# Asynchronous Methods for Deep Reinforcement Learning

**Introduction：**

在深度神经网络控制器的优化中使用异步梯度下降训练的方法 asynchronous gradient descent。有 4 种异步变种，效果最好的是 AC 使用异步方法，叫做 A3C。

经验回放 Experience replay memory 可以减少 on-policy 观测数据的相关性，但同时限制 off-policy。它需要 off-policy 从旧策略当中更新。我们异步并行执行多个智能体，在多个不同的环境部分中。

并行也能使得智能体在稳定进程中互不相关，因为每一步在不同环境部分下并行的智能体是处于不同状态的。

异步中最推荐的方法是 Asynchronous Advantage Actor-critic (A3C)。

**Related Work：**

异步方法通过每个智能体使用不同的探索的策略来最大化这种差异。每一个进程都有一个 actor 根据独立的回放记忆进行选择动作，policy-based，而每一个学习者都会从回放记忆中采样计算 DQN loss，是 value-based 的。

早些方法是应用 Map Reduce 并行框架下使用带线性函数近似器的批量强化学习方法。

https://baike.baidu.com/item/MapReduce/133425?fr=aladdin

异步也收敛，异步在 q-learning 下依然可以收敛。

也有早些方法叫进化方法直接在多个计算机和线程中发布合适的进化版本实现并行。

**RL Background:**

$$L_i(\theta_i) = \mathbb{E}\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right)^2$$

where $s'$ is the state encountered after state $s$.

上式表示一步 Q-learning，因为它更新 action value 只通过一步累计回报值 one-step return r + γmaxa0 Q(s0, a0; θ)。

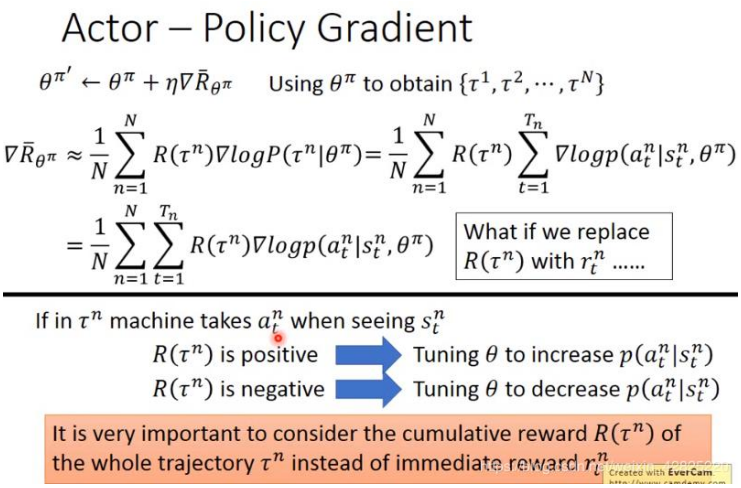
其他的状态动作对只是间接影响 Q 值的更新。这会使得更新很慢，因为需要一步一步通过相关的状态动作对进行更新。

n-step Q-learning

AC：

衡量 actor：expected total reward 期望回报累加值(因为即使是相同的 actor，每次得到的回报还是不同：①面对同样的画面，随机策略会导致结果不同；②即使是确定性策略，环境也有可能是随机的)。

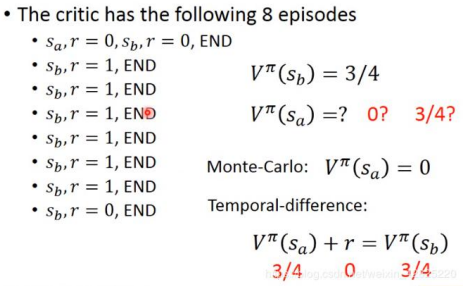
Actor 的梯度更新：提升正向回报的几率，降低负值回报的几率，这里的回报值采用的是累计回报值而不是即时回报。



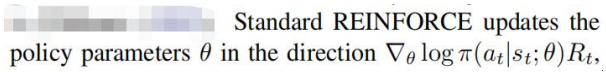

Critic：评估 actor π的好坏

MC 方法/TD 方法

MC 比 TD 的方差更大，MC 是无偏估计，TD 是有偏估计




如果环境有马尔科夫特性，即为不受前一状态影响，那么 TD 比较准确，否则 MC 比较准确。

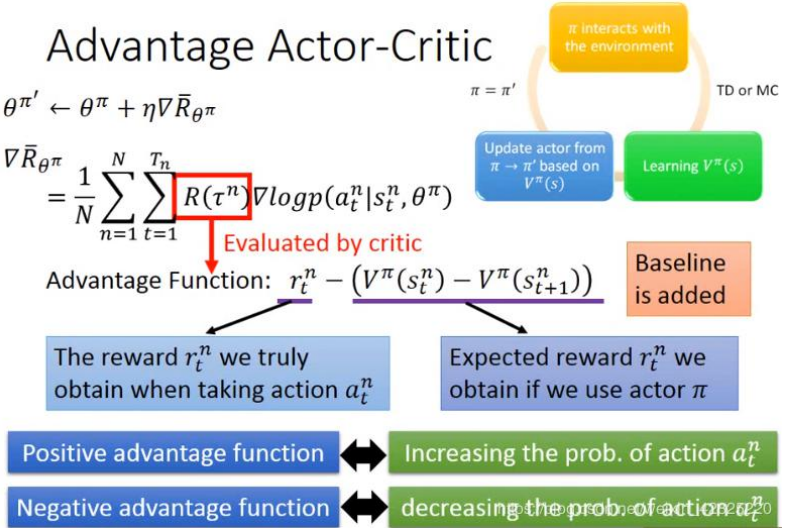



我们可以通过使用上式减去一个状态估计值 bt 作为 baseline 以减少估计的方差。

$$\nabla_\theta \log \pi(a_t|s_t;\theta)\,(R_t - b_t(s_t)).$$

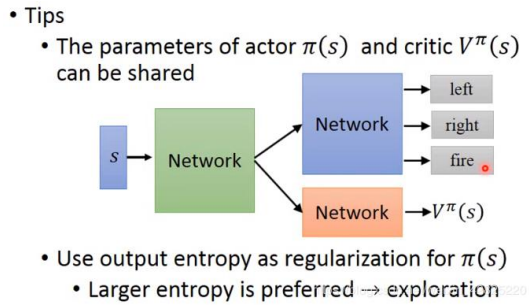
这种方法下带有 baseline 的是 critic。

而将 R 改造为优势函数是 Advantage Actor-Critic，A2C，如下所示：



Actor 和 critic 训练技巧：

Actor 和 critic 部分网络参数可以共享。输出的熵可以尽量大->输出的分布偏向于平滑而不是集中->actor 可以增加探索的机会。




**Asynchronous RL Framework：**

AC 是 on-policy 的搜索方法是 policy-based 和 value-based 的交叉。Q-learning 是 off-policy 的 value-based 方法。第二点我们通过给并行的多个 actors 接收不同的 observation，使得它们尽可能探索环境的不同部分。

通过在不同的线程进程中探索不同的策略，能使得多个并行的 actors 在线更新参数在时间上有比单智能体在线更新参数具有更少的相关性。增加并行性，进行多线程，每个线程使用不同的 policy 可以减小 policy 和数据的相关性。

此外，并行的 actors 还有其他实用性优点。第一，减少了训练时间；第二不需要依赖经验回放技巧。

1. Asynchronous one-step Q-learning:

**Algorithm 1** Asynchronous one-step Q-learning - pseudocode for each actor-learner thread.

*// Assume global shared* $\theta$, $\theta^{-}$, *and counter* $T = 0$.
Initialize thread step counter $t \leftarrow 0$
Initialize target network weights $\theta^{-} \leftarrow \theta$
Initialize network gradients $d\theta \leftarrow 0$
Get initial state $s$
**repeat**
  Take action $a$ with $\epsilon$-greedy policy based on $Q(s, a; \theta)$
  Receive new state $s'$ and reward $r$
  $y = \begin{cases} r & \text{for terminal } s' \\ r + \gamma \max_{a'} Q(s', a'; \theta^{-}) & \text{for non-terminal } s' \end{cases}$
  Accumulate gradients wrt $\theta$: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$
  $s = s'$
  $T \leftarrow T + 1$ and $t \leftarrow t + 1$
  **if** $T \mod I_{target} == 0$ **then**
    Update the target network $\theta^{-} \leftarrow \theta$
  **end if**
  **if** $t \mod I_{AsyncUpdate} == 0$ or $s$ is terminal **then**
    Perform asynchronous update of $\theta$ using $d\theta$.
    Clear gradients $d\theta \leftarrow 0$.
  **end if**
**until** $T > T_{max}$

两种更新网络参数方式：类似于 DQN，在一段时间后，将当前的网络参数复制到目标网络中；每一个 learner 对当前网络的参数的更新是异步的 diversity to exploration。在一段时间后，将梯度求和用于更新网络参数，这样就模仿了 mini batch。是平衡计算效率和数据利用效率的方法。

2. Asynchronous one-step Sarsa:

One-step Sarsa 的 target value 估计值使用 r +γQ(s', a'; θ- ) 其中 a'是根据参数θ-下在状态 s'选择的动作。

3. Asynchronous n-step Q-learning:

算法有些许不同，因为算法会使用前瞻性的技巧，根据 policy 选择多步 action，直到 Tn 步或者达到终止状态，得到 n 个奖励值，根据累计回报值的公式进行计算，然后使用 eligibility traces 技巧进行参数更新。

**Algorithm S2** Asynchronous n-step Q-learning - pseudocode for each actor-learner thread.

*// Assume global shared parameter vector $\theta$.*
*// Assume global shared target parameter vector $\theta^-$.*
*// Assume global shared counter $T = 0$.*
Initialize thread step counter $t \leftarrow 1$
Initialize target network parameters $\theta^- \leftarrow \theta$
Initialize thread-specific parameters $\theta' = \theta$
Initialize network gradients $d\theta \leftarrow 0$
**repeat**
  Clear gradients $d\theta \leftarrow 0$
  Synchronize thread-specific parameters $\theta' = \theta$
  $t_{start} = t$
  Get state $s_t$
  **repeat**
    Take action $a_t$ according to the $\epsilon$-greedy policy based on $Q(s_t, a; \theta')$
    Receive reward $r_t$ and new state $s_{t+1}$
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  **until** terminal $s_t$ **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ \max_a Q(s_t, a; \theta^-) & \text{for non-terminal } s_t \end{cases}$
  **for** $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \frac{\partial (R - Q(s_i, a_i; \theta'))^2}{\partial \theta'}$
  **end for**
  Perform asynchronous update of $\theta$ using $d\theta$.
  **if** $T \mod I_{target} == 0$ **then**
    $\theta^- \leftarrow \theta$
  **end if**
**until** $T > T_{max}$

4. Asynchronous advantage actor-critic:

maintains a policy $\pi(a_t|s_t;\theta)$ and an estimate of the value function $V(s_t;\theta_v)$.

AC 的变种也可以有前瞻性，使用 n-step 进行更新策略和估计值。

The update performed by the algorithm can be seen as $\nabla_{\theta'} \log \pi(a_t|s_t;\theta')A(s_t, a_t; \theta, \theta_v)$ where $A(s_t, a_t; \theta, \theta_v)$ is an estimate of the advantage function given by $\sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v)$, where $k$ can vary from state to state and is upper-bounded by t-max.

特别的策略的参数θ和价值函数的参数θv 为了泛化是互相独立的，但我们为了算法的实用总是会共享一些参数。

特别的我们使用带有一个 soft-max 输出策略π(at|st; θ)和一个线性输出价值函数 V (st; θv)的卷积神经网络这两个输出层被所有不输出层共享。

将策略的熵 entropy 加入到目标函数，以增加探索性，以防止过早收敛到局部最优的 policy。

策略的熵特别对需要分级的任务效果拔群。

**Algorithm S3** Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

*// Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$*
*// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$*
Initialize thread step counter $t \leftarrow 1$
**repeat**
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state $s_t$
  **repeat**
    Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
    Receive reward $r_t$ and new state $s_{t+1}$
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  **until** terminal $s_t$ **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  **for** $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt $\theta'_v$: $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i;\theta'_v))^2 / \partial \theta'_v$
  **end for**
  Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
**until** $T > T_{max}$

The gradient of the full objective function including the entropy regularization term with respect to the policy parameters takes the form $\nabla_{\theta'} \log \pi(a_t|s_t;\theta')(R_t - V(s_t;\theta_v)) + \beta\nabla_{\theta'} H(\pi(s_t;\theta'))$, where $H$ is the entropy. The hyperparameter $\beta$ controls the strength of the entropy regularization term.

β表示 entropy regularization 的强度。

首先说一下 A3C 算法的个人理解，他是基于 Actor-Critic 的一种改进算法，主要是利用多线程维持多个 agent 学习并分享各自的经验，达到更快的学习速率和更好的训练效用。类似于三个工人和一个管理者共同解决一个难题，三个工人在解决过程中向管理者汇报自己的解题经验，而管理者汇总三个人的经验再将总结后的经验发放给三个工人，让他们获得集体的经验并继续解决问题。

具体到 A3C 算法上，这个算法维持着一个中央大脑 global，以及 n 个独立个体 worker（n 一般取 cpu 数量，有多少个 cpu 就有多少个 worker）。global 和 worker 各自维持着结构完全相同的网络，Actor 网络和 Critic 网络。worker 通过 actor—critic 网络获取近几个回合所获得的经验并上传到 global 网络中，global 汇集几个 worker 的经验后下发给每一个 worker，这里所说的经验，其实就是 actor 和 critic 网络的参数。

需要着重注意的两点，首先是这个上传经验，上传的是损失 loss 对神经网络参数求导所得到的梯度，global 从 worker 中获取到这个梯度并更新自己的网络参数。下载经验，是 worker 直接将 global 中的神经网络参数全盘接受，并覆盖当前自己的神经网络 actor 和 critic。

另外，相较于普通的 Actor-critic 网络，每一个 worker 没有自我学习的过程，在 Actor-critic 中，worker 通过 optimizer 最小化损失 loss 并更新自己的网络参数。而在 A3C 算法中，这一过程被 global-net 取代了，更新自己的网络参数只需从 global 中获取最新的参数即可（集体的智慧）。

5. Optimization:

对于 momentum SGD，每一个线程单独维护了梯度，momentum，用于更新参数 SGD。with momentum, RMSProp without shared statistics 各自维护自己的 g 的 RMSProp, RMSProp with shared statistics 维护共有的 g 的 RMSProp.

作者使用标准去中心化 RMSProp，就是线程各自维护自己的 g：

$$g = \alpha g + (1-\alpha)\Delta\theta^2 \text{ and } \theta \leftarrow \theta - \eta\frac{\Delta\theta}{\sqrt{g+\epsilon}}, \qquad (1)$$

有些版本也会让所有线程共享一个 g，但是异步更新它，并且不加锁。

**Experiments：**

5.1. Atari 2600 Games

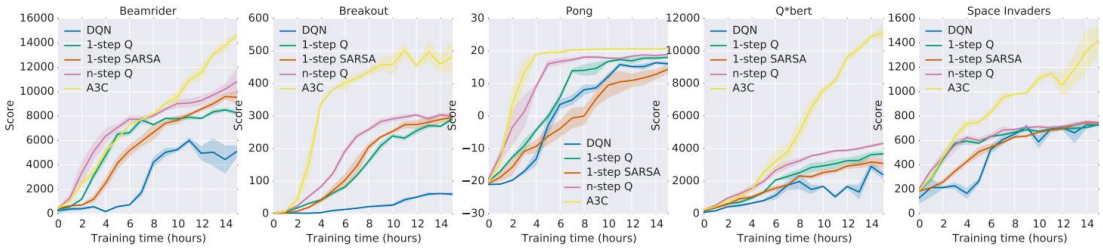



*Figure 1.* Learning speed comparison for DQN and the new asynchronous algorithms on five Atari 2600 games. DQN was trained on a single Nvidia K40 GPU while the asynchronous methods were trained using 16 CPU cores. The plots are averaged over 5 runs. In the case of DQN the runs were for different seeds 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.

总之，A3C 的效果优于其他三种带异步的 value-based 方法。

| 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% |

*Table 1.* Mean and median human-normalized scores on 57 Atari games using the human starts evaluation metric. Supplementary Table S3 shows the raw scores for all games.

## 5.2. TORCS Car Racing Simulator

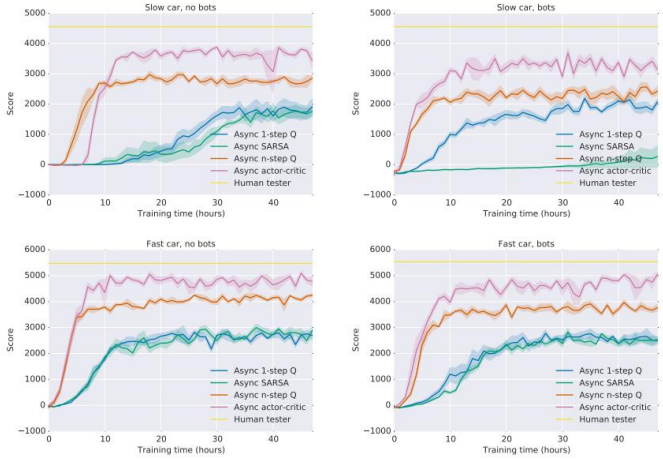



*Figure S6.* Comparison of algorithms on the TORCS car racing simulator. Four different configurations of car speed and opponent presence or absence are shown. In each plot, all four algorithms (one-step Q, one-step Sarsa, $n$-step Q and Advantage Actor-Critic) are compared on score vs training time in wall clock hours. Multi-step algorithms achieve better policies much faster than one-step algorithms on all four levels. The curves show averages over the 5 best runs from 50 experiments with learning rates sampled from $LogUniform(10^{-4}, 10^{-2})$ and all other hyperparameters fixed.

## 5.3. Continuous Action Control Using the MuJoCo Physics Simulator

**Found in Section 9**

## 5.4. Labyrinth 魔幻迷宫

魔幻迷宫是一个新的 3D 环境，在该随机产生的初始化视觉图像迷宫的环境下，智能体必须学会发现奖励。**(Details in Section 8)**

5.5. Scalability and Data Efficiency 可扩展性和数据效率

我们通过增加并行智能体的个数观察训练时间和数据效率是怎么改变的以分析我们推荐的算法框架下的效果。

Table 2 表示在 7 个 Atari games 中，并行智能体数量的增加显著减小了训练时间。

| | Number of threads | | | | |
|---|---|---|---|---|---|
| Method | 1 | 2 | 4 | 8 | 16 |
| 1-step Q | 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 | 1.0 | **2.7** | **5.9** | **10.7** | **17.2** |
| A3C | 1.0 | 2.1 | 3.7 | 6.9 | 12.5 |

*Table 2.* The average training speedup for each method and number of threads averaged over seven Atari games. To compute the training speed-up on a single game we measured the time to required reach a fixed reference score using each method and number of threads. The speedup from using $n$ threads on a game was defined as the time required to reach a fixed reference score using one thread divided the time required to reach the reference score using $n$ threads. The table shows the speedups averaged over seven Atari games (Beamrider, Breakout, Enduro, Pong, Q*bert, Seaquest, and Space Invaders).

这是因为多线程并行对于减少单步更新方法的偏差有正向作用。

**5.6.** Robustness and Stability 鲁棒性和稳定性

在具有良好学习率的区域中几乎没有分数为 0 的点的事实表明该方法是稳定的并且一旦它们被学习就不会崩溃或发散。

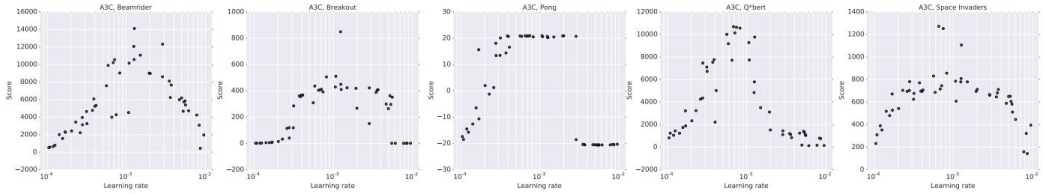



*Figure 2.* Scatter plots of scores obtained by asynchronous advantage actor-critic on five games (Beamrider, Breakout, Pong, Q*bert, Space Invaders) for 50 different learning rates and random initializations. On each game, there is a wide range of learning rates for which all random initializations acheive good scores. This shows that A3C is quite robust to learning rates and initial random weights.

**Conclusions and Discussion：**

通常训练时间减少了一半。

我们其中的一个主要发现是使用并行 AC 更新一个 global Net 的方法比其他异步 value-based 方法更加稳定和有效。

在异步学习的不相关经验回放可以充分利用旧数据，从中获得有效信息。

相对于使用 n-steps 方法利用前瞻性的 n 步相关回报计算累计回报直接作为 target 值，**更经常地做法是使用后看方法 eligibility traces 隐式组合不同的累计回报值。**

未来的方向是在真实的在线的 TD 方法中加入近来的工作。

**https://www.jianshu.com/p/38d997805bc6**

**https://blog.csdn.net/weixin_42825220/article/details/97297546**

**https://blog.csdn.net/Tiberium_discover/article/details/84198209**