【深度强化学习突破】OpenAI Gym 玩游戏达到人类水平

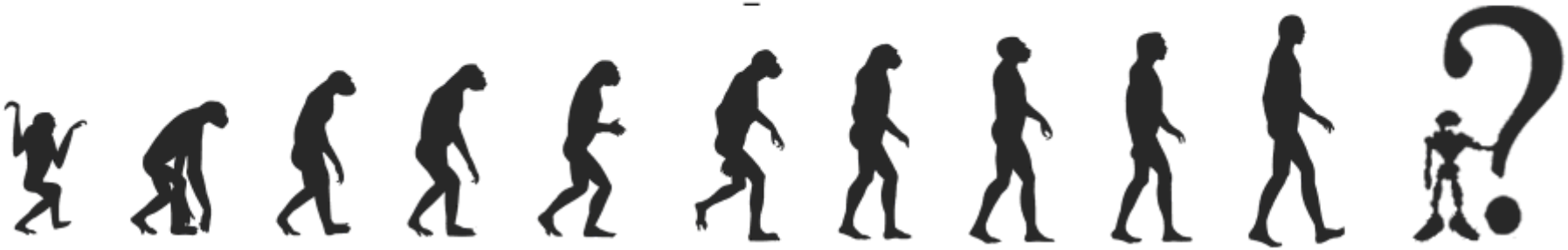
2016-06-10 新智元

新智元编译

来源：karpathy.github.io

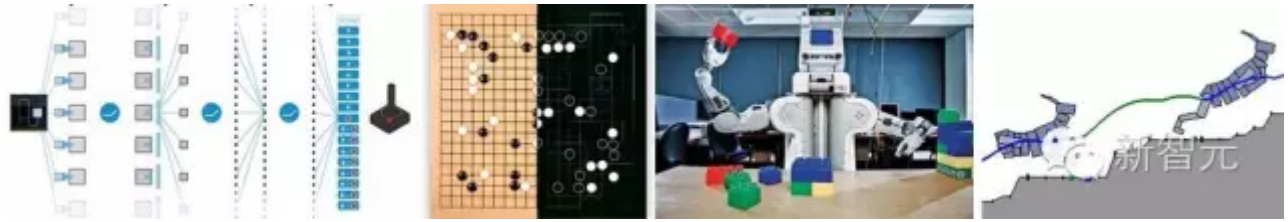
翻译：王婉婷

【新智元导读】许多人不信只用1个强化学习算法，就能让计算机从零开始从像素中自动学会玩大部分ATARI游戏，并达到人类的表现水平。本文中，参与设计与研发OpenAI Gym的Kar Pathy，以Pong!这款ATARI游戏为例，利用强大的策略梯度算法，颠覆上述认知。本文总结了深度强化学习为何意义重大、怎样开发，并展望了深度强化学习推动人工智能的发展，在复杂机器人环境中的应用以及解决实际问题。



（文 / Kar Pathy）这是一篇关于强化学习（Reinforcement Learning，LR）迟来已久的文章。RL非常热门！你可能已经注意到了，计算机现在能够自动（从游戏原始的像素中）学会玩ATARI游戏，它们在围棋比赛中战胜了世界冠军，模拟的四足机器人正在学习如何奔跑和跳跃，而机器人在学习着如何在无需人工为任务编程的情况下完成复杂的操作任务。事实上，所有这些进展都需要归功于RL研究。

我差不多也在去年对RL产生了兴趣：我参与编写了Richard Sutton关于RL的书、念完了David Silver关于RL的课程、观看了John Schulmann关于RL的讲座、用Javascript编写了一个RL库、夏天一直在DeepMind的DeepRL小组实习、而最近则为OpenAI Gym——一款全新的RL基准测试工具包——的设计和开发提供了一些帮助。所以，我已经至少在这个有趣的领域中待了一年，不过我始终没有写过什么文章来全面地描述RL——为什么它意义重大、它是什么、怎样开发、以及它将何去何从——直到今天。



RL的实际案例。从左到右分别是：学习玩ATARI游戏的深度 Q学习网络、AlphaGo、Berkeley的机器人正在堆乐高积木、模拟物理规则的四足机器人跳跃过（无法行走的）地形。

对RL领域近期进展的本质进行思考是一件很有趣的事。我宽泛地将人工智能受限的原因分为4个独立的因素：

1. 计算力（明显的有：摩尔定律、GPU、ASIC）
2. 数据（拥有良好的形式，比如ImageNet，而非网络上随处可得的那种数据）
3. 算法（来自于一些研究和想法，比如反向传播、CNN、LSTM）
4. 基础设施（你拥有的软件——Linux、TCP/IP、Git、ROS、PR2、AWS、AMT、TensorFlow等等）

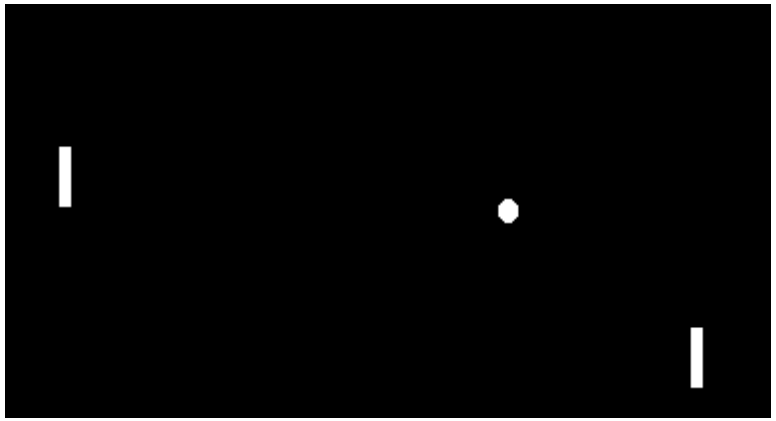
与计算机视觉领域非常相似，RL领域的进展所受到的新奇想法的推动力并不像你可能以为的那样巨大。在计算机视觉领域，2012年的AlexNet基本上就是1990年ConvNets提升规模后（更深、更广）的版本。与此类似，2013年玩ATARI游戏的深度Q学习论文完成的，是对一个常规算法的实现（在Q学习中运用函数逼近，你可以在Sutton于1998年出版的RL书籍中找到这个想法），而其中的函数逼近器恰好是一个ConvNet。AlphaGo用的是策略梯度（Policy Gradients，PG）和蒙特卡洛树搜索（MCTS）——这些也都是常规配置。当然，让它成功运作起来需要大量的精力和耐心，在旧有的算法之上也有了一些机智的微调，但是，对于一阶近似（first-order approximation）来说，近期进展的主要推动力并非来自算法、而是（像计算机视觉领域那样）来自计算力/数据/基础设施。

现在让我们回到RL。每当在一样东西看上去有多神奇和它实际上有多简单之间有所割裂的时候，我总会急得想要给它写一篇文章说道说道。在RL方面，我见过许多人都不相信我们能够用1个算法、从像素中、从计算机对游戏毫无理解开始，让它自动学会玩大部分ATARI游戏，并达到人类的表现水平——这令人惊奇，并且我对此有亲身体验。不过，从核心来看，我们使用的方法也并不怎么新颖（虽然我明白当我们回顾过去的时候说出这种话，是再正常不过的事了），随它去吧。

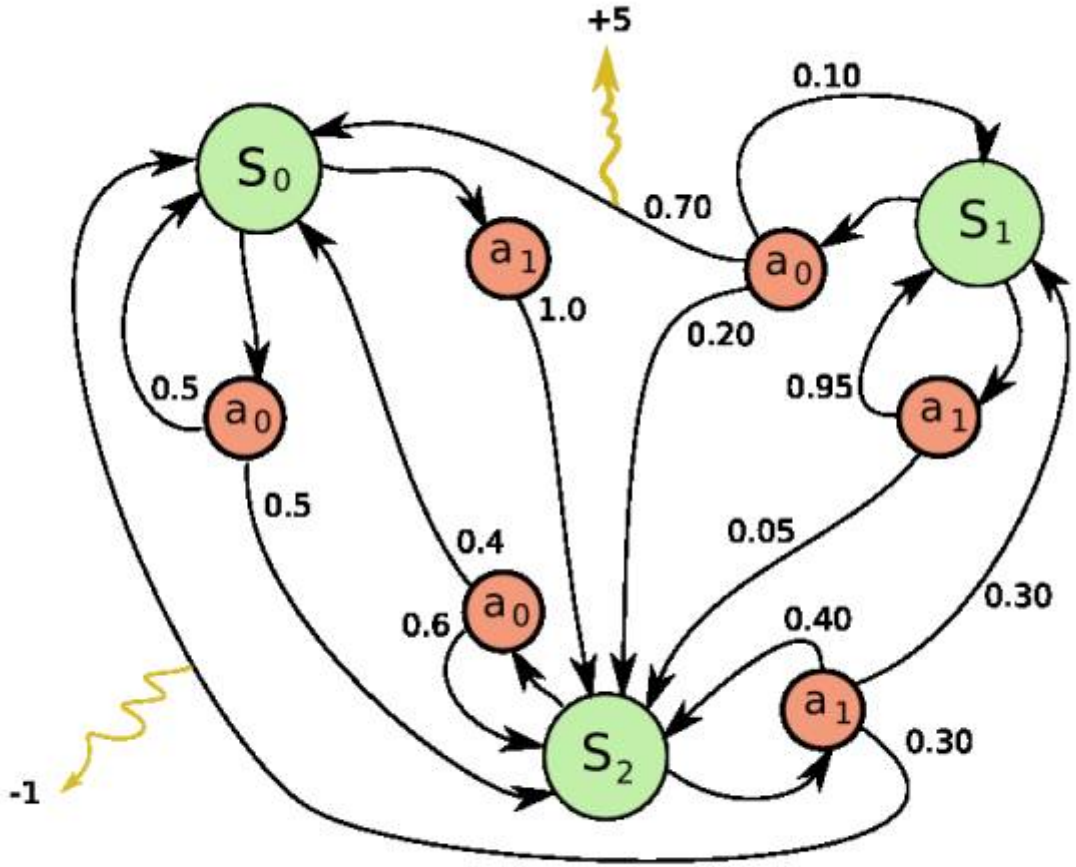
我将要带你看策略梯度（Policy Gradients，PG），当前我们解决RL问题时最喜欢用的默认选择。如果你对RL领域不太了解，你可能会感到好奇：为什么不展示RL领域中更广为人知的、在学习玩ATARI游戏的论文中用到的DQN算法呢？事实上，Q学习并不是一个很好的算法（你甚至可以说DQN只属于2013年（好吧有一半是我在开玩笑））。RL领域中，大部分人偏爱使用的是策略梯度（PG）——包括最初那篇DQN论文的作者，他们展示了经过良好调整的策略梯度能够取得比Q学习更好的效果。策略梯度受到欢迎是因为它是端到端的：它有显型策略（explicit policy）、有方法能够直接对预期回报进行优化。

我将以一款ATARI游戏（Pong!）作为例子，介绍如何用PG和深度神经网络来从像素中、从程序对游戏毫无了解开始，让程序学会玩这款游戏，而我所需的一切是130行Python代码，只用到了numpy包。让我们开始吧。

Pong 的像素



Pong游戏



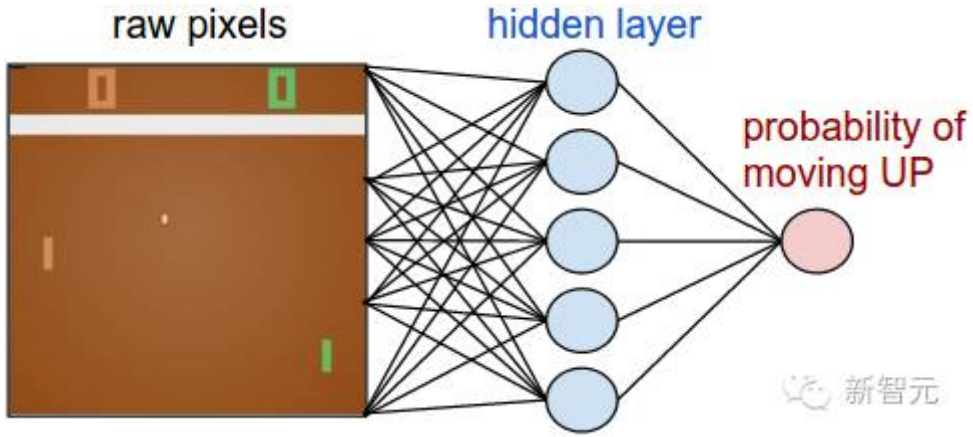
Pong是马尔可夫决策过程（Markov Decision Process，MDP）的一个特例：图中，每一个节点都是一个特定的游戏状态，每一条边（edge）都是一种在普遍概率上可能的移动方向。每一条边也都给出了特定的回报，而目标是在任意游戏状态下计算出让回报最大化的最优方法。

Pong游戏是一个非常好的简单RL任务的例子。在ATARI 2600版本中，你控制其中一块挡板（另一块挡板由一个不错的AI来控制），你需要接住球、把球打回给对方选手。在程序的层面上，这个游戏是这样运作的：我们获得了一帧游戏画面（一个210*160*3字节的数组，用0~255的数字来表示像素的值），然后我们试着决定要把挡板往上移动还是往下移动（也就是一个二元选择）。在每个选择做出之后，游戏模拟器将执行这个行动并给我们回报：或者是球成功越过对手的防御、获得+1的回报，或者是没接到球、获得-1的回报，或者其他情况、获得0的回报。当然，我们的目标是通过移动挡板来获得大量回报。

当我们深入了解这个解决方案时，请务必记得，我们对Pong做的假定非常少，因为我们暗地里并不真正关心Pong这个游戏——我们关心的是复杂的、高维度空间的问题，比如机器人的操纵、装配和引导。**Pong只不过是一个好玩的测试例子，让我们从中了解如何编写非常通用的、未来将能够完成任意的实用任务的AI系统。**

策略网络

首先，我们要定义一个策略网络来实现我们玩家的操作（或者称作“agent”）。这个网络将会获得游戏状态的输入，并决定我们采取什么动作（将挡板上移或是下移）。作为我们最爱的一种简单模块，我们将用一个2层神经网络，获得原始图像像素（总共100800个数字，210*160*3）的输入后，产生一个数字来表示将挡板上移的概率。请注意，常规来说我们会用一个随机游走（stochastic）策略，这意味着我们只会获得一个将挡板上移的概率。每次迭代中，我们会从这个分布中进行抽取（也就是抛一枚重心偏移的硬币）来得到实际行动的决定。这背后的原因，在我们谈论训练的时候就会变得更清晰明白。



我们的策略网络是一个2层的全连接（fully-connected）网络。

为了让这些更落地，以下是你可以用来在使用了numpy的Python中实现这个策略网络的代码。假定我们有一个存有（预处理过的）像素信息的向量 x。我们可以这样写代码：

```
h = np.dot(W1, x) # compute hidden layer neuron activations
h[h<0] = 0 # ReLU nonlinearity: threshold at zero
logp = np.dot(W2, h) # compute log probability of going up
p = 1.0 / (1.0 + np.exp(-logp)) # sigmoid function (gives probability of going up)
```

在这段代码中，W1 和 W2是两个我们随机初始化的矩阵。我们不对它们使用任何改变权重的方法，因为这没什么必要。你可以看到，我们在最后用了sigmoid非线性函数，它会把输出的概率调整到[0,1]的范围里。直观地看，隐藏层中的神经元（它们的权重分配来自于 W1）能够探测到各式各样的游戏情景（例如，球在上方，而我们的挡板在中部），随后W2中的权重能够决定我们是应该把挡板上移还是下移。现在，一开始随机生成的W1和W2毫无疑问已经让玩家蓄势待发。所以，唯一的问题就是找出能够在Pong游戏中发挥出色的W1和W2了！

附属操作：预处理

理想状况中，你会想要将至少2帧画面输入到策略网络里，这样它就能探测到物体的运动。为了将这些稍做简化（我用的是我的Macbook来做这些实验），我会做一些小小的预处理，比如，实际上我向网络中输入的是帧间差（也就是将当前帧减去前一帧）。

这听上去有些不可能

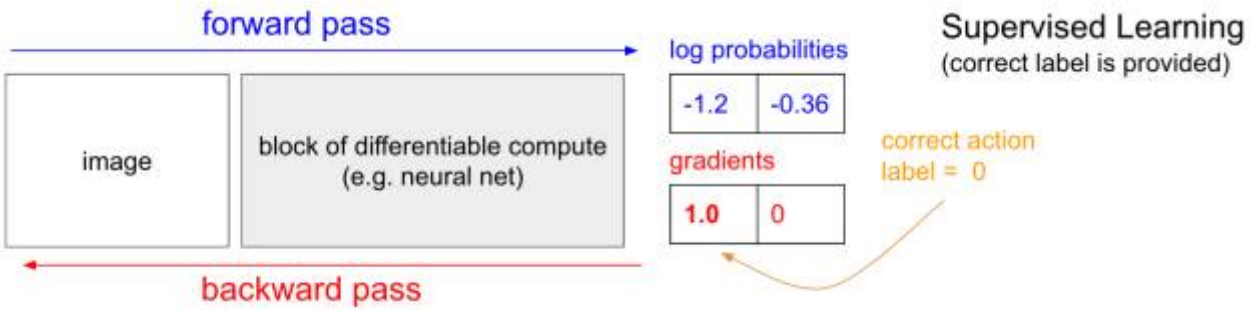
现在我想让你感叹一下RL问题有多么困难。我们有100800个数字（210*160*3）输入到策略网络中（它很容易就会包含像W1和W2中那样的数十万个参数）。假设我们决定要将挡板上移。这时游戏中可能会反馈给我们0回报，并给我们另外100800个代表下一帧的数字。我们可以将这一过程重复几百遍，直到我们获得一个非0的回报！

比如说，假设我们最后得到了+1的回报。这很棒，但是我们要怎么知道这是由什么导致的呢？是我们刚刚做的举动？还是我们在76帧前的动作？或者它与我们在第10帧和第90帧的表现有关？我们又要如何找出在这数十万个参数中，要调整哪个、怎样调整才能带来更好的表现？我们把这个称为信度分配问题（credit assignment problem）。在Pong游戏的例子中，我们知道，如果球越过了对手的防御，我们就会获得+1的回报。真正的原因是，我们击出的球恰好在一个不错的运动轨迹上，不过这可能是我们在许多帧前做的事了——比如说，在Pong中可能是大约20帧前的事，而我们在之后做的每个动作对于我们最后是否获得+1回报都毫无影响。也就是说，我们面对的是一个非常困难的问题，而一切看上去都令人灰心丧气。

监督式学习

在我们继续深入策略梯度的解决方案之前，我想让你简单地回忆一下监督式学习；这是因为，我们很快就会看到，RL与监督式学习非常相似。参见下面的示意图。在常规的监督式学习中，我们将一个图像输入到网络里、随后获得一些概率，比如说，类别“挡板上移”和类别“挡板下移”的概率。图中我为这两个类别用的是log概率(-1.2, -0.36)，而非原始的概率（相当于30%和70%），因为我们优化的是正确标签的log概率——这让数学语言的表达更优美，也相当于优化原始概率，因为是单调的（mnotonic）。现在，在监督式学习中，我们将能访问一个标签。例如，我们可能被告知现在正确的做法是将挡板上移（标签0）.在实际实现中，我们将为“挡板上移”的log概率输入1的梯度，然后运行反向传播来计算梯度向量 $\nabla_W \log p(y = UP \mid x)$。这个梯度将会告诉我们。如何为这数十万个参数做调整，让网络稍微更有可能预测出挡板上移这个动作。

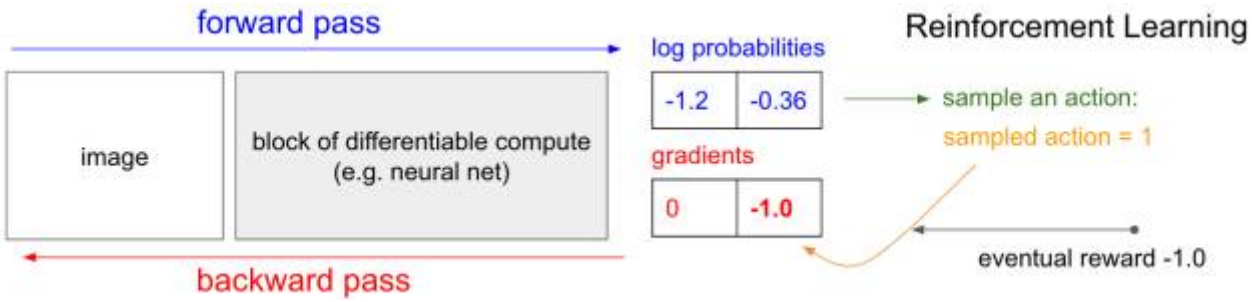
例如，网络中数十万个参数的某一个可能梯度是-2.1，这意味着如果我们稍微提高一点这个参数（比如说，提高0.001），那么预测出挡板上移的log概率会下降2.1*0.001（因为是负数所以是下降）。如果我们随后更新了参数，那么，哇，我们的网络会在面对相似的图像时，变得稍微更有可能预测做出挡板上移的动作。



策略梯度

好了，那么如果我们在RL环境中没有正确的标签的话，该要怎么办？这里是策略梯度的解决方案（参见下面的示意图）。我们的策略网络计算出将挡板上移的概率是30%（log概率-1.2）而下移的概率是70%（log概率-0.36）。现在我们从这个分布中进行抽取，比如说我们抽到了“下移”，然后我们在游戏中这样执行。在这时，你可以注意到一件有趣的事：就像我们在监督式学习中做的那样，我们可以立刻为“下移”填入1的梯度，随后找出让网络在未来稍微更有可能选择“下移”动作的梯度向量。所以我们可以马上评估这个梯度，这很好，不过问题是至少现在我们还不知道将挡板下移是不是一个好的选择。但是关键是，这没关系，因为我们可以等一会儿看看结果！

例如，在Pong游戏中，我们可以等到游戏结束，获得了回报（或者是赢了获得+1，或者是输了获得-1），随后将这个数字输入到我们采取的动作（在这个例子里是“挡板下移”）的梯度中。下面这个例子里，挡板下移最后让我们输了游戏（-1回报）。所以，如果我们在“下移”的log概率中输入-1，然后进行反向传播，我们会获得一个让网络在未来面对相似图像输入时更不倾向于选择“下移”的梯度（这很正确，因为做出这个动作让我们输了游戏）。



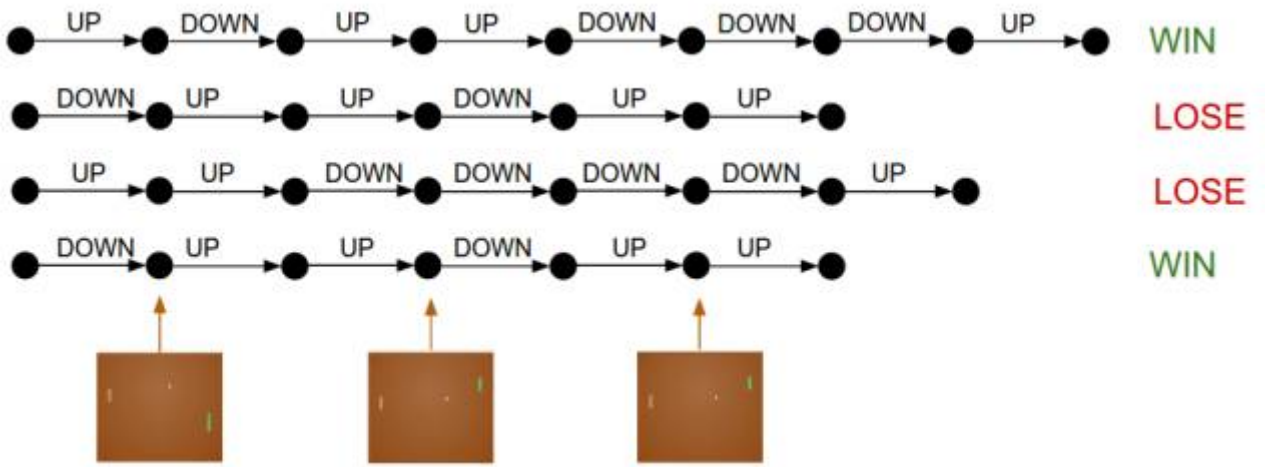
所以，它是这样的：我们有一个随机游走策略，它会从分布中抽取行动，而那些恰好最终带来好的结果的行动会获得更多的青睐，而带来坏的结果的行动则会受到更少的关注。另外，回报并不一定需要以+1和-1的形式来表示我们最终赢得了比赛。它可以是对于结果质量的某种衡量。比如说，如果一切发展得非常好，回报可以是10，我们可以将这个数字替代原先的梯度进行反向传播。这就是神经网络吸引人的地方，它们用起来像是在作弊：你可以在1个teraFLOPS（每秒10^12次浮点运算）的计算力中运行10万个参数，你可以用SGD来让它做任何事。它不应该成立的，但我们的确好巧不巧地生活在一个它能够成立的宇宙中。

训练步骤

接下来要讲的是如何训练的详细步骤。我们用某个W1、W2来初始化策略网络，玩100局Pong游戏（我们称之为策略的“rollouts”）。假设每局游戏都由200帧组成，那么我们总共做出了20000次挡板上移或是挡板下移的决定，对于其中的每一次我们都知道参数的梯度，它告诉我们应该如何调整参数，在未来提高这个游戏状态时的某个动作选择。现在还剩下的是标记我们作出的每个动作选择，将它们标记为好的或是坏的。

比如说，假设我们赢了12局，输了88局。我们会为获胜的12局中的所有决定——200*12=2400个选择——做一次正向更新（为这些动作的梯度输入+1，进行反向传播，而参数更新会让网络更有可能在这些游戏状态中选择我们之前采取的行动）。然后我们也会为输掉的88局中的所有决定——200*88=17600个选择——进行负向更新（让网络更不可能选择这些行动）。然后…就是这样了。现在，这个网络将会稍微更有可能重复那些行得通的选择，并且稍微更不可能重复那些行不通的选择。接着，我们用这个新的、稍微改良过的策略网络来再进行100局游戏，如法炮制。

策略梯度：用一种策略运行一段时间。看看什么行动会带来高额回报，然后提高这些行动的概率。



4局游戏的示意图。每个黑色的圆圈都是某个游戏状态（下方有3张游戏画面图作为例子），每个箭头都是状态之间的转移、并标注了当时抽取出的行动选择。在这个例子里，我们赢得了2局游戏、输掉了2局游戏。我们将会在策略网络中提高获胜的游戏中采取的行动的概率，降低在输掉的游戏中采取的行动的概率。

如果你从头到尾思考一遍这个过程，你会发现一些有趣的地方。比如说，如果我们在第50帧的时候做出了一个好的行动（正确将球击回给对手），但随后在第150帧与球擦肩而过，那会怎么样呢？如果这一局中的每一个行动都被标记为坏的（因为我们输了这一局），这难道不会降低在第50帧做出正确回击的概率吗？你是对的——的确会。然而，当这个过程对数千数万局游戏不断重复时，第一次的正确回击能让你稍微更有可能最终赢得胜利，所以平均来说在这一步上正确回击获得的正向更新次数会超过负向更新次数，于是你的策略最终还是会做出正确的选择。

更通用的优势函数（advantage function）

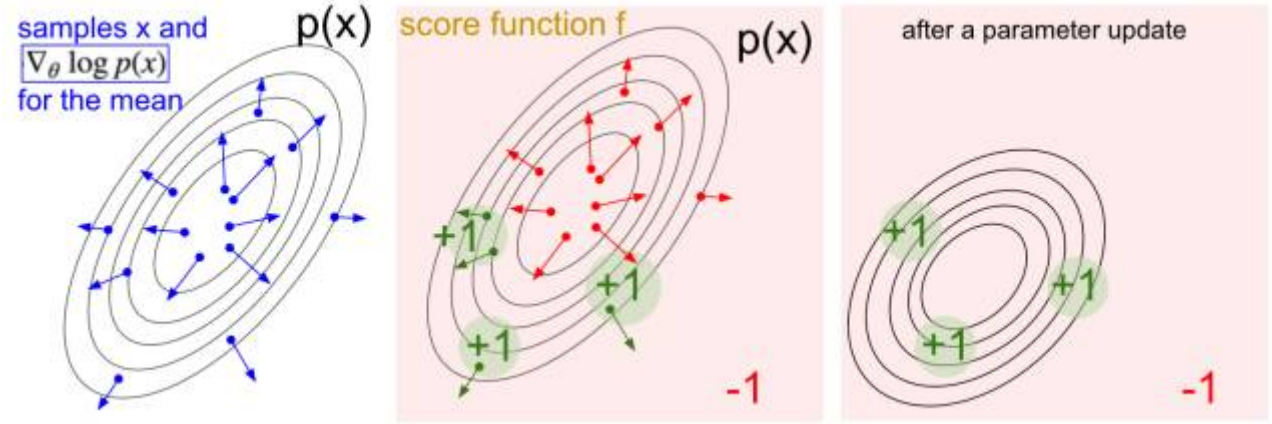
我之前承诺过要对回报做一些讨论。目前为止，我们已经根据是否最终赢得了游戏来判断了每个行动的好坏。在更通用的RL情境中，我们可能会在每个时间点都获得某种奖励 r_t。一种常见的方法是使用贴现回报（discounted reward），那么上面示意图中的“最终回报”就变成了 $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$，其中 γ 是一个处于0到1之间的数字，被称为贴现因子（discount factor，比如说0.99）。这个表达式描述了我们提高某个行动的概率的幅度是随后得到的所有回报经过加权后的总和，只不过越晚得到的回报其重要程度就呈指数级下降。在实际操作中，对这个数字进行正态化也是非常重要的。例如，假设我们为200局Pong游戏中的20000个行动都计算了 r_t，一个不错的主意是，在我们将这些回报的数字输入到反向传播中之前，先对它们进行“标准化”（比如说，将它减去平均值、再除以标准差）。通过这个方式，我们差不多总是在提高一半行动的概率、降低另一半行动的概率。从数学上来说，你也可以将这种技巧解释为一种方法来控制策略梯度估计值的方差。**更深度的解释可以点击“阅读原文”下载论文。**

策略梯度溯源

我也想稍微谈一下策略梯度从数学上是如何得到的。策略梯度是更通用的得分函数梯度估计（score function gradient estimator）的一个特例。通用的例子是，当我们有一个形式为 $E_{x\sim p(x|\theta)}[f(x)]$ 的表达式时——也就是说，某个标量得分函数 f(x) 在某种以某个 θ 为参数的概率分布 p(x;θ) 下的期望值。f(x) 将会是我们的回报函数（或者更通用地说，优势函数），而 p(x) 将会是我们的策略网络，它其实是一个 $p(a \mid I)$ 的模型，为任意图像 ***I*** 提供行动的概率分布。随后我们感兴趣的是，如何移动这个分布（通过改变它的参数 ***θ***）来提高从中抽取出的行动获得的分数（由函数f (x) 定义）？换句话说，我们要如何改变网络的参数来让抽取出的行动获得更高的回报？我们有：

$$
\begin{aligned}
\nabla_\theta E_x[f(x)] &= \nabla_\theta \sum_x p(x) f(x) && \text{definition of expectation} \\
&= \sum_x \nabla_\theta p(x) f(x) && \text{swap sum and gradient} \\
&= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) && \text{both multiply and divide by } p(x) \\
&= \sum_x p(x) \nabla_\theta \log p(x) f(x) && \text{use the fact that } \nabla_\theta \log(z) = \frac{1}{z}\nabla_\theta z \\
&= E_x[f(x) \nabla_\theta \log p(x)] && \text{definition of expectation}
\end{aligned}
$$

用语言来描述的话，我们有可以抽取行动的某种分布 p(x; ***θ***)（我用缩写 p(x) 让它看上去更整洁）。对于每一个抽取出的行动来说，我们有对它进行估算的得分函数 ***f***，向函数输入抽取出的行动后会得到一个标量分数。这个等式告诉我们，我们应该如何（通过调整参数 ***θ***）来移动分布，以此来让从中抽取出的行动获得更高的、由 ***f*** 定义的分数。详细来说，是这样的：从分布中抽取一些行动 ***x***，通过函数 ***f***(x) 估算它们的得分，也为每一个 ***x*** 估算第二个式子 $\nabla_\theta \log p(x;\theta)$。第二个式子是什么？它是一个向量——这个梯度能在参数空间中告诉我们提高一个 ***x*** 的概率的方向。也就是说，如果我们往 $\nabla_\theta \log p(x;\theta)$ 的方向稍微调整一下 ***θ***，我们会看到某个 ***x*** 的概率有了略微的提升。如果你仔细观察公式，它告诉我们应该在这个方向上乘以标量分数 f(x)。这将会使得抽取出的分数越高的动作越“用力拉拢”概率密度，所以如果我们要根据一些 ***p*** 中抽取的动作来做更新，概率密度将会网分数更高的方向移动，让取得高分的动作变得更有可能被选中。



得分函数梯度估计的可视化展示。左：高斯分布及一些从中抽取的动作（蓝点）。在每个蓝点上，我们根据高斯分布的参数平均值画出了log概率的梯度。箭头表示的是，为了提高该动作被选中的概率，高斯分布的平均值应该被“拉动”的方向。中：某个得分函数在大部分区域都给出了-1，只在小部分区域给出了+1（请注意，这可以是一个任意的、不一定可微分的标量函数）。箭头现在用了不同的颜色，在更新中我们将会对所有的红色箭头的反方向与所有的绿色箭头做平均。右：在参数完成更新后，绿色箭头和反向的红色箭头将我们拖往了高斯分布的左边和底部。现在，就像我们想要的那样，从这个分布中抽取的动作将会有更高的预期分数。

我希望我把这与RL的联系写得很明晰了。我们的策略网络为我们从分布中抽取行动，其中一些比另一些效果更好（根据优势函数的计算）。这一小块数学内容告诉我们，调整策略参数的方法是做一些展开（rollouts）、获得被抽取出的行动的梯度、用得分与之相乘并与一切其他的东西相加——就像我们刚才做的那样。

学习

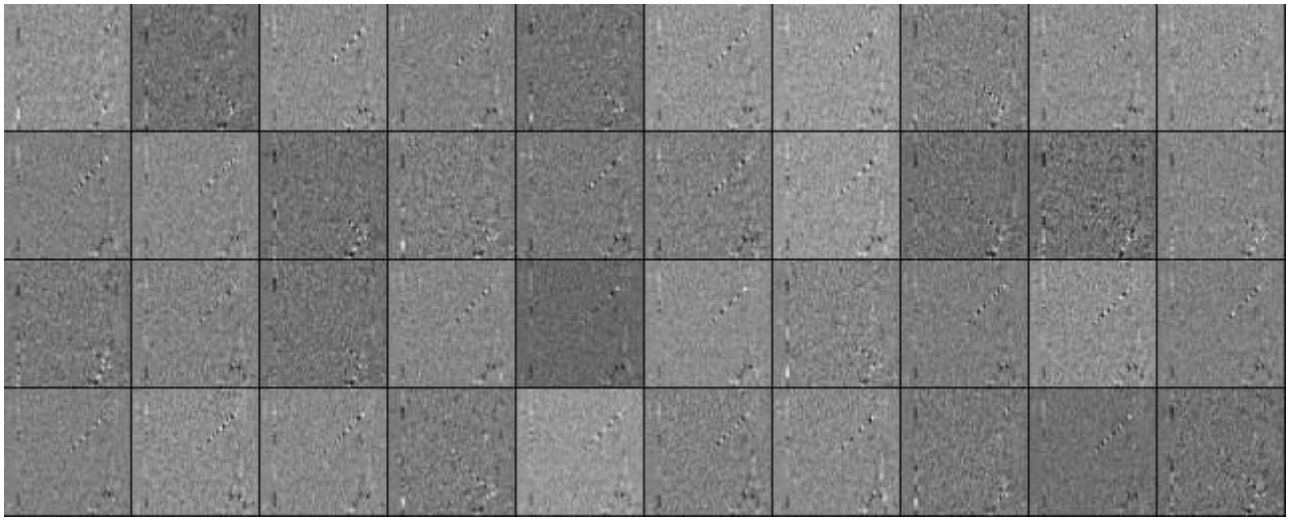
好了，我们已经有了对策略梯度的直观感受，也草草看过了它们的求导。通过一份130行的Python脚本，我实现了整个方法，它用到了OpenAI Gym的ATARI 2600 Pong。基于10次游戏的数据（每一次都是几十局游戏，因为游戏在某一方获得21分时才结束），我用RMSProp训练了一个拥有200个隐藏层单元的2层策略网络。我没对超参数做太多调整，在我的（速度很慢的）Macbook上运行了这个实验。即使如此，在训练了3个晚上之后，我得到了稍微强过AI的策略。总共进行了差不多8000次游戏，所以这个算法玩了大概200000局Pong游戏（相当多了，不是吗！）并进行了总共约800次更新。我的朋友告诉我，如果你在GPU上用ConvNets进行几天的训练，你可以更好地甩开AI选手，而如果你也仔细优化了超参数的话，你将能够完全压制AI（也就是说，每一次游戏的每一局都胜过AI）。然而，我没有花太多时间运算或是调整，所以我们最后得到了一个下面视频中那样的AI——它表现得相当不错：

学习后的代理（agent，右边的绿色挡板）与游戏内置的AI对手（左边）抗衡

习得权重

我们也可以看一下习得权重。由于经过了预处理，我们输入的每一张图像都是80*80的帧间差（当前帧减去前一帧）。现在我们可以取出W1中的每一行，将它们扩展成80*80的样子进行可视化。下图是200个神经元中的40个。白色像素是正向权重，黑色像素是负向权重。请注意，有一些神经元被调整成了特定回击的轨迹，在整条线上都是黑白交错。球只会是一个单独的像素点，所以这些神经元都在做多任务处理，对于球在这条轨迹上的多个位置都会“激发（fire）”。交替的黑与白非常有趣，

因为当球沿着轨迹移动时，神经元的活动会以sine波的样子波动，而由于ReLU的原因，它会在轨迹上离散的、不相连的位置“激发（fire）”。图像上有一些噪音，我猜想如果我用了L2正则化的话能有所缓解。



那些没有发生的事

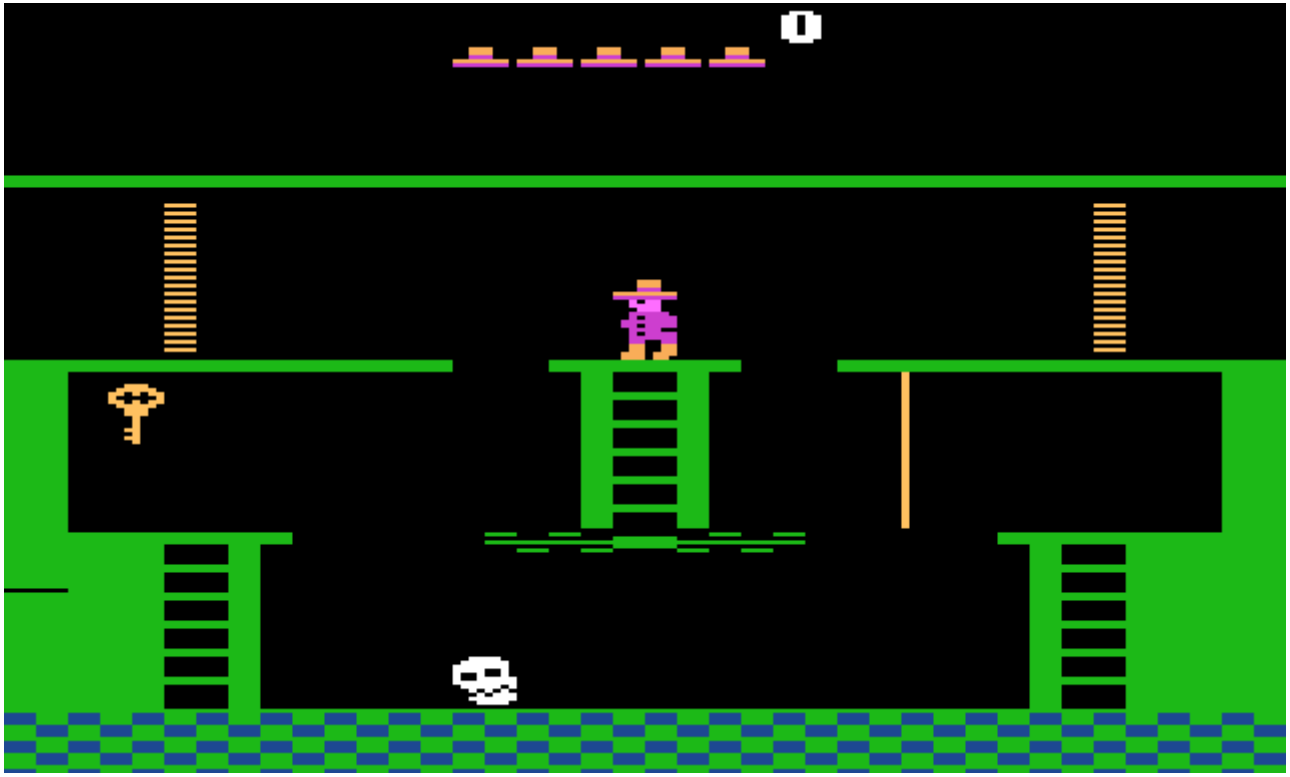
现在你做到了——我们学会了用策略梯度从原始像素中学习玩Pong游戏，并且它的表现相当不错。这种方法是“猜测并检验（guess-and-check）”的一种神奇的形式，其中“猜测”指的是从当前策略中抽取行动并进行展开（rollouts），而“检验”指的是提高带来好结果的行动的概率。不考虑一些细节问题的话，这代表了我们目前使用的最先进的用来解决强化学习问题的方法。我们可以学习这些行为，这令人印象深刻，不过如果你直观地理解了它的算法、知道它是如何工作的，你可能会多少有些失望。尤其是，它怎么没那么好呢？

让我们将它与人类学习玩Pong的方式做个比较。你把这个游戏展示给人们看，告诉他们“你控制着一块挡板，你可以把它往上移或者往下移，你的任务是把球击回对面，让AI控制的对手接不到球”之类的话，然后万事俱备。你会注意到有一些区别的存在：

- 在实际环境中，我们通常会用某种方式来进行任务的交流（这里用了英文），不过在一个标准的RL问题中，你假定有一个任意的回报函数，你需要通过与环境做交互来找出它。如果一个人类接触到Pong游戏但对它毫无了解（确实如此，特别是当回报函数是某个静态但随机的函数的时候）他在习得需要做什么时会有很大

的困难，但是不了解任务对策略梯度来说没有什么意义，甚至可能让策略梯度的表现更为出色。相似地，如果我们将每一帧的像素随机打乱，那么人类很有可能会败北，但是我们的策略梯度解决方案甚至无法察觉到这一点（如果像我们刚才那样用的是全连接网络的话）。

- 人类自带大量的先验知识，比如直觉性物理学（击中球时，球不可能瞬间转移位置、不可能突然静止、球维持着不变的运动速度等等）和直觉性心理学（AI对手“想要”获得胜利、可能会采用向球的方向移动的策略等等）。你也能理解“掌控”一块挡板的概念，明白它会根据你按的向上或是向下的按键移动。相反，我们的算法在一开始对游戏没有任何概念，这既让人印象深刻（因为它行得通）又让人失望不已（因为我们不知道如何让它行不通）。
- 策略梯度是一种暴力破解的解决方案，正确的行动最终会被发现并内化到策略中。人类则是建立一个丰富的抽象模型，并在模型内规划行动。在Pong游戏中，我可以推理出，因为对手移动得相当慢，因此用很快的速度将球击回去可能会是一个不错的策略，这可能导致对手无法及时接到球。然而，我们似乎最终也会将好的解决方案“内化”到像是肌肉反射记忆策略之类的地方。比如，如果你在学习一种新的运动任务（例如驾驶一辆手动档的汽车），你通常都会发现一开始时你会想很多，但到了最后，你的身体不用思考也能自发完成任务了。
- 策略梯度需要真正接受到正向的回报，并且以不低的频率接受正向回报，以此来缓慢地调整策略参数、更多重复带来高回报的行动。在人类的抽象模型中，即使没有真正体验过好的结果或坏的结果，我们也能分辨出哪些行动可能带来好的结果。我不需要通过开车撞墙几百次来慢慢开始避免这种行动。



《魔宫寻宝》（Montezuma’s Revenge）：这对于我们的RL算法来说是一款非常难的游戏。玩家需要先跳下中央的平台，爬到左侧平台上取得钥匙，然后开门。人类能够理解钥匙的重要性。计

算机抽样进行了数十亿次随机行动，其中99%的时候都会死掉或是被怪物杀掉。换句话说，算法难以“无意中踏入”正向回报的情景里。最近，谷歌DeepMind团队使用一个叫“intrinsic motivation”的算法，成功提高谷歌AI在《魔宫寻宝》这款游戏里的表现，试验中Deep Q在25关当中顺利通关14关。

另一款难度很高的游戏叫做《寒霜》（Frostbite），人类玩家能够理解游戏里的物体在移动，有一些物体可以触碰，有一些物体不能触碰，而游戏目标是一点一点地搭起雪屋。《搭建像人类一样学习和思考的机器》一文中很好地分析了这款游戏，并讨论了人类玩家的方法与计算机的方法的区别。

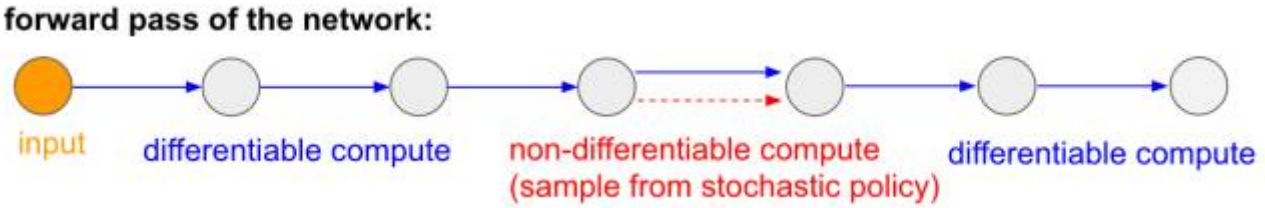
我也想强调一点，也有许多游戏是策略梯度能在其中相当轻易就击败人类的。尤其是那些有频繁的回报、需要精准的操作和迅速的反应、并且不怎么依赖长远规划的游戏，都是极为理想的，因为RL方法可以轻松“注意到”回报和行动之间短期的相关性，而策略网络能细致地优化行动执行。你可以在我们的Pong代理（agent）中窥

到一些：它发展出了一种等待球来到面前随后快速用挡板边缘击球的策略，这会让球在垂直方向有很高的速度。我们的代理（agent）通过连续重复这一策略得到了好几分。在许多ATARI游戏中，深度Q学习都通过这种方式碾压了人类表现的基线——比如弹珠台、打砖块等等。

总而言之，一旦你理解了这些算法借以运行的“小花招”，你就可以推理出它们的长处和短处。特别是，在对游戏建立抽象丰富的表征、在其中进行规划并以此进行快速学习这一方面，我们距离人类水平依然非常遥远。有一天，一台计算机会看着一串像素，从中发现一把钥匙、一扇门，然后想着去捡起钥匙然后开门也许会是个不错的主意。目前，这一目标让我们望尘莫及，试图达到那样的水平则是一个相当活跃的研究领域。

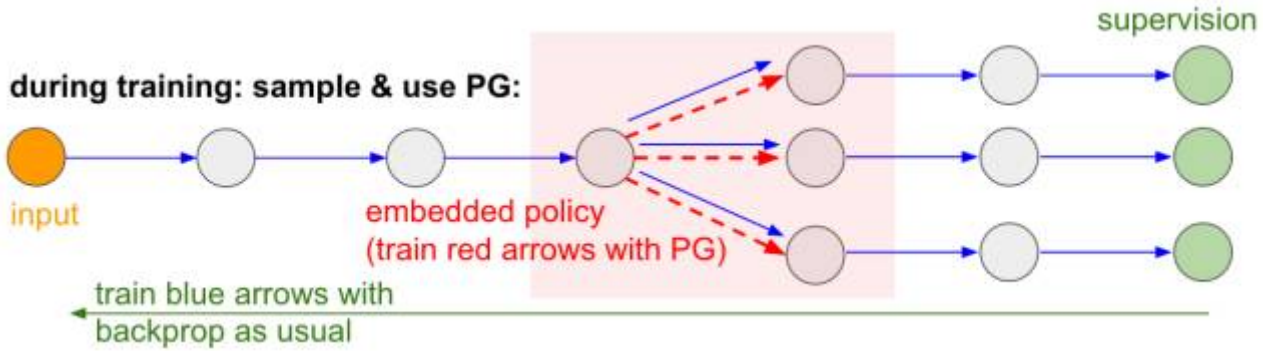
神经网络中不可微分的计算

我想提一下策略梯度与游戏无关但更有趣的一种应用：它让我们能设计和训练包含不可微分计算成分或是与不可微分计算成分有交互的神经网络。这个想法在《视觉注意的递归模型》一文中首次出现，文中的模型使用一系列来自一小片视觉注意区的低分辨率影像来处理图像（受到人类视网膜的启发）。详细来说，在每一次迭代中，一个递归神经网络（RNN）会获得一小块区域的图像，随后抽取一个位置进行下一次观看。例如，这个RNN可能看向了(5,30)的位置，获得了一小块区域的图像，随后决定看向(24,50)。这个想法的问题在于，网络中需要有一个部分生成下一次观看位置的分布、随后从中抽取。不幸的是，这个操作是不可微分的，因为，从直观上就能理解，我们不知道如果我们看向的是不同的位置会发生什么。更宽泛地来看，想象一个神经网络从输入到输出：



请注意，大部分箭头（蓝色的箭头）是通常的可微分的过程，但是其中一些表征转化可能也会包含不可微分的抽样操作（红色的箭头）。我们可以顺利地沿着蓝色箭头进行反向传播，但是红色箭头的地方反向传播无法进行。

策略梯度像救世主一样降临！我们可以把网络中执行抽样操作的部分视为一个大网络中嵌套的小型随机游走策略。因此，在训练中，我们会生成一些抽取的样本（下图中用分支来表示），随后我们提高带来好结果的那些样本的概率（在这个例子里用最后的输赢来衡量结果）。换句话说，我们可以像往常一样用反向传播训练蓝色箭头中涉及的参数，而红色箭头中涉及的参数现在将使用策略梯度进行独立的、与蓝色箭头的反向传播无关的更新，提高带来好结果的样本的概率。



可训练的内存I/O

你可以从许多其他论文中找到这个想法。比如，神经网络图灵机（Neural Turing Machine，NTM）有一个用来读写的记忆体（memory tape）。要进行一次写入操作的话，你可能会执行类似于 m[i] = x 的代码，其中i和x由一个作为控制网络的RNN来预测。然而，这个操作是不可微分的，因为没有迹象能告诉我们如果我们向另一个不为i的位置j执行写入操作的话会发生什么。因此，神经网络图灵机不得不执行软读取和软写入。它会预测一个注意力分布a（其中的元素在0到1之间，相加之和为1，峰值在我们想要写入的下标附近），随后执行 for all i: m[i] = a[i]*x。现在，这可微分了，但是我们不得不付出大量运算资源的代价，因为我们需要遍历记忆体中的每个角落来将之写入一个位置中。想象一下如果我们计算机中的每次赋值操作都需要遍历整个内存会是什么情景吧！

不过，（理论上）我们可以运用策略梯度来规避这个问题，就像强化学习神经网络图灵机那样。我们仍然要预测一个注意力分布a，但是现在我们从分布中抽取位置来执行写入操作而不是执行软写入：i = sample(a); m[i] = x。在训练中我们将会用这种方法得到小批量的i，在最后让效果最好的分支变得更有概率被选中。计算资源上的优势是巨大的，现在我们测试时只需要在某一个位置进行读写操作。然而，就像论文中指出的那样，这个策略很难成功运行，因为你需要在抽样中恰好遇见一个行得通的算法。目前的共识是，策略梯度只在选择是离散变量并且数量很少时才适用，这样它才无需在广阔的搜索空间中大海捞针。

不过，有了策略梯度、以及在有大量数据/计算力的情景中，原则上我们可以充满野心——比如，我们可以设计能够学习与大型不可微分模块交互的神经网络，比如说Latex编译器（如果你想要一个字符递归神经网络来生成能够编译的Latex)。或者是SLAM系统，或者是求解LQR，或者是其他什么东西。又或者，比如说，为了攫取掌控全球所需的关键信息，一个超级智能可能会想要学习如何通过TCP/IP（这不幸是不可微分的）与互联网交互。

结论

我们可以看到，策略梯度是一种强大并且通用的算法，这从例子中我们用130行Python代码从零开始使用原始像素训练了一个ATARI Pong游戏的代理（agent）可以看出。而在更为通用的情景下，这个算法同样还可以为任意的游戏训练代理（agent），甚至终有一日可能应用于实际生活中许多非常有价值的控制问题上。在此，我想以一些笔记收尾：

推进人工智能发展

我们看到了这个算法首先使用暴力搜索的方式随机跳跃地进行搜索，并且必须至少一次偶然地进入回报情景（rewarding situation），最好能经常进入，然后不停地重复这个过程直至策略分布调整了它的参数来增加目标行为的重复概率。我们也看到了人类在解决这类问题的方法与计算机大相径庭，更像是在快速地建立抽象模型——我们在研究中甚至尚未勘破这种方法的表面（虽然有许多人都在不断尝试）。因为这些抽象模型非常难以（如果不说不可能的话）进行明确的标注，这也是为什么最近人们投注在（非监督式）生成模型和程序归纳方面的兴趣越来越多。

复杂机器人环境中的应用

多个算法不能简单地提升规模后直接应用于难以获得大量探索结果的环境中。例如，在机器人环境中， 可能有一个（或多个）机器人与世界在进行实时交互。这令我们无法将我在这篇文章中提到的算法直接简单地移植应用。与此相关的一类试图缓解这个问题的研究是确定性策略梯度（deterministic policy gradients）——不同于随机游走策略中对样本的需求和对更高的分数的追求，这个方法使用的是确定性策略，直接从与得分函数相似的二级网络（被称为 critic）中获取梯度信息。它原则上在那些行动的高维度非常高、从分布中抽样无法达到令人满意的覆盖面的环境中会比一般的策略梯度有更高的效率，但是到目前为止，这种方法在实际操作中看上去要求太多而难以实行。另一种相关的方法是通过提升机器人的规模，正如我们将会看到的Google的机械臂园地（robot arm farm），甚至可能是Tesla的 Model S + Autopilot。

另一个研究方向是通过增加额外的监督来试图让搜索过程减少毫无头绪的搜索。例如，在很多实际案例中，你可以从人类专家那里获得借鉴。比如，AlphaGo首先使用监督式学习来从人类专家的围棋棋谱中学习预测人类的行动，随之而来的模仿人类行动的策略则随后用策略梯度针对赢得围棋比赛的“真实”目标进行了精细的调整。在一些案例中， 你能获得的专家经验可能很少（例如机器人遥控操作方面），而有一些技巧可以让你通过学徒学习（apprenticeship learning）来利用这种数据。最后，如果你没有任何来自于人类的监督式数据（supervised data），也有一些例子运用了代价高昂的优化技巧来进行计算，比如知名的动态模型（例如物理模拟器中的 F=ma）中的轨迹优化；另外一些例子里学习的是局部近似动态模型，你可以在前景广阔的引导式策略搜索（Guided Policy Search）框架中看到这种运用。

将策略梯度应用于实际

作为最后一条笔记，我打算做一些早就想在我的RNN博客中做的事。我想我应该已经已经把RNN塑造成了一个非常神奇、能自动解决任意的序列问题的模型的形象。事实是，让这些模型真正运作起来是非常棘手的一件事，谨慎小心和专业知识都不可或缺，并且在很多时候中也可能出现杀鸡用牛刀的情况——一些简单得多的模型，能为你做到超过90%的效果。策略梯度同样如此，它不是自动化的：你需要大量样本，不停训练，效果糟糕时也难以除错。一个人在打火箭炮的主意之前总是应该先学会用气枪才对。**在强化学习这个例子中，一个不能省略的底线是，首先应该试着使用交叉熵方法（CEM），这是一个从进化中脱胎的随机游走的爬山式"猜测和检验"方法。**如果你坚持使用策略梯度来解决你的问题，请确保你对于论文中的技巧（tricks）看得非常仔细，从简单的开始尝试，用一种被称为TRPO的策略梯度的变体。这种方法往往比原本的PG效果更好、一致性更高。核心思想是，避免那些使你的策略改变太多的参数更新，让你的策略在同一批次数据上新旧两种分布的KL差异受到强制限制（这种想法最简单的实例不是共轭梯度法，而是进行线性搜索并始终核查KL）。

好，就是这样！我希望我让你们体验到了强化学习进展到了什么程度，面临什么挑战，并且如果你希望帮助强化学习进一步发展，我邀请你在OpenAI Gym中这么做 :)

下次见！

扫描二维码关注

阅读原文