第八章　深度模型中的优化

深度学习算法在许多情况下都涉及到优化。例如，模型中的进行推断（如PCA）涉及到求解优化问题。我们经常使用解析优化去证明或设计算法。在深度学习涉及到的诸多优化问题中，最难的是神经网络训练。甚至是用几百台机器投入几天到几个月来解决单个神经网络训练问题，也是很常见的。因为这其中的优化问题很重要，代价也很高，因此研究者们开发了一组专门为此设计的优化技术。本章会介绍神经网络训练中的这些优化技术。

如果你不熟悉基于梯度优化的基本原则，我们建议回顾第四章。该章简要概述了一般的数值优化。

本章主要关注这一类特定的优化问题：寻找神经网络上的一组参数 $\boldsymbol{\theta}$，它能显著地降低代价函数 $J(\boldsymbol{\theta})$，该代价函数通常包括整个训练集上的性能评估和额外的正则化项。

首先，我们会介绍在机器学习任务中作为训练算法使用的优化与纯优化有哪些不同。接下来，我们会介绍导致神经网络优化困难的几个具体挑战。然后，我们会介绍几个实用算法，包括优化算法本身和初始化参数的策略。更高级的算法能够在训练中自适应调整学习率，或者使用代价函数二阶导数包含的信息。最后，我们会介绍几个将简单优化算法结合成高级过程的优化策略，以此作为总结。

8.1　学习和纯优化有什么不同

用于深度模型训练的优化算法与传统的优化算法在几个方面有所不同。机器学习通常是间接作用的。在大多数机器学习问题中，我们关注某些性能度量 P，其定义于测试集上并且可能是不可解的。因此，我们只是间接地优化 P。我们希望通过

降低代价函数 $J(\boldsymbol{\theta})$ 来提高 P。这一点与纯优化不同，纯优化最小化目标 J 本身。训练深度模型的优化算法通常也会包括一些针对机器学习目标函数的特定结构进行的特化。

通常，代价函数可写为训练集上的平均，如

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},\mathrm{y})\sim\hat{p}_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y), \tag{8.1}$$

其中 L 是每个样本的损失函数，$f(\boldsymbol{x};\boldsymbol{\theta})$ 是输入 $\boldsymbol{x}$ 时所预测的输出，$\hat{p}_{\mathrm{data}}$ 是经验分布。监督学习中，y 是目标输出。在本章中，我们会介绍不带正则化的监督学习，L 的变量是 $f(\boldsymbol{x};\boldsymbol{\theta})$ 和 y。不难将这种监督学习扩展成其他形式，如包括 $\boldsymbol{\theta}$ 或者 $\boldsymbol{x}$ 作为参数，或是去掉参数 y，以发展不同形式的正则化或是无监督学习。

式 (8.1) 定义了训练集上的目标函数。通常，我们更希望最小化取自*数据生成分布* p_{data} 的期望，而不仅仅是有限训练集上的对应目标函数：

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x},\mathrm{y})\sim p_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y). \tag{8.2}$$

8.1.1 经验风险最小化

机器学习算法的目标是降低式 (8.2) 所示的期望泛化误差。这个数据量被称为**风险**（risk）。在这里，我们强调该期望取自真实的潜在分布 p_{data}。如果我们知道了真实分布 $p_{\mathrm{data}}(\boldsymbol{x}, y)$，那么最小化风险变成了一个可以被优化算法解决的优化问题。然而，我们遇到的机器学习问题，通常是不知道 $p_{\mathrm{data}}(\boldsymbol{x}, y)$，只知道训练集中的样本。

将机器学习问题转化回一个优化问题的最简单方法是最小化训练集上的期望损失。这意味着用训练集上的经验分布 $\hat{p}(\boldsymbol{x}, y)$ 替代真实分布 $p(\boldsymbol{x}, y)$。现在，我们将最小化**经验风险**（empirical risk）：

$$\mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}), \tag{8.3}$$

其中 m 表示训练样本的数目。

基于最小化这种平均训练误差的训练过程被称为**经验风险最小化**（empirical risk minimization）。在这种情况下，机器学习仍然和传统的直接优化很相似。我们并不直接最优化风险，而是最优化经验风险，希望也能够很大地降低风险。一系列不同的理论构造了一些条件，使得在这些条件下真实风险的期望可以下降不同的量。

然而，经验风险最小化很容易导致过拟合。高容量的模型会简单地记住训练集。在很多情况下，经验风险最小化并非真的可行。最有效的现代优化算法是基于梯度下降的，但是很多有用的损失函数，如 $0-1$ 损失，没有有效的导数（导数要么为零，要么处处未定义）。这两个问题说明，在深度学习中我们很少使用经验风险最小化。反之，我们会使用一个稍有不同的方法，我们真正优化的目标会更加不同于我们希望优化的目标。

8.1.2 代理损失函数和提前终止

有时，我们真正关心的损失函数（比如分类误差）并不能被高效地优化。例如，即使对于线性分类器而言，精确地最小化 $0-1$ 损失通常是不可解的（复杂度是输入维数的指数级别）(Marcotte and Savard, 1992)。在这种情况下，我们通常会优化**代理损失函数**（surrogate loss function）。代理损失函数作为原目标的代理，还具备一些优点。例如，正确类别的负对数似然通常用作 $0-1$ 损失的替代。负对数似然允许模型估计给定样本的类别的条件概率，如果该模型效果好，那么它能够输出期望最小分类误差所对应的类别。

在某些情况下，代理损失函数比原函数学到的更多。例如，使用对数似然替代函数时，在训练集上的 $0-1$ 损失达到 0 之后，测试集上的 $0-1$ 损失还能持续下降很长一段时间。这是因为即使 $0-1$ 损失期望是零时，我们还能拉开不同类别的距离以改进分类器的鲁棒性，获得一个更强壮的、更值得信赖的分类器，从而，相对于简单地最小化训练集上的平均 $0-1$ 损失，它能够从训练数据中抽取更多信息。

一般的优化和我们用于训练算法的优化有一个重要不同：训练算法通常不会停止在局部极小点。反之，机器学习通常优化代理损失函数，但是在基于提前终止（第 7.8 节）的收敛条件满足时停止。通常，提前终止使用真实潜在损失函数，如验证集上的 $0-1$ 损失，并设计为在过拟合发生之前终止。与纯优化不同的是，提前终止时代理损失函数仍然有较大的导数，而纯优化终止时导数较小。

8.1.3 批量算法和小批量算法

机器学习算法和一般优化算法不同的一点是，机器学习算法的目标函数通常可以分解为训练样本上的求和。机器学习中的优化算法在计算参数的每一次更新时通常仅使用整个代价函数中一部分项来估计代价函数的期望值。

例如，最大似然估计问题可以在对数空间中分解成各个样本的总和：

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\mathrm{model}}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}). \tag{8.4}$$

最大化这个总和等价于最大化训练集在经验分布上的期望：

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(\boldsymbol{x}, y; \boldsymbol{\theta}). \tag{8.5}$$

优化算法用到的目标函数J 中的大多数属性也是训练集上的期望。例如，最常用的属性是梯度：

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\mathrm{model}}(\boldsymbol{x}, y; \boldsymbol{\theta}). \tag{8.6}$$

准确计算这个期望的计算代价非常大，因为我们需要在整个数据集上的每个样本上评估模型。在实践中，我们可以从数据集中随机采样少量的样本，然后计算这些样本上的平均值。

回想一下，n 个样本均值的标准差（式 (5.46)）是 $\sigma/\sqrt{n}$，其中 σ 是样本值真实的标准差。分母 $\sqrt{n}$ 表明使用更多样本来估计梯度的方法的回报是低于线性的。比较两个假想的梯度计算，一个基于 100 个样本，另一个基于 10,000 个样本。后者需要的计算量是前者的 100 倍，但却只降低了 10 倍的均值标准差。如果能够快速地计算出梯度估计值，而不是缓慢地计算准确值，那么大多数优化算法会收敛地更快（就总的计算量而言，而不是指更新次数）。

另一个促使我们从小数目样本中获得梯度的统计估计的动机是训练集的冗余。在最坏的情况下，训练集中所有的 m 个样本都是彼此相同的拷贝。基于采样的梯度估计可以使用单个样本计算出正确的梯度，而比原来的做法少花了 m 倍时间。实践中，我们不太可能真的遇到这种最坏情况，但我们可能会发现大量样本都对梯度做出了非常相似的贡献。

使用整个训练集的优化算法被称为 **批量**（batch）或 **确定性**（deterministic）梯度算法，因为它们会在一个大批量中同时处理所有样本。这个术语可能有点令人困惑，因为这个词 “批量” 也经常被用来描述小批量随机梯度下降算法中用到的小批量样本。通常，术语 “批量梯度下降” 指使用全部训练集，而术语 “批量” 单独出现时指一组样本。例如，我们普遍使用术语 “批量大小” 表示小批量的大小。

每次只使用单个样本的优化算法有时被称为 **随机**（stochastic）或者 **在线**（online）算法。术语 “在线” 通常是指从连续产生样本的数据流中抽取样本的情况，而不是从一个固定大小的训练集中遍历多次采样的情况。

大多数用于深度学习的算法介于以上两者之间，使用一个以上，而又不是全部的训练样本。传统上，这些会被称为 **小批量**（minibatch）或 **小批量随机**（minibatch stochastic）方法，现在通常将它们简单地称为 **随机**（stochastic）方法。

随机方法的典型示例是随机梯度下降，这将在第 8.3.1 节中详细描述。

小批量的大小通常由以下几个因素决定：

- 更大的批量会计算更精确的梯度估计，但是回报却是小于线性的。
- 极小批量通常难以充分利用多核架构。这促使我们使用一些绝对最小批量，低于这个值的小批量处理不会减少计算时间。
- 如果批量处理中的所有样本可以并行地处理（通常确是如此），那么内存消耗和批量大小会正比。对于很多硬件设施，这是批量大小的限制因素。
- 在某些硬件上使用特定大小的数组时，运行时间会更少。尤其是在使用 GPU 时，通常使用 2 的幂数作为批量大小可以获得更少的运行时间。一般，2 的幂数的取值范围是 32 到 256，16 有时在尝试大模型时使用。
- 可能是由于小批量在学习过程中加入了噪声，它们会有一些正则化效果 (Wilson and Martinez, 2003)。泛化误差通常在批量大小为 1 时最好。因为梯度估计的高方差，小批量训练需要较小的学习率以保持稳定性。因为降低的学习率和消耗更多步骤来遍历整个训练集都会产生更多的步骤，所以会导致总的运行时间非常大。

不同的算法使用不同的方法从小批量中获取不同的信息。有些算法对采样误差比其他算法更敏感，这通常有两个可能原因。一个是它们使用了很难在少量样本上精确估计的信息，另一个是它们以放大采样误差的方式使用了信息。仅基于梯度 $\boldsymbol{g}$ 的更新方法通常相对鲁棒，并能使用较小的批量获得成功，如 100。使用Hessian矩阵 $\boldsymbol{H}$，计算如 $\boldsymbol{H}^{-1}\boldsymbol{g}$ 更新的二阶方法通常需要更大的批量，如 $10,000$。这些大批量需要最小化估计 $\boldsymbol{H}^{-1}\boldsymbol{g}$ 的波动。假设 $\boldsymbol{H}$ 被精确估计，但是有病态条件数。乘以 $\boldsymbol{H}$ 或是其逆会放大之前存在的误差（这个示例中是指 $\boldsymbol{g}$ 的估计误差）。即使 $\boldsymbol{H}$ 被精确估计，$\boldsymbol{g}$ 中非常小的变化也会导致更新值 $\boldsymbol{H}^{-1}\boldsymbol{g}$ 中非常大的变化。当然，我们通常只会近似地估计 $\boldsymbol{H}$，因此相对于我们使用具有较差条件的操作去估计 $\boldsymbol{g}$，更新 $\boldsymbol{H}^{-1}\boldsymbol{g}$ 会含有更多的误差。

小批量是随机抽取的这点也很重要。从一组样本中计算出梯度期望的无偏估计要求这些样本是独立的。我们也希望两个连续的梯度估计是互相独立的，因此两个连续的小批量样本也应该是彼此独立的。很多现实的数据集自然排列，从而使得连续的样本之间具有高度相关性。例如，假设我们有一个很长的血液样本测试结果清单。清单上的数据有可能是这样获取的，头五个血液样本于不同时间段取自第一个病人，接下来三个血液样本取自第二个病人，再随后的血液样本取自第三个病人，等等。如果我们从这个清单上顺序抽取样本，那么我们的每个小批量数据的偏差都很大，因为这个小批量很可能只代表着数据集上众多患者中的某一个患者。在这种数据集中的顺序有很大影响的情况下，很有必要在抽取小批量样本前打乱样本顺序。对于非常大的数据集，如数据中心含有几十亿样本的数据集，我们每次构建小批量样本时都将样本完全均匀地抽取出来是不太现实的。幸运的是，实践中通常将样本顺序打乱一次，然后按照这个顺序存储起来就足够了。之后训练模型时会用到的一组组小批量连续样本是固定的，每个独立的模型每次遍历训练数据时都会重复使用这个顺序。然而，这种偏离真实随机采样的方法并没有很严重的有害影响。不以某种方式打乱样本顺序才会极大地降低算法的性能。

很多机器学习上的优化问题都可以分解成并行地计算不同样本上单独的更新。换言之，我们在计算小批量样本 $\boldsymbol{X}$ 上最小化 $J(\boldsymbol{X})$ 的更新时，同时可以计算其他小批量样本上的更新。这类异步并行分布式方法将在第 12.1.3 节中进一步讨论。

小批量随机梯度下降的一个有趣动机是，只要没有重复使用样本，它将遵循着真实*泛化误差*（式 (8.2)）的梯度。很多小批量随机梯度下降方法的实现都会打乱数据顺序一次，然后多次遍历数据来更新参数。第一次遍历时，每个小批量样本都用来计算真实泛化误差的无偏估计。第二次遍历时，估计将会是有偏的，因为它重新抽取了已经用过的样本，而不是从和原先样本相同的数据生成分布中获取新的无偏的样本。

我们不难从在线学习的情况中看出随机梯度下降最小化泛化误差的原因。这时样本或者小批量都是从数据**流**（stream）中抽取出来的。换言之，学习器好像是一个每次看到新样本的人，每个样本 $(\boldsymbol{x}, y)$ 都来自数据生成分布 $p_{\text{data}}(\boldsymbol{x}, y)$，而不是使用大小固定的训练集。这种情况下，样本永远不会重复；每次更新的样本是从分布 p_{data} 中采样获得的无偏样本。

在 $\boldsymbol{x}$ 和 y 是离散时，以上的等价性很容易得到。在这种情况下，泛化误差

（式 (8.2)）可以表示为

$$J^*(\boldsymbol{\theta}) = \sum_{\boldsymbol{x}} \sum_{y} p_{\text{data}}(\boldsymbol{x}, y) L(f(\boldsymbol{x}; \boldsymbol{\theta}), y), \tag{8.7}$$

上式的准确梯度为

$$\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}) = \sum_{\boldsymbol{x}} \sum_{y} p_{\text{data}}(\boldsymbol{x}, y) \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}; \boldsymbol{\theta}), y). \tag{8.8}$$

在式 (8.5) 和式 (8.6) 中，我们已经在对数似然中看到了相同的结果；现在我们发现这一点在包括似然的其他函数 L 上也是成立的。在一些关于 p_{data} 和 L 的温和假设下，在 $\boldsymbol{x}$ 和 y 是连续时也能得到类似的结果。

因此，我们可以从数据生成分布 p_{data} 抽取小批量样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 以及对应的目标 $y^{(i)}$，然后计算该小批量上损失函数关于对应参数的梯度

$$\hat{\boldsymbol{g}} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}). \tag{8.9}$$

以此获得泛化误差准确梯度的无偏估计。最后，在泛化误差上使用SGD 方法在方向 $\hat{\boldsymbol{g}}$ 上更新 $\boldsymbol{\theta}$。

当然，这个解释只能用于样本没有重复使用的情况。然而，除非训练集特别大，通常最好是多次遍历训练集。当多次遍历数据集更新时，只有第一遍满足泛化误差梯度的无偏估计。但是，额外的遍历更新当然会由于减小训练误差而得到足够的好处，以抵消其带来的训练误差和测试误差间差距的增加。

随着数据集的规模迅速增长，超越了计算能力的增速，机器学习应用每个样本只使用一次的情况变得越来越常见，甚至是不完整地使用训练集。在使用一个非常大的训练集时，过拟合不再是问题，而欠拟合和计算效率变成了主要的顾虑。读者也可以参考 Bottou and Bousquet (2008a) 中关于训练样本数目增长时，泛化误差上计算瓶颈影响的讨论。

8.2 神经网络优化中的挑战

优化通常是一个极其困难的任务。传统的机器学习会小心设计目标函数和约束，以确保优化问题是凸的，从而避免一般优化问题的复杂度。在训练神经网络时，我们肯定会遇到一般的非凸情况。即使是凸优化，也并非没有任何问题。在这一节中，我们会总结几个训练深度模型时会涉及到的主要挑战。

8.2.1 病态

在优化凸函数时，会遇到一些挑战。这其中最突出的是Hessian矩阵 $\boldsymbol{H}$ 的病态。这是数值优化、凸优化或其他形式的优化中普遍存在的问题，更多细节请回顾第4.3.1节。

病态问题一般被认为存在于神经网络训练过程中。病态体现在随机梯度下降会“卡”在某些情况，此时即使很小的更新步长也会增加代价函数。

回顾式(4.9)，代价函数的二阶泰勒级数展开预测梯度下降中的 $-\epsilon \boldsymbol{g}$ 会增加

$$\frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g} - \epsilon \boldsymbol{g}^\top \boldsymbol{g} \tag{8.10}$$

到代价中。当 $\frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ 超过 $\epsilon \boldsymbol{g}^\top \boldsymbol{g}$ 时，梯度的病态会成为问题。判断病态是否不利于神经网络训练任务，我们可以监测平方梯度范数 $\boldsymbol{g}^\top \boldsymbol{g}$ 和 $\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$。在很多情况中，梯度范数不会在训练过程中显著缩小，但是 $\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ 的增长会超过一个数量级。其结果是尽管梯度很强，学习会变得非常缓慢，因为学习率必须收缩以弥补更强的曲率。如图8.1所示，成功训练的神经网络中，梯度显著增加。

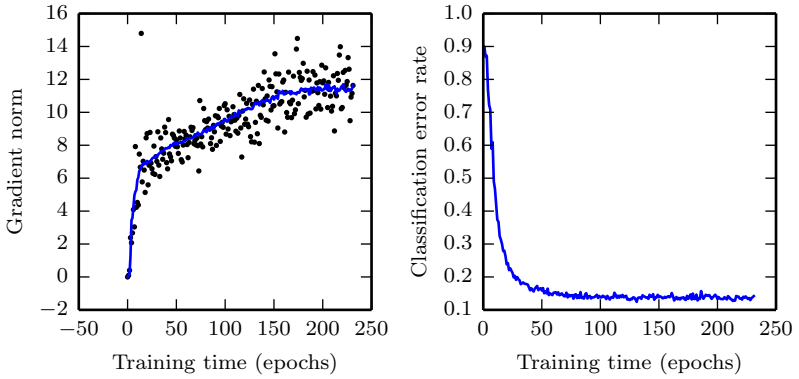

图 8.1: 梯度下降通常不会到达任何类型的临界点。此示例中，在用于对象检测的卷积网络的整个训练期间，梯度范数持续增加。*(左)* 各个梯度计算的范数如何随时间分布的散点图。为了方便作图，每轮仅绘制一个梯度范数。我们将所有梯度范数的移动平均绘制为实曲线。梯度范数明显随时间增加，而不是如我们所期望的那样随训练过程收敛到临界点而减小。*(右)* 尽管梯度递增，训练过程却相当成功。验证集上的分类误差可以降低到较低水平。

尽管病态还存在于除了神经网络训练的其他情况中，有些适用于其他情况的解决病态的技术并不适用于神经网络。例如，牛顿法在解决带有病态条件的Hessian矩

阵的凸优化问题时，是一个非常优秀的工具，但是我们将会在以下小节中说明牛顿法运用到神经网络时需要很大的改动。

8.2.2 局部极小值

凸优化问题的一个突出特点是其可以简化为寻找一个局部极小点的问题。任何一个局部极小点都是全局最小点。有些凸函数的底部是一个平坦的区域，而不是单一的全局最小点，但该平坦区域中的任意点都是一个可以接受的解。优化一个凸问题时，若发现了任何形式的临界点，我们都会知道已经找到了一个不错的可行解。

对于非凸函数时，如神经网络，有可能会存在多个局部极小值。事实上，几乎所有的深度模型基本上都会有非常多的局部极小值。然而，我们会发现这并不是主要问题。

由于**模型可辨识性**（model identifiability）问题，神经网络和任意具有多个等效参数化潜变量的模型都会具有多个局部极小值。如果一个足够大的训练集可以唯一确定一组模型参数，那么该模型被称为可辨认的。带有潜变量的模型通常是不可辨认的，因为通过相互交换潜变量我们能得到等价的模型。例如，考虑神经网络的第一层，我们可以交换单元 i 和单元 j 的传入权重向量、传出权重向量而得到等价的模型。如果神经网络有 m 层，每层有 n 个单元，那么会有 $n!^m$ 种排列隐藏单元的方式。这种不可辨认性被称为**权重空间对称性**（weight space symmetry）。

除了权重空间对称性，很多神经网络还有其他导致不可辨认的原因。例如，在任意整流线性网络或者 maxout 网络中，我们可以将传入权重和偏置扩大 α 倍，然后将传出权重扩大 $\frac{1}{\alpha}$ 倍，而保持模型等价。这意味着，如果代价函数不包括如权重衰减这种直接依赖于权重而非模型输出的项，那么整流线性网络或者 maxout 网络的每一个局部极小点都在等价的局部极小值的 $(m \times n)$ 维双曲线上。

这些模型可辨识性问题意味着神经网络代价函数具有非常多、甚至不可数无限多的局部极小值。然而，所有这些由于不可辨识性问题而产生的局部极小值都有相同的代价函数值。因此，这些局部极小值并非是非凸所带来的问题。

如果局部极小值相比全局最小点拥有很大的代价，局部极小值会带来很大的隐患。我们可以构建没有隐藏单元的小规模神经网络，其局部极小值的代价比全局最小点的代价大很多 (Sontag and Sussman, 1989; Brady *et al.*, 1989; Gori and Tesi, 1992)。如果具有很大代价的局部极小值是常见的，那么这将给基于梯度的优化算法

带来极大的问题。

对于实际中感兴趣的网络，是否存在大量代价很高的局部极小值，优化算法是否会碰到这些局部极小值，都是尚未解决的公开问题。多年来，大多数从业者认为局部极小值是困扰神经网络优化的常见问题。如今，情况有所变化。这个问题仍然是学术界的热点问题，但是学者们现在猜想，对于足够大的神经网络而言，大部分局部极小值都具有很小的代价函数，我们能不能找到真正的全局最小点并不重要，而是需要在参数空间中找到一个代价很小（但不是最小）的点 (Saxe *et al.*, 2013; Dauphin *et al.*, 2014; Goodfellow *et al.*, 2015; Choromanska *et al.*, 2014)。

很多从业者将神经网络优化中的所有困难都归结于局部极小值。我们鼓励从业者要仔细分析特定的问题。一种能够排除局部极小值是主要问题的检测方法是画出梯度范数随时间的变化。如果梯度范数没有缩小到一个微小的值，那么该问题既不是局部极小值，也不是其他形式的临界点。在高维空间中，很难明确证明局部极小值是导致问题的原因。许多并非局部极小值的结构也具有很小的梯度。

8.2.3 高原、鞍点和其他平坦区域

对于很多高维非凸函数而言，局部极小值（以及极大值）事实上都远少于另一类梯度为零的点：鞍点。鞍点附近的某些点比鞍点有更大的代价，而其他点则有更小的代价。在鞍点处，Hessian 矩阵同时具有正负特征值。位于正特征值对应的特征向量方向的点比鞍点有更大的代价，反之，位于负特征值对应的特征向量方向的点有更小的代价。我们可以将鞍点视为代价函数某个横截面上的局部极小点，同时也可以视为代价函数某个横截面上的局部极大点。图 4.5 给了一个示例。

多类随机函数表现出以下性质：低维空间中，局部极小值很普遍。在更高维空间中，局部极小值很罕见，而鞍点则很常见。对于这类函数 $f : \mathbb{R}^n \to \mathbb{R}$ 而言，鞍点和局部极小值的数目比率的期望随 n 指数级增长。我们可以从直觉上理解这种现象——Hessian 矩阵在局部极小点处只有正特征值。而在鞍点处，Hessian 矩阵则同时具有正负特征值。试想一下，每个特征值的正负号由抛硬币决定。在一维情况下，很容易抛硬币得到正面朝上一次而获取局部极小点。在 n-维空间中，要抛掷 n 次硬币都正面朝上的难度是指数级的。具体可以参考 Dauphin *et al.* (2014)，它回顾了相关的理论工作。

很多随机函数一个惊人性质是，当我们到达代价较低的区间时，Hessian 矩阵的特征值为正的可能性更大。和抛硬币类比，这意味着如果我们处于低代价的临界

点时，抛掷硬币正面朝上 n 次的概率更大。这也意味着，局部极小值具有低代价的可能性比高代价要大得多。具有高代价的临界点更有可能是鞍点。具有极高代价的临界点就很可能是局部极大值了。

以上现象出现在许多种类的随机函数中。那么是否在神经网络中也有发生呢？Baldi and Hornik (1989) 从理论上证明，不具非线性的浅层自编码器（第十四章中将介绍的一种将输出训练为输入拷贝的前馈网络）只有全局极小值和鞍点，没有代价比全局极小值更大的局部极小值。他们还发现这些结果能够扩展到不具非线性的更深的网络上，不过没有证明。这类网络的输出是其输入的线性函数，但它们仍然有助于分析非线性神经网络模型，因为它们的损失函数是关于参数的非凸函数。这类网络本质上是多个矩阵组合在一起。Saxe *et al.* (2013) 精确解析了这类网络中完整的学习动态，表明这些模型的学习能够捕捉到许多在训练具有非线性激活函数的深度模型时观察到的定性特征。Dauphin *et al.* (2014) 通过实验表明，真实的神经网络也存在包含很多高代价鞍点的损失函数。Choromanska *et al.* (2014) 提供了额外的理论论点，表明另一类和神经网络相关的高维随机函数也满足这种情况。

鞍点激增对于训练算法来说有哪些影响呢？对于只使用梯度信息的一阶优化算法而言，目前情况还不清楚。鞍点附近的梯度通常会非常小。另一方面，实验中梯度下降似乎可以在许多情况下逃离鞍点。Goodfellow *et al.* (2015) 可视化了最新神经网络的几个学习轨迹，图 8.2 给了一个例子。这些可视化显示，在突出的鞍点附近，代价函数都是平坦的，权重都为零。但是他们也展示了梯度下降轨迹能够迅速逸出该区间。Goodfellow *et al.* (2015) 也主张，应该可以通过分析来表明连续时间的梯度下降会逃离而不是吸引到鞍点，但对梯度下降更现实的使用场景来说，情况或许会有所不同。

对于牛顿法而言，鞍点显然是一个问题。梯度下降旨在朝“下坡”移动，而非明确寻求临界点。而牛顿法的目标是寻求梯度为零的点。如果没有适当的修改，牛顿法就会跳进一个鞍点。高维空间中鞍点的激增或许解释了在神经网络训练中为什么二阶方法无法成功取代梯度下降。Dauphin *et al.* (2014) 介绍了二阶优化的**无鞍牛顿法**（saddle-free Newton method），并表明和传统算法相比有显著改进。二阶方法仍然难以扩展到大型神经网络，但是如果这类无鞍算法能够扩展的话，还是很有希望的。

除了极小值和鞍点，还存在其他梯度为零的点。例如从优化的角度看与鞍点很相似的极大值，很多算法不会被吸引到极大值，除了未经修改的牛顿法。和极小值一样，许多种类的随机函数的极大值在高维空间中也是指数级稀少。

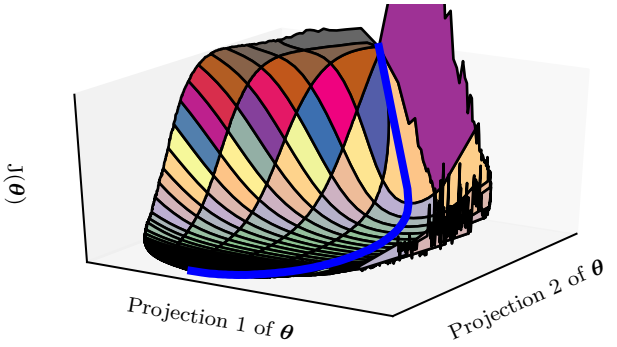

图 8.2: 神经网络代价函数的可视化。这些可视化对应用于真实对象识别和自然语言处理任务的前馈神经网络、卷积网络和循环网络而言是类似的。令人惊讶的是，这些可视化通常不会显示出很多明显的障碍。大约 2012 年，在随机梯度下降开始成功训练非常大的模型之前，相比这些投影所显示的神经网络代价函数的表面通常被认为有更多的非凸结构。该投影所显示的主要障碍是初始参数附近的高代价鞍点，但如由蓝色路径所示，SGD 训练轨迹能轻易地逃脱该鞍点。大多数训练时间花费在横穿代价函数中相对平坦的峡谷，可能由于梯度中的高噪声、或该区域中 Hessian 矩阵的病态条件，或者需要经过间接的弧路径绕过图中可见的高“山”。图经 Goodfellow *et al.* (2015) 许可改编。

也可能存在恒值的、宽且平坦的区域。在这些区域，梯度和 Hessian 矩阵都是零。这种退化的情形是所有数值优化算法的主要问题。在凸问题中，一个宽而平坦的区间肯定包含全局极小值，但是对于一般的优化问题而言，这样的区域可能会对应着目标函数中一个较高的值。

8.2.4 悬崖和梯度爆炸

多层神经网络通常存在像悬崖一样的斜率较大区域，如图 8.3 所示。这是由于几个较大的权重相乘导致的。遇到斜率极大的悬崖结构时，梯度更新会很大程度地改变参数值，通常会完全跳过这类悬崖结构。

不管我们是从上还是从下接近悬崖，情况都很糟糕，但幸运的是我们可以用使用第 10.11.1 节介绍的启发式 **梯度截断**（gradient clipping）来避免其严重的后果。其基本想法源自梯度并没有指明最佳步长，只说明了在无限小区域内的最佳方向。当传统的梯度下降算法提议更新很大一步时，启发式梯度截断会干涉来减小步长，从

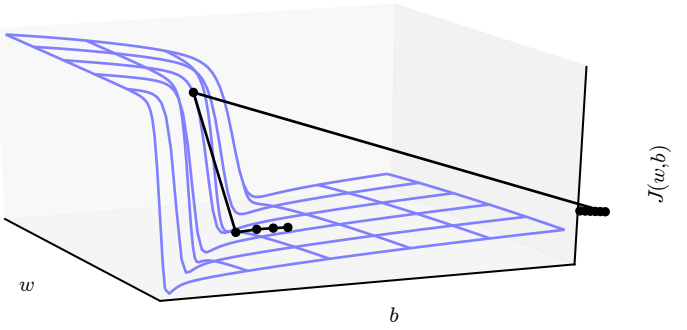

图 8.3: 高度非线性的深度神经网络或循环神经网络的目标函数通常包含由几个参数连乘而导致的参数空间中尖锐非线性。这些非线性在某些区域会产生非常大的导数。当参数接近这样的悬崖区域时，梯度下降更新可以使参数弹射得非常远，可能会使大量已完成的优化工作成为无用功。图经 Pascanu *et al.* (2013a) 许可改编。

而使其不太可能走出梯度近似为最陡下降方向的悬崖区域。悬崖结构在循环神经网络的代价函数中很常见，因为这类模型会涉及到多个因子的相乘，其中每个因子对应一个时间步。因此，长期时间序列会产生大量相乘。

8.2.5 长期依赖

当计算图变得极深时，神经网络优化算法会面临的另外一个难题就是长期依赖问题——由于变深的结构使模型丧失了学习到先前信息的能力，让优化变得极其困难。深层的计算图不仅存在于前馈网络，还存在于之后介绍的循环网络中（在第十章中描述）。因为循环网络要在很长时间序列的各个时刻重复应用相同操作来构建非常深的计算图，并且模型参数共享，这使问题更加凸显。

例如，假设某个计算图中包含一条反复与矩阵 $\boldsymbol{W}$ 相乘的路径。那么 t 步后，相当于乘以 $\boldsymbol{W}^t$。假设 $\boldsymbol{W}$ 有特征值分解 $\boldsymbol{W} = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\,\boldsymbol{V}^{-1}$。在这种简单的情况下，很容易看出

$$\boldsymbol{W}^t = (\boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\,\boldsymbol{V}^{-1})^t = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})^t\,\boldsymbol{V}^{-1}. \tag{8.11}$$

当特征值 λ_i 不在 1 附近时，若在量级上大于 1 则会爆炸；若小于 1 时则会消失。**梯度消失与爆炸问题**（vanishing and exploding gradient problem）是指该计算图上的

梯度也会因为 $\mathrm{diag}(\boldsymbol{\lambda})^t$ 大幅度变化。梯度消失使得我们难以知道参数朝哪个方向移动能够改进代价函数，而梯度爆炸会使得学习不稳定。之前描述的促使我们使用梯度截断的悬崖结构便是梯度爆炸现象的一个例子。

此处描述的在各时间步重复与 $\boldsymbol{W}$ 相乘非常类似于寻求矩阵 $\boldsymbol{W}$ 的最大特征值及对应特征向量的**幂方法**（power method）。从这个观点来看，$\boldsymbol{x}^\top \boldsymbol{W}^t$ 最终会丢弃 $\boldsymbol{x}$ 中所有与 $\boldsymbol{W}$ 的主特征向量正交的成分。

循环网络在各时间步上使用相同的矩阵 $\boldsymbol{W}$，而前馈网络并没有。所以即使使用非常深层的前馈网络，也能很大程度上有效地避免梯度消失与爆炸问题 (Sussillo, 2014)。

在更详细地描述循环网络之后，我们将会在第 10.7 节进一步讨论循环网络训练中的挑战。

8.2.6 非精确梯度

大多数优化算法的先决条件都是我们知道精确的梯度或是 Hessian 矩阵。在实践中，通常这些量会有噪声，甚至是有偏的估计。几乎每一个深度学习算法都需要基于采样的估计，至少使用训练样本的小批量来计算梯度。

在其他情况，我们希望最小化的目标函数实际上是难以处理的。当目标函数不可解时，通常其梯度也是难以处理的。在这种情况下，我们只能近似梯度。这些问题主要出现在第三部分中更高级的模型中。例如，对比散度是用来近似玻尔兹曼机中难以处理的对数似然梯度的一种技术。

各种神经网络优化算法的设计都考虑到了梯度估计的缺陷。我们可以选择比真实损失函数更容易估计的代理损失函数来避免这个问题。

8.2.7 局部和全局结构间的弱对应

迄今为止，我们讨论的许多问题都是关于损失函数在单个点的性质——若 $J(\boldsymbol{\theta})$ 是当前点 $\boldsymbol{\theta}$ 的病态条件，或者 $\boldsymbol{\theta}$ 在悬崖中，或者 $\boldsymbol{\theta}$ 是一个下降方向不明显的鞍点，那么会很难更新当前步。

如果该方向在局部改进很大，但并没有指向代价低得多的遥远区域，那么我们有可能在单点处克服以上所有困难，但仍然表现不佳。

Goodfellow *et al.* (2015) 认为大部分训练的运行时间取决于到达解决方案的轨迹长度。如图 8.2 所示，学习轨迹将花费大量的时间探寻一个围绕山形结构的宽弧。

大多数优化研究的难点集中于训练是否找到了全局最小点、局部极小点或是鞍点，但在实践中神经网络不会到达任何一种临界点。图 8.1 表明神经网络通常不会到达梯度很小的区域。甚至，这些临界点不一定存在。例如，损失函数 $-\log p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$ 可以没有全局最小点，而是当随着训练模型逐渐稳定后，渐近地收敛于某个值。对于具有离散的 y 和 softmax 分布 $p(y \mid \boldsymbol{x})$ 的分类器而言，若模型能够正确分类训练集上的每个样本，则负对数似然可以无限趋近但不会等于零。同样地，实值模型 $p(y \mid \boldsymbol{x}) = \mathcal{N}(y; f(\boldsymbol{\theta}), \beta^{-1})$ 的负对数似然会趋向于负无穷——如果 $f(\boldsymbol{\theta})$ 能够正确预测所有训练集中的目标 y，学习算法会无限制地增加 β。图 8.4 给出了一个失败的例子，即使没有局部极小值和鞍点，该例还是不能从局部优化中找到一个良好的代价函数值。

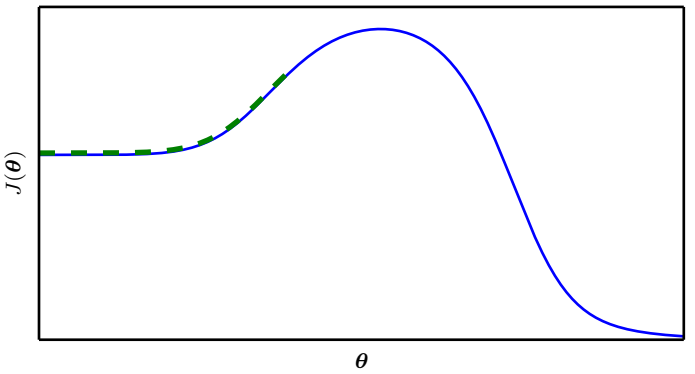

图 8.4: 如果局部表面没有指向全局解，基于局部下坡移动的优化可能就会失败。这里我们提供一个例子，说明即使在没有鞍点或局部极小值的情况下，优化过程会如何失败。此例中的代价函数仅包含朝向低值而不是极小值的渐近线。在这种情况下，造成这种困难的主要原因是初始化在 “山” 的错误一侧，并且无法遍历。在高维空间中，学习算法通常可以环绕过这样的高山，但是相关的轨迹可能会很长，并且导致过长的训练时间，如图 8.2 所示。

未来的研究需要进一步探索影响学习轨迹长度和更好地表征训练过程的结果。

许多现有研究方法在求解具有困难全局结构的问题时，旨在寻求良好的初始点，而不是开发非局部范围更新的算法。

梯度下降和基本上所有的可以有效训练神经网络的学习算法，都是基于局部较

小更新。之前的小节主要集中于为何这些局部范围更新的正确方向难以计算。我们也许能计算目标函数的一些性质，如近似的有偏梯度或正确方向估计的方差。在这些情况下，难以确定局部下降能否定义通向有效解的足够短的路径，但我们并不能真的遵循局部下降的路径。目标函数可能有诸如病态条件或不连续梯度的问题，使得梯度为目标函数提供较好近似的区间非常小。在这些情况下，步长为 ϵ 的局部下降可能定义了到达解的合理的短路经，但是我们只能计算步长为 $\delta \ll \epsilon$ 的局部下降方向。在这些情况下，局部下降或许能定义通向解的路径，但是该路径包含很多次更新，因此遵循该路径会带来很高的计算代价。有时，比如说当目标函数有一个宽而平的区域，或是我们试图寻求精确的临界点（通常来说后一种情况只发生于显式求解临界点的方法，如牛顿法）时，局部信息不能为我们提供任何指导。在这些情况下，局部下降完全无法定义通向解的路径。在其他情况下，局部移动可能太过贪心，朝着下坡方向移动，却和所有可行解南辕北辙，如图 8.4 所示，或者是用舍近求远的方法来求解问题，如图 8.2 所示。目前，我们还不了解这些问题中的哪一个与神经网络优化中的难点最相关，这是研究领域的热点方向。

不管哪个问题最重要，如果存在一个区域，我们遵循局部下降便能合理地直接到达某个解，并且我们能够在该良好区域上初始化学习，那么这些问题都可以避免。最终的观点还是建议在传统优化算法上研究怎样选择更佳的初始化点，以此来实现目标更切实可行。

8.2.8 优化的理论限制

一些理论结果表明，我们为神经网络设计的任何优化算法都有性能限制 (Blum and Rivest, 1992; Judd, 1989; Wolpert and MacReady, 1997)。通常这些结果不影响神经网络在实践中的应用。

一些理论结果仅适用于神经网络的单元输出离散值的情况。然而，大多数神经网络单元输出光滑的连续值，使得局部搜索求解优化可行。一些理论结果表明，存在某类问题是不可解的，但很难判断一个特定问题是否属于该类。其他结果表明，寻找给定规模的网络的一个可行解是很困难的，但在实际情况中，我们通过设置更多参数，使用更大的网络，能轻松找到可接受的解。此外，在神经网络训练中，我们通常不关注某个函数的精确极小点，而只关注将其值下降到足够小以获得一个良好的泛化误差。对优化算法是否能完成此目标进行理论分析是非常困难的。因此，研究优化算法更现实的性能上界仍然是学术界的一个重要目标。

8.3 基本算法

之前我们已经介绍了梯度下降（第4.3节），即沿着整个训练集的梯度方向下降。这可以使用随机梯度下降很大程度地加速，沿着随机挑选的小批量数据的梯度下降方向，就像第5.9节和第8.1.3节中讨论的一样。

8.3.1 随机梯度下降

随机梯度下降（SGD）及其变种很可能是一般机器学习中应用最多的优化算法，特别是在深度学习中。如第8.1.3节中所讨论的，按照数据生成分布抽取 m 个小批量（独立同分布的）样本，通过计算它们梯度均值，我们可以得到梯度的无偏估计。

算法8.1展示了如何沿着这个梯度的估计下降。

算法 8.1 随机梯度下降（SGD）在第 k 个训练迭代的更新

Require: 学习率 ϵ_k

Require: 初始参数 $\boldsymbol{\theta}$

while 停止准则未满足 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 的小批量，其中 $\boldsymbol{x}^{(i)}$ 对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度估计：$\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\hat{\boldsymbol{g}}$

end while

SGD算法中的一个关键参数是学习率。之前，我们介绍的SGD使用固定的学习率。在实践中，有必要随着时间的推移逐渐降低学习率，因此我们将第 k 步迭代的学习率记作 ϵ_k。

这是因为SGD中梯度估计引入的噪声源（m 个训练样本的随机采样）并不会在极小点处消失。相比之下，当我们使用批量梯度下降到达极小点时，整个代价函数的真实梯度会变得很小，之后为 $\mathbf{0}$，因此批量梯度下降可以使用固定的学习率。保证SGD收敛的一个充分条件是

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \tag{8.12}$$

且

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty. \tag{8.13}$$

实践中，一般会线性衰减学习率直到第 τ 次迭代：

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau \tag{8.14}$$

其中 $\alpha = \frac{k}{\tau}$。在 τ 步迭代之后，一般使 ϵ 保持常数。

学习率可通过试验和误差来选取，通常最好的选择方法是监测目标函数值随时间变化的学习曲线。与其说是科学，这更像是一门艺术，我们应该谨慎地参考关于这个问题的大部分指导。使用线性策略时，需要选择的参数为 ϵ_0，ϵ_τ，τ。通常 τ 被设为需要反复遍历训练集几百次的迭代次数。通常 ϵ_τ 应设为大约 ϵ_0 的 1%。主要问题是如何设置 ϵ_0。若 ϵ_0 太大，学习曲线将会剧烈振荡，代价函数值通常会明显增加。温和的振荡是良好的，容易在训练随机代价函数（例如使用 Dropout 的代价函数）时出现。如果学习率太小，那么学习过程会很缓慢。如果初始学习率太低，那么学习可能会卡在一个相当高的代价值。通常，就总训练时间和最终代价值而言，最优初始学习率会高于大约迭代 100 次左右后达到最佳效果的学习率。因此，通常最好是检测最早的几轮迭代，选择一个比在效果上表现最佳的学习率更大的学习率，但又不能太大导致严重的震荡。

SGD 及相关的小批量亦或更广义的基于梯度优化的在线学习算法，一个重要的性质是每一步更新的计算时间不依赖训练样本数目的多寡。即使训练样本数目非常大时，它们也能收敛。对于足够大的数据集，SGD 可能会在处理整个训练集之前就收敛到最终测试集误差的某个固定容差范围内。

研究优化算法的收敛率，一般会衡量**额外误差**（excess error）$J(\boldsymbol{\theta}) - \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$，即当前代价函数超出最低可能代价的量。SGD 应用于凸问题时，k 步迭代后的额外误差量级是 $O(\frac{1}{\sqrt{k}})$，在强凸情况下是 $O(\frac{1}{k})$。除非假定额外的条件，否则这些界限不能进一步改进。批量梯度下降在理论上比随机梯度下降有更好的收敛率。然而，Cramér-Rao 界限 (Cramér, 1946; Rao, 1945) 指出，泛化误差的下降速度不会快于 $O(\frac{1}{k})$。Bottou and Bousquet (2008b) 因此认为对于机器学习任务，不值得探寻收敛快于 $O(\frac{1}{k})$ 的优化算法——更快的收敛可能对应着过拟合。此外，渐近分析掩盖了随机梯度下降在少量更新步之后的很多优点。对于大数据集，SGD 只需非常少量样本计算梯度从而实现初始快速更新，远远超过了其缓慢的渐近收敛。本章剩余部分介绍的大多数算法在实践中都受益于这种性质，但是损失了常数倍 $O(\frac{1}{k})$ 的渐近分析。

我们也可以在学习过程中逐渐增大小批量的大小，以此权衡批量梯度下降和随机梯度下降两者的优点。

了解 SGD 更多的信息，请查看 Bottou (1998)。

8.3.2 动量

虽然随机梯度下降仍然是非常受欢迎的优化方法，但其学习过程有时会很慢。动量方法 (Polyak, 1964) 旨在加速学习，特别是处理高曲率、小但一致的梯度，或是带噪声的梯度。动量算法积累了之前梯度指数级衰减的移动平均，并且继续沿该方向移动。动量的效果如图 8.5 所示。

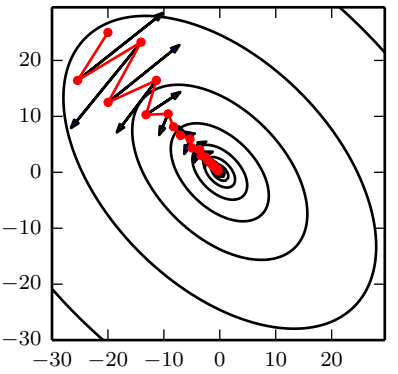

图 8.5: 动量的主要目的是解决两个问题：Hessian 矩阵的病态条件和随机梯度的方差。我们通过此图说明动量如何克服这两个问题的第一个。等高线描绘了一个二次损失函数（具有病态条件的 Hessian 矩阵）。横跨轮廓的红色路径表示动量学习规则所遵循的路径，它使该函数最小化。我们在该路径的每个步骤画一个箭头，表示梯度下降将在该点采取的步骤。我们可以看到，一个病态条件的二次目标函数看起来像一个长而窄的山谷或具有陡峭边的峡谷。动量正确地纵向穿过峡谷，而普通的梯度步骤则会浪费时间在峡谷的窄轴上来回移动。比较图 4.6，它也显示了没有动量的梯度下降的行为。

从形式上看，动量算法引入了变量 $\boldsymbol{v}$ 充当速度角色——它代表参数在参数空间移动的方向和速率。速度被设为负梯度的指数衰减平均。名称 **动量**（momentum）来自物理类比，根据牛顿运动定律，负梯度是移动参数空间中粒子的力。动量在物理学上定义为质量乘以速度。在动量学习算法中，我们假设是单位质量，因此速度向量 $\boldsymbol{v}$ 也可以看作是粒子的动量。超参数$\alpha \in [0, 1)$ 决定了之前梯度的贡献衰减得有

多快。更新规则如下：

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right), \tag{8.15}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}. \tag{8.16}$$

速度 $\boldsymbol{v}$ 累积了梯度元素 $\nabla_{\boldsymbol{\theta}}(\frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)}))$。相对于 ϵ，α 越大，之前梯度对现在方向的影响也越大。带动量的SGD算法如算法8.2所示。

算法 8.2 使用动量的随机梯度下降（SGD）

Require: 学习率 ϵ，动量参数 α

Require: 初始参数 $\boldsymbol{\theta}$，初始速度 $\boldsymbol{v}$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度估计：$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

计算速度更新：$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

end while

之前，步长只是梯度范数乘以学习率。现在，步长取决于梯度序列的大小和排列。当许多连续的梯度指向相同的方向时，步长最大。如果动量算法总是观测到梯度 $\boldsymbol{g}$，那么它会在方向 $-g$ 上不停加速，直到达到最终速度，其中步长大小为

$$\frac{\epsilon \|\boldsymbol{g}\|}{1-\alpha}. \tag{8.17}$$

因此将动量的超参数视为 $\frac{1}{1-\alpha}$ 有助于理解。例如，$\alpha = 0.9$ 对应着最大速度 10 倍于梯度下降算法。

在实践中，α 的一般取值为 0.5，0.9 和 0.99。和学习率一样，α 也会随着时间不断调整。一般初始值是一个较小的值，随后会慢慢变大。随着时间推移调整 α 没有收缩 ϵ 重要。

我们可以将动量算法视为模拟连续时间下牛顿动力学下的粒子。这种物理类比有助于直觉上理解动量和梯度下降算法是如何表现的。

粒子在任意时间点的位置由 $\boldsymbol{\theta}(t)$ 给定。粒子会受到净力 $\boldsymbol{f}(t)$。该力会导致粒子加速：

$$\boldsymbol{f}(t) = \frac{\partial^2}{\partial t^2} \boldsymbol{\theta}(t). \tag{8.18}$$

与其将其视为位置的二阶微分方程，我们不如引入表示粒子在时间 t 处速度的变量 $\boldsymbol{v}(t)$，将牛顿动力学重写为一阶微分方程：

$$\boldsymbol{v}(t) = \frac{\partial}{\partial t}\boldsymbol{\theta}(t), \tag{8.19}$$

$$\boldsymbol{f}(t) = \frac{\partial}{\partial t}\boldsymbol{v}(t). \tag{8.20}$$

由此，动量算法包括通过数值模拟求解微分方程。求解微分方程的一个简单数值方法是欧拉方法，通过在每个梯度方向上小且有限的步来简单模拟该等式定义的动力学。

这解释了动量更新的基本形式，但具体什么是力呢？力正比于代价函数的负梯度 $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$。该力推动粒子沿着代价函数表面下坡的方向移动。梯度下降算法基于每个梯度简单地更新一步，而使用动量算法的牛顿方案则使用该力改变粒子的速度。我们可以将粒子视作在冰面上滑行的冰球。每当它沿着表面最陡的部分下降时，它会累积继续在该方向上滑行的速度，直到其开始向上滑动为止。

另一个力也是必要的。如果代价函数的梯度是唯一的力，那么粒子可能永远不会停下来。想象一下，假设理想情况下冰面没有摩擦，一个冰球从山谷的一端下滑，上升到另一端，永远来回振荡。要解决这个问题，我们添加另一个正比于 $-\boldsymbol{v}(t)$ 的力。在物理术语中，此力对应于粘性阻力，就像粒子必须通过一个抵抗介质，如糖浆。这会导致粒子随着时间推移逐渐失去能量，最终收敛到局部极小点。

为什么要特别使用 $-\boldsymbol{v}(t)$ 和粘性阻力呢？部分原因是因为 $-\boldsymbol{v}(t)$ 在数学上的便利——速度的整数幂很容易处理。然而，其他物理系统具有基于速度的其他整数幂的其他类型的阻力。例如，颗粒通过空气时会受到正比于速度平方的湍流阻力，而颗粒沿着地面移动时会受到恒定大小的摩擦力。这些选择都不合适。湍流阻力，正比于速度的平方，在速度很小时会很弱。不够强到使粒子停下来。非零值初始速度的粒子仅受到湍流阻力，会从初始位置永远地移动下去，和初始位置的距离大概正比于 $O(\log t)$。因此我们必须使用速度较低幂次的力。如果幂次为零，相当于干摩擦，那么力太强了。当代价函数的梯度表示的力很小但非零时，由于摩擦导致的恒力会使得粒子在达到局部极小点之前就停下来。粘性阻力避免了这两个问题——它足够弱，可以使梯度引起的运动直到达到最小，但又足够强，使得坡度不够时可以阻止运动。

8.3.3 Nesterov 动量

受 Nesterov 加速梯度算法 (Nesterov, 1983, 2004) 启发，Sutskever *et al.* (2013) 提出了动量算法的一个变种。这种情况的更新规则如下：

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^{m} L\big(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)}\big) \right], \tag{8.21}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}, \tag{8.22}$$

其中参数 α 和 ϵ 发挥了和标准动量方法中类似的作用。Nesterov 动量和标准动量之间的区别体现在梯度计算上。Nesterov 动量中，梯度计算在施加当前速度之后。因此，Nesterov 动量可以解释为往标准动量方法中添加了一个校正因子。完整的 Nesterov 动量算法如算法 8.3 所示。

算法 8.3 使用 Nesterov 动量的随机梯度下降（SGD）

Require: 学习率 ϵ，动量参数 α

Require: 初始参数 $\boldsymbol{\theta}$，初始速度 $\boldsymbol{v}$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

应用临时更新：$\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$

计算梯度（在临时点）：$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$

计算速度更新：$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

end while

在凸批量梯度的情况下，Nesterov 动量将额外误差收敛率从 $O(1/k)$（k 步后）改进到 $O(1/k^2)$，如 Nesterov (1983) 所示。可惜，在随机梯度的情况下，Nesterov 动量没有改进收敛率。

8.4 参数初始化策略

有些优化算法本质上是非迭代的，只是求解一个解点。有些其它优化算法本质上是迭代的，但是应用于这一类的优化问题时，能在可接受的时间内收敛到可接受的解，并且与初始值无关。深度学习训练算法通常没有这两种奢侈的性质。深度学

习模型的训练算法通常是迭代的，因此要求使用者指定一些开始迭代的初始点。此外，训练深度模型是一个足够困难的问题，以致于大多数算法都很大程度地受到初始化选择的影响。初始点能够决定算法是否收敛，有些初始点十分不稳定，使得该算法会遭遇数值困难，并完全失败。当学习收敛时，初始点可以决定学习收敛得多快，以及是否收敛到一个代价高或低的点。此外，差不多代价的点可以具有区别极大的泛化误差，初始点也可以影响泛化。

现代的初始化策略是简单的、启发式的。设定改进的初始化策略是一项困难的任务，因为神经网络优化至今还未被很好地理解。大多数初始化策略基于在神经网络初始化时实现一些很好的性质。然而，我们并没有很好地理解这些性质中的哪些会在学习开始进行后的哪些情况下得以保持。进一步的难点是，有些初始点从优化的观点看或许是有利的，但是从泛化的观点看是不利的。我们对于初始点如何影响泛化的理解是相当原始的，几乎没有提供如何选择初始点的任何指导。

也许完全确知的唯一特性是初始参数需要在不同单元间“破坏对称性”。如果具有相同激活函数的两个隐藏单元连接到相同的输入，那么这些单元必须具有不同的初始参数。如果它们具有相同的初始参数，然后应用到确定性损失和模型的确定性学习算法将一直以相同的方式更新这两个单元。即使模型或训练算法能够使用随机性为不同的单元计算不同的更新（例如使用Dropout的训练），通常来说，最好还是初始化每个单元使其和其他单元计算不同的函数。这或许有助于确保没有输入模式丢失在前向传播的零空间中，没有梯度模式丢失在反向传播的零空间中。每个单元计算不同函数的目标促使了参数的随机初始化。我们可以明确地搜索一大组彼此互不相同的基函数，但这经常会导致明显的计算代价。例如，如果我们有和输出一样多的输入，我们可以使用 Gram-Schmidt 正交化于初始的权重矩阵，保证每个单元计算彼此非常不同的函数。在高维空间上使用高熵分布来随机初始化，计算代价小并且不太可能分配单元计算彼此相同的函数。

通常情况下，我们可以为每个单元的偏置设置启发式挑选的常数，仅随机初始化权重。额外的参数（例如用于编码预测条件方差的参数）通常和偏置一样设置为启发式选择的常数。

我们几乎总是初始化模型的权重为高斯或均匀分布中随机抽取的值。高斯或均匀分布的选择似乎不会有很大的差别，但也没有被详尽地研究。然而，初始分布的大小确实对优化过程的结果和网络泛化能力都有很大的影响。

更大的初始权重具有更强的破坏对称性的作用，有助于避免冗余的单元。它们

也有助于避免在每层线性成分的前向或反向传播中丢失信号——矩阵中更大的值在矩阵乘法中有更大的输出。如果初始权重太大，那么会在前向传播或反向传播中产生爆炸的值。在循环网络中，很大的权重也可能导致**混沌**（chaos）（对于输入中很小的扰动非常敏感，导致确定性前向传播过程表现随机）。在一定程度上，梯度爆炸问题可以通过梯度截断来缓解（执行梯度下降步骤之前设置梯度的阈值）。较大的权重也会产生使得激活函数饱和的值，导致饱和单元的梯度完全丢失。这些竞争因素决定了权重的理想初始大小。

关于如何初始化网络，正则化和优化有着非常不同的观点。优化观点建议权重应该足够大以成功传播信息，但是正则化希望其小一点。诸如随机梯度下降这类对权重较小的增量更新，趋于停止在更靠近初始参数的区域（不管是由于卡在低梯度的区域，还是由于触发了基于过拟合 的提前终止准则）的优化算法倾向于最终参数应接近于初始参数。回顾第 7.8 节，在某些模型上，提前终止的梯度下降等价于权重衰减。在一般情况下，提前终止的梯度下降和权重衰减不同，但是提供了一个宽松的类比去考虑初始化的影响。我们可以将初始化参数 $\boldsymbol{\theta}$ 为 $\boldsymbol{\theta}_0$ 类比于强置均值为 $\boldsymbol{\theta}_0$ 的高斯先验 $p(\boldsymbol{\theta})$。从这个角度来看，选择 $\boldsymbol{\theta}_0$ 接近 0 是有道理的。这个先验表明，单元间彼此互不交互比交互更有可能。只有在目标函数的似然项表达出对交互很强的偏好时，单元才会交互。另一方面，如果我们初始化 $\boldsymbol{\theta}_0$ 为很大的值，那么我们的先验指定了哪些单元应互相交互，以及它们应如何交互。

有些启发式方法可用于选择权重的初始大小。一种初始化 m 个输入和 n 输出的全连接层的权重的启发式方法是从分布 $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$ 中采样权重，而 Glorot and Bengio (2010) 建议使用**标准初始化**（normalized initialization）

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right). \tag{8.23}$$

后一种启发式方法初始化所有的层，折衷于使其具有相同激活方差和使其具有相同梯度方差之间。这假设网络是不含非线性的链式矩阵乘法，据此推导得出。现实的神经网络显然会违反这个假设，但很多设计于线性模型的策略在其非线性对应中的效果也不错。

Saxe *et al.* (2013) 推荐初始化为随机正交矩阵，仔细挑选负责每一层非线性缩放或**增益** (gain) 因子 g。他们得到了用于不同类型的非线性激活函数的特定缩放因子。这种初始化方案也是启发于不含非线性的矩阵相乘序列的深度网络。在该模型下，这个初始化方案保证了达到收敛所需的训练迭代总数独立于深度。

增加缩放因子 g 将网络推向网络前向传播时激活范数增加，反向传播时梯度范数增加的区域。Sussillo (2014) 表明，正确设置缩放因子足以训练深达 1000 层的网络，而不需要使用正交初始化。这种方法的一个重要观点是，在前馈网络中，激活和梯度会在每一步前向传播或反向传播中增加或缩小，遵循随机游走行为。这是因为前馈网络在每一层使用了不同的权重矩阵。如果该随机游走调整到保持范数，那么前馈网络能够很大程度地避免相同权重矩阵用于每层的梯度消失与爆炸问题，如第 8.2.5 节所述。

可惜，这些初始权重的最佳准则往往不会带来最佳效果。这可能有三种不同的原因。首先，我们可能使用了错误的标准——它实际上并不利于保持整个网络信号的范数。其次，初始化时强加的性质可能在学习开始进行后不能保持。最后，该标准可能成功提高了优化速度，但意外地增大了泛化误差。在实践中，我们通常需要将权重范围视为超参数，其最优值大致接近，但并不完全等于理论预测。

数值范围准则的一个缺点是，设置所有的初始权重具有相同的标准差，例如 $\frac{1}{\sqrt{m}}$，会使得层很大时每个单一权重会变得极其小。Martens (2010) 提出了一种被称为 **稀疏初始化**（sparse initialization）的替代方案，每个单元初始化为恰好有 k 个非零权重。这个想法保持该单元输入的总数量独立于输入数目 m，而不使单一权重元素的大小随 m 缩小。稀疏初始化有助于实现单元之间在初始化时更具多样性。但是，获得较大取值的权重也同时被加了很强的先验。因为梯度下降需要很长时间缩小 “不正确” 的大值，这个初始化方案可能会导致某些单元出问题，例如 maxout 单元有几个过滤器，互相之间必须仔细调整。

计算资源允许的话，将每层权重的初始数值范围设为超参数通常是个好主意，使用第 11.4.2 节介绍的超参数搜索算法，如随机搜索，挑选这些数值范围。是否选择使用密集或稀疏初始化也可以设为一个超参数。作为替代，我们可以手动搜索最优初始范围。一个好的挑选初始数值范围的经验法则是观测单个小批量数据上的激活或梯度的幅度或标准差。如果权重太小，那么当激活值在小批量上前向传播于网络时，激活值的幅度会缩小。通过重复识别具有小得不可接受的激活值的第一层，并提高其权重，最终有可能得到一个初始激活全部合理的网络。如果学习在这点上仍然很慢，观测梯度的幅度或标准差可能也会有所帮助。这个过程原则上是自动的，且通常计算量低于基于验证集误差的超参数优化，因为它是基于初始模型在单批数据上的行为反馈，而不是在验证集上训练模型的反馈。由于这个协议很长时间都被启发式使用，最近 Mishkin and Matas (2015) 更正式地研究了该协议。

目前为止，我们关注在权重的初始化上。幸运的是，其他参数的初始化通常更

容易。

设置偏置的方法必须和设置权重的方法协调。设置偏置为零通常在大多数权重初始化方案中是可行的。存在一些我们可能设置偏置为非零值的情况：

- 如果偏置是作为输出单元，那么初始化偏置以获取正确的输出边缘统计通常是有利的。要做到这一点，我们假设初始权重足够小，该单元的输出仅由偏置决定。这说明设置偏置为应用于训练集上输出边缘统计的激活函数的逆。例如，如果输出是类上的分布，且该分布是高度偏态分布，第 i 类的边缘概率由某个向量 $\boldsymbol{c}$ 的第 i 个元素给定，那么我们可以通过求解方程 $\text{softmax}(\boldsymbol{b}) = \boldsymbol{c}$ 来设置偏置向量 $\boldsymbol{b}$。这不仅适用于分类器，也适用于我们将在第三部分遇到的模型，例如自编码器和玻尔兹曼机。这些模型拥有输出类似于输入数据 $\boldsymbol{x}$ 的网络层，非常有助于初始化这些层的偏置以匹配 $\boldsymbol{x}$ 上的边缘分布。

- 有时，我们可能想要选择偏置以避免初始化引起太大饱和。例如，我们可能会将 ReLU 的隐藏单元设为 0.1 而非 0，以避免 ReLU 在初始化时饱和。尽管这种方法违背不希望偏置具有很强输入的权重初始化准则。例如，不建议使用随机游走初始化 (Sussillo, 2014)。

- 有时，一个单元会控制其他单元能否参与到等式中。在这种情况下，我们有一个单元输出 u，另一个单元 $h \in [0, 1]$，那么我们可以将 h 视作门，以决定 $uh \approx 1$ 还是 $uh \approx 0$。在这种情形下，我们希望设置偏置 h，使得在初始化的大多数情况下 $h \approx 1$。否则，u 没有机会学习。例如，Jozefowicz *et al.* (2015) 提议设置 LSTM 模型遗忘门的偏置为 1，如第 10.10 节所述。

另一种常见类型的参数是方差或精确度参数。例如，我们用以下模型进行带条件方差估计的线性回归

$$p(y \mid \boldsymbol{x}) = \mathcal{N}(y \mid \boldsymbol{w}^\top \boldsymbol{x} + b, 1/\beta), \tag{8.24}$$

其中 β 是精确度参数。通常我们能安全地初始化方差或精确度参数为 1。另一种方法假设初始权重足够接近零，设置偏置可以忽略权重的影响，然后设定偏置以产生输出的正确边缘均值，并将方差参数设置为训练集输出的边缘方差。

除了这些初始化模型参数的简单常数或随机方法，还有可能使用机器学习初始化模型参数。在本书第三部分讨论的一个常用策略是使用相同的输入数据集，用无监督模型训练出来的参数来初始化监督模型。我们也可以在相关问题上使用监督训

练。即使是在一个不相关的任务上运行监督训练，有时也能得到一个比随机初始化具有更快收敛率的初始值。这些初始化策略有些能够得到更快的收敛率和更好的泛化误差，因为它们编码了模型初始参数的分布信息。其他策略显然效果不错的原因主要在于它们设置参数为正确的数值范围，或是设置不同单元计算互相不同的函数。

8.5 自适应学习率算法

神经网络研究员早就意识到学习率肯定是难以设置的超参数之一，因为它对模型的性能有显著的影响。正如我们在第 4.3 节和第 8.2 节中所探讨的，损失通常高度敏感于参数空间中的某些方向，而不敏感于其他。动量算法可以在一定程度缓解这些问题，但这样做的代价是引入了另一个超参数。在这种情况下，自然会问有没有其他方法。如果我们相信方向敏感度在某种程度是轴对齐的，那么每个参数设置不同的学习率，在整个学习过程中自动适应这些学习率是有道理的。

Delta-bar-delta 算法 (Jacobs, 1988) 是一个早期的在训练时适应模型参数各自学习率的启发式方法。该方法基于一个很简单的想法，如果损失对于某个给定模型参数的偏导保持相同的符号，那么学习率应该增加。如果对于该参数的偏导变化了符号，那么学习率应减小。当然，这种方法只能应用于全批量优化中。

最近，提出了一些增量（或者基于小批量）的算法来自适应模型参数的学习率。这节将简要回顾其中一些算法。

8.5.1 AdaGrad

AdaGrad 算法，如算法 8.4 所示，独立地适应所有模型参数的学习率，缩放每个参数反比于其所有梯度历史平方值总和的平方根 (Duchi *et al.*, 2011)。具有损失最大偏导的参数相应地有一个快速下降的学习率，而具有小偏导的参数在学习率上有相对较小的下降。净效果是在参数空间中更为平缓的倾斜方向会取得更大的进步。

在凸优化背景中，AdaGrad 算法具有一些令人满意的理论性质。然而，经验上已经发现，对于训练深度神经网络模型而言，从训练开始时积累梯度平方会导致有效学习率过早和过量的减小。AdaGrad 在某些深度学习模型上效果不错，但不是全部。

算法 8.4 AdaGrad 算法

Require: 全局学习率 ϵ

Require: 初始参数 $\boldsymbol{\theta}$

Require: 小常数 δ，为了数值稳定大约设为 10^{-7}

初始化梯度累积变量 $\boldsymbol{r} = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

累积平方梯度：$\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

计算更新：$\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$ （逐元素地应用除和求平方根）

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

8.5.2 RMSProp

RMSProp 算法 (Hinton, 2012) 修改 AdaGrad 以在非凸设定下效果更好，改变梯度积累为指数加权的移动平均。AdaGrad 旨在应用于凸问题时快速收敛。当应用于非凸函数训练神经网络时，学习轨迹可能穿过了很多不同的结构，最终到达一个局部是凸碗的区域。AdaGrad 根据平方梯度的整个历史收缩学习率，可能使得学习率在达到这样的凸结构前就变得太小了。RMSProp 使用指数衰减平均以丢弃遥远过去的历史，使其能够在找到凸碗状结构后快速收敛，它就像一个初始化于该碗状结构的 AdaGrad 算法实例。

RMSProp 的标准形式如算法 8.5 所示，结合 Nesterov 动量的形式如算法 8.6 所示。相比于 AdaGrad，使用移动平均引入了一个新的超参数ρ，用来控制移动平均的长度范围。

经验上，RMSProp 已被证明是一种有效且实用的深度神经网络优化算法。目前它是深度学习从业者经常采用的优化方法之一。

8.5.3 Adam

Adam (Kingma and Ba, 2014) 是另一种学习率自适应的优化算法，如算法 8.7 所示。“Adam'' 这个名字派生自短语 “adaptive moments''。早期算法背景下，它也许

算法 8.5 RMSProp 算法

Require: 全局学习率 ϵ，衰减速率 ρ

Require: 初始参数 $\boldsymbol{\theta}$

Require: 小常数 δ，通常设为 10^{-6}（用于被小数除时的数值稳定）

初始化累积变量 $\boldsymbol{r} = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

累积平方梯度：$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$

计算参数更新：$\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$ （$\frac{1}{\sqrt{\delta+\boldsymbol{r}}}$ 逐元素应用）

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$

end while

最好被看作结合 RMSProp 和具有一些重要区别的动量的变种。首先，在 Adam 中，动量直接并入了梯度一阶矩（指数加权）的估计。将动量加入 RMSProp 最直观的方法是将动量应用于缩放后的梯度。结合缩放的动量使用没有明确的理论动机。其次，Adam 包括偏置修正，修正从原点初始化的一阶矩（动量项）和（非中心的）二阶矩的估计（算法 8.7 ）。RMSProp 也采用了（非中心的）二阶矩估计，然而缺失了修正因子。因此，不像 Adam，RMSProp 二阶矩估计可能在训练初期有很高的偏置。Adam 通常被认为对超参数的选择相当鲁棒，尽管学习率有时需要从建议的默认修改。

8.5.4 选择正确的优化算法

在本节中，我们讨论了一系列算法，通过自适应每个模型参数的学习率以解决优化深度模型中的难题。此时，一个自然的问题是：该选择哪种算法呢?

遗憾的是，目前在这一点上没有达成共识。Schaul *et al.* (2014) 展示了许多优化算法在大量学习任务上极具价值的比较。虽然结果表明，具有自适应学习率（以 RMSProp 和 AdaDelta 为代表）的算法族表现得相当鲁棒，不分伯仲，但没有哪个算法能脱颖而出。

目前，最流行并且使用很高的优化算法包括 SGD、具动量的 SGD、RMSProp、具动量的 RMSProp、AdaDelta 和 Adam。此时，选择哪一个算法似乎主要取决于

算法 8.6 使用 Nesterov 动量的 RMSProp 算法

Require: 全局学习率 ϵ，衰减速率 ρ，动量系数 α

Require: 初始参数 $\boldsymbol{\theta}$，初始参数 $\boldsymbol{v}$

初始化累积变量 $\boldsymbol{r} = 0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

计算临时更新：$\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$

累积梯度：$\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$

计算速度更新：$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$ （$\frac{1}{\sqrt{\boldsymbol{r}}}$ 逐元素应用）

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

end while

使用者对算法的熟悉程度（以便调节超参数）。

算法 8.7 Adam 算法

Require: 步长 ϵ（建议默认为：0.001）

Require: 矩估计的指数衰减速率，ρ_1 和 ρ_2 在区间 $[0,1)$ 内。（建议默认为：分别为 0.9 和 0.999）

Require: 用于数值稳定的小常数 δ（建议默认为：10^{-8}）

Require: 初始参数 $\boldsymbol{\theta}$

初始化一阶和二阶矩变量 $\boldsymbol{s}=0, \boldsymbol{r}=0$

初始化时间步 $t=0$

while 没有达到停止准则 **do**

从训练集中采包含 m 个样本 $\{\boldsymbol{x}^{(1)},\dots,\boldsymbol{x}^{(m)}\}$ 的小批量，对应目标为 $\boldsymbol{y}^{(i)}$。

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}}\sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),\boldsymbol{y}^{(i)})$

$t \leftarrow t+1$

更新有偏一阶矩估计：$\boldsymbol{s} \leftarrow \rho_1\boldsymbol{s}+(1-\rho_1)\boldsymbol{g}$

更新有偏二阶矩估计：$\boldsymbol{r} \leftarrow \rho_2\boldsymbol{r}+(1-\rho_2)\boldsymbol{g}\odot\boldsymbol{g}$

修正一阶矩的偏差：$\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$

修正二阶矩的偏差：$\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$

计算更新：$\Delta\boldsymbol{\theta} = -\epsilon\frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$（逐元素应用操作）

应用更新：$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}+\Delta\boldsymbol{\theta}$

end while

8.6 二阶近似方法

在本节中，我们会讨论训练深度神经网络的二阶方法。参考LeCun *et al.* (1998a)了解该问题的早期处理方法。为表述简单起见，我们只考察目标函数为经验风险：

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),y^{(i)}). \tag{8.25}$$

然而，我们在这里讨论的方法很容易扩展到更一般的目标函数，例如，第七章讨论的包括参数正则项的函数。

8.6.1 牛顿法

在第 4.3 节，我们介绍了二阶梯度方法。与一阶方法相比，二阶方法使用二阶导数改进了优化。最广泛使用的二阶方法是牛顿法。我们现在更详细地描述牛顿法，重点在其应用于神经网络的训练。

牛顿法是基于二阶泰勒级数展开在某点 $\boldsymbol{\theta}_0$ 附近来近似 $J(\boldsymbol{\theta})$ 的优化方法，其忽略了高阶导数：

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0), \tag{8.26}$$

其中 $\boldsymbol{H}$ 是 J 相对于 $\boldsymbol{\theta}$ 的Hessian矩阵在 $\boldsymbol{\theta}_0$ 处的估计。如果我们再求解这个函数的临界点，我们将得到牛顿参数更新规则：

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0). \tag{8.27}$$

因此，对于局部的二次函数（具有正定的 $\boldsymbol{H}$），用 $\boldsymbol{H}^{-1}$ 重新调整梯度，牛顿法会直接跳到极小值。如果目标函数是凸的但非二次的（有高阶项），该更新将是迭代的，得到和牛顿法相关的算法，如算法 8.8 所示。

对于非二次的表面，只要Hessian矩阵保持正定，牛顿法能够迭代地应用。这意味着一个两步迭代过程。首先，更新或计算Hessian逆（通过更新二阶近似）。其次，根据式 (8.27) 更新参数。

在第 8.2.3 节，我们讨论了牛顿法只适用于Hessian矩阵是正定的情况。在深度学习中，目标函数的表面通常非凸（有很多特征），如鞍点。因此使用牛顿法是有问题的。如果Hessian矩阵的特征值并不都是正的，例如，靠近鞍点处，牛顿法实际上会导致更新朝错误的方向移动。这种情况可以通过正则化Hessian矩阵来避免。常用的正则化策略包括在Hessian矩阵对角线上增加常数 α。正则化更新变为

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - [H(f(\boldsymbol{\theta}_0)) + \alpha \boldsymbol{I}]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0). \tag{8.28}$$

这个正则化策略用于牛顿法的近似，例如 Levenberg-Marquardt 算法 (Levenberg, 1944; Marquardt, 1963)，只要Hessian矩阵的负特征值仍然相对接近零，效果就会很好。在曲率方向更极端的情况下，α 的值必须足够大，以抵消负特征值。然而，如果 α 持续增加，Hessian矩阵会变得由对角矩阵 $\alpha \boldsymbol{I}$ 主导，通过牛顿法所选择的方向会收敛到普通梯度除以 α。当很强的负曲率存在时，α 可能需要特别大，以致于牛顿法比选择合适学习率的梯度下降的步长更小。

算法 8.8 目标为 $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$ 的牛顿法

Require: 初始参数 $\boldsymbol{\theta}_0$

Require: 包含 m 个样本的训练集

while 没有达到停止准则 **do**

计算梯度：$\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

计算 Hessian 矩阵：$\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

计算 Hessian 逆：$\boldsymbol{H}^{-1}$

计算更新：$\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$

应用更新：$\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

除了目标函数的某些特征带来的挑战，如鞍点，牛顿法用于训练大型神经网络还受限于其显著的计算负担。Hessian 矩阵中元素数目是参数数量的平方，因此，如果参数数目为 k（甚至是在非常小的神经网络中 k 也可能是百万级别），牛顿法需要计算 $k \times k$ 矩阵的逆，计算复杂度为 $O(k^3)$。另外，由于参数将每次更新都会改变，每次训练迭代都需要计算 Hessian 矩阵的逆。其结果是，只有参数很少的网络才能在实际中用牛顿法训练。在本节的剩余部分，我们将讨论一些试图保持牛顿法优点，同时避免计算障碍的替代算法。

8.6.2 共轭梯度

共轭梯度是一种通过迭代下降的**共轭方向**（conjugate directions）以有效避免 Hessian 矩阵求逆计算的方法。这种方法的灵感来自于对最速下降方法弱点的仔细研究（详细信息请查看第 4.3 节），其中线搜索迭代地用于与梯度相关的方向上。图 8.6 说明了该方法在二次碗型目标中如何表现的，是一个相当低效的来回往复，锯齿形模式。这是因为每一个由梯度给定的线搜索方向，都保证正交于上一个线搜索方向。

假设上一个搜索方向是 $\boldsymbol{d}_{t-1}$。在极小值处，线搜索终止，方向 $\boldsymbol{d}_{t-1}$ 处的方向导数为零：$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \cdot \boldsymbol{d}_{t-1} = 0$。因为该点的梯度定义了当前的搜索方向，$\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ 将不会贡献于方向 $\boldsymbol{d}_{t-1}$。因此方向 $\boldsymbol{d}_t$ 正交于 $\boldsymbol{d}_{t-1}$。最速下降多次迭代中，方向 $\boldsymbol{d}_{t-1}$ 和 $\boldsymbol{d}_t$ 之间的关系如图 8.6 所示。如图展示的，下降正交方向的选择不会保持前一搜索方向上的最小值。这产生了锯齿形的过程。在当前梯度方向下降到极小值，我们

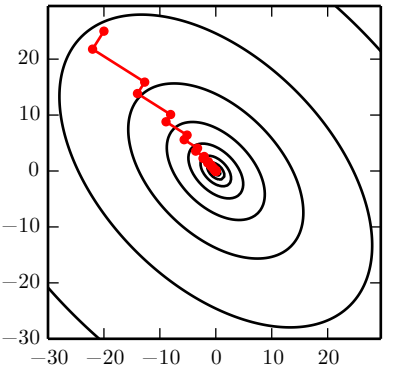

图 8.6: 将最速下降法应用于二次代价表面。在每个步骤，最速下降法沿着由初始点处的梯度定义的线跳到最低代价的点。这解决了图 4.6 中使用固定学习率所遇到的一些问题，但即使使用最佳步长，算法仍然朝最优方向曲折前进。根据定义，在沿着给定方向的目标最小值处，最终点处的梯度与该方向正交。

必须重新最小化之前梯度方向上的目标。因此，通过遵循每次线搜索结束时的梯度，我们在某种程度上撤销了在之前线搜索的方向上取得的进展。共轭梯度试图解决这个问题。

在共轭梯度法中，我们寻求一个和先前线搜索方向 **共轭**（conjugate）的搜索方向，即它不会撤销该方向上的进展。在训练迭代 t 时，下一步的搜索方向 $\boldsymbol{d}_t$ 的形式如下：

$$\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{d}_{t-1}, \tag{8.29}$$

其中，系数 β_t 的大小控制我们应沿方向 $\boldsymbol{d}_{t-1}$ 加回多少到当前搜索方向上。

如果 $\boldsymbol{d}_t^\top \boldsymbol{H} \boldsymbol{d}_{t-1} = 0$，其中 $\boldsymbol{H}$ 是Hessian矩阵，则两个方向 $\boldsymbol{d}_t$ 和 $\boldsymbol{d}_{t-1}$ 被称为共轭的。

适应共轭的直接方法会涉及到 $\boldsymbol{H}$ 特征向量的计算以选择 β_t。这将无法满足我们的开发目标：寻找在大问题比牛顿法计算更加可行的方法。我们能否不进行这些计算而得到共轭方向？幸运的是这个问题的答案是肯定的。

两种用于计算 β_t 的流行方法是：

1. Fletcher-Reeves:

$$\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \tag{8.30}$$

2. Polak-Ribière:

$$\beta_t = \frac{(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}))^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \tag{8.31}$$

对于二次曲面而言，共轭方向确保梯度沿着前一方向大小不变。因此，我们在前一方向上仍然是极小值。其结果是，在 k-维参数空间中，共轭梯度只需要至多 k 次线搜索就能达到极小值。共轭梯度算法如算法 8.9 所示。

算法 8.9 共轭梯度方法

Require: 初始参数 $\boldsymbol{\theta}_0$

Require: 包含 m 个样本的训练集

初始化 $\boldsymbol{\rho}_0 = 0$

初始化 $g_0 = 0$

初始化 $t = 1$

while 没有达到停止准则 **do**

初始化梯度 $\boldsymbol{g}_t = 0$

计算梯度：$\boldsymbol{g}_t \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

计算 $\beta_t = \frac{(\boldsymbol{g}_t - \boldsymbol{g}_{t-1})^\top \boldsymbol{g}_t}{\boldsymbol{g}_{t-1}^\top \boldsymbol{g}_{t-1}}$ (Polak-Ribière)

(非线性共轭梯度：视情况可重置 β_t 为零，例如 t 是常数 k 的倍数时，如 $k = 5$)

计算搜索方向：$\boldsymbol{\rho}_t = -\boldsymbol{g}_t + \beta_t \boldsymbol{\rho}_{t-1}$

执行线搜索寻找：$\epsilon^* = \text{argmin}_\epsilon \frac{1}{m} \sum_{i=1}^m L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_t + \epsilon \boldsymbol{\rho}_t), \boldsymbol{y}^{(i)})$

(对于真正二次的代价函数，存在 ϵ^* 的解析解，而无需显式地搜索)

应用更新：$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t$

$t \leftarrow t + 1$

end while

非线性共轭梯度： 目前，我们已经讨论了用于二次目标函数的共轭梯度法。当然，本章我们主要关注于探索训练神经网络和其他相关深度学习模型的优化方法，其对应的目标函数比二次函数复杂得多。或许令人惊讶，共轭梯度法在这种情况下仍然是适用的，尽管需要作一些修改。没有目标是二次的保证，共轭方向也不再保证在以前方向上的目标仍是极小值。其结果是，**非线性共轭梯度** 算法会包括一些偶尔的重设，共轭梯度法沿未修改的梯度重启线搜索。

实践者报告在实践中使用非线性共轭梯度算法训练神经网络是合理的，尽管在开始非线性共轭梯度前使用随机梯度下降迭代若干步来初始化效果更好。另外，尽管（非线性）共轭梯度算法传统上作为批方法，小批量版本已经成功用于训练神经网络 (Le *et al.*, 2011)。针对神经网路的共轭梯度应用早已被提出，例如缩放的共轭梯度算法 (Moller, 1993)。

8.6.3 BFGS

Broyden-Fletcher-Goldfarb-Shanno（BFGS）算法具有牛顿法的一些优点，但没有牛顿法的计算负担。在这方面，BFGS 和 CG 很像。然而，BFGS 使用了一个更直接的方法近似牛顿更新。回顾牛顿更新由下式给出

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0), \tag{8.32}$$

其中，$\boldsymbol{H}$ 是 J 相对于 $\boldsymbol{\theta}$ 的 Hessian 矩阵在 $\boldsymbol{\theta}_0$ 处的估计。运用牛顿法的主要计算难点在于计算 Hessian 逆 $\boldsymbol{H}^{-1}$。拟牛顿法所采用的方法（BFGS 是其中最突出的）是使用矩阵 $\boldsymbol{M}_t$ 近似逆，迭代地低秩更新精度以更好地近似 $\boldsymbol{H}^{-1}$。

BFGS 近似的说明和推导出现在很多关于优化的教科书中，包括Luenberger (1984)。

当 Hessian 逆近似 $\boldsymbol{M}_t$ 更新时，下降方向 $\boldsymbol{\rho}_t$ 为 $\boldsymbol{\rho}_t = \boldsymbol{M}_t \boldsymbol{g}_t$。该方向上的线搜索用于决定该方向上的步长 ϵ^*。参数的最后更新为：

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t. \tag{8.33}$$

和共轭梯度法相似，BFGS 算法迭代一系列线搜索，其方向含二阶信息。然而和共轭梯度不同的是，该方法的成功并不严重依赖于线搜索寻找该方向上和真正极小值很近的一点。因此，相比于共轭梯度，BFGS 的优点是其花费较少的时间改进每个线搜索。在另一方面，BFGS 算法必须存储 Hessian 逆矩阵 $\boldsymbol{M}$，需要 $O(n^2)$ 的存储空间，使 BFGS 不适用于大多数具有百万级参数的现代深度学习模型。

存储受限的 BFGS（或 L-BFGS） 通过避免存储完整的 Hessian 逆近似 $\boldsymbol{M}$，BFGS 算法的存储代价可以显著降低。L-BFGS 算法使用和 BFGS 算法相同的方法计算 $\boldsymbol{M}$ 的近似，但起始假设是 $\boldsymbol{M}^{(t-1)}$ 是单位矩阵，而不是一步一步都要存储近似。如果使用精确的线搜索，L-BFGS 定义的方向会是相互共轭的。然而，不同于共轭梯

度法，即使只是近似线搜索的极小值，该过程的效果仍然不错。这里描述的无存储的L-BFGS方法可以拓展为包含Hessian矩阵更多的信息，每步存储一些用于更新 $\boldsymbol{M}$ 的向量，且每步的存储代价是 $O(n)$。

8.7 优化策略和元算法

许多优化技术并非真正的算法，而是一般化的模板，可以特定地产生算法，或是并入到很多不同的算法中。

8.7.1 批标准化

批标准化 (Ioffe and Szegedy, 2015) 是优化深度神经网络中最激动人心的最新创新之一。实际上它并不是一个优化算法，而是一个自适应的重参数化的方法，试图解决训练非常深的模型的困难。

非常深的模型会涉及多个函数或层组合。在其他层不改变的假设下，梯度用于如何更新每一个参数。在实践中，我们同时更新所有层。当我们进行更新时，可能会发生一些意想不到的结果，这是因为许多组合在一起的函数同时改变时，计算更新的假设是其他函数保持不变。举一个简单的例子，假设我们有一个深度神经网络，每一层只有一个单元，并且在每个隐藏层不使用激活函数：$\hat{y} = xw_1w_2w_3 \dots w_l$。此处，$w_i$ 表示用于层 i 的权重。层 i 的输出是 $h_i = h_{i-1}w_i$。输出 $\hat{y}$ 是输入 x 的线性函数，但是权重 w_i 的非线性函数。假设我们的代价函数 $\hat{y}$ 上的梯度为 1，所以我们希望稍稍降低 $\hat{y}$。然后反向传播算法可以计算梯度 $\boldsymbol{g} = \nabla_{\boldsymbol{w}}\hat{y}$。想想我们在更新 $\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon\boldsymbol{g}$ 时会发生什么。近似 $\hat{y}$ 的一阶泰勒级数会预测 $\hat{y}$ 的值下降 $\epsilon\boldsymbol{g}^\top\boldsymbol{g}$。如果我们希望 $\hat{y}$ 下降 0.1，那么梯度中的一阶信息表明我们应设置学习率ϵ 为 $\frac{0.1}{\boldsymbol{g}^\top\boldsymbol{g}}$。然而，实际的更新将包括二阶，三阶，直到 l 阶的影响。$\hat{y}$ 的更新值为

$$x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \dots (w_l - \epsilon g_l), \tag{8.34}$$

这个更新中所产生的一个二阶项示例是 $\epsilon^2 g_1 g_2 \prod_{i=3}^{l} w_i$ 。如果 $\prod_{i=3}^{l} w_i$ 很小，那么该项可以忽略不计。而如果层 3 到层 l 的权重都比 1 大时，该项可能会指数级大。这使得我们很难选择一个合适的学习率，因为某一层中参数更新的效果很大程度上取决于其他所有层。二阶优化算法通过考虑二阶相互影响来解决这个问题，但我们可以看到，在非常深的网络中，更高阶的相互影响会很显著。即使是二阶优化算法，计

算代价也很高，并且通常需要大量近似，以免真正计算所有的重要二阶相互作用。因此对于 $n > 2$ 的情况，建立 n 阶优化算法似乎是无望的。那么我们可以做些什么呢?

批标准化提出了一种几乎可以重参数化所有深度网络的优雅方法。重参数化显著减少了多层之间协调更新的问题。批标准化可应用于网络的任何输入层或隐藏层。设 $\boldsymbol{H}$ 是需要标准化的某层的小批量激活函数，排布为设计矩阵，每个样本的激活出现在矩阵的每一行中。为了标准化 $\boldsymbol{H}$，我们将其替换为

$$\boldsymbol{H}' = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \tag{8.35}$$

其中 $\boldsymbol{\mu}$ 是包含每个单元均值的向量，$\boldsymbol{\sigma}$ 是包含每个单元标准差的向量。此处的算术是基于广播向量 $\boldsymbol{\mu}$ 和向量 $\boldsymbol{\sigma}$ 应用于矩阵 $\boldsymbol{H}$ 的每一行。在每一行内，运算是逐元素的，因此 $H_{i,j}$ 标准化为减去 μ_j 再除以 σ_j。网络的其余部分操作 $\boldsymbol{H}'$ 的方式和原网络操作 $\boldsymbol{H}$ 的方式一样。

在训练阶段，

$$\boldsymbol{\mu} = \frac{1}{m}\sum_i \boldsymbol{H}_{i,:} \tag{8.36}$$

和

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m}\sum_i (\boldsymbol{H} - \boldsymbol{\mu})_i^2}, \tag{8.37}$$

其中 δ 是个很小的正值，比如 10^{-8}，以强制避免遇到 $\sqrt{z}$ 的梯度在 $z = 0$ 处未定义的问题。至关重要的是，我们反向传播这些操作，来计算均值和标准差，并应用它们于标准化 $\boldsymbol{H}$。这意味着，梯度不会再简单地增加 h_i 的标准差或均值；标准化操作会除掉这一操作的影响，归零其在梯度中的元素。这是批标准化方法的一个重大创新。以前的方法添加代价函数的惩罚，以鼓励单元标准化激活统计量，或是在每个梯度下降步骤之后重新标准化单元统计量。前者通常会导致不完全的标准化，而后者通常会显著地消耗时间，因为学习算法会反复改变均值和方差而标准化步骤会反复抵消这种变化。批标准化重参数化模型，以使一些单元总是被定义标准化，巧妙地回避了这两个问题。

在测试阶段，$\boldsymbol{\mu}$ 和 $\boldsymbol{\sigma}$ 可以被替换为训练阶段收集的运行均值。这使得模型可以对单一样本评估，而无需使用定义于整个小批量的 $\boldsymbol{\mu}$ 和 $\boldsymbol{\sigma}$。

回顾例子 $\hat{y} = xw_1w_2\dots w_l$，我们看到，我们可以通过标准化 h_{l-1} 很大程度地解决了学习这个模型的问题。假设 x 采样自一个单位高斯。那么 h_{l-1} 也是来自高斯，因为从 x 到 h_l 的变换是线性的。然而，h_{l-1} 不再有零均值和单位方差。使用批

标准化后，我们得到的归一化 $\hat{h}_{l-1}$ 恢复了零均值和单位方差的特性。对于底层的几乎任意更新而言，$\hat{h}_{l-1}$ 仍然保持着单位高斯。然后输出 $\hat{y}$ 可以学习为一个简单的线性函数 $\hat{y} = w_l \hat{h}_{l-1}$。现在学习这个模型非常简单，因为低层的参数在大多数情况下没有什么影响；它们的输出总是重新标准化为单位高斯。只在少数个例中，低层会有影响。改变某个低层权重为 0，可能使输出退化；改变低层权重的符号可能反转 $\hat{h}_{l-1}$ 和 y 之间的关系。这些情况都是非常罕见的。没有标准化，几乎每一个更新都会对 h_{l-1} 的统计量有着极端的影响。因此，批标准化显著地使得模型更易学习。在这个示例中，容易学习的代价是使得底层网络没有用。在我们的线性示例中，较低层不再有任何有害的影响，但它们也不再有任何有益的影响。这是因为我们已经标准化了一阶和二阶统计量，这是线性网络可以影响的所有因素。在具有非线性激活函数的深度神经网络中，较低层可以进行数据的非线性变换，所以它们仍然是有用的。批标准化仅标准化每个单元的均值和方差，以稳定化学习，但允许单元和单个单元的非线性统计量之间的关系发生变化。

由于网络的最后一层能够学习线性变换，实际上我们可能希望移除一层内单元之间的所有线性关系。事实上，这是 Guillaume Desjardins (2015) 中采用的方法，为批标准化提供了灵感。令人遗憾的是，消除所有的线性关联比标准化各个独立单元的均值和标准差代价更高，因此批标准化仍是迄今最实用的方法。

标准化一个单元的均值和标准差会降低包含该单元的神经网络的表达能力。为了保持网络的表现力，通常会将批量隐藏单元激活 $\boldsymbol{H}$ 替换为 $\gamma \boldsymbol{H}' + \boldsymbol{\beta}$，而不是简单地使用标准化的 $\boldsymbol{H}'$。变量 $\boldsymbol{\gamma}$ 和 $\boldsymbol{\beta}$ 是允许新变量有任意均值和标准差的学习参数。乍一看，这似乎是无用的——为什么我们将均值设为 0，然后又引入参数允许它被重设为任意值 $\boldsymbol{\beta}$？答案是新的参数可以表示旧参数作为输入的同一族函数，但是新参数有不同的学习动态。在旧参数中，$\boldsymbol{H}$ 的均值取决于 $\boldsymbol{H}$ 下层中参数的复杂关联。在新参数中，$\boldsymbol{\gamma H}' + \boldsymbol{\beta}$ 的均值仅由 $\boldsymbol{\beta}$ 确定。新参数很容易通过梯度下降来学习。

大多数神经网络层会采取 $\phi(\boldsymbol{XW} + \boldsymbol{b})$ 的形式，其中 ϕ 是某个固定的非线性激活函数，如整流线性变换。自然想到我们应该将批标准化应用于输入 $\boldsymbol{X}$ 还是变换后的值 $\boldsymbol{XW} + \boldsymbol{b}$。Ioffe and Szegedy (2015) 推荐后者。更具体地，$\boldsymbol{XW} + \boldsymbol{b}$ 应替换为 $\boldsymbol{XW}$ 的标准化形式。偏置项应被忽略，因为参数 $\boldsymbol{\beta}$ 会加入批标准化重参数化，它是冗余的。一层的输入通常是前一层的非线性激活函数（如整流线性函数）的输出。因此，输入的统计量更符合非高斯，而更不服从线性操作的标准化。

第九章所述的卷积网络，在特征映射中每个空间位置同样地标准化 $\boldsymbol{\mu}$ 和 $\boldsymbol{\sigma}$ 是很重要的，能使特征映射的统计量在不同的空间位置，仍然保持相同。

8.7.2 坐标下降

在某些情况下，将一个优化问题分解成几个部分，可以更快地解决原问题。如果我们相对于某个单一变量 x_i 最小化 $f(\boldsymbol{x})$，然后相对于另一个变量 x_j 等等，反复循环所有的变量，我们会保证到达（局部）极小值。这种做法被称为**坐标下降**（coordinate descent），因为我们一次优化一个坐标。更一般地，**块坐标下降**（block coordinate descent）是指对于某个子集的变量同时最小化。术语“坐标下降”通常既指块坐标下降，也指严格的单个坐标下降。

当优化问题中的不同变量能够清楚地分成相对独立的组，或是当优化一组变量明显比优化所有变量效率更高时，坐标下降最有意义。例如，考虑代价函数

$$J(\boldsymbol{H}, \boldsymbol{W}) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} \left(\boldsymbol{X} - \boldsymbol{W}^\top \boldsymbol{H}\right)^2_{i,j}. \tag{8.38}$$

该函数描述了一种被称为稀疏编码的学习问题，其目标是寻求一个权重矩阵 $\boldsymbol{W}$，可以线性解码激活值矩阵 $\boldsymbol{H}$ 以重构训练集 $\boldsymbol{X}$。稀疏编码的大多数应用还涉及到权重衰减或 $\boldsymbol{W}$ 列范数的约束，以避免极小 $\boldsymbol{H}$ 和极大 $\boldsymbol{W}$ 的病态解。

函数 J 不是凸的。然而，我们可以将训练算法的输入分成两个集合：字典参数 $\boldsymbol{W}$ 和编码表示 $\boldsymbol{H}$。最小化关于这两者之一的任意一组变量的目标函数都是凸问题。因此，块坐标下降允许我们使用高效的凸优化算法，交替固定 $\boldsymbol{H}$ 优化 $\boldsymbol{W}$ 和固定 $\boldsymbol{W}$ 优化 $\boldsymbol{H}$。

当一个变量的值很大程度地影响另一个变量的最优值时，坐标下降不是一个很好的方法，如函数 $f(\boldsymbol{x}) = (x_1 - x_2)^2 + \alpha(x_1^2 + x_2^2)$，其中 α 是正值常数。第一项鼓励两个变量具有相似的值，而第二项鼓励它们接近零。解是两者都为零。牛顿法可以一步解决这个问题，因为它是一个正定二次问题。但是，对于小值 α 而言，坐标下降会使进展非常缓慢，因为第一项不允许单个变量变为和其他变量当前值显著不同的值。

8.7.3 Polyak 平均

Polyak 平均 (Polyak and Juditsky, 1992) 会平均优化算法在参数空间访问轨迹中的几个点。如果 t 次迭代梯度下降访问了点 $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(t)}$，那么 Polyak 平均算法的输出是 $\hat{\boldsymbol{\theta}}^{(t)} = \frac{1}{t}\sum_i \boldsymbol{\theta}^{(i)}$。在某些问题中，如梯度下降应用于凸问题时，这种方法具有较强的收敛保证。当应用于神经网络时，其验证更多是启发式的，但在实践中表

现良好。基本想法是，优化算法可能会来回穿过山谷好几次而没经过山谷底部附近的点。尽管两边所有位置的均值应比较接近谷底。

在非凸问题中，优化轨迹的路径可以非常复杂，并且经过了许多不同的区域。包括参数空间中遥远过去的点，可能与当前点在代价函数上相隔很大的障碍，看上去不像一个有用的行为。其结果是，当应用 Polyak 平均于非凸问题时，通常会使用指数衰减计算平均值：

$$\hat{\boldsymbol{\theta}}^{(t)} = \alpha\hat{\boldsymbol{\theta}}^{(t-1)} + (1-\alpha)\boldsymbol{\theta}^{(t)}. \tag{8.39}$$

这个计算平均值的方法被用于大量数值应用中。最近的例子请查看Szegedy *et al.* (2015)。

8.7.4 监督预训练

有时，如果模型太复杂难以优化，或是如果任务非常困难，直接训练模型来解决特定任务的挑战可能太大。有时训练一个较简单的模型来求解问题，然后使模型更复杂会更有效。训练模型来求解一个简化的问题，然后转移到最后的问题，有时也会更有效些。这些在直接训练目标模型求解目标问题之前，训练简单模型求解简化问题的方法统称为 **预训练**（ pretraining ）。

贪心算法（ greedy algorithm ）将问题分解成许多部分，然后独立地在每个部分求解最优值。令人遗憾的是，结合各个最佳的部分不能保证得到一个最佳的完整解。然而，贪心算法计算上比求解最优联合解的算法高效得多，并且贪心算法的解在不是最优的情况下，往往也是可以接受的。贪心算法也可以紧接一个 **精调**（ fine-tuning ）阶段，联合优化算法搜索全问题的最优解。使用贪心解初始化联合优化算法，可以极大地加速算法，并提高寻找到的解的质量。

预训练算法，特别是贪心预训练，在深度学习中是普遍存在的。在本节中，我们会具体描述这些将监督学习问题分解成其他简化的监督学习问题的预训练算法。这种方法被称为 **贪心监督预训练**（ greedy supervised pretraining)。

在贪心监督预训练的原始版本 (Bengio *et al.*, 2007c) 中，每个阶段包括一个仅涉及最终神经网络的子集层的监督学习训练任务。贪心监督预训练的一个例子如图 8.7 所示，其中每个附加的隐藏层作为浅层监督多层感知机的一部分预训练，以先前训练的隐藏层输出作为输入。Simonyan and Zisserman (2015) 预训练深度卷积网络（ 11 层权重 ），然后使用该网络前四层和最后三层初始化更深的网络（ 多达 19 层

权重），并非一次预训练一层。非常深的新网络的中间层是随机初始化的。然后联合训练新网络。还有一种选择，由Yu *et al.* (2010) 提出，将先前训练多层感知机的输出，以及原始输入，作为每个附加阶段的输入。

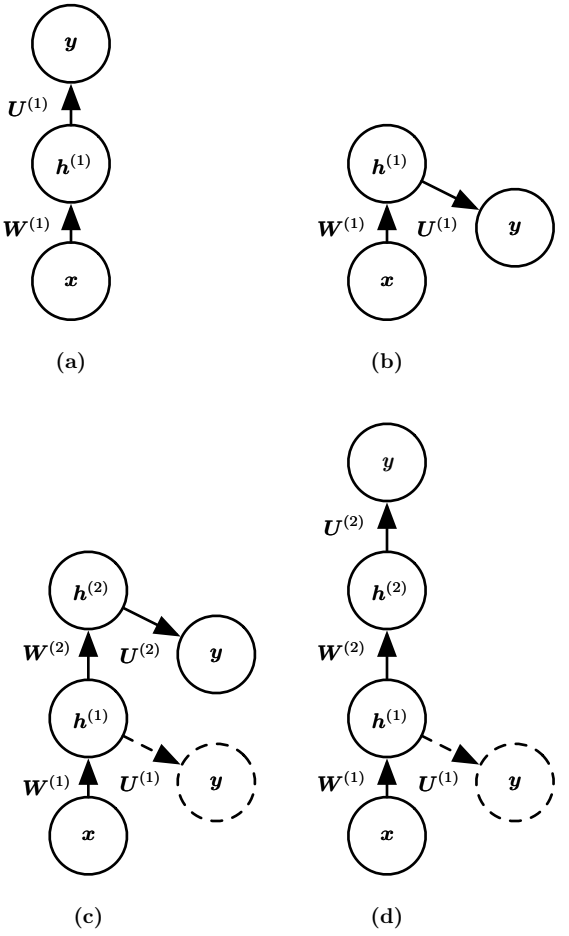

图 8.7: 一种形式的贪心监督预训练的示意图 (Bengio *et al.*, 2007a)。(a) 我们从训练一个足够浅的架构开始。(b) 同一个架构的另一描绘。(c) 我们只保留原始网络的输入到隐藏层，并丢弃隐藏到输出层。我们将第一层隐藏层的输出作为输入发送到另一监督单隐层 MLP（使用与第一个网络相同的目标训练），从而可以添加第二层隐藏层。这可以根据需要重复多层。(d) 所得架构的另一种描绘，可视为前馈网络。为了进一步改进优化，我们可以联合地精调所有层（仅在该过程的结束或者该过程的每个阶段）。

为什么贪心监督预训练会有帮助呢？最初由 Bengio *et al.* (2007d) 提出的假说

是，其有助于更好地指导深层结构的中间层的学习。一般情况下，预训练对于优化和泛化都是有帮助的。

另一个与监督预训练有关的方法扩展了迁移学习的想法：Yosinski *et al.* (2014) 在一组任务上预训练了 8 层权重的深度卷积网络（1000 个 ImageNet 对象类的子集），然而用该网络的前 k 层初始化同样规模的网络。然后第二个网络的所有层（上层随机初始化）联合训练以执行不同的任务（1000 个 ImageNet 对象类的另一个子集），但训练样本少于第一个任务。神经网络中另一个和迁移学习相关的方法将在第 15.2 节讨论。

另一条相关的工作线是 **FitNets** (Romero *et al.*, 2015) 方法。这种方法始于训练深度足够低和宽度足够大（每层单元数），容易训练的网络。然后，这个网络成为第二个网络（被指定为 **学生**）的 **老师**。学生网络更深更窄（11 至 19 层），且在正常情况下很难用 SGD 训练。训练学生网络不仅需要预测原任务的输出，还需要预测教师网络中间层的值，这样使得训练学生网络变得更容易。这个额外的任务说明了隐藏层应如何使用，并且能够简化优化问题。附加参数被引入来从更深的学生网络中间层去回归 5 层教师网络的中间层。然而，该目标是预测教师网络的中间隐藏层，并非预测最终分类目标。学生网络的低层因而具有两个目标：帮助学生网络的输出完成其目标和预测教师网络的中间层。尽管一个窄而深的网络似乎比宽而浅的网络更难训练，但窄而深网络的泛化能力可能更好，并且如果其足够窄，参数足够少，那么其计算代价更小。没有隐藏层的提示，学生网络在训练集和测试集上的实验表现都很差。因而中间层的提示是有助于训练很难训练的网络的方法之一，但是其他优化技术或是架构上的变化也可能解决这个问题。

8.7.5 设计有助于优化的模型

改进优化的最好方法并不总是改进优化算法。相反，深度模型中优化的许多改进来自于设计易于优化的模型。

原则上，我们可以使用呈锯齿非单调模式上上下下的激活函数，但是，这将使优化极为困难。在实践中，*选择一族容易优化的模型比使用一个强大的优化算法更重要*。神经网络学习在过去 30 年的大多数进步主要来自于改变模型族，而非改变优化过程。1980 年代用于训练神经网络的带动量的随机梯度下降，仍然是现代神经网络应用中的前沿算法。

具体来说，现代神经网络的设计选择体现在层之间的线性变换，几乎处处可导

的激活函数，和大部分定义域都有明显的梯度。特别地，创新的模型，如LSTM，整流线性单元和maxout单元都比先前的模型（如基于sigmoid单元的深度网络）使用更多的线性函数。这些模型都具有简化优化的性质。如果线性变换的Jacobian具有相对合理的奇异值，那么梯度能够流经很多层。此外，线性函数在一个方向上一致增加，所以即使模型的输出远离正确值，也可以简单清晰地计算梯度，使其输出方向朝降低损失函数的方向移动。换言之，现代神经网络的设计方案旨在使其*局部*梯度信息合理地对应着移向一个遥远的解。

其他的模型设计策略有助于使优化更简单。例如，层之间的线性路径或是跳跃连接减少了从较低层参数到输出最短路径的长度，因而缓解了梯度消失的问题(Srivastava *et al.*, 2015)。一个和跳跃连接相关的想法是添加和网络中间隐藏层相连的输出的额外副本，如 GoogLeNet (Szegedy *et al.*, 2014a) 和深度监督网络 (Lee *et al.*, 2014)。这些 ‘‘辅助头’’ 被训练来执行和网络顶层主要输出相同的任务，以确保底层网络能够接受较大的梯度。当训练完成时，辅助头可能被丢弃。这是之前小节介绍到的预训练策略的替代方法。以这种方式，我们可以在一个阶段联合训练所有层，而不改变架构，使得中间层（特别是低层）能够通过更短的路径得到一些如何更新的有用信息。这些信息为底层提供了误差信号。

8.7.6 延拓法和课程学习

正如第8.2.7节探讨的，许多优化挑战都来自于代价函数的全局结构，不能仅通过局部更新方向上更好的估计来解决。解决这个问题的主要方法是尝试初始化参数到某种区域内，该区域可以通过局部下降很快连接到参数空间中的解。

延拓法（continuation method）是一族通过挑选初始点使优化更容易的方法，以确保局部优化花费大部分时间在表现良好的空间。延拓法的背后想法是构造一系列具有相同参数的目标函数。为了最小化代价函数 $J(\boldsymbol{\theta})$，我们构建新的代价函数 $\{J^{(0)}, \dots, J^{(n)}\}$。这些代价函数的难度逐步提高，其中 $J^{(0)}$ 是最容易最小化的，$J^{(n)}$ 是最难的，真正的代价函数驱动整个过程。当我们说 $J^{(i)}$ 比 $J^{(i+1)}$ 更容易时，是指其在更多的 $\boldsymbol{\theta}$ 空间上表现良好。随机初始化更有可能落入局部下降可以成功最小化代价函数的区域，因为其良好区域更大。这系列代价函数设计为前一个解是下一个的良好初始点。因此，我们首先解决一个简单的问题，然后改进解以解决逐步变难的问题，直到我们求解真正问题的解。

传统的延拓法（用于神经网络训练之前的延拓法）通常基于平滑目标函数。读

者可以查看 Wu (1997) 了解这类方法的示例，以及一些相关方法的综述。延拓法也和参数中加入噪声的模拟退火紧密相关 (Kirkpatrick *et al.*, 1983)。延拓法在最近几年非常成功。参考 Mobahi and Fisher (2015) 了解近期文献的概述，特别是在 AI 方面的应用。

传统上，延拓法主要用来克服局部极小值的问题。具体地，它被设计来在有很多局部极小值的情况下，求解一个全局最小点。这些连续方法会通过 ‘‘模糊’’ 原来的代价函数来构建更容易的代价函数。这些模糊操作可以是用采样来近似

$$J^{(i)}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\theta}' \sim \mathcal{N}(\boldsymbol{\theta}'; \boldsymbol{\theta}, \sigma^{(i)2})} J(\boldsymbol{\theta}') \tag{8.40}$$

这个方法的直觉是有些非凸函数在模糊后会近似凸的。在许多情况下，这种模糊保留了关于全局极小值的足够信息，我们可以通过逐步求解模糊更少的问题来求解全局极小值。这种方法有三种可能失败的方式。首先，它可能成功地定义了一连串代价函数，并从开始的一个凸函数起（逐一地）沿着函数链最佳轨迹逼近全局最小值，但可能需要非常多的逐步代价函数，整个过程的成本仍然很高。另外，即使延拓法可以适用，NP-hard 的优化问题仍然是 NP-hard。其他两种延拓法失败的原因是不实用。其一，不管如何模糊，函数都没法变成凸的，比如函数 $J(\boldsymbol{\theta}) = -\boldsymbol{\theta}^\top \boldsymbol{\theta}$。其二，函数可能在模糊后是凸的，但模糊函数的最小值可能会追踪到一个局部最小值，而非原始代价函数的全局最小值。

尽管延拓法最初用来解决局部最小值的问题，而局部最小值已不再认为是神经网络优化中的主要问题了。幸运的是，延拓法仍然有所帮助。延拓法引入的简化目标函数能够消除平坦区域，减少梯度估计的方差，提高 Hessian 矩阵的条件数，使局部更新更容易计算，或是改进局部更新方向与朝向全局解方向之间的对应关系。

Bengio *et al.* (2009) 指出被称为**课程学习**（curriculum learning）或者**塑造**（shaping）的方法可以被解释为延拓法。课程学习基于规划学习过程的想法，首先学习简单的概念，然后逐步学习依赖于这些简化概念的复杂概念。之前这一基本策略被用来加速动物训练过程 (Skinner, 1958; Peterson, 2004; Krueger and Dayan, 2009) 和机器学习过程 (Solomonoff, 1989; Elman, 1993; Sanger, 1994)。Bengio *et al.* (2009) 验证这一策略为延拓法，通过增加简单样本的影响（通过分配它们较大的系数到代价函数，或者更频繁地采样），先前的 $J^{(i)}$ 会变得更容易。实验证明，在大规模的神经语言模型任务上使用课程学习，可以获得更好的结果。课程学习已经成功应用于大量的自然语言 (Spitkovsky *et al.*, 2010; Collobert *et al.*, 2011a; Mikolov *et al.*, 2011b; Tu and Honavar, 2011) 和计算机视觉 (Kumar *et al.*, 2010; Lee and