Deep Learning Cheat Sheet

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

October 23, 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

理论分析中，向量均指列向量。

Softmax 回归是 logistic 回归（适用于二类分类）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，是一个概率分布。

1.2 单样本的矢量计算表达式

为了更方便地定义 torch tensor，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（将 $\boldsymbol{b}$ 转变为 $(\underbrace{\boldsymbol{b}; ...; \boldsymbol{b}}_{n \text{ elements}}) \in \mathbb{R}^{n\times C}$）。

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（此处不计算batch上的平均值）
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。

推导过程：

针对单样本进行分析。令 $\boldsymbol{z} = W^\top \boldsymbol{x} + \boldsymbol{b} \in \mathbb{R}^C$，则 $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{z})$，所以[1]

$$\frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}} = \text{diag}\Big(\text{softmax}(\boldsymbol{z})\Big) - \text{softmax}(\boldsymbol{z}) \cdot \Big(\text{softmax}(\boldsymbol{z})\Big)^\top.$$

其次，因为 $\boldsymbol{z} = W^\top \boldsymbol{x} + \boldsymbol{b} = \Big(\boldsymbol{w}_1^\top \boldsymbol{x}, ..., \boldsymbol{w}_C^\top \boldsymbol{x}\Big)^\top + \boldsymbol{b} \in \mathbb{R}^C$，$\boldsymbol{w}_c \in \mathbb{R}^d$，所以 $\forall c = 1, ..., C$，

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{w}_c} \in \mathbb{R}^{d \times c} = \Big(\frac{\partial \boldsymbol{w}_1^\top \boldsymbol{x}}{\partial \boldsymbol{w}_c}, ..., \frac{\partial \boldsymbol{w}_C^\top \boldsymbol{x}}{\partial \boldsymbol{w}_c}\Big)^\top = \big(\boldsymbol{0}, ..., \underbrace{\boldsymbol{x}}_{\text{the } c\text{-th col}}, ..., \boldsymbol{0}\big) \triangleq M_c(\boldsymbol{x}).$$

因为 $l(W, \boldsymbol{b}) = -\boldsymbol{y}^\top \log \hat{\boldsymbol{y}} \in \mathbb{R}$ 且其中的 $\boldsymbol{y} \in \mathbb{R}^C$ 为 one-hot 向量，$\hat{\boldsymbol{y}} \in \mathbb{R}^C$ 是 softmax 回归的输出，所以根据链式法则有

$$\begin{aligned}
\frac{\partial l(W, \boldsymbol{b})}{\partial \boldsymbol{w}_c} &= -\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{w}_c} \cdot \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{z}} \cdot \frac{\partial \log \hat{\boldsymbol{y}}}{\partial \hat{\boldsymbol{y}}} \cdot \boldsymbol{y} \\
&= -M_c(\boldsymbol{x})\Big(\text{diag}\hat{\boldsymbol{y}} - \hat{\boldsymbol{y}} \cdot \hat{\boldsymbol{y}}^\top\Big)\big(\text{diag}\hat{\boldsymbol{y}}\big)^{-1} \cdot \boldsymbol{y} \\
&= -M_c(\boldsymbol{x})\big(I - \hat{\boldsymbol{y}}\boldsymbol{1}^\top\big)\boldsymbol{y} \qquad \triangleright \quad \hat{\boldsymbol{y}}^\top\big(\text{diag}\hat{\boldsymbol{y}}\big)^{-1} = \boldsymbol{1}^\top \\
&= -M_c(\boldsymbol{x})\big(\boldsymbol{y} - \hat{\boldsymbol{y}}\boldsymbol{1}^\top\boldsymbol{y}\big) \\
&= -M_c(\boldsymbol{x})\big(\boldsymbol{y} - \hat{\boldsymbol{y}}\big) \qquad \triangleright \quad \boldsymbol{y}\text{是 one-hot 向量，所以}\boldsymbol{1}^\top\boldsymbol{y} = 1 \\
&= -\boldsymbol{x}\big(\boldsymbol{y} - \hat{\boldsymbol{y}}\big)_c \\
&= -\boldsymbol{x}\Big(\underbrace{\boldsymbol{1}(\boldsymbol{y}_c = 1) - \hat{\boldsymbol{y}}_c}_{\text{scalar}}\Big).
\end{aligned}$$

由此可得

$$\frac{\partial l(W, \boldsymbol{b})}{\partial W} = -\boldsymbol{x}\big(\boldsymbol{y} - \hat{\boldsymbol{y}}\big)^\top.$$

同理可得

$$\frac{\partial l(W, \boldsymbol{b})}{\partial \boldsymbol{b}} = -\big(\boldsymbol{y} - \hat{\boldsymbol{y}}\big)^\top.$$

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。图2.1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

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

[1]请参考文档AI 数学基础的二-14-(14)。

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。

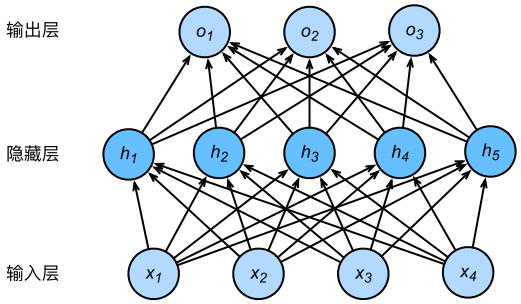

Figure 2.1: 多层感知机结构。

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**。以二维线性回归问题为例，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：

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

$\boldsymbol{w}$ 的更新公式变为

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

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

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）。

图4.1给出了 dropout 的一个示例。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。

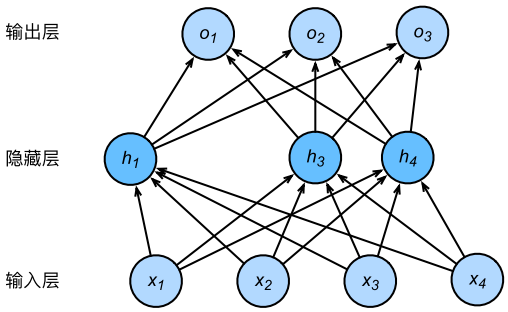

Figure 4.1: 允许 dropout 时，单隐藏层 MLP 的一种可能结构。

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：

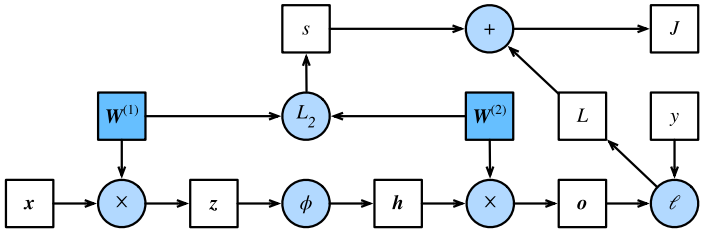

Figure 5.1: 正向传播的计算图。

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
```

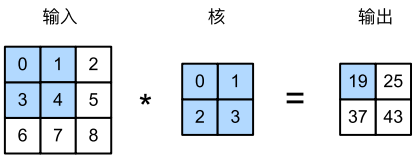

Figure 6.1: 互相关运算。

6.2 填充与步长

对于单个维度而言，设输入特征大小[2]为 n，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。

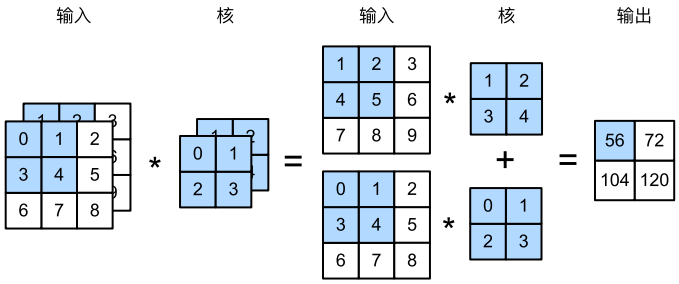

Figure 6.2: 多输入通道下的互相关运算。

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，并将它们在输出通道维上连结，因此卷积核的形状即 $c_o \times c_i \times k_h \times k_w$ 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的互相关运算得到。

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。

6.5 池化层

池化层常常跟在卷积层之后。在图6.4所示的 2×2 最大池化的例子中，只要卷积层识别的（最显著的）模式在高和宽上移动不超过一个元素，池化层就可以将它检测出来，从而缓解卷积层对位置

[2]输入特征和卷积核的宽和高可以不相同，此时输出神经元在宽和高上的数量需要分别计算。

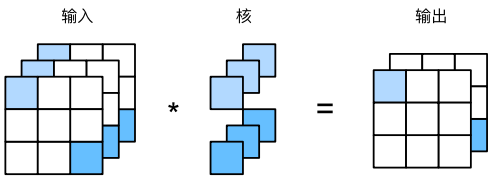

Figure 6.3: 1×1 卷积层。

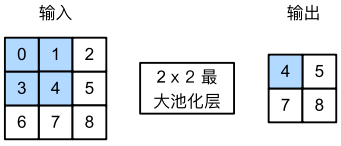

Figure 6.4: 最大池化层。

的过度敏感性。

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 (`batch_size`, `num_channels`, `width`, `height`)。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 `batch_size`，第二维是每个样本变平后的向量表示，长度为通道、高和宽的乘积。这种对 tensor 的变换可使用 `X.view(X.shape[0], -1)` 实现。

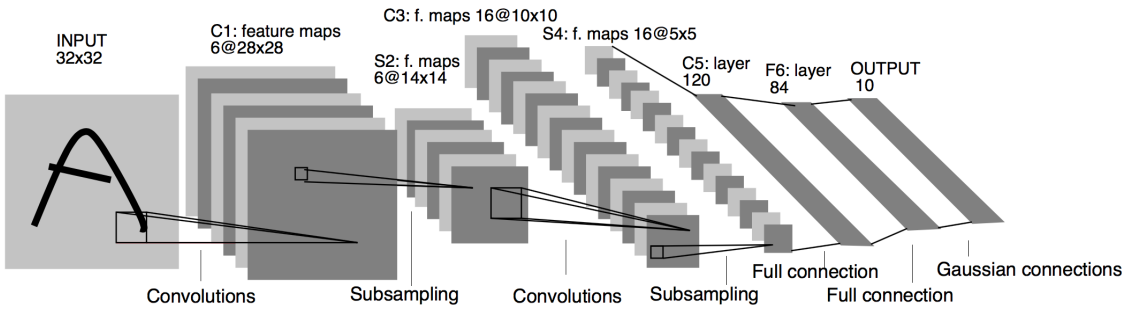

Figure 6.5: LeNet-5 的结构。

```
class LeNet5(nn.Module):
    def __init__(self):
        # input feature: (batch_size, 1, 28, 28)
        super(LeNet5, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, 6, 5),  # (batch_size, 6, 24, 24)
            nn.Sigmoid(),
            nn.MaxPool2d(2),     # (batch_size, 6, 12, 12)

            nn.Conv2d(6, 16, 5), # (batch_size, 16, 8, 8)
            nn.Sigmoid(),
            nn.MaxPool2d(2)      # (batch_size, 16, 4, 4)
        )
        self.fc = nn.Sequential(
            nn.Linear(16 * 4 * 4, 120), # (batch_size, 256)
            nn.Sigmoid(),
            nn.Linear(120, 84),         # (batch_size, 84)
            nn.Sigmoid(),
            nn.Linear(84, 10)           # (batch_size, 10)
        )

    def forward(self, img):
        feature = self.conv(img)
        return self.fc(feature.view(img.shape[0], -1))
```

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。

```
class AlexNet(nn.Module):
    """
```

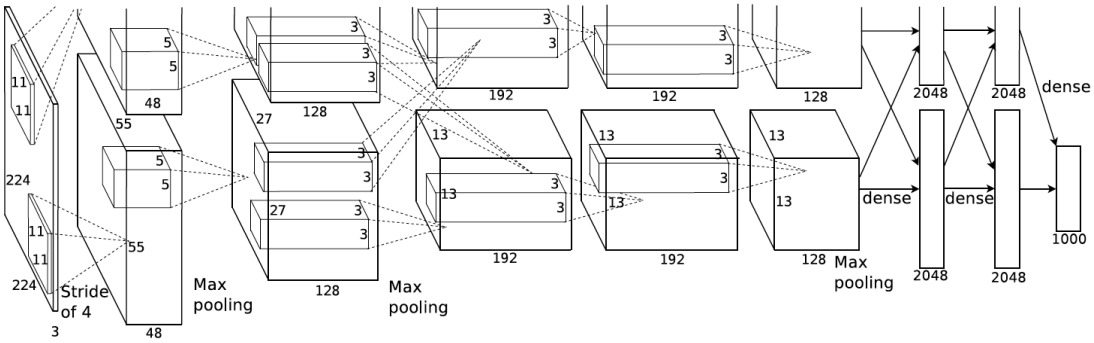

Figure 6.6: AlexNet 的结构。在最初的版本中，受限于当年 GPU 的显存，AlexNet 被拆分成两个部分，分别放到了两个 GPU 上。

```
    The images input are resize into (1, 224, 224).
    """
    def __init__(self):
        super(AlexNet, self).__init__()
        # input feature is of size (batch_size, 1, 224, 224)
        self.conv = nn.Sequential(
            nn.Conv2d(1, 96, 11, 4),       # (batch_size, 96, 54, 54)
            nn.ReLU(),
            nn.MaxPool2d(3, 2),            # (batch_size, 96, 26, 26)

            nn.Conv2d(96, 256, 5, 1, 2),   # (batch_size, 256, 26, 26)
            nn.ReLU(),
            nn.MaxPool2d(3, 2),            # (batch_size, 256, 11, 11)

            nn.Conv2d(256, 384, 3, 1, 1),  # (batch_size, 384, 11, 11)
            nn.ReLU(),
            nn.Conv2d(384, 384, 3, 1, 1),  # (batch_size, 384, 11, 11)
            nn.ReLU(),
            nn.Conv2d(384, 256, 3, 1, 1),  # (batch_size, 256, 11, 11)
            nn.ReLU(),
            nn.MaxPool2d(3, 2)             # (batch_size, 256, 5, 5)
        )
        self.fc = nn.Sequential(
            nn.Linear(256 * 5 * 5, 4096),  # (batch_size, 256 * 5 * 5)
            nn.ReLU(),
            nn.Dropout(0.5),

            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),

            nn.Linear(4096, 10)
        )

    def forward(self, img):
        feature = self.conv(img)
        return self.fc(feature.view(img.shape[0], -1))
```

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。

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

# VGG-11
def VGG11(conv_arch, fc_features, fc_hidden_units=4096, fc_out_units=10):
    net = nn.Sequential()
    for i, (num_conv, in_channels, out_channels) in enumerate(conv_arch):
        net.add_module('vgg_block_' + str(i + 1),
                       vgg_block(num_conv, in_channels, out_channels))
    net.add_module('fc', nn.Sequential(
        metrics.FlattenLayer(),
        nn.Linear(fc_features, fc_hidden_units),
        nn.Dropout(0.5),
        nn.Linear(fc_hidden_units, fc_hidden_units),
        nn.Dropout(0.5),
        nn.Linear(fc_hidden_units, fc_out_units)
    ))
    return net
```

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。

6.9 Network in Network (NiN)

卷积层的输入和输出通常是四维数组 (`batch_size`, `num_channels`, `width`, `height`)，而全连接层的输入和输出则通常是二维数组 (`batch_size`, `num_features`)。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**。

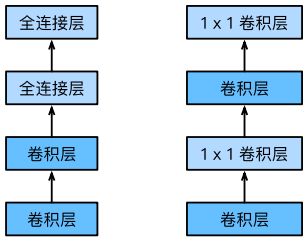

Figure 6.7: NiN 块的结构。

```
# 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。

```
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:])
```

```
# NiN
def NiN():
    # input feature is of size (batch_size, 1, 224, 224)
    return nn.Sequential(
        nin_block(1, 96, kernel_size=11, stride=4, padding=0),
        nn.MaxPool2d(3, 2),

        nin_block(96, 256, kernel_size=5, stride=1, padding=2),
        nn.MaxPool2d(3, 2),

        nin_block(256, 384, kernel_size=3, stride=1, padding=1),
```

```
        nn.MaxPool2d(3, 2),

        nn.Dropout(0.5),
        nin_block(384, 10, kernel_size=3, stride=1, padding=1), # (batch_size, 10,
            5, 5)
        GlobalAvgPool2d(),                          # (batch_size, 10, 1, 1)
        FlattenLayer()                              # (batch_size, 10)
    )
```

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。

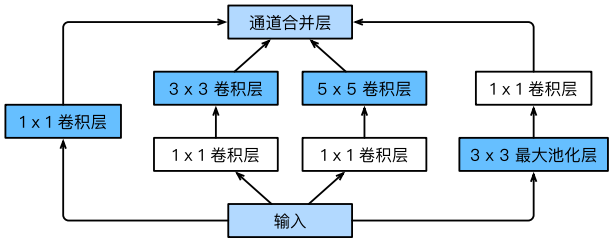

Figure 6.8: Inception 块的结构。

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。实际上，**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

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

其中 $\boldsymbol{u}$ 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

在图6.9右图所示的残差块中，虚线框内要学习的是残差映射 $f(\boldsymbol{x})-\boldsymbol{x}$，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。

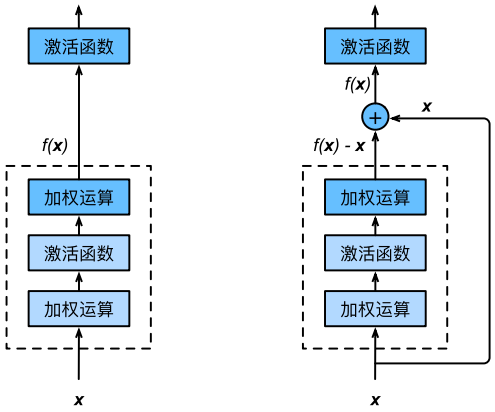

Figure 6.9: 残差网络的基本结构（右）。

```
# 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 在卷积层后增加了批量归一化层。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。

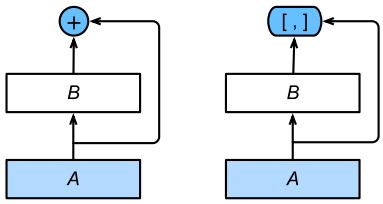

Figure 6.10: DenseNet 的基本结构：稠密块（dense block）和过渡层（transition layer）。

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 $W_{xh} \in \mathbb{R}^{d\times h}, W_{hh} \in \mathbb{R}^{h\times h}, \boldsymbol{b}_h \in \mathbb{R}^{1\times h}, W_{hq} \in \mathbb{R}^{h\times q}, \boldsymbol{b}_q \in \mathbb{R}^{1\times q}$。

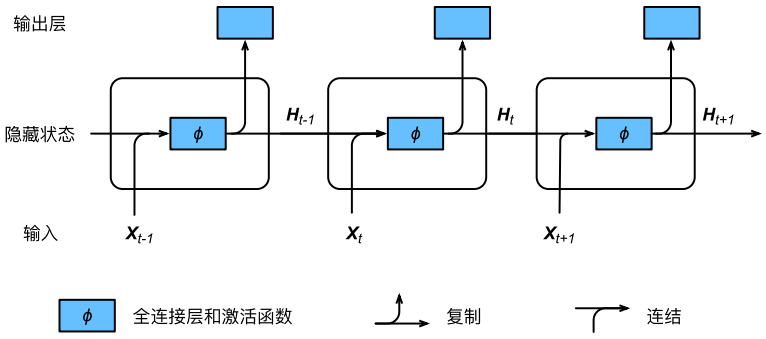

Figure 7.1: 包含单层隐藏状态的循环神经网络的结构。

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 $\boldsymbol{b}_h$。

```
# 定义RNN模型
def rnn(inputs, hidden_state, params):
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = hidden_state
    outputs = []
    for X in inputs:      # iteration over num_steps
        H = torch.tanh(torch.matmul(X, W_xh) + torch.matmul(H, W_hh) + b_h)
        Y = torch.matmul(H, W_hq) + b_q
        outputs.append(Y)
    return outputs, (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。

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。

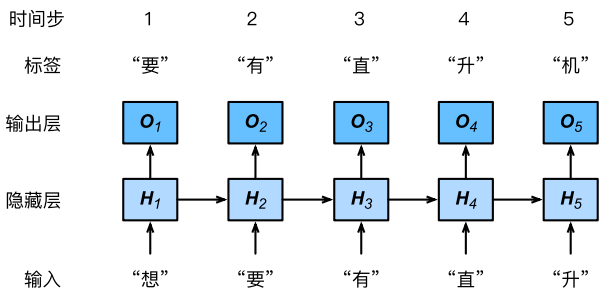

Figure 7.2: 基于字符级循环神经网络创建的语言模型。

- **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: (batch_size, num_steps)
        # 经过独热编码之后，得到 (num_steps, batch_size, vocab_size)
        X = to_onehot(inputs, self.vocab_size)
        Y, self.state = self.rnn(torch.stack(X), state)
        # change Y's 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。

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。

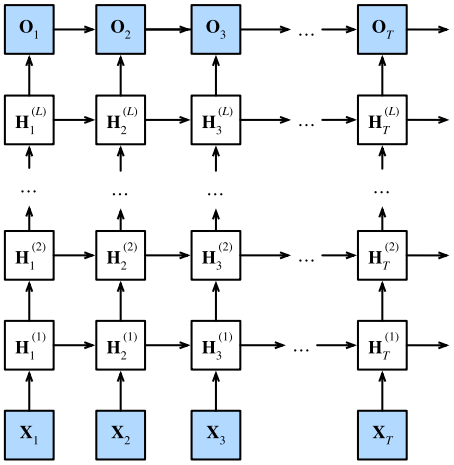

Figure 7.3: 深度循环神经网络的输入、输出与隐藏状态。

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。

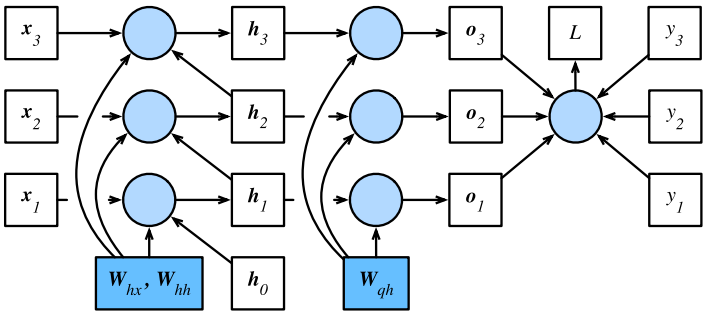

Figure 7.4: 含有单隐藏层的 RNN 的模型计算图。

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：

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

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

可以发现，$\frac{\partial L}{\partial \boldsymbol{h}_t}$ 用到了权重 W_{hh} 的指数运算。如果 W_{hh} 很大，就会发生梯度爆炸的现象。因此裁剪梯度在 RNN 的训练中是十分有必要的。

7.8 门控逻辑单元（GRU）

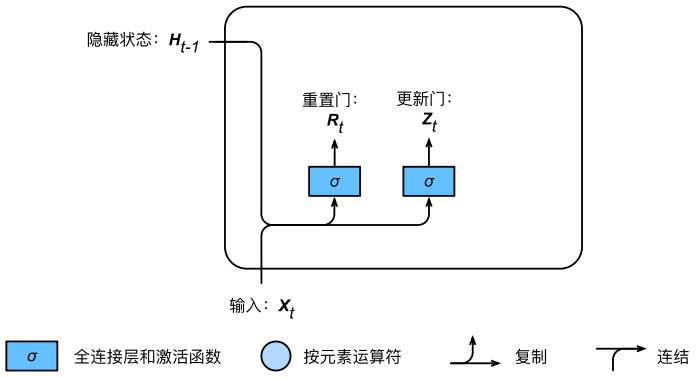

Figure 7.5: GRU：重置门和更新门。

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:

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

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

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。

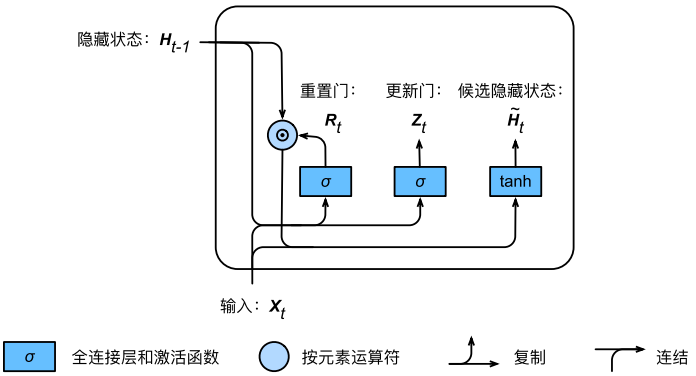

Figure 7.6: GRU：计算候选隐藏状态。

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。

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,)
```

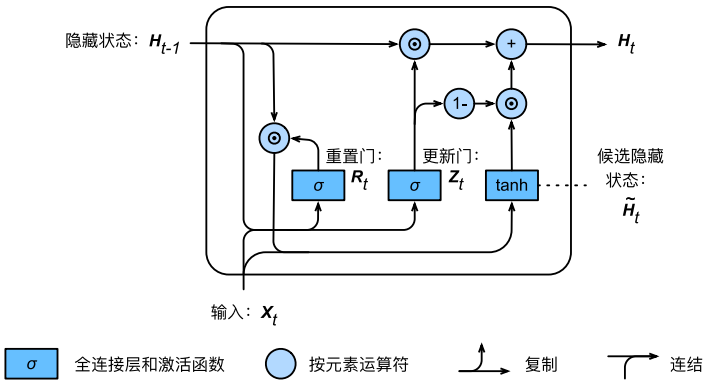

Figure 7.7: GRU：计算隐藏状态。

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 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。

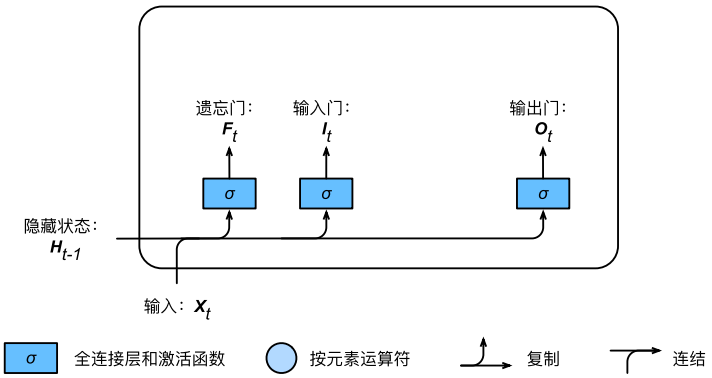

Figure 7.8: LSTM：输入门、遗忘门和输出门。

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。

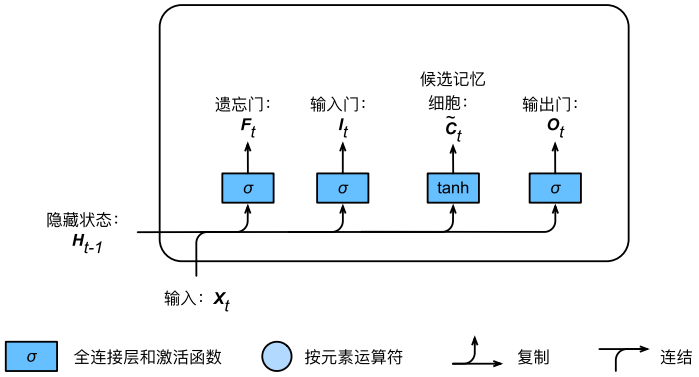

Figure 7.9: LSTM：计算候选记忆细胞。

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，输出变量的计算表达式和 GRU 一致，不再赘述。

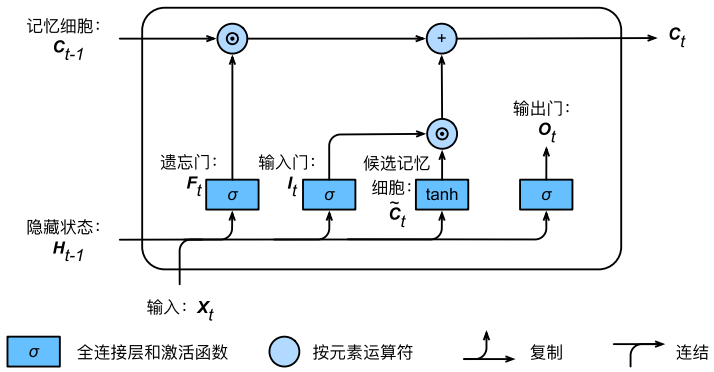

Figure 7.10: LSTM：计算记忆细胞。

```
# 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。

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

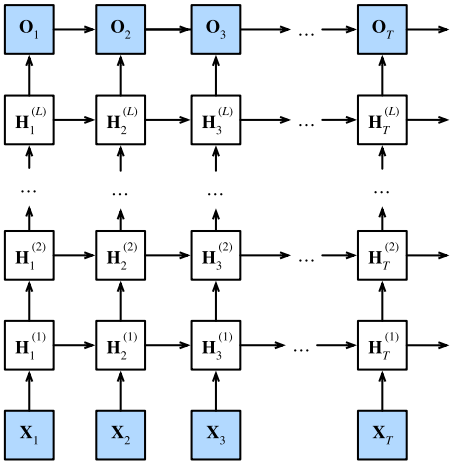

Figure 7.11: 深度循环神经网络。

输出层变量仅依赖于最后一层隐藏状态：

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

模型最后会返回输出层变量和每一个隐藏层的隐藏变量。

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。

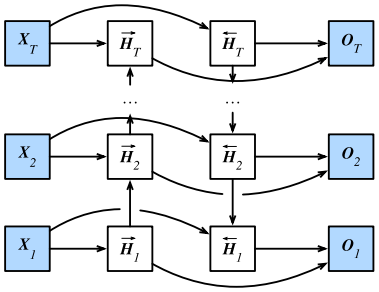

Figure 7.12: 双向循环神经网络。

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：

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

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

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**。

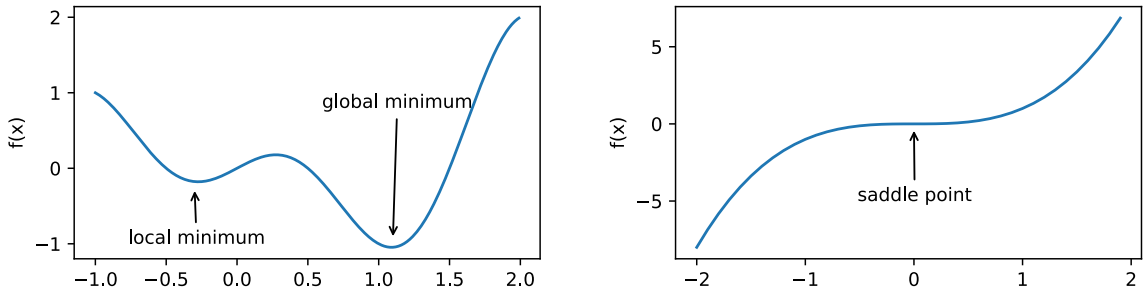

Figure 8.1: 左：局部极小值示例；右：鞍点示例。

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。

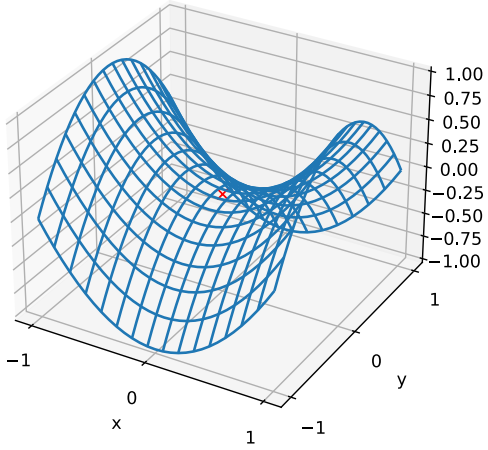

Figure 8.2: 三维空间中的鞍点示例。

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（gradient descent）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）和小批量梯度下降（mini-batch 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。

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8.3所示，会带来一些问题。

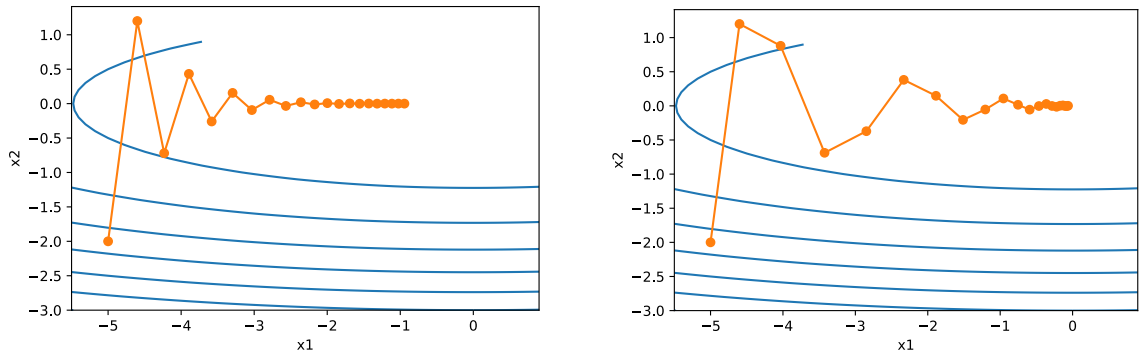

Figure 8.3: 左：梯度下降引发了如下问题——该例子中，自变量在竖直方向比在水平方向移动幅度更大。因此，需要一个较小的学习率从而避免自变量在竖直方向上越过目标函数最优解。然而，这会造成自变量在水平方向上朝最优解移动变慢。右：引入动量之后问题明显缓解。

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 $\mathbf{0}$。

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：

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

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

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

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，可以在解决梯度下降的问题的同时更快收敛。

8.3 AdaGrad 算法

在梯度下降中，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 $\boldsymbol{0}$），

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

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。因此，对于图8.3所示的问题，AdaGrad 可以保证竖直方向的学习率较小，从而减缓竖直方向上的震荡现象。

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**。这个问题的解决办法是：增大学习率的初值。图8.4给出了图示。

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。虽然最简单的方式是增大学习率的初值，但是并未根除这个问题。造成这个问题的根本原因是学习率一直在减小。因此，可以对 AdaGrad 做一些调整，使得学习率不要一直降低。这就是 RMSProp 的做法。

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

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

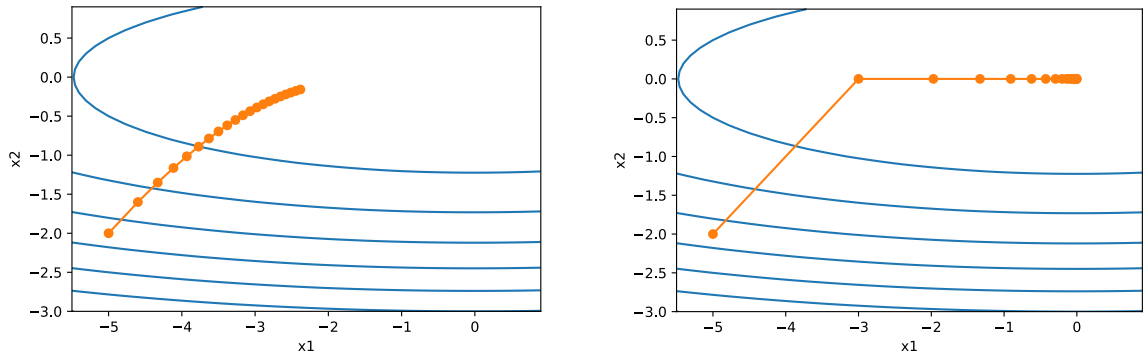

Figure 8.4: 左：当学习率在迭代早期降得较快且当前解依然不佳时，AdaGrad 算法在迭代后期由于学习率过小，可能较难找到一个有用的解。右：增大学习率后自变量更为迅速地逼近了最优解。

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。

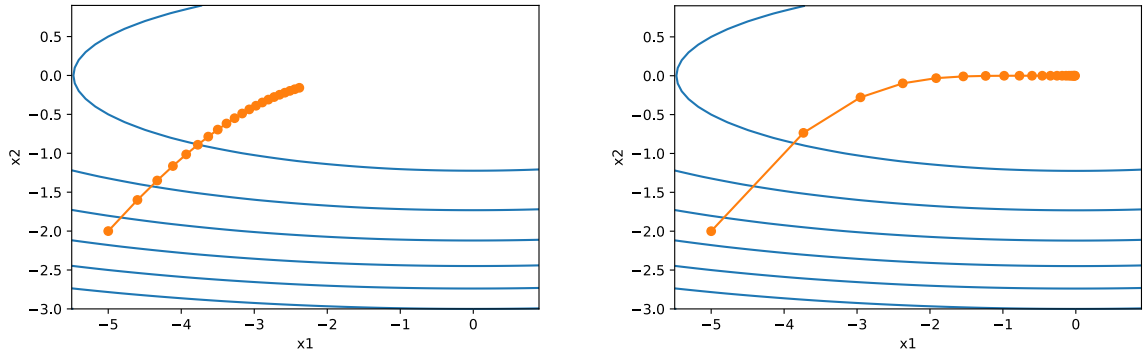

Figure 8.5: 左：当学习率在迭代早期降得较快且当前解依然不佳时，AdaGrad 算法在迭代后期由于学习率过小，可能较难找到一个有用的解。右：RMSProp 解决了这一问题。

8.5 AdaDelta 算法

对于 AdaGrad 算法的问题，与 RMSProp 不同，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，相当于用过去的梯度的变化状态来学习一个较为合适的学习率。

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：

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

再给定超参数 $\beta_2 \in [0,1)$（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 自然语言处理

9.1 词嵌入（word2vec）

对于词典中的单词而言，one-hot 编码方式无法衡量不同单词之间的相似度（任何两个不同词的 one-hot 向量的余弦相似度 $\frac{\boldsymbol{x}^\top \boldsymbol{y}}{\|\boldsymbol{x}\|\cdot\|\boldsymbol{y}\|}$ 都为 0）。

word2vec 工具的提出正是为了解决这个问题。它将每个词表示成一个定长的向量，并使得这些向量能较好地表达不同词之间的相似和类比关系。word2vec 工具包含了两个模型：跳字模型（skip-gram）和连续词袋模型（continuous bag of words，CBOW）。

9.1.1 跳字模型（skip-gram）

跳字模型假设基于某个词（定义为中心词）来生成它在文本序列周围的词（定义为背景词）。

例如，假设文本序列是 “the” “man” “loves” “his” “son”。以 “loves” 作为中心词，设背景窗口大小为 2，我们想要计算的是与它距离不超过 2 个词的背景词 “the” “man” “his” “son” 在 “loves” 出现的条件下出现的概率。

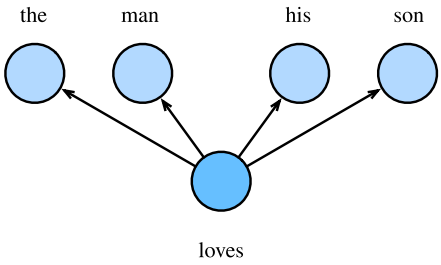

Figure 9.1: 跳字模型。

如何计算这一概率？将每个词被表示成两个 d 维向量，假设这个词在词典中索引为 i，当它为中心词时向量表示为 $\boldsymbol{v}_i \in \mathbb{R}^d$，为背景词时向量表示为 $\boldsymbol{u}_i \in \mathbb{R}^d$。设中心词 w_c 在词典中索引为 c，背景词 w_o 在词典中索引为 o，给定中心词生成背景词的条件概率可以通过对向量内积做 softmax 运算而得到：

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\boldsymbol{u}_o^\top \boldsymbol{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)},$$

其中 $\mathcal{V} \triangleq \{0, 1, ..., |\mathcal{V}| - 1\}$ 为词典中所有单词的索引的集合。

给定一个长度为 T 的文本序列，设时间步 t 的词为 $w^{(t)}$。假设给定中心词的情况下背景词的生成相互独立，当背景窗口大小为 m 的时候，跳字模型的似然函数，即给定**任意**中心词生成**所有**背景词的概率，为

$$\prod_{t=1}^{T} \prod_{-m \le j \le m, j \ne 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)}).$$

自动忽略大于 T 和小于 1 的时间步。

训练跳字模型：

跳字模型的参数是每个词所对应的中心词向量和背景词向量，可以通过最大化似然来训练参数。具体地，最大化上文中的似然函数等价于最小化以下损失函数：

$$-\sum_{t=1}^{T} \sum_{-m \le j \le m,\ j \ne 0} \log \mathbb{P}(w^{(t+j)} \mid w^{(t)}).$$

如果使用随机梯度下降，那么在每一次迭代中随机采样一个较短的子序列来计算有关该子序列的损失，然后计算梯度来更新模型参数。对于所有属于以 w_c 为中心词的窗口内的背景词 w_o，

$$\log \mathbb{P}(w_o \mid w_c) = \boldsymbol{u}_o^\top \boldsymbol{v}_c - \log\left(\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)\right).$$

所以

$$\begin{aligned}
\frac{\partial \log \mathbb{P}(w_o \mid w_c)}{\partial \boldsymbol{v}_c} &= \boldsymbol{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\boldsymbol{u}_j^\top \boldsymbol{v}_c) \boldsymbol{u}_j}{\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)} \\
&= \boldsymbol{u}_o - \sum_{j \in \mathcal{V}} \left(\frac{\exp(\boldsymbol{u}_j^\top \boldsymbol{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)}\right) \boldsymbol{u}_j \\
&= \boldsymbol{u}_o - \sum_{j \in \mathcal{V}} \mathbb{P}(w_j \mid w_c) \boldsymbol{u}_j.
\end{aligned}$$

它的计算需要词典中所有词以 w_c 为中心词的条件概率。同样地，对于所有属于以 w_c 为中心词的窗口内的背景词 w_o 的背景词向量 $\boldsymbol{u}_o$，

$$\frac{\partial \log \mathbb{P}(w_o \mid w_c)}{\partial \boldsymbol{u}_o} = \boldsymbol{v}_c - \mathbb{P}(w_o \mid w_c)\boldsymbol{v}_c.$$

训练结束后，对于词典中的任一索引为 i 的词，我们均得到该词作为中心词和背景词的两组词向量 $\boldsymbol{v}_i$ 和 $\boldsymbol{u}_i$。**一般使用跳字模型的中心词向量作为词的表征向量参与后续的操作**（如判定单词相似度等）。

9.1.2 连续词袋模型（CBOW）

连续词袋模型假设基于某中心词在文本序列前后的背景词来生成该中心词。在同样的文本序列“the”“man”“loves”“his”“son”里，以“loves”作为中心词，且背景窗口大小为 2 时，连续词袋模型关心的是，给定背景词“the”“man”“his”“son”生成中心词“loves”的条件概率。

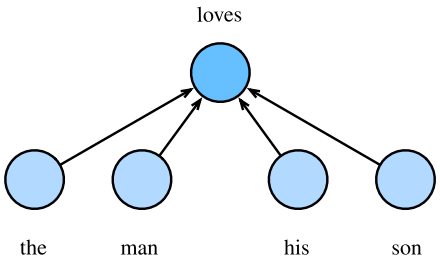

Figure 9.2: 连续词袋模型。

因为连续词袋模型的背景词有多个，我们将这些背景词向量取平均，然后使用和跳字模型一样的方法来计算条件概率。设 $\boldsymbol{v_i} \in \mathbb{R}^d$ 和 $\boldsymbol{u_i} \in \mathbb{R}^d$ 分别表示词典中索引为 i 的词作为背景词和中心词的向量（注意符号的含义与跳字模型中的相反）。设中心词 w_c 在词典中索引为 c，背景词 $w_{o_1}, \ldots, w_{o_{2m}}$ 在词典中索引分别为 $o_1, \ldots, o_{2m}$，那么给定背景词生成中心词的条件概率

$$\mathbb{P}(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\boldsymbol{u}_c^\top (\boldsymbol{v}_{o_1} + \ldots + \boldsymbol{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\boldsymbol{u}_i^\top (\boldsymbol{v}_{o_1} + \ldots + \boldsymbol{v}_{o_{2m}})\right)}.$$

定义 $\mathcal{W}_o \triangleq \{w_{o_1}, \ldots, w_{o_{2m}}\}$，$\bar{\boldsymbol{v}}_o \triangleq \frac{1}{2m}\left(\boldsymbol{v}_{o_1} + \ldots + \boldsymbol{v}_{o_{2m}}\right)$，则上式简化为

$$\mathbb{P}(w_c \mid \mathcal{W}_o) = \frac{\exp\left(\boldsymbol{u}_c^\top \bar{\boldsymbol{v}}_o\right)}{\sum_{i \in \mathcal{V}} \exp\left(\boldsymbol{u}_i^\top \bar{\boldsymbol{v}}_o\right)}.$$

给定一个长度为 T 的文本序列，设时间步 t 的词为 $w^{(t)}$，背景窗口大小为 m，连续词袋模型的似然函数是由背景词生成**任意**中心词的概率：

$$\prod_{t=1}^{T} \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

训练连续词袋模型：

连续词袋模型的最大似然估计等价于最小化损失函数

$$-\sum_{t=1}^{T} \log \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

注意到

$$\log \mathbb{P}(w_c \mid \mathcal{W}_o) = \boldsymbol{u}_c^\top \bar{\boldsymbol{v}}_o - \log\left(\sum_{i \in \mathcal{V}} \exp\left(\boldsymbol{u}_i^\top \bar{\boldsymbol{v}}_o\right)\right).$$

所以对于 $\boldsymbol{v}_{o_i}$（$i = 1, \ldots, 2m$），

$$\frac{\partial \log \mathbb{P}(w_c \mid \mathcal{W}_o)}{\partial \boldsymbol{v}_{o_i}} = \frac{1}{2m}\left(\boldsymbol{u}_c - \sum_{j\in\mathcal{V}} \frac{\exp(\boldsymbol{u}_j^\top \bar{\boldsymbol{v}}_o)\boldsymbol{u}_j}{\sum_{i\in\mathcal{V}}\exp(\boldsymbol{u}_i^\top \bar{\boldsymbol{v}}_o)}\right) = \frac{1}{2m}\left(\boldsymbol{u}_c - \sum_{j\in\mathcal{V}} P(w_j \mid \mathcal{W}_o)\boldsymbol{u}_j\right).$$

同理

$$\frac{\partial \log \mathbb{P}(w_c \mid \mathcal{W}_o)}{\partial \boldsymbol{u}_c} = \bar{\boldsymbol{v}}_o - \mathbb{P}(w_c \mid \mathcal{W}_o)\bar{\boldsymbol{v}}_o.$$

连续词袋模型中，**一般使用连续词袋模型的背景词向量作为词的表征向量**。

9.2 近似训练

跳字模型的核心在于使用 softmax 运算得到给定中心词 w_c 来生成背景词 w_o 的条件概率

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\boldsymbol{u}_o^\top \boldsymbol{v}_c)}{\sum_{i\in\mathcal{V}}\exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)}.$$

该条件概率相应的对数损失为

$$-\log \mathbb{P}(w_o \mid w_c) = -\boldsymbol{u}_o^\top \boldsymbol{v}_c + \log\left(\sum_{i\in\mathcal{V}}\exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)\right).$$

以上损失包含了词典大小数目的项的累加。对于含几十万或上百万词的较大词典，每次的梯度计算开销可能过大。为了降低该计算复杂度，常常使用两种近似训练方法：负采样（negative sampling）和层序 softmax（hierarchical softmax）。

接下来以跳字模型为例介绍这两种方法。

9.2.1 负采样（negative sampling）

负采样修改了原来的目标函数。给定中心词 w_c 的一个背景窗口，我们把背景词 w_o 出现在该背景窗口看作一个事件，并将该事件的概率计算为

$$\mathbb{P}(D = 1 \mid w_c, w_o) = \sigma(\boldsymbol{u}_o^\top \boldsymbol{v}_c),$$

其中的 σ 是 sigmoid 激活函数，二值性的随机变量 D 取值 1 表示 w_o **出现在了以** w_c **作为中心词的背景窗口中**。

先考虑最大化文本序列中所有该事件的联合概率来训练词向量。具体来说，给定一个长度为 T 的文本序列，设时间步 t 的词为 $w^{(t)}$ 且背景窗口大小为 m，考虑最大化联合概率

$$\prod_{t=1}^{T}\prod_{-m\le j\le m,\ j\neq 0} P(D = 1 \mid w^{(t)}, w^{(t+j)}).$$

以上模型中包含的事件仅考虑了正类样本，只有当所有词向量相等且值为无穷大时，以上的联合概率才被最大化为 1。很明显，这样的词向量毫无意义。负采样通过采样并添加负类样本使目标函数更有意义。具体地，设背景词 w_o 出现在中心词 w_c 的一个背景窗口为事件 P，我们根据分布 P 采样 K 个未出现在该背景窗口中的词，即噪声词。设噪声词 w_k（$k = 1, \ldots, K$）不出现在中心词 w_c 的该背景窗口为事件 N_k。假设同时含有正类样本和负类样本的事件 $P, N_1, \ldots, N_K$ 相互独立，则上文的联合概率被改写为

$$\prod_{t=1}^{T}\prod_{-m\le j\le m,\ j\neq 0} P(w^{(t+j)} \mid w^{(t)}),$$

其中

$$\mathbb{P}(w^{(t+j)} \mid w^{(t)}) = \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) \cdot \prod_{k=1,\ w_k\sim P(w)}^{K} \mathbb{P}(D = 0 \mid w^{(t)}, w_k).$$

设文本序列中时间步 t 的词 $w^{(t)}$ 在词典中的索引为 i_t，噪声词 w_k 在词典中的索引为 h_k。有关以上条件概率的对数损失（二元交叉熵损失函数）为

$$
\begin{aligned}
-\log \mathbb{P}(w^{(t+j)} \mid w^{(t)}) &= -\log \mathbb{P}(D=1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log \mathbb{P}(D=0 \mid w^{(t)}, w_k) \\
&= -\log \sigma\left(\boldsymbol{u}_{i_{t+j}}^\top \boldsymbol{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log\left(1-\sigma\left(\boldsymbol{u}_{h_k}^\top \boldsymbol{v}_{i_t}\right)\right) \\
&= -\log \sigma\left(\boldsymbol{u}_{i_{t+j}}^\top \boldsymbol{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log \sigma\left(-\boldsymbol{u}_{h_k}^\top \boldsymbol{v}_{i_t}\right).
\end{aligned}
$$

现在，训练中每一步的梯度计算开销不再与词典大小相关，而与 K 线性相关。当 K 取较小的常数时，负采样在每一步的梯度计算开销较小。简言之，**负采样通过考虑同时含有正类样本和负类样本的相互独立事件来构造损失函数，其训练中每一步的梯度计算的开销仅与采样的噪声词的个数线性相关**。

9.2.2 层序 softmax（hierarchical softmax）

层序 softmax 使用了二叉树这一数据结构，树的每个叶结点代表词典 $\mathcal{V}$ 中的每个词。

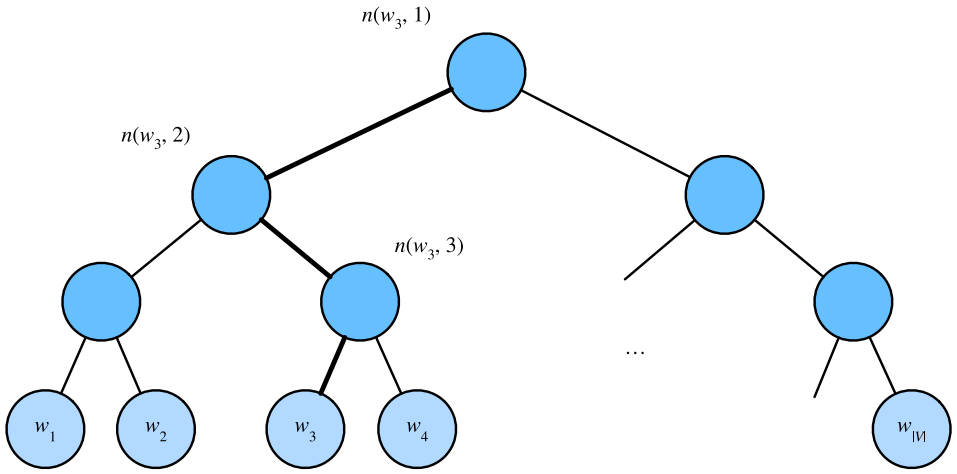

Figure 9.3: 二叉树的每个叶结点代表词典 $\mathcal{V}$ 中的每个词。

假设 $L(w)$ 为从二叉树的根结点到词 w 的叶结点的路径（包括根结点和叶结点）上的结点数。设 $n(w,j)$ 为该路径上第 j 个结点，并设该结点的背景词向量为 $\boldsymbol{u}_{n(w,j)}$。以图9.3为例，$L(w_3)=4$。层序 softmax 将跳字模型中的条件概率近似表示为

$$
P(w_o \mid w_c) = \prod_{j=1}^{L(w_o)-1} \sigma\left(\llbracket n(w_o, j+1) = \texttt{leftChild}(n(w_o,j)) \rrbracket \cdot \boldsymbol{u}_{n(w_o,j)}^\top \boldsymbol{v}_c\right),
$$

其中 σ 是 sigmoid 激活函数，$\texttt{leftChild}(n)$ 是结点 n 的左子结点，且 $\llbracket x \rrbracket = 1$ if x is `True` else -1。

以计算图9.3中给定词 w_c 生成词 w_3 的条件概率为例，需要将 w_c 的词向量 $\boldsymbol{v}_c$ 和根结点到 w_3 路径上的非叶结点向量一一求内积。由于在二叉树中由根结点到叶结点 w_3 的路径上需要向左、向右再向左地遍历（上图中加粗的路径），所以

$$
P(w_3 \mid w_c) = \sigma(\boldsymbol{u}_{n(w_3,1)}^\top \boldsymbol{v}_c) \cdot \sigma(-\boldsymbol{u}_{n(w_3,2)}^\top \boldsymbol{v}_c) \cdot \sigma(\boldsymbol{u}_{n(w_3,3)}^\top \boldsymbol{v}_c).
$$

由于 $\sigma(x)+\sigma(-x)=1$，给定中心词 w_c，生成词典 $\mathcal{V}$ 中任一词的条件概率之和为 1 这一条件也将满足 $\sum_{w\in\mathcal{V}} P(w \mid w_c)=1$（这是因为二叉树中左右子树个数相等）。

由于 $L(w_o)-1$ 的数量级为 $\mathcal{O}(\log_2|\mathcal{V}|)$，所以当词典 $\mathcal{V}$ 很大时，层序 softmax 在训练中每一步的梯度计算开销相较未使用近似训练时大幅降低。

9.3 二次采样

文本数据中一般会出现一些高频词，如英文中的“the”“a”和“in”。通常来说，在一个背景窗口中，一个词（如“chip”）和较低频词（如“microprocessor”）同时出现比和较高频词（如“the”）同时出现对训练词嵌入模型更有益。因此，训练词嵌入模型之前可以进行二次采样，即数据集中的任意被索引词 w_i 都有一定的概率会被丢弃：

$$\mathbb{P}=\max\left(1-\sqrt{\frac{t}{f(w_i)}},0\right),$$

其中 $f(w_i)$ 是单词 w_i 的个数与总词数之比，t 是一个超参数，我们设置为 10^{-4}。具体地，对于每一个词，若 $\texttt{rand}() < \mathbb{P}$，则该词被丢弃。

9.4 子词嵌入（fastText）

英语单词通常有其内部结构和形成方式。例如，可以从“dog”“dogs”和“dogcatcher”的字面上推测它们的关系。这些词都有同一个词根“dog”，但使用不同的后缀来改变词的含义。而且，这个关联可以推广至其他词汇。例如，“dog”和“dogs”的关系如同“cat”和“cats”的关系，“boy”和“boyfriend”的关系如同“girl”和“girlfriend”的关系。构词学（morphology）作为语言学的一个重要分支，研究的正是词的内部结构和形成方式。

word2vec 没有直接利用构词学中的信息。无论是在跳字模型还是连续词袋模型中，我们都将形态不同的单词用不同的向量来表示。例如，“dog”和“dogs”分别用两个不同的向量表示，而模型中并未直接表达这两个向量之间的关系。鉴于此，fastText 提出了子词嵌入（subword embedding）的方法，从而试图将构词信息引入 word2vec 中的跳字模型。

在 fastText 中，每个中心词被表示成子词的集合。以单词“where”为例，首先，我们在单词的首尾分别添加特殊字符“<”和“>”以区分作为前后缀的子词。然后，将单词当成一个由字符构成的序列来提取 n 元语法。例如，当 $n=3$ 时，我们得到所有长度为 3 的子词：“<wh>”“whe”“her”“ere”“<re>”以及特殊子词“<where>”。

在 fastText 中，对于一个词 w，我们将它所有长度在 $3\sim6$ 的子词和特殊子词的并集记为 $\mathcal{G}_w$。那么词典则是所有词的子词集合的并集。假设词典中子词 g 的向量为 $\boldsymbol{z}_g$，那么跳字模型中词 w 的作为中心词的向量 $\boldsymbol{v}_w$ 则表示成

$$\boldsymbol{v}_w=\sum_{g\in\mathcal{G}_w}\boldsymbol{z}_g.$$

fastText 的其余部分同跳字模型一致。

可以看到，与跳字模型相比，fastText 中词典规模更大，造成模型参数更多，同时一个词的向量需要对所有子词向量求和，继而导致计算复杂度更高。但与此同时，较生僻的复杂单词，甚至是词典中没有的单词，可能会从同它结构类似的其他词那里获取更好的词向量表示。

9.5 GloVe

将跳字模型中使用 softmax 运算表达的条件概率 $P(w_j \mid w_i)$ 记作 q_{ij}，即

$$q_{ij} = \frac{\exp(\boldsymbol{u}_j^\top \boldsymbol{v}_i)}{\sum_{k\in\mathcal{V}} \exp(\boldsymbol{u}_k^\top \boldsymbol{v}_i)},$$

其中 $\boldsymbol{v}_i$ 和 $\boldsymbol{u}_i$ 分别是索引为 i 的词 w_i 作为中心词和背景词时的向量表示，$\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$ 为词典索引集。

对于词 w_i，它在数据集中可能多次出现。我们将每一次以它作为中心词的所有背景词全部汇总并保留重复元素，记作多重集 $\mathcal{C}_i$（multiset）。一个元素在多重集中的个数称为该元素的重数（multiplicity）。例如，假设词 w_i 在数据集中出现 2 次：文本序列中以这两个 w_i 作为中心词的背景窗口分别包含背景词索引 $2, 1, 5, 2$ 和 $2, 3, 2, 1$。那么多重集 $\mathcal{C}_i = \{1, 1, 2, 2, 2, 2, 3, 5\}$，其中元素 1 的重数为 2，元素 2 的重数为 4，元素 3 和 5 的重数均为 1。将多重集 $\mathcal{C}_i$ 中元素 j 的重数记作 x_{ij}，它表示了整个数据集中所有以 w_i 为中心词的背景窗口中词 w_j 的个数。因此，跳字模型的损失函数还可以表达为：

$$-\sum_{i\in\mathcal{V}} \sum_{j\in\mathcal{V}} x_{ij} \log\, q_{ij}.$$

我们将数据集中所有以词 w_i 为中心词的背景词的数量之和 $|\mathcal{C}_i|$ 记为 x_i，并将以 w_i 为中心词生成背景词 w_j 的条件概率 x_{ij}/x_i 记作 p_{ij}。我们可以进一步改写跳字模型的损失函数为

$$-\sum_{i\in\mathcal{V}} x_i \sum_{j\in\mathcal{V}} p_{ij} \log\, q_{ij}.$$

上式中，$-\sum_{j\in\mathcal{V}} p_{ij} \log\, q_{ij}$ 计算的是以 w_i 为中心词的背景词条件概率分布 p_{ij} 和模型预测的条件概率分布 q_{ij} 的交叉熵，且损失函数使用所有以词 w_i 为中心词的背景词的数量之和来加权。**最小化上式中的损失函数会令预测的条件概率分布尽可能接近真实的条件概率分布**。

然而，交叉熵损失函数有时并不是好的选择。一方面，令模型预测 q_{ij} 成为合法概率分布的代价是它在分母中基于**整个**词典的累加项，计算开销很大；另一方面，词典中往往有大量生僻词，它们在数据集中出现的次数极少。而有关大量生僻词的条件概率分布在交叉熵损失函数中的最终预测往往并不准确。

鉴于此，GloVe 模型采用了平方损失，并对跳字模型做了 3 点改动：

1. 使用非概率分布的变量 $p'_{ij} = x_{ij}$ 和 $q'_{ij} = \exp(\boldsymbol{u}_j^\top \boldsymbol{v}_i)$，并对它们取对数。因此，平方损失项是 $\left(\log\, p'_{ij} - \log\, q'_{ij}\right)^2 = \left(\boldsymbol{u}_j^\top \boldsymbol{v}_i - \log\, x_{ij}\right)^2$。

2. 为每个词 w_i 增加两个标量模型参数：中心词偏差项 b_i 和背景词偏差项 c_i。

3. 将每个损失项的权重替换成函数 $h(x_{ij})$，该函数是一个值域在 $[0, 1]$ 的单调递增函数。

所以，GloVe 模型的目标是最小化损失函数

$$\sum_{i\in\mathcal{V}} \sum_{j\in\mathcal{V}} h(x_{ij}) \left(\boldsymbol{u}_j^\top \boldsymbol{v}_i + b_i + c_j - \log\, x_{ij}\right)^2.$$

其中权重函数 $h(x)$ 的一个建议选择是：当 $x < c$ 时（如 $c = 100$），令 $h(x) = (x/c)^\alpha$（如 $\alpha = 0.75$），反之令 $h(x) = 1$。因为 $h(0) = 0$，所以对于 $x_{ij} = 0$ 的平方损失项可以直接忽略。当使用小批量随机梯度下降来训练时，每个时间步我们随机采样小批量非零 x_{ij}，然后计算梯度来迭代模型参数。这些非零 x_{ij} 是预先基于整个数据集计算得到的，包含了数据集的全局统计信息。因此，GloVe 模型的命名取“全局向量”（Global Vectors）之意。

需要强调的是，如果词 w_i 出现在词 w_j 的背景窗口里，那么词 w_j 也会出现在词 w_i 的背景窗口里。也就是说，$x_{ij} = x_{ji}$。不同于 word2vec 中拟合的是非对称的条件概率 p_{ij}，loVe 模型拟合的是对称的 $\log\, x_{ij}$。因此，任意词的中心词向量和背景词向量在 GloVe 模型中是等价的。但由于初始化

值的不同，同一个词最终学习到的两组词向量可能不同。当学习得到所有词向量以后，GloVe 模型使用**中心词向量与背景词向量之和**作为该词的最终词向量。

从条件概率的比值来理解 GloVe 模型所做的改动：

作为源于某大型语料库的真实例子，图9.4列举了两组分别以“ice”（冰）和“steam”（蒸汽）为中心词的条件概率以及它们之间的比值：

w_k=	“solid”	“gas”	“water”	“fashion”
$p_1 = P(w_k \mid \text{“ice”})$	0.00019	0.000066	0.003	0.000017
$p_2 = P(w_k \mid \text{“steam”})$	0.000022	0.00078	0.0022	0.000018
p_1/p_2	8.9	0.085	1.36	0.96

Figure 9.4: 计算条件概率的比值。

可以看到：

- 对于与“ice”相关而与“steam”不相关的词 w_k，如 $w_k =$ “solid”（固体），我们期望条件概率比值较大，如上表最后一行中的值 8.9；
- 对于与“ice”不相关而与“steam”相关的词 w_k，如 $w_k =$ “gas”（气体），我们期望条件概率比值较小，如上表最后一行中的值 0.085；
- 对于与“ice”和“steam”都相关的词 w_k，如 $w_k =$ “water”（水），我们期望条件概率比值接近 1，如上表最后一行中的值 1.36；
- 对于与“ice”和“steam”都不相关的词 w_k，如 $w_k =$ “fashion”（时尚），我们期望条件概率比值接近 1，如上表最后一行中的值 0.96。

由此可见，条件概率比值能比较直观地表达**词与词之间的关系**。我们可以构造一个词向量函数使它能有效拟合条件概率比值：

$$f(\boldsymbol{u}_j, \boldsymbol{u}_k, \boldsymbol{v}_i) \approx \frac{p_{ij}}{p_{ik}}.$$

接下来给出 f 的一种可行的构造方式。注意到条件概率比值是一个标量，我们可以将 f 限制为一个标量函数：$f(\boldsymbol{u}_j, \boldsymbol{u}_k, \boldsymbol{v}_i) = f\left((\boldsymbol{u}_j - \boldsymbol{u}_k)^\top \boldsymbol{v}_i\right)$。交换索引 j 和 k 后可以看到函数 f 应该满足 $f(x)f(-x) = 1$，因此一种可能是 $f(x) = \exp(x)$，于是

$$f(\boldsymbol{u}_j, \boldsymbol{u}_k, \boldsymbol{v}_i) = \frac{\exp\left(\boldsymbol{u}_j^\top \boldsymbol{v}_i\right)}{\exp\left(\boldsymbol{u}_k^\top \boldsymbol{v}_i\right)} \approx \frac{p_{ij}}{p_{ik}}.$$

满足最右边约等号的一种可能是 $\exp\left(\boldsymbol{u}_j^\top \boldsymbol{v}_i\right) \approx \alpha p_{ij}$，这里 α 是一个常数。考虑到 $p_{ij} = x_{ij}/x_i$，取对数后 $\boldsymbol{u}_j^\top \boldsymbol{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$。我们使用额外的偏差项来拟合 $-\log \alpha + \log x_i$，因此引入中心词偏差项 b_i 和背景词偏差项 c_j：

$$\boldsymbol{u}_j^\top \boldsymbol{v}_i + b_i + c_j \approx \log(x_{ij}).$$

对上式左右两边取平方误差并加权，即得到 GloVe 模型的损失函数。

到目前为止，我们介绍了如何用词向量表征单词，并给出了几种训练词向量的方式。实际使用中，我们大多是直接下载已在各类大型文本库上经过训练的词向量，常用的有：

```
# vocab.pretrained_aliases.keys()
dict_keys([
    'charngram.100d',
    'fasttext.en.300d', 'fasttext.simple.300d',
    'glove.42B.300d', 'glove.840B.300d',
    'glove.twitter.27B.25d', 'glove.twitter.27B.50d',
    'glove.twitter.27B.100d', 'glove.twitter.27B.200d',
    'glove.6B.50d', 'glove.6B.100d', 'glove.6B.200d', 'glove.6B.300d'
])
```

可以直接基于这些预训练的词向量借助 top-k 余弦相似度求解近义词和类比词。

9.6 文本情感分类

本节展示如何基于预训练词向量做情感分类。给定的数据集中，一个样本是特征：评论（一条文本）和类别：情感判断（positive 或 negative）的组合。在数据预处理阶段，需要将评论拆分为独立的词，筛除词典中出现频率过低的词，并通过截断或者补 0 来将每条评论长度固定。

情感分类归根到底是一个分类问题。接下来分别给出基于双向循环神经网络和卷积神经网络的模型。

9.6.1 双向循环神经网络

具体地，每个词先通过嵌入层（nn.Embedding）得到对应的词向量。然后，我们使用双向循环神经网络对特征序列进一步编码得到序列信息。最后，我们将编码的序列信息通过全连接层变换为输出。具体地，我们可以**将双向长短期记忆在最初时间步和最终时间步的隐藏状态连结，作为特征序列的表征传递给输出层分类**。

```
# 双向循环神经网络
class BiRNN(nn.Module):
    def __init__(self, vocab, embed_size, num_hiddens, num_layers):
        super(BiRNN, self).__init__()
        self.embedding = nn.Embedding(len(vocab), embed_size)
        self.encoder = nn.LSTM(input_size=embed_size,
                               hidden_size=num_hiddens,
                               num_layers=num_layers,
                               bidirectional=True)
        # 初始时间步和最终时间步的隐藏状态作为全连接层输入
        # 双向循环神经网络的隐藏层变量是num_hiddens*2，首尾都算上所以*4
        # 输出的是pos或者neg，因此num_outputs=2
        self.decoder = nn.Linear(4*num_hiddens, 2)

    def forward(self, inputs):
        # inputs的形状是(批量大小，序列长度)
        # 因为LSTM需要将序列长度(seq_len)作为第一维，所以将输入转置后
        # 再提取词特征，输出形状为(seq_len，批量大小，词向量维度)
        # 然后用embedding的方式取代了one-hot encoding
        embeddings = self.embedding(inputs.permute(1, 0))
        # outputs形状是(seq_len，批量大小，2*num_hiddens)
        outputs, _ = self.encoder(embeddings) # output，(h，c)
        # 连结初始时间步和最终时间步的隐藏状态作为全连接层输入
```

```
        # 形状为(批量大小, 4*num_hiddens)
        encoding = torch.cat((outputs[0], outputs[-1]), -1)
        outs = self.decoder(encoding)
        return outs
```

9.6.2 卷积神经网络（textCNN）

假设输入的文本序列由 n 个词组成，每个词用 d 维的词向量表示。那么输入样本的宽为 n（一维向量的特征就是输入的文本序列），高为 1，输入通道数为 d（词向量的大小则是深度）。

然后按照如下步骤计算输出：

1. 定义多个一维卷积核，并使用这些卷积核对输入分别做卷积计算。宽度不同的卷积核可能会捕捉到不同个数的相邻词的相关性；

2. 对输出的所有通道分别做时序最大池化，再将这些通道的池化输出值连结为向；

3. 通过全连接层将连结后的向量变换为有关各类别的输出。这一步可以使用丢弃层应对过拟合。

图9.5给出了 textCNN 的模型结构。这里的输入是一个有 11 个词的句子，每个词用 6 维词向量表示。因此输入序列的宽为 11，输入通道数为 6。给定 2 个一维卷积核，核宽分别为 2 和 4，输出通道数分别设为 4 和 5。因此，一维卷积计算后，4 个输出通道的宽为 $11-2+1=10$，而其他 5 个通道的宽为 $11-4+1=8$。尽管每个通道的宽不同，我们依然可以对各个通道做时序最大池化，并将 9 个通道的池化输出连结成一个 9 维向量。最终，使用全连接将 9 维向量变换为 2 维输出，即正面情感和负面情感的预测。

```
# textCNN
class TextCNN(nn.Module):
    def __init__(self, vocab, embed_size, kernel_sizes, num_channels):
        super(TextCNN, self).__init__()
        self.embedding = nn.Embedding(len(vocab), embed_size)
        # 不参与训练的嵌入层
        self.constant_embedding = nn.Embedding(len(vocab), embed_size)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Linear(sum(num_channels), 2)
        # 时序最大池化层没有权重，所以可以共用一个实例
        self.pool = GlobalMaxPool1d()
        self.convs = nn.ModuleList() # 创建多个一维卷积层
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.append(nn.Conv1d(in_channels=2 * embed_size,
                                        out_channels=c,
                                        kernel_size=k))

    def forward(self, inputs):
        # inputs: (batch_size, seq_len)
        # embeddings: (batch_size, seq_len, 2 * embed_size)
        embeddings = torch.cat((self.embedding(inputs),
            self.constant_embedding(inputs)), dim=2)
        # 根据Conv1D要求的输入格式，将词向量维，即一维卷积层的通道维，变换到前一维
        # embeddings: (batch_size, 2 * embed_size, seq_len)
        embeddings = embeddings.permute(0, 2, 1)
        # (batch_size, out_channels, out_seq_len) --->
        # (batch_size, out_channels, 1) --->
        # (batch_size, out_channels) --->
```

```
        # (batch_size, sum_out_channels)
        encoding = torch.cat(
            [self.pool(F.relu(conv(embeddings))).squeeze(-1)
            for conv in self.convs], dim=1
        )
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

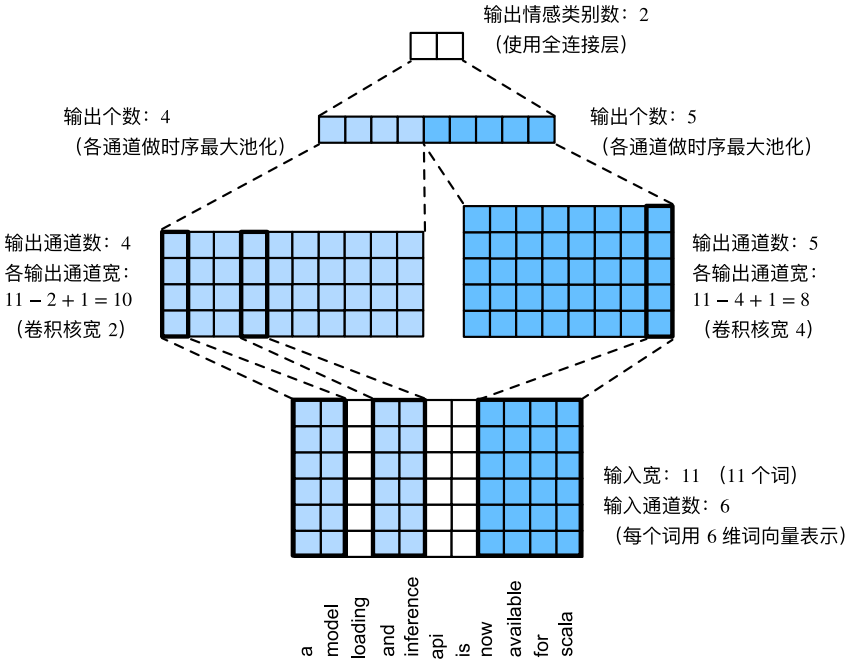

Figure 9.5: textCNN 的结构示例。

9.7 编码器—解码器（seq2seq）

在自然语言处理的很多应用中，输入和输出都可以是不定长序列。以机器翻译为例，输入可以是一段不定长的英语文本序列，输出也是一段不定长的法语文本序列，例如

- 英语输入：“They”、“are”、“watching”、“.”
- 法语输出：“Ils”、“regardent”、“.”

当输入和输出都是不定长序列时，我们可以使用编码器—解码器（encoder-decoder）或者 seq2seq 模型。这两个模型本质上都用到了两个循环神经网络，分别叫做编码器和解码器。编码器用来分析输入序列，解码器用来生成输出序列。

下图描述了使用编码器—解码器将上述英语句子翻译成法语句子的一种方法。在训练数据集中，我们可以在每个句子后附上特殊符号“<eos>”（end of sequence）以表示序列的终止。编码器每个时间步的输入依次为英语句子中的单词、标点和特殊符号“<eos>”。图中使用了编码器在**最终时间步的隐藏状态**作为输入句子的编码信息。解码器在各个时间步中使用输入句子的编码信息、上个时间步的输出以及隐藏状态作为输入。我们希望解码器在各个时间步能正确依次输出翻译后的法语单词、标点和特殊符号“<eos>”。此外，解码器在**最初时间步**的输入用到了一个表示序列开始的特殊符号“<bos>”（beginning of sequence）。

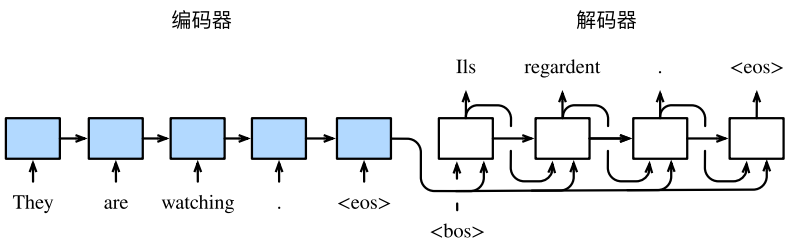

Figure 9.6: seq2seq 模型。

接下来分别介绍编码器和解码器的定义。

9.7.1 编码器

编码器的作用是把一个不定长的输入序列的信息编码为一个定长的背景变量 $\boldsymbol{c}$，常用的编码器是循环神经网络。

以批量大小为 1 的时序数据样本为例，假设输入序列是 $x_1, \ldots, x_T$，其中 x_i 是输入句子中的第 i 个词。在时间步 t，循环神经网络将 x_t 的特征向量 $\boldsymbol{x}_t$ 和上个时间步的隐藏状态 $\boldsymbol{h}_{t-1}$ 作为输入，并得到当前时间步的隐藏状态 $\boldsymbol{h}_t$。我们可以用函数 f 表达循环神经网络隐藏层的变换：

$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}).$$

接下来，编码器通过自定义函数 q 将各个时间步的隐藏状态变换为背景变量

$$\boldsymbol{c} = q(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T).$$

例如，**当选择 $q(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T) = \boldsymbol{h}_T$ 时，背景变量是输入序列最终时间步的隐藏状态 $\boldsymbol{h}_T$**。

以上描述的编码器是一个单向的循环神经网络，每个时间步的隐藏状态只取决于该时间步及之前的输入子序列。也可以使用双向循环神经网络构造编码器。在这种情况下，编码器每个时间步的隐藏状态同时取决于该时间步之前和之后的子序列（包括当前时间步的输入），并编码了整个序列的信息。

```
# 编码器
class Encoder(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 drop_prob=0, **kwargs):
        super(Encoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(embed_size, num_hiddens, num_layers, dropout=drop_prob)

    def forward(self, inputs, state):
        # inputs: (batch_size, num_steps) --> (num_steps, batch_size, embed_size)
        embeddings = self.embedding(inputs.long()).permute(1, 0, 2)
        # outputs: (num_steps, batch_size, hidden_size)
        # state: (num_hiddens, batch_size, hidden_size)
        return self.rnn(embeddings, state)

    def begin_state(self):
        # initialized as zero
        return None
```

9.7.2 解码器

编码器输出的背景变量 $\boldsymbol{c}$ 编码了整个输入序列 $x_1, \ldots, x_T$ 的信息。给定训练样本中的输出序列 $y_1, y_2, \ldots, y_{T'}$，对每个时间步 t'（符号与输入序列或编码器的时间步 t 有区别），解码器输出 $y_{t'}$ 的条件概率将基于之前的输出序列 $y_1, \ldots, y_{t'-1}$ 和背景变量 $\boldsymbol{c}$，即 $P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c})$。

为此，我们可以使用另一个循环神经网络作为解码器。在输出序列的时间步 t'，解码器将上一时间步的输出 $y_{t'-1}$ 以及背景变量 $\boldsymbol{c}$ 作为输入，并将它们与上一时间步的隐藏状态 $\boldsymbol{s}_{t'-1}$ 变换为当前时间步的隐藏状态 $\boldsymbol{s}_{t'}$。因此，我们可以用函数 g 表达解码器隐藏层的变换：

$$\boldsymbol{s}_{t'} = g(y_{t'-1}, \boldsymbol{c}, \boldsymbol{s}_{t'-1}).$$

接下来我们可以使用自定义的输出层和 softmax 运算来计算 $P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c})$。例如，基于当前时间步的解码器隐藏状态 $\boldsymbol{s}_{t'}$、上一时间步的输出 $y_{t'-1}$ 以及背景变量 $\boldsymbol{c}$ 来计算当前时间步输出 $y_{t'}$ 的概率分布。

9.7.3 束搜索

接下来说明如何在得到 $P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c})$ 之后输出不定长的序列。

设输出文本词典 $\mathcal{Y}$（包含特殊符号“<eos>”）的大小为 $|\mathcal{Y}|$，输出序列的最大长度为 T'。所有可能的输出序列一共有 $\mathcal{O}(|\mathcal{Y}|^{T'})$ 种。这些输出序列中所有特殊符号“<eos>”后面的子序列将被舍弃。

贪婪搜索:

一种得到输出序列的方法是贪婪搜索（greedy search）。对于输出序列任一时间步 t'，我们从 $|\mathcal{Y}|$ 个词中搜索出条件概率最大的词

$$y_{t'} = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} P\left(y | y_1, \ldots, y_{t'-1}, c\right)$$

作为输出。一旦搜索出“<eos>”符号，或者输出序列长度已经达到了最大长度 T'，便完成输出。

基于输入序列生成输出序列的条件概率是 $\prod_{t'=1}^{T'} P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c})$。我们将该条件概率最大的输出序列称为最优输出序列。贪婪搜索的问题是**不能保证得到最优输出序列**。接下来用一个例子说明这一问题。

假设输出词典里面有“A”“B”“C”和“<eos>”这 4 个词。图9.7中每个时间步下的 4 个数字分别代表了该时间步生成“A”“B”“C”和“<eos>”这 4 个词的条件概率。在每个时间步，贪婪搜索选取条件概率最大的词。因此，图9.7将生成输出序列“A”“B”“C”“<eos>”。该输出序列的条件概率是 $0.5 \times 0.4 \times 0.4 \times 0.6 = 0.048$。

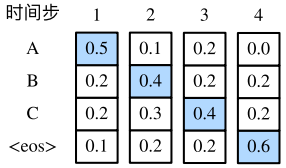

Figure 9.7: 贪婪搜索将得到“A”“B”“C”和“<eos>”。

接下来，观察图9.8演示的例子。与图9.7中不同，图9.7在时间步 2 中选取了条件概率第二大的词“C”。由于时间步 3 所基于的时间步 1 和 2 的输出子序列由图9.7中的“A”“B”变为了图9.8中的“A”“C”，导致图9.8中时间步 3 生成各个词的条件概率发生了变化。在时间步 3 和 4 分别选取“B”和“<eos>”，此时的输出序列“A”“C”“B”“<eos>”的条件概率是 $0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$，大于贪婪搜索得到的输出序列的条件概率。因此，贪婪搜索得到的输出序列“A”“B”“C”“<eos>”并非最优输出序列。

时间步	1	2	3	4
A	0.5	0.1	0.1	0.1
B	0.2	0.4	0.6	0.2
C	0.2	0.3	0.2	0.1
<eos>	0.1	0.2	0.1	0.6

Figure 9.8: 非贪婪搜索得到了“A”“C”“B”和“<eos>”。

贪婪搜索的计算开销是 $\mathcal{O}(|\mathcal{Y}|\,T')$。

穷举搜索：

如果目标是得到最优输出序列，可以考虑穷举搜索（exhaustive search）：穷举所有可能的输出序列，输出条件概率最大的序列。此时的计算开销为 $\mathcal{O}(|\mathcal{Y}|^{T'})$。

束搜索：

束搜索（beam search）是对贪婪搜索的一个改进算法。它有一个束宽（beam size）超参数。我们将它设为 k。在时间步 1 时，选取当前时间步条件概率最大的 k 个词，分别组成 k 个候选输出序列的首词。在之后的每个时间步，基于上个时间步的 k 个候选输出序列，从 $k\,|\mathcal{Y}|$ 个可能的输出序列中选取条件概率最大的 k 个，作为该时间步的候选输出序列。最终，我们从各个时间步的候选输出序列中筛选出包含特殊符号“<eos>”的序列，并将它们中所有特殊符号“<eos>”后面的子序列舍弃，得到最终候选输出序列的集合。

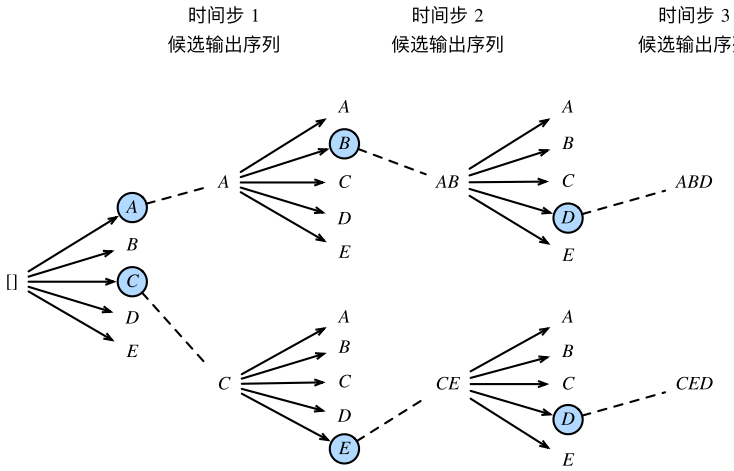

Figure 9.9: 束搜索。

图9.9演示了束搜索的过程。假设输出序列的词典中只包含 5 个元素，即 $\mathcal{Y} = \{A, B, C, D, E\}$，且其中一个为特殊符号“<eos>”。设束搜索的束宽等于 2，输出序列最大长度为 3。在输出序列的时间步 1 时，假设条件概率 $P(y_1 \mid \boldsymbol{c})$ 最大的 2 个词为 A 和 C。我们在时间步 2 时将对所有的 $y_2 \in \mathcal{Y}$ 都分别计算 $P(y_2 \mid A, \boldsymbol{c})$ 和 $P(y_2 \mid C, \boldsymbol{c})$，并从计算出的 10 个条件概率中取最大的 2 个，假设为 $P(B \mid A, \boldsymbol{c})$ 和 $P(E \mid C, \boldsymbol{c})$。那么，我们在时间步 3 时将对所有的 $y_3 \in \mathcal{Y}$ 都分别计算 $P(y_3 \mid A, B, \boldsymbol{c})$ 和 $P(y_3 \mid C, E, \boldsymbol{c})$，并从计算出的 10 个条件概率中取最大的 2 个，假设为 $P(D \mid A, B, \boldsymbol{c})$ 和 $P(D \mid C, E, \boldsymbol{c})$。如此一来，我们得到 6 个候选输出序列：（1）$A$；（2）$C$；（3）$A$、$B$；（4）$C$、$E$；（5）$A$、$B$、$D$ 和（6）C、E、D。接下来，我们将根据这 6 个序列得出最终候选输出序列的集合（注意每一个最终候选输出的所有特殊符号“<eos>”后面的子序列需要被舍弃）。

在最终候选输出序列的集合中，我们取以下分数最高的序列作为输出序列：

$$\frac{1}{L^\alpha}\log P(y_1,\ldots,y_L)=\frac{1}{L^\alpha}\sum_{t'=1}^{L}\log P(y_{t'}\mid y_1,\ldots,y_{t'-1},\boldsymbol{c}),$$

其中 L 为最终候选序列长度，α 一般可选为 0.75。分母上的 L^α 是为了惩罚较长序列在以上分数中较多的对数相加项。

束搜索的计算开销为 $\mathcal{O}(k\,|\mathcal{Y}|\,T')$，介于贪婪搜索和穷举搜索的计算开销之间。**贪婪搜索可看作是束宽为 1 的束搜索**。束搜索通过灵活的束宽 k 来权衡计算开销和搜索质量。

9.7.4 注意力机制

请思考一个翻译的例子：输入为英语序列 “They” “are” “watching” “.”，输出为法语序列 “Ils” “regardent” “.”。不难想到，解码器在生成输出序列中的每一个词时可能只需利用输入序列某一部分的信息。例如，在输出序列的时间步 1，解码器可以主要依赖 “They” “are” 的信息来生成 “Ils”，在时间步 2 则主要使用来自 “watching” 的编码信息生成 “regardent”，最后在时间步 3 则直接映射句号 “.”。这看上去就像是在解码器的每一时间步**对输入序列中不同时间步的表征或编码信息分配不同的注意力**一样，这就是注意力机制。

仍然以循环神经网络为例，注意力机制通过对编码器所有时间步的隐藏状态做加权平均来得到背景变量。解码器在每一时间步调整这些权重，即注意力权重，从而能够在不同时间步分别关注输入序列中的不同部分并编码进相应时间步的背景变量。接下来给出注意力机制的工作原理。

回顾 seq2seq 模型，解码器在时间步 t' 的隐藏状态 $\boldsymbol{s}_{t'}=g(\boldsymbol{y}_{t'-1},\boldsymbol{c},\boldsymbol{s}_{t'-1})$，其中 $\boldsymbol{y}_{t'-1}$ 是上一时间步 $t'-1$ 的输出 $y_{t'-1}$ 的表征，且任一时间步 t' 使用相同的背景变量 $\boldsymbol{c}$。但在注意力机制中，解码器的每一时间步将使用可变的背景变量。记 $\boldsymbol{c}_{t'}$ 是解码器在时间步 t' 的背景变量，那么解码器在该时间步的隐藏状态可以改写为

$$\boldsymbol{s}_{t'}=g(\boldsymbol{y}_{t'-1},\boldsymbol{c}_{t'},\boldsymbol{s}_{t'-1}).$$

这里的关键是如何计算背景变量 $\boldsymbol{c}_{t'}$ 和如何利用它来更新隐藏状态 $\boldsymbol{s}_{t'}$。下面将分别描述这两个关键点。

计算背景变量：

令编码器在时间步 t 的隐藏状态为 $\boldsymbol{h}_t$，且总时间步数为 T。那么解码器在时间步 t' 的背景变量为所有编码器隐藏状态的加权平均：

$$\boldsymbol{c}_{t'}=\sum_{t=1}^{T}\alpha_{t't}\boldsymbol{h}_t,$$

其中给定 t' 时，权重 $\alpha_{t't}$ 在 $t=1,\ldots,T$ 的值是一个概率分布。为了得到概率分布，我们可以使用 softmax 运算:

$$\alpha_{t't}=\frac{\exp(e_{t't})}{\sum_{k=1}^{T}\exp(e_{t'k})},\quad t=1,\ldots,T.$$

现在，我们需要定义如何计算上式中 softmax 运算的输入 $e_{t't}$。由于 $e_{t't}$ 同时取决于解码器的时间步 t' 和编码器的时间步 t，我们不妨以解码器在时间步 $t'-1$ 的隐藏状态 $\boldsymbol{s}_{t'-1}$ 与编码器在时间步 t 的隐藏状态 $\boldsymbol{h}_t$ 为输入，并通过函数 a 计算 $e_{t't}$：

$$e_{t't}=a(\boldsymbol{s}_{t'-1},\boldsymbol{h}_t).$$

这里函数 a 有多种选择，如果两个输入向量长度相同，一个简单的选择是计算它们的内积 $a(\boldsymbol{s},\boldsymbol{h})=\boldsymbol{s}^\top\boldsymbol{h}$。最早提出注意力机制的论文是将输入连结后通过含单隐藏层的多层感知机变换：

$$a(\boldsymbol{s},\boldsymbol{h})=\boldsymbol{v}^\top\tanh(\boldsymbol{W}_s\boldsymbol{s}+\boldsymbol{W}_h\boldsymbol{h}),$$

其中 $\boldsymbol{v}$、$\boldsymbol{W}_s$、$\boldsymbol{W}_h$ 都是可以学习的模型参数。

图9.10描绘了注意力机制如何为解码器在时间步 2 计算背景变量。首先，函数 a 根据解码器在时间步 1 的隐藏状态和编码器在各个时间步的隐藏状态计算 softmax 运算的输入。softmax 运算输出概率分布并对编码器各个时间步的隐藏状态做加权平均，从而得到背景变量。

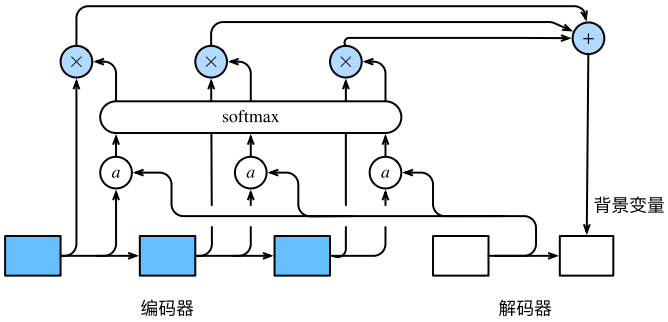

Figure 9.10: 注意力机制下，解码器在时间步 2 的背景变量的计算过程。

我们还可以对注意力机制采用更高效的**矢量化计算**。广义上，注意力机制的输入包括查询项以及一一对应的键项和值项，其中值项是需要加权平均的一组项。在加权平均中，值项的权重来自查询项以及与该值项对应的键项的计算。

这里，查询项为解码器的隐藏状态，键项和值项均为编码器的隐藏状态。让我们考虑一个常见的简单情形，即编码器和解码器的隐藏单元个数均为 h，且函数 $a(\boldsymbol{s}, \boldsymbol{h}) = \boldsymbol{s}^\top \boldsymbol{h}$。假设我们希望根据解码器单个隐藏状态 $\boldsymbol{s}_{t'-1} \in \mathbb{R}^h$ 和编码器所有隐藏状态 $\boldsymbol{h}_t \in \mathbb{R}^h, t = 1, \ldots, T$ 来计算背景向量 $\boldsymbol{c}_{t'} \in \mathbb{R}^h$。我们可以将查询项矩阵 $\boldsymbol{Q} \in \mathbb{R}^{1\times h}$ 设为 $\boldsymbol{s}_{t'-1}^\top$，并令键项矩阵 $\boldsymbol{K} \in \mathbb{R}^{T\times h}$ 和值项矩阵 $\boldsymbol{V} \in \mathbb{R}^{T\times h}$ 相同且第 t 行均为 $\boldsymbol{h}_t^\top$。此时，我们只需要通过矢量化计算

$$\text{softmax}(\boldsymbol{Q}\boldsymbol{K}^\top)\boldsymbol{V}$$

即可算出转置后的背景向量 $\boldsymbol{c}_{t'}^\top$。当查询项矩阵 $\boldsymbol{Q}$ 的行数为 n 时，上式将得到 n 行的输出矩阵。输出矩阵与查询项矩阵在相同行上一一对应。

更新隐藏状态：

现在我们描述第二个关键点，即更新隐藏状态。以门控循环单元为例，在解码器中我们可以对 GRU 中的门控循环单元的设计稍作修改，从而变换上一时间步 $t'-1$ 的输出 $\boldsymbol{y}_{t'-1}$、隐藏状态 $\boldsymbol{s}_{t'-1}$ 和当前时间步 t' 的含注意力机制的背景变量 $\boldsymbol{c}_{t'}$。解码器在时间步 t' 的隐藏状态为

$$\boldsymbol{s}_{t'} = \boldsymbol{z}_{t'} \odot \boldsymbol{s}_{t'-1} + (1 - \boldsymbol{z}_{t'}) \odot \tilde{\boldsymbol{s}}_{t'},$$

其中的重置门、更新门和候选隐藏状态分别为

$$\boldsymbol{r}_{t'} = \sigma(\boldsymbol{W}_{yr}\boldsymbol{y}_{t'-1} + \boldsymbol{W}_{sr}\boldsymbol{s}_{t'-1} + \boldsymbol{W}_{cr}\boldsymbol{c}_{t'} + \boldsymbol{b}_r),$$

$$\boldsymbol{z}_{t'} = \sigma(\boldsymbol{W}_{yz}\boldsymbol{y}_{t'-1} + \boldsymbol{W}_{sz}\boldsymbol{s}_{t'-1} + \boldsymbol{W}_{cz}\boldsymbol{c}_{t'} + \boldsymbol{b}_z),$$

$$\tilde{\boldsymbol{s}}_{t'} = \tanh(\boldsymbol{W}_{ys}\boldsymbol{y}_{t'-1} + \boldsymbol{W}_{ss}(\boldsymbol{s}_{t'-1} \odot \boldsymbol{r}_{t'}) + \boldsymbol{W}_{cs}\boldsymbol{c}_{t'} + \boldsymbol{b}_s),$$

其中含下标的 $\boldsymbol{W}$ 和 $\boldsymbol{b}$ 分别为门控循环单元的权重参数和偏差参数。

```
# 实现注意力机制
def attention_model(input_size, attention_size):
    return nn.Sequential(
        nn.Linear(input_size, attention_size, bias=False),
        nn.Tanh(),
```

```
        nn.Linear(attention_size, 1, bias=False)
    )

def attention_forward(model , enc_states, dec_state):
    """
    enc_states: (num_steps, batch_size, hidden_size)
    dec_state: (batch_size, hidden_size)
    """
    # 将解码器隐藏状态广播到和编码器隐藏状态形状相同后进行连结
    dec_states = dec_state.unsqueeze(dim=0).expand_as(enc_states)
    enc_and_dec_states = torch.cat((enc_states, dec_states), dim=2)
    # e: (num_steps, batch_size, 1)
    e = model(enc_and_dec_states)
    # alpha: (num_steps, batch_size, 1)
    alpha = F.softmax(e, dim=0)
    # alpha在每个时间步上的softmax分布作为权重和enc_states的各个时间步加权平均
    # 自动对所有隐藏单元执行加权平均
    # (num_steps, batch_size, 1) * (num_steps, batch_size, hidden_size) --->
    # (num_steps, batch_size, hidden_size)
    return (alpha * enc_states).sum(dim=0) # (batch_size, hidden_size)
```

```
# 含注意力机制的解码器
class Decoder(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 attention_size, drop_prob=0):
        super(Decoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # 解码器隐藏状态广播到和编码器隐藏状态形状相同后进行连结
        # 因此attention的输入大小为2*num_hiddens
        self.attention = attention_model(2*num_hiddens, attention_size)
        # GRU的输入包含attention输出的c和实际输入，所以为num_hiddens + embed_size
        self.rnn = nn.GRU(num_hiddens + embed_size, num_hiddens,
                          num_layers, dropout=drop_prob)
        self.out = nn.Linear(num_hiddens, vocab_size)

    def forward(self, cur_input, state, enc_states):
        """
        cur_input shape: (batch_size)
        state shape: (num_hiddens, batch_size, hidden_size)
        """
        # 使用注意力机制计算背景向量
        c = attention_forward(self.attention, enc_states, state[-1])
        # (batch_size, embed_size) + (batch_size, hidden_size) on dim 1
        input_and_c = torch.cat((self.embedding(cur_input), c), dim=1)
        # 为输入和背景向量的连结增加时间步维，时间步个数为1
        output, state = self.rnn(input_and_c.unsqueeze(0), state)
        # 移除时间步维，输出形状为(batch_size, vocab_size)
        output = self.out(output).squeeze(dim=0)
        return output, state

    def begin_state(self, enc_state):
        # 直接将编码器最终时间步的隐藏状态作为解码器的初始隐藏状态
```

```
        return enc_state
```

本质上，注意力机制能够为编码信息中较有价值的部分分配较多的计算资源。这个有趣的想法自提出后得到了快速发展，特别是启发了依靠注意力机制来编码输入序列并解码出输出序列的 Transformer 模型的设计。Transformer 抛弃了卷积神经网络和循环神经网络的架构，在计算效率上比基于循环神经网络的编码器—解码器模型通常更具明显优势。

9.7.5 BLEU

seq2seq 是一个通用的文本生成的框架，典型应用场景是机器翻译。判断机器翻译优劣的一个标准是 BLEU（Bilingual Evaluation Understudy）。对于模型预测序列中任意的子序列，BLEU 考察这个子序列是否出现在标签序列中。

具体地，设词数为 n 的子序列的精度为 p_n。它是预测序列与标签序列匹配词数为 n 的子序列的数量与预测序列中词数为 n 的子序列的数量之比。例如，标签序列为 A、B、C、D、E、F，预测序列为 A、B、B、C、D，那么 $p_1 = 4/5, p_2 = 3/4, p_3 = 1/3, p_4 = 0$。设 $len(\text{label})$ 和 $len(\text{pred})$ 分别为标签序列和预测序列的词数，那么，BLEU 的定义为

$$\exp\left(\min\left(0, 1 - \frac{len(\text{label})}{len(\text{pred})}\right)\right) \prod_{n=1}^{k} (p_n)^{1/2^n},$$

其中 k 是我们希望匹配的子序列的最大词数。可以看到当预测序列和标签序列完全一致时，BLEU 为 1。

因为匹配较长子序列比匹配较短子序列更难，BLEU 对匹配较长子序列的精度赋予了更大权重。例如，当 p_n 固定在 0.5 时，随着 n 的增大，$0.5^{1/2} \approx 0.7, 0.5^{1/4} \approx 0.84, 0.5^{1/8} \approx 0.92, 0.5^{1/16} \approx 0.96$。另外，模型预测较短序列往往会得到较高 p_n 值。因此，上式中连乘项前面的系数是为了惩罚较短的输出而设的。当 $k = 2$ 时，假设标签序列为 A、B、C、D、E、F，而预测序列为 A、B。虽然 $p_1 = p_2 = 1$，但惩罚系数 $\exp(1 - 6/2) \approx 0.14$，因此 BLEU 也接近 0.14。

10 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。

这份文档是我个人在学习深度学习理论与实现的过程中做的总结，最新版本的地址为http://hliangzhao.me/math/cheatsheet.pdf。