

实施 AlphaZero 的经验教训



阿迪亚普拉萨德

跟随



2018 年 6 月 6 日 · 4 分钟阅读

DeepMind 的 [AlphaZero](#) 出版物是棋盘游戏强化学习 (RL) 的里程碑。该算法在国际象棋、将棋和围棋中实现了超人的表现，每一种都有不到 24 小时的自我对弈，除了规则之外，几乎没有使用任何专业或硬编码的游戏知识。

我们认为以新颖的方式复制和扩展他们的结果将是有益的，在此过程中发现各种选择如何影响算法的性能。机器学习作为一个整体只是慢慢地建立在坚实的理论基础上，因此探索新方向通常会收获很多。

本文是我们分享探索过程中收集到的见解系列的第一篇。

在第一篇文章中，我们将概述未修改的 AlphaZero 算法。我们不会让您自己阅读论文，而是尝试以相对 RL 新手可以理解的方式进行布局，并链接到外部资源以获取更多详细信息。

在高层次上，它使用由深度残差神经网络（或“ResNet”）支持的修改后的蒙特卡洛树搜索（MCTS）。AlphaZero 与自己对战，双方选择由 MCTS 选择的动作。这些自我博弈的结果用于不断改进 ResNet。self-play 和 ResNet 训练并行进行，相互改进。

让我们从 ResNet 开始解压。

神经网络

ResNet 架构是训练深度网络的一种流行方式，通常用于图像识别。对于 AlphaZero 的网络，输入是棋盘状态，有两个输出：

1. 位置的估计值 (v)，范围从 1 (赢) 到 -1 (输)。
2. 用于播放每个下一个可能动作的先验概率(p)向量。

网络是随机初始化的。在发生一些自我对弈后，训练数据由从玩过的游戏中随机选择的棋盘位置组成，形式为（棋盘状态、游戏结果、来自 MCTS 的儿童访问计数）。

MCTS

该视频对标准 MCTS 进行了出色的演练。为了在给定棋盘位置的情况下选择走法，我们执行以下算法的 800 次迭代。我们构建一棵树，最初只有一个节点（代表当前的棋盘状态）。

比如是状态给出当前状态 s_i ，随机选择动作 a_i 形成新的状态 s_{i+1} ，计算 (s_i, a_i) 这个动作状态的相应的变量值，在 (s_i, a_i) 这个状态下完成一次 MCTS 迭代，同样由在 s_i 的状态开始下一次迭代，动作 a'_i 形成 (s_i, a'_i) 的新状态 s_{i+1}' ，在这个状态下完成 MCTS 计算迭代；又下一轮，可能有采用 a_i 动作，形成 s_{i+1} ，但 s_{i+1} 可以展开，继续 MCTS 计算

1. 从根节点（当前板状态）开始。走到在利用当前信息和探索新动作之间做出最佳权衡的孩子（由 UCB1 公式正式化）。递归直到你碰到一个叶子节点。

2. 如果这是对该节点的第一次访问，则执行一次滚动：随机模拟移动直到游戏结束，然后使用该游戏结果将所有节点的值从叶子更新回根。
3. 如果是第二次访问，则展开它（即创建其子项），并访问+推出其中一个。
4. 没有第三次访问，因为它不再是叶子节点。

在 AlphaZero 中，rollouts 被从 NN 中获取预测所取代，而 UCB1 被 PUCT（多项式上置信树）所取代。该算法如下所示：

1. 从根开始。走向 PUCT 得分最高的孩子。递归直到你碰到一个叶子节点。
2. 在第一次访问时，调用 NN 来获取 (1) 估计的游戏分数（或 *value*, *v*）和 (2) 访问每个孩子的建议概率 (*p*)，以用于 PUCT。创建孩子，但不要拜访他们。
3. 没有第二次访问。

把它绑在一起

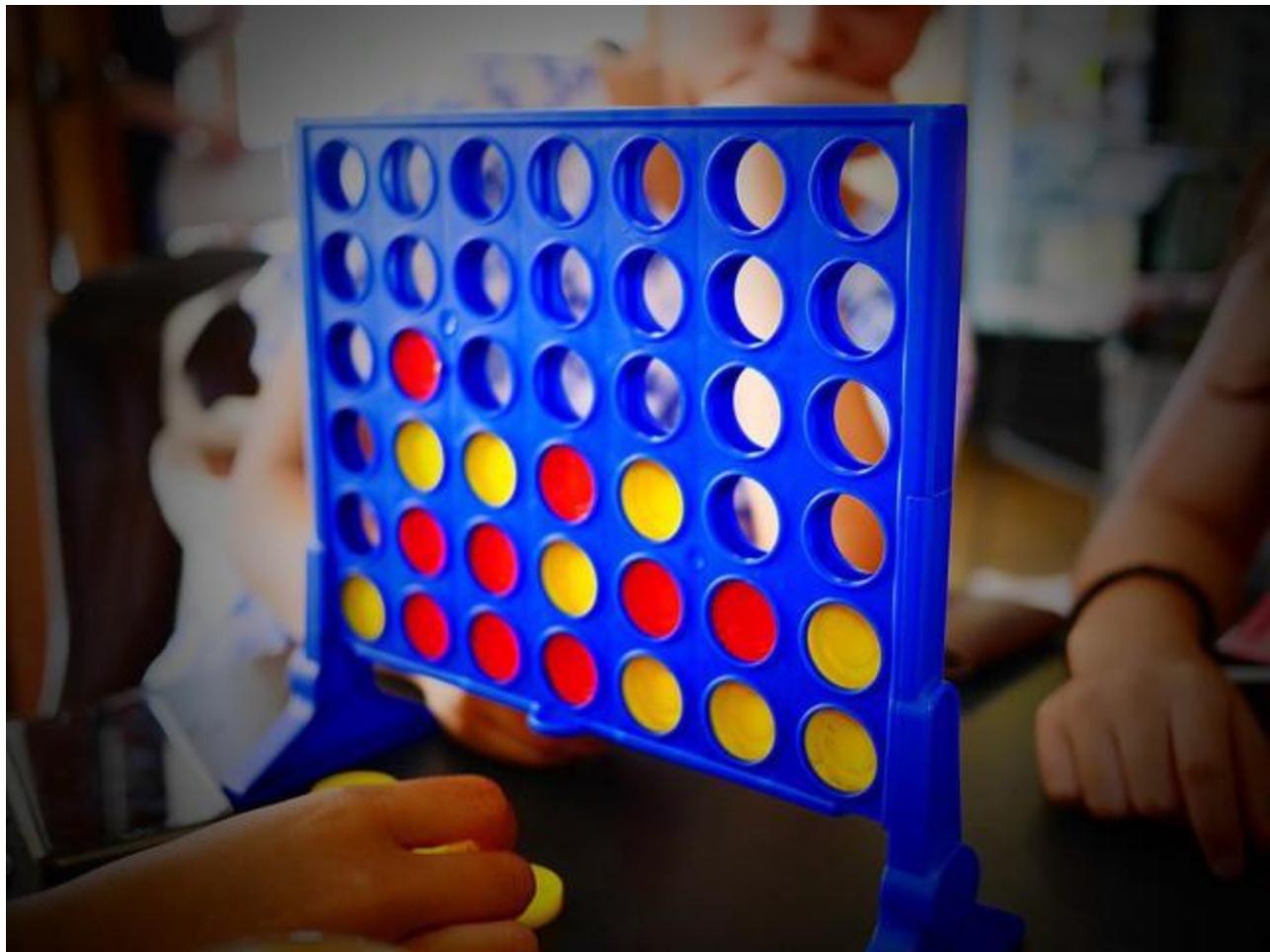
在 800 次 MCTS 迭代之后，通过选择访问次数最多的孩子来选择移动。然后对方以同样的方式进行比赛。这种情况一直持续到游戏结束。那时，游戏中的每个棋盘位置都标有游戏得分和儿童访问计数。这些样本被添加到 ResNet 的训练集中。随着 ResNet 的训练，它被用于后续的 MCTS self-play。

这种 MCTS self-play 和 ResNet 训练之间的来回切换非常强大。ResNet 有助于泛化从 MCTS 中学到的知识，而自我对弈使用泛化的知识来更深入地学习游戏。正如我们从论文中看到的那样，这使得 AlphaZero 的性能基本上优于所有现有算法——其中最好的算法是在各自游戏大师的帮助下精心手工调整的。

这是对 AlphaZero 训练的一个非常粗略的概述。我们遗漏了过多的细节，我们将在接下来的剧集中进行探讨。

敬请关注！

[第2部分](#)、[第3部分](#)、[第4部分](#)、[第5部分](#)和[第6部分](#)现已发布。



AlphaZero 的经验教训：连接四个



阿迪亚普拉萨德 [跟随](#)

2018年6月14日 · 5分钟阅读

欢迎回来！在我们的上一期文章中，我们简要回顾了DeepMind的AlphaZero算法如何快速学会比最优秀的人类（有时甚至是最好的现有算法）更好地玩游戏。回想一下，它使用基于神经网络的蒙特卡洛树搜索(MCTS)与自己对战，并在自我强化循环中使用这些自我对弈游戏的结果来进一步训练网络。

在接下来的几篇文章中，我们将分享一些我们从教它玩Connect Four中学到的经验。

为什么连接四个？

为了实现和测试 AlphaZero，我们首先必须选择一个游戏让它玩。与围棋（一个极端难度）或井字游戏（另一个极端）不同，四连接似乎提供了足够有趣的复杂性，同时仍然足够小以快速迭代。特别是，我们选择了常见的 6x7 变体（如上图所示）。

它还有另一个好处：作为一个完全解决的游戏，我们可以根据最优策略测试我们的模型。也就是说，一旦我们使用 AlphaZero 训练了我们的网络，我们就可以向它提供我们知道正确答案的测试板位置，以获得有意义的“基本事实”准确性。

此外，它使我们能够发现在给定架构下我们的网络可以变得多么接近完美。让我们来看看它是如何工作的。

训练

训练和评估过程如下：

1. 使用 Connect Four 求解器（即，将告诉您任何棋盘位置的正确移动的程序）生成带标签的训练和测试集。
2. 选择神经网络架构。AlphaZero 论文提供了一个很好的起点，但有理由认为不同的游戏将受益于它的调整。
3. 使用监督学习来训练网络标记的训练集。这为我们提供了架构应该能够学习多好的上限。
4. 使用相同的网络架构训练 AlphaZero，并使用标记的测试集对其进行评估，以查看它与上限的接近程度。

假设 AlphaZero 训练的网络达到 90% 的测试准确率。这告诉我们什么？

- 如果相同的架构通过监督训练达到 99%，那么网络可以比 AlphaZero 教它学习更多，我们应该改进 AlphaZero 的自我游戏组件。
- 如果监督训练只达到 91%，那么网络架构本身可能是限制因素，应该改进。

使用此策略帮助我们调整网络架构、发现错误并在我们的 MCTS 自我游戏中进行其他改进。

挑战

在生成带标签的训练数据时，我们将随机棋盘位置提供给 Connect Four 求解器。问题是，什么是“随机”？

因为 AlphaZero 应该学会只玩完美的游戏，它的网络最终将不再看到玩得不好的位置。因此，在他们身上进行测试是不公平的。另一方面，连接四很简单，只有少数完美的游戏，因此不可能仅使用完美的棋盘位置来构建大型训练集。

另一个困难是确定一个职位的正确答案是什么。在任何给定的位置上，都可能有多个导致获胜的动作。越快获胜越好吗？好多少？如果没有保证赢，你应该选择输得最慢的一步（假设一个完美的对手），还是最有可能绊倒他们的一步（如果他们不完美）？你如何做到这一点？

最后，我们决定使用非常不完美的游戏中的位置来生成我们的标记集，这使得我们经过 AlphaZero 训练的网络在比较中处于劣势。然而，正如我们将很快看到的那样，它在这个障碍上表现得非常好。

我们还决定使用两种不同的“最佳移动”定义。在生成“强大”的训练/测试集时，我们仅在可能导致最快获胜（或最慢可能失败）的情况下标记正确的移动。在“弱”的情况下，所有获胜的动作都被赋予相同的权重。为了保持比较公平，我们还在强模式和弱模式下训练了 AlphaZero，鼓励它在前一种情况下更喜欢更快的胜利（通过根据游戏长度缩放最终结果值）。让我们简要地看一下。

强 vs 弱 AlphaZero

在 MCTS 模拟过程中，非终端位置从网络中获取一个初始值，表示预期的博弈结果（在 [-1, 1] 范围内）。另一方面，终端位置不需要这样的估计；我们可以直接为它们分配 +/-1。这对于弱训练来说很好，无论需要多长时间，两个获胜位置都被认为是同样好的。对于强训练，我们按游戏长度缩放最终结果。

最快的四连胜是总共 7 步，最长的是 42 步（用完整个棋盘）。一些快速代数表明，如果 n 是实际游戏长度，那么表达式 $1.18 - (9*n/350)$ 将产生 [0.1, 1] 中的值以获得胜利。这使得 MCTS 更喜欢快速获胜（和缓慢失败）。

评价

我们进行了两种评估：

- **Network-only:** 向网络提供测试板位置并检查其预测的移动是否正确。
- **MCTS:** 运行由网络支持的 MCTS（具有 800 次模拟）并评估其首选动作。

回想一下，我们使用了两种不同的训练方法：

- **AlphaZero (“AZ”）培训。**

- 监督训练（即，使用来自求解器的标记训练数据）。

这为我们提供了四种评估组合：**AZ-Network**、**AZ-MCTS**、**Supervised-Network**、**Supervised-MCTS**。

最后，有两种训练模式（强与弱）。我们将依次查看每一个。

结果

强劲的结果

- 监督网络: **96.20%**
- 监督-MCTS: **97.29%**
- AZ-网络: **95.70%**
- AZ-MCTS: **96.95%**

这告诉我们的：

1. 我们的 AlphaZero 训练几乎与监督训练一样好（尽管处于前面提到的劣势）。
2. 尽管单独的网络做得非常好，但在 MCTS 中使用它进一步降低了大约 29% 的错误率（从 ~4% 到 ~3%）。

结果不佳

- 监督网络: **98.93%**
- 监督-MCTS: **99.79%**
- AZ-网络: **98.83%**
- AZ-MCTS: **99.76%**

这一次，AlphaZero 训练更接近于监督训练，几乎与之匹配。此外，MCTS 将错误率降低了约 80%，几乎每个位置都正确。

在弱测试集中，一个位置平均有 4.07 次正确移动（而强测试集中为 2.16）。这意味着在 7 个可能的移动中随机猜测只会让我们达到 58.1%。

结论

Connect Four 是一款出色的游戏，可用于训练（并发现其中的错误）您的 AlphaZero 实现。它允许速迭代，并且足够复杂以至于有趣。由于这是一个已解决的游戏，因

此也可以客观地衡量培训的成功程度。

当只寻找任何最佳移动时（无论它以多快的速度获胜或避免失败），并与 MCTS 相结合，经过 AlphaZero 训练的网络在 400 个位置中只会出现一次错误移动。即使必须找到最快的赢（或不输），并且仅使用网络预测，它也只会在 25 个位置中出错一次。非常好！

在接下来的几周里，我们将看看我们为获得最佳结果所做的一些调整。回头见！

第 3 部分现已[在此处发布](#)。

机器学习

关于 写 帮助 合法的

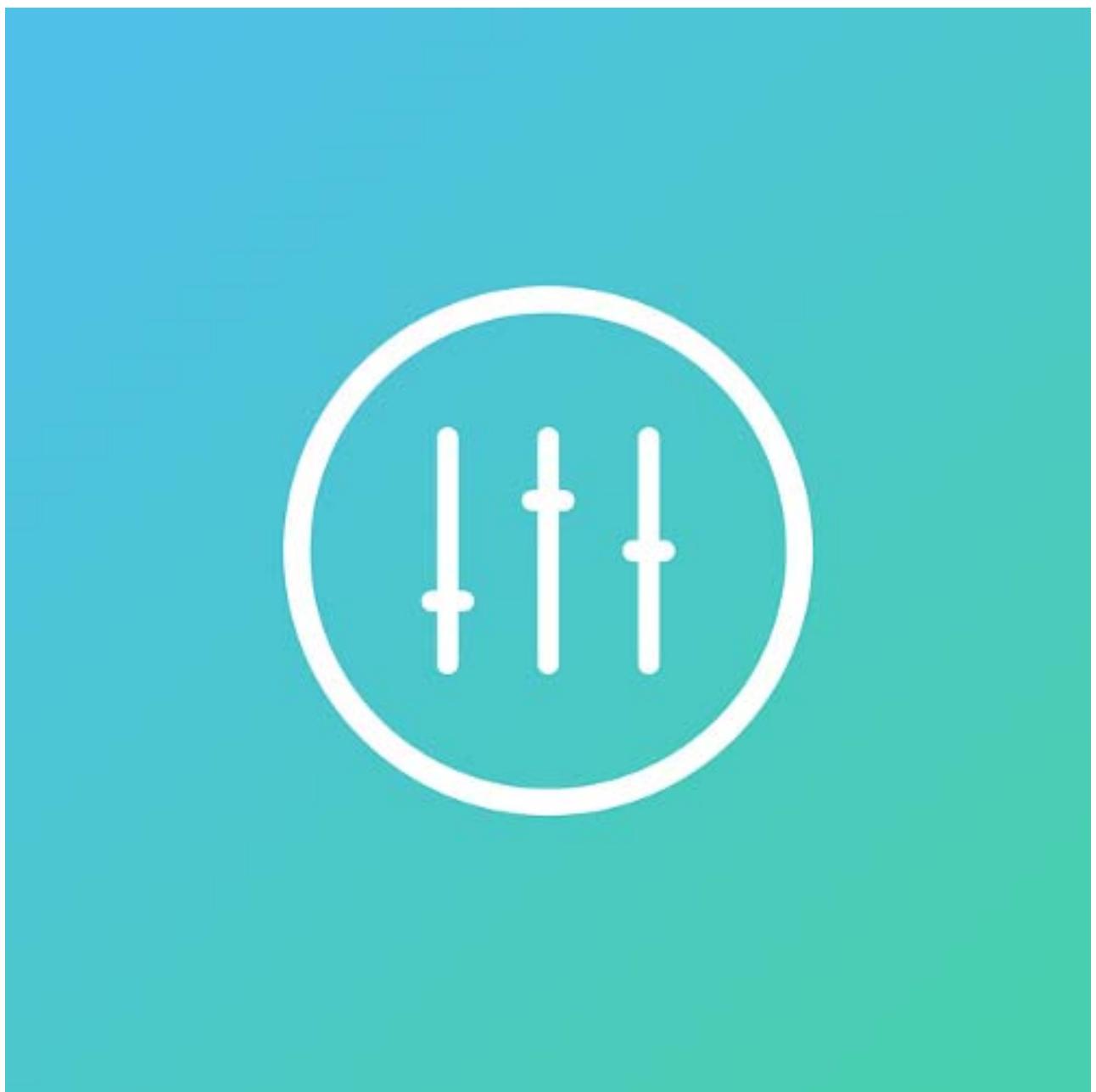
获取中等应用程序



AlphaZero 的经验教训（第 3 部分）：参数调整



阿迪亚普拉萨德 跟随
2018 年 6 月 21 日 · 7 分钟阅读



这是讨论我们如何以有趣的方式复制和扩展 DeepMind 的 AlphaZero 算法的系列文章的第三部分。你可以在这里找到第一部分。

再一次问好！在本周的部分中，我们将分享一些关于在 AlphaZero 中调整（超）参数的想法。这些是在训练期间没有学习到的参数（如网络权重），而是预先固定的。通过选择好的值，训练可能会进行得更快或更高。

其中许多参数会影响所谓的探索-利用权衡：算法需要在利用现有知识和探索新可能性之间取得平衡。在训练 AlphaZero 对抗高技能对手（如论文中所述）时，探索很重要，但在训练它在更广泛的位置上发挥出色时更为重要。如上一篇文章所述，我们的测试集包括不完美的游戏。因此，我们花了一些时间调整我们的实现，以进行足够广泛的探索，使其不仅是一个优秀的最佳玩家，而且是一个优秀的普通玩家。

AlphaZero 中的超参数

让我们看一下我们可能想要优化的一些超参数。在这里，我们只看一眼它们是什么和做什么。

c_puct

在蒙特卡洛树搜索 (MCTS) 模拟期间，该算法根据预期的游戏结果以及已经探索了多少来评估潜在的下一步行动。常数 c 用于控制这种权衡。

标准 MCTS 使用上置信界限 1 (UCB-1) 公式的变体，称为 UCT (树的 UCB)。

AlphaZero 使用称为多项式上置信树 (PUCT) 的版本。如果我们处于状态 s 并考虑动作 a ，我们需要三个值来计算 PUCT(s, a)：

- Q - 平均行动值。这是当前模拟中采取行动 a 的平均游戏结果。
- P ——从网络中获取的先验概率。
- N - 访问次数，或我们在当前模拟期间采取此操作的次数。

我们计算 $\text{PUCT}(s, a) = Q(s, a) + U(s, a)$ ，其中 U 计算如下：

$$U(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

请注意我们如何对分子中的所有潜在操作 (b) 进行求和，而考虑中的操作 (a) 的访问次数在分母中。因此，我们尝试这个动作的次数越少， U 就会越大。这鼓励探索。

通过增加 c_puct ，我们更加重视这个探索项。通过减少它，我们更重视利用预期结果(Q)。

在尝试了一些值（从 1 到 6）后，我们发现大约4是最佳值。

狄利克雷 α (阿尔法)

AlphaZero 的一项创新是将噪声引入 MCTS。

在 MCTS 模拟期间，我们反复模拟从代表当前棋盘状态的根节点 (s) 开始的游戏。我们要做的第一件事是向神经网络查询来自 s 的潜在动作的先验概率向量 (p)。

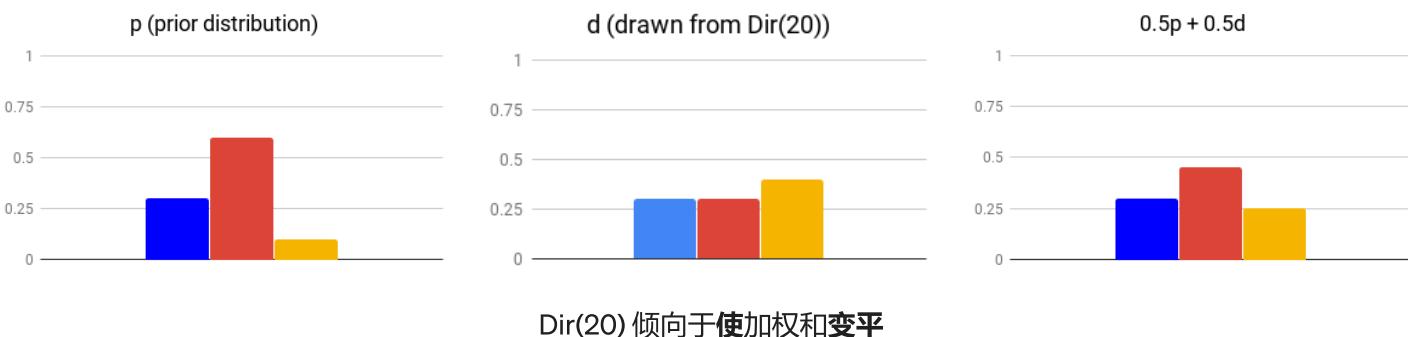
AlphaZero 不是按原样使用 p ，而是根据 Dirichlet 分布添加噪声，参数为 α ，即 $Dir(\alpha)$ 。为了帮助理解这意味着什么，让我们看一些具有不同 α 的 Dirichlet 分布示例。对于这些示例，我们将假设有 n 个潜在的下一步操作。

首先，回想一下狄利克雷的支持（即定义它的值）是一组向量 (x_1, x_2, \dots, x_n) ，其坐标为非负且总和为 1。如果 $n=3$ ，那么一些示例将为 $(0.1, 0.5, 0.4)$ 和 $(0.3, 0, 0.7)$ 。该集合也称为 (n 维) 单纯形。

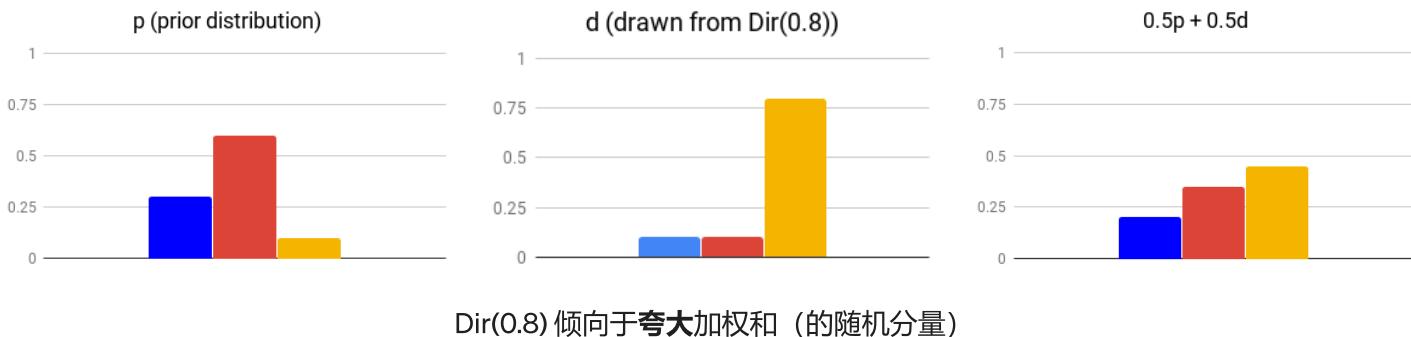
现在注意 $Dir(1)$ 是均匀的。这意味着从中采样将产生上述两个向量（或任何其他有效向量）的机会均等。随着 α 变小，Dirichlet 开始更喜欢基向量附近的向量： $(0.98, 0.01, 0.01)$ 或 $(0.1, 0.9, 0)$ 将比 $(0.3, 0.3, 0.4)$ 更有可能被绘制。对于大于 1 的值，情况正好相反——基向量不被强调，而更平衡的向量是首选。

为了给 p 添加噪声，我们从 $Dir(\alpha)$ 中采样一个值 d ，并取一个加权和： $x^* p + (1-x)^* d$ 。因为 p 和 d 的坐标总和为 1（权重 x 和 $1-x$ 总和为 1），所以此属性保留在加权和中，使其成为有效的概率分布。在论文中， x 设置为 0.75。下面，我们将使用 $x = 0.5$ 来夸大 d 的效果。

当 α 大于 1 时，这往往会使 p “压平”，从而使我们不太喜欢任何一步：



对于 $\alpha < 1$ ，它会导致随机分量变得更加强调：



在这两种情况下，我们都被鼓励远离我们之前的探索（强烈倾向于中间移动）。

为了选择一个值，我们首先查看了他们用于国际象棋、将棋和围棋的值：0.3、0.15和0.03。这些游戏中的平均合法移动数为35、92和250。很难知道如何进行最佳推断，但一个合理的初步猜测似乎是选择 $\alpha = 10/n$ 。由于在四连接位置大约有四个合法移动，这给了我们 $\alpha=2.5$ 。确实，这大于1，而其余的则较小，但这个数字似乎在我们的测试中表现良好。稍微玩了一下，我们发现**1.75**做得更好。

更新：虽然我们训练的早期版本在 $\alpha=1.75$ 时表现最好，但我们最终确定 $\alpha=1.0$ 作为我们训练的最佳值。

更多的研究（和计算时间）将有助于在这里获得更多的洞察力。

温度, τ (tau)

在 MCTS 完成一轮模拟之后，在它选择一个动作之前，它已经为每个潜在的下一步动作累积了访问次数（以上 N ）。MCTS 的工作原理是最终访问好棋子的次数比访问坏棋子的频率高，并且很好地指示了在哪里下棋。

通常，这些计数被归一化并用作选择实际移动的分布。但是可以使用一个所谓的温度参数 (τ) 来首先对这些计数求幂： $N^{(1/\tau)}$ 。他们为游戏的前30步设置 $\tau=1$ （没有影响），然后在游戏的其余部分将其设置为无穷小值（标准化后，抑制除最大值之外的所有值）。

- $N = (1, 2, 3, 4)$
- 归一化 N : $(0.1, 0.2, 0.3, 0.4)$
- 使用 $\tau=1$: $(0.1, 0.2, 0.3, 0.4)$
- 使用 $\tau \sim 0$: $(0, 0, 0, 1)$

因此，前30个动作有点探索性，其余的只会播放访问次数最多的动作。

我们决定试一试。首先，我们尝试将 τ 保留为 1 而不是减小它。这导致了性能的显著提高。受到这次成功的鼓舞，我们尝试了一些在边缘的生活，并在整个游戏中将其提高到 1.75：

- 使用 $\tau=1.75$: (0.15, 0.23, 0.29, 0.34)

τ 越大，分布越平坦，从而导致更均匀的探索。尽管 $\tau=1.75$ 在我们尝试的值中给了我们最好的结果，但我们发现它并不比简单地增加 `c_puct` 更好。所以我们只是把它留在了 1。

其他调整

除了调整超参数之外，我们还尝试了一些其他修改。

Nvidia-TensorRT 和 int8

Nvidia 提供了一种推理优化工具，可将我们的推理速度提高 3-4 倍。除了优化 TensorFlow 图，它还支持8 位量化，以牺牲精度换取吞吐量。

这种量化的副作用是它鼓励探索。例如，如果神经网络预测两个位置的结果略有不同（范围 [-1, 1] 内的数字），则它们可能会在精度上有少许损失时看起来相等。

位置重复数据删除

在训练神经网络时，会在最近的游戏中随机选择位置。由于四连线很少有合理的开局线，我们发现这导致早期董事局位置的比例显著过高。这导致网络过于关注他们。

我们通过在随机选择位置进行训练之前对位置进行重复数据删除来缓解这个问题。在这个过程中，我们平均了先验值和结果值。这似乎运作良好。

结论

与机器学习的大部分内容一样，AlphaZero 论文有很多没有得到充分解释的“幻数”。很难知道为确定提供的值做了多少探索。对于某些参数，有原则的方法可以帮助缩小选择范围，但总的来说，这是一个“试试看”的游戏。

探索与利用是强化学习的核心概念，因此，我们所做的大部分调整都以各种方式影响了这种权衡。

我们当然还没有找到最佳选择，但通过大量实验，我们帮助 AlphaZero 在 Connect Four 上的表现比“开箱即用”要好得多。

第 4 部分现已[在此处发布](#)。

AlphaZero 的经验教训（第 4 部分）：改进训练目标



维什 (Ishaya) 艾布拉姆斯 [跟随](#)

2018 年 6 月 28 日 · 6 分钟阅读



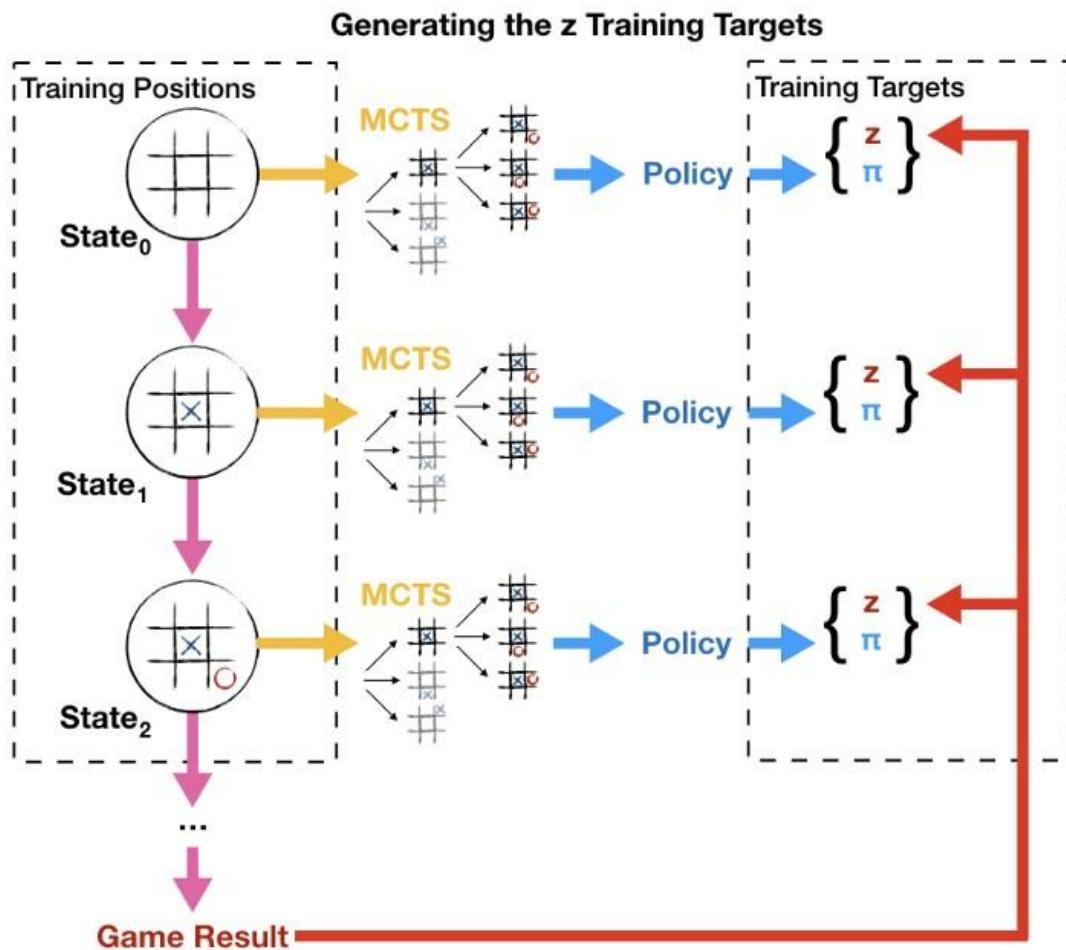
这是我们从实施 *AlphaZero* 中吸取的经验教训系列的第四部分。查看[第 1 部分](#)、[第 2 部分](#)和[第 3 部分](#)。

我们的 *AlphaZero* 方法中的一项创新涉及网络价值输出的目标。我们为网络的价值输出找到了一个替代的训练目标，它优于 *AlphaZero* 方法。

培训目标审查

原始论文指定了神经网络的两个输出：策略输出和值输出。预测当前游戏位置的最佳移动的策略输出是针对由 π 表示的值进行训练的，该值是基于自我对战期间 MCTS 搜索累积的访问计数的概率分布。预测游戏结果的值输出是针对 z 值进行训练的， z 值是从当前玩家的角度来看自我博弈的结果。换句话说，如果当前玩家最终赢得了采样

位置的游戏，则训练的 z 值为 +1，如果它输掉了游戏，则 z 值为 -1。平局得分为 0。如果此描述令人困惑，下图可能会有所帮助：



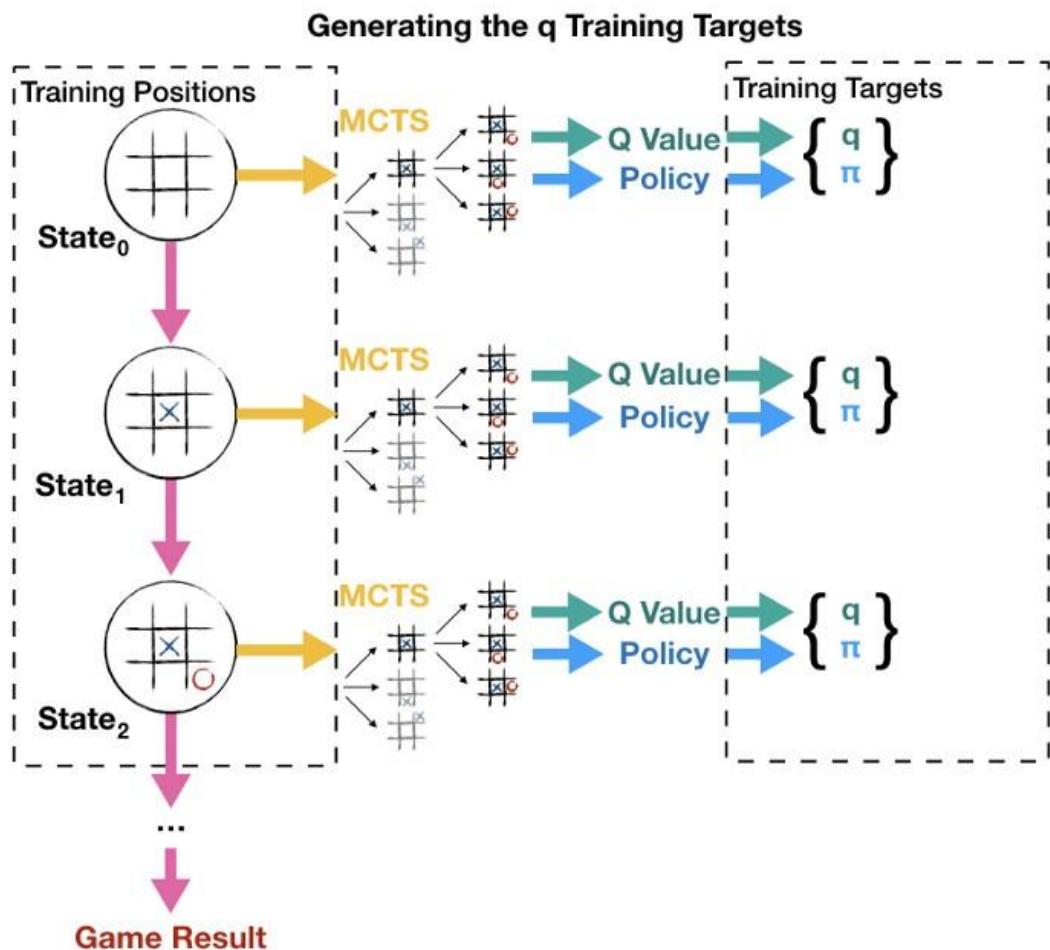
让我们解开在 AlphaZero self-play 过程中使用 z 生成训练目标的过程。从空棋盘开始（玩家 1 移动），游戏按如下方式进行：

1. 从当前状态运行 800 次模拟。
2. 根据当前状态的孩子的访问计数生成策略。
3. 根据策略概率性地打出一招，产生一个新的状态。
4. 对对方执行以上两步。
5. 重复直到游戏结束。游戏结果在 $\{-1, 0, 1\}$ 中。

在游戏结束时，每个状态都被标记为策略和游戏结果（注意否定玩家 2 移动位置的值）。神经网络的两个输出是针对这个策略 (π) 和游戏结果 (z) 进行训练的。如果 MCTS 模拟部分还是模糊不清，这里有很详细的解释。

替代培训目标

还有另一个用于训练价值输出的潜在目标。在 MCTS 搜索过程中，每个节点通过备份步骤累积游戏的预期结果。这个值被称为节点的**Q**值，简单来说就是 W / N ，其中 W 是在模拟过程中沿树向上传播的总分， N 是对节点的访问次数。搜索树的根节点代表游戏中的当前位置，因此它的**Q**值代表从该位置开始的游戏预期结果。当我们根据访问次数保存 π 值时，我们也可以将这个根节点的**Q**值保存为 q 并针对 q 而不是 z 训练网络。使用 q 生成训练目标的过程如下所示：



生成这个训练目标的self-play步骤与上面列出的五个步骤完全相同。唯一的区别在于游戏结束时的标记过程。每个状态都没有使用游戏结果，而是使用策略和根节点的 **Q** 值进行标记。神经网络的两个输出是针对这个策略 (π) 和 **Q** 值 (q) 进行训练的。

在使用深度学习和树搜索快速思考和慢速思考中使用了类似的目标，但有一个关键区别。他们的 MCTS 使用完全播放，而 AlphaZero 使用截断策略，直接从神经网络预测中备份值，而不是为每个模拟播放游戏。作者建议这个训练目标是 z 的一个很好的代理，但需要一个较小的数据集才能显着。

z 和 **q** 的区别

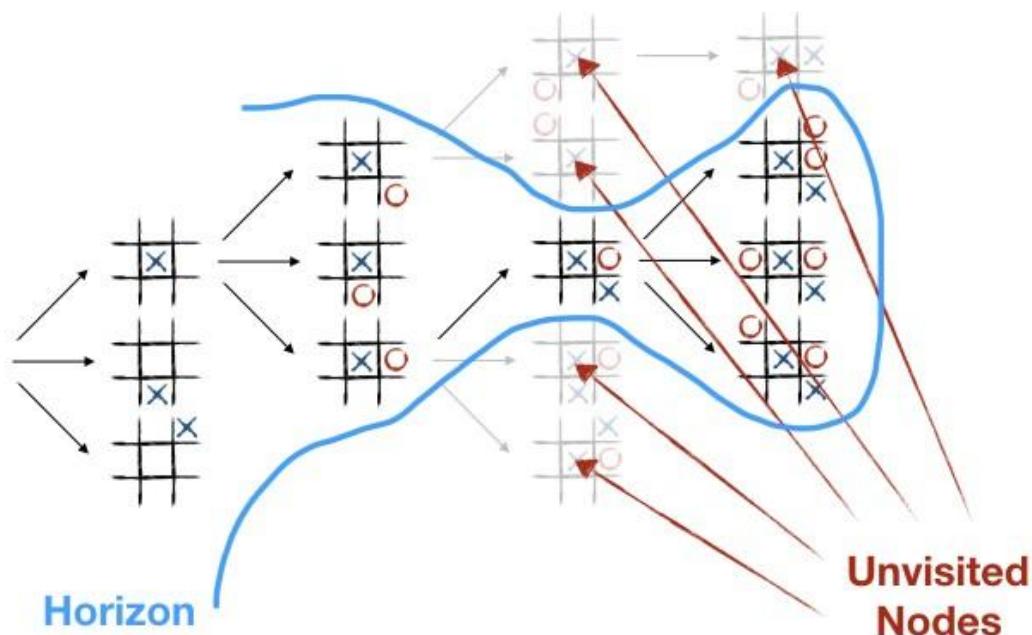
在使用 **z** 或 **q** 进行训练时，我们试图将从 MCTS 学习到的信息编码到网络中，但这两种方法之间存在有意义的差异。针对 **z** 的训练试图通过将比赛中的所有模拟压缩为单个离散值来编码游戏的预期结果：赢、输或平。相反，针对 **q** 的训练尝试仅使用来自当前位置的 800 次模拟将游戏的预期结果编码为连续值。

基于这些描述，**z** 似乎更胜一筹，但使用 **z** 有一个很大的缺点：每个游戏只有一个结果，并且该单个结果可能会受到随机性的严重影响。例如：想象一下游戏早期的一个位置，网络已经做出了正确的举动，但后来最终选择了次优的举动并输掉了比赛。在这种情况下，**z** 将为 -1，并且训练会错误地将低分与该位置相关联。

有了足够的训练数据，人们会希望这些错误会被正确的游戏所掩盖。不幸的是，完全消除错误是不可能的，因为由于其概率策略，网络在自我博弈中进行探索。我们推测这是 30 步后降温对 AlphaZero 如此重要的原因之一。否则，游戏结束时移动选择的随机性可能会影响 **z** 的准确性。

针对 **q** 的训练不会受到随机性问题的影响。网络最终输掉比赛并不重要。如果模拟为移动提供了良好的结果，则网络将针对正值进行训练。从某种意义上说，您可以将 **q** 视为 800 次自我游戏的平均值，而不是 1 次。这些“游戏”是网络的猜测，因此它们不如 800 个真实 **z** 值准确，但它们可以通过更小的自我游戏集实现比 **z** 值更高的一致性。

The Horizon Effect



不幸的是，**q**不是一个完美的解决方案。它可能会受到称为“地平线效应”的影响。当模拟返回一个肯定的结果时会发生这种情况，但是在搜索范围之外有一个杀手响应，即在 800 次模拟期间没有访问过。此外，**q**对于前几代训练中的早期移动有些意义，因为网络不知道如何评估位置。在这种情况下，**q**保持接近 0，并且可能需要很多代才能使价值输出变得重要。

测试新目标

我们从上述分析得出的结论是，**z**和**q**都有可能发挥作用，但每个都可能存在独特的缺点。根据经验，我们可以使用**q**而不是**z**成功训练网络。事实上，对于像 Connect Four 这样的短游戏，针对**q**的训练效果略好于针对**z**的训练。但是，我们可以利用我们的理解来显着改进 AlphaZero 的方法吗？

我们的实验表明，同时使用**q**和**z**可以获得更好的结果。组合**q**和**z**的一种方法是对每个示例位置进行平均，并使用该平均值来训练网络。这似乎带来了两者的好处：**z**有助于抵消**q**的水平效应，而**q**有助于抵消**z**的随机性。另一种有前途的方法是从对**z**进行训练开始，但在几代后线性衰减到**q**。

在训练 Connect Four 的早期周期中，平均和衰减都表现得一样好。最终，衰减方法能够实现比平均方法略低的错误百分比。与单独使用**z**或**q**相比，这两种方法的训练速度都有显著提高。您可以在下面看到我们的前 20 代连接四模型使用四个目标中的每一个实现的错误百分比图表。在线性衰减情况下，我们在第一代开始使用 100% **z**，到第20 代过渡到 100% **q**。



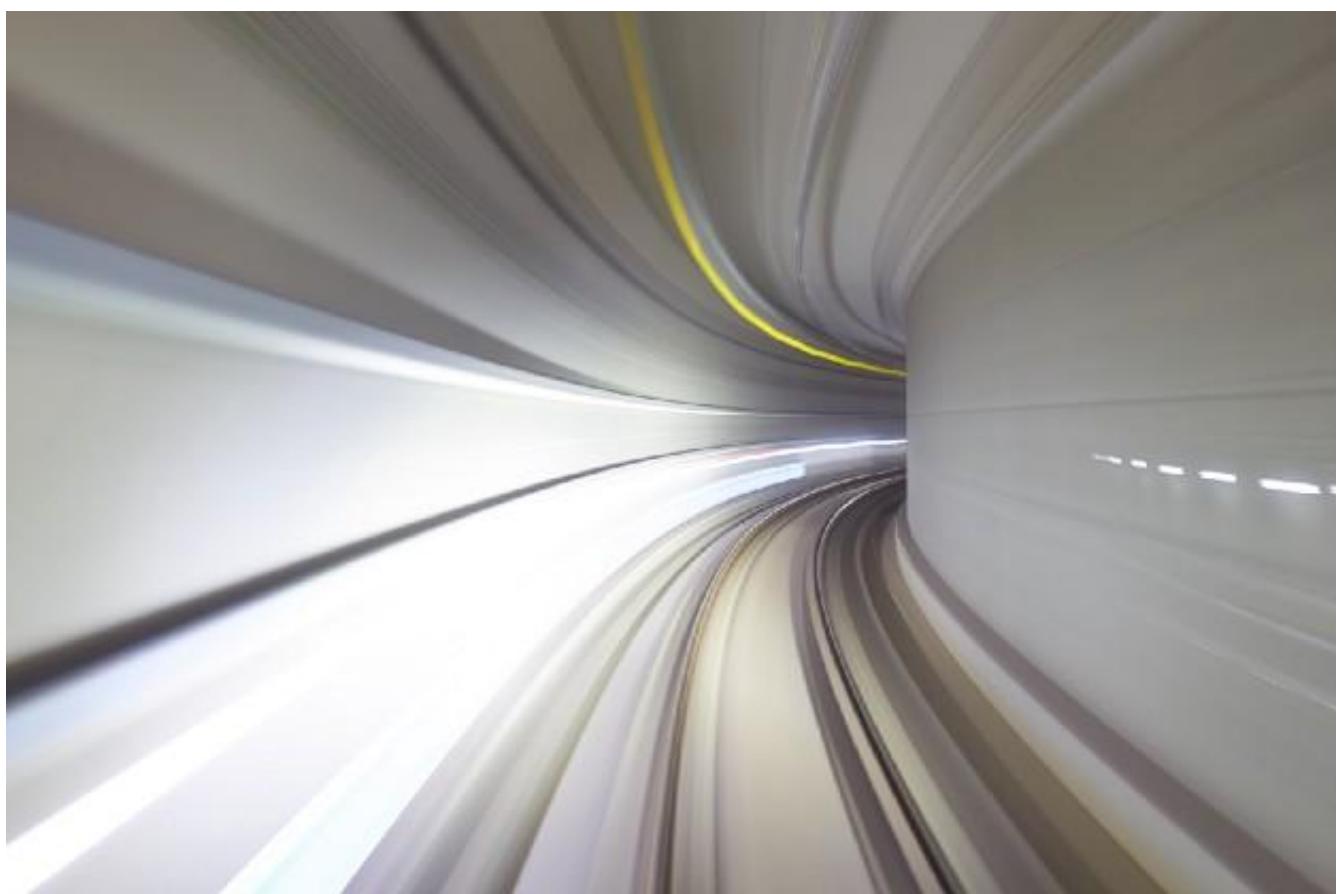
Alpha Zero 的经验教训（第 5 部分）：性能优化



安东尼杨

[跟随](#)

2018 年 7 月 4 日 · 8 分钟阅读



照片由Mathew Schwartz在Unsplash上拍摄

这是我们关于实施 *AlphaZero* 的经验教训系列的第五部分。查看[第 1 部分](#)、[第 2 部分](#)、[第 3 部分](#)和[第 4 部分](#)。

在这篇文章中，我们回顾了 *AlphaZero* 实施的各个方面，这些方面使我们能够显着提高游戏生成和训练的速度。

概述

实现 AlphaZero 的任务是艰巨的，不仅因为算法本身很复杂，还因为作者用于研究的大量资源：在数小时内使用了 5000 个 TPU 来训练他们的算法，那就是大概是在花费大量时间确定最佳参数以使其能够快速训练之后。

通过选择 Connect Four 作为我们的第一款游戏，我们希望在利用更适度资源的同时实现 AlphaZero 的可靠实现。但在开始后不久，我们意识到即使是像 Connect Four 这样的简单游戏也可能需要大量资源来训练：在我们最初的实施中，在一台支持 GPU 的计算机上训练需要数周时间。

幸运的是，我们能够进行一些改进，使我们的训练周期时间从几周缩短到大约一天。在这篇文章中，我将介绍一些我们最有影响力的变化。

瓶颈

在深入探讨我们为减少 AZ 训练时间所做的一些调整之前，让我们描述一下我们的训练周期。尽管 AlphaZero 的作者使用连续和异步的过程来执行模型训练和更新，但在我们的实验中，我们使用了以下三个阶段的同步过程，我们选择它是因为它的简单性和可调试性：

虽然（我的模型不够好）：

1. 生成游戏：每个模型周期，使用最新模型，游戏代理生成 7168 个游戏，相当于大约 140-220K 游戏位置。
2. 训练新模型：基于窗口算法，我们从历史数据中采样并训练改进的神经网络。
3. 部署新模型：我们现在采用新模型，将其转换为可部署的格式，并将其推送到我们的云中以进行下一个训练周期

毫无疑问，这个过程的最大瓶颈是游戏生成，当我们刚开始时，每个周期需要一个多小时。正因为如此，最小化游戏生成时间成为我们关注的焦点。

型号尺寸

Alpha Zero 在自我游戏中非常依赖推理。事实上，在我们的一个典型游戏生成周期中，MCTS 需要超过 1.2 亿次位置评估。根据模型的大小，这可以转化为显着的 GPU 时间。

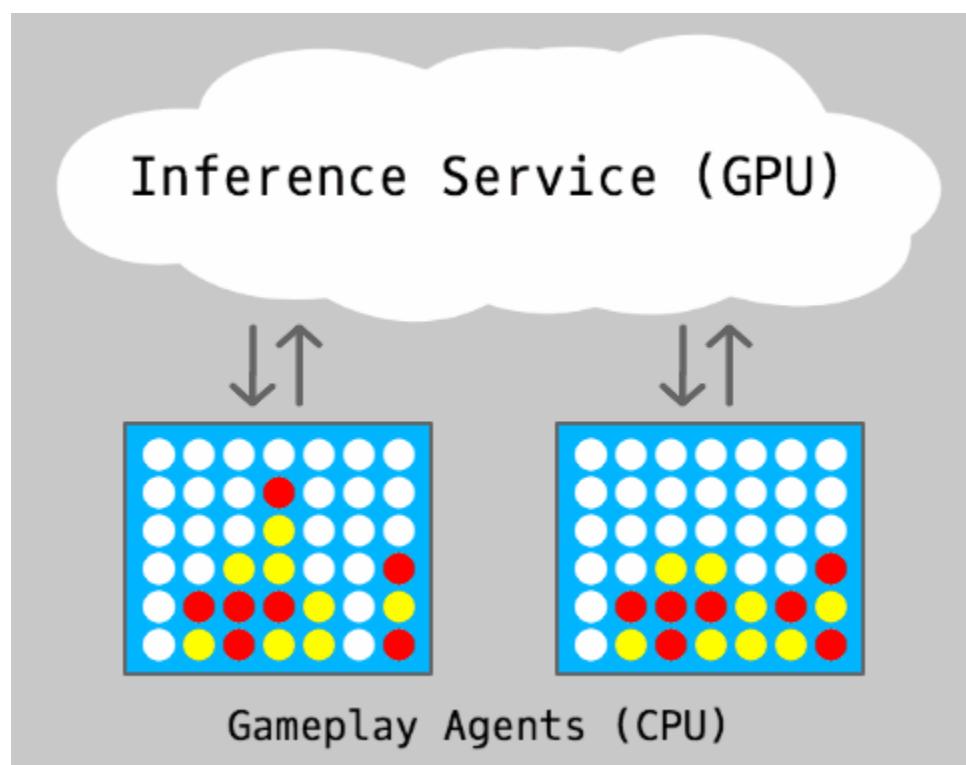
在 AlphaZero 的最初实现中，作者使用了一种架构，其中大部分计算在 20 个残差层中执行，每个残差层有 256 个过滤器。这相当于一个超过 90 兆字节的模型，这对于 Connect Four 来说似乎是多余的。此外，考虑到我们最初有限的 GPU 资源，使用这种大小的模型是不切实际的。

相反，我们从一个非常小的模型开始，只使用 5 层和 64 个过滤器，只是为了看看我们是否可以让我们的实现学到任何东西。随着我们继续优化我们的流程并改进我们的结果，我们能够将模型大小提高到 20X128，同时仍然在我们的硬件上保持合理的游戏生成速度。

	AZ Orig	Initial	Final
Res Layers	20	5	20
Res Filters	256	64	128
Model Size	95MB	386KB	24MB
Param Count	~24,000,000	386,403	6,290,312

分布式推理

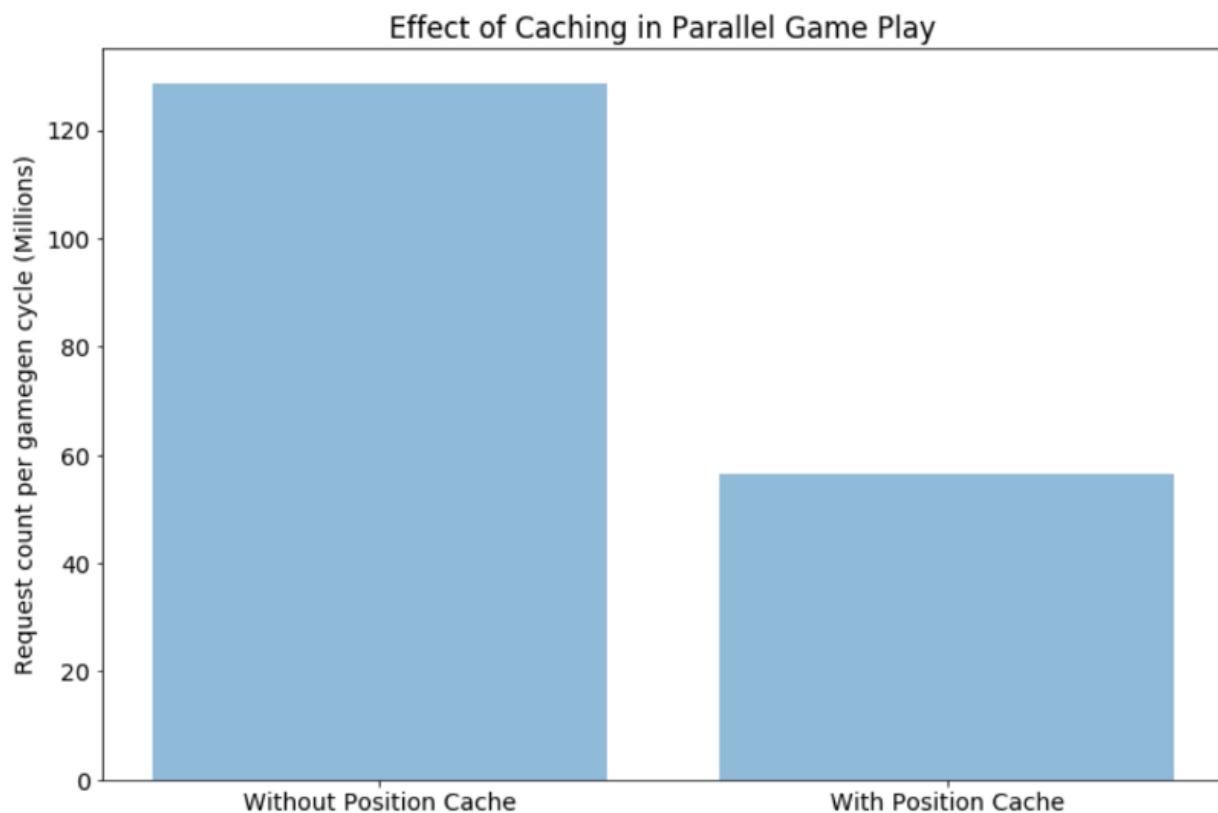
从一开始，我们就知道我们需要不止一个 GPU 才能实现我们所寻求的训练周期时间，因此我们创建了允许我们的 Connect 4 游戏代理执行远程推理以评估位置的软件。这使我们能够将 GPU 密集型推理资源与仅需要 CPU 的游戏资源分开扩展。



并行游戏生成

GPU 资源很昂贵，因此我们希望确保在播出期间尽可能地使它们饱和。事实证明这比我们想象的要棘手。

我们首先进行的优化之一是在同一进程的并行线程上运行许多游戏。也许最大的直接好处是它允许我们缓存位置评估，这可以在不同的线程之间共享。这将发送到我们的远程推理服务器的请求数量减少了 2 倍以上：



缓存是一个巨大的胜利，但我们仍然希望以有效的方式处理剩余的未缓存请求。为了最大限度地减少网络延迟并最好地利用 GPU 并行化，我们将来自不同工作线程的推理请求组合到一个存储桶中，然后再将它们发送到我们的推理服务。这样做的缺点是，如果一个桶没有被及时填满，任何调用线程都会被卡住，直到桶的超时到期。在这种方案下，选择合适的推理桶大小和超时值非常重要。

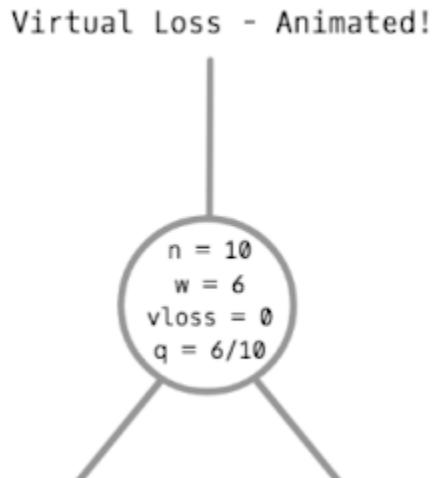
我们发现桶填充率在游戏生成批次的整个过程中会发生变化，主要是因为一些游戏会比其他游戏更快地完成，留下越来越少的线程来填充桶。这导致一批游戏的最后一个游戏需要很长时间才能完成，而 GPU 利用率却降至零。我们需要一种更好的方法来保持我们的水桶装满。

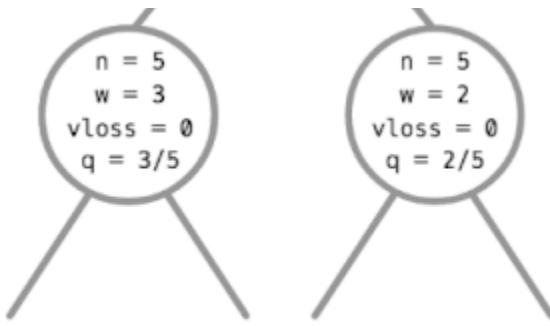
并行 MCTS

为了帮助解决未填充的桶问题，我们实施了 Parallel MCTS，这在 AZ 论文中进行了讨论。最初我们在这个细节上下注，因为它似乎对竞争性的一对一游戏非常重要，而并行游戏不适用。在遇到前面提到的问题后，我们决定试一试。

Parallel MCTS 背后的想法是允许多个线程承担累积树统计数据的工作。虽然这听起来很简单，但这种幼稚的方法存在一个基本问题：如果 N 个线程都同时启动并根据当前树的统计信息选择一条路径，它们都会选择完全相同的路径，从而削弱 MCTS 的探索组件。

为了解决这个问题，AlphaZero 使用了 Virtual Loss 的概念，这是一种临时将游戏损失添加到模拟期间遍历的任何节点的算法。锁用于防止多个线程同时修改节点的模拟和虚拟损失统计。在访问了一个节点并应用了虚拟损失之后，当下一个线程访问同一个节点时，将不鼓励它遵循相同的路径。一旦线程到达终点并备份它的结果，这个虚拟损失就会被移除，从模拟中恢复真实的统计数据。





有了虚拟损失，我们终于能够在大部分游戏生成周期中实现超过 95% 的 GPU 利用率，这表明我们正在接近硬件设置的真正极限。

从技术上讲，虚拟损失在游戏播放中增加了一定程度的探索，因为它迫使选择沿着 MCTS 可能不会自然倾向于访问的路径移动，但我们从未测量过由于它的使用而产生的任何有害（或有益）影响。

张量RT/张量RT+INT8

尽管没有必要使用像 AlphaZero 论文中描述的那样大的模型，但我们看到从更大的模型中学习效果更好，因此希望尽可能使用最大的模型。为了解决这个问题，我们尝试了 TensorRT，这是 Nvidia 创建的一项用于优化模型推理性能的技术。

只需几个脚本即可轻松将现有的 Tensorflow/Keras 模型转换为 TensorRT。不幸的是，在我们做这个的时候，还没有发布 TensorRT 远程服务组件，所以我们自己写了。

使用 TensorRT 的默认配置，我们注意到推理吞吐量略有增加 (~11%)。我们对这种适度的改进感到高兴，但希望通过使用 TensorRT 的 INT8 模式看到更大的性能提升。INT8 模式需要更多的努力才能开始，因为在使用 INT8 时，您必须首先生成一个校准文件，以告诉推理引擎在使用 8 位近似数学时将哪些比例因子应用于您的层激活。此校准是通过将您的数据样本输入 Nvidia 的校准库来完成的。

因为我们观察到校准运行质量的一些变化，我们将尝试对 3 组不同的样本数据进行校准，然后根据保留数据验证生成的配置。在三种校准尝试中，我们选择了验证误差最低的一种。



一旦我们的 INT8 实施到位，我们看到推理吞吐量与库存 libtensorflow 相比增加了近 4 倍，这使我们能够使用比其他方式可行的更大的模型。



使用 INT8 的一个缺点是它在某些情况下可能有损且不精确。虽然我们在训练的早期阶段没有观察到严重的精度问题，但随着学习的进展，我们会观察到推理的质量开始下降，尤其是在我们的价值输出方面。这最初导致我们仅在训练的早期阶段使用 INT8。

巧合的是，当我们开始尝试增加头部网络中卷积滤波器的数量时，我们几乎可以消除 INT8 精度问题，这是我们从 [Leela Chess](#) 那里得到的想法。下面是我们的值输出的平均误差图表，在值头中有 32 个过滤器，而 AZ 默认值为 1：



我们推测，向这些层添加额外的基数会减少激活的方差，从而使模型更容易准确量化。这些天来，我们总是在启用 INT8 的情况下执行我们的游戏生成，并且即使在 AZ 训练结束时也没有看到任何不良影响。

概括

通过使用所有这些方法，我们终于能够训练出具有高 GPU 利用率和良好周期时间的体面模型。最初看起来需要数周才能完成一次完整的训练，但现在我们可以在不到一天的时间内训练出一个体面的模型。这很棒，但事实证明我们才刚刚开始——在下一篇文章中，我们将讨论如何调整 AlphaZero 本身以获得更好的学习速度。

第 6 部分 现已出炉。

感谢Vish (Ishaya) Abrams和Aditya Prasad。

Alpha Zero 的经验教训（第 6 部分）——超参数调优



安东尼杨

跟随



2018 年 7 月 12 日 · 6 分钟阅读



丹尼斯莱昂在Unsplash上的照片

这是我们关于实施 *AlphaZero* 的经验教训系列的第六部分。查看[第 1 部分](#)、[第 2 部分](#)、[第 3 部分](#)、[第 4 部分](#)和[第 5 部分](#)。

在这篇文章中，我们总结了使用 Connect Four 为我们提供了最佳训练性能的配置和超参数选择。

概述

在我们实现 AlphaZero 的过程中，我们花了一些时间才意识到这个算法有多么挑剔，因为即使你没有所有的超参数，它仍然可以学习，尽管速度很慢。此外，还有很多超参数，其中许多在 AZ 论文中没有完全解释。

由于我们的性能优化工作，我们能够比刚开始时更快地生成游戏。但是仍然需要许多模型代来创建一个近乎完美的四人连接播放器。这是与我们原始配置类似的运行图。



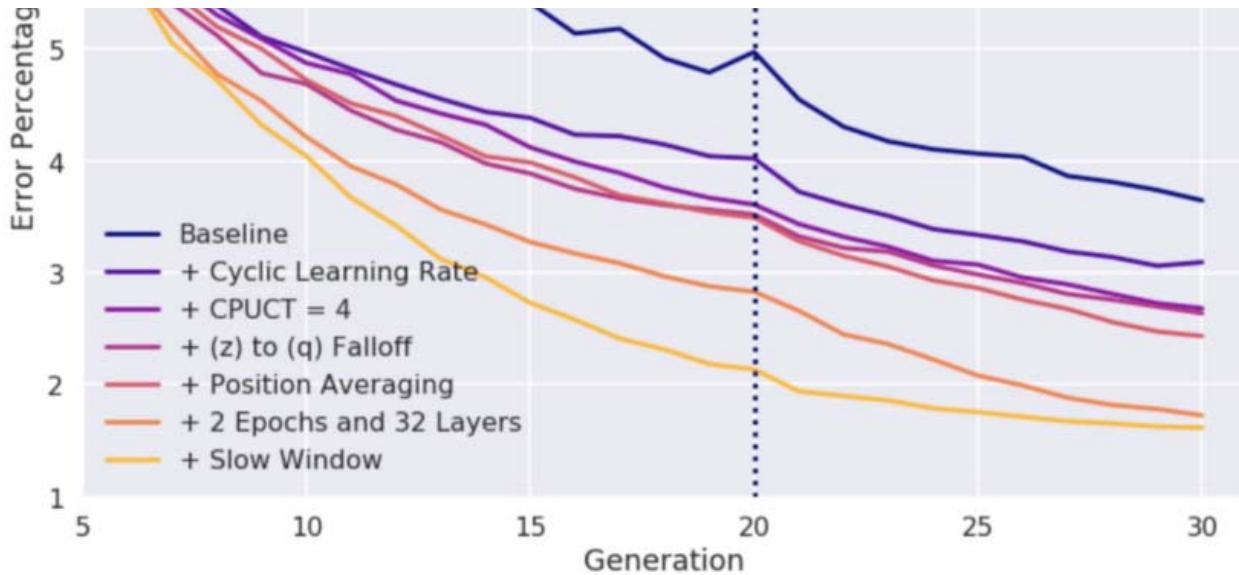
大约 150 代模型，每代 12 分钟，我们花了一天多的时间来执行上述运行。

在上一篇文章中，我们研究了加快训练周期的方法。现在，我们将研究实际减少获得专家算法所需的训练周期数的方法。

我们的改进

在深入了解我们每项改进的细节之前，让我们看一下总结我们所有调整的图表：

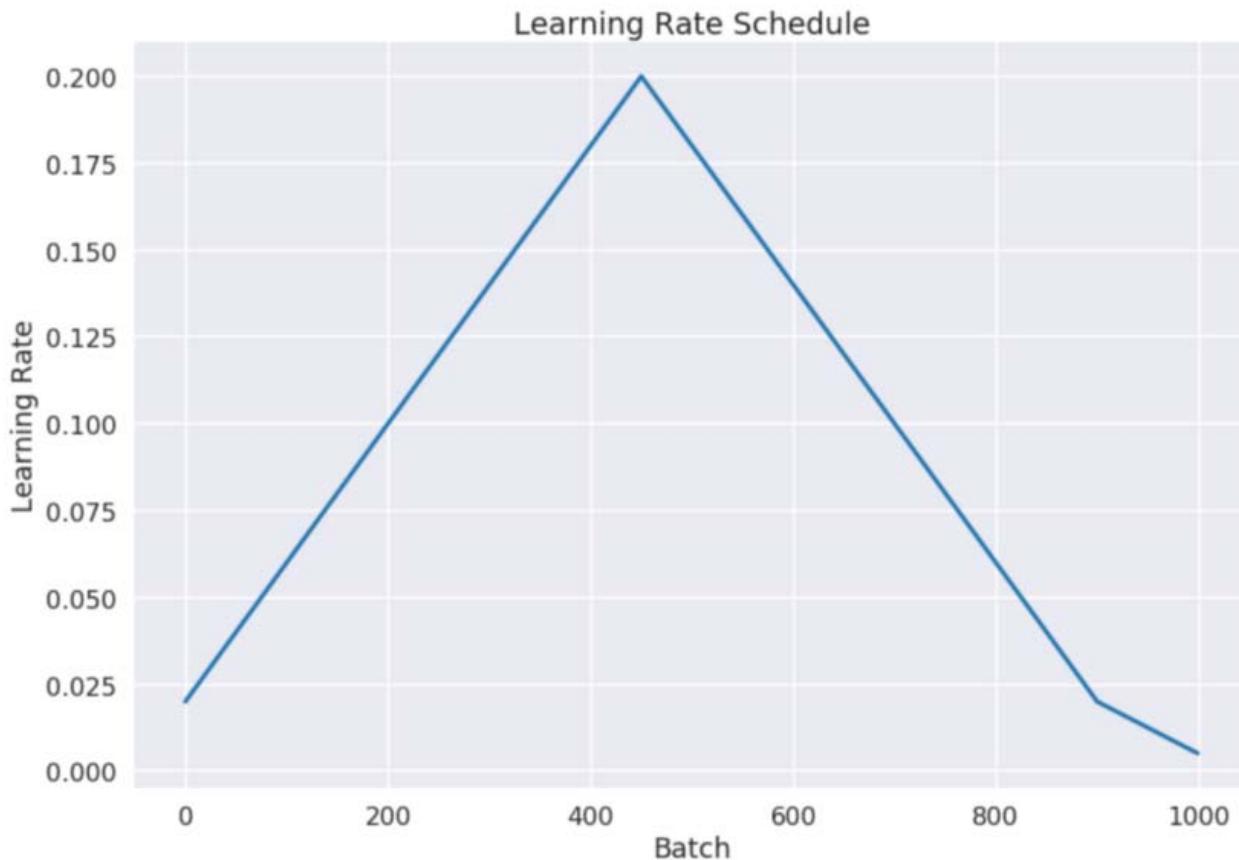




没有单一的灵丹妙药，而是参数调整的组合导致了我们的训练加速。

周期性学习率

在 AlphaZero 中，作者使用他们定期调整的固定学习率来训练他们的网络。虽然我们从这种方法开始，但我们最终实施了一个 1 周期学习率计划，如此处所述。这背后的想法是，我们不是为训练选择一个单一的学习率，而是随着训练的进行显式地上下改变学习率。我们尝试了一些循环调度，例如重新启动的余弦，但发现 1cycle 是我们尝试过的最好的方法。



基础学习率为 0.02 的循环时间表

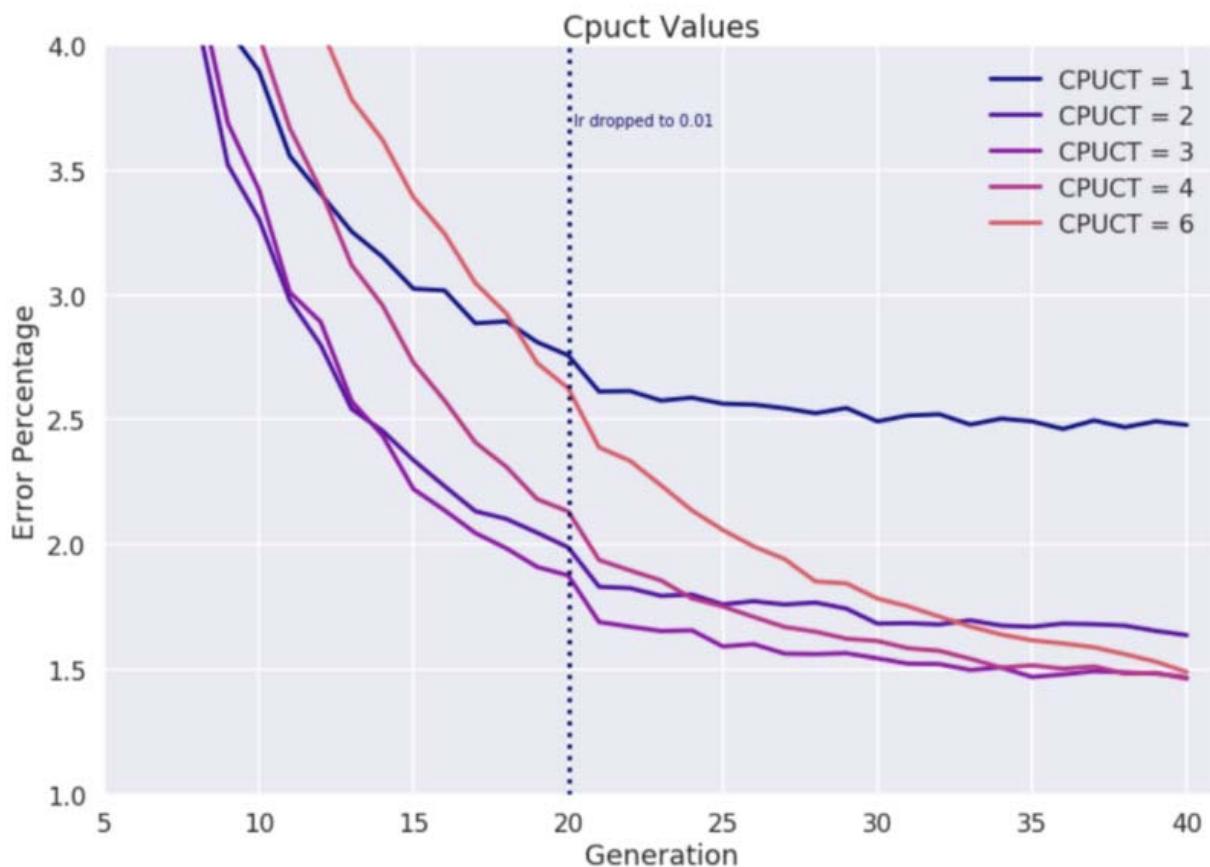
尽管 1cycle 的计划通常应用于多个 epoch，但我们发现即使在单个 epoch 中调整批次时它也很有帮助。

随着训练的进行，我们确实降低了基础学习率，但发现我们不必像没有 1cycle 时那样精确或频繁地这样做。

C-PUCT

在播放期间，MCTS 使用 PUCT（UCT 的一种变体）来平衡探索与利用。有关该算法如何工作的详细信息，请参阅我们之前关于此主题的帖子。

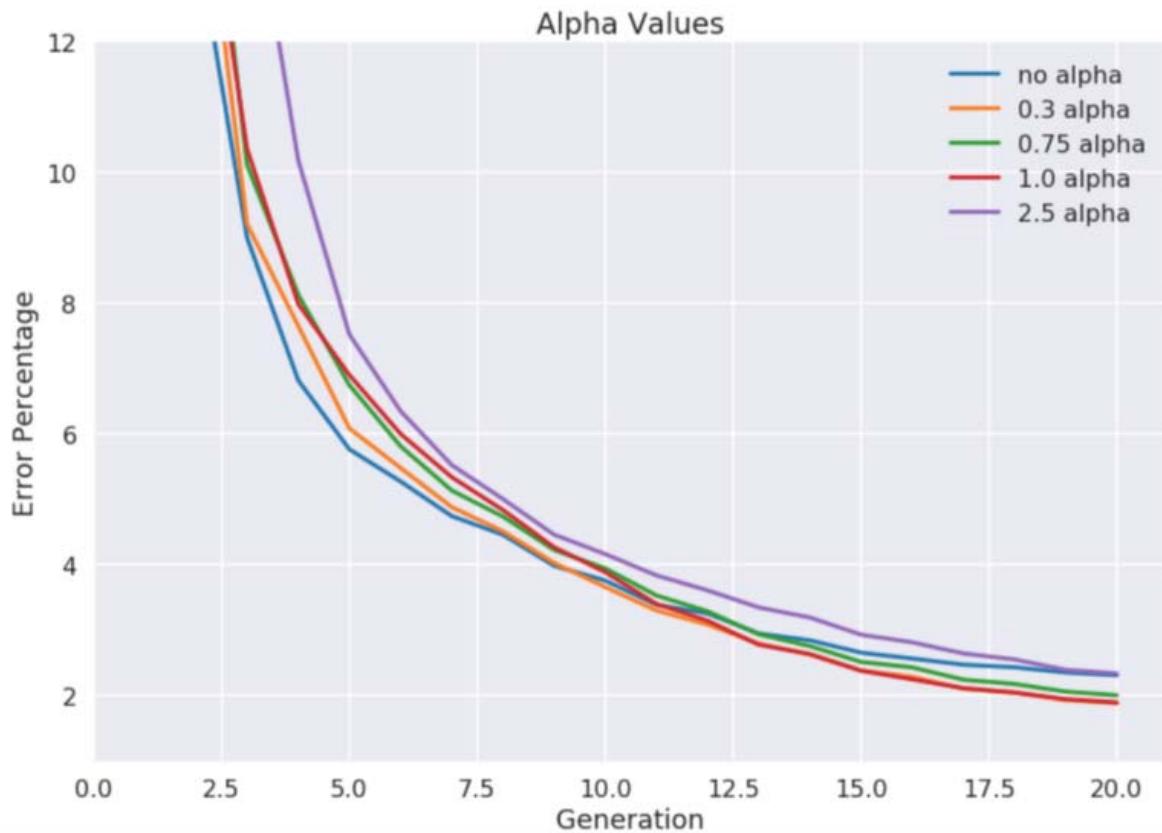
在我们的实验过程中，我们尝试了 C-PUCT 的各种值，但最终发现 c 在 3-4 的范围内是一个最佳点。



A

使用 Dirichlet 噪声是 AlphaGo 的一项有趣创新，它在游戏进行期间向根节点的先验添加噪声。这种噪音在推荐的配置中往往是尖峰的，产生的噪音将探索从根节点集中到离策略路径的一小部分，这一属性对于鼓励在具有高分支因子的游戏中进行集中探索可能特别有用。要了解有关 Dirichlet 噪声和 alpha 的更多信息，请参阅我们之前关于该主题的帖子。

下面是各种 alpha 值的学习曲线图。我们发现 1 的 alpha 在我们的测试中表现最好。



位置平均

在我们的实验过程中，我们开始跟踪训练集中存在的独特位置的数量，以此来监控 AlphaZero 的各种探索参数的效果。例如，当 $C=1$ 时，我们会在生成的位置中观察到大量冗余，这表明该算法正在选择高频相同的路径，并且可能探索不够。在 $C=4$ 时，重复位置的数量较低。一般来说，使用 Connect Four，在您的训练窗口中出现约 30–50% 的重复数据并不罕见。

当在您的训练窗口中发现重复位置时，它们可能来自不同的模型生成，这意味着它们相关的先验和值可能不同。通过将这些具有不同目标的位置呈现给神经网络，我们实际上是在要求它为我们平均目标值。

我们没有要求我们的网络自行对数据进行平均，而是在将数据呈现给网络之前尝试对数据执行重复数据删除和平均。从理论上讲，这为网络创造了更少的工作，因为它不必自己学习这个平均值。此外，重复数据删除允许我们在每个训练周期向网络呈现更多独特的位置。

额外的最后一层过滤器

在 AlphaZero 论文中，神经网络获取游戏输入，然后通过 20 个残差卷积层运行它。然后将这些残差卷积层的输出馈入卷积策略和值头，它们分别具有 2 个和 1 个过滤

器。

我们最初按照论文中的描述实现了模型及其头部网络。根据Leela Chess报告的发现，我们将头部网络中的过滤器数量增加到 32 个，这显着加快了训练速度。

添加额外的头部过滤器具有意想不到的副作用，即减少了我们在 INT8 训练期间的精度误差，这使我们能够在整个训练周期中使用 TensorRT+INT8。更多关于这个[here](#)。

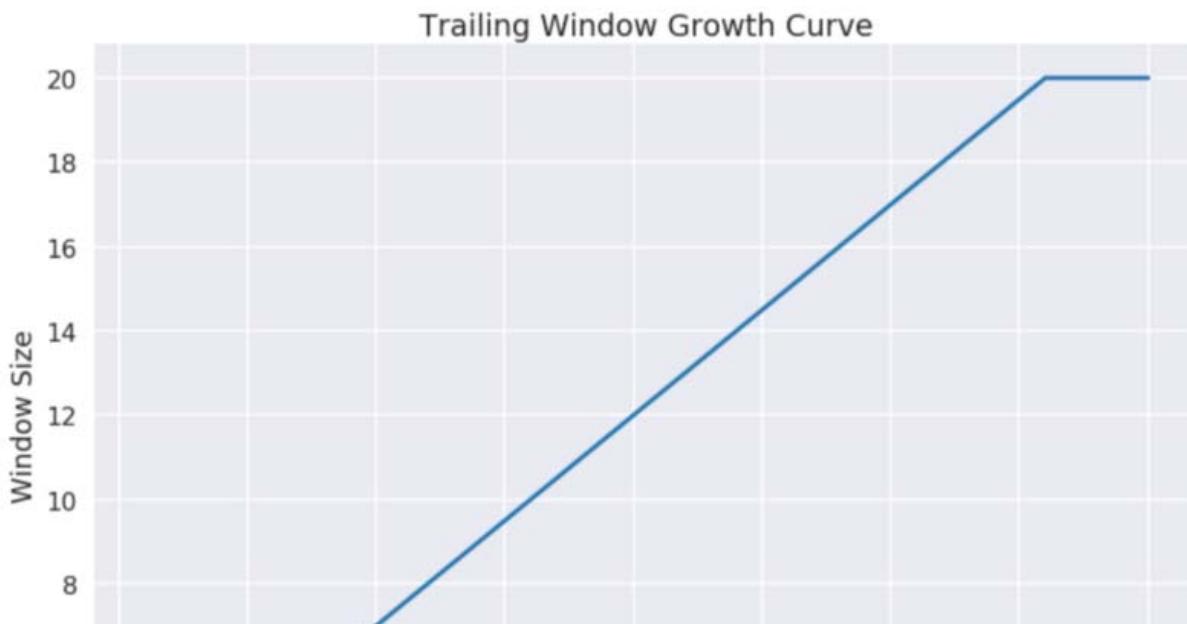
每代多个时期

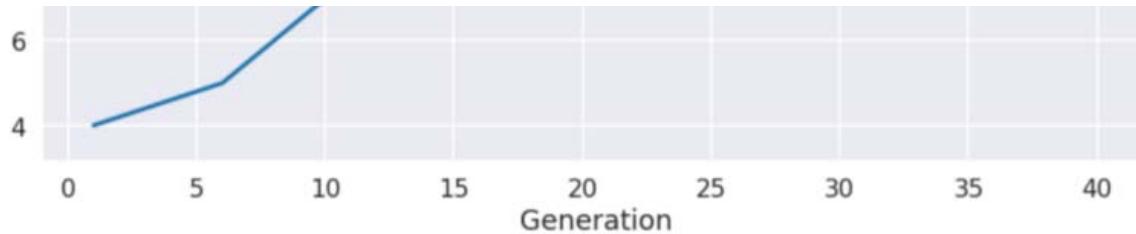
在游戏生成之后，然后进行训练，以便我们的模型可以从最近生成的数据中学习，以便在下一个周期中创建更精细的游戏示例。我们发现，每个窗口样本使用 2 个 epoch 的训练在单 epoch 训练的学习上提供了一个很好的提升，而不会让我们的同步训练周期成为瓶颈太长。

慢窗口

在 AlphaZero 中，作者使用了一个大小为 500,000 个游戏的滑动窗口，他们从中统一采样训练数据。在我们的第一个实现中，我们使用了一个由 20 代数据组成的滑动训练窗口，总计 143360 场比赛。在我们的实验中，我们注意到在模型 21 中，训练误差会大幅下降，评估性能会出现显着提升，正如可用数据量超过训练窗口大小并且旧数据开始被删除一样。这似乎意味着旧的、不太精确的数据可能会阻碍学习。

为了解决这个问题，我们实现了一个缓慢增加的采样窗口，窗口的大小开始时很小，然后随着模型生成次数的增加而缓慢增加。这使我们能够在确定固定窗口大小之前快速淘汰非常早期的数据。我们从 4 的窗口大小开始，因此通过模型 5，第一代（也是最差的）数据被逐步淘汰。然后，我们将历史大小每两个模型增加一个，直到在第 35 代时达到完整的 20 个模型历史大小。

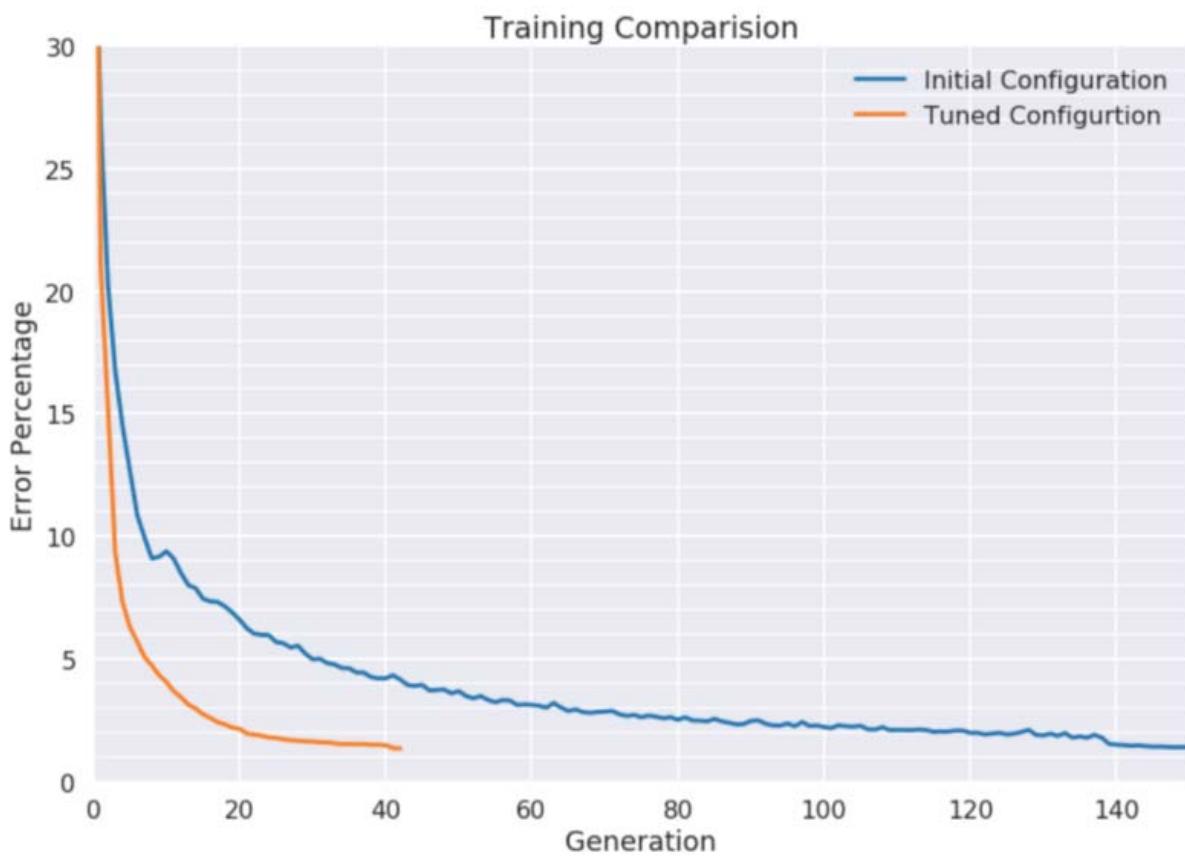




实现类似效果的另一种方法是改变我们的采样分布，尽管我们选择上述方法是为了简单。

把它放在一起

那么通过所有这些调整，我们能学多快？速度快了近 4 倍：虽然过去需要约 150 代来训练 Connect Four 玩家，但我们现在可以训练约 40 代模型。



对我们来说，这相当于从 77 个 GPU 小时减少到 21 个 GPU 小时。我们估计，如果没有此处或上一篇文章中提到的改进（例如 INT8、并行缓存等），我们最初的训练将花费超过 450 个 GPU 小时。

通过调整这些参数的练习，我们了解了超参数调整在 Alpha Zero 中的重要性。希望我们能够将其中的一些知识应用到更大的游戏中。

感谢阿迪亚普拉萨德。

蒂姆·惠勒

[关于](#) [档案](#) [博客](#)

←使用 C3.js 的 Apteryx 飞行数据

[我们如何编写教科书→](#)

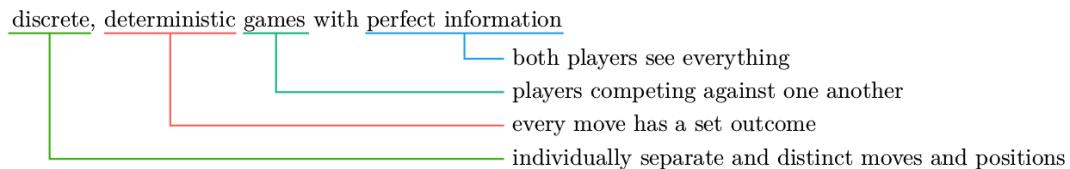
AlphaGo 零 - 如何以及为何工作

2017 年 11 月 2 日通过蒂姆

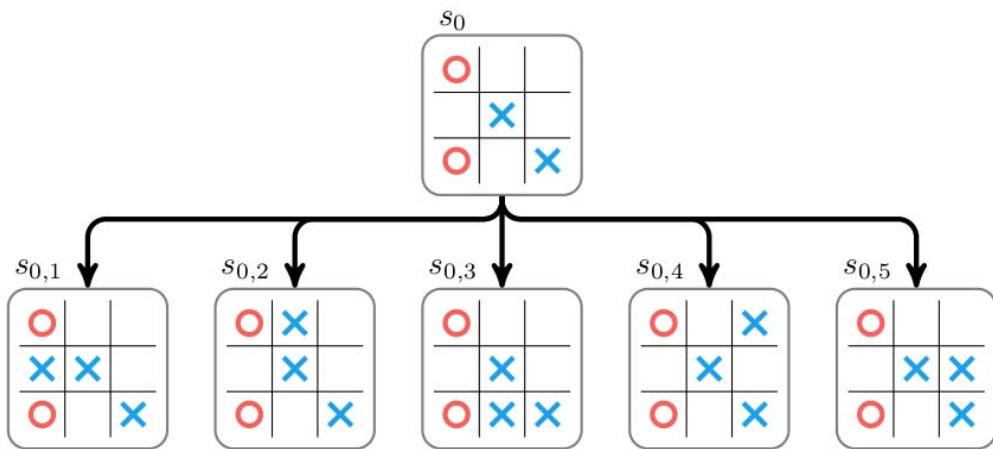
DeepMind 的 AlphaGo 在 2016 年 3 月成为第一个击败人类顶级围棋棋手的 AI 时引起了轰动。这个版本的 AlphaGo——AlphaGo Lee——在训练过程中使用了来自世界上最优秀棋手的大量围棋游戏。几天前发布了一篇新论文，详细介绍了一种新的神经网络——AlphaGo Zero——它不需要人类向它展示如何下围棋。它不仅超越了所有以前的围棋选手，无论是人类还是机器，而且仅在三天的训练时间后就达到了这一点。本文将解释它的工作原理和原因。

蒙特卡洛树搜索

编写机器人来玩具有完美信息的离散、确定性游戏的首选算法是蒙特卡洛树搜索 (MCTS)。玩围棋、国际象棋或跳棋等游戏的机器人可以通过尝试所有动作，然后检查对手所有可能的反应，之后所有可能的动作等来确定它应该采取什么行动。对于像围棋这样的游戏，尝试的动作增长得非常快。蒙特卡洛树搜索将根据它认为的好坏有选择地尝试移动，从而将精力集中在最有可能发生的移动上。



从技术上讲，该算法的工作原理如下。进行中的游戏处于初始状态 s_0 ，轮到机器人上场了。机器人可以从一组动作中进行选择——蒙特卡洛树搜索从一棵由单个节点组成的树开始 s_0 。通过尝试每个操作来扩展此节点一个，并为每个动作构造一个对应的子节点。下面我们展示了这款井字游戏的扩展：



然后必须确定每个新子节点的值。子节点中的游戏通过从子状态随机采取移动来展开，直到达到赢、输或平局。胜利得分为

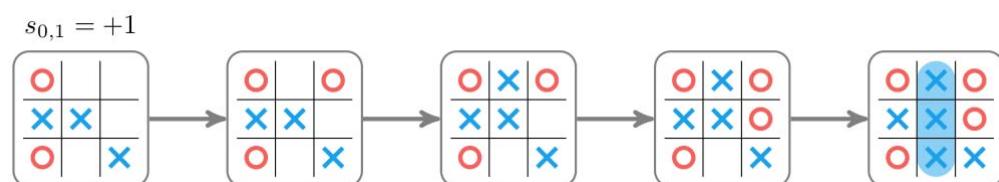
+ 1

，损失在

- 1

， 并且关系在

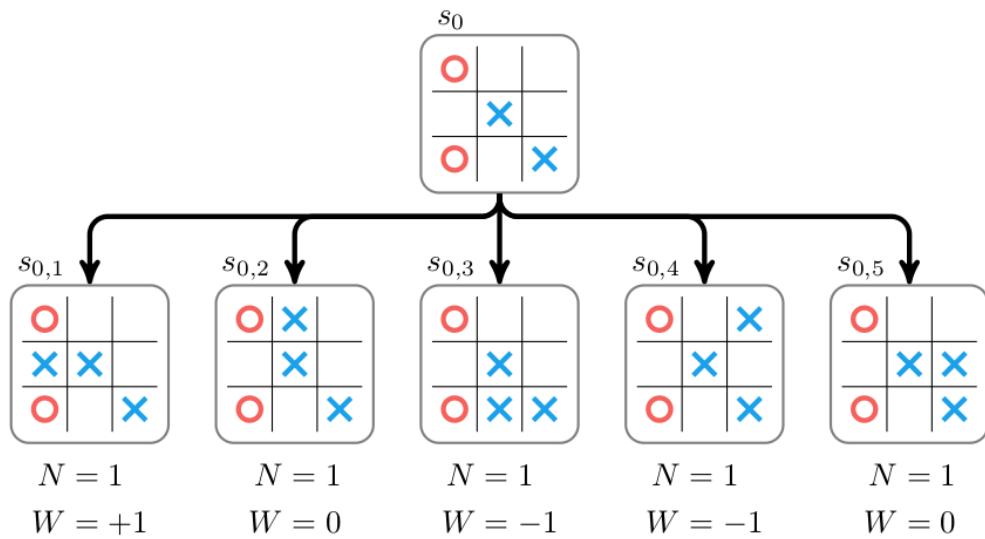
0



上面给出的第一个孩子的随机推出估计值为

+ 1

. 此值可能不代表最佳播放 - 它可能会根据部署的进展情况而有所不同。一个人可以不智能地进行部署，随机均匀地绘制移动。人们通常可以通过遵循一种更好的——尽管仍然是典型的随机策略，或者通过直接估计状态的价值来做得更好。稍后再谈。



上面我们显示了每个子节点具有近似值的扩展树。注意我们存储两个属性：累计值

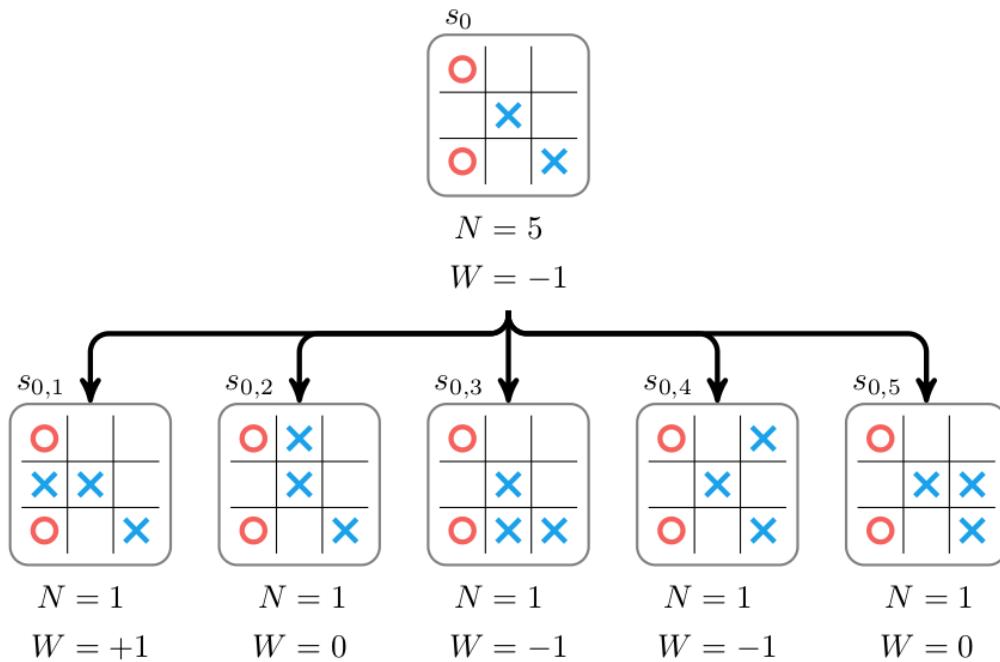
W

以及在该节点或该节点下运行推出的次数，

\tilde{n}

. 我们只访问了每个节点一次。

然后通过增加父节点的值和访问次数，将来自子节点的信息沿树向上传播。然后将其累积值设置为
其子项的总累积值：



蒙特卡洛树搜索继续进行多次迭代，包括选择一个节点、扩展它并传播新信息。扩展和传播已经涵
盖。

蒙特卡洛树搜索不会扩展所有叶节点，因为这会非常昂贵。相反，选择过程会选择在利润丰厚（具有高估计值）和相对未开发（具有低访问次数）之间取得平衡的节点。

通过从根节点向下遍历树来选择叶节点，始终选择子节点

一世

具有最高的上置信树 (UCT) 分数：

$$U_{\text{一世}} = \frac{W_{\text{一世}}}{\tilde{n}_{\text{一世}}} + c \sqrt{\frac{\ln \tilde{n}_p}{\tilde{n}_{\text{一世}}}}$$

在哪里

$W_{\text{一世}}$

是的累计值

一世

第一个孩子，

$\tilde{n}_{\text{一世}}$

是访问次数

一世

第一个孩子，和

\tilde{n}_p

是父节点的访问次数。参数

$c \geq 0$

控制选择有利可图的节点之间的权衡（低

C

) 并探索访问次数少的节点 (高

C

)。它通常是根据经验设置的。

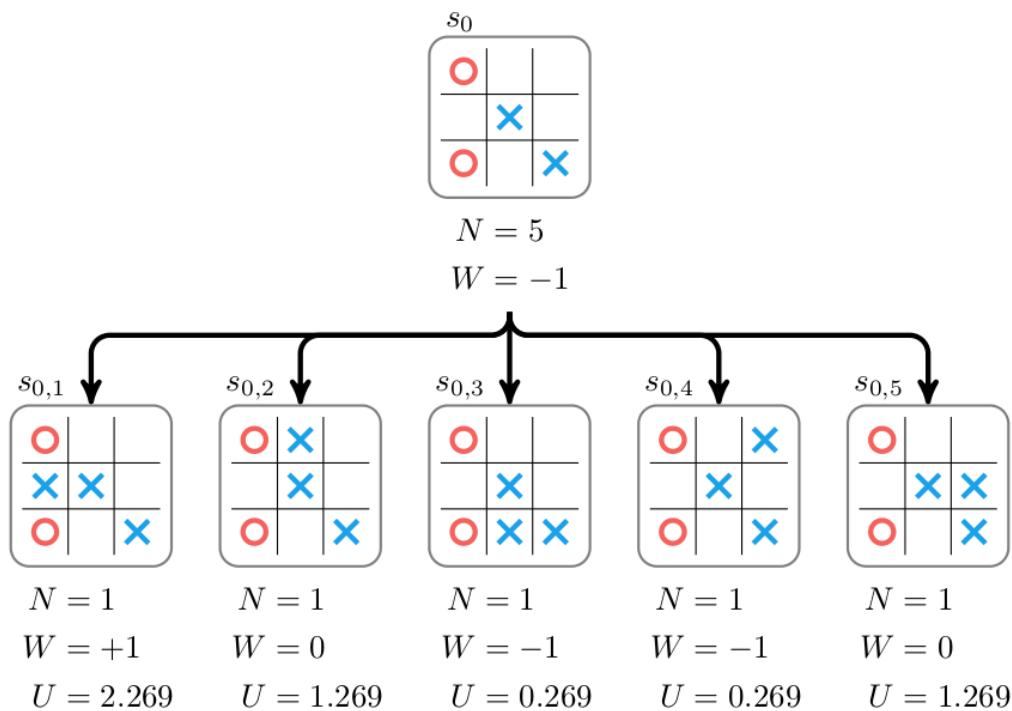
UCT 分数 (

ü

's) 用于井字游戏树

$$c = 1$$

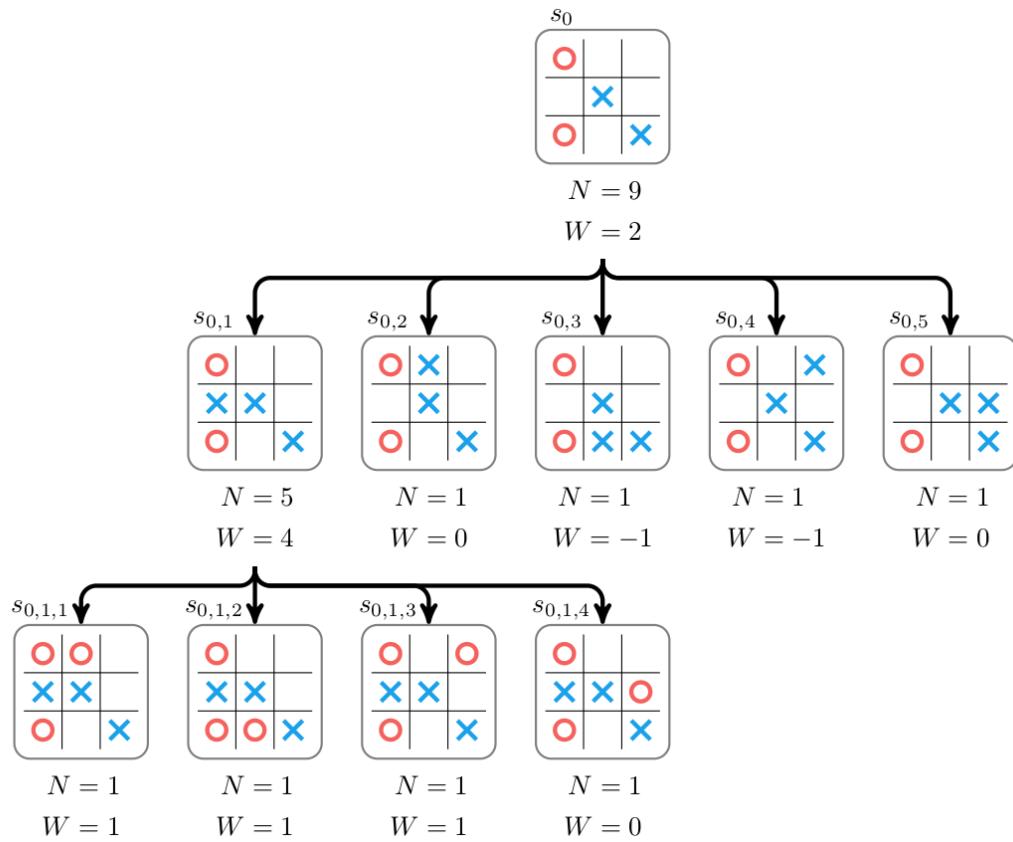
是：



在这种情况下，我们选择第一个节点，

$$s_{0,1}$$

. (如果出现平局，则可以随机打破平局，或者只选择第一个下注节点。) 该节点被扩展并且值被传播回来：



请注意，每个累积值

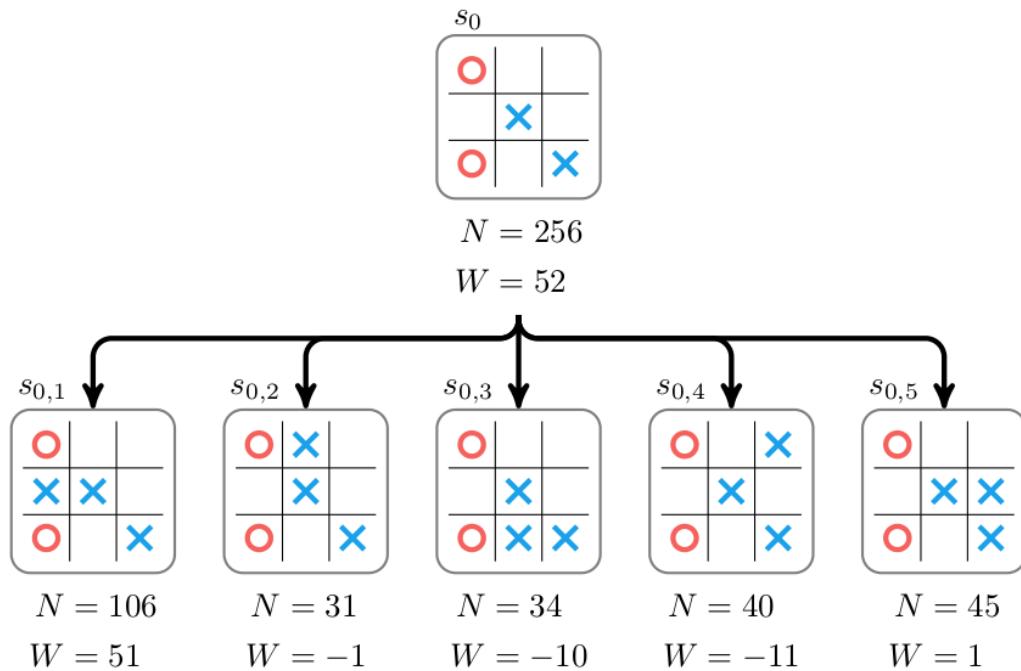
W

反映 X 的输赢。在选择过程中，我们跟踪是轮到 X 还是 O 移动，并翻转

W

每当轮到O时。

我们继续运行蒙特卡洛树搜索的迭代，直到时间用完。树逐渐扩展，我们（希望）探索可能的移动，确定最佳移动。然后，机器人实际上通过选择访问次数最多的第一个孩子来在原始的真实游戏中采取行动。例如，如果我们的树的顶部看起来像：



然后机器人会选择第一个动作并继续

$s_{0,1}$

通过专家策略提高效率

国际象棋和围棋等游戏具有非常大的分支因子。在给定的游戏状态下，有许多可能的动作要采取，因此很难充分探索未来的游戏状态。结果，估计有

10^{46}

国际象棋中的棋盘状态和传统的围棋

19×19

板子周围有

10^{170}

(井字游戏只有

5478

状态)。

使用 vanilla Monte Carlo 树搜索进行移动评估的效率不够高。我们需要一种方法来进一步将注意力集中在有价值的举措上。

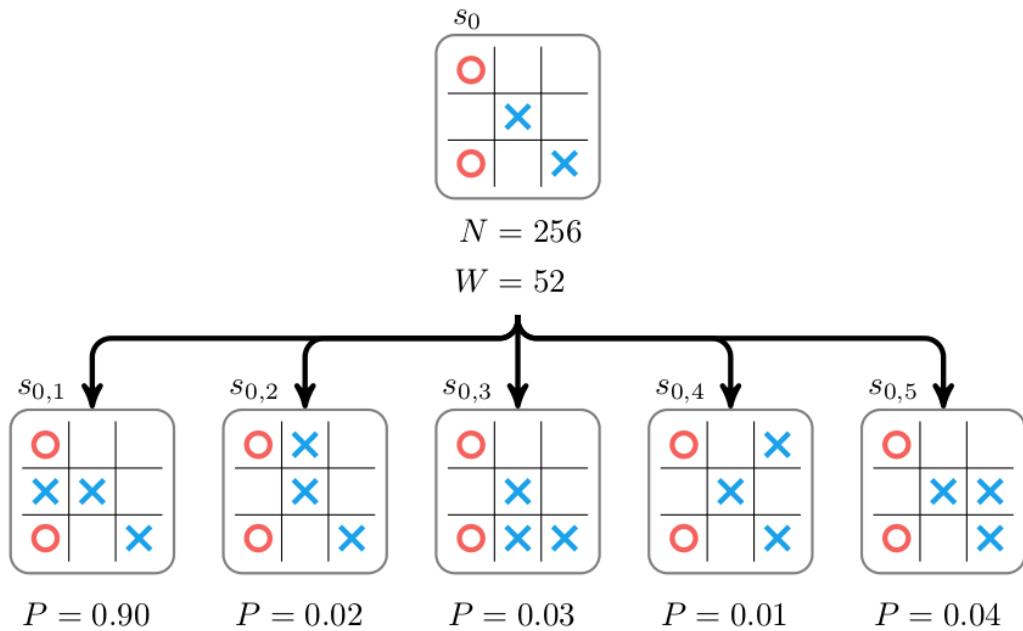
假设我们有一个专家策略

π

那，对于给定的状态

 s

，告诉我们专家级玩家做出每一个可能的动作的可能性有多大。对于井字游戏示例，这可能如下所示：



其中每个

磷世 $\pi(s_0)$

是选择的概率

一世

动作

一世

给定根状态

s_0

如果专家策略真的很好，那么我们可以通过根据产生的概率直接绘制我们的下一个动作来产生一个强大的机器人

 π

, 或者更好的是, 以最高概率采取行动。不幸的是, 获得专家策略很困难, 验证一个人的策略是否最优也很困难。

幸运的是, 可以通过使用蒙特卡洛树搜索的修改形式来改进策略。这个版本还会根据策略存储每个节点的概率, 这个概率用于在选择时调整节点的得分。DeepMind 使用的概率上置信树分数是:

$$\hat{U}_{\text{一世}} = \frac{W_{\text{一世}}}{\tilde{n}_{\text{一世}}} + c_{\text{磷}} \sqrt{\frac{\ln \tilde{n}_p}{1 + \tilde{n}_{\text{一世}}}}$$

和以前一样, 分数在持续产生高分的节点和未探索的节点之间进行权衡。现在, 节点探索由专家策略引导, 将探索偏向于专家策略认为可能的移动。如果专家策略真的很好, 那么蒙特卡洛树搜索有效地关注游戏状态的良好演变。如果专家策略很差, 那么蒙特卡洛树搜索可能会关注游戏状态的不良演变。无论哪种方式, 在样本数量变大的限制下, 节点的价值由赢/输比支配

$$W_{\text{一世}} / \tilde{n}_{\text{一世}}$$

, 和以前一样。

通过价值近似提高效率

第二种形式的效率可以通过避免昂贵且可能不准确的随机推出实现。一种选择是使用上一节中的专家策略来指导随机推出。如果策略是好的, 那么推出应该反映更现实的、专家级的游戏进程, 从而更可靠地估计一个状态的价值。

第二种选择是完全避免推出, 并使用值逼近函数直接逼近状态的值

$$\hat{W}(\cdot)$$

. 这个函数接受一个状态并直接计算一个值

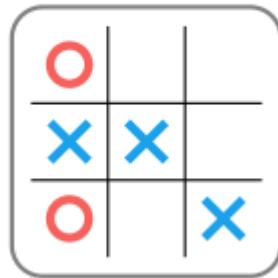
$$[-1, 1]$$

, 不进行部署。显然, 如果

$$\hat{W}$$

是真实值的一个很好的近似值, 但可以比 rollout 更快地执行, 因此可以在不牺牲性能的情况下节省执行时间。

$$\hat{W}(s_{0,1}) = 0.1$$



值近似可以与专家策略一起使用，以加速蒙特卡洛树搜索。一个严重的问题仍然存在——如何获得专家策略和价值函数？是否存在用于训练专家策略和价值函数的算法？

Alpha 零神经网络

随着时间的推移，Alpha Zero 算法通过加速蒙特卡洛树搜索与自己进行博弈，从而产生越来越好的专家策略和价值函数。专家政策

$$\pi$$

和近似值函数

$$\hat{W}$$

都由深度神经网络表示。事实上，为了提高效率，Alpha Zero 使用了一个神经网络

$$F$$

它接受游戏状态并产生下一步移动的概率和近似状态值。（从技术上讲，它需要前八种游戏状态和一个指示轮到它的指示器。）

$$F(s) \rightarrow [\mathbf{p}, W]$$

通过使用神经网络评估它们来扩展搜索树中的叶子。每个孩子都初始化为

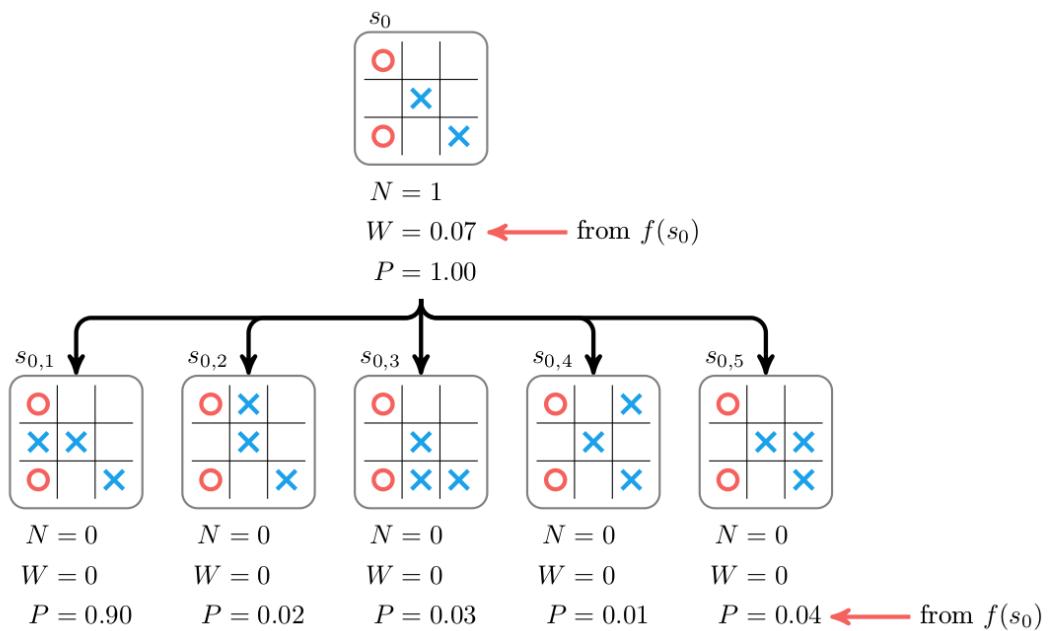
$$\tilde{n} = 0$$

$$W = 0$$

，与

$$\text{磷}$$

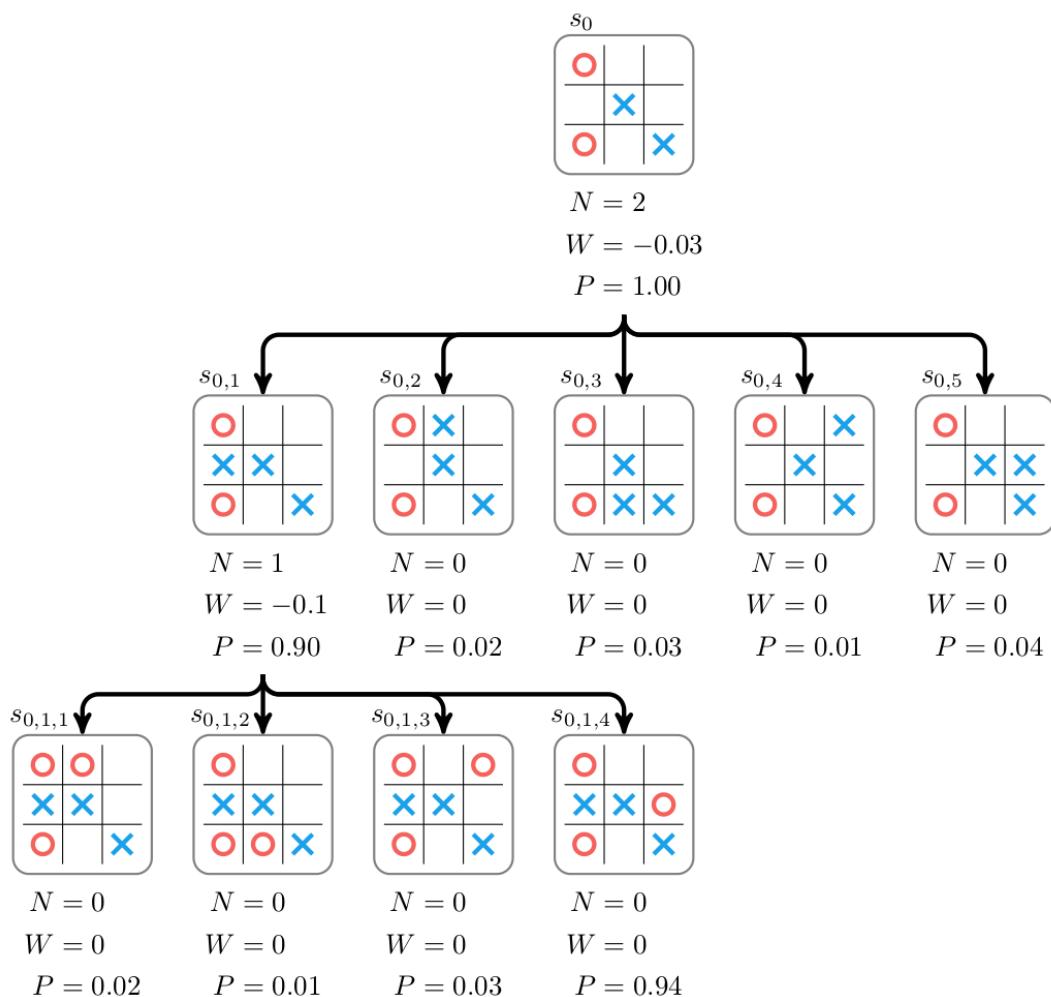
对应于网络的预测。扩展节点的值设置为预测值，然后将该值备份到树中。



选择和备份不变。简而言之，在备份期间，父母的访问计数会增加，并且其值会根据

$$W$$

在另一个选择、扩展和备份步骤之后的搜索树是：



Alpha Zero 算法的核心思想是可以提高神经网络的预测能力，利用蒙特卡洛树搜索产生的玩法来提供训练数据。通过训练预测概率来改进神经网络的策略部分

\mathbf{p}

为了

s_0

匹配改进的概率

π

从运行蒙特卡洛树获得

s_0

. 运行蒙特卡洛树搜索后，改进后的策略预测为：

$$\pi = \tilde{\pi}^{1/\tau}$$

对于一个常数

τ

的价值观

τ

接近零产生根据蒙特卡洛树搜索评估选择最佳移动的策略。

通过训练预测值以匹配游戏的最终赢/输/平结果，提高了神经网络的价值部分，

Z

. 他们的损失函数是：

$$(W - Z)^2 + \pi^\top \ln \mathbf{p} + \lambda \|\boldsymbol{\theta}\|_2^2$$

在哪里

$$(W - Z)^2$$

是价值损失，

$$\pi^\top \ln \mathbf{p}$$

是策略损失，并且

$$\lambda \|\theta\|_2^2$$

是一个带有参数的额外正则化项

$$\lambda \geq 0 \quad \dots$$

和

$$\theta$$

表示神经网络中的参数。

训练完全是在自我游戏中完成的。一个从一组随机初始化的神经网络参数开始

$$\theta$$

. 然后，这个神经网络被用于它自己玩的多个游戏中。在这些游戏中，对于每一步，蒙特卡洛树搜索用于计算

$$\pi$$

. 每场比赛的最终结果决定了该场比赛的价值

$$Z$$

. 参数

$$\theta$$

然后通过在损失函数上使用梯度下降（或任何更复杂的加速下降方法 - Alpha Zero 使用具有动量和学习率退火的随机梯度下降）来改进随机选择的状态。

结束评论

就是这样。DeepMind 的人们贡献了一种干净且稳定的学习算法，该算法仅使用来自自我游戏的数据有效地训练游戏代理。虽然当前的零算法仅适用于离散游戏，但它是否会在未来扩展到 MDP 或它们的部分可观察对应物将会很有趣。

有趣的是，人工智能领域的发展速度有多快。那些声称我们将能够及时看到机器人霸主到来的人应该注意——这些人工智能只会在短暂的瞬间达到人类水平，然后冲过我们进入超人类领域，永不回头。

此条目发表在未分类。[为永久链接添加书签。](#)

[←使用 C3.js 的 Apteryx 飞行数据](#)

[我们如何编写教科书→](#)

关于“AlphaGo 零 - 如何以及为什么工作”的 2 个想法

Pingback: [如何使用 Python 和 Keras 构建自己的 AlphaZero AI | 复制粘贴程序员](#)

Pingback: [AlphaZero 和人类知识的诅咒 | 复制粘贴程序员](#)

评论被关闭。

元

- [登录](#)

在 WordPress 上运行 | 主题: [hndr](#) 的 Mog。

Tim Wheeler

[About](#) [Archive](#) [Blog](#)

[**← Apteryx Flight Data with C3.js**](#)[**How We Wrote a Textbook →**](#)

AlphaGo Zero – How and Why it Works

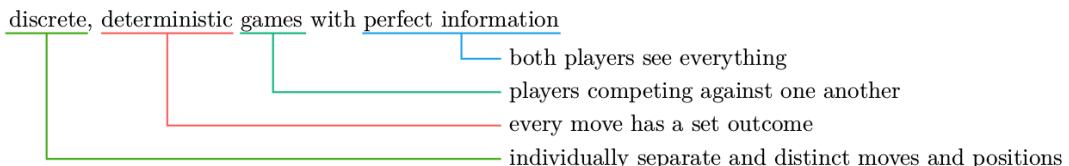
November 2, 2017 by Tim

DeepMind's AlphaGo made waves when it became the first AI to beat a top human Go player in March of 2016. This version of AlphaGo – AlphaGo Lee – used a large set of Go games from the best players in the world during its training process. A new paper was released a few days ago detailing a new neural net—AlphaGo Zero—that does not need humans to show it how to play Go. Not only does it outperform all previous Go players, human or machine, it does so after only three days of training time. This article will explain how and why it works.

Monte Carlo Tree Search

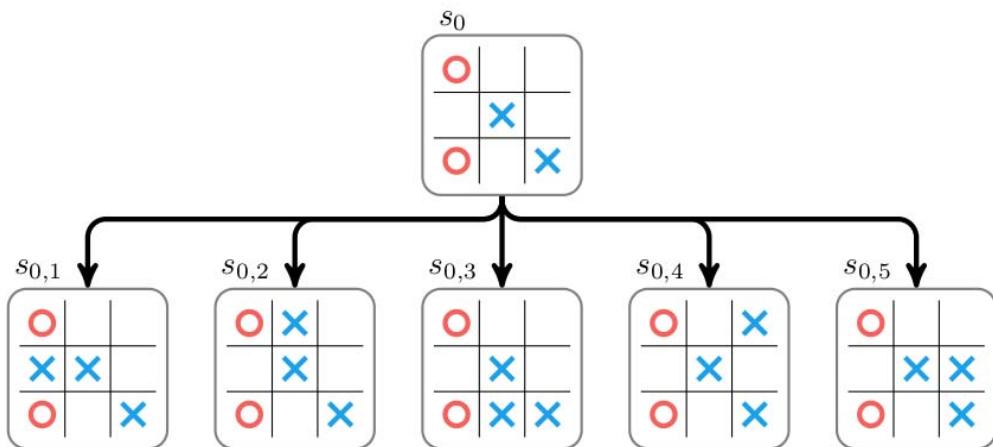
The go-to algorithm for writing bots to play discrete, deterministic games with perfect information is Monte Carlo tree search (MCTS). A bot playing a game like Go, chess, or checkers can figure out what move it should make by trying them all, then checking all possible responses by the opponent, all possible moves after that, etc. For a game like Go the number of moves to try grows really fast.

Monte Carlo tree search will selectively try moves based on how good it thinks they are, thereby focusing its effort on moves that are most likely to happen.



More technically, the algorithm works as follows. The game-in-progress is in an initial state s_0 , and it is the bot's turn to play. The bot can choose from a set of actions \mathcal{A} . Monte Carlo tree search begins with a tree consisting of a single node for s_0 . This node is *expanded* by

trying every action $a \in \mathcal{A}$ and constructing a corresponding child node for each action. Below we show this expansion for a game of tic-tac-toe:



The value of each new child node must then be determined. The game in the child node is *rolled out* by randomly taking moves from the child state until a win, loss, or tie is reached. Wins are scored at

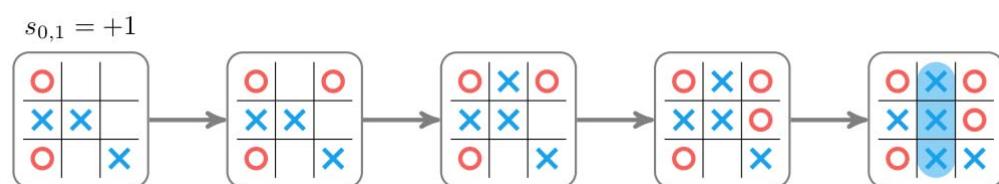
+1

, losses at

-1

, and ties at

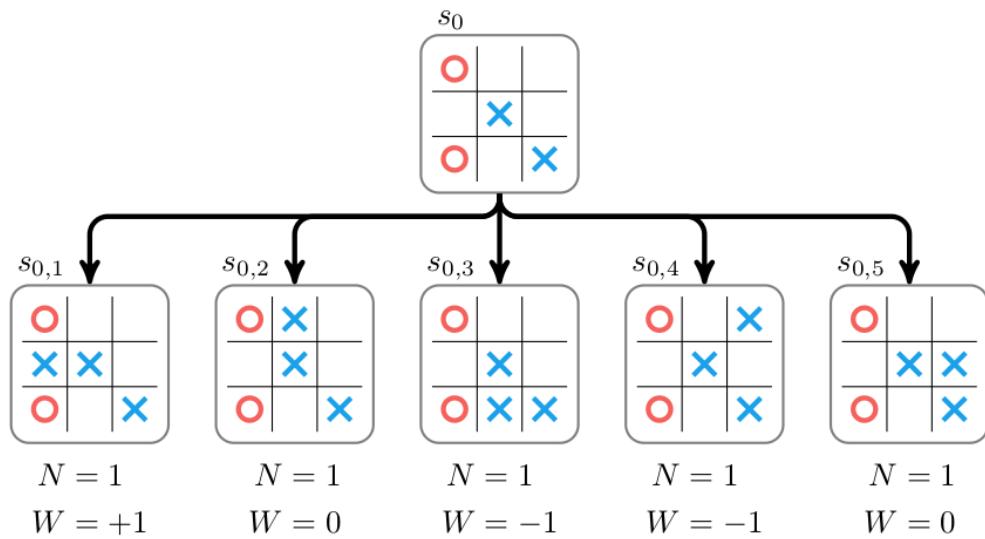
0



The random rollout for the first child given above estimates a value of

+1

- . This value may not represent optimal play—it can vary based on how the rollout progresses. One can run rollouts unintelligently, drawing moves uniformly at random. One can often do better by following a better—though still typically random—strategy, or by estimating the value of the state directly. More on that later.



Above we show the expanded tree with approximate values for each child node. Note that we store two properties: the accumulated value

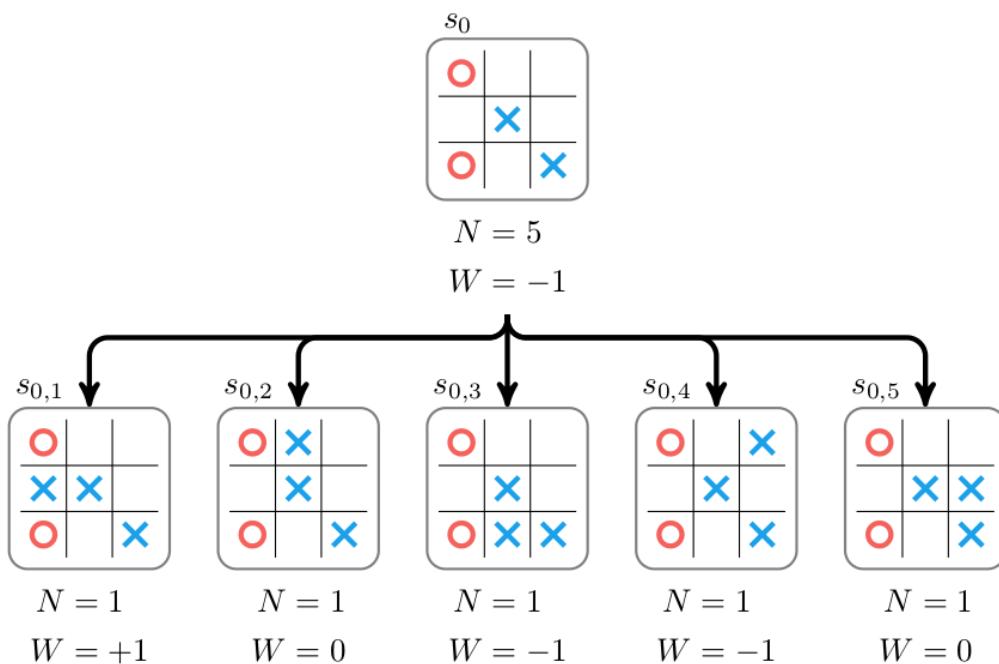
$$W$$

and the number of times rollouts have been run at or below that node,

$$N$$

. We have only visited each node once.

The information from the child nodes is then propagated back up the tree by increasing the parent's value and visit count. Its accumulated value is then set to the total accumulated value of its children:



Monte Carlo tree search continues for multiple iterations consisting of selecting a node, expanding it, and propagating back up the new information. Expansion and propagation have

already been covered.

Monte Carlo tree search does not expand all leaf nodes, as that would be very expensive. Instead, the selection process chooses nodes that strike a balance between being lucrative-having high estimated values-and being relatively unexplored-having low visit counts.

A leaf node is selected by traversing down the tree from the root node, always choosing the child

i

with the highest upper confidence tree (UCT) score:

$$U_i = \frac{W_i}{N_i} + c \sqrt{\frac{\ln N_p}{N_i}}$$

where

W_i

is the accumulated value of the

i

th child,

N_i

is the visit count for

i

th child, and

N_p

is the number of visit counts for the parent node. The parameter

$c \geq 0$

controls the tradeoff between choosing lucrative nodes (low

c

) and exploring nodes with low visit counts (high

c

). It is often set empirically.

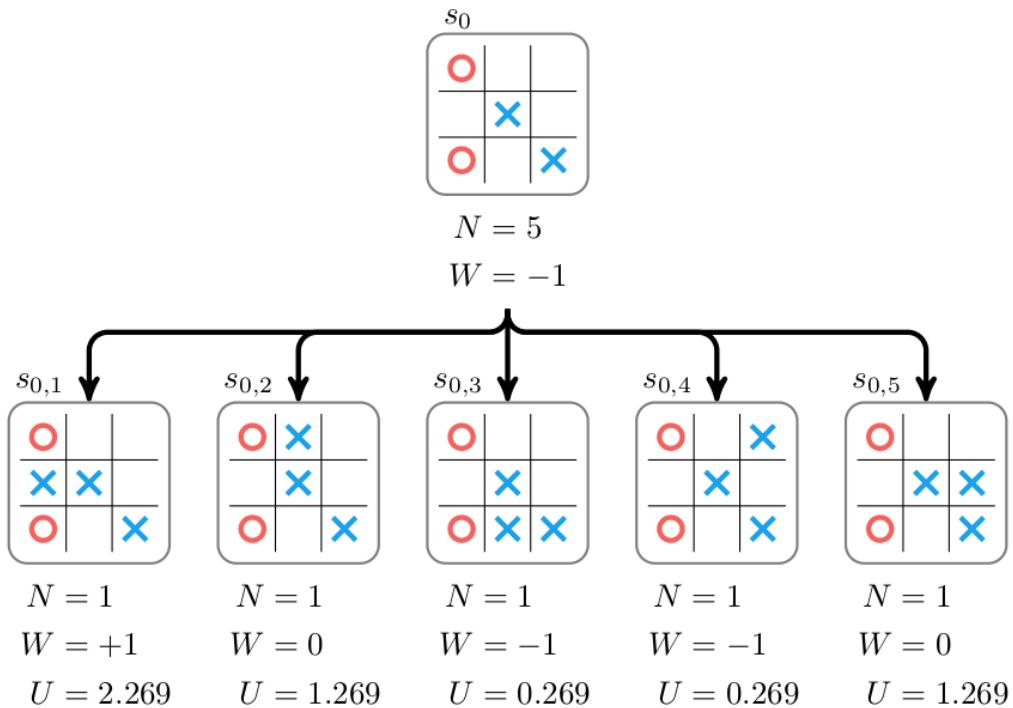
The UCT scores (

$$U$$

's) for the tic-tac-toe tree with

$$c = 1$$

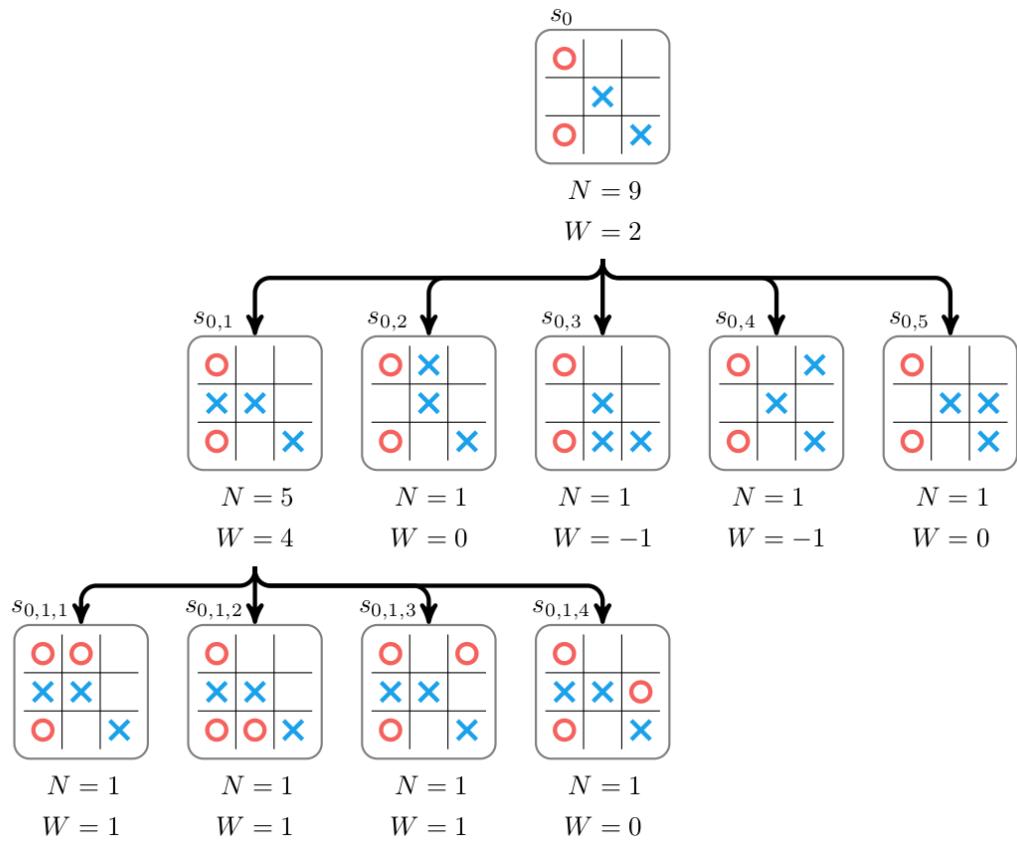
are:



In this case we pick the first node,

$$s_{0,1}$$

. (In the event of a tie one can either randomly break the tie or just pick the first of the best nodes.) That node is expanded and the values are propagated back up:



Note that each accumulated value

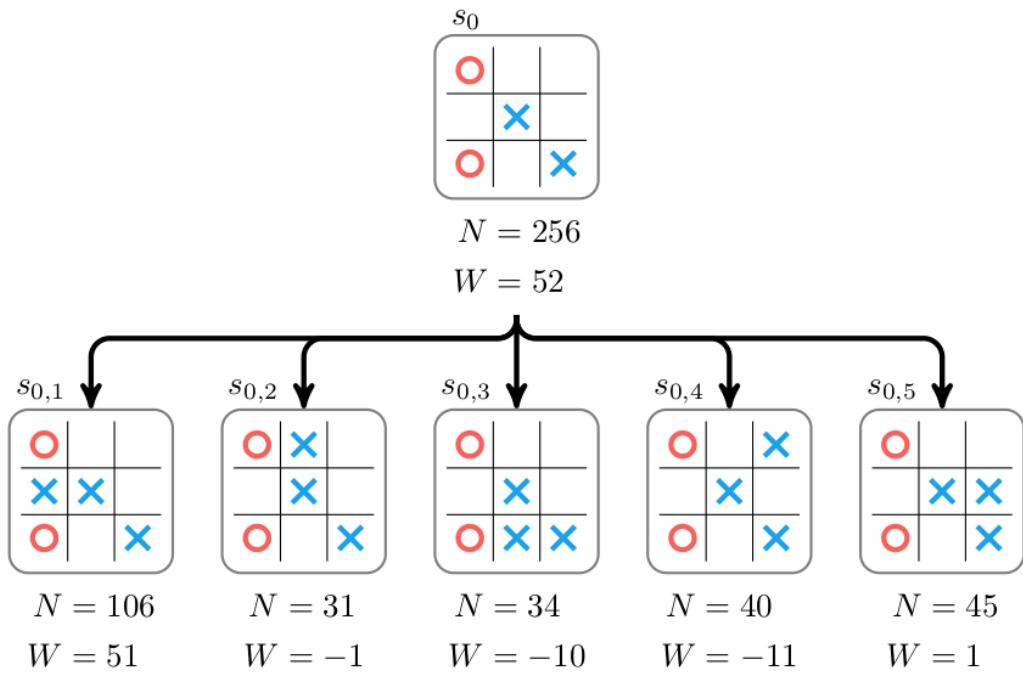
W

reflects whether X's won or lost. During selection, we keep track of whether it is X's or O's turn to move, and flip the sign of

W

whenever it is O's turn.

We continue to run iterations of Monte Carlo tree search until we run out of time. The tree is gradually expanded and we (hopefully) explore the possible moves, identifying the best move to take. The bot then actually makes a move in the original, real game by picking the first child with the highest number of visits. For example, if the top of our tree looks like:



then the bot would choose the first action and proceed to

$$s_{0,1}$$

Efficiency Through Expert Policies

Games like chess and Go have very large branching factors. In a given game state there are many possible actions to take, making it very difficult to adequately explore the future game states. As a result, there are an estimated

$$10^{46}$$

board states in chess, and Go played on a traditional

$$19 \times 19$$

board has around

$$10^{170}$$

(Tic-tac-toe only has

$$5478$$

states).

Move evaluation with vanilla Monte Carlo tree search just isn't efficient enough. We need a way to further focus our attention to worthwhile moves.

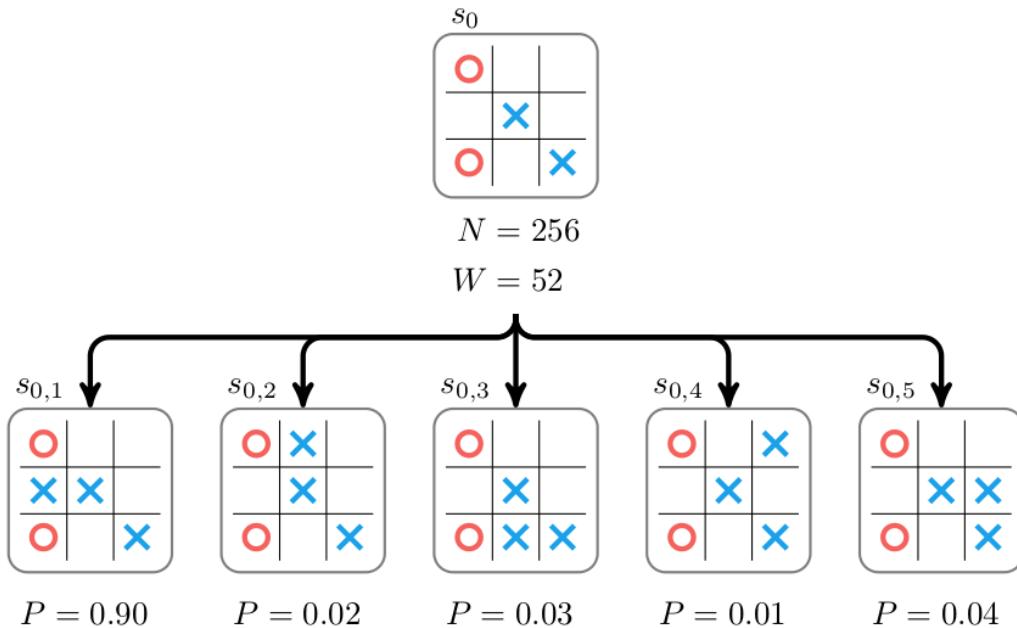
Suppose we have an *expert policy*

$$\pi$$

that, for a given state

$$s$$

, tells us how likely an expert-level player is to make each possible action. For the tic-tac-toe example, this might look like:



where each

$$P_i = \pi(a_i \mid s_0)$$

is the probability of choosing the

i

th action

$$a_i$$

given the root state

$$s_0$$

If the expert policy is really good then we can produce a strong bot by directly drawing our next action according to the probabilities produced by

$$\pi$$

, or better yet, by taking the move with the highest probability. Unfortunately, getting an expert policy is difficult, and verifying that one's policy is optimal is difficult as well.

Fortunately, one can improve on a policy by using a modified form of Monte Carlo tree search. This version will also store the probability of each node according to the policy, and this probability is used to adjust the node's score during selection. The probabilistic upper confidence tree score used by DeepMind is:

$$U_i = \frac{W_i}{N_i} + cP_i \sqrt{\frac{\ln N_p}{1 + N_i}}$$

As before, the score trades off between nodes that consistently produce high scores and nodes that are unexplored. Now, node exploration is guided by the expert policy, biasing exploration towards moves the expert policy considers likely. If the expert policy truly is good, then Monte Carlo tree search efficiently focuses on good evolutions of the game state. If the expert policy is poor, then Monte Carlo tree search may focus on bad evolutions of the game state. Either way, in the limit as the number of samples gets large, the value of a node is dominated by the win/loss ratio

$$W_i/N_i$$

, as before.

Efficiency Through Value Approximation

A second form of efficiency can be achieved by avoiding expensive and potentially inaccurate random rollouts. One option is to use the expert policy from the previous section to guide the random rollout. If the policy is good, then the rollout should reflect more realistic, expert-level game progressions and thus more reliably estimate a state's value.

A second option is to avoid rollouts altogether, and directly approximate the value of a state with a value approximator function

$$\hat{W}(x)$$

. This function takes a state and directly computes a value in

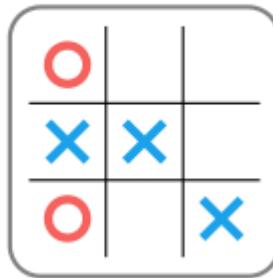
$$[-1, 1]$$

, without conducting rollouts. Clearly, if

$$\hat{W}$$

is a good approximation of the true value, but can be executed faster than a rollout, then execution time can be saved without sacrificing performance.

$$\hat{W}(s_{0,1}) = 0.1$$



Value approximation can be used in tandem with an expert policy to speed up Monte Carlo tree search. A serious concern remains—how does one obtain an expert policy and a value function? Does an algorithm exist for training the expert policy and value function?

The Alpha Zero Neural Net

The Alpha Zero algorithm produces better and better expert policies and value functions over time by playing games against itself with accelerated Monte Carlo tree search. The expert policy

$$\pi$$

and the approximate value function

$$\hat{W}$$

are both represented by deep neural networks. In fact, to increase efficiency, Alpha Zero uses one neural network

$$f$$

that takes in the game state and produces both the probabilities over the next move and the approximate state value. (Technically, it takes in the previous eight game states and an indicator telling it whose turn it is.)

$$f(s) \rightarrow [\mathbf{p}, W]$$

Leaves in the search tree are expanded by evaluating them with the neural network. Each child is initialized with

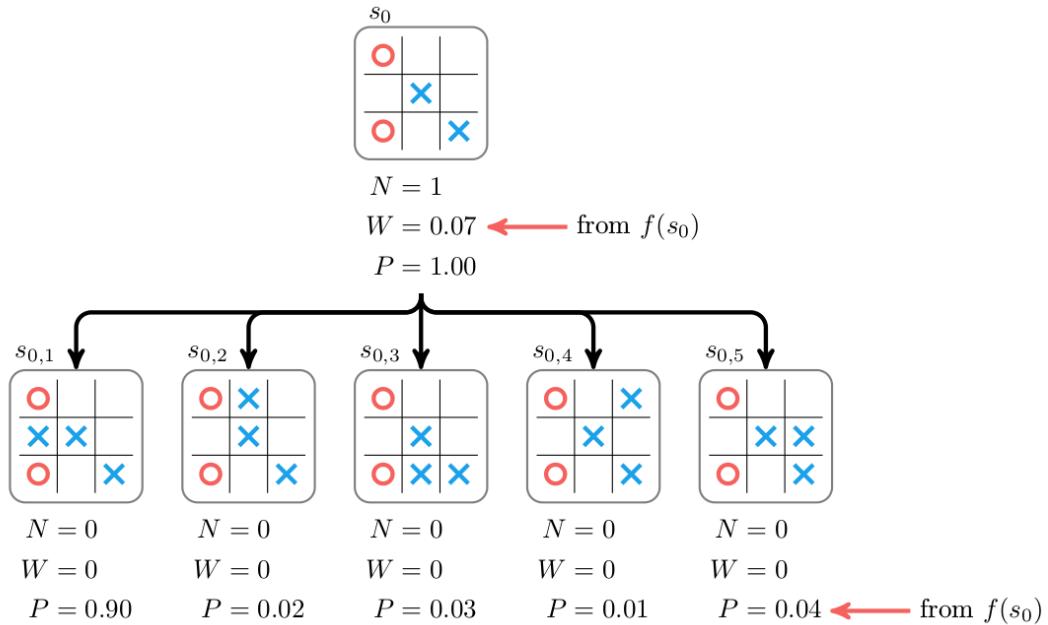
$$N = 0$$

$$W = 0$$

, and with

P

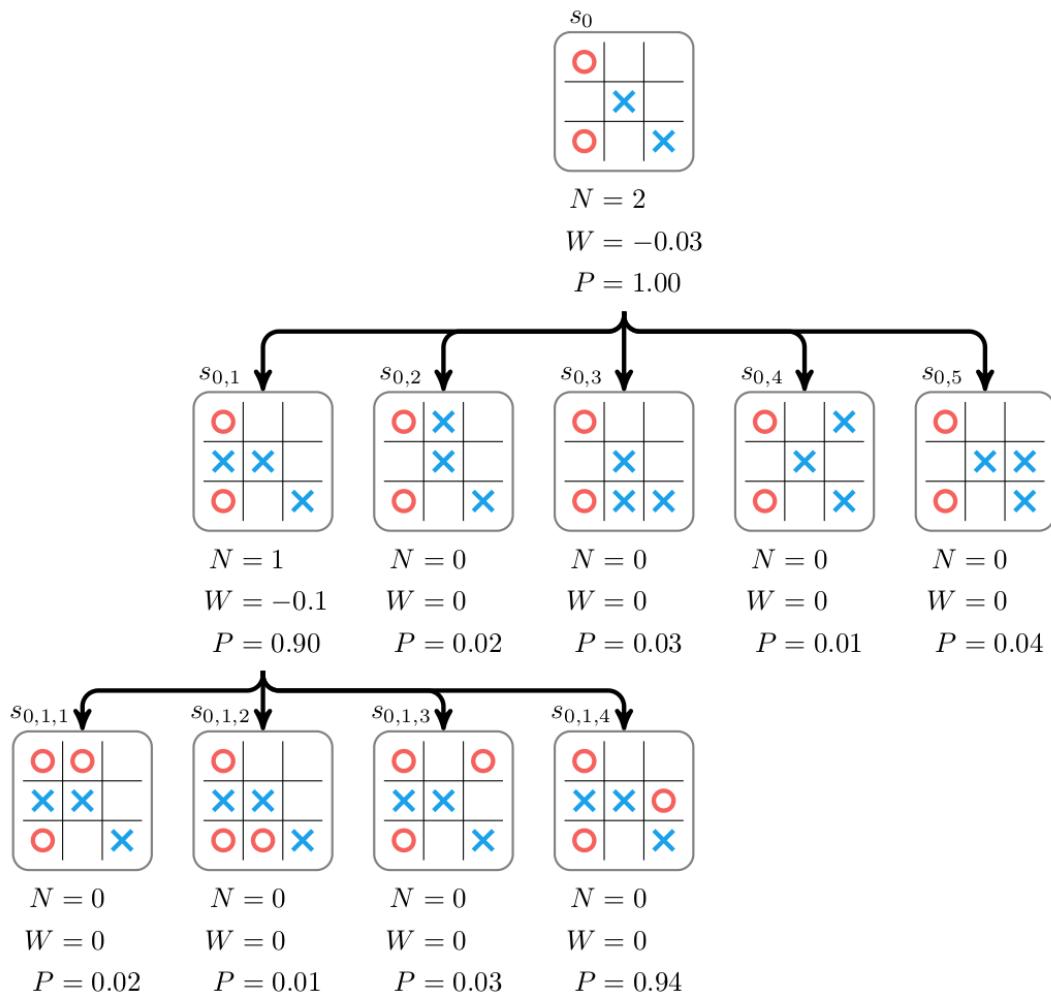
corresponding to the prediction from the network. The value of the expanded node is set to the predicted value and this value is then backed up the tree.



Selection and backup are unchanged. Simply put, during backup a parent's visit counts are incremented and its value is increased according to

 W

The search tree following another selection, expansion, and backup step is:



The core idea of the Alpha Zero algorithm is that the predictions of the neural network can be improved, and the play generated by Monte Carlo tree search can be used to provide the training data. The policy portion of the neural network is improved by training the predicted probabilities

\mathbf{p}

for

s_0

to match the improved probability

π

obtained from running Monte Carlo tree on

s_0

. After running Monte Carlo tree search, the improved policy prediction is:

$$\pi_i = N_i^{1/\tau}$$

for a constant

$$\tau$$

. Values of

$$\tau$$

close to zero produce policies that choose the best move according to the Monte Carlo tree search evaluation.

The value portion of the neural network is improved by training the predicted value to match the eventual win/loss/tie result of the game,

$$Z$$

. Their loss function is:

$$(W - Z)^2 + \pi^\top \ln p + \lambda \|\theta\|_2^2$$

where

$$(W - Z)^2$$

is the value loss,

$$\pi^\top \ln p$$

is the policy loss, and

$$\lambda \|\theta\|_2^2$$

is an extra regularization term with parameter

$$\lambda \geq 0$$

and

$$\theta$$

represents the parameters in the neural network.

Training is done entirely in self-play. One starts with a randomly initialized set of neural network parameters

$$\theta$$

. This neural network is then used in multiple games in which it plays itself. In each of these games, for each move, Monte Carlo tree search is used to calculate

$$\pi$$

. The final outcome of each game determines that game's value for

$$Z$$

. The parameters

$$\theta$$

are then improved by using gradient descent (Or any of the more sophisticated accelerated descent methods-Alpha Zero used stochastic gradient descent with momentum and learning rate annealing.) on the loss function for a random selection of states played.

Closing Comments

And that's it. The folks at DeepMind contributed a clean and stable learning algorithm that trains game-playing agents efficiently using only data from self-play. While the current Zero algorithm only works for discrete games, it will be interesting whether it will be extended to MDPs or their partially observable counterparts in the future.

It is interesting to see how quickly the field of AI is progressing. Those who claim we will be able to see the robot overlords coming in time should take heed – these AI's will only be human-level for a brief instant before blasting past us into superhuman territories, never to look back.

This entry was posted in Uncategorized. Bookmark the [permalink](#).

[← Apteryx Flight Data with C3.js](#)

[How We Wrote a Textbook →](#)

2 thoughts on “AlphaGo Zero – How and Why it Works”

Pingback: [How to build your own AlphaZero AI using Python and Keras | Copy Paste Programmers](#)

Pingback: [AlphaZero and the Curse of Human Knowledge | Copy Paste Programmers](#)

Comments are closed.