

Deep Reinforcement Learning

Advantage actor-critic (A2C, A3C)

Julien Vitay Professur für Künstliche Intelligenz - Fakultät für Informatik



1 - Distributed RL

Advantage actor-critic

 \equiv

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

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



action
$$a \sim \pi_{\theta}(s, a)$$

$$\sum_{\rho_{\theta}, a \sim \pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) (R^{n} - V_{\varphi}(s))]$$

Advantage actor-critic

Ξ

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

$$egin{aligned} & egin{split}
abla_ heta \mathcal{J}(heta) = \mathbb{E}_{s_t \sim
ho_ heta, a_t \sim \pi_ heta} [
abla_ heta \log \pi_ heta(s_t) \ & egin{split}
olimits \mathcal{L}(arphi) = \mathbb{E}_{s_t \sim
ho_ heta, a_t \sim \pi_ heta} [(R_t^n - V_arphi(s_t))^2 \ & eta) \end{bmatrix} \end{split}$$

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

 $(k,a_t)\left(R_t^n-V_arphi(s_t))
ight)$

Distributed RL

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



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



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

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

Distributed RL



Distributed RL

- Initialize global policy or value network θ .
- Initialize N copies of the environment in different states.
- while True:

- **for** each worker in parallel:
 - Copy the global network parameters θ to each worker:

$$heta_k \leftarrow heta$$

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

2 - A3C: Asynchronous advantage actor-critic

Asynchronous Methods for Deep Reinforcement Learning

Volodymyr Mnih¹ Adrià Puigdomènech Badia¹ Mehdi Mirza^{1,2} Alex Graves¹ Tim Harley¹ **Timothy P. Lillicrap**¹ **David Silver**¹ Koray Kavukcuoglu¹ ¹ Google DeepMind ² Montreal Institute for Learning Algorithms (MILA), University of Montreal

Ξ

VMNIH@GOOGLE.COM ADRIAP@GOOGLE.COM MIRZAMOM@IRO.UMONTREAL.CA GRAVESA@GOOGLE.COM THARLEY@GOOGLE.COM COUNTZERO@GOOGLE.COM DAVIDSILVER@GOOGLE.COM KORAYK@GOOGLE.COM

A3C: Asynchronous advantage actor-critic

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

$$abla_ heta \mathcal{J}(heta) = \mathbb{E}_{s_t \sim
ho_ heta, a_t \sim \pi_ heta} [
abla_ heta \log \pi_ heta(s_t, a_t) \left(R_t^n - V_arphi(s_t)
ight)]$$



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

 \equiv

• The value of a state $V_{arphi}(s)$ is produced by the **critic** with weights φ , which minimizes the mse with the

$$egin{split} &arphi) = \mathbb{E}_{s_t \sim
ho_ heta, a_t \sim \pi_ heta} [(R_t^n - V_arphi(s_t))^2] \ &arphi & arphi = \sum_{k=0}^{n-1} \gamma^k \, r_{t+k+1} + \gamma^n \, V_arphi(s_{t+n}) \end{split}$$

A3C: Asynchronous advantage actor-critic



- **def** worker(θ, φ):
 - Initialize empty transition buffer \mathcal{D} . Initialize the environment to the **last** state visited by this worker.
 - for *n* steps:
 - Select an action using π_{θ} , store the transition in the transition buffer.
 - for each transition in \mathcal{D} :
 - $\circ~$ Compute the **n-step return** in each state $R_t^n = \sum \gamma^k$ $k{=}0$
 - Compute **policy gradient** for the actor on the transition buffer:

$$d heta =
abla_ heta \mathcal{J}(heta) = rac{1}{n} \sum_{t=1}^n
abla_ heta \log \pi_ heta(s_t, a_t) \left(R_t^n - V_arphi(s_t)
ight)$$

 $n{-}1$

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

$$darphi =
abla_arphi \mathcal{L}(arphi) = -rac{1}{n}\sum_{t=1}^n (R_t^n - V_arphi(s_t)) \,
abla_arphi V_arphi(s_t)$$

- return d heta, darphi

$$^k \, r_{t+k+1} + \gamma^n \, V_arphi(s_{t+n})$$

A2C: global networks

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

$$\circ \ d heta_k$$
, $darphi_k$ = worker($heta$, $arphi$)

join()

 \equiv

Merge all gradients:

$$d heta=rac{1}{K}\sum_{i=1}^{K}d heta_k \ ; \ darphi=rac{1}{K}\sum_{i=1}^{K}darphi_k$$

Update the actor and critic using gradient ascent/descent:

$$heta \leftarrow heta + \eta \, d heta \; ; \; arphi \leftarrow arphi - \eta \, darphi$$

A3C: Asynchronous advantage actor-critic

- The previous slide depicts **A2C**, the synchronous version of A3C.
- A2C synchronizes the workers (threads), i.e. it waits for the K workers to finish their job before merging the gradients and updating the global networks.
- A3C is **asynchronous**:

- the partial gradients are applied to the global networks as soon as they are available.
- No need to wait for all workers to finish their job.
- As the workers are not synchronized, this means that one worker could be copying the global networks hetaand φ while another worker is writing them.
- This is called a **Hogwild!** update: no locks, no semaphores. Many workers can read/write the same data.
- It turns out NN are robust enough for this kind of updates.

A3C: asynchronous updates

- Initialize actor θ and critic φ .
- Initialize K workers with a copy of the environment.
- for K workers in parallel:
 - for $t \in [0, T_{ ext{total}}]$:

 \equiv

- Copy the global networks heta and arphi.
- Compute partial gradients:

$$d heta_k, darphi_k = \mathrm{worker}(heta, arphi)$$

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

$$egin{aligned} & heta & \leftarrow heta & + \eta \, d heta_k \ & arphi & \leftarrow arphi & - \eta \, darphi_k \end{aligned}$$

A3C: Asynchronous advantage actor-critic



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

• A3C does not use an *experience replay memory*, but relies on **multiple parallel workers** to distribute

Each worker has a copy of the actor and critic networks, as well as an instance of the

• Weight updates are synchronized regularly though a **master network** using Hogwild!-style updates (every n = 5 steps!).

Because the workers learn different parts of the state-action space, the weight updates are not very

A3C : results

 \equiv



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

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

- lacksquarewith DQN.

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

• A3C set a new record for Atari games in 2016.

The main advantage is that the workers gather experience in parallel: training is much faster than

• LSTMs can be used to improve the performance.

A3C : results

• Learning is only marginally better with more threads:



but much faster!



A3C: TORCS simulator





A3C: Labyrinth





A3C: continuous control problems



Comparison with DQN

 \equiv

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



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

3 - Actor-critic neural architectures

Actor-critic neural architectures

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



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



$$\sum_{\rho_{\theta}, a \sim \pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) (R^{n} - V_{\varphi}(s))]$$

Early visual features

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



Shared architectures

Ξ

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

$$\mathcal{L}(heta) = \mathcal{L}_{ ext{actor}}(heta) + \mathcal{L}_{ ext{cri}}$$

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



 $_{
m itic}(heta)$



4 - Continuous stochastic policies

Continuous action spaces

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

$$\mathbf{a} = egin{bmatrix} \Delta heta_1 & \Delta heta_2 & \dots & \Delta heta_n \end{bmatrix}^T$$

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



Gaussian policies

 \equiv

- A Gaussian policy considers the vector **a** to be sampled from the **normal distribution** $\mathcal{N}(\mu_{\theta}(s), \sigma_{\theta}(s))$.
- The mean $\mu_{\theta}(s)$ and standard deviation $\sigma_{\theta}(s)$ are output vectors of the **actor** with parameters θ .
- **Sampling** an action from the normal distribution is done through the **reparameterization trick**:

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

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



ξ

Gaussian policies

Ξ

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

$$\pi_ heta(s,a) = rac{1}{\sqrt{2\pi\sigma_ heta^2(s)}} \, \exp -rac{(a-\mu_ heta(s))^2}{2\sigma_ heta^2(s)}$$

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

$$\log \pi_ heta(s,a) = -rac{(a-\mu_ heta(s))^2}{2\sigma_ heta^2(s)} - rac{1}{2}\,\log 2\pi\sigma_ heta^2(s)$$

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

$$egin{aligned} & \displaystyle \left\{
abla_{\mu_{ heta}(s)} \log \pi_{ heta}(s,a) = rac{a-\mu_{ heta}(s)}{\sigma_{ heta}(s)^2} \ & \displaystyle \left\{
abla_{\sigma_{ heta}(s)} \log \pi_{ heta}(s,a) = rac{(a-\mu_{ heta}(s))^2}{\sigma_{ heta}(s)^3} - rac{1}{\sigma_{ heta}(s)} \end{aligned}
ight.$$

/ \

Gaussian policies

 \equiv

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

$$\mathbf{a} = \mu_ heta(s) + \sigma_ heta(s)$$
 .

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

$$egin{aligned} & \left\{
abla_{\mu_{ heta}(s)} \log \pi_{ heta}(s,a) = rac{a-\mu_{ heta}(s)}{\sigma_{ heta}(s)^2} \ & \left\{
abla_{\sigma_{ heta}(s)} \log \pi_{ heta}(s,a) = rac{(a-\mu_{ heta}(s))^2}{\sigma_{ heta}(s)^3} - rac{1}{\sigma_{ heta}(s)} \end{aligned}
ight\}$$

- The rest of the score ($\nabla_{\theta} \mu_{\theta}(s)$ and $\nabla_{\theta} \sigma_{\theta}(s)$) is the problem of tensorflow/pytorch.
- This is the same **reparametrization trick** used in variational autoencoders to allow backpropagation to work through a sampling operation.
- Beta distributions are an even better choice to parameterize stochastic policies (Chou et al., 2017).

ξ



References

- Chou, P.-W., Maturana, D., and Scherer, S. (2017). Improving Stochastic Policy Gradients in Continuous Control with Deep Reinforcement Learning using the Beta Distribution. in International Conference on Machine Learning http://proceedings.mlr.press/v70/chou17a/chou17a.pdf.
- Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., et al. (2017). Rainbow: Combining Improvements in Deep Reinforcement Learning. http://arxiv.org/abs/1710.02298.
- Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., et al. (2016). Asynchronous Methods for Deep Reinforcement Learning. in Proc. ICML http://arxiv.org/abs/1602.01783.
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., et al. (2013). Playing Atari with Deep Reinforcement Learning. http://arxiv.org/abs/1312.5602.