Deep Learning Cheat Sheet

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

October 16, 2020

Contents

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

1 SoftMax 回归

1.1 模型定义

如无特别说明，向量均指列向量。

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

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

因此 softmax 回归的预测结果为

$$\hat{y} = \operatorname*{argmax}_{c=1}^{C} p(y = c|\boldsymbol{x}) = \operatorname*{argmax}_{c=1}^{C} \boldsymbol{w}_c^\top \boldsymbol{x}.$$

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

1.2 单样本的矢量计算表达式

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

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

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

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，假设所有向量均为行向量。

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

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

PyTorch 会自动对 $\boldsymbol{b}$ 进行广播。

1.4 参数学习

采用交叉熵损失函数，只关心正确类别的预测概率：

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

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

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

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

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

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

2 多层感知机（MLP）

2.1 隐藏层

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

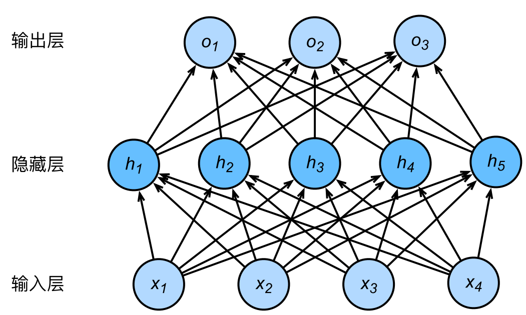

Figure 2.1: 多层感知机结构。

2.2 矢量计算表达式

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

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

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

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

因此

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

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

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）
$$\mathrm{ReLU}(x)=\max(x,0)$$
- sigmoid
$$\mathrm{sigmoid}(x)=\frac{1}{1+\exp(-x)}$$
- tanh
$$\tanh(x)=\frac{1-\exp(-2x)}{1+\exp(-2x)}$$

2.4 多层感知机

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

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

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

3 权重衰减（weight decay）

应对过拟合的方法是**正则化**。以线性回归问题为例，默认的均方误差为

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

若加上 L_2 范数惩罚项，则得到新损失函数：

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

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

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

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

4 Dropout

在讲解 MLP 时我们给出了如图2.1所示的带有隐藏层的神经网络。

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

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

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

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

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

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

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

对上述 MLP 训练的时候使用 dropout，一种可能的网络结构如下：此时 MLP 的输出不依赖 h_2

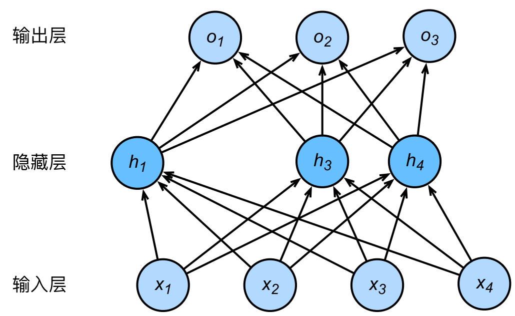

Figure 4.1: Dropout。

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。

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：

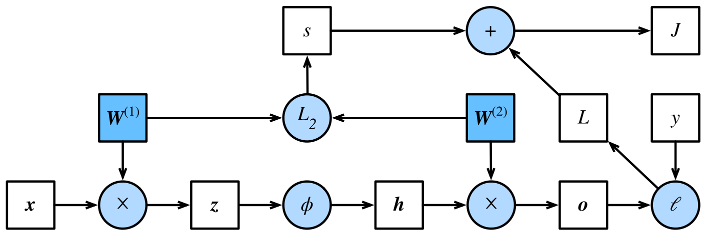

Figure 5.1: 正向传播的计算图。

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，

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

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

互相关运算如图所示（$corr = \mathrm{rot180}(conv)$）：

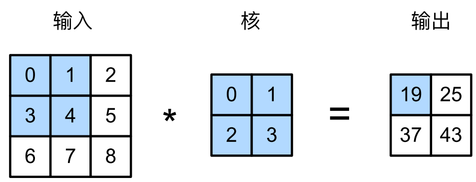

Figure 6.1: 互相关运算。

6.2 填充与步长

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

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

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

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。

6.4 1×1 卷积层

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

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

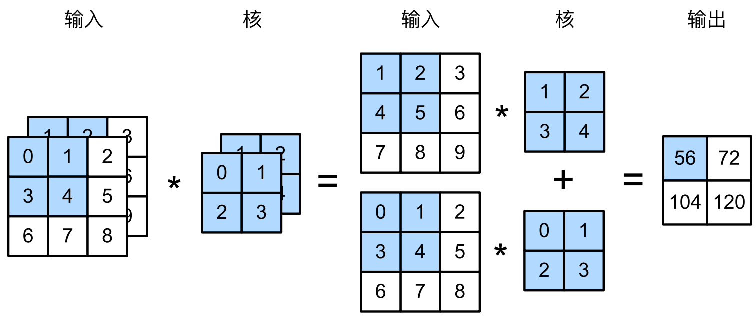

Figure 6.2: 多输入通道与多输出通道下的互相关运算。

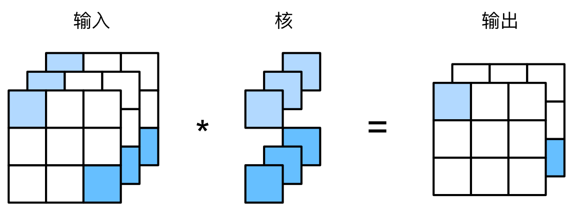

Figure 6.3: 1×1 卷积层。

6.5 池化层

池化层可以缓解卷积层对位置的过度敏感性。

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

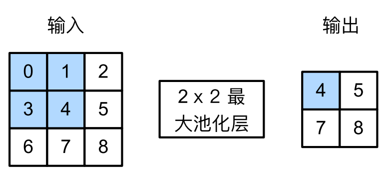

Figure 6.4: 最大池化层。

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。

6.6 LeNet-5

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

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

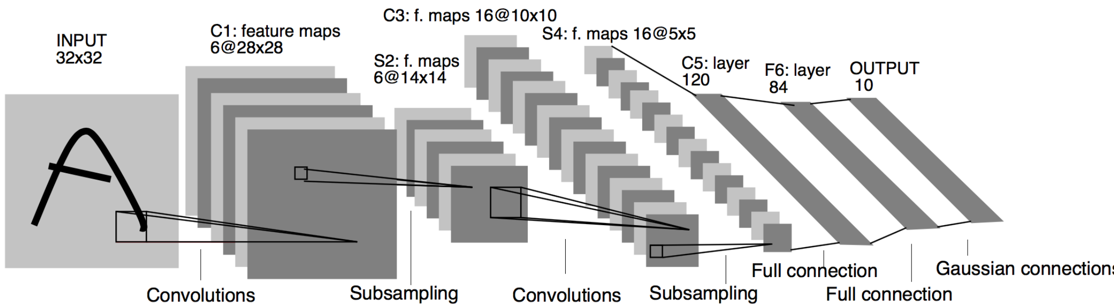

Figure 6.5: LeNet-5 的结构。

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。

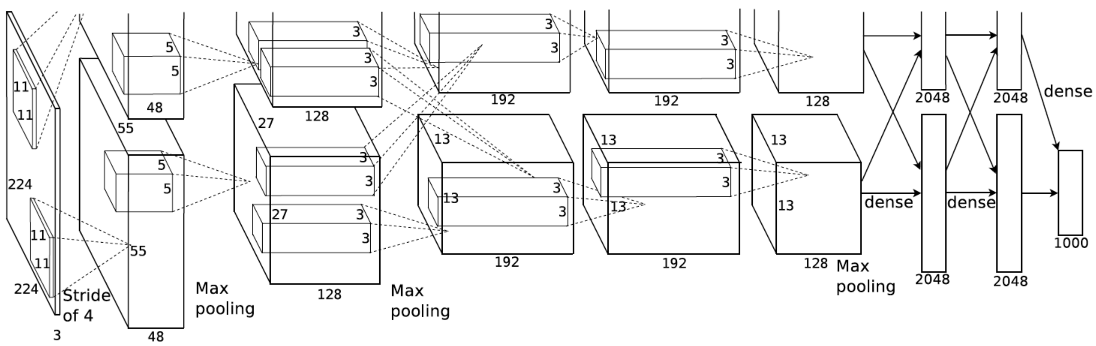

Figure 6.6: AlexNet 的结构。

6.8 VGG

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。

```python
# vgg_block
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)
```

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。

6.9 Network in Network (NiN)

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

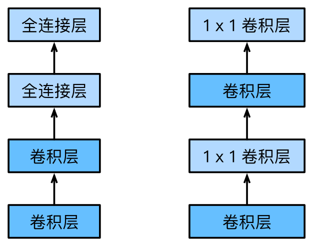

Figure 6.7: NiN 结构。

```
# 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
```

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。

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
```

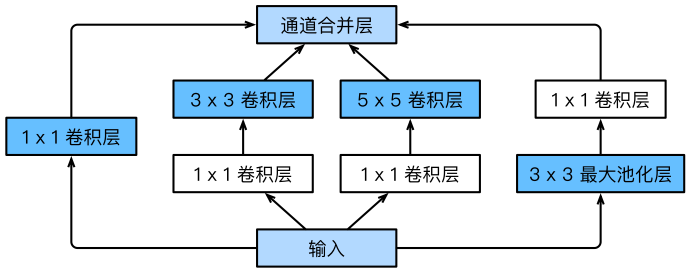

Figure 6.8: NiN 的结构。

```
    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
```

6.11 批量归一化

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

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

6.11.1 对全连接层批量归一化

使用批量归一化的全连接层的输出为

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

其中 u 为全连接层的输入，BN 为批量归一化运算符。

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

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

$BN(\cdot)$ 的具体步骤如下：

首先对小批量 $\mathcal{B}$ 求均值和方差：

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

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

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

这里 $\epsilon > 0$ 是一个很小的常数，保证分母大于 0。

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

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

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

6.11.2 对卷积层批量归一化

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

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

6.11.3 预测时的批量归一化

使用批量归一化训练时，我们可以将批量大小设得大一点，从而使批量内样本的均值和方差的计算都较为准确。将训练好的模型用于预测时，我们希望模型对于任意输入都有确定的输出。因此，单个样本的输出不应取决于批量归一化所需要的随机小批量中的均值和方差。一种常用的方法是通过移动平均估算整个训练数据集的样本均值和方差，并在预测时使用它们得到确定的输出。可见，和丢弃层一样，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。

6.12 ResNet

普通的网络结构（左）与加入残差连接的网络结构（右）：在右图所示的残差块中，虚线框内要学习的是残差映射 $f(\boldsymbol{x}) - \boldsymbol{x}$，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。

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。

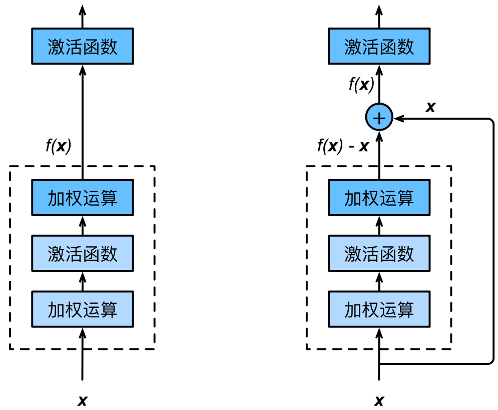

Figure 6.9: 残差网络的基本结构（右）。

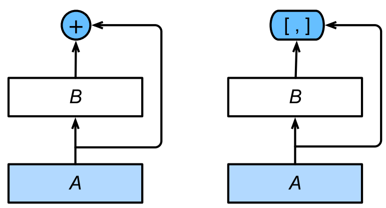

Figure 6.10: DenseNet 的基本结构：稠密块（dense block）和过渡层（transition layer）。

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。

7 后记

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

本清单仍在持续更新中。