Deep Learning Cheat Sheet

Hailiang Zhao*
College of Computer Science and Technology, Zhejiang University
hliangzhao@zju.edu.cn

October 18, 2020

Contents

*Hailiang is a second-year Ph.D. student of ZJU-CS. His homepage is http://hliangzhao.me.

1 SoftMax 回归

1.1 模型定义

如无特别说明，向量均指列向量。

Softmax 回归是 logistic 回归（适用于 2 类分类）扩展到多类分类的结果。设标签 $c \in \{1, ..., C\}$，对于样本 $(\boldsymbol{x}, y)$，softmax 回归预测样本标签为 c 的概率为

$$p(y = c|\boldsymbol{x}) = \text{softmax}(\boldsymbol{w}_c^\top \boldsymbol{x}) = \frac{\exp(\boldsymbol{w}_c^\top \boldsymbol{x})}{\sum_{c'=1}^{C} \exp(\boldsymbol{w}_{c'}^\top \boldsymbol{x})},$$

因此 softmax 回归的预测结果为

$$\hat{y} = \underset{c=1}{\overset{C}{\text{argmax}}}\, p(y = c|\boldsymbol{x}) = \underset{c=1}{\overset{C}{\text{argmax}}}\, \boldsymbol{w}_c^\top \boldsymbol{x}.$$

本质上，softmax 回归是一个单层神经网络，输出层为 softmax 层。

1.2 单样本的矢量计算表达式

为了更方便地定义 torch tensor，假设所有向量均为**行向量**。

设 d 为样本特征个数且 $\boldsymbol{x} \in \mathbb{R}^{1\times d}$，$W \in \mathbb{R}^{d\times C}$ 为待学习的权重，$\boldsymbol{b} \in \mathbb{R}^{1\times C}$ 为偏置，则对于样本 $(\boldsymbol{x}^{(i)}, y^{(i)})$，softmax 回归的矢量计算表达式为

$$\hat{\boldsymbol{y}}^{(i)} = \text{softmax}(\boldsymbol{x}^{(i)} W + \boldsymbol{b}),$$

其中 $\hat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^C$ 的各个元素反应了 softmax 回归预测各标签的概率。

1.3 多样本的矢量计算表达式

为了更方便地定义 torch tensor，假设 $\boldsymbol{b}$ 为**行向量**。

令 $X \in \mathbb{R}^{n\times d}$ 是 n 个样本的特征矩阵，则

$$\hat{Y} = \text{softmax}(XW + \boldsymbol{b}).$$

PyTorch 会自动对 $\boldsymbol{b}$ 进行广播。

1.4 参数学习

采用交叉熵损失函数，只关心正确类别的预测概率：

$$l(W, \boldsymbol{b}) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{c=1}^{C} y_c^{(n)} \log \hat{y}_c^{(n)}.$$

```
# 交叉熵的PyTorch实现
def cross_entropy(y_hat, y):
    return -torch.log(y_hat.gather(1, y.view(-1, 1)))
```

根据该损失函数，参数 W 和 $\boldsymbol{b}$ 的更新公式为

$$W_{t+1} \leftarrow W_t + \alpha \left(\frac{1}{|\mathcal{B}|} \sum_{n\in\mathcal{B}} \boldsymbol{x}^{(n)} \left(\boldsymbol{y}^{(n)} - \hat{\boldsymbol{y}}_{W_t}^{(n)} \right)^\top \right)$$

$$b_{t+1} \leftarrow b_t + \alpha \left(\frac{1}{|\mathcal{B}|} \sum_{n\in\mathcal{B}} \left(\boldsymbol{y}^{(n)} - \hat{\boldsymbol{y}}_{b_t}^{(n)} \right) \right),$$

其中 $\boldsymbol{y}^{(n)}$ 是一个 one-hot 向量，仅有 true label 位置对应的元素为 1。

2 多层感知机（MLP）

2.1 隐藏层

多层感知机（Multi-layer Perceptron）在单层神经网络的基础上引入了一到多个隐藏层（hidden layer）。隐藏层位于输入层和输出层之间。下图展示了一个多层感知机的神经网络图，它含有一个隐藏层，该层中有 5 个隐藏单元。多层感知机中的隐藏层和输出层都是全连接层。

2.2 矢量计算表达式

在不考虑激活函数的前提下，设输入样本 $X \in \mathbb{R}^{n\times d}$，标签个数为 q。对于仅包含单个隐藏层的神经网络，记隐藏层的输出为 $H \in \mathbb{R}^{n\times h}$。则

$$H = XW_h + \boldsymbol{b_h}$$

$$O = HW_o + \boldsymbol{b_o},$$

其中 $W_h \in \mathbb{R}^{d\times h}$，$\boldsymbol{b_h} \in \mathbb{R}^{1\times h}$，$W_o \in \mathbb{R}^{h\times q}$，$\boldsymbol{b_o} \in \mathbb{R}^{1\times q}$ 分别为隐藏层和输出层的权重及偏置。

因此

$$O = XW_hW_o + (\boldsymbol{b_h}W_o + \boldsymbol{b_o}),$$

这相当于是一个权重为 W_hW_o，偏置为 $\boldsymbol{b_h}W_o + \boldsymbol{b_o}$ 的单层神经网络。

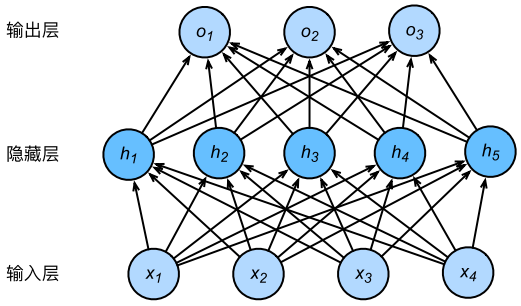

Figure 2.1: 多层感知机结构。

2.3 激活函数

全连接层只是对数据做仿射变换（affine transformation），而**多个仿射变换的复合仍然是一个仿射变换**。解决问题的一个方法是引入非线性变换，例如对隐藏变量使用按元素运算的非线性函数进行变换，然后再作为下一个全连接层的输入。这个非线性函数被称为激活函数（activation function）。

- ReLU（Rectified Linear Unit）：

$$\text{ReLU}(x) = \max(x, 0)$$

- sigmoid：

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

- tanh：

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

2.4 多层感知机

多层感知机就是含有至少一个隐藏层的由全连接层组成的神经网络，且每个隐藏层的输出通过激活函数进行变换。多层感知机的层数和各隐藏层中隐藏单元个数都是超参数。以单隐藏层为例，

$$H = \phi(XW_h + \boldsymbol{b_h})$$
$$O = HW_o + \boldsymbol{b_o}.$$

在分类问题中，我们可以对输出 O 做 softmax 运算，并使用 softmax 回归中的交叉熵损失函数。在回归问题中，我们将输出层的输出个数设为 1，并将输出 O 直接提供给线性回归中使用的平方损失函数。

3 权重衰减（weight decay）

应对过拟合的方法是**正则化**。以线性回归问题为例，默认的均方误差为

$$l(w_1, w_2, b) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left(x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)}\right)^2,$$

若加上 L_2 范数惩罚项，则得到新损失函数：

$$l(w_1, w_2, b) + \frac{\lambda}{2n}\|\boldsymbol{w}\|^2.$$

很容易计算，若对新损失函数求 $\boldsymbol{w}$ 的偏导，则可以得到 $\boldsymbol{w}$ 的更新公式为

$$w_1 \leftarrow \left(1 - \frac{\eta\lambda}{|\mathcal{B}|}\right)w_1 - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}x_1^{(i)}\left(x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)}\right)$$
$$w_2 \leftarrow \left(1 - \frac{\eta\lambda}{|\mathcal{B}|}\right)w_2 - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}x_2^{(i)}\left(x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)}\right),$$

这相当于是令权重先自乘小于 1 的数，再减去不含惩罚项的梯度。因此，L_2 范数正则化又叫权重衰减。

4 Dropout

在讲解 MLP 时我们给出了如图2.1所示的带有隐藏层的神经网络。

其中，对于单个样本 $\left([x_1, ..., x_4]^\top, y\right)$，隐藏单元 h_i 的计算表达式为

$$h_i = \phi\Big(\boldsymbol{x}^\top W_h(:, i) + \boldsymbol{b_h}(i)\Big).$$

若对该隐藏层使用 dropout，则该层的每个隐藏单元有一定概率会被丢弃掉。设丢弃概率（超参数）为 p，则 $\forall i, h_i$ 有 p 的概率会被清零，有 $1-p$ 的概率会被做拉伸。用数学语言描述即

$$h_i' = \frac{\xi_i}{1-p}h_i,$$

其中 ξ_i 是一个随机变量，$p(\xi_i = 0) = p$，$p(\xi_i = 1) = 1 - p$。则

$$\mathbb{E}[h_i'] = h_i.$$

这意味着 dropout **不改变输入的期望输出**（这就是要除以 $1-p$ 的原因）。

对上述 MLP 训练的时候使用 dropout，一种可能的网络结构如下：此时 MLP 的输出不依赖 h_2 和 h_5。由于在训练中隐藏层神经元的丢弃是随机的，即 $h_1, ..., h_5$ 都有可能被清零，输出层的计算无法过度依赖 $h_1, ..., h_5$ 中的任一个，从而在训练模型时起到正则化的作用，并可以用来应对过拟合。

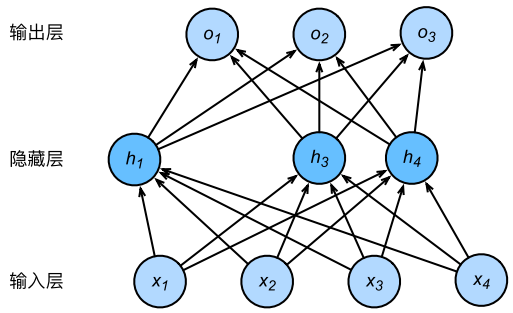

Figure 4.1: 允许 dropout 时，单隐藏层 MLP 的一种可能结构。

Dropout 是一种训练时应对过拟合的方法，并未改变网络的结构。**当参数训练完毕并用于测试时，任何参数都不会被** dropout。

5 反向传播的数学原理

到目前为止，我们只定义了模型的正向传播（forward）的过程，梯度的反向传播则是 PyTorch 自动实现的。接下来将以带 L_2 范数正则化项的、包含单个隐藏层的 MLP 解释反向传播的数学原理。

5.1 正向传播

不考虑偏置，设输入 $\boldsymbol{x} \in \mathbb{R}^d$，则得到中间变量 $\boldsymbol{z} = W^{(1)}\boldsymbol{x} \in \mathbb{R}^h$，其中 $W^{(1)} \in \mathbb{R}^{h\times d}$ 为隐藏层的权重，其中 h 是隐藏层神经元的个数；

$\boldsymbol{z}$ 作为输入传递给激活函数 ϕ，得到 $\boldsymbol{h} = \phi(\boldsymbol{z}) \in \mathbb{R}^h$；

将 $\boldsymbol{h}$ 传递给输出层，得到 $\boldsymbol{o} = W^{(2)}\boldsymbol{h} \in \mathbb{R}^q$，其中 $W^{(2)} \in \mathbb{R}^{q\times h}$ 为输出层的权重，q 为输出层神经元的个数（即 label 的个数）。

设损失函数为 l，且样本标签为 y，则单个样本的 loss 为 $L = l(\boldsymbol{o}, y)$。考虑 L_2 正则化项 $s = \frac{\lambda}{2}\left(\|W^{(1)}\|_F^2 + \|W^{(2)}\|_F^2\right)$，则单个样本上的优化目标为

$$J = L + s = l(\boldsymbol{o}, y) + \frac{\lambda}{2}\left(\|W^{(1)}\|_F^2 + \|W^{(2)}\|_F^2\right).$$

正向传播的计算图如下：

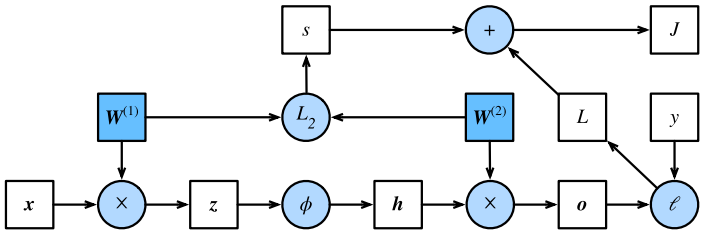

Figure 5.1: 正向传播的计算图。

5.2 反向传播

反向传播依据微积分中的链式法则，沿着从输出层到输入层的顺序，依次计算并存储目标函数有关神经网络各层的中间变量以及参数的梯度。**第 l 层的误差可由第 $l+1$ 层的误差得到**。

5.2.1 张量求导的链式法则

对于任意形状的张量 X, Y, Z，若 $Y = f(X), Z = f(Y)$，则

$$\frac{\partial Z}{\partial X} = prod(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X}),$$

其中 $prod(\cdot)$ 运算符将根据两个输入的形状，在必要的操作（如转置和互换输入位置）后对两个输入做乘法。

5.2.2 计算 $\frac{\partial J}{\partial W^{(2)}}$

将应用链式法则依次计算各中间变量和参数的梯度，其计算次序与前向传播中相应中间变量的计算次序恰恰相反。

首先 $J = L + s$（简单起见，仅考虑单个样本），所以 $\frac{\partial J}{\partial L} = 1, \frac{\partial J}{\partial s} = 1$；

其次，由于 $L = l(\boldsymbol{o}, y)$，所以 $\frac{\partial J}{\partial \boldsymbol{o}} = prod(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \boldsymbol{o}}) = \frac{\partial L}{\partial \boldsymbol{o}}$；

因为 $s = \frac{\lambda}{2}\left(\|W^{(1)}\|_F^2 + \|W^{(2)}\|_F^2\right)$，所以 $\frac{\partial s}{\partial W^{(1)}} = \lambda W^{(1)}$，$\frac{\partial s}{\partial W^{(2)}} = \lambda W^{(2)}$。因为 $\boldsymbol{o} = W^{(2)}\boldsymbol{h}$，所以 $\frac{\partial \boldsymbol{o}}{\partial (W^{(2)})^\top} = \boldsymbol{h}$。因此

$$\frac{\partial J}{\partial W^{(2)}} = prod(\frac{\partial J}{\partial \boldsymbol{o}}, \frac{\partial \boldsymbol{o}}{\partial W^{(2)}}) + prod(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial W^{(2)}}) = prod(\frac{\partial L}{\partial \boldsymbol{o}}, \boldsymbol{h}) + \lambda W^{(2)}.$$

5.2.3 计算 $\frac{\partial J}{\partial W^{(1)}}$

因为 $\frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}} = (W^{(2)})^\top$，所以 $\frac{\partial J}{\partial \boldsymbol{h}} = prod(\frac{\partial L}{\partial \boldsymbol{o}}, (W^{(2)})^\top)$；

进一步地，$\frac{\partial J}{\partial \boldsymbol{z}} = prod(\frac{\partial J}{\partial h}, \frac{\partial h}{\partial z}) = prod\left(\frac{\partial L}{\partial \boldsymbol{o}}, (W^{(2)})^\top\right) \odot \phi'(\boldsymbol{z})$；

最终，

$$\begin{aligned}\frac{\partial J}{\partial W^{(1)}} &= prod(\frac{\partial J}{\partial \boldsymbol{z}}, \frac{\partial \boldsymbol{z}}{\partial W^{(1)}}) + prod(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial W^{(1)}}) \\ &= prod\Big(prod(\frac{\partial L}{\partial \boldsymbol{o}}, (W^{(2)})^\top) \odot \phi'(\boldsymbol{z}), \boldsymbol{x}\Big) + \lambda W^{(1)}.\end{aligned}$$

在模型参数初始化完成后，我们交替地进行正向传播和反向传播，并根据反向传播计算的梯度迭代模型参数。我们在反向传播中使用了正向传播中计算得到的中间变量来避免重复计算，这导致正向传播结束后不能立即释放中间变量内存，因此训练要比预测占用更多的内存。另外需要指出的是，这些中间变量的个数大体上与网络层数线性相关，每个变量的大小跟批量大小和输入个数也是线性相关的，它们是导致较深的神经网络使用较大批量训练时更容易超内存的主要原因。

6 卷积神经网络

6.1 二维互相关运算

互相关运算图6.1所示（$corr = \text{rot180}(conv)$）：

```
# 二维互相关运算
def corr2d(X, K):
    h, w = K.shape
    # 窄卷积 (N - n + 1)
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i: i + h, j: j + w] * K).sum()
    return Y
```

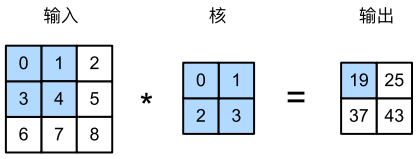

Figure 6.1: 互相关运算。

6.2 填充与步长

对于单个维度而言，设卷积核大小为 m，步长为 s，输入神经元两端各补 p 个零，则输出神经元的数量为

$$\left\lfloor \frac{n-m+2p}{s} \right\rfloor + 1.$$

- 窄卷积：$s=1, p=0 \to n-m+1$；
- 宽卷积：$s=1, p=m-1 \to n+m-1$；
- 等宽卷积：$s=1, p=\frac{m-1}{2} \to n$。

6.3 多输入通道与多输出通道

若输入数据的通道数为 c_i，则卷积核应当是一个大小为 $c_i \times k_h \times k_w$ 的 tensor。由于输入和卷积核各有 c_i 个通道，我们可以在各个通道上对输入的二维数组和卷积核的二维核数组做互相关运算，再将这 c_i 个互相关运算的二维输出按通道相加，得到一个二维数组。这就是含多个通道的输入数据与多输入通道的卷积核做二维互相关运算的输出。

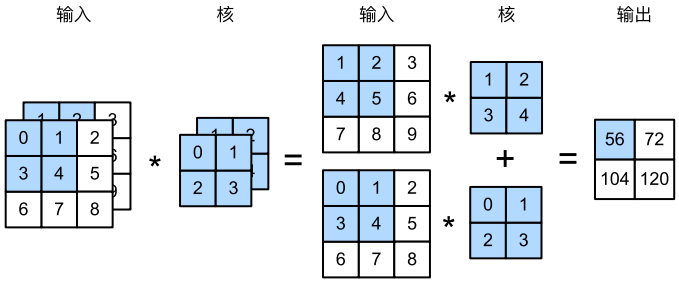

Figure 6.2: 多输入通道下的互相关运算。

当输入通道有多个时，因为我们对各个通道的结果做了累加，所以不论输入通道数是多少，输出通道数总是为 1。设卷积核输入通道数和输出通道数分别为 c_i 和 c_o，高和宽分别为 k_h 和 k_w。如果希望得到含多个通道的输出，我们可以为每个输出通道分别创建形状为 $c_i \times k_h \times k_w$ 的 tensor 的核数组。将它们在输出通道维上连结，卷积核的形状即 $c_o \times c_i \times k_h \times k_w$ 的 tensor。在做互相关运算时，每个输出通道上的结果由卷积核在该输出通道上的核数组与整个输入数组计算而来。

6.4 1×1 卷积层

输出中的每个元素来自输入中在高和宽上相同位置的元素在不同通道之间的按权重累加。假设我们将通道维当作特征维，将高和宽维度上的元素当成数据样本，那么 1×1 卷积层的作用与全连接层等价（区别在于 1×1 卷积层允许权重参数共享，因而拥有更少的参数数量）。

1×1 卷积层被当作保持高和宽维度形状不变的全连接层使用，可以通过调整网络层之间的通道数来控制模型复杂度。

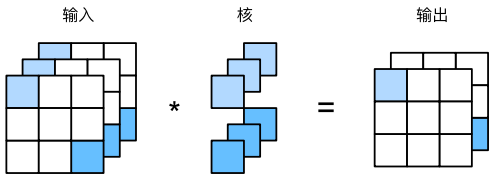

Figure 6.3: 1×1 卷积层。

6.5 池化层

池化层可以缓解卷积层对位置的过度敏感性。

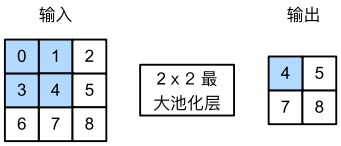

Figure 6.4: 最大池化层。

将卷积层的输出作为 2×2 最大池化的输入。设该卷积层输入是 `X`、池化层输出为 `Y`。无论是 `X[i, j]` 和 `X[i, j+1]` 值不同，还是 `X[i, j+1]` 和 `X[i, j+2]` 不同，池化层输出均有 `Y[i, j]=1`。也就是说，使用 2×2 最大池化层时，只要卷积层识别的模式在高和宽上移动不超过一个元素，我们依然可以将它检测出来。

池化层也可以有多通道。只不过，池化层是对每个输入通道分别池化，而不是像卷积层那样将各通道的输入按通道相加。这意味着池化层的输出通道数与输入通道数相等。

```
# 平均池化与最大池化
def pool2d(X, pool_size, mode='max'):
    X = X.float()
    p_h, p_w = pool_size
    Y = torch.zeros(X.shape[0] - p_h + 1, X.shape[1] - p_w + 1)
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i: i + p_h, j: j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y
```

6.6 LeNet-5

LeNet 分为卷积层块和全连接层块两个部分。

卷积层块里的基本单位是卷积层后接最大池化层：**卷积层用来识别图像里的空间模式**，如线条和物体局部，之后的**最大池化层则用来降低卷积层对位置的敏感性**。卷积层块由两个这样的基本单位重复堆叠构成。在卷积层块中，每个卷积层都使用 5×5 的窗口，并在输出上使用 sigmoid 激活函数。第一个卷积层输出通道数为 6，第二个卷积层输出通道数则增加到 16。这是因为第二个卷积层比第一个卷积层的输入的高和宽要小，所以增加输出通道使两个卷积层的参数尺寸类似。卷积层块的两个最大池化层的窗口形状均为 2×2，且步幅为 2。由于池化窗口与步幅形状相同，池化窗口在输入上每次滑动所覆盖的区域互不重叠。

卷积层块的输出形状为 (批量大小, 通道, 高, 宽)。当卷积层块的输出传入全连接层块时，全连接层块会将小批量中每个样本变平（flatten）。也就是说，全连接层的输入形状将变成二维，其中第一维是小批量中的样本，第二维是每个样本变平后的向量表示，且向量长度为通道、高和宽的乘积。可使用 `X.view(X.shape[0], -1)` 实现。

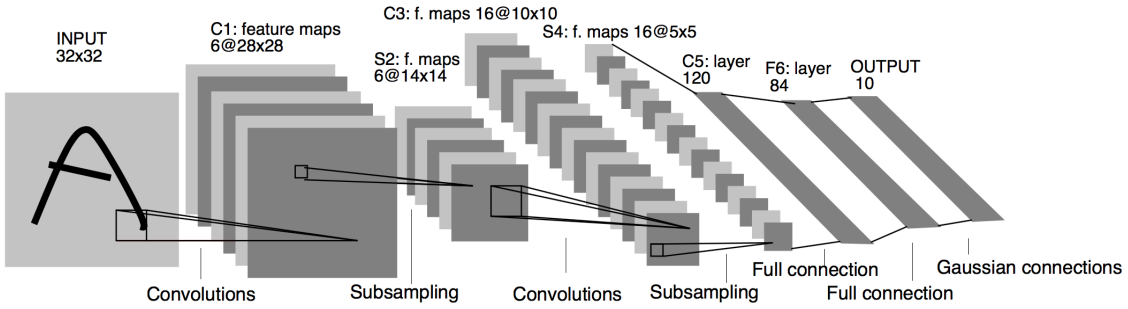

Figure 6.5: LeNet-5 的结构。

6.7 AlexNet

AlexNet 包含 8 层变换，其中有 5 层卷积和 2 层全连接隐藏层，以及 1 个全连接输出层。

AlexNet 第一层中的卷积窗口形状是 11×11。因为 ImageNet 中绝大多数图像的高和宽均比 MNIST 图像的高和宽大 10 倍以上，ImageNet 图像的物体占用更多的像素，所以需要更大的卷积窗口来捕获物体。第二层中的卷积窗口形状减小到 5×5，之后全采用 3×3。此外，第一、第二和第五个卷积层之后都使用了窗口形状为 3×3、步幅为 2 的最大池化层。而且，AlexNet 使用的卷积通道数也大于 LeNet 中的卷积通道数数十倍。

6.8 VGG-11

VGG 块的组成规律是：连续使用数个相同的填充为 1、窗口形状为 3×3 的卷积层后接上一个步幅为 2、窗口形状为 2×2 的最大池化层。卷积层保持输入的高和宽不变，而池化层则对其减半。

```
# VGG块
def vgg_block(num_convs, in_channels, out_channels):
    blk = []
    for i in range(num_convs):
        if i == 0:
```

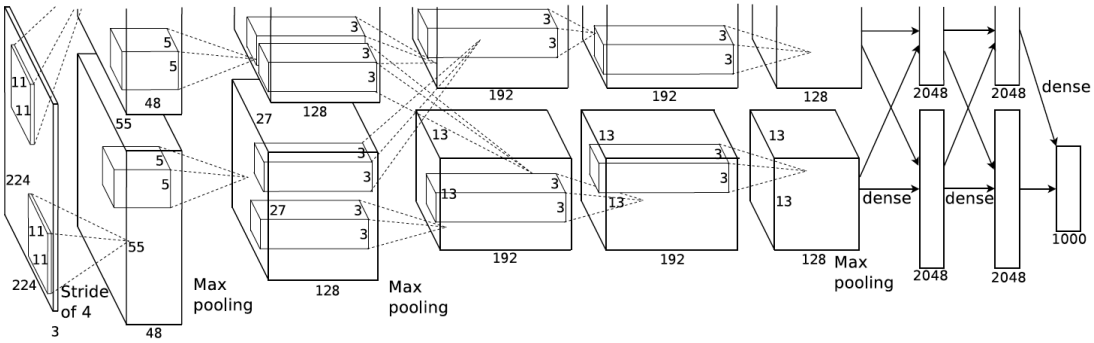

Figure 6.6: AlexNet 的结构。在最初的版本中，受限于当年 GPU 的显存，AlexNet 被拆分成两个部分，分别放到了两个 GPU 上。

```
        blk.append(nn.Conv2d(in_channels, out_channels, kernel_size=3,
            padding=1))
    else:
        blk.append(nn.Conv2d(out_channels, out_channels, kernel_size=3,
            padding=1))
    blk.append(nn.ReLU())
blk.append(nn.MaxPool2d(kernel_size=2, stride=2))
return nn.Sequential(*blk)
```

对于给定的感受野（与输出有关的输入图片的局部大小），**采用堆积的小卷积核优于采用大的卷积核，因为可以增加网络深度来保证学习更复杂的模式，而且代价还比较小（参数更少）**。例如，在 VGG 中，使用了 3 个 3×3 卷积核来代替 7×7 卷积核，使用了 2 个 3×3 卷积核来代替 5×5 卷积核，这样做的主要目的是在保证具有相同感知野的条件下，提升了网络的深度，在一定程度上提升了神经网络的效果。

VGG 网络有 5 个 VGG 块，前 2 块使用单卷积层，而后 3 块使用双卷积层。因为这个网络使用了 8 个卷积层和 3 个全连接层，所以经常被称为 VGG-11。

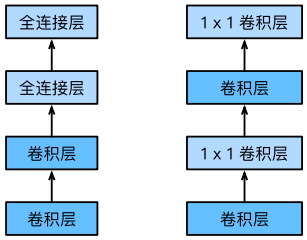

Figure 6.7: NiN 块的结构。

6.9 Network in Network (NiN)

卷积层的输入和输出通常是四维数组（样本，通道，高，宽），而全连接层的输入和输出则通常是二维数组（样本，特征）。如果想在全连接层后再接上卷积层，则需要将全连接层的输出变换为四维。可以用 1×1 卷积层代替全连接层，其中**空间维度（高和宽）上的每个元素相当于样本，通道相当于特征**。

```python
# nin_block
def nin_block(in_channels, out_channels, kernel_size, stride, padding):
    blk = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding),
        nn.ReLU(),
        nn.Conv2d(out_channels, out_channels, kernel_size=1),
        nn.ReLU(),
        nn.Conv2d(out_channels, out_channels, kernel_size=1),
        nn.ReLU()
    )
    return blk
```

NiN 使用卷积窗口形状分别为 11×11、5×5 和 3×3 的卷积层，相应的输出通道数也与 AlexNet 中的一致。每个 NiN 块后接一个步幅为 2、窗口形状为 3×3 的最大池化层。

NiN 去掉了 AlexNet 最后的 3 个全连接层，取而代之地，NiN 使用了输出通道数等于标签类别数的 NiN 块，然后使用**全局平均池化层**对每个通道中所有元素求平均并直接用于分类。这里的全局平均池化层即窗口形状等于输入空间维形状的平均池化层。NiN 的这个设计的好处是可以显著减小模型参数尺寸，从而缓解过拟合。然而，该设计有时会造成获得有效模型的训练时间的增加。

```python
class GlobalAvgPool2d(nn.Module):
    # 全局平均池化层可通过将池化窗口形状设置成输入的高和宽实现
    def __init__(self):
        super(GlobalAvgPool2d, self).__init__()
    def forward(self, x):
        # 将单个通道上（宽 * 高个元素的平均值计算出来）
        return F.avg_pool2d(x, kernel_size=x.size()[2:])
```

6.10 GoogLeNet

GoogLeNet 吸收了 NiN 中网络串联网络的思想，并在此基础上做了很大改进。

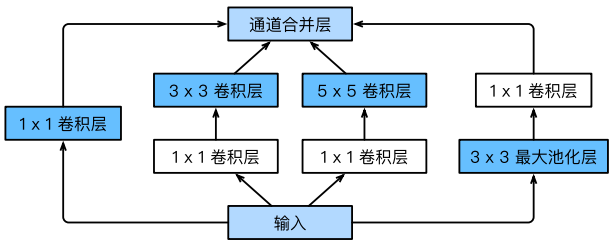

Figure 6.8: Inception 块的结构。

GoogLeNet 中的基础卷积块叫作 Inception 块，有 4 条并行的线路。前 3 条线路使用窗口大小分别是 1×1、3×3 和 5×5 的卷积层来抽取不同空间尺寸下的信息，其中中间 2 个线路会对输入先做 1×1 卷积来减少输入通道数，以降低模型复杂度。第四条线路则使用 3×3 最大池化层，后接 1×1 卷积层来改变通道数。4 条线路都使用了合适的填充来使输入与输出的高和宽一致。最后将每条线路的输出在通道维上连结，并输入接下来的层中去。

```
class Inception(nn.Module):
    # c1 - c4为每条线路里的层的输出通道数
    def __init__(self, in_c, c1, c2, c3, c4):
        super(Inception, self).__init__()
        # 线路1, 单1 x 1卷积层
        self.p1_1 = nn.Conv2d(in_c, c1, kernel_size=1)
        # 线路2, 1 x 1卷积层后接3 x 3卷积层
        self.p2_1 = nn.Conv2d(in_c, c2[0], kernel_size=1)
        self.p2_2 = nn.Conv2d(c2[0], c2[1], kernel_size=3, padding=1)
        # 线路3, 1 x 1卷积层后接5 x 5卷积层
        self.p3_1 = nn.Conv2d(in_c, c3[0], kernel_size=1)
        self.p3_2 = nn.Conv2d(c3[0], c3[1], kernel_size=5, padding=2)
        # 线路4, 3 x 3最大池化层后接1 x 1卷积层
        self.p4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.p4_2 = nn.Conv2d(in_c, c4, kernel_size=1)

    def forward(self, x):
        p1 = F.relu(self.p1_1(x))
        p2 = F.relu(self.p2_2(F.relu(self.p2_1(x))))
        p3 = F.relu(self.p3_2(F.relu(self.p3_1(x))))
        p4 = F.relu(self.p4_2(self.p4_1(x)))
        return torch.cat((p1, p2, p3, p4), dim=1) # 在通道维上连结输出
```

GoogLeNet 跟 VGG 一样，在主体卷积部分中使用 5 个模块，每个模块之间使用步幅为 2 的 3×3 最大池化层来减小输出高宽。第一模块使用一个 64 通道的 7×7 卷积层。第二模块使用 2 个卷积层：首先是 64 通道的 1×1 卷积层，然后是将通道增大 3 倍的 3×3 卷积层（和 Inception 模块中的线路 2 一致）。第三模块串联 2 个完整的 Inception 块。第一个 Inception 块的输出通道数为 64+128+32+32=256。第二个 Inception 块输出通道数增至 128+192+96+64=480。第四模块串联了 5 个 Inception 块。第五模块串联了 2 个 Inception 块并使用全局平均池化层直接得到分类结果。

6.11 批量归一化

通常来说，数据标准化预处理对于浅层模型就足够有效了。随着模型训练的进行，当每层中参数更新时，靠近输出层的输出较难出现剧烈变化。但对深层神经网络来说，即使输入数据已做标准化，训练中模型参数的更新依然很容易造成靠近输出层输出的剧烈变化。这种计算数值的不稳定性通常令我们难以训练出有效的深度模型。

批量归一化的提出正是为了应对深度模型训练的挑战。在模型训练时，批量归一化利用小批量上的均值和标准差，不断调整神经网络中间输出，从而使整个神经网络在各层的中间输出的数值更稳定。**批量归一化**和**残差网络**为训练和设计深度模型提供了两类重要思路。

6.11.1 对全连接层批量归一化

使用批量归一化的全连接层的输出为

$$\phi(BN(x)) = \phi(BN(W\boldsymbol{u} + \boldsymbol{b})),$$

其中 u 为全连接层的输入，BN 为批量归一化运算符。

对于小批量的**仿射变换的输出** $\mathcal{B} = \{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$，其中 $\boldsymbol{x}^{(i)} \in \mathbb{R}^d$，则批量归一化的输出为

$$\boldsymbol{y}^{(i)} = BN(\boldsymbol{x}^{(i)}) \in \mathbb{R}^d.$$

$BN(\cdot)$ 的具体步骤如下：

首先对小批量 $\mathcal{B}$ 求均值和方差：

$$\boldsymbol{\mu}_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}\boldsymbol{x}^{(i)}, \boldsymbol{\sigma}_{\mathcal{B}}^2 \leftarrow \frac{1}{m-1}\sum_{i=1}^{m}(\boldsymbol{x}^{(i)}-\boldsymbol{\mu}_{\mathcal{B}})^2,$$

其中的平方计算是按元素求平方。接下来，使用按元素开方和按元素除法对 $\boldsymbol{x}^{(i)}$ 标准化：

$$\hat{\boldsymbol{x}}^{(i)} \leftarrow \frac{\boldsymbol{x}^{(i)}-\boldsymbol{\mu}_{\mathcal{B}}}{\sqrt{\boldsymbol{\sigma}_{\mathcal{B}}^2+\epsilon}},$$

这里 $\epsilon > 0$ 是一个很小的常数，保证分母大于 0。

批量归一化层引入了两个可以学习的模型参数，拉伸（scale）参数 $\boldsymbol{\gamma}$ 和偏移（shift）参数 $\boldsymbol{\beta}$。这两个参数和 $\boldsymbol{x}^{(i)}$ 形状相同，皆为 d 维向量。它们与分别做按元素乘法和加法计算：

$$\boldsymbol{y}^{(i)} \leftarrow \boldsymbol{\gamma} \odot \hat{\boldsymbol{x}}^{(i)} + \boldsymbol{\beta}.$$

可学习的拉伸和偏移参数保留了不对 $\boldsymbol{x}^{(i)}$ 做批量归一化的可能：此时只需学出 $\boldsymbol{\gamma} = \sqrt{\boldsymbol{\sigma}_{\mathcal{B}}^2+\epsilon}$，$\boldsymbol{\beta} = \boldsymbol{\mu}_{\mathcal{B}}$。我们可以对此这样理解：如果批量归一化无益，理论上，学出的模型可以不使用批量归一化。

6.11.2 对卷积层批量归一化

对卷积层来说，批量归一化发生在卷积计算之后、应用激活函数之前。如果卷积计算输出多个通道，我们需要对这些通道的输出分别做批量归一化，且每个通道都拥有独立的拉伸和偏移参数，并均为**标量**。

设小批量中有 m 个样本。在单个通道上，假设卷积计算输出的高和宽分别为 p 和 q。我们需要对该通道中 $m \times p \times q$ 个元素同时做批量归一化。对这些元素做标准化计算时，我们使用相同的均值和方差，即该通道中 $m \times p \times q$ 个元素的均值和方差。

6.11.3 预测时的批量归一化

使用批量归一化训练时，我们可以将批量大小设得大一点，从而使批量内样本的均值和方差的计算都较为准确。将训练好的模型用于预测时，我们希望模型对于任意输入都有确定的输出。因此，单个样本的输出不应取决于批量归一化所需要的随机小批量中的均值和方差。一种常用的方法是通过移动平均估算整个训练数据集的样本均值和方差，并在预测时使用它们得到确定的输出。可见，和 dropout 一样，批量归一化层在训练模式和预测模式下的计算结果也是不一样的。

```
# 对输入的minibatch进行批量归一化
def batch_norm(is_training, X, gamma, beta, moving_mean, moving_var, eps,
    momentum):
    # 判断当前模式是训练模式还是预测模式
    if not is_training:
        # 如果是在预测模式下，直接使用传入的移动平均所得的均值和方差
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # 使用全连接层的情况，计算特征维上的均值和方差
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # 使用二维卷积层的情况，计算通道维上（axis=1）的均值和方差。
```

```
            # 这里我们需要保持X的形状以便后面可以做广播运算
            mean = X.mean(dim=0, keepdim=True).mean(dim=2,
                keepdim=True).mean(dim=3, keepdim=True)
            var = ((X - mean) ** 2).mean(dim=0, keepdim=True).mean(dim=2,
                keepdim=True).mean(dim=3, keepdim=True)
        # 训练模式下用当前的均值和方差做标准化
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # 更新移动平均的均值和方差
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta         # 拉伸和偏移
    return Y, moving_mean, moving_var
```

6.12 ResNet-18

普通的网络结构（左）与加入残差连接的网络结构（右）：在右图所示的残差块中，虚线框内要

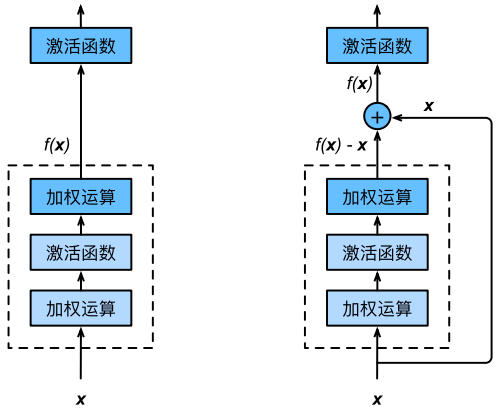

Figure 6.9: 残差网络的基本结构（右）。

学习的是残差映射 $f(\boldsymbol{x})-\boldsymbol{x}$，当理想映射接近恒等映射时（即 $f(\boldsymbol{x})=\boldsymbol{x}$），虚线框内上方的加权运算的权重和偏差参数会被学习为 $\mathbf{0}$。此时的残差映射可以捕捉恒等映射的细微波动。

```
# 实现图6.9（右）所示的残差块
class Residual(nn.Module):
    # ResNet沿用了VGG全3×3卷积层的设计。残差块里首先有2个有相同输出通道数的3×3卷积层
    # 每个卷积层后接一个批量归一化层
    def __init__(self, in_channels, out_channels, use_1x1conv=False, stride=1):
        super(Residual, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
            padding=1, stride=stride)
        self.bn1 = nn.BatchNorm2d(out_channels)

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
            padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)

        if use_1x1conv:
            # 想要改变通道数
```

```
            self.conv3 = nn.Conv2d(in_channels, out_channels, kernel_size=1,
                stride=stride)
        else:
            self.conv3 = None

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))

        # 将输入跳过这两个卷积运算后直接加在最后的ReLU激活函数前
        if self.conv3:
            X = self.conv3(X)
        return F.relu(Y + X)
```

ResNet 第一层与 GooLeNet 第一层一样，在输出通道数为 64、步幅为 2 的 7×7 卷积层后接步幅为 2 的 3×3 的最大池化层。不同之处在于 ResNet 在卷积层后增加的批量归一化层。GoogLeNet 在后面接了 4 个由 Inception 块组成的模块。ResNet 则使用 4 个由残差块组成的模块，每个模块使用若干个同样输出通道数的残差块，第一个模块的通道数同输入通道数一致。每个模块在第一个残差块里将上一个模块的通道数翻倍，并将高和宽减半。最后，使用**全局平均池化层**对每个通道中所有元素求平均并输入给全连接层用于分类。

这里每个模块里有 4 个卷积层（不计算 1×1 卷积层），加上最开始的卷积层和最后的全连接层，共计 18 层。这个模型通常也被称为 ResNet-18。

```
# 由四个残差块组成的模块
def resnet_block(in_channels, out_channels, num_residuals, first_block=False):
    if first_block:
        assert in_channels == out_channels
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(in_channels, out_channels, use_1x1conv=True,
                stride=2))
        else:
            blk.append(Residual(out_channels, out_channels))
    return nn.Sequential(*blk)
```

6.13 DenseNet

DenseNet 里模块 B 的输出不是像 ResNet 那样和模块 A 的输出相加，而是在通道维上连结。这样模块 A 的输出可以直接传入模块 B 后面的层。

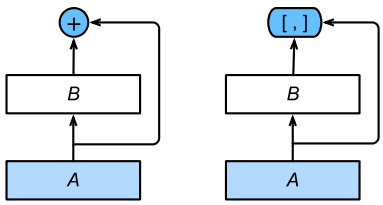

Figure 6.10: DenseNet 的基本结构：稠密块（dense block）和过渡层（transition layer）。

DenseNet 的主要构建模块是稠密块（dense block）和过渡层（transition layer）。前者定义了输入和输出是如何连结的，后者则用来控制通道数，使之不过大。

```
# 稠密块
def conv_block(in_channels, out_channels):
    blk = nn.Sequential(
        nn.BatchNorm2d(in_channels),
        nn.ReLU(),
        nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
    )
    return blk

# 稠密块由多个conv_block组成，每块使用相同的输出通道数
class DenseBlock(nn.Module):
    def __init__(self, num_convs, in_channels, out_channels):
        super(DenseBlock, self).__init__()
        net = []
        for i in range(num_convs):
            in_c = in_channels + i * out_channels
            net.append(conv_block(in_c, out_channels))
        self.net = nn.ModuleList(net)
        self.out_channels = in_channels + num_convs * out_channels

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            X = torch.cat((X, Y), dim=1)
        return X
```

```
# 过渡层
def transition_block(in_channels, out_channels):
    return nn.Sequential(
        nn.BatchNorm2d(in_channels),
        nn.ReLU(),
        nn.Conv2d(in_channels, out_channels, kernel_size=1),
        nn.AvgPool2d(kernel_size=2, stride=2)
    )
```

DenseNet 首先使用同 ResNet 一样的单卷积层和最大池化层。随后，类似于 ResNet 接下来使用的 4 个残差块，DenseNet 使用的是 4 个稠密块。同 ResNet 一样，我们可以设置每个稠密块使用多少个卷积层。这里我们设成 4，从而与上一节的 ResNet 保持一致。稠密块里的卷积层通道数（即增长率）设为 32，所以每个稠密块将增加 128 个通道。

ResNet 里通过步幅为 2 的残差块在每个模块之间减小高和宽。这里我们则使用过渡层来减半高和宽，并减半通道数。同样地，最后接上全局池化层和全连接层来输出。

7 循环神经网络

7.1 语言模型

假设序列 $w_1, w_2, ..., w_T$ 的每个词是依次生成的，则

$$P(w_1, ..., w_T) = \prod_{t=1}^{T} P(w_t | w_1, ..., w_{t-1}).$$

基于 $n-1$ 阶马尔可夫链，语言模型可改写为

$$P(w_1, ..., w_T) \approx \prod_{t=1}^{T} P(w_t | w_{t-(n-1)}, ..., w_{t-1}),$$

即当前词的出现仅和前面的 $n-1$ 个词有关，这就是 n **元语法**。

7.2 RNN 的基本结构

循环神经网络并非刚性地记忆所有固定长度的序列，而是通过隐藏状态来存储之前时间步的信息。

在 MLP 中，设输入的小批量数据样本为 $X \in \mathbb{R}^{n \times d}$，则隐藏层的输出为 $H = \phi(XW_{xh} + \boldsymbol{b}_h) \in \mathbb{R}^{n \times h}$，输出层的输出为 $O = HW_{hq} + \boldsymbol{b}_q \in \mathbb{R}^{n \times q}$，最后通过 softmax$(O)$ 得到输出类别的概率分布。

在 MLP 的基础上，将上一时间步隐藏层的输出作为这一时间步隐藏层计算的输入，即

$$H_t = \phi(X_t W_{xh} + H_{t-1} W_{hh} + \boldsymbol{b}_h),$$

通过引入新的权重参数将上一轮隐藏层的输出作为本轮隐藏层计算的依据之一。输出层的计算和 MLP 一致。

采用这种方式构建的循环神经网络的参数包含 $W_{xh} \in \mathbb{R}^{d \times h}, W_{hh} \in \mathbb{R}^{h \times h}, \boldsymbol{b}_h \in \mathbb{R}^{1 \times h}, W_{hq} \in \mathbb{R}^{h \times q}, \boldsymbol{b}_q \in \mathbb{R}^{1 \times q}$。

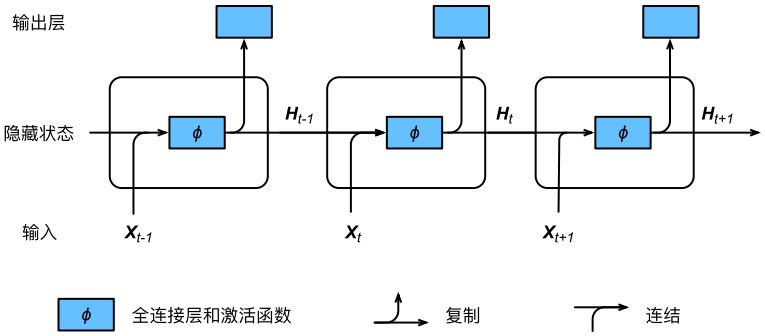

Figure 7.1: 包含单层隐藏状态的循环神经网络的结构。

在时间步 t，隐藏状态的计算可以看成是将输入 X_t 和前一时间步隐藏状态 H_{t-1} 连结后输入一个激活函数为 ϕ 的全连接层。该全连接层的输出就是当前时间步的隐藏状态 H_t 且模型参数为 W_{xh} 和 W_{hh} 的连结，偏差为 $\boldsymbol{b}_h$。

基于字符级循环神经网络来创建语言模型：输入是一个字符，神经网络基于当前和过去的字符来预测下一个字符。在训练时，我们对每个时间步的输出层输出使用 softmax 运算，然后使用交叉熵损失函数来计算它与标签的误差。

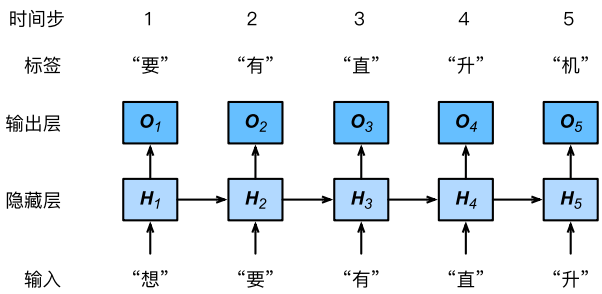

Figure 7.2: 基于字符级循环神经网络创建的语言模型。

7.3 时序数据的采样

- **随机采样**：每次从数据里随机采样一个小批量。其中批量大小 `batch size` 指每个小批量的样本数，`num steps` 为每个样本所包含的时间步数。在随机采样中，每个样本是原始序列上任意截取的一段序列。相邻的两个随机小批量在原始序列上的位置不一定相毗邻。因此，我们无法用一个小批量最终时间步的隐藏状态来初始化下一个小批量的隐藏状态。在训练模型时，每次随机采样前都需要重新初始化隐藏状态。

- **相邻采样**：相邻的两个随机小批量在原始序列上的位置相毗邻。这时候，我们就可以用一个小批量最终时间步的隐藏状态来初始化下一个小批量的隐藏状态，从而使下一个小批量的输出也取决于当前小批量的输入，并如此循环下去。这对实现循环神经网络造成了两方面影响：一方面，在训练模型时，我们只需在每一个迭代周期开始时初始化隐藏状态；另一方面，当多个相邻小批量通过传递隐藏状态串联起来时，模型参数的梯度计算将依赖所有串联起来的小批量序列。同一迭代周期中，随着迭代次数的增加，梯度的计算开销会越来越大。为了使模型参数的梯度计算只依赖一次迭代读取的小批量序列，我们可以在每次读取小批量前将隐藏状态从计算图中分离出来。

7.4 裁剪梯度

循环神经网络中较容易出现梯度衰减或梯度爆炸。为了应对梯度爆炸，我们可以裁剪梯度（clip gradient），裁剪后的梯度的 $\|\cdot\|_2$ 不超过 θ:

$$\min\left(\frac{\theta}{\|\boldsymbol{g}\|}, 1\right)\cdot\boldsymbol{g}.$$

```
# 裁剪梯度
def grad_clipping(params, theta, device):
    norm = torch.tensor([0.], device=device)
    for param in params:
        norm += (param.grad.data ** 2).sum()
    norm = norm.sqrt().item()
    if norm > theta:
        for param in params:
            param.grad.data *= theta / norm
```

7.5 困惑度

使用困惑度（perplexity）评价语言模型的好坏。困惑度是对交叉熵损失函数做指数运算后得到的值。

- 最佳情况下，模型总是把标签类别的概率预测为 1，此时困惑度为 1；
- 最坏情况下，模型总是把标签类别的概率预测为 0，此时困惑度为正无穷；
- 基线情况下，模型总是预测所有类别的概率都相同，此时困惑度为类别个数。

一个有效地模型的困惑度应在 1 和 `vocab size` 之间。

7.6 RNN 的实现

首先按照如下方式实现 `rnn layer`：

```
rnn_layer = nn.RNN(input_size=vocab_size, hidden_size=hidden_size)
```

作为 `nn.RNN` 的实例，`rnn layer` 在前向计算后会分别返回输出和隐藏状态。其中输出指的是隐藏层在各个时间步上计算并输出的隐藏状态，它们通常作为后续输出层的输入，形状为 `(num steps, batch size, hidden size)`。需要强调的是，该输出本身并不涉及输出层计算。隐藏状态指的是隐藏层在**最后时间步**的隐藏状态（图7.3中的 $H_T^{(1)}, ..., H_T^{(L)}$）。当隐藏层有多层时，每一层的隐藏状态都会记录在该变量中。

基于 `rnn layer`，实现 RNN 模型：

```
class RNNModel(nn.Module):
    def __init__(self, rnn_layer, vocab_size):
        super(RNNModel, self).__init__()
        self.rnn = rnn_layer
        self.hidden_size = rnn_layer.hidden_size * (2 if rnn_layer.bidirectional
            else 1)
        self.vocab_size = vocab_size
        self.dense = nn.Linear(self.hidden_size, vocab_size)
        self.state = None

    def forward(self, inputs, state):
        # input is of size (batch_size, num_steps)
        X = my_utils.to_onehot(inputs, self.vocab_size)
        Y, self.state = self.rnn(torch.stack(X), state)
        # change size into (num_steps * batch_size, num_hiddens)
        output = self.dense(Y.view(-1, Y.shape[-1]))
        return output, self.state
```

7.7 通过时间反向传播（BPTT）

如果不裁剪梯度，RNN 模型将无法正常训练。为了深刻理解这一现象，本节将介绍循环神经网络中梯度的计算和存储方法，即通过时间反向传播（back-propagation through time）。需要将循环神经网络按时间步展开，从而得到模型变量和参数之间的依赖关系，并依据链式法则应用反向传播计算并存储梯度。

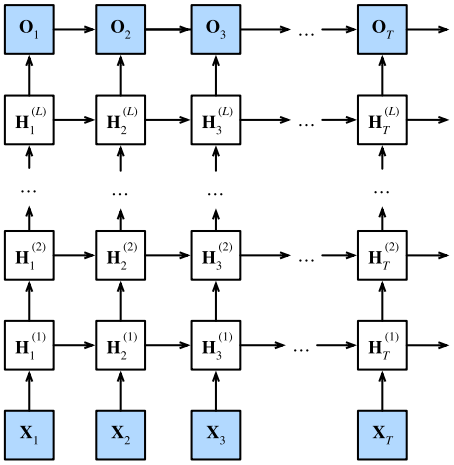

Figure 7.3: 深度循环神经网络的输入、输出与隐藏状态。

7.7.1 含有单隐藏层的 RNN

考虑一个无偏差项的循环神经网络，且激活函数为恒等映射 $\phi(\boldsymbol{x}) = \boldsymbol{x}$。设时间步 t 的输入为单个样本 $\boldsymbol{x}_t \in \mathbb{R}^d$，标签为 y_t，则隐藏状态 $\boldsymbol{h}_t \in \mathbb{R}^h$ 的计算表达式为

$$\boldsymbol{h}_t = W_{hx}\boldsymbol{x}_t + W_{hh}\boldsymbol{h}_{t-1},$$

其中 $W_{hx} \in \mathbb{R}^{h\times d}$ 和 $W_{hh} \in \mathbb{R}^{h\times h}$ 是隐藏层权重参数。

设输出层权重参数为 $X_{qh} \in \mathbb{R}^{q\times h}$，则时间步 t 的输出层变量 $\boldsymbol{o}_t \in \mathbb{R}^q$ 的计算表达式为

$$\boldsymbol{o}_t = W_{qh}\boldsymbol{h}_t.$$

设时间步 t 的损失为 $l(\boldsymbol{o}_t, y_t)$，则时间步数为 T 的损失函数定义为

$$L \triangleq \frac{1}{T}\sum_{t=1}^{T} l(\boldsymbol{o}_t, y_t).$$

7.7.2 模型计算图

图7.4给出了时间步数为 3 的循环神经网络模型计算中的依赖关系。方框代表变量（无阴影）或参数（有阴影），圆圈代表运算符。

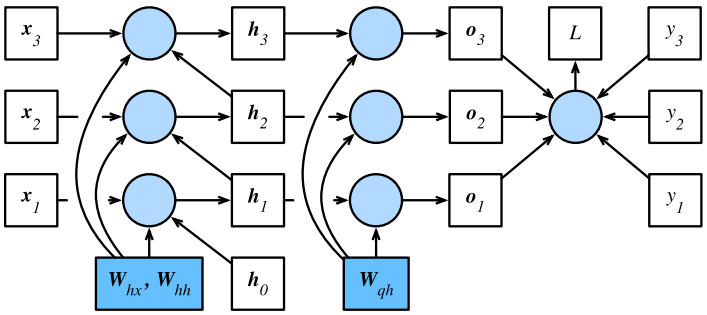

Figure 7.4: 含有单隐藏层的 RNN 的模型计算图。

7.7.3 通过时间反向传播

计算 L 关于各时间步输出层变量 $\boldsymbol{o}_t$ 的梯度：

$$\forall t \in \{1, ..., T\} : \frac{\partial L}{\partial \boldsymbol{o}_t} = \frac{\partial l(\boldsymbol{o}_t, y_t)}{T \cdot \partial \boldsymbol{o}_t}.$$

计算 L 关于输出层权重参数 W_{qh} 的梯度：

L 通过 $\boldsymbol{o}_1, ..., \boldsymbol{o}_T$ 依赖 W_{qh}。所以

$$\frac{\partial L}{\partial W_{qh}} = \sum_{t=1}^{T} prod\Big(\frac{\partial L}{\partial \boldsymbol{o}_t}, \frac{\partial \boldsymbol{o}_t}{\partial W_{qh}}\Big) = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{o}_t} \boldsymbol{h}_t^\top.$$

计算 L 关于各时间步 t 隐藏层变量 $\boldsymbol{h}_t$ 的梯度：对于 $t = T$ 和 $t = 1, .., T-1$ 而言，L 对 $\boldsymbol{h}_t$ 的依赖不同。对于 $t = T$，L 只通过 $\boldsymbol{o}_T$ 依赖隐藏状态 $\boldsymbol{h}_T$。因此，梯度计算表达式为

$$\frac{\partial L}{\partial \boldsymbol{h}_T} = prod\Big(\frac{\partial L}{\partial \boldsymbol{o}_T}, \frac{\partial \boldsymbol{o}_T}{\partial \boldsymbol{h}_T}\Big) = W_{qh}^\top \frac{\partial L}{\partial \boldsymbol{o}_T}.$$

对于 $t = 1, ..., T-1$，L 通过 $\boldsymbol{o}_t$ 和 $\boldsymbol{h}_{t+1}$ 依赖隐藏状态 $\boldsymbol{h}_t$。因此，梯度计算表达式为

$$\frac{\partial L}{\partial \boldsymbol{h}_t} = prod\Big(\frac{\partial L}{\partial \boldsymbol{h}_{t+1}}, \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t}\Big) + prod\Big(\frac{\partial L}{\partial \boldsymbol{o}_t}, \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{h}_t}\Big) = W_{hh}^\top \frac{\partial L}{\partial \boldsymbol{h}_{t+1}} + W_{qh}^\top \frac{\partial L}{\partial \boldsymbol{o}_t}.$$

将上面的递归公式展开，对任意时间步 $1 \le t \le T$，我们可以得到目标函数有关隐藏状态梯度的通项公式：

$$\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{h}_t} &= \Big(W_{hh}^\top\Big)^2 \frac{\partial L}{\partial \boldsymbol{o}_{t+2}} + W_{hh}^\top W_{qh}^\top \frac{\partial L}{\partial \boldsymbol{o}_{t+1}} \\
&= \Big(W_{hh}^\top\Big)^3 \frac{\partial L}{\partial \boldsymbol{o}_{t+3}} + \Big(W_{hh}^\top\Big)^2 W_{qh}^\top \frac{\partial L}{\partial \boldsymbol{o}_{t+2}} + W_{hh}^\top W_{qh}^\top \frac{\partial L}{\partial \boldsymbol{o}_{t+1}} \\
&= ... \\
&= \sum_{i=t}^{T} \Big(W_{hh}^\top\Big)^{T-i} W_{qh}^\top \frac{\partial L}{\partial \boldsymbol{o}_{T-i+t}}.
\end{aligned}$$

计算 L 关于隐藏层权重参数 W_{qh} 的梯度：

$$\frac{\partial L}{\partial W_{hx}} = \sum_{t=1}^{T} prod\Big(\frac{\partial T}{\partial \boldsymbol{h}_t}, \frac{\boldsymbol{h}_t}{\partial W_{hx}}\Big) = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{h}_t} \boldsymbol{x}_t^\top$$

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} prod\Big(\frac{\partial T}{\partial \boldsymbol{h}_t}, \frac{\boldsymbol{h}_t}{\partial W_{hh}}\Big) = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{h}_t} \boldsymbol{h}_{t-1}^\top.$$

7.8 门控逻辑单元（GRU）

当时间步数较大或者时间步较小时，循环神经网络的梯度较容易出现衰减或爆炸。虽然裁剪梯度可以应对梯度爆炸，但无法解决梯度衰减的问题。通常由于这个原因，循环神经网络在实际中较难捕捉时间序列中时间步距离较大的依赖关系。

门控循环神经网络（gated recurrent neural network）的提出，正是为了更好地捕捉时间序列中时间步距离较大的依赖关系。它通过可以学习的门来控制信息的流动。其中，门控循环单元（gated recurrent unit，GRU）是一种常用的门控循环神经网络。

GRU 引入了重置门（reset gate）和更新门（update gate）的概念，从而修改了循环神经网络中隐藏状态的计算方式。

重置门和更新门的输入均为当前时间步的小批量输入 $X_t \in \mathbb{R}^{n \times d}$ 和上一时间步的隐藏状态 $H_{t-1} \in \mathbb{R}^{n \times h}$，输出由激活函数为 sigmoid 函数的全连接层得到：

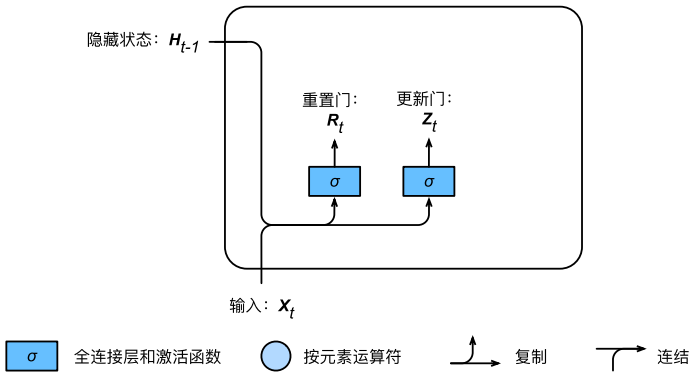

Figure 7.5: 重置门和更新门。

$$R_t = \sigma\Big(X_t W_{xr} + H_{t-1} W_{hr} + \boldsymbol{b}_r\Big) \in \mathbb{R}^{n\times h}$$
$$Z_t = \sigma\Big(X_t W_{xz} + H_{t-1} W_{hz} + \boldsymbol{b}_z\Big) \in \mathbb{R}^{n\times h},$$

其中 $W_{xr}, W_{xz} \in \mathbb{R}^{d\times h}$ 和 $W_{hr}, W_{hz} \in \mathbb{R}^{h\times h}$ 为权重参数，$\boldsymbol{b}_r, \boldsymbol{b}_z \in \mathbb{R}^{1\times h}$ 为偏置。选择 sigmoid 作为激活函数是为了将这两个逻辑门的输出限定在 0 到 1 之间。

随后，将当前时间步重置门的输出与上一时间步隐藏状态做按元素乘法。如果重置门中元素值接近 0，那么意味着重置对应隐藏状态元素为 0，即丢弃上一时间步的隐藏状态。如果元素值接近 1，那么表示保留上一时间步的隐藏状态。然后，将按元素乘法的结果与当前时间步的输入连结，再通过含激活函数 tanh 的全连接层计算出候选隐藏状态，其所有元素的值域为 $[-1, 1]$。因此，当前时间步候选隐藏状态的计算表达式为

$$\tilde{H}_t = \tanh\Big(X_t W_{xh} + \big(H_{t-1} \odot R_t\big) W_{hh} + \boldsymbol{b}_h\Big) \in \mathbb{R}^{n\times h},$$

其中 $W_{xh} \in \mathbb{R}^{d\times h}, W_{hh} \in \mathbb{R}^{h\times h}$ 为权重参数，$\boldsymbol{b}_h \in \mathbb{R}^{1\times h}$ 为偏置。**重置门控制了上一时间步的隐藏状态如何流入当前时间步的候选隐藏状态，从而更好地捕捉时间序列里短期的依赖关系**。而上一时间步的隐藏状态可能包含了时间序列截至上一时间步的全部历史信息。因此，重置门可以用来丢弃与预测无关的历史信息。

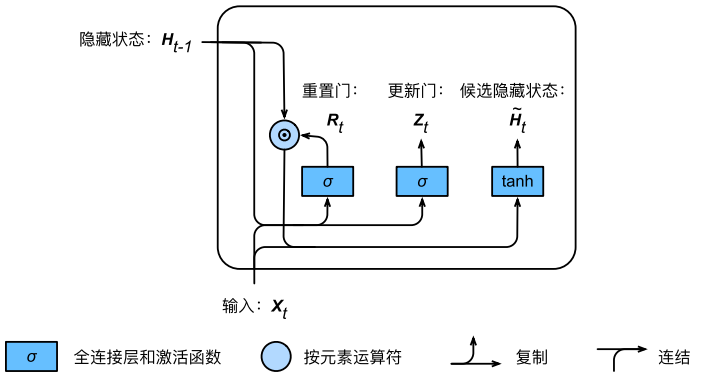

Figure 7.6: 计算候选隐藏状态。

然后，计算当前时间步的隐藏状态 $H_t \in \mathbb{R}^{n\times h}$：

$$H_t = Z_t \odot H_{t-1} + (1 - Z_t) \odot \tilde{H}_t.$$

更新门可以控制隐藏状态应该如何被包含当前时间步信息的候选隐藏状态所更新。假设更新门在时间步 t' 到 t $(t' < t)$ 之间一直近似 1，那么在时间步 t' 到 t 之间的输入信息几乎没有流入时间步 t 的隐藏状态 H_t。这种现象可以理解为：**较早时刻的隐藏状态 $H_{t'-1}$ 一直通过时间保存并传递至当前的时间步 t**。这个设计可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时间序列中时间步距离较大的依赖关系。

最后，时间步 t 的输出的计算方式不变，仍为

$$O_t = HW_{hq} + \boldsymbol{b}_q \in \mathbb{R}^{n\times q},$$

其中 $W_{hq} \in \mathbb{R}^{h\times q}$ 为权重参数，$\boldsymbol{b}_q \in \mathbb{R}^{1\times q}$ 为偏置。

```
# 定义GRU模型
def gru(inputs, state, params):
    W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        Z = torch.sigmoid(torch.matmul(X, W_xz) + torch.matmul(H, W_hz) + b_z)
        R = torch.sigmoid(torch.matmul(X, W_xr) + torch.matmul(H, W_hr) + b_r)
        H_t = torch.tanh(torch.matmul(X, W_xh) + torch.matmul(R * H, W_hh) + b_h)
        H = Z * H + (1 - Z) * H_t
        Y = torch.matmul(H, W_hq) + b_q
        outputs.append(Y)
    return outputs, (H,)
```

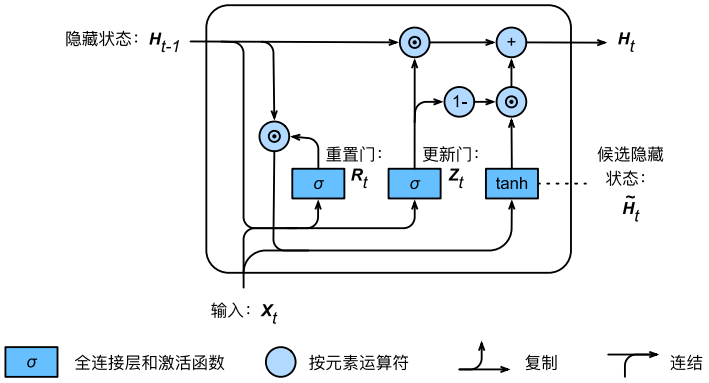

Figure 7.7: 计算隐藏状态。

7.9 长短时记忆（LSTM）

长短时记忆（long short-term memory，LSTM）中引入了 3 个门，即输入门（input gate）、遗忘门（forget gate）和输出门（output gate），以及与隐藏状态形状相同的记忆细胞（某些文献把记忆细胞当成一种特殊的隐藏状态），从而记录额外的信息。、

与 GRU 一样，LSTM 的遗忘门、输入门和输出门的输入均为当前时间步的输入 X_t 和上一时间步的隐藏状态 H_{t-1}，输出由激活函数为 sigmoid 函数的全连接层计算得到。三个门的输出均在 0 到

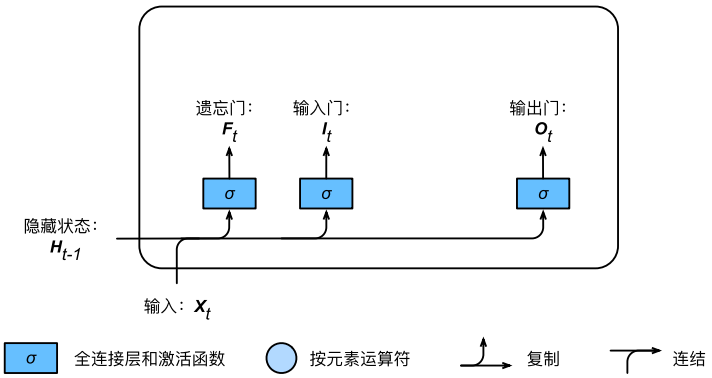

Figure 7.8: 输入门、遗忘门和输出门。

1 之间。计算表达式如下：

$$I_t = \sigma\Big(X_t W_{xi} + H_{t-1} W_{hi} + \boldsymbol{b}_i\Big) \in \mathbb{R}^{n\times h}$$

$$F_t = \sigma\Big(X_t W_{xf} + H_{t-1} W_{hf} + \boldsymbol{b}_f\Big) \in \mathbb{R}^{n\times h}$$

$$O_t = \sigma\Big(X_t W_{xo} + H_{t-1} W_{ho} + \boldsymbol{b}_o\Big) \in \mathbb{R}^{n\times h},$$

其中 $W_{xi}, W_{xf}, W_{xo} \in \mathbb{R}^{d\times h}$ 和 $W_{hi}, W_{hf}, W_{ho} \in \mathbb{R}^{h\times h}$ 为权重参数，$\boldsymbol{b}_i, \boldsymbol{b}_f, \boldsymbol{b}_o \in \mathbb{R}^{1\times h}$ 为偏置。

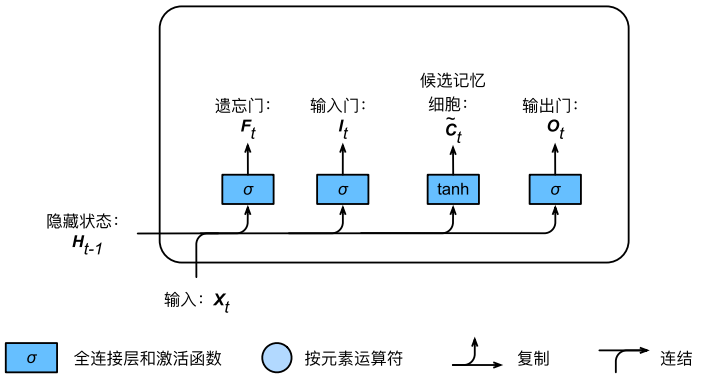

Figure 7.9: 计算候选记忆细胞。

计算候选记忆细胞 $\tilde{C}_t \in \mathbb{R}^{n\times h}$：采用 tanh 作为激活函数，有

$$\tilde{C}_t = \tanh\Big(X_t W_{xc} + H_{t-1} W_{hc} + \boldsymbol{b}_c\Big) \in \mathbb{R}^{n\times h},$$

其中 $W_{xc} \in \mathbb{R}^{d\times h}$ 和 $W_{hc} \in \mathbb{R}^{h\times h}$ 为权重参数，$\boldsymbol{b}_c \in \mathbb{R}^{1\times h}$ 为偏置。

计算记忆细胞 $C_t \in \mathbb{R}^{n\times h}$：我们可以通过元素值域在 $[0, 1]$ 的输入门、遗忘门和输出门来控制隐藏状态中信息的流动，这一般也是通过使用按元素乘法来实现的。当前时间步记忆细胞 C_t 的计算组合了上一时间步记忆细胞和当前时间步候选记忆细胞的信息，并通过遗忘门和输入门来控制信息的流动：

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t.$$

遗忘门控制上一时间步的记忆细胞 C_{t-1} 中的信息是否传递到当前时间步，而输入门则通过候选记忆细胞 $\tilde{C}_t$ 控制当前时间步的输入 X_t 如何流入当前时间步的记忆细胞。如果遗忘门一直近似 1 且

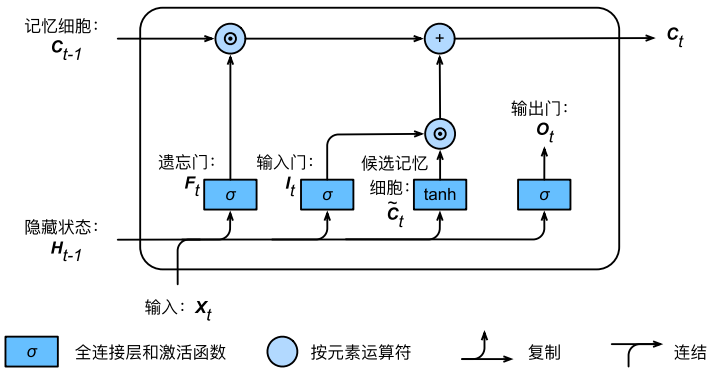

Figure 7.10: 计算记忆细胞。

输入门一直近似 0，过去的记忆细胞将一直通过时间保存并传递至当前时间步。这个设计可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时间序列中时间步距离较大的依赖关系。

计算隐藏状态 $H_t \in \mathbb{R}^{n\times h}$：输出门来控制从记忆细胞到隐藏状态 H_t 的信息的流动：

$$H_t = O_t \odot \tanh(C_t).$$

这里的 tanh 函数确保隐藏状态元素值在-1 到 1 之间。需要注意的是，当输出门近似 1 时，记忆细胞信息将传递到隐藏状态供输出层使用；当输出门近似 0 时，记忆细胞信息只自己保留。

最后，输出变量和 GRU 一致，不再赘述。

```
# 定义LSTM模型
def lstm(inputs, state, params):
    [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c, W_hq,
        b_q] = params
    (H, C) = state
    outputs = []
    for X in inputs:
        I = torch.sigmoid(torch.matmul(X, W_xi) + torch.matmul(H, W_hi) + b_i)
        F = torch.sigmoid(torch.matmul(X, W_xf) + torch.matmul(H, W_hf) + b_f)
        O = torch.sigmoid(torch.matmul(X, W_xo) + torch.matmul(H, W_ho) + b_o)
        C_t = torch.tanh(torch.matmul(X, W_xc) + torch.matmul(H, W_hc) + b_c)
        C = F * C + I * C_t
        H = O * C.tanh()
        Y = torch.matmul(H, W_hq) + b_q
        outputs.append(Y)
    return outputs, (H, C)
```

7.10 深度循环神经网络

本章到目前为止介绍的循环神经网络只有一个单向的隐藏层，在深度学习应用里，我们通常会用到含有多个隐藏层的循环神经网络，也称作深度循环神经网络。图7.11（同图7.3）演示了一个有 L 个隐藏层的深度循环神经网络，每个隐藏状态不断传递至当前层的下一时间步和当前时间步的下一层。

在时间步 t，设小批量输入为 $X_t \in \mathbb{R}^{n\times d}$，第 l 个隐藏层 $(l = 1, ..., L)$ 的隐藏状态为 $H_t^{(l)} \in \mathbb{R}^{n\times h}$，输出变量为 $O_t \in \mathbb{R}^{n\times q}$，且隐藏层的激活函数为 ϕ。

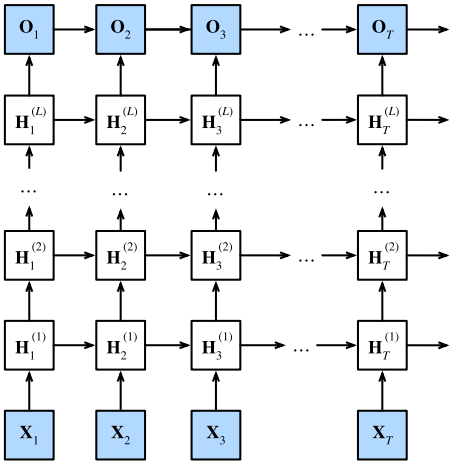

Figure 7.11: 深度循环神经网络。

则第 1 个隐藏层的隐藏状态的计算和之前一样：

$$H_t^{(1)} = \phi(X_t W_{xh}^{(1)} + H_{t-1}^{(1)} W_{hh}^{(1)} + \boldsymbol{b}_h^{(1)}).$$

对于后续隐藏层，

$$H_t^{(l)} = \phi(H_t^{(l-1)} W_{xh}^{(l)} + H_{t-1}^{(l)} W_{hh}^{(l)} + \boldsymbol{b}_h^{(l)}).$$

注意 $W_{xh}^{(1)} \in \mathbb{R}^{d\times h}$，$\forall l = 2, ..., L : W_{xh}^{(l)} \in \mathbb{R}^{h\times h}$。

输出层变量仅依赖于最后一层隐藏状态：

$$O_t = H_t^{(L)} W_{hq} + \boldsymbol{b}_q.$$

7.11 双向循环神经网络

之前介绍的循环神经网络模型都是假设当前时间步是由前面的较早时间步的序列决定的，因此它们都将信息通过隐藏状态从前往后传递。有时候，当前时间步也可能由后面时间步决定。例如，当我们写下一个句子时，可能会根据句子后面的词来修改句子前面的用词。双向循环神经网络通过增加从后往前传递信息的隐藏层来更灵活地处理这类信息。图7.12演示了一个含单隐藏层的双向循环神经网络的架构：

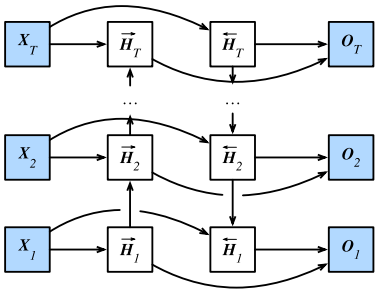

Figure 7.12: 双向循环神经网络。

给定时间步 t 的小批量输入 $X_t \in \mathbb{R}^{n\times d}$ 和隐藏层激活函数 ϕ。设时间步正向隐藏状态为 $\overrightarrow{H}_t \in \mathbb{R}^{n\times h}$，反向隐藏状态为 $\overleftarrow{H}_t \in \mathbb{R}^{n\times h}$，分别按如下方式计算：

$$\overrightarrow{H}_t = \phi\Big(X_t W_{xh}^{(f)} + \overrightarrow{H}_{t-1} W_{hh}^{(f)} + \boldsymbol{b}_h^{(f)}\Big)$$

$$\overleftarrow{H}_t = \phi\Big(X_t W_{xh}^{(b)} + \overleftarrow{H}_{t+1} W_{hh}^{(b)} + \boldsymbol{b}_h^{(b)}\Big).$$

两个方向上的隐藏单元个数可以不同。

连接两个方向的隐藏状态 $\overrightarrow{H}_t$ 和 $\overleftarrow{H}_t$ 得到 $H_t \in \mathbb{R}^{n\times 2h}$，传递给输出层：

$$O_t = H_t W_{hq} + \boldsymbol{b}_q \in \mathbb{R}^{n\times q}.$$

8 优化算法

优化算法的目标函数通常是一个基于训练数据集的损失函数，优化的目标在于降低训练误差。而深度学习的目标在于降低泛化误差。为了降低泛化误差，除了使用优化算法降低训练误差以外，还需要注意应对过拟合。在接下来的内容中，我们只关注优化算法在最小化目标函数上的表现，而不关注模型的泛化误差。

深度学习中绝大多数目标函数都很复杂。因此，很多优化问题并不存在解析解，而需要使用基于数值方法的优化算法找到近似解，即数值解。为了求得最小化目标函数的数值解，我们将通过优化算法有限次迭代模型参数来尽可能降低损失函数的值。

其中的两大挑战是**局部最小值**和**鞍点**。

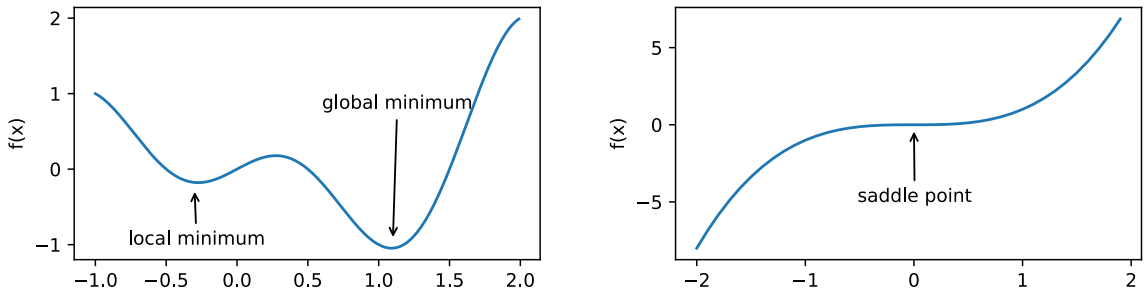

Figure 8.1: 左：局部极小值示例；右：鞍点示例。

图8.2给出了三维空间中的鞍点示例。在该示例中，目标函数在 x 轴方向上是局部最小值，但在 y 轴方向上是局部最大值。

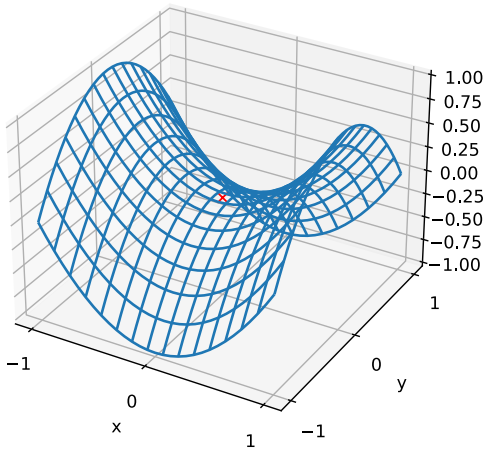

Figure 8.2: 三维空间中的鞍点示例。

假设一个函数的输入为 k 维向量，输出为标量，那么它的海森矩阵（Hessian matrix）有 k 个特征值。该函数在梯度为 0 的位置上可能是局部最小值、局部最大值或者鞍点。

- 当函数的海森矩阵在梯度为零的位置上的特征值全为正时，该函数得到局部最小值。
- 当函数的海森矩阵在梯度为零的位置上的特征值全为负时，该函数得到局部最大值。
- 当函数的海森矩阵在梯度为零的位置上的特征值有正有负时，该函数得到鞍点。

随机矩阵理论表明，**对于一个大的高斯随机矩阵来说，任一特征值是正或者是负的概率都是** 0.5。由于深度学习模型参数通常都是高维的，所以目标函数的鞍点通常比局部最小值更常见。

接下来将依次介绍梯度下降系列算法以及在其上的各种改进方法。

8.1 梯度下降方法

虽然梯度下降在深度学习中很少被直接使用，但理解梯度的意义以及沿着梯度反方向更新自变量可能降低目标函数值的原因是学习后续优化算法的基础。随后，我们将引出随机梯度下降（stochastic gradient descent）。

8.1.1 一维梯度下降

设函数 $f:\mathbb{R}\to\mathbb{R}$ 是一个连续可导的函数，将其在 x 处泰勒展开：

$$f(x+\epsilon)\approx f(x)+\epsilon f'(x),$$

其中 ϵ 是一个很小的数。选取 $\epsilon=-|\eta f'(x)|$，其中 η 是一个很小的正常数，则

$$f\big(x-\eta f'(x)\big)\approx f(x)-\eta f'(x)^2.$$

若 $f'(x)\neq 0$，则

$$f\big(x-\eta f'(x)\big)\lesssim f(x),$$

这意味着可以通过

$$x\leftarrow x-\eta f'(x)$$

来迭代 x，从而使得函数值降低。

8.1.2 多维梯度下降

设 $f:\mathbb{R}^d\to\mathbb{R}$ 是一个连续可导的函数，则任意点 $\boldsymbol{x}$ 上的梯度为

$$\nabla f(\boldsymbol{x})=\Big[\frac{\partial f(\boldsymbol{x})}{\partial x_1},...,\frac{\partial f(\boldsymbol{x})}{\partial x_d}\Big]^\top\in\mathbb{R}^d.$$

梯度中每个偏导数 $\frac{\partial f(\boldsymbol{x})}{\partial x_i}$ 代表 f 在 $\boldsymbol{x}$ 有关输入 x_i 的变化率。定义 f 在 $\boldsymbol{x}$ 沿着单位方向 $\boldsymbol{u}$ 的方向导数为

$$D_{\boldsymbol{u}}f(\boldsymbol{x})\triangleq\lim_{h\to 0}\frac{f(\boldsymbol{x}+h\boldsymbol{u})-f(\boldsymbol{x})}{h}=\nabla f(\boldsymbol{x})\cdot\boldsymbol{u}=\|\nabla f(\boldsymbol{x})\|\cdot\cos(\theta),$$

其中 θ 为梯度和 $\boldsymbol{u}$ 之间的夹角。

方向导数 $D_{\boldsymbol{u}}f(\boldsymbol{x})$ 给出了 f 在 $\boldsymbol{x}$ 上沿着所有可能方向的变量率。当 $\theta=\pi$，即 $\boldsymbol{u}$ 在梯度方向的反方向时，方向导数 $D_{\boldsymbol{u}}f(\boldsymbol{x})$ 区最小值，这就是 f 被降低最快的方向。这意味着可以通过

$$\boldsymbol{x}\leftarrow\boldsymbol{x}-\eta\nabla f(\boldsymbol{x})$$

来迭代 $\boldsymbol{x}$，其中 η 是人为添加的学习率。

8.1.3 随机梯度下降

设 $f_i(\boldsymbol{x})$ 是有关索引为 i 的训练数据样本的损失函数，n 是训练数据样本数，$\boldsymbol{x}$ 是模型的参数向量，那么目标函数定义为

$$f(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{x}),$$

目标在 $\boldsymbol{x}$ 处的梯度计算为

$$\nabla f(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}).$$

在随机梯度下降中，每一轮迭代随机采样一个样本索引 i 来更新模型参数 $\boldsymbol{x}$：

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \nabla f_i(\boldsymbol{x}).$$

因为 $\mathbb{E}\big[\nabla f_i(\boldsymbol{x})\big] \approx \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}) = \nabla f(\boldsymbol{x})$，所以随机梯度是对梯度的**无偏估计**。

8.1.4 小批量梯度下降

在每一次迭代中，梯度下降使用整个训练数据集来计算梯度，因此它有时也被称为批量梯度下降（batch gradient descent）。*如无特别说明，本章节后文的梯度下降均指批量梯度下降。*而随机梯度下降（stochastic gradient descent）在每次迭代中只随机采样一个样本来计算梯度。我们还可以在每轮迭代中随机均匀采样多个样本来组成一个小批量，然后使用这个小批量来计算梯度，这就是小批量梯度下降（mini-batch gradient descent）。

设目标函数 $f(\boldsymbol{x}) : \mathbb{R}^d \to \mathbb{R}$，其中 $\boldsymbol{x}$ 为待学习的参数。在迭代开始前的时间步设为 0。该时间步的自变量记为 $\boldsymbol{x}_0$，通常由随机初始化得到。在接下来的每一个时间步 $t > 0$ 中，小批量随机梯度下降随机均匀采样一个由训练数据样本索引组成的小批量 $\mathcal{B}_t$。我们可以通过重复采样（sampling with replacement）或者不重复采样（sampling without replacement）得到一个小批量中的各个样本。前者允许同一个小批量中出现重复的样本，后者则不允许如此，且更常见。对于这两者间的任一种方式，都可以使用

$$\boldsymbol{g}_t \leftarrow \nabla f_{\mathcal{B}_t}(\boldsymbol{x}_{t-1}) = \frac{1}{|\mathcal{B}_t|}\sum_{i\in\mathcal{B}_t} \nabla f_i(\boldsymbol{x}_{t-1})$$

来计算时间步 t 的小批量 $\mathcal{B}_t$ 上目标函数位于 $\boldsymbol{x}_{t-1}$ 处的梯度 $\boldsymbol{g}_t$。这里 $\mathcal{B}_t$ 代表小批量中样本的个数，是一个超参数。$\boldsymbol{g}_t$ 是对 $\nabla f(\boldsymbol{x}_{t-1})$ 的**无偏估计**。给定学习率 $\eta_t > 0$，小批量随机梯度下降对自变量的迭代如下：

$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \eta_t \boldsymbol{g}_t.$$

为什么小批量梯度下降需要衰减学习率？因为基于随机采样得到的梯度的方差在迭代过程中无法减小，在逼近局部极小值点时不可避免会震荡。相比之下，梯度下降则不需要衰减学习率。这是因为梯度下降中的 $\boldsymbol{g}_t$ 就是目标函数的真实梯度，而非无偏估计的结果。

8.2 动量法

目标函数有关自变量的梯度代表了目标函数在自变量当前位置下降最快的方向。因此，梯度下降也叫作最陡下降（steepest descent）。在每次迭代中，梯度下降根据自变量当前位置，沿着当前位置的梯度更新自变量。然而，如果自变量的迭代方向仅仅取决于自变量当前位置，如图**??**所示，会带来一些问题。

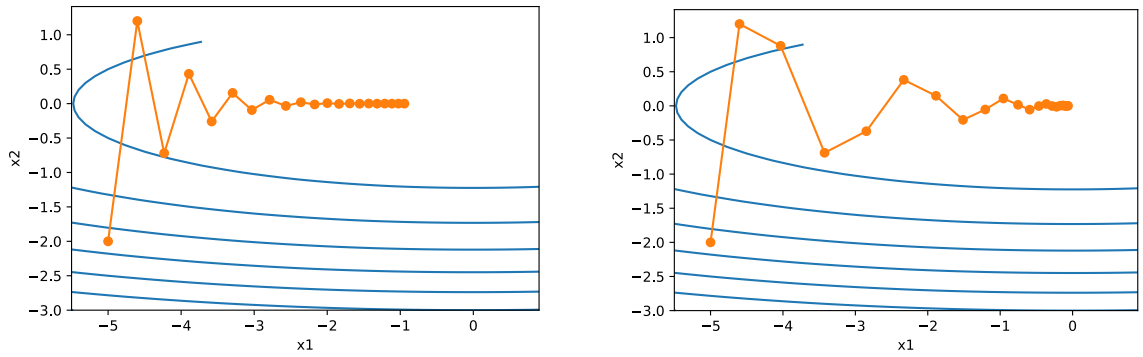

Figure 8.3: 左：梯度下降引发了如下问题——该例子中，自变量在竖直方向比在水平方向移动幅度更大。因此，需要一个较小的学习率从而避免自变量在竖直方向上越过目标函数最优解。然而，这会造成自变量在水平方向上朝最优解移动变慢。右：引入动量之后问题明显缓解。

引入动量法可解决这一问题。设 $\boldsymbol{g}_t$ 表示时间步 t 的小批量随机梯度，引入速度 $\boldsymbol{v}_t$，将模型参数 $\boldsymbol{x}_t$ 按照如下方式更新：

$$\begin{aligned}\boldsymbol{v}_t &\leftarrow \gamma \boldsymbol{v}_{t-1} + \eta_t \boldsymbol{g}_t \\ \boldsymbol{x}_t &\leftarrow \boldsymbol{x}_{t-1} - \boldsymbol{v}_t,\end{aligned}$$

其中 $\gamma \in [0,1)$，$\boldsymbol{v}_0$ 初始化为 $\boldsymbol{0}$。

接下来借助指数加权移动平均的概念来解释动量法的数学原理。

8.2.1 指数加权移动平均

给定超参数 $\gamma \in [0,1)$，当前时间步 t 的变量 y_t 是上一时间步 $t-1$ 的变量 y_{t-1} 和当前时间步的另一变量 x_t 的线性组合：

$$y_t = \gamma y_{t-1} + (1-\gamma) x_t,$$

对 y_t 按时间步展开：

$$\begin{aligned}y_t &= (1-\gamma)x_t + \gamma y_{t-1} \\ &= (1-\gamma)\left(x_t + \gamma x_{t-1} + \gamma^2 x_{t-2}\right) + \gamma^3 y_{t-3} \\ &= \ldots\end{aligned}$$

我们想忽略更高阶的项，因此需要分析 γ 的指数 n。为了和 e 建立关联，令 $n = \frac{1}{1-\gamma}$。则 $\gamma = 1 - \frac{1}{n}$，且

$$\lim_{\gamma \to 1} \gamma^n = \lim_{\gamma \to 1} \gamma^{1/(1-\gamma)} = \lim_{n \to \infty} \left(1 - \frac{1}{n}\right)^n = \exp(-1) \approx 0.3679.$$

我们将 $\exp(-1)$（即 $\lim_{\gamma \to 1} \gamma^{1/(1-\gamma)}$ 的近似）视为一个比较小的数，并忽略包含 $\lim_{\gamma \to 1} \gamma^{1/(1-\gamma)}$ 在内的高阶小量，那么，以 $\gamma = 0.95$ 为例，y_t 近似为

$$y_t \approx 0.05 \sum_{i=0}^{19} 0.95^i x_{t-i} = \frac{\sum_{i=0}^{19} 0.95^i x_{t-i}}{20}.$$

因此，我们常常将 y_t 看作是对最近 $\frac{1}{1-\gamma}$ 个时间步的 x_t 值的加权平均，距离当前时间步越近权重越大。

8.2.2 理解动量法

在动量法中，速度 $\boldsymbol{v}_t$ 的更新可改写为：

$$\boldsymbol{v}_t \leftarrow \gamma \boldsymbol{v}_{t-1}(1-\gamma)\Big(\frac{\eta_t}{1-\gamma}\Big).$$

这实际上是对序列 $\{\frac{\eta_{t-i}}{(1-\gamma)}\boldsymbol{g}_{t-i} : i = 0, ..., \frac{1}{1-\gamma} - 1\}$ 做了指数加权移动平均。动量法在每个时间步的自变量更新量近似于将最近 $\frac{1}{1-\gamma}$ 个时间步的普通更新量（即学习率乘以梯度）做了指数加权移动平均后再除以 $1-\gamma$（因为又除以了 $1-\gamma$，所以其实是加权和）。这意味着自变量在各个方向上的移动幅度不仅取决当前梯度，还取决于过去的各个梯度在各个方向上是否一致。在上文的例子中，数值方向的更新量在两个方向上相互抵消，从而减缓了在竖直方向上的移动幅度。相比之下，所有梯度在水平方向上为正。因此，我们可以采用稍大的学习率，可以在解决上述问题的同时更快收敛。

8.3 AdaGrad 算法

在之前介绍过的优化算法中，目标函数自变量的每一个元素在相同时间步都使用同一个学习率来自我迭代。如果学习率过小，那么梯度较小的维度梯度更新太慢；如果学习率过大，那么容易在梯度过大的维度上发散（震荡）。与动量法不同，AdaGrad 算法根据自变量在每个维度的梯度值的大小来调整各个维度上的学习率，从而避免统一的学习率难以适应所有维度的问题。

在每一时间步 t，引入变量 $\boldsymbol{s}_t$（初始化为 $\mathbf{0}$），

$$\boldsymbol{s}_t \leftarrow \boldsymbol{s}_{t-1} + \boldsymbol{g}_t \odot \boldsymbol{g}_t,$$

并将待优化参数 $\boldsymbol{x}_t$ 中每个元素的学习率通过按元素运算重新调整一下：

$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \frac{\eta}{\sqrt{\boldsymbol{s}_t + \epsilon}} \odot \boldsymbol{g}_t,$$

其中 η 是学习率，ϵ 是为了维持数值稳定性而添加的常数，如 10^{-6}。

观察上式可发现，如果目标函数有关自变量中某个元素的偏导数一直都较大，那么该元素的学习率将下降较快；反之，如果目标函数有关自变量中某个元素的偏导数一直都较小，那么该元素的学习率将下降较慢。然而，由于 $\boldsymbol{s}_t$ 一直在累加按元素平方的梯度，自变量中每个元素的学习率在迭代过程中一直在降低（或不变）。所以，**当学习率在迭代早期降得较快且当前解依然不佳时**，AdaGrad **算法在迭代后期由于学习率过小，可能较难找到一个有用的解**。

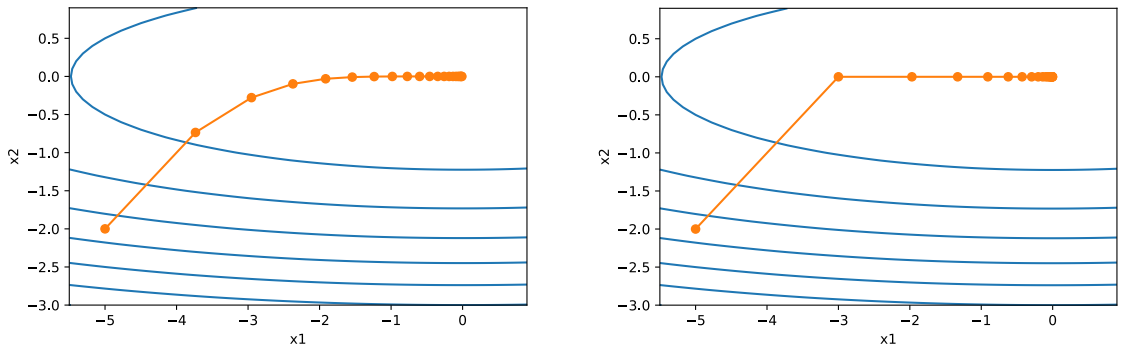

Figure 8.4: 左：验证了当学习率在迭代早期降得较快且当前解依然不佳时，AdaGrad 算法在迭代后期由于学习率过小，可能较难找到一个有用的解。右：增大学习率后自变量更为迅速地逼近了最优解。

8.4 RMSProp 算法

AdaGrad 算法的问题是，当学习率在迭代早期降得较快且当前解依然不佳时，AdaGrad 算法在迭代后期由于学习率过小，可能较难找到一个有用的解。可以对 AdaGrad 做一些调整，使得学习率

不要一直降低。

具体地，在每一时间步 t，变量 $\boldsymbol{s}_t$ 按如下方式更新：

$$\boldsymbol{s}_t \leftarrow \gamma \boldsymbol{s}_{t-1} + (1-\gamma)\boldsymbol{g}_t \odot \boldsymbol{g}_t,$$

其中 $\gamma \in [0,1)$。待优化参数 $\boldsymbol{x}_t$ 的更新方式保持不变：

$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \frac{\eta}{\sqrt{\boldsymbol{s}_t + \epsilon}} \odot \boldsymbol{g}_t,$$

其中 η 是学习率，ϵ 是为了维持数值稳定性而添加的常数，如 10^{-6}。此时 $\boldsymbol{s}_t$ 是对平方项 $\boldsymbol{g}_t \odot \boldsymbol{g}_t$ 的指数加权移动平均，因此 $\boldsymbol{s}_t$ 可能变大也可能变小，从而保证学习率不会一直减小。

8.5 AdaDelta 算法

对于 AdaGrad 算法的问题，AdaDelta 算法是另一种改进方式。

具体地，在每一时间步 t，变量 $\boldsymbol{s}_t$ 按如下方式更新：

$$\boldsymbol{s}_t \leftarrow \rho \boldsymbol{s}_{t-1} + (1-\rho)\boldsymbol{g}_t \odot \boldsymbol{g}_t,$$

其中 $\rho \in [0,1)$。AdaDelta 额外维护了一个状态变量 $\Delta\boldsymbol{x}_t$（初始化为 $\mathbf{0}$），用于代替 $\frac{\eta}{\sqrt{\boldsymbol{s}_t+\epsilon}} \odot \boldsymbol{g}_t$ 成为待学习参数 $\boldsymbol{x}_t$ 的变化量：

$$\boldsymbol{g}'_t \leftarrow \sqrt{\frac{\Delta\boldsymbol{x}_{t-1} + \epsilon}{\boldsymbol{s}_t + \epsilon}} \odot \boldsymbol{g}_t,$$

其中 ϵ 是为了维持数值稳定性而添加的常数，如 10^{-5}。所以 $\boldsymbol{x}_t$ 的更新方式为：

$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \boldsymbol{g}'_t.$$

最后，更新状态变量 $\Delta\boldsymbol{x}_t$：

$$\Delta\boldsymbol{x}_t \leftarrow \rho\Delta\boldsymbol{x}_{t-1} + (1-\rho)\boldsymbol{g}'_t \odot \boldsymbol{g}'_t.$$

本质上，AdaDelta **是用** $\sqrt{\Delta\boldsymbol{x}_{t-1}}$ **代替学习率** η。

8.6 Adam 算法

Adam 算法在 RMSProp 算法基础上对小批量随机梯度也做了指数加权移动平均，可以看做是 RMSProp 算法与动量法的结合。

给定超参数 $\beta_1 \in [0,1)$（算法作者建议设为 0.9），时间步 t 的动量变量 $\boldsymbol{v}_t$ 即小批量随机梯度 $\boldsymbol{g}_t$ 的指数加权移动平均：

$$\boldsymbol{v}_t \leftarrow \beta_1 \boldsymbol{v}_{t-1} + (1-\beta_1)\boldsymbol{g}_t.$$

再定超参数 $\beta_2 \in [0,1)$（算法作者建议设为 0.999），对 $\boldsymbol{g}_t \odot \boldsymbol{g}_t$ 做指数加权移动平均：

$$\boldsymbol{s}_t \leftarrow \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t,$$

由于我们将 $\boldsymbol{v}_0$ 和 $\boldsymbol{s}_0$ 中的元素都初始化为 0，在时间步 t 我们得到

$$\boldsymbol{v}_t = (1-\beta_1)\sum_{i=1}^{t} \beta_1^{t-i}\boldsymbol{g}_i = (1-\beta_1^t)\boldsymbol{g}_i.$$

显然，当 t 较小时，过去各时间步小批量随机梯度权值之和会较小。为了消除这样的影响，通过除以 $(1-\beta_1^t)$ 使各时间步小批量随机梯度权值之和为 1，这就是**偏差修正**。对 $\boldsymbol{v}_t$ 和 $\boldsymbol{s}_t$ 做偏差修正：

$$\begin{aligned} \hat{\boldsymbol{v}}_t &\leftarrow \frac{\boldsymbol{v}_t}{1-\beta_1^t} \\ \hat{\boldsymbol{s}}_t &\leftarrow \frac{\boldsymbol{s}_t}{1-\beta_2^t}. \end{aligned}$$

待学习参数 $\boldsymbol{x}_t$ 的变化量用修正后的变量计算：

$$\boldsymbol{g}_t' \leftarrow \frac{\eta \hat{\boldsymbol{v}}_t}{\sqrt{\hat{\boldsymbol{s}}_t} + \epsilon},$$

其中 η 是学习率，ϵ 是为了维持数值稳定性而添加的常数，如 10^{-8}。最后，更新待学习参数 $\boldsymbol{x}_t$：

$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \boldsymbol{g}_t'.$$

9 后记

这份速查清单整理自《Dive into Deep Learning》的 PyTorch 版本，图片均来自该在线文档。本清单仅包含理论模型，对应的代码实现在https://github.com/hliangzhao/Torch-Tools。

本清单仍在持续更新中。最新版本的地址为http://hliangzhao.me/math/cheatsheet.pdf。