Asynchronous Methods for Deep Reinforcement Learning

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, Koray Kavukcuoglu

Xiangyu Zhao

xz398@cam.ac.uk

11 February 2022

Background

Previous belief: deep RL is fundamentally unstable, because the sequence of observed data encountered by an online RL agent is non-stationary, and online RL updates are strongly correlated

Major deep RL solution prior to this paper: store the agent's data in an *experience replay* memory, and train deep models via mini-batches or random sampling

Problems:

- Requires very heavy computational and memory resources
- Only works for off-policy methods

Previous work on asynchronous RL training (General Reinforcement Learning Architecture, Gorila) has shown a promising result [Nair et al., 2015]

This paper

- Proposes an asynchronous RL framework
  - Conceptually simple, lightweight
  - Enables deep RL for on-policy methods
- Presents asynchronous variants of 4 standard RL algorithms:
  - One-step Q-Learning
  - One-step Sarsa
  - n-step Q-Learning
  - Advantage actor-critic
- Best performing method: asynchronous advantage actor-critic (A3C)
- A3C succeeds in a wide variety of continuous motor control problems, in addition to Atari games

Asynchronous RL framework

- Asynchronously execute multiple agents in parallel, on multiple instances of the environment
  - Each agent can run different exploration policy to maximise diversity
- Asynchronously update the parameters of the global NN using the gradients computed by the agents
- Synchronise global and agents' parameters at fixed intervals
- Benefits:
  - Decorrelates the agents' data, and stabilise learning
  - Roughly linear reduction in training time
- Comparison:
  - Gorila: uses separate machines for the agents, and maintains a parameter server
  - This paper: all agents are executed on a single machine, using multiple CPU threads – removes communication costs of sending gradients and parameters

Asynchronous RL framework

Global Network

Policy $\pi(s)$ $V(s)$

Network

Input (s)

Worker 1 Worker 2 Worker 3 ... Worker n

Environment 1 Environment 2 Environment 3 ... Environment n

https://medium.com/sciforce/reinforcement-learning-and-asynchronous-actor-critic-agent-a3c-algorithm-explained-f0f3146a14ab

Recall: Q-learning and Sarsa

- Q-learning: value-based, off-policy TD control

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\Big)$$

- Sarsa: value-based, on-policy TD control

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\Big)$$

Recall: Q-learning and Sarsa

- Q-learning: value-based, off-policy TD control

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \Big)$$

- Sarsa: value-based, on-policy TD control

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \Big)$$

Recall: Q-learning and Sarsa

- Q-learning: value-based, off-policy TD control

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\Big)$$

- Sarsa: value-based, on-policy TD control

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\Big)$$

Asynchronous one-step Q-learning

Algorithm Asynchronous one-step Q-learning: individual actor-learner thread

Require: global shared NN weights θ and target NN weights θ^*
Require: global shared counter $T = 0$
Initialise thread step counter $t \leftarrow 0$
Initialise target NN weights $\theta^* \leftarrow \theta$
Initialise NN gradients $d\theta \leftarrow 0$
Get initial state s_0
repeat
Take action a_t with ϵ-greedy policy based on $Q_\theta(s_t, a)$
Receive reward r_t and new state s_{t+1}

$$R \leftarrow \begin{cases} r_t & \text{for terminal } s_{t+1} \\ r_t + \gamma \max_a Q_{\theta^*}(s_{t+1}, a) & \text{for non-terminal } s_{t+1} \end{cases}$$

Accumulate gradients w.r.t. θ: $d\theta \leftarrow d\theta + \nabla_\theta (R - Q_\theta(s_t, a_t))^2$
$T \leftarrow T + 1$, $t \leftarrow t + 1$
if T mod $I_{\text{target}} = 0$ **then**
Update target NN weights $\theta^* \leftarrow \theta$
end if
if t mod $I_{\text{AsyncUpdate}} = 0$ **or** s_t is terminal **then**
Perform asynchronous update of θ using $d\theta$
Clear gradients $d\theta \leftarrow 0$
end if
until $T > T_{\max}$

One-step vs. n-step Q-learning

- Recall Monte Carlo methods:

$$V(S_t) \leftarrow V(S_t) + \alpha\big(G_t - V(S_t)\big)$$

- One-step Q-learning: bootstrapping using one-step return

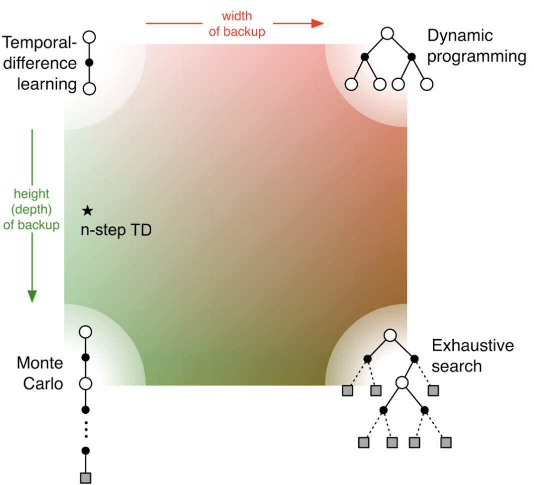
$$G_t = r_t + \gamma \max_a Q(s_{t+1}, a)$$

- n-step Q-learning: bootstrapping using n-step return

$$G_t = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_a Q(s_{t+n}, a)$$

- In practice:
  - Take actions for up to $t_{\max}$ steps or a terminal state
  - Compute gradients of the n-step updates
  - Apply accumulated updates in a single gradient step
- This makes learning more efficient...
- ...but becomes on-policy!

One-step vs. *n*-step Q-learning



Adapted from [Sutton & Barto, 2018]

Asynchronous n-step Q-learning

Algorithm Asynchronous n-step Q-learning: individual actor-learner thread

Require: global shared NN weights θ and target NN weights θ^*
Require: global shared counter $T = 0$
Initialise thread step counter $t \leftarrow 0$
Initialise target NN weights $\theta^* \leftarrow \theta$
Initialise thread-specific NN weights $\theta' \leftarrow \theta$
Initialise NN gradients $d\theta \leftarrow 0$
repeat
Clear gradients $d\theta \leftarrow 0$
Synchronise thread-specific NN weights $\theta' \leftarrow \theta$
Get state s_t
repeat
Take action a_t with ϵ-greedy policy based on $Q_{\theta'}(s_t, a)$
Receive reward r_t and new state s_{t+1}
$T \leftarrow T + 1$, $t \leftarrow t + 1$
until s_t is terminal **or** $t = t_{\max}$

$$R \leftarrow \begin{cases} 0 & \text{for terminal } s_t \\ \max_a Q_{\theta^*}(s_t, a) & \text{for non-terminal } s_t \end{cases} \quad \text{// Bootstrap from last state}$$

for $i = t - 1$ **downto** 0 **do**
$R \leftarrow r_i + \gamma R$
Accumulate gradients w.r.t. θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} (R - Q_{\theta'}(s_i, a_i))^2$
end for
Perform asynchronous update of θ using $d\theta$
if $T \bmod I_{\text{target}} = 0$ **then**
Update target NN weights $\theta^* \leftarrow \theta$
end if
until $T > T_{\max}$

Primer on actor-critic: policy gradient methods

- Policy gradient methods: learn a parametrised policy $\pi_\theta(a_t|s_t)$ through stochastic gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)}$$

  where $J(\theta_t)$ is some scalar performance measure w.r.t. the policy parameters θ_t.

- The policy gradient theorem:

$$J(\theta) =_{\text{def}} v_{\pi_\theta}(s_0)$$

$$\begin{aligned}
\nabla J(\theta) &= \nabla v_{\pi_\theta}(s_0) \\
&\propto \sum_s \mu(s) \sum_a q_{\pi_\theta}(s, a) \nabla \pi_\theta(a|s) \\
&= \mathbb{E}_{\pi_\theta}\left[\sum_a q_{\pi_\theta}(s_t, a) \nabla \pi_\theta(a|s_t)\right]
\end{aligned}$$

REINFORCE: Monte Carlo policy gradient

Standard REINFORCE:

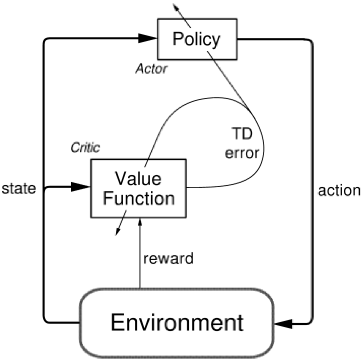
$$\begin{aligned}
\nabla J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\sum_a q_{\pi_\theta}(s_t, a)\nabla \pi_\theta(a|s_t)\right] \\
&= \mathbb{E}_{\pi_\theta}\left[\sum_a \pi_\theta(a|s_t) q_{\pi_\theta}(s_t, a)\frac{\nabla \pi_\theta(a|s_t)}{\pi_\theta(a|s_t)}\right] \\
&= \mathbb{E}_{\pi_\theta}\left[q_{\pi_\theta}(s_t, a_t)\frac{\nabla \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right] \\
&= \mathbb{E}_{\pi_\theta}\left[G_t\frac{\nabla \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right] \\
&= \mathbb{E}_{\pi_\theta}\left[G_t \nabla \ln \pi_\theta(a_t|s_t)\right]
\end{aligned}$$

REINFORCE with baseline:

$$\nabla J(\theta) =_{\text{def}} \mathbb{E}_{\pi_\theta}\left[\left(G_t - b(s_t)\right)\nabla \ln \pi_\theta(a_t|s_t)\right]$$

Actor-critic methods

- Learn approximations to both policy and value functions
- Actor: the learned policy, decides which action to take
- Critic: the learned value function, tells the actor how good its action was, and how it should adjust



[Sutton & Barto, 1998]

Advantage actor-critic (A2C)

Parametrise policy $\pi(a_t|s_t)$ by θ and value function $V(s_t)$ by θ_v

Define *advantage* of action a_t in state s_t:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$$

therefore

$$\begin{aligned}
\nabla J(\theta) &=_{\text{def}} \mathbb{E}_{\pi_\theta}\left[\left(G_t - V_{\theta_v}(s_t)\right)\nabla \ln \pi_\theta(a_t|s_t)\right] \\
&= \mathbb{E}_{\pi_\theta}\left[\left(\underline{r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1}} + \underline{\gamma^n V_{\theta_v}(s_{t+n})} - \underline{V_{\theta_v}(s_t)}\right)\nabla \ln \pi_\theta(a_t|s_t)\right] \\
&= \mathbb{E}_{\pi_\theta}\left[\left(\sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n V_{\theta_v}(s_{t+n}) - V_{\theta_v}(s_t)\right) \nabla \ln \pi_\theta(a_t|s_t)\right] \\
&= \mathbb{E}_{\pi_\theta}\left[A_{\theta,\theta_v}(s_t, a_t)\nabla \ln \pi_\theta(a_t|s_t)\right]
\end{aligned}$$

Asynchronous advantage actor-critic (A3C)

Algorithm Asynchronous advantage actor-critic (A3C): individual actor-learner thread

Require: global shared NN weights θ and θ_v
Require: global shared counter $T = 0$
  Initialise thread step counter $t \leftarrow 0$
  Initialise thread-specific NN weights $\theta' \leftarrow \theta$ and $\theta'_v \leftarrow \theta_v$
  Initialise NN gradients $d\theta \leftarrow 0$
  repeat
    Clear gradients $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    Synchronise thread-specific weights $\theta' \leftarrow \theta$ and $\theta'_v \leftarrow \theta_v$
    Get state s_t
    repeat
      Take action a_t according to policy $\pi_{\theta'}(a_t|s_t)$
      Receive reward r_t and new state s_{t+1}
      $T \leftarrow T + 1$, $t \leftarrow t + 1$
    until s_t is terminal **or** $t = t_{\max}$

$$R \leftarrow \begin{cases} 0 & \text{for terminal } s_t \\ V_{\theta'_v}(s_t) & \text{for non-terminal } s_t \end{cases} \quad \text{// Bootstrap from last state}$$

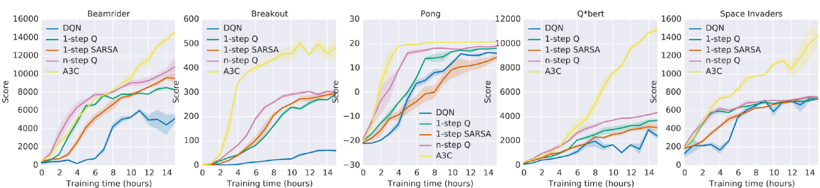
    for $i = t - 1$ **downto** 0 **do**
      $R \leftarrow r_i + \gamma R$
      Accumulate gradients w.r.t. θ': $d\theta \leftarrow d\theta + \big(R - V_{\theta'_v}(s_i)\big)\nabla_{\theta'} \ln \pi_{\theta'}(a_i|s_i)$
      Accumulate gradients w.r.t. θ'_v: $d\theta_v \leftarrow d\theta_v + \nabla_{\theta'_v}\big(R - V_{\theta'_v}(s_i)\big)^2$
    end for
    Perform asynchronous update of θ using $d\theta$
    Perform asynchronous update of θ_v using $d\theta_v$
  until $T > T_{\max}$

Experiments

- Evaluated training speeds of all 4 asynchronous methods on 5 Atari 2600 games
- Evaluated A3C's performance on 57 Atari 2600 games
- Previous methods trained on an NVIDIA K40 GPUs
- Asynchronous methods trained on 16 CPU cores
- For A3C, trained both a feedforward agent and a recurrent agent with an additional 256 LSTM cells
- Evaluation metric: *human starts* [Nair et al., 2015]
  - Obtain 100 starting points by random sampling from a human expert's trajectory
  - Run from each starting point for up to 30 minutes emulator time (108,000 frames)
  - Scores are *human-normalised* [van Hasselt et al., 2016]:

$$\text{score}_{\text{normalised}} = \frac{\text{score}_{\text{agent}} - \text{score}_{\text{random}}}{\text{score}_{\text{human}} - \text{score}_{\text{random}}}$$

Training speeds



Results

Game	DQN	Gorila	Double	Dueling	Prioritized	A3C FF, 1 day	A3C FF	A3C LSTM
Alien	570.2	813.5	1033.4	**1486.5**	900.5	182.1	518.4	945.3
Amidar	133.4	189.2	169.1	172.7	218.4	**283.9**	263.9	173.0
Assault	3332.3	1195.8	6060.8	3994.8	7748.5	3746.1	5474.9	**14497.9**
Asterix	124.5	3324.7	16837.0	15840.0	**31907.5**	6723.0	22140.5	17244.5
Asteroids	697.1	933.6	1193.2	2035.4	1654.0	3009.4	4474.5	**5093.1**
Atlantis	76108.0	629166.5	319688.0	445360.0	593642.0	772392.0	**911091.0**	875822.0
Bank Heist	176.3	399.4	886.0	**1129.3**	816.8	946.0	970.1	932.8
Battle Zone	17560.0	19938.0	24740.0	**31320.0**	29100.0	11340.0	12950.0	20760.0
Beam Rider	8672.4	3822.1	17417.2	14591.3	**26172.7**	13235.9	22707.9	24622.2
Berzerk			1011.1	910.6	1165.6	**1433.4**	817.9	862.2
Bowling	41.2	54.0	**69.6**	65.7	65.8	36.2	35.1	41.8
Boxing	25.8	74.2	73.5	**77.3**	68.6	33.7	59.8	37.3
Breakout	303.9	313.0	368.9	411.6	371.6	551.6	681.9	**766.8**
Centipede	3773.1	**6296.9**	3853.5	4881.0	3421.9	3306.5	3755.8	1997.0
Chopper Comman	3046.0	3191.8	3495.0	3784.0	6604.0	4669.0	7021.0	**10150.0**
Crazy Climber	50992.0	65451.0	113782.0	124566.0	131086.0	101624.0	112646.0	**138518.0**
Defender			27510.0	33996.0	21093.5	36242.5	56533.0	**233021.5**
Demon Attack	12835.2	14880.1	69803.4	56322.8	73185.8	84997.5	113308.4	**115201.9**
Double Dunk	-21.6	-11.3	-0.3	-0.8	**2.7**	0.1	-0.1	0.1
Enduro	475.6	71.0	1216.6	**2077.4**	1884.4	-82.2	-82.5	-82.5
Fishing Derby	-2.3	4.6	3.2	-4.1	9.2	13.6	18.8	**22.6**
Freeway	25.8	10.2	**28.8**	0.2	27.9	0.1	0.1	0.1
Frostbite	157.4	426.6	1448.1	2332.4	**2930.2**	180.1	190.5	197.6
Gopher	2731.8	4373.0	15253.0	20051.4	**57783.8**	8442.8	10022.8	17106.8
Gravitar	216.5	**538.4**	200.5	297.0	218.0	269.5	303.5	320.0
H.E.R.O.	12952.5	8963.4	14892.5	15207.9	20506.4	28765.8	**32464.1**	28889.5
Ice Hockey	-3.8	-1.7	-2.5	-1.3	**-1.0**	-4.7	-2.8	-1.7
James Bond	348.5	444.0	573.0	835.5	**3511.5**	351.5	541.0	613.0
Kangaroo	2696.0	1431.0	**11204.0**	10334.0	10241.0	106.0	94.0	125.0
Krull	3864.0	6363.1	6796.1	8051.6	7406.5	**8066.6**	5560.0	5911.4
Kung-Fu Master	11875.0	20620.0	30207.0	24288.0	31244.0	3046.0	28819.0	**40835.0**
Montezuma's Revenge	50.0	**84.0**	42.0	22.0	13.0	53.0	67.0	41.0
Ms. Pacman	763.5	1263.0	1241.3	**2250.6**	1824.6	594.4	653.7	850.7
Name This Game	5439.9	9238.5	8960.3	11185.1	11836.1	5614.0	10476.1	**12093.7**
Phoenix			12366.5	20410.5	27430.1	28181.8	52894.1	**74786.7**
Pit Fall			-186.7	-46.9	**-14.8**	-123.0	-78.5	-135.7
Pong	16.2	16.7	**19.1**	18.8	18.9	11.4	5.6	10.7
Private Eye	298.2	**2598.6**	-575.5	292.6	179.0	194.4	206.9	421.1
Q*Bert	4589.8	7089.8	11020.8	14175.8	11277.0	13752.3	15148.8	**21307.5**
River Raid	4065.3	5310.3	10838.4	16569.4	**18184.4**	10001.2	12201.8	6591.9
Road Runner	9264.0	43079.8	43156.0	58549.0	56990.0	31769.0	34216.0	**73949.0**
Robotank	58.5	61.8	59.1	**62.0**	55.4	2.3	32.8	2.6
Seaquest	2793.9	10145.9	14498.0	37361.6	**39096.7**	2300.2	2355.4	1326.1
Skiing			-11490.4	-11928.0	**-10852.8**	-13700.0	-10911.1	-14863.8
Solaris			810.0	1768.4	**2238.2**	1884.8	1956.0	1936.4
Space Invaders	1449.7	1183.3	2628.7	5993.1	9063.0	2214.7	15730.5	**23846.0**
Star Gunner	34081.0	14919.2	58365.0	90804.0	51959.0	64393.0	138218.0	**164766.0**
Surround			1.9	**4.0**	-0.9	-9.6	-9.7	-8.3
Tennis	-2.3	-0.7	-7.8	**4.4**	-2.0	-10.2	-6.3	-6.4
Time Pilot	5640.0	8267.8	6608.0	6601.0	7448.0	5825.0	12679.0	**27202.0**
Tutankham	32.4	118.5	92.2	48.0	33.6	26.1	**156.3**	144.2
Up and Down	3311.3	8747.7	19086.9	24759.2	29443.7	54525.4	74705.7	**105728.7**
Venture	54.0	**523.4**	21.0	200.0	244.0	19.0	23.0	25.0
Video Pinball	20228.1	112093.4	367823.7	110976.2	374886.9	185852.6	331628.1	**470310.5**
Wizard of Wor	246.0	10431.0	6201.0	7054.0	7451.0	5278.0	17244.0	**18082.0**
Yars Revenge			6270.6	**25976.5**	5965.1	7270.8	7157.5	5615.5
Zaxxon	831.0	6159.4	8593.0	10164.0	9501.0	2659.0	**24622.0**	23519.0

Results

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

Table: Mean and median human-normalised scores on 57 Atari games

More challenging tasks

- TORCS 3D car racing: more realistic graphics & dynamics

  https://youtu.be/0xo1Ldx3L5Q

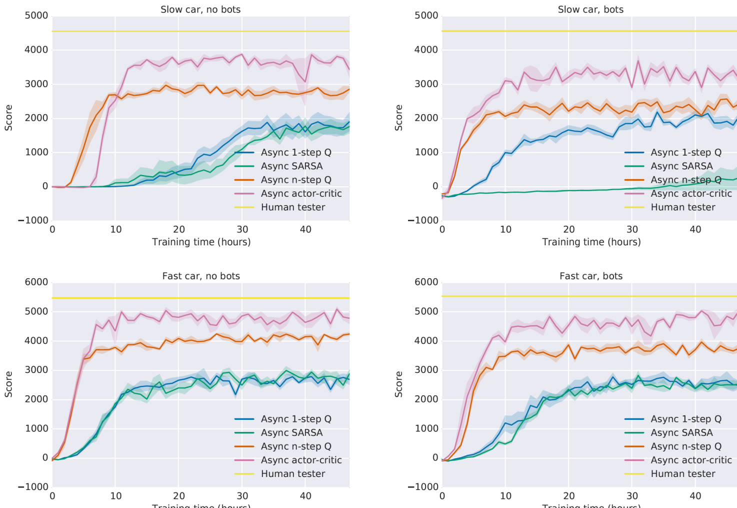
- MuJoCo physics engine: continuous action control
  - Pole swing-up, quadruped locomotion, planar biped walking, balancing, 2D target reaching, 3D manipulation, etc.

  https://youtu.be/Ajjc08-iPx8

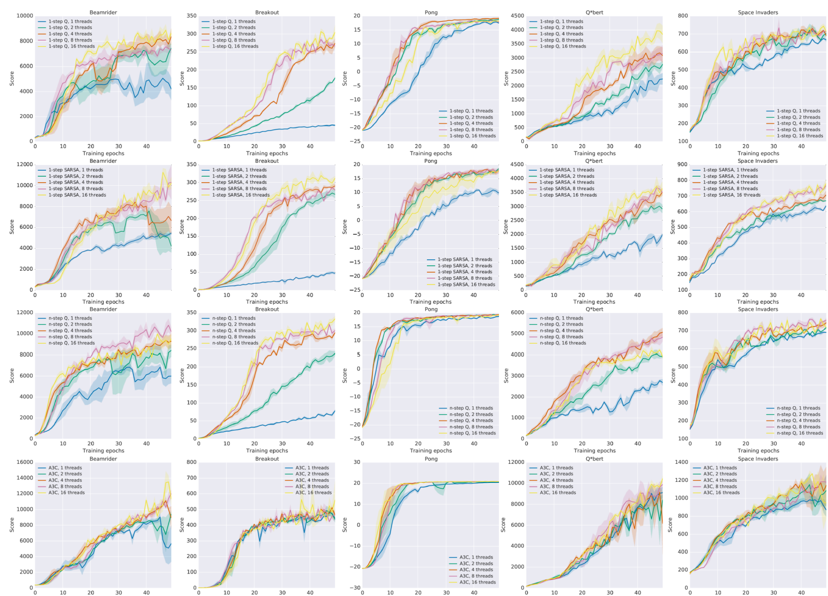
- Labyrinth: 3D environment, maze randomly generated at every episode

  https://youtu.be/nMR5mjCFZCw

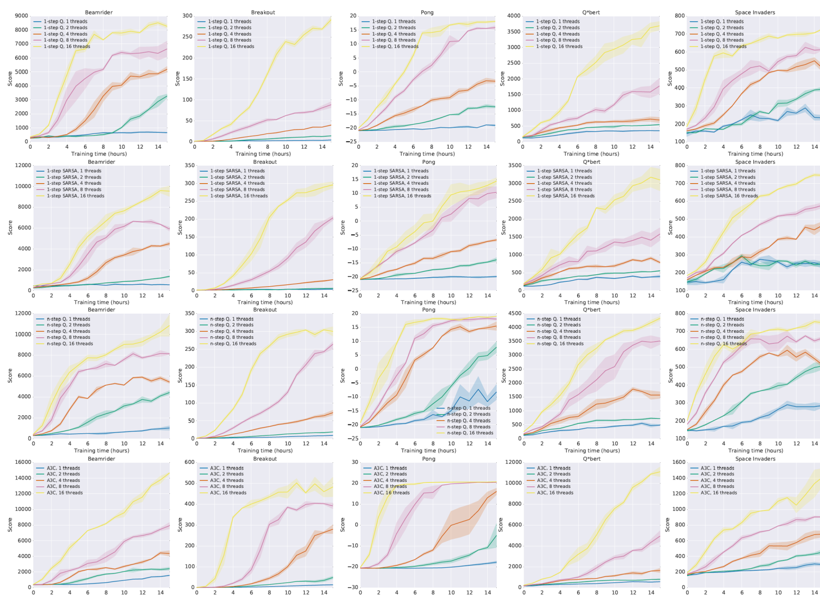
TORCS 3D car racing results



Data efficiency



Training speed-up



Training speed-up

Method	Number of threads				
	1	2	4	8	16
1-step Q-learning	1.0	**3.0**	**6.3**	**13.3**	**24.1**
1-step Sarsa	1.0	**2.8**	**5.9**	**13.1**	**22.1**
n-step Q-learning	1.0	**2.7**	**5.9**	**10.7**	**17.2**
A3C	1.0	2.1	3.7	6.9	12.5

Table: Average training speed-up over 7 Atari games

Super-linear speed-up for the one-step methods!

Final discussions

Other good things about this paper:

- Reports hyperparameter tuning details
- Performs robustness and stability tests

Limitations:

- No comparison between A3C and A2C
- No analysis on performance of methods other than A3C
- Can we improve other asynchronous methods, and how?

Outlooks:

- Can be combined with DDPG
- Multi-agent RL – MADDPG

References

Hado van Hasselt, Arthur Guez, and David Silver (2016).
Deep Reinforcement Learning with Double Q-Learning.
In *Proceedings of AAAI-16*, *30*(1):2094–2100.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves et al. (2013).
Playing Atari with Deep Reinforcement Learning.
In *NIPS Deep Learning Workshop*.

Volodymyr Mnih, Adrià P. Badia, Mehdi Mirza, Alex Graves et al. (2016).
Asynchronous Methods for Deep Reinforcement Learning.
In *Proceedings of ICML 2016*, PMLR *48*:1928–1937.

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek et al. (2015).
Massively Parallel Methods for Deep Reinforcement Learning.
In *ICML Workshop on Deep Learning*.

Richard S. Sutton and Andrew G. Barto (1998 & 2018).
Reinforcement Learning: An Introduction (1st & 2nd editions).
MIT Press.

Thank you!