强化学习第五次作业 - 阅读 AlphaGo 文献报告并设计相关实验

吴天阳 强基数学 002 2204210460

1 AlphaGo 基本原理

AlphaGo 文献参考 Deepmind 官网上的Mastering the game of Go with deep neural networks and tree search文章，该文章说明了 AlphaGo 是通过 **深度神经网络和蒙特卡洛树搜索**两种方法的结合来学习围棋的，深度神经网络主要包括两个部分：**策略网络和（状态）价值神经网络**.

使用蒙特卡洛树搜索作为 AI 围棋算法在 AlphaGo 之前就有很多程序实现了，例如文章中与 AlphaGo 进行对比的 Fan Hui, Crazy Stone, Zen 等等，AlphaGo 只是利用深度神经网络帮助蒙特卡洛树搜索进行后续步骤评估，并用蒙特卡洛树搜索作为强化学习中生成一幕数据的方法，并自己和自己下棋得到更多的训练数据，从而不断改进深度神经网络，最终以 5:0 战胜人类顶级围棋选手.

这种无模型的强化学习方法的特点在于没有任何的先验围棋技巧知识，完全通过已有的棋局和自我下棋的强化学习方法对网络进行更新，无需人工设计复杂的奖励函数和具体的策略.

文章将 AlphaGo 的训练分为以下三个主要部分：

1. 基于有监督数据的策略网络学习；
2. 基于强化学习的策略网络学习；
3. 基于强化学习的价值网络；

神经网络使用的是卷积神经网络，并使用蒙特卡洛树搜索对策略进行确定.

1.1 基于有监督数据的策略网络学习

设 $s \in \mathbb{R}^{19\times 19}$ 表示棋盘的状态，a 表示落子的位置，$p(a|s)$ 表示棋盘为 s 时，落子在 a 处的概率. 通过这篇文献可以总结出以下几点：

- 网络结构：$p_\sigma(a|s)$ 表示权重参数为 σ 的 13 层的卷积神经网络（激活函数为 ReLU 函数），输入层为 s，输出通过 softmax 得到 a 的概率分布；
- 训练数据：KGS Go Server 数据集中包含 3 千万个训练样本 $\{(s_i, a_i)\}$；
- 网络训练方法：随机梯度下降最大化对数似然 $\max\limits_{\sigma} \log p_\sigma(a|s)$，即损失函数为交叉熵损失函数.

同时还他们训练了一个简单版本的小型神经网络 $p_\pi(a|s)$，网络结构与 $p_\sigma(a|s)$ 类似，只是层数更少，且该网络运算速度更快，该网络用于蒙特卡洛树搜索中进行决策评估时所使用的.

1.2 基于强化学习的策略网络学习

基于强化学习的策略网络记为 $p_\rho(a|s)$，网络结构与 $p_\sigma(a|s)$ 完全一致，并且初始权重参数来自训练完成的 $p_\sigma(a|s)$. 强化学习的训练关键为以下几部分：

训练数据 由于是分幕式任务，所以将自己和自己下棋的一幕完整的棋局作为一个训练样本（由蒙特卡洛树搜索得到，为避免过拟合，会随机从之前训练好的网络中选择策略）将每一次落子定义为一个时刻的划分，设一个棋局总共有 T 个时刻，将一幕中下棋的某一方数据记为 $s_0, a_0, r_0, \cdots, s_T, a_T, r_T$.

奖励函数 将所有动作到达的非终止状态的奖励均记为 0，到达终止状态并胜利的奖励记为 1，到达终止状态并失败的奖励记为 -1，即

$$r_t = 0,\ (t = 0, 1, \cdots, T-1), \quad r_T = \begin{cases} +1, & \text{胜利}, \\ -1, & \text{失败}. \end{cases}$$

上述奖励的定义满足围棋的策略，因为围棋中最终的输赢往往仅取决于某一步的结果，所以**网络的更新是基于 (s_t, a_t) 进行的，而不是一整幕数据进行更新**.

用强化学习理论进一步定义回报值 $z_t = r_T$，也就是在 AlphaGo 程序中回报折扣率 $\gamma = 1$，奖励不随当前状态距离最终状态的相对时间长短而发生改变.

网络训练 如果当前幕最终状态为胜利，则最大化似然函数，否则最小化似然函数，也就是对于每个每一幕中某个训练样本 (s_t, a_t)，网络参数 ρ 梯度下降方向为

$$\Delta\rho \propto \frac{\partial p_\rho(a_t|s_t)}{\partial \rho} z_t$$

同样可以用交叉熵作为损失函数，输入特征为 s_t，真实值为 a_t 对应的 one-hot 向量，如果 $z_t = +1$，则向 $-\Delta\rho$ 方向更新参数，否则向 $\Delta\rho$ 方向更新参数.

1.3 基于强化学习的价值网络

设策略 p 对应的真实状态价值网络为 $v^p(s) : \mathbb{R}^{19\times 19} \to \mathbb{R}$，其定义与强化学习理论中的状态价值相同

$$v^p(s) = \mathbb{E}[z_t | s_t = s, a_{t\cdots T} \sim p] = \mathbb{E}_p[Z_t | S_t = s]$$

使用神经网络 $v_\theta(s)$ 去近似 $v^p(s)$，再通过强化学习理论中经典的广义策略迭代可以说明，通过不断的迭代更新最终有 $v_\theta \approx v^{p_\rho} \approx v^*$，其中 $v^*(s)$ 状态 s 下的理论最优真实价值. 价值网络的学习中包含以下关键点：

- 使用与 $p_\rho(a|s)$ 相似的神经网络结构，只是将输出层的 softmax 换为输出层为 1 的全连接层，从而将网络从预测概率分布的模型转化为回归模型；

- 训练数据：对于一幕的输赢可以得到 z，对于第 t 时刻下训练数据为 (s_t, z)，文献中给出的梯度为

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}(z - v_\theta(s))$$

也就是用均方误差 $[z - v_\theta(s)]^2$ 作为损失函数.

1.4 基于策略函数和价值函数的蒙特卡洛树搜索

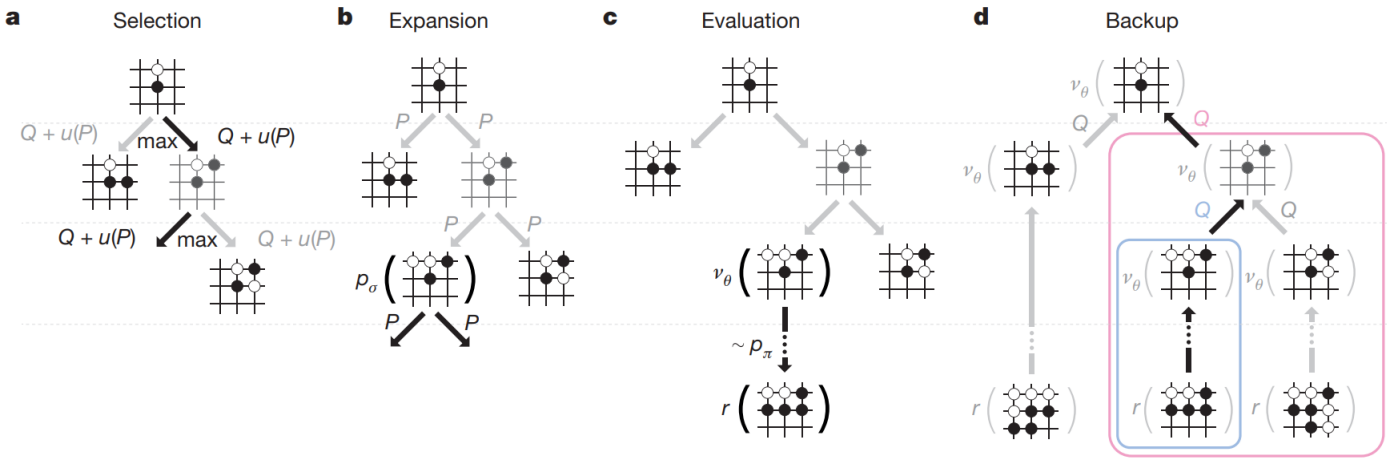

图 1: 蒙特卡洛树搜索

AlphaGo 的强化学习中动作状态价值函数 Q 是通过蒙特卡洛树搜索确定的，蒙特卡洛树搜索算法如图 1 所示，搜索树的跟节点状态表示当前的状态 s，由于搜索状态非常庞大，所以利用贪心的思想，每次优先搜索最有可能获得最高价值的路径，并将终止状态作为搜索树的叶子节点，每次搜索完成后回溯更新动作价值函数 $Q(s,a)$；但是这样贪心地搜索缺少了试探的过程，容易陷入局部最优，所以搜索下个动作的概率还应该和访问次数成反比，具体实现见下面的详细过程解释.

设 $P(s,a) = p_\rho(a|s)$，$N(s,a)$ 为动作-状态 (s,a) 的访问次数，最开始只有跟节点（当前状态）被访问到，搜索树上仅有该一个节点，假设现在是第 n 次搜索，每次搜索均从根节点开始，蒙特卡洛搜索分为一下四步：

1. 动作选择：根据动作价值函数 $Q(s,a)$ 和一个奖励值 $u(s,a)$ 确定选择的动作

$$a_t = \arg\max_a (Q(s_t,a) + u(s_t,a))$$

其中 $u(s,a) \propto \dfrac{P(s,a)}{1+N(s,a)}$，分子 $P(s,a)$ 就是策略网络 $p_\sigma(s,a)$ 或 $p_\rho(s,a)$，分母为该条边的访问次数，加入该项可以是的访问次数少的节点有更多的访问几率，从而模型的增强试探能力.

2. 状态扩展：当搜索到达一个从未见过的状态节点 s_L 时（也称 s_L 为叶子节点），则需对 s_L 进行扩展，利用策略网络 $p_\sigma(s,a)$ 或 $p_\rho(s,a)$ 对 s_L 的所有合法状态 a 进行预测，得到 $P(s_L,a)$.

3. 状态评估：记 s_L 处的状态价值函数为 $V(s_L)$，第一部分肯定是使用状态价值函数 $v_\theta(s_L)$ 对其进行估计，第二部分则是利用最开始通过监督学习得到的简单决策网络 $p_\pi(s,a)$ 做搜索得到的链，这条链一直搜索到终止状态，记最终的奖励值为 z_L；结合上述二者确定 s_L 处的状态价值函数（λ 为混合因子，用于平衡两者的重要度）

$$V(s_L) = (1-\lambda)v_\theta(s_L) + \lambda z_L$$

4. 状态回溯：在得到 s_L 的状态后，通过回溯更新 $N(s,a)$ 和 $Q(s,a)$，$N(s,a)$ 就是边 (s,a) 通过的累计次数，$Q(s,a)$ 则可通过强化学习理论的定义式得到

$$\begin{aligned} Q(s,a) &= \mathbb{E}[z_t|S_t = s, A_t = a] = \sum_{s_L,r} p(s_L|s,a)(r + V(s_L)) \\ &\xlongequal{r=0} \sum_{s_L} p(s_L|s,a)V(s_L) = \frac{1}{N(s,a)}\sum_{i=1}^{n} 1(s,a,i)V(s_L^i) \end{aligned}$$

其中 $1(s,a,i)$ 表示第 i 次搜索是否经过 (s,a)，$V(s_L^i)$ 表示第 i 次搜索的叶节点.

最后，**选择访问次数最多的动作作为当前状态的实际动作**.

1.5 总结

最终版本的 AlphaGo 分为两个版本，一个单个机器上计算 40 线程，48 个 CPU 和 8 个 GPU 的小型版本；另一个是分布式版本 40 线程，1202 个 CPU 和 176 个 GPU 的大型版本. 总的来说，AlphaGo 做出了一下几个特点：

1. 将蒙特卡洛树搜索与深度神经网络相结合，利用神经网络近似策略函数和状态价值函数，用于搜索过程中状态价值的再估计与搜索方向；
2. 深度神经网络将有监督学习和强化学习相结合，先进行有监督学习，再在其基础上进行强化学习；
3. 仅从与环境交互中学习，无需人工加入的特殊估值函数.

2 实验设计

2.1 实验目标

使用论文Playing Atari with Deep Reinforcement Learning - 2013中的 DQN 网络，对 Cartpole 平衡小车问题进行求解，模型来自 OpenAI 在 Python 中的gymnasium 库，我通过单线程与多线程实现的方法，有效地实现了简单的 DQN 网络解决该问题.

Cartpole 问题可以视为一个倒立摆问题，倒立摆是一个重心高于其枢轴点的摆，它是不稳定的，但是可以通过移动枢轴点位置以保持该系统的稳定性. 我们的目标是尽可能长地保持摆的垂直状态.

- 紫色方块代表枢纽点.
- 红色和绿色箭头分别表示枢纽点可移动的水平方向.

具体的规则：一个摆通过一个无摩擦的枢轴连接到推车上，该推车沿着水平方向的无摩擦轨道移动. 通过对推车施加 $+1$ 和 -1 的推力来维持该系统平衡，保持摆的直立状态. 当杆在一个时间戳保持直立，则获得 $+1$ 的奖励. 当杆与竖直方向的夹角超过 $15°$，或者小车相对中心的移动距离超过 2.4 个单位时，游戏结束.

2.2 算法原理

Deep Q-Learning Network (DQN) 由 DeepMind 团队在 2013 年提出，论文中在多个 Atari 游戏中超过了人类水平，核心更新公式如下

$$Q(s_t, a_t; \boldsymbol{w}) \leftarrow r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \boldsymbol{w})$$

其中 $Q(s, a; \boldsymbol{w})$ 表示网络参数为 $\boldsymbol{w}$，输入为 s, a 的神经网络，输出为最优动作价值函数的估计，$\mathcal{A}$ 表示全部的动作集合.

在 2015 年在 Nature 上提出的论文Human-level control through deep reinforcement learning - 2015 中，更新公式不变，提出要在两个网络 $Q(s, a; \boldsymbol{w}')$ 和 $Q(s, a; \boldsymbol{w})$，其中 $\boldsymbol{w}$ 为当前更新的网络，$\boldsymbol{w}'$ 表示之前某次的网络，论文中说可以保持策略的稳定性，从而更容易收敛

$$Q(s_t, a_t; \boldsymbol{w}) \leftarrow r_t + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a'; \boldsymbol{w}')$$

我是按照第一篇论文实现的网络，由于问题较为简单，所以约 20 分钟之内能够达到最优解.

2.3 结果分析

下图为单线程的某次拟合效果：GIF 动图

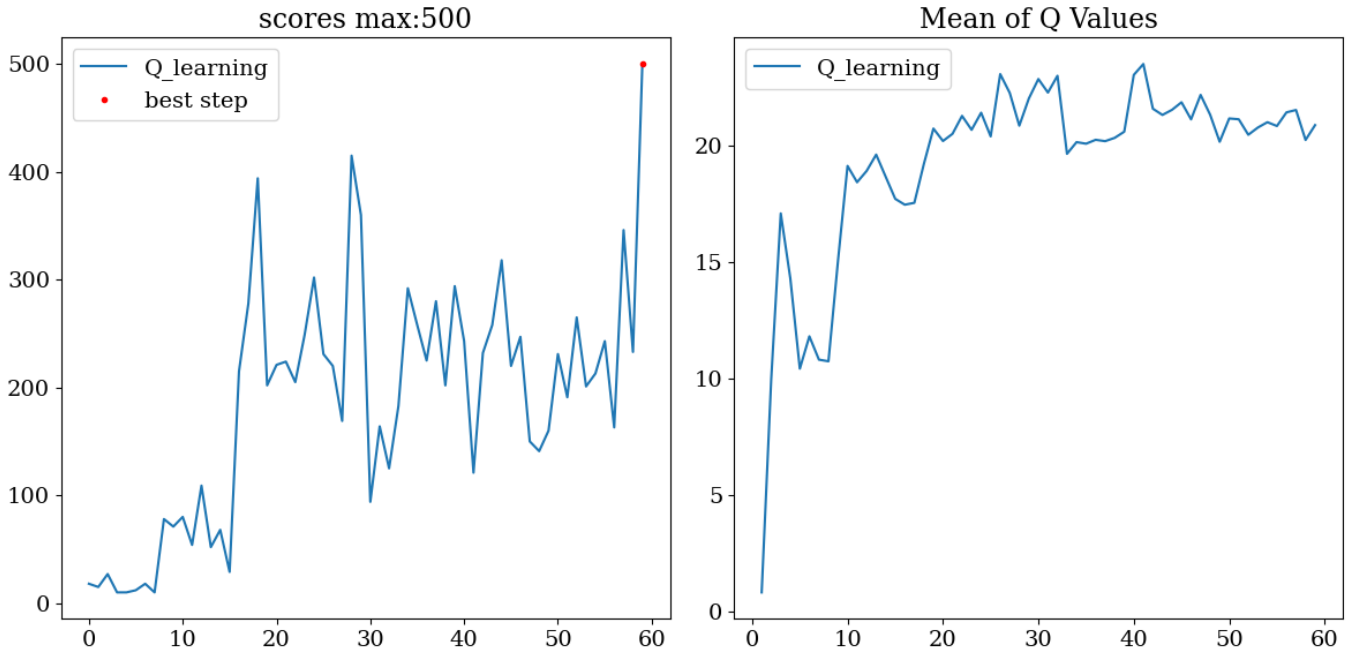

图 2: 单线程拟合效果

左侧为每幕的最优得分（达到的步数），右图为每幕下对 Q 值的估计. 由于单线程训练样本非常少，每步之后只能对网络以大小为 32batch 对网络进行一次更新. 网络参数为：

```
    learning_rate = 0.001
    gamma = 0.95
    BATCH_SIZE = 32
    MEMORY_SIZE = 10000

    EPSILON_MAX = 1.0
    EPSILON_MIN = 0.01
    EPSILON_DECAY = 0.995

    Dense1: units=32, activation="relu", type="Dense"
    Dense2: units=32, activation="relu", type="Dense"
    Output:  units=2, type="Dense"

    LOSS = MSE (Mean square of error)
    optimizer = Adam
```

下图为 6 线程的某次拟合效果：GIF 动图

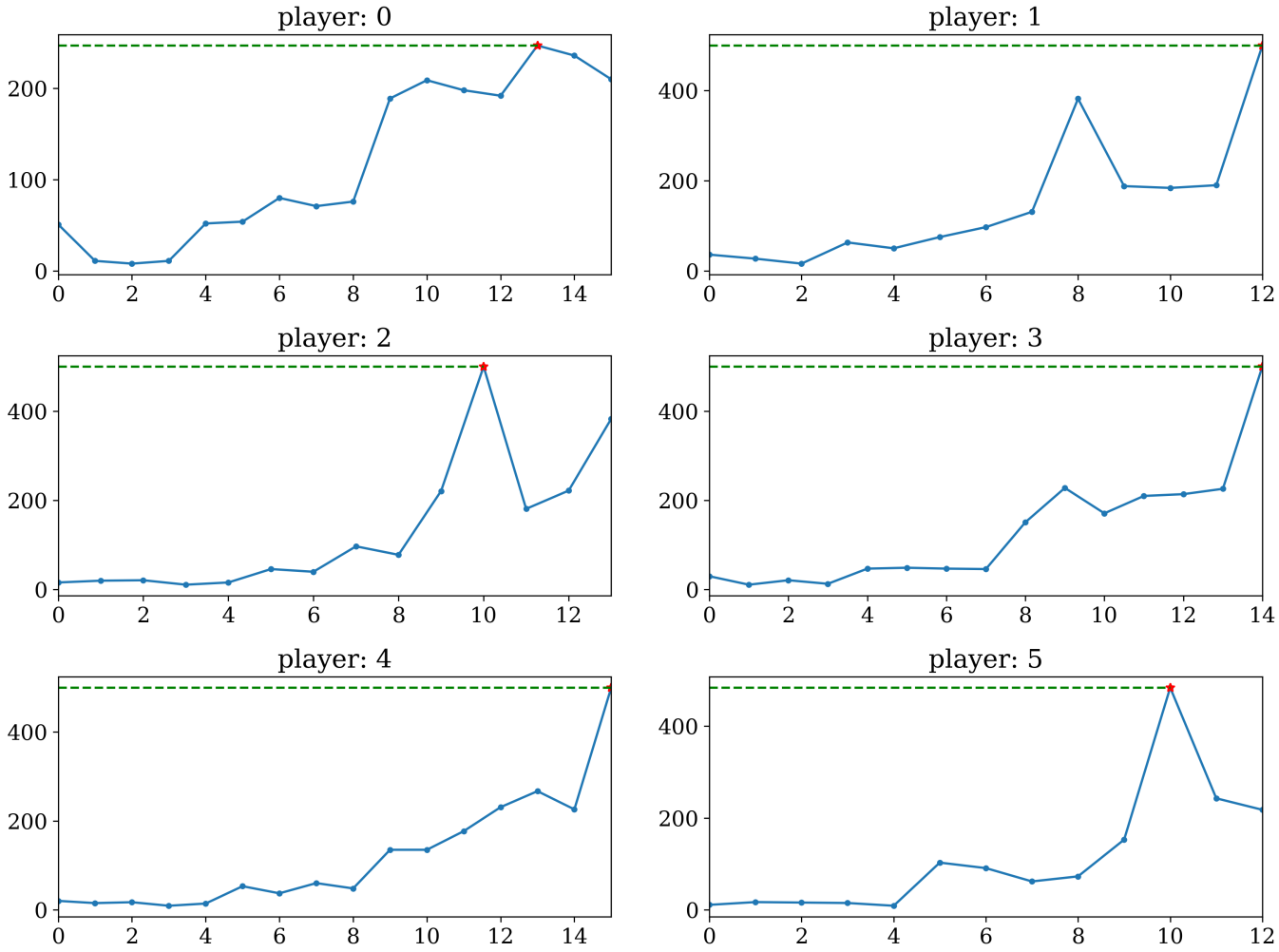

图 3: 6 个线程的拟合效果

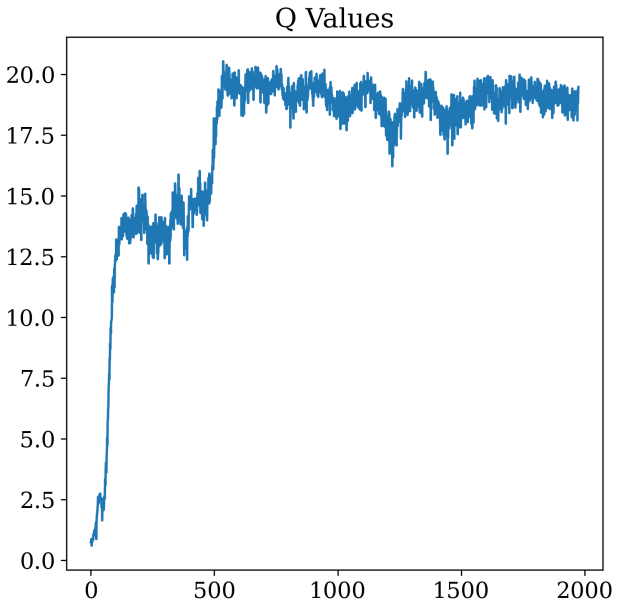

图 4: 6 线程的 Q 函数估计值

6 线程中 Q 值更新更加稳定，而且训练速度更快，并且最终得到的动图中滑块移动范围更小，更加稳定，超参数与单线程相同.

具体更新的方法为统一收到 6 个线程返回的状态值，通过返回值传入到网络中得到每个线程的 Q 值估计值并记录到记忆中，并给出最优动作，并对网络进行一次训练，每次采样的训练集大小为 6×32，每个 Batch 大小为 32.

2.4 代码结构

使用到的库版本如下：

```
tensorflow.__version__ = '2.12.0'  # tensorflow 2 以上均可
numpy.__version__ = '1.23.5'
matplotlib.__version__ = '3.7.1'
gymnasium.__version__ = '0.27.1'
```

单线程的代码在文件夹 `code_Catpole/single_process` 中，主要包含两个代码文件

- `main.py`：单线程运行的主程序，主要由 `Agent` 类构成，包含数据的输入、日志保存、环境交互的功能.
- `dense_net.py`：主要包含类 `DenseModel`，实现网络的记忆、训练与 Q 值预测.

多线程的代码在文件夹 `code_Catpole/multi_process` 中，主要包含五个代码文件

- `run.py`：启动训练.
- `agent.py`：包含智能体 `Agent` 类，主要实现网络部分，具有记忆、训练与 Q 值预测.
- `player.py`：包含 `Player` 类，实现每个子线程的内部日志记录，独立环境创建和交互功能；`LogManager` 用于处理每个 `Player` 类的日志，并输出到图像中.

- `swapper.py`：作为 `Agent` 与 `Player` 类之间的数据进行交换，开启线程池，并进行多线程操作.
- `constant.py`：包含全部重要超参数.