天津工業大學

TIANJIN POLYTECHNIC UNIVERSITY

计算机科学与技术学院

基于强化学习的‘Pong’游戏玩家

成　员：

王　殊　1611640413

李　彪　1610650213

全　力　1611640320

苏　宁　1611640421

班　级：　计算机 s1601

指 导 老 师：　刘丁

2019 年 1 月 10 日

目录

一、 简介

强化学习 (Reinforcement Learning) 是机器学习的一个领域, 其强调如何基于环境而行动, 以取得最大化的预期利益.

我们的工作基于强化学习, 引导 Agent 去玩“Atair”系列的“乓”(Pong) 游戏. 我们主要使用了基于策略梯度 (Policy Gradient) 的算法: Asynchronous Advantage Actor-Critic (A3C) 来对 Agent 进行训练.

二、 实验环境与分工

2.1 实验环境

- 机器学习框架: TensorFlow
- 游戏模拟平台: OpenAI Gym

2.2 实验分工

实验主要分工为:

- 王殊: 环境搭建与代码调试
- 李彪: 结果分析和论文书写
- 全力: 收集资料与协助
- 苏宁: PPT 准备和演讲

三、 相关知识

3.1 游戏规则

1972 年, 美国的雅达利公司的《PONG》是模拟两个人打乒乓球, 在两条线中间有一个点在动. 当时是很著名的游戏……(Atari 2600). 它的游戏截图如下:

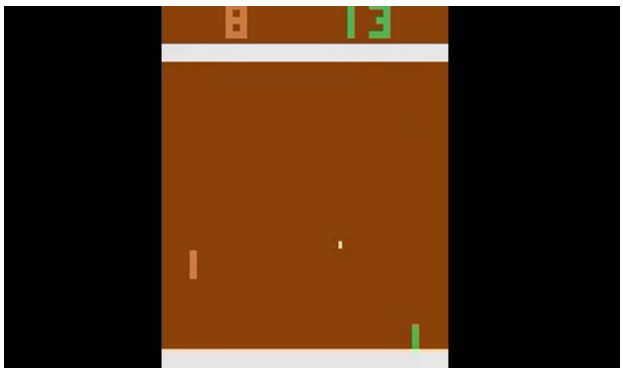

图 **1**　游戏截图

Pong 游戏左侧的板子 (paddle) 是计算机玩家, 右侧的是我们的 Agent. 如果我们的 Agent 采取的行动导致了:

1 我方无法接球, 即时回报为-1

2 对方无法接球, 即时回报为 1

3 否则, 即时回报为 0

3.2 马尔可夫决策过程

马尔科夫决策过程 (Markov Decision Processes, MDP)[1], 是强化学习研究的理论基石, MDP 过程是一种随机过程, 该模型能够提供一种非常简便的表达方式, 对于解决序贯决策问题 (Sequential Decision) 十分有效, 几乎所有的 RL 问题都能通过 MDP 来描述.

马尔科夫决策过程以马尔可夫随机过程为理论基础, 马尔科夫决策过程可以用一个元组 (S, A, P, R, γ) 来表示. S 是决策过程中的状态集合; A 是决策过程中的动作集合; P 是状态之间的转移概率; R 是采取某一动作到达下一状态后的回报 (也可看作奖励) 值; γ 是折扣因子. 特别地, 这里的转移概率与马尔科夫随机过程不同, 这里的转移概率是加入了动作 A 的概率, 如果当前状态采用不同动作, 那么到达的下一个状态也不一样, 自然转移概率也不一样. 转移概率形式化描述是:

$$p^{a}_{(s\hat{s})} = P(\hat{s}|s, a) = P(S_{t+1} = \hat{s}|S_t = s, A_t = a)$$

上式中 $p^{a}_{(s\hat{s})}$ 表示: 在 t 时刻所处的状态是 s, 采取 a 动作后在 $t+1$ 时刻到达 $\hat{s}$ 的概率.

在马尔科夫随机过程中我们专注于状态之间的改变, 而状态之间是怎么改变的我们并不关心, 往往是通过转移概率来衡量不同状态之间转移的可能性大小; 而在马尔科夫决策过程中, 多了一个决策, 这个决策也就是我们前面所说的动作, 在采用什么动作后, 到达下一时刻的状态, 并且给这个决策一个回报值来衡量该决策的好坏.

决策可以用 $\pi(a \,\big|\, s) = p(A_t = a \,\big|\, S_t = s)$ 来表示, 意思时在 t 时刻处于状态 s 的情况下, 选用 a 动作的概率. 即: 在每个状态 s 采取的动作 a 并不确定, 那么状态序列也不一样. 状态 s_t 到 s_{t+1} 会有一个回报, s_{t+1} 到 s_{t+2} 同样会有一个回报, 以此类推. 但 s_t 对 s_{t+1} 的影响很大, 对于 $s_{t+2}, s_{t+3}...$ 的影响会越来越小, 所以提出了一个折扣因子 γ 来减小后面状态的回报对当前状态衡量的影响. 于是我们得出所谓的累计回报函数:

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

由于 G_t 并非一个确定值 (中间涉及到概率选择动作), 所以采用期望来计算这个累计回报函数, 也叫做状态值函数, 也就是:

$$v_\pi(s) = E_\pi[G_t] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s]$$

在这个式子中, 状态值函数 $V_\pi(s)$ 仅仅和策略 π 有关, 也就是说策略唯一确定了状态值函数的分布. 而策略由动作 a 确定, 所以将动作加入其中:

$$q_\pi(s,a) = E_\pi[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s, A_t = a]$$

$q_\pi(s,a)$ 又叫做状态-动作函数. 可以发现 $v_\pi(s)$ 存在如下关系:

$$\begin{aligned} v_\pi(s) &= E[G_t|S_t = s] \\ &= E[R_{t+1} + \gamma R_{t+2} + ...|S_t = s] \\ &= E[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3})|S_t = s] \\ &= E[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= E[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \end{aligned}$$

这个关系叫做贝尔曼方程, 同样可以得到状态-行为函数的贝尔曼方程:

$$q_\pi(s,a) = E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$$

由于状态值函数是以期望定义的, 根据期望的计算规则, 状态值函数应该是:

$$v_\pi(s) = \sum_{a\in A}\pi(a|s)q_\pi(s,a)$$

粗略的理解是在 t 时刻的状态 s, 以某以概率选择动作 a, 动作 a 产生的状态-动作值是 $q_\pi(s,a)$, 于是也就有上面的期望计算式. 再将 $q_\pi(s,a)$ 改写成:

$$q_\pi(s,a) = R_s^a + \gamma\sum_{\hat{s}\in S}P_{s\hat{s}}^a v_\pi(\hat{s})$$

所以:

$$v_\pi(s) = \sum_{a\in A}\pi(a|s)\big(R_s^a + \gamma\sum_{\hat{s}\in S}P_{s\hat{s}}^a v_\pi(\hat{s})\big)$$

而:

$$v_\pi(\hat{s}) = \sum_{\hat{a}\in A}\pi(\hat{a}|\hat{s})q_\pi(\hat{s},\hat{a})$$

定义最优的状态值函数 $v^*(s) = \max_\pi v_\pi(s)$, 也就是所有策略中最大的状态值函数; 同理最优的状态-动作函数 $q^*(s,a) = \max_\pi q(s,a)$, 所有策略中最大的状态-动作值函数值. 即:

$$v^*(s) = \max R_s^a + \gamma\sum_{\hat{s}\in S}P_{s\hat{s}}^a v_\pi^*(\hat{s})$$

$$q^*(s,a) = R_s^a + \gamma\sum_{\hat{s}\in S}P_{s\hat{s}}^a max_{\hat{a}}q_\pi^*(\hat{s},\hat{a})$$

3.3 Actor-Critic

3.3.1 背景

传统强化学习中除了 value-based 方法, 另一大类就是 policy-based 方法. 在 RL 任务中, 本质上最终要学习的是策略 (Policy) . 前者用的是间接方法, 即通过学习值函数 (value function) 或者动作-值函数 (action-value function) 来得到策略. 而后者是直接对策略进行建模和学习, 因此后者也称为 policy optimization.

Policy-based 方法又可分为两大类: gradient-based 方法和 gradient-free 方法. 前者也称为 policy gradient (PG) 方法. 而 policy gradient 方法又可细分为几类, 如 finite difference, Monte-Carlo 和 Actor-Critic 等. Actor-Critic (AC) 方法其实是 policy-based 和 value-based 方法的结合. 因为它本身是一种 PG 方法, 同时又结合了 value estimation 方法, 所以有些地方将之归为 PG 方法的一种, 有些地方把它列为 policy-based 和 value-based 以外的另一种方法.

在 AC 框架中, actor 负责 policy gradient 学习策略, 而 critic 负责 policy evaluation 估计 value function. 可以看到, 一方面 actor 学习策略, 而策略更新依赖 critic 估计的 value function; 另一方面 critic 估计 value function, 而 value function 又是策略的函数. Policy 和 value function 互为依赖, 相互影响, 因此需要在训练过程中迭代优化. 这种多元优化的迭代思想其实在机器学习中有很多体现.

Actor-Critic 的发展历程如下:

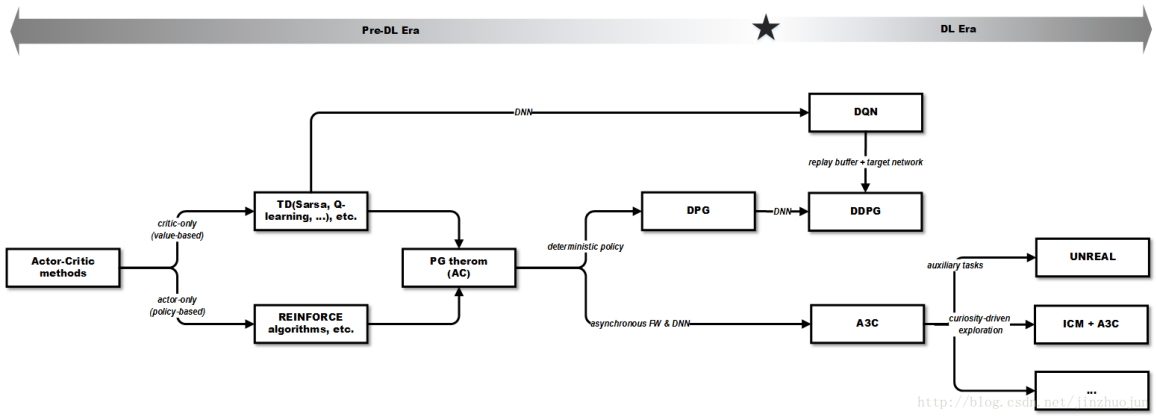

图 **2** **Actor-Critic** 发展历程

3.3.2 理论部分

PG 算法的基本做法是先定义 policy 为参数 θ 的函数, 记作 $\pi(\theta)$, 然后定义 $J(\theta)$ 为目标函数 (Objective function, or performance objective) , 接着通过利用目标函数相对于参数的梯度更新参数 θ, 从而达到目标函数的局部最大值. 记作:

$$\Delta\theta = \alpha\nabla_\theta J(\theta)$$

在 Sutton 的奠基性论文《Policy Gradient Methods for Reinforcement Learning with Function Approximation》中证明了结合可微 FA 的 PG 方法可收敛到局部最优点. 论文中提出了两种 objective function 定义, 但结论是一致的. 以 average reward formulation 为例, 它把 J 定义为平均回报 (average reward) , 对其求导, 得到了 (Stochastic) Policy Graidient 定理. 其核心结论给出了目标函数相对于策略参数的梯度公式:

$$\frac{\partial J(\theta)}{\partial \theta}=\sum_{s} d^{\pi}(s) \sum_{a} \frac{\partial \pi(s, a)}{\partial \theta} Q^{\pi}(s, a)$$

重点是它其中不含有状态稳态分布 $d^{\pi}(s)$ 对于策略参数的导数, 这意味着参数的变化对于状态稳态分布没有影响, 这样有利于通过采样来逼近和估计梯度. 因为这样的话通过执行策略 π 来得到状态的分布就可以得到无偏估计. 注意如果其中的 Q 函数用真实的累积回报来估计, 那就是 REINFORCE 算法. 但结合了 FA, 即 $Q_w(s,a)$ 后, 会引入 bias. 我们可以采用类似蒙特卡洛的方法采样一条轨迹后对策略进行更新, 即:

$$\nabla_{\theta} \mathrm{J}(\theta)=\frac{1}{N} \sum_{i=0}^{N} \sum_{t=0}^{T}\left[\nabla_{\theta} \log \pi_{\theta}\left(a_{i_1 t} | s_{i_1 t}\right)\left(\sum_{t^{\prime}=t}^{T} r\left(s_{i_1 t}, a_{i_1 t}\right)-b\right)\right] \tag{1}$$

机器学习的一个核心问题就是平衡偏差和方差 [5]. 对于策略梯度来说, 它的方差相差较大, 为了模型的稳定, 我们可以牺牲一定的偏差来换取方差变小. 这其中的一种方法就是 Actor-Critic 算法, 其主要特点是通过一个单独的模型来估计轨迹的累积回报, 而不再用采样完一条轨迹后的真实累积回报.

公式 (1) 为随机策略梯度模型的梯度 (优化) 方向, 我们可以使用如下的方法来替代轨迹梯度权值 $\sum_{t^{\prime}=t}^{T} r\left(s_{i_1 t}, a_{i_1 t}\right)-b$

1 直接使用状态值函数估计的轨迹回报作为轨迹梯度权值: $q(s,a)$

2 直接使用优势函数作为轨迹的梯度权重: $A(s,a)=q(s,a)-V(s)$

3 使用 TD-error 作为轨迹梯度权值: $r(s_t,a_t)+v(s_{t+1})-v(s_t)$

上述三种替代方式都可以在一定程度上减低方差, 但是偏差会增大. 更进一步, 还可以利用 $TD(\lambda)$ 的方式来平衡偏差与方差的关系. Actor-Critic 选用第三种 TD-error 的方式作为轨迹梯度的权值, 这种方法在计算量以及减小方差上都有较好的效果. 为了估计 $v(s)$ 引入价值模型, 这也是 Actor-Critic 命名的由来, Actor 表示策略模型, Critic 表示价值模型.

Actor-Critic 可以每运行一步进行一次学习, 其核心公式为

$$\begin{aligned}
&\delta=r_t\left(s_t, \mathrm{a}_t\right)+\nu_{\omega}\left(\mathrm{s}_{t+1}\right)-\nu_{\omega}\left(s_t\right)\\
&\omega_{t+1}=\omega_t+\alpha^{\omega} \delta_t \nabla_{\omega} \nu_{\omega}\left(s_t\right)\\
&\theta_{t+1}=\theta_t+\alpha^{\theta} \nabla_{\theta} \log \pi_{\theta}\left(a_t | s_t\right) \delta_t
\end{aligned}$$

即: 1. 计算该步的 TD-error; 2. 更新价值模型参数; 3. 更新策略模型参数.

3.4 Asynchronous Advantage Actor-Critic

A3C(Asynchronous Advantage AC) 采用异步更新的方式, 加快了模型的学习速度. 它不仅适用于离散也适用于连续动作空间的控制问题. AC 及 policy gradient 是一个 on-policy 的策略, 每次模型与环境交互后更新模型, 再用新的模型去与环境交互进行下一次更新. A3C 是一种并行采样交互与训练的算法, 归类为 off-policy.

A3C 在训练过程中同时启动 N 个线程, 使用 N 个 agent 同时与环境进行交互, 只要对环境设置不同, 则每个线程采样得到的数据将会有区别. 每个线程按照 Actor-Critic 算法的方式独立的进行环境交互与学习, 每个线程完成训练并得到参数更新量后, 异步的将该线程的参数更新量更新到全局模型中, 下一次训练开始前, 从全局模型中获取最新的模型参数, 使用新同步到的全局模型参数进行下一次交互更新.

在 Actor-Critic 算法中, 使用了 TD-error 的形式: $r(s_t, a_t) + v(s_{t+1}) - v(s_t)$ 来估计回报, 而 A3C 算法则使用了多步回报估计法, 这个方法可以在模型训练早期更快得提升价值模型, 对应的公式为:

$$A(s_t, a_t; \theta', \theta'_v) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta'_v) - V(s_t; \theta'_v)$$

其中: θ' 是策略 π 的参数, θ'_v 是状态值函数的参数. k 是可变化的, 上界由 n-step 决定, 即 n. 在实际操作中, A3C 在该公式中加入了策略 π 的熵项 $\beta \nabla_{\theta'} H(\pi(s_t; \theta'))$, 防止过早的进入次优策略.

最终, 完整的策略梯度计算公式已经更新为:

$$\nabla_\theta J(\theta) = \frac{1}{T} \sum_t^T \nabla_\theta \log \pi(a_t | s_t; \theta) \left(\sum_{i=1}^n \gamma^{i-1} r_{t+1} + \nu(s_{t+n}) - \nu(s_t) \right) + \beta \nabla_\theta H(\pi(s_t, \theta))$$

其中, 其中 H 是信息熵, 超参数 β 为策略的熵在目标中的权重.

四、 模型的建立与求解

4.1 模型的输入输出

本实验使用了 openAI 的强化学习平台 gym, gym 提供了很多内置的环境. 我们使用了其中的'Pong-v0' 模块来实现游戏功能部分.

当用于强化学习的智能体 (Agent) 进行决策时, 它需要获取游戏的当前状态 (S), 即: 我们需要为智能体准备用于决策的输入数据. 通过截图可以获取每时每刻的游戏画面 (如图 3 所示), 并将其灰度化、标准化、然后分别裁切为 84*84 像素的图像, 最后将图像中的每四帧作为一个单独的状态, 即完成了模型输入数据的准备.

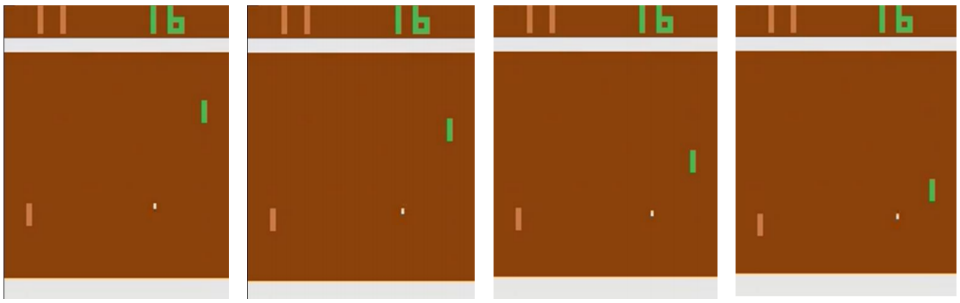

图 **3**　连续四帧游戏画面

在本实验中, Agent 面临一个游戏状态时决策的结果只有三种类型: 向上移动、向下移动、保持不动; Agent 的 Actor 结构需要输出决策的结果, Critic 结构需要输出决策结果的好坏. 所以模型需要输出为一个长度为 3 的向量 (动作空间: A) 和一个对动作的评分.

4.2 Actor-Critic Net 架构

本次实验的模型主要使用一个两层的卷积网络来提取游戏进行时的画面特征, 并设计了一个 LSTM 网络层来对卷积层提取的特征进行进一步处理, 其中 CNN 层和 RNN 层使用了一个全连接层进行过渡, 最后分别使用一层全连接层来输出动作和值函数.

Actor-Critic 的架构如下:

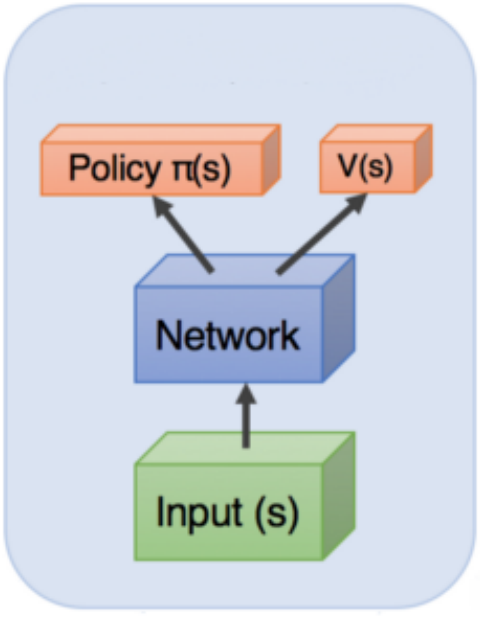

图 **4**　**Actor-Critic**

其中 Network 的输出 256 维的向量, Actor 部分是一个 256×3 的全连接层, 用于输出 Policy $\pi(s)$; Critic 部分是一个 256×1 的全连接层, 用于输出 $V(s)$. Network 的网络结构为:

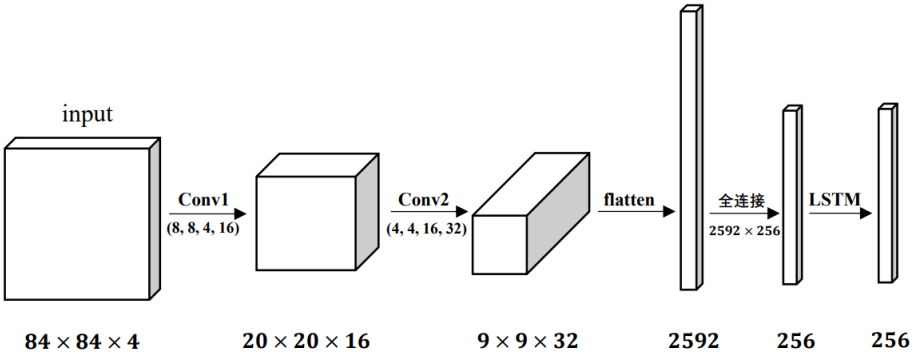

图 **5**　**Network** 的网络结构

Network 的卷积层中 conv1 的 stride 为 4, conv2 的 stride 为 2. Network 的最后一层使用了 LSTM 结构对卷积层提取的特征进行进一步处理, 这里也可以使用全连接层, 但是 LSTM 通常要比全连接层有更好的效果 [2], 这是因为 LSTM 能更好地捕捉帧与帧之间的关系 (比如游戏中小球的速度, 方向等特征). 网络的其他具体细节可在附件 logs 中使用 Tensorboard 查看.

4.3 网络的训练

在训练过程中我们定义了一个全局网络 (Global Net) 和八个局部网络 (Local Net, 即独立训练的 Agent), 每个局部网络都能从全局网络获取参数用于下一次训练, 而且也都能将训练结果用于全局网络的参数更新上, 各部分的关系如下图所示:

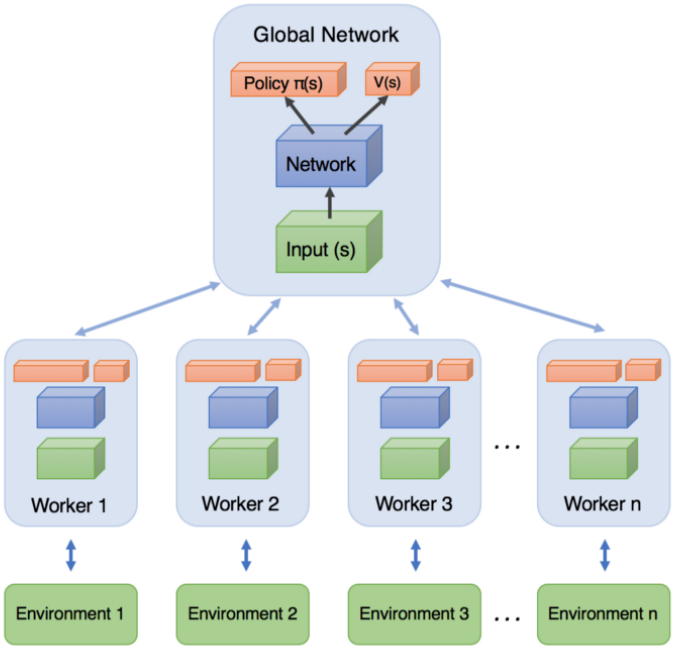

图 **6**　各网络训练间的关系

4.4 超参数选择

本次实验使用了 RMSProp [4] 方法进行优化, 虽然 RMSProp 已被广泛用于深度学习, 但它尚未在异步优化设置中进行广泛研究. 论文 [2] 中的研究表明 RMSProp 通常具有更好的效果, 作者在 *Optimization Details* 章节给出了详细的理由, 这里不再赘述, 本实验中 RMSProp 的动力项系数和全局初始学习率分别为 0.99 和 0.1.

在训练过程中, 每当 Agent 执行若干步 (local AC 网络进行采样的最大的时间步:LMS) 以后就要对网络进行优化, 并将权重更新同步到全局网络中. 几个周期之后, 全局网络将获得所有局部网络的累积梯度.

主要的超参数选择如下表:

表 **1**　超参数选择

STEPS	LMS	Lr	n_{AGENT}	γ	β
76M	20	[1e-4,1e-2]	8	0.99	0.01

训练过程一共进行了 76M 步, local AC 网络进行采样的最大的时间步为 20, 初始学习率为 0.01 并不断衰减, 最小值为 10^{-4}, 实验一共训练了 8 个 Agent, 奖励折扣为 0.99, 策略的熵在目标函数中的权重为 0.01.

4.5 实验结果

经过 23h(76M 步) 的训练, Agent 的得分变化如下:

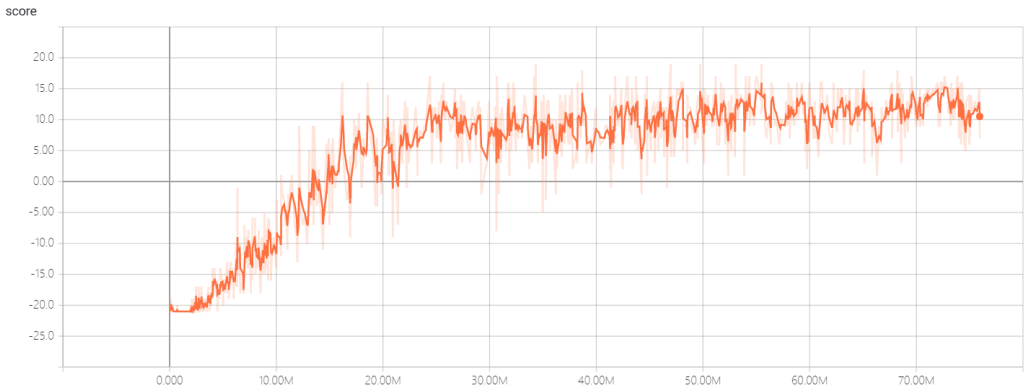

图 **7**　得分变化图

可以看到, 随着训练的进行得分不断提高. 限于时间和硬件限制, 本实验没有对参数选择进行细致的研究, 实验中的参数参考了论文 [6] 和 [2] 并作了适当调整. 单就结果来看, 参数的选择确实是有效的, 附件内的视频记录了 Agent 训练后在游戏中的表现.

五、 总结

这次实验我们主要针对'Pong' 游戏建立了一个 Actor-Critic 网络, 并使用 A3C 算法进行优化. 实验期间遇到过许多困难, 但是小组成员互相帮助问题最终都一一解决. 但时间紧迫, 本实验尚有可完善的地方. 这次实验给予了我们知识和经验, 提高了我们的解决问题的能力, 在日后的的学习过程中, 这次大作业的实验经历将是一份宝贵的财富.

参考文献

[1] Hausknecht, M. and P. Stone 2015. Deep Recurrent Q-Learning for Partially Observable MDPs. arXiv:1507.06527.

[2] Mnih, V., A. Puigdomènech Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver and K. Kavukcuoglu 2016. Asynchronous Methods for Deep Reinforcement Learning. arXiv:1602.01783.

[3] Thomas, P. S. and E. Brunskill 2017. Policy Gradient Methods for Reinforcement Learning with Function Approximation and Action-Dependent Baselines. arXiv:1706.06643.

[4] Tieleman, Tijmen and Hinton, Geoffrey. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 4, 2012.

[5] Arulkumaran, K., M. P. Deisenroth, M. Brundage and A. A. Bharath 2017. A Brief Survey of Deep Reinforcement Learning. arXiv:1708.05866.

[6] Lowe, R., Y. Wu, A. Tamar, J. Harb, P. Abbeel and I. Mordatch 2017. Multi-Agent Actor-Critic for Mixed Cooperative- Competitive Environments. arXiv:1706.02275.