第七章　网络优化与正则化

任何数学技巧都不能弥补信息的缺失。

— Cornelius Lanczos,1964

虽然神经网络具有非常强的表达能力，但是当应用神经网络模型到机器学习时依然存在一些难点。主要分为两大类：（1）优化问题：神经网络模型是一个非凸函数，再加上在深度网络中的梯度消失问题，很难进行优化；另外，深度神经网络模型一般参数比较多，训练数据也比较大，会导致训练的效率比较低。（2）泛化问题：因为神经网络的拟合能力强，反而容易在训练集上产生过拟合。因此，在训练深度神经网络时，同时也需要掌握一定的技巧。目前，人们在大量的实践中总结了一些经验技巧，从优化和正则化两个方面来提高学习效率并得到一个好的网络模型。

7.1　网络优化

深度神经网络是一个高度非线性的模型，其风险函数也是一个非凸问题。在非凸问题中，一个会存在一些局部最优点。

有效地学习深度神经网络的参数是一个具有挑战性的问题，其主要原因有以下几个方面。

网络结构多样性 神经网络的种类非常多，比如卷积网络、循环网络等，其结构也非常不同。有些比较深，有些比较宽。不同参数在网络中的作用也有很大的差异，比如连接权重和偏置的不同，以及循环网络中循环连接上的权重和其它权重的不同。

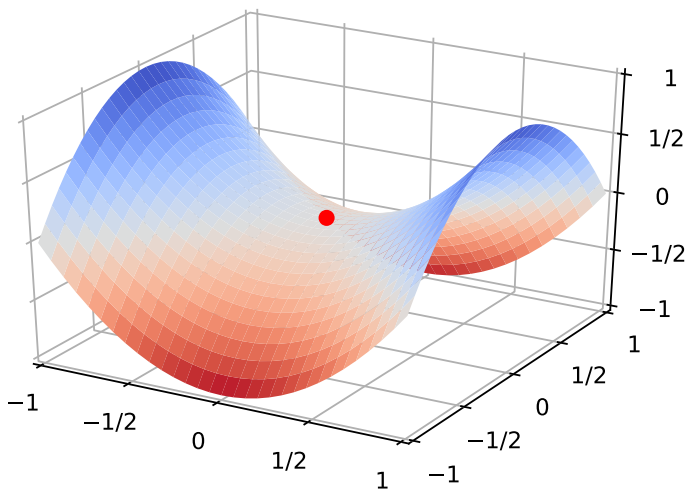


图 7.1: 鞍点示例。

网络结构的多样性导致了很难找到一种通用的优化方法。不同的优化方法在不同网络结构上的差异也都比较大。

此外，网络的超参数一般也比较多，这也给优化带来很大的挑战。

高维变量的非凸优化 深度神经网络的参数非常多，其参数学习是在非常高维空间中的非凸优化问题，其挑战和在低维空间的非凸优化问题有所不同。低维空间的非凸优化问题主要是存在一些局部最优点。采用梯度下降方法时，不合适的参数初始化会导致陷入局部最优点，因此主要的难点是如何选择初始化参数和逃离局部最优点。

Dauphin et al. [2014] 指出在高维空间中，非凸优化的难点并不在于如果逃离局部最优点，而是如何逃离鞍点。鞍点（saddle point）是梯度为0，但是在一些维度上是最高点，在另一些维度上是最低点，如图7.1所示。梯度下降方法同样很难从这些鞍点中逃离。

鞍点的叫法是因为其形状像马鞍。

目前，深度神经网络的参数学习主要是通过梯度下降方法来寻找一组可以最小化结构风险的参数。在具体实现中，梯度下降法可以分为：批量梯度下降、随机梯度下降以及小批量梯度下降三种形式。根据不同的数据量和参数量，可以选择一种具体的实现形式。除了在收敛效果和效率上的差异，这三种方法都

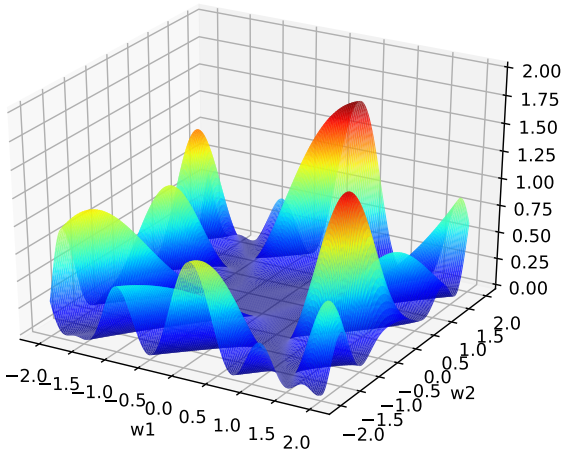


图 7.2: 神经网络中的非凸优化问题。

存在一些共同的问题，比如1）如何初始化参数；2）预处理数据；3）如何选择合适的学习率，避免陷入局部最优等。

7.1.1 参数初始化

神经网络的训练过程中的参数学习是基于梯度下降法进行优化的。梯度下降法需要在开始训练时给每一个参数赋一个初始值。这个初始值的选取十分关键。在感知器和 logistic 回归的训练中，我们一般将参数全部初始化为0。但是这在神经网络的训练中会存在一些问题。因为如果参数都为0，在第一遍前向计算时，所有的隐层神经元的激活值都相同。这样会导致深层神经元没有区分性。这种现象也称为**对称权重**现象。

为了打破这个平衡，比较好的方式是对每个参数都随机初始化，这样使得不同神经元之间的区分性更好。

但是一个问题是如何选取随机初始化的区间呢？如果参数太小，会导致神经元的输入过小。经过多层之后信号就慢慢消失了。参数过小还会使得 sigmoid 型激活函数丢失非线性的能力。以 logistic 函数为例，在 0 附近基本上是近似线

性的。这样多层神经网络的优势也就不存在了。如果参数取得太大，会导致输入状态过大。对于 sigmoid 型激活函数来说，激活值变得饱和，从而导致梯度接近于 0。

因此，如果要高质量地训练一个网络，给参数选取一个合适的初始化区间是非常重要的。一般而言，参数初始化的区间应该不用神经元的性质进行差异化的设置。如果一个神经元的输入连接很多，它的每个输入连接上的权重就应该小一些，以避免神经元的输出过大（当激活函数为 ReLU 时）或过饱和（当激活函数为 sigmoid 函数时）。

经常使用的初始化方法有以下几种：

Gaussian 分布初始化

Gaussian 初始化方法是最简单的初始化方法，参数从一个固定均值（比如 0）和固定方差（比如 0.01）的 Gaussian 分布进行随机初始化。

初始化一个深度网络时，一个比较好的初始化方案是保持每个神经元输入的方差为一个常量。当一个神经元的输入连接数量为 n_{in} 时，可以设置其输入连接权重以 $\mathcal{N}(0, \sqrt{\frac{1}{n_{in}}})$ 的 Gaussian 分布进行初始化。如果同时考虑输出连接的数量 n_{out}，则可以按 $\mathcal{N}(0, \sqrt{\frac{2}{n_{in}+n_{out}}})$ 的 Gaussian 分布进行初始化。

均匀分布初始化

均匀分布初始化是在一个给定的区间 $[-r, r]$ 内采用均匀分布来初始化参数。超参数 r 的设置也可以按神经元的连接数量进行自适应的调整。

Xavier 初始化方法中，Xavier 是发明者 Xavier Glorot 的名字。

***Xavier* 初始化方法** Glorot and Bengio [2010] 提出一个自动计算超参数 r 的方法，参数可以在 $[-r, r]$ 内采用均匀分布进行初始化。

如果神经元激活函数为 logistic 函数，对于第 $l-1$ 到 l 层的权重参数区间 r 可以设置为

$$r = \sqrt{\frac{6}{n^{l-1}+n^{1}}}, \tag{7.1}$$

这里 n^l 是第 l 层神经元个数，n^{l-1} 是第 $l-1$ 层神经元个数。

对于 tanh 函数，r 可以设置为

$$r = 4\sqrt{\frac{6}{n^{l-1}+n^{l}}}. \tag{7.2}$$

假设第 l 层的一个隐藏层神经元 z^l，其接受前一层的 n^{l-1} 个神经元的输出 $a_i^{(l-1)}$，$i \in [1, n^{(l-1)}]$，

$$z^l = \sum_{i=1}^{n} w_i^l a_i^{(l-1)} \tag{7.3}$$

为了避免初始化参数使得激活值变得饱和，我们需要尽量使得 z^l 处于激活函数的线性区间，也就是其绝对值比较小的值。这时该神经元的激活值为 $a^l = f(z^l) \approx z^l$。

假设 w_i^l 和 $a_i^{(l-1)}$ 都是相互独立，并且均值都为0，则 a 的均值为

$$\mathbb{E}[a^l] = \mathbb{E}[\sum_{i=1}^{n} w_i^l a_i^{(l-1)}] = \sum_{i=1}^{d} \mathbb{E}[\mathbf{w}_i]\mathbb{E}[a_i^{(l-1)}] = 0. \tag{7.4}$$

a^l 的方差为

$$\begin{aligned} \mathrm{Var}[a^l] &= \mathrm{Var}[\sum_{i=1}^{n^{(l-1)}} w_i^l a_i^{(l-1)}] && (7.5)\\ &= \sum_{i=1}^{n^{(l-1)}} \mathrm{Var}[w_i^l]\,\mathrm{Var}[a_i^{(l-1)}] && (7.6)\\ &= n^{(l-1)}\,\mathrm{Var}[w_i^l]\,\mathrm{Var}[a_i^{(l-1)}]. && (7.7) \end{aligned}$$

也就是说，输入信号的方差在经过该神经元后被放大或缩小了 $n^{(l-1)}\,\mathrm{Var}[w_i^l]$ 倍。为了使得在经过多层网络后，信号不被过分放大或过分减弱，我们尽可能保持每个神经元的输入和输出的方差一致。这样 $n^{(l-1)}\,\mathrm{Var}[w_i^l]$ 设为1比较合理，即

$$\mathrm{Var}[w_i^l] = \frac{1}{n^{(l-1)}}. \tag{7.8}$$

同理，为了使得在反向传播中，误差信号也不被放大或缩小，需要将 w_i^l 的方差保持为

$$\mathrm{Var}[w_i^l] = \frac{1}{n^{(l)}}. \tag{7.9}$$

作为折中，同时考虑信号在前向和反向传播中都不被放大或缩小，可以设置

$$\mathrm{Var}[w_i^l] = \frac{2}{n^{(l-1)} + n^{(l)}}. \tag{7.10}$$

假设随机变量 x 在区间 $[a, b]$ 内均匀分布，则其方差为：

$$\mathrm{Var}[x] = \frac{(b-a)^2}{12}. \tag{7.11}$$

因此，若让 $w_i^l \in [-r, r]$，并且 $\mathrm{Var}[w_i^l] = 1$，则 r 的取值为

$$r = \sqrt{\frac{6}{n^{l-1} + n^1}}. \tag{7.12}$$

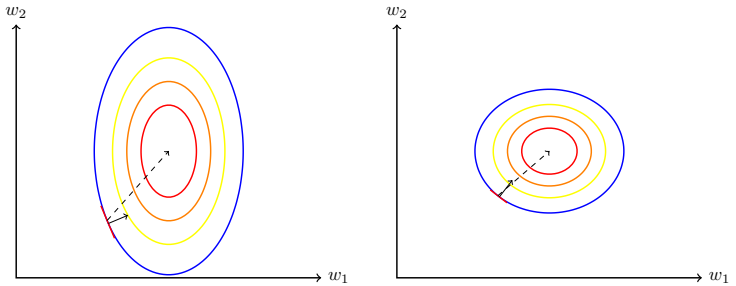
7.1.2 数据预处理

一般而言，原始的训练数据中，每一维特征的来源以及度量单位不同，会造成这些特征值的分布范围往往差异很大。当我们计算不同样本之间的欧式距离时，取值范围大的特征会起到主导作用。这样，对于基于相似度比较的机器学习方法（比如最近邻分类器），必须先对样本进行预处理，将各个维度的特征归一化到同一个取值区间，并且消除不同特征之间的相关性，才能获得比较理想的结果。虽然神经网络可以通过参数的调整来适应不同特征的取值范围，但是会导致训练效率比较低。

假设一个只有一层的网络 $y = \tanh(w_1 x_1 + w_2 x_2 + b)$，其中 $x_1 \in [0, 10]$，$x_2 \in [0, 1]$。之前我们提到 tanh 函数的导数在区间 $[-2, 2]$ 上是敏感的，其余的导数接近于 0。因此，如果 $w_1 x_1 + w_2 x_2 + b$ 过大或过小，都会导致梯度过小，难以训练。为了提高训练效率，我们需要使 $w_1 x_1 + w_2 x_2 + b$ 在 $[-2, 2]$ 区间，我们需要将 w_1 设得小一点，比如在 $[-0.1, 0.1]$ 之间。可以想象，如果数据维数很多时，我们很难这样精心去选择每一个参数。因此，如果每一个特征的取值范围都在相似的区间，比如 $[0, 1]$ 或者 $[-1, 1]$，我们就不太需要区别对待每一个参数，减少人工干预。

除了参数初始化之外，不同特征取值范围差异比较大时还会梯度下降法的搜索效率。图7.3给出了数据归一化对梯度的影响。其中，图7.3a为未归一化数据的等高线图。取值范围不同会造成在大多数位置上的梯度方向并不是最优的搜索方向。当使用梯度下降法寻求最优解时，会导致需要很多次迭代才能收敛。如果我们把数据归一化为取值范围相同，如图7.3b所示，大部分位置的梯度方向近似于最优搜索方向。这样，在梯度下降求解时，每一步梯度的方向都基本指向最小值，训练效率会大大提高。

归一化的方法有很多种，比如之前我们介绍的 sigmoid 型函数等都可以将不同取值范围的特征挤压到一个比较受限的区间。这里，我们介绍几种在神经网络中经常使用的归一化方法。



(a) 未归一化数据的梯度 (b) 归一化数据的梯度

图 7.3: 数据归一化对梯度的影响。

标准归一化 标准归一化也叫z-score归一化，来源于统计上的标准分数。将每一个维特征都处理为符合标准正态分布（均值为0，标准差为1）。假设有 N 个样本 $\{\mathbf{x}^{(i)}\}, i = 1, \cdots, N$，对于每一维特征 x，我们先计算它的均值和标准差：

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x^{(i)}, \tag{7.13}$$

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x^{(i)} - \mu)^2. \tag{7.14}$$

然后，将特征 $x^{(i)}$ 减去均值，并除以标准差，得到新的特征值 $\hat{x}^{(i)}$。

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mu}{\sigma}, \tag{7.15}$$

这里 σ 不能为0。如果标准差为0，说明这一维特征没有任务区分性，可以直接删掉。

在标准归一化之后，每一维特征都服从标准正态分布。

缩放归一化 另外一种非常简单的归一化是缩放归一化：通过缩放将特征取值范围归一到 $[0, 1]$ 或 $[-1, 1]$ 之间：

$$\hat{x}^{(i)} = \frac{x^{(i)} - \min(x)}{\max(x) - \min(x)}, \tag{7.16}$$

其中，$\min(x)$ 和 $\max(x)$ 分别为这一维特征在所有样本上的最小值和最大值。

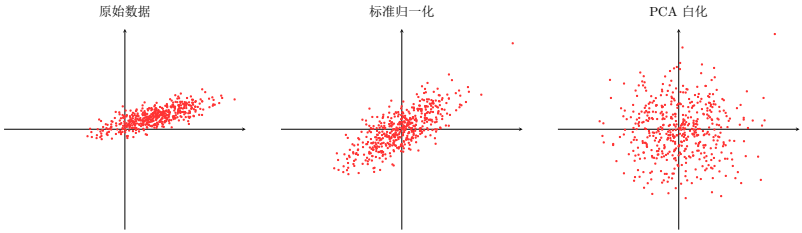


图 7.4: 数据归一化示例

白化 白化（whitening）是一种重要的预处理方法，用来降低输入数据特征之间的冗余性。输入数据经过白化处理后，特征之间相关性较低，并且所有特征具有相同的方差。

白化的一个主要实现方式是使用主成分分析（Principal Component Analysis，PCA）方法去除掉各个成分之间的相关性。

7.1.3 逐层归一化

在深层神经网络中，中间某一层的输入是其之前的神经层的输出。因此，其之前的神经层的参数变化会导致其输入的分布发生较大的差异。在使用随机梯度下降来训练网络时，每次参数更新都会导致网络中间每一层的输入的分布发生改变。越深的层，其输入的分布会改变得越明显。就像一栋高楼，低楼层发生一个较小的偏移，都会导致高楼层较大的偏移。

从机器学习角度来看，如果某个神经层的输入分布发生了改变，那么其参数需要重新学习。这种现象叫做内部协变量偏移（internal covariate shift）。

为了解决内部协方差偏移问题，就要使得每一个神经层的输入的分布在训练过程中要保持一致。最简单直接的方法就是对每一个神经层都进行归一化操作，使其分布保存稳定。下面介绍几种比较常用的逐层归一化方法：批量归一化、层归一化和其它一些方法。

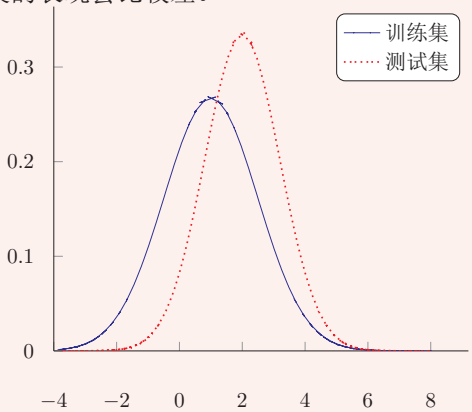
这里的逐层归一化方法是指可以应用在深层神经网络中的任何一个中间层。一般情况下不需要对每一层进行归一化。

批量归一化

批量归一化（batch normalization）方法 [Ioffe and Szegedy, 2015] 是一种有效的逐层归一化方法，可以对神经网络中任意的中间层进行归一化操作。

机器学习小知识 | 协变量偏移

在传统机器学习中，一个常见的问题的协变量偏移（Covariate Shift）。协变量是一个统计学概念，是可能影响预测结果的统计变量。在机器学习中，协变量可以看作是输入。一般的机器学习算法都要求输入在训练集和测试集上的分布是相似的。如果不满足这个要求，这些学习算法在测试集的表现会比较差。



对于一个深层神经网络，假设第 l 层的净输入为 $\mathbf{z}^{(l)}$，神经元的输出为 $\mathbf{a}^{(l)}$，有

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)}) = f(W\mathbf{a}^{(l-1)} + \mathbf{b}), \tag{7.17}$$

其中 $f(\cdot)$ 是激活函数，W 和 $\mathbf{b}$ 是可学习的参数。

虽然归一化操作可以应用在输入 $\mathbf{a}^{(l-1)}$ 上，但其分布性质不如 $\mathbf{z}^{(l)}$ 稳定。因此，在实践中归一化操作一般应用仿射变换之后，在激活函数之前。

为了减少内部协变量偏移问题，就要使得净输入 $\mathbf{z}^{(l)}$ 的分布一致，比如都归一化到标准正态分布。我们可以利用类似第7.1.2节中数据预处理的方法进行对 $\mathbf{z}^{(l)}$ 进行归一化。相当于每一层都进行一次数据预处理，从而加速收敛速度。但是，这里的归一化需要在每一层进行操作，所以要求效率要高。因此白化的归一化方法就不太合适。为了提高归一化效率，这里使用标准归一化，对净输入 $\mathbf{z}^{(l)}$ 的每一维都归一到标准正态分布。

$$\hat{\mathbf{z}}^{(l)} = \frac{\mathbf{z}^{(l)} - \mathbb{E}[\mathbf{z}^{(l)}]}{\sqrt{\mathrm{Var}[\mathbf{z}^{(l)}] + \epsilon}}, \tag{7.18}$$

其中 $\mathbb{E}[\mathbf{z}^{(l)}]$ 和 $\text{Var}[\mathbf{z}^{(l)}]$ 是指当前参数下，$\mathbf{z}^{(l)}$ 的每一维在整个训练集上的期望和方差。因为目前主要的训练方法是基于小批量的随机梯度下降方法，所以准确地计算 $\mathbf{z}^{(l)}$ 的期望和方差是不可行的。因此，$\mathbf{z}^{(l)}$ 的期望和方差通常用当前小批量样本集的均值和方差近似估计。

给定一个小批量的样本集合，假设神经网络的第 l 层神经的净输入 $\{\mathbf{z}^{(1,l)}, \cdots, \mathbf{z}^{(K,l)}\}$，其均值和方差为

$$\mu_{\mathcal{B}} = \frac{1}{K}\sum_{i=1}^{K}\mathbf{z}^{(k,l)}, \tag{7.19}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{K}\sum_{k=1}^{K}(\mathbf{z}^{(k,l)} - \mu_{\mathcal{B}}) \odot (\mathbf{z}^{(k,l)} - \mu_{\mathcal{B}}). \tag{7.20}$$

对净输入 $\mathbf{z}^{(l)}$ 的标准归一化会使得其取值集中的 0 附近，如果使用 sigmoid 型激活函数时，这个取值区间刚好是接近线性变换的区间，减弱了神经网络的非线性性质。因此，为了使得归一化不对网络的表示能力造成负面影响，我们可以通过一个附加的缩放和平移变换改变取值区间。

$$\hat{\mathbf{z}}^{(l)} = \frac{\mathbf{z}^{(l)} - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \odot \gamma + \beta \tag{7.21}$$

$$\equiv \text{BN}_{\gamma,\beta}(\mathbf{z}^{(l)}), \tag{7.22}$$

其中 γ 和 β 分别代表缩放和平移的参数向量。从最保守的角度考虑，可以通过来标准归一化的逆变换来使得归一化后的变量可以被还原为原来的值。当 $\gamma = \sqrt{\sigma_{\mathcal{B}}^2}$，$\beta = \mu_{\mathcal{B}}$ 时，$\hat{\mathbf{z}}^{(l)} = \mathbf{z}^{(l)}$。

批量归一化操作可以看作是一个特殊的神经层，加在每一层非线性激活函数之前，即

$$\mathbf{a}^{(l)} = f(\text{BN}_{\gamma,\beta}(\mathbf{z}^{(l)})) = f\left(\text{BN}_{\gamma,\beta}(W\mathbf{a}^{(l-1)})\right), \tag{7.23}$$

其中因为批量归一化本身具有平移变换，因此仿射变换 $W\mathbf{a}^{(l-1)}$ 不再需要偏置参数。

这里要注意的是，每次小批量样本的 $\mu_{\mathcal{B}}$ 和方差 $\sigma_{\mathcal{B}}^2$ 是净输入 $\mathbf{z}^{(l)}$ 的函数，而不是常量。因此在计算参数梯度时需要考虑 $\mu_{\mathcal{B}}$ 和 $\sigma_{\mathcal{B}}^2$ 的影响。当训练完成时，用整个数据集上的均值 μ 和方差 σ 来分别代替每次小批量样本的 $\mu_{\mathcal{B}}$ 和方差 $\sigma_{\mathcal{B}}^2$。在实践中，$\mu_{\mathcal{B}}$ 和 $\sigma_{\mathcal{B}}^2$ 也可以用移动平均来计算。

层归一化

批量归一化是对一个中间层的单个神经元进行归一化操作，因此要求小批量样本的数量不能太小，否则难以计算单个神经元的统计信息。此外，如果一个神经元的净输入的分布在神经网络中是动态变化的，比如循环神经网络，那么就无法应用批量归一化操作。

参见习题（7-1），第157页。

层归一化（layer normalization）[Ba et al., 2016] 是和批量归一化非常类似的方法。和批量归一化不同的是，层归一化是对一个中间层的所有神经元进行归一化。

假设一个深层神经网络中，第 l 层神经的净输入为 $\{\mathbf{z}^{(l)}$，其均值和方差为

$$\mu^{(l)} = \frac{1}{n^l}\sum_{i=1}^{n^l} z_i^{(l)}, \tag{7.24}$$

$$\sigma^{(l)^2} = \frac{1}{n^l}\sum_{k=1}^{n^l}(z_i^{(l)} - \mu^{(l)})^2, \tag{7.25}$$

其中 n^l 为第 l 层神经元的数量。

层归一化定义为

$$\hat{\mathbf{z}}^{(l)} = \frac{\mathbf{z}^{(l)} - \mu^{(l)}}{\sqrt{\sigma^{(l)^2} + \epsilon}} \odot \gamma + \beta \tag{7.26}$$

$$\equiv \mathrm{LN}_{\gamma,\beta}(\mathbf{z}^{(l)}), \tag{7.27}$$

其中 γ 和 β 分别代表缩放和平移的参数向量，和 $\mathbf{z}^{(l)}$ 维数相同。

层归一化的循环神经网络 层归一化可以应用在循环神经网络中，对循环神经层进行归一化操作。假设在时刻 t，循环神经网络的隐藏层为 $\mathbf{h}_t$，其层归一化的更新为

参见公式(6.4),第110页。

$$\mathbf{z}_t = U\mathbf{h}_{t-1} + W\mathbf{x}_t, \tag{7.28}$$

$$\mathbf{h}_t = f(\mathrm{LN}_{\gamma,\beta}(\mathbf{z}_t)), \tag{7.29}$$

其中输入为 $\mathbf{x}_t$ 为第 t 时刻的输入，U, W 为网络参数。

在标准循环神经网络中，循环神经层的净输入一般会随着时间慢慢变大或变小，从而导致梯度爆炸或消失。而层归一化的循环神经网络可以有效地缓解这种状况。

层归一化和批量归一化整体上是十分类似的，差别在于归一化的方法不同。对于 K 个样本的一个小批量集合 $Z^{(l)} = [\mathbf{z}^{(1,l)}; \cdots ; \mathbf{z}^{(K,l)}]$，层归一化是对矩阵 $Z^{(l)}$ 对每一列进行归一化，而批量归一化是对每一行进行归一化。一般而言，批量归一化是一种更好的选择。当小批量样本数量比较小时，可以选择层归一化。

其它归一化方法

除了上述两种归一化方法外，也有一起其它的归一化方法。

权重归一化 *权重归一化*（weight normalization）[Salimans and Kingma, 2016] 是对神经网络的连接权重进行归一化，通过*再参数化*（reparameterization）方法，将连接权重分解为长度和方向两种参数。假设第 l 层神经元 $\mathbf{a}^{(l)} = f(W\mathbf{a}^{(l-1)} + \mathbf{b})$，我们将 W 再参数化为

$$W_{i,:} = \frac{g_i}{\|\mathbf{v}_i\|}\mathbf{v}_i, \qquad 1 \leq i \leq n^l \tag{7.30}$$

其中 $W_{i,:}$ 表示权重 W 的第 i 行，n^l 为神经元数量。新引入的参数 g_i 为标量，$\mathbf{v}_i$ 和 $\mathbf{a}^{(l-1)}$ 维数相同。

由于在神经网络中权重经常是共享的，权重数量往往比神经元数量要少，因此权重归一化的开销会比较小。

局部相应归一化 *局部响应归一化*（local response normalization）[Krizhevsky et al., 2012] 是一种受生物学启发的归一化方法，通常用在基于卷积的图像处理上。

假设一个卷积层的输出特征映射 $\mathbf{Y} \in \mathbb{R}^{M' \times N' \times P}$ 为三维张量，其中每个切片矩阵 $Y^p \in \mathbb{R}^{M' \times N'}$ 为一个输出特征映射，$1 \leq p \leq P$。

参见公式(5.22)，第95页。

局部响应归一化是对邻近的特征映射进行局部归一化。

$$\hat{Y}^p = Y^p / \left(k + \alpha \sum_{j=\max(1,p-\frac{n}{2})}^{\min(P,p+\frac{n}{2})} (Y^j)^2 \right)^{\beta} \tag{7.31}$$

$$\equiv \mathrm{LRN}_{n,k,\alpha,\beta}(Y^p), \tag{7.32}$$

其中除和幂运算都是按元素运算，n, k, α, β 为超参，n 为局部归一化的特征窗口大小。在 AlexNet 中，这些超参的取值为 $n = 5, k = 2, \alpha = 10\mathrm{e}^{-4}, \beta = 0.75$。

局部响应归一化和层归一化都是对同层的神经元进行归一化。不同的是局部响应归一化应用在激活函数之后，只是对邻近的神经元进行局部归一化，并且不减去均值。

邻近的神经元指对应同样位置的邻近特征映射

局部响应归一化和生物神经元中的侧抑制（lateral inhibition）现象比较类似，即活跃神经元对相邻神经元具有抑制作用。当使用ReLU作为激活函数时，神经元的活性值是没有限制的，局部响应归一化可以起到平衡和约束左右。如果一个神经元的活性值非常大，那么和它邻近的神经元就近似地归一化为0，从而起到抑制作用，增强模型的泛化能力。最大汇聚也具有侧抑制作用。但最大汇聚是对同一个特征映射中的邻近位置中的神经元进行抑制，而局部响应归一化是对同一个位置的邻近特征映射中的神经元进行抑制。

上述的归一化方法可以根据需要应用在神经网络的中间层，从而减少前面网络参数更新对后面网络输入带来的内部协变量偏移问题，提高深层神经网络的训练效率。同时，归一化方法也可以作为一种有效的正则化方法，从而提高网络的泛化能力，避免过拟合。

7.1.4 梯度下降方法的改进

由于深度学习经常用来处理比较大规模的数据，参数也非常多，因此如果每次迭代都计算整个数据集上的梯度需要计算资源比较多。并且在大规模的数据集中，数据也非常冗余，也没有必要在整个数据集上批量计算梯度。因此，在训练深度模型时，经常使用小批量梯度下降算法。图7.5给出了小批量梯度下降中，每次选取样本数量对损失下降的影响。

假设 $f(\mathbf{x}^{(i)}, \theta)$ 代表神经网络，θ 为网络参数，以小批量梯度下降为例，在第 t 次迭代（epoch）时，选取 m 个训练样本 $\mathcal{I}_t = \{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^{m}$，首先计算梯度 $\mathbf{g}_t$

$$\mathbf{g}_t = \frac{1}{m} \sum_{i \in I_t} \frac{\partial \mathcal{L}\big(\mathbf{y}^{(i)}, f(\mathbf{x}^{(i)}, \theta)\big)}{\partial \theta} + \lambda \|\theta\|^2, \tag{7.33}$$

其中，$\mathcal{L}(\cdot, \cdot)$ 为可微分的损失函数，λ 为正则化系数。

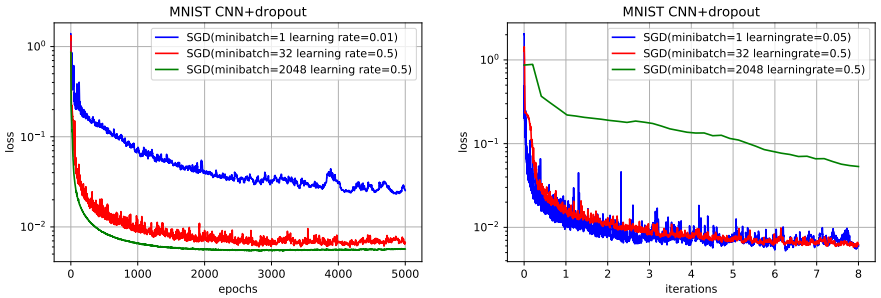
然后使用梯度下降来更新参数，

$$\theta_t \leftarrow \theta_{t-1} - \alpha \mathbf{g}_t, \tag{7.34}$$

其中 $\alpha > 0$ 为学习率。

我们定义每次迭代时参数更新的差值 $\Delta\theta_t$ 为

$$\Delta\theta_t = \theta_t - \theta_{t-1}. \tag{7.35}$$



(a) 按每次小批量更新的损失变化 (b) 按整个数据集迭代的损失变化

图 7.5: 小批量梯度下降中，每次选取样本数量对损失下降的影响。从 (a) 可以看出，每次迭代选取的批量样本数越多，下降效果越明显，并且取现越平滑。当每次选取一个样本时（相当于随机梯度下降），损失整体是下降趋势，但局部看会来回震荡。从 (b) 可以看出，如果按整个数据集迭代的来看损失变化情况，则小批量样本数越小，下降效果越明显。

$\Delta\theta_t$ 为每次迭代时参数的实际更新差值，即 $\theta_t = \theta_{t-1} + \Delta\theta_t$。$\Delta\theta_t$ 和梯度 $\mathbf{g}_t$ 并不需要完全一致。在标准的小批量梯度下降中，$\Delta\theta_t = -\alpha\mathbf{g}_t$。

为了更有效地进行训练深度神经网络，在标准的小批量梯度下降方法的基础上，也经常使用一些改进方法以加快优化速度。常见的改进方法主要从以下两个方面进行改进：学习率衰减和动量法。

这些改进的优化方法也同样可以应用在批量或随机梯度下降方法上。

学习率衰减

在梯度下降中，学习率 α 的取值非常关键，如果过大就不会收敛，如果过小则收敛速度太慢。从经验上看，学习率在一开始要保持大些来保证收敛速度，在收敛到最优点附近时要小些以避免来回震荡。因此，比较简单直接的学习率调整可以通过学习率衰减（learning rate decay）的方式来实现。

假设初始化学习率为 α_0，在第 t 次迭代时的学习率 α_t。常用的衰减方式为可以设置为按迭代次数进行衰减。比如反时衰减（inverse time decay）

$$\alpha_t = \alpha_0 \frac{1}{1 + \beta \times t}, \tag{7.36}$$

或指数衰减（exponential decay）

$$\alpha_t = \alpha_0 \beta^t, \tag{7.37}$$

或自然指数衰减（natural exponential decay）

$$\alpha_t = \alpha_0 \exp(-\beta \times t), \tag{7.38}$$

其中β为衰减率，一般取值为0.96。

除了这些固定衰减率的调整学习率方法外，还有些自适应地调整学习率的方法，比如AdaGrad、RMSprop、AdaDelta等。这些方法都对每个参数设置不同的学习率。

AdaGrad 在标准的梯度下降方法中，每个参数在每次迭代时都使用相同的学习率。由于每个参数的维度上收敛速度都不相同，因此根据不同参数的收敛情况分别设置学习率。

AdaGrad（Adaptive Gradient）算法[Duchi et al., 2011]是借鉴L2正则化的思想，每次迭代时自适应地调整每个参数的学习率。在第t迭代时，先计算每个参数梯度平方的累计值

$$G_t = \sum_{\tau=1}^{t} \mathbf{g}_\tau \odot \mathbf{g}_\tau, \tag{7.39}$$

其中$\odot$为按元素乘积，$\mathbf{g}_\tau \in \mathbb{R}^{|\theta|}$是第$\tau$次迭代时的梯度。

AdaGrad算法的参数更新差值为

$$\Delta\theta_t = -\frac{\alpha}{\sqrt{G_t + \epsilon}} \odot \mathbf{g}_t, \tag{7.40}$$

其中α是初始的学习率，ϵ是为了保持数值稳定性而设置的非常小的常数，一般取值e^{-7}到e^{-10}。此外，这里的开平方、除、加运算都是按元素进行的操作。

在Adagrad算法中，如果某个参数的偏导数累积比较大，其学习率相对较小；相反，如果其偏导数累积较小，其学习率相对较大。但整体是随着迭代次数的增加，学习率逐渐缩小。

Adagrad算法的缺点是在经过一定次数的迭代依然没有找到最优点时，由于这时的学习率已经非常小，很难再继续找到最优点。

RMSprop *RMSprop*算法是Geoff Hinton提出的一种自适应学习率的方法，可以在有些情况下避免AdaGrad算法中学习率不断单调下降以至于过早衰减的缺点。

RMSprop 算法首先计算每次迭代梯度 $\mathbf{g}_t$ 平方的指数衰减移动平均，

$$G_t = \beta G_{t-1} + (1-\beta)\mathbf{g}_t \odot \mathbf{g}_t, \tag{7.41}$$

其中 β 为衰减率，一般取值为 0.9。

RMSprop 算法的参数更新差值为

$$\Delta\theta_t = -\frac{\alpha}{\sqrt{G_t + \epsilon}} \odot \mathbf{g}_t, \tag{7.42}$$

其中 α 是初始的学习率，比如 0.001。

从上式可以看出，RMSProp 算法和 Adagrad 算法的区别在于 G_t 的计算有累积方式变成了指数衰减移动平均。在迭代过程中，每个参数的学习率并不是呈衰减趋势，既可以变小也可以变大。

AdaDelta *AdaDelta*算法 [Zeiler, 2012] 也是 Adagrad 算法的一个改进。和 RMSprop 算法类似，AdaDelta 算法通过梯度平方的指数衰减移动平均来调整学习率。此外，AdaDelta 算法还引入了每个平方的指数衰减权移动平均。

第 t 次迭代时，每次参数更新差 $\Delta\theta_\tau, 1 \le \tau \le t-1$ 的指数衰减权移动平均为

此时 $\Delta\theta_t$ 未未知，因此只能计算到 ΔX_{t-1}。

$$\Delta X_{t-1}^2 = \beta \Delta X_{t-2}^2 + (1-\beta)\Delta\theta_{t-1} \odot \Delta\theta_{t-1}. \tag{7.43}$$

其中 β 为衰减率。

AdaDelta 算法的参数更新差值为

$$\Delta\theta_t = -\frac{\sqrt{\Delta X_{t-1}^2 + \epsilon}}{\sqrt{G_t + \epsilon}}\mathbf{g}_t \tag{7.44}$$

其中 G_t 的计算方式和 RMSprop 算法一样（公式 (7.41)），ΔX_{t-1}^2 为参数更新差 $\Delta\theta$ 的指数衰减权移动平均。

从上式可以看出，AdaDelta 算法将 RMSprop 算法中的初始学习率 α 改为动态计算的 $\sqrt{\Delta X_{t-1}^2}$，在一定程度上平抑了学习率的波动。

动量法

除了调整学习率之外，还可以通过使用最近一段时间内的平均梯度来代替当前时刻的梯度来作为参数更新的方向。从图7.5看出，在小批量梯度下降中，

如果每次选取样本数量比较小，损失会呈现震荡的方式下降。有效地缓解梯度下降中的震荡的方式是通过用梯度的移动平均来代替每次的实际梯度，并提高优化速度。这就是*动量法*。

*动量*是模拟物理中的概念。一般而言，一个物体的动量指的是这个物体在它运动方向上保持运动的趋势，是物体的质量和速度的乘积。*动量法*（Momentum Method）[Rumelhart et al., 1988] 是用之前积累动量来替代真正的梯度。每次迭代的梯度可以看作是加速度。

实际上是相当于对 $-\frac{\alpha}{1-\rho}\mathbf{g}_t$ 做指数衰减移动平均。

在第 t 次迭代时，计算*负梯度*的“加权移动平均”作为参数的更新方向，

$$\Delta\theta_t = \rho\Delta\theta_{t-1} - \alpha\mathbf{g}_t, \tag{7.45}$$

其中 ρ 为动量因子，通常设为 0.9；α 为学习率。

这样每个参数的实际更新差值取决于最近一段时间内梯度的加权平均值。当某个参数在最近一段时间内的梯度方向不一致时，其真实的参数更新幅度变小；相反，当在最近一段时间内的梯度方向都一致时，其真实的参数更新幅度变大，起到加速作用。一般而言，在迭代初期，梯度方法都比较一致，动量法会起到加速作用，可以更快地到达最优点。在迭代后期，梯度方法会取决不一致，在收敛值附近震荡，动量法会起到减速作用，增加稳定性。从某种角度来说，当前梯度叠加上部分的上次梯度，一定程度上可以近似看作二阶梯度。

AdaM *Adam*（Adaptive Moment Estimation）算法 [Kingma and Ba, 2015] 可以看作是动量法和 RMSprop 的结合，不但使用动量作为参数更新方向，而且可以自适应调整学习率。

Adam 算法一方面计算梯度平方 $\mathbf{g}_t^2$ 的指数加权平均（和 RMSprop 类似），另一方面计算梯度 $\mathbf{g}_t$ 的指数加权平均（和动量法类似）。

$$M_t = \beta_1 M_{t-1} + (1-\beta_1)\mathbf{g}_t, \tag{7.46}$$

$$G_t = \beta_2 G_{t-1} + (1-\beta_2)\mathbf{g}_t \odot \mathbf{g}_t, \tag{7.47}$$

其中 β_1 和 β_2 分别为两个移动平均的衰减率，通常取值为 $\beta_1 = 0.9, \beta_2 = 0.99$。

M_t 可以看作是梯度的均值（一阶矩），G_t 可以看作是梯度的未减去均值的方差（二阶矩）。

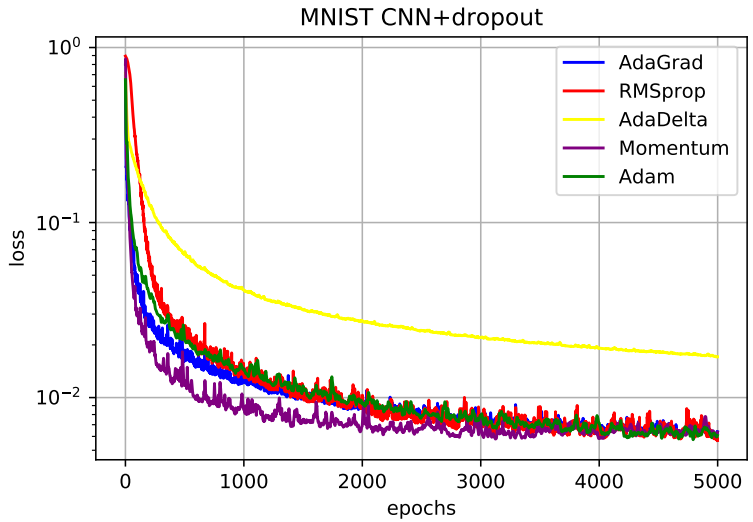


图 7.6: 不同优化方法的比较。

假设 $M_0 = 0, G_0 = 0$，那么在迭代初期 M_t 和 G_t 的值会比真实的均值和方差要小。特别是当 β_1 和 β_2 都接近于 1 时，偏差会很大。因此，需要对偏差进行修正。

$$\hat{M}_t = \frac{M_t}{1-\beta_1^t}, \tag{7.48}$$

$$\hat{G}_t = \frac{G_t}{1-\beta_2^t}. \tag{7.49}$$

Adam 算法的参数更新差值为

$$\Delta\theta_t = -\frac{\alpha}{\sqrt{G_t+\epsilon}}\hat{M}_t, \tag{7.50}$$

其中学习率 α 通常设为 0.001，并且也可以进行衰减，比如 $\alpha_t = \frac{\alpha_0}{\sqrt{t}}$。

图7.6给出了不同优化方法的比较。

梯度截断

在深层神经网络或循环神经网络中，梯度消失或爆炸是影响学习效率的主要因素。除了上面介绍优化算法之外，还可以使用一种比较简单的启发式方法：*梯度截断*（gradient clipping）[Pascanu et al., 2013]。

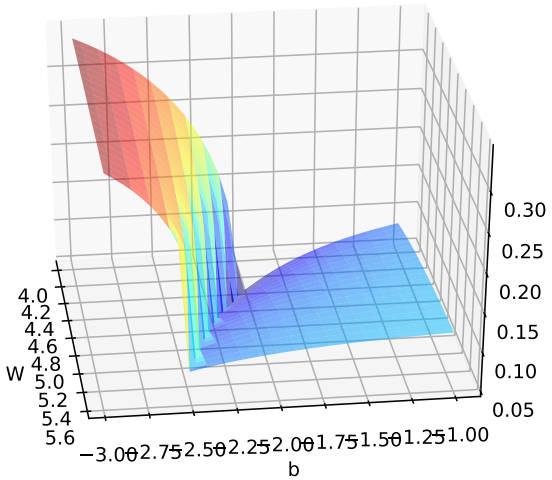


图 7.7: 梯度爆炸问题示例。图中的曲面为只有一个隐藏神经元的循环神经网络 $h_t = \sigma(wh_{t1} + b)$ 的损失函数，其中 w 和 b 为参数。假如 h_0 初始值为 0.3，损失函数为 $\mathcal{L} = (h_{100} - 0.65)^2$。

梯度截断是限定一个梯度的模限定一个区间，当梯度的模小于或大于这个区间时就进行截断。一般截断的方式有以下几种：

按值截断 在第 t 次迭代时，梯度为 $\mathbf{g}_t$，给定一个区间 $[a, b]$，如果一个参数的梯度小于 a 时，就将其设为 a；如果小于 b 时，就将其设为 b。

$$\mathbf{g}_t = \max(\min(\mathbf{g}_t, b), a). \tag{7.51}$$

按模截断 按模截断是将梯度的模截断到一个给定的截断阈值 b。

如果 $\|\mathbf{g}_t\|^2 \leq b$，保持 $\mathbf{g}_t$ 不变。如果 $\|\mathbf{g}_t\|^2 > b$，令

$$\mathbf{g}_t = \frac{b}{\|\mathbf{g}_t\|}\mathbf{g}_t. \tag{7.52}$$

截断阈值 b 是一个超参数，也可以根据一段时间内的平均梯度来自动调整。Pascanu et al. [2013] 在实验中发现训练过程对阈值 a 并不十分敏感，通常一个小的

阈值就可以得到很好的结果。

在训练循环神经网络时，按模截断是避免梯度爆炸问题的有效方法。图7.7给出了一个循环神经网络的损失函数关于参数的曲面。在使用梯度下降方法来进行参数学习的过程中，有时梯度会突然增大，如果用大的梯度进行更新参数，反而会导致其远离最优点。为了避免这种情况，当梯度的模大于一定阈值时，对梯度进行截断。

7.1.5 超参数优化

在神经网络中，除了可学习的参数之外，还存在很多超参数。这些超参数对网络性能的影响也很大。不同的机器学习任务需要往往需要不同的超参数。常见的超参数有

- 网络结构，包括神经元之间的连接关系、层数、每层的神经元数量、激活函数的类型等。
- 优化参数，包括优化方法、学习率、小批量的样本数量等
- 正则化系数

超参数优化（hyperparameter optimization）主要存在两方面的困难。（1）超参数优化是一个组合优化问题，无法像一般参数那样通过梯度下降方法来优化，也没有一种通用有效的优化方法。（2）评估一组超参数配置（configuration）的时间代价非常高，从而导致一些优化方法（比如演化算法（evolution algorithm））在超参数优化中难以应用。

假设一个神经网络中总共有 K 个超参数，每个超参数配置表示为一个向量 $\mathbf{x} \in \mathcal{X}$，$\mathcal{X} \subset \mathbb{R}^K$ 是超参数配置的取值空间。超参数优化的目标函数定义为 $f(\mathbf{x}) : \mathcal{X} \to \mathbb{R}$，$f(\mathbf{x})$ 是衡量一组超参数配置 $\mathbf{x}$ 效果的函数，一般设置为开发集上的错误率。目标函数 $f(\mathbf{x})$ 可以看作是一个黑盒（block-box）函数，不需要知道其具体形式。

虽然在神经网络的超参数优化中，$f(\mathbf{x})$ 的函数形式虽然已知，但 $f(\mathbf{x})$ 不是关于 $\mathbf{x}$ 的连续函数，并且 $\mathbf{x}$ 不同，$f(\mathbf{x})$ 的函数形式也不同，因此关无法使用梯度下降等优化方法。

对于超参数的设置，比较简单的方法有人工搜索、网格搜索和随机搜索。

网格搜索 网格搜索（grid search）是一种通过尝试所有超参数的组合来寻址合适一组超参数配置的方法。假设总共有 K 个超参数，第 k 个超参数的可以取 m_k 个值。那么总共的配置组合数量为 $m_1 \times m_2 \times \cdots \times m_K$。如果超参数是连续的，可以将超参数离散化，选择几个“经验”值。比如学习率 α，我们可以设置

$$\alpha \in \{0.01, 0.1, 0.5, 1.0\}.$$

一般而言，对于连续的超参数，我们不能按等间隔的方式进行离散化，需要根据超参数自身的特点进行离散化。

网格搜索根据这些超参数的不同组合分别训练一个模型，然后测试这些模型在开发集上的性能，选取一组性能最好的配置。

随机搜索 如果不同超参数对模型性能的影响有很大差异。有些超参数（比如正则化系数）对模型性能的影响有限，而有些超参数（比如学习率）对模型性能影响比较大。在这种情况下，采用网格搜索会在不重要的超参数上进行不必要的尝试。一种在实践中比较有效的改进方法是对超参数进行随机组合，然后选取一个性能最好的配置，这就是*随机搜索*（random search）[Bergstra and Bengio, 2012]。随机搜索在实践中更容易实现，一般会比网格搜索更加有效。

网格搜索和随机搜索都没有利用不同超参数组合之间的相关性，即如果模型的超参数组合比较类似，其模型性能也是比较接近的。因此这两种搜索方式一般都比较低效。下面我们介绍两种自适应的超参数优化方法：贝叶斯优化和动态资源分配。

贝叶斯优化 *贝叶斯优化*（Bayesian optimization）[Bergstra et al., 2011, Snoek et al., 2012] 是一种自适应的超参数搜索方法，根据当前已经试验的超参数组合，来预测下一个可能带来最大收益的组合。一种比较比较常用的贝叶斯优化方法为*时序模型优化*（sequential model-based optimization，SMBO）[Hutter et al., 2011]。假设超参数优化的函数 $f(\mathbf{x})$ 服从高斯过程，则 $p(f(\mathbf{x})|\mathbf{x})$ 为一个正态分布。贝叶斯优化过程是根据已有的 N 组试验结果 $\mathcal{H}=\{\mathbf{x}_n, y_n\}_{n=1}^N$（$y_n$ 为 $f(\mathbf{x}_n)$ 的观测值）来建模高斯过程，并计算 $f(\mathbf{x})$ 的后验分布 $p_{\mathcal{GP}}(f(\mathbf{x})|\mathbf{x},\mathcal{H})$。

高斯过程参见第D.3.2节，第336页。

为了使得 $p_{\mathcal{GP}}(f(\mathbf{x})|\mathbf{x},\mathcal{H})$ 接近其真实分布，就需要对样本空间进行足够多的采样。但是超参数优化中每一个样本的生成成本很高，需要用尽可能少的样本来使得 $p_\theta(f(\mathbf{x})|\mathbf{x},\mathcal{H})$ 接近于真实分布。因此，需要通过定义一个*收益函数*（acquisition function）$a(x,\mathcal{H})$ 来判断一个样本是否能够给建模 $p_\theta(f(\mathbf{x})|\mathbf{x},\mathcal{H})$ 提供更多的收益。收益越大，其修正的高斯过程会越接近目标函数的真实分布。时序模型优化的过程如算法7.1所示。

收益函数的定义有很多种方式，一个常用的是*期望改善*（expected improvement，EI）函数。假设 $y^*=\min\{y_n, 1\leq n\leq N\}$ 是当前已有样本中的最优值，期望改善函数为，

$$\mathbf{EI}(\mathbf{x},\mathcal{H})=\int_{-\infty}^{\infty}\max(y^*-y,0)p_{\mathcal{GP}}(y|\mathbf{x},\mathcal{H})dy. \tag{7.53}$$

算法 7.1: 时序模型优化，一种贝叶斯优化方法

输入: 优化目标函数 $f(\mathbf{x})$，迭代次数：T，收益函数 $a(x, \mathcal{H})$

$\mathcal{H} \leftarrow \emptyset$;

随机初始化高斯过程，并计算 $p_{\mathcal{GP}}(f(\mathbf{x})|\mathbf{x}, \mathcal{H})$;

for $t \leftarrow 1$ **to** T **do**

  $\mathbf{x}' \leftarrow \arg\max_x a(x, \mathcal{H})$;

  评价 $y' = f(\mathbf{x}')$; // 代价高

  $\mathcal{H} \leftarrow \mathcal{H} \cup (\mathbf{x}', y')$;

  根据 $\mathcal{H}$ 重新建模高斯过程，并计算 $p_{\mathcal{GP}}(f(\mathbf{x})|\mathbf{x}, \mathcal{H})$;

end

输出: $\mathcal{H}$

期望改善是定义一个样本 $\mathbf{x}$ 在当前模型 $p_{\mathcal{GP}}(f(\mathbf{x})|\mathbf{x}, \mathcal{H})$ 下，$f(\mathbf{x})$ 超过最好结果 y^* 的期望。除了期望改善函数之外，收益函数还有其它定义形式，比如改善概率（probability of improvement）、高斯过程置信上界（GP upper confidence bound，GP-UCB）等。

贝叶斯优化的一个缺点是高斯过程建模需要计算协方差矩阵的逆，时间复杂度是 $O(n^3)$，因此不能很好地处理高维情况。深度神经网络的超参数一般比较多，为了使用贝叶斯优化来搜索神经网络的超参数，需要一些更高效的高斯过程建模。也有一些方法可以将时间复杂度降从 $O(n^3)$ 降低到 $O(n)$[Snoek et al., 2015]。

动态资源分配 在超参数优化中，每组超参数配置的评估代价比较高。如果我们可以在较早的阶段就可以估计出一组配置的效果会比较差，那么我们就可以中止这组配置的评估，将更多的资源留给其它配置。这个问题可以归结为多臂赌博机问题的一个泛化问题：*最优臂问题*（best-arm problem），即在给定有限的机会次数下，如何玩这些赌博机并找到收益最大的臂。和多臂赌博机类似，最优臂问题也是在利用和探索之间找到最佳的平衡。

多臂赌博机问题参见第14.1.1节，第246页。

由于目前神经网络的优化方法一般都采取随机梯度下降，因此我们可以通过一组超参数的学习曲线来预估这组超参数配置是否有希望得到比较好的结果。如果一组超参数配置的学习曲线不收敛或者收敛比较差，我们可以应用*早期停止*（early-stopping）策略来中止当前的训练。

动态资源分配的一种有效方法是*逐次减半*（successive halving）方法 [Jamieson

算法 7.2: 一种逐次减半的动态资源分配方法

输入: 预算 B，N 个超参数配置 $\{\mathbf{x}_n\}_{n=1}^N$

$T \leftarrow \lceil \log_2(N) \rceil - 1$;

随机初始化 $\mathcal{S}_0 = \{\mathbf{x}_n\}_{n=1}^N$;

for $t \leftarrow 1$ **to** T **do**

  $r_t \leftarrow \lfloor \frac{B}{|\mathcal{S}_t| \times T} \rfloor$;

  给 S_t 中的每组配置分配 r_t 的资源;

  运行 S_t 所有配置，评估结果为 $\mathbf{y}_t$;

  根据评估结果，选取 $|S_t|/2$ 组最优的配置

  $\mathcal{S}_t \leftarrow \arg\max(\mathcal{S}_t, \mathbf{y}_t, |\mathcal{S}_t|/2)$; // $\arg\max(\mathcal{S}, \mathbf{y}, m)$ 为从集合 $\mathcal{S}$ 中选取 m 个元素，对应最优的 m 个评估结果。

end

输出: 最优配置 $\mathcal{S}_K$

and Talwalkar, 2016]，将超参数优化看作是一种非随机的最优臂问题。假设要尝试 N 组超参数配置，总共可利用的资源预算（摇臂的次数）为 B，我们可以通过 $T = \lceil \log_2(N) \rceil - 1$ 轮逐次减半的方法来选取最优的配置，具体过程如算法7.2所示。

在逐次减半方法中，尝试的超参数配置数量 N 十分关键。如果 N 越大，得到最佳配置的机会也越大，但每组配置分到的资源就越少，这样早期的评估结果可能不准确。反之如果 N 越小，每组超参数配置的评估会越准确，但有可能无法得到最优的配置。因此，如何设置 N 是平衡“利用-探索”的一个关键因素。一种改进的方法是 HyperBand 方法 [Li et al., 2017]，通过尝试不同的 N 来选取最优参数。

神经架构搜索 上面介绍的超参数优化方法都是在固定（或变化比较小）的超参数空间 $\mathcal{X}$ 中进行最优配置搜索，而最重要的神经网络架构一般还是需要由有经验的专家来进行设计。*神经架构搜索*（neural architecture search，NAS）[Zoph and Le, 2017] 是一个新的比较有前景的研究方向，通过神经网络来自动实现网络架构的设计。一个神经网络的架构可以用一个变长的字符串来描述。利用元学习的思想，神经架构搜索利用一个*控制器*来生成另一个*子网络*的架构描述。控制器可以由一个循环神经网络来实现。控制器的训练可以通过*强化学习*来完成，其奖励信号为生成的子网络在开发集上的准确率。

深度学习使得机器学习中的“特征工程”问题转变为“网络架构工程”问题。

强化学习参见第14.1节，第246页。

7.2 网络正则化

机器学习模型的关键是泛化问题，即在样本真实分布上的期望风险最小化。对于同样，最小化神经网络模型在训练数据集上的经验风险并不是唯一目标。由于神经网络的拟合能力非常强，其在训练数据上的错误往往都可以降到非常低（比如错误率为0），因此如果提高神经网络的泛化能力反而成为影响模型能力的最关键因素。

在传统的机器学习中，提高泛化能力的方法主要是限制模型复杂度，比如采用权重衰减等方式。而在训练深度神经网络时，特别是在过度参数（over-parameterized）时，权重衰减的效果往往不如浅层机器学习模型中显著。

过度参数是指模型参数的数量远远大于训练数据的数量。

因此训练深度学习模型时，往往还会使用其它的正则化方法，比如数据增强、早期停止、丢弃法、集成法等。

7.2.1 权重递减

深度神经网络很容易产生过拟合现象，因为增加的抽象层使得模型能够对训练数据中较为罕见的依赖关系进行建模。对此，权重递减（ℓ_2 正规化）或者稀疏（ℓ_1 -正规化）等方法可以利用在训练过程中以减小过拟合现象 [Bengio et al., 2013]。

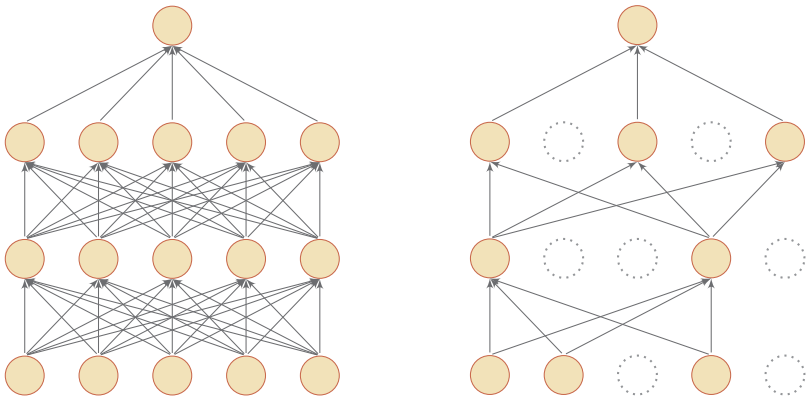
7.2.2 数据增强

深层神经网络一般都需要大量的训练数据才能获得比较理想的结果。在数据量有限的情况下，可以通过数据增强（Data Augmentation）来增加数据量，提高模型鲁棒性，避免过拟合。目前，数据增强还主要应用在图像数据上，在文本等其它类型的数据还没有太好的方法。

图像数据的增强主要是通过算法对图像进行转变，引入噪声等方法来增加数据的多样性。增强的方法主要有几种：

- 旋转（Rotation）：将图像按顺时针或逆时针方向随机旋转一定角度；
- 翻转（Flip）：将图像沿水平或垂直方法随机翻转一定角度；
- 缩放（Zoom In/Out）：将图像放大或缩小一定比例；
- 平移（Shift）：将图像沿水平或垂直方法平移一定步长；
- 加噪声（Noise）：加入随机噪声。

7.2.3 Dropout



(a) 标准网络 (b) Dropout 神经网络

图 7.8: Dropout 神经网络模型

训练一个大规模的神经网络经常容易过拟合。过拟合在很多机器学习中都会出现。一般解决过拟合的方法有正则化、早期终止、集成学习以及使用验证集等。Srivastava et al. [2014] 提出了适用于神经网络的避免过拟合的方法，叫**dropout**方法（丢弃法），即在训练中随机丢弃一部分神经元（同时丢弃其对应的连接边）来避免过拟合。图7.8给出了 dropout 网络的示例。

每做一次 dropout，相当于从原始的网络中采样得到一个更瘦的网络。如果一个神经网络有 n 个神经元，那么总共可以采样出 2^n 个子网络。每次迭代都相当于训练一个不同的子网络，这些子网络都共享原始网络的参数。那么，最终的网络可以近似看作是集成了指数级个不同网络的组合模型。每次选择丢弃的神经元是随机的。最简单的方法是设置一个固定的概率 p。对每一个神经元都一个概率 p 来判定要不要保留。p 可以通过验证集来选取一个最优的值。或者，p 也可以设为0.5，这对大部分的网络和任务有比较有效。

一般情况下 dropout 是针对神经元进行随机丢弃，但是也可以扩展到对每条边进行随机丢弃，或每一层进行随机丢弃。

当 $p = 0.5$ 时，在训练时有一半的神经元被丢弃，只剩余一半的神经元是可以激活的。而在测试时，所有的神经元都是可以激活的。因此每个神经元训练时的净输入值平均比测试时小一半左右。这会造成训练和测试时网络的输出不

一致。为了缓解这个问题，在测试时需要将每一个神经元的输出都折半，也相当于把不同的神经网络做了平均。

一般来讲，对于隐藏层的神经元，其 dropout 率等于 0.5 时效果最好，因为此时通过 dropout 方法，随机生成的网络结构最具多样性。对于输入层的神经元，其 dropout 率通常设为更接近 1 的数，使得输入变化不会太大。当对输入层神经元进行 dropout 时，相当于给数据增加噪声，以此来提高网络的鲁棒性。

bayesian view

7.3 模型压缩

待写。。。

习题 7-1 分析为什么批量归一化不能循环神经网络。

参考文献

Lei Jimmy Ba, Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016. URL http://arxiv.org/abs/1607.06450.

Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8624–8628. IEEE, 2013.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941, 2014.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial in-*

telligence and statistics, pages 249–256, 2010.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 448–456, 2015.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*, pages 240–248, 2016.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of International Conference on Learning Representations*, 2015.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. In *Proceedings of 5th International Conference on Learning Representations*, 2017.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the International Conference on Machine Learning*, pages 1310–1318, 2013.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5:3, 1988.

Tim Salimans and Diederik P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–909, 2016.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable bayesian optimization using deep neural networks. In *International Conference on Machine Learning*, pages 2171–2180, 2015.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Matthew D Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. In *Proceedings of 5th International Conference on Learning Representations*, 2017.