第十二章 并行计算

机器学习的实践中普遍使用并行计算，利用大量的计算资源（比如很多块 GPU）缩短训练所需的时间，用几个小时就能完成原本需要很多天才能完成的训练。深度强化学习自然也不例外；可以用很多处理器同时收集经验、计算梯度，让原本需要很长时间的训练在较短的时间内完成。第 12.1 以并行梯度下降为例讲解并行计算基础知识。第 12.2 介绍异步并行梯度下降算法。第 12.3 介绍两种异步强化学习算法。

12.1 并行计算基础

本节以并行梯度下降 (Parallel Gradient Descent) 为例讲解并行计算的基础知识，用 MapReduce 架构实现并行梯度下降，并且分析并行计算中的时间开销。

12.1.1 并行梯度下降

本节用最小二乘回归 (Least Squares Regression) 为例讲解并行梯度下降的基本原理。把训练数据记作 $(\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$。最小二乘回归定义为：

$$\min_{\boldsymbol{w}} \ \left\{ L(\boldsymbol{w}) \ \triangleq \ \frac{1}{2n} \sum_{j=1}^{n} \left(\boldsymbol{x}_j^T \boldsymbol{w} \ - \ y_j \right)^2 \right\}.$$

这个优化问题的目标是寻找向量 $\boldsymbol{w}^\star \in \mathbb{R}^d$，使得对于所有的 j，$\boldsymbol{x}_j^T \boldsymbol{w}^\star$ 都很接近 y_j。我们可以用梯度下降算法求解这个优化问题。梯度下降重复这个步骤，直到收敛：

$$\boldsymbol{w} \ \leftarrow \ \boldsymbol{w} \ - \ \eta \cdot \nabla_{\boldsymbol{w}} L(\boldsymbol{w}).$$

公式中的 η 是学习率。如果 η 的取值比较合理，那么梯度下降可以保证 $\boldsymbol{w}$ 收敛到最优解 $\boldsymbol{w}^\star$。目标函数 $L(\boldsymbol{w})$ 的梯度可以写作：

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) \ = \ \frac{1}{n} \sum_{j=1}^{n} \boldsymbol{g}(\boldsymbol{x}_j, y_j; \boldsymbol{w}), \qquad \text{其中} \quad \boldsymbol{g}(\boldsymbol{x}_j, y_j; \boldsymbol{w}) \ \triangleq \ \left(\boldsymbol{x}_j^T \boldsymbol{w} \ - \ y_j \right) \boldsymbol{x}_j \ \in \ \mathbb{R}^d.$$

由于 $\boldsymbol{x}_j$ 和 $\boldsymbol{w}$ 都是 d 维向量，因此计算一个 $\boldsymbol{g}(\boldsymbol{x}_j, y_j; \boldsymbol{w})$ 的时间复杂度是 $\mathcal{O}(d)$。计算梯度 $\nabla_{\boldsymbol{w}} L(\boldsymbol{w})$ 需要计算 $\boldsymbol{g}$ 函数 n 次，所以计算 $\nabla_{\boldsymbol{w}} L(\boldsymbol{w})$ 的时间复杂度是 $\mathcal{O}(nd)$。如果用 m 块处理器做并行计算，那么理想情况下每块处理器的计算量是 $\mathcal{O}(\frac{nd}{m})$。

下面举一个简单的例子讲解并行梯度下降。假设我们有两块处理器。把梯度 $\nabla_{\boldsymbol{w}} L(\boldsymbol{w})$ 展开，得到：

$$\begin{aligned} & \nabla_{\boldsymbol{w}} L(\boldsymbol{w}) \\ & = \ \frac{1}{n} \Big[\underbrace{\boldsymbol{g}(\boldsymbol{x}_1, y_1; \boldsymbol{w}) + \cdots + \boldsymbol{g}(\boldsymbol{x}_{\frac{n}{2}}, y_{\frac{n}{2}}; \boldsymbol{w})}_{\text{用一号处理器计算，把结果记作 } \tilde{\boldsymbol{g}}^1} + \underbrace{\boldsymbol{g}(\boldsymbol{x}_{\frac{n}{2}+1}, y_{\frac{n}{2}+1}; \boldsymbol{w}) + \cdots + \boldsymbol{g}(\boldsymbol{x}_n, y_n; \boldsymbol{w})}_{\text{用二号处理器计算，把结果记作 } \tilde{\boldsymbol{g}}^2} \Big]. \end{aligned}$$

两块处理器各承担一半的计算量，分别输出 d 维向量 $\tilde{\boldsymbol{g}}^1$ 和 $\tilde{\boldsymbol{g}}^2$。将两块处理器的结果汇总，得到梯度：

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) \ = \ \frac{1}{n} \left(\tilde{\boldsymbol{g}}^1 \ + \ \tilde{\boldsymbol{g}}^2 \right).$$

并行梯度下降中的“计算”非常简单；而并行计算的复杂之处在于通信。在一轮梯度下降开始之前，需要把最新的模型参数 $\boldsymbol{w}$ 发送给两块处理器，否则处理器无法计算梯度。在两块处理器完成计算之后，需要做通信，把结果 $\tilde{\boldsymbol{g}}^1$ 和 $\tilde{\boldsymbol{g}}^2$ 汇总到一块处理器上。下一小节以 MapReduce 架构为例，讲解并行梯度下降的实现。

12.1.2 MapReduce

并行计算需要在计算机集群上完成。一个集群有很多处理器和内存条，它们被划分到多个节点 (Compute Node) 上。一个节点上可以有多个处理器，处理器可以共享内存。节点之间不能共享内存，即一个节点不能访问另一个节点的内存。如果两个节点相连接，它们可以通过计算机网络通信（比如 TCP/IP 协议）。

为了协调节点的计算和通信，需要有相应的软件系统。MapReduce 是由 Google 开发的一种软件系统，用于大规模的数据分析和机器学习。MapReduce 原本是软件系统的名字，但是后来人们把类似的系统架构都称作 MapReduce。除了 Google 自己的 MapReduce，比较有名的系统还有 Hadoop[1] 和 Spark [2]。MapReduce 属于 Server-Client 架构，有一个节点作为中央服务器，其余节点作为 Worker，受服务器控制。服务器用于协调整个系统，而计算主要由 Worker 节点并行完成。

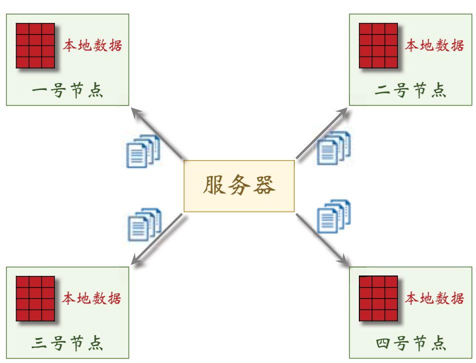

图 12.1: MapReduce 中的广播 (Broadcast) 操作。

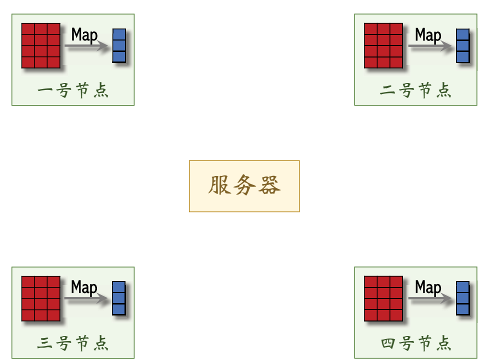

图 12.2: MapReduce 中的映射 (Map) 操作。

服务器可以与 Worker 节点做通信传输数据（但是 Worker 节点之间不能相互通信）。一种通信方式是**广播** (Broadcast)，即服务器将同一条信息同时发送给所有 Worker 节点；如图 12.1 所示。比如做并行梯度下降的时候，服务器需要把更新过的参数 $\boldsymbol{w} \in \mathbb{R}^d$ 广播到所有 Worker 节点。MapReduce 架构不允许服务器将一条信息只发送给一号节点，而将一条不同的信息只发送给二号节点。服务器只能把相同信息广播到所有节点。

每个节点都可以做计算。**映射** (Map) 操作让所有 Worker 节点同时并行做计算；如图 12.2 所示。如果我们要编程实现一个算法，需要自己定义一个函数，它可以让每个 Worker 节点把它的本地数据映射到一些输出值。比如做并行梯度下降的时候，定义函数 $\boldsymbol{g}$ 把三

[1]https://hadoop.apache.org/

[2]https://spark.apache.org/

元组 $(\boldsymbol{x}_j, y_j, \boldsymbol{w})$ 映射到向量

$$\boldsymbol{z}_j \;=\; \left(\boldsymbol{x}_j^T \boldsymbol{w} \;-\; y_j\right) \boldsymbol{x}_j.$$

映射操作要求所有节点都要同时执行同一个函数，比如 $\boldsymbol{g}(\boldsymbol{x}_j, y_j, \boldsymbol{w})$。节点不能各自执行不同的函数。

Worker 节点可以向服务器发送信息，最常用的通信操作是**规约** (Reduce)。这种操作可以把 Worker 节点上的数据做归并，并且传输到服务器上。如图 12.3 所示，系统对 Worker 节点输出的蓝色向量做规约。如果执行 `sum` 规约函数，那么结果是四个蓝色向量的加和。如果执行 `mean` 规约函数，那么结果是四个蓝色向量的均值。如果执行 `count` 规约函数，那么结果是整数 4，即蓝色向量的数量。

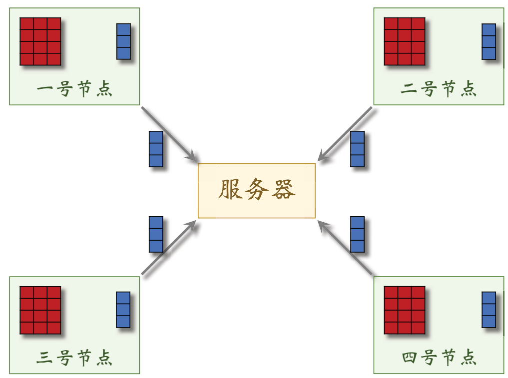

图 12.3: MapReduce 中的规约 (Reduce) 操作。

12.1.3 用 MapReduce 实现并行梯度下降

数据并发 (Data Parallelism)：为了使用 MapReduce 实现并行梯度下降，我们需要把数据集 $(\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_n, y_n)$ 划分到 m 个 Worker 节点上，每个节点上存一部分数据；见图 12.4。这种划分方式叫做数据并发。与数据并发相对的是模型并发 (Model Parallelism)，即将模型参数 $\boldsymbol{w}$ 划分到 m 个 Worker 节点上；每个节点有全部数据，但是只有一部分模型参数。本书只介绍数据并发，不讨论模型并发。

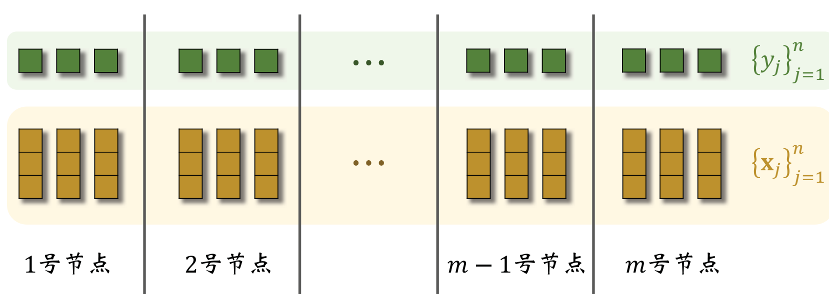

图 12.4: 将数据集 $(\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_n, y_n)$ 划分到 m 个 Worker 节点上。

并行梯度下降的流程： 用数据并发，设集合 $\mathcal{I}_1, \cdots, \mathcal{I}_m$ 是集合 $\{1, 2, \cdots, n\}$ 的划分；集合 $\mathcal{I}_k$ 包含第 k 个 Worker 节点上所有样本的序号。并行梯度下降需要重复——广播、映射、规约、更新参数——这四个步骤，直到算法收敛；见示意图 12.5。

1. 广播 (Broadcast)： 服务器将当前的模型参数 $\boldsymbol{w}_{\text{now}}$ 广播到 m 个 Worker 节点。这样一来，所有节点都知道 $\boldsymbol{w}_{\text{now}}$。
2. 映射 (Map)： 这一步让 m 个 Worker 节点做并行计算，用本地数据计算梯度。需要

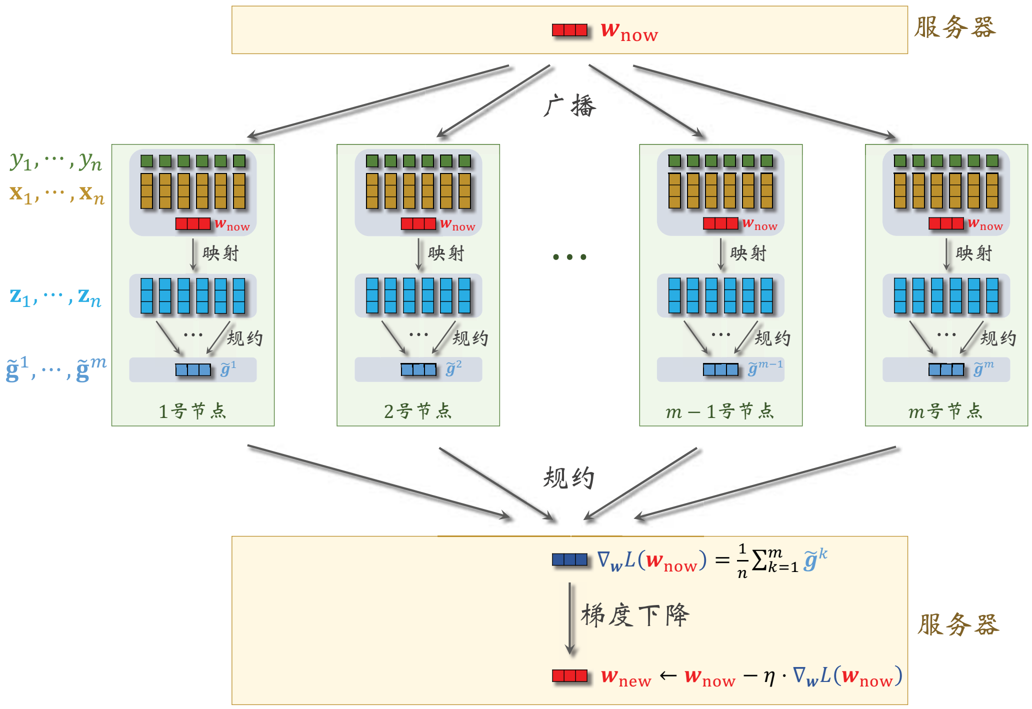

图 12.5: 并行梯度下降的流程。

在编程的时候定义这样一个映射函数：

$$g(x,\, y,\, w) \;=\; (x^T w - y)\, x.$$

第 k 号 Worker 节点做如下映射：

$$g \;:\; (x_j,\, y_j,\, w_{\text{now}}) \quad\mapsto\quad z_j = (x_j^T w_{\text{now}} - y_j)\, x_j, \qquad \forall\, j \in \mathcal{I}_k.$$

这样一来，第 k 号 Worker 节点得到向量的集合 $\{z_j\}_{j\in\mathcal{I}_k}$。

3. 规约 (Reduce)：在做完映射之后，向量 $z_1, \cdots, z_n \in \mathbb{R}^d$ 分布式存储在 m 个 Worker 节点上，每个节点有一个子集。不难看出，目标函数 $L(w) = \frac{1}{2n}\sum_{j=1}^n (x_j^T w - y_j)^2$ 在 w_{now} 处的梯度等于：

$$\nabla_w L(w_{\text{now}}) \;=\; \frac{1}{n}\sum_{j=1}^n z_j.$$

因此，我们应该使用 sum 规约函数。每个 Worker 节点首先会规约自己本地的 $\{z_j\}_{j\in\mathcal{I}_k}$，得到

$$\tilde{g}^k \;\triangleq\; \sum_{j\in\mathcal{I}_k} z_j, \qquad \forall\, k = 1, \cdots, m.$$

然后将 $\tilde{g}_k \in \mathbb{R}^d$ 发送给服务器，服务器对 $\tilde{g}^1, \cdots, \tilde{g}^k$ 求和，再除以 n，得到梯度：

$$\nabla_w L(w_{\text{now}}) \;\leftarrow\; \frac{1}{n}\sum_{k=1}^m \tilde{g}^k.$$

先在本地做规约，再做通信，只需要传输 md 个浮点数；如果不先在本地归约，直接把所有的 $\{z_j\}_{j=1}^n$ 都发送给服务器，那么需要传输 nd 个浮点数，通信代价大得

多。

4. 更新参数： 最后，服务器在本地做梯度下降，更新模型参数：

$$\boldsymbol{w}_{\text{new}} \;\leftarrow\; \boldsymbol{w}_{\text{now}} \;-\; \eta \cdot \nabla_{\boldsymbol{w}} L\big(\boldsymbol{w}_{\text{now}}\big).$$

这样就完成了一轮梯度下降，对参数做了一次更新。

12.1.4 并行计算的代价

通常用算法实际运行所需的时间来衡量并行计算的表现。时间有两种定义，请读者注意区分。

1. **钟表时间** (Wall-clock Time)，也叫 Elapsed Real Time，意思是程序实际运行的时间。可以这样理解钟表时间：在程序开始运行的时候，记录下墙上钟表的时刻；在程序结束的时候，再记录钟表的时刻；两者之差就是钟表时间。
2. **处理器时间** (CPU Time 或 GPU Time) 是所有处理器运行时间的总和。比如使用 4 块 CPU 做并行计算，程序运行的钟表时间是 1 分钟，期间 CPU 没有空闲，那么系统的 CPU 时间等于 4 分钟。

处理器数量越多，每块处理器承担的计算量就越小，那么程序运行速度就会越快。所以并行计算可以让钟表时间更短。用多个处理器做并行计算，然而总计算量没有减少，因此并行计算不会让处理器时间更短。

通常用**加速比** (Speedup Ratio) 衡量并行计算带来的速度提升。加速比是这样计算的：

$$\text{加速比} \;=\; \frac{\text{使用一个节点所需的钟表时间}}{\text{使用 } m \text{ 个节点所需的钟表时间}}.$$

通常来说，节点数量越多，算力越强，加速比就越大。在实验报告中，通常需要把加速比绘制成一条曲线。把节点数量设置为不同的值，比如 $m = 1, 2, 4, 8, 16, 32$，得到相应的加速比。把 m 作为横轴，把加速比作为纵轴，绘制出加速比曲线；见图 12.6。

在最理想的情况下，使用 m 个节点，每个节点承担 $\frac{1}{m}$ 的计算量，那么钟表时间会减小到原来的 $\frac{1}{m}$，即加速比等于 m。图 12.6 中的蓝色直线是理想情况下的加速比。但实际的加速比往往是图中的红色曲线，即加速比小于 m。其原因在于计算所需时间只占总的钟表时间的一部分。通信等操作也要花费时间，导致加速比达不到 m。下面分析并行计算中常见的时间开销。

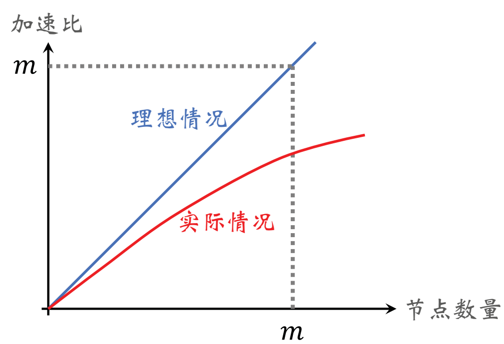

图 12.6: 加速比曲线。

通信量 (Communication Complexity) 的意思是有多少个比特或者浮点数在服务器与 Worker 节点之间传输。在并行梯度下降的例子中，每一轮梯度下降需要做两次通信：服务器将模型参数 $\boldsymbol{w} \in \mathbb{R}^d$ 广播给 m 个 Worker 节点，Worker 节点将计算出的梯度 $\tilde{\boldsymbol{g}}^1, \cdots, \tilde{\boldsymbol{g}}^m$ 发送给服务器。因此每一轮梯度下降的通信量都是 $\mathcal{O}(md)$。很显然，通信量越大，通信

花的时间越长。

延迟 (Latency) 是由计算机网络的硬件和软件系统决定的。做通信的时候，需要把大的矩阵、向量拆分成小数据包，通过计算机网络逐个传输数据包。即使数据包再小，从发送到接收之间也需要花费一定时间，这个时间就是延迟。通常来说，延迟与通信次数成正比，而跟通信量关系不大。

通信时间 主要由通信量和延迟造成。我们无法准确预估通信时间 (指的是钟表时间)，除非实际做实验测量。但我们不妨用下面的公式粗略估计通信时间：

$$\text{通信时间} \;\approx\; \frac{\text{通信量}}{\text{带宽}} + \text{延迟}.$$

在并行计算中，通信时间是不容忽视的，通信时间甚至有可能超过计算时间。降低通信量和通信次数是设计并行算法的关键。只有当通信时间远低于计算时间，才能取得较高的加速比。

12.2 同步与异步

本节讨论同步算法、异步算法的区别，重点介绍异步并行梯度下降。用在机器学习中，异步算法的表现通常优于同步算法。

12.2.1 同步算法

上一节介绍的并行梯度下降算法属于**同步算法** (Synchronous Algorithm)。如图 12.7 所示，在所有 Worker 节点都完成映射 (Map) 的计算之后，系统才能执行规约 (Reduce) 通信。这意味着即使有些节点先完成计算，也必须等待最慢节点；在等待期间，节点处于空闲状态。图 12.7 中黑色的竖线表示同步屏障，即所有节点都完成计算之后才能开始通信，当通信完成之后才能开始下一轮计算。

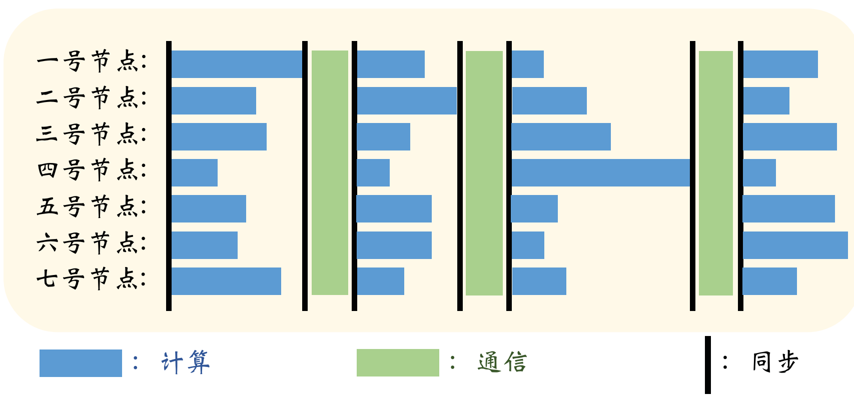

图 12.7: **同步梯度下降**中的计算、通信、同步。图中横向表示时间。

同步的代价: 实际软硬件系统中存在负载不平衡、软硬件不稳定、I/O 速度不稳定等因素。因此 Worker 节点会有先后、快慢之分，不会恰好在同一时刻完成任务。同步要求每一轮都必须等待所有节点完成计算，这势必导致 "短板效应"，即任务所需时间取决于最慢的节点。同步会造成很多节点处于空闲状态，无法有效利用集群的算力。

Straggler Effect 意思是一个节点的速度远慢于其余节点，导致整个系统长时间处于空闲状态，等待最慢的节点。Straggler 也叫 Outlier，字面意思是 "掉队者"。产生 Straggler 的原因有很多种，比如在某个节点的硬件或软件出错之后，节点死掉或者重启，导致计算时间多几倍。如果把 MapReduce 这样的需要同步的系统部署到廉价、可靠性低的硬件上，Straggler Effect 可能会很严重。

12.2.2 异步算法

如果把图 12.7 中的同步屏障去掉，得到的算法就叫做**异步算法** (Asynchronous Algorithm)，如图 12.8 所示。在异步算法中，一个 Worker 节点无需等待其余节点完成计算或通信。当一个 Worker 节点完成计算，它立刻跟 Server 通信，然后开始下一轮的计算。异步算法避免了等待，节点几乎没有空闲的时间，因此系统的利用率很高。

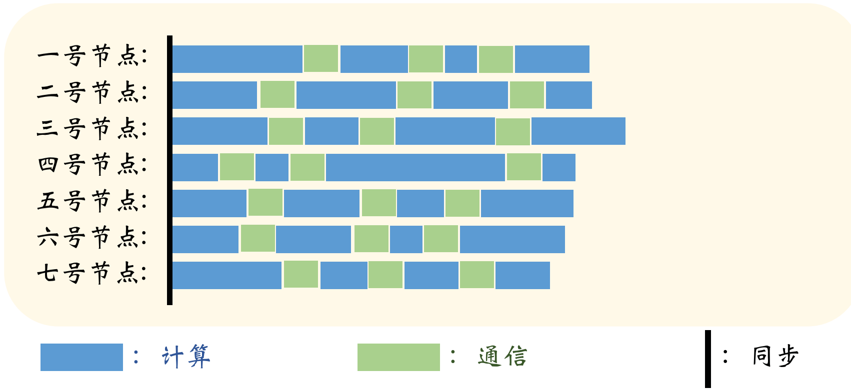

图 12.8: **异步算法**中的计算、通信、同步。图中横向表示时间。

下面介绍异步梯度下降算法。我们仍然采用数据并发的方式，即把数据集 $\{(\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_n, y_n)\}$ 划分到 m 个 Worker 节点上。如图 12.9 所示，服务器可以单独与某个 Worker 节点通信：Worker 节点把计算出梯度发送给服务器，服务器把最新的参数发送给这个 Worker 节点。如果想要编程实现异步算法，可以用 Message Passing Interface (MPI) 这样底层的库，也可以借助 Ray[3] 这样的框架。用户需要做的工作是编程实现 Worker 端、服务器端的计算。

图 12.9: 异步梯度下降。

Worker 端的计算： 每个 Worker 节点独立做计算，独立与服务器通信；Worker 节点之间不通信，不等待。第 k 号 Worker 节点重复下面的步骤：

1. 向服务器发出请求，索要最新的模型参数。把接收到的参数记作 $\boldsymbol{w}_{\text{now}}$。
2. 利用本地的数据 $\{(\boldsymbol{x}_j, y_j)\}_{j \in \mathcal{I}_k}$ 和参数 $\boldsymbol{w}_{\text{now}}$ 计算本地的梯度：

$$\tilde{\boldsymbol{g}}^k = \frac{1}{|\mathcal{I}_k|} \sum_{j \in \mathcal{I}_k} (\boldsymbol{x}_j^T \boldsymbol{w}_{\text{now}} - y_j) \boldsymbol{x}_j.$$

[3] https://ray.io/

3. 把计算出的梯度 $\tilde{\boldsymbol{g}}^k$ 发送给服务器。

服务器端的计算： 服务器上储存一份模型参数，并且用 Worker 发来的梯度更新参数。每当收到一个 Worker（比如第 k 号 Worker）发送来的梯度（记作 $\tilde{\boldsymbol{g}}^k$），服务器就立刻做梯度下降更新参数：

$$\boldsymbol{w}_{\text{new}} \;\leftarrow\; \boldsymbol{w}_{\text{now}} \;-\; \eta \cdot \tilde{\boldsymbol{g}}^k.$$

服务器还需要监听 Worker 发送的请求。如果有 Worker 索要参数，就把当前的参数 $\boldsymbol{w}_{\text{new}}$ 发送给这个 Worker。

12.2.3 同步与异步梯度下降的对比

上一节介绍的同步并行梯度下降完全**等价于**标准的梯度下降，只是把计算分配到了多个 Worker 节点上而已。然而异步梯度下降算法与标准的梯度下降是**不等价**的。同步与异步梯度下降不只是编程实现有区别，更是在算法上有本质区别。

1. 不难证明，**同步并行梯度下降**更新参数的方式为：

$$\boldsymbol{w}_{\text{new}} \;\leftarrow\; \boldsymbol{w}_{\text{now}} \;-\; \eta \cdot \nabla_{\boldsymbol{w}} L\big(\boldsymbol{w}_{\text{now}}\big),$$

即标准的梯度下降。在同一时刻，所有 Worker 节点上的参数是相同的，都是 $\boldsymbol{w}_{\text{now}}$。所有 Worker 节点都基于相同的 $\boldsymbol{w}_{\text{now}}$ 计算梯度。

2. 对于**异步并行梯度下降**，在同一时刻，不同 Worker 节点上的参数 $\boldsymbol{w}$ 通常是不同的。比如两个 Worker 分别在 t_1 和 t_2 时刻向服务器索要参数。在两个时刻之间，服务器可能已经对参数做了多次更新，导致在 t_1 和 t_2 时刻取回的参数不同。两个 Worker 节点会基于不同的参数计算梯度。

在理论上，异步梯度下降的收敛速度慢于同步算法，即需要更多的计算量才能达到相同的精度。但是实践中异步梯度下降远比同步算法快（指的是钟表时间），这是因为异步算法无需等待，Worker 节点几乎不会空闲，利用率很高。

12.3 并行强化学习

并行强化学习的目的在于用更少的钟表时间完成训练。第 12.3.1、12.3.2 小节分别用异步并行算法训练 DQN、Actor-Critic。本节介绍的异步算法与上一节的异步算法很类似，都是由 Worker 节点计算梯度，由服务器更新模型参数。

12.3.1 异步并行双 Q 学习

DQN 和双 Q 学习： DQN 是一个神经网络，记作 $Q(s,a;\boldsymbol{w})$，其中 s 是状态，a 是动作，$\boldsymbol{w}$ 表示神经网络参数（包含多个向量、矩阵、张量）。通常用双 Q 学习等算法训练 DQN。双 Q 学习需要目标网络 $Q(s,a;\boldsymbol{w}^{-})$，它的结构与 DQN 相同，但是参数不同。双 Q 学习属于异策略，即由任意策略控制智能体收集经验，事后做经验回放更新 DQN 参数。第 6 章介绍的高级技巧可以很容易地与双 Q 学习结合，此处就不详细解释了。

系统架构： 如图 12.10 所示，系统中有一个服务器和 m 个 Worker 节点。服务器可以随时给某个 Worker 发送一条信息，一个 Worker 也可以随时给服务器发送信息，但是 Worker 之间不能通信。服务器和 Worker 都存储 DQN 的参数。服务器上的参数是最新的，服务器用 Worker 发来的梯度对参数做更新。Worker 节点的参数可能是过时的，所以 Worker 需要频繁向服务器索要最新的参数。Worker 节点有自己的目标网络，而服务器上不存储目标网络。每个 Worker 节点有自己的环境，比如运行一个超级玛丽游戏，用 DQN 控制智能体与环境交互，收集经验，把 (s,a,r,s') 这样的四元组存储到本地的经验回放数组。在收集经验的同时，Worker 节点做经验回放，计算梯度，把梯度发送给服务器。

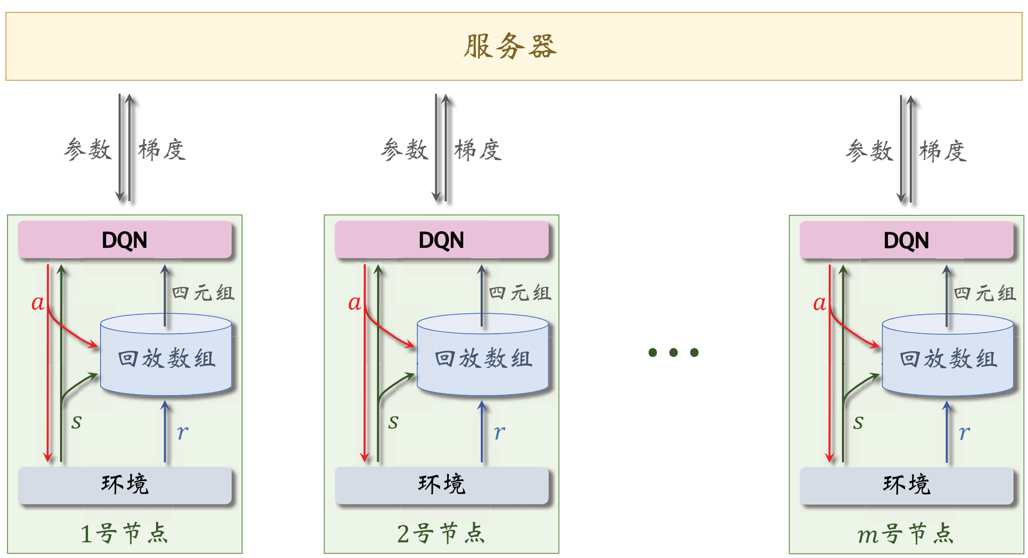

图 12.10: 用异步并行算法训练 DQN。图中没有画出目标网络。

Worker 端的计算： 每个 Worker 节点本地有独立的环境，独立的经验回放数组，还有一个 DQN 和一个目标网络。（图 12.10 中没有画出目标网络。）设某个 Worker 节点当

前参数为 $\boldsymbol{w}_{\text{now}}$。它用 ϵ-greedy 策略控制智能体与本地环境交互，收集经验。ϵ-greedy 的定义是：

$$a_t \;=\; \begin{cases} \operatorname{argmax}_a Q(s_t, a; \boldsymbol{w}_{\text{now}}), & \text{以概率 } (1-\epsilon); \\ \text{均匀抽取 } \mathcal{A} \text{ 中的一个动作}, & \text{以概率 } \epsilon. \end{cases}$$

把收集到的经验 (s_t, a_t, r_t, s_{t+1}) 存入本地的经验回放数组。

与此同时，所有的 Worker 节点都要参与异步梯度下降。Worker 节点在本地做计算，还要与服务器通信。第 k 号 Worker 节点重复下面的步骤：

1. 向服务器发出请求，索要最新的 DQN 参数。把接收到的参数记作 $\boldsymbol{w}_{\text{new}}$。
2. 更新本地的目标网络：
$$\boldsymbol{w}_{\text{new}}^{-} \;\leftarrow\; \tau \cdot \boldsymbol{w}_{\text{new}} \;+\; (1-\tau) \cdot \boldsymbol{w}_{\text{now}}^{-}.$$
3. 在本地做经验回放，计算本地梯度：
   (a). 从本地的经验回放数组中随机抽取 b 个四元组，记作
   $$(s_1, a_1, r_1, s_1'), \;\; (s_2, a_2, r_2, s_2'), \;\; \cdots, \;\; (s_b, a_b, r_b, s_b').$$
   b 是批量的大小，由用户自己设定，比如 $b=16$。
   (b). 用双 Q 学习计算 TD 目标。对于所有的 $j=1,\cdots,b$，分别计算
   $$\widehat{y}_j \;=\; r_j + \gamma \cdot Q(s_j', a_j'; \boldsymbol{w}_{\text{new}}^{-}), \qquad \text{其中} \;\; a_j' \;=\; \operatorname*{argmax}_a Q(s_j', a; \boldsymbol{w}_{\text{new}}).$$
   (c). 定义目标函数：
   $$L(\boldsymbol{w}) \;\triangleq\; \frac{1}{2b} \sum_{j=1}^{b} \Big[Q(s_j, a_j; \boldsymbol{w}) \;-\; \widehat{y}_j \Big]^2.$$
   (d). 计算梯度：
   $$\tilde{\boldsymbol{g}}^k \;=\; \nabla_{\boldsymbol{w}} L(\boldsymbol{w}_{\text{new}}).$$
4. 把计算出的梯度 $\tilde{\boldsymbol{g}}^k$ 发送给服务器。

服务器端的计算： 服务器上储存有一份模型参数，记作 $\boldsymbol{w}_{\text{now}}$。每当一个 Worker 节点发来请求，服务器就把 $\boldsymbol{w}_{\text{now}}$ 发送给该 Worker 节点。每当一个 Worker 节点发来梯度 $\tilde{\boldsymbol{g}}^k$，服务器就立刻做梯度下降更新参数：

$$\boldsymbol{w}_{\text{new}} \;\leftarrow\; \boldsymbol{w}_{\text{now}} \;-\; \alpha \cdot \tilde{\boldsymbol{g}}^k.$$

12.3.2 A3C: 异步并行 A2C

A2C 有一个策略网络 $\pi(a|s;\boldsymbol{\theta})$ 和一个价值网络 $v(s;\boldsymbol{w})$。通常用策略梯度更新策略网络，用 TD 算法更新价值网络。为了让 TD 算法更稳定，需要一个目标网络 $v(s;\boldsymbol{w}^{-})$，它的结构与价值网络相同，但是参数不同。A2C 属于同策略，不能使用经验回放。A2C 的实现详见第 8.3 节。异步并行 A2C 被称作 Asynchronous Advantage Actor-Critic (A3C)。

系统架构： 如图 12.10 所示，系统中有一个服务器和 m 个 Worker 节点。服务器维护策略网络和价值网络最新的参数，并用 Worker 节点发来的梯度更新参数。每个 Worker 节点有一份参数的拷贝，并每隔一段时间向服务器索要最新的参数。每个 Worker 节点有

一个目标网络，而服务器上不储存目标网络。每个 Worker 节点有独立的环境，用本地的策略网络控制智能体与环境交互，用状态、动作、奖励计算梯度。

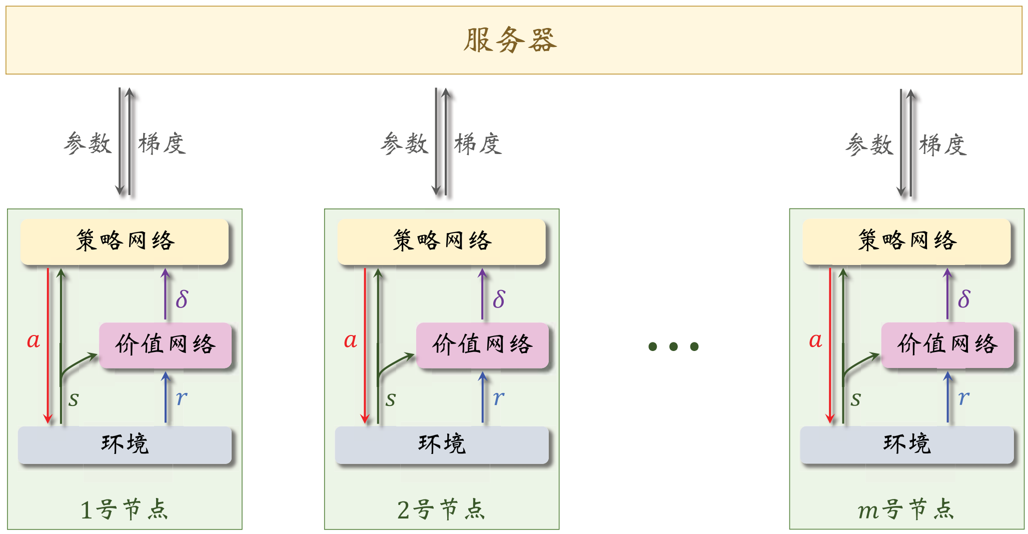

图 12.11: A3C，即异步并行 A2C。图中没有画出目标网络。

Worker 端的计算： 每个 Worker 节点有独立的环境，独立做计算，随时可以与服务器通信。每个 Worker 节点本地有一个策略网络 $\pi(a|s;\boldsymbol{\theta})$、一个价值网络 $v(s;\boldsymbol{w})$、一个目标网络 $v(s;\boldsymbol{w}^{-})$。设第 k 个 Worker 节点当前参数为 $\boldsymbol{\theta}_{\text{now}}$、$\boldsymbol{w}_{\text{now}}$、$\boldsymbol{w}^{-}_{\text{now}}$。第 k 个 Worker 节点重复下面的步骤：

1. 向服务器发出请求，索要最新的参数。把接收到的参数记作 $\boldsymbol{\theta}_{\text{new}}$、$\boldsymbol{w}_{\text{new}}$。
2. 更新本地的目标网络：
$$\boldsymbol{w}^{-}_{\text{new}} \ \leftarrow\ \tau\cdot\boldsymbol{w}_{\text{new}} \ +\ \big(1-\tau\big)\cdot\boldsymbol{w}^{-}_{\text{now}}.$$
3. 重复下面的步骤 b 次（b 是用户设置的超参数），或是从头到尾完成一回合游戏。让智能体与环境交互，计算策略梯度，并累积策略梯度。全零初始化 $\tilde{\boldsymbol{g}}_\theta^k\leftarrow\boldsymbol{0}$、$\tilde{\boldsymbol{g}}_w^k\leftarrow\boldsymbol{0}$，用它们累积梯度。
   (a). 基于当前状态 s_t，根据策略网络做决策 $a_t\sim\pi(\cdot\,|\,s_t,\boldsymbol{\theta})$，让智能体执行动作 a_t。随后观测到奖励 r_t 和新状态 s_{t+1}。
   (b). 计算 TD 目标 $\widehat{y}_t$ 和 TD 误差 δ_t：[4]
$$\begin{aligned}\widehat{y}_t &= r_t + \gamma\cdot v\big(s_{t+1};\,\boldsymbol{w}^{-}_{\text{new}}\big),\\ \delta_t &= v\big(s_t;\,\boldsymbol{w}_{\text{new}}\big) - \widehat{y}_t.\end{aligned}$$
   (c). 累积梯度：
$$\begin{aligned}\tilde{\boldsymbol{g}}_w^k &\leftarrow \tilde{\boldsymbol{g}}_w^k + \delta_t\cdot\nabla_{\boldsymbol{w}} v\big(s_t;\,\boldsymbol{w}_{\text{new}}\big),\\ \tilde{\boldsymbol{g}}_\theta^k &\leftarrow \tilde{\boldsymbol{g}}_\theta^k + \delta_t\cdot\nabla_{\boldsymbol{\theta}}\ln\pi\big(a_t\,\big|\,s_t;\,\boldsymbol{\theta}_{\text{new}}\big).\end{aligned}$$

[4]此处可以用多步 TD 目标等技巧；详见第 5.3 节。

4. 把累积的梯度 $\tilde{\boldsymbol{g}}_{\theta}^{k}$ 和 $\tilde{\boldsymbol{g}}_{w}^{k}$ 发送给服务器。

服务器端的计算： 服务器上储存有一份模型参数，记作 $\boldsymbol{\theta}_{\text{now}}$ 和 $\boldsymbol{w}_{\text{now}}$。每当一个 Worker 节点发来请求，服务器就把 $\boldsymbol{\theta}_{\text{now}}$ 和 $\boldsymbol{w}_{\text{now}}$ 发送给该 Worker 节点。每当一个 Worker 节点发来梯度 $\tilde{\boldsymbol{g}}_{\theta}^{k}$ 和 $\tilde{\boldsymbol{g}}_{w}^{k}$，服务器就立刻做梯度下降更新参数：

$$
\begin{aligned}
\boldsymbol{w}_{\text{new}} \quad &\leftarrow \quad \boldsymbol{w}_{\text{now}} - \alpha \cdot \tilde{\boldsymbol{g}}_{w}^{k}, \\
\boldsymbol{\theta}_{\text{new}} \quad &\leftarrow \quad \boldsymbol{\theta}_{\text{now}} - \beta \cdot \tilde{\boldsymbol{g}}_{\theta}^{k}.
\end{aligned}
$$

第十二章 相关文献

MapReduce 原本是指 Google 内部使用的软件系统，现在泛指这类系统架构。Google 的 MapReduce 系统不对外开源，但是外界可以通过 2008 年的论文 [33] 了解系统的设计。外界有多个开源项目力图实现 MapReduce 系统，其中最有名的是 Hadoop。后来基于 Hadoop 等项目开发的 Spark [123] 比 Hadoop MapReduce 的速度更快。本章介绍的异步并行算法主要基于 Parameter Server [60] 的思想。Ray [71] 是一个开源的软件系统，包含 Parameter Server 的功能。用 Ray 很容易实现异步并行算法，而且 Ray 对强化学习有很好的支持。

本章介绍的并行强化学习算法主要基于 2015 年的论文 [73] 和 2016 年的论文 [68]。两篇论文都是异步算法，主要区别在于 2015 年的论文 [73] 使用经验回放，而 2016 年的论文 [68] 不用经验回放。对于 Atari 游戏这类问题，获取经验非常容易，于是使用经验回放与否其实无关紧要。