to time '***t***'
- Usually used in Monte Carlo Tree Search (MCTS)
- However, the UCB algorithm focuses on a current state
- It is also limited to simpler, tabular based tasks

Multi-Armed Bandit

The upper confidence bound (UCB) exploration

- UCB :

$$A_t = \arg\max_a [Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}}]$$

- Exploitation part: $Q_t(a)$
  - **Optimism in the fact of uncertainty**
  - If you don’t know which action is best then choose the one that currently looks to be the best
  - the action that currently has the highest estimated reward will be the chosen action

Multi-Armed Bandit

The upper confidence bound (UCB) exploration

- Expand the idea : add a state bonus into rewards

$$r_{new} = r + B(N(s))$$

- $B(N(s))$ defined in different types
  - UCB $\sqrt{\frac{\log n}{N(s)}}$
  - MBIE-EB (Strehl & Littman, 2008) $\sqrt{\frac{1}{N(s)}}$
  - BEB (Kolter & Ng, 2009) $\frac{1}{N(s)}$

Multi-Armed Bandit

The upper confidence bound (UCB) exploration

- UCB :

$$A_t = \arg\max_a [Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}}]$$

- Exploration part: $c\sqrt{\frac{\log t}{N_t(a)}}$
  - The degree of exploration is controlled by the hyper-parameter ‘c’
  - $N_t(a)$ is small if an action hasn’t been tried very often

Multi-Armed Bandit

Posterior sampling

- Thompson sampling
  - Sample a distribution
  - Take this distribution to select action
  - Update the distribution

Algorithm 4.2

Thompson$(\mathcal{X}, p, q, r)$

for $t = 1, 2, \ldots$ **do**
  #sample model:
  Sample $\hat{\theta} \sim p$

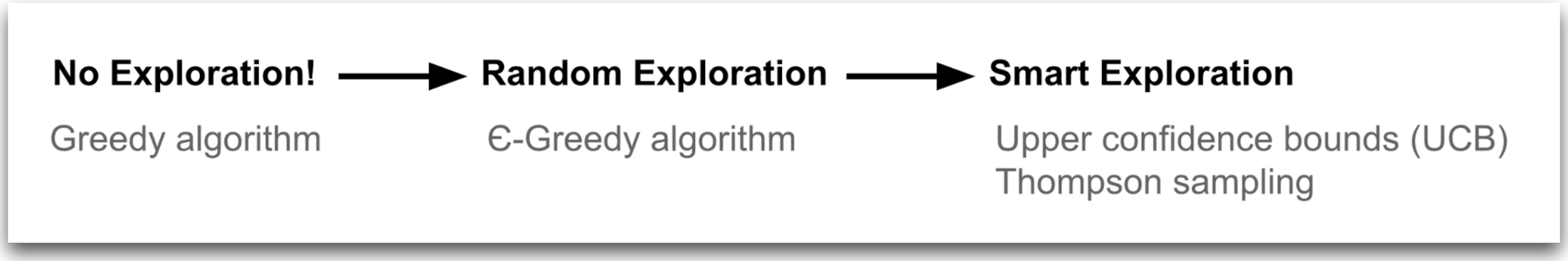
  #select and apply action:
  $x_t \leftarrow \text{argmax}_{x \in \mathcal{X}} \mathbb{E}_{q_{\hat{\theta}}}[r(y_t)|x_t = x]$
  Apply x_t and observe y_t

  #update distribution:
  $p \leftarrow \mathbb{P}_{p,q}(\theta \in \cdot | x_t, y_t)$
end for

Multi-Armed Bandit

Summary

- Exploration is necessary for multi-armed bandit problems
  - In terms of the exploration strategies, we can do no exploration at all, focusing on the short-term returns
  - We can also occasionally explore at random
  - Or even further, we explore and we are picky about which options to explore — actions with higher uncertainty are favored because they can provide higher information gain



Outline

- Agent57 Family Overview
- DQN Series
- R2D2
- Never Give Up
- Multi-Armed Bandit
- Agent57

Agent57

Learning to Balance Exploration with Exploitation

- **Agent57** is built on the following observation
  - What if an agent can learn when it's better to exploit?
  - When it's better to explore?

- **Agent57** introduced a meta-controller for two purposes
  - Adapting the exploration-exploitation trade-off
  - Adjusting the time horizon for games requiring longer temporal credit assignment

- **Agent57** is therefore able to get the best of both worlds: **above human-level performance** on both **easy games** and **hard games**

Agent57

Online Adaptation Mechanism

- **Two shortcomings** of intrinsic motivation methods:
  - **Exploration**: Much of the experience produced by exploratory policies in **NGU** will eventually become wasteful after the agent explores all relevant states
  - **Time horizon**:
    - Some tasks will require long time horizons (e.g. **Skiing**, **Solaris**), where **valuing rewards that will be earned in the far future might be important** for eventually learning a good exploitative policy, or even to learn a good policy at all
    - Other tasks may be slow and unstable to learn **if future rewards are overly weighted**
    - This trade-off is commonly controlled by **the discount factor** in reinforcement learning, where a higher discount factor enables learning from longer time horizons

Agent57

Contributions

- A new parameterization of the state-action value function that decomposes the contributions of the intrinsic and extrinsic rewards
  - This significantly increases the **training stability** over a large range of intrinsic reward scales
- A **meta-controller**: an adaptive mechanism to select which of the policies (parameterized by **exploration rate** and **discount factors**) to prioritize throughout the training process
  - This allows the agent to control the **exploration/exploitation trade-off** by dedicating more resources to one or the other

Agent57

The issues in NGU

- **NGU** can be unstable and fail to learn an appropriate approximation of $Q^*_{r_j}$ for all the state-action value functions in the family, even in simple environments
- This is especially the case when the scale and sparseness of r^e_t and r^i_t are both different
    - Or when one reward is more noisy than the other
- This may due to the fact that learning a common state-action value function for a mix of rewards is difficult when the rewards are very different in nature

Agent57

Improvement 1: splitting the state-action value function

- **Agent57** splits the state-action value function as follows
  - $Q(x, a, j; \theta) = Q(x, a, j; \theta^e) + Q(x, a, j; \theta^i)$

    where $Q(x, a, j; \theta^e)$ and $Q(x, a, j; \theta^i)$ are the extrinsic and intrinsic components, respectively
- The sets of weights θ^e and θ^i separately parameterize two neural networks with identical architecture
  - $Q(x, a, j; \theta^e)$ and $Q(x, a, j; \theta^i)$ are optimized separately in the learner with rewards r^e and r^i, respectively
- By doing this, **Agent57** allows each network to adapt to the scale and variance associated with their corresponding reward.
  - It also allows for the associated optimizer state to be separated for intrinsic and extrinsic state-action value functions

Agent57

Improvement 2: Adaptive exploration over a family of policies

- The core idea of **NGU** is to jointly train a family of policies with different degrees of exploratory behavior using a single network architecture
  - In this way, training these exploratory policies plays the role of a set of **auxiliary tasks** that can help train the shared architecture even in the absence of extrinsic reward
  - A major limitation of this approach is that **all policies are trained equally**, **regardless of their contribution to the learning progress**
- As a result, **Agent57** proposes to incorporate **a meta-controller** that can **adaptively select which policies to use** both at training and evaluation time

Agent57

The motivation of introducing meta-controller

- The meta-controller carries two important consequence
  - By selecting which policies to prioritize during training, **Agent57** can allocate more of the **capacity** of the network to better represent the state-action value function of the policies that are **most relevant for the task at hand**
  - This is likely to change throughout the training process, naturally building a **curriculum** to facilitate training
    - It is expected that policies with higher β_j and lower γ_j in the early training phase
    - As training progresses, it becomes the opposite
  - The mechanism also provides a way of choosing the best policy in the family to use at evaluation time

Agent57

The meta-controller (1/2)

- Agent57 proposes to implement the meta-controller using a non-stationary (i.e., **the actors are changing over time**) multi-arm bandit algorithm running independently on each actor
  - Each arm j from the N-arm bandit is linked to a policy in the family and corresponds to a pair (β_j, γ_j)
  - At the beginning of each episode, the meta-controller chooses an arm J_k (which is a random variable) setting which policy will be executed
  - Then the l-th actor acts ϵ_l-greedily with respect to the corresponding state-action value function, $Q(x, a, J_k; \theta_l)$, for the whole episode
  - The un-discounted extrinsic episode returns $R_k^e(J_k)$ are used as a reward signal to train the multi-arm bandit algorithm of the meta-controller

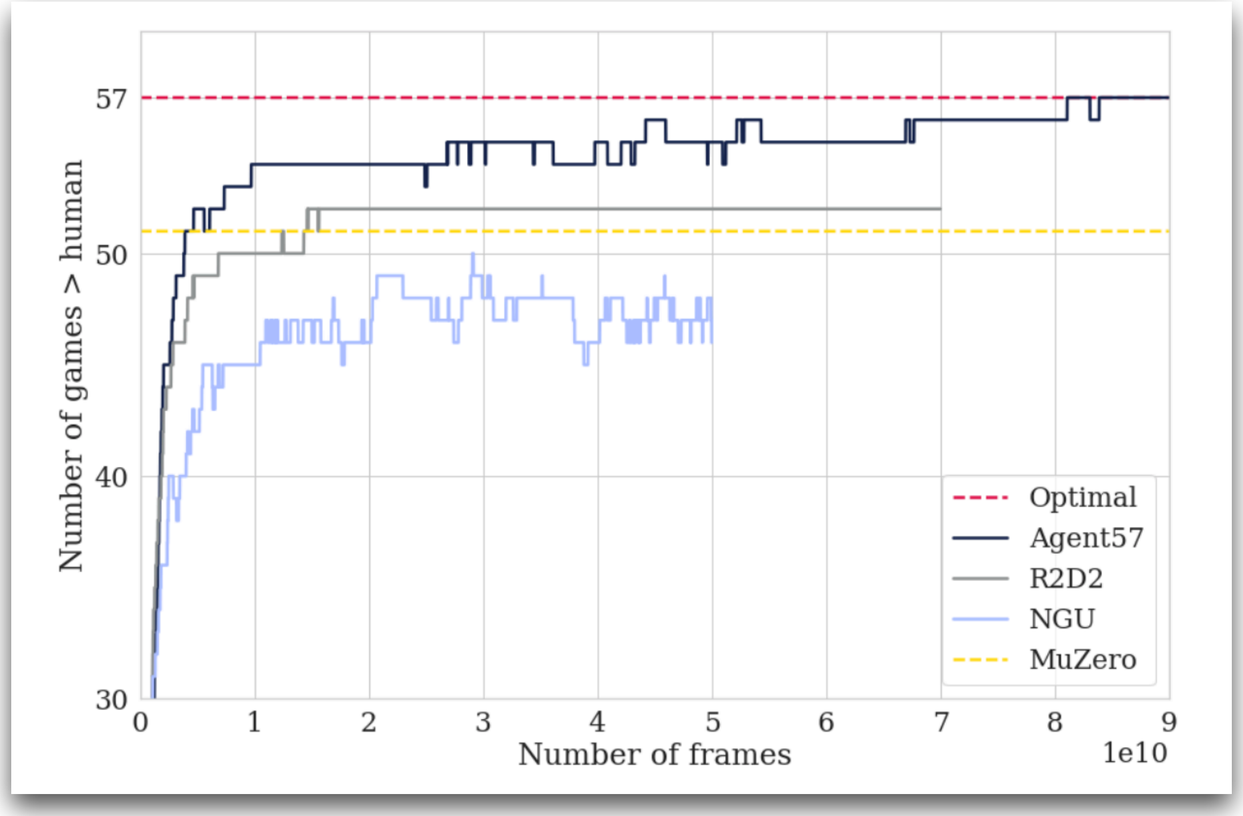
Agent57

The meta-controller (2/2)

- The reward signal $R_k^e(J_k)$ is non-stationary, as the agent changes throughout training
  - A classical bandit algorithm such as Upper Confidence Bound (UCB) is thus not able to adapt to the changes of the reward through time
  - Therefore, **Agent57** employees a simplified sliding-window UCB with UCB $\epsilon_{UCB}-$ greedy exploration
  - With probability $1 - \epsilon_{UCB}$, this algorithm runs a slight modification of classic UCB on a sliding window of size $\tau = 160 \sim 3600$ and selects a random arm with probability ϵ_{UCB}

Agent57

Performance comparison with the other baselines



- Published at **ICML 2020**
- **Agent57** was able to scale with increasing amounts of computation: the longer it trained, the higher its score got

Statisitics	Agent57	NGU	R2D2	MuZero
Number of games > human	57	51	52	51
Mean HNS	4766.25%	3421.80%	4622.09%	**5661.84%**
Median HNS	1933.49%	1359.78%	1935.86%	**2381.51%**
5th percentile of HNS	**116.67%**	64.10%	50.27%	0.03%

- Number of games where algorithms are better than the human benchmark throughout training for Agent57 and state-ofthe-art baselines on the 57 Atari games.

ありがとうございます

ありがとうございます