

10.1 注意力提示

感谢读者对本书的关注，因为读者的注意力是一种稀缺的资源：此刻读者正在阅读本书（而忽略了其他的书），因此读者的注意力是用机会成本（与金钱类似）来支付的。为了确保读者现在投入的注意力是值得的，作者们尽全力（全部的注意力）创作一本好书。

自经济学研究稀缺资源分配以来，人们正处在“注意力经济”时代，即人类的注意力被视为可以交换的、有限的、有价值的且稀缺的商品。许多商业模式也被开发出来去利用这一点：在音乐或视频流媒体服务上，人们要么消耗注意力在广告上，要么付钱来隐藏广告；为了在网络游戏世界的成长，人们要么消耗注意力在游戏战斗中，从而帮助吸引新的玩家，要么付钱立即变得强大。总之，注意力不是免费的。

注意力是稀缺的，而环境中的干扰注意力的信息却并不少。比如人类的视觉神经系统大约每秒收到 10^8 位的信息，这远远超过了大脑能够完全处理的水平。幸运的是，人类的祖先已经从经验（也称为数据）中认识到“并非感官的所有输入都是一样的”。在整个人类历史中，这种只将注意力引向感兴趣的一小部分信息的能力，使人类的大脑能够更明智地分配资源来生存、成长和社交，例如发现天敌、找寻食物和伴侣。

10.1.1 生物学中的注意力提示

注意力是如何应用于视觉世界中的呢？这要从当今十分普及的双组件（two-component）的框架开始讲起：这个框架的出现可以追溯到19世纪90年代的威廉·詹姆斯，他被认为是“美国心理学之父”（James, 2007）。在这个框架中，受试者基于非自主性提示和自主性提示有选择地引导注意力的焦点。

非自主性提示是基于环境中物体的突出性和易见性。想象一下，假如我们面前有五个物品：一份报纸、一篇研究论文、一杯咖啡、一本笔记本和一本书，就像图10.1.1。所有纸制品都是黑白印刷的，但咖啡杯是红色的。换句话说，这个咖啡杯在这种视觉环境中是突出和显眼的，不由自主地引起人们的注意。所以我们会把视力最敏锐的地方放到咖啡上，如图10.1.1所示。

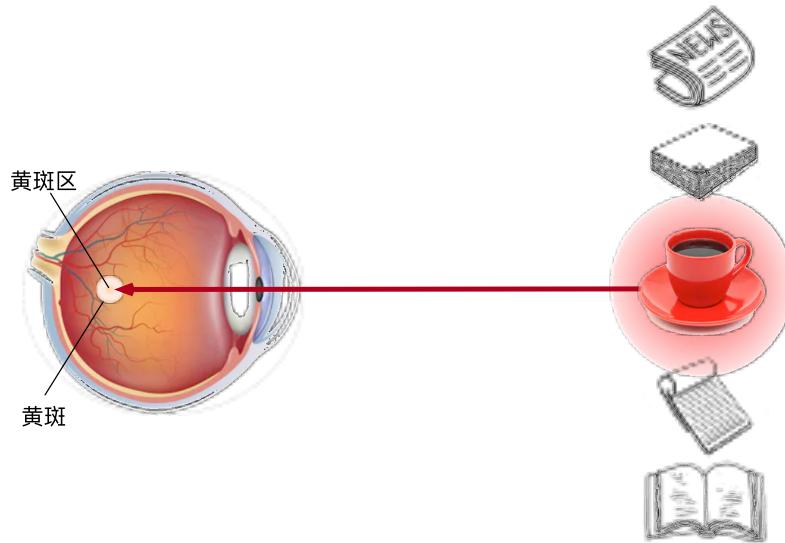


图10.1.1：由于突出性的非自主性提示（红杯子），注意力不自主地指向了咖啡杯

喝咖啡后，我们会变得兴奋并想读书，所以转过头，重新聚焦眼睛，然后看看书，就像图10.1.2中描述那样。与图10.1.1中由于突出性导致的选择不同，此时选择书是受到了认知和意识的控制，因此注意力在基于自主性提示去辅助选择时将更为谨慎。受试者的主观意愿推动，选择的力量也就更强大。

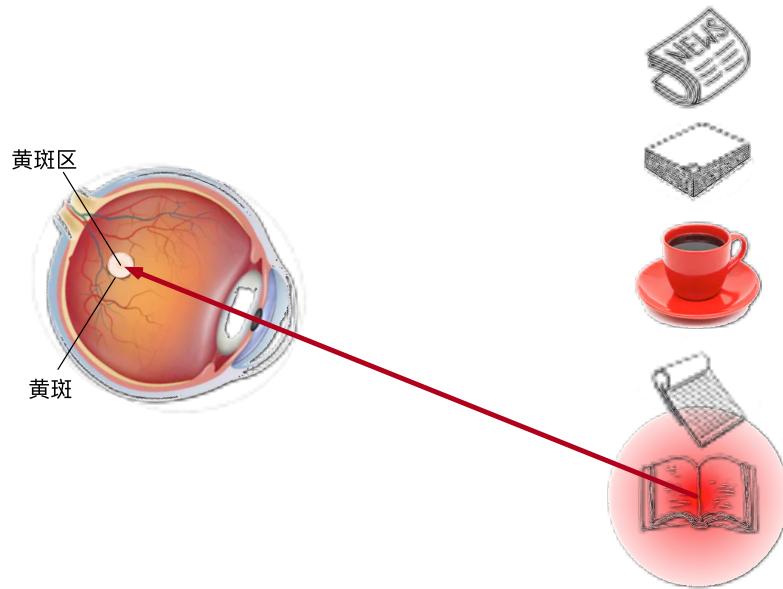


图10.1.2：依赖于任务的意志提示（想读一本书），注意力被自主引导到书上

10.1.2 查询、键和值

自主性的与非自主性的注意力提示解释了人类的注意力的方式，下面来看看如何通过这两种注意力提示，用神经网络来设计注意力机制的框架，

首先，考虑一个相对简单的状况，即只使用非自主性提示。要想将选择偏向于感官输入，则可以简单地使用参数化的全连接层，甚至是参数化的最大汇聚层或平均汇聚层。

因此，“是否包含自主性提示”将注意力机制与全连接层或汇聚层区别开来。在注意力机制的背景下，自主性提示被称为查询（query）。给定任何查询，注意力机制通过注意力汇聚（attention pooling）将选择引导至感官输入（sensory inputs，例如中间特征表示）。在注意力机制中，这些感官输入被称为值（value）。更通俗的解释，每个值都与一个键（key）配对，这可以想象为感官输入的非自主提示。如图10.1.3所示，可以通过设计注意力汇聚的方式，便于给定的查询（自主性提示）与键（非自主性提示）进行匹配，这将引导得出最匹配的值（感官输入）。

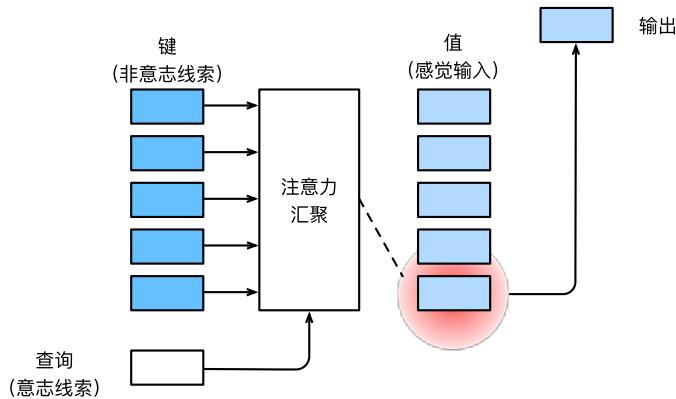


图10.1.3: 注意力机制通过注意力汇聚将查询（自主性提示）和键（非自主性提示）结合在一起，实现对值（感官输入）的选择倾向

鉴于上面所提框架在 图10.1.3中的主导地位，因此这个框架下的模型将成为本章的中心。然而，注意力机制的设计有许多替代方案。例如可以设计一个不可微的注意力模型，该模型可以使用强化学习方法 (Mnih et al., 2014)进行训练。

10.1.3 注意力的可视化

平均汇聚层可以被视为输入的加权平均值，其中各输入的权重是一样的。实际上，注意力汇聚得到的是加权平均的总和值，其中权重是在给定的查询和不同的键之间计算得出的。

```
import torch
from d2l import torch as d2l
```

为了可视化注意力权重，需要定义一个`show_heatmaps`函数。其输入`matrices`的形状是（要显示的行数，要显示的列数，查询的数目，键的数目）。

```
#@save
def show_heatmaps(matrices, xlabel, ylabel, titles=None, figsize=(2.5, 2.5),
                  cmap='Reds'):
    """显示矩阵热图"""
    d2l.use_svg_display()
    num_rows, num_cols = matrices.shape[0], matrices.shape[1]
    fig, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize,
                                 sharex=True, sharey=True, squeeze=False)
    for i, (row_axes, row_matrices) in enumerate(zip(axes, matrices)):
        for j, (ax, matrix) in enumerate(zip(row_axes, row_matrices)):
            pcm = ax.imshow(matrix.detach().numpy(), cmap=cmap)
            if i == num_rows - 1:
                ax.set_xlabel(xlabel)
```

(continues on next page)

```

if j == 0:
    ax.set_ylabel(ylabel)
if titles:
    ax.set_title(titles[j])
fig.colorbar(pcm, ax=axes, shrink=0.6);

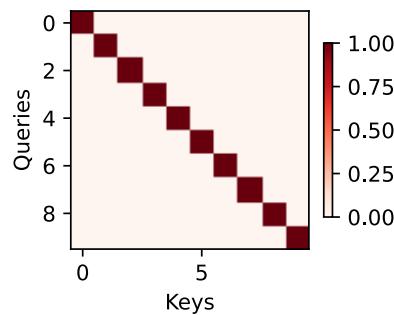
```

下面使用一个简单的例子进行演示。在本例子中，仅当查询和键相同时，注意力权重为1，否则为0。

```

attention_weights = torch.eye(10).reshape((1, 1, 10, 10))
show_heatmaps(attention_weights, xlabel='Keys', ylabel='Queries')

```



后面的章节内容将经常调用`show_heatmaps`函数来显示注意力权重。

小结

- 人类的注意力是有限的、有价值和稀缺的资源。
- 受试者使用非自主性和自主性提示有选择性地引导注意力。前者基于突出性，后者则依赖于意识。
- 注意力机制与全连接层或者汇聚层的区别源于增加的自主提示。
- 由于包含了自主性提示，注意力机制与全连接的层或汇聚层不同。
- 注意力机制通过注意力汇聚使选择偏向于值（感官输入），其中包含查询（自主性提示）和键（非自主性提示）。键和值是成对的。
- 可可视化查询和键之间的注意力权重是可行的。

练习

1. 在机器翻译中通过解码序列词元时，其自主性提示可能是什么？非自主性提示和感官输入又是什么？
2. 随机生成一个 10×10 矩阵并使用softmax运算来确保每行都是有效的概率分布，然后可视化输出注意力权重。

Discussions¹¹⁸

10.2 注意力汇聚：Nadaraya-Watson 核回归

上节介绍了框架下的注意力机制的主要成分 图10.1.3：查询（自主提示）和键（非自主提示）之间的交互形成了注意力汇聚；注意力汇聚有选择地聚合了值（感官输入）以生成最终的输出。本节将介绍注意力汇聚的更多细节，以便从宏观上了解注意力机制在实践中的运作方式。具体来说，1964年提出的Nadaraya-Watson核回归模型是一个简单但完整的例子，可以用于演示具有注意力机制的机器学习。

```
import torch
from torch import nn
from d2l import torch as d2l
```

10.2.1 生成数据集

简单起见，考虑下面这个回归问题：给定的成对的“输入—输出”数据集 $\{(x_1, y_1), \dots, (x_n, y_n)\}$ ，如何学习 f 来预测任意新输入 x 的输出 $\hat{y} = f(x)$ ？

根据下面的非线性函数生成一个人工数据集，其中加入的噪声项为 ϵ ：

$$y_i = 2 \sin(x_i) + x_i^{0.8} + \epsilon, \quad (10.2.1)$$

其中 ϵ 服从均值为 0 和标准差为 0.5 的正态分布。在这里生成了 50 个训练样本和 50 个测试样本。为了更好地可视化之后的注意力模式，需要将训练样本进行排序。

```
n_train = 50 # 训练样本数
x_train, _ = torch.sort(torch.rand(n_train) * 5) # 排序后的训练样本
```

```
def f(x):
    return 2 * torch.sin(x) + x**0.8

y_train = f(x_train) + torch.normal(0.0, 0.5, (n_train,)) # 训练样本的输出
x_test = torch.arange(0, 5, 0.1) # 测试样本
```

(continues on next page)

¹¹⁸ <https://discuss.d2l.ai/t/5764>

(continued from previous page)

```
y_truth = f(x_test) # 测试样本的真实输出  
n_test = len(x_test) # 测试样本数  
n_test
```

50

下面的函数将绘制所有的训练样本（样本由圆圈表示），不带噪声项的真实数据生成函数 f （标记为“Truth”），以及学习得到的预测函数（标记为“Pred”）。

```
def plot_kernel_reg(y_hat):  
    d2l.plot(x_test, [y_truth, y_hat], 'x', 'y', legend=['Truth', 'Pred'],  
            xlim=[0, 5], ylim=[-1, 5])  
    d2l.plt.plot(x_train, y_train, 'o', alpha=0.5);
```

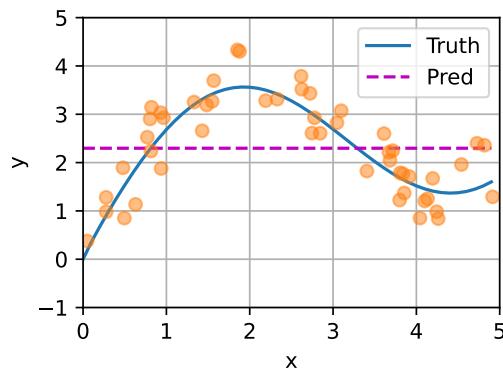
10.2.2 平均汇聚

先使用最简单的估计器来解决回归问题。基于平均汇聚来计算所有训练样本输出值的平均值：

$$f(x) = \frac{1}{n} \sum_{i=1}^n y_i, \quad (10.2.2)$$

如下图所示，这个估计器确实不够聪明。真实函数 f （“Truth”）和预测函数（“Pred”）相差很大。

```
y_hat = torch.repeat_interleave(y_train.mean(), n_test)  
plot_kernel_reg(y_hat)
```



10.2.3 非参数注意力汇聚

显然，平均汇聚忽略了输入 x_i 。于是Nadaraya (Nadaraya, 1964)和 Watson (Watson, 1964)提出了一个更好的想法，根据输入的位置对输出 y_i 进行加权：

$$f(x) = \sum_{i=1}^n \frac{K(x - x_i)}{\sum_{j=1}^n K(x - x_j)} y_i, \quad (10.2.3)$$

其中 K 是核(kernel)。公式(10.2.3)所描述的估计器被称为 Nadaraya-Watson核回归(Nadaraya-Watson kernel regression)。这里不会深入讨论核函数的细节，但受此启发，我们可以从图10.1.3中的注意力机制框架的角度重写(10.2.3)，成为一个更加通用的注意力汇聚(attention pooling)公式：

$$f(x) = \sum_{i=1}^n \alpha(x, x_i) y_i, \quad (10.2.4)$$

其中 x 是查询， (x_i, y_i) 是键值对。比较(10.2.4)和(10.2.2)，注意力汇聚是 y_i 的加权平均。将查询 x 和键 x_i 之间的关系建模为注意力权重(attention weight) $\alpha(x, x_i)$ ，如(10.2.4)所示，这个权重将被分配给每一个对应值 y_i 。对于任何查询，模型在所有键值对注意力权重都是一个有效的概率分布：它们是非负的，并且总和为1。

为了更好地理解注意力汇聚，下面考虑一个高斯核(Gaussian kernel)，其定义为：

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{u^2}{2}\right). \quad (10.2.5)$$

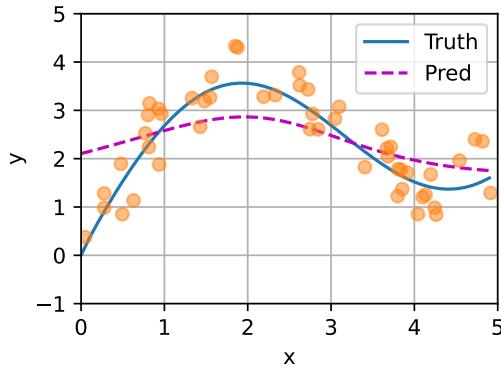
将高斯核代入(10.2.4)和(10.2.3)可以得到：

$$\begin{aligned} f(x) &= \sum_{i=1}^n \alpha(x, x_i) y_i \\ &= \sum_{i=1}^n \frac{\exp\left(-\frac{1}{2}(x - x_i)^2\right)}{\sum_{j=1}^n \exp\left(-\frac{1}{2}(x - x_j)^2\right)} y_i \\ &= \sum_{i=1}^n \text{softmax}\left(-\frac{1}{2}(x - x_i)^2\right) y_i. \end{aligned} \quad (10.2.6)$$

在(10.2.6)中，如果一个键 x_i 越是接近给定的查询 x ，那么分配给这个键对应值 y_i 的注意力权重就会越大，也就“获得了更多的注意力”。

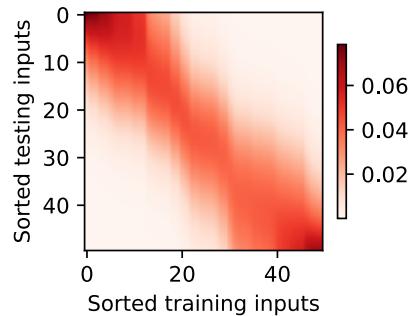
值得注意的是，Nadaraya-Watson核回归是一个非参数模型。因此，(10.2.6)是非参数的注意力汇聚(non-parametric attention pooling)模型。接下来，我们将基于这个非参数的注意力汇聚模型来绘制预测结果。从绘制的结果会发现新的模型预测线是平滑的，并且比平均汇聚的预测更接近真实。

```
# X_repeat的形状: (n_test, n_train),
# 每一行都包含着相同的测试输入（例如：同样的查询）
X_repeat = x_test.repeat_interleave(n_train).reshape((-1, n_train))
# x_train包含着键。attention_weights的形状: (n_test, n_train),
# 每一行都包含着要在给定的每个查询的值 (y_train) 之间分配的注意力权重
attention_weights = nn.functional.softmax(-(X_repeat - x_train)**2 / 2, dim=1)
# y_hat的每个元素都是值的加权平均值，其中的权重是注意力权重
y_hat = torch.matmul(attention_weights, y_train)
plot_kernel_reg(y_hat)
```



现在来观察注意力的权重。这里测试数据的输入相当于查询，而训练数据的输入相当于键。因为两个输入都是经过排序的，因此由观察可知“查询-键”对越接近，注意力汇聚的注意力权重就越高。

```
d2l.show_heatmaps(attention_weights.unsqueeze(0).unsqueeze(0),
    xlabel='Sorted training inputs',
    ylabel='Sorted testing inputs')
```



10.2.4 带参数注意力汇聚

非参数的Nadaraya-Watson核回归具有一致性（consistency）的优点：如果有足够的数据，此模型会收敛到最优结果。尽管如此，我们还是可以轻松地将可学习的参数集成到注意力汇聚中。

例如，与 (10.2.6)略有不同，在下面的查询 x 和键 x_i 之间的距离乘以可学习参数 w ：

$$\begin{aligned}
 f(x) &= \sum_{i=1}^n \alpha(x, x_i) y_i \\
 &= \sum_{i=1}^n \frac{\exp\left(-\frac{1}{2}((x - x_i)w)^2\right)}{\sum_{j=1}^n \exp\left(-\frac{1}{2}((x - x_j)w)^2\right)} y_i \\
 &= \sum_{i=1}^n \text{softmax}\left(-\frac{1}{2}((x - x_i)w)^2\right) y_i.
 \end{aligned} \tag{10.2.7}$$

本节的余下部分将通过训练这个模型 (10.2.7) 来学习注意力汇聚的参数。

批量矩阵乘法

为了更有效地计算小批量数据的注意力，我们可以利用深度学习开发框架中提供的批量矩阵乘法。

假设第一个小批量数据包含 n 个矩阵 $\mathbf{X}_1, \dots, \mathbf{X}_n$ ，形状为 $a \times b$ ，第二个小批量包含 n 个矩阵 $\mathbf{Y}_1, \dots, \mathbf{Y}_n$ ，形状为 $b \times c$ 。它们的批量矩阵乘法得到 n 个矩阵 $\mathbf{X}_1\mathbf{Y}_1, \dots, \mathbf{X}_n\mathbf{Y}_n$ ，形状为 $a \times c$ 。因此，假定两个张量的形状分别是 (n, a, b) 和 (n, b, c) ，它们的批量矩阵乘法输出的形状为 (n, a, c) 。

```
X = torch.ones((2, 1, 4))
Y = torch.ones((2, 4, 6))
torch.bmm(X, Y).shape
```

```
torch.Size([2, 1, 6])
```

在注意力机制的背景中，我们可以使用小批量矩阵乘法来计算小批量数据中的加权平均值。

```
weights = torch.ones((2, 10)) * 0.1
values = torch.arange(20.0).reshape((2, 10))
torch.bmm(weights.unsqueeze(1), values.unsqueeze(-1))
```

```
tensor([[[ 4.5000]],
       [[14.5000]]])
```

定义模型

基于 (10.2.7) 中的带参数的注意力汇聚，使用小批量矩阵乘法，定义 Nadaraya-Watson 核回归的带参数版本为：

```
class NWKernelRegression(nn.Module):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.w = nn.Parameter(torch.rand((1,), requires_grad=True))

    def forward(self, queries, keys, values):
        # queries 和 attention_weights 的形状为(查询个数, “键-值” 对个数)
        queries = queries.repeat_interleave(keys.shape[1]).reshape((-1, keys.shape[1]))
        self.attention_weights = nn.functional.softmax(
            -((queries - keys) * self.w)**2 / 2, dim=1)
        # values 的形状为(查询个数, “键-值” 对个数)
        return torch.bmm(self.attention_weights.unsqueeze(1),
                         values.unsqueeze(-1)).reshape(-1)
```

训练

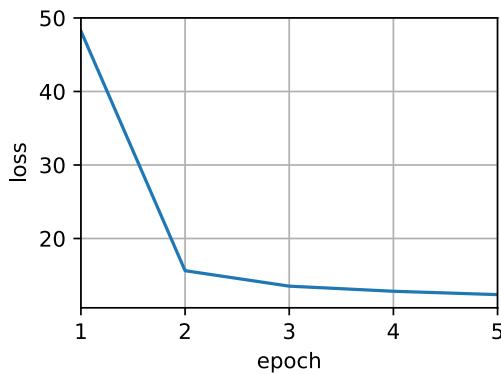
接下来，将训练数据集变换为键和值用于训练注意力模型。在带参数的注意力汇聚模型中，任何一个训练样本的输入都会和除自己以外的所有训练样本的“键一值”对进行计算，从而得到其对应的预测输出。

```
# X_tile的形状:(n_train, n_train), 每一行都包含着相同的训练输入
X_tile = x_train.repeat((n_train, 1))
# Y_tile的形状:(n_train, n_train), 每一行都包含着相同的训练输出
Y_tile = y_train.repeat((n_train, 1))
# keys的形状:('n_train', 'n_train'-1)
keys = X_tile[(1 - torch.eye(n_train)).type(torch.bool)].reshape((n_train, -1))
# values的形状:('n_train', 'n_train'-1)
values = Y_tile[(1 - torch.eye(n_train)).type(torch.bool)].reshape((n_train, -1))
```

训练带参数的注意力汇聚模型时，使用平方损失函数和随机梯度下降。

```
net = NWKernelRegression()
loss = nn.MSELoss(reduction='none')
trainer = torch.optim.SGD(net.parameters(), lr=0.5)
animator = d2l.Animator(xlabel='epoch', ylabel='loss', xlim=[1, 5])

for epoch in range(5):
    trainer.zero_grad()
    l = loss(net(x_train, keys, values), y_train)
    l.sum().backward()
    trainer.step()
    print(f'epoch {epoch + 1}, loss {float(l.sum()):.6f}')
    animator.add(epoch + 1, float(l.sum()))
```

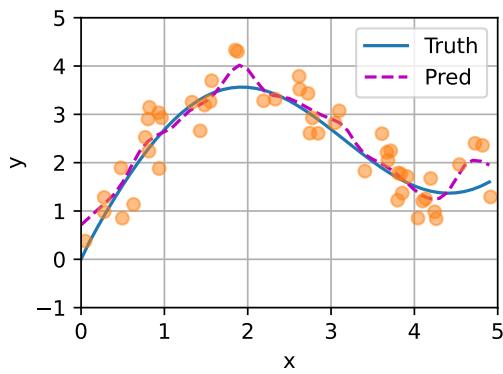


如下所示，训练完带参数的注意力汇聚模型后可以发现：在尝试拟合带噪声的训练数据时，预测结果绘制的线不如之前非参数模型的平滑。

```

# keys的形状:(n_test, n_train), 每一行包含着相同的训练输入 (例如, 相同的键)
keys = x_train.repeat((n_test, 1))
# value的形状:(n_test, n_train)
values = y_train.repeat((n_test, 1))
y_hat = net(x_test, keys, values).unsqueeze(1).detach()
plot_kernel_reg(y_hat)

```

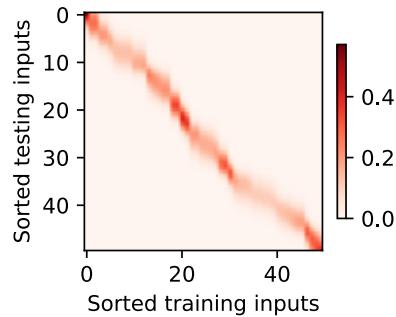


为什么新的模型更不平滑了呢？下面看一下输出结果的绘制图：与非参数的注意力汇聚模型相比，带参数的模型加入可学习的参数后，曲线在注意力权重较大的区域变得更不平滑。

```

d2l.show_heatmaps(net.attention_weights.unsqueeze(0).unsqueeze(0),
                  xlabel='Sorted training inputs',
                  ylabel='Sorted testing inputs')

```



小结

- Nadaraya-Watson核回归是具有注意力机制的机器学习范例。
- Nadaraya-Watson核回归的注意力汇聚是对训练数据中输出的加权平均。从注意力的角度来看，分配给每个值的注意力权重取决于将值所对应的键和查询作为输入的函数。
- 注意力汇聚可以分为非参数型和带参数型。

练习

1. 增加训练数据的样本数量，能否得到更好的非参数的Nadaraya-Watson核回归模型？
2. 在带参数的注意力汇聚的实验中学习得到的参数 w 的价值是什么？为什么在可视化注意力权重时，它会使加权区域更加尖锐？
3. 如何将超参数添加到非参数的Nadaraya-Watson核回归中以实现更好地预测结果？
4. 为本节的核回归设计一个新的带参数的注意力汇聚模型。训练这个新模型并可视化其注意力权重。

Discussions¹¹⁹

10.3 注意力评分函数

10.2节使用了高斯核来对查询和键之间的关系建模。(10.2.6)中的高斯核指数部分可以视为注意力评分函数(attention scoring function)，简称评分函数(scoring function)，然后把这个函数的输出结果输入到softmax函数中进行运算。通过上述步骤，将得到与键对应的值的概率分布(即注意力权重)。最后，注意力汇聚的输出就是基于这些注意力权重的值的加权和。

从宏观来看，上述算法可以用来实现图10.1.3中的注意力机制框架。图10.3.1说明了如何将注意力汇聚的输出计算成为值的加权和，其中 a 表示注意力评分函数。由于注意力权重是概率分布，因此加权和本质上是加权平均值。

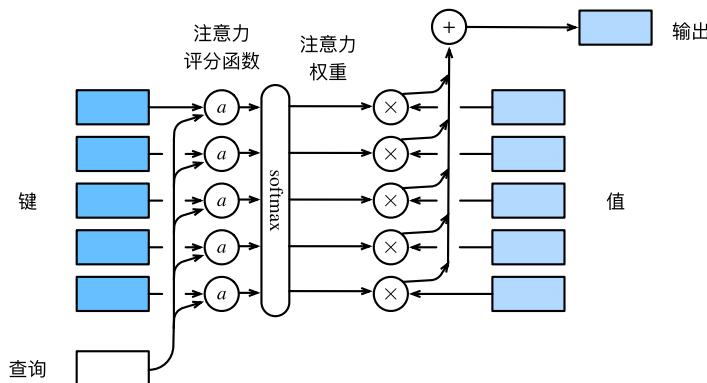


图10.3.1：计算注意力汇聚的输出为值的加权和

¹¹⁹ <https://discuss.d2l.ai/t/5760>

用数学语言描述，假设有一个查询 $\mathbf{q} \in \mathbb{R}^q$ 和 m 个“键-值”对 $(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_m, \mathbf{v}_m)$ ，其中 $\mathbf{k}_i \in \mathbb{R}^k$, $\mathbf{v}_i \in \mathbb{R}^v$ 。注意力汇聚函数 f 就被表示成值的加权和：

$$f(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_m, \mathbf{v}_m)) = \sum_{i=1}^m \alpha(\mathbf{q}, \mathbf{k}_i) \mathbf{v}_i \in \mathbb{R}^v, \quad (10.3.1)$$

其中查询 \mathbf{q} 和键 \mathbf{k}_i 的注意力权重（标量）是通过注意力评分函数 a 将两个向量映射成标量，再经过 softmax 运算得到的：

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \text{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^m \exp(a(\mathbf{q}, \mathbf{k}_j))} \in \mathbb{R}. \quad (10.3.2)$$

正如上图所示，选择不同的注意力评分函数 a 会导致不同的注意力汇聚操作。本节将介绍两个流行的评分函数，稍后将用它们来实现更复杂的注意力机制。

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

10.3.1 掩蔽softmax操作

正如上面提到的，softmax 操作用于输出一个概率分布作为注意力权重。在某些情况下，并非所有的值都应该被纳入到注意力汇聚中。例如，为了在 9.5 节中高效处理小批量数据集，某些文本序列被填充了没有意义的特殊词元。为了仅将有意义的词元作为值来获取注意力汇聚，可以指定一个有效序列长度（即词元的个数），以便在计算 softmax 时过滤掉超出指定范围的位置。下面的 `masked_softmax` 函数实现了这样的掩蔽 softmax 操作（masked softmax operation），其中任何超出有效长度的位置都被掩蔽并置为 0。

```
#@save
def masked_softmax(X, valid_lens):
    """通过在最后一个轴上掩蔽元素来执行softmax操作"""
    # X:3D张量, valid_lens:1D或2D张量
    if valid_lens is None:
        return nn.functional.softmax(X, dim=-1)
    else:
        shape = X.shape
        if valid_lens.dim() == 1:
            valid_lens = torch.repeat_interleave(valid_lens, shape[1])
        else:
            valid_lens = valid_lens.reshape(-1)
        # 最后一轴上被掩蔽的元素使用一个非常大的负值替换，从而其softmax输出为0
        X = d2l.sequence_mask(X.reshape(-1, shape[-1]), valid_lens,
                              value=-1e6)
        return nn.functional.softmax(X.reshape(shape), dim=-1)
```

为了演示此函数是如何工作的，考虑由两个 2×4 矩阵表示的样本，这两个样本的有效长度分别为2和3。经过掩蔽softmax操作，超出有效长度的值都被掩蔽为0。

```
masked_softmax(torch.rand(2, 2, 4), torch.tensor([2, 3]))
```

```
tensor([[[0.5980, 0.4020, 0.0000, 0.0000],  
        [0.5548, 0.4452, 0.0000, 0.0000]],  
  
       [[0.3716, 0.3926, 0.2358, 0.0000],  
        [0.3455, 0.3337, 0.3208, 0.0000]]])
```

同样，也可以使用二维张量，为矩阵样本中的每一行指定有效长度。

```
masked_softmax(torch.rand(2, 2, 4), torch.tensor([[1, 3], [2, 4]]))
```

```
tensor([[[1.0000, 0.0000, 0.0000, 0.0000],  
        [0.4125, 0.3273, 0.2602, 0.0000]],  
  
       [[0.5254, 0.4746, 0.0000, 0.0000],  
        [0.3117, 0.2130, 0.1801, 0.2952]]])
```

10.3.2 加性注意力

一般来说，当查询和键是不同长度的矢量时，可以使用加性注意力作为评分函数。给定查询 $\mathbf{q} \in \mathbb{R}^q$ 和键 $\mathbf{k} \in \mathbb{R}^k$ ，加性注意力（additive attention）的评分函数为

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q \mathbf{q} + \mathbf{W}_k \mathbf{k}) \in \mathbb{R}, \quad (10.3.3)$$

其中可学习的参数是 $\mathbf{W}_q \in \mathbb{R}^{h \times q}$ 、 $\mathbf{W}_k \in \mathbb{R}^{h \times k}$ 和 $\mathbf{w}_v \in \mathbb{R}^h$ 。如(10.3.3)所示，将查询和键连结起来后输入到一个多层次感知机（MLP）中，感知机包含一个隐藏层，其隐藏单元数是一个超参数 h 。通过使用tanh作为激活函数，并且禁用偏置项。

下面来实现加性注意力。

```
#@save  
class AdditiveAttention(nn.Module):  
    """加性注意力"""\n    def __init__(self, key_size, query_size, num_hiddens, dropout, **kwargs):  
        super(AdditiveAttention, self).__init__(**kwargs)  
        self.W_k = nn.Linear(key_size, num_hiddens, bias=False)  
        self.W_q = nn.Linear(query_size, num_hiddens, bias=False)  
        self.w_v = nn.Linear(num_hiddens, 1, bias=False)
```

(continues on next page)

(continued from previous page)

```
self.dropout = nn.Dropout(dropout)

def forward(self, queries, keys, values, valid_lens):
    queries, keys = self.W_q(queries), self.W_k(keys)
    # 在维度扩展后,
    # queries的形状: (batch_size, 查询的个数, 1, num_hidden)
    # key的形状: (batch_size, 1, “键-值” 对的个数, num_hiddens)
    # 使用广播方式进行求和
    features = queries.unsqueeze(2) + keys.unsqueeze(1)
    features = torch.tanh(features)
    # self.w_v仅有一个输出, 因此从形状中移除最后那个维度。
    # scores的形状: (batch_size, 查询的个数, “键-值” 对的个数)
    scores = self.W_v(features).squeeze(-1)
    self.attention_weights = masked_softmax(scores, valid_lens)
    # values的形状: (batch_size, “键-值” 对的个数, 值的维度)
    return torch.bmm(self.dropout(self.attention_weights), values)
```

用一个小例子来演示上面的AdditiveAttention类，其中查询、键和值的形状为（批量大小，步数或词元序列长度，特征大小），实际输出为(2, 1, 20)、(2, 10, 2)和(2, 10, 4)。注意力汇聚输出的形状为（批量大小，查询的步数，值的维度）。

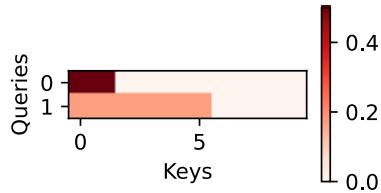
```
queries, keys = torch.normal(0, 1, (2, 1, 20)), torch.ones((2, 10, 2))
# values的小批量, 两个值矩阵是相同的
values = torch.arange(40, dtype=torch.float32).reshape(1, 10, 4).repeat(
    2, 1, 1)
valid_lens = torch.tensor([2, 6])

attention = AdditiveAttention(key_size=2, query_size=20, num_hiddens=8,
                             dropout=0.1)
attention.eval()
attention(queries, keys, values, valid_lens)
```

```
tensor([[[ 2.0000,  3.0000,  4.0000,  5.0000],
        [[10.0000, 11.0000, 12.0000, 13.0000]]], grad_fn=<BmmBackward0>)
```

尽管加性注意力包含了可学习的参数，但由于本例子中每个键都是相同的，所以注意力权重是均匀的，由指定的有效长度决定。

```
d2l.show_heatmaps(attention.attention_weights.reshape((1, 1, 2, 10)),
                   xlabel='Keys', ylabel='Queries')
```



10.3.3 缩放点积注意力

使用点积可以得到计算效率更高的评分函数，但是点积操作要求查询和键具有相同的长度 d 。假设查询和键的所有元素都是独立的随机变量，并且都满足零均值和单位方差，那么两个向量的点积的均值为0，方差为 d 。为确保无论向量长度如何，点积的方差在不考虑向量长度的情况下仍然是1，我们再将点积除以 \sqrt{d} ，则缩放点积注意力（scaled dot-product attention）评分函数为：

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k} / \sqrt{d}. \quad (10.3.4)$$

在实践中，我们通常从小批量的角度来考虑提高效率，例如基于 n 个查询和 m 个键—值对计算注意力，其中查询和键的长度为 d ，值的长度为 v 。查询 $\mathbf{Q} \in \mathbb{R}^{n \times d}$ 、键 $\mathbf{K} \in \mathbb{R}^{m \times d}$ 和值 $\mathbf{V} \in \mathbb{R}^{m \times v}$ 的缩放点积注意力是：

$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n \times v}. \quad (10.3.5)$$

下面的缩放点积注意力的实现使用了暂退法进行模型正则化。

```
#@save
class DotProductAttention(nn.Module):
    """缩放点积注意力"""
    def __init__(self, dropout, **kwargs):
        super(DotProductAttention, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)

    # queries的形状: (batch_size, 查询的个数, d)
    # keys的形状: (batch_size, “键—值” 对的个数, d)
    # values的形状: (batch_size, “键—值” 对的个数, 值的维度)
    # valid_lens的形状:(batch_size,)或者(batch_size, 查询的个数)
    def forward(self, queries, keys, values, valid_lens=None):
        d = queries.shape[-1]
        # 设置transpose_b=True为了交换keys的最后两个维度
        scores = torch.bmm(queries, keys.transpose(1, 2)) / math.sqrt(d)
        self.attention_weights = masked_softmax(scores, valid_lens)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

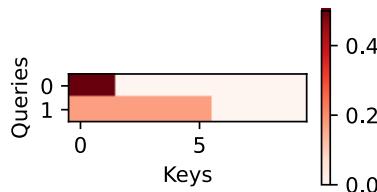
为了演示上述的DotProductAttention类，我们使用与先前加性注意力例子中相同的键、值和有效长度。对于点积操作，我们令查询的特征维度与键的特征维度大小相同。

```
queries = torch.normal(0, 1, (2, 1, 2))
attention = DotProductAttention(dropout=0.5)
attention.eval()
attention(queries, keys, values, valid_lens)
```

```
tensor([[[ 2.0000,  3.0000,  4.0000,  5.0000],
        [[10.0000, 11.0000, 12.0000, 13.0000]]])
```

与加性注意力演示相同，由于键包含的是相同的元素，而这些元素无法通过任何查询进行区分，因此获得了均匀的注意力权重。

```
d2l.show_heatmaps(attention.attention_weights.reshape((1, 1, 2, 10)),
                    xlabel='Keys', ylabel='Queries')
```



小结

- 将注意力汇聚的输出计算可以作为值的加权平均，选择不同的注意力评分函数会带来不同的注意力汇聚操作。
- 当查询和键是不同长度的矢量时，可以使用可加性注意力评分函数。当它们的长度相同时，使用缩放的“点一积”注意力评分函数的计算效率更高。

练习

- 修改小例子中的键，并且可视化注意力权重。可加性注意力和缩放的“点一积”注意力是否仍然产生相同的结果？为什么？
- 只使用矩阵乘法，能否为具有不同矢量长度的查询和键设计新的评分函数？
- 当查询和键具有相同的矢量长度时，矢量求和作为评分函数是否比“点一积”更好？为什么？

Discussions¹²⁰

¹²⁰ <https://discuss.d2l.ai/t/5752>

10.4 Bahdanau 注意力

9.7节中探讨了机器翻译问题：通过设计一个基于两个循环神经网络的编码器-解码器架构，用于序列到序列学习。具体来说，循环神经网络编码器将长度可变的序列转换为固定形状的上下文变量，然后循环神经网络解码器根据生成的词元和上下文变量按词元生成输出（目标）序列词元。然而，即使并非所有输入（源）词元都对解码某个词元都有用，在每个解码步骤中仍使用编码相同的上下文变量。有什么方法能改变上下文变量呢？

我们试着从(Graves, 2013)中找到灵感：在为给定文本序列生成手写的挑战中，Graves设计了一种可微注意力模型，将文本字符与更长的笔迹对齐，其中对齐方式仅向一个方向移动。受学习对齐想法的启发，Bahdanau等人提出了一个没有严格单向对齐限制的可微注意力模型(Bahdanau et al., 2014)。在预测词元时，如果不是所有输入词元都相关，模型将仅对齐（或参与）输入序列中与当前预测相关的部分。这是通过将上下文变量视为注意力集中的输出来实现的。

10.4.1 模型

下面描述的Bahdanau注意力模型将遵循9.7节中的相同符号表达。这个新的基于注意力的模型与9.7节中的模型相同，只不过(9.7.3)中的上下文变量 \mathbf{c} 在任何解码时间步 t' 都会被 $\mathbf{c}_{t'}$ 替换。假设输入序列中有 T 个词元，解码时间步 t' 的上下文变量是注意力集中的输出：

$$\mathbf{c}_{t'} = \sum_{t=1}^T \alpha(\mathbf{s}_{t'-1}, \mathbf{h}_t) \mathbf{h}_t, \quad (10.4.1)$$

其中，时间步 $t'-1$ 时的解码器隐状态 $\mathbf{s}_{t'-1}$ 是查询，编码器隐状态 \mathbf{h}_t 既是键，也是值，注意力权重 α 是使用(10.3.2)所定义的加性注意力打分函数计算的。

与图9.7.2中的循环神经网络编码器-解码器架构略有不同，图10.4.1描述了Bahdanau注意力的架构。

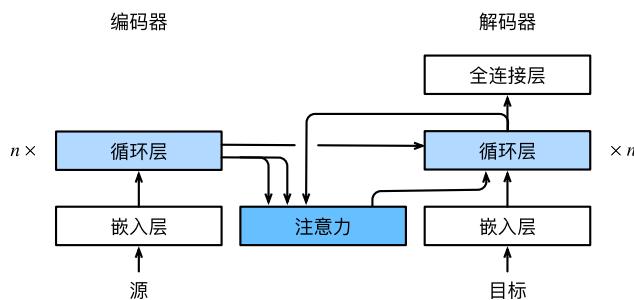


图10.4.1：一个带有Bahdanau注意力的循环神经网络编码器-解码器模型

```
import torch
from torch import nn
from d2l import torch as d2l
```

10.4.2 定义注意力解码器

下面看看如何定义Bahdanau注意力，实现循环神经网络编码器-解码器。其实，我们只需重新定义解码器即可。为了更方便地显示学习的注意力权重，以下AttentionDecoder类定义了带有注意力机制解码器的基本接口。

```
#@save
class AttentionDecoder(d2l.Decoder):
    """带有注意力机制解码器的基本接口"""
    def __init__(self, **kwargs):
        super(AttentionDecoder, self).__init__(**kwargs)

    @property
    def attention_weights(self):
        raise NotImplementedError
```

接下来，让我们在接下来的Seq2SeqAttentionDecoder类中实现带有Bahdanau注意力的循环神经网络解码器。首先，初始化解码器的状态，需要下面的输入：

1. 编码器在所有时间步的最终层隐状态，将作为注意力的键和值；
2. 上一时间步的编码器全层隐状态，将作为初始化解码器的隐状态；
3. 编码器有效长度（排除在注意力池中填充词元）。

在每个解码时间步骤中，解码器上一个时间步的最终层隐状态将用作查询。因此，注意力输出和输入嵌入都连结为循环神经网络解码器的输入。

```
class Seq2SeqAttentionDecoder(AttentionDecoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqAttentionDecoder, self).__init__(**kwargs)
        self.attention = d2l.AdditiveAttention(
            num_hiddens, num_hiddens, num_hiddens, dropout)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(
            embed_size + num_hiddens, num_hiddens, num_layers,
            dropout=dropout)
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens, *args):
        # outputs的形状为(batch_size, num_steps, num_hiddens).
        # hidden_state的形状为(num_layers, batch_size, num_hiddens)
        outputs, hidden_state = enc_outputs
        return (outputs.permute(1, 0, 2), hidden_state, enc_valid_lens)
```

(continues on next page)

```

def forward(self, X, state):
    # enc_outputs的形状为(batch_size,num_steps,num_hiddens).
    # hidden_state的形状为(num_layers,batch_size,
    # num_hiddens)
    enc_outputs, hidden_state, enc_valid_lens = state
    # 输出X的形状为(num_steps,batch_size,embed_size)
    X = self.embedding(X).permute(1, 0, 2)
    outputs, self._attention_weights = [], []
    for x in X:
        # query的形状为(batch_size,1,num_hiddens)
        query = torch.unsqueeze(hidden_state[-1], dim=1)
        # context的形状为(batch_size,1,num_hiddens)
        context = self.attention(
            query, enc_outputs, enc_outputs, enc_valid_lens)
        # 在特征维度上连结
        x = torch.cat((context, torch.unsqueeze(x, dim=1)), dim=-1)
        # 将x变形为(1,batch_size,embed_size+num_hiddens)
        out, hidden_state = self.rnn(x.permute(1, 0, 2), hidden_state)
        outputs.append(out)
        self._attention_weights.append(self.attention.attention_weights)
    # 全连接层变换后, outputs的形状为
    # (num_steps,batch_size,vocab_size)
    outputs = self.dense(torch.cat(outputs, dim=0))
    return outputs.permute(1, 0, 2), [enc_outputs, hidden_state,
                                         enc_valid_lens]

@property
def attention_weights(self):
    return self._attention_weights

```

接下来，使用包含7个时间步的4个序列输入的小批量测试Bahdanau注意力解码器。

```

encoder = d2l.Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
                               num_layers=2)
encoder.eval()
decoder = Seq2SeqAttentionDecoder(vocab_size=10, embed_size=8, num_hiddens=16,
                                   num_layers=2)
decoder.eval()
X = torch.zeros((4, 7), dtype=torch.long)  # (batch_size,num_steps)
state = decoder.init_state(encoder(X), None)
output, state = decoder(X, state)
output.shape, len(state), state[0].shape, len(state[1]), state[1][0].shape

```

```
(torch.Size([4, 7, 10]), 3, torch.Size([4, 7, 16]), 2, torch.Size([4, 16]))
```

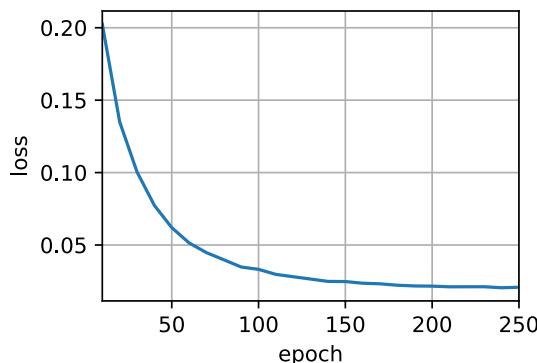
10.4.3 训练

与 9.7.4 节类似，我们在这里指定超参数，实例化一个带有 Bahdanau 注意力的编码器和解码器，并对这个模型进行机器翻译训练。由于新增的注意力机制，训练要比没有注意力机制的 9.7.4 节慢得多。

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.1
batch_size, num_steps = 64, 10
lr, num_epochs, device = 0.005, 250, d2l.try_gpu()

train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)
encoder = d2l.Seq2SeqEncoder(
    len(src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqAttentionDecoder(
    len(tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
net = d2l.EncoderDecoder(encoder, decoder)
d2l.train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.021, 4948.7 tokens/sec on cuda:0
```



模型训练后，我们用它将几个英语句子翻译成法语并计算它们的 BLEU 分数。

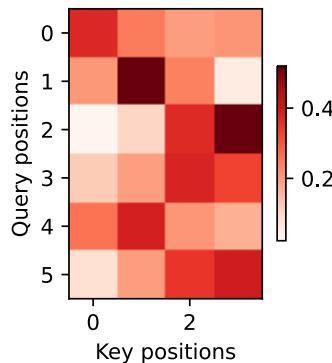
```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, dec_attention_weight_seq = d2l.predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device, True)
    print(f'{eng} => {translation}, '
          f'bleu {d2l.bleu(translation, fra, k=2):.3f}')
```

```
go . => va !, bleu 1.000
i lost . => j'ai perdu ., bleu 1.000
he's calm . => il est paresseux ., bleu 0.658
i'm home . => je suis chez moi ., bleu 1.000
```

```
attention_weights = torch.cat([step[0][0][0] for step in dec_attention_weight_seq], 0).reshape((1, 1, -1, num_steps))
```

训练结束后，下面通过可视化注意力权重会发现，每个查询都会在键值对上分配不同的权重，这说明在每个解码步中，输入序列的不同部分被选择性地聚集在注意力池中。

```
# 加上一个包含序列结束词元
d2l.show_heatmaps(
    attention_weights[:, :, :, :len(engs[-1].split()) + 1].cpu(),
    xlabel='Key positions', ylabel='Query positions')
```



小结

- 在预测词元时，如果不是所有输入词元都是相关的，那么具有Bahdanau注意力的循环神经网络编码器-解码器会有选择地统计输入序列的不同部分。这是通过将上下文变量视为加性注意力池化的输出来实现的。
- 在循环神经网络编码器-解码器中，Bahdanau注意力将上一时间步的解码器隐状态视为查询，在所有时间步的编码器隐状态同时视为键和值。

练习

1. 在实验中用LSTM替换GRU。
2. 修改实验以将加性注意力打分函数替换为缩放点积注意力，它如何影响训练效率？

Discussions¹²¹

10.5 多头注意力

在实践中，当给定相同的查询、键和值的集合时，我们希望模型可以基于相同的注意力机制学习到不同的行为，然后将不同的行为作为知识组合起来，捕获序列内各种范围的依赖关系（例如，短距离依赖和长距离依赖关系）。因此，允许注意力机制组合使用查询、键和值的不同子空间表示（representation subspaces）可能是有益的。

为此，与其只使用单独一个注意力汇聚，我们可以用独立学习得到的 h 组不同的线性投影（linear projections）来变换查询、键和值。然后，这 h 组变换后的查询、键和值将并行地送到注意力汇聚中。最后，将这 h 个注意力汇聚的输出拼接在一起，并且通过另一个可以学习的线性投影进行变换，以产生最终输出。这种设计被称为多头注意力（multihead attention）(Vaswani et al., 2017)。对于 h 个注意力汇聚输出，每一个注意力汇聚都被称作一个头（head）。图10.5.1展示了使用全连接层来实现可学习的线性变换的多头注意力。

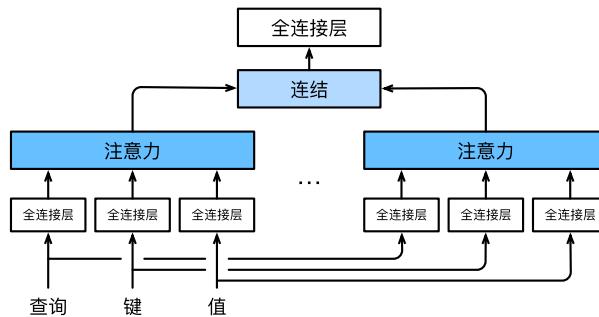


图10.5.1: 多头注意力：多个头连结然后线性变换

10.5.1 模型

在实现多头注意力之前，让我们用数学语言将这个模型形式化地描述出来。给定查询 $\mathbf{q} \in \mathbb{R}^{d_q}$ 、键 $\mathbf{k} \in \mathbb{R}^{d_k}$ 和值 $\mathbf{v} \in \mathbb{R}^{d_v}$ ，每个注意力头 \mathbf{h}_i ($i = 1, \dots, h$) 的计算方法为：

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)} \mathbf{q}, \mathbf{W}_i^{(k)} \mathbf{k}, \mathbf{W}_i^{(v)} \mathbf{v}) \in \mathbb{R}^{p_v}, \quad (10.5.1)$$

其中，可学习的参数包括 $\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$ 、 $\mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$ 和 $\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$ ，以及代表注意力汇聚的函数 f 。 f 可以是 10.3 节中的加性注意力和缩放点积注意力。多头注意力的输出需要经过另一个线性转换，它对应着 h 个

¹²¹ <https://discuss.d2l.ai/t/5754>

头连结后的结果，因此其可学习参数是 $\mathbf{W}_o \in \mathbb{R}^{p_o \times h p_v}$ ：

$$\mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix} \in \mathbb{R}^{p_o}. \quad (10.5.2)$$

基于这种设计，每个头都可能会关注输入的不同部分，可以表示比简单加权平均值更复杂的函数。

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

10.5.2 实现

在实现过程中通常选择缩放点积注意力作为每一个注意力头。为了避免计算代价和参数代价的大幅增长，我们设定 $p_q = p_k = p_v = p_o/h$ 。值得注意的是，如果将查询、键和值的线性变换的输出数量设置为 $p_q h = p_k h = p_v h = p_o$ ，则可以并行计算 h 个头。在下面的实现中， p_o 是通过参数 `num_hiddens` 指定的。

```
#@save
class MultiHeadAttention(nn.Module):
    """多头注意力"""
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 num_heads, dropout, bias=False, **kwargs):
        super(MultiHeadAttention, self).__init__(**kwargs)
        self.num_heads = num_heads
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.Linear(query_size, num_hiddens, bias=bias)
        self.W_k = nn.Linear(key_size, num_hiddens, bias=bias)
        self.W_v = nn.Linear(value_size, num_hiddens, bias=bias)
        self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

    def forward(self, queries, keys, values, valid_lens):
        # queries, keys, values 的形状:
        # (batch_size, 查询或者“键—值”对的个数, num_hiddens)
        # valid_lens 的形状:
        # (batch_size,) 或 (batch_size, 查询的个数)
        # 经过变换后, outputs 的形状:
        # (batch_size * num_heads, 查询或者“键—值”对的个数,
        # num_hiddens / num_heads)
        queries = transpose_qkv(self.W_q(queries), self.num_heads)
        keys = transpose_qkv(self.W_k(keys), self.num_heads)
        values = transpose_qkv(self.W_v(values), self.num_heads)
```

(continues on next page)

```

if valid_lens is not None:
    # 在轴0, 将第一项(标量或者矢量)复制num_heads次,
    # 然后如此复制第二项, 然后诸如此类。
    valid_lens = torch.repeat_interleave(
        valid_lens, repeats=self.num_heads, dim=0)

    # output的形状:(batch_size*num_heads, 查询的个数,
    # num_hiddens/num_heads)
    output = self.attention(queries, keys, values, valid_lens)

    # output_concat的形状:(batch_size, 查询的个数, num_hiddens)
    output_concat = transpose_output(output, self.num_heads)
return self.W_o(output_concat)

```

为了能够使多个头并行计算,上面的MultiHeadAttention类将使用下面定义的两个转置函数。具体来说,transpose_output函数反转了transpose_qkv函数的操作。

```

#@save
def transpose_qkv(X, num_heads):
    """为了多注意力头的并行计算而变换形状"""
    # 输入X的形状:(batch_size, 查询或者“键-值”对的个数, num_hiddens)
    # 输出X的形状:(batch_size, 查询或者“键-值”对的个数, num_heads,
    # num_hiddens/num_heads)
    X = X.reshape(X.shape[0], X.shape[1], num_heads, -1)

    # 输出X的形状:(batch_size, num_heads, 查询或者“键-值”对的个数,
    # num_hiddens/num_heads)
    X = X.permute(0, 2, 1, 3)

    # 最终输出的形状:(batch_size*num_heads, 查询或者“键-值”对的个数,
    # num_hiddens/num_heads)
    return X.reshape(-1, X.shape[2], X.shape[3])

#@save
def transpose_output(X, num_heads):
    """逆转transpose_qkv函数的操作"""
    X = X.reshape(-1, num_heads, X.shape[1], X.shape[2])
    X = X.permute(0, 2, 1, 3)
    return X.reshape(X.shape[0], X.shape[1], -1)

```

下面使用键和值相同的小例子来测试我们编写的MultiHeadAttention类。多头注意力输出的形状是

(batch_size, num_queries, num_hiddens)。

```
num_hiddens, num_heads = 100, 5
attention = MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
                               num_hiddens, num_heads, 0.5)
attention.eval()
```

```
MultiHeadAttention(
    (attention): DotProductAttention(
        (dropout): Dropout(p=0.5, inplace=False)
    )
    (W_q): Linear(in_features=100, out_features=100, bias=False)
    (W_k): Linear(in_features=100, out_features=100, bias=False)
    (W_v): Linear(in_features=100, out_features=100, bias=False)
    (W_o): Linear(in_features=100, out_features=100, bias=False)
)
```

```
batch_size, num_queries = 2, 4
num_kv_pairs, valid_lens = 6, torch.tensor([3, 2])
X = torch.ones((batch_size, num_queries, num_hiddens))
Y = torch.ones((batch_size, num_kv_pairs, num_hiddens))
attention(X, Y, Y, valid_lens).shape
```

```
torch.Size([2, 4, 100])
```

小结

- 多头注意力融合了来自于多个注意力汇聚的不同知识，这些知识的不同来源于相同的查询、键和值的不同的子空间表示。
- 基于适当的张量操作，可以实现多头注意力的并行计算。

练习

1. 分别可视化这个实验中的多个头的注意力权重。
2. 假设有一个完成训练的基于多头注意力的模型，现在希望修剪最不重要的注意力头以提高预测速度。如何设计实验来衡量注意力头的重要性呢？

Discussions¹²²

¹²² <https://discuss.d2l.ai/t/5758>

10.6 自注意力和位置编码

在深度学习中，经常使用卷积神经网络（CNN）或循环神经网络（RNN）对序列进行编码。想象一下，有了注意力机制之后，我们将词元序列输入注意力池化中，以便同一组词元同时充当查询、键和值。具体来说，每个查询都会关注所有的键—值对并生成一个注意力输出。由于查询、键和值来自同一组输入，因此被称为自注意力（self-attention）（Lin *et al.*, 2017, Vaswani *et al.*, 2017），也被称为内部注意力（intra-attention）（Cheng *et al.*, 2016, Parikh *et al.*, 2016, Paulus *et al.*, 2017）。本节将使用自注意力进行序列编码，以及如何使用序列的顺序作为补充信息。

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

10.6.1 自注意力

给定一个由词元组成的输入序列 $\mathbf{x}_1, \dots, \mathbf{x}_n$ ，其中任意 $\mathbf{x}_i \in \mathbb{R}^d$ ($1 \leq i \leq n$)。该序列的自注意力输出为一个长度相同的序列 $\mathbf{y}_1, \dots, \mathbf{y}_n$ ，其中：

$$\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \dots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d \quad (10.6.1)$$

根据 (10.2.4) 中定义的注意力汇聚函数 f 。下面的代码片段是基于多头注意力对一个张量完成自注意力的计算，张量的形状为（批量大小，时间步的数目或词元序列的长度， d ）。输出与输入的张量形状相同。

```
num_hiddens, num_heads = 100, 5
attention = d2l.MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
                                    num_hiddens, num_heads, 0.5)
attention.eval()
```

```
MultiHeadAttention(
    (attention): DotProductAttention(
        (dropout): Dropout(p=0.5, inplace=False)
    )
    (W_q): Linear(in_features=100, out_features=100, bias=False)
    (W_k): Linear(in_features=100, out_features=100, bias=False)
    (W_v): Linear(in_features=100, out_features=100, bias=False)
    (W_o): Linear(in_features=100, out_features=100, bias=False)
)
```

```

batch_size, num_queries, valid_lens = 2, 4, torch.tensor([3, 2])
X = torch.ones((batch_size, num_queries, num_hiddens))
attention(X, X, X, valid_lens).shape

```

```
torch.Size([2, 4, 100])
```

10.6.2 比较卷积神经网络、循环神经网络和自注意力

接下来比较下面几个架构，目标都是将由 n 个词元组成的序列映射到另一个长度相等的序列，其中的每个输入词元或输出词元都由 d 维向量表示。具体来说，将比较的是卷积神经网络、循环神经网络和自注意力这几个架构的计算复杂性、顺序操作和最大路径长度。请注意，顺序操作会妨碍并行计算，而任意的序列位置组合之间的路径越短，则能更轻松地学习序列中的远距离依赖关系 (Hochreiter *et al.*, 2001)。

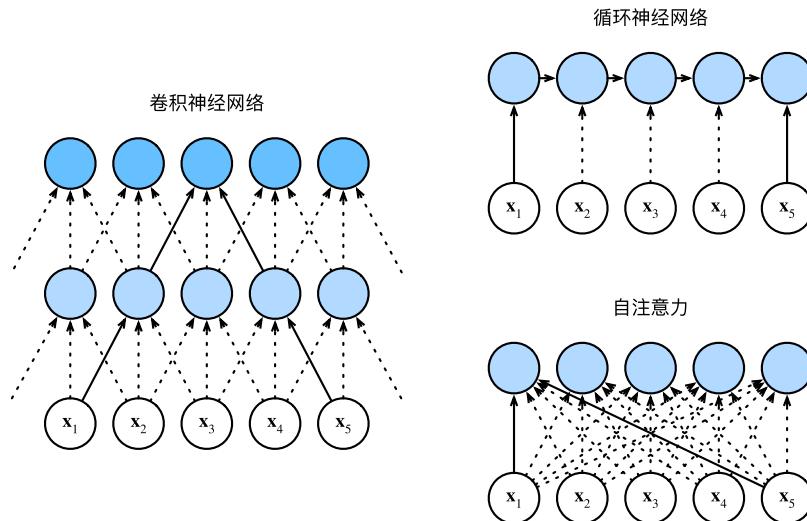


图10.6.1: 比较卷积神经网络（填充词元被忽略）、循环神经网络和自注意力三种架构

考虑一个卷积核大小为 k 的卷积层。在后面的章节将提供关于使用卷积神经网络处理序列的更多详细信息。目前只需要知道的是，由于序列长度是 n ，输入和输出的通道数量都是 d ，所以卷积层的计算复杂度为 $\mathcal{O}(knd^2)$ 。如图10.6.1所示，卷积神经网络是分层的，因此为有 $\mathcal{O}(1)$ 个顺序操作，最大路径长度为 $\mathcal{O}(n/k)$ 。例如， x_1 和 x_5 处于图10.6.1中卷积核大小为3的双层卷积神经网络的感受野内。

当更新循环神经网络的隐状态时， $d \times d$ 权重矩阵和 d 维隐状态的乘法计算复杂度为 $\mathcal{O}(d^2)$ 。由于序列长度为 n ，因此循环神经网络层的计算复杂度为 $\mathcal{O}(nd^2)$ 。根据图10.6.1，有 $\mathcal{O}(n)$ 个顺序操作无法并行化，最大路径长度也是 $\mathcal{O}(n)$ 。

在自注意力中，查询、键和值都是 $n \times d$ 矩阵。考虑(10.3.5)中缩放的“点—积”注意力，其中 $n \times d$ 矩阵乘以 $d \times n$ 矩阵。之后输出的 $n \times n$ 矩阵乘以 $n \times d$ 矩阵。因此，自注意力具有 $\mathcal{O}(n^2d)$ 计算复杂性。正如在图10.6.1中

所讲，每个词元都通过自注意力直接连接到任何其他词元。因此，有 $\mathcal{O}(1)$ 个顺序操作可以并行计算，最大路径长度也是 $\mathcal{O}(1)$ 。

总而言之，卷积神经网络和自注意力都拥有并行计算的优势，而且自注意力的最大路径长度最短。但是因为其计算复杂度是关于序列长度的二次方，所以在很长的序列中计算会非常慢。

10.6.3 位置编码

在处理词元序列时，循环神经网络是逐个的重复地处理词元的，而自注意力则因为并行计算而放弃了顺序操作。为了使用序列的顺序信息，通过在输入表示中添加位置编码（positional encoding）来注入绝对的或相对的位置信息。位置编码可以通过学习得到也可以直接固定得到。接下来描述的是基于正弦函数和余弦函数的固定位置编码 (Vaswani *et al.*, 2017)。

假设输入表示 $\mathbf{X} \in \mathbb{R}^{n \times d}$ 包含一个序列中 n 个词元的 d 维嵌入表示。位置编码使用相同形状的位置嵌入矩阵 $\mathbf{P} \in \mathbb{R}^{n \times d}$ 输出 $\mathbf{X} + \mathbf{P}$ ，矩阵第 i 行、第 $2j$ 列和 $2j + 1$ 列上的元素为：

$$\begin{aligned} p_{i,2j} &= \sin\left(\frac{i}{10000^{2j/d}}\right), \\ p_{i,2j+1} &= \cos\left(\frac{i}{10000^{2j/d}}\right). \end{aligned} \quad (10.6.2)$$

乍一看，这种基于三角函数的设计看起来很奇怪。在解释这个设计之前，让我们先在下面的PositionalEncoding类中实现它。

```
#@save
class PositionalEncoding(nn.Module):
    """位置编码"""
    def __init__(self, num_hiddens, dropout, max_len=1000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(dropout)
        # 创建一个足够长的P
        self.P = torch.zeros((1, max_len, num_hiddens))
        X = torch.arange(max_len, dtype=torch.float32).reshape(
            -1, 1) / torch.pow(10000, torch.arange(
                0, num_hiddens, 2, dtype=torch.float32) / num_hiddens)
        self.P[:, :, 0::2] = torch.sin(X)
        self.P[:, :, 1::2] = torch.cos(X)

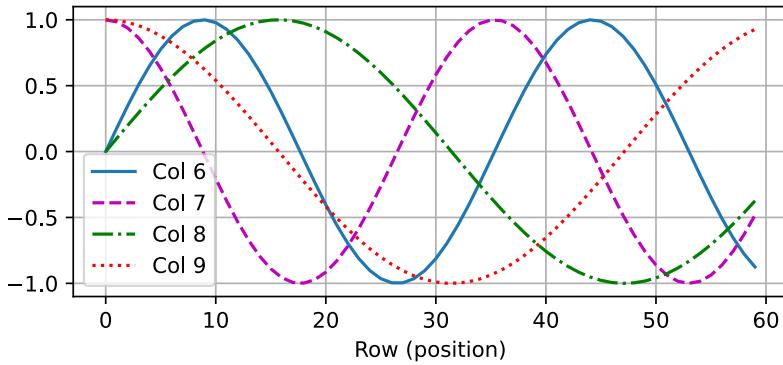
    def forward(self, X):
        X = X + self.P[:, :X.shape[1], :].to(X.device)
        return self.dropout(X)
```

在位置嵌入矩阵 \mathbf{P} 中，行代表词元在序列中的位置，列代表位置编码的不同维度。从下面的例子中可以看到位置嵌入矩阵的第6列和第7列的频率高于第8列和第9列。第6列和第7列之间的偏移量（第8列和第9列相同）是由于正弦函数和余弦函数的交替。

```

encoding_dim, num_steps = 32, 60
pos_encoding = PositionalEncoding(encoding_dim, 0)
pos_encoding.eval()
X = pos_encoding(torch.zeros((1, num_steps, encoding_dim)))
P = pos_encoding.P[:, :X.shape[1], :]
d2l.plot(torch.arange(num_steps), P[0, :, 6:10].T, xlabel='Row (position)',
         figsize=(6, 2.5), legend=["Col %d" % d for d in torch.arange(6, 10)])

```



绝对位置信息

为了明白沿着编码维度单调降低的频率与绝对位置信息的关系，让我们打印出 $0, 1, \dots, 7$ 的二进制表示形式。正如所看到的，每个数字、每两个数字和每四个数字上的比特值在第一个最低位、第二个最低位和第三个最低位上分别交替。

```

for i in range(8):
    print(f'{i}的二进制是: {i:>03b}')

```

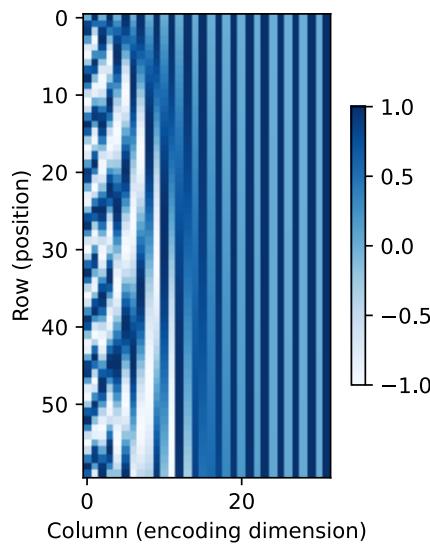
```

0的二进制是: 000
1的二进制是: 001
2的二进制是: 010
3的二进制是: 011
4的二进制是: 100
5的二进制是: 101
6的二进制是: 110
7的二进制是: 111

```

在二进制表示中，较高比特位的交替频率低于较低比特位，与下面的热图所示相似，只是位置编码通过使用三角函数在编码维度上降低频率。由于输出是浮点数，因此此类连续表示比二进制表示法更节省空间。

```
P = P[0, :, :].unsqueeze(0).unsqueeze(0)
d2l.show_heatmaps(P, xlabel='Column (encoding dimension)',
                   ylabel='Row (position)', figsize=(3.5, 4), cmap='Blues')
```



相对位置信息

除了捕获绝对位置信息之外，上述的位置编码还允许模型学习得到输入序列中相对位置信息。这是因为对于任何确定的位置偏移 δ ，位置 $i + \delta$ 处的位置编码可以线性投影位置 i 处的位置编码来表示。

这种投影的数学解释是，令 $\omega_j = 1/10000^{2j/d}$ ，对于任何确定的位置偏移 δ ，(10.6.2)中的任何一对 $(p_{i,2j}, p_{i,2j+1})$ 都可以线性投影到 $(p_{i+\delta,2j}, p_{i+\delta,2j+1})$ ：

$$\begin{aligned}
& \begin{bmatrix} \cos(\delta\omega_j) & \sin(\delta\omega_j) \\ -\sin(\delta\omega_j) & \cos(\delta\omega_j) \end{bmatrix} \begin{bmatrix} p_{i,2j} \\ p_{i,2j+1} \end{bmatrix} \\
&= \begin{bmatrix} \cos(\delta\omega_j) \sin(i\omega_j) + \sin(\delta\omega_j) \cos(i\omega_j) \\ -\sin(\delta\omega_j) \sin(i\omega_j) + \cos(\delta\omega_j) \cos(i\omega_j) \end{bmatrix} \\
&= \begin{bmatrix} \sin((i+\delta)\omega_j) \\ \cos((i+\delta)\omega_j) \end{bmatrix} \\
&= \begin{bmatrix} p_{i+\delta,2j} \\ p_{i+\delta,2j+1} \end{bmatrix},
\end{aligned} \tag{10.6.3}$$

2×2 投影矩阵不依赖于任何位置的索引 i 。

小结

- 在自注意力中，查询、键和值都来自同一组输入。
- 卷积神经网络和自注意力都拥有并行计算的优势，而且自注意力的最大路径长度最短。但是因为其计算复杂度是关于序列长度的二次方，所以在很长的序列中计算会非常慢。
- 为了使用序列的顺序信息，可以通过在输入表示中添加位置编码，来注入绝对的或相对的位置信息。

练习

1. 假设设计一个深度架构，通过堆叠基于位置编码的自注意力层来表示序列。可能会存在什么问题？
2. 请设计一种可学习的位置编码方法。

Discussions¹²³

10.7 Transformer

10.6.2节中比较了卷积神经网络（CNN）、循环神经网络（RNN）和自注意力（self-attention）。值得注意的是，自注意力同时具有并行计算和最短的最大路径长度这两个优势。因此，使用自注意力来设计深度架构是很有吸引力的。对比之前仍然依赖循环神经网络实现输入表示的自注意力模型（Cheng *et al.*, 2016, Lin *et al.*, 2017, Paulus *et al.*, 2017），Transformer模型完全基于注意力机制，没有任何卷积层或循环神经网络层（Vaswani *et al.*, 2017）。尽管Transformer最初是应用于在文本数据上的序列到序列学习，但现在已经推广到各种现代的深度学习中，例如语言、视觉、语音和强化学习领域。

10.7.1 模型

Transformer作为编码器—解码器架构的一个实例，其整体架构图在图10.7.1中展示。正如所见到的，Transformer是由编码器和解码器组成的。与图10.4.1中基于Bahdanau注意力实现的序列到序列的学习相比，Transformer的编码器和解码器是基于自注意力的模块叠加而成的，源（输入）序列和目标（输出）序列的嵌入（embedding）表示将加上位置编码（positional encoding），再分别输入到编码器和解码器中。

¹²³ <https://discuss.d2l.ai/t/5762>

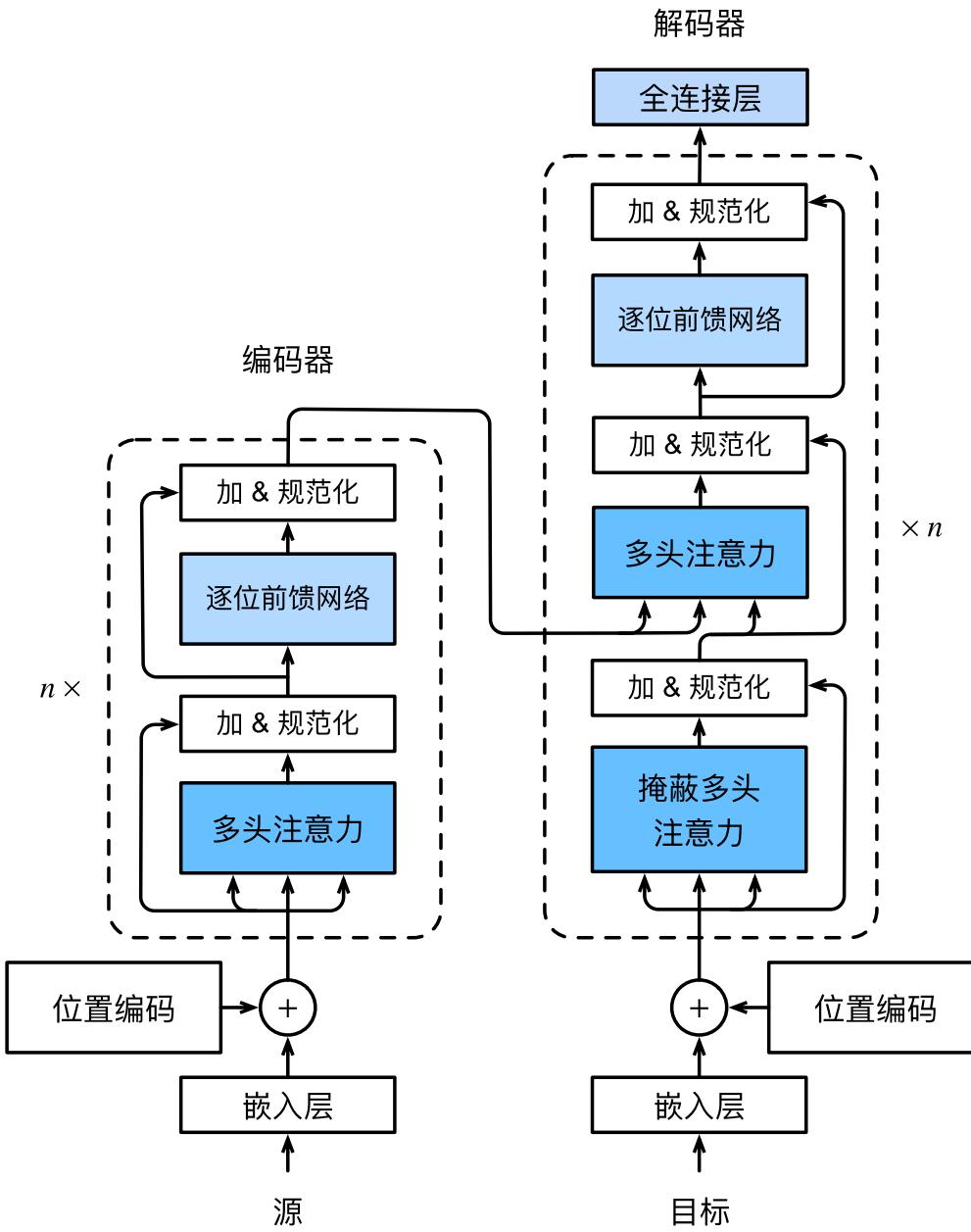


图10.7.1: transformer架构

图10.7.1中概述了Transformer的架构。从宏观角度来看，Transformer的编码器是由多个相同的层叠加而成的，每个层都有两个子层（子层表示为sublayer）。第一个子层是多头自注意力（multi-head self-attention）汇聚；第二个子层是基于位置的前馈网络（positionwise feed-forward network）。具体来说，在计算编码器的自注意力时，查询、键和值都来自前一个编码器层的输出。受7.6节中残差网络的启发，每个子层都采用了残差连接（residual connection）。在Transformer中，对于序列中任何位置的任何输入 $\mathbf{x} \in \mathbb{R}^d$ ，都要求满足 $\text{sublayer}(\mathbf{x}) \in \mathbb{R}^d$ ，以便残差连接满足 $\mathbf{x} + \text{sublayer}(\mathbf{x}) \in \mathbb{R}^d$ 。在残差连接的加法计算之后，紧接着应用层规范化（layer normalization）(Ba et al., 2016)。因此，输入序列对应的每个位置，Transformer编码器都将

输出一个 d 维表示向量。

Transformer解码器也是由多个相同的层叠加而成的，并且层中使用了残差连接和层规范化。除了编码器中描述的两个子层之外，解码器还在这两个子层之间插入了第三个子层，称为编码器—解码器注意力（encoder-decoder attention）层。在编码器—解码器注意力中，查询来自前一个解码器层的输出，而键和值来自整个编码器的输出。在解码器自注意力中，查询、键和值都来自上一个解码器层的输出。但是，解码器中的每个位置只能考虑该位置之前的所有位置。这种掩蔽（masked）注意力保留了自回归（auto-regressive）属性，确保预测仅依赖于已生成的输出词元。

在此之前已经描述并实现了基于缩放点积多头注意力 10.5节和位置编码 10.6.3节。接下来将实现Transformer模型的剩余部分。

```
import math
import pandas as pd
import torch
from torch import nn
from d2l import torch as d2l
```

10.7.2 基于位置的前馈网络

基于位置的前馈网络对序列中的所有位置的表示进行变换时使用的是同一个多层次感知机（MLP），这就是称前馈网络是基于位置的（positionwise）的原因。在下面的实现中，输入 X 的形状（批量大小，时间步数或序列长度，隐单元数或特征维度）将被一个两层的感知机转换成形状为（批量大小，时间步数， ffn_num_outputs ）的输出张量。

```
#@save
class PositionWiseFFN(nn.Module):
    """基于位置的前馈网络"""
    def __init__(self, ffn_num_input, ffn_num_hiddens, ffn_num_outputs,
                 **kwargs):
        super(PositionWiseFFN, self).__init__(**kwargs)
        self.dense1 = nn.Linear(ffn_num_input, ffn_num_hiddens)
        self.relu = nn.ReLU()
        self.dense2 = nn.Linear(ffn_num_hiddens, ffn_num_outputs)

    def forward(self, X):
        return self.dense2(self.relu(self.dense1(X)))
```

下面的例子显示，改变张量的最里层维度的尺寸，会改变成基于位置的前馈网络的输出尺寸。因为用同一个多层次感知机对所有位置上的输入进行变换，所以当所有这些位置的输入相同时，它们的输出也是相同的。

```
ffn = PositionWiseFFN(4, 4, 8)
ffn.eval()
```

(continues on next page)

(continued from previous page)

```
ffn(torch.ones((2, 3, 4)))[0]
```

```
tensor([[-0.8290,  1.0067,  0.3619,  0.3594, -0.5328,  0.2712,  0.7394,  0.0747],
       [-0.8290,  1.0067,  0.3619,  0.3594, -0.5328,  0.2712,  0.7394,  0.0747],
       [-0.8290,  1.0067,  0.3619,  0.3594, -0.5328,  0.2712,  0.7394,  0.0747]],  
grad_fn=<SelectBackward0>)
```

10.7.3 残差连接和层规范化

现在让我们关注 图10.7.1 中的加法和规范化（add&norm）组件。正如在本节开头所述，这是由残差连接和紧随其后的层规范化组成的。两者都是构建有效的深度架构的关键。

7.5 节中解释了在一个小批量的样本内基于批量规范化对数据进行重新中心化和重新缩放的调整。层规范化和批量规范化的目标相同，但层规范化是基于特征维度进行规范化。尽管批量规范化在计算机视觉中被广泛应用，但在自然语言处理任务中（输入通常是变长序列）批量规范化通常不如层规范化效果好。

以下代码对比不同维度的层规范化和批量规范化效果。

```
ln = nn.LayerNorm(2)  
bn = nn.BatchNorm1d(2)  
X = torch.tensor([[1, 2], [2, 3]], dtype=torch.float32)  
# 在训练模式下计算x的均值和方差  
print('layer norm:', ln(X), '\nbatch norm:', bn(X))
```

```
layer norm: tensor([-1.0000,  1.0000],
                   [-1.0000,  1.0000]), grad_fn=<NativeLayerNormBackward0>  
batch norm: tensor([-1.0000, -1.0000],
                   [ 1.0000,  1.0000]), grad_fn=<NativeBatchNormBackward0>
```

现在可以使用残差连接和层规范化来实现AddNorm类。暂退法也被作为正则化方法使用。

```
#@save  
class AddNorm(nn.Module):  
    """残差连接后进行层规范化"""  
    def __init__(self, normalized_shape, dropout, **kwargs):  
        super(AddNorm, self).__init__(**kwargs)  
        self.dropout = nn.Dropout(dropout)  
        self.ln = nn.LayerNorm(normalized_shape)  
  
    def forward(self, X, Y):  
        return self.ln(self.dropout(Y) + X)
```

残差连接要求两个输入的形状相同，以便加法操作后输出张量的形状相同。

```
add_norm = AddNorm([3, 4], 0.5)
add_norm.eval()
add_norm(torch.ones((2, 3, 4)), torch.ones((2, 3, 4))).shape
```

```
torch.Size([2, 3, 4])
```

10.7.4 编码器

有了组成Transformer编码器的基础组件，现在可以先实现编码器中的一个层。下面的EncoderBlock类包含两个子层：多头自注意力和基于位置的前馈网络，这两个子层都使用了残差连接和紧随的层规范化。

```
#@save
class EncoderBlock(nn.Module):
    """Transformer编码器块"""
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
                 dropout, use_bias=False, **kwargs):
        super(EncoderBlock, self).__init__(**kwargs)
        self.attention = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout,
            use_bias)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(
            ffn_num_input, ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(norm_shape, dropout)

    def forward(self, X, valid_lens):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
        return self.addnorm2(Y, self.ffn(Y))
```

正如从代码中所看到的，Transformer编码器中的任何层都不会改变其输入的形状。

```
X = torch.ones((2, 100, 24))
valid_lens = torch.tensor([3, 2])
encoder_blk = EncoderBlock(24, 24, 24, 24, [100, 24], 24, 48, 8, 0.5)
encoder_blk.eval()
encoder_blk(X, valid_lens).shape
```

```
torch.Size([2, 100, 24])
```

下面实现的Transformer编码器的代码中，堆叠了num_layers个EncoderBlock类的实例。由于这里使用的是值范围在-1和1之间的固定位置编码，因此通过学习得到的输入的嵌入表示的值需要先乘以嵌入维度的平方根进行重新缩放，然后再与位置编码相加。

```
#@save
class TransformerEncoder(d2l.Encoder):
    """Transformer编码器"""
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
                 num_heads, num_layers, dropout, use_bias=False, **kwargs):
        super(TransformerEncoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
                EncoderBlock(key_size, query_size, value_size, num_hiddens,
                            norm_shape, ffn_num_input, ffn_num_hiddens,
                            num_heads, dropout, use_bias))

    def forward(self, X, valid_lens, *args):
        # 因为位置编码值在-1和1之间,
        # 因此嵌入值乘以嵌入维度的平方根进行缩放,
        # 然后再与位置编码相加。
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
            self.attention_weights[i] = blk.attention.attention.attention_weights
        return X
```

下面我们指定了超参数来创建一个两层的Transformer编码器。Transformer编码器输出的形状是（批量大小，时间步数目， num_hiddens）。

```
encoder = TransformerEncoder(
    200, 24, 24, 24, 24, [100, 24], 24, 48, 8, 2, 0.5)
encoder.eval()
encoder(torch.ones((2, 100), dtype=torch.long), valid_lens).shape
```

```
torch.Size([2, 100, 24])
```

10.7.5 解码器

如图10.7.1所示，Transformer解码器也是由多个相同的层组成。在DecoderBlock类中实现的每个层包含了三个子层：解码器自注意力、“编码器-解码器”注意力和基于位置的前馈网络。这些子层也都被残差连接和紧随的层规范化围绕。

正如在本节前面所述，在掩蔽多头解码器自注意力层（第一个子层）中，查询、键和值都来自上一个解码器层的输出。关于序列到序列模型（sequence-to-sequence model），在训练阶段，其输出序列的所有位置（时间步）的词元都是已知的；然而，在预测阶段，其输出序列的词元是逐个生成的。因此，在任何解码器时间步中，只有生成的词元才能用于解码器的自注意力计算中。为了在解码器中保留自回归的属性，其掩蔽自注意力设定了参数dec_valid_lens，以便任何查询都只会与解码器中所有已经生成词元的位置（即直到该查询位置为止）进行注意力计算。

```
class DecoderBlock(nn.Module):
    """解码器中第i个块"""
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
                 dropout, i, **kwargs):
        super(DecoderBlock, self).__init__(**kwargs)
        self.i = i
        self.attention1 = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.attention2 = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm2 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(ffn_num_input, ffn_num_hiddens,
                                   num_hiddens)
        self.addnorm3 = AddNorm(norm_shape, dropout)

    def forward(self, X, state):
        enc_outputs, enc_valid_lens = state[0], state[1]
        # 训练阶段，输出序列的所有词元都在同一时间处理，
        # 因此state[2][self.i]初始化为None。
        # 预测阶段，输出序列是通过词元一个接着一个解码的，
        # 因此state[2][self.i]包含着直到当前时间步第i个块解码的输出表示
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = torch.cat((state[2][self.i], X), axis=1)
```

(continues on next page)

(continued from previous page)

```
state[2][self.i] = key_values
if self.training:
    batch_size, num_steps, _ = X.shape
    # dec_valid_lens的开头:(batch_size,num_steps),
    # 其中每一行是[1,2,...,num_steps]
    dec_valid_lens = torch.arange(
        1, num_steps + 1, device=X.device).repeat(batch_size, 1)
else:
    dec_valid_lens = None

# 自注意力
X2 = self.attention1(X, key_values, key_values, dec_valid_lens)
Y = self.addnorm1(X, X2)
# 编码器—解码器注意力。
# enc_outputs的开头:(batch_size,num_steps,num_hiddens)
Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_lens)
Z = self.addnorm2(Y, Y2)
return self.addnorm3(Z, self.ffn(Z)), state
```

为了便于在“编码器—解码器”注意力中进行缩放点积计算和残差连接中进行加法计算，编码器和解码器的特征维度都是num_hiddens。

```
decoder_blk = DecoderBlock(24, 24, 24, 24, [100, 24], 24, 48, 8, 0.5, 0)
decoder_blk.eval()
X = torch.ones((2, 100, 24))
state = [encoder_blk(X, valid_lens), valid_lens, [None]]
decoder_blk(X, state)[0].shape
```

```
torch.Size([2, 100, 24])
```

现在我们构建了由num_layers个DecoderBlock实例组成的完整的Transformer解码器。最后，通过一个全连接层计算所有vocab_size个可能的输出词元的预测值。解码器的自注意力权重和编码器解码器注意力权重都被存储下来，方便日后可视化的需要。

```
class TransformerDecoder(d2l.AttentionDecoder):
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
                 num_heads, num_layers, dropout, **kwargs):
        super(TransformerDecoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.num_layers = num_layers
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
```

(continues on next page)

(continued from previous page)

```
self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
self.blks = nn.Sequential()
for i in range(num_layers):
    self.blks.add_module("block"+str(i),
        DecoderBlock(key_size, query_size, value_size, num_hiddens,
                    norm_shape, ffn_num_input, ffn_num_hiddens,
                    num_heads, dropout, i))
self.dense = nn.Linear(num_hiddens, vocab_size)

def init_state(self, enc_outputs, enc_valid_lens, *args):
    return [enc_outputs, enc_valid_lens, [None] * self.num_layers]

def forward(self, X, state):
    X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
    self._attention_weights = [[None] * len(self.blks) for _ in range(2)]
    for i, blk in enumerate(self.blks):
        X, state = blk(X, state)
        # 解码器自注意力权重
        self._attention_weights[0][i] = blk.attention1.attention.attention_weights
        # “编码器—解码器” 自注意力权重
        self._attention_weights[1][i] = blk.attention2.attention.attention_weights
    return self.dense(X), state

@property
def attention_weights(self):
    return self._attention_weights
```

10.7.6 训练

依照Transformer架构来实例化编码器—解码器模型。在这里，指定Transformer的编码器和解码器都是2层，都使用4头注意力。与9.7.4节类似，为了进行序列到序列的学习，下面在“英语—法语”机器翻译数据集上训练Transformer模型。

```
num_hiddens, num_layers, dropout, batch_size, num_steps = 32, 2, 0.1, 64, 10
lr, num_epochs, device = 0.005, 200, d2l.try_gpu()
ffn_num_input, ffn_num_hiddens, num_heads = 32, 64, 4
key_size, query_size, value_size = 32, 32, 32
norm_shape = [32]
```

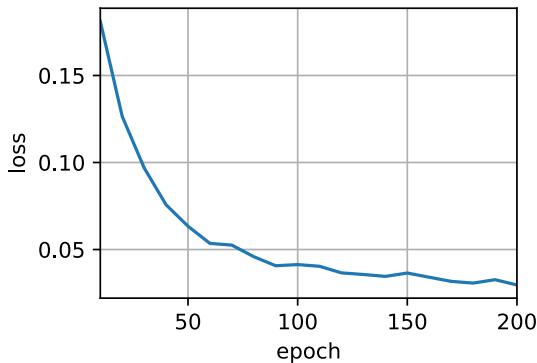
(continues on next page)

(continued from previous page)

```
train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)

encoder = TransformerEncoder(
    len(src_vocab), key_size, query_size, value_size, num_hiddens,
    norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
    num_layers, dropout)
decoder = TransformerDecoder(
    len(tgt_vocab), key_size, query_size, value_size, num_hiddens,
    norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
    num_layers, dropout)
net = d2l.EncoderDecoder(encoder, decoder)
d2l.train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.030, 5202.9 tokens/sec on cuda:0
```



训练结束后，使用Transformer模型将一些英语句子翻译成法语，并且计算它们的BLEU分数。

```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, dec_attention_weight_seq = d2l.predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device, True)
    print(f'{eng} => {translation}, ',
          f'bleu {d2l.bleu(translation, fra, k=2):.3f}')
```

```
go . => va !, bleu 1.000
i lost . => j'ai perdu ., bleu 1.000
he's calm . => il est calme ., bleu 1.000
i'm home . => je suis chez moi ., bleu 1.000
```

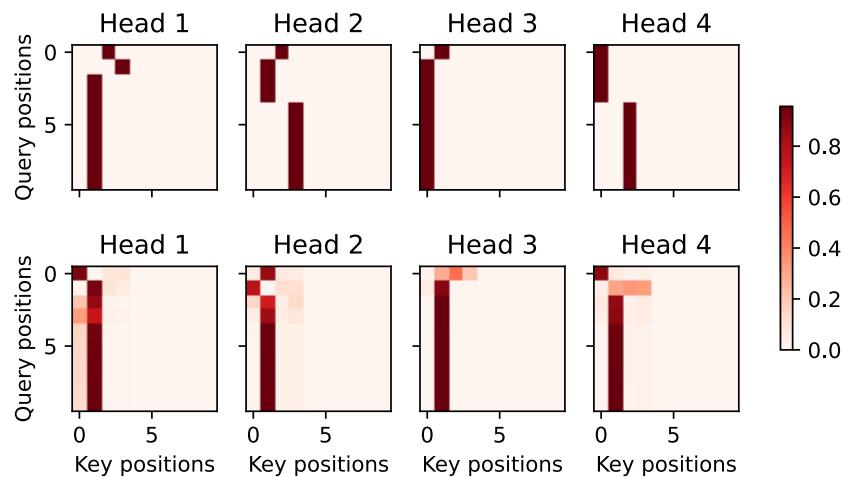
当进行最后一个英语到法语的句子翻译工作时，让我们可视化Transformer的注意力权重。编码器自注意力权重的形状为（编码器层数， 注意力头数， num_steps或查询的数目， num_steps或“键一值”对的数目）。

```
enc_attention_weights = torch.cat(net.encoder.attention_weights, 0).reshape((num_layers, num_heads, -1, num_steps))
enc_attention_weights.shape
```

```
torch.Size([2, 4, 10, 10])
```

在编码器的自注意力中，查询和键都来自相同的输入序列。因为填充词元是不携带信息的，因此通过指定输入序列的有效长度可以避免查询与使用填充词元的位置计算注意力。接下来，将逐行呈现两层多头注意力的权重。每个注意力头都根据查询、键和值的不同的表示子空间来表示不同的注意力。

```
d2l.show_heatmaps(
    enc_attention_weights.cpu(), xlabel='Key positions',
    ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
    figsize=(7, 3.5))
```



为了可视化解码器的自注意力权重和“编码器一解码器”的注意力权重，我们需要完成更多的数据操作工作。例如用零填充被掩蔽住的注意力权重。值得注意的是，解码器的自注意力权重和“编码器一解码器”的注意力权重都有相同的查询：即以序列开始词元（beginning-of-sequence, BOS）打头，再与后续输出的词元共同组成序列。

```
dec_attention_weights_2d = [head[0].tolist()
                             for step in dec_attention_weight_seq
                             for attn in step for blk in attn for head in blk]
dec_attention_weights_filled = torch.tensor([
    pd.DataFrame(dec_attention_weights_2d).fillna(0.0).values
])
```

(continues on next page)

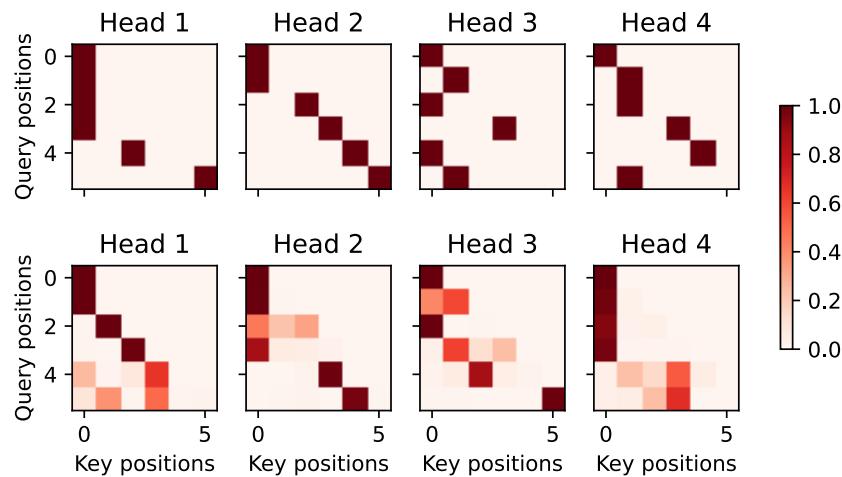
(continued from previous page)

```
dec_attention_weights = dec_attention_weights_filled.reshape((-1, 2, num_layers, num_heads, num_steps))
dec_self_attention_weights, dec_inter_attention_weights = \
    dec_attention_weights.permute(1, 2, 3, 0, 4)
dec_self_attention_weights.shape, dec_inter_attention_weights.shape
```

```
(torch.Size([2, 4, 6, 10]), torch.Size([2, 4, 6, 10]))
```

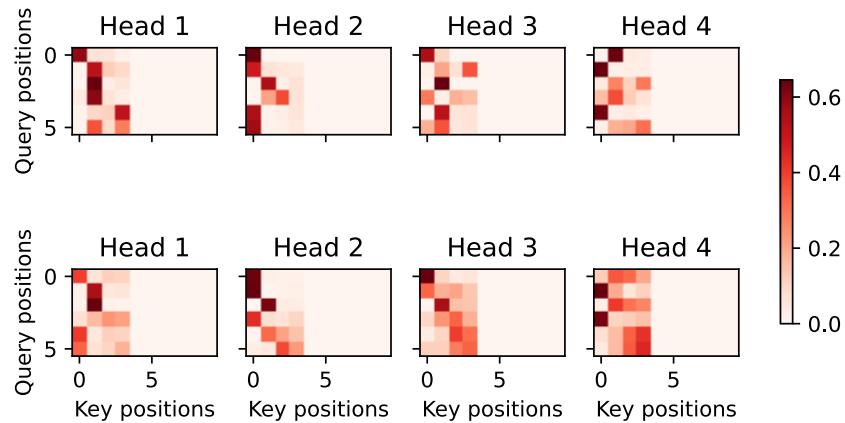
由于解码器自注意力的自回归属性，查询不会对当前位置之后的“键一值”对进行注意力计算。

```
# Plus one to include the beginning-of-sequence token
d2l.show_heatmaps(
    dec_self_attention_weights[:, :, :, :len(translation.split()) + 1],
    xlabel='Key positions', ylabel='Query positions',
    titles=['Head %d' % i for i in range(1, 5)], figsize=(7, 3.5))
```



与编码器的自注意力的情况类似，通过指定输入序列的有效长度，输出序列的查询不会与输入序列中填充位置的词元进行注意力计算。

```
d2l.show_heatmaps(
    dec_inter_attention_weights, xlabel='Key positions',
    ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
    figsize=(7, 3.5))
```



尽管Transformer架构是为了序列到序列的学习而提出的，但正如本书后面将提及的那样，Transformer编码器或Transformer解码器通常被单独用于不同的深度学习任务中。

小结

- Transformer是编码器—解码器架构的一个实践，尽管在实际情况中编码器或解码器可以单独使用。
- 在Transformer中，多头自注意力用于表示输入序列和输出序列，不过解码器必须通过掩蔽机制来保留自回归属性。
- Transformer中的残差连接和层规范化是训练非常深度模型的重要工具。
- Transformer模型中基于位置的前馈网络使用同一个多层次感知机，作用是对所有序列位置的表示进行转换。

练习

1. 在实验中训练更深的Transformer将如何影响训练速度和翻译效果？
2. 在Transformer中使用加性注意力取代缩放点积注意力是不是个好办法？为什么？
3. 对于语言模型，应该使用Transformer的编码器还是解码器，或者两者都用？如何设计？
4. 如果输入序列很长，Transformer会面临什么挑战？为什么？
5. 如何提高Transformer的计算速度和内存使用效率？提示：可以参考论文 (Tay *et al.*, 2020)。
6. 如果不使用卷积神经网络，如何设计基于Transformer模型的图像分类任务？提示：可以参考Vision Transformer (Dosovitskiy *et al.*, 2021)。

Discussions¹²⁴

¹²⁴ <https://discuss.d2l.ai/t/5756>

11

优化算法

截止到目前，本书已经使用了许多优化算法来训练深度学习模型。优化算法使我们能够继续更新模型参数，并使损失函数的值最小化。这就像在训练集上评估一样。事实上，任何满足于将优化视为黑盒装置，以在简单的设置中最小化目标函数的人，都可能会知道存在着一系列此类“咒语”（名称如“SGD”和“Adam”）。

但是，为了做得更好，还需要更深入的知识。优化算法对于深度学习非常重要。一方面，训练复杂的深度学习模型可能需要数小时、几天甚至数周。优化算法的性能直接影响模型的训练效率。另一方面，了解不同优化算法的原则及其超参数的作用将使我们能够以有针对性的方式调整超参数，以提高深度学习模型的性能。

在本章中，我们深入探讨常见的深度学习优化算法。深度学习中出现的几乎所有优化问题都是非凸的。尽管如此，在凸问题背景下设计和分析算法是非常有启发性的。正是出于这个原因，本章包括了凸优化的入门，以及凸目标函数上非常简单的随机梯度下降算法的证明。

11.1 优化和深度学习

本节将讨论优化与深度学习之间的关系以及在深度学习中使用优化的挑战。对于深度学习问题，我们通常会先定义损失函数。一旦我们有了损失函数，我们就可以使用优化算法来尝试最小化损失。在优化中，损失函数通常被称为优化问题的目标函数。按照传统惯例，大多数优化算法都关注的是最小化。如果我们需要最大化目标，那么有一个简单的解决方案：在目标函数前加负号即可。

11.1.1 优化的目标

尽管优化提供了一种最大限度地减少深度学习损失函数的方法，但本质上，优化和深度学习的目标是根本不同的。前者主要关注的是最小化目标，后者则关注在给定有限数据量的情况下寻找合适的模型。在 4.4 节中，我们详细讨论了这两个目标之间的区别。例如，训练误差和泛化误差通常不同：由于优化算法的目标函数通常是基于训练数据集的损失函数，因此优化的目标是减少训练误差。但是，深度学习（或更广义地说，统计推断）的目标是减少泛化误差。为了实现后者，除了使用优化算法来减少训练误差之外，我们还需要注意过拟合。

```
%matplotlib inline
import numpy as np
import torch
from mpl_toolkits import mplot3d
from d2l import torch as d2l
```

为了说明上述不同的目标，引入两个概念风险和经验风险。如 4.9.3 节所述，经验风险是训练数据集的平均损失，而风险则是整个数据群的预期损失。下面我们定义了两个函数：风险函数 f 和经验风险函数 g 。假设我们只有有限的训练数据。因此，这里的 g 不如 f 平滑。

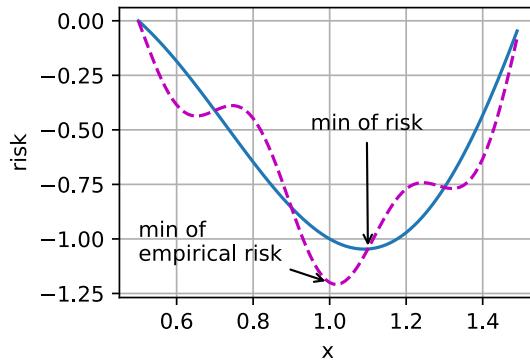
```
def f(x):
    return x * torch.cos(np.pi * x)

def g(x):
    return f(x) + 0.2 * torch.cos(5 * np.pi * x)
```

下图说明，训练数据集的最低经验风险可能与最低风险（泛化误差）不同。

```
def annotate(text, xy, xytext):  # @save
    d2l.plt.gca().annotate(text, xy=xy, xytext=xytext,
                           arrowprops=dict(arrowstyle='->'))

x = torch.arange(0.5, 1.5, 0.01)
d2l.set_figsize((4.5, 2.5))
d2l.plot(x, [f(x), g(x)], 'x', 'risk')
annotate('min of\nempirical risk', (1.0, -1.2), (0.5, -1.1))
annotate('min of risk', (1.1, -1.05), (0.95, -0.5))
```



11.1.2 深度学习中的优化挑战

本章将关注优化算法在最小化目标函数方面的性能，而不是模型的泛化误差。在 3.1 节中，我们区分了优化问题中的解析解和数值解。在深度学习中，大多数目标函数都很复杂，没有解析解。相反，我们必须使用数值优化算法。本章中的优化算法都属于此类别。

深度学习优化存在许多挑战。其中最令人烦恼的是局部最小值、鞍点和梯度消失。

局部最小值

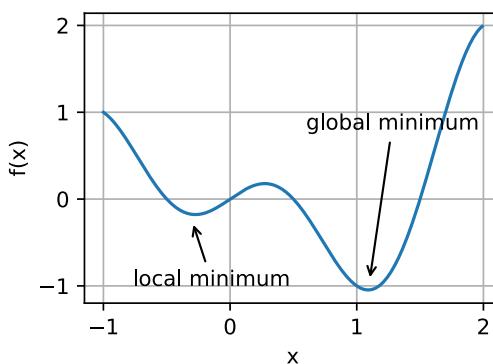
对于任何目标函数 $f(x)$ ，如果在 x 处对应的 $f(x)$ 值小于在 x 附近任意其他点的 $f(x)$ 值，那么 $f(x)$ 可能是局部最小值。如果 $f(x)$ 在 x 处的值是整个域中目标函数的最小值，那么 $f(x)$ 是全局最小值。

例如，给定函数

$$f(x) = x \cdot \cos(\pi x) \text{ for } -1.0 \leq x \leq 2.0, \quad (11.1.1)$$

我们可以近似该函数的局部最小值和全局最小值。

```
x = torch.arange(-1.0, 2.0, 0.01)
d2l.plot(x, [f(x)], 'x', 'f(x)')
annotate('local minimum', (-0.3, -0.25), (-0.77, -1.0))
annotate('global minimum', (1.1, -0.95), (0.6, 0.8))
```

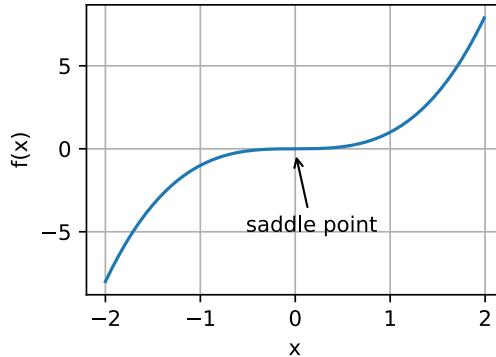


深度学习模型的目标函数通常有许多局部最优解。当优化问题的数值解接近局部最优值时，随着目标函数解的梯度接近或变为零，通过最终迭代获得的数值解可能仅使目标函数局部最优，而不是全局最优。只有一定程度的噪声可能会使参数跳出局部最小值。事实上，这是小批量随机梯度下降的有利特性之一。在这种情况下，小批量上梯度的自然变化能够将参数从局部极小值中跳出。

鞍点

除了局部最小值之外，鞍点是梯度消失的另一个原因。鞍点（saddle point）是指函数的所有梯度都消失但既不是全局最小值也不是局部最小值的任何位置。考虑这个函数 $f(x) = x^3$ 。它的一阶和二阶导数在 $x = 0$ 时消失。这时优化可能会停止，尽管它不是最小值。

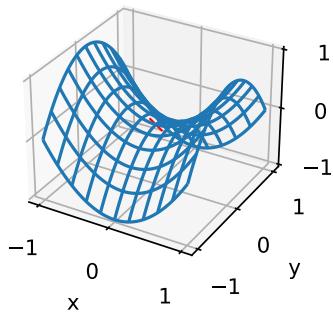
```
x = torch.arange(-2.0, 2.0, 0.01)
d2l.plot(x, [x**3], 'x', 'f(x)')
annotate('saddle point', (0, -0.2), (-0.52, -5.0))
```



如下例所示，较高维度的鞍点甚至更加隐蔽。考虑这个函数 $f(x, y) = x^2 - y^2$ 。它的鞍点为 $(0, 0)$ 。这是关于 y 的最大值，也是关于 x 的最小值。此外，它看起来像个马鞍，这就是鞍点的名字由来。

```
x, y = torch.meshgrid(
    torch.linspace(-1.0, 1.0, 101), torch.linspace(-1.0, 1.0, 101))
z = x**2 - y**2

ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.plot([0], [0], [0], 'rx')
ticks = [-1, 0, 1]
d2l.plt.xticks(ticks)
d2l.plt.yticks(ticks)
ax.set_zticks(ticks)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y');
```



我们假设函数的输入是 k 维向量，其输出是标量，因此其Hessian矩阵（也称黑塞矩阵）将有 k 个特征值（参考特征分解的在线附录¹²⁵）。函数的解可能是局部最小值、局部最大值或函数梯度为零位置处的鞍点：

- 当函数在零梯度位置处的Hessian矩阵的特征值全部为正值时，我们有该函数的局部最小值；
- 当函数在零梯度位置处的Hessian矩阵的特征值全部为负值时，我们有该函数的局部最大值；
- 当函数在零梯度位置处的Hessian矩阵的特征值为负值和正值时，我们有该函数的一个鞍点。

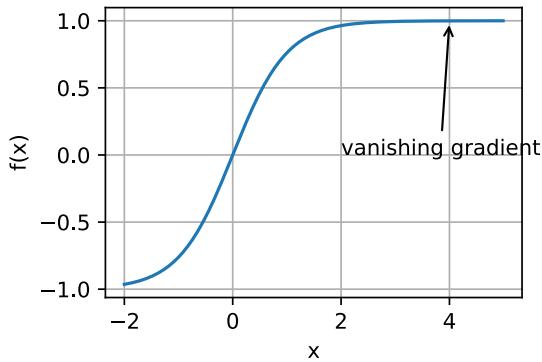
对于高维度问题，至少部分特征值为负的可能性相当高。这使得鞍点比局部最小值更有可能出现。我们将在下一节介绍凸性时讨论这种情况的一些例外。简而言之，凸函数是Hessian函数的特征值永远不为负值的函数。不幸的是，大多数深度学习问题并不属于这一类。尽管如此，它还是研究优化算法的一个很好的工具。

梯度消失

可能遇到的最隐蔽问题是梯度消失。回想一下我们在 4.1.2 节中常用的激活函数及其衍生函数。例如，假设我们想最小化函数 $f(x) = \tanh(x)$ ，然后我们恰好从 $x = 4$ 开始。正如我们所看到的那样， f 的梯度接近零。更具体地说， $f'(x) = 1 - \tanh^2(x)$ ，因此是 $f'(4) = 0.0013$ 。因此，在我们取得进展之前，优化将会停滞很长一段时间。事实证明，这是在引入ReLU激活函数之前训练深度学习模型相当棘手的原因之一。

```
x = torch.arange(-2.0, 5.0, 0.01)
d2l.plot(x, [torch.tanh(x)], 'x', 'f(x)')
annotate('vanishing gradient', (4, 1), (2, 0.0))
```

¹²⁵ https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/eigendecomposition.html



正如我们所看到的那样，深度学习的优化充满挑战。幸运的是，有一系列强大的算法表现良好，即使对于初学者也很容易使用。此外，没有必要找到最优解。局部最优解或其近似解仍然非常有用。

小结

- 最小化训练误差并不能保证我们找到最佳的参数集来最小化泛化误差。
- 优化问题可能有许多局部最小值。
- 一个问题可能有很多的鞍点，因为问题通常不是凸的。
- 梯度消失可能会导致优化停滞，重参数化通常会有所帮助。对参数进行良好的初始化也可能是有益的。

练习

1. 考虑一个简单的MLP，它有一个隐藏层，比如，隐藏层中维度为 d 和一个输出。证明对于任何局部最小值，至少有 d 个等效方案。
2. 假设我们有一个对称随机矩阵 \mathbf{M} ，其中条目 $M_{ij} = M_{ji}$ 各自从某种概率分布 p_{ij} 中抽取。此外，假设 $p_{ij}(x) = p_{ij}(-x)$ ，即分布是对称的（详情请参见 (Wigner, 1958)）。
 1. 证明特征值的分布也是对称的。也就是说，对于任何特征向量 \mathbf{v} ，关联的特征值 λ 满足 $P(\lambda > 0) = P(\lambda < 0)$ 的概率为 $P(\lambda > 0) = P(\lambda < 0)$ 。
 2. 为什么以上没有暗示 $P(\lambda > 0) = 0.5$ ？
3. 你能想到深度学习优化还涉及哪些其他挑战？
4. 假设你想在（真实的）鞍上平衡一个（真实的）球。
 1. 为什么这很难？
 2. 能利用这种效应来优化算法吗？

Discussions¹²⁶

¹²⁶ <https://discuss.d2l.ai/t/3841>

11.2 凸性

凸性 (convexity) 在优化算法的设计中起到至关重要的作用，这主要是由于在这种情况下对算法进行分析和测试要容易。换言之，如果算法在凸性条件设定下的效果很差，那通常我们很难在其他条件下看到好的结果。此外，即使深度学习中的优化问题通常是非凸的，它们也经常在局部极小值附近表现出一些凸性。这可能会产生一些像 (Izmailov et al., 2018) 这样比较有意思的新优化变体。

```
%matplotlib inline
import numpy as np
import torch
from mpl_toolkits import mplot3d
from d2l import torch as d2l
```

11.2.1 定义

在进行凸分析之前，我们需要定义凸集（convex sets）和凸函数（convex functions）。

凸集

凸集（convex set）是凸性的基础。简单地说，如果对于任何 $a, b \in \mathcal{X}$ ，连接 a 和 b 的线段也位于 \mathcal{X} 中，则向量空间中的一个集合 \mathcal{X} 是凸（convex）的。在数学术语上，这意味着对于所有 $\lambda \in [0, 1]$ ，我们得到

$$\lambda a + (1 - \lambda)b \in \mathcal{X} \text{ 当 } a, b \in \mathcal{X}. \quad (11.2.1)$$

这听起来有点抽象，那我们来看一下 图11.2.1 里的例子。第一组存在不包含在集合内部的线段，所以该集合是非凸的，而另外两组则没有这样的问题。

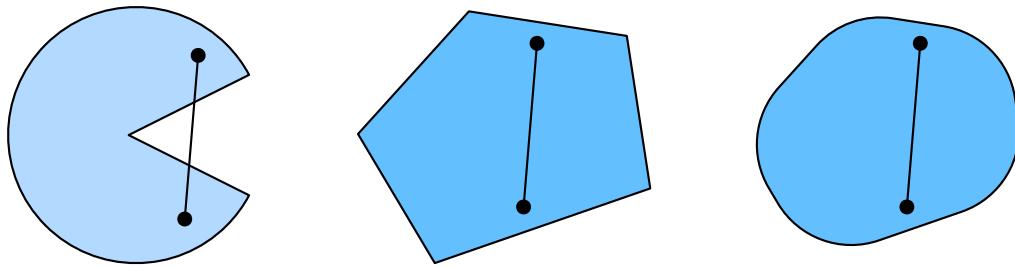


图11.2.1：第一组是非凸的，另外两组是凸的。

接下来来看一下交集 图11.2.2。假设 \mathcal{X} 和 \mathcal{Y} 是凸集，那么 $\mathcal{X} \cap \mathcal{Y}$ 也是凸集的。现在考虑任意 $a, b \in \mathcal{X} \cap \mathcal{Y}$ ，因为 \mathcal{X} 和 \mathcal{Y} 是凸集，所以连接 a 和 b 的线段包含在 \mathcal{X} 和 \mathcal{Y} 中。鉴于此，它们也需要包含在 $\mathcal{X} \cap \mathcal{Y}$ 中，从而证明我们的定理。

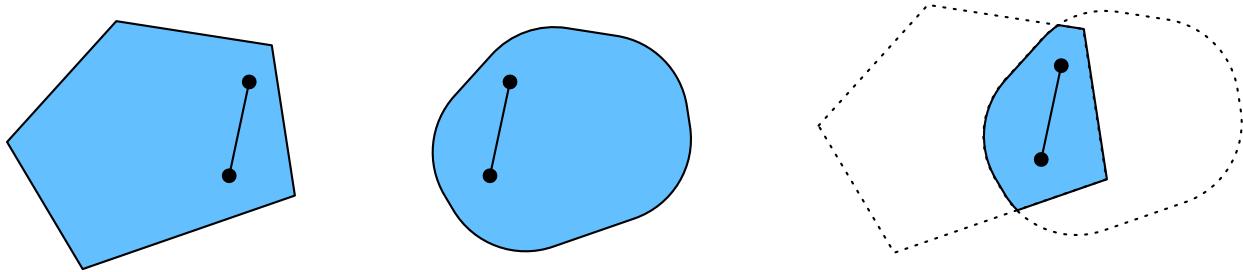


图11.2.2: 两个凸集的交集是凸的。

我们可以毫不费力地进一步得到这样的结果: 给定凸集 \mathcal{X}_i , 它们的交集 $\cap_i \mathcal{X}_i$ 是凸的。但是反向是不正确的, 考虑两个不相交的集合 $\mathcal{X} \cap \mathcal{Y} = \emptyset$, 取 $a \in \mathcal{X}$ 和 $b \in \mathcal{Y}$ 。因为我们假设 $\mathcal{X} \cap \mathcal{Y} = \emptyset$, 在图11.2.3中连接 a 和 b 的线段需要包含一部分既不在 \mathcal{X} 也不在 \mathcal{Y} 中。因此线段也不在 $\mathcal{X} \cup \mathcal{Y}$ 中, 因此证明了凸集的并集不一定是凸的, 即非凸 (nonconvex) 的。

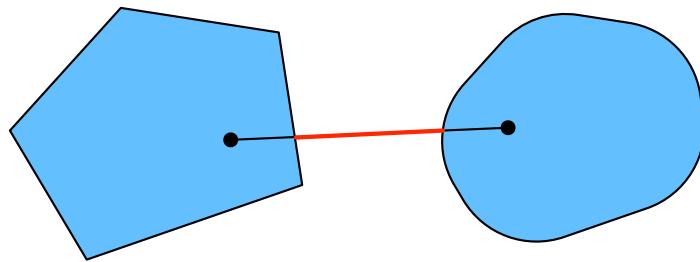


图11.2.3: 两个凸集的并集不一定是凸的。

通常, 深度学习中的问题是在凸集上定义的。例如, \mathbb{R}^d , 即实数的 d -维向量的集合是凸集 (毕竟 \mathbb{R}^d 中任意两点之间的线存在 \mathbb{R}^d 中)。在某些情况下, 我们使用有界长度的变量, 例如球的半径定义为 $\{\mathbf{x} | \mathbf{x} \in \mathbb{R}^d \text{ 且 } \|\mathbf{x}\| \leq r\}$ 。

凸函数

现在我们有了凸集, 我们可以引入凸函数 (convex function) f 。给定一个凸集 \mathcal{X} , 如果对于所有 $x, x' \in \mathcal{X}$ 和所有 $\lambda \in [0, 1]$, 函数 $f : \mathcal{X} \rightarrow \mathbb{R}$ 是凸的, 我们可以得到

$$\lambda f(x) + (1 - \lambda)f(x') \geq f(\lambda x + (1 - \lambda)x'). \quad (11.2.2)$$

为了说明这一点, 让我们绘制一些函数并检查哪些函数满足要求。下面我们定义一些函数, 包括凸函数和非凸函数。

```

f = lambda x: 0.5 * x**2 # 凸函数
g = lambda x: torch.cos(np.pi * x) # 非凸函数
h = lambda x: torch.exp(0.5 * x) # 凸函数

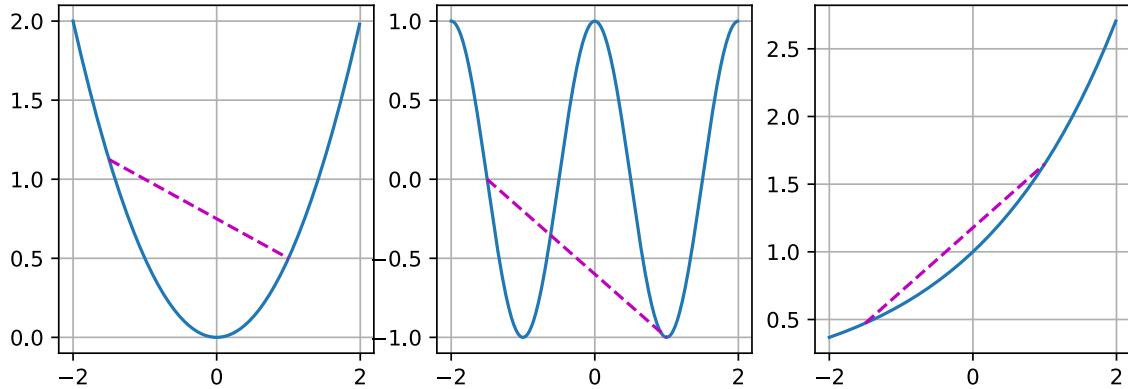
x, segment = torch.arange(-2, 2, 0.01), torch.tensor([-1.5, 1])

```

(continues on next page)

(continued from previous page)

```
d2l.use_svg_display()
_, axes = d2l.plt.subplots(1, 3, figsize=(9, 3))
for ax, func in zip(axes, [f, g, h]):
    d2l.plot([x, segment], [func(x), func(segment)], axes=ax)
```



不出所料，余弦函数为非凸的，而抛物线函数和指数函数为凸的。请注意，为使该条件有意义， \mathcal{X} 是凸集的要求是必要的。否则可能无法很好地界定 $f(\lambda x + (1 - \lambda)x')$ 的结果。

詹森不等式

给定一个凸函数 f ，最有用的数学工具之一就是詹森不等式 (Jensen's inequality)。它是凸性定义的一种推广：

$$\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right) \text{ and } E_X[f(X)] \geq f(E_X[X]), \quad (11.2.3)$$

其中 α_i 是满足 $\sum_i \alpha_i = 1$ 的非负实数， X 是随机变量。换句话说，凸函数的期望不小于期望的凸函数，其中后者通常是一个更简单的表达式。为了证明第一个不等式，我们多次将凸性的定义应用于一次求和中的一项。

詹森不等式的一个常见应用：用一个较简单的表达式约束一个较复杂的表达式。例如，它可以应用于部分观察到的随机变量的对数似然。具体地说，由于 $\int P(Y)P(X | Y)dY = P(X)$ ，所以

$$E_{Y \sim P(Y)}[-\log P(X | Y)] \geq -\log P(X), \quad (11.2.4)$$

这里， Y 是典型的未观察到的随机变量， $P(Y)$ 是它可能如何分布的最佳猜测， $P(X)$ 是将 Y 积分后的分布。例如，在聚类中 Y 可能是簇标签，而在应用簇标签时， $P(X | Y)$ 是生成模型。

11.2.2 性质

下面我们来看一下凸函数一些有趣的性质。

局部极小值是全局极小值

首先凸函数的局部极小值也是全局极小值。下面我们用反证法给出证明。

假设 $x^* \in \mathcal{X}$ 是一个局部最小值，则存在一个很小的正值 p ，使得当 $x \in \mathcal{X}$ 满足 $0 < |x - x^*| \leq p$ 时，有 $f(x^*) < f(x)$ 。

现在假设局部极小值 x^* 不是 f 的全局极小值：存在 $x' \in \mathcal{X}$ 使得 $f(x') < f(x^*)$ 。则存在 $\lambda \in [0, 1]$ ，比如 $\lambda = 1 - \frac{p}{|x^* - x'|}$ ，使得 $0 < |\lambda x^* + (1 - \lambda)x' - x^*| \leq p$ 。

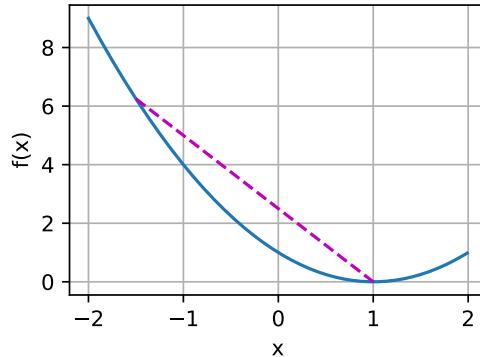
然而，根据凸性的性质，有

$$\begin{aligned} f(\lambda x^* + (1 - \lambda)x') &\leq \lambda f(x^*) + (1 - \lambda)f(x') \\ &< \lambda f(x^*) + (1 - \lambda)f(x^*) \\ &= f(x^*), \end{aligned} \tag{11.2.5}$$

这与 x^* 是局部最小值相矛盾。因此，不存在 $x' \in \mathcal{X}$ 满足 $f(x') < f(x^*)$ 。综上所述，局部最小值 x^* 也是全局最小值。

例如，对于凸函数 $f(x) = (x - 1)^2$ ，有一个局部最小值 $x = 1$ ，它也是全局最小值。

```
f = lambda x: (x - 1) ** 2
d2l.set_figsize()
d2l.plot([x, segment], [f(x), f(segment)], 'x', 'f(x)')
```



凸函数的局部极小值同时也是全局极小值这一性质是很方便的。这意味着如果我们最小化函数，我们就不会“卡住”。但是请注意，这并不意味着不能有多个全局最小值，或者可能不存在一个全局最小值。例如，函数 $f(x) = \max(|x| - 1, 0)$ 在 $[-1, 1]$ 区间上都是最小值。相反，函数 $f(x) = \exp(x)$ 在 \mathbb{R} 上没有取得最小值。对于 $x \rightarrow -\infty$ ，它趋近于 0，但是没有 $f(x) = 0$ 的 x 。

凸函数的下水平集是凸的

我们可以方便地通过凸函数的下水平集 (below sets) 定义凸集。具体来说，给定一个定义在凸集 \mathcal{X} 上的凸函数 f ，其任意一个下水平集

$$\mathcal{S}_b := \{x | x \in \mathcal{X} \text{ and } f(x) \leq b\} \quad (11.2.6)$$

是凸的。

让我们快速证明一下。对于任何 $x, x' \in \mathcal{S}_b$ ，我们需要证明：当 $\lambda \in [0, 1]$ 时， $\lambda x + (1 - \lambda)x' \in \mathcal{S}_b$ 。因为 $f(x) \leq b$ 且 $f(x') \leq b$ ，所以

$$f(\lambda x + (1 - \lambda)x') \leq \lambda f(x) + (1 - \lambda)f(x') \leq b. \quad (11.2.7)$$

凸性和二阶导数

当一个函数的二阶导数 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 存在时，我们很容易检查这个函数的凸性。我们需要做的是检查 $\nabla^2 f \succeq 0$ ，即对于所有 $\mathbf{x} \in \mathbb{R}^n$ ， $\mathbf{x}^\top \mathbf{H}\mathbf{x} \geq 0$ 。例如，函数 $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}\|^2$ 是凸的，因为 $\nabla^2 f = \mathbf{1}$ ，即其导数是单位矩阵。

更正式地讲， f 为凸函数，当且仅当任意二次可微一维函数 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 是凸的。对于任意二次可微多维函数 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ ，它是凸的当且仅当它的Hessian $\nabla^2 f \succeq 0$ 。

首先，我们来证明一下一维情况。为了证明凸函数的 $f''(x) \geq 0$ ，我们使用：

$$\frac{1}{2}f(x + \epsilon) + \frac{1}{2}f(x - \epsilon) \geq f\left(\frac{x + \epsilon}{2} + \frac{x - \epsilon}{2}\right) = f(x). \quad (11.2.8)$$

因为二阶导数是由有限差分的极限给出的，所以遵循

$$f''(x) = \lim_{\epsilon \rightarrow 0} \frac{f(x + \epsilon) + f(x - \epsilon) - 2f(x)}{\epsilon^2} \geq 0. \quad (11.2.9)$$

为了证明 $f'' \geq 0$ 可以推导 f 是凸的，我们使用这样一个事实： $f'' \geq 0$ 意味着 f' 是一个单调的非递减函数。假设 $a < x < b$ 是 \mathbb{R} 中的三个点，其中， $x = (1 - \lambda)a + \lambda b$ 且 $\lambda \in (0, 1)$ 。根据中值定理，存在 $\alpha \in [a, x]$ ， $\beta \in [x, b]$ ，使得

$$f'(\alpha) = \frac{f(x) - f(a)}{x - a} \text{ 且 } f'(\beta) = \frac{f(b) - f(x)}{b - x}. \quad (11.2.10)$$

通过单调性 $f'(\beta) \geq f'(\alpha)$ ，因此

$$\frac{x - a}{b - a}f(b) + \frac{b - x}{b - a}f(a) \geq f(x). \quad (11.2.11)$$

由于 $x = (1 - \lambda)a + \lambda b$ ，所以

$$\lambda f(b) + (1 - \lambda)f(a) \geq f((1 - \lambda)a + \lambda b), \quad (11.2.12)$$

从而证明了凸性。

第二，我们需要一个引理证明多维情况： $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 是凸的当且仅当对于所有 $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

$$g(z) \stackrel{\text{def}}{=} f(z\mathbf{x} + (1 - z)\mathbf{y}) \text{ where } z \in [0, 1] \quad (11.2.13)$$

是凸的。

为了证明 f 的凸性意味着 g 是凸的，我们可以证明，对于所有的 $a \otimes b \otimes \lambda \in [0 \otimes 1]$ （这样有 $0 \leq \lambda a + (1 - \lambda)b \leq 1$ ），

$$\begin{aligned} & g(\lambda a + (1 - \lambda)b) \\ &= f((\lambda a + (1 - \lambda)b)\mathbf{x} + (1 - \lambda a - (1 - \lambda)b)\mathbf{y}) \\ &= f(\lambda(a\mathbf{x} + (1 - a)\mathbf{y}) + (1 - \lambda)(b\mathbf{x} + (1 - b)\mathbf{y})) \\ &\leq \lambda f(a\mathbf{x} + (1 - a)\mathbf{y}) + (1 - \lambda)f(b\mathbf{x} + (1 - b)\mathbf{y}) \\ &= \lambda g(a) + (1 - \lambda)g(b). \end{aligned} \tag{11.2.14}$$

为了证明这一点，我们可以证明对 $[0 \otimes 1]$ 中所有的 λ :

$$\begin{aligned} & f(\lambda\mathbf{x} + (1 - \lambda)\mathbf{y}) \\ &= g(\lambda \cdot 1 + (1 - \lambda) \cdot 0) \\ &\leq \lambda g(1) + (1 - \lambda)g(0) \\ &= \lambda f(\mathbf{x}) + (1 - \lambda)f(\mathbf{y}). \end{aligned} \tag{11.2.15}$$

最后，利用上面的引理和一维情况的结果，我们可以证明多维情况：多维函数 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 是凸函数，当且仅当 $g(z) \stackrel{\text{def}}{=} f(z\mathbf{x} + (1 - z)\mathbf{y})$ 是凸的，这里 $z \in [0, 1]$, $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ 。根据一维情况，此条成立的条件为，当且仅当对于所有 $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, $g'' = (\mathbf{x} - \mathbf{y})^\top \mathbf{H}(\mathbf{x} - \mathbf{y}) \geq 0$ ($\mathbf{H} \stackrel{\text{def}}{=} \nabla^2 f$)。这相当于根据半正定矩阵的定义， $\mathbf{H} \succeq 0$ 。

11.2.3 约束

凸优化的一个很好的特性是能够让我们有效地处理约束（constraints）。即它使我们能够解决以下形式的约束优化（constrained optimization）问题：

$$\begin{aligned} & \underset{\mathbf{x}}{\text{minimize}} \quad f(\mathbf{x}) \\ & \text{subject to } c_i(\mathbf{x}) \leq 0 \text{ for all } i \in \{1, \dots, N\}. \end{aligned} \tag{11.2.16}$$

这里 f 是目标函数， c_i 是约束函数。例如第一个约束 $c_1(\mathbf{x}) = \|\mathbf{x}\|_2 - 1$ ，则参数 \mathbf{x} 被限制为单位球。如果第二个约束 $c_2(\mathbf{x}) = \mathbf{v}^\top \mathbf{x} + b$ ，那么这对应于半空间上所有的 \mathbf{x} 。同时满足这两个约束等于选择一个球的切片作为约束集。

拉格朗日函数

通常，求解一个有约束的优化问题是困难的，解决这个问题的一种方法来自物理中相当简单的直觉。想象一个球在一个盒子里，球会滚到最低的地方，重力将与盒子两侧对球施加的力平衡。简而言之，目标函数（即重力）的梯度将被约束函数的梯度所抵消（由于墙壁的“推回”作用，需要保持在盒子内）。请注意，任何不起作用的约束（即球不接触壁）都将无法对球施加任何力。

这里我们简略拉格朗日函数 L 的推导，上述推理可以通过以下鞍点优化问题来表示：

$$L(\mathbf{x}, \alpha_1, \dots, \alpha_n) = f(\mathbf{x}) + \sum_{i=1}^n \alpha_i c_i(\mathbf{x}) \text{ where } \alpha_i \geq 0. \tag{11.2.17}$$

这里的变量 α_i ($i = 1, \dots, n$) 是所谓的拉格朗日乘数 (Lagrange multipliers)，它确保约束被正确地执行。选择它们的大小足以确保所有 i 的 $c_i(\mathbf{x}) \leq 0$ 。例如，对于 $c_i(\mathbf{x}) < 0$ 中任意 \mathbf{x} ，我们最终会选择 $\alpha_i = 0$ 。此外，这是一个鞍点 (saddlepoint) 优化问题。在这个问题中，我们想要使 L 相对于 α_i 最大化 (maximize)，同时使它相对于 \mathbf{x} 最小化 (minimize)。有大量的文献解释如何得出函数 $L(\mathbf{x}, \alpha_1, \dots, \alpha_n)$ 。我们这里只需要知道 L 的鞍点是原始约束优化问题的最优解就足够了。

惩罚

一种至少近似地满足约束优化问题的方法是采用拉格朗日函数 L 。除了满足 $c_i(\mathbf{x}) \leq 0$ 之外，我们只需将 $\alpha_i c_i(\mathbf{x})$ 添加到目标函数 $f(x)$ 。这样可以确保不会严重违反约束。

事实上，我们一直在使用这个技巧。比如权重衰减 4.5 节，在目标函数中加入 $\frac{\lambda}{2} \|\mathbf{w}\|^2$ ，以确保 \mathbf{w} 不会增长太大。使用约束优化的观点，我们可以看到，对于若干半径 r ，这将确保 $\|\mathbf{w}\|^2 - r^2 \leq 0$ 。通过调整 λ 的值，我们可以改变 \mathbf{w} 的大小。

通常，添加惩罚是确保近似满足约束的一种好方法。在实践中，这被证明比精确的满意度更可靠。此外，对于非凸问题，许多使精确方法在凸情况下的性质（例如，可求最优解）不再成立。

投影

满足约束条件的另一种策略是投影 (projections)。同样，我们之前也遇到过，例如在 8.5 节中处理梯度截断时，我们通过

$$\mathbf{g} \leftarrow \mathbf{g} \cdot \min(1, \theta / \|\mathbf{g}\|), \quad (11.2.18)$$

确保梯度的长度以 θ 为界限。

这就是 \mathbf{g} 在半径为 θ 的球上的投影 (projection)。更泛化地说，在凸集 \mathcal{X} 上的投影被定义为

$$\text{Proj}_{\mathcal{X}}(\mathbf{x}) = \underset{\mathbf{x}' \in \mathcal{X}}{\operatorname{argmin}} \|\mathbf{x} - \mathbf{x}'\|. \quad (11.2.19)$$

它是 \mathcal{X} 中离 \mathbf{x} 最近的点。

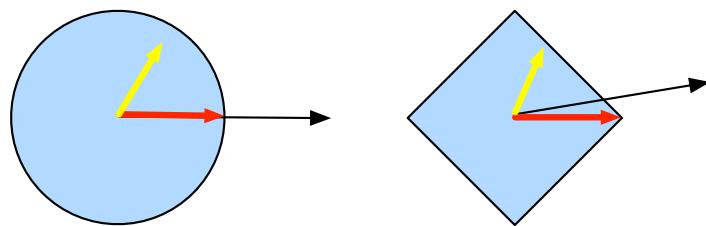


图11.2.4: Convex Projections.

投影的数学定义听起来可能有点抽象，为了解释得更清楚一些，请看 图11.2.4。图中有两个凸集，一个圆和一个菱形。两个集合内的点（黄色）在投影期间保持不变。两个集合（黑色）之外的点投影到集合中接近原始点（黑色）的点（红色）。虽然对 L_2 的球面来说，方向保持不变，但一般情况下不需要这样。

凸投影的一个用途是计算稀疏权重向量。在本例中，我们将权重向量投影到一个 L_1 的球上，这是图11.2.4中菱形例子的一个广义版本。

小结

在深度学习的背景下，凸函数的主要目的是帮助我们详细了解优化算法。我们由此得出梯度下降法和随机梯度下降法是如何相应推导出来的。

- 凸集的交点是凸的，并集不是。
- 根据詹森不等式，“一个多变量凸函数的总期望值”大于或等于“用每个变量的期望值计算这个函数的总值”。
- 一个二次可微函数是凸函数，当且仅当其Hessian（二阶导数矩阵）是半正定的。
- 凸约束可以通过拉格朗日函数来添加。在实践中，只需在目标函数中加上一个惩罚就可以了。
- 投影映射到凸集中最接近原始点的点。

练习

1. 假设我们想要通过绘制集合内点之间的所有直线并检查这些直线是否包含来验证集合的凸性。*i.* 证明只检查边界上的点是充分的。*ii.* 证明只检查集合的顶点是充分的。
2. 用 p -范数表示半径为 r 的球，证明 $\mathcal{B}_p[r] := \{\mathbf{x} | \mathbf{x} \in \mathbb{R}^d \text{ and } \|\mathbf{x}\|_p \leq r\}$ ， $\mathcal{B}_p[r]$ 对于所有 $p \geq 1$ 是凸的。
3. 已知凸函数 f 和 g 表明 $\max(f, g)$ 也是凸函数。证明 $\min(f, g)$ 是非凸的。
4. 证明Softmax函数的规范化是凸的，即 $f(x) = \log \sum_i \exp(x_i)$ 的凸性。
5. 证明线性子空间 $\mathcal{X} = \{\mathbf{x} | \mathbf{W}\mathbf{x} = \mathbf{b}\}$ 是凸集。
6. 证明在线性子空间 $\mathbf{b} = \mathbf{0}$ 的情况下，对于矩阵 \mathbf{M} 的投影 $\text{Proj}_{\mathcal{X}}$ 可以写成 $\mathbf{M}\mathbf{x}$ 。
7. 证明对于凸二次可微函数 f ，对于 $\xi \in [0, \epsilon]$ ，我们可以写成 $f(x + \epsilon) = f(x) + \epsilon f'(x) + \frac{1}{2}\epsilon^2 f''(x + \xi)$ 。
8. 给定一个凸集 \mathcal{X} 和两个向量 \mathbf{x} 和 \mathbf{y} 证明了投影不会增加距离，即 $\|\mathbf{x} - \mathbf{y}\| \geq \|\text{Proj}_{\mathcal{X}}(\mathbf{x}) - \text{Proj}_{\mathcal{X}}(\mathbf{y})\|$ 。

Discussions¹²⁷

11.3 梯度下降

尽管梯度下降（gradient descent）很少直接用于深度学习，但了解它是理解下一节随机梯度下降算法的关键。例如，由于学习率过大，优化问题可能会发散，这种现象早已在梯度下降中出现。同样地，预处理（preconditioning）是梯度下降中的一种常用技术，还被沿用到更高级的算法中。让我们从简单的一维梯度下降开始。

¹²⁷ <https://discuss.d2l.ai/t/3815>

11.3.1 一维梯度下降

为什么梯度下降算法可以优化目标函数？一维中的梯度下降给我们很好的启发。考虑一类连续可微实值函数 $f : \mathbb{R} \rightarrow \mathbb{R}$ ，利用泰勒展开，我们可以得到

$$f(x + \epsilon) = f(x) + \epsilon f'(x) + \mathcal{O}(\epsilon^2). \quad (11.3.1)$$

即在一阶近似中， $f(x + \epsilon)$ 可通过 x 处的函数值 $f(x)$ 和一阶导数 $f'(x)$ 得出。我们可以假设在负梯度方向上移动的 ϵ 会减少 f 。为了简单起见，我们选择固定步长 $\eta > 0$ ，然后取 $\epsilon = -\eta f'(x)$ 。将其代入泰勒展开式我们可以得到

$$f(x - \eta f'(x)) = f(x) - \eta f'^2(x) + \mathcal{O}(\eta^2 f'^2(x)). \quad (11.3.2)$$

如果其导数 $f'(x) \neq 0$ 没有消失，我们就能继续展开，这是因为 $\eta f'^2(x) > 0$ 。此外，我们总是可以令 η 小到足以使高阶项变得不相关。因此，

$$f(x - \eta f'(x)) \lesssim f(x). \quad (11.3.3)$$

这意味着，如果我们使用

$$x \leftarrow x - \eta f'(x) \quad (11.3.4)$$

来迭代 x ，函数 $f(x)$ 的值可能会下降。因此，在梯度下降中，我们首先选择初始值 x 和常数 $\eta > 0$ ，然后使用它们连续迭代 x ，直到停止条件达成。例如，当梯度 $|f'(x)|$ 的幅度足够小或迭代次数达到某个值时。

下面我们来展示如何实现梯度下降。为了简单起见，我们选用目标函数 $f(x) = x^2$ 。尽管我们知道 $x = 0$ 时 $f(x)$ 能取得最小值，但我们仍然使用这个简单的函数来观察 x 的变化。

```
%matplotlib inline
import numpy as np
import torch
from d2l import torch as d2l
```

```
def f(x):  # 目标函数
    return x ** 2

def f_grad(x):  # 目标函数的梯度(导数)
    return 2 * x
```

接下来，我们使用 $x = 10$ 作为初始值，并假设 $\eta = 0.2$ 。使用梯度下降法迭代 x 共 10 次，我们可以看到， x 的值最终将接近最优解。

```
def gd(eta, f_grad):
    x = 10.0
```

(continues on next page)

(continued from previous page)

```
results = [x]
for i in range(10):
    x -= eta * f_grad(x)
    results.append(float(x))
print(f'epoch 10, x: {x:f}')
return results

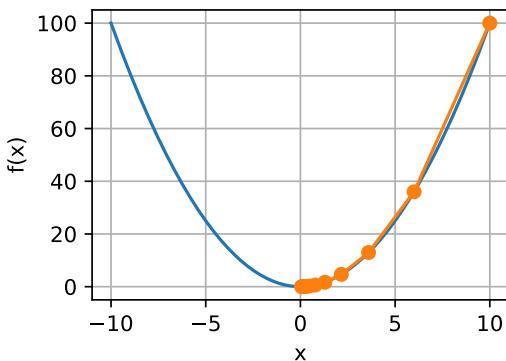
results = gd(0.2, f_grad)
```

```
epoch 10, x: 0.060466
```

对进行 x 优化的过程可以绘制如下。

```
def show_trace(results, f):
    n = max(abs(min(results)), abs(max(results)))
    f_line = torch.arange(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, results], [[f(x) for x in f_line], [
        f(x) for x in results]], 'x', 'f(x)', fmts=['-', '-o'])
```

```
show_trace(results, f)
```

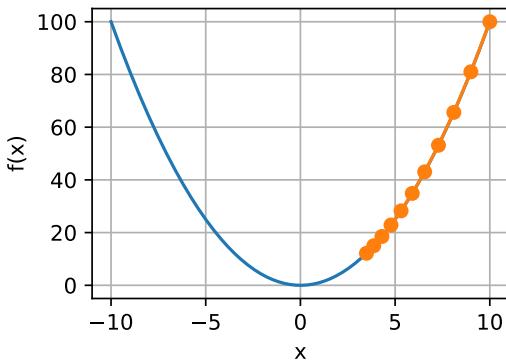


学习率

学习率 (learning rate) 决定目标函数能否收敛到局部最小值，以及何时收敛到最小值。学习率 η 可由算法设计者设置。请注意，如果我们使用的学习率太小，将导致 x 的更新非常缓慢，需要更多的迭代。例如，考虑同一优化问题中 $\eta = 0.05$ 的进度。如下所示，尽管经过了 10 个步骤，我们仍然离最优解很远。

```
show_trace(gd(0.05, f_grad), f)
```

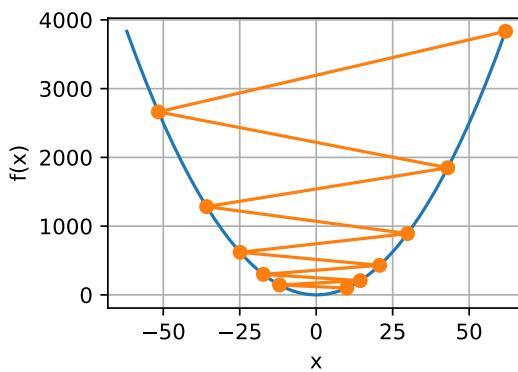
```
epoch 10, x: 3.486784
```



相反，如果我们使用过高的学习率， $|\eta f'(x)|$ 对于一阶泰勒展开式可能太大。也就是说，(11.3.1)中的 $\mathcal{O}(\eta^2 f'^2(x))$ 可能变得显著了。在这种情况下， x 的迭代不能保证降低 $f(x)$ 的值。例如，当学习率为 $\eta = 1.1$ 时， x 超出了最优解 $x = 0$ 并逐渐发散。

```
show_trace(gd(1.1, f_grad), f)
```

```
epoch 10, x: 61.917364
```



局部最小值

为了演示非凸函数的梯度下降，考虑函数 $f(x) = x \cdot \cos(cx)$ ，其中 c 为某常数。这个函数有无穷多个局部最小值。根据我们选择的学习率，我们最终可能只会得到许多解的一个。下面的例子说明了（不切实际的）高学习率如何导致较差的局部最小值。

```
c = torch.tensor(0.15 * np.pi)
```

(continues on next page)

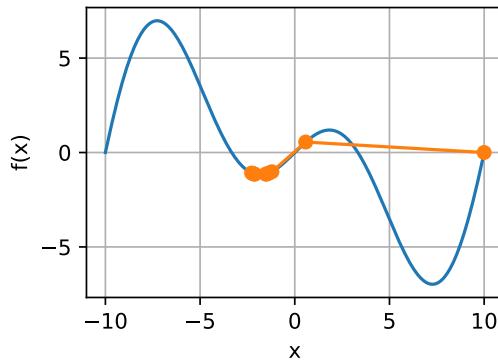
(continued from previous page)

```
def f(x): # 目标函数
    return x * torch.cos(c * x)

def f_grad(x): # 目标函数的梯度
    return torch.cos(c * x) - c * x * torch.sin(c * x)

show_trace(gd(2, f_grad), f)
```

```
epoch 10, x: -1.528166
```



11.3.2 多元梯度下降

现在我们对单变量的情况有了更好的理解，让我们考虑一下 $\mathbf{x} = [x_1, x_2, \dots, x_d]^\top$ 的情况。即目标函数 $f : \mathbb{R}^d \rightarrow \mathbb{R}$ 将向量映射成标量。相应地，它的梯度也是多元的，它是一个由 d 个偏导数组成的向量：

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_d} \right]^\top. \quad (11.3.5)$$

梯度中的每个偏导数元素 $\partial f(\mathbf{x})/\partial x_i$ 代表了当输入 x_i 时 f 在 \mathbf{x} 处的变化率。和先前单变量的情况一样，我们可以对多变量函数使用相应的泰勒近似来思考。具体来说，

$$f(\mathbf{x} + \boldsymbol{\epsilon}) = f(\mathbf{x}) + \boldsymbol{\epsilon}^\top \nabla f(\mathbf{x}) + \mathcal{O}(\|\boldsymbol{\epsilon}\|^2). \quad (11.3.6)$$

换句话说，在 $\boldsymbol{\epsilon}$ 的二阶项中，最陡下降的方向由负梯度 $-\nabla f(\mathbf{x})$ 得出。选择合适的学习率 $\eta > 0$ 来生成典型的梯度下降算法：

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x}). \quad (11.3.7)$$

这个算法在实践中的表现如何呢？我们构造一个目标函数 $f(\mathbf{x}) = x_1^2 + 2x_2^2$ ，并有二维向量 $\mathbf{x} = [x_1, x_2]^\top$ 作为输入，标量作为输出。梯度由 $\nabla f(\mathbf{x}) = [2x_1, 4x_2]^\top$ 给出。我们将从初始位置 $[-5, -2]$ 通过梯度下降观察 \mathbf{x} 的轨迹。

我们还需要两个辅助函数：第一个是 `update` 函数，并将其应用于初始值 20 次；第二个函数会显示 \mathbf{x} 的轨迹。

```

def train_2d(trainer, steps=20, f_grad=None): #@save
    """用定制的训练机优化2D目标函数"""
    # s1和s2是稍后将使用的内部状态变量
    x1, x2, s1, s2 = -5, -2, 0, 0
    results = [(x1, x2)]
    for i in range(steps):
        if f_grad:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2, f_grad)
        else:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
        results.append((x1, x2))
    print(f'epoch {i + 1}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results

```

```

def show_trace_2d(f, results): #@save
    """显示优化过程中2D变量的轨迹"""
    d2l.set_figsize()
    d2l.plt.plot(*zip(*results), '-o', color='#ff7f0e')
    x1, x2 = torch.meshgrid(torch.arange(-5.5, 1.0, 0.1),
                           torch.arange(-3.0, 1.0, 0.1), indexing='ij')
    d2l.plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')
    d2l.plt.xlabel('x1')
    d2l.plt.ylabel('x2')

```

接下来，我们观察学习率 $\eta = 0.1$ 时优化变量 \mathbf{x} 的轨迹。可以看到，经过20步之后， \mathbf{x} 的值接近其位于 $[0, 0]$ 的最小值。虽然进展相当顺利，但相当缓慢。

```

def f_2d(x1, x2): # 目标函数
    return x1 ** 2 + 2 * x2 ** 2

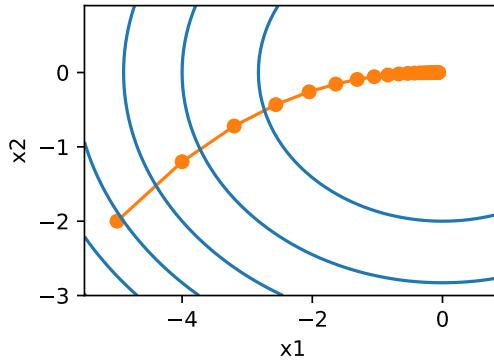
def f_2d_grad(x1, x2): # 目标函数的梯度
    return (2 * x1, 4 * x2)

def gd_2d(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    return (x1 - eta * g1, x2 - eta * g2, 0, 0)

eta = 0.1
show_trace_2d(f_2d, train_2d(gd_2d, f_grad=f_2d_grad))

```

```
epoch 20, x1: -0.057646, x2: -0.000073
```



11.3.3 自适应方法

正如我们在 11.3.1 节中所看到的，选择“恰到好处”的学习率 η 是很棘手的。如果我们把它选得太小，就没有什么进展；如果太大，得到的解就会振荡，甚至可能发散。如果我们可以自动确定 η ，或者完全不必选择学习率，会怎么样？除了考虑目标函数的值和梯度、还考虑它的曲率的二阶方法可以帮我们解决这个问题。虽然由于计算代价的原因，这些方法不能直接应用于深度学习，但它们为如何设计高级优化算法提供了有用的思想直觉，这些算法可以模拟下面概述的算法的许多理想特性。

牛顿法

回顾一些函数 $f : \mathbb{R}^d \rightarrow \mathbb{R}$ 的泰勒展开式，事实上我们可以把它写成

$$f(\mathbf{x} + \boldsymbol{\epsilon}) = f(\mathbf{x}) + \boldsymbol{\epsilon}^\top \nabla f(\mathbf{x}) + \frac{1}{2} \boldsymbol{\epsilon}^\top \nabla^2 f(\mathbf{x}) \boldsymbol{\epsilon} + \mathcal{O}(\|\boldsymbol{\epsilon}\|^3). \quad (11.3.8)$$

为了避免繁琐的符号，我们将 $\mathbf{H} \stackrel{\text{def}}{=} \nabla^2 f(\mathbf{x})$ 定义为 f 的 Hessian，是 $d \times d$ 矩阵。当 d 的值很小且问题很简单时， \mathbf{H} 很容易计算。但是对于深度神经网络而言，考虑到 \mathbf{H} 可能非常大， $\mathcal{O}(d^2)$ 个条目的存储代价会很高，此外通过反向传播进行计算可能雪上加霜。然而，我们姑且先忽略这些考量，看看会得到什么算法。

毕竟， f 的最小值满足 $\nabla f = 0$ 。遵循 2.4 节中的微积分规则，通过取 $\boldsymbol{\epsilon}$ 对 (11.3.8) 的导数，再忽略不重要的高阶项，我们便得到

$$\nabla f(\mathbf{x}) + \mathbf{H} \boldsymbol{\epsilon} = 0 \text{ and hence } \boldsymbol{\epsilon} = -\mathbf{H}^{-1} \nabla f(\mathbf{x}). \quad (11.3.9)$$

也就是说，作为优化问题的一部分，我们需要将 Hessian 矩阵 \mathbf{H} 求逆。

举一个简单的例子，对于 $f(x) = \frac{1}{2}x^2$ ，我们有 $\nabla f(x) = x$ 和 $\mathbf{H} = 1$ 。因此，对于任何 x ，我们可以获得 $\boldsymbol{\epsilon} = -x$ 。换言之，单单一步就足以完美地收敛，而无须任何调整。我们在这里比较幸运：泰勒展开式是确切的，因为 $f(x + \boldsymbol{\epsilon}) = \frac{1}{2}x^2 + \epsilon x + \frac{1}{2}\epsilon^2$ 。

让我们看看其他问题。给定一个凸双曲余弦函数 c ，其中 c 为某些常数，我们可以看到经过几次迭代后，得到了 $x = 0$ 处的全局最小值。

```

c = torch.tensor(0.5)

def f(x): # 目标函数
    return torch.cosh(c * x)

def f_grad(x): # 目标函数的梯度
    return c * torch.sinh(c * x)

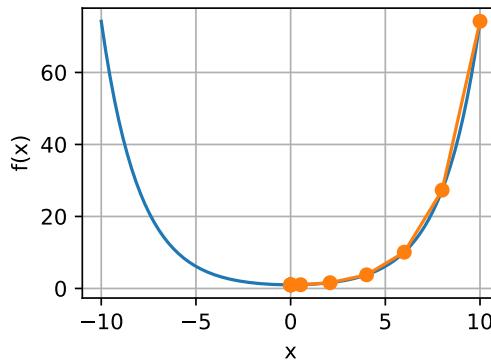
def f_hess(x): # 目标函数的Hessian
    return c**2 * torch.cosh(c * x)

def newton(eta=1):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x) / f_hess(x)
        results.append(float(x))
    print('epoch 10, x:', x)
    return results

show_trace(newton(), f)

```

```
epoch 10, x: tensor(0.)
```



现在让我们考虑一个非凸函数，比如 $f(x) = x \cos(cx)$, c 为某些常数。请注意在牛顿法中，我们最终将除以Hessian。这意味着如果二阶导数是负的， f 的值可能会趋于增加。这是这个算法的致命缺陷！让我们看看实践中会发生什么。

```

c = torch.tensor(0.15 * np.pi)

def f(x): # 目标函数

```

(continues on next page)

(continued from previous page)

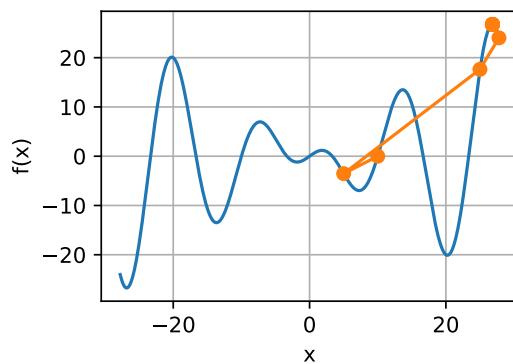
```
return x * torch.cos(c * x)

def f_grad(x): # 目标函数的梯度
    return torch.cos(c * x) - c * x * torch.sin(c * x)

def f_hess(x): # 目标函数的Hessian
    return - 2 * c * torch.sin(c * x) - x * c**2 * torch.cos(c * x)

show_trace(newton(), f)
```

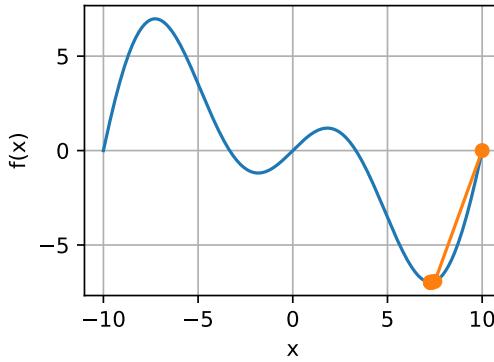
```
epoch 10, x: tensor(26.8341)
```



这发生了惊人的错误。我们怎样才能修正它？一种方法是用取Hessian的绝对值来修正，另一个策略是重新引入学习率。这似乎违背了初衷，但不完全是——拥有二阶信息可以使我们在曲率较大时保持谨慎，而在目标函数较平坦时则采用较大的学习率。让我们看看在学习率稍小的情况下它是如何生效的，比如 $\eta = 0.5$ 。如我们所见，我们有了一个相当高效的算法。

```
show_trace(newton(0.5), f)
```

```
epoch 10, x: tensor(7.2699)
```



收敛性分析

在此，我们以部分目标凸函数 f 为例，分析它们的牛顿法收敛速度。这些目标凸函数三次可微，而且二阶导数不为零，即 $f'' > 0$ 。由于多变量情况下的证明是对以下一维参数情况证明的直接拓展，对我们理解这个问题不能提供更多帮助，因此我们省略了多变量情况的证明。

用 $x^{(k)}$ 表示 x 在第 k^{th} 次迭代时的值，令 $e^{(k)} \stackrel{\text{def}}{=} x^{(k)} - x^*$ 表示 k^{th} 迭代时与最优性的距离。通过泰勒展开，我们得到条件 $f'(x^*) = 0$ 可以写成

$$0 = f'(x^{(k)} - e^{(k)}) = f'(x^{(k)}) - e^{(k)}f''(x^{(k)}) + \frac{1}{2}(e^{(k)})^2 f'''(\xi^{(k)}), \quad (11.3.10)$$

这对某些 $\xi^{(k)} \in [x^{(k)} - e^{(k)}, x^{(k)}]$ 成立。将上述展开除以 $f''(x^{(k)})$ 得到

$$e^{(k)} - \frac{f'(x^{(k)})}{f''(x^{(k)})} = \frac{1}{2}(e^{(k)})^2 \frac{f'''(\xi^{(k)})}{f''(x^{(k)})}. \quad (11.3.11)$$

回想之前的方程 $x^{(k+1)} = x^{(k)} - f'(x^{(k)})/f''(x^{(k)})$ 。代入这个更新方程，取两边的绝对值，我们得到

$$\left| e^{(k+1)} \right| = \frac{1}{2}(e^{(k)})^2 \frac{|f'''(\xi^{(k)})|}{|f''(x^{(k)})|}. \quad (11.3.12)$$

因此，每当我们处于有界区域 $|f'''(\xi^{(k)})| / (2f''(x^{(k)})) \leq c$ ，我们就有一个二次递减误差

$$\left| e^{(k+1)} \right| \leq c(e^{(k)})^2. \quad (11.3.13)$$

另一方面，优化研究人员称之为“线性”收敛，而将 $|e^{(k+1)}| \leq \alpha |e^{(k)}|$ 这样的条件称为“恒定”收敛速度。请注意，我们无法估计整体收敛的速度，但是一旦我们接近极小值，收敛将变得非常快。另外，这种分析要求 f 在高阶导数上表现良好，即确保 f 在如何变化它的值方面没有任何“超常”的特性。

预处理

计算和存储完整的Hessian非常昂贵，而改善这个问题的一种方法是“预处理”。它回避了计算整个Hessian，而只计算“对角线”项，即如下的算法更新：

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \text{diag}(\mathbf{H})^{-1} \nabla f(\mathbf{x}). \quad (11.3.14)$$

虽然这不如完整的牛顿法精确，但它仍然比不使用要好得多。为什么预处理有效呢？假设一个变量以毫米表示高度，另一个变量以公里表示高度的情况。假设这两种自然尺度都以米为单位，那么我们的参数化就出现了严重的不匹配。幸运的是，使用预处理可以消除这种情况。梯度下降的有效预处理相当于为每个变量选择不同的学习率（矢量 \mathbf{x} 的坐标）。我们将在后面一节看到，预处理推动了随机梯度下降优化算法的一些创新。

梯度下降和线搜索

梯度下降的一个关键问题是可能会超过目标或进展不足，解决这一问题的简单方法是结合使用线搜索和梯度下降。也就是说，我们使用 $\nabla f(\mathbf{x})$ 给出的方向，然后进行二分搜索，以确定哪个学习率 η 使 $f(\mathbf{x} - \eta \nabla f(\mathbf{x}))$ 取最小值。

有关分析和证明，此算法收敛迅速（请参见 (Boyd and Vandenberghe, 2004)）。然而，对深度学习而言，这不太可行。因为线搜索的每一步都需要评估整个数据集上的目标函数，实现它的方式太昂贵了。

小结

- 学习率的大小很重要：学习率太大会使模型发散，学习率太小会没有进展。
- 梯度下降会可能陷入局部极小值，而得不到全局最小值。
- 在高维模型中，调整学习率是很复杂的。
- 预处理有助于调节比例。
- 牛顿法在凸问题中一旦开始正常工作，速度就会快得多。
- 对于非凸问题，不要不作任何调整就使用牛顿法。

练习

1. 用不同的学习率和目标函数进行梯度下降实验。
2. 在区间 $[a, b]$ 中实现线搜索以最小化凸函数。
 1. 是否需要导数来进行二分搜索，即决定选择 $[a, (a + b)/2]$ 还是 $[(a + b)/2, b]$ 。
 2. 算法的收敛速度有多快？
 3. 实现该算法，并将其应用于求 $\log(\exp(x) + \exp(-2x - 3))$ 的最小值。
3. 设计一个定义在 \mathbb{R}^2 上的目标函数，它的梯度下降非常缓慢。提示：不同坐标的缩放方式不同。
4. 使用预处理实现牛顿方法的轻量版本。
 1. 使用对角Hessian作为预条件子。
 2. 使用它的绝对值，而不是实际值（可能有符号）。
 3. 将此应用于上述问题。
5. 将上述算法应用于多个目标函数（凸或非凸）。如果把坐标旋转45度会怎么样？

11.4 随机梯度下降

在前面的章节中，我们一直在训练过程中使用随机梯度下降，但没有解释它为什么起作用。为了澄清这一点，我们刚在 11.3 节中描述了梯度下降的基本原则。本节继续更详细地说明随机梯度下降（stochastic gradient descent）。

```
%matplotlib inline
import math
import torch
from d2l import torch as d2l
```

11.4.1 随机梯度更新

在深度学习中，目标函数通常是训练数据集中每个样本的损失函数的平均值。给定 n 个样本的训练数据集，我们假设 $f_i(\mathbf{x})$ 是关于索引 i 的训练样本的损失函数，其中 \mathbf{x} 是参数向量。然后我们得到目标函数

$$f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n f_i(\mathbf{x}). \quad (11.4.1)$$

\mathbf{x} 的目标函数的梯度计算为

$$\nabla f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{x}). \quad (11.4.2)$$

如果使用梯度下降法，则每个自变量迭代的计算代价为 $\mathcal{O}(n)$ ，它随 n 线性增长。因此，当训练数据集较大时，每次迭代的梯度下降计算代价将较高。

随机梯度下降（SGD）可降低每次迭代时的计算代价。在随机梯度下降的每次迭代中，我们对数据样本随机均匀采样一个索引 i ，其中 $i \in \{1, \dots, n\}$ ，并计算梯度 $\nabla f_i(\mathbf{x})$ 以更新 \mathbf{x} ：

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}), \quad (11.4.3)$$

其中 η 是学习率。我们可以看到，每次迭代的计算代价从梯度下降的 $\mathcal{O}(n)$ 降至常数 $\mathcal{O}(1)$ 。此外，我们要强调，随机梯度 $\nabla f_i(\mathbf{x})$ 是对完整梯度 $\nabla f(\mathbf{x})$ 的无偏估计，因为

$$\mathbb{E}_i \nabla f_i(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}). \quad (11.4.4)$$

这意味着，平均而言，随机梯度是对梯度的良好估计。

现在，我们将把它与梯度下降进行比较，方法是向梯度添加均值为 0、方差为 1 的随机噪声，以模拟随机梯度下降。

¹²⁸ <https://discuss.d2l.ai/t/3836>

```

def f(x1, x2): # 目标函数
    return x1 ** 2 + 2 * x2 ** 2

def f_grad(x1, x2): # 目标函数的梯度
    return 2 * x1, 4 * x2

```

```

def sgd(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    # 模拟有噪声的梯度
    g1 += torch.normal(0.0, 1, (1,)).item()
    g2 += torch.normal(0.0, 1, (1,)).item()
    eta_t = eta * lr()
    return (x1 - eta_t * g1, x2 - eta_t * g2, 0, 0)

```

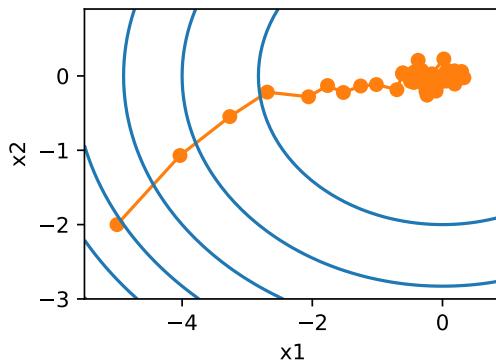
```

def constant_lr():
    return 1

eta = 0.1
lr = constant_lr # 常数学习速度
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))

```

epoch 50, x1: 0.020569, x2: 0.227895



正如我们所看到的，随机梯度下降中变量的轨迹比我们在 11.3 节中观察到的梯度下降中观察到的轨迹嘈杂得多。这是由于梯度的随机性质。也就是说，即使我们接近最小值，我们仍然受到通过 $\eta \nabla f_i(\mathbf{x})$ 的瞬间梯度所注入的不确定性的影响。即使经过 50 次迭代，质量仍然不那么好。更糟糕的是，经过额外的步骤，它不会得到改善。这给我们留下了唯一的选择：改变学习率 η 。但是，如果我们选择的学习率太小，我们一开始就不会取得任何有意义的进展。另一方面，如果我们选择的学习率太大，我们将无法获得一个好的解决方案，如上所示。解决这些相互冲突的目标的唯一方法是在优化过程中动态降低学习率。

这也是在 sgd 步长函数中添加学习率函数 lr 的原因。在上面的示例中，学习率调度的任何功能都处于休眠状

态，因为我们将相关的lr函数设置为常量。

11.4.2 动态学习率

用与时间相关的学习率 $\eta(t)$ 取代 η 增加了控制优化算法收敛的复杂性。特别是，我们需要弄清 η 的衰减速度。如果太快，我们将过早停止优化。如果减少的太慢，我们会在优化上浪费太多时间。以下是随着时间推移调整 η 时使用的一些基本策略（稍后我们将讨论更高级的策略）：

$$\begin{aligned}\eta(t) &= \eta_i \text{ if } t_i \leq t \leq t_{i+1} && \text{分段常数} \\ \eta(t) &= \eta_0 \cdot e^{-\lambda t} && \text{指数衰减} \\ \eta(t) &= \eta_0 \cdot (\beta t + 1)^{-\alpha} && \text{多项式衰减}\end{aligned}\quad (11.4.5)$$

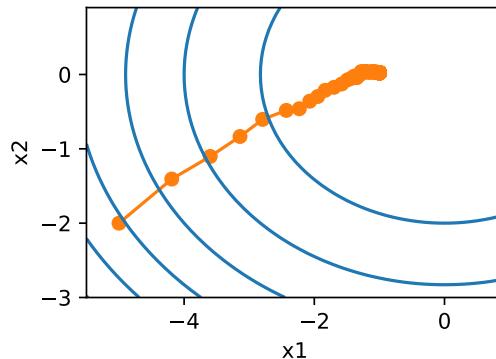
在第一个分段常数（piecewise constant）场景中，我们会降低学习率，例如，每当优化进度停顿时。这是训练深度网络的常见策略。或者，我们可以通过指数衰减（exponential decay）来更积极地减低它。不幸的是，这往往会导致算法收敛之前过早停止。一个受欢迎的选择是 $\alpha = 0.5$ 的多项式衰减（polynomial decay）。在凸优化的情况下，有许多证据表明这种速率表现良好。

让我们看看指数衰减在实践中是什么样子。

```
def exponential_lr():
    # 在函数外部定义，而在内部更新的全局变量
    global t
    t += 1
    return math.exp(-0.1 * t)

t = 1
lr = exponential_lr
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=1000, f_grad=f_grad))
```

```
epoch 1000, x1: -0.998659, x2: 0.023408
```



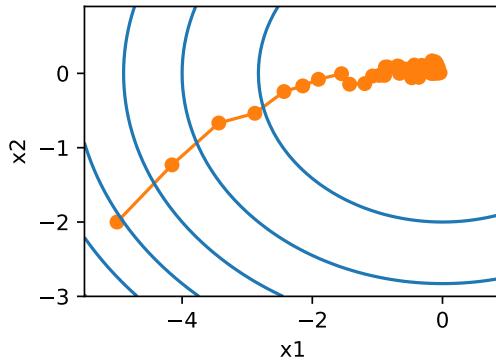
正如预期的那样，参数的方差大大减少。但是，这是以未能收敛到最优解 $\mathbf{x} = (0, 0)$ 为代价的。即使经过1000个

迭代步骤，我们仍然离最优解很远。事实上，该算法根本无法收敛。另一方面，如果我们使用多项式衰减，其中学习率随迭代次数的平方根倒数衰减，那么仅在50次迭代之后，收敛就会更好。

```
def polynomial_lr():
    # 在函数外部定义，而在内部更新的全局变量
    global t
    t += 1
    return (1 + 0.1 * t) ** (-0.5)

t = 1
lr = polynomial_lr
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))
```

epoch 50, x1: -0.174174, x2: -0.000615



关于如何设置学习率，还有更多的选择。例如，我们可以从较小的学习率开始，然后使其迅速上涨，再让它降低，尽管这会更慢。我们甚至可以在较小和较大的学习率之间切换。现在，让我们专注于可以进行全面理论分析的学习率计划，即凸环境下的学习率。对一般的非凸问题，很难获得有意义的收敛保证，因为总的来说，最大限度地减少非线性非凸问题是NP困难的。有关的研究调查，请参阅例如2015年Tibshirani的优秀讲义笔记¹²⁹。

11.4.3 凸目标的收敛性分析

以下对凸目标函数的随机梯度下降的收敛性分析是可选读的，主要用于传达对问题的更多直觉。我们只限于最简单的证明之一 (Nesterov and Vial, 2000)。存在着明显更先进的证明技术，例如，当目标函数表现特别好时。

假设所有 ξ 的目标函数 $f(\xi, \mathbf{x})$ 在 \mathbf{x} 中都是凸的。更具体地说，我们考虑随机梯度下降更新：

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta_t \partial_{\mathbf{x}} f(\xi_t, \mathbf{x}), \quad (11.4.6)$$

¹²⁹ <https://www.stat.cmu.edu/~ryantibs/convexopt-F15/lectures/26-nonconvex.pdf>

其中 $f(\xi_t, \mathbf{x})$ 是训练样本 $f(\xi_t, \mathbf{x})$ 的目标函数： ξ_t 从第 t 步的某个分布中提取， \mathbf{x} 是模型参数。用

$$R(\mathbf{x}) = E_{\xi}[f(\xi, \mathbf{x})] \quad (11.4.7)$$

表示期望风险， R^* 表示对于 \mathbf{x} 的最低风险。最后让 \mathbf{x}^* 表示最小值（我们假设它存在于定义 \mathbf{x} 的域中）。在这种情况下，我们可以跟踪时间 t 处的当前参数 \mathbf{x}_t 和风险最小化器 \mathbf{x}^* 之间的距离，看看它是否随着时间的推移而改善：

$$\begin{aligned} & \| \mathbf{x}_{t+1} - \mathbf{x}^* \|^2 \\ &= \| \mathbf{x}_t - \eta_t \partial_{\mathbf{x}} f(\xi_t, \mathbf{x}) - \mathbf{x}^* \|^2 \\ &= \| \mathbf{x}_t - \mathbf{x}^* \|^2 + \eta_t^2 \| \partial_{\mathbf{x}} f(\xi_t, \mathbf{x}) \|^2 - 2\eta_t \langle \mathbf{x}_t - \mathbf{x}^*, \partial_{\mathbf{x}} f(\xi_t, \mathbf{x}) \rangle. \end{aligned} \quad (11.4.8)$$

我们假设随机梯度 $\partial_{\mathbf{x}} f(\xi_t, \mathbf{x})$ 的 L_2 范数受到某个常数 L 的限制，因此我们有

$$\eta_t^2 \| \partial_{\mathbf{x}} f(\xi_t, \mathbf{x}) \|^2 \leq \eta_t^2 L^2. \quad (11.4.9)$$

我们最感兴趣的是 \mathbf{x}_t 和 \mathbf{x}^* 之间的距离如何变化的期望。事实上，对于任何具体的步骤序列，距离可能会增加，这取决于我们遇到的 ξ_t 。因此我们需要点积的边界。因为对于任何凸函数 f ，所有 \mathbf{x} 和 \mathbf{y} 都满足 $f(\mathbf{y}) \geq f(\mathbf{x}) + \langle f'(\mathbf{x}), \mathbf{y} - \mathbf{x} \rangle$ ，按凸性我们有

$$f(\xi_t, \mathbf{x}^*) \geq f(\xi_t, \mathbf{x}_t) + \langle \mathbf{x}^* - \mathbf{x}_t, \partial_{\mathbf{x}} f(\xi_t, \mathbf{x}_t) \rangle. \quad (11.4.10)$$

将不等式 (11.4.9) 和 (11.4.10) 代入 (11.4.8) 我们在时间 $t+1$ 时获得参数之间距离的边界，如下所示：

$$\| \mathbf{x}_t - \mathbf{x}^* \|^2 - \| \mathbf{x}_{t+1} - \mathbf{x}^* \|^2 \geq 2\eta_t (f(\xi_t, \mathbf{x}_t) - f(\xi_t, \mathbf{x}^*)) - \eta_t^2 L^2. \quad (11.4.11)$$

这意味着，只要当前损失和最优损失之间的差异超过 $\eta_t L^2 / 2$ ，我们就会取得进展。由于这种差异必然会收敛到零，因此学习率 η_t 也需要消失。

接下来，我们根据 (11.4.11) 取期望。得到

$$E[\| \mathbf{x}_t - \mathbf{x}^* \|^2] - E[\| \mathbf{x}_{t+1} - \mathbf{x}^* \|^2] \geq 2\eta_t [E[R(\mathbf{x}_t)] - R^*] - \eta_t^2 L^2. \quad (11.4.12)$$

最后一步是对 $t \in \{1, \dots, T\}$ 的不等式求和。在求和过程中抵消中间项，然后舍去低阶项，可以得到

$$\| \mathbf{x}_1 - \mathbf{x}^* \|^2 \geq 2 \left(\sum_{t=1}^T \eta_t \right) [E[R(\mathbf{x}_t)] - R^*] - L^2 \sum_{t=1}^T \eta_t^2. \quad (11.4.13)$$

请注意，我们利用了给定的 \mathbf{x}_1 ，因而可以去掉期望。最后定义

$$\bar{\mathbf{x}} \stackrel{\text{def}}{=} \frac{\sum_{t=1}^T \eta_t \mathbf{x}_t}{\sum_{t=1}^T \eta_t}. \quad (11.4.14)$$

因为有

$$E \left(\frac{\sum_{t=1}^T \eta_t R(\mathbf{x}_t)}{\sum_{t=1}^T \eta_t} \right) = \frac{\sum_{t=1}^T \eta_t E[R(\mathbf{x}_t)]}{\sum_{t=1}^T \eta_t} = E[R(\bar{\mathbf{x}})], \quad (11.4.15)$$

根据詹森不等式（令 (11.2.3) 中 $i = t$, $\alpha_i = \eta_t / \sum_{t=1}^T \eta_t$ ）和 R 的凸性使其满足的 $E[R(\mathbf{x}_t)] \geq E[R(\bar{\mathbf{x}})]$ ，因此，

$$\sum_{t=1}^T \eta_t E[R(\mathbf{x}_t)] \geq \sum_{t=1}^T \eta_t E[R(\bar{\mathbf{x}})]. \quad (11.4.16)$$

将其代入不等式 (11.4.13) 得到边界

$$[E[\bar{\mathbf{x}}]] - R^* \leq \frac{r^2 + L^2 \sum_{t=1}^T \eta_t^2}{2 \sum_{t=1}^T \eta_t}, \quad (11.4.17)$$

其中 $r^2 \stackrel{\text{def}}{=} \|\mathbf{x}_1 - \mathbf{x}^*\|^2$ 是初始选择参数与最终结果之间距离的边界。简而言之，收敛速度取决于随机梯度标准的限制方式 (L) 以及初始参数值与最优结果的距离 (r)。请注意，边界由 $\bar{\mathbf{x}}$ 而不是 \mathbf{x}_T 表示。因为 $\bar{\mathbf{x}}$ 是优化路径的平滑版本。只要知道 r, L 和 T ，我们就可以选择学习率 $\eta = r/(L\sqrt{T})$ 。这个就是上界 rL/\sqrt{T} 。也就是说，我们将按照速度 $\mathcal{O}(1/\sqrt{T})$ 收敛到最优解。

11.4.4 随机梯度和有限样本

到目前为止，在谈论随机梯度下降时，我们进行得有点快而松散。我们假设从分布 $p(x, y)$ 中采样得到样本 x_i (通常带有标签 y_i)，并且用它来以某种方式更新模型参数。特别是，对于有限的样本数量，我们仅仅讨论了由某些允许我们在其上执行随机梯度下降的函数 δ_{x_i} 和 δ_{y_i} 组成的离散分布 $p(x, y) = \frac{1}{n} \sum_{i=1}^n \delta_{x_i}(x) \delta_{y_i}(y)$ 。

但是，这不是我们真正做的。在本节的简单示例中，我们只是将噪声添加到其他非随机梯度上，也就是说，我们假装有成对的 (x_i, y_i) 。事实证明，这种做法在这里是合理的（有关详细讨论，请参阅练习）。更麻烦的是，在以前的所有讨论中，我们显然没有这样做。相反，我们遍历了所有实例恰好一次。要了解为什么这更可取，可以反向考虑一下，即我们有替换地从离散分布中采样 n 个观测值。随机选择一个元素 i 的概率是 $1/n$ 。因此选择它至少一次就是

$$P(\text{choose } i) = 1 - P(\text{omit } i) = 1 - (1 - 1/n)^n \approx 1 - e^{-1} \approx 0.63. \quad (11.4.18)$$

类似的推理表明，挑选一些样本（即训练示例）恰好一次的概率是

$$\binom{n}{1} \frac{1}{n} \left(1 - \frac{1}{n}\right)^{n-1} = \frac{n}{n-1} \left(1 - \frac{1}{n}\right)^n \approx e^{-1} \approx 0.37. \quad (11.4.19)$$

这导致与无替换采样相比，方差增加并且数据效率降低。因此，在实践中我们执行后者（这是本书中的默认选择）。最后一点注意，重复采用训练数据集的时候，会以不同的随机顺序遍历它。

小结

- 对于凸问题，我们可以证明，对于广泛的学习率选择，随机梯度下降将收敛到最优解。
- 对于深度学习而言，情况通常并非如此。但是，对凸问题的分析使我们能够深入了解如何进行优化，即逐步降低学习率，尽管不是太快。
- 如果学习率太小或太大，就会出现问题。实际上，通常只有经过多次实验后才能找到合适的学习率。
- 当训练数据集中有更多样本时，计算梯度下降的每次迭代的代价更高，因此在这些情况下，首选随机梯度下降。
- 随机梯度下降的最优化保证在非凸情况下一般不可用，因为需要检查的局部最小值的数量可能是指数级的。

练习

- 尝试不同的随机梯度下降学习率计划和不同的迭代次数进行实验。特别是，根据迭代次数的函数来绘制与最优解 $(0, 0)$ 的距离。
- 证明对于函数 $f(x_1, x_2) = x_1^2 + 2x_2^2$ 而言，向梯度添加正态噪声等同于最小化损失函数 $f(\mathbf{x}, \mathbf{w}) = (x_1 - w_1)^2 + 2(x_2 - w_2)^2$ ，其中 \mathbf{x} 是从正态分布中提取的。
- 从 $\{(x_1, y_1), \dots, (x_n, y_n)\}$ 分别使用替换方法以及不替换方法进行采样时，比较随机梯度下降的收敛性。
- 如果某些梯度（或者更确切地说与之相关的某些坐标）始终比所有其他梯度都大，将如何更改随机梯度下降求解器？
- 假设 $f(x) = x^2(1 + \sin x)$ 。 f 有多少局部最小值？请试着改变 f 以尽量减少它需要评估所有局部最小值的方式。

Discussions¹³⁰

11.5 小批量随机梯度下降

到目前为止，我们在基于梯度的学习方法中遇到了两个极端情况：[11.3节](#)中使用完整数据集来计算梯度并更新参数，[11.4节](#)中一次处理一个训练样本来取得进展。二者各有利弊：每当数据非常相似时，梯度下降并不是非常“数据高效”。而由于CPU和GPU无法充分利用向量化，随机梯度下降并不特别“计算高效”。这暗示了两者之间可能有折中方案，这便涉及到小批量随机梯度下降（minibatch gradient descent）。

11.5.1 向量化和缓存

使用小批量的决策的核心是计算效率。当考虑与多个GPU和多台服务器并行处理时，这一点最容易被理解。在这种情况下，我们需要向每个GPU发送至少一张图像。有了每台服务器8个GPU和16台服务器，我们就能得到大小为128的小批量。

当涉及到单个GPU甚至CPU时，事情会更微妙一些：这些设备有多种类型的内存、通常情况下多种类型的计算单元以及在它们之间不同的带宽限制。例如，一个CPU有少量寄存器（register），L1和L2缓存，以及L3缓存（在不同的处理器内核之间共享）。随着缓存的大小的增加，它们的延迟也在增加，同时带宽在减少。可以说，处理器能够执行的操作远比主内存接口所能提供的多得多。

首先，具有16个内核和AVX-512向量化的2GHz CPU每秒可处理高达 $2 \cdot 10^9 \cdot 16 \cdot 32 = 10^{12}$ 个字节。同时，GPU的性能很容易超过该数字100倍。而另一方面，中端服务器处理器的带宽可能不超过100Gb/s，即不到处理器满负荷所需的十分之一。更糟糕的是，并非所有的内存入口都是相等的：内存接口通常为64位或更宽（例如，在最多384位的GPU上）。因此读取单个字节会导致由于更宽的存取而产生的代价。

其次，第一次存取的额外开销很大，而按序存取（sequential access）或突发读取（burst read）相对开销较小。有关更深入的讨论，请参阅此[维基百科文章](#)¹³¹。

¹³⁰ <https://discuss.d2l.ai/t/3838>

¹³¹ https://en.wikipedia.org/wiki/Cache_hierarchy

减轻这些限制的方法是使用足够快的CPU缓存层次结构来为处理器提供数据。这是深度学习中批量处理背后的推动力。举一个简单的例子：矩阵-矩阵乘法。比如 $\mathbf{A} = \mathbf{BC}$ ，我们有很多方法来计算 \mathbf{A} 。例如，我们可以尝试以下方法：

1. 我们可以计算 $\mathbf{A}_{ij} = \mathbf{B}_{i,:}\mathbf{C}_{:,j}^\top$ ，也就是说，我们可以通过点积进行逐元素计算。
2. 我们可以计算 $\mathbf{A}_{:,j} = \mathbf{B}\mathbf{C}_{:,j}^\top$ ，也就是说，我们可以一次计算一列。同样，我们可以一次计算 \mathbf{A} 一行 $\mathbf{A}_{i,:}$ 。
3. 我们可以简单地计算 $\mathbf{A} = \mathbf{BC}$ 。
4. 我们可以将 \mathbf{B} 和 \mathbf{C} 分成较小的区块矩阵，然后一次计算 \mathbf{A} 的一个区块。

如果我们使用第一个选择，每次我们计算一个元素 \mathbf{A}_{ij} 时，都需要将一行和一列向量复制到CPU中。更糟糕的是，由于矩阵元素是按顺序对齐的，因此当从内存中读取它们时，我们需要访问两个向量中许多不相交的位置。第二种选择相对更有利：我们能够在遍历 \mathbf{B} 的同时，将列向量 $\mathbf{C}_{:,j}$ 保留在CPU缓存中。它将内存带宽需求减半，相应地提高了访问速度。第三种选择表面上是最可取的，然而大多数矩阵可能不能完全放入缓存中。第四种选择提供了一个实践上很有用的方案：我们可以将矩阵的区块移到缓存中然后在本地将它们相乘。让我们来看看这些操作在实践中的效率如何。

除了计算效率之外，Python和深度学习框架本身带来的额外开销也是相当大的。回想一下，每次我们执行代码时，Python解释器都会向深度学习框架发送一个命令，要求将其插入到计算图中并在调度过程中处理它。这样的额外开销可能是非常不利的。总而言之，我们最好用向量化（和矩阵）。

```
%matplotlib inline
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l

timer = d2l.Timer()
A = torch.zeros(256, 256)
B = torch.randn(256, 256)
C = torch.randn(256, 256)
```

按元素分配只需遍历分别为 \mathbf{B} 和 \mathbf{C} 的所有行和列，即可将该值分配给 \mathbf{A} 。

```
# 逐元素计算A=BC
timer.start()
for i in range(256):
    for j in range(256):
        A[i, j] = torch.dot(B[i, :], C[:, j])
timer.stop()
```

```
1.6609790325164795
```

更快的策略是执行按列分配。

```
# 逐列计算A=BC
timer.start()
for j in range(256):
    A[:, j] = torch.mv(B, C[:, j])
timer.stop()
```

0.022588253021240234

最有效的方法是在一个区块中执行整个操作。让我们看看它们各自的操作速度是多少。

```
# 一次性计算A=BC
timer.start()
A = torch.mm(B, C)
timer.stop()

# 乘法和加法作为单独的操作（在实践中融合）
gigaflops = [2/i for i in timer.times]
print(f'performance in Gigaflops: element {gigaflops[0]:.3f}, '
      f'column {gigaflops[1]:.3f}, full {gigaflops[2]:.3f}')
```

performance in Gigaflops: element 1.204, column 88.542, full 195.625

11.5.2 小批量

之前我们会理所当然地读取数据的小批量，而不是观测单个数据来更新参数，现在简要解释一下原因。处理单个观测值需要我们执行许多单一矩阵-矢量（甚至矢量-矢量）乘法，这耗费相当大，而且对应深度学习框架也要巨大的开销。这既适用于计算梯度以更新参数时，也适用于用神经网络预测。也就是说，每当我们执行 $\mathbf{w} \leftarrow \mathbf{w} - \eta_t \mathbf{g}_t$ 时，消耗巨大。其中

$$\mathbf{g}_t = \partial_{\mathbf{w}} f(\mathbf{x}_t, \mathbf{w}). \quad (11.5.1)$$

我们可以通过将其应用于一个小批量观测值来提高此操作的计算效率。也就是说，我们将梯度 \mathbf{g}_t 替换为一个小批量而不是单个观测值

$$\mathbf{g}_t = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}). \quad (11.5.2)$$

让我们看看这对 \mathbf{g}_t 的统计属性有什么影响：由于 \mathbf{x}_t 和小批量 \mathcal{B}_t 的所有元素都是从训练集中随机抽出的，因此梯度的期望保持不变。另一方面，方差显著降低。由于小批量梯度由正在被平均计算的 $b := |\mathcal{B}_t|$ 个独立梯度组成，其标准差降低了 $b^{-\frac{1}{2}}$ 。这本身就是一件好事，因为这意味着更新与完整的梯度更接近了。

直观来说，这表明选择大型的小批量 \mathcal{B}_t 将是普遍可行的。然而，经过一段时间后，与计算代价的线性增长相比，标准差的额外减少是微乎其微的。在实践中我们选择一个足够大的小批量，它可以提供良好的计算效率

同时仍适合GPU的内存。下面，我们来看看这些高效的代码。在里面我们执行相同的矩阵-矩阵乘法，但是这次我们将其一次性分为64列的“小批量”。

```
timer.start()
for j in range(0, 256, 64):
    A[:, j:j+64] = torch.mm(B, C[:, j:j+64])
timer.stop()
print(f'performance in GigaFlops: block {2 / timer.times[3]:.3f}')
```

```
performance in GigaFlops: block 2056.535
```

显而易见，小批量上的计算基本上与完整矩阵一样有效。需要注意的是，在7.5节中，我们使用了一种在很大程度上取决于小批量中的方差的正则化。随着后者增加，方差会减少，随之而来的是批量规范化带来的噪声注入的好处。关于实例，请参阅(Ioffe, 2017)，了解有关如何重新缩放并计算适当项目。

11.5.3 读取数据集

让我们来看看如何从数据中有效地生成小批量。下面我们使用NASA开发的测试机翼的数据集不同飞行器产生的噪声¹³²来比较这些优化算法。为方便起见，我们只使用前1,500样本。数据已作预处理：我们移除了均值并将方差重新缩放到每个坐标为1。

```
#@save
d2l.DATA_HUB['airfoil'] = (d2l.DATA_URL + 'airfoil_self_noise.dat',
                            '76e5be1548fd8222e5074cf0faae75edff8cf93f')

#@save
def get_data_ch11(batch_size=10, n=1500):
    data = np.genfromtxt(d2l.download('airfoil'),
                         dtype=np.float32, delimiter='\t')
    data = torch.from_numpy((data - data.mean(axis=0)) / data.std(axis=0))
    data_iter = d2l.load_array((data[:n, :-1], data[:n, -1]),
                               batch_size, is_train=True)
    return data_iter, data.shape[1]-1
```

¹³² <https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise>

11.5.4 从零开始实现

3.2节一节中已经实现过小批量随机梯度下降算法。我们在这里将它的输入参数变得更加通用，主要是为了方便本章后面介绍的其他优化算法也可以使用同样的输入。具体来说，我们添加了一个状态输入states并将超参数放在字典hyperparams中。此外，我们将在训练函数里对各个小批量样本的损失求平均，因此优化算法中的梯度不需要除以批量大小。

```
def sgd(params, states, hyperparams):
    for p in params:
        p.data.sub_(hyperparams['lr'] * p.grad)
        p.grad.data.zero_()
```

下面实现一个通用的训练函数，以方便本章后面介绍的其他优化算法使用。它初始化了一个线性回归模型，然后可以使用小批量随机梯度下降以及后续小节介绍的其他算法来训练模型。

```
#@save
def train_ch11(trainer_fn, states, hyperparams, data_iter,
               feature_dim, num_epochs=2):
    # 初始化模型
    w = torch.normal(mean=0.0, std=0.01, size=(feature_dim, 1),
                     requires_grad=True)
    b = torch.zeros((1), requires_grad=True)
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    # 训练模型
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                             xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            l = loss(net(X), y).mean()
            l.backward()
            trainer_fn([w, b], states, hyperparams)
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss),))
                timer.start()
        print(f'loss: {animator.Y[-1]:.3f}, {timer.avg():.3f} sec/epoch')
    return timer.cumsum(), animator.Y
```

让我们来看看批量梯度下降的优化是如何进行的。这可以通过将小批量设置为1500（即样本总数）来实现。因此，模型参数每个迭代轮数只迭代一次。

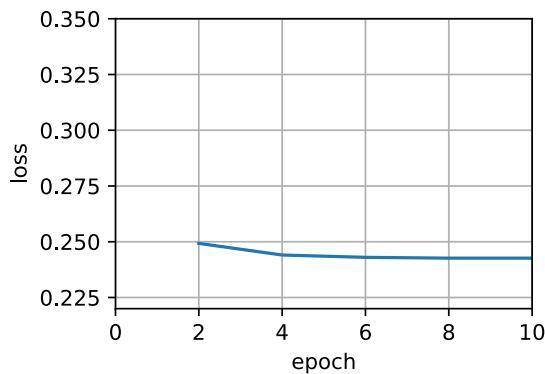
```

def train_sgd(lr, batch_size, num_epochs=2):
    data_iter, feature_dim = get_data_ch11(batch_size)
    return train_ch11(
        sgd, None, {'lr': lr}, data_iter, feature_dim, num_epochs)

gd_res = train_sgd(1, 1500, 10)

```

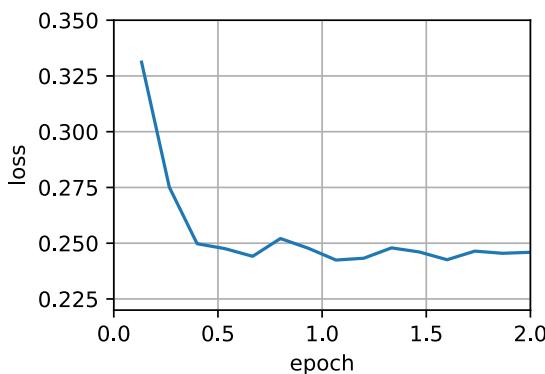
loss: 0.243, 0.055 sec/epoch



当批量大小为1时，优化使用的是随机梯度下降。为了简化实现，我们选择了很小的学习率。在随机梯度下降的实验中，每当一个样本被处理，模型参数都会更新。在这个例子中，这相当于每个迭代轮数有1500次更新。可以看到，目标函数值的下降在1个迭代轮数后就变得较为平缓。尽管两个例子在一个迭代轮数内都处理了1500个样本，但实验中随机梯度下降的一个迭代轮数耗时更多。这是因为随机梯度下降更频繁地更新了参数，而且一次处理单个观测值效率较低。

```
sgd_res = train_sgd(0.005, 1)
```

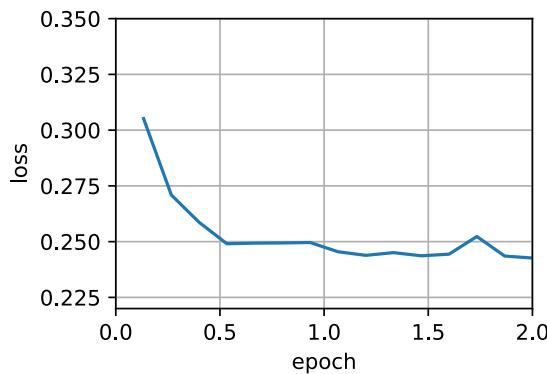
loss: 0.246, 0.102 sec/epoch



最后，当批量大小等于100时，我们使用小批量随机梯度下降进行优化。每个迭代轮数所需的时间比随机梯度下降和批量梯度下降所需的时间短。

```
mini1_res = train_sgd(.4, 100)
```

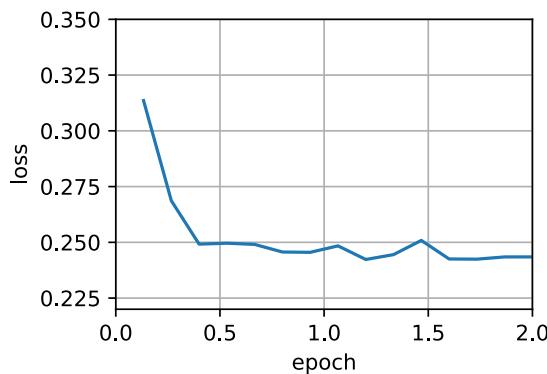
```
loss: 0.243, 0.003 sec/epoch
```



将批量大小减少到10，每个迭代轮数的时间都会增加，因为每批工作负载的执行效率变得更低。

```
mini2_res = train_sgd(.05, 10)
```

```
loss: 0.243, 0.013 sec/epoch
```

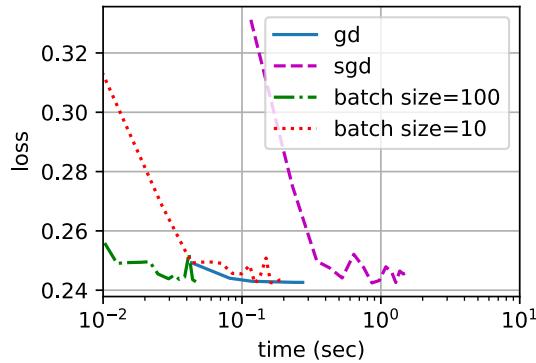


现在我们可以比较前四个实验的时间与损失。可以看出，尽管在处理的样本数方面，随机梯度下降的收敛速度快于梯度下降，但与梯度下降相比，它需要更多的时间来达到同样的损失，因为逐个样本来计算梯度并不那么有效。小批量随机梯度下降能够平衡收敛速度和计算效率。大小为10的小批量比随机梯度下降更有效；大小为100的小批量在运行时间上甚至优于梯度下降。

```

d2l.set_figsize([6, 3])
d2l.plot(*list(map(list, zip(gd_res, sgd_res, mini1_res, mini2_res))),
    'time (sec)', 'loss', xlim=[1e-2, 10],
    legend=['gd', 'sgd', 'batch size=100', 'batch size=10'])
d2l.plt.gca().set_xscale('log')

```



11.5.5 简洁实现

下面用深度学习框架自带算法实现一个通用的训练函数，我们将在本章中其它小节使用它。

```

#@save
def train_concise_ch11(trainer_fn, hyperparams, data_iter, num_epochs=4):
    # 初始化模型
    net = nn.Sequential(nn.Linear(5, 1))
    def init_weights(m):
        if type(m) == nn.Linear:
            torch.nn.init.normal_(m.weight, std=0.01)
    net.apply(init_weights)

    optimizer = trainer_fn(net.parameters(), **hyperparams)
    loss = nn.MSELoss(reduction='none')
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            optimizer.zero_grad()
            out = net(X)
            y = y.reshape(out.shape)
            l = loss(out, y)
            l.mean().backward()
            animator.add(n, l)
            n += 1
            timer.stop()
            timer.start()
    print(f'{timer.avg:.2f} sec/epoch on {d2l.device}')

```

(continues on next page)

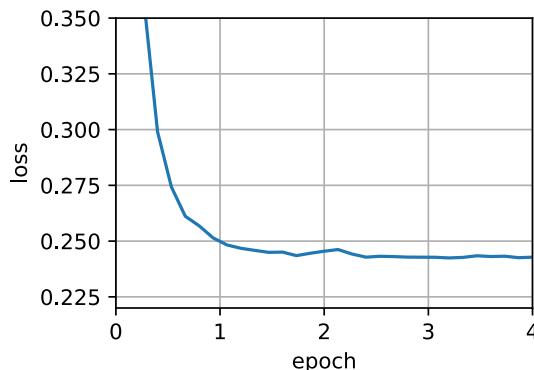
(continued from previous page)

```
optimizer.step()
n += X.shape[0]
if n % 200 == 0:
    timer.stop()
    # MSELoss计算平方误差时不带系数1/2
    animator.add(n/X.shape[0]/len(data_iter),
                  (d2l.evaluate_loss(net, data_iter, loss) / 2, ))
    timer.start()
print(f'loss: {animator.Y[0][-1]:.3f}, {timer.avg():.3f} sec/epoch')
```

下面使用这个训练函数，复现之前的实验。

```
data_iter, _ = get_data_ch11(10)
trainer = torch.optim.SGD
train_concise_ch11(trainer, {'lr': 0.01}, data_iter)
```

```
loss: 0.243, 0.015 sec/epoch
```



小结

- 由于减少了深度学习框架的额外开销，使用更好的内存定位以及CPU和GPU上的缓存，向量化使代码更加高效。
- 随机梯度下降的“统计效率”与大批量一次处理数据的“计算效率”之间存在权衡。小批量随机梯度下降提供了两全其美的答案：计算和统计效率。
- 在小批量随机梯度下降中，我们处理通过训练数据的随机排列获得的批量数据（即每个观测值只处理一次，但按随机顺序）。
- 在训练期间降低学习率有助于训练。
- 一般来说，小批量随机梯度下降比随机梯度下降和梯度下降的速度快，收敛风险较小。

练习

1. 修改批量大小和学习率，并观察目标函数值的下降率以及每个迭代轮数消耗的时间。
2. 将小批量随机梯度下降与实际从训练集中取样替换的变体进行比较。会看出什么？
3. 一个邪恶的精灵在没通知你的情况下复制了你的数据集（即每个观测发生两次，数据集增加到原始大小的两倍，但没有人告诉你）。随机梯度下降、小批量随机梯度下降和梯度下降的表现将如何变化？

Discussions¹³³

11.6 动量法

在 11.4 节一节中，我们详述了如何执行随机梯度下降，即在只有嘈杂的梯度可用的情况下执行优化时会发生什么。对于嘈杂的梯度，我们在选择学习率需要格外谨慎。如果衰减速度太快，收敛就会停滞。相反，如果太宽松，我们可能无法收敛到最优解。

11.6.1 基础

本节将探讨更有效的优化算法，尤其是针对实验中常见的某些类型的优化问题。

泄漏平均值

上一节中我们讨论了小批量随机梯度下降作为加速计算的手段。它也有很好的副作用，即平均梯度减小了方差。小批量随机梯度下降可以通过以下方式计算：

$$\mathbf{g}_{t,t-1} = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}_{t-1}) = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \mathbf{h}_{i,t-1}. \quad (11.6.1)$$

为了保持记法简单，在这里我们使用 $\mathbf{h}_{i,t-1} = \partial_{\mathbf{w}} f(\mathbf{x}_i, \mathbf{w}_{t-1})$ 作为样本 i 的随机梯度下降，使用时间 $t-1$ 时更新的权重 $t-1$ 。如果我们能够从方差减少的影响中受益，甚至超过小批量上的梯度平均值，那很不错。完成这项任务的一种选择是用泄漏平均值（leaky average）取代梯度计算：

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1} \quad (11.6.2)$$

其中 $\beta \in (0, 1)$ 。这有效地将瞬时梯度替换为多个“过去”梯度的平均值。 \mathbf{v} 被称为动量（momentum），它累加了过去的梯度。为了更详细地解释，让我们递归地将 \mathbf{v}_t 扩展到

$$\mathbf{v}_t = \beta^2 \mathbf{v}_{t-2} + \beta \mathbf{g}_{t-1,t-2} + \mathbf{g}_{t,t-1} = \dots = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau,t-\tau-1}. \quad (11.6.3)$$

其中，较大的 β 相当于长期平均值，而较小的 β 相对于梯度法只是略有修正。新的梯度替换不再指向特定实例下降最陡的方向，而是指向过去梯度的加权平均值的方向。这使我们能够实现对单批量计算平均值的大部分好处，而不产生实际计算其梯度的代价。

¹³³ <https://discuss.d2l.ai/t/4325>

上述推理构成了“加速”梯度方法的基础，例如具有动量的梯度。在优化问题条件不佳的情况下（例如，有些方向的进展比其他方向慢得多，类似狭窄的峡谷），“加速”梯度还额外享受更有效的好处。此外，它们允许我们对随后的梯度计算平均值，以获得更稳定的下降方向。诚然，即使是对于无噪声凸问题，加速度这方面也是动量如此起效的关键原因之一。

正如人们所期望的，由于其功效，动量是深度学习及其后优化中一个深入研究的主题。例如，请参阅文章¹³⁴，观看深入分析和互动动画。动量是由 (Polyak, 1964) 提出的。(Nesterov, 2018) 在凸优化的背景下进行了详细的理论讨论。长期以来，深度学习的动量一直被认为是有益的。有关实例的详细信息，请参阅 (Sutskever et al., 2013) 的讨论。

条件不佳的问题

为了更好地了解动量法的几何属性，我们复习一下梯度下降，尽管它的目标函数明显不那么令人愉快。回想我们在 11.3 节中使用了 $f(\mathbf{x}) = x_1^2 + 2x_2^2$ ，即中度扭曲的椭球目标。我们通过向 x_1 方向伸展它来进一步扭曲这个函数

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2. \quad (11.6.4)$$

与之前一样， f 在 $(0, 0)$ 有最小值，该函数在 x_1 的方向上非常平坦。让我们看看在这个新函数上执行梯度下降时会发生什么。

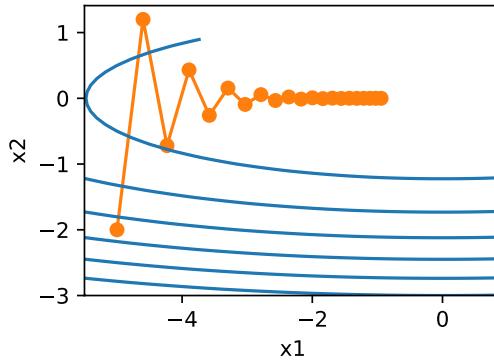
```
%matplotlib inline
import torch
from d2l import torch as d2l

eta = 0.4
def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2
def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)

d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

```
epoch 20, x1: -0.943467, x2: -0.000073
```

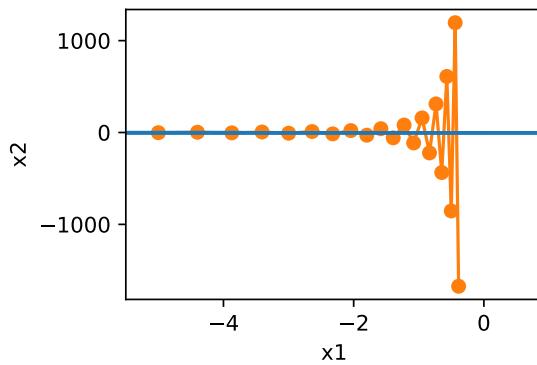
¹³⁴ <https://distill.pub/2017/momentum/> %20:cite:%60Goh.2017%60



从构造来看， x_2 方向的梯度比水平 x_1 方向的梯度大得多，变化也快得多。因此，我们陷入两难：如果选择较小的学习率，我们会确保解不会在 x_2 方向发散，但要承受在 x_1 方向的缓慢收敛。相反，如果学习率较高，我们在 x_1 方向上进展很快，但在 x_2 方向将会发散。下面的例子说明了即使学习率从0.4略微提高到0.6，也会发生变化。 x_1 方向上的收敛有所改善，但整体来看解的质量更差了。

```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

epoch 20, x1: -0.387814, x2: -1673.365109



动量法

动量法（momentum）使我们能够解决上面描述的梯度下降问题。观察上面的优化轨迹，我们可能会直觉到计算过去的平均梯度效果会很好。毕竟，在 x_1 方向上，这将聚合非常对齐的梯度，从而增加我们在每一步中覆盖的距离。相反，在梯度振荡的 x_2 方向，由于相互抵消了对方的振荡，聚合梯度将减小步长大小。使用 \mathbf{v}_t 而不是梯度 \mathbf{g}_t 可以生成以下更新等式：

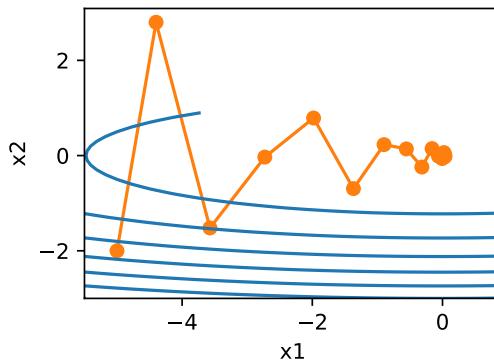
$$\begin{aligned} \mathbf{v}_t &\leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1}, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t. \end{aligned} \tag{11.6.5}$$

请注意，对于 $\beta = 0$ ，我们恢复常规的梯度下降。在深入研究它的数学属性之前，让我们快速看一下算法在实验中的表现如何。

```
def momentum_2d(x1, x2, v1, v2):
    v1 = beta * v1 + 0.2 * x1
    v2 = beta * v2 + 4 * x2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

eta, beta = 0.6, 0.5
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

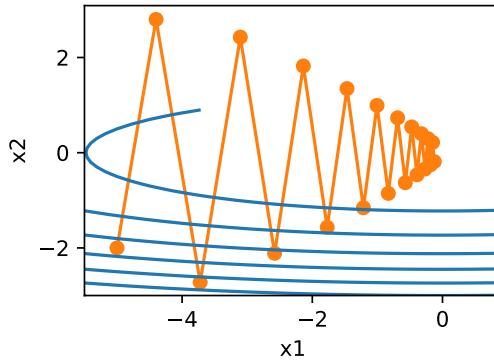
```
epoch 20, x1: 0.007188, x2: 0.002553
```



正如所见，尽管学习率与我们以前使用的相同，动量法仍然很好地收敛了。让我们看看当降低动量参数时会发生什么。将其减半至 $\beta = 0.25$ 会导致一条几乎没有收敛的轨迹。尽管如此，它比没有动量时解将会发散要好得多。

```
eta, beta = 0.6, 0.25
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

```
epoch 20, x1: -0.126340, x2: -0.186632
```

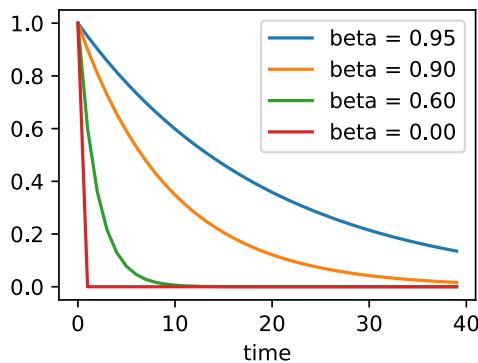


请注意，我们可以将动量法与随机梯度下降，特别是小批量随机梯度下降结合起来。唯一的变化是，在这种情况下，我们将梯度 $\mathbf{g}_{t,t-1}$ 替换为 \mathbf{g}_t 。为了方便起见，我们在时间 $t = 0$ 初始化 $\mathbf{v}_0 = 0$ 。

有效样本权重

回想一下 $\mathbf{v}_t = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau, t-\tau-1}$ 。极限条件下， $\sum_{\tau=0}^{\infty} \beta^\tau = \frac{1}{1-\beta}$ 。换句话说，不同于在梯度下降或者随机梯度下降中取步长 η ，我们选取步长 $\frac{\eta}{1-\beta}$ ，同时处理潜在表现可能会更好的下降方向。这是集两种好处于一身的做法。为了说明 β 的不同选择的权重效果如何，请参考下面的图表。

```
d2l.set_figsize()
betas = [0.95, 0.9, 0.6, 0]
for beta in betas:
    x = torch.arange(40).detach().numpy()
    d2l.plt.plot(x, beta ** x, label=f'beta = {beta:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```



11.6.2 实际实验

让我们来看看动量法在实验中是如何运作的。为此，我们需要一个更加可扩展的实现。

从零开始实现

相比于小批量随机梯度下降，动量方法需要维护一组辅助变量，即速度。它与梯度以及优化问题的变量具有相同的形状。在下面的实现中，我们称这些变量为`states`。

```
def init_momentum_states(feature_dim):
    v_w = torch.zeros((feature_dim, 1))
    v_b = torch.zeros(1)
    return (v_w, v_b)
```

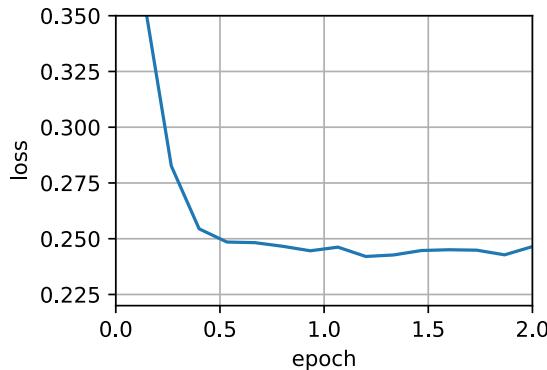
```
def sgd_momentum(params, states, hyperparams):
    for p, v in zip(params, states):
        with torch.no_grad():
            v[:] = hyperparams['momentum'] * v + p.grad
            p[:] -= hyperparams['lr'] * v
        p.grad.data.zero_()
```

让我们看看它在实验中是如何运作的。

```
def train_momentum(lr, momentum, num_epochs=2):
    d2l.train_ch11(sgd_momentum, init_momentum_states(feature_dim),
                   {'lr': lr, 'momentum': momentum}, data_iter,
                   feature_dim, num_epochs)

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
train_momentum(0.02, 0.5)
```

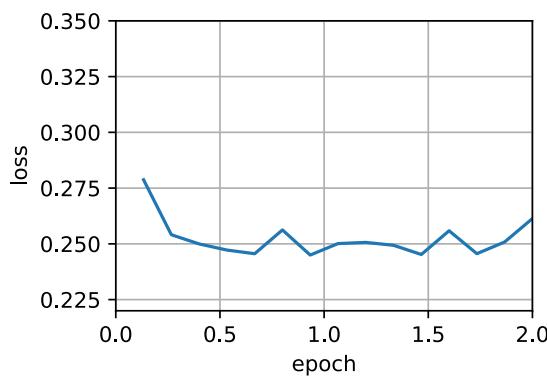
```
loss: 0.246, 0.013 sec/epoch
```



当我们将动量超参数momentum增加到0.9时，它相当于有效样本数量增加到 $\frac{1}{1-0.9} = 10$ 。我们将学习率略微降至0.01，以确保可控。

```
train_momentum(0.01, 0.9)
```

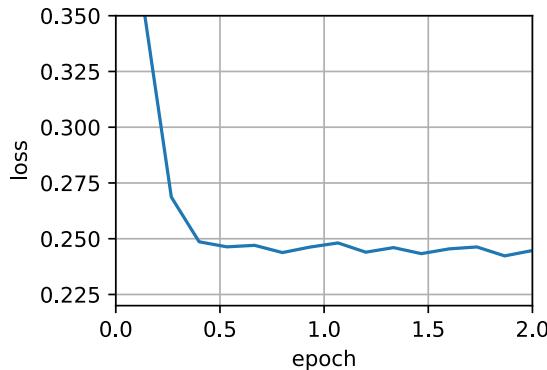
```
loss: 0.261, 0.013 sec/epoch
```



降低学习率进一步解决了任何非平滑优化问题的困难，将其设置为0.005会产生良好的收敛性能。

```
train_momentum(0.005, 0.9)
```

```
loss: 0.245, 0.013 sec/epoch
```

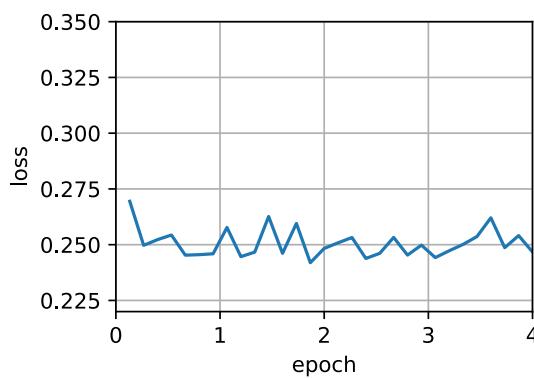


简洁实现

由于深度学习框架中的优化求解器早已构建了动量法，设置匹配参数会产生非常类似的轨迹。

```
trainer = torch.optim.SGD  
d2l.train_concise_ch11(trainer, {'lr': 0.005, 'momentum': 0.9}, data_iter)
```

```
loss: 0.247, 0.012 sec/epoch
```



11.6.3 理论分析

$f(x) = 0.1x_1^2 + 2x_2^2$ 的 2D 示例似乎相当牵强。下面我们将看到，它在实际生活中非常具有代表性，至少最小化凸二次目标函数的情况下是如此。

二次凸函数

考虑这个函数

$$h(\mathbf{x}) = \frac{1}{2} \mathbf{x}^\top \mathbf{Q} \mathbf{x} + \mathbf{x}^\top \mathbf{c} + b. \quad (11.6.6)$$

这是一个普通的二次函数。对于正定矩阵 $\mathbf{Q} \succ 0$ ，即对于具有正特征值的矩阵，有最小化器为 $\mathbf{x}^* = -\mathbf{Q}^{-1}\mathbf{c}$ ，最小值为 $b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$ 。因此我们可以将 h 重写为

$$h(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})^\top \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c}) + b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}. \quad (11.6.7)$$

梯度由 $\partial_{\mathbf{x}} f(\mathbf{x}) = \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$ 给出。也就是说，它是由 \mathbf{x} 和最小化器之间的距离乘以 \mathbf{Q} 所得出的。因此，动量法还是 $\mathbf{Q}(\mathbf{x}_t - \mathbf{Q}^{-1}\mathbf{c})$ 的线性组合。

由于 \mathbf{Q} 是正定的，因此可以通过 $\mathbf{Q} = \mathbf{O}^\top \mathbf{\Lambda} \mathbf{O}$ 分解为正交（旋转）矩阵 \mathbf{O} 和正特征值的对角矩阵 $\mathbf{\Lambda}$ 。这使我们能够将变量从 \mathbf{x} 更改为 $\mathbf{z} := \mathbf{O}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$ ，以获得一个非常简化的表达式：

$$h(\mathbf{z}) = \frac{1}{2} \mathbf{z}^\top \mathbf{\Lambda} \mathbf{z} + b'. \quad (11.6.8)$$

这里 $b' = b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$ 。由于 \mathbf{O} 只是一个正交矩阵，因此不会真正意义上扰动梯度。以 \mathbf{z} 表示的梯度下降变成

$$\mathbf{z}_t = \mathbf{z}_{t-1} - \mathbf{\Lambda} \mathbf{z}_{t-1} = (\mathbf{I} - \mathbf{\Lambda}) \mathbf{z}_{t-1}. \quad (11.6.9)$$

这个表达式中的重要事实是梯度下降在不同的特征空间之间不会混合。也就是说，如果用 \mathbf{Q} 的特征系统来表示，优化问题是以逐坐标顺序的方式进行的。这在动量法中也适用。

$$\begin{aligned} \mathbf{v}_t &= \beta \mathbf{v}_{t-1} + \mathbf{\Lambda} \mathbf{z}_{t-1} \\ \mathbf{z}_t &= \mathbf{z}_{t-1} - \eta (\beta \mathbf{v}_{t-1} + \mathbf{\Lambda} \mathbf{z}_{t-1}) \\ &= (\mathbf{I} - \eta \mathbf{\Lambda}) \mathbf{z}_{t-1} - \eta \beta \mathbf{v}_{t-1}. \end{aligned} \quad (11.6.10)$$

在这样做的过程中，我们只是证明了以下定理：带有和带有不凸二次函数动量的梯度下降，可以分解为朝二次矩阵特征向量方向坐标顺序的优化。

标量函数

鉴于上述结果，让我们看看当我们最小化函数 $f(x) = \frac{\lambda}{2}x^2$ 时会发生什么。对于梯度下降我们有

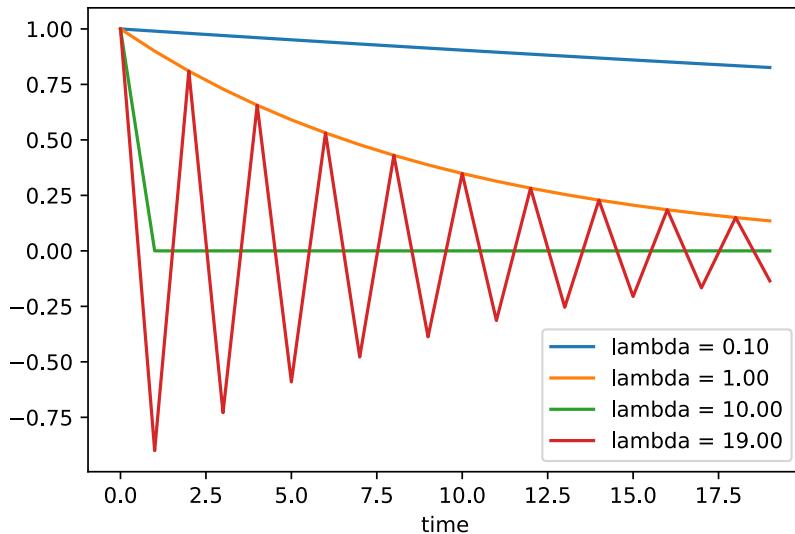
$$x_{t+1} = x_t - \eta \lambda x_t = (1 - \eta \lambda) x_t. \quad (11.6.11)$$

每 $|1 - \eta \lambda| < 1$ 时，这种优化以指数速度收敛，因为在 t 步之后我们可以得到 $x_t = (1 - \eta \lambda)^t x_0$ 。这显示了在我们将学习率 η 提高到 $\eta \lambda = 1$ 之前，收敛率最初是如何提高的。超过该数值之后，梯度开始发散，对于 $\eta \lambda > 2$ 而言，优化问题将会发散。

```

lambdas = [0.1, 1, 10, 19]
eta = 0.1
d2l.set_figsize((6, 4))
for lam in lambdas:
    t = torch.arange(20).detach().numpy()
    d2l.plt.plot(t, (1 - eta * lam) ** t, label=f'lambda = {lam:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();

```



为了分析动量的收敛情况，我们首先用两个标量重写更新方程：一个用于 x ，另一个用于动量 v 。这产生了：

$$\begin{bmatrix} v_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \beta & \lambda \\ -\eta\beta & (1-\eta\lambda) \end{bmatrix} \begin{bmatrix} v_t \\ x_t \end{bmatrix} = \mathbf{R}(\beta, \eta, \lambda) \begin{bmatrix} v_t \\ x_t \end{bmatrix}. \quad (11.6.12)$$

我们用 \mathbf{R} 来表示 2×2 矩阵的收敛表现。在 t 步之后，最初的值 $[v_0, x_0]$ 变为 $\mathbf{R}(\beta, \eta, \lambda)^t [v_0, x_0]$ 。因此，收敛速度是由 \mathbf{R} 的特征值决定的。请参阅文章¹³⁵ (Goh, 2017) 了解精彩动画。请参阅 (Flammarion and Bach, 2015) 了解详细分析。简而言之，当 $0 < \eta\lambda < 2 + 2\beta$ 时动量收敛。与梯度下降的 $0 < \eta\lambda < 2$ 相比，这是更大范围的可行参数。另外，一般而言较大值的 β 是可取的。

小结

- 动量法用过去梯度的平均值来替换梯度，这大大加快了收敛速度。
- 对于无噪声梯度下降和嘈杂随机梯度下降，动量法都是可取的。
- 动量法可以防止在随机梯度下降的优化过程停滞的问题。
- 由于对过去的数据进行了指数降权，有效梯度数为 $\frac{1}{1-\beta}$ 。

¹³⁵ <https://distill.pub/2017/momentum/>

- 在凸二次问题中，可以对动量法进行明确而详细的分析。
- 动量法的实现非常简单，但它需要我们存储额外的状态向量（动量 \mathbf{v} ）。

练习

1. 使用动量超参数和学习率的其他组合，观察和分析不同的实验结果。
2. 试试梯度下降和动量法来解决一个二次问题，其中有多个特征值，即 $f(\mathbf{x}) = \frac{1}{2} \sum_i \lambda_i x_i^2$ ，例如 $\lambda_i = 2^{-i}$ 。绘制出 x 的值在初始化 $x_i = 1$ 时如何下降。
3. 推导 $h(\mathbf{x}) = \frac{1}{2} \mathbf{x}^\top \mathbf{Q} \mathbf{x} + \mathbf{x}^\top \mathbf{c} + b$ 的最小值和最小化器。
4. 当我们执行带动量法的随机梯度下降时会有什么变化？当我们使用带动量法的小批量随机梯度下降时会发生什么？试验参数如何？

Discussions¹³⁶

11.7 AdaGrad算法

我们从有关特征学习中并不常见的问题入手。

11.7.1 稀疏特征和学习率

假设我们正在训练一个语言模型。为了获得良好的准确性，我们大多希望在训练的过程中降低学习率，速度通常为 $\mathcal{O}(t^{-\frac{1}{2}})$ 或更低。现在讨论关于稀疏特征（即只在偶尔出现的特征）的模型训练，这对自然语言来说很常见。例如，我们看到“预先条件”这个词比“学习”这个词的可能性要小得多。但是，它在计算广告学和个性化协同过滤等其他领域也很常见。

只有在这些不常见的特征出现时，与其相关的参数才会得到有意义的更新。鉴于学习率下降，我们可能最终会面临这样的情况：常见特征的参数相当迅速地收敛到最佳值，而对于不常见的特征，我们仍缺乏足够的观测以确定其最佳值。换句话说，学习率要么对于常见特征而言降低太慢，要么对于不常见特征而言降低太快。

解决此问题的一个方法是记录我们看到特定特征的次数，然后将其用作调整学习率。即我们可以使用大小为 $\eta_i = \frac{\eta_0}{\sqrt{s(i,t)+c}}$ 的学习率，而不是 $\eta = \frac{\eta_0}{\sqrt{t+c}}$ 。在这里 $s(i,t)$ 计下了我们截至 t 时观察到功能 i 的次数。这其实很容易实施且不产生额外损耗。

AdaGrad算法 (Duchi *et al.*, 2011) 通过将粗略的计数器 $s(i,t)$ 替换为先前观察所得梯度的平方之和来解决这个问题。它使用 $s(i,t+1) = s(i,t) + (\partial_i f(\mathbf{x}))^2$ 来调整学习率。这有两个好处：首先，我们不再需要决定梯度何时算足够大。其次，它会随梯度的大小自动变化。通常对应于较大梯度的坐标会显著缩小，而其他梯度较小的坐标则会得到更平滑的处理。在实际应用中，它促成了计算广告学及其相关问题中非常有效的优化程序。但是，它遮盖了AdaGrad固有的一些额外优势，这些优势在预处理环境中很容易被理解。

¹³⁶ <https://discuss.d2l.ai/t/4328>

11.7.2 预处理

凸优化问题有助于分析算法的特点。毕竟对大多数非凸问题来说，获得有意义的理论保证很难，但是直觉和洞察往往会延续。让我们来看看最小化 $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{c}^\top \mathbf{x} + b$ 这一问题。

正如在 11.6 节中那样，我们可以根据其特征分解 $\mathbf{Q} = \mathbf{U}^\top \boldsymbol{\Lambda} \mathbf{U}$ 重写这个问题，来得到一个简化得多的问题，使每个坐标都可以单独解出：

$$f(\mathbf{x}) = \bar{f}(\bar{\mathbf{x}}) = \frac{1}{2}\bar{\mathbf{x}}^\top \boldsymbol{\Lambda}\bar{\mathbf{x}} + \bar{\mathbf{c}}^\top \bar{\mathbf{x}} + b. \quad (11.7.1)$$

在这里我们使用了 $\mathbf{x} = \mathbf{U}\bar{\mathbf{x}}$ ，且因此 $\mathbf{c} = \mathbf{U}\bar{\mathbf{c}}$ 。修改后优化器为 $\bar{\mathbf{x}} = -\boldsymbol{\Lambda}^{-1}\bar{\mathbf{c}}$ 且最小值为 $-\frac{1}{2}\bar{\mathbf{c}}^\top \boldsymbol{\Lambda}^{-1}\bar{\mathbf{c}} + b$ 。这样更容易计算，因为 $\boldsymbol{\Lambda}$ 是一个包含 \mathbf{Q} 特征值的对角矩阵。

如果稍微扰动 \mathbf{c} ，我们会期望在 f 的最小化器中只产生微小的变化。遗憾的是，情况并非如此。虽然 \mathbf{c} 的微小变化导致了 $\bar{\mathbf{c}}$ 同样的微小变化，但 f 的（以及 \bar{f} 的）最小化器并非如此。每当特征值 $\boldsymbol{\Lambda}_i$ 很大时，我们只会看到 \bar{x}_i 和 \bar{f} 的最小值发声微小变化。相反，对小的 $\boldsymbol{\Lambda}_i$ 来说， \bar{x}_i 的变化可能是剧烈的。最大和最小的特征值之比称为优化问题的条件数 (condition number)。

$$\kappa = \frac{\boldsymbol{\Lambda}_1}{\boldsymbol{\Lambda}_d}. \quad (11.7.2)$$

如果条件编号 κ 很大，准确解决优化问题就会很难。我们需要确保在获取大量动态的特征值范围时足够谨慎：难道我们不能简单地通过扭曲空间来“修复”这个问题，从而使所有特征值都是1？理论上这很容易：我们只需要 \mathbf{Q} 的特征值和特征向量即可将问题从 \mathbf{x} 整理到 $\mathbf{z} := \boldsymbol{\Lambda}^{\frac{1}{2}}\mathbf{U}\mathbf{x}$ 中的一个。在新的坐标系中， $\mathbf{x}^\top \mathbf{Q}\mathbf{x}$ 可以被简化为 $\|\mathbf{z}\|^2$ 。可惜，这是一个相当不切实际的想法。一般而言，计算特征值和特征向量要比解决实际问题“贵”得多。

虽然准确计算特征值可能会很昂贵，但即便只是大致猜测并计算它们，也可能已经比不做任何事情好得多。特别是，我们可以使用 \mathbf{Q} 的对角线条目并相应地重新缩放它。这比计算特征值开销小的多。

$$\tilde{\mathbf{Q}} = \text{diag}^{-\frac{1}{2}}(\mathbf{Q})\mathbf{Q}\text{diag}^{-\frac{1}{2}}(\mathbf{Q}). \quad (11.7.3)$$

在这种情况下，我们得到了 $\tilde{\mathbf{Q}}_{ij} = \mathbf{Q}_{ij}/\sqrt{\mathbf{Q}_{ii}\mathbf{Q}_{jj}}$ ，特别注意对于所有 i ， $\tilde{\mathbf{Q}}_{ii} = 1$ 。在大多数情况下，这大大简化了条件数。例如我们之前讨论的案例，它将完全消除眼下的问题，因为问题是轴对齐的。

遗憾的是，我们还面临另一个问题：在深度学习中，我们通常情况甚至无法计算目标函数的二阶导数：对于 $\mathbf{x} \in \mathbb{R}^d$ ，即使只在小批量上，二阶导数可能也需要 $\mathcal{O}(d^2)$ 空间来计算，导致几乎不可行。AdaGrad算法巧妙的思路是，使用一个代理来表示黑塞矩阵 (Hessian) 的对角线，既相对易于计算又高效。

为了了解它是如何生效的，让我们来看看 $\bar{f}(\bar{\mathbf{x}})$ 。我们有

$$\partial_{\bar{\mathbf{x}}}\bar{f}(\bar{\mathbf{x}}) = \boldsymbol{\Lambda}\bar{\mathbf{x}} + \bar{\mathbf{c}} = \boldsymbol{\Lambda}(\bar{\mathbf{x}} - \bar{\mathbf{x}}_0), \quad (11.7.4)$$

其中 $\bar{\mathbf{x}}_0$ 是 \bar{f} 的优化器。因此，梯度的大小取决于 $\boldsymbol{\Lambda}$ 和与最佳值的差值。如果 $\bar{\mathbf{x}} - \bar{\mathbf{x}}_0$ 没有改变，那这就是我们所求的。毕竟在这种情况下，梯度 $\partial_{\bar{\mathbf{x}}}\bar{f}(\bar{\mathbf{x}})$ 的大小就足够了。由于AdaGrad算法是一种随机梯度下降算法，所以即使是在最佳值中，我们也会看到具有非零方差的梯度。因此，我们可以放心地使用梯度的方差作为黑塞矩阵比例的廉价替代。详尽的分析（要花几页解释）超出了本节的范围，请读者参考 (Duchi et al., 2011)。

11.7.3 算法

让我们接着上面正式开始讨论。我们使用变量 \mathbf{s}_t 来累加过去的梯度方差，如下所示：

$$\begin{aligned}\mathbf{g}_t &= \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})), \\ \mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.\end{aligned}\tag{11.7.5}$$

在这里，操作是按照坐标顺序应用。也就是说， \mathbf{v}^2 有条目 v_i^2 。同样， $\frac{1}{\sqrt{v}}$ 有条目 $\frac{1}{\sqrt{v_i}}$ ，并且 $\mathbf{u} \cdot \mathbf{v}$ 有条目 $u_i v_i$ 。与之前一样， η 是学习率， ϵ 是一个为维持数值稳定性而添加的常数，用来确保我们不会除以0。最后，我们初始化 $\mathbf{s}_0 = \mathbf{0}$ 。

就像在动量法中我们需要跟踪一个辅助变量一样，在AdaGrad算法中，我们允许每个坐标有单独的学习率。与SGD算法相比，这并没有明显增加AdaGrad的计算代价，因为主要计算用在 $l(y_t, f(\mathbf{x}_t, \mathbf{w}))$ 及其导数。

请注意，在 \mathbf{s}_t 中累加平方梯度意味着 \mathbf{s}_t 基本上以线性速率增长（由于梯度从最初开始衰减，实际上比线性慢一些）。这产生了一个学习率 $\mathcal{O}(t^{-\frac{1}{2}})$ ，但是在单个坐标的层面上进行了调整。对于凸问题，这完全足够了。然而，在深度学习中，我们可能希望更慢地降低学习率。这引出了许多AdaGrad算法的变体，我们将在后续章节中讨论它们。眼下让我们先看看它在二次凸问题中的表现如何。我们仍然以同一函数为例：

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2.\tag{11.7.6}$$

我们将使用与之前相同的学习率来实现AdaGrad算法，即 $\eta = 0.4$ 。可以看到，自变量的迭代轨迹较平滑。但由于 s_t 的累加效果使学习率不断衰减，自变量在迭代后期的移动幅度较小。

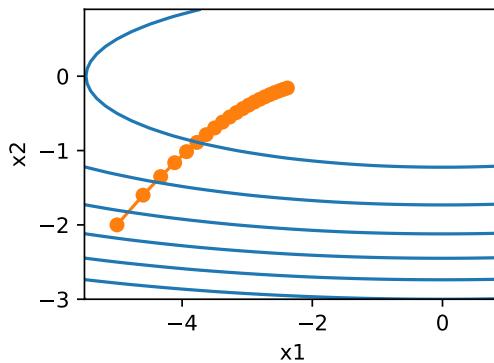
```
%matplotlib inline
import math
import torch
from d2l import torch as d2l
```

```
def adagrad_2d(x1, x2, s1, s2):
    eps = 1e-6
    g1, g2 = 0.2 * x1, 4 * x2
    s1 += g1 ** 2
    s2 += g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta = 0.4
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

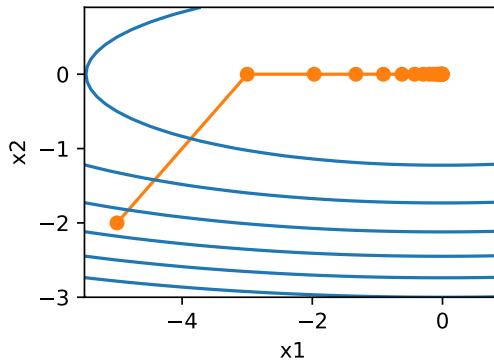
```
epoch 20, x1: -2.382563, x2: -0.158591
```



我们将学习率提高到2，可以看到更好的表现。这已经表明，即使在无噪声的情况下，学习率的降低可能相当剧烈，我们需要确保参数能够适当地收敛。

```
eta = 2  
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

```
epoch 20, x1: -0.002295, x2: -0.000000
```



11.7.4 从零开始实现

同动量法一样，AdaGrad算法需要对每个自变量维护同它一样形状的状态变量。

```
def init_adagrad_states(feature_dim):  
    s_w = torch.zeros((feature_dim, 1))  
    s_b = torch.zeros(1)  
    return (s_w, s_b)
```

(continues on next page)

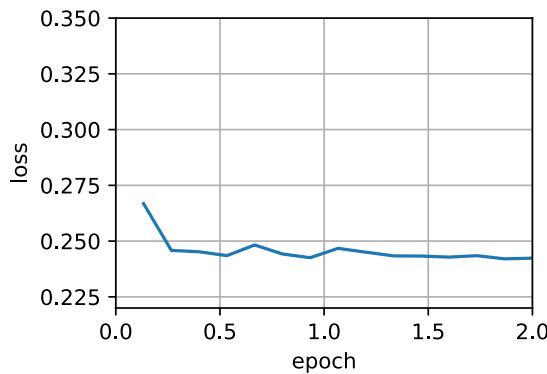
(continued from previous page)

```
def adagrad(params, states, hyperparams):
    eps = 1e-6
    for p, s in zip(params, states):
        with torch.no_grad():
            s[:] += torch.square(p.grad)
            p[:] -= hyperparams['lr'] * p.grad / torch.sqrt(s + eps)
            p.grad.data.zero_()
```

与 11.5 节一节中的实验相比，这里使用更大的学习率来训练模型。

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adagrad, init_adagrad_states(feature_dim),
{'lr': 0.1}, data_iter, feature_dim);
```

```
loss: 0.242, 0.012 sec/epoch
```

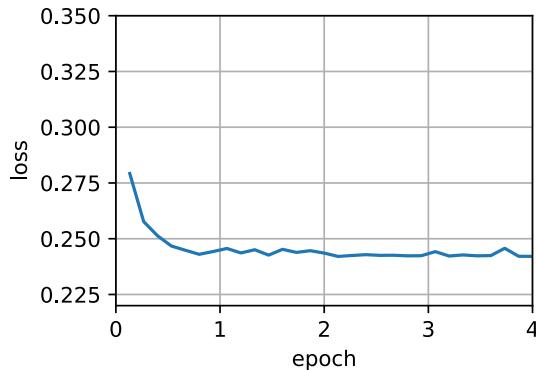


11.7.5 简洁实现

我们可直接使用深度学习框架中提供的AdaGrad算法来训练模型。

```
trainer = torch.optim.Adagrad
d2l.train_concise_ch11(trainer, {'lr': 0.1}, data_iter)
```

```
loss: 0.242, 0.013 sec/epoch
```



小结

- AdaGrad算法会在单个坐标层面动态降低学习率。
- AdaGrad算法利用梯度的大小作为调整进度速率的手段：用较小的学习率来补偿带有较大梯度的坐标。
- 在深度学习问题中，由于内存和计算限制，计算准确的二阶导数通常是不可行的。梯度可以作为一个有效的代理。
- 如果优化问题的结构相当不均匀，AdaGrad算法可以帮助缓解扭曲。
- AdaGrad算法对于稀疏特征特别有效，在此情况下由于不常出现的问题，学习率需要更慢地降低。
- 在深度学习问题上，AdaGrad算法有时在降低学习率方面可能过于剧烈。我们将在 11.10 节一节讨论缓解这种情况的策略。

练习

1. 证明对于正交矩阵 \mathbf{U} 和向量 \mathbf{c} ，以下等式成立： $\|\mathbf{c} - \delta\|_2 = \|\mathbf{U}\mathbf{c} - \mathbf{U}\delta\|_2$ 。为什么这意味着在变量的正交变化之后，扰动的程度不会改变？
2. 尝试对函数 $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ 、以及它旋转45度后的函数即 $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$ 使用AdaGrad算法。它的表现会不同吗？
3. 证明格什戈林圆盘定理¹³⁷，其中提到，矩阵 \mathbf{M} 的特征值 λ_i 在至少一个 j 的选项中满足 $|\lambda_i - \mathbf{M}_{jj}| \leq \sum_{k \neq j} |\mathbf{M}_{jk}|$ 的要求。
4. 关于对角线预处理矩阵 $\text{diag}^{-\frac{1}{2}}(\mathbf{M})\mathbf{M}\text{diag}^{-\frac{1}{2}}(\mathbf{M})$ 的特征值，格什戈林的定理告诉了我们什么？
5. 尝试对适当的深度网络使用AdaGrad算法，例如，6.6节中应用于Fashion-MNIST的深度网络。
6. 要如何修改AdaGrad算法，才能使其在学习率方面的衰减不那么激进？

Discussions¹³⁸

¹³⁷ https://en.wikipedia.org/wiki/Gershgorin_circle_theorem

¹³⁸ <https://discuss.d2l.ai/t/4319>

11.8 RMSProp算法

11.7节中的关键问题之一，是学习率按预定时间表 $\mathcal{O}(t^{-\frac{1}{2}})$ 显著降低。虽然这通常适用于凸问题，但对于深度学习中遇到的非凸问题，可能并不理想。但是，作为一个预处理器，Adagrad算法按坐标顺序的适应性是非常可取的。

(Tieleman and Hinton, 2012)建议以RMSProp算法作为将速率调度与坐标自适应学习率分离的简单修复方法。问题在于，Adagrad算法将梯度 \mathbf{g}_t 的平方累加成状态矢量 $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$ 。因此，由于缺乏规范化，没有约束力， \mathbf{s}_t 持续增长，几乎上是在算法收敛时呈线性递增。

解决此问题的一种方法是使用 \mathbf{s}_t/t 。对 \mathbf{g}_t 的合理分布来说，它将收敛。遗憾的是，限制行为生效可能需要很长时间，因为该流程记住了值的完整轨迹。另一种方法是按动量法中的方式使用泄漏平均值，即 $\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t^2$ ，其中参数 $\gamma > 0$ 。保持所有其它部分不变就产生了RMSProp算法。

11.8.1 算法

让我们详细写出这些方程式。

$$\begin{aligned}\mathbf{s}_t &\leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t^2, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t.\end{aligned}\tag{11.8.1}$$

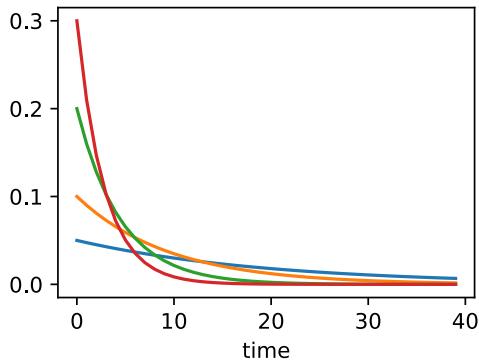
常数 $\epsilon > 0$ 通常设置为 10^{-6} ，以确保我们不会因除以零或步长过大而受到影响。鉴于这种扩展，我们现在可以自由控制学习率 η ，而不考虑基于每个坐标应用的缩放。就泄漏平均值而言，我们可以采用与之前在动量法中适用的相同推理。扩展 \mathbf{s}_t 定义可获得

$$\begin{aligned}\mathbf{s}_t &= (1 - \gamma) \mathbf{g}_t^2 + \gamma \mathbf{s}_{t-1} \\ &= (1 - \gamma) (\mathbf{g}_t^2 + \gamma \mathbf{g}_{t-1}^2 + \gamma^2 \mathbf{g}_{t-2}^2 + \dots).\end{aligned}\tag{11.8.2}$$

同之前在 11.6 节小节一样，我们使用 $1 + \gamma + \gamma^2 + \dots = \frac{1}{1-\gamma}$ 。因此，权重总和标准化为1且观测值的半衰期为 γ^{-1} 。让我们图像化各种数值的 γ 在过去40个时间步长的权重。

```
import math
import torch
from d2l import torch as d2l

d2l.set_figsize()
gammas = [0.95, 0.9, 0.8, 0.7]
for gamma in gammas:
    x = torch.arange(40).detach().numpy()
    d2l.plt.plot(x, (1-gamma) * gamma ** x, label=f'gamma = {gamma:.2f}')
d2l.plt.xlabel('time');
```



11.8.2 从零开始实现

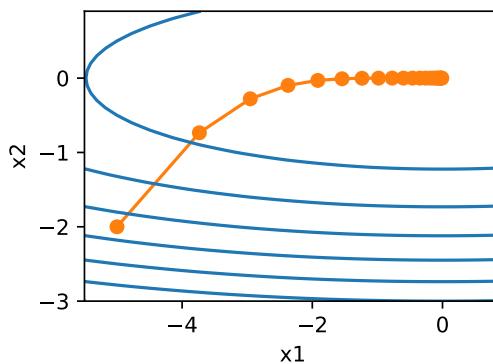
和之前一样，我们使用二次函数 $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ 来观察RMSProp算法的轨迹。回想在 11.7 节一节中，当我们使用学习率为0.4的Adagrad算法时，变量在算法的后期阶段移动非常缓慢，因为学习率衰减太快。RMSProp算法中不会发生这种情况，因为 η 是单独控制的。

```
def rmsprop_2d(x1, x2, s1, s2):
    g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
    s1 = gamma * s1 + (1 - gamma) * g1 ** 2
    s2 = gamma * s2 + (1 - gamma) * g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta, gamma = 0.4, 0.9
d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))
```

epoch 20, x1: -0.010599, x2: 0.000000



接下来，我们在深度网络中实现RMSProp算法。

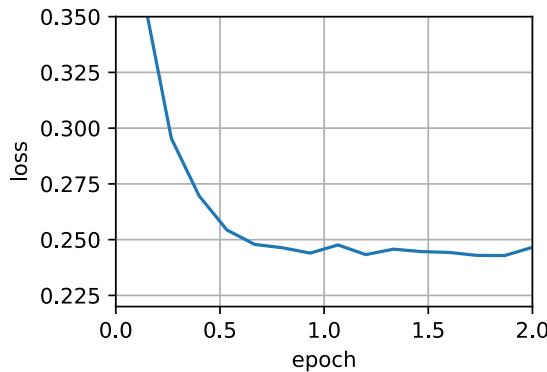
```
def init_rmsprop_states(feature_dim):
    s_w = torch.zeros((feature_dim, 1))
    s_b = torch.zeros(1)
    return (s_w, s_b)
```

```
def rmsprop(params, states, hyperparams):
    gamma, eps = hyperparams['gamma'], 1e-6
    for p, s in zip(params, states):
        with torch.no_grad():
            s[:] = gamma * s + (1 - gamma) * torch.square(p.grad)
            p[:] -= hyperparams['lr'] * p.grad / torch.sqrt(s + eps)
    p.grad.data.zero_()
```

我们将初始学习率设置为0.01，加权项 γ 设置为0.9。也就是说， \mathbf{s} 累加了过去的 $1/(1 - \gamma) = 10$ 次平方梯度观测值的平均值。

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(rmsprop, init_rmsprop_states(feature_dim),
{'lr': 0.01, 'gamma': 0.9}, data_iter, feature_dim);
```

```
loss: 0.247, 0.014 sec/epoch
```

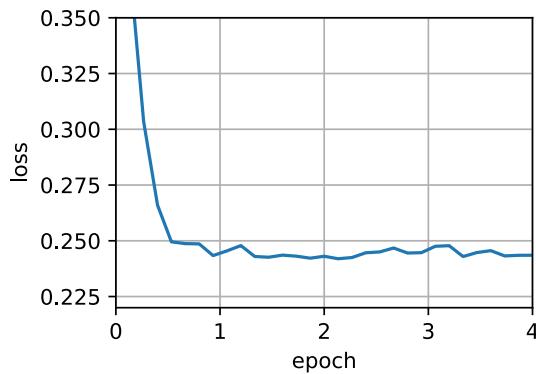


11.8.3 简洁实现

我们可直接使用深度学习框架中提供的RMSProp算法来训练模型。

```
trainer = torch.optim.RMSprop  
d2l.train_concise_ch11(trainer, {'lr': 0.01, 'alpha': 0.9},  
                         data_iter)
```

```
loss: 0.244, 0.017 sec/epoch
```



小结

- RMSProp算法与Adagrad算法非常相似，因为两者都使用梯度的平方来缩放系数。
- RMSProp算法与动量法都使用泄漏平均值。但是，RMSProp算法使用该技术来调整按系数顺序的预处理器。
- 在实验中，学习率需要由实验者调度。
- 系数 γ 决定了在调整每坐标比例时历史记录的时长。

练习

1. 如果我们设置 $\gamma = 1$ ，实验会发生什么？为什么？
2. 旋转优化问题以最小化 $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$ 。收敛会是什么？
3. 试试在真正的机器学习问题上应用RMSProp算法会发生什么，例如在Fashion-MNIST上的训练。试验不同的取值来调整学习率。
4. 随着优化的进展，需要调整 γ 吗？RMSProp算法对此有多敏感？

Discussions¹³⁹

¹³⁹ <https://discuss.d2l.ai/t/4322>

11.9 Adadelta

Adadelta是AdaGrad的另一种变体（11.7节），主要区别在于前者减少了学习率适应坐标的数量。此外，广义上Adadelta被称为没有学习率，因为它使用变化量本身作为未来变化的校准。Adadelta算法是在（Zeiler, 2012）中提出的。

11.9.1 Adadelta算法

简而言之，Adadelta使用两个状态变量， \mathbf{s}_t 用于存储梯度二阶导数的泄露平均值， $\Delta\mathbf{x}_t$ 用于存储模型本身中参数变化二阶导数的泄露平均值。请注意，为了与其他出版物和实现的兼容性，我们使用作者的原始符号和命名（没有其它真正理由让大家使用不同的希腊变量来表示在动量法、AdaGrad、RMSProp和Adadelta中用于相同用途的参数）。

以下是Adadelta的技术细节。鉴于参数du jour是 ρ ，我们获得了与11.8节类似的以下泄漏更新：

$$\mathbf{s}_t = \rho\mathbf{s}_{t-1} + (1 - \rho)\mathbf{g}_t^2. \quad (11.9.1)$$

与11.8节的区别在于，我们使用重新缩放的梯度 \mathbf{g}'_t 执行更新，即

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \mathbf{g}'_t. \quad (11.9.2)$$

那么，调整后的梯度 \mathbf{g}'_t 是什么？我们可以按如下方式计算它：

$$\mathbf{g}'_t = \frac{\sqrt{\Delta\mathbf{x}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t, \quad (11.9.3)$$

其中 $\Delta\mathbf{x}_{t-1}$ 是重新缩放梯度的平方 \mathbf{g}'_t 的泄露平均值。我们将 $\Delta\mathbf{x}_0$ 初始化为0，然后在每个步骤中使用 \mathbf{g}'_t 更新它，即

$$\Delta\mathbf{x}_t = \rho\Delta\mathbf{x}_{t-1} + (1 - \rho)\mathbf{g}'_t^2, \quad (11.9.4)$$

和 ϵ （例如 10^{-5} 这样的小值）是为了保持数字稳定性而加入的。

11.9.2 代码实现

Adadelta需要为每个变量维护两个状态变量，即 \mathbf{s}_t 和 $\Delta\mathbf{x}_t$ 。这将产生以下实现。

```
%matplotlib inline
import torch
from d2l import torch as d2l

def init_adadelta_states(feature_dim):
    s_w, s_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
```

(continues on next page)

(continued from previous page)

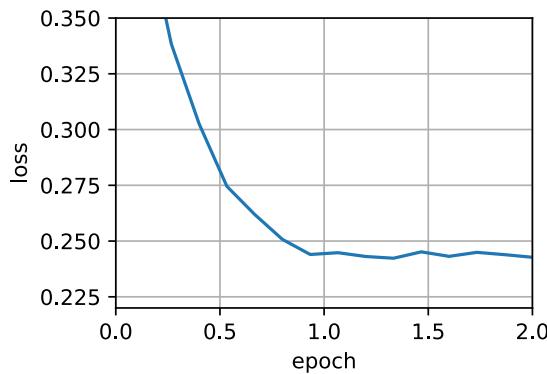
```
delta_w, delta_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
return ((s_w, delta_w), (s_b, delta_b))

def adadelta(params, states, hyperparams):
    rho, eps = hyperparams['rho'], 1e-5
    for p, (s, delta) in zip(params, states):
        with torch.no_grad():
            # In-place updates via [:]
            s[:] = rho * s + (1 - rho) * torch.square(p.grad)
            g = (torch.sqrt(delta + eps) / torch.sqrt(s + eps)) * p.grad
            p[:] -= g
            delta[:] = rho * delta + (1 - rho) * g * g
            p.grad.data.zero_()
```

对于每次参数更新，选择 $\rho = 0.9$ 相当于10个半衰期。由此我们得到：

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adadelta, init_adadelta_states(feature_dim),
    {'rho': 0.9}, data_iter, feature_dim);
```

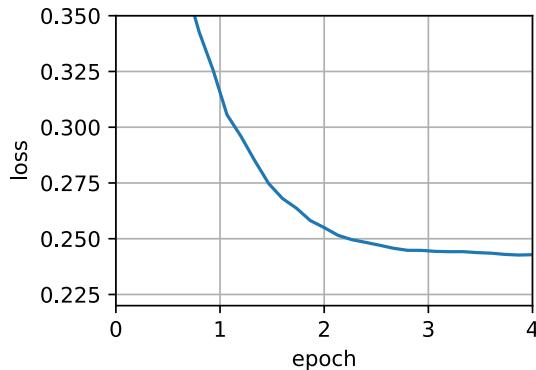
loss: 0.243, 0.014 sec/epoch



为了简洁实现，我们只需使用高级API中的Adadelta算法。

```
trainer = torch.optim.Adadelta
d2l.train_concise_ch11(trainer, {'rho': 0.9}, data_iter)
```

loss: 0.243, 0.013 sec/epoch



小结

- Adadelta没有学习率参数。相反，它使用参数本身的变化率来调整学习率。
- Adadelta需要两个状态变量来存储梯度的二阶导数和参数的变化。
- Adadelta使用泄漏的平均值来保持对适当统计数据的运行估计。

练习

1. 调整 ρ 的值，会发生什么？
2. 展示如何在不使用 \mathbf{g}'_t 的情况下实现算法。为什么这是个好主意？
3. Adadelta真的是学习率为0吗？能找到Adadelta无法解决的优化问题吗？
4. 将Adadelta的收敛行为与AdaGrad和RMSProp进行比较。

Discussions¹⁴⁰

11.10 Adam算法

本章我们已经学习了许多有效优化的技术。在本节讨论之前，我们先详细回顾一下这些技术：

- 在 11.4 节中，我们学习了：随机梯度下降在解决优化问题时比梯度下降更有效。
- 在 11.5 节中，我们学习了：在一个小批量中使用更大的观测值集，可以通过向量化提供额外效率。这是高效的多机、多GPU和整体并行处理的关键。
- 在 11.6 节中我们添加了一种机制，用于汇总过去梯度的历史以加速收敛。
- 在 11.7 节中，我们通过对每个坐标缩放来实现高效计算的预处理器。
- 在 11.8 节中，我们通过学习率的调整来分离每个坐标的缩放。

¹⁴⁰ <https://discuss.d2l.ai/t/5772>

Adam算法 (Kingma and Ba, 2014) 将所有这些技术汇总到一个高效的学习算法中。不出预料，作为深度学习中使用的更强大和有效的优化算法之一，它非常受欢迎。但是它并非没有问题，尤其是 (Reddi *et al.*, 2019) 表明，有时 Adam 算法可能由于方差控制不良而发散。在完善工作中，(Zaheer *et al.*, 2018) 给 Adam 算法提供了一个称为 Yogi 的热补丁来解决这些问题。下面我们了解一下 Adam 算法。

11.10.1 算法

Adam 算法的关键组成部分之一是：它使用指数加权移动平均值来估算梯度的动量和二次矩，即它使用状态变量

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1) \mathbf{g}_t, \\ \mathbf{s}_t &\leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2.\end{aligned}\tag{11.10.1}$$

这里 β_1 和 β_2 是非负加权参数。常将它们设置为 $\beta_1 = 0.9$ 和 $\beta_2 = 0.999$ 。也就是说，方差估计的移动远远慢于动量估计的移动。注意，如果我们初始化 $\mathbf{v}_0 = \mathbf{s}_0 = 0$ ，就会获得一个相当大的初始偏差。我们可以通过使用 $\sum_{i=0}^t \beta^i = \frac{1-\beta^t}{1-\beta}$ 来解决这个问题。相应地，标准化状态变量由下式获得

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}.\tag{11.10.2}$$

有了正确的估计，我们现在可以写出更新方程。首先，我们以非常类似于 RMSProp 算法的方式重新缩放梯度以获得

$$\mathbf{g}'_t = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t + \epsilon}}.\tag{11.10.3}$$

与 RMSProp 不同，我们的更新使用动量 $\hat{\mathbf{v}}_t$ 而不是梯度本身。此外，由于使用 $\frac{1}{\sqrt{\hat{\mathbf{s}}_t + \epsilon}}$ 而不是 $\frac{1}{\sqrt{\mathbf{s}_t + \epsilon}}$ 进行缩放，两者会略有差异。前者在实践中效果略好一些，因此与 RMSProp 算法有所区分。通常，我们选择 $\epsilon = 10^{-6}$ ，这是为了在数值稳定性和逼真度之间取得良好的平衡。

最后，我们简单更新：

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}'_t.\tag{11.10.4}$$

回顾 Adam 算法，它的设计灵感很清楚：首先，动量和规模在状态变量中清晰可见，它们相当独特的定义使我们移除偏项（这可以通过稍微不同的初始化和更新条件来修正）。其次，RMSProp 算法中两项的组合都非常简单。最后，明确的学习率 η 使我们能够控制步长来解决收敛问题。

11.10.2 实现

从头开始实现 Adam 算法并不难。为方便起见，我们将时间步 t 存储在 `hyperparams` 字典中。除此之外，一切都很简单。

```
%matplotlib inline
import torch
```

(continues on next page)

(continued from previous page)

```
from d2l import torch as d2l

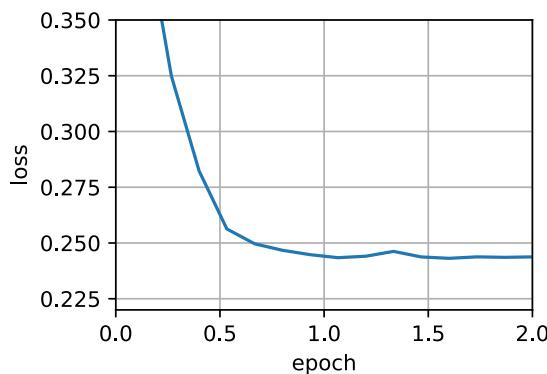
def init_adam_states(feature_dim):
    v_w, v_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    s_w, s_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    return ((v_w, s_w), (v_b, s_b))

def adam(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s) in zip(params, states):
        with torch.no_grad():
            v[:] = beta1 * v + (1 - beta1) * p.grad
            s[:] = beta2 * s + (1 - beta2) * torch.square(p.grad)
            v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
            s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
            p[:] -= hyperparams['lr'] * v_bias_corr / (torch.sqrt(s_bias_corr)
                                                       + eps)
        p.grad.data.zero_()
    hyperparams['t'] += 1
```

现在，我们用以上Adam算法来训练模型，这里我们使用 $\eta = 0.01$ 的学习率。

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adam, init_adam_states(feature_dim),
{'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

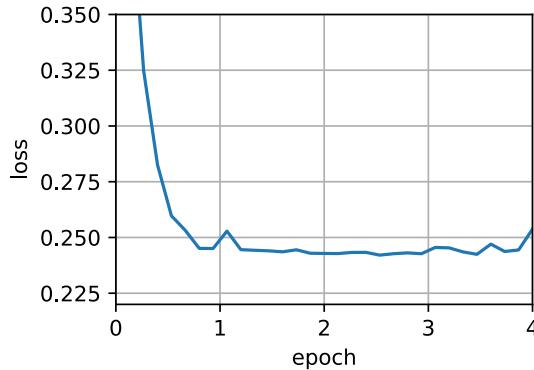
loss: 0.244, 0.015 sec/epoch



此外，我们可以用深度学习框架自带算法应用Adam算法，这里我们只需要传递配置参数。

```
trainer = torch.optim.Adam
d2l.train_concise_ch11(trainer, {'lr': 0.01}, data_iter)
```

loss: 0.254, 0.015 sec/epoch



11.10.3 Yogi

Adam算法也存在一些问题：即使在凸环境下，当 \mathbf{s}_t 的二次矩估计值爆炸时，它可能无法收敛。(Zaheer et al., 2018)为 \mathbf{s}_t 提出了的改进更新和参数初始化。论文中建议我们重写Adam算法更新如下：

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2) (\mathbf{g}_t^2 - \mathbf{s}_{t-1}). \quad (11.10.5)$$

每当 \mathbf{g}_t^2 具有值很大的变量或更新很稀疏时， \mathbf{s}_t 可能会太快地“忘记”过去的值。一个有效的解决方法是将 $\mathbf{g}_t^2 - \mathbf{s}_{t-1}$ 替换为 $\mathbf{g}_t^2 \odot \text{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1})$ 。这就是Yogi更新，现在更新的规模不再取决于偏差的量。

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2 \odot \text{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1}). \quad (11.10.6)$$

论文中，作者还进一步建议用更大的初始批量来初始化动量，而不仅仅是初始的逐点估计。

```
def yogi(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-3
    for p, (v, s) in zip(params, states):
        with torch.no_grad():
            v[:] = beta1 * v + (1 - beta1) * p.grad
            s[:] = s + (1 - beta2) * torch.sign(
                torch.square(p.grad) - s) * torch.square(p.grad)
            v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
            s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
            p[:] -= hyperparams['lr'] * v_bias_corr / (torch.sqrt(s_bias_corr)
                                                       + eps)
    p.grad.data.zero_()
```

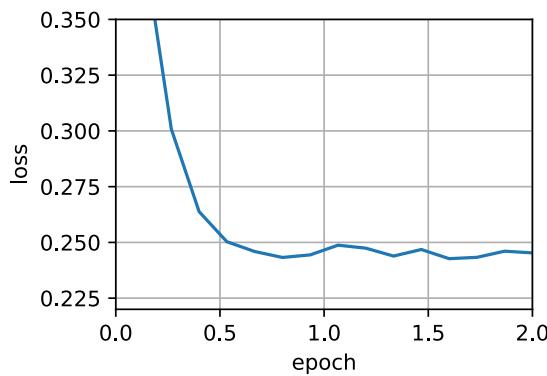
(continues on next page)

(continued from previous page)

```
hyperparams['t'] += 1

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(yogi, init_adam_states(feature_dim),
{'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

```
loss: 0.245, 0.015 sec/epoch
```



小结

- Adam算法将许多优化算法的功能结合到了相当强大的更新规则中。
- Adam算法在RMSPProp算法基础上创建的，还在小批量的随机梯度上使用EWMA。
- 在估计动量和二次矩时，Adam算法使用偏差校正来调整缓慢的启动速度。
- 对于具有显著差异的梯度，我们可能会遇到收敛性问题。我们可以通过使用更大的小批量或者切换到改进的估计值 \mathbf{s}_t 来修正它们。Yogi提供了这样的替代方案。

练习

1. 调节学习率，观察并分析实验结果。
2. 试着重写动量和二次矩更新，从而使其不需要偏差校正。
3. 收敛时为什么需要降低学习率 η ？
4. 尝试构造一个使用Adam算法会发散而Yogi会收敛的例子。

Discussions¹⁴¹

¹⁴¹ <https://discuss.d2l.ai/t/4331>

11.11 学习率调度器

到目前为止，我们主要关注如何更新权重向量的优化算法，而不是它们的更新速率。然而，调整学习率通常与实际算法同样重要，有如下几方面需要考虑：

- 首先，学习率的大小很重要。如果它太大，优化就会发散；如果它太小，训练就会需要过长时间，或者我们最终只能得到次优的结果。我们之前看到问题的条件数很重要（有关详细信息，请参见 11.6 节）。直观地说，这是最不敏感与最敏感方向的变化量的比率。
- 其次，衰减速率同样很重要。如果学习率持续过高，我们可能最终会在最小值附近弹跳，从而无法达到最优解。11.5 节比较详细地讨论了这一点，在 11.4 节中我们则分析了性能保证。简而言之，我们希望速率衰减，但要比 $\mathcal{O}(t^{-\frac{1}{2}})$ 慢，这样能成为解决凸问题的不错选择。
- 另一个同样重要的方面是初始化。这既涉及参数最初的设计方式（详情请参阅 4.8 节），又关系到它们最初的演变方式。这被戏称为预热（warmup），即我们最初开始向着解决方案迈进的速度有多快。一开始的大步可能没有好处，特别是因为最初的参数集是随机的。最初的更新方向可能也是毫无意义的。
- 最后，还有许多优化变体可以执行周期性学习率调整。这超出了本章的范围，我们建议读者阅读 (Izmailov et al., 2018) 来了解个中细节。例如，如何通过对整个路径参数求平均值来获得更好的解。

鉴于管理学习率需要很多细节，因此大多数深度学习框架都有自动应对这个问题的工具。在本章中，我们将梳理不同的调度策略对准确性的影响，并展示如何通过学习率调度器（learning rate scheduler）来有效管理。

11.11.1 一个简单的问题

我们从一个简单的问题开始，这个问题可以轻松计算，但足以说明要义。为此，我们选择了一个稍微现代化的LeNet版本（激活函数使用relu而不是sigmoid，汇聚层使用最大汇聚层而不是平均汇聚层），并应用于Fashion-MNIST数据集。此外，我们混合网络以提高性能。由于大多数代码都是标准的，我们只介绍基础知识，而不做进一步的详细讨论。如果需要，请参阅 6 节进行复习。

```
%matplotlib inline
import math
import torch
from torch import nn
from torch.optim import lr_scheduler
from d2l import torch as d2l

def net_fn():
    model = nn.Sequential(
        nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(6, 16, kernel_size=5), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
```

(continues on next page)

(continued from previous page)

```
nn.Flatten(),
nn.Linear(16 * 5 * 5, 120), nn.ReLU(),
nn.Linear(120, 84), nn.ReLU(),
nn.Linear(84, 10))

return model

loss = nn.CrossEntropyLoss()
device = d2l.try_gpu()

batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

# 代码几乎与d2l.train_ch6定义在卷积神经网络一章LeNet一节中的相同
def train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
          scheduler=None):
    net.to(device)
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                             legend=['train loss', 'train acc', 'test acc'])

    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3) # train_loss, train_acc, num_examples
        for i, (X, y) in enumerate(train_iter):
            net.train()
            trainer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            trainer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
        train_loss = metric[0] / metric[2]
        train_acc = metric[1] / metric[2]
        if (i + 1) % 50 == 0:
            animator.add(epoch + i / len(train_iter),
                         (train_loss, train_acc, None))

        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch+1, (None, None, test_acc))

    if scheduler:
        if scheduler.__module__ == lr_scheduler.__name__:
```

(continues on next page)

(continued from previous page)

```
# UsingPyTorchIn-BuiltScheduler
scheduler.step()

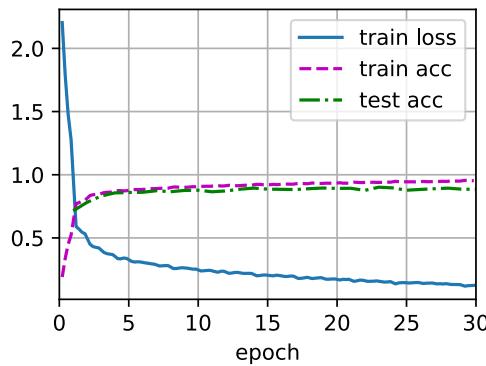
else:
    # Usingcustomdefinedscheduler
    for param_group in trainer.param_groups:
        param_group['lr'] = scheduler(epoch)

print(f'train loss {train_loss:.3f}, train acc {train_acc:.3f}, '
      f'test acc {test_acc:.3f}')
```

让我们来看看如果使用默认设置，调用此算法会发生什么。例如设学习率为0.3并训练30次迭代。留意在超过了某点、测试准确度方面的进展停滞时，训练准确度将如何继续提高。两条曲线之间的间隙表示过拟合。

```
lr, num_epochs = 0.3, 30
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=lr)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device)
```

```
train loss 0.128, train acc 0.951, test acc 0.885
```



11.11.2 学习率调度器

我们可以在每个迭代轮数（甚至在每个小批量）之后向下调整学习率。例如，以动态的方式来响应优化的进展情况。

```
lr = 0.1
trainer.param_groups[0]["lr"] = lr
print(f'learning rate is now {trainer.param_groups[0]["lr"]:.2f}')
```

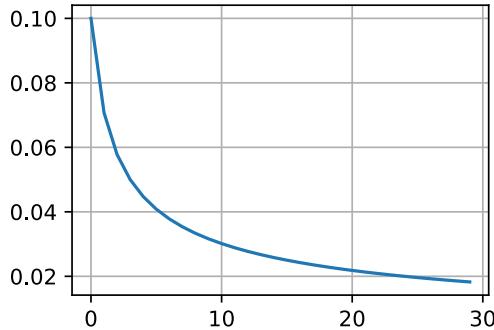
```
learning rate is now 0.10
```

更通常而言，我们应该定义一个调度器。当调用更新次数时，它将返回学习率的适当值。让我们定义一个简单的方法，将学习率设置为 $\eta = \eta_0(t + 1)^{-\frac{1}{2}}$ 。

```
class SquareRootScheduler:  
    def __init__(self, lr=0.1):  
        self.lr = lr  
  
    def __call__(self, num_update):  
        return self.lr * pow(num_update + 1.0, -0.5)
```

让我们在一系列值上绘制它的行为。

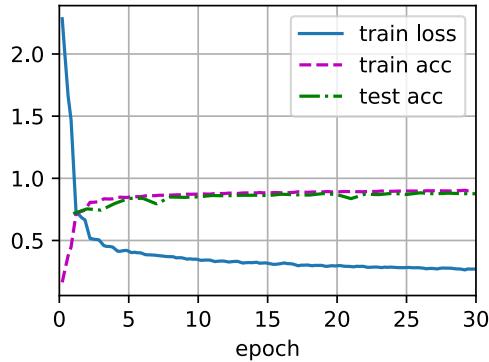
```
scheduler = SquareRootScheduler(lr=0.1)  
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



现在让我们来看看这对在Fashion-MNIST数据集上的训练有何影响。我们只是提供调度器作为训练算法的额外参数。

```
net = net_fn()  
trainer = torch.optim.SGD(net.parameters(), lr)  
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,  
      scheduler)
```

```
train loss 0.270, train acc 0.901, test acc 0.876
```



这比以前好一些：曲线比以前更加平滑，并且过拟合更小了。遗憾的是，关于为什么在理论上某些策略会导致较轻的过拟合，有一些观点认为，较小的步长将导致参数更接近零，因此更简单。但是，这并不能完全解释这种现象，因为我们并没有真正地提前停止，而只是轻柔地降低了学习率。

11.11.3 策略

虽然我们不可能涵盖所有类型的学习率调度器，但我们会尝试在下面简要概述常用的策略：多项式衰减和分段常数表。此外，余弦学习率调度在实践中的一些问题上运行效果很好。在某些问题上，最好在使用较高的学习率之前预热优化器。

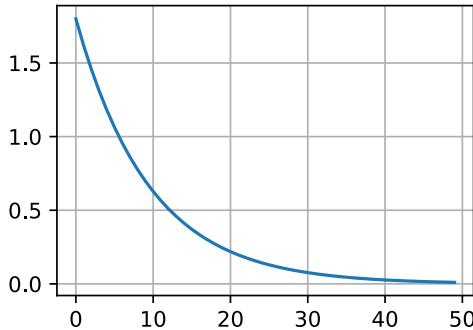
单因子调度器

多项式衰减的一种替代方案是乘法衰减，即 $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ 其中 $\alpha \in (0, 1)$ 。为了防止学习率衰减到一个合理的下界之下，更新方程经常修改为 $\eta_{t+1} \leftarrow \max(\eta_{\min}, \eta_t \cdot \alpha)$ 。

```
class FactorScheduler:
    def __init__(self, factor=1, stop_factor_lr=1e-7, base_lr=0.1):
        self.factor = factor
        self.stop_factor_lr = stop_factor_lr
        self.base_lr = base_lr

    def __call__(self, num_update):
        self.base_lr = max(self.stop_factor_lr, self.base_lr * self.factor)
        return self.base_lr

scheduler = FactorScheduler(factor=0.9, stop_factor_lr=1e-2, base_lr=2.0)
d2l.plot(torch.arange(50), [scheduler(t) for t in range(50)])
```



接下来，我们将使用内置的调度器，但在这里仅解释它们的功能。

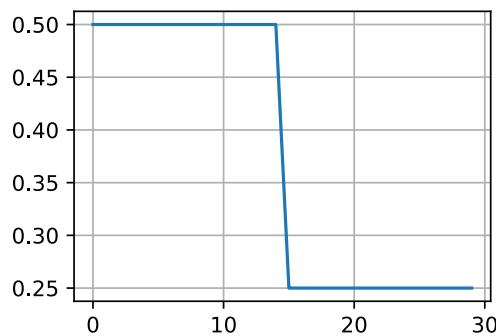
多因子调度器

训练深度网络的常见策略之一是保持学习率为一组分段的常量，并且不时地按给定的参数对学习率做乘法衰减。具体地说，给定一组降低学习率的时间点，例如 $s = \{5, 10, 20\}$ ，每当 $t \in s$ 时，降低 $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ 。假设每步中的值减半，我们可以按如下方式实现这一点。

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.5)
scheduler = lr_scheduler.MultiStepLR(trainer, milestones=[15, 30], gamma=0.5)

def get_lr(trainer, scheduler):
    lr = scheduler.get_last_lr()[0]
    trainer.step()
    scheduler.step()
    return lr

d2l.plot(torch.arange(num_epochs), [get_lr(trainer, scheduler)
                                    for t in range(num_epochs)])
```

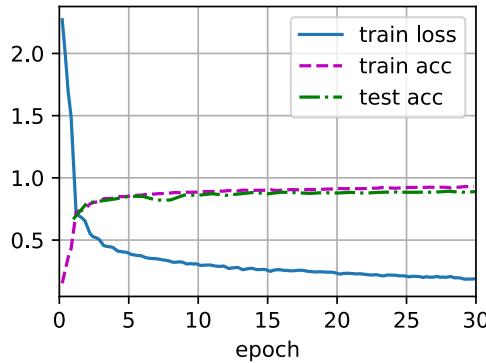


这种分段恒定学习率调度背后的直觉是，让优化持续进行，直到权重向量的分布达到一个驻点。此时，我们

才将学习率降低，以获得更高质量的代理来达到一个良好的局部最小值。下面的例子展示了如何使用这种方法产生更好的解决方案。

```
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.191, train acc 0.928, test acc 0.889
```



余弦调度器

余弦调度器是 (Loshchilov and Hutter, 2016) 提出的一种启发式算法。它所依据的观点是：我们可能不想在一开始就太大地降低学习率，而且可能希望最终能用非常小的学习率来“改进”解决方案。这产生了一个类似于余弦的调度，函数形式如下所示，学习率的值在 $t \in [0, T]$ 之间。

$$\eta_t = \eta_T + \frac{\eta_0 - \eta_T}{2} (1 + \cos(\pi t/T)) \quad (11.11.1)$$

这里 η_0 是初始学习率， η_T 是当 T 时的目标学习率。此外，对于 $t > T$ ，我们只需将值固定到 η_T 而不再增加它。在下面的示例中，我们设置了最大更新步数 $T = 20$ 。

```
class CosineScheduler:
    def __init__(self, max_update, base_lr=0.01, final_lr=0,
                 warmup_steps=0, warmup_begin_lr=0):
        self.base_lr_orig = base_lr
        self.max_update = max_update
        self.final_lr = final_lr
        self.warmup_steps = warmup_steps
        self.warmup_begin_lr = warmup_begin_lr
        self.max_steps = self.max_update - self.warmup_steps

    def get_warmup_lr(self, epoch):
        increase = (self.base_lr_orig - self.warmup_begin_lr) \
```

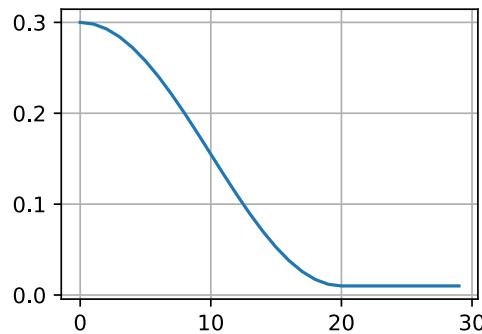
(continues on next page)

(continued from previous page)

```
* float(epoch) / float(self.warmup_steps)
return self.warmup_begin_lr + increase

def __call__(self, epoch):
    if epoch < self.warmup_steps:
        return self.get_warmup_lr(epoch)
    if epoch <= self.max_update:
        self.base_lr = self.final_lr + (
            self.base_lr_orig - self.final_lr) * (1 + math.cos(
                math.pi * (epoch - self.warmup_steps) / self.max_steps)) / 2
    return self.base_lr

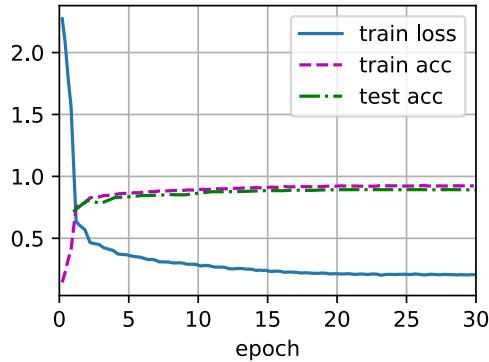
scheduler = CosineScheduler(max_update=20, base_lr=0.3, final_lr=0.01)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



在计算机视觉的背景下，这个调度方式可能产生改进的结果。但请注意，如下所示，这种改进并不一定成立。

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.3)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.207, train acc 0.923, test acc 0.892
```

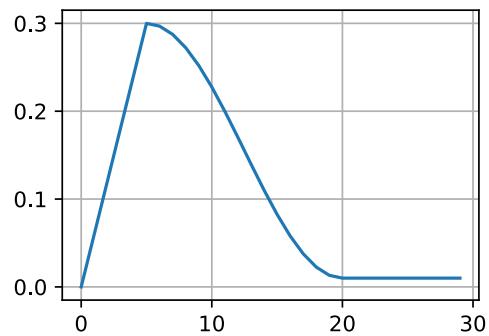


预热

在某些情况下，初始化参数不足以得到良好的解。这对某些高级网络设计来说尤其棘手，可能导致不稳定的优化结果。对此，一方面，我们可以选择一个足够小的学习率，从而防止一开始发散，然而这样进展太缓慢。另一方面，较高的学习率最初就会导致发散。

解决这种困境的一个相当简单的解决方法是使用预热期，在此期间学习率将增加至初始最大值，然后冷却直到优化过程结束。为了简单起见，通常使用线性递增。这引出了如下表所示的时间表。

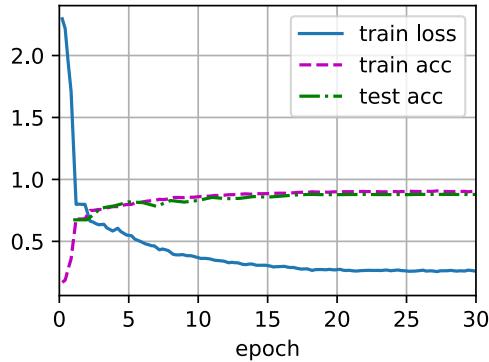
```
scheduler = CosineScheduler(20, warmup_steps=5, base_lr=0.3, final_lr=0.01)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



注意，观察前5个迭代轮数的性能，网络最初收敛得更好。

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.3)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.261, train acc 0.904, test acc 0.878
```



预热可以应用于任何调度器，而不仅仅是余弦。有关学习率调度的更多实验和更详细讨论，请参阅 (Gotmare *et al.*, 2018)。其中，这篇论文的点睛之笔的发现：预热阶段限制了非常深的网络中参数的发散程度。这在直觉上是有道理的：在网络中那些一开始花费最多时间取得进展的部分，随机初始化会产生巨大的发散。

小结

- 在训练期间逐步降低学习率可以提高准确性，并且减少模型的过拟合。
- 在实验中，每当进展趋于稳定时就降低学习率，这是很有效的。从本质上说，这可以确保我们有效地收敛到一个适当的解，也只有这样才能通过降低学习率来减小参数的固有方差。
- 余弦调度器在某些计算机视觉问题中很受欢迎。
- 优化之前的预热期可以防止发散。
- 优化在深度学习中有多种用途。对于同样的训练误差而言，选择不同的优化算法和学习率调度，除了最大限度地减少训练时间，可以导致测试集上不同的泛化和过拟合量。

练习

1. 试验给定固定学习率的优化行为。这种情况下可以获得的最佳模型是什么？
2. 如果改变学习率下降的指数，收敛性会如何改变？在实验中方便起见，使用PolyScheduler。
3. 将余弦调度器应用于大型计算机视觉问题，例如训练ImageNet数据集。与其他调度器相比，它如何影响性能？
4. 预热应该持续多长时间？
5. 可以试着把优化和采样联系起来吗？首先，在随机梯度朗之万动力学上使用 (Welling and Teh, 2011) 的结果。

Discussions¹⁴²

¹⁴² <https://discuss.d2l.ai/t/4334>

计算性能

在深度学习中，数据集和模型通常都很大，导致计算量也会很大。因此，计算的性能非常重要。本章将集中讨论影响计算性能的主要因素：命令式编程、符号编程、异步计算、自动并行和多GPU计算。通过学习本章，对于前几章中实现的那些模型，可以进一步提高它们的计算性能。例如，我们可以在不影响准确性的前提下，大大减少训练时间。

12.1 编译器和解释器

目前为止，本书主要关注的是命令式编程（imperative programming）。命令式编程使用诸如print、“+”和if之类的语句来更改程序的状态。考虑下面这段简单的命令式程序：

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))
```

Python是一种解释型语言 (interpreted language)。因此，当对上面的fancy_func函数求值时，它按顺序执行函数体的操作。也就是说，它将通过对 $e = add(a, b)$ 求值，并将结果存储为变量e，从而更改程序的状态。接下来的两个语句 $f = add(c, d)$ 和 $g = add(e, f)$ 也将执行类似地操作，即执行加法计算并将结果存储为变量。图12.1.1说明了数据流。

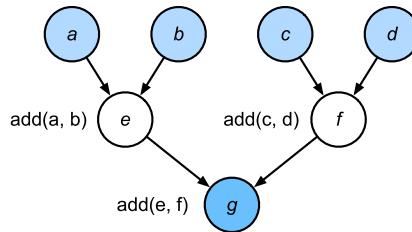


图12.1.1: 命令式编程中的数据流

尽管命令式编程很方便，但可能效率不高。一方面原因，Python会单独执行这三个函数的调用，而没有考虑add函数在fancy_func中被重复调用。如果在一个GPU（甚至多个GPU）上执行这些命令，那么Python解释器产生的开销可能会非常大。此外，它需要保存e和f的变量值，直到fancy_func中的所有语句都执行完毕。这是因为程序不知道在执行语句 $e = add(a, b)$ 和 $f = add(c, d)$ 之后，其他部分是否会使用变量e和f。

12.1.1 符号式编程

考虑另一种选择符号式编程 (symbolic programming)，即代码通常只在完全定义了过程之后才执行计算。这个策略被多个深度学习框架使用，包括Theano和TensorFlow（后者已经获得了命令式编程的扩展）。一般包括以下步骤：

1. 定义计算流程；
2. 将流程编译成可执行的程序；
3. 给定输入，调用编译好的程序执行。

这将允许进行大量的优化。首先，在大多数情况下，我们可以跳过Python解释器。从而消除因为多个更快的GPU与单个CPU上的单个Python线程搭配使用时产生的性能瓶颈。其次，编译器可以将上述代码优化和重写为 $\text{print}((1 + 2) + (3 + 4))$ 甚至 $\text{print}(10)$ 。因为编译器在将其转换为机器指令之前可以看到完整的代码，所以这种优化是可以实现的。例如，只要某个变量不再需要，编译器就可以释放内存（或者从不分配内存），或者将代码转换为一个完全等价的片段。下面，我们将通过模拟命令式编程来进一步了解符号式编程的概念。

```

def add_():
    return ''
def add(a, b):
  
```

(continues on next page)

(continued from previous page)

```
    return a + b
    ...

def fancy_func_():
    return ''
def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
    ...

def evoke_():
    return add_() + fancy_func_() + 'print(fancy_func(1, 2, 3, 4))'

prog = evoke_()
print(prog)
y = compile(prog, '', 'exec')
exec(y)
```

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
print(fancy_func(1, 2, 3, 4))
10
```

命令式（解释型）编程和符号式编程的区别如下：

- 命令式编程更容易使用。在Python中，命令式编程的大部分代码都是简单易懂的。命令式编程也更容易调试，这是因为无论是获取和打印所有的中间变量值，或者使用Python的内置调试工具都更加简单；
- 符号式编程运行效率更高，更易于移植。符号式编程更容易在编译期间优化代码，同时还能够将程序移植到与Python无关的格式中，从而允许程序在非Python环境中运行，避免了任何潜在的与Python解释器相关的性能问题。

12.1.2 混合式编程

历史上，大部分深度学习框架都在命令式编程与符号式编程之间进行选择。例如，Theano、TensorFlow（灵感来自前者）、Keras和CNTK采用了符号式编程。相反地，Chainer和PyTorch采取了命令式编程。在后来的版本更新中，TensorFlow2.0和Keras增加了命令式编程。

如上所述，PyTorch是基于命令式编程并且使用动态计算图。为了能够利用符号式编程的可移植性和效率，开发人员思考能否将这两种编程模型的优点结合起来，于是就产生了torchscript。torchscript允许用户使用纯命令式编程进行开发和调试，同时能够将大多数程序转换为符号式程序，以便在需要产品级计算性能和部署时使用。

12.1.3 Sequential的混合式编程

要了解混合式编程的工作原理，最简单的方法是考虑具有多层的深层网络。按照惯例，Python解释器需要执行所有层的代码来生成一条指令，然后将该指令转发到CPU或GPU。对于单个的（快速的）计算设备，这不会导致任何重大问题。另一方面，如果我们使用先进的8-GPU服务器，比如AWS P3dn.24xlarge实例，Python将很难让所有的GPU都保持忙碌。在这里，瓶颈是单线程的Python解释器。让我们看看如何通过将Sequential替换为HybridSequential来解决代码中这个瓶颈。首先，我们定义一个简单的多层感知机。

```
import torch
from torch import nn
from d2l import torch as d2l

# 生产网络的工厂模式
def get_net():
    net = nn.Sequential(nn.Linear(512, 256),
                        nn.ReLU(),
                        nn.Linear(256, 128),
                        nn.ReLU(),
                        nn.Linear(128, 2))
    return net

x = torch.randn(size=(1, 512))
net = get_net()
net(x)
```

```
tensor([[ 0.0722, -0.0190]], grad_fn=<AddmmBackward0>)
```

通过使用`torch.jit.script`函数来转换模型，我们就有能力编译和优化多层感知机中的计算，而模型的计算结果保持不变。

```
net = torch.jit.script(net)
net(x)
```

```
tensor([[ 0.0722, -0.0190]], grad_fn=<AddmmBackward0>)
```

我们编写与之前相同的代码，再使用`torch.jit.script`简单地转换模型，当完成这些任务后，网络就将得到优化（我们将在下面对性能进行基准测试）。

通过混合式编程加速

为了证明通过编译获得了性能改进，我们比较了混合编程前后执行`net(x)`所需的时间。让我们先定义一个度量时间的类，它在本章中在衡量（和改进）模型性能时将非常有用。

```
#@save
class Benchmark:
    """用于测量运行时间"""
    def __init__(self, description='Done'):
        self.description = description

    def __enter__(self):
        self.timer = d2l.Timer()
        return self

    def __exit__(self, *args):
        print(f'{self.description}: {self.timer.stop():.4f} sec')
```

现在我们可以调用网络两次，一次使用`torchscript`，一次不使用`torchscript`。

```
net = get_net()
with Benchmark('无torchscript'):
    for i in range(1000): net(x)

net = torch.jit.script(net)
with Benchmark('有torchscript'):
    for i in range(1000): net(x)
```

```
无torchscript: 0.1361 sec
有torchscript: 0.1204 sec
```

如以上结果所示，在`nn.Sequential`的实例被函数`torch.jit.script`脚本化后，通过使用符号式编程提高了计算性能。

序列化

编译模型的好处之一是我们可以将模型及其参数序列化（保存）到磁盘。这允许这些训练好的模型部署到其他设备上，并且还能方便地使用其他前端编程语言。同时，通常编译模型的代码执行速度也比命令式编程更快。让我们看看`save`的实际功能。

```
net.save('my_mlp')  
!ls -lh my_mlp*
```

```
-rw-r--r-- 1 ci ci 651K Aug 18 06:58 my_mlp
```

小结

- 命令式编程使得新模型的设计变得容易，因为可以依据控制流编写代码，并拥有相对成熟的Python软件生态。
- 符号式编程要求我们先定义并且编译程序，然后再执行程序，其好处是提高了计算性能。

练习

- 回顾前几章中感兴趣的模型，能提高它们的计算性能吗？

Discussions¹⁴³

12.2 异步计算

今天的计算机是高度并行的系统，由多个CPU核、多个GPU、多个处理单元组成。通常每个CPU核有多个线程，每个设备通常有多个GPU，每个GPU有多个处理单元。总之，我们可以同时处理许多不同的事情，并且通常是在不同的设备上。不幸的是，Python并不善于编写并行和异步代码，至少在没有额外帮助的情况下不是好选择。归根结底，Python是单线程的，将来也是不太可能改变的。因此在诸多的深度学习框架中，MXNet和TensorFlow之类则采用了一种异步编程（asynchronous programming）模型来提高性能，而PyTorch则使用了Python自己的调度器来实现不同的性能权衡。对PyTorch来说GPU操作在默认情况下是异步的。当调用一个使用GPU的函数时，操作会排队到特定的设备上，但不一定要等到以后才执行。这允许我们并行执行更多的计算，包括在CPU或其他GPU上的操作。

因此，了解异步编程是如何工作的，通过主动地减少计算需求和相互依赖，有助于我们开发更高效的程序。这能够减少内存开销并提高处理器利用率。

¹⁴³ <https://discuss.d2l.ai/t/2788>

```
import os
import subprocess
import numpy
import torch
from torch import nn
from d2l import torch as d2l
```

12.2.1 通过后端异步处理

作为热身，考虑一个简单问题：生成一个随机矩阵并将其相乘。让我们在NumPy和PyTorch张量中都这样做，看看它们的区别。请注意，PyTorch的tensor是在GPU上定义的。

```
# GPU计算热身
device = d2l.try_gpu()
a = torch.randn(size=(1000, 1000), device=device)
b = torch.mm(a, a)

with d2l.Benchmark('numpy'):
    for _ in range(10):
        a = numpy.random.normal(size=(1000, 1000))
        b = numpy.dot(a, a)

with d2l.Benchmark('torch'):
    for _ in range(10):
        a = torch.randn(size=(1000, 1000), device=device)
        b = torch.mm(a, a)
```

```
numpy: 1.0704 sec
torch: 0.0013 sec
```

通过PyTorch的基准输出比较快了几个数量级。NumPy点积是在CPU上执行的，而PyTorch矩阵乘法是在GPU上执行的，后者的速度要快得多。但巨大的时间差距表明一定还有其他原因。默认情况下，GPU操作在PyTorch中是异步的。强制PyTorch在返回之前完成所有计算，这种强制说明了之前发生的情况：计算是由后端执行，而前端将控制权返回给了Python。

```
with d2l.Benchmark():
    for _ in range(10):
        a = torch.randn(size=(1000, 1000), device=device)
        b = torch.mm(a, a)
        torch.cuda.synchronize(device)
```

Done: 0.0049 sec

广义上说，PyTorch有一个用于与用户直接交互的前端（例如通过Python），还有一个由系统用来执行计算的后端。如图12.2.1所示，用户可以用各种前端语言编写PyTorch程序，如Python和C++。不管使用的前端编程语言是什么，PyTorch程序的执行主要发生在C++实现的后端。由前端语言发出的操作被传递到后端执行。后端管理自己的线程，这些线程不断收集和执行排队的任务。请注意，要使其工作，后端必须能够跟踪计算图中各个步骤之间的依赖关系。因此，不可能并行化相互依赖的操作。

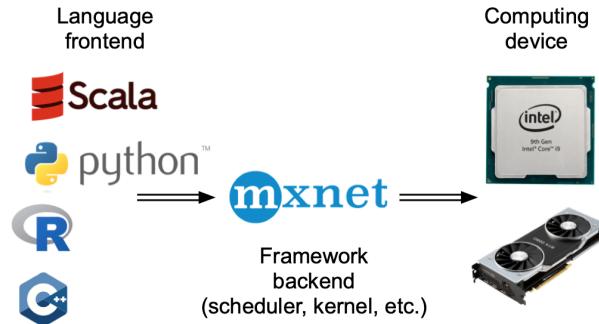


图12.2.1：编程语言前端和深度学习框架后端

接下来看看另一个简单例子，以便更好地理解依赖关系图。

```
x = torch.ones((1, 2), device=device)
y = torch.ones((1, 2), device=device)
z = x * y + 2
z
```

```
tensor([[3., 3.]], device='cuda:0')
```

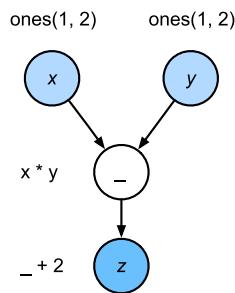


图12.2.2：后端跟踪计算图中各个步骤之间的依赖关系

上面的代码片段在图12.2.2中进行了说明。每当Python前端线程执行前三条语句中的一条语句时，它只是将任务返回到后端队列。当最后一个语句的结果需要被打印出来时，Python前端线程将等待C++后端线程完成

变量 z 的结果计算。这种设计的一个好处是Python前端线程不需要执行实际的计算。因此，不管Python的性能如何，对程序的整体性能几乎没有影响。图12.2.3演示了前端和后端如何交互。

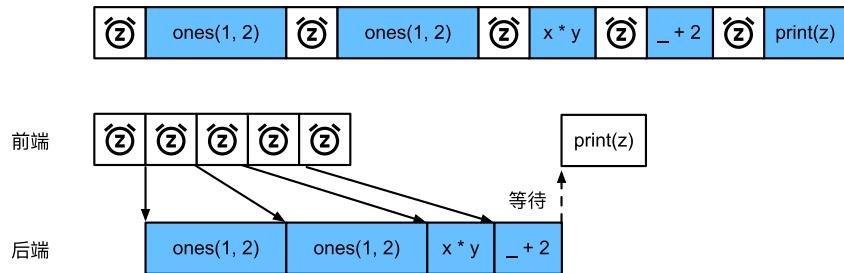


图12.2.3: 前端和后端的交互

12.2.2 障碍器与阻塞器

12.2.3 改进计算

Python前端线程和C++后端线程之间的简化交互可以概括如下：

1. 前端命令后端将计算任务 $y = x + 1$ 插入队列；
2. 然后后端从队列接收计算任务并执行；
3. 然后后端将计算结果返回到前端。

假设这三个阶段的持续时间为 t_1, t_2, t_3 。如果不使用异步编程，执行10000次计算所需的总时间为 $10000(t_1 + t_2 + t_3)$ 。如果使用异步编程，因为前端不必等待后端为每个循环返回计算结果，执行10000次计算所花费的总时间可以减少到 $t_1 + 10000t_2 + t_3$ （假设 $10000t_2 > 9999t_1$ ）。

小结

- 深度学习框架可以将Python前端的控制与后端的执行解耦，使得命令可以快速地异步插入后端、并行执行。
- 异步产生了一个相当灵活的前端，但请注意：过度填充任务队列可能会导致内存消耗过多。建议对每个小批量进行同步，以保持前端和后端大致同步。
- 芯片供应商提供了复杂的性能分析工具，以获得对深度学习效率更精确的洞察。

练习

- 在CPU上，对本节中相同的矩阵乘法操作进行基准测试，仍然可以通过后端观察异步吗？

Discussions¹⁴⁴

12.3 自动并行

深度学习框架（例如，MxNet、飞桨和PyTorch）会在后端自动构建计算图。利用计算图，系统可以了解所有依赖关系，并且可以选择性地并行执行多个不相互依赖的任务以提高速度。例如，12.2节中的图12.2.2独立初始化两个变量。因此，系统可以选择并行执行它们。

通常情况下单个操作符将使用所有CPU或单个GPU上的所有计算资源。例如，即使在一台机器上有多个CPU处理器，dot操作符也将使用所有CPU上的所有核心（和线程）。这样的行为同样适用于单个GPU。因此，并行化对单设备计算机来说并不是很有用，而并行化对于多个设备就很重要了。虽然并行化通常应用在多个GPU之间，但增加本地CPU以后还将提高少许性能。例如，(Hadjis et al., 2016)则把结合GPU和CPU的训练应用到计算机视觉模型中。借助自动并行化框架的便利性，我们可以依靠几行Python代码实现相同的目标。对自动并行计算的讨论主要集中在使用CPU和GPU的并行计算上，以及计算和通信的并行化内容。

请注意，本节中的实验至少需要两个GPU来运行。

```
import torch
from d2l import torch as d2l
```

12.3.1 基于GPU的并行计算

从定义一个具有参考性的用于测试的工作负载开始：下面的run函数将执行10次矩阵—矩阵乘法时需要使用的数据分配到两个变量（x_gpu1和x_gpu2）中，这两个变量分别位于选择的不同设备上。

```
devices = d2l.try_all_gpus()
def run(x):
    return [x.mm(x) for _ in range(50)]

x_gpu1 = torch.rand(size=(4000, 4000), device=devices[0])
x_gpu2 = torch.rand(size=(4000, 4000), device=devices[1])
```

现在使用函数来处理数据。通过在测量之前需要预热设备（对设备执行一次传递）来确保缓存的作用不影响最终的结果。`torch.cuda.synchronize()`函数将会等待一个CUDA设备上的所有流中的所有核心的计算完成。函数接受一个device参数，代表是哪个设备需要同步。如果device参数是None（默认值），它将使用`current_device()`找出的当前设备。

¹⁴⁴ <https://discuss.d2l.ai/t/2791>

```
run(x_gpu1)
run(x_gpu2) # 预热设备
torch.cuda.synchronize(devices[0])
torch.cuda.synchronize(devices[1])

with d2l.Benchmark('GPU1 time'):
    run(x_gpu1)
    torch.cuda.synchronize(devices[0])

with d2l.Benchmark('GPU2 time'):
    run(x_gpu2)
    torch.cuda.synchronize(devices[1])
```

```
GPU1 time: 0.4600 sec
GPU2 time: 0.4706 sec
```

如果删除两个任务之间的synchronize语句，系统就可以在两个设备上自动实现并行计算。

```
with d2l.Benchmark('GPU1 & GPU2'):
    run(x_gpu1)
    run(x_gpu2)
    torch.cuda.synchronize()
```

```
GPU1 & GPU2: 0.4580 sec
```

在上述情况下，总执行时间小于两个部分执行时间的总和，因为深度学习框架自动调度两个GPU设备上的计算，而不需要用户编写复杂的代码。

12.3.2 并行计算与通信

在许多情况下，我们需要在不同的设备之间移动数据，比如在CPU和GPU之间，或者在不同的GPU之间。例如，当执行分布式优化时，就需要移动数据来聚合多个加速卡上的梯度。让我们通过在GPU上计算，然后将结果复制回CPU来模拟这个过程。

```
def copy_to_cpu(x, non_blocking=False):
    return [y.to('cpu', non_blocking=non_blocking) for y in x]

with d2l.Benchmark('在GPU1上运行'):
    y = run(x_gpu1)
    torch.cuda.synchronize()
```

(continues on next page)

(continued from previous page)

```
with d2l.Benchmark('复制到CPU'):  
    y_cpu = copy_to_cpu(y)  
    torch.cuda.synchronize()
```

在GPU1上运行: 0.4608 sec
复制到CPU: 2.3504 sec

这种方式效率不高。注意到当列表中的其余部分还在计算时，我们可能就已经开始将y的部分复制到CPU了。例如，当计算一个小批量的（反传）梯度时。某些参数的梯度将比其他参数的梯度更早可用。因此，在GPU仍在运行时就开始使用PCI-Express总线带宽来移动数据是有利的。在PyTorch中，`to()`和`copy_()`等函数都允许显式的`non_blocking`参数，这允许在不需要同步时调用方可以绕过同步。设置`non_blocking=True`以模拟这个场景。

```
with d2l.Benchmark('在GPU1上运行并复制到CPU'):  
    y = run(x_gpu1)  
    y_cpu = copy_to_cpu(y, True)  
    torch.cuda.synchronize()
```

在GPU1上运行并复制到CPU: 1.7703 sec

两个操作所需的总时间少于它们各部分操作所需时间的总和。请注意，与并行计算的区别是通信操作使用的资源：CPU和GPU之间的总线。事实上，我们可以在两个设备上同时进行计算和通信。如上所述，计算和通信之间存在的依赖关系是必须先计算 $y[i]$ ，然后才能将其复制到CPU。幸运的是，系统可以在计算 $y[i]$ 的同时复制 $y[i-1]$ ，以减少总的运行时间。

最后，本节给出了一个简单的两层多层感知机在CPU和两个GPU上训练时的计算图及其依赖关系的例子，如图12.3.1所示。手动调度由此产生的并行程序将是相当痛苦的。这就是基于图的计算后端进行优化的优势所在。

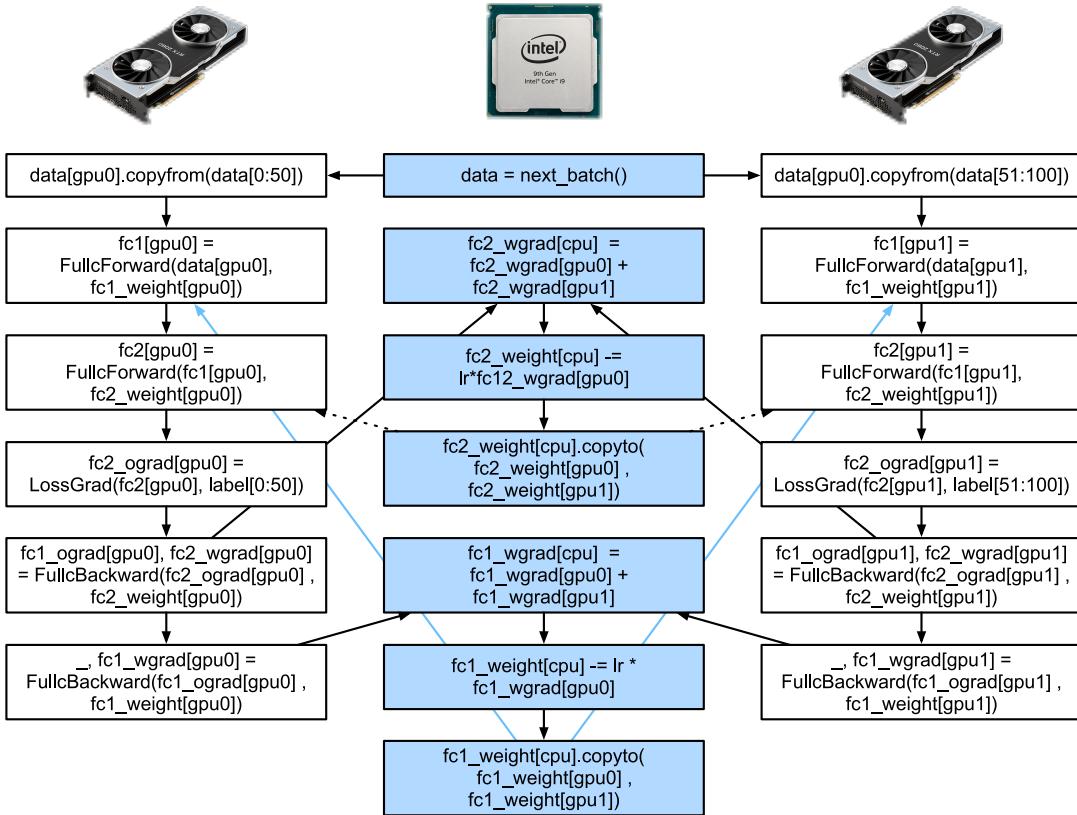


图12.3.1: 在一个CPU和两个GPU上的两层的多层感知机的计算图及其依赖关系

小结

- 现代系统拥有多种设备，如多个GPU和多个CPU，还可以并行地、异步地使用它们。
- 现代系统还拥有各种通信资源，如PCI Express、存储（通常是固态硬盘或网络存储）和网络带宽，为了达到最高效率可以并行使用它们。
- 后端可以通过自动化地并行计算和通信来提高性能。

练习

- 在本节定义的run函数中执行了八个操作，并且操作之间没有依赖关系。设计一个实验，看看深度学习框架是否会自动地并行地执行它们。
- 当单个操作符的工作量足够小，即使在单个CPU或GPU上，并行化也会有所帮助。设计一个实验来验证这一点。
- 设计一个实验，在CPU和GPU这两种设备上使用并行计算和通信。
- 使用诸如NVIDIA的Nsight¹⁴⁵之类的调试器来验证代码是否有效。

¹⁴⁵ https://developer.nvidia.com/nsight-compute-2019_5

5. 设计并实验具有更加复杂的数据依赖关系的计算任务，以查看是否可以在提高性能的同时获得正确的结果。

Discussions¹⁴⁶

12.4 硬件

很好地理解算法和模型才可以捕获统计方面的问题，构建出具有出色性能的系统。同时，至少对底层硬件有一定的了解也是必不可少的。本节不能替代硬件和系统设计的相关课程。相反，本节的内容可以作为理解某些算法为什么比其他算法更高效以及如何实现良好吞吐量的起点。一个好的设计可以很容易地在性能上造就数量级的差异，这也是后续产生的能够训练网络（例如，训练时间为1周）和无法训练网络（训练时间为3个月，导致错过截止期）之间的差异。我们先从计算机的研究开始。然后深入查看CPU和GPU。最后，再查看数据中心或云中的多台计算机的连接方式。

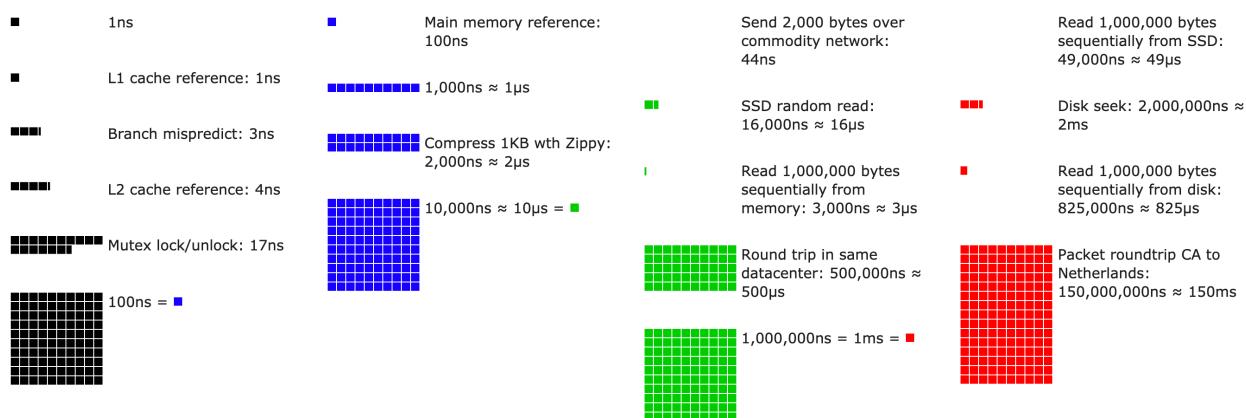


图12.4.1：每个程序员都应该知道的延迟数字

也可以通过 图12.4.1 进行简单的了解，图片源自科林·斯科特的[互动帖子](#)¹⁴⁷，在帖子中很好地概述了过去十年的进展。原始的数字是取自于杰夫迪恩的Stanford讲座¹⁴⁸。下面的讨论解释了这些数字的一些基本原理，以及它们如何指导我们去设计算法。下面的讨论是非常笼统和粗略的。很显然，它并不能代替一门完整的课程，而只是为了给统计建模者提供足够的信息，让他们做出合适的设计决策。对于计算机体系结构的深入概述，建议读者参考 (Hennessy and Patterson, 2011) 或关于该主题的最新课程，例如 Arste Asanovic¹⁴⁹。

¹⁴⁶ <https://discuss.d2l.ai/t/2794>

¹⁴⁷ https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

¹⁴⁸ <https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf>

¹⁴⁹ <http://inst.eecs.berkeley.edu/~cs152/sp19/>

12.4.1 计算机

大多数深度学习研究者和实践者都可以使用一台具有相当数量的内存、计算资源、某种形式的加速器（如一个或者多个GPU）的计算机。计算机由以下关键部件组成：

- 一个处理器（也被称为CPU），它除了能够运行操作系统和许多其他功能之外，还能够执行给定的程序。它通常由8个或更多个核心组成；
- 内存（随机访问存储，RAM）用于存储和检索计算结果，如权重向量和激活参数，以及训练数据；
- 一个或多个以太网连接，速度从1GB/s到100GB/s不等。在高端服务器上可能用到更高级的互连；
- 高速扩展总线（PCIe）用于系统连接一个或多个GPU。服务器最多有8个加速卡，通常以更高级的拓扑方式连接，而桌面系统则有1个或2个加速卡，具体取决于用户的预算和电源负载的大小；
- 持久性存储设备，如磁盘驱动器、固态驱动器，在许多情况下使用高速扩展总线连接。它为系统需要的训练数据和中间检查点需要的存储提供了足够的传输速度。

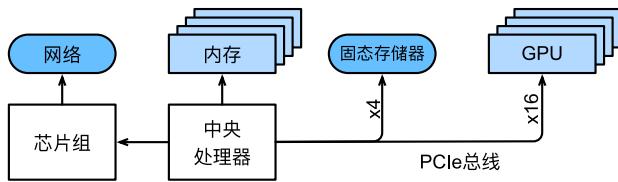


图12.4.2: 计算机组件的连接

如图12.4.2所示，高速扩展总线由直接连接到CPU的多个通道组成，将CPU与大多数组件（网络、GPU和存储）连接在一起。例如，AMD的Threadripper3有64个PCIe4.0通道，每个通道都能够双向传输16Gbit/s的数据。内存直接连接到CPU，总带宽高达100GB/s。

当我们在计算机上运行代码时，需要将数据转移到处理器上（CPU或GPU）执行计算，然后将结果从处理器移回到随机访问存储和持久存储器中。因此，为了获得良好的性能，需要确保每一步工作都能无缝链接，而不希望系统中的任何一部分成为主要的瓶颈。例如，如果不能快速加载图像，那么处理器就无事可做。同样地，如果不能快速移动矩阵到CPU（或GPU）上，那么CPU（或GPU）就会无法全速运行。最后，如果希望在网络上同步多台计算机，那么网络就不应该拖累计算速度。一种选择是通信和计算交错进行。接下来将详细介绍各个组件。

12.4.2 内存

最基本的内存主要用于存储需要随时访问的数据。目前，CPU的内存通常为DDR4¹⁵⁰类型，每个模块提供20-25Gb/s的带宽。每个模块都有一条64位宽的总线。通常使用成对的内存模块来允许多个通道。CPU有2到4个内存通道，也就是说，它们内存带宽的峰值在40GB/s到100GB/s之间。一般每个通道有两个物理存储体（bank）。例如AMD的Zen 3 Threadripper有8个插槽。

¹⁵⁰ https://en.wikipedia.org/wiki/DDR4_SDRAM

虽然这些数字令人印象深刻，但实际上它们只能说明了一部分故事。当我们想要从内存中读取一部分内容时，需要先告诉内存模块在哪里可以找到信息。也就是说，我们需要先将地址（address）发送到RAM。然后我们可以选择只读取一条64位记录还是一长串记录。后者称为突发读取（burst read）。概括地说，向内存发送地址并设置传输大约需要100ns（细节取决于所用内存芯片的特定定时系数），每个后续传输只需要0.2ns。总之，第一次读取的成本是后续读取的500倍！请注意，每秒最多可以执行一千万次随机读取。这说明应该尽可能地避免随机内存访问，而是使用突发模式读取和写入。

当考虑到拥有多个物理存储体时，事情就更加复杂了。每个存储体大部分时候都可以独立地读取内存。这意味着两件事。一方面，如果随机读操作均匀分布在内存中，那么有效的随机读操作次数将高达4倍。这也意味着执行随机读取仍然不是一个好主意，因为突发读取的速度也快了4倍。另一方面，由于内存对齐是64位边界，因此最好将任何数据结构与相同的边界对齐。当设置了适当的标志时，编译器基本上就是自动化¹⁵¹地执行对齐操作。我们鼓励好奇的读者回顾一下Zeshan Chishti关于DRAM的讲座¹⁵²。

GPU内存的带宽要求甚至更高，因为它们的处理单元比CPU多得多。总的来说，解决这些问题有两种选择。首先是使内存总线变得更宽。例如，NVIDIA的RTX 2080Ti有一条352位宽的总线。这样就可以同时传输更多的信息。其次，GPU使用特定的高性能内存。消费级设备，如NVIDIA的RTX和Titan系列，通常使用GDDR6¹⁵³模块。它们使用截然不同的接口，直接与专用硅片上的GPU连接。这使得它们非常昂贵，通常仅限于高端服务器芯片，如NVIDIA Volta V100系列加速卡。毫不意外的是GPU的内存通常比CPU的内存小得多，因为前者的成本更高。就目的而言，它们的性能与特征大体上是相似的，只是GPU的速度更快。就本书而言，我们完全可以忽略细节，因为这些技术只在调整GPU核心以获得高吞吐量时才起作用。

12.4.3 存储器

随机访问存储的一些关键特性是带宽（bandwidth）和延迟（latency）。存储设备也是如此，只是不同设备之间的特性差异可能更大。

硬盘驱动器

硬盘驱动器（hard disk drive, HDD）已经使用了半个多世纪。简单的说，它们包含许多旋转的盘片，这些盘片的磁头可以放置在任何给定的磁道上进行读写。高端磁盘在9个盘片上可容纳高达16TB的容量。硬盘的主要优点之一是相对便宜，而它们的众多缺点之一是典型的灾难性故障模式和相对较高的读取延迟。

要理解后者，请了解一个事实即硬盘驱动器的转速大约为7200RPM（每分钟转数）。它们如果转速再快些，就会由于施加在碟片上的离心力而破碎。在访问磁盘上的特定扇区时，还有一个关键问题：需要等待碟片旋转到位（可以移动磁头，但是无法对磁盘加速）。因此，可能需要8毫秒才能使用请求的数据。一种常见的描述方式是，硬盘驱动器可以以大约100IOPs（每秒输入/输出操作）的速度工作，并且在过去二十年中这个数字基本上没变。同样糟糕的是，带宽（大约为100-200MB/s）也很难增加。毕竟，每个磁头读取一个磁道的比特，因此比特率只随信息密度的平方根缩放。因此，对于非常大的数据集，HDD正迅速降级为归档存储和低级存储。

¹⁵¹ https://en.wikipedia.org/wiki/Data_structure_alignment

¹⁵² http://web.cecs.pdx.edu/~zeshan/ece585_lec5.pdf

¹⁵³ https://en.wikipedia.org/wiki/GDDR6_SDRAM 500GB/s HBM

固态驱动器

固态驱动器 (solid state drives, SSD) 使用闪存持久地存储信息。这允许更快地访问存储的记录。现代的固态驱动器的IOPs可以达到10万到50万，比硬盘驱动器快3个数量级。而且，它们的带宽可以达到1-3GB/s，比硬盘驱动器快一个数量级。这些改进听起来好的难以置信，而事实上受固态驱动器的设计方式，它仍然存在下面的附加条件。

- 固态驱动器以块的方式（256KB或更大）存储信息。块只能作为一个整体来写入，因此需要耗费大量的时间，导致固态驱动器在按位随机写入时性能非常差。而且通常数据写入需要大量的时间还因为块必须被读取、擦除，然后再重新写入新的信息。如今固态驱动器的控制器和固件已经开发出了缓解这种情况的算法。尽管有了算法，写入速度仍然会比读取慢得多，特别是对于QLC（四层单元）固态驱动器。提高性能的关键是维护操作的“队列”，在队列中尽可能地优先读取和写入大的块。
- 固态驱动器中的存储单元磨损得比较快（通常在几千次写入之后就已经老化了）。磨损程度保护算法能够将退化平摊到许多单元。也就是说，不建议将固态驱动器用于交换分区文件或大型日志文件。
- 最后，带宽的大幅增加迫使计算机设计者将固态驱动器与PCIe总线相连接，这种驱动器称为NVMe（非易失性内存增强），其最多可以使用4个PCIe通道。在PCIe4.0上最高可达8GB/s。

云存储

云存储提供了一系列可配置的性能。也就是说，虚拟机的存储在数量和速度上都能根据用户需要进行动态分配。建议用户在延迟太高时（例如，在训练期间存在许多小记录时）增加IOPs的配置数。

12.4.4 CPU

中央处理器 (central processing unit, CPU) 是任何计算机的核心。它们由许多关键组件组成：处理器核心 (processor cores) 用于执行机器代码的；总线 (bus) 用于连接不同组件（注意，总线会因为处理器型号、各代产品和供应商之间的特定拓扑结构有明显不同）；缓存 (cach) 相比主内存实现更高的读取带宽和更低的延迟内存访问。最后，因为高性能线性代数和卷积运算常见于媒体处理和机器学习中，所以几乎所有的现代CPU都包含向量处理单元 (vector processing unit) 为这些计算提供辅助。

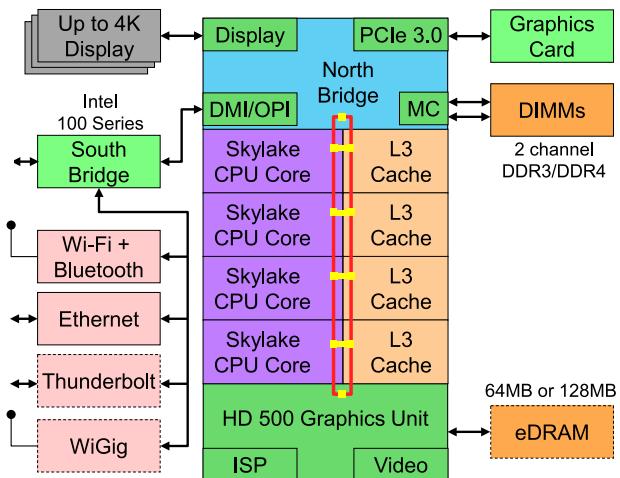


图12.4.3: Intel Skylake消费级四核CPU

图12.4.3描述了Intel Skylake消费级四核CPU。它包含一个集成GPU、缓存和一个连接四个核心的环总线。例如，以太网、WiFi、蓝牙、SSD控制器和USB这些外围设备要么是芯片组的一部分，要么通过PCIe直接连接到CPU。

微体系结构

每个处理器核心都由一组相当复杂的组件组成。虽然不同时代的产品和供应商的细节有所不同，但基本功能都是标准的。前端加载指令并尝试预测将采用哪条路径（例如，为了控制流），然后将指令从汇编代码解码为微指令。汇编代码通常不是处理器执行的最低级别代码，而复杂的微指令却可以被解码成一组更低级的操作，然后由实际的执行核心处理。通常执行核心能够同时执行许多操作，例如，图12.4.4的ARM Cortex A77核心可以同时执行多达8个操作。

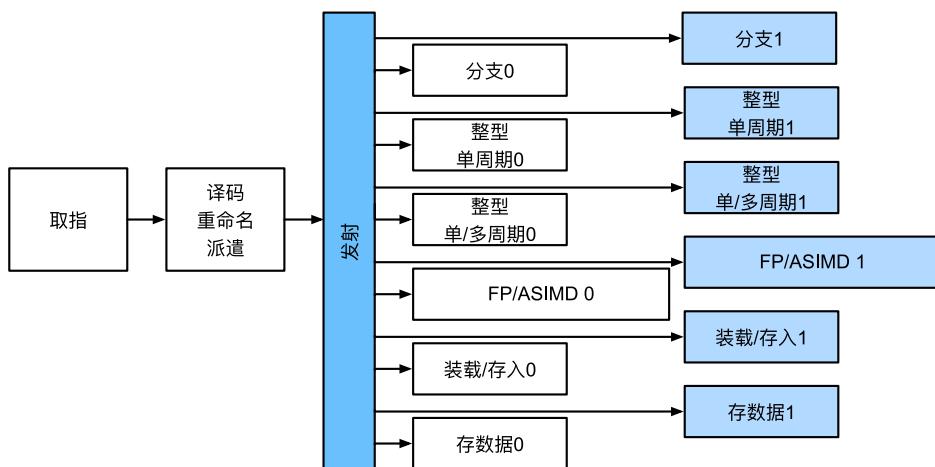


图12.4.4: ARM Cortex A77微体系结构

这意味着高效的程序可以在每个时钟周期内执行多条指令，前提是这些指令可以独立执行。不是所有的处理单元都是平等的。一些专用于处理整数指令，而另一些则针对浮点性能进行了优化。为了提高吞吐量，处理器还可以在分支指令中同时执行多条代码路径，然后丢弃未选择分支的结果。这就是为什么前端的分支预测单元很重要，因为只有最有希望的路径才会被继续执行。

矢量化

深度学习的计算量非常大。因此，为了满足机器学习的需要，CPU需要在一个时钟周期内执行许多操作。这种执行方式是通过向量处理单元实现的。这些处理单元有不同的名称：在ARM上叫做NEON，在x86上被称为AVX2¹⁵⁴。一个常见的功能是它们能够执行单指令多数据（single instruction multiple data, SIMD）操作。图12.4.5显示了如何在ARM上的一个时钟周期中完成8个整数加法。

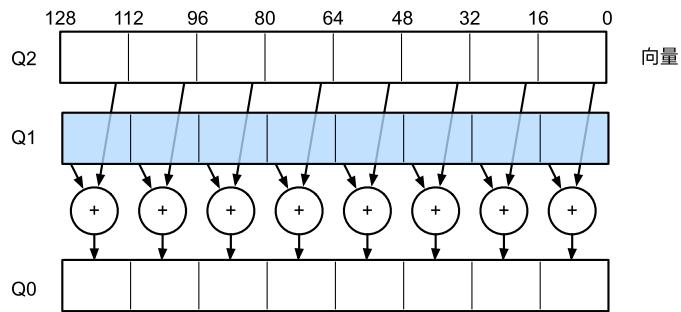


图12.4.5: 128位NEON矢量化

根据体系结构的选择，此类寄存器最长可达512位，最多可组合64对数字。例如，我们可能会将两个数字相乘，然后与第三个数字相加，这也称为乘加融合（fused multiply-add）。Intel的OpenVino¹⁵⁵就是使用这些处理器来获得可观的吞吐量，以便在服务器级CPU上进行深度学习。不过请注意，这个数字与GPU的能力相比则相形见绌。例如，NVIDIA的RTX 2080Ti拥有4352个CUDA核心，每个核心都能够在任何时候处理这样的操作。

缓存

考虑以下情况：我们有一个中等规模的4核心的CPU，如图12.4.3所示，运行在2GHz频率。此外，假设向量处理单元启用了256位带宽的AVX2，其IPC（指令/时钟）计数为1。进一步假设从内存中获取用于AVX2操作的指令至少需要一个寄存器。这意味着CPU每个时钟周期需要消耗 $4 \times 256 \text{ bit} = 128 \text{ bytes}$ 的数据。除非我们能够每秒向处理器传输 $2 \times 10^9 \times 128 = 256 \times 10^9 \text{字节}$ ，否则用于处理的数据将会不足。不幸的是，这种芯片的存储器接口仅支持20-40Gb/s的数据传输，即少了一个数量级。解决方法是尽可能避免从内存中加载新数据，而是将数据放在CPU的缓存上。这就是使用缓存的地方。通常使用以下名称或概念。

- 寄存器，严格来说不是缓存的一部分，用于帮助组织指令。也就是说，寄存器是CPU可以以时钟速度访问而没有延迟的存储位置。CPU有几十个寄存器，因此有效地使用寄存器取决于编译器（或程序员）。例如，C语言有一个register关键字。

¹⁵⁴ https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

¹⁵⁵ <https://01.org/openvino/toolkit>

- 一级缓存是应对高内存带宽要求的第一道防线。一级缓存很小（常见的大小可能是32-64KB），内容通常分为数据和指令。当数据在一级缓存中被找到时，其访问速度非常快，如果没有在那里找到，搜索将沿着缓存层次结构向下寻找。
- 二级缓存是下一站。根据架构设计和处理器大小的不同，它们可能是独占的也可能是共享的。即它们可能只能由给定的核心访问，或者在多个核心之间共享。二级缓存比一级缓存大（通常每个核心256-512KB），而速度也更慢。此外，我们首先需要检查以确定数据不在一级缓存中，才会访问二级缓存中的内容，这会增加少量的额外延迟。
- 三级缓存在多个核之间共享，并且可以非常大。AMD的EPYC 3服务器的CPU在多个芯片上拥有高达256MB的高速缓存。更常见的数字在4-8MB范围内。

预测下一步需要哪个存储设备是优化芯片设计的关键参数之一。例如，建议以向前的方向遍历内存，因为大多数缓存算法将试图向前读取（read forward）而不是向后读取。同样，将内存访问模式保持在本地也是提高性能的一个好方法。

添加缓存是一把双刃剑。一方面，它能确保处理器核心不缺乏数据。但同时，它也增加了芯片尺寸，消耗了原本可以用来提高处理能力的面积。此外，缓存未命中的代价可能会很昂贵。考虑最坏的情况，如图12.4.6所示的错误共享（false sharing）。当处理器1上的线程请求数据时，内存位置缓存在处理器0上。为了满足获取需要，处理器0需要停止它正在做的事情，将信息写回主内存，然后让处理器1从内存中读取它。在此操作期间，两个处理器都需要等待。与高效的单处理器实现相比，这种代码在多个处理器上运行的速度可能要慢得多。这就是为什么缓存大小（除了物理大小之外）有实际限制的另一个原因。

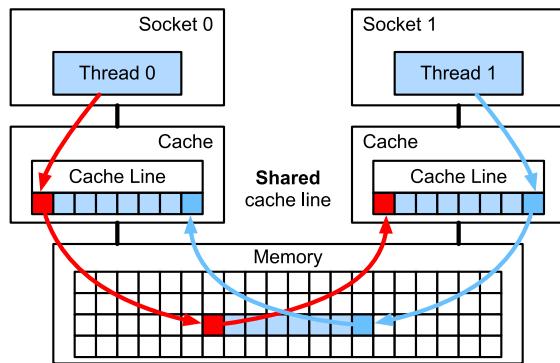


图12.4.6: 错误共享 (图片由英特尔提供)

12.4.5 GPU和其他加速卡

毫不夸张地说，如果没有GPU，深度学习就不会成功。基于同样的原因，有理由认为GPU制造商的财富由于深度学习而显著增加。这种硬件和算法的协同进化导致了这样一种情况：无论好坏，深度学习都是更可取的统计建模范式。因此，了解GPU和其他加速卡（如TPU (*Jouppi et al., 2017*)）的具体好处是值得的。

值得注意的是，在实践中经常会有这样一个判别：加速卡是为训练还是推断而优化的。对于后者，我们只需要计算网络中的前向传播。而反向传播不需要存储中间数据。还有，我们可能不需要非常精确的计算（FP16或INT8通常就足够了）。对于前者，即训练过程中需要存储所有的中间结果用来计算梯度。而且，累积梯度也需要更高的精度，以避免数值下溢（或溢出）。这意味着最低要求也是FP16（或FP16与FP32的混合精度）。所有这些都

需要更快、更大的内存（HBM2或者GDDR6）和更高的处理能力。例如，NVIDIA优化了Turing¹⁵⁶ T4 GPU用于推断和V100 GPU用于训练。

回想一下如图12.4.5所示的矢量化。处理器核心中添加向量处理单元可以显著提高吞吐量。例如，在图12.4.5的例子中，我们能够同时执行16个操作。首先，如果我们添加的运算不仅优化了向量运算，而且优化了矩阵运算，会有什么好处？稍后我们将讨论基于这个策略引入的张量核（tensor cores）。第二，如果我们增加更多的核心呢？简而言之，以上就是GPU设计决策中的两种策略。图12.4.7给出了基本处理块的概述。它包含16个整数单位和16个浮点单位。除此之外，两个张量核加速了与深度学习相关的附加操作的狭窄的子集。每个流式多处理器都由这样的四个块组成。

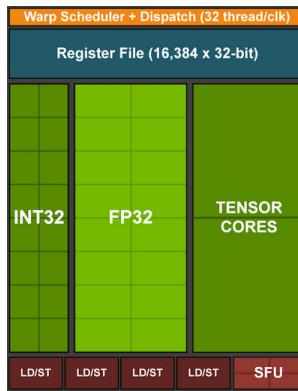


图12.4.7: NVIDIA Turing处理块（图片由英伟达提供）

接下来，将12个流式多处理器分组为图形处理集群，这些集群构成了高端TU102处理器。充足的内存通道和二级缓存完善了配置。图12.4.8有相关的细节。设计这种设备的原因之一是可以根据需要独立地添加或删除模块，从而满足设计更紧凑的芯片和处理良品率问题（故障模块可能无法激活）的需要。幸运的是，在CUDA和框架代码层之下，这类设备的编程对深度学习的临时研究员隐藏得很好。特别是，只要有可用的资源GPU上就可以同时执行多个程序。尽管如此，了解设备的局限性是值得的，以避免对应的设备内存的型号不合适。

¹⁵⁶ <https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/>



图12.4.8: NVIDIA Turing架构 (图片由英伟达提供)

最后值得一提的是张量核 (tensor core)。它们是最近增加更多优化电路趋势的一个例子，这些优化电路对深度学习特别有效。例如，TPU添加了用于快速矩阵乘法的脉动阵列 (Kung, 1988)，这种设计是为了支持非常小数量 (第一代TPU支持数量为1) 的大型操作。而张量核是另一个极端。它们针对 4×4 和 16×16 矩阵之间的小型运算进行了优化，具体取决于它们的数值精度。图12.4.9给出了优化的概述。

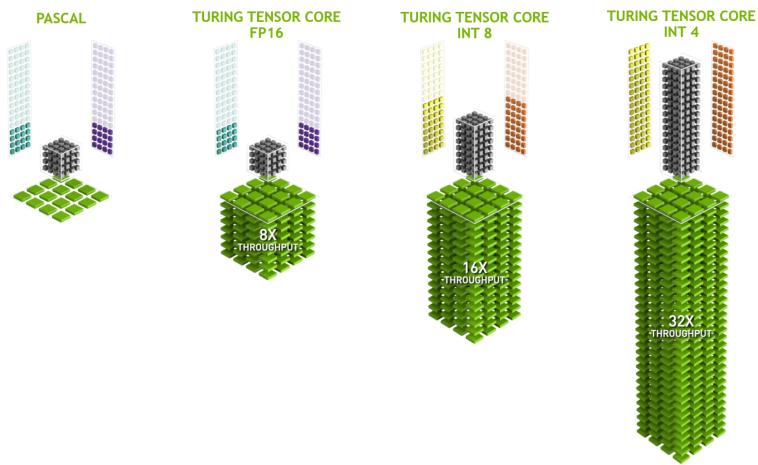


图12.4.9: NVIDIA Turing架构中的张量核心 (图片由英伟达提供)

显然，我们最终会在优化计算时做出某些妥协。其中之一是GPU不太擅长处理稀疏数据和中断。尽管有一些明显的例外，如Gunrock¹⁵⁷ (Wang et al., 2016)，但GPU擅长的高带宽突发读取操作并不适合稀疏的矩阵和向量的访问模式。访问稀疏数据和处理中断这两个目标是一个积极研究的领域。例如：DGL¹⁵⁸，一个专为图深

¹⁵⁷ <https://github.com/gunrock/gunrock>

¹⁵⁸ <http://dgl.ai>

度学习而设计的库。

12.4.6 网络和总线

每当单个设备不足以进行优化时，我们就需要来回传输数据以实现同步处理，于是网络和总线就派上了用场。我们有许多设计参数：带宽、成本、距离和灵活性。应用的末端有WiFi，它有非常好的使用范围，非常容易使用（毕竟没有线缆），而且还便宜，但它提供的带宽和延迟相对一般。头脑正常的机器学习研究人员都不会用它来构建服务器集群。接下来的内容中将重点关注适合深度学习的互连方式。

- **PCIe**, 一种专用总线，用于每个通道点到点连接的高带宽需求（在16通道插槽中的PCIe4.0上高达32GB/s），延迟时间为个位数的微秒（ $5\mu\text{s}$ ）。PCIe链接非常宝贵。处理器拥有的数量：AMD的EPYC 3有128个通道，Intel的Xeon每个芯片有48个通道；在桌面级CPU上，数字分别是20（Ryzen9）和16（Core i9）。由于GPU通常有16个通道，这就限制了以全带宽与CPU连接的GPU数量。毕竟，它们还需要与其他高带宽外围设备（如存储和以太网）共享链路。与RAM访问一样，由于减少了数据包的开销，因此更适合大批量数据传输。
- 以太网，连接计算机最常用的方式。虽然它比PCIe慢得多，但它的安装成本非常低，而且具有很强的弹性，覆盖的距离也要长得多。低级服务器的典型带宽为1Gbit/s。高端设备（如云中的C5实例¹⁵⁹）。这进一步增加了开销。与PCIe类似，以太网旨在连接两个设备，例如计算机和交换机。
- 交换机，一种连接多个设备的方式，该连接方式下的任何一对设备都可以同时执行（通常是全带宽）点对点连接。例如，以太网交换机可能以高带宽连接40台服务器。请注意，交换机并不是传统计算机网络所独有的。甚至PCIe通道也可以是可交换的¹⁶⁰，例如：P2实例¹⁶¹就是将大量GPU连接到主机处理器。
- **NVLink**，是PCIe的替代品，适用于非常高带宽的互连。它为每条链路提供高达300Gbit/s的数据传输速率。服务器GPU（Volta V100）有六个链路。而消费级GPU（RTX 2080Ti）只有一个链路，运行速度也降低到100Gbit/s。建议使用NCCL¹⁶²来实现GPU之间的高速数据传输。

12.4.7 更多延迟

表12.4.1和 表12.4.2中的小结来自Eliot Eshelman¹⁶³，他们将数字的更新版本保存到GitHub gist¹⁶⁴。

表12.4.1: 常见延迟。

Action	Time	Notes
L1 cache reference hit	1.5 ns	4 cycles
Floating-point add/mult/FMA	1.5 ns	4 cycles

continues on next page

¹⁵⁹ <https://aws.amazon.com/ec2/instance-types/c5/> 100Gbit/s
IP

¹⁶⁰ <https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches>

¹⁶¹ <https://aws.amazon.com/ec2/instance-types/p2/>

¹⁶² <https://github.com/NVIDIA/nccl>

¹⁶³ <https://gist.github.com/eshelman>

¹⁶⁴ <https://gist.github.com/eshelman/343a1c46cb3fba142c1afdcdeec17646>

表 12.4.1 – continued from previous page

Action	Time	Notes
L2 cache reference/hit	5 ns	12 ~ 17 cycles
Branch mispredict	6 ns	15 ~ 20 cycles
L3 cache hit (unshared cache)	16 ns	42 cycles
L3 cache hit (shared in another core)	25 ns	65 cycles
Mutex lock/unlock	25 ns	
L3 cache hit (modified in another core)	29 ns	75 cycles
L3 cache hit (on a remote CPU socket)	40 ns	100 ~ 300 cycles (40 ~ 116 ns)
QPI hop to another CPU (per hop)	40 ns	
64MB memory ref. (local CPU)	46 ns	TinyMemBench on Broadwell E5-2690v4
64MB memory ref. (remote CPU)	70 ns	TinyMemBench on Broadwell E5-2690v4
256MB memory ref. (local CPU)	75 ns	TinyMemBench on Broadwell E5-2690v4
Intel Optane random write	94 ns	UCSD Non-Volatile Systems Lab
256MB memory ref. (remote CPU)	120 ns	TinyMemBench on Broadwell E5-2690v4
Intel Optane random read	305 ns	UCSD Non-Volatile Systems Lab
Send 4KB over 100 Gbps HPC fabric	1 μ s	MVAPICH2 over Intel Omni-Path
Compress 1KB with Google Snappy	3 μ s	
Send 4KB over 10 Gbps ethernet	10 μ s	
Write 4KB randomly to NVMe SSD	30 μ s	DC P3608 NVMe SSD (QOS 99% is 500 μ s)
Transfer 1MB to/from NVLink GPU	30 μ s	~33GB/s on NVIDIA 40GB NVLink
Transfer 1MB to/from PCI-E GPU	80 μ s	~12GB/s on PCIe 3.0 x16 link
Read 4KB randomly from NVMe SSD	120 μ s	DC P3608 NVMe SSD (QOS 99%)
Read 1MB sequentially from NVMe SSD	208 μ s	~4.8GB/s DC P3608 NVMe SSD
Write 4KB randomly to SATA SSD	500 μ s	DC S3510 SATA SSD (QOS 99.9%)
Read 4KB randomly from SATA SSD	500 μ s	DC S3510 SATA SSD (QOS 99.9%)
Round trip within same datacenter	500 μ s	One-way ping is ~250 μ s
Read 1MB sequentially from SATA SSD	2 ms	~550MB/s DC S3510 SATA SSD
Read 1MB sequentially from disk	5 ms	~200MB/s server HDD
Random Disk Access (seek+rotation)	10 ms	
Send packet CA->Netherlands->CA	150 ms	

表12.4.2: NVIDIA Tesla GPU的延迟.

Action	Time	Notes
GPU Shared Memory access	30 ns	30~90 cycles (bank conflicts add latency)
GPU Global Memory access	200 ns	200~800 cycles
Launch CUDA kernel on GPU	10 μ s	Host CPU instructs GPU to start kernel
Transfer 1MB to/from NVLink GPU	30 μ s	~33GB/s on NVIDIA 40GB NVLink
Transfer 1MB to/from PCI-E GPU	80 μ s	~12GB/s on PCI-Express x16 link

小结

- 设备有运行开销。因此，数据传输要争取量大次少而不是量少次多。这适用于RAM、固态驱动器、网络和GPU。
- 矢量化是性能的关键。确保充分了解加速器的特定功能。例如，一些Intel Xeon CPU特别适用于INT8操作，NVIDIA Volta GPU擅长FP16矩阵操作，NVIDIA Turing擅长FP16、INT8和INT4操作。
- 在训练过程中数据类型过小导致的数值溢出可能是个问题（在推断过程中则影响不大）。
- 数据混叠现象会导致严重的性能退化。64位CPU应该按照64位边界进行内存对齐。在GPU上建议保持卷积大小对齐，例如：与张量核对齐。
- 将算法与硬件相匹配（例如，内存占用和带宽）。将命中参数装入缓存后，可以实现很大量级的加速比。
- 在验证实验结果之前，建议先在纸上勾勒出新算法的性能。关注的原因是数量级及以上的差异。
- 使用调试器跟踪调试寻找性能的瓶颈。
- 训练硬件和推断硬件在性能和价格方面有不同的优点。

练习

- 编写C语言来测试访问对齐的内存和未对齐的内存之间的速度是否有任何差异。（提示：小心缓存影响。）
- 测试按顺序访问或按给定步幅访问内存时的速度差异。
- 如何测量CPU上的缓存大小？
- 如何在多个内存通道中分配数据以获得最大带宽？如果有许多小的线程，会怎么布置？
- 一个企业级硬盘正在以10000转/分的速度旋转。在最坏的情况下，硬盘读取数据所需的最短时间是多少（假设磁头几乎是瞬间移动的）？为什么2.5英寸硬盘在商用服务器上越来越流行（相对于3.5英寸硬盘和5.25英寸硬盘）？
- 假设HDD制造商将存储密度从每平方英寸1 Tbit增加到每平方英寸5 Tbit。在一个2.5英寸的硬盘上，多少信息能够存储一个环中？内轨和外轨有区别吗？
- 从8位数据类型到16位数据类型，硅片的数量大约增加了四倍，为什么？为什么NVIDIA会在其图灵GPU中添加INT4运算？
- 在内存中向前读比向后读快多少？该数字在不同的计算机和CPU供应商之间是否有所不同？为什么？编写C代码进行实验。
- 磁盘的缓存大小能否测量？典型的硬盘是多少？固态驱动器需要缓存吗？
- 测量通过以太网发送消息时的数据包开销。查找UDP和TCP/IP连接之间的差异。
- 直接内存访问允许CPU以外的设备直接向内存写入（和读取）。为什么要这样？
- 看看Turing T4GPU的性能数字。为什么从FP16到INT8和INT4的性能只翻倍？

13. 一个网络包从旧金山到阿姆斯特丹的往返旅行需要多长时间？提示：可以假设距离为10000公里。

Discussions¹⁶⁵

12.5 多GPU训练

到目前为止，我们讨论了如何在CPU和GPU上高效地训练模型，同时在 12.3 节中展示了深度学习框架如何在CPU和GPU之间自动地并行化计算和通信，还在 5.6 节中展示了如何使用nvidia-smi命令列出计算机上所有可用的GPU。但是我们没有讨论如何真正实现深度学习训练的并行化。是否一种方法，以某种方式分割数据到多个设备上，并使其能够正常工作呢？本节将详细介绍如何从零开始并行地训练网络，这里需要运用小批量随机梯度下降算法（详见 11.5 节）。后面我还讲介绍如何使用高级API并行训练网络（请参阅 12.6 节）。

12.5.1 问题拆分

我们从一个简单的计算机视觉问题和一个稍稍过时的网络开始。这个网络有多个卷积层和汇聚层，最后可能有几个全连接的层，看起来非常类似于LeNet (LeCun *et al.*, 1998)或AlexNet (Krizhevsky *et al.*, 2012)。假设我们有多个GPU（如果是桌面服务器则有2个，AWS g4dn.12xlarge上有4个，p3.16xlarge上有8个，p2.16xlarge上有16个）。我们希望以一种方式对训练进行拆分，为实现良好的加速比，还能同时受益于简单且可重复的设计选择。毕竟，多个GPU同时增加了内存和计算能力。简而言之，对于需要分类的小批量训练数据，我们有以下选择。

第一种方法，在多个GPU之间拆分网络。也就是说，每个GPU将流入特定层的数据作为输入，跨多个后续层对数据进行处理，然后将数据发送到下一个GPU。与单个GPU所能处理的数据相比，我们可以用更大的网络处理数据。此外，每个GPU占用的显存（memory footprint）可以得到很好的控制，虽然它只是整个网络显存的一小部分。

然而，GPU的接口之间需要的密集同步可能是很难办的，特别是层之间计算的工作负载不能正确匹配的时候，还有层之间的接口需要大量的数据传输的时候（例如：激活值和梯度，数据量可能会超出GPU总线的带宽）。此外，计算密集型操作的顺序对拆分来说也是非常重要的，这方面的最好研究可参见 (Mirhoseini *et al.*, 2017)，其本质仍然是一个困难的问题，目前还不清楚研究是否能在特定问题上实现良好的线性缩放。综上所述，除非存框架或操作系统本身支持将多个GPU连接在一起，否则不建议这种方法。

第二种方法，拆分层内的工作。例如，将问题分散到4个GPU，每个GPU生成16个通道的数据，而不是在单个GPU上计算64个通道。对于全连接的层，同样可以拆分输出单元的数量。图12.5.1描述了这种设计，其策略用于处理显存非常小（当时为2GB）的GPU。当通道或单元的数量不太小时，使计算性能有良好的提升。此外，由于可用的显存呈线性扩展，多个GPU能够处理不断变大的网络。

¹⁶⁵ <https://discuss.d2l.ai/t/5717>

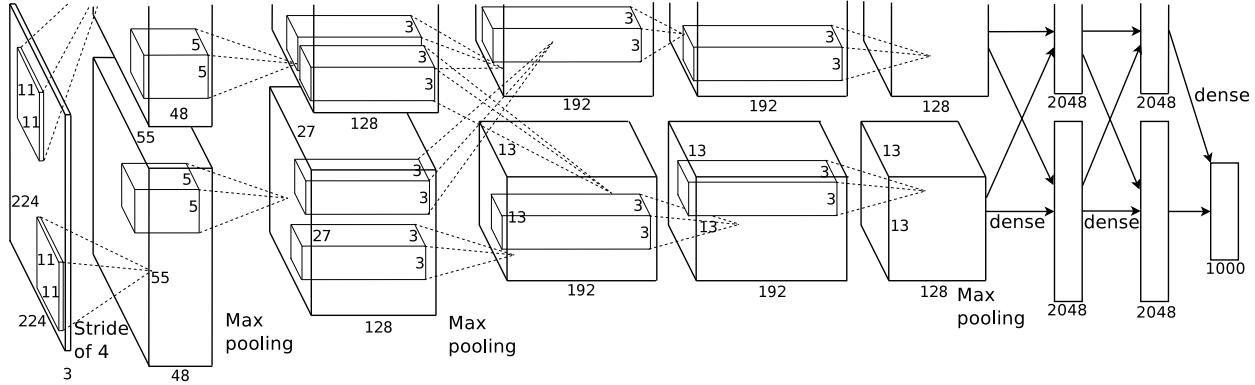


图12.5.1: 由于GPU显存有限, 原有AlexNet设计中的模型并行

然而, 我们需要大量的同步或屏障操作 (barrier operation), 因为每一层都依赖于所有其他层的结果。此外, 需要传输的数据量也可能比跨GPU拆分层时还要大。因此, 基于带宽的成本和复杂性, 我们同样不推荐这种方法。

最后一种方法, 跨多个GPU对数据进行拆分。这种方式下, 所有GPU尽管有不同的观测结果, 但是执行着相同类型的工作。在完成每个小批量数据的训练之后, 梯度在GPU上聚合。这种方法最简单, 并可以应用于任何情况, 同步只需要在每个小批量数据处理之后进行。也就是说, 当其他梯度参数仍在计算时, 完成计算的梯度参数就可以开始交换。而且, GPU的数量越多, 小批量包含的数据量就越大, 从而就能提高训练效率。但是, 添加更多的GPU并不能让我们训练更大的模型。

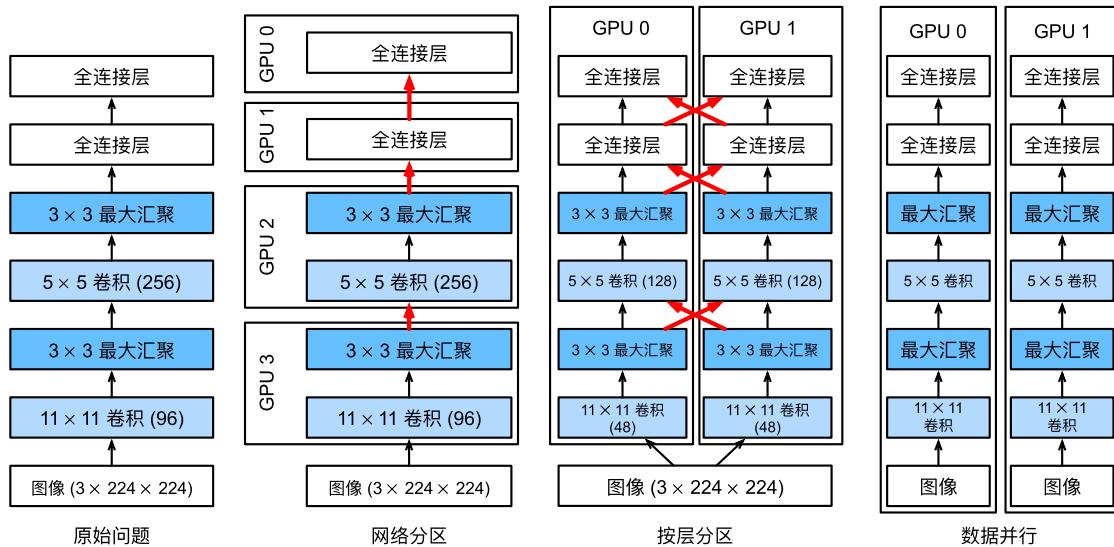


图12.5.2: 在多个GPU上并行化。从左到右: 原始问题、网络并行、分层并行、数据并行

图12.5.2中比较了多个GPU上不同的并行方式。总体而言, 只要GPU的显存足够大, 数据并行是最方便的。有关分布式训练分区的详细描述, 请参见 (Li et al., 2014)。在深度学习的早期, GPU的显存曾经是一个棘手的问题, 然而如今除了非常特殊的情况, 这个问题已经解决。下面我们将重点讨论数据并行性。

12.5.2 数据并行性

假设一台机器有 k 个GPU。给定需要训练的模型，虽然每个GPU上的参数值都是相同且同步的，但是每个GPU都将独立地维护一组完整的模型参数。例如，图12.5.3演示了在 $k = 2$ 时基于数据并行方法训练模型。

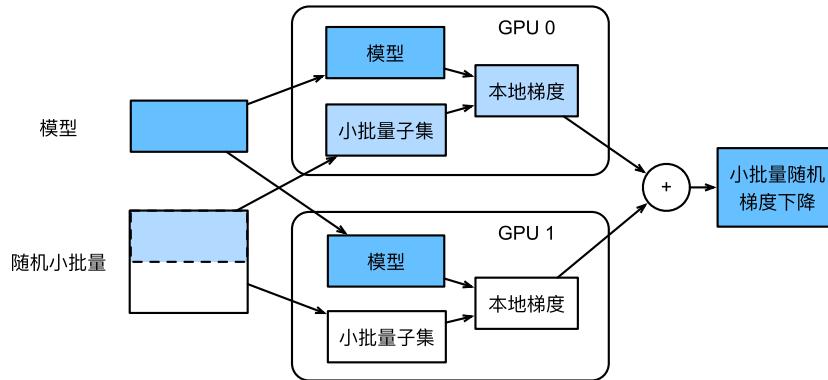


图12.5.3: 利用两个GPU上的数据，并行计算小批量随机梯度下降

一般来说， k 个GPU并行训练过程如下：

- 在任何一次训练迭代中，给定的随机的小批量样本都将被分成 k 个部分，并均匀地分配到GPU上；
- 每个GPU根据分配给它的小批量子集，计算模型参数的损失和梯度；
- 将 k 个GPU中的局部梯度聚合，以获得当前小批量的随机梯度；
- 聚合梯度被重新分发到每个GPU中；
- 每个GPU使用这个小批量随机梯度，来更新它所维护的完整的模型参数集。

在实践中请注意，当在 k 个GPU上训练时，需要扩大小批量的大小为 k 的倍数，这样每个GPU都有相同的工作量，就像只在单个GPU上训练一样。因此，在16-GPU服务器上可以显著地增加小批量数据量的大小，同时可能还需要相应地提高学习率。还请注意，7.5节中的批量规范化也需要调整，例如，为每个GPU保留单独的批量规范化参数。

下面我们将使用一个简单网络来演示多GPU训练。

```
%matplotlib inline
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

12.5.3 简单网络

我们使用 6.6 节中介绍的（稍加修改的）LeNet，从零开始定义它，从而详细说明参数交换和同步。

```
# 初始化模型参数
scale = 0.01
W1 = torch.randn(size=(20, 1, 3, 3)) * scale
b1 = torch.zeros(20)
W2 = torch.randn(size=(50, 20, 5, 5)) * scale
b2 = torch.zeros(50)
W3 = torch.randn(size=(800, 128)) * scale
b3 = torch.zeros(128)
W4 = torch.randn(size=(128, 10)) * scale
b4 = torch.zeros(10)
params = [W1, b1, W2, b2, W3, b3, W4, b4]

# 定义模型
def lenet(X, params):
    h1_conv = F.conv2d(input=X, weight=params[0], bias=params[1])
    h1_activation = F.relu(h1_conv)
    h1 = F.avg_pool2d(input=h1_activation, kernel_size=(2, 2), stride=(2, 2))
    h2_conv = F.conv2d(input=h1, weight=params[2], bias=params[3])
    h2_activation = F.relu(h2_conv)
    h2 = F.avg_pool2d(input=h2_activation, kernel_size=(2, 2), stride=(2, 2))
    h2 = h2.reshape(h2.shape[0], -1)
    h3_linear = torch.mm(h2, params[4]) + params[5]
    h3 = F.relu(h3_linear)
    y_hat = torch.mm(h3, params[6]) + params[7]
    return y_hat

# 交叉熵损失函数
loss = nn.CrossEntropyLoss(reduction='none')
```

12.5.4 数据同步

对于高效的多GPU训练，我们需要两个基本操作。首先，我们需要向多个设备分发参数并附加梯度(`get_params`)。如果没有参数，就不可能在GPU上评估网络。第二，需要跨多个设备对参数求和，也就是说，需要一个`allreduce`函数。

```
def get_params(params, device):
    new_params = [p.to(device) for p in params]
    for p in new_params:
```

(continues on next page)

(continued from previous page)

```
p.requires_grad_()
return new_params
```

通过将模型参数复制到一个GPU。

```
new_params = get_params(params, d2l.try_gpu(0))
print('b1 权重:', new_params[1])
print('b1 梯度:', new_params[1].grad)
```

```
b1 权重: tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
device='cuda:0', requires_grad=True)
b1 梯度: None
```

由于还没有进行任何计算，因此权重参数的梯度仍然为零。假设现在有一个向量分布在多个GPU上，下面的allreduce函数将所有向量相加，并将结果广播给所有GPU。请注意，我们需要将数据复制到累积结果的设备，才能使函数正常工作。

```
def allreduce(data):
    for i in range(1, len(data)):
        data[0][:] += data[i].to(data[0].device)
    for i in range(1, len(data)):
        data[i][:] = data[0].to(data[i].device)
```

通过在不同设备上创建具有不同值的向量并聚合它们。

```
data = [torch.ones((1, 2), device=d2l.try_gpu(i)) * (i + 1) for i in range(2)]
print('allreduce之前: \n', data[0], '\n', data[1])
allreduce(data)
print('allreduce之后: \n', data[0], '\n', data[1])
```

```
allreduce之前:
tensor([[1., 1.]], device='cuda:0')
tensor([[2., 2.]], device='cuda:1')
allreduce之后:
tensor([[3., 3.]], device='cuda:0')
tensor([[3., 3.]], device='cuda:1')
```

12.5.5 数据分发

我们需要一个简单的工具函数，将一个小批量数据均匀地分布在多个GPU上。例如，有两个GPU时，我们希望每个GPU可以复制一半的数据。因为深度学习框架的内置函数编写代码更方便、更简洁，所以在 4×5 矩阵上使用它进行尝试。

```
data = torch.arange(20).reshape(4, 5)
devices = [torch.device('cuda:0'), torch.device('cuda:1')]
split = nn.parallel.scatter(data, devices)
print('input :', data)
print('load into', devices)
print('output:', split)
```

```
input : tensor([[ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14],
               [15, 16, 17, 18, 19]])
load into [device(type='cuda', index=0), device(type='cuda', index=1)]
output: (tensor([[0, 1, 2, 3, 4],
                 [5, 6, 7, 8, 9]], device='cuda:0'), tensor([[10, 11, 12, 13, 14],
                 [15, 16, 17, 18, 19]], device='cuda:1'))
```

为了方便以后复用，我们定义了可以同时拆分数据和标签的`split_batch`函数。

```
#@save
def split_batch(X, y, devices):
    """将X和y拆分到多个设备上"""
    assert X.shape[0] == y.shape[0]
    return (nn.parallel.scatter(X, devices),
            nn.parallel.scatter(y, devices))
```

12.5.6 训练

现在我们可以在一个小批量上实现多GPU训练。在多个GPU之间同步数据将使用刚才讨论的辅助函数`allreduce`和`split_and_load`。我们不需要编写任何特定的代码来实现并行性。因为计算图在小批量内的设备之间没有任何依赖关系，因此它是“自动地”并行执行。

```
def train_batch(X, y, device_params, devices, lr):
    X_shards, y_shards = split_batch(X, y, devices)
    # 在每个GPU上分别计算损失
    ls = [loss(lenet(X_shard, device_W), y_shard).sum()
```

(continues on next page)

(continued from previous page)

```
for X_shard, y_shard, device_W in zip(
    X_shards, y_shards, device_params):
    for l in ls: # 反向传播在每个GPU上分别执行
        l.backward()
    # 将每个GPU的所有梯度相加，并将其广播到所有GPU
    with torch.no_grad():
        for i in range(len(device_params[0])):
            allreduce(
                [device_params[c][i].grad for c in range(len(devices))])
    # 在每个GPU上分别更新模型参数
    for param in device_params:
        d2l.sgd(param, lr, X.shape[0]) # 在这里，我们使用全尺寸的小批量
```

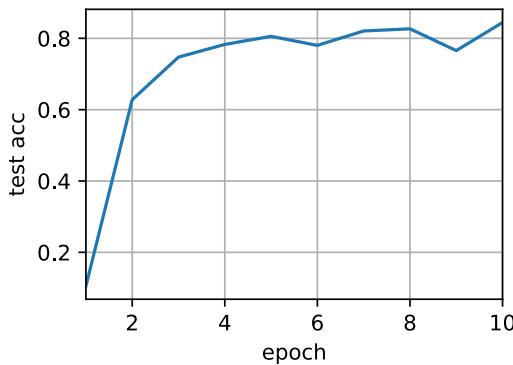
现在，我们可以定义训练函数。与前几章中略有不同：训练函数需要分配GPU并将所有模型参数复制到所有设备。显然，每个小批量都是使用train_batch函数来处理多个GPU。我们只在一个GPU上计算模型的精确度，而让其他GPU保持空闲，尽管这是相对低效的，但是使用方便且代码简洁。

```
def train(num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    devices = [d2l.try_gpu(i) for i in range(num_gpus)]
    # 将模型参数复制到num_gpus个GPU
    device_params = [get_params(params, d) for d in devices]
    num_epochs = 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    timer = d2l.Timer()
    for epoch in range(num_epochs):
        timer.start()
        for X, y in train_iter:
            # 为单个小批量执行多GPU训练
            train_batch(X, y, device_params, devices, lr)
            torch.cuda.synchronize()
        timer.stop()
        # 在GPU0上评估模型
        animator.add(epoch + 1, (d2l.evaluate_accuracy_gpu(
            lambda x: lenet(x, device_params[0]), test_iter, devices[0]),))
    print(f'测试精度: {animator.Y[0][-1]:.2f}, {timer.avg():.1f}秒/轮, '
          f'在{str(devices)})')
```

让我们看看在单个GPU上运行效果得有多好。首先使用的批量大小是256，学习率是0.2。

```
train(num_gpus=1, batch_size=256, lr=0.2)
```

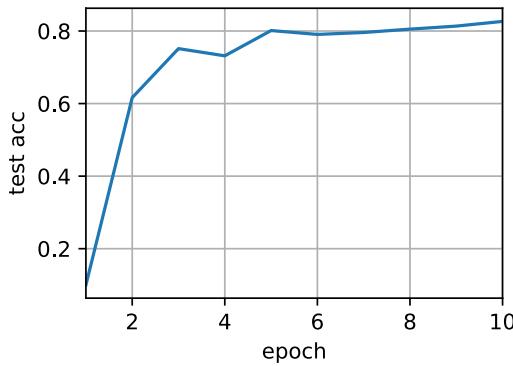
测试精度: 0.84, 2.7秒/轮, 在[device(type='cuda', index=0)]



保持批量大小和学习率不变, 并增加为2个GPU, 我们可以看到测试精度与之前的实验基本相同。不同的GPU个数在算法寻优方面是相同的。不幸的是, 这里没有任何有意义的加速: 模型实在太小了; 而且数据集也太小了。在这个数据集中, 我们实现的多GPU训练的简单方法受到了巨大的Python开销的影响。在未来, 我们将遇到更复杂的模型和更复杂的并行化方法。尽管如此, 让我们看看Fashion-MNIST数据集上会发生什么。

```
train(num_gpus=2, batch_size=256, lr=0.2)
```

测试精度: 0.83, 3.6秒/轮, 在[device(type='cuda', index=0), device(type='cuda', index=1)]



小结

- 有多种方法可以在多个GPU上拆分深度网络的训练。拆分可以在层之间、跨层或跨数据上实现。前两者需要对数据传输过程进行严格编排, 而最后一种则是最简单的策略。
- 数据并行训练本身是不复杂的, 它通过增加有效的小批量数据量的大小提高了训练效率。
- 在数据并行中, 数据需要跨多个GPU拆分, 其中每个GPU执行自己的前向传播和反向传播, 随后所有的梯度被聚合为一, 之后聚合结果向所有的GPU广播。

- 小批量数据量更大时，学习率也需要稍微提高一些。

练习

1. 在 k 个GPU上进行训练时，将批量大小从 b 更改为 $k \cdot b$ ，即按GPU的数量进行扩展。
2. 比较不同学习率时模型的精确度，随着GPU数量的增加学习率应该如何扩展？
3. 实现一个更高效的allreduce函数用于在不同的GPU上聚合不同的参数？为什么这样的效率更高？
4. 实现模型在多GPU下测试精度的计算。

Discussions¹⁶⁶

12.6 多GPU的简洁实现

每个新模型的并行计算都从零开始实现是无趣的。此外，优化同步工具以获得高性能也是有好处的。下面我们将展示如何使用深度学习框架的高级API来实现这一点。数学和算法与 12.5 节中的相同。本节的代码至少需要两个GPU来运行。

```
import torch
from torch import nn
from d2l import torch as d2l
```

12.6.1 简单网络

让我们使用一个比 12.5 节的LeNet更有意义的网络，它依然能够容易地和快速地训练。我们选择的是 (He et al., 2016) 中的ResNet-18。因为输入的图像很小，所以稍微修改了一下。与 7.6 节的区别在于，我们在开始时使用了更小的卷积核、步长和填充，而且删除了最大汇聚层。

```
#@save
def resnet18(num_classes, in_channels=1):
    """稍加修改的ResNet-18模型"""
    def resnet_block(in_channels, out_channels, num_residuals,
                    first_block=False):
        blk = []
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.append(d2l.Residual(in_channels, out_channels,
                                       use_1x1conv=True, strides=2))
            else:
                blk.append(d2l.Residual(out_channels, out_channels))
        return blk
    return nn.Sequential([
        nn.Conv2d(in_channels, 64, kernel_size=7, stride=2, padding=3),
        nn.BatchNorm2d(64), nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
        resnet_block(64, 64, 2, first_block=True),
        resnet_block(64, 128, 2),
        nn.BatchNorm2d(128), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
        resnet_block(128, 256, 2),
        nn.BatchNorm2d(256), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
        resnet_block(256, 512, 2),
        nn.BatchNorm2d(512), nn.ReLU(),
        nn.GlobalAvgPool2d(),
        nn.Flatten(),
        nn.Linear(512, num_classes)
    ])
```

(continues on next page)

¹⁶⁶ <https://discuss.d2l.ai/t/2800>

```

blk.append(d2l.Residual(out_channels, out_channels))
return nn.Sequential(*blk)

# 该模型使用了更小的卷积核、步长和填充，而且删除了最大汇聚层
net = nn.Sequential(
    nn.Conv2d(in_channels, 64, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(64),
    nn.ReLU())
net.add_module("resnet_block1", resnet_block(
    64, 64, 2, first_block=True))
net.add_module("resnet_block2", resnet_block(64, 128, 2))
net.add_module("resnet_block3", resnet_block(128, 256, 2))
net.add_module("resnet_block4", resnet_block(256, 512, 2))
net.add_module("global_avg_pool", nn.AdaptiveAvgPool2d((1, 1)))
net.add_module("fc", nn.Sequential(nn.Flatten(),
                                  nn.Linear(512, num_classes)))
return net

```

12.6.2 网络初始化

我们将在训练回路中初始化网络。请参见 4.8 节复习初始化方法。

```

net = resnet18(10)
# 获取GPU列表
devices = d2l.try_all_gpus()
# 我们将在训练代码实现中初始化网络

```

12.6.3 训练

如前所述，用于训练的代码需要执行几个基本功能才能实现高效并行：

- 需要在所有设备上初始化网络参数；
- 在数据集上迭代时，要将小批量数据分配到所有设备上；
- 跨设备并行计算损失及其梯度；
- 聚合梯度，并相应地更新参数。

最后，并行地计算精确度和发布网络的最终性能。除了需要拆分和聚合数据外，训练代码与前几章的实现非常相似。

```

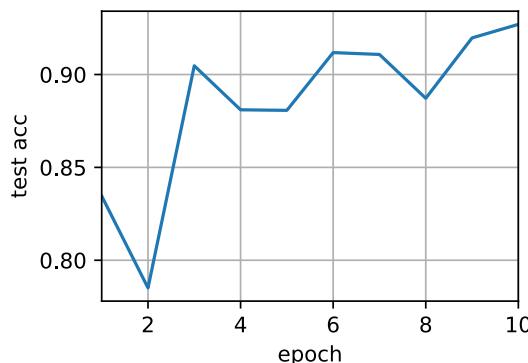
def train(net, num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    devices = [d2l.try_gpu(i) for i in range(num_gpus)]
    def init_weights(m):
        if type(m) in [nn.Linear, nn.Conv2d]:
            nn.init.normal_(m.weight, std=0.01)
    net.apply(init_weights)
    # 在多个GPU上设置模型
    net = nn.DataParallel(net, device_ids=devices)
    trainer = torch.optim.SGD(net.parameters(), lr)
    loss = nn.CrossEntropyLoss()
    timer, num_epochs = d2l.Timer(), 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    for epoch in range(num_epochs):
        net.train()
        timer.start()
        for X, y in train_iter:
            trainer.zero_grad()
            X, y = X.to(devices[0]), y.to(devices[0])
            l = loss(net(X), y)
            l.backward()
            trainer.step()
        timer.stop()
        animator.add(epoch + 1, (d2l.evaluate_accuracy_gpu(net, test_iter),))
    print(f'测试精度: {animator.Y[-1]:.2f}, {timer.avg():.1f}秒/轮, '
          f'在{str(devices)}')

```

接下来看看这在实践中是如何运作的。我们先在单个GPU上训练网络进行预热。

```
train(net, num_gpus=1, batch_size=256, lr=0.1)
```

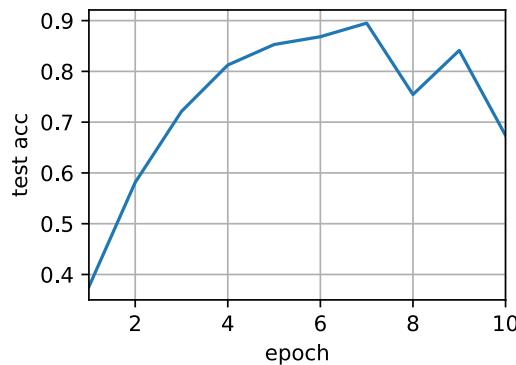
测试精度: 0.93, 12.2秒/轮, 在[device(type='cuda', index=0)]



接下来我们使用2个GPU进行训练。与 12.5节 中评估的LeNet相比，ResNet-18的模型要复杂得多。这就是显示并行化优势的地方，计算所需时间明显大于同步参数需要的时间。因为并行化开销的相关性较小，因此这种操作提高了模型的可伸缩性。

```
train(net, num_gpus=2, batch_size=512, lr=0.2)
```

测试精度：0.67，7.4秒/轮，在[device(type='cuda', index=0), device(type='cuda', index=1)]



小结

- 神经网络可以在（可找到数据的）单GPU上进行自动评估。
- 每台设备上的网络需要先初始化，然后再尝试访问该设备上的参数，否则会遇到错误。
- 优化算法在多个GPU上自动聚合。

练习

1. 本节使用ResNet-18，请尝试不同的迭代周期数、批量大小和学习率，以及使用更多的GPU进行计算。如果使用16个GPU（例如，在AWS p2.16xlarge实例上）尝试此操作，会发生什么？
2. 有时候不同的设备提供了不同的计算能力，我们可以同时使用GPU和CPU，那应该如何分配工作？为什么？

Discussions¹⁶⁷

¹⁶⁷ <https://discuss.d2l.ai/t/2803>

12.7 参数服务器

当我们从一个GPU迁移到多个GPU时，以及再迁移到包含多个GPU的多个服务器时（可能所有服务器的分布跨越了多个机架和多个网络交换机），分布式并行训练算法也需要变得更加复杂。通过细节可以知道，一方面是不同的互连方式的带宽存在极大的区别（例如，NVLink可以通过设置实现跨6条链路的高达100GB/s的带宽，16通道的PCIe4.0提供32GB/s的带宽，而即使是高速100GbE以太网也只能提供大约10GB/s的带宽）；另一方面是期望开发者既能完成统计学习建模还精通系统和网络也是不切实际的。

参数服务器的核心思想首先是由 (Smola and Narayananurthy, 2010) 在分布式隐变量模型的背景下引入的。然后，在 (Ahmed et al., 2012) 中描述了Push和Pull的语义，又在 (Li et al., 2014) 中描述了系统和开源库。下面，我们将介绍用于提高计算效率的组件。

12.7.1 数据并行训练

让我们回顾一下在分布式架构中数据并行的训练方法，因为在实践中它的实现相对简单，因此本节将排除其他内容只对其进行介绍。由于当今的GPU拥有大量的显存，因此在实际场景中（不包括图深度学习）只有数据并行这种并行训练策略值得推荐。图 图12.7.1 描述了在 12.5 节中实现的数据并行的变体。其中的关键是梯度的聚合需要在单个GPU（GPU 0）上完成，然后再将更新后的参数广播给所有GPU。

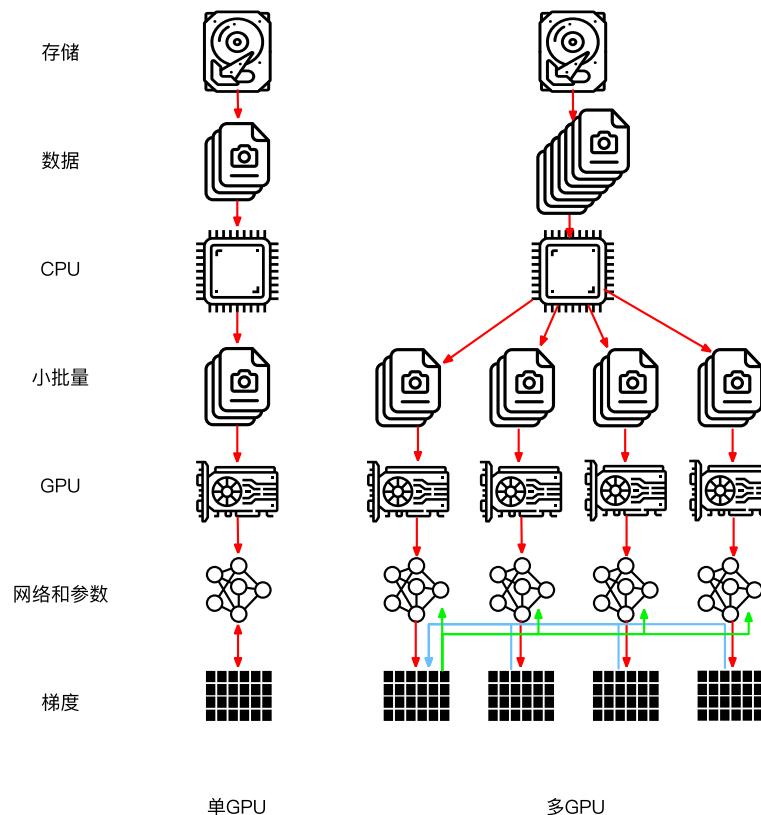


图12.7.1: 左图是单GPU训练；右图是多GPU训练的一个变体：(1) 计算损失和梯度，(2) 所有梯度聚合在一个GPU上，(3) 发生参数更新，并将参数重新广播给所有GPU

回顾来看，选择GPU 0进行聚合似乎是个很随便的决定，当然也可以选择CPU上聚合，事实上只要优化算法支持，在实际操作中甚至可以在某个GPU上聚合其中一些参数，而在另一个GPU上聚合另一些参数。例如，如果有四个与参数向量相关的梯度 $\mathbf{g}_1, \dots, \mathbf{g}_4$ ，还可以一个GPU对一个 $\mathbf{g}_i (i = 1, \dots, 4)$ 地进行梯度聚合。

这样的推断似乎是轻率和武断的，毕竟数学应该是逻辑自治的。但是，我们处理的是如 12.4 节中所述的真实的物理硬件，其中不同的总线具有不同的带宽。考虑一个如 12.4 节中所述的真实的4路GPU服务器。如果它的连接是特别完整的，那么可能拥有一个100GbE的网卡。更有代表性的数字是1-10GbE范围内，其有效带宽为100MB/s到1GB/s。因为CPU的PCIe通道太少（例如，消费级的Intel CPU有24个通道），所以无法直接与所有的GPU相连接，因此需要multiplexer¹⁶⁸。CPU在16x Gen3链路上的带宽为16GB/s，这也是每个GPU连接到交换机的速度，这意味着GPU设备之间的通信更有效。

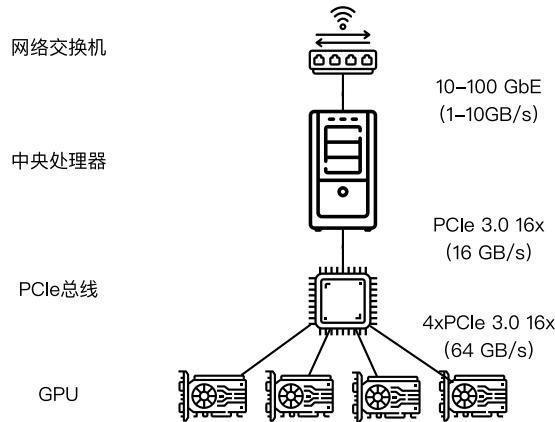


图12.7.2: 一个4路GPU服务器

为了便于讨论，我们假设所有梯度共需160MB。在这种情况下，将其中3个GPU的梯度发送到第4个GPU上需要30毫秒（每次传输需要10毫秒=160MB/16GB/s）。再加上30毫秒将权重向量传输回来，得到的结果是总共需要60毫秒。如果将所有的数据发送到CPU，总共需要80毫秒，其中将有40毫秒的惩罚，因为4个GPU都需要将数据发送到CPU。最后，假设能够将梯度分为4个部分，每个部分为40MB，现在可以在不同的GPU上同时聚合每个部分。因为PCIe交换机在所有链路之间提供全带宽操作，所以传输需要 $2.5 \times 3 = 7.5$ 毫秒，而不是30毫秒，因此同步操作总共需要15毫秒。简而言之，一样的参数同步操作基于不同的策略时间可能在15毫秒到80毫秒之间。图12.7.3描述了交换参数的不同策略。

¹⁶⁸ <https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches>

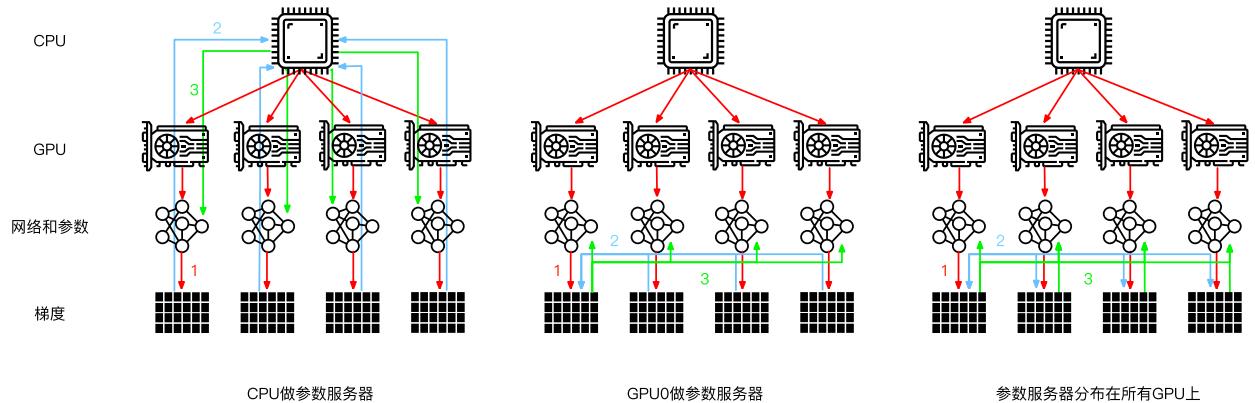


图12.7.3: 参数同步策略

请注意，我们还可以使用另一个工具来改善性能：在深度网络中，从顶部到底部计算所有梯度需要一些时间，因此即使还在忙着为某些参数计算梯度时，就可以开始为准备好的参数同步梯度了。想了解详细信息可以参见 (Sergeev and Del Balso, 2018)，想知道如何操作可参考Horovod¹⁶⁹。

12.7.2 环同步 (Ring Synchronization)

当谈及现代深度学习硬件的同步问题时，我们经常会遇到大量的定制的网络连接。例如，AWS p3.16xlarge和NVIDIA DGX-2实例中的连接都使用了 图12.7.4中的结构。每个GPU通过PCIe链路连接到主机CPU，该链路最多只能以16GB/s的速度运行。此外，每个GPU还具有6个NVLink连接，每个NVLink连接都能够以300Gbit/s进行双向传输。这相当于每个链路每个方向约 $300 \div 8 \div 2 \approx 18\text{GB/s}$ 。简言之，聚合的NVLink带宽明显高于PCIe带宽，问题是是如何有效地使用它。

¹⁶⁹ <https://github.com/horovod/horovod>

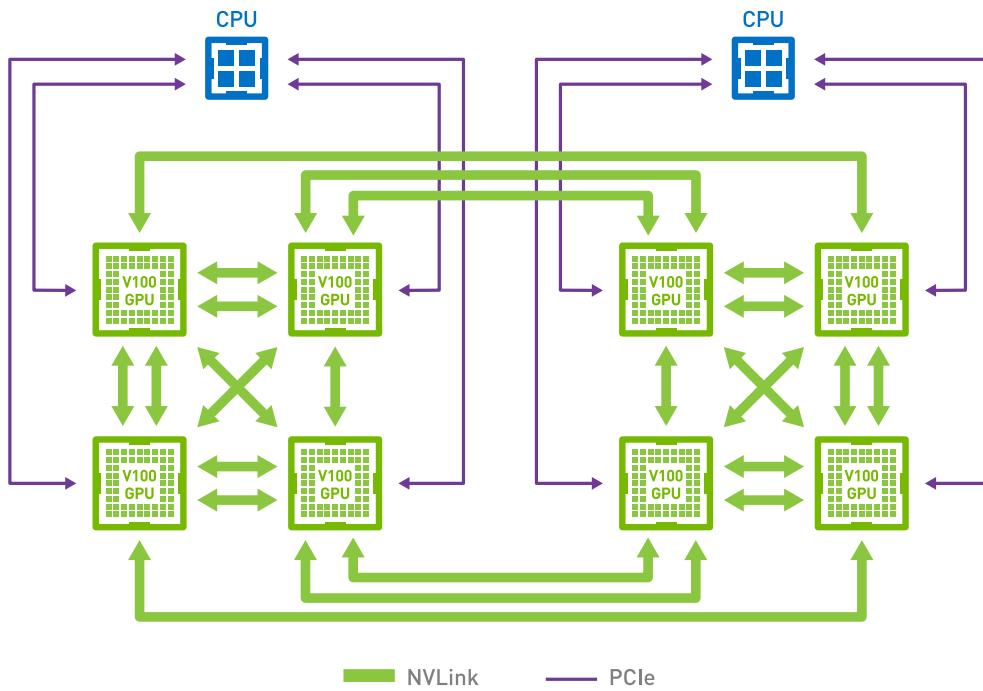


图12.7.4: 在8台V100 GPU服务器上连接NVLink (图片由英伟达提供)

(Wang et al., 2018)的研究结果表明最优的同步策略是将网络分解成两个环，并基于两个环直接同步数据。图12.7.5描述了网络可以分解为一个具有双NVLink带宽的环（1-2-3-4-5-6-7-8-1）和一个具有常规带宽的环（1-4-6-3-5-8-2-7-1）。在这种情况下，设计一个高效的同步协议是非常重要的。

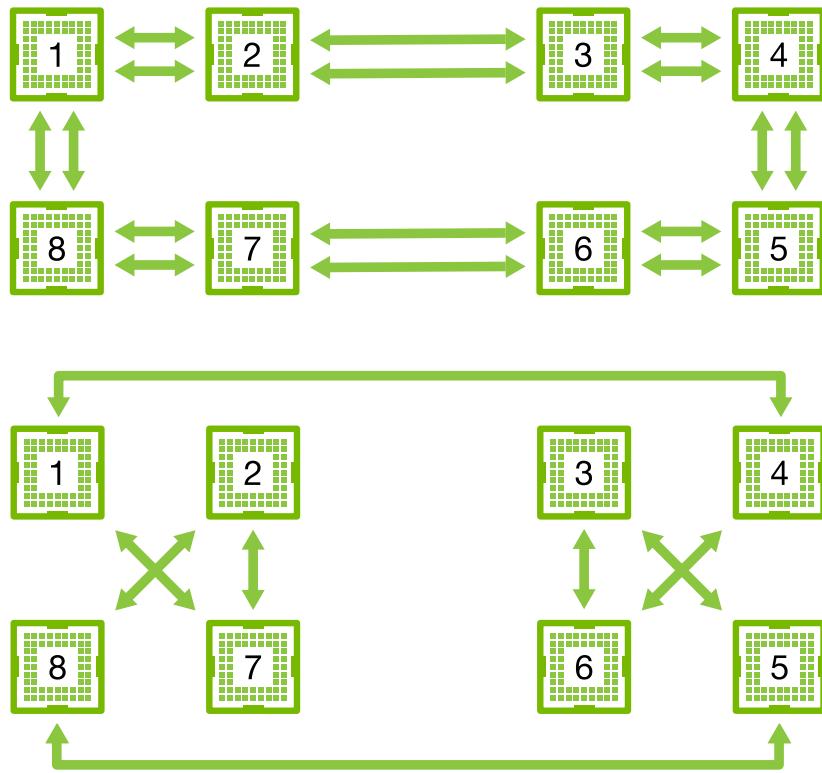


图12.7.5: 将NVLink网络分解为两个环。

考虑下面的思维试验: 给定由 n 个计算节点(或GPU)组成的一个环, 梯度可以从第一个节点发送到第二个节点, 在第二个结点将本地的梯度与传送的梯度相加并发送到第三个节点, 依此类推。在 $n - 1$ 步之后, 可以在最后访问的节点中找到聚合梯度。也就是说, 聚合梯度的时间随节点数线性增长。但如果照此操作, 算法是相当低效的。归根结底, 在任何时候都只有一个节点在通信。如果我们将梯度分为 n 个块, 并从节点 i 开始同步块 i , 会怎么样? 因为每个块的大小是 $1/n$, 所以总时间现在是 $(n - 1)/n \approx 1$ 。换句话说, 当我们增大环的大小时, 聚合梯度所花费的时间不会增加。这是一个相当惊人的结果。图12.7.6说明了 $n = 4$ 个节点上的步骤顺序。

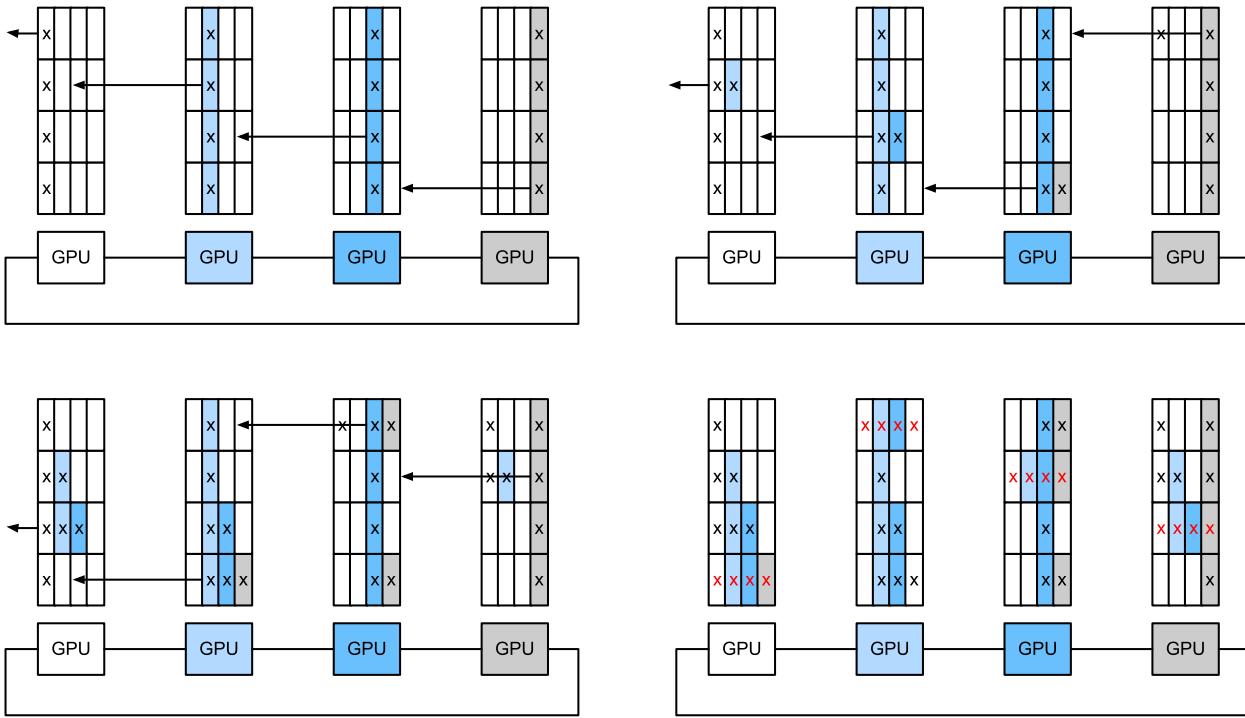


图12.7.6: 跨4个节点的环同步。每个节点开始向其左邻居发送部分梯度，直到在其右邻居中找到聚合的梯度

如果我们使用相同的例子，跨8个V100 GPU同步160MB，我们得到的结果大约是 $2 \times 160\text{MB} \div (3 \times 18\text{GB/s}) \approx 6\text{ms}$ 。这比使用PCIe总线要好，即使我们现在使用的是8个GPU。请注意，这些数字在实践中通常会差一些，因为深度学习框架无法将通信组合成大的突发传输。

注意到有一种常见的误解认为环同步与其他同步算法在本质上是不同的，实际上与简单的树算法相比其唯一的区别是同步路径稍微精细一些。

12.7.3 多机训练

新的挑战出现在多台机器上进行分布式训练：我们需要服务器之间相互通信，而这些服务器又只通过相对较低的带宽结构连接，在某些情况下这种连接的速度可能会慢一个数量级，因此跨设备同步是个棘手的问题。毕竟，在不同机器上运行训练代码的速度会有细微的差别，因此如果想使用分布式优化的同步算法就需要同步（synchronize）这些机器。[图12.7.7](#)说明了分布式并行训练是如何发生的。

1. 在每台机器上读取一组（不同的）批量数据，在多个GPU之间分割数据并传输到GPU的显存中。基于每个GPU上的批量数据分别计算预测和梯度。
2. 来自一台机器上的所有的本地GPU的梯度聚合在一个GPU上（或者在不同的GPU上聚合梯度的某些部分）。
3. 每台机器的梯度被发送到其本地CPU中。
4. 所有的CPU将梯度发送到中央参数服务器中，由该服务器聚合所有梯度。
5. 然后使用聚合后的梯度来更新参数，并将更新后的参数广播回各个CPU中。

6. 更新后的参数信息发送到本地一个（或多个）GPU中。

7. 所有GPU上的参数更新完成。

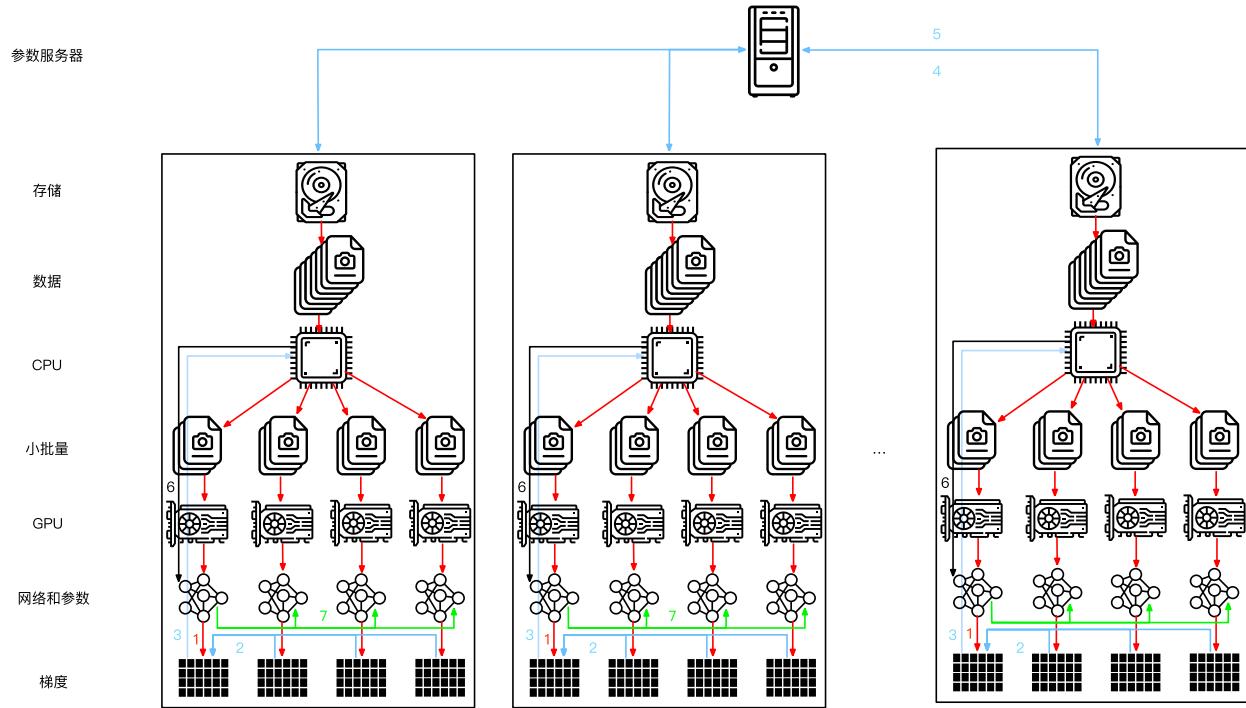


图12.7.7: 多机多GPU分布式并行训练

以上这些操作似乎都相当简单，而且事实上它们可以在一台机器内高效地执行，但是当我们考虑多台机器时，就会发现中央的参数服务器成为了瓶颈。毕竟，每个服务器的带宽是有限的，因此对 m 个工作节点来说，将所有梯度发送到服务器所需的时间是 $\mathcal{O}(m)$ 。我们也可以通过将参数服务器数量增加到 n 来突破这一障碍。此时，每个服务器只需要存储 $\mathcal{O}(1/n)$ 个参数，因此更新和优化的总时间变为 $\mathcal{O}(m/n)$ 。这两个数字的匹配会产生稳定的伸缩性，而不用在乎我们需要处理多少工作节点。在实际应用中，我们使用同一台机器既作为工作节点还作为服务器。设计说明请参考 图12.7.8（技术细节请参考 (Li et al., 2014))。特别是，确保多台机器只在没有不合理延迟的情况下工作是相当困难的。

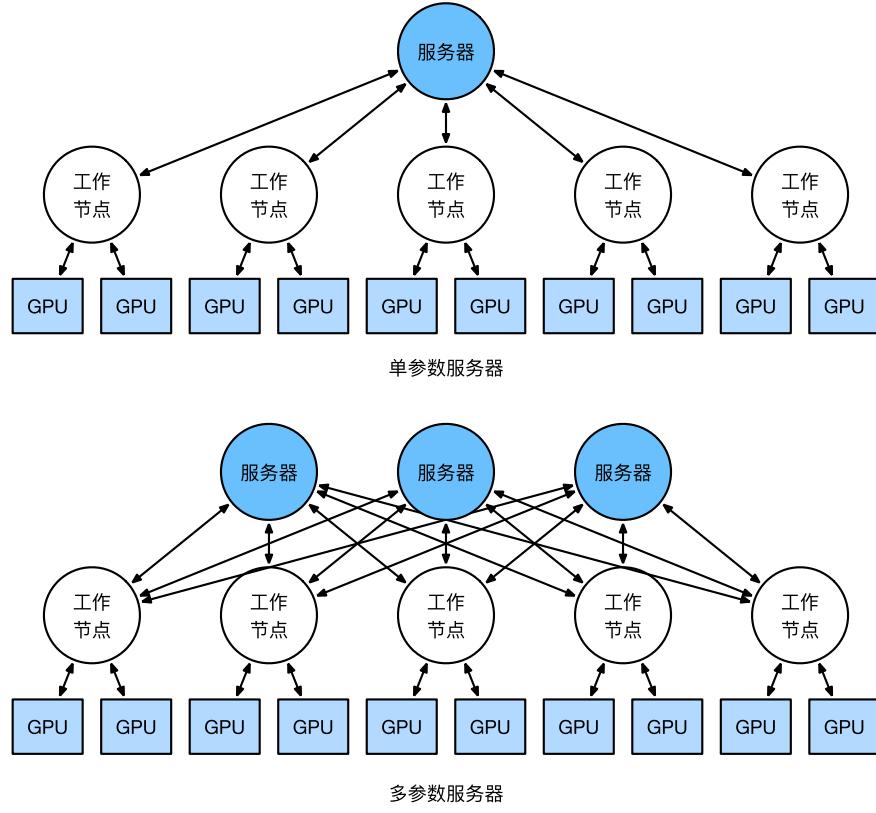


图12.7.8: 上图: 单参数服务器是一个瓶颈, 因为它的带宽是有限的; 下图: 多参数服务器使用聚合带宽存储部分参数

12.7.4 键值存储

在实践中, 实现分布式多GPU训练所需要的步骤绝非易事。这就是公共抽象值得使用的原因, 公共抽象即重新定义具有更新语义的键一值存储 (key-value store) 的抽象。

在许多工作节点和许多GPU中, 梯度 i 的计算可以定义为

$$\mathbf{g}_i = \sum_{k \in \text{workers}} \sum_{j \in \text{GPUs}} \mathbf{g}_{ijk}, \quad (12.7.1)$$

其中 \mathbf{g}_{ijk} 是在工作节点 k 的GPU j 上拆分的梯度 i 的一部分。这个运算的关键在于它是一个交换归约 (commutative reduction), 也就是说, 它把许多向量变换为一个向量, 而运算顺序在完成向量变换时并不重要。这对实现我们的目标来说是非常好的, 因为不需要为何时接收哪个梯度进行细粒度的控制。此外, 请注意, 这个操作在不同的 i 之间是独立的。

这就允许我们定义下面两个操作: *push* (用于累积梯度) 和 *pull* (用于取得聚合梯度)。因为我们有很多层, 也就有很多不同的梯度集合, 因此需要用一个键 i 来对梯度建索引。这个与Dynamo (DeCandia et al., 2007)中引入的键一值存储之间存在相似性并非巧合。它们两个定义都拥有许多相似的性质, 特别是在多个服务器之间分发参数时。

键一值存储的*push*与*pull*操作描述如下:

- **push (key, value)** 将特定的梯度值从工作节点发送到公共存储，在那里通过某种方式（例如，相加）来聚合值；
- **pull (key, value)** 从公共存储中取得某种方式（例如，组合来自所有工作节点的梯度）的聚合值。

通过将同步的所有复杂性隐藏在一个简单的push和pull操作背后，我们可以将统计建模人员（他们希望能够用简单的术语表达优化）和系统工程师（他们需要处理分布式同步中固有的复杂性）的关注点解耦。

小结

- 同步需要高度适应特定的网络基础设施和服务器内的连接，这种适应会严重影响同步所需的时间。
- 环同步对于p3和DGX-2服务器是最佳的，而对于其他服务器则未必。
- 当添加多个参数服务器以增加带宽时，分层同步策略可以工作的很好。

练习

1. 请尝试进一步提高环同步的性能吗。（提示：可以双向发送消息。）
2. 在计算仍在进行中，可否允许执行异步通信？它将如何影响性能？
3. 怎样处理在长时间运行的计算过程中丢失了一台服务器这种问题？尝试设计一种容错机制来避免重启计算这种解决方案？

Discussions¹⁷⁰

¹⁷⁰ <https://discuss.d2l.ai/t/5774>

近年来，深度学习一直是提高计算机视觉系统性能的变革力量。无论是医疗诊断、自动驾驶，还是智能滤波器、摄像头监控，许多计算机视觉领域的应用都与我们当前和未来的生活密切相关。可以说，最先进的计算机视觉应用与深度学习几乎是不可分割的。有鉴于此，本章将重点介绍计算机视觉领域，并探讨最近在学术界和行业中具有影响力的方法和应用。

在 6 节 和 7 节 中，我们研究了计算机视觉中常用的各种卷积神经网络，并将它们应用到简单的图像分类任务中。本章开头，我们将介绍两种可以改进模型泛化的方法，即图像增广和微调，并将它们应用于图像分类。由于深度神经网络可以有效地表示多个层次的图像，因此这种分层表示已成功用于各种计算机视觉任务，例如目标检测（object detection）、语义分割（semantic segmentation）和样式迁移（style transfer）。秉承计算机视觉中利用分层表示的关键思想，我们将从物体检测的主要组件和技术开始，继而展示如何使用完全卷积网络对图像进行语义分割，然后我们将解释如何使用样式迁移技术来生成像本书封面一样的图像。最后在结束本章时，我们将本章和前几章的知识应用于两个流行的计算机视觉基准数据集。

13.1 图像增广

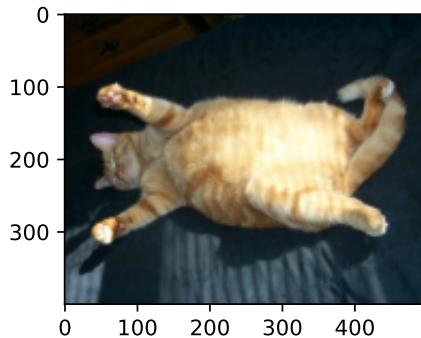
7.1 节 提到过大型数据集是成功应用深度神经网络的先决条件。图像增广在对训练图像进行一系列的随机变化之后，生成相似但不同的训练样本，从而扩大了训练集的规模。此外，应用图像增广的原因是，随机改变训练样本可以减少模型对某些属性的依赖，从而提高模型的泛化能力。例如，我们可以以不同的方式裁剪图像，使感兴趣的对象出现在不同的位置，减少模型对于对象出现位置的依赖。我们还可以调整亮度、颜色等因素来降低模型对颜色的敏感度。可以说，图像增广技术对于 AlexNet 的成功是必不可少的。本节将讨论这项广泛应用于计算机视觉的技术。

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

13.1.1 常用的图像增广方法

在对常用图像增广方法的探索时，我们将使用下面这个尺寸为 400×500 的图像作为示例。

```
d2l.set_figsize()
img = d2l.Image.open('../img/cat1.jpg')
d2l.plt.imshow(img);
```



大多数图像增广方法都具有一定的随机性。为了便于观察图像增广的效果，我们下面定义辅助函数apply。此函数在输入图像上多次运行图像增广方法aug并显示所有结果。

```
def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
    Y = [aug(img) for _ in range(num_rows * num_cols)]
    d2l.show_images(Y, num_rows, num_cols, scale=scale)
```

翻转和裁剪

左右翻转图像通常不会改变对象的类别。这是最早且最广泛使用的图像增广方法之一。接下来，我们使用transforms模块来创建RandomFlipLeftRight实例，这样就各有50%的几率使图像向左或向右翻转。

```
apply(img, torchvision.transforms.RandomHorizontalFlip())
```



上下翻转图像不如左右图像翻转那样常用。但是，至少对于这个示例图像，上下翻转不会妨碍识别。接下来，我们创建一个RandomFlipTopBottom实例，使图像各有50%的几率向上或向下翻转。

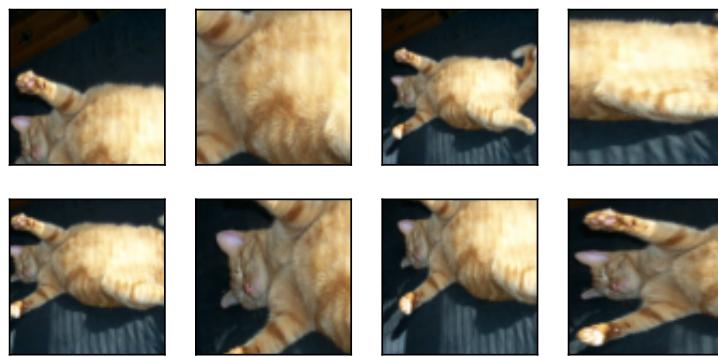
```
apply(img, torchvision.transforms.RandomVerticalFlip())
```



在我们使用的示例图像中，猫位于图像的中间，但并非所有图像都是这样。在[6.5节](#)中，我们解释了汇聚层可以降低卷积层对目标位置的敏感性。另外，我们可以通过对图像进行随机裁剪，使物体以不同的比例出现在图像的不同位置。这也降低模型对目标位置的敏感性。

下面的代码将随机裁剪一个面积为原始面积10%到100%的区域，该区域的宽高比从0.5~2之间随机取值。然后，区域的宽度和高度都被缩放到200像素。在本节中（除非另有说明）， a 和**b**之间的随机数指的是在区间 $[a, b]$ 中通过均匀采样获得的连续值。

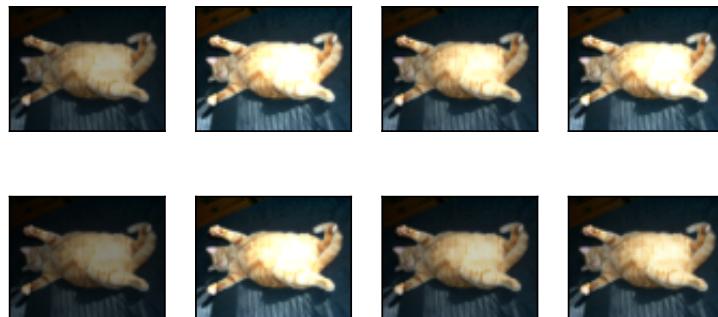
```
shape_aug = torchvision.transforms.RandomResizedCrop(  
    (200, 200), scale=(0.1, 1), ratio=(0.5, 2))  
apply(img, shape_aug)
```



改变颜色

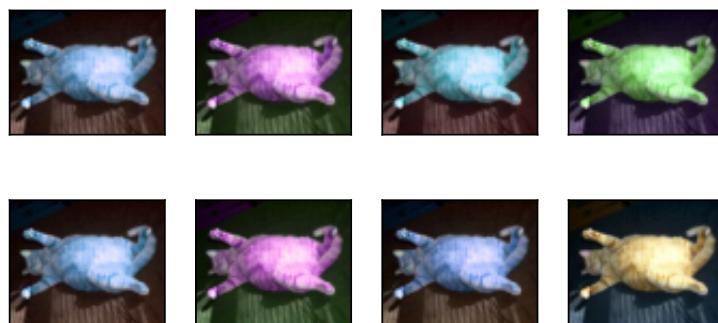
另一种增广方法是改变颜色。我们可以改变图像颜色的四个方面：亮度、对比度、饱和度和色调。在下面的示例中，我们随机更改图像的亮度，随机值为原始图像的50% ($1 - 0.5$) 到150% ($1 + 0.5$) 之间。

```
apply(img, torchvision.transforms.ColorJitter(  
    brightness=0.5, contrast=0, saturation=0, hue=0))
```



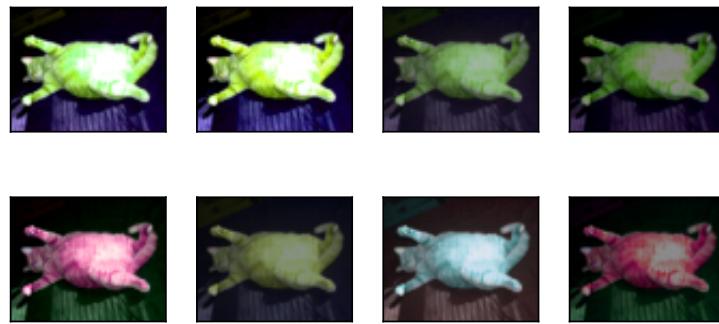
同样，我们可以随机更改图像的色调。

```
apply(img, torchvision.transforms.ColorJitter(  
    brightness=0, contrast=0, saturation=0, hue=0.5))
```



我们还可以创建一个RandomColorJitter实例，并设置如何同时随机更改图像的亮度（brightness）、对比度（contrast）、饱和度（saturation）和色调（hue）。

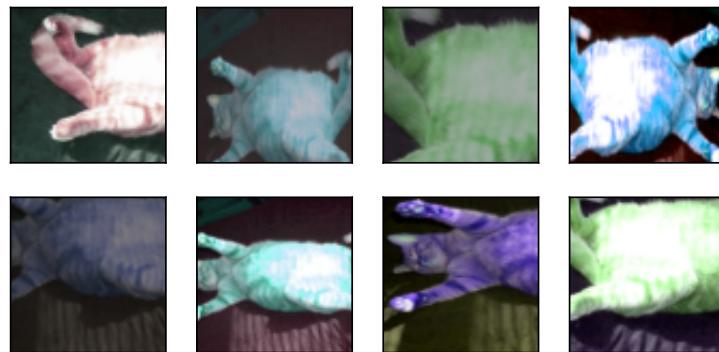
```
color_aug = torchvision.transforms.ColorJitter(  
    brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)  
apply(img, color_aug)
```



结合多种图像增广方法

在实践中，我们将结合多种图像增广方法。比如，我们可以通过使用一个Compose实例来综合上面定义的不同的图像增广方法，并将它们应用到每个图像。

```
augs = torchvision.transforms.Compose([  
    torchvision.transforms.RandomHorizontalFlip(), color_aug, shape_aug])  
apply(img, augs)
```



13.1.2 使用图像增广进行训练

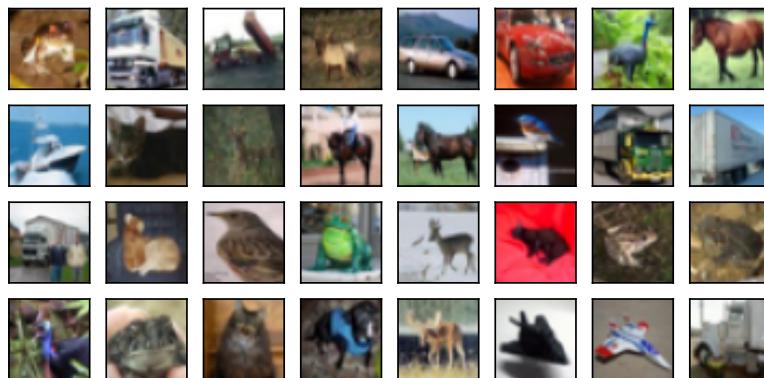
让我们使用图像增广来训练模型。这里，我们使用CIFAR-10数据集，而不是我们之前使用的Fashion-MNIST数据集。这是因为Fashion-MNIST数据集中对象的位置和大小已被规范化，而CIFAR-10数据集中对象的颜色和大小差异更明显。CIFAR-10数据集中的前32个训练图像如下所示。

```
all_images = torchvision.datasets.CIFAR10(train=True, root="../data",
                                         download=True)
d2l.show_images([all_images[i][0] for i in range(32)], 4, 8, scale=0.8);
```

Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ../data/cifar-10-python.tar.gz

0% | 0/170498071 [00:00<?, ?it/s]

Extracting ../data/cifar-10-python.tar.gz to ../data



为了在预测过程中得到确切的结果，我们通常对训练样本只进行图像增广，且在预测过程中不使用随机操作的图像增广。在这里，我们只使用最简单的随机左右翻转。此外，我们使用ToTensor实例将一批图像转换为深度学习框架所要求的格式，即形状为（批量大小，通道数，高度，宽度）的32位浮点数，取值范围为0~1。

```
train_augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor()])

test_augs = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor()])
```

接下来，我们定义一个辅助函数，以便于读取图像和应用图像增广。PyTorch数据集提供的transform参数应用图像增广来转化图像。有关DataLoader的详细介绍，请参阅 3.5节。

```

def load_cifar10(is_train, augs, batch_size):
    dataset = torchvision.datasets.CIFAR10(root="../data", train=is_train,
                                           transform=augs, download=True)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                             shuffle=is_train, num_workers=d2l.get_dataloader_workers())
    return dataloader

```

多GPU训练

我们在CIFAR-10数据集上训练 7.6节 中的ResNet-18模型。回想一下 12.6节 中对多GPU训练的介绍。接下来，我们定义一个函数，使用多GPU对模型进行训练和评估。

```

#@save
def train_batch_ch13(net, X, y, loss, trainer, devices):
    """用多GPU进行小批量训练"""
    if isinstance(X, list):
        # 微调BERT中所需
        X = [x.to(devices[0]) for x in X]
    else:
        X = X.to(devices[0])
    y = y.to(devices[0])
    net.train()
    trainer.zero_grad()
    pred = net(X)
    l = loss(pred, y)
    l.sum().backward()
    trainer.step()
    train_loss_sum = l.sum()
    train_acc_sum = d2l.accuracy(pred, y)
    return train_loss_sum, train_acc_sum

```

```

#@save
def train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
               devices=d2l.try_all_gpus()):
    """用多GPU进行模型训练"""
    timer, num_batches = d2l.Timer(), len(train_iter)
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 1],
                            legend=['train loss', 'train acc', 'test acc'])
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    for epoch in range(num_epochs):
        # 4个维度：储存训练损失，训练准确度，实例数，特点数
        metric = d2l.Accumulator(4)

```

(continues on next page)

(continued from previous page)

```
for i, (features, labels) in enumerate(train_iter):
    timer.start()
    l, acc = train_batch_ch13(
        net, features, labels, loss, trainer, devices)
    metric.add(l, acc, labels.shape[0], labels.numel())
    timer.stop()
    if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
        animator.add(epoch + (i + 1) / num_batches,
                      (metric[0] / metric[2], metric[1] / metric[3],
                       None))
    test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
    animator.add(epoch + 1, (None, None, test_acc))
print(f'loss {metric[0] / metric[2]:.3f}, train acc '
      f'{metric[1] / metric[3]:.3f}, test acc {test_acc:.3f}')
print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec on '
      f'{str(devices)})'
```

现在，我们可以定义train_with_data_aug函数，使用图像增广来训练模型。该函数获取所有的GPU，并使用Adam作为训练的优化算法，将图像增广应用于训练集，最后调用刚刚定义的用于训练和评估模型的train_ch13函数。

```
batch_size, devices, net = 256, d2l.try_all_gpus(), d2l.resnet18(10, 3)

def init_weights(m):
    if type(m) in [nn.Linear, nn.Conv2d]:
        nn.init.xavier_uniform_(m.weight)

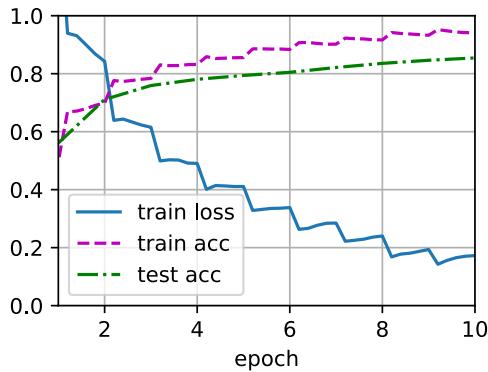
net.apply(init_weights)

def train_with_data_aug(train_augs, test_augs, net, lr=0.001):
    train_iter = load_cifar10(True, train_augs, batch_size)
    test_iter = load_cifar10(False, test_augs, batch_size)
    loss = nn.CrossEntropyLoss(reduction="none")
    trainer = torch.optim.Adam(net.parameters(), lr=lr)
    train_ch13(net, train_iter, test_iter, loss, trainer, 10, devices)
```

让我们使用基于随机左右翻转的图像增广来训练模型。

```
train_with_data_aug(train_augs, test_augs, net)
```

```
loss 0.173, train acc 0.941, test acc 0.854
4183.9 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



小结

- 图像增广基于现有的训练数据生成随机图像，来提高模型的泛化能力。
- 为了在预测过程中得到确切的结果，我们通常对训练样本只进行图像增广，而在预测过程中不使用带随机操作的图像增广。
- 深度学习框架提供了许多不同的图像增广方法，这些方法可以被同时应用。

练习

- 在不使用图像增广的情况下训练模型：`train_with_data_aug(no_aug, no_aug)`。比较使用和不使用图像增广的训练结果和测试精度。这个对比实验能支持图像增广可以减轻过拟合的论点吗？为什么？
- 在基于CIFAR-10数据集的模型训练中结合多种不同的图像增广方法。它能提高测试准确性吗？
- 参阅深度学习框架的在线文档。它还提供了哪些其他的图像增广方法？

Discussions¹⁷¹

13.2 微调

前面的一些章节介绍了如何在只有6万张图像的Fashion-MNIST训练数据集上训练模型。我们还描述了学术界当下使用最广泛的大规模图像数据集ImageNet，它有超过1000万的图像和1000类的物体。然而，我们平常接触到的数据集的规模通常在这两者之间。

假如我们想识别图片中不同类型的椅子，然后向用户推荐购买链接。一种可能的方法是首先识别100把普通椅子，为每把椅子拍摄1000张不同角度的图像，然后在收集的图像数据集上训练一个分类模型。尽管这个椅子数据集可能大于Fashion-MNIST数据集，但实例数量仍然不到ImageNet中的十分之一。适合ImageNet的复杂模型可能会在这个椅子数据集上过拟合。此外，由于训练样本数量有限，训练模型的准确性可能无法满足实际要求。

¹⁷¹ <https://discuss.d2l.ai/t/2829>

为了解决上述问题，一个显而易见的解决方案是收集更多的数据。但是，收集和标记数据可能需要大量的时间和金钱。例如，为了收集ImageNet数据集，研究人员花费了数百万美元的研究资金。尽管目前的数据收集成本已大幅降低，但这一成本仍不能忽视。

另一种解决方案是应用迁移学习（transfer learning）将从源数据集学到的知识迁移到目标数据集。例如，尽管ImageNet数据集中的大多数图像与椅子无关，但在此数据集上训练的模型可能会提取更通用的图像特征，这有助于识别边缘、纹理、形状和对象组合。这些类似的特征也可能有效地识别椅子。

13.2.1 步骤

本节将介绍迁移学习中的常见技巧：微调（fine-tuning）。如图13.2.1所示，微调包括以下四个步骤。

1. 在源数据集（例如ImageNet数据集）上预训练神经网络模型，即源模型。
2. 创建一个新的神经网络模型，即目标模型。这将复制源模型上的所有模型设计及其参数（输出层除外）。我们假定这些模型参数包含从源数据集中学到的知识，这些知识也将适用于目标数据集。我们还假定源模型的输出层与源数据集的标签密切相关；因此不在目标模型中使用该层。
3. 向目标模型添加输出层，其输出数是目标数据集中的类别数。然后随机初始化该层的模型参数。
4. 在目标数据集（如椅子数据集）上训练目标模型。输出层将从头开始进行训练，而所有其他层的参数将根据源模型的参数进行微调。

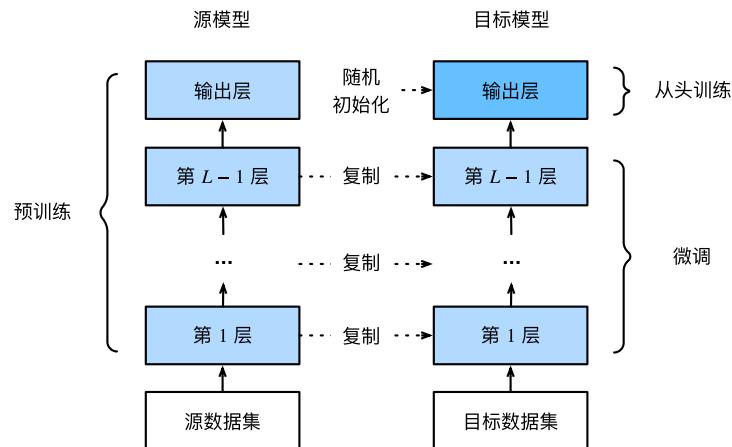


图13.2.1：微调。

当目标数据集比源数据集小得多时，微调有助于提高模型的泛化能力。

13.2.2 热狗识别

让我们通过具体案例演示微调：热狗识别。我们将在一个小型数据集上微调ResNet模型。该模型已在ImageNet数据集上进行了预训练。这个小型数据集包含数千张包含热狗和不包含热狗的图像，我们将使用微调模型来识别图像中是否包含热狗。

```
%matplotlib inline
import os
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

获取数据集

我们使用的热狗数据集来源于网络。该数据集包含1400张热狗的“正类”图像，以及包含尽可能多的其他食物的“负类”图像。含着两个类别的1000张图片用于训练，其余的则用于测试。

解压下载的数据集，我们获得了两个文件夹hotdog/train和hotdog/test。这两个文件夹都有hotdog（有热狗）和not-hotdog（无热狗）两个子文件夹，子文件夹内都包含相应类的图像。

```
#@save
d2l.DATA_HUB['hotdog'] = (d2l.DATA_URL + 'hotdog.zip',
                           'fba480ffa8aa7e0febbb511d181409f899b9baa5')

data_dir = d2l.download_extract('hotdog')
```

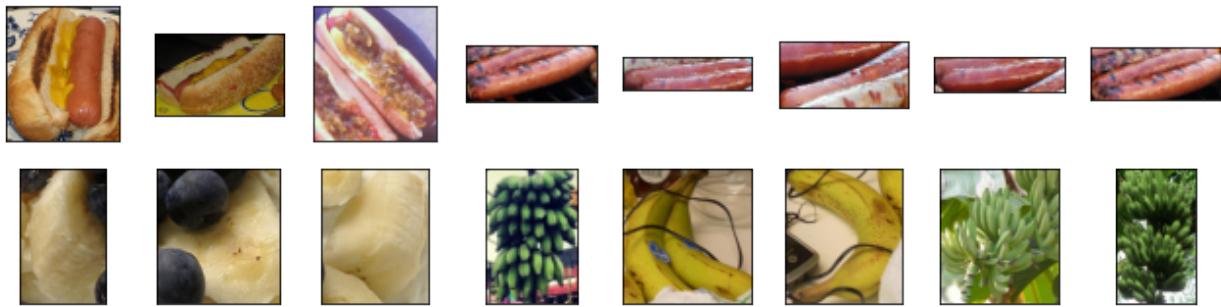
```
Downloading ../data/hotdog.zip from http://d2l-data.s3-accelerate.amazonaws.com/hotdog.zip...
```

我们创建两个实例来分别读取训练和测试数据集中的所有图像文件。

```
train_imgs = torchvision.datasets.ImageFolder(os.path.join(data_dir, 'train'))
test_imgs = torchvision.datasets.ImageFolder(os.path.join(data_dir, 'test'))
```

下面显示了前8个正类样本图片和最后8张负类样本图片。正如所看到的，图像的大小和纵横比各有不同。

```
hotdogs = [train_imgs[i][0] for i in range(8)]
not_hotdogs = [train_imgs[-i - 1][0] for i in range(8)]
d2l.show_images(hotdogs + not_hotdogs, 2, 8, scale=1.4);
```



在训练期间，我们首先从图像中裁切随机大小和随机长宽比的区域，然后将该区域缩放为 224×224 输入图像。在测试过程中，我们将图像的高度和宽度都缩放到256像素，然后裁剪中央 224×224 区域作为输入。此外，对于RGB（红、绿和蓝）颜色通道，我们分别标准化每个通道。具体而言，该通道的每个值减去该通道的平均值，然后将结果除以该通道的标准差。

```
# 使用RGB通道的均值和标准差，以标准化每个通道
normalize = torchvision.transforms.Normalize(
    [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomResizedCrop(224),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    normalize])

test_augs = torchvision.transforms.Compose([
    torchvision.transforms.Resize([256, 256]),
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    normalize])
```

定义和初始化模型

我们使用在ImageNet数据集上预训练的ResNet-18作为源模型。在这里，我们指定`pretrained=True`以自动下载预训练的模型参数。如果首次使用此模型，则需要连接互联网才能下载。

```
pretrained_net = torchvision.models.resnet18(pretrained=True)
```

预训练的源模型实例包含许多特征层和一个输出层`fc`。此划分的主要目的是促进对除输出层以外所有层的模型参数进行微调。下面给出了源模型的成员变量`fc`。

```
pretrained_net.fc
```

```
Linear(in_features=512, out_features=1000, bias=True)
```

在ResNet的全局平均汇聚层后，全连接层转换为ImageNet数据集的1000个类输出。之后，我们构建一个新的神经网络作为目标模型。它的定义方式与预训练源模型的定义方式相同，只是最终层中的输出数量被设置为目标数据集中的类数（而不是1000个）。

在下面的代码中，目标模型finetune_net中成员变量features的参数被初始化为源模型相应层的模型参数。由于模型参数是在ImageNet数据集上预训练的，并且足够好，因此通常只需要较小的学习率即可微调这些参数。

成员变量output的参数是随机初始化的，通常需要更高的学习率才能从头开始训练。假设Trainer实例中的学习率为 η ，我们将成员变量output中参数的学习率设置为 10η 。

```
finetune_net = torchvision.models.resnet18(pretrained=True)
finetune_net.fc = nn.Linear(finetune_net.fc.in_features, 2)
nn.init.xavier_uniform_(finetune_net.fc.weight);
```

微调模型

首先，我们定义了一个训练函数train_fine_tuning，该函数使用微调，因此可以多次调用。

```
# 如果param_group=True，输出层中的模型参数将使用十倍的学习率
def train_fine_tuning(net, learning_rate, batch_size=128, num_epochs=5,
                      param_group=True):
    train_iter = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(
        os.path.join(data_dir, 'train'), transform=train_augs),
        batch_size=batch_size, shuffle=True)
    test_iter = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(
        os.path.join(data_dir, 'test'), transform=test_augs),
        batch_size=batch_size)
    devices = d2l.try_all_gpus()
    loss = nn.CrossEntropyLoss(reduction="none")
    if param_group:
        params_1x = [param for name, param in net.named_parameters()
                     if name not in ["fc.weight", "fc.bias"]]
        trainer = torch.optim.SGD([{'params': params_1x,
                                    'params': net.fc.parameters(),
                                    'lr': learning_rate * 10}],
                                  lr=learning_rate, weight_decay=0.001)
    else:
        trainer = torch.optim.SGD(net.parameters(), lr=learning_rate,
                                 weight_decay=0.001)
```

(continues on next page)

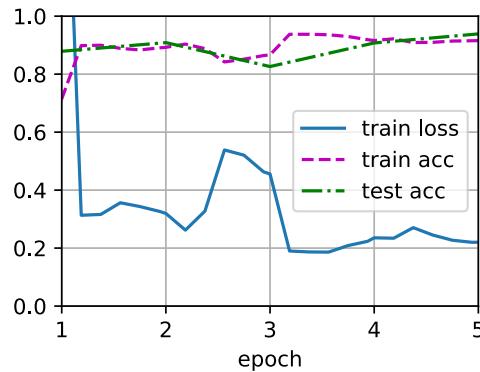
(continued from previous page)

```
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
                devices)
```

我们使用较小的学习率，通过微调预训练获得的模型参数。

```
train_fine_tuning(finetune_net, 5e-5)
```

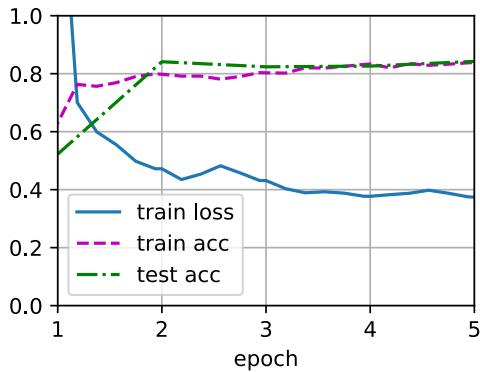
```
loss 0.220, train acc 0.915, test acc 0.939
999.1 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



为了进行比较，我们定义了一个相同的模型，但是将其所有模型参数初始化为随机值。由于整个模型需要从头开始训练，因此我们需要使用更大的学习率。

```
scratch_net = torchvision.models.resnet18()
scratch_net.fc = nn.Linear(scratch_net.fc.in_features, 2)
train_fine_tuning(scratch_net, 5e-4, param_group=False)
```

```
loss 0.374, train acc 0.839, test acc 0.843
1623.8 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



意料之中，微调模型往往表现更好，因为它的初始参数值更有效。

小结

- 迁移学习将从源数据集中学到的知识迁移到目标数据集，微调是迁移学习的常见技巧。
- 除输出层外，目标模型从源模型中复制所有模型设计及其参数，并根据目标数据集对这些参数进行微调。但是，目标模型的输出层需要从头开始训练。
- 通常，微调参数使用较小的学习率，而从头开始训练输出层可以使用更大的学习率。

练习

- 继续提高finetune_net的学习率，模型的准确性如何变化？
- 在比较实验中进一步调整finetune_net和scratch_net的超参数。它们的准确性还有不同吗？
- 将输出层finetune_net之前的参数设置为源模型的参数，在训练期间不要更新它们。模型的准确性如何变化？提示：可以使用以下代码。

```
for param in finetune_net.parameters():
    param.requires_grad = False
```

- 事实上，ImageNet数据集中有一个“热狗”类别。我们可以通过以下代码获取其输出层中的相应权重参数，但是我们怎样才能利用这个权重参数？

```
weight = pretrained_net.fc.weight
hotdog_w = torch.split(weight.data, 1, dim=0)[934]
hotdog_w.shape
```

```
torch.Size([1, 512])
```

13.3 目标检测和边界框

前面的章节（例如 7.1 节—7.4 节）介绍了各种图像分类模型。在图像分类任务中，我们假设图像中只有一个主要物体对象，我们只关注如何识别其类别。然而，很多时候图像里有多个我们感兴趣的目标，我们不仅想知道它们的类别，还想得到它们在图像中的具体位置。在计算机视觉里，我们将这类任务称为目标检测（object detection）或目标识别（object recognition）。

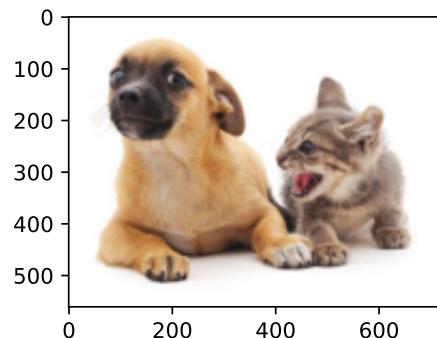
目标检测在多个领域中被广泛使用。例如，在无人驾驶里，我们需要通过识别拍摄到的视频图像里的车辆、行人、道路和障碍物的位置来规划行进线路。机器人也常通过该任务来检测感兴趣的目标。安防领域则需要检测异常目标，如歹徒或者炸弹。

接下来的几节将介绍几种用于目标检测的深度学习方法。我们将首先介绍目标的位置。

```
%matplotlib inline
import torch
from d2l import torch as d2l
```

下面加载本节将使用的示例图像。可以看到图像左边是一只狗，右边是一只猫。它们是这张图像里的两个主要目标。

```
d2l.set_figsize()
img = d2l.plt.imread('../img/catdog.jpg')
d2l.plt.imshow(img);
```



¹⁷² <https://discuss.d2l.ai/t/2894>

13.3.1 边界框

在目标检测中，我们通常使用边界框（bounding box）来描述对象的空间位置。边界框是矩形的，由矩形左上角的以及右下角的 x 和 y 坐标决定。另一种常用的边界框表示方法是边界框中心的 (x, y) 轴坐标以及框的宽度和高度。

在这里，我们定义在这两种表示法之间进行转换的函数：`box_corner_to_center`从两角表示法转换为中心宽度表示法，而`box_center_to_corner`反之亦然。输入参数`boxes`可以是长度为4的张量，也可以是形状为 $(n, 4)$ 的二维张量，其中 n 是边界框的数量。

```
#@save
def box_corner_to_center(boxes):
    """从 (左上, 右下) 转换到 (中间, 宽度, 高度)
    x1, y1, x2, y2 = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2
    w = x2 - x1
    h = y2 - y1
    boxes = torch.stack((cx, cy, w, h), axis=-1)
    return boxes

#@save
def box_center_to_corner(boxes):
    """从 (中间, 宽度, 高度) 转换到 (左上, 右下)
    cx, cy, w, h = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    x1 = cx - 0.5 * w
    y1 = cy - 0.5 * h
    x2 = cx + 0.5 * w
    y2 = cy + 0.5 * h
    boxes = torch.stack((x1, y1, x2, y2), axis=-1)
    return boxes
```

我们将根据坐标信息定义图像中狗和猫的边界框。图像中坐标的原点是图像的左上角，向右的方向为 x 轴的正方向，向下的方向为 y 轴的正方向。

```
# bbox是边界框的英文缩写
dog_bbox, cat_bbox = [60.0, 45.0, 378.0, 516.0], [400.0, 112.0, 655.0, 493.0]
```

我们可以通过转换两次来验证边界框转换函数的正确性。

```
boxes = torch.tensor((dog_bbox, cat_bbox))
box_center_to_corner(box_corner_to_center(boxes)) == boxes
```

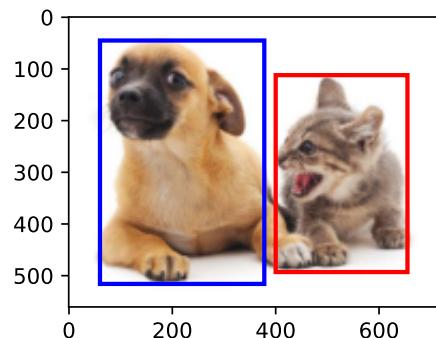
```
tensor([[True, True, True, True],  
       [True, True, True, True]])
```

我们可以将边界框在图中画出，以检查其是否准确。画之前，我们定义一个辅助函数bbox_to_rect。它将边界框表示成matplotlib的边界框格式。

```
#@save  
def bbox_to_rect(bbox, color):  
    # 将边界框(左上x,左上y,右下x,右下y)格式转换成matplotlib格式:  
    # ((左上x,左上y),宽,高)  
    return d2l.plt.Rectangle(  
        xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],  
        fill=False, edgecolor=color, linewidth=2)
```

在图像上添加边界框之后，我们可以看到两个物体的主要轮廓基本上在两个框内。

```
fig = d2l.plt.imshow(img)  
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))  
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
```



小结

- 目标检测不仅可以识别图像中所有感兴趣的物体，还能识别它们的位置，该位置通常由矩形边界框表示。
- 我们可以在两种常用的边界框表示（中间，宽度，高度）和（左上，右下）坐标之间进行转换。

练习

1. 找到另一张图像，然后尝试标记包含该对象的边界框。比较标注边界框和标注类别哪个需要更长的时间？
2. 为什么`box_corner_to_center`和`box_center_to_corner`的输入参数的最内层维度总是4？

Discussions¹⁷³

13.4 锚框

目标检测算法通常会在输入图像中采样大量的区域，然后判断这些区域中是否包含我们感兴趣的目标，并调整区域边界从而更准确地预测目标的真实边界框（ground-truth bounding box）。不同的模型使用的区域采样方法可能不同。这里我们介绍其中的一种方法：以每个像素为中心，生成多个缩放比和宽高比（aspect ratio）不同的边界框。这些边界框被称为锚框（anchor box）。我们将在 13.7 节中设计一个基于锚框的目标检测模型。

首先，让我们修改输出精度，以获得更简洁的输出。

```
%matplotlib inline
import torch
from d2l import torch as d2l

torch.set_printoptions(2) # 精简输出精度
```

13.4.1 生成多个锚框

假设输入图像的高度为 h ，宽度为 w 。我们以图像的每个像素为中心生成不同形状的锚框：缩放比为 $s \in (0, 1]$ ，宽高比为 $r > 0$ 。那么锚框的宽度和高度分别是 $hs\sqrt{r}$ 和 hs/\sqrt{r} 。请注意，当中心位置给定时，已知宽和高的锚框是确定的。

要生成多个不同形状的锚框，让我们设置许多缩放比（scale）取值 s_1, \dots, s_n 和许多宽高比（aspect ratio）取值 r_1, \dots, r_m 。当使用这些比例和长宽比的所有组合以每个像素为中心时，输入图像将总共有 $whnm$ 个锚框。尽管这些锚框可能会覆盖所有真实边界框，但计算复杂性很容易过高。在实践中，我们只考虑包含 s_1 或 r_1 的组合：

$$(s_1, r_1), (s_1, r_2), \dots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \dots, (s_n, r_1). \quad (13.4.1)$$

也就是说，以同一像素为中心的锚框的数量是 $n + m - 1$ 。对于整个输入图像，将共生成 $wh(n + m - 1)$ 个锚框。

上述生成锚框的方法在下面的`multibox_prior`函数中实现。我们指定输入图像、尺寸列表和宽高比列表，然后此函数将返回所有的锚框。

¹⁷³ <https://discuss.d2l.ai/t/2944>

```

#@save
def multibox_prior(data, sizes, ratios):
    """生成以每个像素为中心具有不同形状的锚框"""
    in_height, in_width = data.shape[-2:]
    device, num_sizes, num_ratios = data.device, len(sizes), len(ratios)
    boxes_per_pixel = (num_sizes + num_ratios - 1)
    size_tensor = torch.tensor(sizes, device=device)
    ratio_tensor = torch.tensor(ratios, device=device)

    # 为了将锚点移动到像素的中心，需要设置偏移量。
    # 因为一个像素的高为1且宽为1，我们选择偏移我们的中心0.5
    offset_h, offset_w = 0.5, 0.5
    steps_h = 1.0 / in_height # 在y轴上缩放步长
    steps_w = 1.0 / in_width # 在x轴上缩放步长

    # 生成锚框的所有中心点
    center_h = (torch.arange(in_height, device=device) + offset_h) * steps_h
    center_w = (torch.arange(in_width, device=device) + offset_w) * steps_w
    shift_y, shift_x = torch.meshgrid(center_h, center_w, indexing='ij')
    shift_y, shift_x = shift_y.reshape(-1), shift_x.reshape(-1)

    # 生成“boxes_per_pixel”个高和宽，
    # 之后用于创建锚框的四角坐标(xmin,xmax,ymin,ymax)
    w = torch.cat((size_tensor * torch.sqrt(ratio_tensor[0]),
                   sizes[0] * torch.sqrt(ratio_tensor[1:]))) \
        * in_height / in_width # 处理矩形输入
    h = torch.cat((size_tensor / torch.sqrt(ratio_tensor[0]),
                   sizes[0] / torch.sqrt(ratio_tensor[1:])))
    # 除以2来获得半高和半宽
    anchor_manipulations = torch.stack((-w, -h, w, h)).T.repeat(
        in_height * in_width, 1) / 2

    # 每个中心点都将有“boxes_per_pixel”个锚框，
    # 所以生成含所有锚框中心的网格，重复了“boxes_per_pixel”次
    out_grid = torch.stack([shift_x, shift_y, shift_x, shift_y],
                          dim=1).repeat_interleave(boxes_per_pixel, dim=0)
    output = out_grid + anchor_manipulations
    return output.unsqueeze(0)

```

可以看到返回的锚框变量Y的形状是（批量大小， 锚框的数量， 4）。

```

img = d2l.plt.imread('../img/catdog.jpg')
h, w = img.shape[:2]

```

(continues on next page)

(continued from previous page)

```
print(h, w)
X = torch.rand(size=(1, 3, h, w))
Y = multibox_prior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
```

561 728

```
torch.Size([1, 2042040, 4])
```

将锚框变量 Y 的形状更改为(图像高度, 图像宽度, 以同一像素为中心的锚框的数量, 4)后, 我们可以获得以指定像素的位置为中心的所有锚框。在接下来的内容中, 我们访问以 (250,250) 为中心的第一个锚框。它有四个元素: 锚框左上角的 (x, y) 轴坐标和右下角的 (x, y) 轴坐标。输出中两个轴的坐标各分别除以了图像的宽度和高度。

```
boxes = Y.reshape(h, w, 5, 4)
boxes[250, 250, 0, :]
```

```
tensor([0.06, 0.07, 0.63, 0.82])
```

为了显示以图像中以某个像素为中心的所有锚框, 定义下面的`show_bboxes`函数来在图像上绘制多个边界框。

```
#@save
def show_bboxes(axes, bboxes, labels=None, colors=None):
    """显示所有边界框"""
    def _make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj

    labels = _make_list(labels)
    colors = _make_list(colors, ['b', 'g', 'r', 'm', 'c'])
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = d2l.bbox_to_rect(bbox.detach().numpy(), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
```

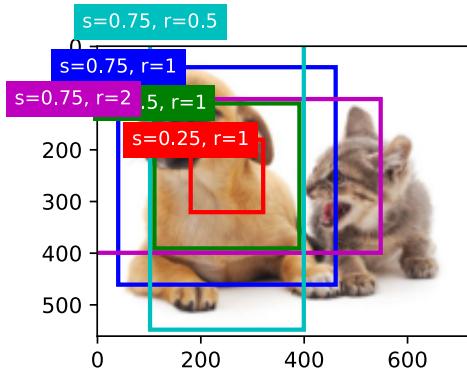
(continues on next page)

(continued from previous page)

```
axes.text(rect.xy[0], rect.xy[1], labels[i],
           va='center', ha='center', fontsize=9, color=text_color,
           bbox=dict(facecolor=color, lw=0))
```

正如从上面代码中所看到的，变量`boxes`中 x 轴和 y 轴的坐标值已分别除以图像的宽度和高度。绘制锚框时，我们需要恢复它们原始的坐标值。因此，在下面定义了变量`bbox_scale`。现在可以绘制出图像中所有以(250,250)为中心的锚框了。如下所示，缩放比为0.75且宽高比为1的蓝色锚框很好地围绕着图像中的狗。

```
d2l.set_figsize()
bbox_scale = torch.tensor((w, h, w, h))
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,
            ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
             's=0.75, r=0.5'])
```



13.4.2 交并比 (IoU)

我们刚刚提到某个锚框“较好地”覆盖了图像中的狗。如果已知目标的真实边界框，那么这里的“好”该如何如何量化呢？直观地说，可以衡量锚框和真实边界框之间的相似性。杰卡德系数（Jaccard）可以衡量两组之间的相似性。给定集合 \mathcal{A} 和 \mathcal{B} ，他们的杰卡德系数是他们交集的大小除以他们并集的大小：

$$J(\mathcal{A}, \mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}. \quad (13.4.2)$$

事实上，我们可以将任何边界框的像素区域视为一组像素。通过这种方式，我们可以通过其像素集的杰卡德系数来测量两个边界框的相似性。对于两个边界框，它们的杰卡德系数通常称为交并比（intersection over union, IoU），即两个边界框相交面积与相并面积之比，如图13.4.1所示。交并比的取值范围在0和1之间：0表示两个边界框无重合像素，1表示两个边界框完全重合。

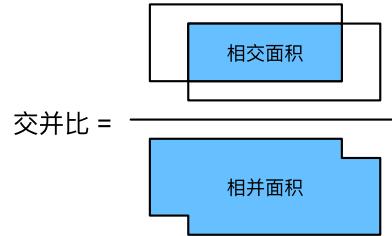


图13.4.1: 交并比是两个边界框相交面积与相并面积之比。

接下来部分将使用交并比来衡量锚框和真实边界框之间、以及不同锚框之间的相似度。给定两个锚框或边界框的列表，以下box_iou函数将在这两个列表中计算它们成对的交并比。

```
#@save
def box_iou(boxes1, boxes2):
    """计算两个锚框或边界框列表中成对的交并比"""
    box_area = lambda boxes: ((boxes[:, 2] - boxes[:, 0]) *
                               (boxes[:, 3] - boxes[:, 1]))
    # boxes1, boxes2, areas1, areas2的形状:
    # boxes1: (boxes1的数量, 4),
    # boxes2: (boxes2的数量, 4),
    # areas1: (boxes1的数量,),
    # areas2: (boxes2的数量,)
    areas1 = box_area(boxes1)
    areas2 = box_area(boxes2)
    # inter_upperlefts, inter_lowerrights, inters的形状:
    # (boxes1的数量, boxes2的数量, 2)
    inter_upperlefts = torch.max(boxes1[:, None, :2], boxes2[:, :2])
    inter_lowerrights = torch.min(boxes1[:, None, 2:], boxes2[:, 2:])
    inters = (inter_lowerrights - inter_upperlefts).clamp(min=0)
    # inter_areasandunion_areas的形状:(boxes1的数量, boxes2的数量)
    inter_areas = inters[:, :, 0] * inters[:, :, 1]
    union_areas = areas1[:, None] + areas2 - inter_areas
    return inter_areas / union_areas
```

13.4.3 在训练数据中标注锚框

在训练集中，我们将每个锚框视为一个训练样本。为了训练目标检测模型，我们需要每个锚框的类别（class）和偏移量（offset）标签，其中前者是与锚框相关的对象的类别，后者是真实边界框相对于锚框的偏移量。在预测时，我们为每个图像生成多个锚框，预测所有锚框的类别和偏移量，根据预测的偏移量调整它们的位置以获得预测的边界框，最后只输出符合特定条件的预测边界框。

目标检测训练集带有真实边界框的位置及其包围物体类别的标签。要标记任何生成的锚框，我们可以参考分配到的最接近此锚框的真实边界框的位置和类别标签。下文将介绍一个算法，它能够把最接近的真实边界框

分配给锚框。

将真实边界框分配给锚框

给定图像，假设锚框是 A_1, A_2, \dots, A_{n_a} ，真实边界框是 B_1, B_2, \dots, B_{n_b} ，其中 $n_a \geq n_b$ 。让我们定义一个矩阵 $\mathbf{X} \in \mathbb{R}^{n_a \times n_b}$ ，其中第*i*行、第*j*列的元素 x_{ij} 是锚框 A_i 和真实边界框 B_j 的IoU。该算法包含以下步骤。

1. 在矩阵 \mathbf{X} 中找到最大的元素，并将它的行索引和列索引分别表示为 i_1 和 j_1 。然后将真实边界框 B_{j_1} 分配给锚框 A_{i_1} 。这很直观，因为 A_{i_1} 和 B_{j_1} 是所有锚框和真实边界框配对中最相近的。在第一个分配完成后，丢弃矩阵中 i_1^{th} 行和 j_1^{th} 列中的所有元素。
2. 在矩阵 \mathbf{X} 中找到剩余元素中最大的元素，并将它的行索引和列索引分别表示为 i_2 和 j_2 。我们将真实边界框 B_{j_2} 分配给锚框 A_{i_2} ，并丢弃矩阵中 i_2^{th} 行和 j_2^{th} 列中的所有元素。
3. 此时，矩阵 \mathbf{X} 中两行和两列中的元素已被丢弃。我们继续，直到丢弃掉矩阵 \mathbf{X} 中 n_b 列中的所有元素。此时已经为这 n_b 个锚框各自分配了一个真实边界框。
4. 只遍历剩下的 $n_a - n_b$ 个锚框。例如，给定任何锚框 A_i ，在矩阵 \mathbf{X} 的第*i*th行中找到与 A_i 的IoU最大的真实边界框 B_j ，只有当此IoU大于预定义的阈值时，才将 B_j 分配给 A_i 。

下面用一个具体的例子来说明上述算法。如图13.4.2（左）所示，假设矩阵 \mathbf{X} 中的最大值为 x_{23} ，我们将真实边界框 B_3 分配给锚框 A_2 。然后，我们丢弃矩阵第2行和第3列中的所有元素，在剩余元素（阴影区域）中找到最大的 x_{71} ，然后将真实边界框 B_1 分配给锚框 A_7 。接下来，如图13.4.2（中）所示，丢弃矩阵第7行和第1列中的所有元素，在剩余元素（阴影区域）中找到最大的 x_{54} ，然后将真实边界框 B_4 分配给锚框 A_5 。最后，如图13.4.2（右）所示，丢弃矩阵第5行和第4列中的所有元素，在剩余元素（阴影区域）中找到最大的 x_{92} ，然后将真实边界框 B_2 分配给锚框 A_9 。之后，我们只需要遍历剩余的锚框 A_1, A_3, A_4, A_6, A_8 ，然后根据阈值确定是否为它们分配真实边界框。

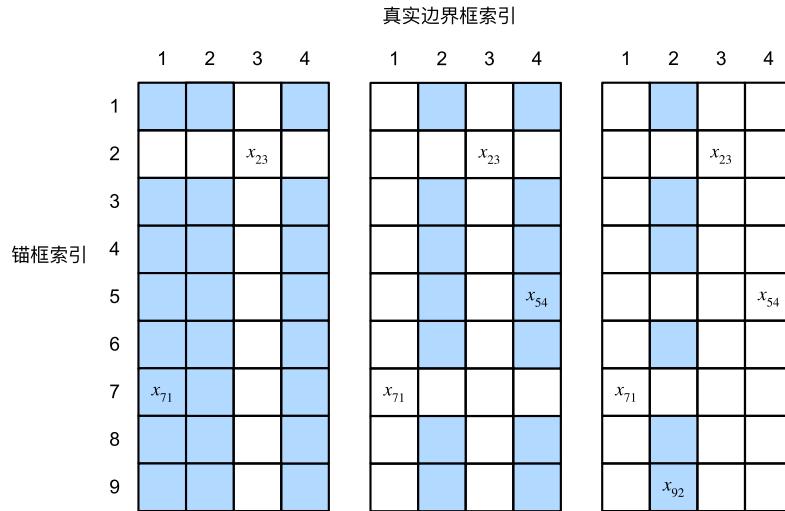


图13.4.2: 将真实边界框分配给锚框。

此算法在下面的assign_anchor_to_bbox函数中实现。

```

#@save
def assign_anchor_to_bbox(ground_truth, anchors, device, iou_threshold=0.5):
    """将最接近的真实边界框分配给锚框"""
    num_anchors, num_gt_boxes = anchors.shape[0], ground_truth.shape[0]
    # 位于第i行和第j列的元素x_ij是锚框i和真实边界框j的IoU
    jaccard = box_iou(anchors, ground_truth)
    # 对于每个锚框，分配的真实边界框的张量
    anchors_bbox_map = torch.full((num_anchors,), -1, dtype=torch.long,
                                  device=device)
    # 根据阈值，决定是否分配真实边界框
    max_ious, indices = torch.max(jaccard, dim=1)
    anc_i = torch.nonzero(max_ious >= iou_threshold).reshape(-1)
    box_j = indices[max_ious >= iou_threshold]
    anchors_bbox_map[anc_i] = box_j
    col_discard = torch.full((num_anchors,), -1)
    row_discard = torch.full((num_gt_boxes,), -1)
    for _ in range(num_gt_boxes):
        max_idx = torch.argmax(jaccard)
        box_idx = (max_idx % num_gt_boxes).long()
        anc_idx = (max_idx / num_gt_boxes).long()
        anchors_bbox_map[anc_idx] = box_idx
        jaccard[:, box_idx] = col_discard
        jaccard[anc_idx, :] = row_discard
    return anchors_bbox_map

```

标记类别和偏移量

现在我们可以为每个锚框标记类别和偏移量了。假设一个锚框 A 被分配了一个真实边界框 B 。一方面，锚框 A 的类别将被标记为与 B 相同。另一方面，锚框 A 的偏移量将根据 B 和 A 中心坐标的相对位置以及这两个框的相对大小进行标记。鉴于数据集内不同的框的位置和大小不同，我们可以对那些相对位置和大小应用变换，使其获得分布更均匀且易于拟合的偏移量。这里介绍一种常见的变换。给定框 A 和 B ，中心坐标分别为 (x_a, y_a) 和 (x_b, y_b) ，宽度分别为 w_a 和 w_b ，高度分别为 h_a 和 h_b ，可以将 A 的偏移量标记为：

$$\left(\frac{\frac{x_b-x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b-y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log \frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log \frac{h_b}{h_a} - \mu_h}{\sigma_h} \right), \quad (13.4.3)$$

其中常量的默认值为 $\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1, \sigma_w = \sigma_h = 0.2$ 。这种转换在下面的 `offset_boxes` 函数中实现。

```

#@save
def offset_boxes(anchors, assigned_bb, eps=1e-6):
    """对锚框偏移量的转换"""
    c_anc = d2l.box_corner_to_center(anchors)

```

(continues on next page)

(continued from previous page)

```
c_assigned_bb = d2l.box_corner_to_center(assigned_bb)
offset_xy = 10 * (c_assigned_bb[:, :2] - c_anc[:, :2]) / c_anc[:, 2:]
offset_wh = 5 * torch.log(eps + c_assigned_bb[:, 2:]) / c_anc[:, 2:])
offset = torch.cat([offset_xy, offset_wh], axis=1)
return offset
```

如果一个锚框没有被分配真实边界框，我们只需将锚框的类别标记为背景 (background)。背景类别的锚框通常被称为负类锚框，其余的被称为正类锚框。我们使用真实边界框 (labels参数) 实现以下multibox_target函数，来标记锚框的类别和偏移量 (anchors参数)。此函数将背景类别的索引设置为零，然后将新类别的整数索引递增一。

```
#@save
def multibox_target(anchors, labels):
    """使用真实边界框标记锚框"""
    batch_size, anchors = labels.shape[0], anchors.squeeze(0)
    batch_offset, batch_mask, batch_class_labels = [], [], []
    device, num_anchors = anchors.device, anchors.shape[0]
    for i in range(batch_size):
        label = labels[i, :, :]
        anchors_bbox_map = assign_anchor_to_bbox(
            label[:, 1:], anchors, device)
        bbox_mask = ((anchors_bbox_map >= 0).float().unsqueeze(-1)).repeat(
            1, 4)
        # 将类标签和分配的边界框坐标初始化为零
        class_labels = torch.zeros(num_anchors, dtype=torch.long,
                                  device=device)
        assigned_bb = torch.zeros((num_anchors, 4), dtype=torch.float32,
                                  device=device)
        # 使用真实边界框来标记锚框的类别。
        # 如果一个锚框没有被分配，标记其为背景（值为零）
        indices_true = torch.nonzero(anchors_bbox_map >= 0)
        bb_idx = anchors_bbox_map[indices_true]
        class_labels[indices_true] = label[bb_idx, 0].long() + 1
        assigned_bb[indices_true] = label[bb_idx, 1:]
        # 偏移量转换
        offset = offset_boxes(anchors, assigned_bb) * bbox_mask
        batch_offset.append(offset.reshape(-1))
        batch_mask.append(bbox_mask.reshape(-1))
        batch_class_labels.append(class_labels)
    bbox_offset = torch.stack(batch_offset)
    bbox_mask = torch.stack(batch_mask)
    class_labels = torch.stack(batch_class_labels)
    return (bbox_offset, bbox_mask, class_labels)
```

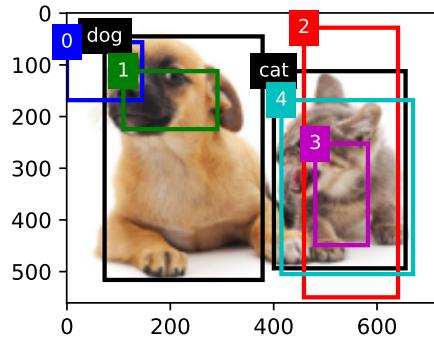
一个例子

下面通过一个具体的例子来说明锚框标签。我们已经为加载图像中的狗和猫定义了真实边界框，其中第一个元素是类别（0代表狗，1代表猫），其余四个元素是左上角和右下角的 (x, y) 轴坐标（范围介于0和1之间）。我们还构建了五个锚框，用左上角和右下角的坐标进行标记： A_0, \dots, A_4 （索引从0开始）。然后我们在图像中绘制这些真实边界框和锚框。

```
ground_truth = torch.tensor([[0, 0.1, 0.08, 0.52, 0.92],
                             [1, 0.55, 0.2, 0.9, 0.88]])
anchors = torch.tensor([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                        [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                        [0.57, 0.3, 0.92, 0.9]])
```



```
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```



使用上面定义的`multibox_target`函数，我们可以根据狗和猫的真实边界框，标注这些锚框的分类和偏移量。在这个例子中，背景、狗和猫的类索引分别为0、1和2。下面我们为锚框和真实边界框样本添加一个维度。

```
labels = multibox_target(anchors.unsqueeze(dim=0),
                         ground_truth.unsqueeze(dim=0))
```

返回的结果中有三个元素，都是张量格式。第三个元素包含标记的输入锚框的类别。

让我们根据图像中的锚框和真实边界框的位置来分析下面返回的类别标签。首先，在所有的锚框和真实边界框配对中，锚框 A_4 与猫的真实边界框的IoU是最大的。因此， A_4 的类别被标记为猫。去除包含 A_4 或猫的真实边界框的配对，在剩下的配对中，锚框 A_1 和狗的真实边界框有最大的IoU。因此， A_1 的类别被标记为狗。接下来，我们需要遍历剩下的三个未标记的锚框： A_0, A_2 和 A_3 。对于 A_0 ，与其拥有最大IoU的真实边界框的类别是狗，但IoU低于预定义的阈值（0.5），因此该类别被标记为背景；对于 A_2 ，与其拥有最大IoU的真实边界框的类别是猫，IoU超过阈值，所以类别被标记为猫；对于 A_3 ，与其拥有最大IoU的真实边界框的类别是猫，但值低于阈值，因此该类别被标记为背景。

```
labels[2]
```

```
tensor([[0, 1, 2, 0, 2]])
```

返回的第二个元素是掩码（mask）变量，形状为（批量大小，锚框数的四倍）。掩码变量中的元素与每个锚框的4个偏移量一一对应。由于我们不关心对背景的检测，负类的偏移量不应影响目标函数。通过元素乘法，掩码变量中的零将在计算目标函数之前过滤掉负类偏移量。

```
labels[1]
```

```
tensor([[0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 1., 1.,
        1., 1.]])
```

返回的第一个元素包含了为每个锚框标记的四个偏移值。请注意，负类锚框的偏移量被标记为零。

```
labels[0]
```

```
tensor([[-0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00,  1.40e+00,  1.00e+01,
        2.59e+00,  7.18e+00, -1.20e+00,  2.69e-01,  1.68e+00, -1.57e+00,
       -0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00, -5.71e-01, -1.00e+00,
        4.17e-06,  6.26e-01]])
```

13.4.4 使用非极大值抑制预测边界框

在预测时，我们先为图像生成多个锚框，再为这些锚框一一预测类别和偏移量。一个预测好的边界框则根据其中某个带有预测偏移量的锚框而生成。下面我们实现了`offset_inverse`函数，该函数将锚框和偏移量预测作为输入，并应用逆偏移变换来返回预测的边界框坐标。

```
#@save
def offset_inverse(anchors, offset_preds):
    """根据带有预测偏移量的锚框来预测边界框"""
    anc = d2l.box_corner_to_center(anchors)
    pred_bbox_xy = (offset_preds[:, :2] * anc[:, 2:] / 10) + anc[:, :2]
    pred_bbox_wh = torch.exp(offset_preds[:, 2:] / 5) * anc[:, 2:]
    pred_bbox = torch.cat((pred_bbox_xy, pred_bbox_wh), axis=1)
    predicted_bbox = d2l.box_center_to_corner(pred_bbox)
    return predicted_bbox
```

当有许多锚框时，可能会输出许多相似的具有明显重叠的预测边界框，都围绕着同一目标。为了简化输出，我们可以使用非极大值抑制（non-maximum suppression, NMS）合并属于同一目标的类似的预测边界框。

以下是非极大值抑制的工作原理。对于一个预测边界框 B ，目标检测模型会计算每个类别的预测概率。假设最大的预测概率为 p ，则该概率所对应的类别 B 即为预测的类别。具体来说，我们将 p 称为预测边界框 B 的置信度（confidence）。在同一张图像中，所有预测的非背景边界框都按置信度降序排序，以生成列表 L 。然后我们通过以下步骤操作排序列表 L 。

1. 从 L 中选取置信度最高的预测边界框 B_1 作为基准，然后将所有与 B_1 的IoU超过预定阈值 ϵ 的非基准预测边界框从 L 中移除。这时， L 保留了置信度最高的预测边界框，去除了与其太过相似的其他预测边界框。简而言之，那些具有非极大值置信度的边界框被抑制了。
2. 从 L 中选取置信度第二高的预测边界框 B_2 作为又一个基准，然后将所有与 B_2 的IoU大于 ϵ 的非基准预测边界框从 L 中移除。
3. 重复上述过程，直到 L 中的所有预测边界框都曾被用作基准。此时， L 中任意一对预测边界框的IoU都小于阈值 ϵ ；因此，没有一对边界框过于相似。
4. 输出列表 L 中的所有预测边界框。

以下nms函数按降序对置信度进行排序并返回其索引。

```
#@save
def nms(bboxes, scores, iou_threshold):
    """对预测边界框的置信度进行排序"""
    B = torch.argsort(scores, dim=-1, descending=True)
    keep = [] # 保留预测边界框的指标
    while B.numel() > 0:
        i = B[0]
        keep.append(i)
        if B.numel() == 1: break
        iou = box_iou(bboxes[i, :], bboxes[B[1:], :].reshape(-1, 4),
                      bboxes[B[1:], :, :].reshape(-1, 4)).reshape(-1)
        inds = torch.nonzero(iou <= iou_threshold).reshape(-1)
        B = B[inds + 1]
    return torch.tensor(keep, device=bboxes.device)
```

我们定义以下multibox_detection函数来将非极大值抑制应用于预测边界框。这里的实现有点复杂，请不要担心。我们将在实现之后，马上用一个具体的例子来展示它是如何工作的。

```
#@save
def multibox_detection(cls_probs, offset_preds, anchors, nms_threshold=0.5,
                      pos_threshold=0.009999999):
    """使用非极大值抑制来预测边界框"""
    device, batch_size = cls_probs.device, cls_probs.shape[0]
    anchors = anchors.squeeze(0)
    num_classes, num_anchors = cls_probs.shape[1], cls_probs.shape[2]
    out = []
    for i in range(batch_size):
```

(continues on next page)

(continued from previous page)

```
cls_prob, offset_pred = cls_probs[i], offset_preds[i].reshape(-1, 4)
conf, class_id = torch.max(cls_prob[1:], 0)
predicted_bb = offset_inverse(anchors, offset_pred)
keep = nms(predicted_bb, conf, nms_threshold)

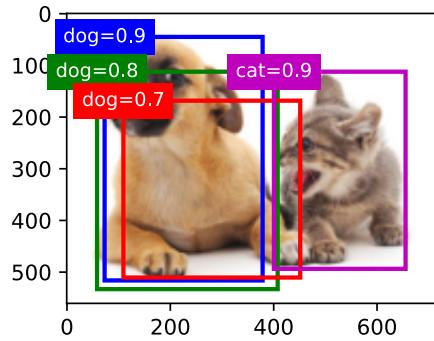
# 找到所有的non_keep索引，并将类设置为背景
all_idx = torch.arange(num_anchors, dtype=torch.long, device=device)
combined = torch.cat((keep, all_idx))
uniques, counts = combined.unique(return_counts=True)
non_keep = uniques[counts == 1]
all_id_sorted = torch.cat((keep, non_keep))
class_id[non_keep] = -1
class_id = class_id[all_id_sorted]
conf, predicted_bb = conf[all_id_sorted], predicted_bb[all_id_sorted]
# pos_threshold是一个用于非背景预测的阈值
below_min_idx = (conf < pos_threshold)
class_id[below_min_idx] = -1
conf[below_min_idx] = 1 - conf[below_min_idx]
pred_info = torch.cat((class_id.unsqueeze(1),
                      conf.unsqueeze(1),
                      predicted_bb), dim=1)
out.append(pred_info)
return torch.stack(out)
```

现在让我们将上述算法应用到一个带有四个锚框的具体示例中。为简单起见，我们假设预测的偏移量都是零，这意味着预测的边界框即是锚框。对于背景、狗和猫其中的每个类，我们还定义了它的预测概率。

```
anchors = torch.tensor([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                       [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
offset_preds = torch.tensor([0] * anchors.numel())
cls_probs = torch.tensor([[0] * 4, # 背景的预测概率
                         [0.9, 0.8, 0.7, 0.1], # 狗的预测概率
                         [0.1, 0.2, 0.3, 0.9]]) # 猫的预测概率
```

我们可以在图像上绘制这些预测边界框和置信度。

```
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, anchors * bbox_scale,
            ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```



现在我们可以调用`multibox_detection`函数来执行非极大值抑制，其中阈值设置为0.5。请注意，我们在示例的张量输入中添加了维度。

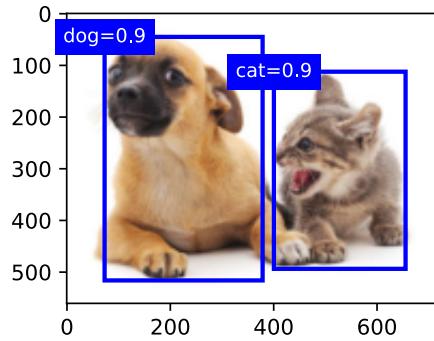
我们可以看到返回结果的形状是（批量大小， 锚框的数量， 6）。最内层维度中的六个元素提供了同一预测边界框的输出信息。第一个元素是预测的类索引，从0开始（0代表狗， 1代表猫），值-1表示背景或在非极大值抑制中被移除了。第二个元素是预测的边界框的置信度。其余四个元素分别是预测边界框左上角和右下角的 (x, y) 轴坐标（范围介于0和1之间）。

```
output = multibox_detection(cls_probs.unsqueeze(dim=0),
                            offset_preds.unsqueeze(dim=0),
                            anchors.unsqueeze(dim=0),
                            nms_threshold=0.5)
output
```

```
tensor([[[ 0.00,  0.90,  0.10,  0.08,  0.52,  0.92],
        [ 1.00,  0.90,  0.55,  0.20,  0.90,  0.88],
        [-1.00,  0.80,  0.08,  0.20,  0.56,  0.95],
        [-1.00,  0.70,  0.15,  0.30,  0.62,  0.91]]])
```

删除-1类别（背景）的预测边界框后，我们可以输出由非极大值抑制保存的最终预测边界框。

```
fig = d2l=plt.imshow(img)
for i in output[0].detach().numpy():
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [torch.tensor(i[2:]) * bbox_scale], label)
```



实践中，在执行非极大值抑制前，我们甚至可以将置信度较低的预测边界框移除，从而减少此算法中的计算量。我们也可以对非极大值抑制的输出结果进行后处理。例如，只保留置信度更高的结果作为最终输出。

小结

- 我们以图像的每个像素为中心生成不同形状的锚框。
- 交并比 (IoU) 也被称为杰卡德系数，用于衡量两个边界框的相似性。它是相交面积与相并面积的比率。
- 在训练集中，我们需要给每个锚框两种类型的标签。一个是与锚框中目标检测的类别，另一个是锚框真实相对于边界框的偏移量。
- 预测期间可以使用非极大值抑制（NMS）来移除类似的预测边界框，从而简化输出。

练习

1. 在`multibox_prior`函数中更改`sizes`和`ratios`的值。生成的锚框有什么变化？
2. 构建并可视化两个IoU为0.5的边界框。它们是怎样重叠的？
3. 在 13.4.3节 和 13.4.4节 中修改变量`anchors`，结果如何变化？
4. 非极大值抑制是一种贪心算法，它通过移除来抑制预测的边界框。是否存在一种可能，被移除的一些框实际上是有用的？如何修改这个算法来柔和地抑制？可以参考Soft-NMS (Bodla et al., 2017)。
5. 如果非手动，非最大限度的抑制可以被学习吗？

Discussions¹⁷⁴

¹⁷⁴ <https://discuss.d2l.ai/t/2946>

13.5 多尺度目标检测

在 13.4 节中，我们以输入图像的每个像素为中心，生成了多个锚框。基本而言，这些锚框代表了图像不同区域的样本。然而，如果为每个像素都生成的锚框，我们最终可能会得到太多需要计算的锚框。想象一个 561×728 的输入图像，如果以每个像素为中心生成五个形状不同的锚框，就需要在图像上标记和预测超过 200 万个锚框 ($561 \times 728 \times 5$)。

13.5.1 多尺度锚框

减少图像上的锚框数量并不困难。比如，我们可以在输入图像中均匀采样一小部分像素，并以它们为中心生成锚框。此外，在不同尺度下，我们可以生成不同数量和不同大小的锚框。直观地说，比起较大的目标，较小的目标在图像上出现的可能性更多样。例如， 1×1 、 1×2 和 2×2 的目标可以分别以 4、2 和 1 种可能的方式出现在 2×2 图像上。因此，当使用较小的锚框检测较小的物体时，我们可以采样更多的区域，而对于较大的物体，我们可以采样较少的区域。

为了演示如何在多个尺度下生成锚框，让我们先读取一张图像。它的高度和宽度分别为 561 和 728 像素。

```
%matplotlib inline
import torch
from d2l import torch as d2l

img = d2l.plt.imread('../img/catdog.jpg')
h, w = img.shape[:2]
h, w
```

(561, 728)

回想一下，在 6.2 节中，我们将卷积图层的二维数组输出称为特征图。通过定义特征图的形状，我们可以确定任何图像上均匀采样锚框的中心。

`display_anchors` 函数定义如下。我们在特征图 (`fmap`) 上生成锚框 (`anchors`)，每个单位（像素）作为锚框的中心。由于锚框中的 (x, y) 轴坐标值 (`anchors`) 已经被除以特征图 (`fmap`) 的宽度和高度，因此这些值介于 0 和 1 之间，表示特征图中锚框的相对位置。

由于锚框 (`anchors`) 的中心分布于特征图 (`fmap`) 上的所有单位，因此这些中心必须根据其相对空间位置在任何输入图像上均匀分布。更具体地说，给定特征图的宽度和高度 `fmap_w` 和 `fmap_h`，以下函数将均匀地对任何输入图像中 `fmap_h` 行和 `fmap_w` 列中的像素进行采样。以这些均匀采样的像素为中心，将会生成大小为 `s`（假设列表 `s` 的长度为 1）且宽高比 (`ratios`) 不同的锚框。

```
def display_anchors(fmap_w, fmap_h, s):
    d2l.set_figsize()
    # 前两个维度上的值不影响输出
```

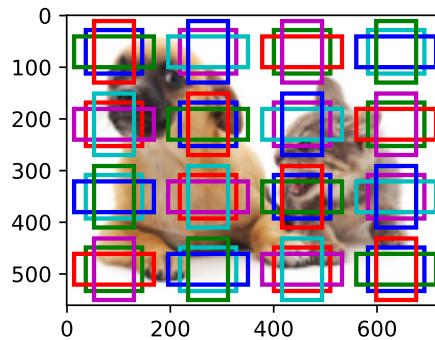
(continues on next page)

(continued from previous page)

```
fmap = torch.zeros((1, 10, fmap_h, fmap_w))
anchors = d2l.multibox_prior(fmap, sizes=s, ratios=[1, 2, 0.5])
bbox_scale = torch.tensor((w, h, w, h))
d2l.show_bboxes(d2l.plt.imshow(img).axes,
    anchors[0] * bbox_scale)
```

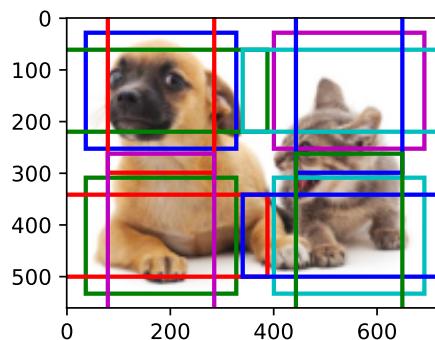
首先，让我们考虑探测小目标。为了在显示时更容易分辨，在这里具有不同中心的锚框不会重叠：锚框的尺度设置为0.15，特征图的高度和宽度设置为4。我们可以看到，图像上4行和4列的锚框的中心是均匀分布的。

```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```



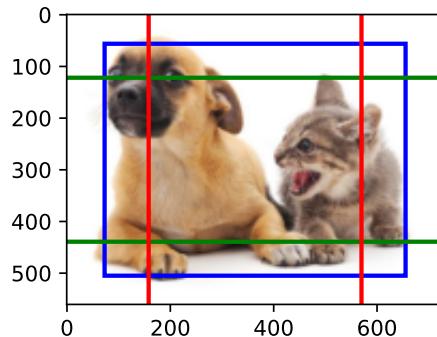
然后，我们将特征图的高度和宽度减小一半，然后使用较大的锚框来检测较大的目标。当尺度设置为0.4时，一些锚框将彼此重叠。

```
display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```



最后，我们进一步将特征图的高度和宽度减小一半，然后将锚框的尺度增加到0.8。此时，锚框的中心即是图像的中心。

```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```



13.5.2 多尺度检测

既然我们已经生成了多尺度的锚框，我们就将使用它们来检测不同尺度下各种大小的目标。下面，我们介绍一种基于CNN的多尺度目标检测方法，将在 13.7 节中实现。

在某种规模上，假设我们有 c 张形状为 $h \times w$ 的特征图。使用 13.5.1 节中的方法，我们生成了 hw 组锚框，其中每组都有 a 个中心相同的锚框。例如，在 13.5.1 节实验的第一个尺度上，给定10个（通道数量） 4×4 的特征图，我们生成了16组锚框，每组包含3个中心相同的锚框。接下来，每个锚框都根据真实值边界框来标记了类和偏移量。在当前尺度下，目标检测模型需要预测输入图像上 hw 组锚框类别和偏移量，其中不同组锚框具有不同的中心。

假设此处的 c 张特征图是CNN基于输入图像的正向传播算法获得的中间输出。既然每张特征图上都有 hw 个不同的空间位置，那么相同空间位置可以看作含有 c 个单元。根据 6.2 节中对感受野的定义，特征图在相同空间位置的 c 个单元在输入图像上的感受野相同：它们表征了同一感受野内的输入图像信息。因此，我们可以将特征图在同一空间位置的 c 个单元变换为使用此空间位置生成的 a 个锚框类别和偏移量。本质上，我们用输入图像在某个感受野区域内的信息，来预测输入图像上与该区域位置相近的锚框类别和偏移量。

当不同层的特征图在输入图像上分别拥有不同大小的感受野时，它们可以用于检测不同大小的目标。例如，我们可以设计一个神经网络，其中靠近输出层的特征图单元具有更宽的感受野，这样它们就可以从输入图像中检测到较大的目标。

简言之，我们可以利用深层神经网络在多个层次上对图像进行分层表示，从而实现多尺度目标检测。在 13.7 节，我们将通过一个具体的例子来说明它是如何工作的。

小结

- 在多个尺度下，我们可以生成不同尺寸的锚框来检测不同尺寸的目标。
- 通过定义特征图的形状，我们可以决定任何图像上均匀采样的锚框的中心。
- 我们使用输入图像在某个感受野区域内的信息，来预测输入图像上与该区域位置相近的锚框类别和偏移量。
- 我们可以通过深入学习，在多个层次上的图像分层表示进行多尺度目标检测。

练习

1. 根据我们在 7.1 节中的讨论，深度神经网络学习图像特征级别抽象层次，随网络深度的增加而升级。在多尺度目标检测中，不同尺度的特征映射是否对应于不同的抽象层次？为什么？
2. 在 13.5.1 节中的实验里的第一个尺度 ($fmap_w=4, fmap_h=4$) 下，生成可能重叠的均匀分布的锚框。
3. 给定形状为 $1 \times c \times h \times w$ 的特征图变量，其中 c, h 和 w 分别是特征图的通道数、高度和宽度。怎样才能将这个变量转换为锚框类别和偏移量？输出的形状是什么？

Discussions¹⁷⁵

13.6 目标检测数据集

目标检测领域没有像MNIST和Fashion-MNIST那样的小数据集。为了快速测试目标检测模型，我们收集并标记了一个小型数据集。首先，我们拍摄了一组香蕉的照片，并生成了1000张不同角度和大小的香蕉图像。然后，我们在一些背景图片的随机位置上放一张香蕉的图像。最后，我们在图片上为这些香蕉标记了边界框。

13.6.1 下载数据集

包含所有图像和CSV标签文件的香蕉检测数据集可以直接从互联网下载。

```
%matplotlib inline
import os
import pandas as pd
import torch
import torchvision
from d2l import torch as d2l
```

¹⁷⁵ <https://discuss.d2l.ai/t/2948>

```
#@save
d2l.DATA_HUB['banana-detection'] = (
    d2l.DATA_URL + 'banana-detection.zip',
    '5de26c8fce5ccdea9f91267273464dc968d20d72')
```

13.6.2 读取数据集

通过`read_data_bananas`函数，我们读取香蕉检测数据集。该数据集包括一个的CSV文件，内含目标类别标签和位于左上角和右下角的真实边界框坐标。

```
#@save
def read_data_bananas(is_train=True):
    """读取香蕉检测数据集中的图像和标签"""
    data_dir = d2l.download_extract('banana-detection')
    csv_fname = os.path.join(data_dir, 'bananas_train' if is_train
                             else 'bananas_val', 'label.csv')
    csv_data = pd.read_csv(csv_fname)
    csv_data = csv_data.set_index('img_name')
    images, targets = [], []
    for img_name, target in csv_data.iterrows():
        images.append(torchvision.io.read_image(
            os.path.join(data_dir, 'bananas_train' if is_train else
                         'bananas_val', 'images', f'{img_name}')))
        # 这里的target包含 (类别, 左上角x, 左上角y, 右下角x, 右下角y),
        # 其中所有图像都具有相同的香蕉类 (索引为0)
        targets.append(list(target))
    return images, torch.tensor(targets).unsqueeze(1) / 256
```

通过使用`read_data_bananas`函数读取图像和标签，以下`BananasDataset`类别将允许我们创建一个自定义Dataset实例来加载香蕉检测数据集。

```
#@save
class BananasDataset(torch.utils.data.Dataset):
    """一个用于加载香蕉检测数据集的自定义数据集"""
    def __init__(self, is_train):
        self.features, self.labels = read_data_bananas(is_train)
        print('read ' + str(len(self.features)) + (f' training examples' if
            is_train else f' validation examples')))

    def __getitem__(self, idx):
        return (self.features[idx].float(), self.labels[idx])
```

(continues on next page)

(continued from previous page)

```
def __len__(self):
    return len(self.features)
```

最后，我们定义`load_data_bananas`函数，来为训练集和测试集返回两个数据加载器实例。对于测试集，无须按随机顺序读取它。

```
#@save
def load_data_bananas(batch_size):
    """加载香蕉检测数据集"""
    train_iter = torch.utils.data.DataLoader(BananasDataset(is_train=True),
                                              batch_size, shuffle=True)
    val_iter = torch.utils.data.DataLoader(BananasDataset(is_train=False),
                                           batch_size)
    return train_iter, val_iter
```

让我们读取一个小批量，并打印其中的图像和标签的形状。图像的小批量的形状为（批量大小、通道数、高度、宽度），看起来很眼熟：它与我们之前图像分类任务中的相同。标签的小批量的形状为（批量大小， m , 5），其中 m 是数据集的任何图像中边界框可能出现的最大数量。

小批量计算虽然高效，但它要求每张图像含有相同数量的边界框，以便放在同一个批量中。通常来说，图像可能拥有不同数量个边界框；因此，在达到 m 之前，边界框少于 m 的图像将被非法边界框填充。这样，每个边界框的标签将被长度为5的数组表示。数组中的第一个元素是边界框中对象的类别，其中-1表示用于填充的非法边界框。数组的其余四个元素是边界框左上角和右下角的 (x, y) 坐标值（值域在0~1之间）。对于香蕉数据集而言，由于每张图像上只有一个边界框，因此 $m = 1$ 。

```
batch_size, edge_size = 32, 256
train_iter, _ = load_data_bananas(batch_size)
batch = next(iter(train_iter))
batch[0].shape, batch[1].shape
```

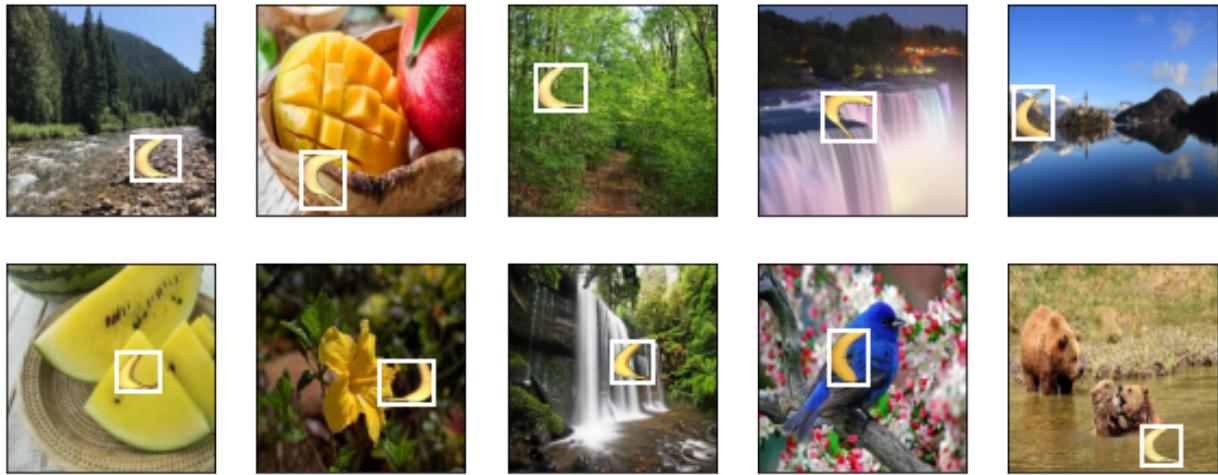
```
Downloading ../data/banana-detection.zip from http://d2l-data.s3-accelerate.amazonaws.com/banana-
↳ detection.zip...
read 1000 training examples
read 100 validation examples
```

```
(torch.Size([32, 3, 256, 256]), torch.Size([32, 1, 5]))
```

13.6.3 演示

让我们展示10幅带有真实边界框的图像。我们可以看到在所有这些图像中香蕉的旋转角度、大小和位置都有所不同。当然，这只是一个简单的人工数据集，实践中真实世界的数据集通常要复杂得多。

```
imgs = (batch[0][0:10].permute(0, 2, 3, 1)) / 255
axes = d2l.show_images(imgs, 2, 5, scale=2)
for ax, label in zip(axes, batch[1][0:10]):
    d2l.show_bboxes(ax, [label[0][1:5] * edge_size], colors=['w'])
```



小结

- 我们收集的香蕉检测数据集可用于演示目标检测模型。
- 用于目标检测的数据加载与图像分类的数据加载类似。但是，在目标检测中，标签还包含真实边界框的信息，它不出现在图像分类中。

练习

1. 在香蕉检测数据集中演示其他带有真实边界框的图像。它们在边界框和目标方面有什么不同？
2. 假设我们想要将数据增强（例如随机裁剪）应用于目标检测。它与图像分类中的有什么不同？提示：如果裁剪的图像只包含物体的一小部分会怎样？

Discussions¹⁷⁶

¹⁷⁶ <https://discuss.d2l.ai/t/3202>

13.7 单发多框检测 (SSD)

在 13.3 节—13.6 节中，我们分别介绍了边界框、锚框、多尺度目标检测和用于目标检测的数据集。现在我们已经准备好使用这样的背景知识来设计一个目标检测模型：单发多框检测 (SSD) (Liu et al., 2016)。该模型简单、快速且被广泛使用。尽管这只是其中一种目标检测模型，但本节中的一些设计原则和实现细节也适用于其他模型。

13.7.1 模型

图13.7.1描述了单发多框检测模型的设计。此模型主要由基础网络组成，其后是几个多尺度特征块。基本网络用于从输入图像中提取特征，因此它可以使用深度卷积神经网络。单发多框检测论文中选用了在分类层之前截断的VGG (Liu et al., 2016)，现在也常用ResNet替代。我们可以设计基础网络，使它输出的高和宽较大。这样一来，基于该特征图生成的锚框数量较多，可以用来检测尺寸较小的目标。接下来的每个多尺度特征块将上一层提供的特征图的高和宽缩小（如减半），并使特征图中每个单元在输入图像上的感受野变得更广阔。

回想一下在 13.5 节中，通过深度神经网络分层表示图像的多尺度目标检测的设计。由于接近 图13.7.1顶部的多尺度特征图较小，但具有较大的感受野，它们适合检测较少但较大的物体。简而言之，通过多尺度特征块，单发多框检测生成不同大小的锚框，并通过预测边界框的类别和偏移量来检测大小不同的目标，因此这是一个多尺度目标检测模型。

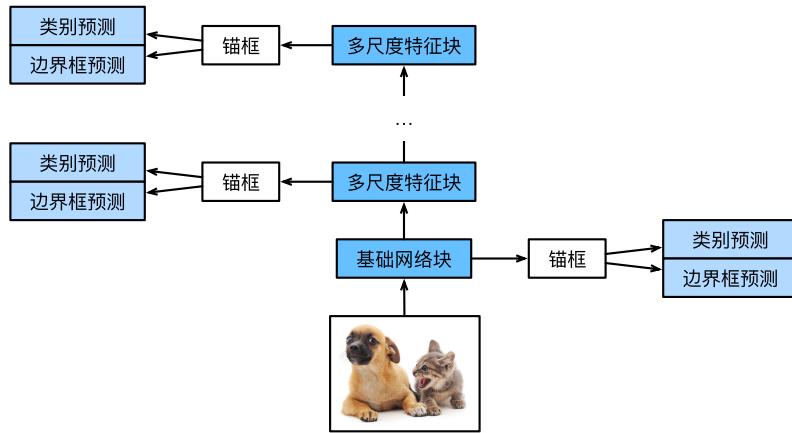


图13.7.1: 单发多框检测模型主要由一个基础网络块和若干多尺度特征块串联而成。

在下面，我们将介绍 图13.7.1 中不同块的实施细节。首先，我们将讨论如何实施类别和边界框预测。

类别预测层

设目标类别的数量为 q 。这样一来，锚框有 $q+1$ 个类别，其中0类是背景。在某个尺度下，设特征图的高和宽分别为 h 和 w 。如果以其中每个单元为中心生成 a 个锚框，那么我们需要对 hwa 个锚框进行分类。如果使用全连接层作为输出，很容易导致模型参数过多。回忆 7.3 节一节介绍的使用卷积层的通道来输出类别预测的方法，单发多框检测采用同样的方法来降低模型复杂度。

具体来说，类别预测层使用一个保持输入高和宽的卷积层。这样一来，输出和输入在特征图宽和高上的空间坐标一一对应。考虑输出和输入同一空间坐标 (x, y) ：输出特征图上 (x, y) 坐标的通道里包含了以输入特征图 (x, y) 坐标为中心生成的所有锚框的类别预测。因此输出通道数为 $a(q+1)$ ，其中索引为 $i(q+1) + j$ ($0 \leq j \leq q$) 的通道代表了索引为 i 的锚框有关类别索引为 j 的预测。

在下面，我们定义了这样一个类别预测层，通过参数`num_anchors`和`num_classes`分别指定了 a 和 q 。该图层使用填充为1的 3×3 的卷积层。此卷积层的输入和输出的宽度和高度保持不变。

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l

def cls_predictor(num_inputs, num_anchors, num_classes):
    return nn.Conv2d(num_inputs, num_anchors * (num_classes + 1),
                   kernel_size=3, padding=1)
```

边界框预测层

边界框预测层的设计与类别预测层的设计类似。唯一不同的是，这里需要为每个锚框预测4个偏移量，而不是 $q+1$ 个类别。

```
def bbox_predictor(num_inputs, num_anchors):
    return nn.Conv2d(num_inputs, num_anchors * 4, kernel_size=3, padding=1)
```

连结多尺度的预测

正如我们所提到的，单发多框检测使用多尺度特征图来生成锚框并预测其类别和偏移量。在不同的尺度下，特征图的形状或以同一单元为中心的锚框的数量可能会有所不同。因此，不同尺度下预测输出的形状可能会有所不同。

在以下示例中，我们为同一个小批量构建两个不同比例（Y1和Y2）的特征图，其中Y2的高度和宽度是Y1的一半。以类别预测为例，假设Y1和Y2的每个单元分别生成了5个和3个锚框。进一步假设目标类别的数量为10，对

于特征图 Y_1 和 Y_2 ，类别预测输出中的通道数分别为 $5 \times (10 + 1) = 55$ 和 $3 \times (10 + 1) = 33$ ，其中任一输出的形状是（批量大小，通道数，高度，宽度）。

```
def forward(x, block):
    return block(x)

Y1 = forward(torch.zeros((2, 8, 20, 20)), cls_predictor(8, 5, 10))
Y2 = forward(torch.zeros((2, 16, 10, 10)), cls_predictor(16, 3, 10))
Y1.shape, Y2.shape
```

```
(torch.Size([2, 55, 20, 20]), torch.Size([2, 33, 10, 10]))
```

正如我们所看到的，除了批量大小这一维度外，其他三个维度都具有不同的尺寸。为了将这两个预测输出链接起来以提高计算效率，我们将把这些张量转换为更一致的格式。

通道维包含中心相同的锚框的预测结果。我们首先将通道维移到最后一维。因为不同尺度下批量大小仍保持不变，我们可以将预测结果转成二维的（批量大小，高×宽×通道数）的格式，以方便之后在维度1上的连结。

```
def flatten_pred(pred):
    return torch.flatten(pred.permute(0, 2, 3, 1), start_dim=1)

def concat_preds(preds):
    return torch.cat([flatten_pred(p) for p in preds], dim=1)
```

这样一来，尽管 Y_1 和 Y_2 在通道数、高度和宽度方面具有不同的大小，我们仍然可以在同一个小批量的两个不同尺度上连接这两个预测输出。

```
concat_preds([Y1, Y2]).shape
```

```
torch.Size([2, 25300])
```

高和宽减半块

为了在多个尺度下检测目标，我们在下面定义了高和宽减半块down_sample_blk，该模块将输入特征图的高度和宽度减半。事实上，该块应用了在subsec_vgg-blocks中的VGG模块设计。更具体地说，每个高和宽减半块由两个填充为1的 3×3 的卷积层、以及步幅为2的 2×2 最大汇聚层组成。我们知道，填充为1的 3×3 卷积层不改变特征图的形状。但是，其后的 2×2 的最大汇聚层将输入特征图的高度和宽度减少了一半。对于此高和宽减半块的输入和输出特征图，因为 $1 \times 2 + (3 - 1) + (3 - 1) = 6$ ，所以输出中的每个单元在输入上都有一个 6×6 的感受野。因此，高和宽减半块会扩大每个单元在其输出特征图中的感受野。

```
def down_sample_blk(in_channels, out_channels):
    blk = []
    for _ in range(2):
        blk.append(nn.Conv2d(in_channels, out_channels,
                            kernel_size=3, padding=1))
        blk.append(nn.BatchNorm2d(out_channels))
        blk.append(nn.ReLU())
        in_channels = out_channels
    blk.append(nn.MaxPool2d(2))
    return nn.Sequential(*blk)
```

在以下示例中，我们构建的高和宽减半块会更改输入通道的数量，并将输入特征图的高度和宽度减半。

```
forward(torch.zeros((2, 3, 20, 20)), down_sample_blk(3, 10)).shape
```

```
torch.Size([2, 10, 10, 10])
```

基本网络块

基本网络块用于从输入图像中抽取特征。为了计算简洁，我们构造了一个小的基础网络，该网络串联3个高和宽减半块，并逐步将通道数翻倍。给定输入图像的形状为 256×256 ，此基本网络块输出的特征图形状为 32×32 ($256/2^3 = 32$)。

```
def base_net():
    blk = []
    num_filters = [3, 16, 32, 64]
    for i in range(len(num_filters) - 1):
        blk.append(down_sample_blk(num_filters[i], num_filters[i+1]))
    return nn.Sequential(*blk)

forward(torch.zeros((2, 3, 256, 256)), base_net()).shape
```

```
torch.Size([2, 64, 32, 32])
```

完整的模型

完整的单发多框检测模型由五个模块组成。每个块生成的特征图既用于生成锚框，又用于预测这些锚框的类别和偏移量。在这五个模块中，第一个是基本网络块，第二个到第四个是高和宽减半块，最后一个模块使用全局最大池将高度和宽度都降到1。从技术上讲，第二到第五个区块都是 图13.7.1中的多尺度特征块。

```
def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 1:
        blk = down_sample_blk(64, 128)
    elif i == 4:
        blk = nn.AdaptiveMaxPool2d((1,1))
    else:
        blk = down_sample_blk(128, 128)
    return blk
```

现在我们为每个块定义前向传播。与图像分类任务不同，此处的输出包括：CNN特征图Y；在当前尺度下根据Y生成的锚框；预测的这些锚框的类别和偏移量（基于Y）。

```
def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
    Y = blk(X)
    anchors = d2l.multibox_prior(Y, sizes=size, ratios=ratio)
    cls_preds = cls_predictor(Y)
    bbox_preds = bbox_predictor(Y)
    return (Y, anchors, cls_preds, bbox_preds)
```

回想一下，在 图13.7.1中，一个较接近顶部的多尺度特征块是用于检测较大目标的，因此需要生成更大的锚框。在上面的前向传播中，在每个多尺度特征块上，我们通过调用的`multibox_prior`函数（见 13.4节）的`sizes`参数传递两个比例值的列表。在下面，0.2和1.05之间的区间被均匀分成五个部分，以确定五个模块的在不同尺度下的较小值：0.2、0.37、0.54、0.71和0.88。之后，他们较大的值由 $\sqrt{0.2 \times 0.37} = 0.272$ 、 $\sqrt{0.37 \times 0.54} = 0.447$ 等给出。

```
sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619], [0.71, 0.79],
         [0.88, 0.961]]
ratios = [[1, 2, 0.5]] * 5
num_anchors = len(sizes[0]) + len(ratios[0]) - 1
```

现在，我们就可以按如下方式定义完整的模型TinySSD了。

```
class TinySSD(nn.Module):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
```

(continues on next page)

(continued from previous page)

```
self.num_classes = num_classes
idx_to_in_channels = [64, 128, 128, 128, 128]
for i in range(5):
    # 即赋值语句self.blk_i=get_blk(i)
    setattr(self, f'blk_{i}', get_blk(i))
    setattr(self, f'cls_{i}', cls_predictor(idx_to_in_channels[i],
                                             num_anchors, num_classes))
    setattr(self, f'bbox_{i}', bbox_predictor(idx_to_in_channels[i],
                                              num_anchors))

def forward(self, X):
    anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
    for i in range(5):
        # getattr(self,'blk_%d'%i)即访问self.blk_i
        X, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(
            X, getattr(self, f'blk_{i}'), sizes[i], ratios[i],
            getattr(self, f'cls_{i}'), getattr(self, f'bbox_{i}'))
    anchors = torch.cat(anchors, dim=1)
    cls_preds = concat_preds(cls_preds)
    cls_preds = cls_preds.reshape(
        cls_preds.shape[0], -1, self.num_classes + 1)
    bbox_preds = concat_preds(bbox_preds)
    return anchors, cls_preds, bbox_preds
```

我们创建一个模型实例，然后使用它对一个 256×256 像素的小批量图像 X 执行前向传播。

如本节前面部分所示，第一个模块输出特征图的形状为 32×32 。回想一下，第二到第四个模块为高和宽减半块，第五个模块为全局汇聚层。由于以特征图的每个单元为中心有4个锚框生成，因此在所有五个尺度下，每个图像总共生成 $(32^2 + 16^2 + 8^2 + 4^2 + 1) \times 4 = 5444$ 个锚框。

```
net = TinySSD(num_classes=1)
X = torch.zeros((32, 3, 256, 256))
anchors, cls_preds, bbox_preds = net(X)

print('output anchors:', anchors.shape)
print('output class preds:', cls_preds.shape)
print('output bbox preds:', bbox_preds.shape)
```

```
output anchors: torch.Size([1, 5444, 4])
output class preds: torch.Size([32, 5444, 2])
output bbox preds: torch.Size([32, 21776])
```

13.7.2 训练模型

现在，我们将描述如何训练用于目标检测的单发多框检测模型。

读取数据集和初始化

首先，让我们读取 13.6 节中描述的香蕉检测数据集。

```
batch_size = 32
train_iter, _ = d2l.load_data_bananas(batch_size)
```

```
read 1000 training examples
read 100 validation examples
```

香蕉检测数据集中，目标的类别数为1。定义好模型后，我们需要初始化其参数并定义优化算法。

```
device, net = d2l.try_gpu(), TinySSD(num_classes=1)
trainer = torch.optim.SGD(net.parameters(), lr=0.2, weight_decay=5e-4)
```

定义损失函数和评价函数

目标检测有两种类型的损失。第一种有关锚框类别的损失：我们可以简单地复用之前图像分类问题里一直使用的交叉熵损失函数来计算；第二种有关正类锚框偏移量的损失：预测偏移量是一个回归问题。但是，对于这个回归问题，我们在这里不使用 3.1.3 节中描述的平方损失，而是使用 L_1 范数损失，即预测值和真实值之差的绝对值。掩码变量 `bbox_masks` 令负类锚框和填充锚框不参与损失的计算。最后，我们将锚框类别和偏移量的损失相加，以获得模型的最终损失函数。

```
cls_loss = nn.CrossEntropyLoss(reduction='none')
bbox_loss = nn.L1Loss(reduction='none')

def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    batch_size, num_classes = cls_preds.shape[0], cls_preds.shape[2]
    cls = cls_loss(cls_preds.reshape(-1, num_classes),
                  cls_labels.reshape(-1)).reshape(batch_size, -1).mean(dim=1)
    bbox = bbox_loss(bbox_preds * bbox_masks,
                     bbox_labels * bbox_masks).mean(dim=1)
    return cls + bbox
```

我们可以沿用准确率评价分类结果。由于偏移量使用了 L_1 范数损失，我们使用平均绝对误差来评价边界框的预测结果。这些预测结果是从生成的锚框及其预测偏移量中获得的。

```

def cls_eval(cls_preds, cls_labels):
    # 由于类别预测结果放在最后一维, argmax需要指定最后一维。
    return float((cls_preds.argmax(dim=-1).type(
        cls_labels.dtype) == cls_labels).sum())

def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
    return float((torch.abs(bbox_labels - bbox_preds) * bbox_masks).sum())

```

训练模型

在训练模型时, 我们需要在模型的前向传播过程中生成多尺度锚框(anchors), 并预测其类别(cls_preds)和偏移量(bbox_preds)。然后, 我们根据标签信息Y为生成的锚框标记类别(cls_labels)和偏移量(bbox_labels)。最后, 我们根据类别和偏移量的预测和标注值计算损失函数。为了代码简洁, 这里没有评价测试数据集。

```

num_epochs, timer = 20, d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                         legend=['class error', 'bbox mae'])
net = net.to(device)
for epoch in range(num_epochs):
    # 训练精确度的和, 训练精确度的和中的示例数
    # 绝对误差的和, 绝对误差的和中的示例数
    metric = d2l.Accumulator(4)
    net.train()
    for features, target in train_iter:
        timer.start()
        trainer.zero_grad()
        X, Y = features.to(device), target.to(device)
        # 生成多尺度的锚框, 为每个锚框预测类别和偏移量
        anchors, cls_preds, bbox_preds = net(X)
        # 为每个锚框标注类别和偏移量
        bbox_labels, bbox_masks, cls_labels = d2l.multibox_target(anchors, Y)
        # 根据类别和偏移量的预测和标注值计算损失函数
        l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                      bbox_masks)
        l.mean().backward()
        trainer.step()
        metric.add(cls_eval(cls_preds, cls_labels), cls_labels.numel(),
                   bbox_eval(bbox_preds, bbox_labels, bbox_masks),
                   bbox_labels.numel())
    cls_err, bbox_mae = 1 - metric[0] / metric[1], metric[2] / metric[3]
    animator.add(epoch + 1, (cls_err, bbox_mae))
print(f'class err {cls_err:.2e}, bbox mae {bbox_mae:.2e}')

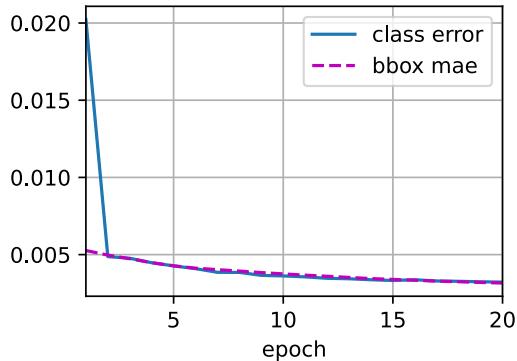
```

(continues on next page)

(continued from previous page)

```
print(f'{len(train_iter.dataset) / timer.stop():.1f} examples/sec on '
      f'{str(device)})')
```

```
class err 3.22e-03, bbox mae 3.16e-03
3353.9 examples/sec on cuda:0
```



13.7.3 预测目标

在预测阶段，我们希望能把图像里面所有我们感兴趣的目标检测出来。在下面，我们读取并调整测试图像的大小，然后将其转成卷积层需要的四维格式。

```
X = torchvision.io.read_image('../img/banana.jpg').unsqueeze(0).float()
img = X.squeeze(0).permute(1, 2, 0).long()
```

使用下面的`multibox_detection`函数，我们可以根据锚框及其预测偏移量得到预测边界框。然后，通过非极大值抑制来移除相似的预测边界框。

```
def predict(X):
    net.eval()
    anchors, cls_preds, bbox_preds = net(X.to(device))
    cls_probs = F.softmax(cls_preds, dim=2).permute(0, 2, 1)
    output = d2l.multibox_detection(cls_probs, bbox_preds, anchors)
    idx = [i for i, row in enumerate(output[0]) if row[0] != -1]
    return output[0], idx

output = predict(X)
```

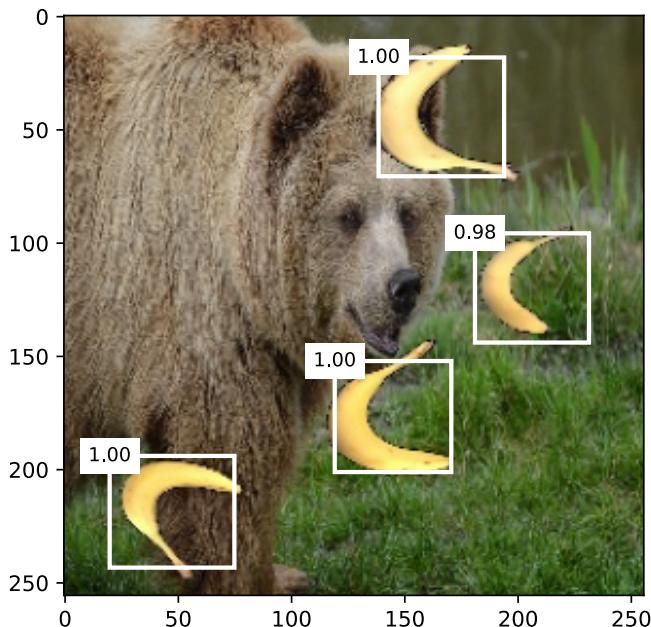
最后，我们筛选所有置信度不低于0.9的边界框，做为最终输出。

```

def display(img, output, threshold):
    d2l.set_figsize((5, 5))
    fig = d2l.plt.imshow(img)
    for row in output:
        score = float(row[1])
        if score < threshold:
            continue
        h, w = img.shape[0:2]
        bbox = [row[2:6] * torch.tensor((w, h, w, h), device=row.device)]
        d2l.show_bboxes(fig.axes, bbox, '%.2f' % score, 'w')

display(img, output.cpu(), threshold=0.9)

```



小结

- 单发多框检测是一种多尺度目标检测模型。基于基础网络块和各个多尺度特征块，单发多框检测生成不同数量和不同大小的锚框，并通过预测这些锚框的类别和偏移量检测不同大小的目标。
- 在训练单发多框检测模型时，损失函数是根据锚框的类别和偏移量的预测及标注值计算得出的。

练习

- 能通过改进损失函数来改进单发多框检测吗？例如，将预测偏移量用到的 L_1 范数损失替换为平滑 L_1 范数损失。它在零点附近使用平方函数从而更加平滑，这是通过一个超参数 σ 来控制平滑区域的：

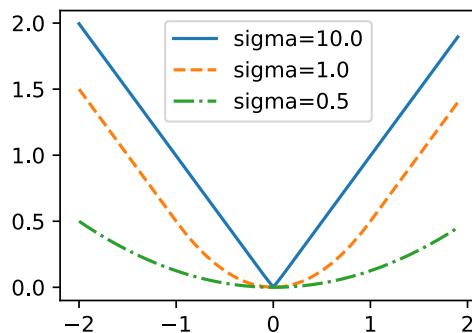
$$f(x) = \begin{cases} (\sigma x)^2 / 2, & \text{if } |x| < 1/\sigma^2 \\ |x| - 0.5/\sigma^2, & \text{otherwise} \end{cases} \quad (13.7.1)$$

当 σ 非常大时，这种损失类似于 L_1 范数损失。当它的值较小时，损失函数较平滑。

```
def smooth_l1(data, scalar):
    out = []
    for i in data:
        if abs(i) < 1 / (scalar ** 2):
            out.append(((scalar * i) ** 2) / 2)
        else:
            out.append(abs(i) - 0.5 / (scalar ** 2))
    return torch.tensor(out)

sigmas = [10, 1, 0.5]
lines = ['-', '--', '-.']
x = torch.arange(-2, 2, 0.1)
d2l.set_figsize()

for l, s in zip(lines, sigmas):
    y = smooth_l1(x, scalar=s)
    d2l.plt.plot(x, y, l, label='sigma=%1f' % s)
d2l.plt.legend();
```



此外，在类别预测时，实验中使用了交叉熵损失：设真实类别 j 的预测概率是 p_j ，交叉熵损失为 $-\log p_j$ 。我们还可以使用焦点损失 (Lin et al., 2017)。给定超参数 $\gamma > 0$ 和 $\alpha > 0$ ，此损失的定义为：

$$-\alpha(1 - p_j)^\gamma \log p_j. \quad (13.7.2)$$

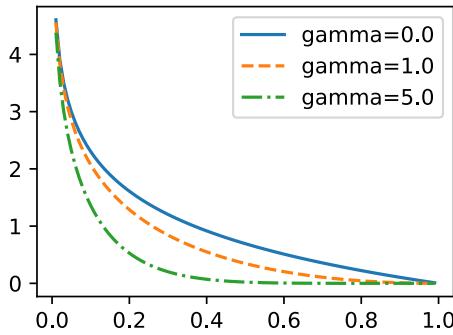
可以看到，增大 γ 可以有效地减少正类预测概率较大时（例如 $p_j > 0.5$ ）的相对损失，因此训练可以更集中在那些错误分类的困难示例上。

```

def focal_loss(gamma, x):
    return -(1 - x) ** gamma * torch.log(x)

x = torch.arange(0.01, 1, 0.01)
for l, gamma in zip(lines, [0, 1, 5]):
    y = d2l.plt.plot(x, focal_loss(gamma, x), l, label='gamma=%1f' % gamma)
d2l.plt.legend();

```



2. 由于篇幅限制，我们在本节中省略了单发多框检测模型的一些实现细节。能否从以下几个方面进一步改进模型：
 1. 当目标比图像小得多时，模型可以将输入图像调大；
 2. 通常会存在大量的负锚框。为了使类别分布更加平衡，我们可以将负锚框的高和宽减半；
 3. 在损失函数中，给类别损失和偏移损失设置不同比重的超参数；
 4. 使用其他方法评估目标检测模型，例如单发多框检测论文 (Liu *et al.*, 2016) 中的方法。

Discussions¹⁷⁷

13.8 区域卷积神经网络（R-CNN）系列

除了 13.7 节中描述的单发多框检测之外，区域卷积神经网络（region-based CNN 或 regions with CNN features, R-CNN）(Girshick *et al.*, 2014) 也是将深度模型应用于目标检测的开创性工作之一。本节将介绍 R-CNN 及其一系列改进方法：快速的 R-CNN（Fast R-CNN）(Girshick, 2015)、更快的 R-CNN（Faster R-CNN）(Ren *et al.*, 2015) 和掩码 R-CNN（Mask R-CNN）(He *et al.*, 2017)。限于篇幅，我们只着重介绍这些模型的设计思路。

¹⁷⁷ <https://discuss.d2l.ai/t/3204>

13.8.1 R-CNN

R-CNN首先从输入图像中选取若干（例如2000个）提议区域（如锚框也是一种选取方法），并标注它们的类别和边界框（如偏移量）。(Girshick *et al.*, 2014)然后，用卷积神经网络对每个提议区域进行前向传播以抽取其特征。接下来，我们用每个提议区域的特征来预测类别和边界框。

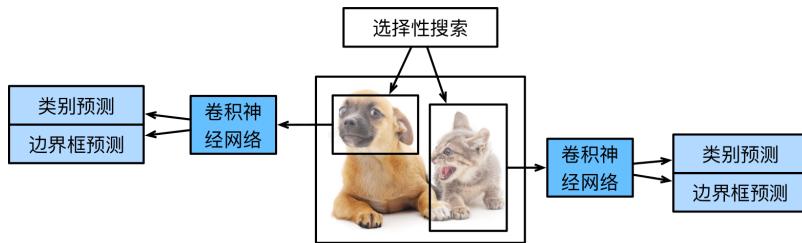


图13.8.1: R-CNN模型

图13.8.1展示了R-CNN模型。具体来说，R-CNN包括以下四个步骤：

1. 对输入图像使用选择性搜索来选取多个高质量的提议区域 (Uijlings *et al.*, 2013)。这些提议区域通常是在多个尺度下选取的，并具有不同的形状和大小。每个提议区域都将被标注类别和真实边界框；
2. 选择一个预训练的卷积神经网络，并将其在输出层之前截断。将每个提议区域变形为网络需要的输入尺寸，并通过前向传播输出抽取的提议区域特征；
3. 将每个提议区域的特征连同其标注的类别作为一个样本。训练多个支持向量机对目标分类，其中每个支持向量机用来判断样本是否属于某一个类别；
4. 将每个提议区域的特征连同其标注的边界框作为一个样本，训练线性回归模型来预测真实边界框。

尽管R-CNN模型通过预训练的卷积神经网络有效地抽取了图像特征，但它的速度很慢。想象一下，我们可能从一张图像中选出上千个提议区域，这需要上千次的卷积神经网络的前向传播来执行目标检测。这种庞大的计算量使得R-CNN在现实世界中难以被广泛应用。

13.8.2 Fast R-CNN

R-CNN的主要性能瓶颈在于，对每个提议区域，卷积神经网络的前向传播是独立的，而没有共享计算。由于这些区域通常有重叠，独立的特征抽取会导致重复的计算。*Fast R-CNN* (Girshick, 2015)对R-CNN的主要改进之一，是仅在整张图象上执行卷积神经网络的前向传播。

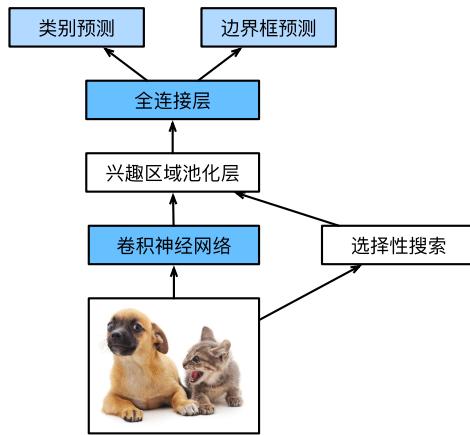


图13.8.2: Fast R-CNN模型

图13.8.2中描述了Fast R-CNN模型。它的主要计算如下：

1. 与R-CNN相比，Fast R-CNN用来提取特征的卷积神经网络的输入是整个图像，而不是各个提议区域。此外，这个网络通常会参与训练。设输入为一张图像，将卷积神经网络的输出的形状记为 $1 \times c \times h_1 \times w_1$ ；
2. 假设选择性搜索生成了 n 个提议区域。这些形状各异的提议区域在卷积神经网络的输出上分别标出了形状各异的兴趣区域。然后，这些感兴趣的区域需要进一步抽取出形状相同的特征（比如指定高度 h_2 和宽度 w_2 ），以便于连结后输出。为了实现这一目标，Fast R-CNN引入了兴趣区域汇聚层（RoI pooling）：将卷积神经网络的输出和提议区域作为输入，输出连结后的各个提议区域抽取的特征，形状为 $n \times c \times h_2 \times w_2$ ；
3. 通过全连接层将输出形状变换为 $n \times d$ ，其中超参数 d 取决于模型设计；
4. 预测 n 个提议区域中每个区域的类别和边界框。更具体地说，在预测类别和边界框时，将全连接层的输出分别转换为形状为 $n \times q$ （ q 是类别的数量）的输出和形状为 $n \times 4$ 的输出。其中预测类别时使用softmax回归。

在Fast R-CNN中提出的兴趣区域汇聚层与 6.5节中介绍的汇聚层有所不同。在汇聚层中，我们通过设置汇聚窗口、填充和步幅的大小来间接控制输出形状。而兴趣区域汇聚层对每个区域的输出形状是可以直接指定的。

例如，指定每个区域输出的高和宽分别为 h_2 和 w_2 。对于任何形状为 $h \times w$ 的兴趣区域窗口，该窗口将被划分为 $h_2 \times w_2$ 子窗口网格，其中每个子窗口的大小约为 $(h/h_2) \times (w/w_2)$ 。在实践中，任何子窗口的高度和宽度都应向上取整，其中的最大元素作为该子窗口的输出。因此，兴趣区域汇聚层可从形状各异的兴趣区域中均抽取出形状相同的特征。

作为说明性示例，图13.8.3中提到，在 4×4 的输入中，我们选取了左上角 3×3 的兴趣区域。对于该兴趣区域，我们通过 2×2 的兴趣区域汇聚层得到一个 2×2 的输出。请注意，四个划分后的子窗口中分别含有元素0、1、4、5（5最大）；2、6（6最大）；8、9（9最大）；以及10。



图13.8.3: 一个 2×2 的兴趣区域汇聚层

下面，我们演示了兴趣区域汇聚层的计算方法。假设卷积神经网络抽取的特征 X 的高度和宽度都是4，且只有单通道。

```
import torch
import torchvision

X = torch.arange(16.).reshape(1, 1, 4, 4)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

让我们进一步假设输入图像的高度和宽度都是40像素，且选择性搜索在此图像上生成了两个提议区域。每个区域由5个元素表示：区域目标类别、左上角和右下角的 (x, y) 坐标。

```
rois = torch.Tensor([[0, 0, 0, 20, 20], [0, 0, 10, 30, 30]])
```

由于 X 的高和宽是输入图像高和宽的 $1/10$ ，因此，两个提议区域的坐标先按`spatial_scale`乘以 0.1 。然后，在 X 上分别标出这两个兴趣区域 $X[:, :, 0:3, 0:3]$ 和 $X[:, :, 1:4, 0:4]$ 。最后，在 2×2 的兴趣区域汇聚层中，每个兴趣区域被划分为子窗口网格，并进一步抽取相同形状 2×2 的特征。

```
torchvision.ops.roi_pool(X, rois, output_size=(2, 2), spatial_scale=0.1)

tensor([[[[ 5.,  6.],
          [ 9., 10.]],

         [[ 9., 11.],
          [13., 15.]]]])
```

13.8.3 Faster R-CNN

为了较精确地检测目标结果，Fast R-CNN模型通常需要在选择性搜索中生成大量的提议区域。Faster R-CNN (Ren et al., 2015)提出将选择性搜索替换为区域提议网络（region proposal network），从而减少提议区域的生成数量，并保证目标检测的精度。

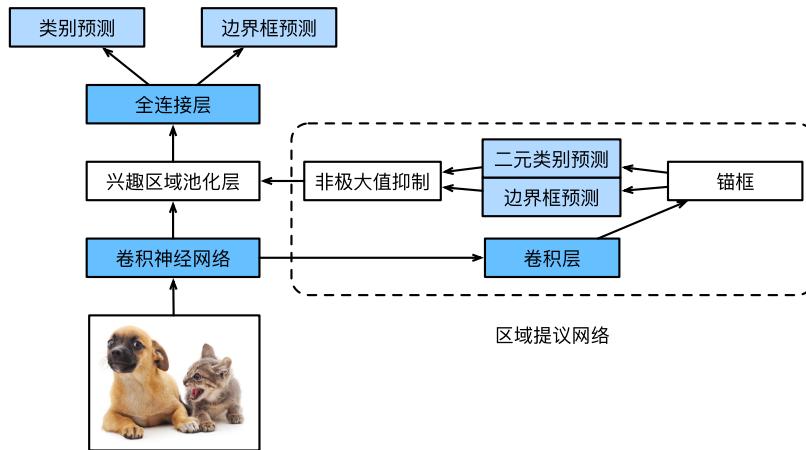


图13.8.4: Faster R-CNN 模型

图13.8.4描述了Faster R-CNN模型。与Fast R-CNN相比，Faster R-CNN只将生成提议区域的方法从选择性搜索改为了区域提议网络，模型的其余部分保持不变。具体来说，区域提议网络的计算步骤如下：

1. 使用填充为1的 3×3 的卷积层变换卷积神经网络的输出，并将输出通道数记为 c 。这样，卷积神经网络为图像抽取的特征图中的每个单元均得到一个长度为 c 的新特征。
2. 以特征图的每个像素为中心，生成多个不同大小和宽高比的锚框并标注它们。
3. 使用锚框中心单元长度为 c 的特征，分别预测该锚框的二元类别（含目标还是背景）和边界框。
4. 使用非极大值抑制，从预测类别为目标的预测边界框中移除相似的结果。最终输出的预测边界框即是兴趣区域汇聚层所需的提议区域。

值得一提的是，区域提议网络作为Faster R-CNN模型的一部分，是和整个模型一起训练得到的。换句话说，Faster R-CNN的目标函数不仅包括目标检测中的类别和边界框预测，还包括区域提议网络中锚框的二元类别和边界框预测。作为端到端训练的结果，区域提议网络能够学习到如何生成高质量的提议区域，从而在减少了从数据中学到的提议区域的数量的情况下，仍保持目标检测的精度。

13.8.4 Mask R-CNN

如果在训练集中还标注了每个目标在图像上的像素级位置，那么Mask R-CNN (He et al., 2017)能够有效地利用这些详尽的标注信息进一步提升目标检测的精度。

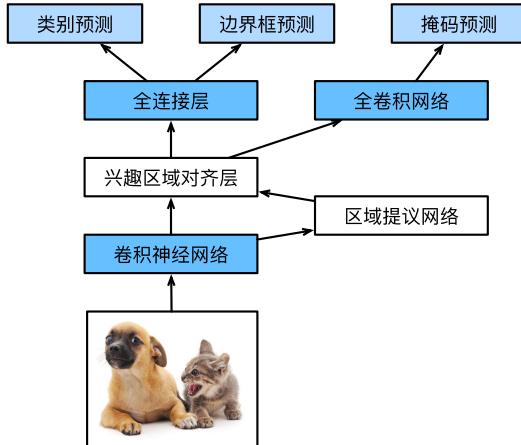


图13.8.5: Mask R-CNN 模型

如图13.8.5所示，Mask R-CNN是基于Faster R-CNN修改而来的。具体来说，Mask R-CNN将兴趣区域汇聚层替换为了兴趣区域对齐层，使用双线性插值（bilinear interpolation）来保留特征图上的空间信息，从而更适合于像素级预测。兴趣区域对齐层的输出包含了所有与兴趣区域的形状相同的特征图。它们不仅被用于预测每个兴趣区域的类别和边界框，还通过额外的全卷积网络预测目标的像素级位置。本章的后续章节将更详细地介绍如何使用全卷积网络预测图像中像素级的语义。

小结

- R-CNN对图像选取若干提议区域，使用卷积神经网络对每个提议区域执行前向传播以抽取其特征，然后再用这些特征来预测提议区域的类别和边界框。
- Fast R-CNN对R-CNN的一个主要改进：只对整个图像做卷积神经网络的前向传播。它还引入了兴趣区域汇聚层，从而为具有不同形状的兴趣区域抽取相同形状的特征。
- Faster R-CNN将Fast R-CNN中使用的选择性搜索替换为参与训练的区域提议网络，这样后者可以在减少提议区域数量的情况下仍保证目标检测的精度。
- Mask R-CNN在Faster R-CNN的基础上引入了一个全卷积网络，从而借助目标的像素级位置进一步提升目标检测的精度。

练习

1. 我们能否将目标检测视为回归问题（例如预测边界框和类别的概率）？可以参考YOLO模型 (Redmon *et al.*, 2016)的设计。
2. 将单发多框检测与本节介绍的方法进行比较。他们的主要区别是什么？可以参考 (Zhao *et al.*, 2019)中的图2。

讨论区¹⁷⁸

13.9 语义分割和数据集

在 13.3 节—13.8 节中讨论的目标检测问题中，我们一直使用方形边界框来标注和预测图像中的目标。本节将探讨语义分割 (semantic segmentation) 问题，它重点关注于如何将图像分割成属于不同语义类别的区域。与目标检测不同，语义分割可以识别并理解图像中每一个像素的内容：其语义区域的标注和预测是像素级的。图13.9.1展示了语义分割中图像有关狗、猫和背景的标签。与目标检测相比，语义分割标注的像素级的边框显然更加精细。

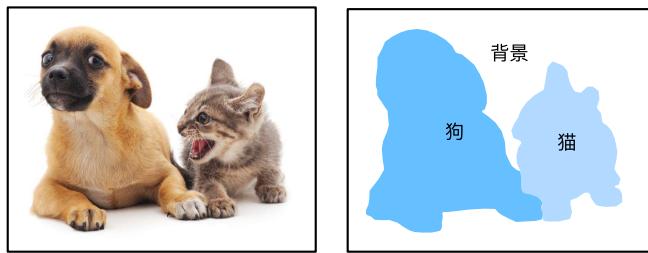


图13.9.1: 语义分割中图像有关狗、猫和背景的标签

13.9.1 图像分割和实例分割

计算机视觉领域还有2个与语义分割相似的重要问题，即图像分割(image segmentation)和实例分割(instance segmentation)。我们在这里将它们同语义分割简单区分一下。

- 图像分割将图像划分为若干组成区域，这类问题的方法通常利用图像中像素之间的相关性。它在训练时不需要有关图像像素的标签信息，在预测时也无法保证分割出的区域具有我们希望得到的语义。以图13.9.1中的图像作为输入，图像分割可能会将狗分为两个区域：一个覆盖以黑色为主的嘴和眼睛，另一个覆盖以黄色为主的其余部分身体。
- 实例分割也叫同时检测并分割 (simultaneous detection and segmentation)，它研究如何识别图像中各个目标实例的像素级区域。与语义分割不同，实例分割不仅需要区分语义，还要区分不同的目标实例。例如，如果图像中有两条狗，则实例分割需要区分像素属于的两条狗中的哪一条。

¹⁷⁸ <https://discuss.d2l.ai/t/3207>

13.9.2 Pascal VOC2012 语义分割数据集

最重要的语义分割数据集之一是Pascal VOC2012¹⁷⁹。下面我们深入了解一下这个数据集。

```
%matplotlib inline
import os
import torch
import torchvision
from d2l import torch as d2l
```

数据集的tar文件大约为2GB,所以下载可能需要一段时间。提取出的数据集位于`..../data/VOCdevkit/VOC2012`。

```
#@save
d2l.DATA_HUB['voc2012'] = (d2l.DATA_URL + 'VOCtrainval_11-May-2012.tar',
                            '4e443f8a2eca6b1dac8a6c57641b67dd40621a49')

voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
```

```
Downloading ..../data/VOCtrainval_11-May-2012.tar from http://d2l-data.s3-accelerate.amazonaws.com/
→ VOCtrainval_11-May-2012.tar...
```

进入路径`..../data/VOCdevkit/VOC2012`之后，我们可以看到数据集的不同组件。`ImageSets/Segmentation`路径包含用于训练和测试样本的文本文件，而`JPEGImages`和`SegmentationClass`路径分别存储着每个示例的输入图像和标签。此处的标签也采用图像格式，其尺寸和它所标注的输入图像的尺寸相同。此外，标签中颜色相同的像素属于同一个语义类别。下面将`read_voc_images`函数定义为将所有输入的图像和标签读入内存。

```
#@save
def read_voc_images(voc_dir, is_train=True):
    """读取所有VOC图像并标注"""
    txt_fname = os.path.join(voc_dir, 'ImageSets', 'Segmentation',
                            'train.txt' if is_train else 'val.txt')
    mode = torchvision.io.image.ImageReadMode.RGB
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    features, labels = [], []
    for i, fname in enumerate(images):
        features.append(torchvision.io.read_image(os.path.join(
            voc_dir, 'JPEGImages', f'{fname}.jpg')))
        labels.append(torchvision.io.read_image(os.path.join(
            voc_dir, 'SegmentationClass', f'{fname}.png'), mode))
    return features, labels
```

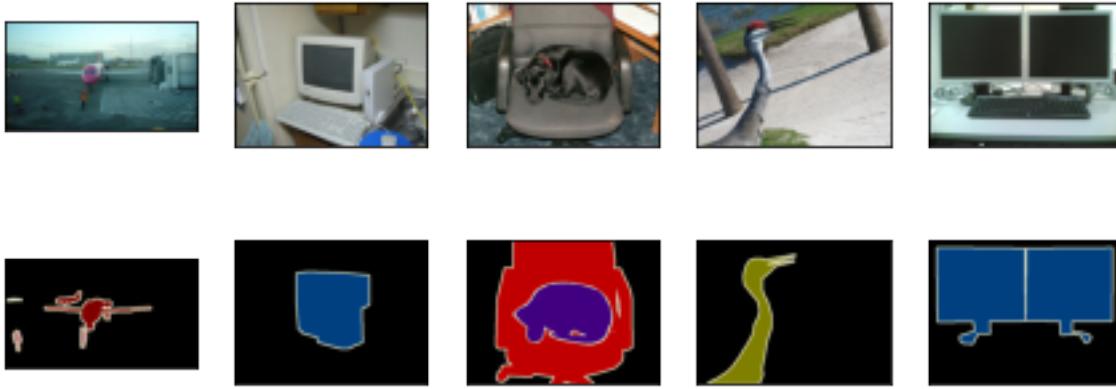
(continues on next page)

¹⁷⁹ <http://host.robots.ox.ac.uk/pascal/VOC/voc2012/>

```
train_features, train_labels = read_voc_images(voc_dir, True)
```

下面我们绘制前5个输入图像及其标签。在标签图像中，白色和黑色分别表示边框和背景，而其他颜色则对应不同的类别。

```
n = 5
imgs = train_features[0:n] + train_labels[0:n]
imgs = [img.permute(1,2,0) for img in imgs]
d2l.show_images(imgs, 2, n);
```



接下来，我们列举RGB颜色值和类名。

```
#@save
VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],
                 [0, 0, 128], [128, 0, 128], [0, 128, 128], [128, 128, 128],
                 [64, 0, 0], [192, 0, 0], [64, 128, 0], [192, 128, 0],
                 [64, 0, 128], [192, 0, 128], [64, 128, 128], [192, 128, 128],
                 [0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],
                 [0, 64, 128]]
```



```
#@save
VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
                'bottle', 'bus', 'car', 'cat', 'chair', 'cow',
                'diningtable', 'dog', 'horse', 'motorbike', 'person',
                'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']
```

通过上面定义的两个常量，我们可以方便地查找标签中每个像素的类索引。我们定义了voc_colormap2label函数来构建从上述RGB颜色值到类别索引的映射，而voc_label_indices函数将RGB值映射到在Pascal VOC2012数据集中的类别索引。

```

#@save
def voc_colormap2label():
    """构建从RGB到VOC类别索引的映射"""
    colormap2label = torch.zeros(256 ** 3, dtype=torch.long)
    for i, colormap in enumerate(VOC_COLORMAP):
        colormap2label[
            (colormap[0] * 256 + colormap[1]) * 256 + colormap[2]] = i
    return colormap2label

#@save
def voc_label_indices(colormap, colormap2label):
    """将VOC标签中的RGB值映射到它们的类别索引"""
    colormap = colormap.permute(1, 2, 0).numpy().astype('int32')
    idx = ((colormap[:, :, 0] * 256 + colormap[:, :, 1]) * 256
           + colormap[:, :, 2])
    return colormap2label[idx]

```

例如，在第一张样本图像中，飞机头部区域的类别索引为1，而背景索引为0。

```

y = voc_label_indices(train_labels[0], voc_colormap2label())
y[105:115, 130:140], VOC_CLASSES[1]

```

```

(tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
         [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1]]),
'aeroplane')

```

预处理数据

在之前的实验，例如 7.1节—7.4节 中，我们通过再缩放图像使其符合模型的输入形状。然而在语义分割中，这样做需要将预测的像素类别重新映射回原始尺寸的输入图像。这样的映射可能不够精确，尤其在不同语义的分割区域。为了避免这个问题，我们将图像裁剪为固定尺寸，而不是再缩放。具体来说，我们使用图像增广中的随机裁剪，裁剪输入图像和标签的相同区域。

```

#@save
def voc_rand_crop(feature, label, height, width):
    """随机裁剪特征和标签图像"""
    rect = torchvision.transforms.RandomCrop.get_params(
        feature, (height, width))
    feature = torchvision.transforms.functional.crop(feature, *rect)
    label = torchvision.transforms.functional.crop(label, *rect)
    return feature, label

```

```

imgs = []
for _ in range(n):
    imgs += voc_rand_crop(train_features[0], train_labels[0], 200, 300)

imgs = [img.permute(1, 2, 0) for img in imgs]
d2l.show_images(imgs[::2] + imgs[1::2], 2, n);

```



自定义语义分割数据集类

我们通过继承高级API提供的Dataset类，自定义了一个语义分割数据集类VOCSegDataset。通过实现`__getitem__`函数，我们可以任意访问数据集中索引为idx的输入图像及其每个像素的类别索引。由于数据集中有些图像的尺寸可能小于随机裁剪所指定的输出尺寸，这些样本可以通过自定义的`filter`函数移除掉。此外，我们还定义了`normalize_image`函数，从而对输入图像的RGB三个通道的值分别做标准化。

```

#@save
class VOCSegDataset(torch.utils.data.Dataset):
    """一个用于加载voc数据集的自定义数据集"""

    def __init__(self, is_train, crop_size, voc_dir):
        self.transform = torchvision.transforms.Normalize(
            mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

```

(continues on next page)

(continued from previous page)

```
self.crop_size = crop_size
features, labels = read_voc_images(voc_dir, is_train=is_train)
self.features = [self.normalize_image(feature)
                 for feature in self.filter(features)]
self.labels = self.filter(labels)
self.colormap2label = voc_colormap2label()
print('read ' + str(len(self.features)) + ' examples')

def normalize_image(self, img):
    return self.transform(img.float() / 255)

def filter(self, imgs):
    return [img for img in imgs if (
        img.shape[1] >= self.crop_size[0] and
        img.shape[2] >= self.crop_size[1])]

def __getitem__(self, idx):
    feature, label = voc_rand_crop(self.features[idx], self.labels[idx],
                                    *self.crop_size)
    return (feature, voc_label_indices(label, self.colormap2label))

def __len__(self):
    return len(self.features)
```

读取数据集

我们通过自定义的VOCSegDataset类来分别创建训练集和测试集的实例。假设我们指定随机裁剪的输出图像的形状为 320×480 ，下面我们可以查看训练集和测试集所保留的样本个数。

```
crop_size = (320, 480)
voc_train = VOCSegDataset(True, crop_size, voc_dir)
voc_test = VOCSegDataset(False, crop_size, voc_dir)
```

```
read 1114 examples
read 1078 examples
```

设批量大小为64，我们定义训练集的迭代器。打印第一个小批量的形状会发现：与图像分类或目标检测不同，这里的标签是一个三维数组。

```
batch_size = 64
train_iter = torch.utils.data.DataLoader(voc_train, batch_size, shuffle=True,
```

(continues on next page)

(continued from previous page)

```
        drop_last=True,
        num_workers=d2l.get_dataloader_workers())

for X, Y in train_iter:
    print(X.shape)
    print(Y.shape)
    break
```

```
torch.Size([64, 3, 320, 480])
torch.Size([64, 320, 480])
```

整合所有组件

最后，我们定义以下load_data_voc函数来下载并读取Pascal VOC2012语义分割数据集。它返回训练集和测试集的数据迭代器。

```
#@save
def load_data_voc(batch_size, crop_size):
    """加载VOC语义分割数据集"""
    voc_dir = d2l.download_extract('voc2012', os.path.join(
        'VOCdevkit', 'VOC2012'))
    num_workers = d2l.get_dataloader_workers()
    train_iter = torch.utils.data.DataLoader(
        VOCSegDataset(True, crop_size, voc_dir), batch_size,
        shuffle=True, drop_last=True, num_workers=num_workers)
    test_iter = torch.utils.data.DataLoader(
        VOCSegDataset(False, crop_size, voc_dir), batch_size,
        drop_last=True, num_workers=num_workers)
    return train_iter, test_iter
```

小结

- 语义分割通过将图像划分为属于不同语义类别的区域，来识别并理解图像中像素级别的内容。
- 语义分割的一个重要的数据集叫做Pascal VOC2012。
- 由于语义分割的输入图像和标签在像素上一一对应，输入图像会被随机裁剪为固定尺寸而不是缩放。

练习

1. 如何在自动驾驶和医疗图像诊断中应用语义分割？还能想到其他领域的应用吗？
2. 回想一下 13.1 节中对数据增强的描述。图像分类中使用的哪种图像增强方法是难以用于语义分割的？

Discussions¹⁸⁰

13.10 转置卷积

到目前为止，我们所见到的卷积神经网络层，例如卷积层（6.2节）和汇聚层（6.5节），通常会减少下采样输入图像的空间维度（高和宽）。然而如果输入和输出图像的空间维度相同，在以像素级分类的语义分割中将会很方便。例如，输出像素所处的通道维可以保有输入像素在同一位置上的分类结果。

为了实现这一点，尤其是在空间维度被卷积神经网络层缩小后，我们可以使用另一种类型的卷积神经网络层，它可以增加上采样中间层特征图的空间维度。本节将介绍 转置卷积（transposed convolution）(Dumoulin and Visin, 2016)，用于逆转下采样导致的空间尺寸减小。

```
import torch
from torch import nn
from d2l import torch as d2l
```

13.10.1 基本操作

让我们暂时忽略通道，从基本的转置卷积开始，设步幅为1且没有填充。假设我们有一个 $n_h \times n_w$ 的输入张量和一个 $k_h \times k_w$ 的卷积核。以步幅为1滑动卷积核窗口，每行 n_w 次，每列 n_h 次，共产生 $n_h n_w$ 个中间结果。每个中间结果都是一个 $(n_h + k_h - 1) \times (n_w + k_w - 1)$ 的张量，初始化为0。为了计算每个中间张量，输入张量中的每个元素都要乘以卷积核，从而使所得的 $k_h \times k_w$ 张量替换中间张量的一部分。请注意，每个中间张量被替换部分的位置与输入张量中元素的位置相对应。最后，所有中间结果相加以获得最终结果。

例如，图13.10.1解释了如何为 2×2 的输入张量计算卷积核为 2×2 的转置卷积。

¹⁸⁰ <https://discuss.d2l.ai/t/3295>

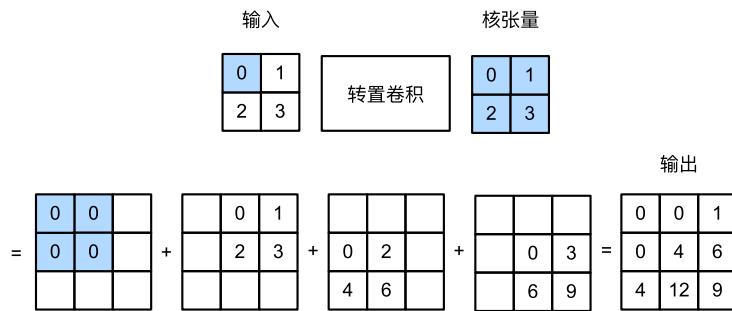


图13.10.1: 卷积核为 2×2 的转置卷积。阴影部分是中间张量的一部分，也是用于计算的输入和卷积核张量元素。

我们可以对输入矩阵 X 和卷积核矩阵 K 实现基本的转置卷积运算`trans_conv`。

```
def trans_conv(X, K):
    h, w = K.shape
    Y = torch.zeros((X.shape[0] + h - 1, X.shape[1] + w - 1))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Y[i: i + h, j: j + w] += X[i, j] * K
    return Y
```

与通过卷积核“减少”输入元素的常规卷积（在 6.2 节中）相比，转置卷积通过卷积核“广播”输入元素，从而产生大于输入的输出。我们可以通过 图13.10.1 来构建输入张量 X 和卷积核张量 K 从而验证上述实现输出。此实现是基本的二维转置卷积运算。

```
X = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
trans_conv(X, K)
```

```
tensor([[ 0.,  0.,  1.],
       [ 0.,  4.,  6.],
       [ 4., 12.,  9.]])
```

或者，当输入 X 和卷积核 K 都是四维张量时，我们可以使用高级API获得相同的结果。

```
X, K = X.reshape(1, 1, 2, 2), K.reshape(1, 1, 2, 2)
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[ 0.,  0.,  1.],
          [ 0.,  4.,  6.],
          [ 4., 12.,  9.]]]], grad_fn=<ConvolutionBackward0>)
```

13.10.2 填充、步幅和多通道

与常规卷积不同，在转置卷积中，填充被应用于的输出（常规卷积将填充应用于输入）。例如，当将高和宽两侧的填充数指定为1时，转置卷积的输出中将删除第一和最后的行与列。

```
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, padding=1, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[4.]]]], grad_fn=<ConvolutionBackward0>)
```

在转置卷积中，步幅被指定为中间结果（输出），而不是输入。使用 图13.10.1 中相同输入和卷积核张量，将步幅从1更改为2会增加中间张量的高和权重，因此输出张量在 图13.10.2 中。

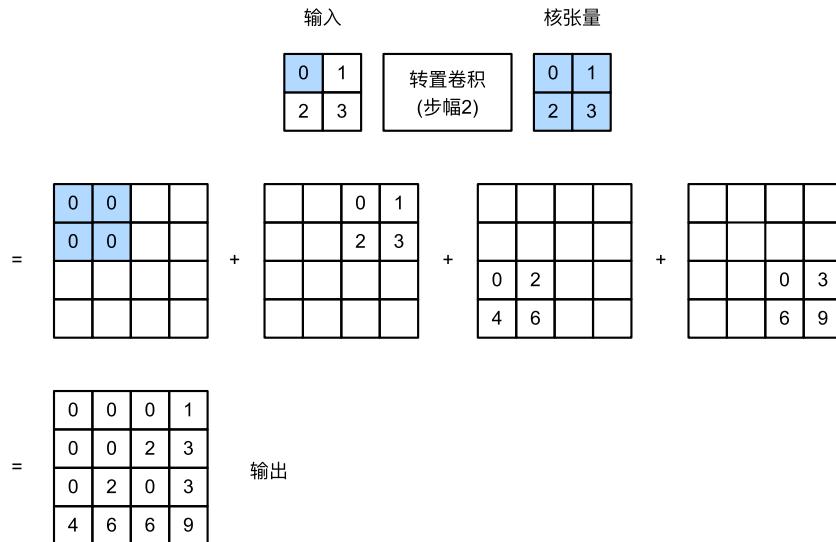


图13.10.2: 卷积核为 2×2 ，步幅为2的转置卷积。阴影部分是中间张量的一部分，也是用于计算的输入和卷积核张量元素。

以下代码可以验证 图13.10.2 中步幅为2的转置卷积的输出。

```
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, stride=2, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[0., 0., 0., 1.],
          [0., 0., 2., 3.],
          [0., 2., 0., 3.],
          [4., 6., 6., 9.]]]], grad_fn=<ConvolutionBackward0>)
```

对于多个输入和输出通道，转置卷积与常规卷积以相同方式运作。假设输入有 c_i 个通道，且转置卷积为每个输入通道分配了一个 $k_h \times k_w$ 的卷积核张量。当指定多个输出通道时，每个输出通道将有一个 $c_i \times k_h \times k_w$ 的卷积核。

同样，如果我们将 X 代入卷积层 f 来输出 $Y = f(X)$ ，并创建一个与 f 具有相同的超参数、但输出通道数量是 X 中通道数的转置卷积层 g ，那么 $g(Y)$ 的形状将与 X 相同。下面的示例可以解释这一点。

```
X = torch.rand(size=(1, 10, 16, 16))
conv = nn.Conv2d(10, 20, kernel_size=5, padding=2, stride=3)
tconv = nn.ConvTranspose2d(20, 10, kernel_size=5, padding=2, stride=3)
tconv(conv(X)).shape == X.shape
```

```
True
```

13.10.3 与矩阵变换的联系

转置卷积为何以矩阵变换命名呢？让我们首先看看如何使用矩阵乘法来实现卷积。在下面的示例中，我们定义了一个 3×3 的输入 X 和 2×2 卷积核 K ，然后使用`corr2d`函数计算卷积输出 Y 。

```
X = torch.arange(9.0).reshape(3, 3)
K = torch.tensor([[1.0, 2.0], [3.0, 4.0]])
Y = d2l.corr2d(X, K)
Y
```

```
tensor([[27., 37.],
        [57., 67.]])
```

接下来，我们将卷积核 K 重写为包含大量0的稀疏权重矩阵 W 。权重矩阵的形状是 $(4, 9)$ ，其中非0元素来自卷积核 K 。

```
def kernel2matrix(K):
    k, W = torch.zeros(5), torch.zeros((4, 9))
    k[:2], k[3:5] = K[0, :], K[1, :]
    W[0, :5], W[1, 1:6], W[2, 3:8], W[3, 4:] = k, k, k, k
    return W
```

(continues on next page)

(continued from previous page)

```
W = kernel2matrix(K)
W
```

```
tensor([[1., 2., 0., 3., 4., 0., 0., 0., 0.],
       [0., 1., 2., 0., 3., 4., 0., 0., 0.],
       [0., 0., 0., 1., 2., 0., 3., 4., 0.],
       [0., 0., 0., 0., 1., 2., 0., 3., 4.]])
```

逐行连结输入 x ，获得了一个长度为9的矢量。然后， w 的矩阵乘法和向量化的 x 给出了一个长度为4的向量。重塑它之后，可以获得与上面的原始卷积操作所得相同的结果 y ：我们刚刚使用矩阵乘法实现了卷积。

```
Y == torch.matmul(W, X.reshape(-1)).reshape(2, 2)
```

```
tensor([[True, True],
       [True, True]])
```

同样，我们可以使用矩阵乘法来实现转置卷积。在下面的示例中，我们将上面的常规卷积 2×2 的输出 y 作为转置卷积的输入。想要通过矩阵相乘来实现它，我们只需要将权重矩阵 w 的形状转置为 $(9, 4)$ 。

```
Z = trans_conv(Y, K)
Z == torch.matmul(W.T, Y.reshape(-1)).reshape(3, 3)
```

```
tensor([[True, True, True],
       [True, True, True],
       [True, True, True]])
```

抽象来看，给定输入向量 x 和权重矩阵 W ，卷积的前向传播函数可以通过将其输入与权重矩阵相乘并输出向量 $y = Wx$ 来实现。由于反向传播遵循链式法则且 $\nabla_{\mathbf{x}} \mathbf{y} = \mathbf{W}^T$ ，卷积的反向传播函数可以通过将其输入与转置的权重矩阵 \mathbf{W}^T 相乘来实现。因此，转置卷积层能够交换卷积层的正向传播函数和反向传播函数：它的正向传播和反向传播函数将输入向量分别与 \mathbf{W}^T 和 \mathbf{W} 相乘。

小结

- 与通过卷积核减少输入元素的常规卷积相反，转置卷积通过卷积核广播输入元素，从而产生形状大于输入的输出。
- 如果我们将 x 输入卷积层 f 来获得输出 $y = f(x)$ 并创造一个与 f 有相同的超参数、但输出通道数是 x 中通道数的转置卷积层 g ，那么 $g(y)$ 的形状将与 x 相同。
- 我们可以使用矩阵乘法来实现卷积。转置卷积层能够交换卷积层的正向传播函数和反向传播函数。

练习

1. 在 13.10.3 节中，卷积输入 x 和转置的卷积输出 z 具有相同的形状。他们的数值也相同吗？为什么？
2. 使用矩阵乘法来实现卷积是否有效率？为什么？

Discussions¹⁸¹

13.11 全卷积网络

如 13.9 节中所介绍的那样，语义分割是对图像中的每个像素分类。全卷积网络（fully convolutional network, FCN）采用卷积神经网络实现了从图像像素到像素类别的变换 (Long *et al.*, 2015)。与我们之前在图像分类或目标检测部分介绍的卷积神经网络不同，全卷积网络将中间层特征图的高和宽变换回输入图像的尺寸：这是通过在 13.10 节中引入的转置卷积（transposed convolution）实现的。因此，输出的类别预测与输入图像在像素级别上具有一一对应关系：通道维的输出即该位置对应像素的类别预测。

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

13.11.1 构造模型

下面我们了解一下全卷积网络模型最基本的设计。如 图13.11.1 所示，全卷积网络先使用卷积神经网络抽取图像特征，然后通过 1×1 卷积层将通道数变换为类别个数，最后在 13.10 节中通过转置卷积层将特征图的高和宽变换为输入图像的尺寸。因此，模型输出与输入图像的高和宽相同，且最终输出通道包含了该空间位置像素的类别预测。

¹⁸¹ <https://discuss.d2l.ai/t/3302>



图13.11.1: 全卷积网络

下面，我们使用在ImageNet数据集上预训练的ResNet-18模型来提取图像特征，并将该网络记为pretrained_net。ResNet-18模型的最后几层包括全局平均汇聚层和全连接层，然而全卷积网络中不需要它们。

```
pretrained_net = torchvision.models.resnet18(pretrained=True)
list(pretrained_net.children())[-3:]
```

```
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /home/ci/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth
```

```
0% | 0.00/44.7M [00:00<?, ?B/s]
```

```
[Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  )
)
```

(continues on next page)

(continued from previous page)

```
(bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu): ReLU(inplace=True)
(conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
),
AdaptiveAvgPool2d(output_size=(1, 1)),
Linear(in_features=512, out_features=1000, bias=True)]
```

接下来，我们创建一个全卷积网络net。它复制了ResNet-18中大部分的预训练层，除了最后的全局平均汇聚层和最接近输出的全连接层。

```
net = nn.Sequential(*list(pretrained_net.children())[:-2])
```

给定高度为320和宽度为480的输入，net的前向传播将输入的高和宽减小至原来的1/32，即10和15。

```
X = torch.rand(size=(1, 3, 320, 480))
net(X).shape
```

```
torch.Size([1, 512, 10, 15])
```

接下来使用 1×1 卷积层将输出通道数转换为Pascal VOC2012数据集的类数（21类）。最后需要将特征图的高度和宽度增加32倍，从而将其变回输入图像的高和宽。回想一下6.3节中卷积层输出形状的计算方法：由于 $(320 - 64 + 16 \times 2 + 32)/32 = 10$ 且 $(480 - 64 + 16 \times 2 + 32)/32 = 15$ ，我们构造一个步幅为32的转置卷积层，并将卷积核的高和宽设为64，填充为16。我们可以看到如果步幅为s，填充为s/2（假设s/2是整数）且卷积核的高和宽为2s，转置卷积核会将输入的高和宽分别放大s倍。

```
num_classes = 21
net.add_module('final_conv', nn.Conv2d(512, num_classes, kernel_size=1))
net.add_module('transpose_conv', nn.ConvTranspose2d(num_classes, num_classes,
                                                 kernel_size=64, padding=16, stride=32))
```

13.11.2 初始化转置卷积层

在图像处理中，我们有时需要将图像放大，即上采样（upsampling）。双线性插值（bilinear interpolation）是常用的上采样方法之一，它也经常用于初始化转置卷积层。

为了解释双线性插值，假设给定输入图像，我们想要计算上采样输出图像上的每个像素。

1. 将输出图像的坐标 (x, y) 映射到输入图像的坐标 (x', y') 上。例如，根据输入与输出的尺寸之比来映射。请注意，映射后的 x' 和 y' 是实数。

2. 在输入图像上找到离坐标 (x', y') 最近的4个像素。
3. 输出图像在坐标 (x, y) 上的像素依据输入图像上这4个像素及其与 (x', y') 的相对距离来计算。

双线性插值的上采样可以通过转置卷积层实现，内核由以下**bilinear_kernel**函数构造。限于篇幅，我们只给出**bilinear_kernel**函数的实现，不讨论算法的原理。

```
def bilinear_kernel(in_channels, out_channels, kernel_size):
    factor = (kernel_size + 1) // 2
    if kernel_size % 2 == 1:
        center = factor - 1
    else:
        center = factor - 0.5
    og = (torch.arange(kernel_size).reshape(-1, 1),
          torch.arange(kernel_size).reshape(1, -1))
    filt = (1 - torch.abs(og[0] - center) / factor) * \
           (1 - torch.abs(og[1] - center) / factor)
    weight = torch.zeros((in_channels, out_channels,
                          kernel_size, kernel_size))
    weight[range(in_channels), range(out_channels), :, :] = filt
    return weight
```

让我们用双线性插值的上采样实验它由转置卷积层实现。我们构造一个将输入的高和宽放大2倍的转置卷积层，并将其卷积核用**bilinear_kernel**函数初始化。

```
conv_trans = nn.ConvTranspose2d(3, 3, kernel_size=4, padding=1, stride=2,
                             bias=False)
conv_trans.weight.data.copy_(bilinear_kernel(3, 3, 4));
```

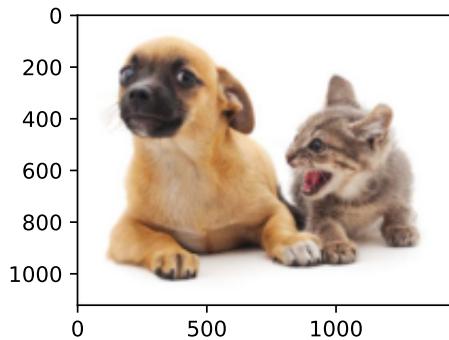
读取图像X，将上采样的结果记作Y。为了打印图像，我们需要调整通道维的位置。

```
img = torchvision.transforms.ToTensor()(d2l.Image.open('../img/catdog.jpg'))
X = img.unsqueeze(0)
Y = conv_trans(X)
out_img = Y[0].permute(1, 2, 0).detach()
```

可以看到，转置卷积层将图像的高和宽分别放大了2倍。除了坐标刻度不同，双线性插值放大的图像和在13.3节中打印出的原图看上去没什么两样。

```
d2l.set_figsize()
print('input image shape:', img.permute(1, 2, 0).shape)
d2l.plt.imshow(img.permute(1, 2, 0));
print('output image shape:', out_img.shape)
d2l.plt.imshow(out_img);
```

```
input image shape: torch.Size([561, 728, 3])
output image shape: torch.Size([1122, 1456, 3])
```



全卷积网络用双线性插值的上采样初始化转置卷积层。对于 1×1 卷积层，我们使用Xavier初始化参数。

```
W = bilinear_kernel(num_classes, num_classes, 64)
net.transpose_conv.weight.data.copy_(W);
```

13.11.3 读取数据集

我们用 13.9 节中介绍的语义分割读取数据集。指定随机裁剪的输出图像的形状为 320×480 ：高和宽都可以被32整除。

```
batch_size, crop_size = 32, (320, 480)
train_iter, test_iter = d2l.load_data_voc(batch_size, crop_size)
```

```
read 1114 examples
read 1078 examples
```

13.11.4 训练

现在我们可以训练全卷积网络了。这里的损失函数和准确率计算与图像分类中的并没有本质上的不同，因为我们使用转置卷积层的通道来预测像素的类别，所以需要在损失计算中指定通道维。此外，模型基于每个像素的预测类别是否正确来计算准确率。

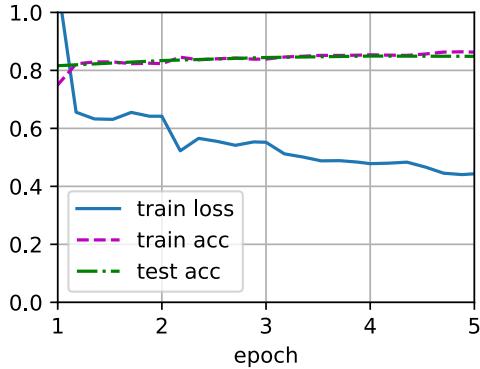
```
def loss(inputs, targets):
    return F.cross_entropy(inputs, targets, reduction='none').mean(1).mean(1)
```

(continues on next page)

(continued from previous page)

```
num_epochs, lr, wd, devices = 5, 0.001, 1e-3, d2l.try_all_gpus()
trainer = torch.optim.SGD(net.parameters(), lr=lr, weight_decay=wd)
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.443, train acc 0.863, test acc 0.848
254.0 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



13.11.5 预测

在预测时，我们需要将输入图像在各个通道做标准化，并转成卷积神经网络所需要的四维输入格式。

```
def predict(img):
    X = test_iter.dataset.normalize_image(img).unsqueeze(0)
    pred = net(X.to(devices[0])).argmax(dim=1)
    return pred.reshape(pred.shape[1], pred.shape[2])
```

为了可视化预测的类别给每个像素，我们将预测类别映射回它们在数据集中的标注颜色。

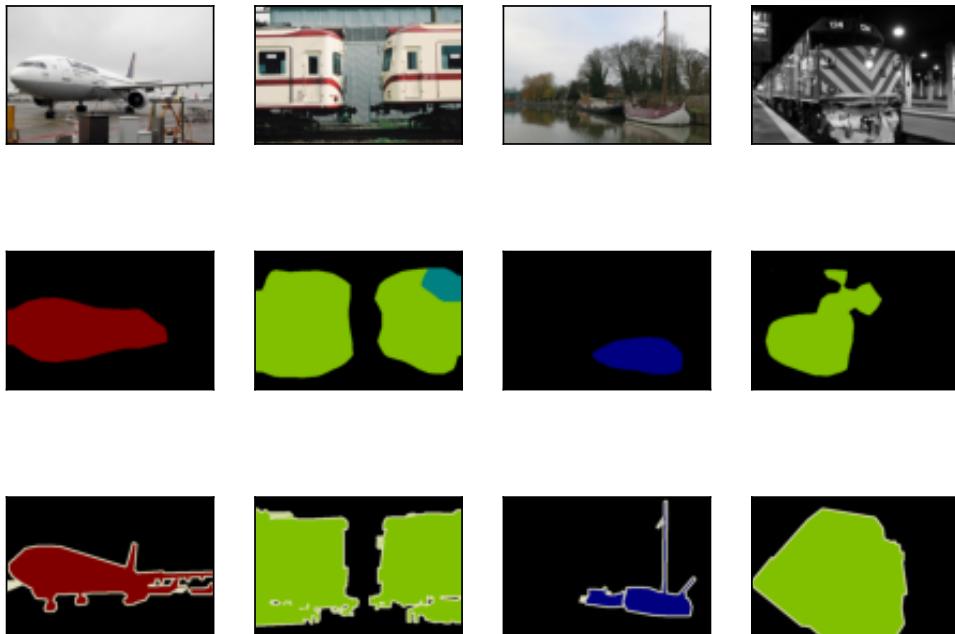
```
def label2image(pred):
    colormap = torch.tensor(d2l.VOC_COLORMAP, device=devices[0])
    X = pred.long()
    return colormap[X, :]
```

测试数据集中的图像大小和形状各异。由于模型使用了步幅为32的转置卷积层，因此当输入图像的高或宽无法被32整除时，转置卷积层输出的高或宽会与输入图像的尺寸有偏差。为了解决这个问题，我们可以在图像中截取多块高和宽为32的整数倍的矩形区域，并分别对这些区域中的像素做前向传播。请注意，这些区域的并集需要完整覆盖输入图像。当一个像素被多个区域所覆盖时，它在不同区域前向传播中转置卷积层输出的平均值可以作为softmax运算的输入，从而预测类别。

为简单起见，我们只读取几张较大的测试图像，并从图像的左上角开始截取形状为 320×480 的区域用于预测。

对于这些测试图像，我们逐一打印它们截取的区域，再打印预测结果，最后打印标注的类别。

```
voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
test_images, test_labels = d2l.read_voc_images(voc_dir, False)
n, imgs = 4, []
for i in range(n):
    crop_rect = (0, 0, 320, 480)
    X = torchvision.transforms.functional.crop(test_images[i], *crop_rect)
    pred = label2image(predict(X))
    imgs += [X.permute(1,2,0), pred.cpu(),
              torchvision.transforms.functional.crop(
                  test_labels[i], *crop_rect).permute(1,2,0)]
d2l.show_images(imgs[::3] + imgs[1::3] + imgs[2::3], 3, n, scale=2);
```



小结

- 全卷积网络先使用卷积神经网络抽取图像特征，然后通过 1×1 卷积层将通道数变换为类别个数，最后通过转置卷积层将特征图的高和宽变换为输入图像的尺寸。
- 在全卷积网络中，我们可以将转置卷积层初始化为双线性插值的上采样。

练习

1. 如果将转置卷积层改用Xavier随机初始化，结果有什么变化？
2. 调节超参数，能进一步提升模型的精度吗？
3. 预测测试图像中所有像素的类别。
4. 最初的全卷积网络的论文中 (Long *et al.*, 2015)还使用了某些卷积神经网络中间层的输出。试着实现这个想法。

Discussions¹⁸²

13.12 风格迁移

摄影爱好者也许接触过滤波器。它能改变照片的颜色风格，从而使风景照更加锐利或者令人像更加美白。但一个滤波器通常只能改变照片的某个方面。如果要照片达到理想中的风格，可能需要尝试大量不同的组合。这个过程的复杂程度不亚于模型调参。

本节将介绍如何使用卷积神经网络，自动将一个图像中的风格应用在另一图像之上，即风格迁移 (style transfer) (Gatys *et al.*, 2016)。这里我们需要两张输入图像：一张是内容图像，另一张是风格图像。我们将使用神经网络修改内容图像，使其在风格上接近风格图像。例如，图13.12.1中的内容图像为本书作者在西雅图郊区的雷尼尔山国家公园拍摄的风景照，而风格图像则是一幅主题为秋天橡树的油画。最终输出的合成图像应用了风格图像的油画笔触让整体颜色更加鲜艳，同时保留了内容图像中物体主体的形状。



图13.12.1: 输入内容图像和风格图像，输出风格迁移后的合成图像

¹⁸² <https://discuss.d2l.ai/t/3297>

13.12.1 方法

图13.12.2用简单的例子阐述了基于卷积神经网络的风格迁移方法。首先，我们初始化合成图像，例如将其初始化为内容图像。该合成图像是风格迁移过程中唯一需要更新的变量，即风格迁移所需迭代的模型参数。然后，我们选择一个预训练的卷积神经网络来抽取图像的特征，其中的模型参数在训练中无须更新。这个深度卷积神经网络凭借多个层逐级抽取图像的特征，我们可以选择其中某些层的输出作为内容特征或风格特征。以图13.12.2为例，这里选取的预训练的神经网络含有3个卷积层，其中第二层输出内容特征，第一层和第三层输出风格特征。

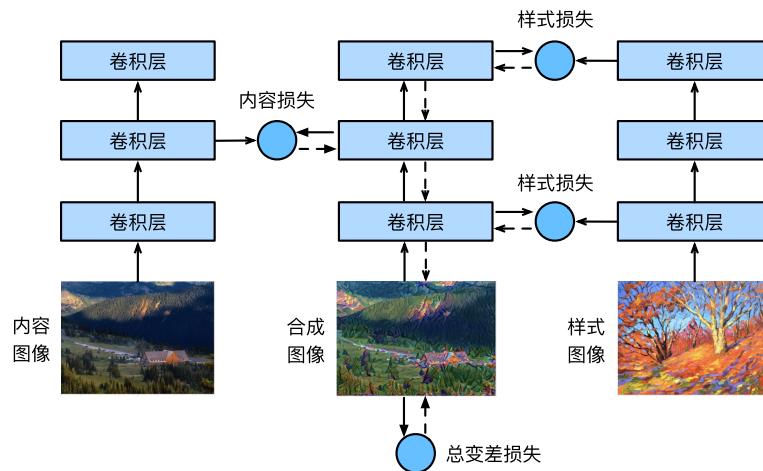


图13.12.2: 基于卷积神经网络的风格迁移。实线箭头和虚线箭头分别表示前向传播和反向传播

接下来，我们通过前向传播（实线箭头方向）计算风格迁移的损失函数，并通过反向传播（虚线箭头方向）迭代模型参数，即不断更新合成图像。风格迁移常用的损失函数由3部分组成：

1. 内容损失使合成图像与内容图像在内容特征上接近；
2. 风格损失使合成图像与风格图像在风格特征上接近；
3. 全变分损失则有助于减少合成图像中的噪点。

最后，当模型训练结束时，我们输出风格迁移的模型参数，即得到最终的合成图像。

在下面，我们将通过代码来进一步了解风格迁移的技术细节。

13.12.2 阅读内容和风格图像

首先，我们读取内容和风格图像。从打印出的图像坐标轴可以看出，它们的尺寸并不一样。

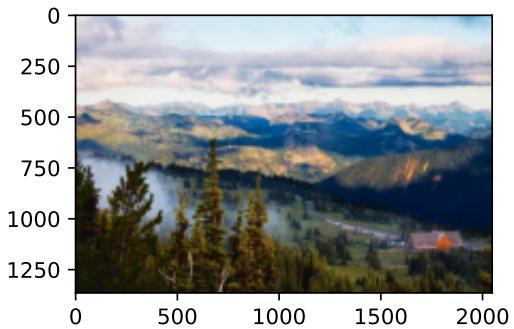
```
%matplotlib inline
import torch
import torchvision
```

(continues on next page)

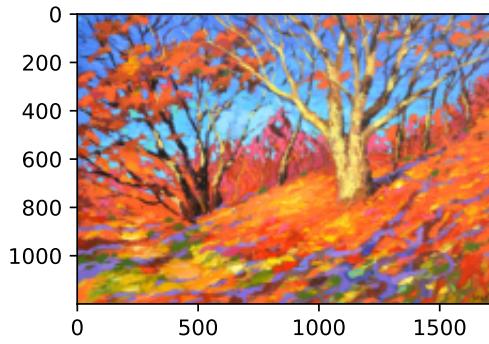
(continued from previous page)

```
from torch import nn
from d2l import torch as d2l

d2l.set_figsize()
content_img = d2l.Image.open('../img/rainier.jpg')
d2l.plt.imshow(content_img);
```



```
style_img = d2l.Image.open('../img/autumn-oak.jpg')
d2l.plt.imshow(style_img);
```



13.12.3 预处理和后处理

下面，定义图像的预处理函数和后处理函数。预处理函数`preprocess`对输入图像在RGB三个通道分别做标准化，并将结果变换为卷积神经网络接受的输入格式。后处理函数`postprocess`则将输出图像中的像素值还原回标准化之前的值。由于图像打印函数要求每个像素的浮点数值在0~1之间，我们对小于0和大于1的值分别取0和1。

```
rgb_mean = torch.tensor([0.485, 0.456, 0.406])
rgb_std = torch.tensor([0.229, 0.224, 0.225])
```

(continues on next page)

```

def preprocess(img, image_shape):
    transforms = torchvision.transforms.Compose([
        torchvision.transforms.Resize(image_shape),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=rgb_mean, std=rgb_std)])
    return transforms(img).unsqueeze(0)

def postprocess(img):
    img = img[0].to(rgb_std.device)
    img = torch.clamp(img.permute(1, 2, 0) * rgb_std + rgb_mean, 0, 1)
    return torchvision.transforms.ToPILImage()(img.permute(2, 0, 1))

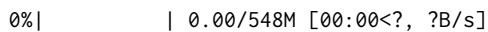
```

13.12.4 抽取图像特征

我们使用基于ImageNet数据集预训练的VGG-19模型来抽取图像特征 (Gatys *et al.*, 2016)。

```
pretrained_net = torchvision.models.vgg19(pretrained=True)
```

```
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to /home/ci/.cache/torch/hub/checkpoints/vgg19-dcbb9e9d.pth
```



为了抽取图像的内容特征和风格特征，我们可以选择VGG网络中某些层的输出。一般来说，越靠近输入层，越容易抽取图像的细节信息；反之，则越容易抽取图像的全局信息。为了避免合成图像过多保留内容图像的细节，我们选择VGG较靠近输出的层，即内容层，来输出图像的内容特征。我们还从VGG中选择不同层的输出来匹配局部和全局的风格，这些图层也称为风格层。正如 7.2 节中所介绍的，VGG网络使用了5个卷积块。实验中，我们选择第四卷积块的最后一个卷积层作为内容层，选择每个卷积块的第一个卷积层作为风格层。这些层的索引可以通过打印pretrained_net实例获取。

```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

使用VGG层抽取特征时，我们只需要用到从输入层到最靠近输出层的内容层或风格层之间的所有层。下面构建一个新的网络net，它只保留需要用到的VGG的所有层。

```
net = nn.Sequential(*[pretrained_net.features[i] for i in
                     range(max(content_layers + style_layers) + 1)])
```

给定输入 X , 如果我们简单地调用前向传播 $\text{net}(X)$, 只能获得最后一层的输出。由于我们还需要中间层的输出, 因此这里我们逐层计算, 并保留内容层和风格层的输出。

```
def extract_features(X, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        X = net[i](X)
        if i in style_layers:
            styles.append(X)
        if i in content_layers:
            contents.append(X)
    return contents, styles
```

下面定义两个函数: `get_contents`函数对内容图像抽取内容特征; `get_styles`函数对风格图像抽取风格特征。因为在训练时无须改变预训练的VGG的模型参数, 所以我们可以在训练开始之前就提取出内容特征和风格特征。由于合成图像是风格迁移所需迭代的模型参数, 我们只能在训练过程中通过调用`extract_features`函数来抽取合成图像的内容特征和风格特征。

```
def get_contents(image_shape, device):
    content_X = preprocess(content_img, image_shape).to(device)
    contents_Y, _ = extract_features(content_X, content_layers, style_layers)
    return content_X, contents_Y

def get_styles(image_shape, device):
    style_X = preprocess(style_img, image_shape).to(device)
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y
```

13.12.5 定义损失函数

下面我们来描述风格迁移的损失函数。它由内容损失、风格损失和全变分损失3部分组成。

内容损失

与线性回归中的损失函数类似, 内容损失通过平方误差函数衡量合成图像与内容图像在内容特征上的差异。平方误差函数的两个输入均为`extract_features`函数计算所得到的内容层的输出。

```
def content_loss(Y_hat, Y):
    # 我们从动态计算梯度的树中分离目标:
    # 这是一个规定的值, 而不是一个变量。
    return torch.square(Y_hat - Y.detach()).mean()
```

风格损失

风格损失与内容损失类似，也通过平方误差函数衡量合成图像与风格图像在风格上的差异。为了表达风格层输出的风格，我们先通过`extract_features`函数计算风格层的输出。假设该输出的样本数为1，通道数为 c ，高和宽分别为 h 和 w ，我们可以将此输出转换为矩阵 \mathbf{X} ，其有 c 行和 hw 列。这个矩阵可以被看作由 c 个长度为 hw 的向量 $\mathbf{x}_1, \dots, \mathbf{x}_c$ 组合而成的。其中向量 \mathbf{x}_i 代表了通道 i 上的风格特征。

在这些向量的格拉姆矩阵 $\mathbf{XX}^\top \in \mathbb{R}^{c \times c}$ 中， i 行 j 列的元素 x_{ij} 即向量 \mathbf{x}_i 和 \mathbf{x}_j 的内积。它表达了通道 i 和通道 j 上风格特征的相关性。我们用这样的格拉姆矩阵来表达风格层输出的风格。需要注意的是，当 hw 的值较大时，格拉姆矩阵中的元素容易出现较大的值。此外，格拉姆矩阵的高和宽皆为通道数 c 。为了让风格损失不受这些值的大小影响，下面定义的`gram`函数将格拉姆矩阵除以了矩阵中元素的个数，即 chw 。

```
def gram(X):
    num_channels, n = X.shape[1], X.numel() // X.shape[1]
    X = X.reshape((num_channels, n))
    return torch.matmul(X, X.T) / (num_channels * n)
```

自然地，风格损失的平方误差函数的两个格拉姆矩阵输入分别基于合成图像与风格图像的风格层输出。这里假设基于风格图像的格拉姆矩阵`gram_Y`已经预先计算好了。

```
def style_loss(Y_hat, gram_Y):
    return torch.square(gram(Y_hat) - gram_Y.detach()).mean()
```

全变分损失

有时候，我们学到的合成图像里面有大量高频噪点，即有特别亮或者特别暗的颗粒像素。一种常见的去噪方法是全变分去噪（total variation denoising）：假设 $x_{i,j}$ 表示坐标 (i, j) 处的像素值，降低全变分损失

$$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}| \quad (13.12.1)$$

能够尽可能使邻近的像素值相似。

```
def tv_loss(Y_hat):
    return 0.5 * (torch.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
                  torch.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

损失函数

风格转移的损失函数是内容损失、风格损失和总变化损失的加权和。通过调节这些权重超参数，我们可以权衡合成图像在保留内容、迁移风格以及去噪三方面的相对重要性。

```
content_weight, style_weight, tv_weight = 1, 1e3, 10

def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # 分别计算内容损失、风格损失和全变分损失
    contents_l = [content_loss(Y_hat, Y) * content_weight for Y_hat, Y in zip(
        contents_Y_hat, contents_Y)]
    styles_l = [style_loss(Y_hat, Y) * style_weight for Y_hat, Y in zip(
        styles_Y_hat, styles_Y_gram)]
    tv_l = tv_loss(X) * tv_weight
    # 对所有损失求和
    l = sum(10 * styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
```

13.12.6 初始化合成图像

在风格迁移中，合成的图像是训练期间唯一需要更新的变量。因此，我们可以定义一个简单的模型SynthesizedImage，并将合成的图像视为模型参数。模型的前向传播只需返回模型参数即可。

```
class SynthesizedImage(nn.Module):
    def __init__(self, img_shape, **kwargs):
        super(SynthesizedImage, self).__init__(**kwargs)
        self.weight = nn.Parameter(torch.rand(*img_shape))

    def forward(self):
        return self.weight
```

下面，我们定义get_inits函数。该函数创建了合成图像的模型实例，并将其初始化为图像X。风格图像在各个风格层的格拉姆矩阵styles_Y_gram将在训练前预先计算好。

```
def get_inits(X, device, lr, styles_Y):
    gen_img = SynthesizedImage(X.shape).to(device)
    gen_img.weight.data.copy_(X.data)
    trainer = torch.optim.Adam(gen_img.parameters(), lr=lr)
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, trainer
```

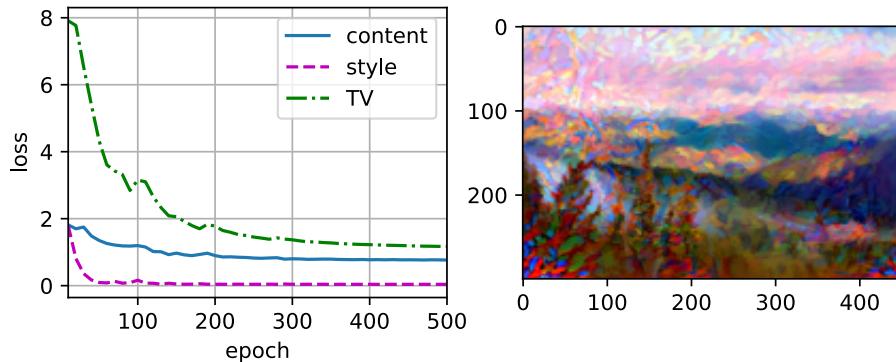
13.12.7 训练模型

在训练模型进行风格迁移时，我们不断抽取合成图像的内容特征和风格特征，然后计算损失函数。下面定义了训练循环。

```
def train(X, contents_Y, styles_Y, device, lr, num_epochs, lr_decay_epoch):
    X, styles_Y_gram, trainer = get_inits(X, device, lr, styles_Y)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_decay_epoch, 0.8)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs],
                            legend=['content', 'style', 'TV'],
                            ncols=2, figsize=(7, 2.5))
    for epoch in range(num_epochs):
        trainer.zero_grad()
        contents_Y_hat, styles_Y_hat = extract_features(
            X, content_layers, style_layers)
        contents_l, styles_l, tv_l, l = compute_loss(
            X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram)
        l.backward()
        trainer.step()
        scheduler.step()
        if (epoch + 1) % 10 == 0:
            animator.axes[1].imshow(postprocess(X))
            animator.add(epoch + 1, [float(sum(contents_l)),
                                    float(sum(styles_l)), float(tv_l)])
    return X
```

现在我们训练模型：首先将内容图像和风格图像的高和宽分别调整为300和450像素，用内容图像来初始化合成图像。

```
device, image_shape = d2l.try_gpu(), (300, 450)
net = net.to(device)
content_X, contents_Y = get_contents(image_shape, device)
_, styles_Y = get_styles(image_shape, device)
output = train(content_X, contents_Y, styles_Y, device, 0.3, 500, 50)
```



我们可以看到，合成图像保留了内容图像的风景和物体，并同时迁移了风格图像的色彩。例如，合成图像具有与风格图像中一样的色彩块，其中一些甚至具有画笔笔触的细微纹理。

小结

- 风格迁移常用的损失函数由3部分组成：（1）内容损失使合成图像与内容图像在内容特征上接近；（2）风格损失令合成图像与风格图像在风格特征上接近；（3）全变分损失则有助于减少合成图像中的噪点。
- 我们可以通过预训练的卷积神经网络来抽取图像的特征，并通过最小化损失函数来不断更新合成图像来作为模型参数。
- 我们使用格拉姆矩阵表达风格层输出的风格。

练习

1. 选择不同的内容和风格层，输出有什么变化？
2. 调整损失函数中的权重超参数。输出是否保留更多内容或减少更多噪点？
3. 替换实验中的内容图像和风格图像，能创作出更有趣的合成图像吗？
4. 我们可以对文本使用风格迁移吗？提示：可以参阅调查报告 (Hu *et al.*, 2020)。

Discussions¹⁸³

13.13 实战 Kaggle 比赛：图像分类 (CIFAR-10)

之前几节中，我们一直在使用深度学习框架的高级API直接获取张量格式的图像数据集。但是在实践中，图像数据集通常以图像文件的形式出现。本节将从原始图像文件开始，然后逐步组织、读取并将它们转换为张量格式。

¹⁸³ <https://discuss.d2l.ai/t/3300>

我们在 13.1 节中对 CIFAR-10 数据集做了一个实验。CIFAR-10 是计算机视觉领域中的一个重要的数据集。本节将运用我们在前几节中学到的知识来参加 CIFAR-10 图像分类问题的 Kaggle 竞赛，比赛的网址是 <https://www.kaggle.com/c/cifar-10>。

图13.13.1显示了竞赛网站页面上的信息。为了能提交结果，首先需要注册一个Kaggle账户。

The screenshot shows the Kaggle competition page for 'CIFAR-10 - Object Recognition in Images'. At the top, there's a grid of small sample images from the dataset. Below it, the title 'CIFAR-10 - Object Recognition in Images' is displayed, followed by the subtitle 'Identify the subject of 60,000 labeled images'. It also shows '231 teams · 4 years ago'. Below the title, there are navigation links: 'Overview' (which is underlined), 'Data', 'Discussion', 'Leaderboard', and 'Rules'. The 'Overview' section is currently active. In the 'Description' tab, it says: 'CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the 80 million tiny images dataset and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.' The 'Evaluation' tab is also visible.

图13.13.1: CIFAR-10 图像分类竞赛页面上的信息。竞赛用的数据集可通过点击“Data”选项卡获取。

首先，导入竞赛所需的包和模块。

```
import collections
import math
import os
import shutil
import pandas as pd
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

13.13.1 获取并组织数据集

比赛数据集分为训练集和测试集，其中训练集包含50000张、测试集包含300000张图像。在测试集中，10000张图像将被用于评估，而剩下的290000张图像将不会被进行评估，包含它们只是为了防止手动标记测试集并提交标记结果。两个数据集中的图像都是png格式，高度和宽度均为32像素并有三个颜色通道（RGB）。这些图片共涵盖10个类别：飞机、汽车、鸟类、猫、鹿、狗、青蛙、马、船和卡车。图13.13.1的左上角显示了数据集中飞机、汽车和鸟类的一些图像。

下载数据集

登录Kaggle后，我们可以点击图13.13.1中显示的CIFAR-10图像分类竞赛网页上的“Data”选项卡，然后单击“Download All”按钮下载数据集。在`../data`中解压下载的文件并在其中解压缩`train.7z`和`test.7z`后，在以下路径中可以找到整个数据集：

- `../data/cifar-10/train/[1-50000].png`
- `../data/cifar-10/test/[1-300000].png`
- `../data/cifar-10/trainLabels.csv`
- `../data/cifar-10/sampleSubmission.csv`

`train`和`test`文件夹分别包含训练和测试图像，`trainLabels.csv`含有训练图像的标签，`sample_submission.csv`是提交文件的范例。

为了便于入门，我们提供包含前1000个训练图像和5个随机测试图像的数据集的小规模样本。要使用Kaggle竞赛的完整数据集，需要将以下`demo`变量设置为`False`。

```
#@save
d2l.DATA_HUB['cifar10_tiny'] = (d2l.DATA_URL + 'kaggle_cifar10_tiny.zip',
                                 '2068874e4b9a9f0fb07ebe0ad2b29754449ccacd')

# 如果使用完整的Kaggle竞赛的数据集，设置demo为False
demo = True

if demo:
    data_dir = d2l.download_extract('cifar10_tiny')
else:
    data_dir = '../data/cifar-10/'
```

```
Downloading ../data/kaggle_cifar10_tiny.zip from http://d2l-data.s3-accelerate.amazonaws.com/kaggle_
↳ cifar10_tiny.zip...
```

整理数据集

我们需要整理数据集来训练和测试模型。首先，我们用以下函数读取CSV文件中的标签，它返回一个字典，该字典将文件名中不带扩展名的部分映射到其标签。

```
#@save
def read_csv_labels(fname):
    """读取fname来给标签字典返回一个文件名"""
    with open(fname, 'r') as f:
        # 跳过文件头行(列名)
```

(continues on next page)

(continued from previous page)

```
lines = f.readlines()[1:]
tokens = [l.rstrip().split(',') for l in lines]
return dict(((name, label) for name, label in tokens))

labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
print('# 训练样本 :', len(labels))
print('# 类别 :', len(set(labels.values())))
```

```
# 训练样本 : 1000
# 类别 : 10
```

接下来，我们定义reorg_train_valid函数来将验证集从原始的训练集中拆分出来。此函数中的参数valid_ratio是验证集中的样本数与原始训练集中的样本数之比。更具体地说，令 n 等于样本最少的类别中的图像数量，而 r 是比率。验证集将为每个类别拆分出 $\max([nr], 1)$ 张图像。让我们以valid_ratio=0.1为例，由于原始的训练集有50000张图像，因此train_valid_test/train路径中将有45000张图像用于训练，而剩下5000张图像将作为路径train_valid_test/valid中的验证集。组织数据集后，同类别的图像将被放置在同一文件夹下。

```
#@save
def copyfile(filename, target_dir):
    """将文件复制到目标目录"""
    os.makedirs(target_dir, exist_ok=True)
    shutil.copy(filename, target_dir)

#@save
def reorg_train_valid(data_dir, labels, valid_ratio):
    """将验证集从原始的训练集中拆分出来"""
    # 训练数据集中样本最少的类别中的样本数
    n = collections.Counter(labels.values()).most_common()[-1][1]
    # 验证集中每个类别的样本数
    n_valid_per_label = max(1, math.floor(n * valid_ratio))
    label_count = {}
    for train_file in os.listdir(os.path.join(data_dir, 'train')):
        label = labels[train_file.split('.')[0]]
        fname = os.path.join(data_dir, 'train', train_file)
        copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                     'train_valid', label))
        if label not in label_count or label_count[label] < n_valid_per_label:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'valid', label))
            label_count[label] = label_count.get(label, 0) + 1
    else:
```

(continues on next page)

(continued from previous page)

```
copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                             'train', label))
return n_valid_per_label
```

下面的reorg_test函数用来在预测期间整理测试集，以方便读取。

```
#@save
def reorg_test(data_dir):
    """在预测期间整理测试集，以方便读取"""
    for test_file in os.listdir(os.path.join(data_dir, 'test')):
        copyfile(os.path.join(data_dir, 'test', test_file),
                 os.path.join(data_dir, 'train_valid_test', 'test',
                             'unknown'))
```

最后，我们使用一个函数来调用前面定义的函数read_csv_labels、reorg_train_valid和reorg_test。

```
def reorg_cifar10_data(data_dir, valid_ratio):
    labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
    reorg_train_valid(data_dir, labels, valid_ratio)
    reorg_test(data_dir)
```

在这里，我们只将样本数据集的批量大小设置为32。在实际训练和测试中，应该使用Kaggle竞赛的完整数据集，并将batch_size设置为更大的整数，例如128。我们将10%的训练样本作为调整超参数的验证集。

```
batch_size = 32 if demo else 128
valid_ratio = 0.1
reorg_cifar10_data(data_dir, valid_ratio)
```

13.13.2 图像增广

我们使用图像增广来解决过拟合的问题。例如在训练中，我们可以随机水平翻转图像。我们还可以对彩色图像的三个RGB通道执行标准化。下面，我们列出了其中一些可以调整的操作。

```
transform_train = torchvision.transforms.Compose([
    # 在高度和宽度上将图像放大到40像素的正方形
    torchvision.transforms.Resize(40),
    # 随机裁剪出一个高度和宽度均为40像素的正方形图像,
    # 生成一个面积为原始图像面积0.64~1倍的小正方形,
    # 然后将其缩放为高度和宽度均为32像素的正方形
    torchvision.transforms.RandomResizedCrop(32, scale=(0.64, 1.0),
                                             ratio=(1.0, 1.0)),
```

(continues on next page)

(continued from previous page)

```
torchvision.transforms.RandomHorizontalFlip(),
torchvision.transforms.ToTensor(),
# 标准化图像的每个通道
torchvision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                [0.2023, 0.1994, 0.2010]))
```

在测试期间，我们只对图像执行标准化，以消除评估结果中的随机性。

```
transform_test = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                    [0.2023, 0.1994, 0.2010]))
```

13.13.3 读取数据集

接下来，我们读取由原始图像组成的数据集，每个样本都包括一张图片和一个标签。

```
train_ds, train_valid_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_train) for folder in ['train', 'train_valid']]

valid_ds, test_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_test) for folder in ['valid', 'test']]
```

在训练期间，我们需要指定上面定义的所有图像增广操作。当验证集在超参数调整过程中用于模型评估时，不应引入图像增广的随机性。在最终预测之前，我们根据训练集和验证集组合而成的训练模型进行训练，以充分利用所有标记的数据。

```
train_iter, train_valid_iter = [torch.utils.data.DataLoader(
    dataset, batch_size, shuffle=True, drop_last=True)
    for dataset in (train_ds, train_valid_ds)]

valid_iter = torch.utils.data.DataLoader(valid_ds, batch_size, shuffle=False,
                                         drop_last=True)

test_iter = torch.utils.data.DataLoader(test_ds, batch_size, shuffle=False,
                                         drop_last=False)
```

13.13.4 定义模型

我们定义了 7.6 节中描述的Resnet-18模型。

```
def get_net():
    num_classes = 10
    net = d2l.resnet18(num_classes, 3)
    return net

loss = nn.CrossEntropyLoss(reduction="none")
```

13.13.5 定义训练函数

我们将根据模型在验证集上的表现来选择模型并调整超参数。下面我们定义了模型训练函数train。

```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    trainer = torch.optim.SGD(net.parameters(), lr=lr, momentum=0.9,
                              weight_decay=wd)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_period, lr_decay)
    num_batches, timer = len(train_iter), d2l.Timer()
    legend = ['train loss', 'train acc']
    if valid_iter is not None:
        legend.append('valid acc')
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=legend)
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    for epoch in range(num_epochs):
        net.train()
        metric = d2l.Accumulator(3)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = d2l.train_batch_ch13(net, features, labels,
                                         loss, trainer, devices)
            metric.add(l, acc, labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[2],
                             None))
        if valid_iter is not None:
            valid_acc = d2l.evaluate_accuracy_gpu(net, valid_iter)
            animator.add(epoch + 1, (None, None, valid_acc))
```

(continues on next page)

(continued from previous page)

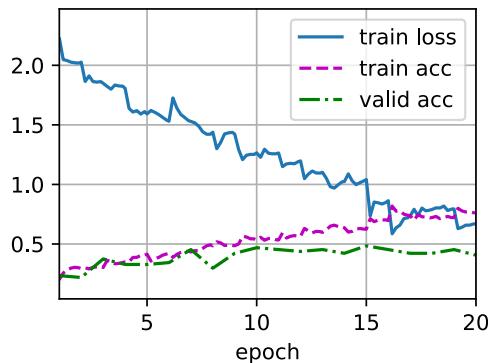
```
scheduler.step()  
measures = (f'train loss {metric[0] / metric[2]:.3f}, '  
            f'train acc {metric[1] / metric[2]:.3f}')  
if valid_iter is not None:  
    measures += f', valid acc {valid_acc:.3f}'  
print(measures + f'\n{metric[2] * num_epochs / timer.sum():.1f}'  
      f' examples/sec on {str(devices)}')
```

13.13.6 训练和验证模型

现在，我们可以训练和验证模型了，而以下所有超参数都可以调整。例如，我们可以增加周期的数量。当lr_period和lr_decay分别设置为4和0.9时，优化算法的学习速率将在每4个周期乘以0.9。为便于演示，我们在这里只训练20个周期。

```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 20, 2e-4, 5e-4  
lr_period, lr_decay, net = 4, 0.9, get_net()  
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,  
      lr_decay)
```

```
train loss 0.668, train acc 0.761, valid acc 0.406  
758.4 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



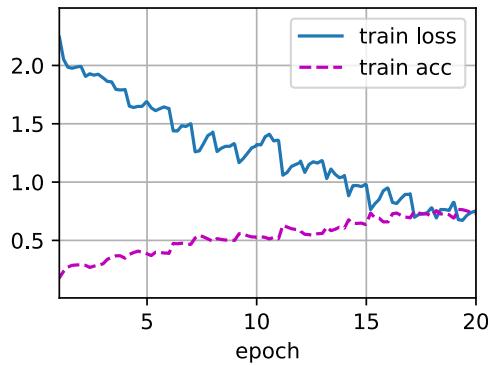
13.13.7 在 Kaggle 上对测试集进行分类并提交结果

在获得具有超参数的满意的模型后，我们使用所有标记的数据（包括验证集）来重新训练模型并对测试集进行分类。

```
net, preds = get_net(), []
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

for X, _ in test_iter:
    y_hat = net(X.to(devices[0]))
    preds.extend(y_hat.argmax(dim=1).type(torch.int32).cpu().numpy())
sorted_ids = list(range(1, len(test_ds) + 1))
sorted_ids.sort(key=lambda x: str(x))
df = pd.DataFrame({'id': sorted_ids, 'label': preds})
df['label'] = df['label'].apply(lambda x: train_valid_ds.classes[x])
df.to_csv('submission.csv', index=False)
```

```
train loss 0.745, train acc 0.734
883.3 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



向Kaggle提交结果的方法与 4.10节中的方法类似，上面的代码将生成一个 `submission.csv`文件，其格式符合Kaggle竞赛的要求。

小结

- 将包含原始图像文件的数据集组织为所需格式后，我们可以读取它们。
- 我们可以在图像分类竞赛中使用卷积神经网络和图像增广。

练习

1. 在这场Kaggle竞赛中使用完整的CIFAR-10数据集。将超参数设为batch_size = 128, num_epochs = 100, lr = 0.1, lr_period = 50, lr_decay = 0.1。看看在这场比赛中能达到什么准确度和排名。能进一步改进吗？
2. 不使用图像增广时，能获得怎样的准确度？

Discussions¹⁸⁴

13.14 实战Kaggle比赛：狗的品种识别（ImageNet Dogs）

本节我们将在Kaggle上实战狗品种识别问题。本次比赛网址是<https://www.kaggle.com/c/dog-breed-identification>。图13.14.1显示了鉴定比赛网页上的信息。需要一个Kaggle账户才能提交结果。

在这场比赛中，我们将识别120类不同品种的狗。这个数据集实际上是著名的ImageNet的数据集子集。与13.13节中CIFAR-10数据集中的图像不同，ImageNet数据集中的图像更高更宽，且尺寸不一。

A screenshot of the Kaggle Dog Breed Identification competition page. The top banner features a large image of a dog's head and the text "Playground Prediction Competition" and "Dog Breed Identification: Determine the breed of a dog in an image". Below the banner, it says "Kaggle · 1,286 teams · 4 months ago". A navigation bar includes "Overview" (which is underlined), "Data", "Kernels", "Discussion", "Leaderboard", and "Rules". The "Overview" section contains two tabs: "Description" and "Evaluation". The "Description" tab contains text about the competition and a 2x5 grid of dog images. The "Evaluation" tab contains text about the dataset and a 2x5 grid of dog images.

图13.14.1: 狗的品种鉴定比赛网站，可以通过单击“数据”选项卡来获得比赛数据集。

¹⁸⁴ <https://discuss.d2l.ai/t/2831>

```
import os
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

13.14.1 获取和整理数据集

比赛数据集分为训练集和测试集，分别包含RGB（彩色）通道的10222张、10357张JPEG图像。在训练数据集中，有120种犬类，如拉布拉多、贵宾、腊肠、萨摩耶、哈士奇、吉娃娃和约克夏等。

下载数据集

登录Kaggle后，可以点击图13.14.1中显示的竞争网页上的“数据”选项卡，然后点击“全部下载”按钮下载数据集。在`../data`中解压下载的文件后，将在以下路径中找到整个数据集：

- `../data/dog-breed-identification/labels.csv`
- `../data/dog-breed-identification/sample_submission.csv`
- `../data/dog-breed-identification/train`
- `../data/dog-breed-identification/test`

上述结构与13.13节的CIFAR-10类似，其中文件夹`train/`和`test/`分别包含训练和测试狗图像，`labels.csv`包含训练图像的标签。

同样，为了便于入门，我们提供完整数据集的小规模样本：`train_valid_test_tiny.zip`。如果要在Kaggle比赛中使用完整的数据集，则需要将下面的`demo`变量更改为`False`。

```
#@save
d2l.DATA_HUB['dog_tiny'] = (d2l.DATA_URL + 'kaggle_dog_tiny.zip',
                             '0cb91d09b814ecdc07b50f31f8dcad3e81d6a86d')

# 如果使用Kaggle比赛的完整数据集，请将下面的变量更改为False
demo = True
if demo:
    data_dir = d2l.download_extract('dog_tiny')
else:
    data_dir = os.path.join('..', 'data', 'dog-breed-identification')
```

```
Downloading ../data/kaggle_dog_tiny.zip from http://d2l-data.s3-accelerate.amazonaws.com/kaggle_dog_
˓→tiny.zip...
```

整理数据集

我们可以像 13.13 节中所做的那样整理数据集，即从原始训练集中拆分验证集，然后将图像移动到按标签分组的子文件夹中。

下面的 reorg_dog_data 函数读取训练数据标签、拆分验证集并整理训练集。

```
def reorg_dog_data(data_dir, valid_ratio):
    labels = d2l.read_csv_labels(os.path.join(data_dir, 'labels.csv'))
    d2l.reorg_train_valid(data_dir, labels, valid_ratio)
    d2l.reorg_test(data_dir)

batch_size = 32 if demo else 128
valid_ratio = 0.1
reorg_dog_data(data_dir, valid_ratio)
```

13.14.2 图像增广

回想一下，这个狗品种数据集是ImageNet数据集的子集，其图像大于 13.13 节中CIFAR-10数据集的图像。下面我们看一下如何在相对较大的图像上使用图像增广。

```
transform_train = torchvision.transforms.Compose([
    # 随机裁剪图像，所得图像为原始面积的0.08~1之间，高宽比在3/4和4/3之间。
    # 然后，缩放图像以创建224x224的新图像
    torchvision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                             ratio=(3.0/4.0, 4.0/3.0)),
    torchvision.transforms.RandomHorizontalFlip(),
    # 随机更改亮度，对比度和饱和度
    torchvision.transforms.ColorJitter(brightness=0.4,
                                       contrast=0.4,
                                       saturation=0.4),
    # 添加随机噪声
    torchvision.transforms.ToTensor(),
    # 标准化图像的每个通道
    torchvision.transforms.Normalize([0.485, 0.456, 0.406],
                                    [0.229, 0.224, 0.225]))
```

测试时，我们只使用确定性的图像预处理操作。

```
transform_test = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    # 从图像中心裁切224x224大小的图片
```

(continues on next page)

(continued from previous page)

```
torchvision.transforms.CenterCrop(224),  
torchvision.transforms.ToTensor(),  
torchvision.transforms.Normalize([0.485, 0.456, 0.406],  
[0.229, 0.224, 0.225]))
```

13.14.3 读取数据集

与 13.13节一样，我们可以读取整理后的含原始图像文件的数据集。

```
train_ds, train_valid_ds = [torchvision.datasets.ImageFolder(  
    os.path.join(data_dir, 'train_valid_test', folder),  
    transform=transform_train) for folder in ['train', 'train_valid']]  
  
valid_ds, test_ds = [torchvision.datasets.ImageFolder(  
    os.path.join(data_dir, 'train_valid_test', folder),  
    transform=transform_test) for folder in ['valid', 'test']]
```

下面我们创建数据加载器实例的方式与 13.13节相同。

```
train_iter, train_valid_iter = [torch.utils.data.DataLoader(  
    dataset, batch_size, shuffle=True, drop_last=True)  
    for dataset in (train_ds, train_valid_ds)]  
  
valid_iter = torch.utils.data.DataLoader(valid_ds, batch_size, shuffle=False,  
                                         drop_last=True)  
  
test_iter = torch.utils.data.DataLoader(test_ds, batch_size, shuffle=False,  
                                         drop_last=False)
```

13.14.4 微调预训练模型

同样，本次比赛的数据集是ImageNet数据集的子集。因此，我们可以使用 13.2节中讨论的方法在完整ImageNet数据集上选择预训练的模型，然后使用该模型提取图像特征，以便将其输入到定制的小规模输出网络中。深度学习框架的高级API提供了在ImageNet数据集上预训练的各种模型。在这里，我们选择预训练的ResNet-34模型，我们只需重复使用此模型的输出层（即提取的特征）的输入。然后，我们可以用一个可以训练的小型自定义输出网络替换原始输出层，例如堆叠两个完全连接的图层。与 13.2节中的实验不同，以下内容不重新训练用于特征提取的预训练模型，这节省了梯度下降的时间和内存空间。

回想一下，我们使用三个RGB通道的均值和标准差来对完整的ImageNet数据集进行图像标准化。事实上，这也符合ImageNet上预训练模型的标准化操作。

```

def get_net(devices):
    finetune_net = nn.Sequential()
    finetune_net.features = torchvision.models.resnet34(pretrained=True)
    # 定义一个新的输出网络，共有120个输出类别
    finetune_net.output_new = nn.Sequential(nn.Linear(1000, 256),
                                            nn.ReLU(),
                                            nn.Linear(256, 120))
    # 将模型参数分配给用于计算的CPU或GPU
    finetune_net = finetune_net.to(devices[0])
    # 冻结参数
    for param in finetune_net.features.parameters():
        param.requires_grad = False
    return finetune_net

```

在计算损失之前，我们首先获取预训练模型的输出层的输入，即提取的特征。然后我们使用此特征作为我们小型自定义输出网络的输入来计算损失。

```

loss = nn.CrossEntropyLoss(reduction='none')

def evaluate_loss(data_iter, net, devices):
    l_sum, n = 0.0, 0
    for features, labels in data_iter:
        features, labels = features.to(devices[0]), labels.to(devices[0])
        outputs = net(features)
        l = loss(outputs, labels)
        l_sum += l.sum()
        n += labels.numel()
    return (l_sum / n).to('cpu')

```

13.14.5 定义训练函数

我们将根据模型在验证集上的表现选择模型并调整超参数。模型训练函数train只迭代小型自定义输出网络的参数。

```

def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    # 只训练小型自定义输出网络
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    trainer = torch.optim.SGD((param for param in net.parameters()
                               if param.requires_grad), lr=lr,
                               momentum=0.9, weight_decay=wd)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_period, lr_decay)

```

(continues on next page)

(continued from previous page)

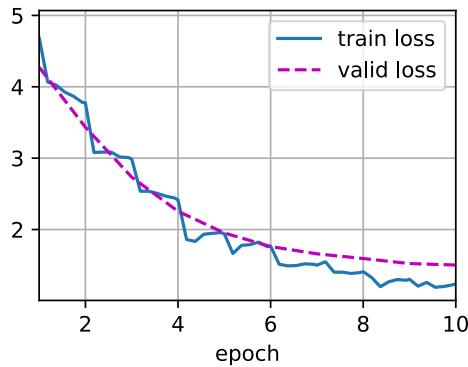
```
num_batches, timer = len(train_iter), d2l.Timer()
legend = ['train loss']
if valid_iter is not None:
    legend.append('valid loss')
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                         legend=legend)
for epoch in range(num_epochs):
    metric = d2l.Accumulator(2)
    for i, (features, labels) in enumerate(train_iter):
        timer.start()
        features, labels = features.to(devices[0]), labels.to(devices[0])
        trainer.zero_grad()
        output = net(features)
        l = loss(output, labels).sum()
        l.backward()
        trainer.step()
        metric.add(l, labels.shape[0])
        timer.stop()
        if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
            animator.add(epoch + (i + 1) / num_batches,
                          (metric[0] / metric[1], None))
    measures = f'train loss {metric[0] / metric[1]:.3f}'
    if valid_iter is not None:
        valid_loss = evaluate_loss(valid_iter, net, devices)
        animator.add(epoch + 1, (None, valid_loss.detach().cpu()))
    scheduler.step()
    if valid_iter is not None:
        measures += f', valid loss {valid_loss:.3f}'
print(measures + f'\n{metric[1] * num_epochs / timer.sum():.1f}')
f' examples/sec on {str(devices)}')
```

13.14.6 训练和验证模型

现在我们可以训练和验证模型了，以下超参数都是可调的。例如，我们可以增加迭代轮数。另外，由于lr_period和lr_decay分别设置为2和0.9，因此优化算法的学习速率将在每2个迭代后乘以0.9。

```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 10, 1e-4, 1e-4
lr_period, lr_decay, net = 2, 0.9, get_net(devices)
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)
```

```
train loss 1.237, valid loss 1.503
442.6 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



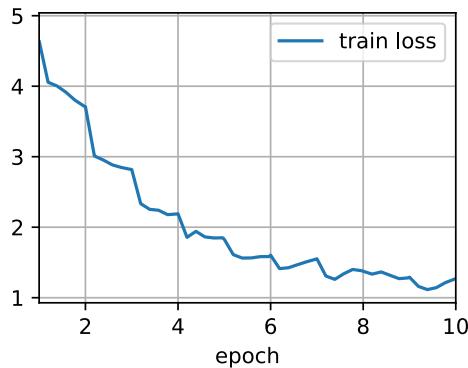
13.14.7 对测试集分类并在Kaggle提交结果

与 13.13 节中的最后一步类似，最终所有标记的数据（包括验证集）都用于训练模型和对测试集进行分类。我们将使用训练好的自定义输出网络进行分类。

```
net = get_net(devices)
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

preds = []
for data, label in test_iter:
    output = torch.nn.functional.softmax(net(data.to(devices[0])), dim=1)
    preds.extend(output.cpu().detach().numpy())
ids = sorted(os.listdir(
    os.path.join(data_dir, 'train_valid_test', 'test', 'unknown')))
with open('submission.csv', 'w') as f:
    f.write('id,' + ','.join(train_valid_ds.classes) + '\n')
    for i, output in zip(ids, preds):
        f.write(i.split('.')[0] + ',' + ','.join(
            [str(num) for num in output]) + '\n')
```

```
train loss 1.273
710.0 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



上面的代码将生成一个`submission.csv`文件，以4.10节中描述的方式提在Kaggle上提交。

小结

- ImageNet数据集中的图像比CIFAR-10图像尺寸大，我们可能会修改不同数据集上任务的图像增广操作。
- 要对ImageNet数据集的子集进行分类，我们可以利用完整ImageNet数据集上的预训练模型来提取特征并仅训练小型自定义输出网络，这将减少计算时间和节省内存空间。

练习

1. 试试使用完整Kaggle比赛数据集，增加`batch_size`（批量大小）和`num_epochs`（迭代轮数），或者设计其它超参数为`lr = 0.01`, `lr_period = 10`, 和`lr_decay = 0.1`时，能取得什么结果？
2. 如果使用更深的预训练模型，会得到更好的结果吗？如何调整超参数？能进一步改善结果吗？

Discussions¹⁸⁵

¹⁸⁵ <https://discuss.d2l.ai/t/2833>

自然语言处理：预训练

人与人之间需要交流。出于人类这种基本需要，每天都有大量的书面文本产生。比如，社交媒体、聊天应用、电子邮件、产品评论、新闻文章、研究论文和书籍中的丰富文本，使计算机能够理解它们以提供帮助或基于人类语言做出决策变得至关重要。

自然语言处理是指研究使用自然语言的计算机和人类之间的交互。在实践中，使用自然语言处理技术来处理和分析文本数据是非常常见的，例如 8.3 节的语言模型和 9.5 节的机器翻译模型。

要理解文本，我们可以从学习它的表示开始。利用来自大型语料库的现有文本序列，自监督学习（self-supervised learning）已被广泛用于预训练文本表示，例如通过使用周围文本的其它部分来预测文本的隐藏部分。通过这种方式，模型可以通过有监督地从海量文本数据中学习，而不需要昂贵的标签标注！

本章我们将看到：当将每个单词或子词视为单个词元时，可以在大型语料库上使用 word2vec、GloVe 或子词嵌入模型预先训练每个词元的词元。经过预训练后，每个词元的表示可以是一个向量。但是，无论上下文是什么，它都保持不变。例如，“bank”（可以译作银行或者河岸）的向量表示在“go to the bank to deposit some money”（去银行存点钱）和“go to the bank to sit down”（去河岸坐下来）中是相同的。因此，许多较新的预训练模型使相同词元的表示适应于不同的上下文，其中包括基于 Transformer 编码器的更深的自监督模型 BERT。在本章中，我们将重点讨论如何预训练文本的这种表示，如图 14.1 中所强调的那样。

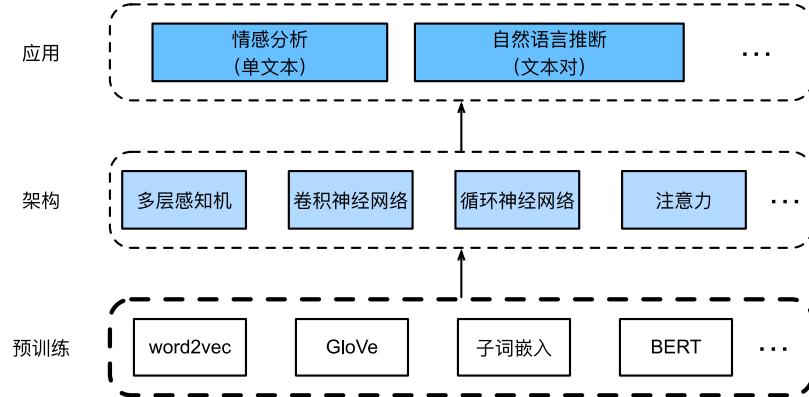


图14.1: 预训练好的文本表示可以放入各种深度学习架构, 应用于不同自然语言处理任务 (本章主要研究上游文本的预训练)

图14.1显示了预训练好的文本表示可以放入各种深度学习架构, 应用于不同自然语言处理任务。我们将在15节中介绍它们。

14.1 词嵌入 (word2vec)

自然语言是用来表达人脑思维的复杂系统。在这个系统中, 词是意义的基本单元。顾名思义, 词向量是用于表示单词意义的向量, 并且还可以被认为是单词的特征向量或表示。将单词映射到实向量的技术称为词嵌入。近年来, 词嵌入逐渐成为自然语言处理的基础知识。

14.1.1 为何独热向量是一个糟糕的选择

在8.5节中, 我们使用独热向量来表示词 (字符就是单词)。假设词典中不同词的数量 (词典大小) 为 N , 每个词对应一个从0到 $N-1$ 的不同整数 (索引)。为了得到索引为 i 的任意词的独热向量表示, 我们创建了一个全为0的长度为 N 的向量, 并将位置 i 的元素设置为1。这样, 每个词都被表示为一个长度为 N 的向量, 可以直接由神经网络使用。

虽然独热向量很容易构建, 但它们通常不是一个好的选择。一个主要原因是独热向量不能准确表达不同词之间的相似度, 比如我们经常使用的“余弦相似度”。对于向量 $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, 它们的余弦相似度是它们之间角度的余弦:

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \in [-1, 1]. \quad (14.1.1)$$

由于任意两个不同词的独热向量之间的余弦相似度为0, 所以独热向量不能编码词之间的相似性。

14.1.2 自监督的word2vec

word2vec¹⁸⁶工具是为了解决上述问题而提出的。它将每个词映射到一个固定长度的向量，这些向量能更好地表达不同词之间的相似性和类比关系。word2vec工具包含两个模型，即跳元模型（skip-gram）（Mikolov et al., 2013）和连续词袋（CBOW）（Mikolov et al., 2013）。对于在语义上有意义的表示，它们的训练依赖于条件概率，条件概率可以被看作使用语料库中一些词来预测另一些单词。由于是不带标签的数据，因此跳元模型和连续词袋都是自监督模型。

下面，我们将介绍这两种模式及其训练方法。

14.1.3 跳元模型（Skip-Gram）

跳元模型假设一个词可以用来在文本序列中生成其周围的单词。以文本序列“the”“man”“loves”“his”“son”为例。假设中心词选择“loves”，并将上下文窗口设置为2，如图图14.1.1所示，给定中心词“loves”，跳元模型考虑生成上下文词“the”“man”“him”“son”的条件概率：

$$P(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} | \text{"loves"}). \quad (14.1.2)$$

假设上下文词是在给定中心词的情况下独立生成的（即条件独立性）。在这种情况下，上述条件概率可以重写为：

$$P(\text{"the"} | \text{"loves"}) \cdot P(\text{"man"} | \text{"loves"}) \cdot P(\text{"his"} | \text{"loves"}) \cdot P(\text{"son"} | \text{"loves"}). \quad (14.1.3)$$

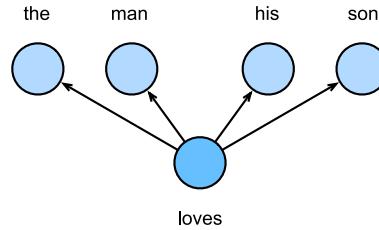


图14.1.1：跳元模型考虑了在给定中心词的情况下生成周围上下文词的条件概率

在跳元模型中，每个词都有两个 d 维向量表示，用于计算条件概率。更具体地说，对于词典中索引为 i 的任何词，分别用 $\mathbf{v}_i \in \mathbb{R}^d$ 和 $\mathbf{u}_i \in \mathbb{R}^d$ 表示其用作中心词和上下文词时的两个向量。给定中心词 w_c （词典中的索引 c ），生成任何上下文词 w_o （词典中的索引 o ）的条件概率可以通过对向量点积的softmax操作来建模：

$$P(w_o | w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}, \quad (14.1.4)$$

其中词表索引集 $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}| - 1\}$ 。给定长度为 T 的文本序列，其中时间步 t 处的词表示为 $w^{(t)}$ 。假设上下文词是在给定任何中心词的情况下独立生成的。对于上下文窗口 m ，跳元模型的似然函数是在给定任何中心词的情况下生成所有上下文词的概率：

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)} | w^{(t)}), \quad (14.1.5)$$

¹⁸⁶ <https://code.google.com/archive/p/word2vec/>

其中可以省略小于1或大于T的任何时间步。

训练

跳元模型参数是词表中每个词的中心词向量和上下文词向量。在训练中，我们通过最大化似然函数（即极大似然估计）来学习模型参数。这相当于最小化以下损失函数：

$$-\sum_{t=1}^T \sum_{-m \leq j \leq m, j \neq 0} \log P(w^{(t+j)} | w^{(t)}). \quad (14.1.6)$$

当使用随机梯度下降来最小化损失时，在每次迭代中可以随机抽样一个较短的子序列来计算该子序列的（随机）梯度，以更新模型参数。为了计算该（随机）梯度，我们需要获得对数条件概率关于中心词向量和上下文词向量的梯度。通常，根据(14.1.4)，涉及中心词 w_c 和上下文词 w_o 的对数条件概率为：

$$\log P(w_o | w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right). \quad (14.1.7)$$

通过微分，我们可以获得其相对于中心词向量 \mathbf{v}_c 的梯度为

$$\begin{aligned} \frac{\partial \log P(w_o | w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c) \mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} \left(\frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \right) \mathbf{u}_j \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} P(w_j | w_c) \mathbf{u}_j. \end{aligned} \quad (14.1.8)$$

注意，(14.1.8)中的计算需要词典中以 w_c 为中心词的所有词的条件概率。其他词向量的梯度可以以相同的方式获得。

对词典中索引为*i*的词进行训练后，得到 \mathbf{v}_i （作为中心词）和 \mathbf{u}_i （作为上下文词）两个词向量。在自然语言处理应用中，跳元模型的中心词向量通常用作词表示。

14.1.4 连续词袋 (CBOW) 模型

连续词袋 (CBOW) 模型类似于跳元模型。与跳元模型的主要区别在于，连续词袋模型假设中心词是基于其在文本序列中的周围上下文词生成的。例如，在文本序列“the”“man”“loves”“his”“son”中，在“loves”为中心词且上下文窗口为2的情况下，连续词袋模型考虑基于上下文词“the”“man”“him”“son”（如图14.1.2所示）生成中心词“loves”的条件概率，即：

$$P("loves" | "the", "man", "his", "son"). \quad (14.1.9)$$

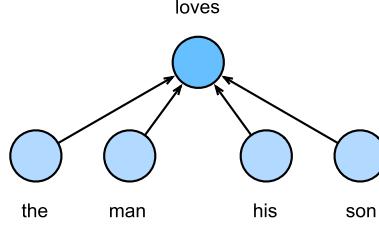


图14.1.2: 连续词袋模型考虑了给定周围上下文词生成中心词条件概率

由于连续词袋模型中存在多个上下文词，因此在计算条件概率时对这些上下文词向量进行平均。具体地说，对于字典中索引*i*的任意词，分别用 $\mathbf{v}_i \in \mathbb{R}^d$ 和 $\mathbf{u}_i \in \mathbb{R}^d$ 表示用作上下文词和中心词的两个向量（符号与跳元模型中相反）。给定上下文词 $w_{o_1}, \dots, w_{o_{2m}}$ （在词表中索引是 o_1, \dots, o_{2m} ）生成任意中心词 w_c （在词表中索引是*c*）的条件概率可以由以下公式建模：

$$P(w_c | w_{o_1}, \dots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\mathbf{u}_c^\top(\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\mathbf{u}_i^\top(\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})\right)}. \quad (14.1.10)$$

为了简洁起见，我们设为 $\mathcal{W}_o = \{w_{o_1}, \dots, w_{o_{2m}}\}$ 和 $\bar{\mathbf{v}}_o = (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}}) / (2m)$ 。那么(14.1.10)可以简化为：

$$P(w_c | \mathcal{W}_o) = \frac{\exp(\mathbf{u}_c^\top \bar{\mathbf{v}}_o)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)}. \quad (14.1.11)$$

给定长度为T的文本序列，其中时间步*t*处的词表示为 $w^{(t)}$ 。对于上下文窗口*m*，连续词袋模型的似然函数是在给定其上下文词的情况下生成所有中心词的概率：

$$\prod_{t=1}^T P(w^{(t)} | w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}). \quad (14.1.12)$$

训练

训练连续词袋模型与训练跳元模型几乎是一样的。连续词袋模型的最大似然估计等价于最小化以下损失函数：

$$-\sum_{t=1}^T \log P(w^{(t)} | w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}). \quad (14.1.13)$$

请注意，

$$\log P(w_c | \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o) \right). \quad (14.1.14)$$

通过微分，我们可以获得其关于任意上下文词向量 \mathbf{v}_{o_i} ($i = 1, \dots, 2m$) 的梯度，如下：

$$\frac{\partial \log P(w_c | \mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \bar{\mathbf{v}}_o) \mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)} \right) = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} P(w_j | \mathcal{W}_o) \mathbf{u}_j \right). \quad (14.1.15)$$

其他词向量的梯度可以以相同的方式获得。与跳元模型不同，连续词袋模型通常使用上下文词向量作为词表示。

小结

- 词向量是用于表示单词意义的向量，也可以看作词的特征向量。将词映射到实向量的技术称为词嵌入。
- word2vec工具包含跳元模型和连续词袋模型。
- 跳元模型假设一个单词可用于在文本序列中，生成其周围的单词；而连续词袋模型假设基于上下文词来生成中心单词。

练习

1. 计算每个梯度的计算复杂度是多少？如果词表很大，会有什么问题呢？
2. 英语中的一些固定短语由多个单词组成，例如“new york”。如何训练它们的词向量？提示：查看word2vec论文的第四节 (*Mikolov et al., 2013*)。
3. 让我们以跳元模型为例来思考word2vec设计。跳元模型中两个词向量的点积与余弦相似度之间有什么关系？对于语义相似的一对词，为什么它们的词向量（由跳元模型训练）的余弦相似度可能很高？

Discussions¹⁸⁷

14.2 近似训练

回想一下我们在 14.1 节中的讨论。跳元模型的主要思想是使用softmax运算来计算基于给定的中心词 w_c 生成上下文字 w_o 的条件概率（如 (14.1.4)），对应的对数损失在 (14.1.7) 给出。

由于softmax操作的性质，上下文词可以是词表 \mathcal{V} 中的任意项，(14.1.7) 包含与整个词表大小一样多的项的求和。因此，(14.1.8) 中跳元模型的梯度计算和 (14.1.15) 中的连续词袋模型的梯度计算都包含求和。不幸的是，在一个词典上（通常有几十万或数百万个单词）求和的梯度的计算成本是巨大的！

为了降低上述计算复杂度，本节将介绍两种近似训练方法：负采样和分层softmax。由于跳元模型和连续词袋模型的相似性，我们将以跳元模型为例来描述这两种近似训练方法。

14.2.1 负采样

负采样修改了原目标函数。给定中心词 w_c 的上下文窗口，任意上下文词 w_o 来自该上下文窗口的被认为是由下式建模概率的事件：

$$P(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c), \quad (14.2.1)$$

其中 σ 使用了 sigmoid 激活函数的定义：

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \quad (14.2.2)$$

¹⁸⁷ <https://discuss.d2l.ai/t/5744>

让我们从最大化文本序列中所有这些事件的联合概率开始训练词嵌入。具体而言，给定长度为 T 的文本序列，以 $w^{(t)}$ 表示时间步 t 的词，并使上下文窗口为 m ，考虑最大化联合概率：

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(D = 1 | w^{(t)}, w^{(t+j)}). \quad (14.2.3)$$

然而，(14.2.3)只考虑那些正样本的事件。仅当所有词向量都等于无穷大时，(14.2.3)中的联合概率才最大化为1。当然，这样的结果毫无意义。为了使目标函数更有意义，负采样添加从预定义分布中采样的负样本。

用 S 表示上下文词 w_o 来自中心词 w_c 的上下文窗口的事件。对于这个涉及 w_o 的事件，从预定义分布 $P(w)$ 中采样 K 个不是来自这个上下文窗口噪声词。用 N_k 表示噪声词 w_k ($k = 1, \dots, K$) 不是来自 w_c 的上下文窗口的事件。假设正例和负例 S, N_1, \dots, N_K 的这些事件是相互独立的。负采样将(14.2.3)中的联合概率（仅涉及正例）重写为

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)} | w^{(t)}), \quad (14.2.4)$$

通过事件 S, N_1, \dots, N_K 近似条件概率：

$$P(w^{(t+j)} | w^{(t)}) = P(D = 1 | w^{(t)}, w^{(t+j)}) \prod_{k=1, w_k \sim P(w)}^K P(D = 0 | w^{(t)}, w_k). \quad (14.2.5)$$

分别用 i_t 和 h_k 表示词 $w^{(t)}$ 和噪声词 w_k 在文本序列的时间步 t 处的索引。(14.2.5)中关于条件概率的对数损失为：

$$\begin{aligned} -\log P(w^{(t+j)} | w^{(t)}) &= -\log P(D = 1 | w^{(t)}, w^{(t+j)}) - \sum_{k=1, w_k \sim P(w)}^K \log P(D = 0 | w^{(t)}, w_k) \\ &= -\log \sigma(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}) - \sum_{k=1, w_k \sim P(w)}^K \log(1 - \sigma(\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t})) \\ &= -\log \sigma(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}) - \sum_{k=1, w_k \sim P(w)}^K \log \sigma(-\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}). \end{aligned} \quad (14.2.6)$$

我们可以看到，现在每个训练步的梯度计算成本与词表大小无关，而是线性依赖于 K 。当将超参数 K 设置为较小的值时，在负采样的每个训练步处的梯度的计算成本较小。

14.2.2 层序Softmax

作为另一种近似训练方法，层序Softmax (hierarchical softmax) 使用二叉树（图14.2.1中说明的数据结构），其中树的每个叶节点表示词表 \mathcal{V} 中的一个词。

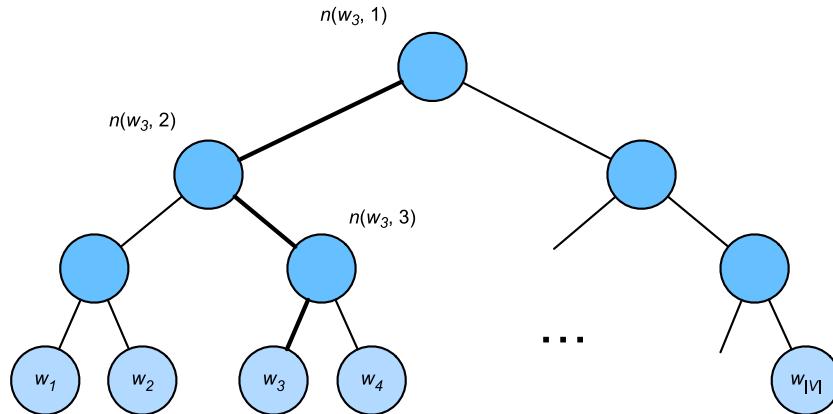


图14.2.1: 用于近似训练的分层softmax，其中树的每个叶节点表示词表中的一个词

用 $L(w)$ 表示二叉树中表示字 w 的从根节点到叶节点的路径上的节点数（包括两端）。设 $n(w, j)$ 为该路径上的 j^{th} 节点，其上下文字向量为 $\mathbf{u}_{n(w, j)}$ 。例如，图14.2.1中的 $L(w_3) = 4$ 。分层softmax将(14.1.4)中的条件概率近似为

$$P(w_o | w_c) = \prod_{j=1}^{L(w_o)-1} \sigma \left(\llbracket n(w_o, j+1) = \text{leftChild}(n(w_o, j)) \rrbracket \cdot \mathbf{u}_{n(w_o, j)}^\top \mathbf{v}_c \right), \quad (14.2.7)$$

其中函数 σ 在(14.2.2)中定义， $\text{leftChild}(n)$ 是节点 n 的左子节点：如果 x 为真， $\llbracket x \rrbracket = 1$ ；否则 $\llbracket x \rrbracket = -1$ 。

为了说明，让我们计算图14.2.1中给定词 w_c 生成词 w_3 的条件概率。这需要 w_c 的词向量 \mathbf{v}_c 和从根到 w_3 的路径（图14.2.1中加粗的路径）上的非叶节点向量之间的点积，该路径依次向左、向右和向左遍历：

$$P(w_3 | w_c) = \sigma(\mathbf{u}_{n(w_3, 1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3, 2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3, 3)}^\top \mathbf{v}_c). \quad (14.2.8)$$

由 $\sigma(x) + \sigma(-x) = 1$ ，它认为基于任意词 w_c 生成词表 \mathcal{V} 中所有词的条件概率总和为1：

$$\sum_{w \in \mathcal{V}} P(w | w_c) = 1. \quad (14.2.9)$$

幸运的是，由于二叉树结构， $L(w_o) - 1$ 大约与 $\mathcal{O}(\log_2 |\mathcal{V}|)$ 是一个数量级。当词表大小 \mathcal{V} 很大时，与没有近似训练的相比，使用分层softmax的每个训练步的计算代价显著降低。

小结

- 负采样通过考虑相互独立的事件来构造损失函数，这些事件同时涉及正例和负例。训练的计算量与每一步的噪声词数成线性关系。
- 分层softmax使用二叉树中从根节点到叶节点的路径构造损失函数。训练的计算成本取决于词表大小的对数。

练习

1. 如何在负采样中对噪声词进行采样?
2. 验证 (14.2.9) 是否有效。
3. 如何分别使用负采样和分层softmax训练连续词袋模型?

Discussions¹⁸⁸

14.3 用于预训练词嵌入的数据集

现在我们已经了解了word2vec模型的技术细节和大致的训练方法，让我们来看看它们的实现。具体地说，我们将以 14.1 节的跳元模型和 14.2 节的负采样为例。本节从用于预训练词嵌入模型的数据集开始：数据的原始格式将被转换为可以在训练期间迭代的小批量。

```
import math
import os
import random
import torch
from d2l import torch as d2l
```

14.3.1 读取数据集

我们在这里使用的数据集是Penn Tree Bank (PTB)¹⁸⁹。该语料库取自“华尔街日报”的文章，分为训练集、验证集和测试集。在原始格式中，文本文件的每一行表示由空格分隔的一句话。在这里，我们将每个单词视为一个词元。

```
#@save
d2l.DATA_HUB['ptb'] = (d2l.DATA_URL + 'ptb.zip',
                        '319d85e578af0cdc590547f26231e4e31cdf1e42')

#@save
def read_ptb():
    """将PTB数据集加载到文本行的列表中"""
    data_dir = d2l.download_extract('ptb')
    # Read the training set.
    with open(os.path.join(data_dir, 'ptb.train.txt')) as f:
        raw_text = f.read()
    return [line.split() for line in raw_text.split('\n')]
```

(continues on next page)

¹⁸⁸ <https://discuss.d2l.ai/t/5741>

¹⁸⁹ <https://catalog.ldc.upenn.edu/LDC99T42>

(continued from previous page)

```
sentences = read_ptb()  
f'# sentences数: {len(sentences)}'
```

```
Downloading ../data/ptb.zip from http://d2l-data.s3-accelerate.amazonaws.com/ptb.zip...
```

```
'# sentences数: 42069'
```

在读取训练集之后，我们为语料库构建了一个词表，其中出现次数少于10次的任何单词都将由“`<unk>`”词元替换。请注意，原始数据集还包含表示稀有（未知）单词的“`<unk>`”词元。

```
vocab = d2l.Vocab(sentences, min_freq=10)  
f'vocab size: {len(vocab)}'
```

```
'vocab size: 6719'
```

14.3.2 下采样

文本数据通常有“the”“a”和“in”等高频词：它们在非常大的语料库中甚至可能出现数十亿次。然而，这些词经常在上下文窗口中与许多不同的词共同出现，提供的有用信息很少。例如，考虑上下文窗口中的词“chip”：直观地说，它与低频单词“intel”的共现比与高频单词“a”的共现在训练中更有用。此外，大量（高频）单词的训练速度很慢。因此，当训练词嵌入模型时，可以对高频单词进行下采样 (Mikolov *et al.*, 2013)。具体地说，数据集中的每个词 w_i 将有概率地被丢弃

$$P(w_i) = \max \left(1 - \sqrt{\frac{t}{f(w_i)}}, 0 \right), \quad (14.3.1)$$

其中 $f(w_i)$ 是 w_i 的词数与数据集中的总词数的比率，常量 t 是超参数（在实验中为 10^{-4} ）。我们可以看到，只有当相对比率 $f(w_i) > t$ 时，（高频）词 w_i 才能被丢弃，且该词的相对比率越高，被丢弃的概率就越大。

```
#@save  
def subsample(sentences, vocab):  
    """下采样高频词"""  
    # 排除未知词元'<unk>'  
    sentences = [[token for token in line if vocab[token] != vocab.unk]  
                 for line in sentences]  
    counter = d2l.count_corpus(sentences)  
    num_tokens = sum(counter.values())
```

(continues on next page)

(continued from previous page)

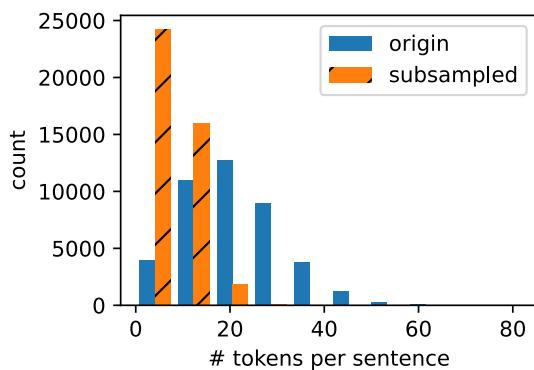
```
# 如果在下采样期间保留词元，则返回True
def keep(token):
    return(random.uniform(0, 1) <
           math.sqrt(1e-4 / counter[token] * num_tokens))

return ([[token for token in line if keep(token)] for line in sentences],
        counter)

subsampled, counter = subsample(sentences, vocab)
```

下面的代码片段绘制了下采样前后每句话的词元数量的直方图。正如预期的那样，下采样通过删除高频词来显著缩短句子，这将使训练加速。

```
d2l.show_list_len_pair_hist(
    ['origin', 'subsampled'], '# tokens per sentence',
    'count', sentences, subsampled);
```



对于单个词元，高频词“the”的采样率不到1/20。

```
def compare_counts(token):
    return (f'{token}'"的数量："
           f'之前={sum([l.count(token) for l in sentences])}, '
           f'之后={sum([l.count(token) for l in subsampled])}')
```

```
compare_counts('the')
```

```
'''the''的数量：之前=50770, 之后=2056'
```

相比之下，低频词“join”则被完全保留。

```
compare_counts('join')
```

```
'''join''的数量：之前=45，之后=45'
```

在下采样之后，我们将词元映射到它们在语料库中的索引。

```
corpus = [vocab[line] for line in subsampled]  
corpus[:3]
```

```
[], [2115, 274, 406], [140, 3, 5277, 3054, 1580]]
```

14.3.3 中心词和上下文词的提取

下面的get_centers_and_contexts函数从corpus中提取所有中心词及其上下文词。它随机采样1到max_window_size之间的整数作为上下文窗口。对于任一中心词，与其距离不超过采样上下文窗口大小的词为其上下文词。

```
#@save  
def get_centers_and_contexts(corpus, max_window_size):  
    """返回跳元模型中的中心词和上下文词"""  
    centers, contexts = [], []  
    for line in corpus:  
        # 要形成“中心词-上下文词”对，每个句子至少需要有2个词  
        if len(line) < 2:  
            continue  
        centers += line  
        for i in range(len(line)):  
            # 上下文窗口中间i  
            window_size = random.randint(1, max_window_size)  
            indices = list(range(max(0, i - window_size),  
                                min(len(line), i + 1 + window_size)))  
            # 从上下文词中排除中心词  
            indices.remove(i)  
            contexts.append([line[idx] for idx in indices])  
    return centers, contexts
```

接下来，我们创建一个人工数据集，分别包含7个和3个单词的两个句子。设置最大上下文窗口大小为2，并打印所有中心词及其上下文词。

```
tiny_dataset = [list(range(7)), list(range(7, 10))]  
print('数据集', tiny_dataset)
```

(continues on next page)

(continued from previous page)

```
for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):  
    print('中心词', center, '的上下文词是', context)
```

```
数据集 [[0, 1, 2, 3, 4, 5, 6], [7, 8, 9]]  
中心词 0 的上下文词是 [1]  
中心词 1 的上下文词是 [0, 2]  
中心词 2 的上下文词是 [0, 1, 3, 4]  
中心词 3 的上下文词是 [2, 4]  
中心词 4 的上下文词是 [3, 5]  
中心词 5 的上下文词是 [4, 6]  
中心词 6 的上下文词是 [5]  
中心词 7 的上下文词是 [8, 9]  
中心词 8 的上下文词是 [7, 9]  
中心词 9 的上下文词是 [7, 8]
```

在PTB数据集上进行训练时，我们将最大上下文窗口大小设置为5。下面提取数据集中的所有中心词及其上下文词。

```
all_centers, all_contexts = get_centers_and_contexts(corpus, 5)  
f'# “中心词-上下文词对”的数量: {sum([len(contexts) for contexts in all_contexts])}'  
  
'# “中心词-上下文词对”的数量: 1499984'
```

14.3.4 负采样

我们使用负采样进行近似训练。为了根据预定义的分布对噪声词进行采样，我们定义以下RandomGenerator类，其中（可能未规范化的）采样分布通过变量sampling_weights传递。

```
#@save  
class RandomGenerator:  
    """根据n个采样权重在{1,...,n}中随机抽取"""  
    def __init__(self, sampling_weights):  
        # Exclude  
        self.population = list(range(1, len(sampling_weights) + 1))  
        self.sampling_weights = sampling_weights  
        self.candidates = []  
        self.i = 0  
  
    def draw(self):  
        if self.i == len(self.candidates):
```

(continues on next page)

(continued from previous page)

```
# 缓存k个随机采样结果
self.candidates = random.choices(
    self.population, self.sampling_weights, k=10000)
self.i = 0
self.i += 1
return self.candidates[self.i - 1]
```

例如，我们可以在索引1、2和3中绘制10个随机变量 X ，采样概率为 $P(X = 1) = 2/9$, $P(X = 2) = 3/9$ 和 $P(X = 3) = 4/9$ ，如下所示。

```
#@save
generator = RandomGenerator([2, 3, 4])
[generator.draw() for _ in range(10)]
```

```
[1, 2, 2, 3, 3, 3, 3, 2, 1, 2]
```

对于一对中心词和上下文词，我们随机抽取了K个（实验中为5个）噪声词。根据word2vec论文中的建议，将噪声词 w 的采样概率 $P(w)$ 设置为其在字典中的相对频率，其幂为0.75 (Mikolov *et al.*, 2013)。

```
#@save
def get_negatives(all_contexts, vocab, counter, K):
    """返回负采样中的噪声词"""
    # 索引为1、2、... (索引0是词表中排除的未知标记)
    sampling_weights = [counter[vocab.to_tokens(i)]**0.75
        for i in range(1, len(vocab))]
    all_negatives, generator = [], RandomGenerator(sampling_weights)
    for contexts in all_contexts:
        negatives = []
        while len(negatives) < len(contexts) * K:
            neg = generator.draw()
            # 噪声词不能是上下文词
            if neg not in contexts:
                negatives.append(neg)
        all_negatives.append(negatives)
    return all_negatives

all_negatives = get_negatives(all_contexts, vocab, counter, 5)
```

14.3.5 小批量加载训练实例

在提取所有中心词及其上下文词和采样噪声词后，将它们转换成小批量的样本，在训练过程中可以迭代加载。

在小批量中， i^{th} 个样本包括中心词及其 n_i 个上下文词和 m_i 个噪声词。由于上下文窗口大小不同， $n_i + m_i$ 对于不同的*i*是不同的。因此，对于每个样本，我们在contexts_negatives变量中将其上下文词和噪声词连结起来，并填充零，直到连结长度达到 $\max_i n_i + m_i (\max_len)$ 。为了在计算损失时排除填充，我们定义了掩码变量masks。在masks中的元素和contexts_negatives中的元素之间存在一一对应关系，其中masks中的0（否则为1）对应于contexts_negatives中的填充。

为了区分正反例，我们在contexts_negatives中通过一个labels变量将上下文词与噪声词分开。类似于masks，在labels中的元素和contexts_negatives中的元素之间也存在一一对应关系，其中labels中的1（否则为0）对应于contexts_negatives中的上下文词的正例。

上述思想在下面的batchify函数中实现。其输入data是长度等于批量大小的列表，其中每个元素是由中心词center、其上下文词context和其噪声词negative组成的样本。此函数返回一个可以在训练期间加载用于计算的小批量，例如包括掩码变量。

```
#@save
def batchify(data):
    """返回带有负采样的跳元模型的小批量样本"""
    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, masks, labels = [], [], [], []
    for center, context, negative in data:
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [
            [context + negative + [0] * (max_len - cur_len)]]
        masks += [[1] * cur_len + [0] * (max_len - cur_len)]]
        labels += [[1] * len(context) + [0] * (max_len - len(context))]]
    return (torch.tensor(centers).reshape((-1, 1)), torch.tensor(
        contexts_negatives), torch.tensor(masks), torch.tensor(labels))
```

让我们使用一个小批量的两个样本来测试此函数。

```
x_1 = (1, [2, 2], [3, 3, 3])
x_2 = (1, [2, 2, 2], [3, 3])
batch = batchify((x_1, x_2))

names = ['centers', 'contexts_negatives', 'masks', 'labels']
for name, data in zip(names, batch):
    print(name, '=', data)
```

```

centers = tensor([[1,
    [1]]])
contexts_negatives = tensor([[2, 2, 3, 3, 3, 3],
    [2, 2, 2, 3, 3, 0]])
masks = tensor([[1, 1, 1, 1, 1, 1],
    [1, 1, 1, 1, 0]])
labels = tensor([[1, 1, 0, 0, 0, 0],
    [1, 1, 1, 0, 0, 0]])

```

14.3.6 整合代码

最后，我们定义了读取PTB数据集并返回数据迭代器和词表的load_data_ptb函数。

```

#@save
def load_data_ptb(batch_size, max_window_size, num_noise_words):
    """下载PTB数据集，然后将其加载到内存中"""
    num_workers = d2l.get_dataloader_workers()
    sentences = read_ptb()
    vocab = d2l.Vocab(sentences, min_freq=10)
    subsampled, counter = subsample(sentences, vocab)
    corpus = [vocab[line] for line in subsampled]
    all_centers, all_contexts = get_centers_and_contexts(
        corpus, max_window_size)
    all_negatives = get_negatives(
        all_contexts, vocab, counter, num_noise_words)

    class PTBDataset(torch.utils.data.Dataset):
        def __init__(self, centers, contexts, negatives):
            assert len(centers) == len(contexts) == len(negatives)
            self.centers = centers
            self.contexts = contexts
            self.negatives = negatives

        def __getitem__(self, index):
            return (self.centers[index], self.contexts[index],
                    self.negatives[index])

        def __len__(self):
            return len(self.centers)

    dataset = PTBDataset(all_centers, all_contexts, all_negatives)

```

(continues on next page)

(continued from previous page)

```
data_iter = torch.utils.data.DataLoader(  
    dataset, batch_size, shuffle=True,  
    collate_fn=batchify, num_workers=num_workers)  
return data_iter, vocab
```

让我们打印数据迭代器的第一个小批量。

```
data_iter, vocab = load_data_ptb(512, 5, 5)  
for batch in data_iter:  
    for name, data in zip(names, batch):  
        print(name, 'shape:', data.shape)  
    break
```

```
centers shape: torch.Size([512, 1])  
contexts_negatives shape: torch.Size([512, 60])  
masks shape: torch.Size([512, 60])  
labels shape: torch.Size([512, 60])
```

小结

- 高频词在训练中可能不是那么有用。我们可以对他们进行下采样，以便在训练中加快速度。
- 为了提高计算效率，我们以小批量方式加载样本。我们可以定义其他变量来区分填充标记和非填充标记，以及正例和负例。

练习

1. 如果不使用下采样，本节中代码的运行时间会发生什么变化？
2. RandomGenerator类缓存k个随机采样结果。将k设置为其他值，看看它如何影响数据加载速度。
3. 本节代码中的哪些其他超参数可能会影响数据加载速度？

Discussions¹⁹⁰

¹⁹⁰ <https://discuss.d2l.ai/t/5735>

14.4 预训练word2vec

我们继续实现 14.1 节中定义的跳元语法模型。然后，我们将在PTB数据集上使用负采样预训练word2vec。首先，让我们通过调用d2l.load_data_ptb函数来获得该数据集的数据迭代器和词表，该函数在 14.3 节中进行了描述。

```
import math
import torch
from torch import nn
from d2l import torch as d2l

batch_size, max_window_size, num_noise_words = 512, 5, 5
data_iter, vocab = d2l.load_data_ptb(batch_size, max_window_size,
                                      num_noise_words)
```

14.4.1 跳元模型

我们通过嵌入层和批量矩阵乘法实现了跳元模型。首先，让我们回顾一下嵌入层是如何工作的。

嵌入层

如 9.7 节中所述，嵌入层将词元的索引映射到其特征向量。该层的权重是一个矩阵，其行数等于字典大小 (input_dim)，列数等于每个标记的向量维数 (output_dim)。在词嵌入模型训练之后，这个权重就是我们需要的。

```
embed = nn.Embedding(num_embeddings=20, embedding_dim=4)
print(f'Parameter embedding_weight ({embed.weight.shape}, '
      f'dtype={embed.weight.dtype})')
```

```
Parameter embedding_weight (torch.Size([20, 4]), dtype=torch.float32)
```

嵌入层的输入是词元（词）的索引。对于任何词元素索引 i ，其向量表示可以从嵌入层中的权重矩阵的第 i 行获得。由于向量维度 (output_dim) 被设置为4，因此当小批量词元素索引的形状为 (2, 3) 时，嵌入层返回具有形状 (2, 3, 4) 的向量。

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
embed(x)
```

```
tensor([[-1.4754, -0.3612, -0.4246,  0.5805],
       [-0.3160,  0.8830,  0.5328,  0.2179],
       [-0.0378, -0.5559,  1.4525,  0.6230]],

      [[ 0.0829, -1.0549,  0.6381,  0.7886],
       [-0.3862, -0.1291,  0.4160, -0.6710],
       [-0.4056,  0.0370, -0.6308, -0.2865]]], grad_fn=<EmbeddingBackward0>)
```

定义前向传播

在前向传播中，跳元语法模型的输入包括形状为（批量大小， 1）的中心词索引center和形状为（批量大小， max_len）的上下文与噪声词索引contexts_and_negatives，其中max_len在 14.3.5节中定义。这两个变量首先通过嵌入层从词元索引转换成向量，然后它们的批量矩阵相乘（在 10.2.4节中描述）返回形状为（批量大小， 1， max_len）的输出。输出中的每个元素是中心词向量和上下文或噪声词向量的点积。

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = torch.bmm(v, u.permute(0, 2, 1))
    return pred
```

让我们为一些样例输入打印此skip_gram函数的输出形状。

```
skip_gram(torch.ones((2, 1), dtype=torch.long),
          torch.ones((2, 4), dtype=torch.long), embed, embed).shape
```

```
torch.Size([2, 1, 4])
```

14.4.2 训练

在训练带负采样的跳元模型之前，我们先定义它的损失函数。

二元交叉熵损失

根据 14.2.1节中负采样损失函数的定义，我们将使用二元交叉熵损失。

```
class SigmoidBCELoss(nn.Module):
    # 带掩码的二元交叉熵损失
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
super().__init__()

def forward(self, inputs, target, mask=None):
    out = nn.functional.binary_cross_entropy_with_logits(
        inputs, target, weight=mask, reduction="none")
    return out.mean(dim=1)

loss = SigmoidBCELoss()
```

回想一下我们在 14.3.5 节中对掩码变量和标签变量的描述。下面计算给定变量的二进制交叉熵损失。

```
pred = torch.tensor([[1.1, -2.2, 3.3, -4.4]] * 2)
label = torch.tensor([[1.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0]])
mask = torch.tensor([[1, 1, 1, 1], [1, 1, 0, 0]])
loss(pred, label, mask) * mask.shape[1] / mask.sum(axis=1)
```

```
tensor([0.9352, 1.8462])
```

下面显示了如何使用二元交叉熵损失中的Sigmoid激活函数（以较低效率的方式）计算上述结果。我们可以将这两个输出视为两个规范化的损失，在非掩码预测上进行平均。

```
def sigmoid(x):
    return -math.log(1 / (1 + math.exp(-x)))

print(f'{(sigmoid(1.1) + sigmoid(2.2) + sigmoid(-3.3) + sigmoid(4.4)) / 4:.4f}')
print(f'{(sigmoid(-1.1) + sigmoid(-2.2)) / 2:.4f}')
```

```
0.9352
1.8462
```

初始化模型参数

我们定义了两个嵌入层，将词表中的所有单词分别作为中心词和上下文词使用。字向量维度`embed_size`被设置为100。

```
embed_size = 100
net = nn.Sequential(nn.Embedding(num_embeddings=len(vocab),
                                 embedding_dim=embed_size),
                    nn.Embedding(num_embeddings=len(vocab),
                                 embedding_dim=embed_size))
```

定义训练阶段代码

训练阶段代码实现定义如下。由于填充的存在，损失函数的计算与以前的训练函数略有不同。

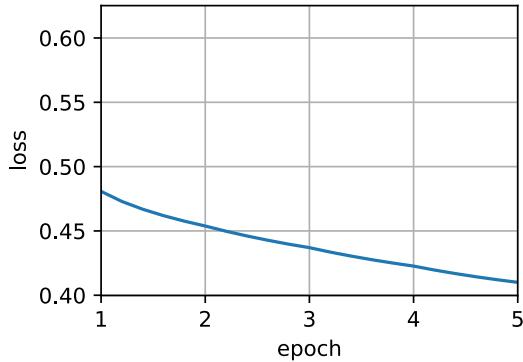
```
def train(net, data_iter, lr, num_epochs, device=d2l.try_gpu()):
    def init_weights(m):
        if type(m) == nn.Embedding:
            nn.init.xavier_uniform_(m.weight)
    net.apply(init_weights)
    net = net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                             xlim=[1, num_epochs])
    # 规范化的损失之和，规范化的损失数
    metric = d2l.Accumulator(2)
    for epoch in range(num_epochs):
        timer, num_batches = d2l.Timer(), len(data_iter)
        for i, batch in enumerate(data_iter):
            optimizer.zero_grad()
            center, context_negative, mask, label = [
                data.to(device) for data in batch]

            pred = skip_gram(center, context_negative, net[0], net[1])
            l = (loss(pred.reshape(label.shape).float(), label.float(), mask)
                 / mask.sum(axis=1) * mask.shape[1])
            l.sum().backward()
            optimizer.step()
            metric.add(l.sum(), l.numel())
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, '
          f'{metric[1] / timer.stop():.1f} tokens/sec on {str(device)})')
```

现在，我们可以使用负采样来训练跳元模型。

```
lr, num_epochs = 0.002, 5
train(net, data_iter, lr, num_epochs)
```

```
loss 0.410, 377799.5 tokens/sec on cuda:0
```



14.4.3 应用词嵌入

在训练word2vec模型之后，我们可以使用训练好模型中词向量的余弦相似度来从词表中找到与输入单词语义最相似的单词。

```
def get_similar_tokens(query_token, k, embed):
    W = embed.weight.data
    x = W[vocab[query_token]]
    # 计算余弦相似性。增加1e-9以获得数值稳定性
    cos = torch.mv(W, x) / torch.sqrt(torch.sum(W * W, dim=1)) *
          torch.sum(x * x) + 1e-9
    topk = torch.topk(cos, k=k+1)[1].cpu().numpy().astype('int32')
    for i in topk[1:]: # 删除输入词
        print(f'cosine sim={float(cos[i]):.3f}: {vocab.to_tokens(i)}')

get_similar_tokens('chip', 3, net[0])
```

```
cosine sim=0.773: microprocessor
cosine sim=0.589: hitachi
cosine sim=0.582: computers
```

小结

- 我们可以使用嵌入层和二元交叉熵损失来训练带负采样的跳元模型。
- 词嵌入的应用包括基于词向量的余弦相似度为给定词找到语义相似的词。

练习

1. 使用训练好的模型，找出其他输入词在语义上相似的词。您能通过调优超参数来改进结果吗？
2. 当训练语料库很大时，在更新模型参数时，我们经常对当前小批量的中心词进行上下文词和噪声词的采样。换言之，同一中心词在不同的训练迭代轮数可以有不同的上下文词或噪声词。这种方法的好处是什么？尝试实现这种训练方法。

Discussions¹⁹¹

14.5 全局向量的词嵌入（GloVe）

上下文窗口内的词共现可以携带丰富的语义信息。例如，在一个大型语料库中，“固体”比“气体”更有可能与“冰”共现，但“气体”一词与“蒸汽”的共现频率可能比与“冰”的共现频率更高。此外，可以预先计算此类共现的全局语料库统计数据：这可以提高训练效率。为了利用整个语料库中的统计信息进行词嵌入，让我们首先回顾 14.1.3 节中的跳元模型，但是使用全局语料库统计（如共现计数）来解释它。

14.5.1 带全局语料统计的跳元模型

用 q_{ij} 表示词 w_j 的条件概率 $P(w_j | w_i)$ ，在跳元模型中给定词 w_i ，我们有：

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)}, \quad (14.5.1)$$

其中，对于任意索引 i ，向量 \mathbf{v}_i 和 \mathbf{u}_i 分别表示词 w_i 作为中心词和上下文词，且 $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}| - 1\}$ 是词表的索引集。

考虑词 w_i 可能在语料库中出现多次。在整个语料库中，所有以 w_i 为中心词的上下文词形成一个词索引的多重集 \mathcal{C}_i ，该索引允许同一元素的多个实例。对于任何元素，其实例数称为其重数。举例说明，假设词 w_i 在语料库中出现两次，并且在两个上下文窗口中以 w_i 为其中心词的上下文词索引是 k, j, m, k 和 k, l, k, j 。因此，多重集 $\mathcal{C}_i = \{j, j, k, k, k, k, l, m\}$ ，其中元素 j, k, l, m 的重数分别为 2、4、1、1。

现在，让我们将多重集 \mathcal{C}_i 中的元素 j 的重数表示为 x_{ij} 。这是词 w_j （作为上下文词）和词 w_i （作为中心词）在整个语料库的同一上下文窗口中的全局共现计数。使用这样的全局语料库统计，跳元模型的损失函数等价于：

$$-\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} x_{ij} \log q_{ij}. \quad (14.5.2)$$

我们用 x_i 表示上下文窗口中的所有上下文词的数量，其中 w_i 作为它们的中心词出现，这相当于 $|\mathcal{C}_i|$ 。设 p_{ij} 为用于生成上下文词 w_j 的条件概率 x_{ij}/x_i 。给定中心词 w_i ，(14.5.2) 可以重写为：

$$-\sum_{i \in \mathcal{V}} x_i \sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}. \quad (14.5.3)$$

在 (14.5.3) 中， $-\sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}$ 计算全局语料统计的条件分布 p_{ij} 和模型预测的条件分布 q_{ij} 的交叉熵。如上所述，这一损失也按 x_i 加权。在 (14.5.3) 中最小化损失函数将使预测的条件分布接近全局语料库统计中的条件分布。

¹⁹¹ <https://discuss.d2l.ai/t/5740>

虽然交叉熵损失函数通常用于测量概率分布之间的距离，但在这里可能不是一个好的选择。一方面，正如我们在 14.2 节中提到的，规范化 q_{ij} 的代价在于整个词表的求和，这在计算上可能非常昂贵。另一方面，来自大型语料库的大量罕见事件往往被交叉熵损失建模，从而赋予过多的权重。

14.5.2 GloVe模型

有鉴于此，GloVe模型基于平方损失 (Pennington *et al.*, 2014)对跳元模型做了三个修改：

1. 使用变量 $p'_{ij} = x_{ij}$ 和 $q'_{ij} = \exp(\mathbf{u}_j^\top \mathbf{v}_i)$ 而非概率分布，并取两者的对数。所以平方损失项是 $(\log p'_{ij} - \log q'_{ij})^2 = (\mathbf{u}_j^\top \mathbf{v}_i - \log x_{ij})^2$ 。
2. 为每个词 w_i 添加两个标量模型参数：中心词偏置 b_i 和上下文词偏置 c_i 。
3. 用权重函数 $h(x_{ij})$ 替换每个损失项的权重，其中 $h(x)$ 在 $[0, 1]$ 的间隔内递增。

整合代码，训练GloVe是为了尽量降低以下损失函数：

$$\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} h(x_{ij}) (\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log x_{ij})^2. \quad (14.5.4)$$

对于权重函数，建议的选择是：当 $x < c$ (例如, $c = 100$) 时, $h(x) = (x/c)^\alpha$ (例如 $\alpha = 0.75$)；否则 $h(x) = 1$ 。在这种情况下，由于 $h(0) = 0$ ，为了提高计算效率，可以省略任意 $x_{ij} = 0$ 的平方损失项。例如，当使用小批量随机梯度下降进行训练时，在每次迭代中，我们随机抽样一小批量非零的 x_{ij} 来计算梯度并更新模型参数。注意，这些非零的 x_{ij} 是预先计算的全局语料库统计数据；因此，该模型GloVe被称为全局向量。

应该强调的是，当词 w_i 出现在词 w_j 的上下文窗口时，词 w_j 也出现在词 w_i 的上下文窗口。因此， $x_{ij} = x_{ji}$ 。与拟合非对称条件概率 p_{ij} 的 word2vec 不同，GloVe 拟合对称概率 $\log x_{ij}$ 。因此，在 GloVe 模型中，任意词的中心词向量和上下文词向量在数学上是等价的。但在实际应用中，由于初始值不同，同一个词经过训练后，在这两个向量中可能得到不同的值：GloVe 将它们相加作为输出向量。

14.5.3 从条件概率比值理解GloVe模型

我们也可以从另一个角度来理解GloVe模型。使用 14.5.1 节中的相同符号，设 $p_{ij} \stackrel{\text{def}}{=} P(w_j | w_i)$ 为生成上下文词 w_j 的条件概率，给定 w_i 作为语料库中的中心词。`tab_glove`根据大量语料库的统计数据，列出了给定单词“ice”和“steam”的共现概率及其比值。

大型语料库中的词-词共现概率及其比值（根据 (Pennington *et al.*, 2014) 中的表1改编）

表14.5.1: label:`tab_glove`

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

从 `tab_glove` 中，我们可以观察到以下几点：

- 对于与“ice”相关但与“steam”无关的单词 w_k , 例如 $w_k = \text{solid}$, 我们预计会有更大的共现概率比值, 例如8.9。
- 对于与“steam”相关但与“ice”无关的单词 w_k , 例如 $w_k = \text{gas}$, 我们预计较小的共现概率比值, 例如0.085。
- 对于同时与“ice”和“steam”相关的单词 w_k , 例如 $w_k = \text{water}$, 我们预计其共现概率的比值接近1, 例如1.36。
- 对于与“ice”和“steam”都不相关的单词 w_k , 例如 $w_k = \text{fashion}$, 我们预计共现概率的比值接近1, 例如0.96。

由此可见, 共现概率的比值能够直观地表达词与词之间的关系。因此, 我们可以设计三个词向量的函数来拟合这个比值。对于共现概率 p_{ij}/p_{ik} 的比值, 其中 w_i 是中心词, w_j 和 w_k 是上下文词, 我们希望使用某个函数 f 来拟合该比值:

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) \approx \frac{p_{ij}}{p_{ik}}. \quad (14.5.5)$$

在 f 的许多可能的设计中, 我们只在以下几点中选择了一个合理的选择。因为共现概率的比值是标量, 所以我们要求 f 是标量函数, 例如 $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = f((\mathbf{u}_j - \mathbf{u}_k)^\top \mathbf{v}_i)$ 。在(14.5.5)中交换词索引 j 和 k , 它必须保持 $f(x)f(-x) = 1$, 所以一种可能性是 $f(x) = \exp(x)$, 即:

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\exp(\mathbf{u}_k^\top \mathbf{v}_i)} \approx \frac{p_{ij}}{p_{ik}}. \quad (14.5.6)$$

现在让我们选择 $\exp(\mathbf{u}_j^\top \mathbf{v}_i) \approx \alpha p_{ij}$, 其中 α 是常数。从 $p_{ij} = x_{ij}/x_i$ 开始, 取两边的对数得到 $\mathbf{u}_j^\top \mathbf{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$ 。我们可以使用附加的偏置项来拟合 $-\log \alpha + \log x_i$, 如中心词偏置 b_i 和上下文词偏置 c_j :

$$\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j \approx \log x_{ij}. \quad (14.5.7)$$

通过对(14.5.7)的加权平方误差的度量, 得到了(14.5.4)的GloVe损失函数。

小结

- 诸如词-词共现计数的全局语料库统计可以来解释跳元模型。
- 交叉熵损失可能不是衡量两种概率分布差异的好选择, 特别是对于大型语料库。GloVe使用平方损失来拟合预先计算的全局语料库统计数据。
- 对于GloVe中的任意词, 中心词向量和上下文词向量在数学上是等价的。
- GloVe可以从词-词共现概率的比率来解释。

练习

- 如果词 w_i 和 w_j 在同一上下文窗口中同时出现，我们如何使用它们在文本序列中的距离来重新设计计算条件概率 p_{ij} 的方法？提示：参见GloVe论文(Pennington et al., 2014)的第4.2节。
- 对于任何一个词，它的中心词偏置和上下文偏置在数学上是等价的吗？为什么？

Discussions¹⁹²

14.6 子词嵌入

在英语中，“helps”“helped”和“helping”等单词都是同一个词“help”的变形形式。“dog”和“dogs”之间的关系与“cat”和“cats”之间的关系相同，“boy”和“boyfriend”之间的关系与“girl”和“girlfriend”之间的关系相同。在法语和西班牙语等其他语言中，许多动词有40多种变形形式，而在芬兰语中，名词最多可能有15种变形。在语言学中，形态学研究单词形成和词汇关系。但是，word2vec和GloVe都没有对词的内部结构进行探讨。

14.6.1 fastText模型

回想一下词在word2vec中是如何表示的。在跳元模型和连续词袋模型中，同一词的不同变形形式直接由不同的向量表示，不需要共享参数。为了使用形态信息，fastText模型提出了一种子词嵌入方法，其中子词是一个字符n-gram(Bojanowski et al., 2017)。fastText可以被认为是子词级跳元模型，而非学习词级向量表示，其中每个中心词由其子词级向量之和表示。

让我们来说明如何以单词“where”为例获得fastText中每个中心词的子词。首先，在词的开头和末尾添加特殊字符“<”和“>”，以将前缀和后缀与其他子词区分开来。然后，从词中提取字符n-gram。例如，值 $n = 3$ 时，我们将获得长度为3的所有子词：“<wh”“whe”“her”“ere”“re>”和特殊子词“<where>”。

在fastText中，对于任意词 w ，用 \mathcal{G}_w 表示其长度在3和6之间的所有子词与其特殊子词的并集。词表是所有词的子词的集合。假设 \mathbf{z}_g 是词典中的子词 g 的向量，则跳元模型中作为中心词的词 w 的向量 \mathbf{v}_w 是其子词向量的和：

$$\mathbf{v}_w = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g. \quad (14.6.1)$$

fastText的其余部分与跳元模型相同。与跳元模型相比，fastText的词量更大，模型参数也更多。此外，为了计算一个词的表示，它的所有子词向量都必须求和，这导致了更高的计算复杂度。然而，由于具有相似结构的词之间共享来自子词的参数，罕见词甚至词表外的词在fastText中可能获得更好的向量表示。

¹⁹² <https://discuss.d2l.ai/t/5736>

14.6.2 字节对编码（Byte Pair Encoding）

在fastText中，所有提取的子词都必须是指定的长度，例如3到6，因此词表大小不能预定义。为了在固定大小的词表中允许可变长度的子词，我们可以应用一种称为字节对编码（Byte Pair Encoding, BPE）的压缩算法来提取子词 (Sennrich *et al.*, 2015)。

字节对编码执行训练数据集的统计分析，以发现单词内的公共符号，诸如任意长度的连续字符。从长度为1的符号开始，字节对编码迭代地合并最频繁的连续符号对以产生新的更长的符号。请注意，为提高效率，不考虑跨越单词边界的对。最后，我们可以使用像子词这样的符号来切分单词。字节对编码及其变体已经用于诸如GPT-2 (Radford *et al.*, 2019)和RoBERTa (Liu *et al.*, 2019)等自然语言处理预训练模型中的输入表示。在下面，我们将说明字节对编码是如何工作的。

首先，我们将符号词表初始化为所有英文小写字符、特殊的词尾符号'_'和特殊的未知符号'[UNK]'。

```
import collections

symbols = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
           'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
           '_', '[UNK']']
```

因为我们不考虑跨越词边界的符号对，所以我们只需要一个字典raw_token_freqs将词映射到数据集中的频率(出现次数)。注意，特殊符号'_'被附加到每个词的尾部，以便我们可以容易地从输出符号序列(例如，“a_all er_man”)恢复单词序列(例如，“a_all er_man”)。由于我们仅从单个字符和特殊符号的词开始合并处理，所以在每个词(词典token_freqs的键)内的每对连续字符之间插入空格。换句话说，空格是词中符号之间的分隔符。

```
raw_token_freqs = {'fast_': 4, 'faster_': 3, 'tall_': 5, 'taller_': 4}
token_freqs = {}
for token, freq in raw_token_freqs.items():
    token_freqs[' '.join(list(token))] = raw_token_freqs[token]
token_freqs
```

```
{'f a s t _': 4, 'f a s t e r _': 3, 't a l l _': 5, 't a l l e r _': 4}
```

我们定义以下get_max_freq_pair函数，其返回词内最频繁的连续符号对，其中词来自输入词典token_freqs的键。

```
def get_max_freq_pair(token_freqs):
    pairs = collections.defaultdict(int)
    for token, freq in token_freqs.items():
        symbols = token.split()
        for i in range(len(symbols) - 1):
            # “pairs”的键是两个连续符号的元组
```

(continues on next page)

(continued from previous page)

```
    pairs[symbols[i], symbols[i + 1]] += freq
return max(pairs, key=pairs.get) # 具有最大值的“pairs”键
```

作为基于连续符号频率的贪心方法，字节对编码将使用以下`merge_symbols`函数来合并最频繁的连续符号对以产生新符号。

```
def merge_symbols(max_freq_pair, token_freqs, symbols):
    symbols.append(''.join(max_freq_pair))
    new_token_freqs = dict()
    for token, freq in token_freqs.items():
        new_token = token.replace(''.join(max_freq_pair),
                                  ''.join(max_freq_pair))
        new_token_freqs[new_token] = token_freqs[token]
    return new_token_freqs
```

现在，我们对词典`token_freqs`的键迭代地执行字节对编码算法。在第一次迭代中，最频繁的连续符号对是't'和'a'，因此字节对编码将它们合并以产生新符号'ta'。在第二次迭代中，字节对编码继续合并'ta'和'l'以产生另一个新符号'tal'。

```
num_merges = 10
for i in range(num_merges):
    max_freq_pair = get_max_freq_pair(token_freqs)
    token_freqs = merge_symbols(max_freq_pair, token_freqs, symbols)
    print(f'合并# {i+1}: {max_freq_pair}'
```

```
合并# 1: ('t', 'a')
合并# 2: ('ta', 'l')
合并# 3: ('tal', 'l')
合并# 4: ('f', 'a')
合并# 5: ('fa', 's')
合并# 6: ('fas', 't')
合并# 7: ('e', 'r')
合并# 8: ('er', '_')
合并# 9: ('tall', '_')
合并# 10: ('fast', '_')
```

在字节对编码的10次迭代之后，我们可以看到列表`symbols`现在又包含10个从其他符号迭代合并而来的符号。

```
print(symbols)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u  
↪', 'v', 'w', 'x', 'y', 'z', '_', '[UNK]', 'ta', 'tal', 'tall', 'fa', 'fas', 'fast', 'er', 'er_',
↪'tall_', 'fast_']
```

对于在词典raw_token_freqs的键中指定的同一数据集，作为字节对编码算法的结果，数据集中的每个词现在被子词“fast_”“fast”“er_”“tall_”和“tall”分割。例如，单词“fast er_”和“tall er_”分别被分割为“fast er_”和“tall er_”。

```
print(list(token_freqs.keys()))
```

```
['fast_', 'fast er_', 'tall_', 'tall er_']
```

请注意，字节对编码的结果取决于正在使用的数据集。我们还可以使用从一个数据集学习的子词来切分另一个数据集的单词。作为一种贪心方法，下面的segment_BPE函数尝试将单词从输入参数symbols分成可能最长的子词。

```
def segment_BPE(tokens, symbols):
    outputs = []
    for token in tokens:
        start, end = 0, len(token)
        cur_output = []
        # 具有符号中可能最长子字的词元段
        while start < len(token) and start < end:
            if token[start: end] in symbols:
                cur_output.append(token[start: end])
                start = end
                end = len(token)
            else:
                end -= 1
        if start < len(token):
            cur_output.append('[UNK]')
        outputs.append(' '.join(cur_output))
    return outputs
```

我们使用列表symbols中的子词（从前面提到的数据集学习）来表示另一个数据集的tokens。

```
tokens = ['tallest_', 'fatter_']
print(segment_BPE(tokens, symbols))
```

```
['tall e s t _', 'fa t t er_']
```

小结

- fastText模型提出了一种子词嵌入方法：基于word2vec中的跳元模型，它将中心词表示为其子词向量之和。
- 字节对编码执行训练数据集的统计分析，以发现词内的公共符号。作为一种贪心方法，字节对编码迭代地合并最频繁的连续符号对。
- 子词嵌入可以提高稀有词和词典外词的表示质量。

练习

1. 例如，英语中大约有 3×10^8 种可能的6-元组。子词太多会有什么问题呢？如何解决这个问题？提示：请参阅fastText论文第3.2节末尾(Bojanowski *et al.*, 2017)。
2. 如何在连续词袋模型的基础上设计一个子词嵌入模型？
3. 要获得大小为 m 的词表，当初始符号词表大小为 n 时，需要多少合并操作？
4. 如何扩展字节对编码的思想来提取短语？

Discussions¹⁹³

14.7 词的相似性和类比任务

在 14.4 节中，我们在一个小的数据集上训练了一个 word2vec 模型，并使用它为一个输入词寻找语义相似的词。实际上，在大型语料库上预先训练的词向量可以应用于下游的自然语言处理任务，这将在后面的 15 节中讨论。为了直观地演示大型语料库中预训练词向量的语义，让我们将预训练词向量应用到词的相似性和类比任务中。

```
import os
import torch
from torch import nn
from d2l import torch as d2l
```

¹⁹³ <https://discuss.d2l.ai/t/5748>

14.7.1 加载预训练词向量

以下列出维度为50、100和300的预训练GloVe嵌入，可从[GloVe网站¹⁹⁴](#)下载。预训练的fastText嵌入有多种语言。这里我们使用可以从[fastText网站¹⁹⁵](#)下载300维度的英文版本（“wiki.en”）。

```
#@save
d2l.DATA_HUB['glove.6b.50d'] = (d2l.DATA_URL + 'glove.6B.50d.zip',
                                  '0b8703943ccdb6eb788e6f091b8946e82231bc4d')

#@save
d2l.DATA_HUB['glove.6b.100d'] = (d2l.DATA_URL + 'glove.6B.100d.zip',
                                   'cd43bfb07e44e6f27cbcc7bc9ae3d80284fdaf5a')

#@save
d2l.DATA_HUB['glove.42b.300d'] = (d2l.DATA_URL + 'glove.42B.300d.zip',
                                   'b5116e234e9eb9076672cfeabf5469f3eec904fa')

#@save
d2l.DATA_HUB['wiki.en'] = (d2l.DATA_URL + 'wiki.en.zip',
                           'c1816da3821ae9f43899be655002f6c723e91b88')
```

为了加载这些预训练的GloVe和fastText嵌入，我们定义了以下TokenEmbedding类。

```
#@save
class TokenEmbedding:
    """GloVe嵌入"""
    def __init__(self, embedding_name):
        self.idx_to_token, self.idx_to_vec = self._load_embedding(
            embedding_name)
        self.unknown_idx = 0
        self.token_to_idx = {token: idx for idx, token in
                            enumerate(self.idx_to_token)}

    def _load_embedding(self, embedding_name):
        idx_to_token, idx_to_vec = ['<unk>'], []
        data_dir = d2l.download_extract(embedding_name)
        # GloVe网站: https://nlp.stanford.edu/projects/glove/
        # fastText网站: https://fasttext.cc/
        with open(os.path.join(data_dir, 'vec.txt'), 'r') as f:
            for line in f:
                elems = line.rstrip().split(' ')
                token, elems = elems[0], [float(elem) for elem in elems[1:]]
```

(continues on next page)

¹⁹⁴ <https://nlp.stanford.edu/projects/glove/>

¹⁹⁵ <https://fasttext.cc/>

(continued from previous page)

```
# 跳过标题信息，例如fastText中的首行
if len(elems) > 1:
    idx_to_token.append(token)
    idx_to_vec.append(elems)

idx_to_vec = [[0] * len(idx_to_vec[0])] + idx_to_vec
return idx_to_token, torch.tensor(idx_to_vec)

def __getitem__(self, tokens):
    indices = [self.token_to_idx.get(token, self.unknown_idx)
               for token in tokens]
    vecs = self.idx_to_vec[torch.tensor(indices)]
    return vecs

def __len__(self):
    return len(self.idx_to_token)
```

下面我们加载50维GloVe嵌入（在维基百科的子集上预训练）。创建TokenEmbedding实例时，如果尚未下载指定的嵌入文件，则必须下载该文件。

```
glove_6b50d = TokenEmbedding('glove.6b.50d')
```

```
Downloading ../data/glove.6B.50d.zip from http://d2l-data.s3-accelerate.amazonaws.com/glove.6B.50d.zip...
→ .
```

输出词表大小。词表包含400000个词（词元）和一个特殊的未知词元。

```
len(glove_6b50d)
```

```
400001
```

我们可以得到词表中一个单词的索引，反之亦然。

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

```
(3367, 'beautiful')
```

14.7.2 应用预训练词向量

使用加载的GloVe向量，我们将通过下面的词相似性和类比任务中来展示词向量的语义。

词相似度

与 14.4.3节类似，为了根据词向量之间的余弦相似性为输入词查找语义相似的词，我们实现了以下knn（ k 近邻）函数。

```
def knn(W, x, k):
    # 增加1e-9以获得数值稳定性
    cos = torch.mv(W, x.reshape(-1,)) / (
        torch.sqrt(torch.sum(W * W, axis=1) + 1e-9) *
        torch.sqrt((x * x).sum()))
    _, topk = torch.topk(cos, k=k)
    return topk, [cos[int(i)] for i in topk]
```

然后，我们使用TokenEmbedding的实例embed中预训练好的词向量来搜索相似的词。

```
def get_similar_tokens(query_token, k, embed):
    topk, cos = knn(embed.idx_to_vec, embed[[query_token]], k + 1)
    for i, c in zip(topk[1:], cos[1:]): # 排除输入词
        print(f'{embed.idx_to_token[int(i)]}: cosine相似度={float(c):.3f}'')
```

glove_6b50d中预训练词向量的词表包含400000个词和一个特殊的未知词元。排除输入词和未知词元后，我们在词表中找到与“chip”一词语义最相似的三个词。

```
get_similar_tokens('chip', 3, glove_6b50d)
```

```
chips: cosine相似度=0.856
intel: cosine相似度=0.749
electronics: cosine相似度=0.749
```

下面输出与“baby”和“beautiful”相似的词。

```
get_similar_tokens('baby', 3, glove_6b50d)
```

```
babies: cosine相似度=0.839
boy: cosine相似度=0.800
girl: cosine相似度=0.792
```

```
get_similar_tokens('beautiful', 3, glove_6b50d)
```

```
lovely: cosine相似度=0.921  
gorgeous: cosine相似度=0.893  
wonderful: cosine相似度=0.830
```

词类比

除了找到相似的词，我们还可以将词向量应用到词类比任务中。例如，“man”：“woman”::“son”：“daughter”是一个词的类比。“man”是对“woman”的类比，“son”是对“daughter”的类比。具体来说，词类比任务可以定义为：对于单词类比 $a:b::c:d$ ，给出前三个词 a 、 b 和 c ，找到 d 。用 $\text{vec}(w)$ 表示词 w 的向量，为了完成这个类比，我们将找到一个词，其向量与 $\text{vec}(c) + \text{vec}(b) - \text{vec}(a)$ 的结果最相似。

```
def get_analogy(token_a, token_b, token_c, embed):  
    vecs = embed[[token_a, token_b, token_c]]  
    x = vecs[1] - vecs[0] + vecs[2]  
    topk, cos = knn(embed.idx_to_vec, x, 1)  
    return embed.idx_to_token[int(topk[0])] # 删除未知词
```

让我们使用加载的词向量来验证“male-female”类比。

```
get_analogy('man', 'woman', 'son', glove_6b50d)
```

```
'daughter'
```

下面完成一个“首都-国家”的类比：“beijing”：“china”::“tokyo”：“japan”。这说明了预训练词向量中的语义。

```
get_analogy('beijing', 'china', 'tokyo', glove_6b50d)
```

```
'japan'
```

另外，对于“bad”：“worst”::“big”：“biggest”等“形容词-形容词最高级”的比喻，预训练词向量可以捕捉到句法信息。

```
get_analogy('bad', 'worst', 'big', glove_6b50d)
```

```
'biggest'
```

为了演示在预训练词向量中捕捉到的过去式概念，我们可以使用“现在式-过去式”的类比来测试句法：“do”：“did” :: “go”：“went”。

```
get_analogy('do', 'did', 'go', glove_6b50d)
```

```
'went'
```

小结

- 在实践中，在大型语料库上预先练的词向量可以应用于下游的自然语言处理任务。
- 预训练的词向量可以应用于词的相似性和类比任务。

练习

1. 使用TokenEmbedding('wiki.en')测试fastText结果。
2. 当词表非常大时，我们怎样才能更快地找到相似的词或完成一个词的类比呢？

Discussions¹⁹⁶

14.8 来自Transformers的双向编码器表示（BERT）

我们已经介绍了几种用于自然语言理解的词嵌入模型。在预训练之后，输出可以被认为是一个矩阵，其中每一行都是一个表示预定义词表中词的向量。事实上，这些词嵌入模型都是与上下文无关的。让我们先来说明这个性质。

14.8.1 从上下文无关到上下文敏感

回想一下 14.4 节和 14.7 节中的实验。例如，word2vec 和 GloVe 都将相同的预训练向量分配给同一个词，而不考虑词的上下文（如果有的话）。形式上，任何词元 x 的上下文无关表示是函数 $f(x)$ ，其仅将 x 作为其输入。考虑到自然语言中丰富的多义现象和复杂的语义，上下文无关表示具有明显的局限性。例如，在“a crane is flying”（一只鹤在飞）和“a crane driver came”（一名吊车司机来了）的上下文中，“crane”一词有完全不同的含义；因此，同一个词可以根据上下文被赋予不同的表示。

这推动了“上下文敏感”词表示的发展，其中词的表征取决于它们的上下文。因此，词元 x 的上下文敏感表示是函数 $f(x, c(x))$ ，其取决于 x 及其上下文 $c(x)$ 。流行的上下文敏感表示包括 TagLM (language-model-augmented sequence tagger, 语言模型增强的序列标记器) (Peters et al., 2017)、CoVe (Context Vectors, 上下文向量) (McCann et al., 2017) 和 ELMo (Embeddings from Language Models, 来自语言模型的嵌入) (Peters et al., 2018)。

¹⁹⁶ <https://discuss.d2l.ai/t/5746>

例如，通过将整个序列作为输入，ELMo是为输入序列中的每个单词分配一个表示的函数。具体来说，ELMo将来自预训练的双向长短期记忆网络的所有中间层表示组合为输出表示。然后，ELMo的表示将作为附加特征添加到下游任务的现有监督模型中，例如通过将ELMo的表示和现有模型中词元的原始表示（例如GloVe）连接起来。一方面，在加入ELMo表示后，冻结了预训练的双向LSTM模型中的所有权重。另一方面，现有的监督模型是专门为给定的任务定制的。利用当时不同任务的不同最佳模型，添加ELMo改进了六种自然语言处理任务的技术水平：情感分析、自然语言推断、语义角色标注、共指消解、命名实体识别和问答。

14.8.2 从特定于任务到不可知任务

尽管ELMo显著改进了各种自然语言处理任务的解决方案，但每个解决方案仍然依赖于一个特定于任务的架构。然而，为每一个自然语言处理任务设计一个特定的架构实际上并不是一件容易的事。GPT（Generative Pre Training，生成式预训练）模型为上下文的敏感表示设计了通用的任务无关模型（Radford et al., 2018）。GPT建立在Transformer解码器的基础上，预训练了一个用于表示文本序列的语言模型。当将GPT应用于下游任务时，语言模型的输出将被送到一个附加的线性输出层，以预测任务的标签。与ELMo冻结预训练模型的参数不同，GPT在下游任务的监督学习过程中对预训练Transformer解码器中的所有参数进行微调。GPT在自然语言推断、问答、句子相似性和分类等12项任务上进行了评估，并在对模型架构进行最小更改的情况下改善了其中9项任务的最新水平。

然而，由于语言模型的自回归特性，GPT只能向前看（从左到右）。在“i went to the bank to deposit cash”（我去银行存现金）和“i went to the bank to sit down”（我去河岸边坐下）的上下文中，由于“bank”对其左边的上下文敏感，GPT将返回“bank”的相同表示，尽管它有不同的含义。

14.8.3 BERT：把两个最好的结合起来

如我们所见，ELMo对上下文进行双向编码，但使用特定于任务的架构；而GPT是任务无关的，但是从左到右编码上下文。BERT（来自Transformers的双向编码器表示）结合了这两个方面的优点。它对上下文进行双向编码，并且对于大多数的自然语言处理任务（Devlin et al., 2018）只需要最少的架构改变。通过使用预训练的Transformer编码器，BERT能够基于其双向上下文表示任何词元。在下游任务的监督学习过程中，BERT在两个方面与GPT相似。首先，BERT表示将被输入到一个添加的输出层中，根据任务的性质对模型架构进行最小的更改，例如预测每个词元与预测整个序列。其次，对预训练Transformer编码器的所有参数进行微调，而额外的输出层将从头开始训练。图14.8.1描述了ELMo、GPT和BERT之间的差异。

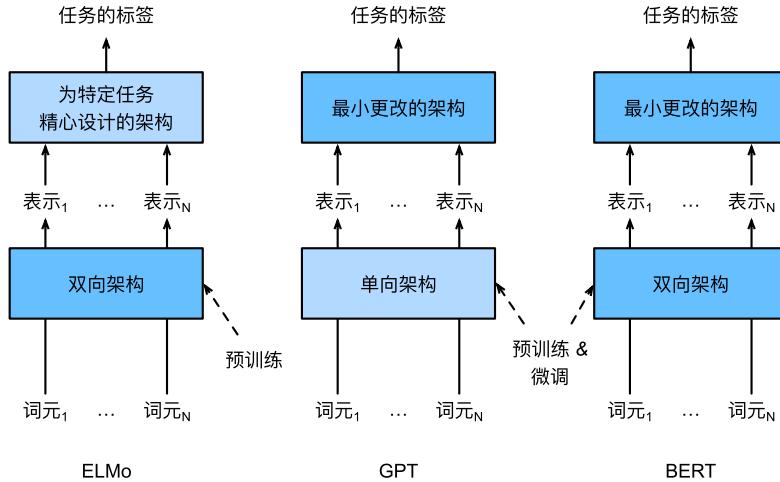


图14.8.1: ELMo、GPT和BERT的比较

BERT进一步改进了11种自然语言处理任务的技术水平，这些任务分为以下几个大类：(1) 单一文本分类（如情感分析）、(2) 文本对分类（如自然语言推断）、(3) 问答、(4) 文本标记（如命名实体识别）。从上下文敏感的ELMo到任务不可知的GPT和BERT，它们都是在2018年提出的。概念上简单但经验上强大的自然语言深度表示预训练已经彻底改变了各种自然语言处理任务的解决方案。

在本章的其余部分，我们将深入了解BERT的训练前准备。当在15节中解释自然语言处理应用时，我们将说明针对下游应用的BERT微调。

```
import torch
from torch import nn
from d2l import torch as d2l
```

14.8.4 输入表示

在自然语言处理中，有些任务（如情感分析）以单个文本作为输入，而有些任务（如自然语言推断）以一对文本序列作为输入。BERT输入序列明确地表示单个文本和文本对。当输入为单个文本时，BERT输入序列是特殊类别词元“<cls>”、文本序列的标记、以及特殊分隔词元“<sep>”的连结。当输入为文本对时，BERT输入序列是“<cls>”、第一个文本序列的标记、“<sep>”、第二个文本序列标记、以及“<sep>”的连结。我们将始终如一地将术语“BERT输入序列”与其他类型的“序列”区分开来。例如，一个BERT输入序列可以包括一个文本序列或两个文本序列。

为了区分文本对，根据输入序列学到的片段嵌入 \mathbf{e}_A 和 \mathbf{e}_B 分别被添加到第一序列和第二序列的词元嵌入中。对于单文本输入，仅使用 \mathbf{e}_A 。

下面的get_tokens_and_segments将一个句子或两个句子作为输入，然后返回BERT输入序列的标记及其相应的片段索引。

```

#@save
def get_tokens_and_segments(tokens_a, tokens_b=None):
    """获取输入序列的词元及其片段索引"""
    tokens = ['<cls>'] + tokens_a + ['<sep>']
    # 0和1分别标记片段A和B
    segments = [0] * (len(tokens_a) + 2)
    if tokens_b is not None:
        tokens += tokens_b + ['<sep>']
        segments += [1] * (len(tokens_b) + 1)
    return tokens, segments

```

BERT选择Transformer编码器作为其双向架构。在Transformer编码器中常见是，位置嵌入被加入到输入序列的每个位置。然而，与原始的Transformer编码器不同，BERT使用可学习的位置嵌入。总之，图14.8.2表明BERT输入序列的嵌入是词元嵌入、片段嵌入和位置嵌入的和。

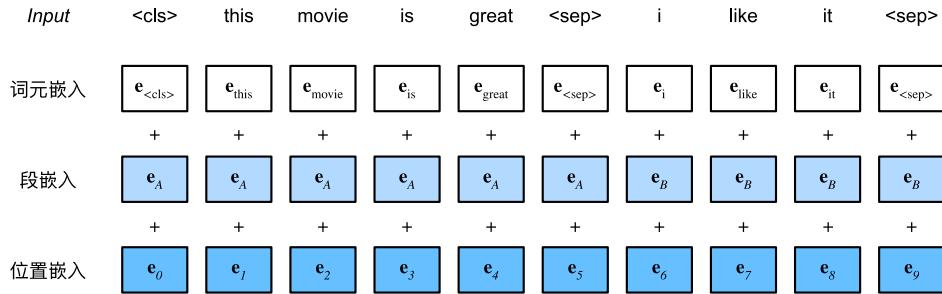


图14.8.2: BERT输入序列的嵌入是词元嵌入、片段嵌入和位置嵌入的和

下面的BERTEncoder类类似于10.7节中实现的TransformerEncoder类。与TransformerEncoder不同，BERTEncoder使用片段嵌入和可学习的位置嵌入。

```

#@save
class BERTEncoder(nn.Module):
    """BERT编码器"""
    def __init__(self, vocab_size, num_hiddens, norm_shape, ffn_num_input,
                 ffn_num_hiddens, num_heads, num_layers, dropout,
                 max_len=1000, key_size=768, query_size=768, value_size=768,
                 **kwargs):
        super(BERTEncoder, self).__init__(**kwargs)
        self.token_embedding = nn.Embedding(vocab_size, num_hiddens)
        self.segment_embedding = nn.Embedding(2, num_hiddens)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module(f"{i}", d2l.EncoderBlock(
                key_size, query_size, value_size, num_hiddens, norm_shape,

```

(continues on next page)

(continued from previous page)

```
        ffn_num_input, ffn_num_hiddens, num_heads, dropout, True))
# 在BERT中，位置嵌入是可学习的，因此我们创建一个足够长的位置嵌入参数
self.pos_embedding = nn.Parameter(torch.randn(1, max_len,
                                              num_hiddens))

def forward(self, tokens, segments, valid_lens):
    # 在以下代码段中，x的形状保持不变：(批量大小, 最大序列长度, num_hiddens)
    X = self.token_embedding(tokens) + self.segment_embedding(segments)
    X = X + self.pos_embedding.data[:, :X.shape[1], :]
    for blk in self.blks:
        X = blk(X, valid_lens)
    return X
```

假设词表大小为10000，为了演示BERTEncoder的前向推断，让我们创建一个实例并初始化它的参数。

```
vocab_size, num_hiddens, ffn_num_hiddens, num_heads = 10000, 768, 1024, 4
norm_shape, ffn_num_input, num_layers, dropout = [768], 768, 2, 0.2
encoder = BERTEncoder(vocab_size, num_hiddens, norm_shape, ffn_num_input,
                      ffn_num_hiddens, num_heads, num_layers, dropout)
```

我们将tokens定义为长度为8的2个输入序列，其中每个词元是词表的索引。使用输入tokens的BERTEncoder的前向推断返回编码结果，其中每个词元由向量表示，其长度由超参数num_hiddens定义。此超参数通常称为Transformer编码器的隐藏大小（隐藏单元数）。

```
tokens = torch.randint(0, vocab_size, (2, 8))
segments = torch.tensor([[0, 0, 0, 0, 1, 1, 1, 1], [0, 0, 0, 1, 1, 1, 1, 1]])
encoded_X = encoder(tokens, segments, None)
encoded_X.shape
```

```
torch.Size([2, 8, 768])
```

14.8.5 预训练任务

BERTEncoder的前向推断给出了输入文本的每个词元和插入的特殊标记“<cls>”及“<seq>”的BERT表示。接下来，我们将使用这些表示来计算预训练BERT的损失函数。预训练包括以下两个任务：掩蔽语言模型和下一句预测。

掩蔽语言模型（Masked Language Modeling）

如 8.3 节所示，语言模型使用左侧的上下文预测词元。为了双向编码上下文以表示每个词元，BERT随机掩蔽词元并使用来自双向上下文的词元以自监督的方式预测掩蔽词元。此任务称为掩蔽语言模型。

在这个预训练任务中，将随机选择15%的词元作为预测的掩蔽词元。要预测一个掩蔽词元而不使用标签作弊，一个简单的方法是总是用一个特殊的“<mask>”替换输入序列中的词元。然而，人造特殊词元“<mask>”不会出现在微调中。为了避免预训练和微调之间的这种不匹配，如果为预测而屏蔽词元（例如，在“this movie is great”中选择掩蔽和预测“great”），则在输入中将其替换为：

- 80%时间为特殊的“<mask>”词元（例如，“this movie is great”变为“this movie is<mask>”）；
- 10%时间为随机词元（例如，“this movie is great”变为“this movie is drink”）；
- 10%时间内为不变的标签词元（例如，“this movie is great”变为“this movie is great”）。

请注意，在15%的时间中，有10%的时间插入了随机词元。这种偶然的噪声鼓励BERT在其双向上下文编码中不那么偏向于掩蔽词元（尤其是当标签词元保持不变时）。

我们实现了下面的MaskLM类来预测BERT预训练的掩蔽语言模型任务中的掩蔽标记。预测使用单隐藏层的多层感知机（self.mlp）。在前向推断中，它需要两个输入：BERTEncoder的编码结果和用于预测的词元位置。输出是这些位置的预测结果。

```
#@save
class MaskLM(nn.Module):
    """BERT的掩蔽语言模型任务"""
    def __init__(self, vocab_size, num_hiddens, num_inputs=768, **kwargs):
        super(MaskLM, self).__init__(**kwargs)
        self.mlp = nn.Sequential(nn.Linear(num_inputs, num_hiddens),
                               nn.ReLU(),
                               nn.LayerNorm(num_hiddens),
                               nn.Linear(num_hiddens, vocab_size))

    def forward(self, X, pred_positions):
        num_pred_positions = pred_positions.shape[1]
        pred_positions = pred_positions.reshape(-1)
        batch_size = X.shape[0]
        batch_idx = torch.arange(0, batch_size)
        # 假设batch_size=2, num_pred_positions=3
        # 那么batch_idx是np.array ([0,0,0,1,1,1])
```

(continues on next page)

(continued from previous page)

```
batch_idx = torch.repeat_interleave(batch_idx, num_pred_positions)
masked_X = X[batch_idx, pred_positions]
masked_X = masked_X.reshape((batch_size, num_pred_positions, -1))
mlm_Y_hat = self.mlp(masked_X)
return mlm_Y_hat
```

为了演示MaskLM的前向推断，我们创建了其实例mlm并对其进行了初始化。回想一下，来自BERTEncoder的正向推断encoded_X表示2个BERT输入序列。我们将mlm_positions定义为在encoded_X的任一输入序列中预测的3个指示。mlm的前向推断返回encoded_X的所有掩蔽位置mlm_positions处的预测结果mlm_Y_hat。对于每个预测，结果的大小等于词表的大小。

```
mlm = MaskLM(vocab_size, num_hiddens)
mlm_positions = torch.tensor([[1, 5, 2], [6, 1, 5]])
mlm_Y_hat = mlm(encoded_X, mlm_positions)
mlm_Y_hat.shape
```

```
torch.Size([2, 3, 10000])
```

通过掩码下的预测词元mlm_Y的真实标签mlm_Y_hat，我们可以计算在BERT预训练中的遮蔽语言模型任务的交叉熵损失。

```
mlm_Y = torch.tensor([[7, 8, 9], [10, 20, 30]])
loss = nn.CrossEntropyLoss(reduction='none')
mlm_l = loss(mlm_Y_hat.reshape((-1, vocab_size)), mlm_Y.reshape(-1))
mlm_l.shape
```

```
torch.Size([6])
```

下一句预测（Next Sentence Prediction）

尽管掩蔽语言建模能够编码双向上下文来表示单词，但它不能显式地建模文本对之间的逻辑关系。为了帮助理解两个文本序列之间的关系，BERT在预训练中考虑了一个二元分类任务——下一句预测。在为预训练生成句子对时，有一半的时间它们确实是标签为“真”的连续句子；在另一半的时间里，第二个句子是从语料库中随机抽取的，标记为“假”。

下面的NextSentencePred类使用单隐藏层的多层感知机来预测第二个句子是否是BERT输入序列中第一个句子的下一个句子。由于Transformer编码器中的自注意力，特殊词元“<cls>”的BERT表示已经对输入的两个句子进行了编码。因此，多层感知机分类器的输出层（self.output）以x作为输入，其中x是多层感知机隐藏层的输出，而MLP隐藏层的输入是编码后的“<cls>”词元。

```
#@save
class NextSentencePred(nn.Module):
    """BERT的下一句预测任务"""
    def __init__(self, num_inputs, **kwargs):
        super(NextSentencePred, self).__init__(**kwargs)
        self.output = nn.Linear(num_inputs, 2)

    def forward(self, X):
        # X的形状: (batchsize, num_hiddens)
        return self.output(X)
```

我们可以看到，`NextSentencePred`实例的前向推断返回每个BERT输入序列的二分类预测。

```
encoded_X = torch.flatten(encoded_X, start_dim=1)
# NSP的输入形状:(batchsize, num_hiddens)
nsp = NextSentencePred(encoded_X.shape[-1])
nsp_Y_hat = nsp(encoded_X)
nsp_Y_hat.shape
```

```
torch.Size([2, 2])
```

还可以计算两个二元分类的交叉熵损失。

```
nsp_y = torch.tensor([0, 1])
nsp_l = loss(nsp_Y_hat, nsp_y)
nsp_l.shape
```

```
torch.Size([2])
```

值得注意的是，上述两个预训练任务中的所有标签都可以从预训练语料库中获得，而无需人工标注。原始的BERT已经在图书语料库 (Zhu et al., 2015)和英文维基百科的连接上进行了预训练。这两个文本语料库非常庞大：它们分别有8亿个单词和25亿个单词。

14.8.6 整合代码

在预训练BERT时，最终的损失函数是掩蔽语言模型损失函数和下一句预测损失函数的线性组合。现在我们可以通过实例化三个类BERTEncoder、MaskLM和NextSentencePred来定义BERTModel类。前向推断返回编码后的BERT表示`encoded_X`、掩蔽语言模型预测`mlm_Y_hat`和下一句预测`nsp_Y_hat`。

```

#@save
class BERTModel(nn.Module):
    """BERT模型"""
    def __init__(self, vocab_size, num_hiddens, norm_shape, ffn_num_input,
                 ffn_num_hiddens, num_heads, num_layers, dropout,
                 max_len=1000, key_size=768, query_size=768, value_size=768,
                 hid_in_features=768, mlm_in_features=768,
                 nsp_in_features=768):
        super(BERTModel, self).__init__()
        self.encoder = BERTEncoder(vocab_size, num_hiddens, norm_shape,
                                   ffn_num_input, ffn_num_hiddens, num_heads, num_layers,
                                   dropout, max_len=max_len, key_size=key_size,
                                   query_size=query_size, value_size=value_size)
        self.hidden = nn.Sequential(nn.Linear(hid_in_features, num_hiddens),
                                   nn.Tanh())
        self.mlm = MaskLM(vocab_size, num_hiddens, mlm_in_features)
        self.nsp = NextSentencePred(nsp_in_features)

    def forward(self, tokens, segments, valid_lens=None,
               pred_positions=None):
        encoded_X = self.encoder(tokens, segments, valid_lens)
        if pred_positions is not None:
            mlm_Y_hat = self.mlm(encoded_X, pred_positions)
        else:
            mlm_Y_hat = None
        # 用于下一句预测的多层感知机分类器的隐藏层，0是“<cls>”标记的索引
        nsp_Y_hat = self.nsp(self.hidden(encoded_X[:, 0, :]))
        return encoded_X, mlm_Y_hat, nsp_Y_hat

```

小结

- word2vec和GloVe等词嵌入模型与上下文无关。它们将相同的预训练向量赋给同一个词，而不考虑词的上下文（如果有的话）。它们很难处理好自然语言中的一词多义或复杂语义。
- 对于上下文敏感的词表示，如ELMo和GPT，词的表示依赖于它们的上下文。
- ELMo对上下文进行双向编码，但使用特定于任务的架构（然而，为每个自然语言处理任务设计一个特定的体系架构实际上不容易）；而GPT是任务无关的，但是从左到右编码上下文。
- BERT结合了这两个方面的优点：它对上下文进行双向编码，并且需要对大量自然语言处理任务进行最小的架构更改。
- BERT输入序列的嵌入是词元嵌入、片段嵌入和位置嵌入的和。
- 预训练包括两个任务：掩蔽语言模型和下一句预测。前者能够编码双向上下文来表示单词，而后者则显

式地建模文本对之间的逻辑关系。

练习

1. 为什么BERT成功了？
2. 在所有其他条件相同的情况下，掩蔽语言模型比从左到右的语言模型需要更多或更少的预训练步骤来收敛吗？为什么？
3. 在BERT的原始实现中，BERTEncoder中的位置前馈网络（通过d2l.EncoderBlock）和MaskLM中的全连接层都使用高斯误差线性单元（Gaussian error linear unit, GELU）(Hendrycks and Gimpel, 2016)作为激活函数。研究GELU与ReLU之间的差异。

Discussions¹⁹⁷

14.9 用于预训练BERT的数据集

为了预训练 14.8 节中实现的BERT模型，我们需要以理想的格式生成数据集，以便于两个预训练任务：遮蔽语言模型和下一句预测。一方面，最初的BERT模型是在两个庞大的图书语料库和英语维基百科（参见 14.8.5节）的合集上预训练的，但它很难吸引这本书的大多数读者。另一方面，现成的预训练BERT模型可能不适合医学等特定领域的应用。因此，在定制的数据集上对BERT进行预训练变得越来越流行。为了方便BERT预训练的演示，我们使用了较小的语料库WikiText-2 (Merity *et al.*, 2016)。

与 14.3 节中用于预训练word2vec的PTB数据集相比，WikiText-2 (1) 保留了原来的标点符号，适合于下一句预测；(2) 保留了原来的大小写和数字；(3) 大了一倍以上。

```
import os
import random
import torch
from d2l import torch as d2l
```

在WikiText-2数据集中，每行代表一个段落，其中在任意标点符号及其前面的词元之间插入空格。保留至少有两句话的段落。为了简单起见，我们仅使用句号作为分隔符来拆分句子。我们将更复杂的句子拆分技术的讨论留在本节末尾的练习中。

```
#@save
d2l.DATA_HUB['wikitext-2'] = (
    'https://s3.amazonaws.com/research.metamind.io/wikitext/'
    'wikitext-2-v1.zip', '3c914d17d80b1459be871a5039ac23e752a53cbe')

#@save
def _read_wiki(data_dir):
```

(continues on next page)

¹⁹⁷ <https://discuss.d2l.ai/t/5750>

(continued from previous page)

```
file_name = os.path.join(data_dir, 'wiki.train.tokens')
with open(file_name, 'r') as f:
    lines = f.readlines()
# 大写字母转换为小写字母
paragraphs = [line.strip().lower().split(' . ')
              for line in lines if len(line.split(' . ')) >= 2]
random.shuffle(paragraphs)
return paragraphs
```

14.9.1 为预训练任务定义辅助函数

在下文中，我们首先为BERT的两个预训练任务实现辅助函数。这些辅助函数将在稍后将原始文本语料库转换为理想格式的数据集时调用，以预训练BERT。

生成下一句预测任务的数据

根据 14.8.5 节的描述，`_get_next_sentence` 函数生成二分类任务的训练样本。

```
#@save
def _get_next_sentence(sentence, next_sentence, paragraphs):
    if random.random() < 0.5:
        is_next = True
    else:
        # paragraphs是三重列表的嵌套
        next_sentence = random.choice(random.choice(paragraphs))
        is_next = False
    return sentence, next_sentence, is_next
```

下面的函数通过调用`_get_next_sentence` 函数从输入`paragraph`生成用于下一句预测的训练样本。这里`paragraph`是句子列表，其中每个句子都是词元列表。自变量`max_len`指定预训练期间的BERT输入序列的最大长度。

```
#@save
def _get_nsp_data_from_paragraph(paragraph, paragraphs, vocab, max_len):
    nsp_data_from_paragraph = []
    for i in range(len(paragraph) - 1):
        tokens_a, tokens_b, is_next = _get_next_sentence(
            paragraph[i], paragraph[i + 1], paragraphs)
        # 考虑1个'<cls>'词元和2个'<sep>'词元
        if len(tokens_a) + len(tokens_b) + 3 > max_len:
            continue
        nsp_data_from_paragraph.append((tokens_a, tokens_b, is_next))
    return nsp_data_from_paragraph
```

(continues on next page)

(continued from previous page)

```
tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
nsp_data_from_paragraph.append((tokens, segments, is_next))
return nsp_data_from_paragraph
```

生成遮蔽语言模型任务的数据

为了从BERT输入序列生成遮蔽语言模型的训练样本，我们定义了以下_replace_mlm_tokens函数。在其输入中，tokens是表示BERT输入序列的词元的列表，candidate_pred_positions是不包括特殊词元的BERT输入序列的词元素索引的列表（特殊词元在遮蔽语言模型任务中不被预测），以及num_mlm_preds指示预测的数量（选择15%要预测的随机词元）。在14.8.5节中定义遮蔽语言模型任务之后，在每个预测位置，输入可以由特殊的“掩码”词元或随机词元替换，或者保持不变。最后，该函数返回可能替换后的输入词元、发生预测的词元素索引和这些预测的标签。

```
#@save
def _replace_mlm_tokens(tokens, candidate_pred_positions, num_mlm_preds,
                        vocab):
    # 为遮蔽语言模型的输入创建新的词元副本，其中输入可能包含替换的“<mask>”或随机词元
    mlm_input_tokens = [token for token in tokens]
    pred_positions_and_labels = []
    # 打乱后用于在遮蔽语言模型任务中获取15%的随机词元进行预测
    random.shuffle(candidate_pred_positions)
    for mlm_pred_position in candidate_pred_positions:
        if len(pred_positions_and_labels) >= num_mlm_preds:
            break
        masked_token = None
        # 80%的时间：将词替换为“<mask>”词元
        if random.random() < 0.8:
            masked_token = '<mask>'
        else:
            # 10%的时间：保持词不变
            if random.random() < 0.5:
                masked_token = tokens[mlm_pred_position]
            # 10%的时间：用随机词替换该词
            else:
                masked_token = random.choice(vocab.idx_to_token)
        mlm_input_tokens[mlm_pred_position] = masked_token
        pred_positions_and_labels.append(
            (mlm_pred_position, tokens[mlm_pred_position]))
    return mlm_input_tokens, pred_positions_and_labels
```

通过调用前述的_replace_mlm_tokens函数，以下函数将BERT输入序列（tokens）作为输入，并返回输入词元的索引（在14.8.5节中描述的可能的词元替换之后）、发生预测的词元素索引以及这些预测的标签索引。

```

#@save
def _get_mlm_data_from_tokens(tokens, vocab):
    candidate_pred_positions = []
    # tokens是一个字符串列表
    for i, token in enumerate(tokens):
        # 在遮蔽语言模型任务中不会预测特殊词元
        if token in ['<cls>', '<sep>']:
            continue
        candidate_pred_positions.append(i)
    # 遮蔽语言模型任务中预测15%的随机词元
    num_mlm_preds = max(1, round(len(tokens) * 0.15))
    mlm_input_tokens, pred_positions_and_labels = _replace_mlm_tokens(
        tokens, candidate_pred_positions, num_mlm_preds, vocab)
    pred_positions_and_labels = sorted(pred_positions_and_labels,
                                       key=lambda x: x[0])
    pred_positions = [v[0] for v in pred_positions_and_labels]
    mlm_pred_labels = [v[1] for v in pred_positions_and_labels]
    return vocab[mlm_input_tokens], pred_positions, vocab[mlm_pred_labels]

```

14.9.2 将文本转换为预训练数据集

现在我们几乎准备好为BERT预训练定制一个Dataset类。在此之前，我们仍然需要定义辅助函数`_pad_bert_inputs`来将特殊的“`<mask>`”词元附加到输入。它的参数`examples`包含来自两个预训练任务的辅助函数`_get_nsp_data_from_paragraph`和`_get_mlm_data_from_tokens`的输出。

```

#@save
def _pad_bert_inputs(examples, max_len, vocab):
    max_num_mlm_preds = round(max_len * 0.15)
    all_token_ids, all_segments, valid_lens, = [], [], []
    all_pred_positions, all_mlm_weights, all_mlm_labels = [], [], []
    nsp_labels = []
    for (token_ids, pred_positions, mlm_pred_label_ids, segments,
          is_next) in examples:
        all_token_ids.append(torch.tensor(token_ids + [vocab['<pad>']] * (
            max_len - len(token_ids)), dtype=torch.long))
        all_segments.append(torch.tensor(segments + [0] * (
            max_len - len(segments)), dtype=torch.long))
    # valid_lens不包括'<pad>'的计数
    valid_lens.append(torch.tensor(len(token_ids), dtype=torch.float32))
    all_pred_positions.append(torch.tensor(pred_positions + [0] * (
        max_num_mlm_preds - len(pred_positions)), dtype=torch.long))
    # 填充词元的预测将通过乘以0权重在损失中过滤掉

```

(continues on next page)

(continued from previous page)

```
all_mlm_weights.append(  
    torch.tensor([1.0] * len(mlm_pred_label_ids) + [0.0] * (  
        max_num_mlm_preds - len(pred_positions)),  
        dtype=torch.float32))  
all_mlm_labels.append(torch.tensor(mlm_pred_label_ids + [0] * (  
    max_num_mlm_preds - len(mlm_pred_label_ids)), dtype=torch.long))  
nsp_labels.append(torch.tensor(is_next, dtype=torch.long))  
return (all_token_ids, all_segments, valid_lens, all_pred_positions,  
    all_mlm_weights, all_mlm_labels, nsp_labels)
```

将用于生成两个预训练任务的训练样本的辅助函数和用于填充输入的辅助函数放在一起，我们定义以下_WikiTextDataset类为用于预训练BERT的WikiText-2数据集。通过实现__getitem__函数，我们可以任意访问WikiText-2语料库的一对句子生成的预训练样本（遮蔽语言模型和下一句预测）样本。

最初的BERT模型使用词表大小为30000的WordPiece嵌入 (Wu et al., 2016)。WordPiece的词元化方法是对14.6.2节中原有的字节对编码算法稍作修改。为简单起见，我们使用d2l.tokenize函数进行词元化。出现次数少于5次的不频繁词元将被过滤掉。

```
#@save  
class _WikiTextDataset(torch.utils.data.Dataset):  
    def __init__(self, paragraphs, max_len):  
        # 输入paragraphs[i]是代表段落的句子字符串列表；  
        # 而输出paragraphs[i]是代表段落的句子列表，其中每个句子都是词元列表  
        paragraphs = [d2l.tokenize(  
            paragraph, token='word') for paragraph in paragraphs]  
        sentences = [sentence for paragraph in paragraphs  
            for sentence in paragraph]  
        self.vocab = d2l.Vocab(sentences, min_freq=5, reserved_tokens=[  
            '<pad>', '<mask>', '<cls>', '<sep>'])  
        # 获取下一句子预测任务的数据  
        examples = []  
        for paragraph in paragraphs:  
            examples.extend(_get_nsp_data_from_paragraph(  
                paragraph, paragraphs, self.vocab, max_len))  
        # 获取遮蔽语言模型任务的数据  
        examples = [(_get_mlm_data_from_tokens(tokens, self.vocab)  
            + (segments, is_next))  
            for tokens, segments, is_next in examples]  
        # 填充输入  
        (self.all_token_ids, self.all_segments, self.valid_lens,  
        self.all_pred_positions, self.all_mlm_weights,  
        self.all_mlm_labels, self.nsp_labels) = _pad_bert_inputs(  
            examples, max_len, self.vocab)
```

(continues on next page)

```

def __getitem__(self, idx):
    return (self.all_token_ids[idx], self.all_segments[idx],
            self.valid_lens[idx], self.all_pred_positions[idx],
            self.all_mlm_weights[idx], self.all_mlm_labels[idx],
            self.nsp_labels[idx])

def __len__(self):
    return len(self.all_token_ids)

```

通过使用`_read_wiki`函数和`_WikiTextDataset`类，我们定义了下面的`load_data_wiki`来下载并生成WikiText-2数据集，并从中生成预训练样本。

```

#@save
def load_data_wiki(batch_size, max_len):
    """加载WikiText-2数据集"""
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('wikitext-2', 'wikitext-2')
    paragraphs = _read_wiki(data_dir)
    train_set = _WikiTextDataset(paragraphs, max_len)
    train_iter = torch.utils.data.DataLoader(train_set, batch_size,
                                             shuffle=True, num_workers=num_workers)
    return train_iter, train_set.vocab

```

将批量大小设置为512，将BERT输入序列的最大长度设置为64，我们打印出小批量的BERT预训练样本的形状。注意，在每个BERT输入序列中，为遮蔽语言模型任务预测10 (64×0.15) 个位置。

```

batch_size, max_len = 512, 64
train_iter, vocab = load_data_wiki(batch_size, max_len)

for (tokens_X, segments_X, valid_lens_x, pred_positions_X, mlm_weights_X,
      mlm_Y, nsp_y) in train_iter:
    print(tokens_X.shape, segments_X.shape, valid_lens_x.shape,
          pred_positions_X.shape, mlm_weights_X.shape, mlm_Y.shape,
          nsp_y.shape)
    break

```

```

Downloading ../data/wikitext-2-v1.zip from https://s3.amazonaws.com/research.metamind.io/wikitext/
→wikitext-2-v1.zip...
torch.Size([512, 64]) torch.Size([512, 64]) torch.Size([512]) torch.Size([512, 10]) torch.Size([512, ↪
→10]) torch.Size([512, 10]) torch.Size([512])

```

最后，我们来看一下词量。即使在过滤掉不频繁的词元之后，它仍然比PTB数据集的大两倍以上。

```
len(vocab)
```

```
20256
```

小结

- 与PTB数据集相比，WikiText-2数据集保留了原来的标点符号、大小写和数字，并且比PTB数据集大了两倍多。
- 我们可以任意访问从WikiText-2语料库中的一对句子生成的预训练（遮蔽语言模型和下一句预测）样本。

练习

- 为简单起见，句号用作拆分句子的唯一分隔符。尝试其他的句子拆分技术，比如Spacy和NLTK。以NLTK为例，需要先安装NLTK：`pip install nltk`。在代码中先`import nltk`。然后下载Punkt语句词元分析器：`nltk.download('punkt')`。要拆分句子，比如`sentences = 'This is great ! Why not ?'`，调用`nltk.tokenize.sent_tokenize(sentences)`将返回两个句子字符串的列表：`['This is great !', 'Why not ?']`。
- 如果我们不过滤出一些不常见的词元，词量会有多大？

Discussions¹⁹⁸

14.10 预训练BERT

利用 14.8 节中实现的BERT模型和 14.9 节中从WikiText-2数据集生成的预训练样本，我们将在本节中在WikiText-2数据集上对BERT进行预训练。

```
import torch
from torch import nn
from d2l import torch as d2l
```

首先，我们加载WikiText-2数据集作为小批量的预训练样本，用于遮蔽语言模型和下一句预测。批量大小是512，BERT输入序列的最大长度是64。注意，在原始BERT模型中，最大长度是512。

```
batch_size, max_len = 512, 64
train_iter, vocab = d2l.load_data_wiki(batch_size, max_len)
```

¹⁹⁸ <https://discuss.d2l.ai/t/5738>

14.10.1 预训练BERT

原始BERT (Devlin *et al.*, 2018)有两个不同模型尺寸的版本。基本模型 (BERT_{BASE}) 使用12层 (Transformer编码器块)，768个隐藏单元 (隐藏大小) 和12个自注意头。大模型 (BERT_{LARGE}) 使用24层，1024个隐藏单元和16个自注意头。值得注意的是，前者有1.1亿个参数，后者有3.4亿个参数。为了便于演示，我们定义了一个小的BERT，使用了2层、128个隐藏单元和2个自注意头。

```
net = d2l.BERTModel(len(vocab), num_hiddens=128, norm_shape=[128],
                     ffn_num_input=128, ffn_num_hiddens=256, num_heads=2,
                     num_layers=2, dropout=0.2, key_size=128, query_size=128,
                     value_size=128, hid_in_features=128, mlm_in_features=128,
                     nsp_in_features=128)
devices = d2l.try_all_gpus()
loss = nn.CrossEntropyLoss()
```

在定义训练代码实现之前，我们定义了一个辅助函数`_get_batch_loss_bert`。给定训练样本，该函数计算遮蔽语言模型和下一句子预测任务的损失。请注意，BERT预训练的最终损失是遮蔽语言模型损失和下一句预测损失的和。

```
#@save
def _get_batch_loss_bert(net, loss, vocab_size, tokens_X,
                        segments_X, valid_lens_x,
                        pred_positions_X, mlm_weights_X,
                        mlm_Y, nsp_y):
    # 前向传播
    _, mlm_Y_hat, nsp_Y_hat = net(tokens_X, segments_X,
                                    valid_lens_x.reshape(-1),
                                    pred_positions_X)

    # 计算遮蔽语言模型损失
    mlm_l = loss(mlm_Y_hat.reshape(-1, vocab_size), mlm_Y.reshape(-1)) *\
        mlm_weights_X.reshape(-1, 1)
    mlm_l = mlm_l.sum() / (mlm_weights_X.sum() + 1e-8)

    # 计算下一句子预测任务的损失
    nsp_l = loss(nsp_Y_hat, nsp_y)
    l = mlm_l + nsp_l
    return mlm_l, nsp_l, l
```

通过调用上述两个辅助函数，下面的`train_bert`函数定义了在WikiText-2 (`train_iter`) 数据集上预训练BERT (`net`) 的过程。训练BERT可能需要很长时间。以下函数的输入`num_steps`指定了训练的迭代步数，而不是像`train_ch13`函数那样指定训练的轮数（参见 13.1 节）。

```
def train_bert(train_iter, net, loss, vocab_size, devices, num_steps):
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
```

(continues on next page)

(continued from previous page)

```
trainer = torch.optim.Adam(net.parameters(), lr=0.01)
step, timer = 0, d2l.Timer()
animator = d2l.Animator(xlabel='step', ylabel='loss',
                        xlim=[1, num_steps], legend=['mlm', 'nsp'])
# 遮蔽语言模型损失的和，下一句预测任务损失的和，句子对的数量，计数
metric = d2l.Accumulator(4)
num_steps_reached = False
while step < num_steps and not num_steps_reached:
    for tokens_X, segments_X, valid_lens_x, pred_positions_X,
        mlm_weights_X, mlm_Y, nsp_y in train_iter:
        tokens_X = tokens_X.to(devices[0])
        segments_X = segments_X.to(devices[0])
        valid_lens_x = valid_lens_x.to(devices[0])
        pred_positions_X = pred_positions_X.to(devices[0])
        mlm_weights_X = mlm_weights_X.to(devices[0])
        mlm_Y, nsp_y = mlm_Y.to(devices[0]), nsp_y.to(devices[0])
        trainer.zero_grad()
        timer.start()
        mlm_l, nsp_l, l = _get_batch_loss_bert(
            net, loss, vocab_size, tokens_X, segments_X, valid_lens_x,
            pred_positions_X, mlm_weights_X, mlm_Y, nsp_y)
        l.backward()
        trainer.step()
        metric.add(mlm_l, nsp_l, tokens_X.shape[0], 1)
        timer.stop()
        animator.add(step + 1,
                     (metric[0] / metric[3], metric[1] / metric[3]))
        step += 1
        if step == num_steps:
            num_steps_reached = True
            break

    print(f'MLM loss {metric[0] / metric[3]:.3f}, '
          f'NSP loss {metric[1] / metric[3]:.3f}')
print(f'{metric[2] / timer.sum():.1f} sentence pairs/sec on '
      f'{[str(devices)]}'
```

在预训练过程中，我们可以绘制出遮蔽语言模型损失和下一句预测损失。

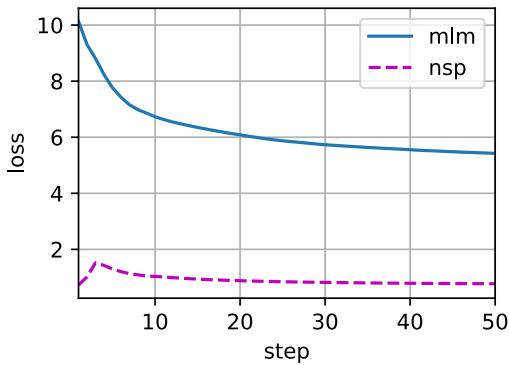
```
train_bert(train_iter, net, loss, len(vocab), devices, 50)
```

```
MLM loss 5.425, NSP loss 0.775
```

(continues on next page)

(continued from previous page)

3485.7 sentence pairs/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]



14.10.2 用BERT表示文本

在预训练BERT之后，我们可以用它来表示单个文本、文本对或其中的任何词元。下面的函数返回tokens_a和tokens_b中所有词元的BERT（net）表示。

```
def get_bert_encoding(net, tokens_a, tokens_b=None):
    tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
    token_ids = torch.tensor(vocab[tokens], device=devices[0]).unsqueeze(0)
    segments = torch.tensor(segments, device=devices[0]).unsqueeze(0)
    valid_len = torch.tensor(len(tokens), device=devices[0]).unsqueeze(0)
    encoded_X, _, _ = net(token_ids, segments, valid_len)
    return encoded_X
```

考虑“a crane is flying”这句话。回想一下 14.8.4 节中讨论的BERT的输入表示。插入特殊标记“<cls>”（用于分类）和“<sep>”（用于分隔）后，BERT输入序列的长度为6。因为零是“<cls>”词元，encoded_text[:, 0, :]是整个输入语句的BERT表示。为了评估一词多义词元“crane”，我们还打印出了该词元的BERT表示的前三个元素。

```
tokens_a = ['a', 'crane', 'is', 'flying']
encoded_text = get_bert_encoding(net, tokens_a)
# 词元: '<cls>', 'a', 'crane', 'is', 'flying', '<sep>'
encoded_text_cls = encoded_text[:, 0, :]
encoded_text_crane = encoded_text[:, 2, :]
encoded_text.shape, encoded_text_cls.shape, encoded_text_crane[0][:3]
```

```
(torch.Size([1, 6, 128]),
 torch.Size([1, 128]),
 tensor([-0.5007, -1.0034,  0.8718], device='cuda:0', grad_fn=<SliceBackward0>))
```

现在考虑一个句子“a crane driver came”和“he just left”。类似地，`encoded_pair[:, 0, :]`是来自预训练BERT的整个句子对的编码结果。注意，多义词元“crane”的前三个元素与上下文不同的元素不同。这支持了BERT表示是上下文敏感的。

```
tokens_a, tokens_b = ['a', 'crane', 'driver', 'came'], ['he', 'just', 'left']
encoded_pair = get_bert_encoding(net, tokens_a, tokens_b)
# 词元: '<cls>', 'a', 'crane', 'driver', 'came', '<sep>', 'he', 'just',
# 'left', '<sep>'
encoded_pair_cls = encoded_pair[:, 0, :]
encoded_pair_crane = encoded_pair[:, 2, :]
encoded_pair.shape, encoded_pair_cls.shape, encoded_pair_crane[0][:3]
```

```
(torch.Size([1, 10, 128]),
 torch.Size([1, 128]),
 tensor([ 0.5101, -0.4041, -1.2749], device='cuda:0', grad_fn=<SliceBackward0>))
```

在 15 节中，我们将为下游自然语言处理应用微调预训练的BERT模型。

小结

- 原始的BERT有两个版本，其中基本模型有1.1亿个参数，大模型有3.4亿个参数。
- 在预训练BERT之后，我们可以用它来表示单个文本、文本对或其中的任何词元。
- 在实验中，同一个词元在不同的上下文中具有不同的BERT表示。这支持BERT表示是上下文敏感的。

练习

1. 在实验中，我们可以看到遮蔽语言模型损失明显高于下一句预测损失。为什么？
2. 将BERT输入序列的最大长度设置为512（与原始BERT模型相同）。使用原始BERT模型的配置，如BERT_{LARGE}。运行此部分时是否遇到错误？为什么？

Discussions¹⁹⁹

¹⁹⁹ <https://discuss.d2l.ai/t/5743>

自然语言处理：应用

前面我们学习了如何在文本序列中表示词元，并在 14 节中训练了词元的表示。这样的预训练文本表示可以通过不同模型架构，放入不同的下游自然语言处理任务。

前一章我们提及到一些自然语言处理应用，这些应用没有预训练，只是为了解释深度学习架构。例如，在 8 节中，我们依赖循环神经网络设计语言模型来生成类似中篇小说的文本。在 9 节和 10 节中，我们还设计了基于循环神经网络和注意力机制的机器翻译模型。

然而，本书并不打算全面涵盖所有此类应用。相反，我们的重点是如何应用深度语言表征学习来解决自然语言处理问题。在给定预训练的文本表示的情况下，本章将探讨两种流行且具有代表性的下游自然语言处理任务：情感分析和自然语言推断，它们分别分析单个文本和文本对之间的关系。

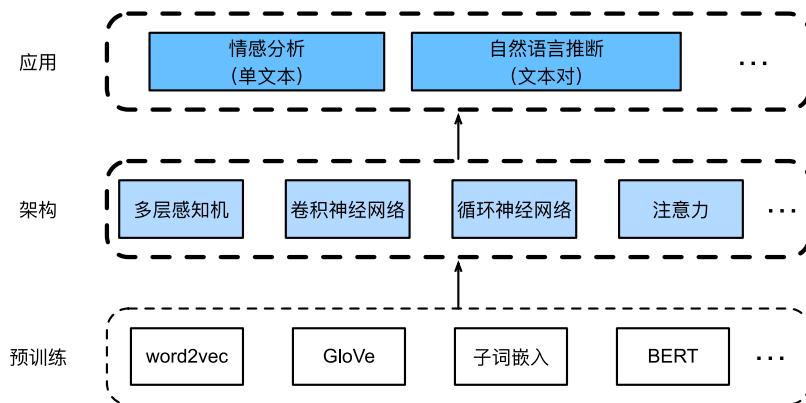


图15.1: 预训练文本表示可以通过不同模型架构，放入不同的下游自然语言处理应用（本章重点介绍如何为不同的下游应用设计模型）

如图15.1所述，本章将重点描述然后使用不同类型的深度学习架构（如多层感知机、卷积神经网络、循环神经网络和注意力）设计自然语言处理模型。尽管在图15.1中，可以将任何预训练的文本表示与任何应用的架构相结合，但我们选择了一些具有代表性的组合。具体来说，我们将探索基于循环神经网络和卷积神经网络的流行架构进行情感分析。对于自然语言推断，我们选择注意力和多层感知机来演示如何分析文本对。最后，我们介绍了如何为广泛的自然语言处理应用，如在序列级（单文本分类和文本对分类）和词元级（文本标注和问答）上对预训练BERT模型进行微调。作为一个具体的经验案例，我们将针对自然语言推断对BERT进行微调。

正如我们在14.8节中介绍的那样，对于广泛的自然语言处理应用，BERT只需要最少的架构更改。然而，这一好处是以微调下游应用的大量BERT参数为代价的。当空间或时间有限时，基于多层感知机、卷积神经网络、循环神经网络和注意力的精心构建的模型更具可行性。下面，我们从情感分析应用开始，分别解读基于循环神经网络和卷积神经网络的模型设计。

15.1 情感分析及数据集

随着在线社交媒体和评论平台的快速发展，大量评论的数据被记录下来。这些数据具有支持决策过程的巨大潜力。情感分析（sentiment analysis）研究人们在文本中（如产品评论、博客评论和论坛讨论等）“隐藏”的情绪。它在广泛应用于政治（如公众对政策的情绪分析）、金融（如市场情绪分析）和营销（如产品研究和品牌管理）等领域。

由于情感可以被分类为离散的极性或尺度（例如，积极的和消极的），我们可以将情感分析看作一项文本分类任务，它将可变长度的文本序列转换为固定长度的文本类别。在本章中，我们将使用斯坦福大学的大型电影评论数据集（large movie review dataset）²⁰⁰进行情感分析。它由一个训练集和一个测试集组成，其中包含从IMDb下载的25000个电影评论。在这两个数据集中，“积极”和“消极”标签的数量相同，表示不同的情感极性。

```
import os
import torch
from torch import nn
from d2l import torch as d2l
```

15.1.1 读取数据集

首先，下载并提取路径`../data/aclImdb`中的IMDb评论数据集。

```
#@save
d2l.DATA_HUB['aclImdb'] = (
    'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz',
    '01ada507287d82875905620988597833ad4e0903')
```

(continues on next page)

²⁰⁰ <https://ai.stanford.edu/~amaas/data/sentiment/>

(continued from previous page)

```
data_dir = d2l.download_extract('aclImdb', 'aclImdb')
```

接下来，读取训练和测试数据集。每个样本都是一个评论及其标签：1表示“积极”，0表示“消极”。

```
#@save
def read_imdb(data_dir, is_train):
    """读取IMDb评论数据集文本序列和标签"""
    data, labels = [], []
    for label in ('pos', 'neg'):
        folder_name = os.path.join(data_dir, 'train' if is_train else 'test',
                                    label)
        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', ' ')
            data.append(review)
            labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb(data_dir, is_train=True)
print('训练集数目: ', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('标签: ', y, 'review: ', x[0:60])
```

```
训练集数目: 25000
标签: 1 review: Zentropa has much in common with The Third Man, another noir
标签: 1 review: Zentropa is the most original movie I've seen in years. If y
标签: 1 review: Lars Von Trier is never backward in trying out new technique
```

15.1.2 预处理数据集

将每个单词作为一个词元，过滤掉出现不到5次的单词，我们从训练数据集中创建一个词表。

```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=['<pad>'])
```

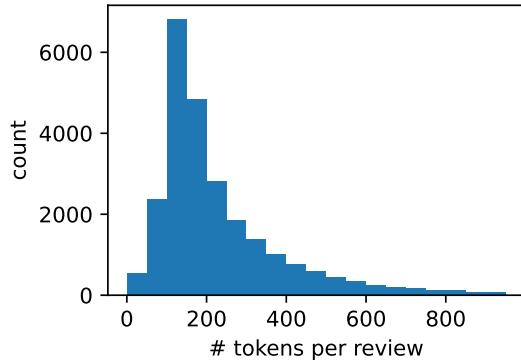
在词元化之后，让我们绘制评论词元长度的直方图。

```
d2l.set_figsize()
d2l.plt.xlabel('# tokens per review')
```

(continues on next page)

(continued from previous page)

```
d2l=plt.ylabel('count')
d2l=plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 50));
```



正如我们所料，评论的长度各不相同。为了每次处理一小批量这样的评论，我们通过截断和填充将每个评论的长度设置为500。这类似于 9.5 节中对机器翻译数据集的预处理步骤。

```
num_steps = 500 # 序列长度
train_features = torch.tensor([d2l.truncate_pad(
    vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
print(train_features.shape)
```

```
torch.Size([25000, 500])
```

15.1.3 创建数据迭代器

现在我们可以创建数据迭代器了。在每次迭代中，都会返回一小批量样本。

```
train_iter = d2l.load_array((train_features,
    torch.tensor(train_data[1])), 64)

for X, y in train_iter:
    print('X:', X.shape, ', y:', y.shape)
    break
print('小批量数目：', len(train_iter))
```

```
X: torch.Size([64, 500]) , y: torch.Size([64])
小批量数目： 391
```

15.1.4 整合代码

最后，我们将上述步骤封装到load_data_imdb函数中。它返回训练和测试数据迭代器以及IMDb评论数据集的词表。

```
#@save
def load_data_imdb(batch_size, num_steps=500):
    """返回数据迭代器和IMDb评论数据集的词表"""
    data_dir = d2l.download_extract('aclImdb', 'aclImdb')
    train_data = read_imdb(data_dir, True)
    test_data = read_imdb(data_dir, False)
    train_tokens = d2l.tokenize(train_data[0], token='word')
    test_tokens = d2l.tokenize(test_data[0], token='word')
    vocab = d2l.Vocab(train_tokens, min_freq=5)
    train_features = torch.tensor([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
    test_features = torch.tensor([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in test_tokens])
    train_iter = d2l.load_array((train_features, torch.tensor(train_data[1])),
                                batch_size)
    test_iter = d2l.load_array((test_features, torch.tensor(test_data[1])),
                               batch_size,
                               is_train=False)
    return train_iter, test_iter, vocab
```

小结

- 情感分析研究人们在文本中的情感，这被认为是一个文本分类问题，它将可变长度的文本序列进行转换转换为固定长度的文本类别。
- 经过预处理后，我们可以使用词表将IMDb评论数据集加载到数据迭代器中。

练习

1. 我们可以修改本节中的哪些超参数来加速训练情感分析模型？
2. 请实现一个函数来将Amazon reviews²⁰¹的数据集加载到数据迭代器中进行情感分析。

Discussions²⁰²

²⁰¹ <https://snap.stanford.edu/data/web-Amazon.html>

²⁰² <https://discuss.d2l.ai/t/5726>

15.2 情感分析：使用循环神经网络

与词相似度和类比任务一样，我们也可以将预先训练的词向量应用于情感分析。由于 15.1节中的IMDb评论数据集不是很大，使用在大规模语料库上预训练的文本表示可以减少模型的过拟合。作为 图15.2.1中所示的具体示例，我们将使用预训练的GloVe模型来表示每个词元，并将这些词元表示送入多层次双向循环神经网络以获得文本序列表示，该文本序列表示将被转换为情感分析输出 (Maas et al., 2011)。对于相同的下游应用，我们稍后将考虑不同的架构选择。

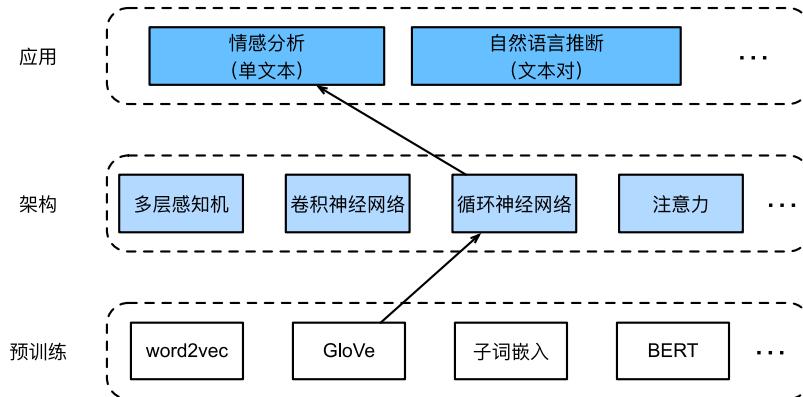


图15.2.1: 将GloVe送入基于循环神经网络的架构，用于情感分析

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

15.2.1 使用循环神经网络表示单个文本

在文本分类任务（如情感分析）中，可变长度的文本序列将被转换为固定长度的类别。在下面的BiRNN类中，虽然文本序列的每个词元经由嵌入层 (`self.embedding`) 获得其单独的预训练GloVe表示，但是整个序列由双向循环神经网络 (`self.encoder`) 编码。更具体地说，双向长短期记忆网络在初始和最终时间步的隐状态（在最后一层）被连结起来作为文本序列的表示。然后，通过一个具有两个输出（“积极”和“消极”）的全连接层 (`self.decoder`)，将此单一文本表示转换为输出类别。

```
class BiRNN(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, **kwargs):
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
```

(continues on next page)

```

# 将bidirectional设置为True以获取双向循环神经网络
self.encoder = nn.LSTM(embed_size, num_hiddens, num_layers=num_layers,
                      bidirectional=True)
self.decoder = nn.Linear(4 * num_hiddens, 2)

def forward(self, inputs):
    # inputs的形状是（批量大小， 时间步数）
    # 因为长短期记忆网络要求其输入的第一个维度是时间维，
    # 所以在获得词元表示之前， 输入会被转置。
    # 输出形状为（时间步数， 批量大小， 词向量维度）
    embeddings = self.embedding(inputs.T)
    self.encoder.flatten_parameters()
    # 返回上一个隐藏层在不同时间步的隐状态，
    # outputs的形状是（时间步数， 批量大小， 2*隐藏单元数）
    outputs, _ = self.encoder(embeddings)
    # 连结初始和最终时间步的隐状态，作为全连接层的输入，
    # 其形状为（批量大小， 4*隐藏单元数）
    encoding = torch.cat((outputs[0], outputs[-1]), dim=1)
    outs = self.decoder(encoding)
    return outs

```

让我们构造一个具有两个隐藏层的双向循环神经网络来表示单个文本以进行情感分析。

```

embed_size, num_hiddens, num_layers = 100, 100, 2
devices = d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)

```

```

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
    if type(m) == nn.LSTM:
        for param in m._flat_weights_names:
            if "weight" in param:
                nn.init.xavier_uniform_(m._parameters[param])
net.apply(init_weights);

```

15.2.2 加载预训练的词向量

下面，我们为词表中的单词加载预训练的100维（需要与embed_size一致）的GloVe嵌入。

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
```

打印词表中所有词元向量的形状。

```
embeds = glove_embedding[vocab.idx_to_token]  
embeds.shape
```

```
torch.Size([49346, 100])
```

我们使用这些预训练的词向量来表示评论中的词元，并且在训练期间不要更新这些向量。

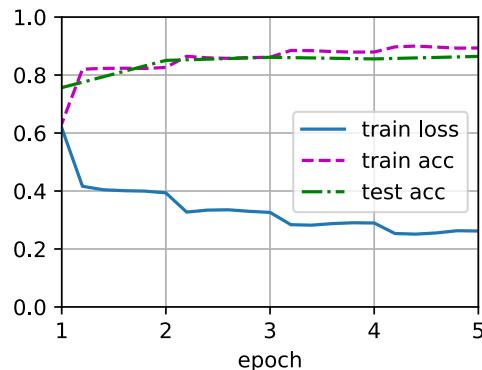
```
net.embedding.weight.data.copy_(embeds)  
net.embedding.weight.requires_grad = False
```

15.2.3 训练和评估模型

现在我们可以训练双向循环神经网络进行情感分析。

```
lr, num_epochs = 0.01, 5  
trainer = torch.optim.Adam(net.parameters(), lr=lr)  
loss = nn.CrossEntropyLoss(reduction="none")  
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,  
    devices)
```

```
loss 0.262, train acc 0.893, test acc 0.864  
2902.4 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



我们定义以下函数来使用训练好的模型net预测文本序列的情感。

```
#@save
def predict_sentiment(net, vocab, sequence):
    """预测文本序列的情感"""
    sequence = torch.tensor(vocab[sequence.split()], device=d2l.try_gpu())
    label = torch.argmax(net(sequence.reshape(1, -1)), dim=1)
    return 'positive' if label == 1 else 'negative'
```

最后，让我们使用训练好的模型对两个简单的句子进行情感预测。

```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

小结

- 预训练的词向量可以表示文本序列中的各个词元。
- 双向循环神经网络可以表示文本序列。例如通过连结初始和最终时间步的隐状态，可以使用全连接的层将该单个文本表示转换为类别。

练习

1. 增加迭代轮数可以提高训练和测试的准确性吗？调优其他超参数怎么样？
2. 使用较大的预训练词向量，例如300维的GloVe嵌入。它是否提高了分类精度？
3. 是否可以通过spaCy词元化来提高分类精度？需要安装Spacy(`pip install spacy`)和英语语言包(`python -m spacy download en`)。在代码中，首先导入Spacy (`import spacy`)。然后，加载Spacy英语软件包 (`spacy_en = spacy.load('en')`)。最后，定义函数`def tokenizer(text): return [tok.text for tok in spacy_en.tokenizer(text)]`并替换原来的`tokenizer`函数。请注意GloVe和spaCy中短语标记的不同形式。例如，短语标记“new york”在GloVe中的形式是“new-york”，而在spaCy词元化之后的形式是“new york”。

Discussions²⁰³

²⁰³ <https://discuss.d2l.ai/t/5724>

15.3 情感分析：使用卷积神经网络

在6节中，我们探讨了使用二维卷积神经网络处理二维图像数据的机制，并将其应用于局部特征，如相邻像素。虽然卷积神经网络最初是为计算机视觉设计的，但它也被广泛用于自然语言处理。简单地说，只要将任何文本序列想象成一维图像即可。通过这种方式，一维卷积神经网络可以处理文本中的局部特征，例如n元语法。

本节将使用textCNN模型来演示如何设计一个表示单个文本(Kim, 2014)的卷积神经网络架构。与图15.2.1中使用带有GloVe预训练的循环神经网络架构进行情感分析相比，图15.3.1中唯一的区别在于架构的选择。

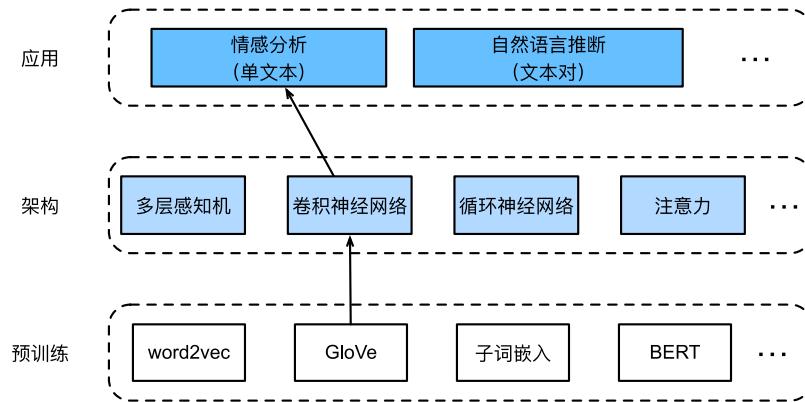


图15.3.1: 将GloVe放入卷积神经网络架构进行情感分析

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

```
Downloading ../data/aclImdb_v1.tar.gz from http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz...
```

15.3.1 一维卷积

在介绍该模型之前，让我们先看看一维卷积是如何工作的。请记住，这只是基于互相关运算的二维卷积的特例。



图15.3.2: 一维互相关运算。阴影部分是第一个输出元素以及用于输出计算的输入和核张量元素: $0 \times 1 + 1 \times 2 = 2$

如 图15.3.2中所示，在一维情况下，卷积窗口在输入张量上从左向右滑动。在滑动期间，卷积窗口中某个位置包含的输入子张量（例如，图15.3.2中的0和1）和核张量（例如，图15.3.2中的1和2）按元素相乘。这些乘法的总和在输出张量的相应位置给出单个标量值（例如，图15.3.2中的 $0 \times 1 + 1 \times 2 = 2$ ）。

我们在下面的corr1d函数中实现了一维互相关。给定输入张量X和核张量K，它返回输出张量Y。

```
def corr1d(X, K):
    w = K.shape[0]
    Y = torch.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i:i + w] * K).sum()
    return Y
```

我们可以从 图15.3.2构造输入张量X和核张量K来验证上述一维互相关实现的输出。

```
X, K = torch.tensor([0, 1, 2, 3, 4, 5, 6]), torch.tensor([1, 2])
corr1d(X, K)
```

```
tensor([ 2.,  5.,  8., 11., 14., 17.])
```

对于任何具有多个通道的一维输入，卷积核需要具有相同数量的输入通道。然后，对于每个通道，对输入的一维张量和卷积核的一维张量执行互相关运算，将所有通道上的结果相加以产生一维输出张量。图15.3.3演示了具有3个输入通道的一维互相关操作。

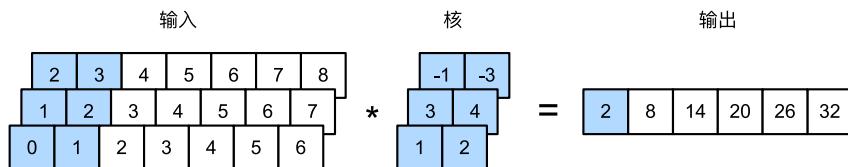


图15.3.3: 具有3个输入通道的一维互相关运算。阴影部分是第一个输出元素以及用于输出计算的输入和核张量元素: $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$

我们可以实现多个输入通道的一维互相关运算，并在 图15.3.3中验证结果。

```

def corr1d_multi_in(X, K):
    # 首先，遍历'X'和'K'的第0维（通道维）。然后，把它们加在一起
    return sum(corr1d(x, k) for x, k in zip(X, K))

X = torch.tensor([[0, 1, 2, 3, 4, 5, 6],
                  [1, 2, 3, 4, 5, 6, 7],
                  [2, 3, 4, 5, 6, 7, 8]])
K = torch.tensor([[1, 2], [3, 4], [-1, -3]])
corr1d_multi_in(X, K)

```

```
tensor([ 2.,  8., 14., 20., 26., 32.])
```

注意，多输入通道的一维互相关等同于单输入通道的二维互相关。举例说明，图15.3.3中的多输入通道一维互相关的等价形式是图15.3.4中的单输入通道二维互相关，其中卷积核的高度必须与输入张量的高度相同。

输入	核	输出																											
<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </table>	2	3	4	5	6	7	8	1	2	3	4	5	6	7	0	1	2	3	4	5	6	*	<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td>-1</td><td>-3</td></tr> <tr><td>3</td><td>4</td></tr> <tr><td>1</td><td>2</td></tr> </table>	-1	-3	3	4	1	2
2	3	4	5	6	7	8																							
1	2	3	4	5	6	7																							
0	1	2	3	4	5	6																							
-1	-3																												
3	4																												
1	2																												
	=	<table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td>2</td><td>8</td><td>14</td><td>20</td><td>26</td><td>32</td></tr> </table>	2	8	14	20	26	32																					
2	8	14	20	26	32																								

图15.3.4: 具有单个输入通道的二维互相关操作。阴影部分是第一个输出元素以及用于输出计算的输入和内核张量元素: $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$

图15.3.2和图15.3.3中的输出都只有一个通道。与 subsec_multi-output-channels 中描述的具有多个输出通道的二维卷积相同，我们也可以为一维卷积指定多个输出通道。

15.3.2 最大时间汇聚层

类似地，我们可以使用汇聚层从序列表示中提取最大值，作为跨时间步的最重要特征。textCNN中使用的最大时间汇聚层的工作原理类似于一维全局汇聚 (Collobert *et al.*, 2011)。对于每个通道在不同时间步存储值的多通道输入，每个通道的输出是该通道的最大值。请注意，最大时间汇聚允许在不同通道上使用不同数量的时间步。

15.3.3 textCNN模型

使用一维卷积和最大时间汇聚，textCNN模型将单个预训练的词元表示作为输入，然后获得并转换用于下游应用的序列表示。

对于具有由 d 维向量表示的 n 个词元的单个文本序列，输入张量的宽度、高度和通道数分别为 n 、1和 d 。textCNN模型将输入转换为输出，如下所示：

1. 定义多个一维卷积核，并分别对输入执行卷积运算。具有不同宽度的卷积核可以捕获不同数目的相邻词元之间的局部特征。
2. 在所有输出通道上执行最大时间汇聚层，然后将所有标量汇聚输出连结为向量。
3. 使用全连接层将连结后的向量转换为输出类别。Dropout可以用来减少过拟合。

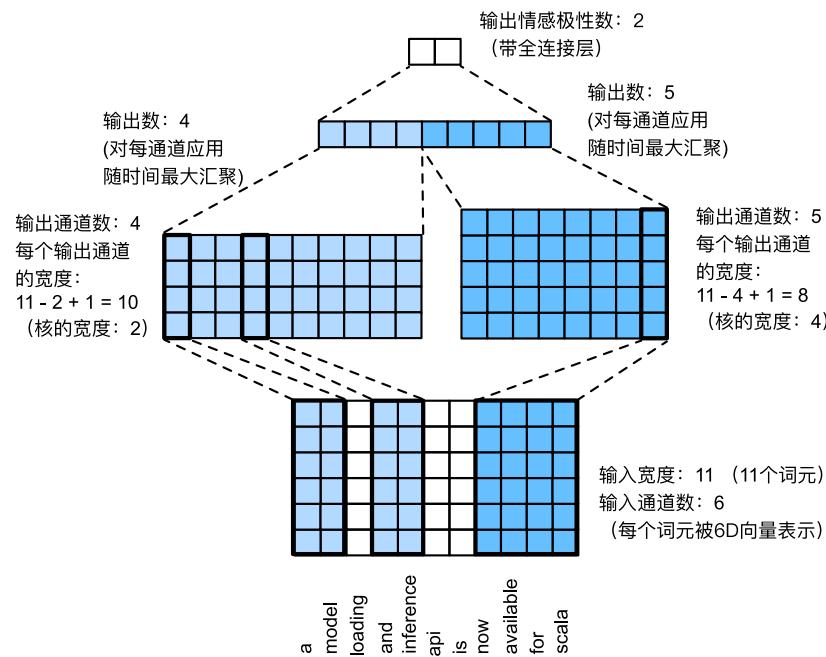


图15.3.5: textCNN的模型架构

图15.3.5通过一个具体的例子说明了textCNN的模型架构。输入是具有11个词元的句子，其中每个词元由6维向量表示。因此，我们有一个宽度为11的6通道输入。定义两个宽度为2和4的一维卷积核，分别具有4个和5个输出通道。它们产生4个宽度为 $11 - 2 + 1 = 10$ 的输出通道和5个宽度为 $11 - 4 + 1 = 8$ 的输出通道。尽管这9个通道的宽度不同，但最大时间汇聚层给出了一个连结的9维向量，该向量最终被转换为用于二元情感预测的2维输出向量。

定义模型

我们在下面的类中实现textCNN模型。与 15.2节的双向循环神经网络模型相比，除了用卷积层代替循环神经网络层外，我们还使用了两个嵌入层：一个是可训练权重，另一个是固定权重。

```
class TextCNN(nn.Module):
    def __init__(self, vocab_size, embed_size, kernel_sizes, num_channels,
                 **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # 这个嵌入层不需要训练
        self.constant_embedding = nn.Embedding(vocab_size, embed_size)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Linear(sum(num_channels), 2)
        # 最大时间汇聚层没有参数，因此可以共享此实例
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.relu = nn.ReLU()
        # 创建多个一维卷积层
        self.convs = nn.ModuleList()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.append(nn.Conv1d(2 * embed_size, c, k))

    def forward(self, inputs):
        # 沿着向量维度将两个嵌入层连结起来,
        # 每个嵌入层的输出形状都是（批量大小，词元数量，词元向量维度）连结起来
        embeddings = torch.cat([
            self.embedding(inputs), self.constant_embedding(inputs), dim=2])
        # 根据一维卷积层的输入格式，重新排列张量，以便通道作为第2维
        embeddings = embeddings.permute(0, 2, 1)
        # 每一个一维卷积层在最大时间汇聚层合并后，获得的张量形状是（批量大小，通道数，1）
        # 删除最后一个维度并沿通道维度连结
        encoding = torch.cat([
            torch.squeeze(self.relu(self.pool(conv(embeddings))), dim=-1)
            for conv in self.convs], dim=1)
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

让我们创建一个textCNN实例。它有3个卷积层，卷积核宽度分别为3、4和5，均有100个输出通道。

```
embed_size, kernel_sizes, num_channels = 100, [3, 4, 5], [100, 100, 100]
devices = d2l.try_all_gpus()
net = TextCNN(len(vocab), embed_size, kernel_sizes, num_channels)

def init_weights(m):
```

(continues on next page)

(continued from previous page)

```
if type(m) in (nn.Linear, nn.Conv1d):
    nn.init.xavier_uniform_(m.weight)

net.apply(init_weights);
```

加载预训练词向量

与 15.2 节相同，我们加载预训练的100维GloVe嵌入作为初始化的词元表示。这些词元表示（嵌入权重）在embedding中将被训练，在constant_embedding中将被固定。

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.requires_grad = False
```

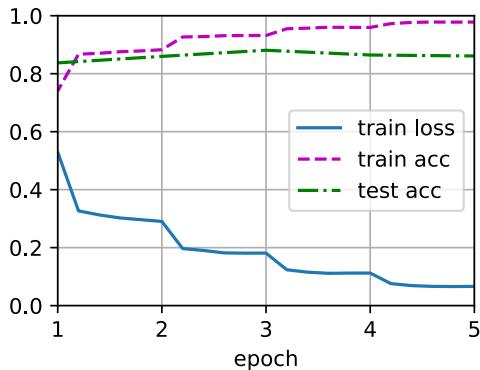
```
Downloading ../data/glove.6B.100d.zip from http://d2l-data.s3-accelerate.amazonaws.com/glove.6B.100d.
→zip...
```

训练和评估模型

现在我们可以训练textCNN模型进行情感分析。

```
lr, num_epochs = 0.001, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.066, train acc 0.978, test acc 0.861
4609.1 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



下面，我们使用训练好的模型来预测两个简单句子的情感。

```
d2l.predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
d2l.predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

小结

- 一维卷积神经网络可以处理文本中的局部特征，例如 n 元语法。
- 多输入通道的一维互相关等价于单输入通道的二维互相关。
- 最大时间汇聚层允许在不同通道上使用不同数量的时间步长。
- textCNN模型使用一维卷积层和最大时间汇聚层将单个词元表示转换为下游应用输出。

练习

- 调整超参数，并比较 15.2 节中用于情感分析的架构和本节中用于情感分析的架构，例如在分类精度和计算效率方面。
- 请试着用 15.2 节练习中介绍的方法进一步提高模型的分类精度。
- 在输入表示中添加位置编码。它是否提高了分类的精度？

Discussions²⁰⁴

²⁰⁴ <https://discuss.d2l.ai/t/5720>

15.4 自然语言推断与数据集

在 15.1 节中，我们讨论了情感分析问题。这个任务的目的是将单个文本序列分类到预定义的类别中，例如一组情感极性中。然而，当需要决定一个句子是否可以从另一个句子推断出来，或者需要通过识别语义等价的句子来消除句子间冗余时，知道如何对一个文本序列进行分类是不够的。相反，我们需要能够对成对的文本序列进行推断。

15.4.1 自然语言推断

自然语言推断 (natural language inference) 主要研究假设 (hypothesis) 是否可以从前提 (premise) 中推断出来，其中两者都是文本序列。换言之，自然语言推断决定了一对文本序列之间的逻辑关系。这类关系通常分为三种类型：

- 蕴涵 (entailment)：假设可以从前提中推断出来。
- 矛盾 (contradiction)：假设的否定可以从前提中推断出来。
- 中性 (neutral)：所有其他情况。

自然语言推断也被称为识别文本蕴涵任务。例如，下面的一个文本对将被贴上“蕴涵”的标签，因为假设中的“表白”可以从前提中的“拥抱”中推断出来。

前提：两个女人拥抱在一起。

假设：两个女人在示爱。

下面是一个“矛盾”的例子，因为“运行编码示例”表示“不睡觉”，而不是“睡觉”。

前提：一名男子正在运行 Dive Into Deep Learning 的编码示例。

假设：该男子正在睡觉。

第三个例子显示了一种“中性”关系，因为“正在为我们表演”这一事实无法推断出“出名”或“不出名”。

前提：音乐家们正在为我们表演。

假设：音乐家很有名。

自然语言推断一直是理解自然语言的中心话题。它有着广泛的应用，从信息检索到开放领域的问答。为了研究这个问题，我们将首先研究一个流行的自然语言推断基准数据集。

15.4.2 斯坦福自然语言推断 (SNLI) 数据集

斯坦福自然语言推断语料库 (Stanford Natural Language Inference, SNLI) 是由 500000 多个带标签的英语句子对组成的集合 (Bowman *et al.*, 2015)。我们在路径 `..../data/snli_1.0` 中下载并存储提取的 SNLI 数据集。

```

import os
import re
import torch
from torch import nn
from d2l import torch as d2l

#@save
d2l.DATA_HUB['SNLI'] = (
    'https://nlp.stanford.edu/projects/snli/snli_1.0.zip',
    '9fcde07509c7e87ec61c640c1b2753d9041758e4')

data_dir = d2l.download_extract('SNLI')

```

读取数据集

原始的SNLI数据集包含的信息比我们在实验中真正需要的信息丰富得多。因此，我们定义函数`read_snli`以仅提取数据集的一部分，然后返回前提、假设及其标签的列表。

```

#@save
def read_snli(data_dir, is_train):
    """将SNLI数据集解析为前提、假设和标签"""
    def extract_text(s):
        # 删除我们不会使用的信息
        s = re.sub('\\(', '', s)
        s = re.sub('\\)', '', s)
        # 用一个空格替换两个或多个连续的空格
        s = re.sub('\\s{2,}', ' ', s)
        return s.strip()
    label_set = {'entailment': 0, 'contradiction': 1, 'neutral': 2}
    file_name = os.path.join(data_dir, 'snli_1.0_train.txt'
                             if is_train else 'snli_1.0_test.txt')
    with open(file_name, 'r') as f:
        rows = [row.split('\t') for row in f.readlines()[1:]]
    premises = [extract_text(row[1]) for row in rows if row[0] in label_set]
    hypotheses = [extract_text(row[2]) for row in rows if row[0] \
                  in label_set]
    labels = [label_set[row[0]] for row in rows if row[0] in label_set]
    return premises, hypotheses, labels

```

现在让我们打印前3对前提和假设，以及它们的标签（“0”“1”和“2”分别对应于“蕴涵”“矛盾”和“中性”）。

```
train_data = read_snli(data_dir, is_train=True)
```

(continues on next page)

(continued from previous page)

```
for x0, x1, y in zip(train_data[0][:3], train_data[1][:3], train_data[2][:3]):  
    print('前提: ', x0)  
    print('假设: ', x1)  
    print('标签: ', y)
```

前提: A person on a horse jumps over a broken down airplane .
假设: A person is training his horse for a competition .
标签: 2
前提: A person on a horse jumps over a broken down airplane .
假设: A person is at a diner , ordering an omelette .
标签: 1
前提: A person on a horse jumps over a broken down airplane .
假设: A person is outdoors , on a horse .
标签: 0

训练集约有550000对，测试集约有10000对。下面显示了训练集和测试集中的三个标签“蕴涵”“矛盾”和“中性”是平衡的。

```
test_data = read_snli(data_dir, is_train=False)  
for data in [train_data, test_data]:  
    print([[row for row in data[2]].count(i) for i in range(3)])
```

```
[183416, 183187, 182764]  
[3368, 3237, 3219]
```

定义用于加载数据集的类

下面我们来定义一个用于加载SNLI数据集的类。类构造函数中的变量num_steps指定文本序列的长度，使得每个小批量序列将具有相同的形状。换句话说，在较长序列中的前num_steps个标记之后的标记被截断，而特殊标记“<pad>”将被附加到较短的序列后，直到它们的长度变为num_steps。通过实现__getitem__功能，我们可以任意访问带有索引idx的前提、假设和标签。

```
#@save  
class SNLIDataset(torch.utils.data.Dataset):  
    """用于加载SNLI数据集的自定义数据集"""\n    def __init__(self, dataset, num_steps, vocab=None):  
        self.num_steps = num_steps  
        all_premise_tokens = d2l.tokenize(dataset[0])  
        all_hypothesis_tokens = d2l.tokenize(dataset[1])  
        if vocab is None:
```

(continues on next page)

(continued from previous page)

```
self.vocab = d2l.Vocab(all_premise_tokens + \
    all_hypothesis_tokens, min_freq=5, reserved_tokens=['<pad>'])

else:
    self.vocab = vocab
    self.premises = self._pad(all_premise_tokens)
    self.hypotheses = self._pad(all_hypothesis_tokens)
    self.labels = torch.tensor(dataset[2])
    print('read ' + str(len(self.premises)) + ' examples')

def _pad(self, lines):
    return torch.tensor([d2l.truncate_pad(
        self.vocab[line], self.num_steps, self.vocab['<pad>'])
        for line in lines])

def __getitem__(self, idx):
    return (self.premises[idx], self.hypotheses[idx]), self.labels[idx]

def __len__(self):
    return len(self.premises)
```

整合代码

现在，我们可以调用read_snli函数和SNLIDataset类来下载SNLI数据集，并返回训练集和测试集的DataLoader实例，以及训练集的词表。值得注意的是，我们必须使用从训练集构造的词表作为测试集的词表。因此，在训练集中训练的模型将不知道来自测试集的任何新词元。

```
#@save
def load_data_snli(batch_size, num_steps=50):
    """下载SNLI数据集并返回数据迭代器和词表"""
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('SNLI')
    train_data = read_snli(data_dir, True)
    test_data = read_snli(data_dir, False)
    train_set = SNLIDataset(train_data, num_steps)
    test_set = SNLIDataset(test_data, num_steps, train_set.vocab)
    train_iter = torch.utils.data.DataLoader(train_set, batch_size,
                                             shuffle=True,
                                             num_workers=num_workers)
    test_iter = torch.utils.data.DataLoader(test_set, batch_size,
                                             shuffle=False,
                                             num_workers=num_workers)
    return train_iter, test_iter, train_set.vocab
```

在这里，我们将批量大小设置为128时，将序列长度设置为50，并调用load_data_snli函数来获取数据迭代器和词表。然后我们打印词表大小。

```
train_iter, test_iter, vocab = load_data_snli(128, 50)
len(vocab)
```

```
read 549367 examples
read 9824 examples
```

```
18678
```

现在我们打印第一个小批量的形状。与情感分析相反，我们有分别代表前提和假设的两个输入 $X[0]$ 和 $X[1]$ 。

```
for X, Y in train_iter:
    print(X[0].shape)
    print(X[1].shape)
    print(Y.shape)
    break
```

```
torch.Size([128, 50])
torch.Size([128, 50])
torch.Size([128])
```

小结

- 自然语言推断研究“假设”是否可以从“前提”推断出来，其中两者都是文本序列。
- 在自然语言推断中，前提和假设之间的关系包括蕴涵关系、矛盾关系和中性关系。
- 斯坦福自然语言推断（SNLI）语料库是一个比较流行的自然语言推断基准数据集。

练习

- 机器翻译长期以来一直是基于翻译输出和翻译真实值之间的表面 n 元语法匹配来进行评估的。可以设计一种用自然语言推断来评价机器翻译结果的方法吗？
- 我们如何更改超参数以减小词表大小？

Discussions²⁰⁵

²⁰⁵ <https://discuss.d2l.ai/t/5722>

15.5 自然语言推断：使用注意力

我们在15.4节中介绍了自然语言推断任务和SNLI数据集。鉴于许多模型都是基于复杂而深度的架构，Parikh等人提出用注意力机制解决自然语言推断问题，并称之为“可分解注意力模型”(Parikh et al., 2016)。这使得模型没有循环层或卷积层，在SNLI数据集上以更少的参数实现了当时的最佳结果。本节将描述并实现这种基于注意力的自然语言推断方法（使用MLP），如图15.5.1中所述。

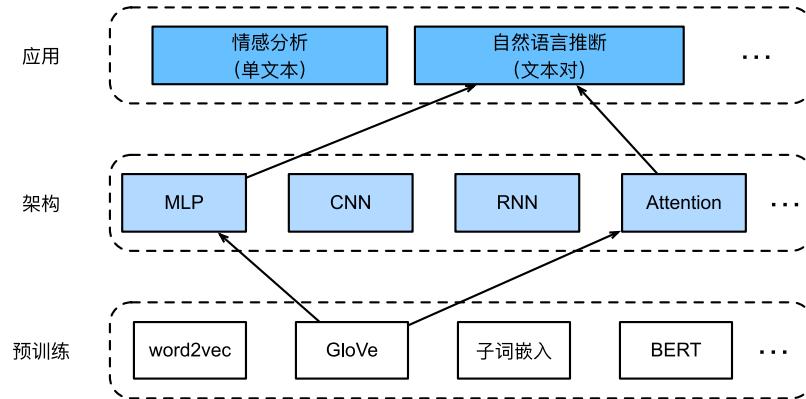


图15.5.1: 将预训练GloVe送入基于注意力和MLP的自然语言推断架构

15.5.1 模型

与保留前提和假设中词元的顺序相比，我们可以将一个文本序列中的词元与另一个文本序列中的每个词元对齐，然后比较和聚合这些信息，以预测前提和假设之间的逻辑关系。与机器翻译中源句和目标句之间的词元对齐类似，前提和假设之间的词元对齐可以通过注意力机制灵活地完成。

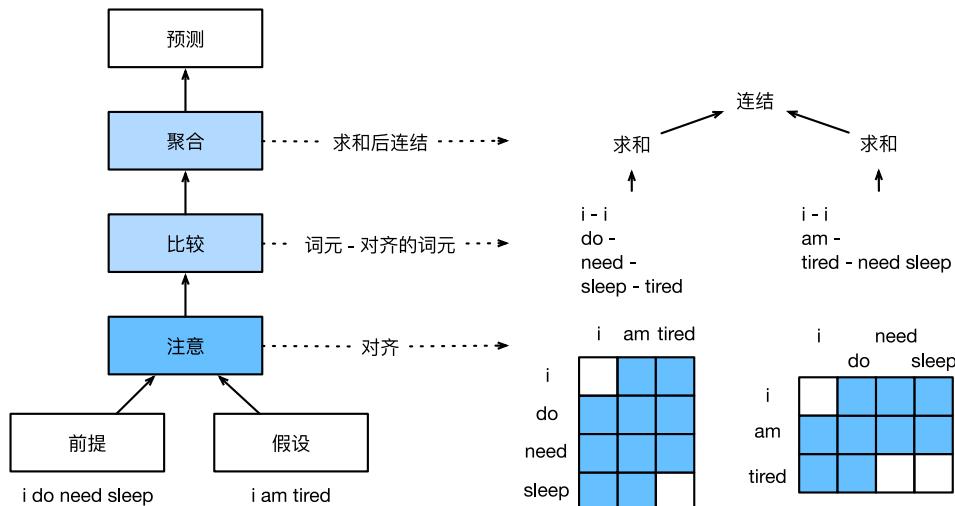


图15.5.2: 利用注意力机制进行自然语言推断

图15.5.2描述了使用注意力机制的自然语言推断方法。从高层次上讲，它由三个联合训练的步骤组成：对齐、

比较和汇总。我们将在下面一步一步地对它们进行说明。

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

注意 (Attending)

第一步是将一个文本序列中的词元与另一个序列中的每个词元对齐。假设前提是“我确实需要睡眠”，假设是“我累了”。由于语义上的相似性，我们不妨将假设中的“我”与前提中的“我”对齐，将假设中的“累”与前提中的“睡眠”对齐。同样，我们可能希望将前提中的“我”与假设中的“我”对齐，将前提中的“需要”和“睡眠”与假设中的“累”对齐。请注意，这种对齐是使用加权平均的“软”对齐，其中理想情况下较大的权重与要对齐的词元相关联。为了便于演示，图15.5.2以“硬”对齐的方式显示了这种对齐方式。

现在，我们更详细地描述使用注意力机制的软对齐。用 $\mathbf{A} = (\mathbf{a}_1, \dots, \mathbf{a}_m)$ 和 $\mathbf{B} = (\mathbf{b}_1, \dots, \mathbf{b}_n)$ 表示前提和假设，其词元数量分别为 m 和 n ，其中 $\mathbf{a}_i, \mathbf{b}_j \in \mathbb{R}^d$ ($i = 1, \dots, m, j = 1, \dots, n$) 是 d 维的词向量。对于软对齐，我们将注意力权重 $e_{ij} \in \mathbb{R}$ 计算为：

$$e_{ij} = f(\mathbf{a}_i)^\top f(\mathbf{b}_j), \quad (15.5.1)$$

其中函数 f 是在下面的mlp函数中定义的多层感知机。输出维度 f 由mlp的num_hiddens参数指定。

```
def mlp(num_inputs, num_hiddens, flatten):
    net = []
    net.append(nn.Dropout(0.2))
    net.append(nn.Linear(num_inputs, num_hiddens))
    net.append(nn.ReLU())
    if flatten:
        net.append(nn.Flatten(start_dim=1))
    net.append(nn.Dropout(0.2))
    net.append(nn.Linear(num_hiddens, num_hiddens))
    net.append(nn.ReLU())
    if flatten:
        net.append(nn.Flatten(start_dim=1))
    return nn.Sequential(*net)
```

值得注意的是，在(15.5.1)中， f 分别输入 \mathbf{a}_i 和 \mathbf{b}_j ，而不是将它们一对放在一起作为输入。这种分解技巧导致 f 只有 $m + n$ 个次计算（线性复杂度），而不是 mn 次计算（二次复杂度）。

对(15.5.1)中的注意力权重进行规范化，我们计算假设中所有词元向量的加权平均值，以获得假设的表示，该假设与前提中索引 i 的词元进行软对齐：

$$\beta_i = \sum_{j=1}^n \frac{\exp(e_{ij})}{\sum_{k=1}^n \exp(e_{ik})} \mathbf{b}_j. \quad (15.5.2)$$

同样，我们计算假设中索引为 j 的每个词元与前提词元的软对齐：

$$\alpha_j = \sum_{i=1}^m \frac{\exp(e_{ij})}{\sum_{k=1}^m \exp(e_{kj})} \mathbf{a}_i. \quad (15.5.3)$$

下面，我们定义Attend类来计算假设(beta)与输入前提A的软对齐以及前提(alpha)与输入假设B的软对齐。

```
class Attend(nn.Module):
    def __init__(self, num_inputs, num_hiddens, **kwargs):
        super(Attend, self).__init__(**kwargs)
        self.f = mlp(num_inputs, num_hiddens, flatten=False)

    def forward(self, A, B):
        # A/B的形状: (批量大小, 序列A/B的词元数, embed_size)
        # f_A/f_B的形状: (批量大小, 序列A/B的词元数, num_hiddens)
        f_A = self.f(A)
        f_B = self.f(B)
        # e的形状: (批量大小, 序列A的词元数, 序列B的词元数)
        e = torch.bmm(f_A, f_B.permute(0, 2, 1))
        # beta的形状: (批量大小, 序列A的词元数, embed_size),
        # 意味着序列B被软对齐到序列A的每个词元(beta的第一个维度)
        beta = torch.bmm(F.softmax(e, dim=-1), B)
        # beta的形状: (批量大小, 序列B的词元数, embed_size),
        # 意味着序列A被软对齐到序列B的每个词元(alpha的第一个维度)
        alpha = torch.bmm(F.softmax(e.permute(0, 2, 1), dim=-1), A)
        return beta, alpha
```

比较

在下一步中，我们将一个序列中的词元与与该词元软对齐的另一个序列进行比较。请注意，在软对齐中，一个序列中的所有词元（尽管可能具有不同的注意力权重）将与另一个序列中的词元进行比较。为便于演示，图15.5.2对词元以硬的方式对齐。例如，上述的注意（attending）步骤确定前提中的“need”和“sleep”都与假设中的“tired”对齐，则将对“疲倦-需要睡眠”进行比较。

在比较步骤中，我们将来自一个序列的词元的连结（运算符 $[., .]$ ）和来自另一序列的对齐的词元送入函数 g （一个多层感知机）：

$$\begin{aligned} \mathbf{v}_{A,i} &= g([\mathbf{a}_i, \beta_i]), i = 1, \dots, m \\ \mathbf{v}_{B,j} &= g([\mathbf{b}_j, \alpha_j]), j = 1, \dots, n. \end{aligned} \quad (15.5.4)$$

在(15.5.4)中， $\mathbf{v}_{A,i}$ 是指，所有假设中的词元与前提中词元*i*软对齐，再与词元*i*的比较；而 $\mathbf{v}_{B,j}$ 是指，所有前提中的词元与假设中词元*i*软对齐，再与词元*i*的比较。下面的Compare个类定义了比较步骤。

```
class Compare(nn.Module):
    def __init__(self, num_inputs, num_hiddens, **kwargs):
```

(continues on next page)

(continued from previous page)

```
super(Compare, self).__init__(**kwargs)
self.g = mlp(num_inputs, num_hiddens, flatten=False)

def forward(self, A, B, beta, alpha):
    V_A = self.g(torch.cat([A, beta], dim=2))
    V_B = self.g(torch.cat([B, alpha], dim=2))
    return V_A, V_B
```

聚合

现在我们有两组比较向量 $\mathbf{v}_{A,i}$ ($i = 1, \dots, m$) 和 $\mathbf{v}_{B,j}$ ($j = 1, \dots, n$)。在最后一步中，我们将聚合这些信息以推断逻辑关系。我们首先求和这两组比较向量：

$$\mathbf{v}_A = \sum_{i=1}^m \mathbf{v}_{A,i}, \quad \mathbf{v}_B = \sum_{j=1}^n \mathbf{v}_{B,j}. \quad (15.5.5)$$

接下来，我们将两个求和结果的连结提供给函数 h （一个多层感知机），以获得逻辑关系的分类结果：

$$\hat{\mathbf{y}} = h([\mathbf{v}_A, \mathbf{v}_B]). \quad (15.5.6)$$

聚合步骤在以下 `Aggregate` 类中定义。

```
class Aggregate(nn.Module):
    def __init__(self, num_inputs, num_hiddens, num_outputs, **kwargs):
        super(Aggregate, self).__init__(**kwargs)
        self.h = mlp(num_inputs, num_hiddens, flatten=True)
        self.linear = nn.Linear(num_hiddens, num_outputs)

    def forward(self, V_A, V_B):
        # 对两组比较向量分别求和
        V_A = V_A.sum(dim=1)
        V_B = V_B.sum(dim=1)
        # 将两个求和结果的连结送到多层感知机中
        Y_hat = self.linear(self.h(torch.cat([V_A, V_B], dim=1)))
        return Y_hat
```

整合代码

通过将注意步骤、比较步骤和聚合步骤组合在一起，我们定义了可分解注意力模型来联合训练这三个步骤。

```
class DecomposableAttention(nn.Module):
    def __init__(self, vocab, embed_size, num_hiddens, num_inputs_attend=100,
                 num_inputs_compare=200, num_inputs_agg=400, **kwargs):
        super(DecomposableAttention, self).__init__(**kwargs)
        self.embedding = nn.Embedding(len(vocab), embed_size)
        self.attend = Attend(num_inputs_attend, num_hiddens)
        self.compare = Compare(num_inputs_compare, num_hiddens)
        # 有3种可能的输出：蕴涵、矛盾和中性
        self.aggregate = Aggregate(num_inputs_agg, num_hiddens, num_outputs=3)

    def forward(self, X):
        premises, hypotheses = X
        A = self.embedding(premises)
        B = self.embedding(hypotheses)
        beta, alpha = self.attend(A, B)
        V_A, V_B = self.compare(A, B, beta, alpha)
        Y_hat = self.aggregate(V_A, V_B)
        return Y_hat
```

15.5.2 训练和评估模型

现在，我们将在SNLI数据集上对定义好的可分解注意力模型进行训练和评估。我们从读取数据集开始。

读取数据集

我们使用 15.4 节中定义的函数下载并读取SNLI数据集。批量大小和序列长度分别设置为256和50。

```
batch_size, num_steps = 256, 50
train_iter, test_iter, vocab = d2l.load_data_snli(batch_size, num_steps)
```

```
Downloading ../data/snli_1.0.zip from https://nlp.stanford.edu/projects/snli/snli_1.0.zip...
read 549367 examples
read 9824 examples
```

创建模型

我们使用预训练好的100维GloVe嵌入来表示输入词元。我们将向量 \mathbf{a}_i 和 \mathbf{b}_j 在(15.5.1)中的维数预定义为100。(15.5.1)中的函数 f 和(15.5.4)中的函数 g 的输出维度被设置为200。然后我们创建一个模型实例，初始化它的参数，并加载GloVe嵌入来初始化输入词元的向量。

```
embed_size, num_hiddens, devices = 100, 200, d2l.try_all_gpus()
net = DecomposableAttention(vocab, embed_size, num_hiddens)
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds);
```

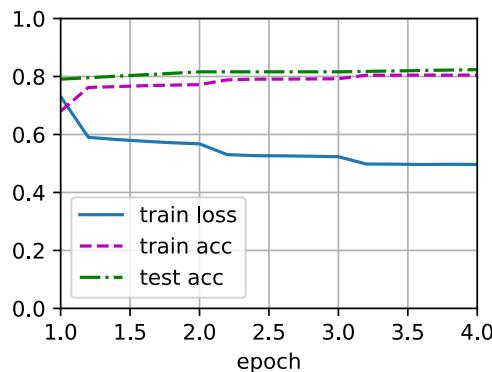
训练和评估模型

与12.5节中接受单一输入（如文本序列或图像）的split_batch函数不同，我们定义了一个split_batch_multi_inputs函数以小批量接受多个输入，如前提和假设。

现在我们可以在SNLI数据集上训练和评估模型。

```
lr, num_epochs = 0.001, 4
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
    devices)
```

```
loss 0.497, train acc 0.805, test acc 0.824
14163.6 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



使用模型

最后，定义预测函数，输出一对前提和假设之间的逻辑关系。

```
#@save
def predict_snli(net, vocab, premise, hypothesis):
    """预测前提和假设之间的逻辑关系"""
    net.eval()
    premise = torch.tensor(vocab[premise], device=d2l.try_gpu())
    hypothesis = torch.tensor(vocab[hypothesis], device=d2l.try_gpu())
    label = torch.argmax(net([premise.reshape((1, -1)),
                             hypothesis.reshape((1, -1))]), dim=1)
    return 'entailment' if label == 0 else 'contradiction' if label == 1 \
        else 'neutral'
```

我们可以使用训练好的模型来获得对示例句子的自然语言推断结果。

```
predict_snli(net, vocab, ['he', 'is', 'good', '.'], ['he', 'is', 'bad', '.'])
```

```
'contradiction'
```

小结

- 可分解注意模型包括三个步骤来预测前提和假设之间的逻辑关系：注意、比较和聚合。
- 通过注意力机制，我们可以将一个文本序列中的词元与另一个文本序列中的每个词元对齐，反之亦然。这种对齐是使用加权平均的软对齐，其中理想情况下较大的权重与要对齐的词元相关联。
- 在计算注意力权重时，分解技巧会带来比二次复杂度更理想的线性复杂度。
- 我们可以使用预训练好的词向量作为下游自然语言处理任务（如自然语言推断）的输入表示。

练习

1. 使用其他超参数组合训练模型，能在测试集上获得更高的准确度吗？
2. 自然语言推断的可分解注意模型的主要缺点是什么？
3. 假设我们想要获得任何一对句子的语义相似级别（例如，0~1之间的连续值）。我们应该如何收集和标注数据集？请尝试设计一个有注意力机制的模型。

Discussions²⁰⁶

²⁰⁶ <https://discuss.d2l.ai/t/5728>

15.6 针对序列级和词元级应用微调BERT

在本章的前几节中，我们为自然语言处理应用设计了不同的模型，例如基于循环神经网络、卷积神经网络、注意力和多层感知机。这些模型在有空间或时间限制的情况下是有帮助的，但是，为每个自然语言处理任务精心设计一个特定的模型实际上是不可行的。在 14.8 节中，我们介绍了一个名为BERT的预训练模型，该模型可以对广泛的自然语言处理任务进行最少的架构更改。一方面，在提出时，BERT改进了各种自然语言处理任务的技术水平。另一方面，正如在 14.10 节中指出的那样，原始BERT模型的两个版本分别带有1.1亿和3.4亿个参数。因此，当有足够的计算资源时，我们可以考虑为下游自然语言处理应用微调BERT。

下面，我们将自然语言处理应用的子集概括为序列级和词元级。在序列层次上，介绍了在单文本分类任务和文本对分类（或回归）任务中，如何将文本输入的BERT表示转换为输出标签。在词元级别，我们将简要介绍新的应用，如文本标注和问答，并说明BERT如何表示它们的输入并转换为输出标签。在微调期间，不同应用之间的BERT所需的“最小架构更改”是额外的全连接层。在下游应用的监督学习期间，额外层的参数是从零开始学习的，而预训练BERT模型中的所有参数都是微调的。

15.6.1 单文本分类

单文本分类将单个文本序列作为输入，并输出其分类结果。除了我们在这一章中探讨的情感分析之外，语言可接受性语料库（Corpus of Linguistic Acceptability, COLA）也是一个单文本分类的数据集，它的要求判断给定的句子在语法上是否可以接受。（Warstadt *et al.*, 2019）。例如，“I should study.” 是可以接受的，但是 “I should studying.” 不是可以接受的。

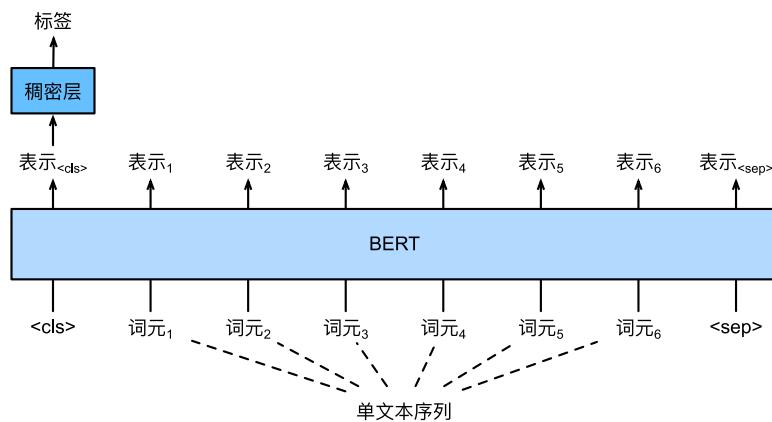


图15.6.1：微调BERT用于单文本分类应用，如情感分析和测试语言可接受性（这里假设输入的单个文本有六个词元）

14.8 节描述了BERT的输入表示。BERT输入序列明确地表示单个文本和文本对，其中特殊分类标记“<cls>”用于序列分类，而特殊分类标记“<sep>”标记单个文本的结束或分隔成对文本。如 图15.6.1所示，在单文本分类应用中，特殊分类标记“<cls>”的BERT表示对整个输入文本序列的信息进行编码。作为输入单个文本的表示，它将被送入到由全连接（稠密）层组成的小多层感知机中，以输出所有离散标签值的分布。

15.6.2 文本对分类或回归

在本章中，我们还研究了自然语言推断。它属于文本对分类，这是一种对文本进行分类的应用类型。

以一对文本作为输入但输出连续值，语义文本相似度是一个流行的“文本对回归”任务。这项任务评估句子的语义相似度。例如，在语义文本相似度基准数据集（Semantic Textual Similarity Benchmark）中，句子对的相似度得分是从0（无语义重叠）到5（语义等价）的分数区间（Cer et al., 2017）。我们的目标是预测这些分数。来自语义文本相似性基准数据集的样本包括（句子1，句子2，相似性得分）：

- “A plane is taking off.” (“一架飞机正在起飞。”), “An air plane is taking off.” (“一架飞机正在起飞。”), 5.000分;
- “A woman is eating something.” (“一个女人在吃东西。”), “A woman is eating meat.” (“一个女人在吃肉。”), 3.000分;
- “A woman is dancing.” (“一个女人在跳舞。”), “A man is talking.” (“一个人在说话。”), 0.000分。

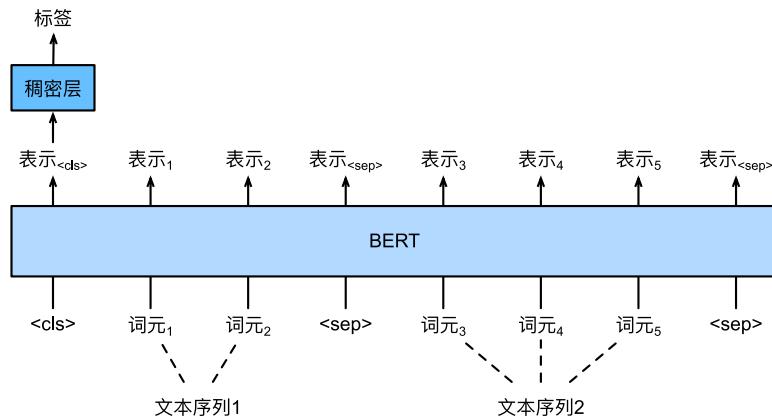


图15.6.2: 文本对分类或回归应用的BERT微调，如自然语言推断和语义文本相似性（假设输入文本对分别有两个词元和三个词元）

与 图15.6.1中的单文本分类相比，图15.6.2中的文本对分类的BERT微调在输入表示上有所不同。对于文本对回归任务（如语义文本相似性），可以应用细微的更改，例如输出连续的标签值和使用均方损失：它们在回归中很常见。

15.6.3 文本标注

现在让我们考虑词元级任务，比如文本标注（text tagging），其中每个词元都被分配了一个标签。在文本标注任务中，词性标注为每个单词分配词性标记（例如，形容词和限定词）。根据单词在句子中的作用。如，在Penn树库II标注集中，句子“John Smith ‘s car is new”应该被标记为“NNP（名词，专有单数） NNP POS（所有格结尾） NN（名词，单数或质量） VB（动词，基本形式） JJ（形容词）”。

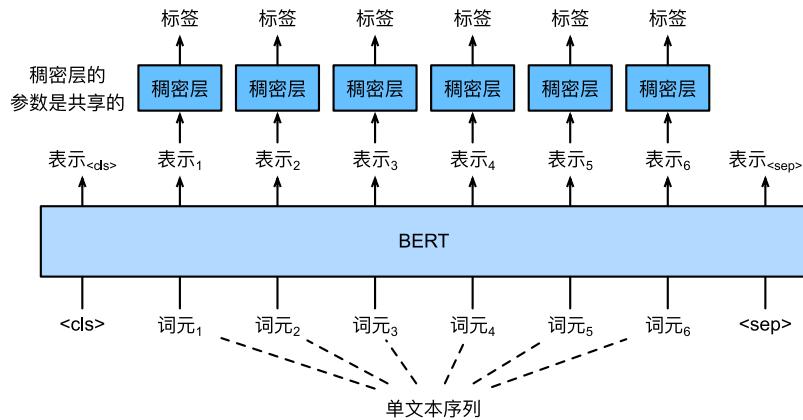


图15.6.3: 文本标记应用的BERT微调，如词性标记。假设输入的单个文本有六个词元。

图15.6.3中说明了文本标记应用的BERT微调。与 图15.6.1相比，唯一的区别在于，在文本标注中，输入文本的每个词元的BERT表示被送到相同的额外全连接层中，以输出词元的标签，例如词性标签。

15.6.4 问答

作为另一个词元级应用，问答反映阅读理解能力。例如，斯坦福问答数据集（Stanford Question Answering Dataset, SQuAD v1.1）由阅读段落和问题组成，其中每个问题的答案只是段落中的一段文本（文本片段）(Rajpurkar *et al.*, 2016)。举个例子，考虑一段话：“Some experts report that a mask’s efficacy is inconclusive. However, mask makers insist that their products, such as N95 respirator masks, can guard against the virus.”（“一些专家报告说面罩的功效是不确定的。然而，口罩制造商坚持他们的产品，如N95口罩，可以预防病毒。”）还有一个问题“Who say that N95 respirator masks can guard against the virus?”（“谁说N95口罩可以预防病毒？”）。答案应该是文章中的文本片段“mask makers”（“口罩制造商”）。因此，SQuAD v1.1的目标是在给定问题和段落的情况下预测段落中文本片段的开始和结束。

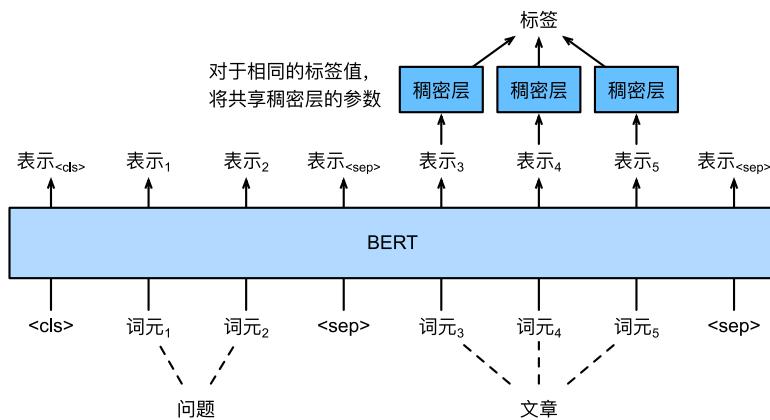


图15.6.4: 对问答进行BERT微调（假设输入文本对分别有两个和三个词元）

为了微调BERT进行问答，在BERT的输入中，将问题和段落分别作为第一个和第二个文本序列。为了预测文

本片段开始的位置，相同的额外的全连接层将把来自位置 i 的任何词元的BERT表示转换成标量分数 s_i 。文章中所有词元的分数还通过softmax转换成概率分布，从而为文章中的每个词元位置 i 分配作为文本片段开始的概率 p_i 。预测文本片段的结束与上面相同，只是其额外的全连接层中的参数与用于预测开始位置的参数无关。当预测结束时，位置 j 的词元由相同的全连接层转换成标量分数 e_j 。[图15.6.4](#)描述了用于问答的微调BERT。

对于问答，监督学习的训练目标就像最大化真实值的开始和结束位置的对数似然一样简单。当预测片段时，我们可以计算从位置 i 到位置 j 的有效片段的分数 $s_i + e_j$ ($i \leq j$)，并输出分数最高的跨度。

小结

- 对于序列级和词元级自然语言处理应用，BERT只需要最小的架构改变（额外的全连接层），如单个文本分类（例如，情感分析和测试语言可接受性）、文本对分类或回归（例如，自然语言推断和语义文本相似性）、文本标记（例如，词性标记）和问答。
- 在下游应用的监督学习期间，额外层的参数是从零开始学习的，而预训练BERT模型中的所有参数都是微调的。

练习

1. 让我们为新闻文章设计一个搜索引擎算法。当系统接收到查询（例如，“冠状病毒爆发期间的石油行业”）时，它应该返回与该查询最相关的新闻文章的排序列表。假设我们有一个巨大的新闻文章池和大量的查询。为了简化问题，假设为每个查询标记了最相关的文章。如何在算法设计中应用负采样（见[14.2.1节](#)）和BERT？
2. 我们如何利用BERT来训练语言模型？
3. 我们能在机器翻译中利用BERT吗？

Discussions²⁰⁷

15.7 自然语言推断：微调BERT

在本章的前面几节中，我们已经为SNLI数据集（[15.4节](#)）上的自然语言推断任务设计了一个基于注意力的结构（[15.5节](#)）。现在，我们通过微调BERT来重新审视这项任务。正如在[15.6节](#)中讨论的那样，自然语言推断是一个序列级别的文本对分类问题，而微调BERT只需要一个额外的基于多层感知机的架构，如[图15.7.1](#)中所示。

²⁰⁷ <https://discuss.d2l.ai/t/5729>

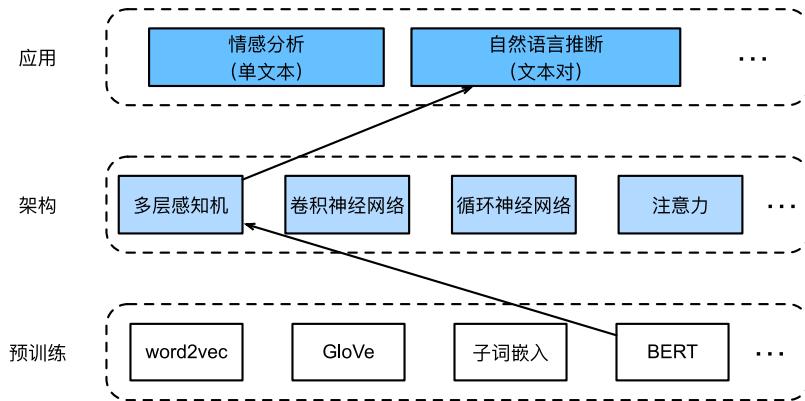


图15.7.1: 将预训练BERT提供给基于多层感知机的自然语言推断架构

本节将下载一个预训练好的小版本的BERT，然后对其进行微调，以便在SNLI数据集上进行自然语言推断。

```
import json
import multiprocessing
import os
import torch
from torch import nn
from d2l import torch as d2l
```

15.7.1 加载预训练的BERT

我们已经在 14.9 节 和 14.10 节 WikiText-2 数据集上预训练 BERT (请注意，原始的 BERT 模型是在更大的语料库上预训练的)。正如在 14.10 节 中所讨论的，原始的 BERT 模型有数以亿计的参数。在下面，我们提供了两个版本的预训练的 BERT：“bert.base” 与原始的 BERT 基础模型一样大，需要大量的计算资源才能进行微调，而 “bert.small” 是一个小版本，以便于演示。

```
d2l.DATA_HUB['bert.base'] = (d2l.DATA_URL + 'bert.base.torch.zip',
                             '225d66f04cae318b841a13d32af3acc165f253ac')
d2l.DATA_HUB['bert.small'] = (d2l.DATA_URL + 'bert.small.torch.zip',
                            'c72329e68a732bef0452e4b96a1c341c8910f81f')
```

两个预训练好的 BERT 模型都包含一个定义词表的 “vocab.json” 文件和一个预训练参数的 “pre-trained.params” 文件。我们实现了以下 `load_pretrained_model` 函数来加载预先训练好的 BERT 参数。

```
def load_pretrained_model(pretrained_model, num_hiddens, ffn_num_hiddens,
                         num_heads, num_layers, dropout, max_len, devices):
    data_dir = d2l.download_extract(pretrained_model)
    # 定义空词表以加载预定义词表
```

(continues on next page)

(continued from previous page)

```
vocab = d2l.Vocab()
vocab.idx_to_token = json.load(open(os.path.join(data_dir,
    'vocab.json')))
vocab.token_to_idx = {token: idx for idx, token in enumerate(
    vocab.idx_to_token)}
bert = d2l.BERTModel(len(vocab), num_hiddens, norm_shape=[256],
    ffn_num_input=256, ffn_num_hiddens=ffn_num_hiddens,
    num_heads=4, num_layers=2, dropout=0.2,
    max_len=max_len, key_size=256, query_size=256,
    value_size=256, hid_in_features=256,
    mlm_in_features=256, nsp_in_features=256)
# 加载预训练BERT参数
bert.load_state_dict(torch.load(os.path.join(data_dir,
    'pretrained.params')))
return bert, vocab
```

为了便于在大多数机器上演示，我们将在本节中加载和微调经过预训练BERT的小版本（“bert.small”）。在练习中，我们将展示如何微调大得多的“bert.base”以显著提高测试精度。

```
devices = d2l.try_all_gpus()
bert, vocab = load_pretrained_model(
    'bert.small', num_hiddens=256, ffn_num_hiddens=512, num_heads=4,
    num_layers=2, dropout=0.1, max_len=512, devices=devices)
```

```
Downloading ../data/bert.small.torch.zip from http://d2l-data.s3-accelerate.amazonaws.com/bert.small.
→torch.zip...
```

15.7.2 微调BERT的数据集

对于SNLI数据集的下游任务自然语言推断，我们定义了一个定制的数据集类SNLIBERTDataset。在每个样本中，前提和假设形成一对文本序列，并被打包成一个BERT输入序列，如图15.6.2所示。回想14.8.4节，片段索引用于区分BERT输入序列中的前提和假设。利用预定义的BERT输入序列的最大长度(max_len)，持续移除输入文本对中较长文本的最后一个标记，直到满足max_len。为了加速生成用于微调BERT的SNLI数据集，我们使用4个工作进程并行生成训练或测试样本。

```
class SNLIBERTDataset(torch.utils.data.Dataset):
    def __init__(self, dataset, max_len, vocab=None):
        all_premise_hypothesis_tokens = [
            p_tokens, h_tokens] for p_tokens, h_tokens in zip(
                *[d2l.tokenize([s.lower() for s in sentences])]
```

(continues on next page)

(continued from previous page)

```
for sentences in dataset[:2])]

self.labels = torch.tensor(dataset[2])
self.vocab = vocab
self.max_len = max_len
(self.all_token_ids, self.all_segments,
 self.valid_lens) = self._preprocess(all_premise_hypothesis_tokens)
print('read ' + str(len(self.all_token_ids)) + ' examples')

def _preprocess(self, all_premise_hypothesis_tokens):
    pool = multiprocessing.Pool(4) # 使用4个进程
    out = pool.map(self._mp_worker, all_premise_hypothesis_tokens)
    all_token_ids = [
        token_ids for token_ids, segments, valid_len in out]
    all_segments = [segments for token_ids, segments, valid_len in out]
    valid_lens = [valid_len for token_ids, segments, valid_len in out]
    return (torch.tensor(all_token_ids, dtype=torch.long),
            torch.tensor(all_segments, dtype=torch.long),
            torch.tensor(valid_lens))

def _mp_worker(self, premise_hypothesis_tokens):
    p_tokens, h_tokens = premise_hypothesis_tokens
    self._truncate_pair_of_tokens(p_tokens, h_tokens)
    tokens, segments = d2l.get_tokens_and_segments(p_tokens, h_tokens)
    token_ids = self.vocab[tokens] + [self.vocab['<pad>']] \
        * (self.max_len - len(tokens))
    segments = segments + [0] * (self.max_len - len(segments))
    valid_len = len(tokens)
    return token_ids, segments, valid_len

def _truncate_pair_of_tokens(self, p_tokens, h_tokens):
    # 为BERT输入中的'<CLS>'、'<SEP>'和'<SEP>'词元保留位置
    while len(p_tokens) + len(h_tokens) > self.max_len - 3:
        if len(p_tokens) > len(h_tokens):
            p_tokens.pop()
        else:
            h_tokens.pop()

def __getitem__(self, idx):
    return (self.all_token_ids[idx], self.all_segments[idx],
            self.valid_lens[idx], self.labels[idx])

def __len__(self):
```

(continues on next page)

(continued from previous page)

```
    return len(self.all_token_ids)
```

下载完SNLI数据集后，我们通过实例化`SNLIBERTDataset`类来生成训练和测试样本。这些样本将在自然语言推断的训练和测试期间进行小批量读取。

```
# 如果出现显存不足错误，请减少“batch_size”。在原始的BERT模型中，max_len=512
batch_size, max_len, num_workers = 512, 128, d2l.get_dataloader_workers()
data_dir = d2l.download_extract('SNLI')
train_set = SNLIBERTDataset(d2l.read_snli(data_dir, True), max_len, vocab)
test_set = SNLIBERTDataset(d2l.read_snli(data_dir, False), max_len, vocab)
train_iter = torch.utils.data.DataLoader(train_set, batch_size, shuffle=True,
                                         num_workers=num_workers)
test_iter = torch.utils.data.DataLoader(test_set, batch_size,
                                         num_workers=num_workers)
```

```
read 549367 examples
read 9824 examples
```

15.7.3 微调BERT

如图15.6.2所示，用于自然语言推断的微调BERT只需要一个额外的多层感知机，该多层感知机由两个全连接层组成（请参见下面`BERTClassifier`类中的`self.hidden`和`self.output`）。这个多层感知机将特殊的“`<cls>`”词元的BERT表示进行了转换，该词元同时编码前提和假设的信息为自然语言推断的三个输出：蕴涵、矛盾和中性。

```
class BERTClassifier(nn.Module):
    def __init__(self, bert):
        super(BERTClassifier, self).__init__()
        self.encoder = bert.encoder
        self.hidden = bert.hidden
        self.output = nn.Linear(256, 3)

    def forward(self, inputs):
        tokens_X, segments_X, valid_lens_x = inputs
        encoded_X = self.encoder(tokens_X, segments_X, valid_lens_x)
        return self.output(self.hidden(encoded_X[:, 0, :]))
```

在下文中，预训练的BERT模型`bert`被送到用于下游应用的`BERTClassifier`实例`net`中。在BERT微调的常见实现中，只有额外的多层感知机（`net.output`）的输出层的参数将从零开始学习。预训练BERT编码器（`net.encoder`）和额外的多层感知机的隐藏层（`net.hidden`）的所有参数都将进行微调。

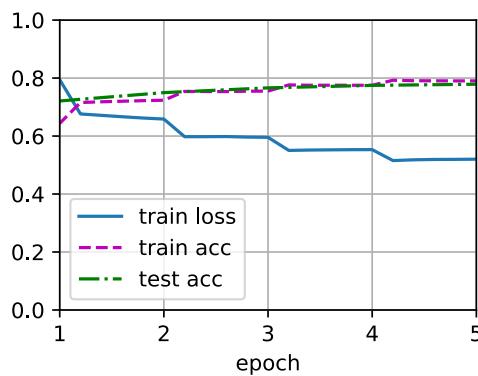
```
net = BERTClassifier(bert)
```

回想一下，在14.8节中，MaskLM类和NextSentencePred类在其使用的多层感知机中都有一些参数。这些参数是预训练BERT模型bert中参数的一部分，因此是net中的参数的一部分。然而，这些参数仅用于计算预训练过程中的遮蔽语言模型损失和下一句预测损失。这两个损失函数与微调下游应用无关，因此当BERT微调时，MaskLM和NextSentencePred中采用的多层感知机的参数不会更新（陈旧的，staled）。

为了允许具有陈旧梯度的参数，标志`ignore_stale_grad=True`在`step`函数`d2l.train_batch_ch13`中被设置。我们通过该函数使用SNLI的训练集（`train_iter`）和测试集（`test_iter`）对`net`模型进行训练和评估。由于计算资源有限，训练和测试精度可以进一步提高：我们把对它的讨论留在练习中。

```
lr, num_epochs = 1e-4, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction='none')
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
    devices)
```

```
loss 0.520, train acc 0.790, test acc 0.779
10442.5 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



小结

- 我们可以针对下游应用对预训练的BERT模型进行微调，例如在SNLI数据集上进行自然语言推断。
- 在微调过程中，BERT模型成为下游应用模型的一部分。仅与训练前损失相关的参数在微调期间不会更新。

练习

1. 如果您的计算资源允许, 请微调一个更大的预训练BERT模型, 该模型与原始的BERT基础模型一样大。修改load_pretrained_model函数中的参数设置: 将“bert.small”替换为“bert.base”, 将num_hiddens=256、ffn_num_hiddens=512、num_heads=4和num_layers=2的值分别增加到768、3072、12和12。通过增加微调迭代轮数 (可能还会调优其他超参数), 你可以获得高于0.86的测试精度吗?
2. 如何根据一对序列的长度比值截断它们? 将此对截断方法与SNLIBERTDataset类中使用的方法进行比较。它们的利弊是什么?

Discussions²⁰⁸

²⁰⁸ <https://discuss.d2l.ai/t/5718>

附录：深度学习工具

为了充分利用《动手学深度学习》，本书将在本附录中介绍不同工具，例如如何运行这本交互式开源书籍和为本书做贡献。

16.1 使用Jupyter Notebook

本节介绍如何使用Jupyter Notebook编辑和运行本书各章中的代码。确保你已按照安装 (page 9)中的说明安装了Jupyter并下载了代码。如果你想了解更多关于Jupyter的信息，请参阅其[文档²⁰⁹](#)中的优秀教程。

16.1.1 在本地编辑和运行代码

假设本书代码的本地路径为xx/yy/d2l-en/。使用shell将目录更改为此路径 (`cd xx/yy/d2l-en`) 并运行命令`jupyter notebook`。如果浏览器未自动打开，请打开<http://localhost:8888>。此时你将看到Jupyter的界面以及包含本书代码的所有文件夹，如图16.1.1所示

²⁰⁹ <https://jupyter.readthedocs.io/en/latest/>

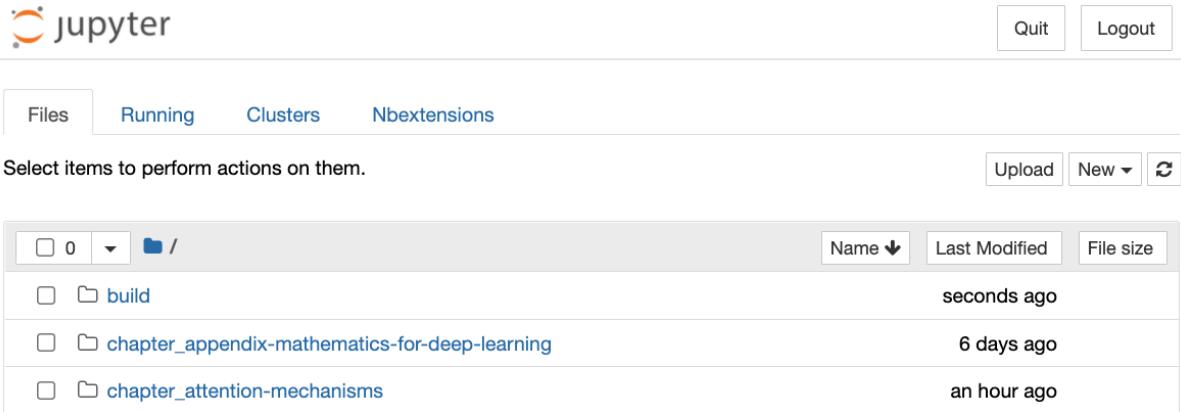


图16.1.1: 包含本书代码的文件夹

你可以通过单击网页上显示的文件夹来访问notebook文件。它们通常有后缀“.ipynb”。为了简洁起见，我们创建了一个临时的“test.ipynb”文件。单击后显示的内容如 图16.1.2所示。此notebook包括一个标记单元格和一个代码单元格。标记单元格中的内容包括“This Is a Title”和“This is text.”。代码单元包含两行Python代码。

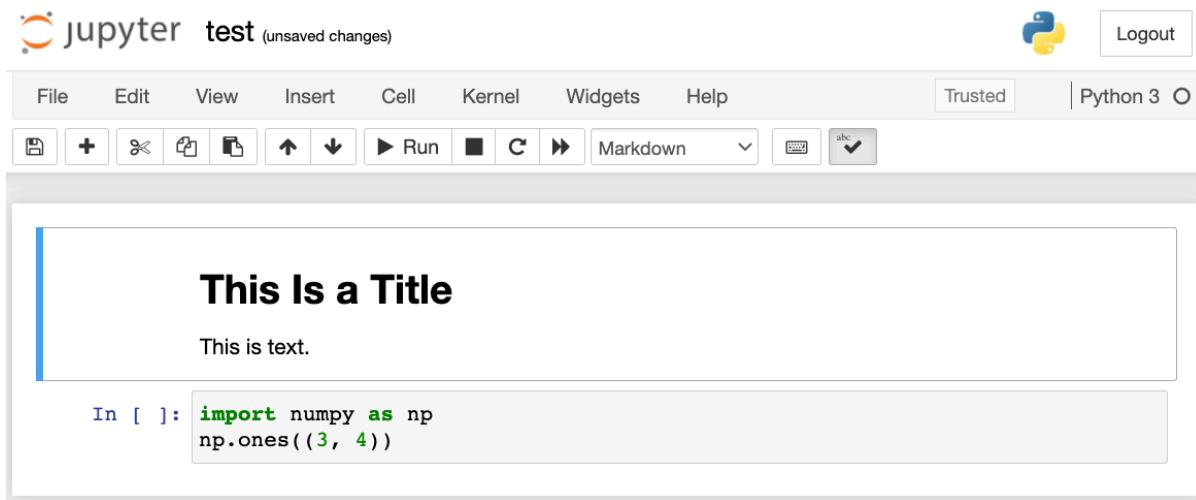


图16.1.2: “test.ipynb”文件中的markdown和代码块

双击标记单元格以进入编辑模式。在单元格末尾添加一个新的文本字符串“Hello world.”，如 图16.1.3所示。

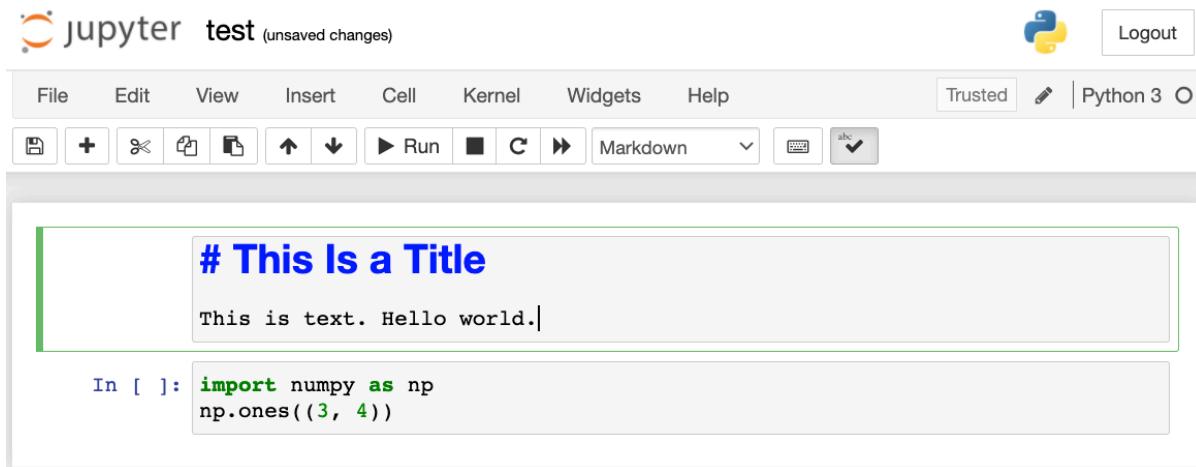


图16.1.3: 编辑markdown单元格

如图16.1.4所示, 单击菜单栏中的“Cell”→“Run Cells”以运行编辑后的单元格。

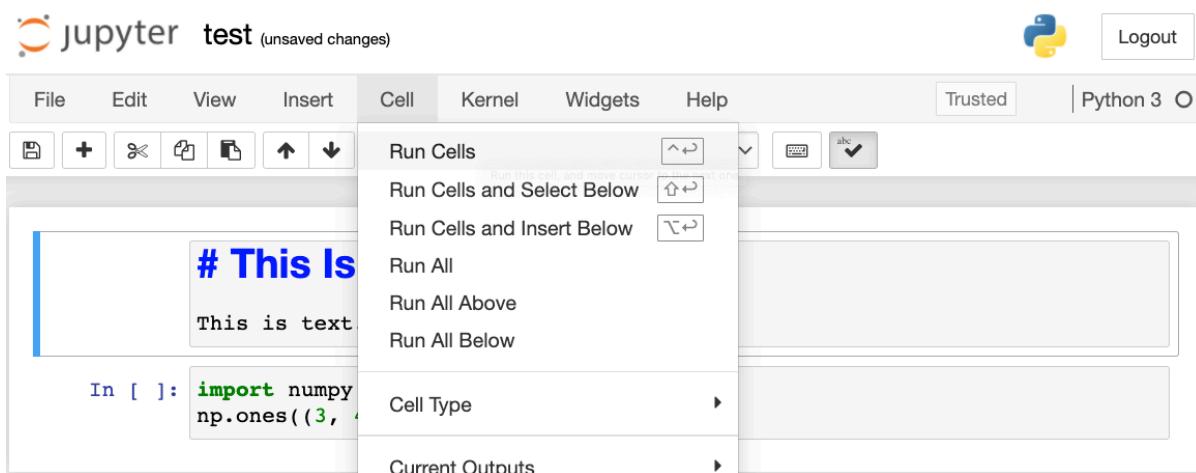


图16.1.4: 运行单元格

运行后, markdown单元格如图16.1.5所示。

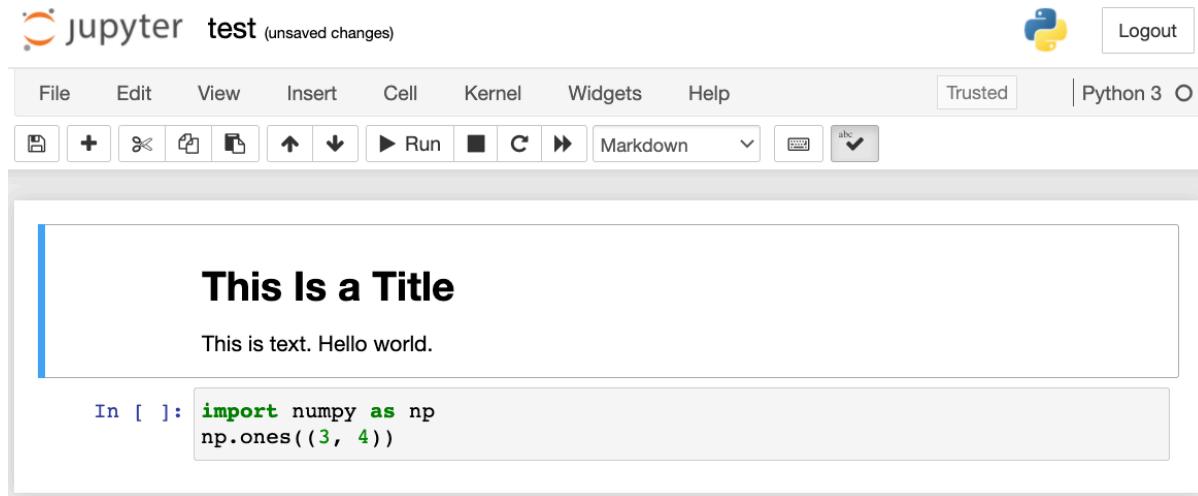


图16.1.5: 编辑后的markdown单元格

接下来, 单击代码单元。将最后一行代码后的元素乘以2, 如 图16.1.6所示。

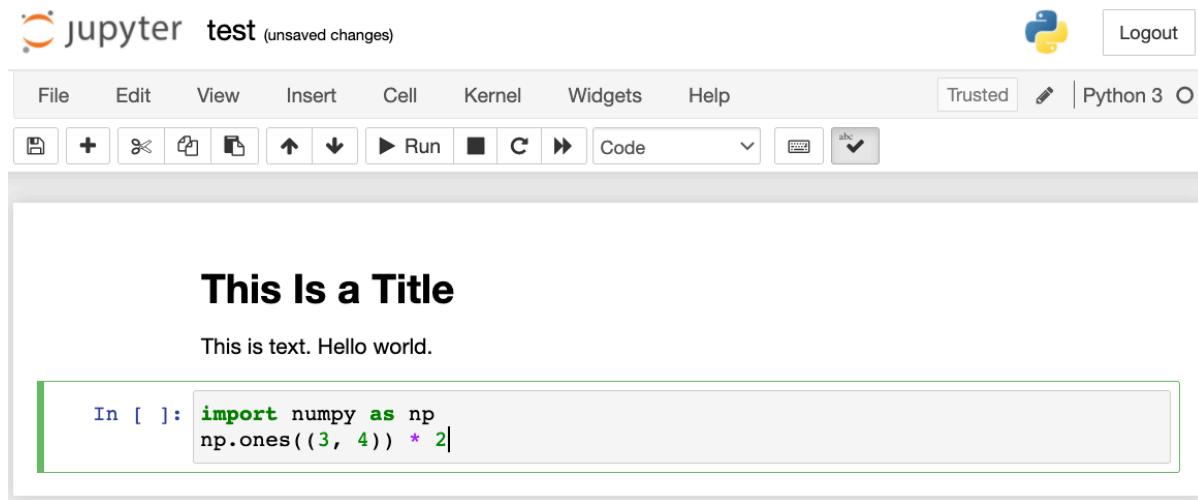


图16.1.6: 编辑代码单元格

你还可以使用快捷键(默认情况下为Ctrl+Enter)运行单元格, 并从图16.1.7获取输出结果。

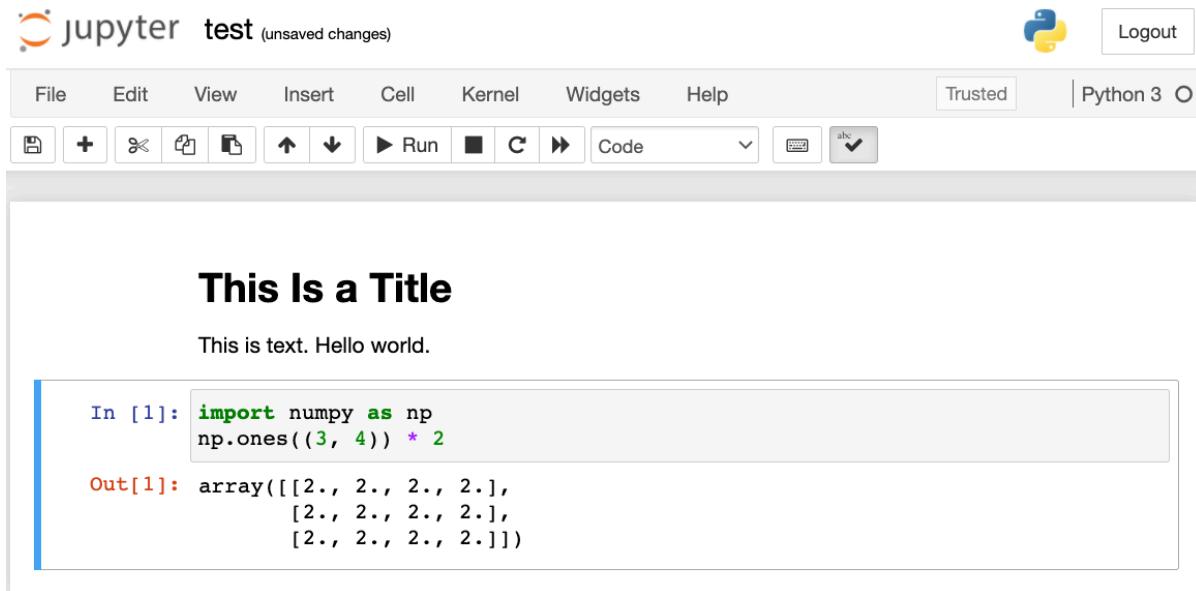


图16.1.7: 运行代码单元格以获得输出

当一个notebook包含更多单元格时，我们可以单击菜单栏中的“Kernel” → “Restart & Run All” 来运行整个notebook中的所有单元格。通过单击菜单栏中的“Help” → “Edit Keyboard Shortcuts”，可以根据你的首选项编辑快捷键。

16.1.2 高级选项

除了本地编辑，还有两件事非常重要：以markdown格式编辑notebook和远程运行Jupyter。当我们想要在更快的服务器上运行代码时，后者很重要。前者很重要，因为Jupyter原生的ipynb格式存储了大量辅助数据，这些数据实际上并不特定于notebook中的内容，主要与代码的运行方式和运行位置有关。这让git感到困惑，并且使得合并贡献非常困难。幸运的是，还有另一种选择——在markdown中进行本地编辑。

Jupyter中的Markdown文件

如果你希望对本书的内容有所贡献，则需要在GitHub上修改源文件（md文件，而不是ipynb文件）。使用notedown插件，我们可以直接在Jupyter中修改md格式的notebook。

首先，安装notedown插件，运行Jupyter Notebook并加载插件：

```
pip install d2l-notedown # 你可能需要卸载原始notedown  
jupyter notebook --NotebookApp.contents_manager_class='notedown.NotedownContentsManager'
```

要在运行Jupyter Notebook时默认打开notedown插件，请执行以下操作：首先，生成一个Jupyter Notebook配置文件（如果已经生成了，可以跳过此步骤）。

```
jupyter notebook --generate-config
```

然后，在Jupyter Notebook配置文件的末尾添加以下行（对于Linux/macOS，通常位于`~/.jupyter/jupyter_notebook_config.py`）：

```
c.NotebookApp.contents_manager_class = 'notedown.NotedownContentsManager'
```

在这之后，你只需要运行`jupyter notebook`命令就可以默认打开notedown插件。

在远程服务器上运行Jupyter Notebook

有时，你可能希望在远程服务器上运行Jupyter Notebook，并通过本地计算机上的浏览器访问它。如果本地计算机上安装了Linux或MacOS（Windows也可以通过PuTTY等第三方软件支持此功能），则可以使用端口转发：

```
ssh myserver -L 8888:localhost:8888
```

以上是远程服务器`myserver`的地址。然后我们可以使用`http://localhost:8888`访问运行Jupyter Notebook的远程服务器`myserver`。下一节将详细介绍如何在AWS实例上运行Jupyter Notebook。

执行时间

我们可以使用`ExecuteTime`插件来计算Jupyter Notebook中每个代码单元的执行时间。使用以下命令安装插件：

```
pip install jupyter_contrib_nbextensions
jupyter contrib nbextension install --user
jupyter nbextension enable execute_time/ExecuteTime
```

小结

- 使用Jupyter Notebook工具，我们可以编辑、运行和为本书做贡献。
- 使用端口转发在远程服务器上运行Jupyter Notebook。

练习

1. 在本地计算机上使用Jupyter Notebook编辑并运行本书中的代码。
2. 使用Jupyter Notebook通过端口转发来远程编辑和运行本书中的代码。
3. 对于两个方矩阵，测量 $\mathbf{A}^\top \mathbf{B}$ 与 \mathbf{AB} 在 $\mathbb{R}^{1024 \times 1024}$ 中的运行时间。哪一个更快？

Discussions²¹⁰

16.2 使用Amazon SageMaker

深度学习程序可能需要很多计算资源，这很容易超出你的本地计算机所能提供的范围。云计算服务允许你使用功能更强大的计算机更轻松地运行本书的GPU密集型代码。本节将介绍如何使用Amazon SageMaker运行本书的代码。

16.2.1 注册

首先，我们需要在注册一个帐户<https://aws.amazon.com/>。为了增加安全性，鼓励使用双因素身份验证。设置详细的计费和支出警报也是一个好主意，以避免任何意外，例如，当忘记停止运行实例时。登录AWS帐户后，转到[console²¹¹](#)并搜索“Amazon SageMaker”（参见 [图16.2.1](#)），然后单击它打开SageMaker面板。

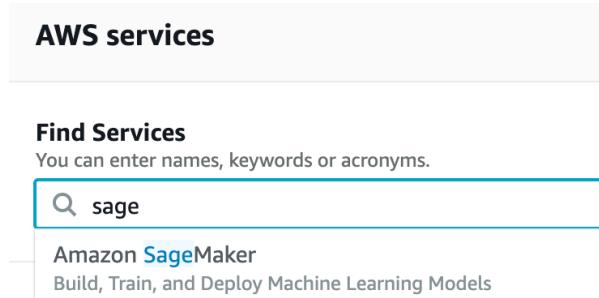


图16.2.1: 搜索并打开SageMaker面板

16.2.2 创建SageMaker实例

接下来，让我们创建一个notebook实例，如 [图16.2.2](#)所示。

²¹⁰ <https://discuss.d2l.ai/t/5731>

²¹¹ <http://console.aws.amazon.com/>

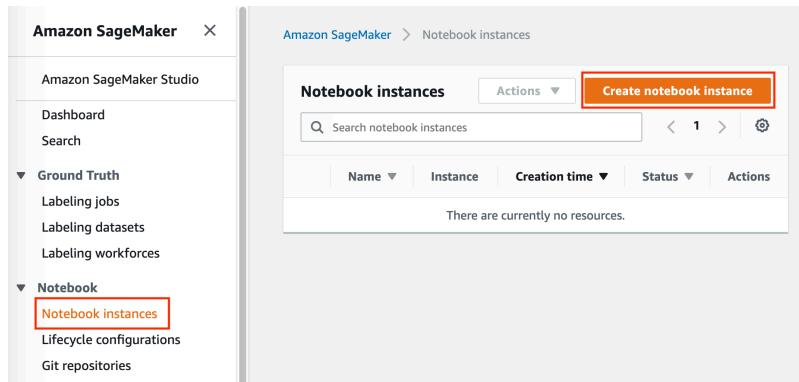


图16.2.2: 创建一个SageMaker实例

SageMaker提供多个具有不同计算能力和价格的实例类型²¹²。创建notebook实例时，可以指定其名称和类型。在图16.2.3中，我们选择ml.p3.2xlarge：使用一个Tesla V100 GPU和一个8核CPU，这个实例的性能足够本书的大部分内容使用。

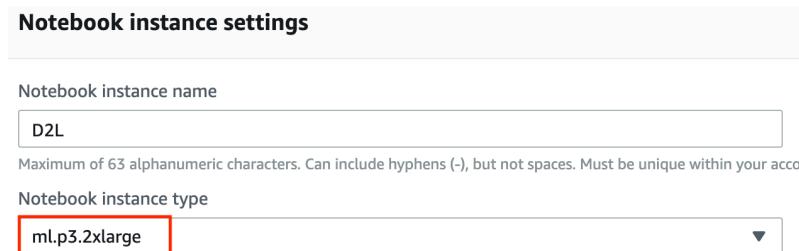


图16.2.3: 选择实例类型

用于与SageMaker一起运行的ipynb格式的整本书可从<https://github.com/d2l-ai/d2l-pytorch-sagemaker>获得。我们可以指定此GitHub存储库URL（图16.2.4），以允许SageMaker在创建实例时克隆它。

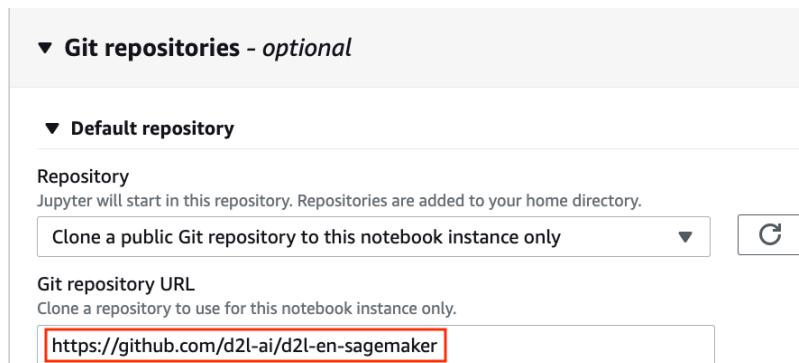


图16.2.4: 指定GitHub存储库

²¹² <https://aws.amazon.com/sagemaker/pricing/instance-types/>

16.2.3 运行和停止实例

创建实例可能需要几分钟的时间。当实例准备就绪时，单击它旁边的“Open Jupyter”链接（图16.2.5），以便你可以在此实例上编辑并运行本书的所有Jupyter Notebook（类似于 16.1节中的步骤）。



图16.2.5: 在创建的SageMaker实例上打开Jupyter

完成工作后，不要忘记停止实例以避免进一步收费（图16.2.6）。

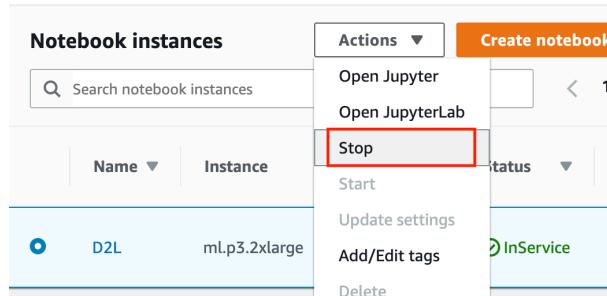


图16.2.6: 停止SageMaker实例

16.2.4 更新Notebook

这本开源书的notebook将定期在GitHub上的d2l-ai/d2l-pytorch-sagemaker²¹³存储库中更新。要更新至最新版本，你可以在SageMaker实例（图16.2.7）上打开终端。

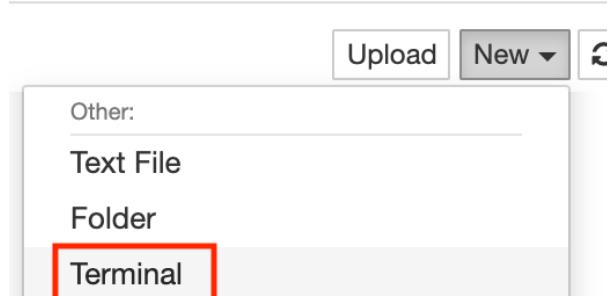


图16.2.7: 在SageMaker实例上打开终端

你可能希望在从远程存储库提取更新之前提交本地更改。否则，只需在终端中使用以下命令放弃所有本地更改：

²¹³ <https://github.com/d2l-ai/d2l-pytorch-sagemaker>

```
cd SageMaker/d2l-pytorch-sagemaker/  
git reset --hard  
git pull
```

小结

- 我们可以使用Amazon SageMaker创建一个GPU的notebook实例来运行本书的密集型代码。
- 我们可以通过Amazon SageMaker实例上的终端更新notebooks。

练习

1. 使用Amazon SageMaker编辑并运行任何需要GPU的部分。
2. 打开终端以访问保存本书所有notebooks的本地目录。

Discussions²¹⁴

16.3 使用Amazon EC2实例

本节将展示如何在原始Linux机器上安装所有库。回想一下，16.2节讨论了如何使用Amazon SageMaker，而在云上自己构建实例的成本更低。本演示包括三个步骤。

1. 从AWS EC2请求GPU Linux实例。
2. 安装CUDA（或使用预装CUDA的Amazon机器映像）。
3. 安装深度学习框架和其他库以运行本书的代码。

此过程也适用于其他实例（和其他云），尽管需要一些细微的修改。在继续操作之前，你需要创建一个AWS帐户，有关更多详细信息，请参阅16.2节。

16.3.1 创建和运行EC2实例

登录到你的aws账户后，单击“EC2”（在图16.3.1中用红色方框标记）进入EC2面板。

²¹⁴ <https://discuss.d2l.ai/t/5732>

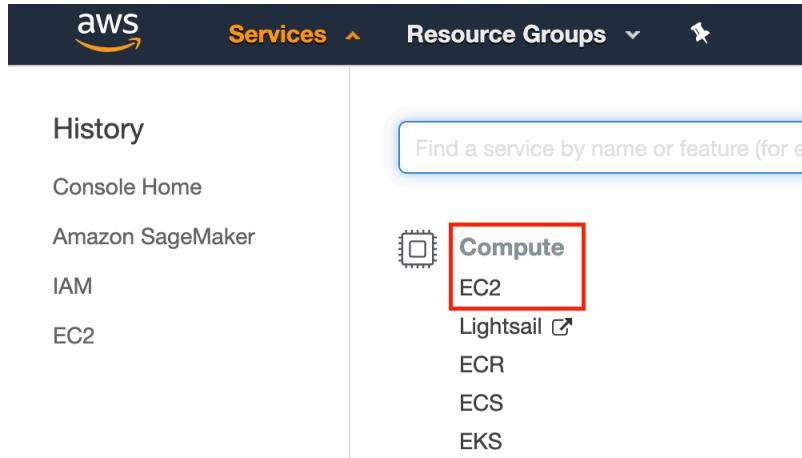


图16.3.1: 打开EC2控制台

图16.3.2显示EC2面板，敏感帐户信息变为灰色。

A screenshot of the AWS EC2 Dashboard. The top navigation bar includes the AWS logo, 'Services' dropdown, 'Resource Groups' dropdown, and a bell icon. A red box highlights the 'Oregon' region selection in the top right. The left sidebar has a 'Limits' section with a red box around it. The main content area is titled 'Resources' and shows a list of EC2 resources: 'Running Instances', 'Dedicated Hosts', 'Volumes', 'Key Pairs', 'Placement Groups', 'Elastic IPs', 'Snapshots', 'Load Balancers', and 'Security Groups'. Below this is a 'Create Instance' section with a 'Launch Instance' button highlighted with a red box. To the right, there's an 'Account Attributes' section with links to 'Supported Platforms', 'VPC', 'Default VPC', and 'Resource ID length management'. Another section titled 'Additional Information' includes links to 'Getting Started Guide', 'Documentation', 'All EC2 Resources', 'Forums', 'Pricing', and 'Contact Us'. A note at the bottom states: 'Note: Your instances will launch in the US East (N. Virginia) region'.

图16.3.2: EC2面板

预置位置

选择附近的数据中心以降低延迟，例如“Oregon”（俄勒冈）(图16.3.2右上角的红色方框)。如果你位于中国，你可以选择附近的亚太地区，例如首尔或东京。请注意，某些数据中心可能没有GPU实例。

增加限制

在选择实例之前，请点击 图16.3.2所示左侧栏中的“Limits”（限制）标签查看是否有数量限制。图16.3.3显示了此类限制的一个例子。账号目前无法按地域打开p2.xlarge实例。如果你需要打开一个或多个实例，请点击“Request limit increase”（请求增加限制）链接，申请更高的实例配额。一般来说，需要一个工作日的时间来处理申请。

The screenshot shows the AWS EC2 Dashboard with the 'Limits' tab selected in the sidebar. The main pane displays a table of instance types and their current running counts, along with a 'Request limit increase' link for each type where the count is less than the limit.

Instance Type	Running On-Demand Instances	Action
m5d.metal	0	Request limit increase
m5d.xlarge	2	Request limit increase
p2.16xlarge	0	Request limit increase
p2.8xlarge	0	Request limit increase
p2.xlarge	0	Request limit increase
p3.16xlarge	0	Request limit increase
p3.2xlarge	0	Request limit increase
p3.8xlarge	0	Request limit increase
p3dn.24xlarge	0	Request limit increase

图16.3.3: 实例数量限制

启动实例

接下来，单击 图16.3.2中红框标记的“Launch Instance”（启动实例）按钮，启动你的实例。

我们首先选择一个合适的Amazon机器映像（Amazon Machine Image, AMI）。在搜索框中输入“ubuntu”（图16.3.4中的红色框标记）。

The screenshot shows the 'Step 1: Choose an Amazon Machine Image (AMI)' step of the instance creation wizard. A search bar at the top contains the text 'Ubuntu'. The results list shows two entries: 'Ubuntu Server 18.04 LTS (HVM), SSD Volume Type' and 'Ubuntu Server 16.04 LTS (HVM), SSD Volume Type'. Each entry includes a 'Select' button, which is highlighted with a red box. To the right of the entries, there are two radio buttons for '64-bit (x86)' and '64-bit (Arm)'.

图16.3.4: 选择一个AMI

EC2提供了许多不同的实例配置可供选择。对初学者来说，这有时会让人感到困惑。表16.3.1列出了不同合适的计算机。

表16.3.1: 不同的EC2实例类型

Name	GPU	Notes
g2	Grid K520	过时的
p2	Kepler K80	旧的GPU但Spot实例通常很便宜
g3	Maxwell M60	好的平衡
p3	Volta V100	FP16的高性能
g4	Turing T4	FP16/INT8推理优化

所有这些服务器都有多种类型,显示了使用的GPU数量。例如,p2.xlarge有1个GPU,而p2.16xlarge有16个GPU和更多内存。有关更多详细信息,请参阅Amazon EC2 文档²¹⁵。



图16.3.5: 选择一个实例

注意,你应该使用支持GPU的实例以及合适的驱动程序和支持GPU的深度学习框架。否则,你将感受不到使用GPU的任何好处。

到目前为止,我们已经完成了启动EC2实例的七个步骤中的前两个步骤,如图16.3.6顶部所示。在本例中,我们保留“3. Configure Instance”(3. 配置实例)、“5. Add Tags”(5. 添加标签)和“6. Configure Security Group”(6. 配置安全组)步骤的默认配置。点击“4. Add Storage”并将默认硬盘大小增加到64GB(图16.3.6中的红色框标记)。请注意,CUDA本身已经占用了4GB空间。

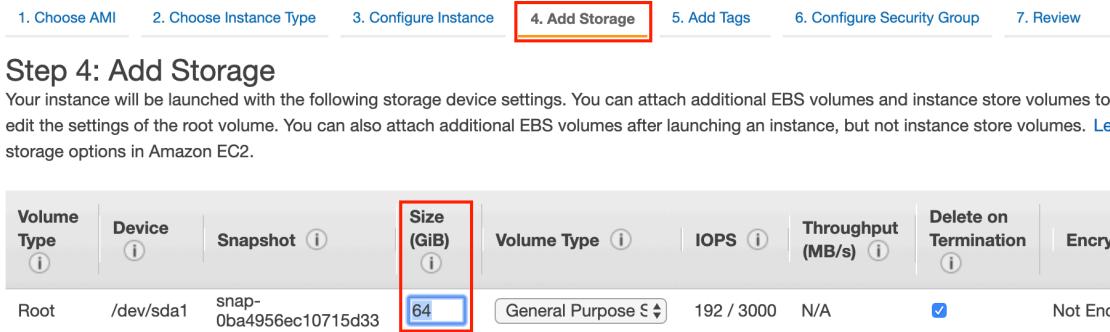


图16.3.6: 修改硬盘大小

最后,进入“7. Review”(7. 查看),点击“Launch”(启动),即可启动配置好的实例。系统现在将提示你选择用于访问实例的密钥对。如果你没有密钥对,请在图16.3.7的第一个下拉菜单中选择“Create a new key

²¹⁵ <https://aws.amazon.com/ec2/instance-types/>

pair”（新建密钥对），即可生成密钥对。之后，你可以在此菜单中选择“Choose an existing key pair”（选择现有密钥对），然后选择之前生成的密钥对。单击“Launch Instances”（启动实例）即可启动创建的实例。

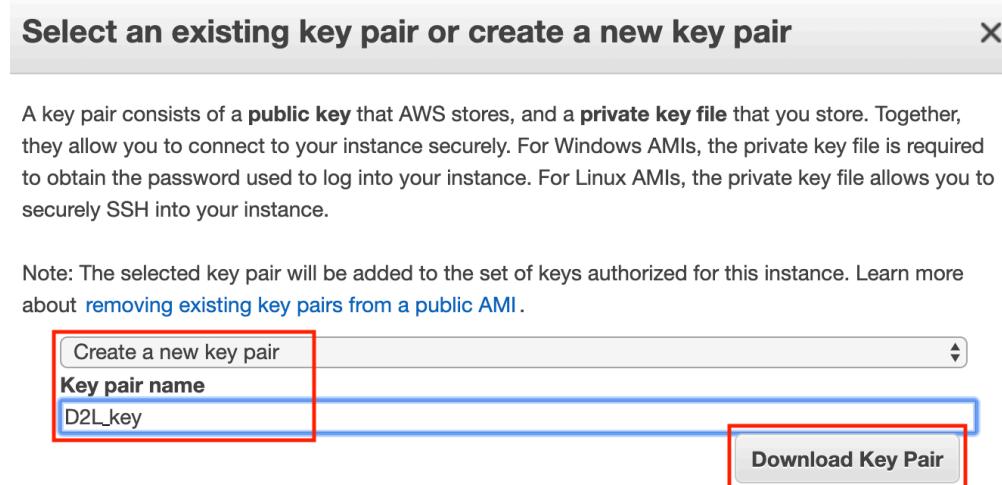


图16.3.7: 选择一个密钥对

如果生成了新密钥对，请确保下载密钥对并将其存储在安全位置。这是你通过SSH连接到服务器的唯一方式。单击图16.3.8中显示的实例ID可查看该实例的状态。

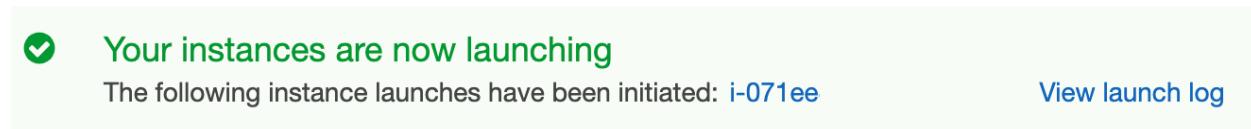


图16.3.8: 单击实例ID

连接到实例

如图16.3.9所示，实例状态变为绿色后，右键单击实例，选择Connect（连接）查看实例访问方式。

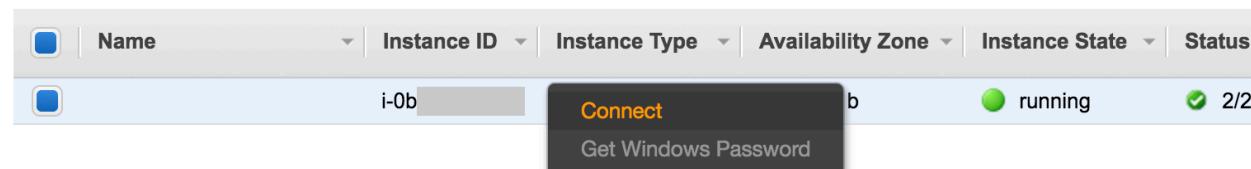


图16.3.9: 查看实例访问方法

如果这是一个新密钥，它必须是不可公开查看的，SSH才能工作。转到存储D2L_key.pem的文件夹，并执行以下命令以使密钥不可公开查看：

```
chmod 400 D2L_key.pem
```

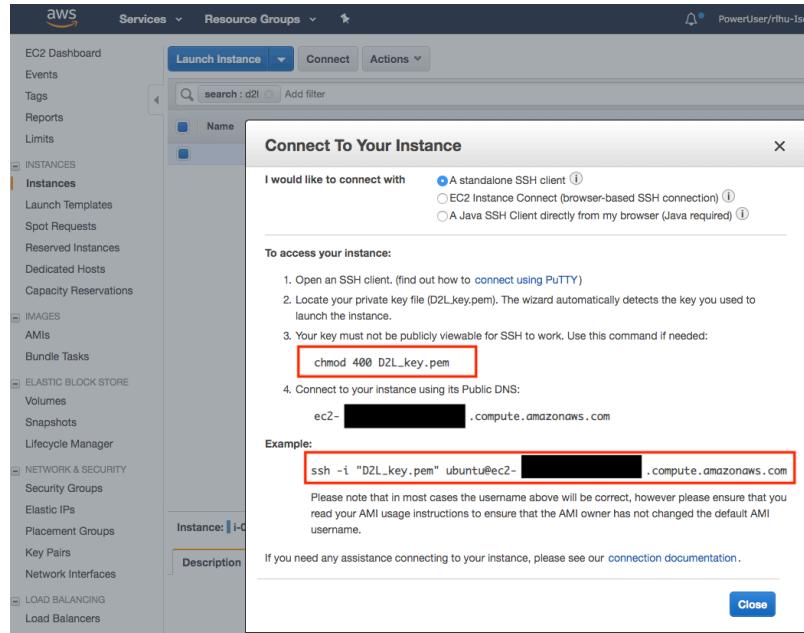


图16.3.10: 查看实例访问和启动方法

现在，复制 图16.3.10下方红色框中的ssh命令并粘贴到命令行：

```
ssh -i "D2L_key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.compute.amazonaws.com
```

当命令行提示“Are you sure you want to continue connecting (yes/no)”（“你确定要继续连接吗？(是/否)”）时，输入“yes”并按回车键登录实例。

你的服务器现在已就绪。

16.3.2 安装CUDA

在安装CUDA之前，请确保使用最新的驱动程序更新实例。

```
sudo apt-get update && sudo apt-get install -y build-essential git libgfortran3
```

我们在这里下载CUDA 10.1。访问NVIDIA的[官方存储库](https://developer.nvidia.com/cuda-toolkit-archive)²¹⁶以找到下载链接，如 图16.3.11中所示。

²¹⁶ <https://developer.nvidia.com/cuda-toolkit-archive>

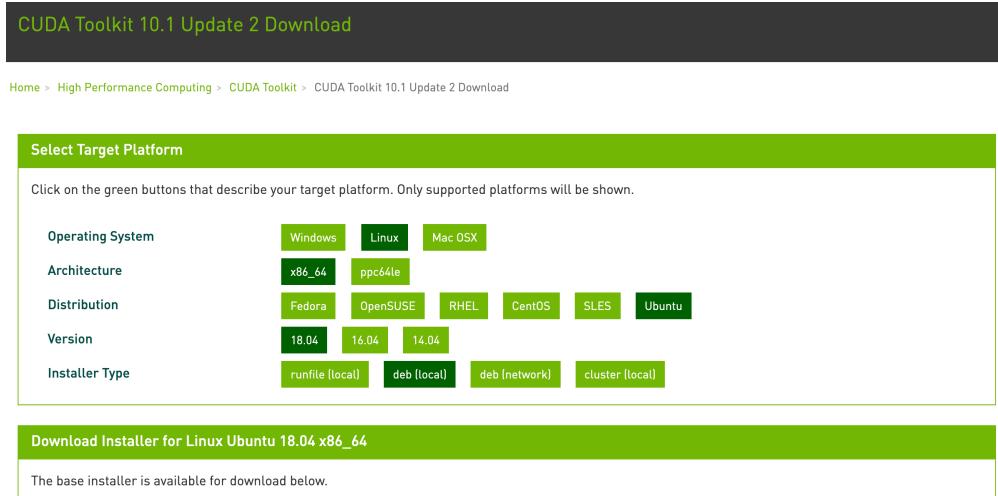


图16.3.11: 查找CUDA 10.1下载地址

将说明复制粘贴到终端上，以安装CUDA 10.1。

```
# 链接和文件名可能会发生更改，以NVIDIA的官方为准
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64/cuda-ubuntu1804.pin
sudo mv cuda-ubuntu1804.pin /etc/apt/preferences.d/cuda-repository-pin-600
wget http://developer.download.nvidia.com/compute/cuda/10.1/Prod/local_installers/cuda-repo-ubuntu1804-
→10-1-local-10.1.243-418.87.00_1.0-1_amd64.deb
sudo dpkg -i cuda-repo-ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.deb
sudo apt-key add /var/cuda-repo-10-1-local-10.1.243-418.87.00/7fa2af80.pub
sudo apt-get update
sudo apt-get -y install cuda
```

安装程序后，运行以下命令查看GPU：

```
nvidia-smi
```

最后，将CUDA添加到库路径以帮助其他库找到它。

```
echo "export LD_LIBRARY_PATH=\${LD_LIBRARY_PATH}:/usr/local/cuda/lib64" >> ~/.bashrc
```

16.3.3 安装库以运行代码

要运行本书的代码，只需在EC2实例上为linux用户执行[安装](#)(page 9)中的步骤，并使用以下提示在远程linux服务器上工作。

- 要在Miniconda安装页面下载bash脚本，请右击下载链接并选择“copy Link address”，然后执行wget [copied link address]。
- 运行~/miniconda3/bin/conda init，你可能需要执行source~/.bashrc，而不是关闭并重新打开当前shell。

16.3.4 远程运行Jupyter笔记本

要远程运行Jupyter笔记本，你需要使用SSH端口转发。毕竟，云中的服务器没有显示器或键盘。为此，请从你的台式机（或笔记本电脑）登录到你的服务器，如下所示：

```
# 此命令必须在本地命令行中运行
ssh -i "/path/to/key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.compute.amazonaws.com -L 8889:localhost:8888
```

接下来，转到EC2实例上本书下载的代码所在的位置，然后运行：

```
conda activate d2l
jupyter notebook
```

图16.3.12显示了运行Jupyter笔记本后可能的输出。最后一行是端口8888的URL。

```
(d2l) [REDACTED] $ jupyter notebook
[I 05:23:30.157 NotebookApp] Writing notebook server cookie secret to /home/ubuntu/.lo
[I 05:23:30.711 NotebookApp] Serving notebooks from local directory: /home/ubuntu/d2l-
[I 05:23:30.711 NotebookApp] Jupyter Notebook 6.4.4 is running at:
[I 05:23:30.711 NotebookApp] http://localhost:8888/?token=7ae1f41cbd5c6ca705c1ad9f5115
[I 05:23:30.711 NotebookApp] or http://127.0.0.1:8888/?token=7ae1f41cbd5c6ca705c1ad9f
[I 05:23:30.712 NotebookApp] Use Control-C to stop this server and shut down all kerne
[W 05:23:30.717 NotebookApp] No web browser found: could not locate runnable browser.
[C 05:23:30.717 NotebookApp]

To access the notebook, open this file in a browser:
  file:///home/ubuntu/.local/share/jupyter/runtime/nbserver-1615-open.html
Or copy and paste one of these URLs:
  http://localhost:8888/?token=7ae1f41cbd5c6ca705c1ad9f5115931bb8929817fd507ba3
```

图16.3.12：运行Jupyter Notebook后的输出（最后一行是端口8888的URL）

由于你使用端口转发到端口8889，请复制图16.3.12红色框中的最后一行，将URL中的“8888”替换为“8889”，然后在本地浏览器中打开它。

16.3.5 关闭未使用的实例

由于云服务是按使用时间计费的，你应该关闭不使用的实例。请注意，还有其他选择：

- “Stopping”（停止）实例意味着你可以重新启动它。这类似于关闭常规服务器的电源。但是，停止的实例仍将按保留的硬盘空间收取少量费用；
- “Terminating”（终止）实例将删除与其关联的所有数据。这包括磁盘，因此你不能再次启动它。只有在你知道将来不需要它的情况下才这样做。

如果你想要将该实例用作更多实例的模板，请右击 图16.3.9中的例子，然后选择“Image”→“Create”以创建该实例的镜像。完成后，选择“实例状态”→“终止”以终止实例。下次要使用此实例时，可以按照本节中的步骤基于保存的镜像创建实例。唯一的区别是，在 图16.3.4所示的“1.选择AMI”中，你必须使用左侧的“我的AMI”选项来选择你保存的镜像。创建的实例将保留镜像硬盘上存储的信息。例如，你不必重新安装CUDA和其他运行时环境。

小结

- 我们可以按需启动和停止实例，而不必购买和制造我们自己的计算机。
- 在使用支持GPU的深度学习框架之前，我们需要安装CUDA。
- 我们可以使用端口转发在远程服务器上运行Jupyter笔记本。

练习

1. 云提供了便利，但价格并不便宜。了解如何启动spot实例²¹⁷以降低成本。
2. 尝试使用不同的GPU服务器。它们有多快？
3. 尝试使用多GPU服务器。你能把事情扩大到什么程度？

Discussions²¹⁸

16.4 选择服务器和GPU

深度学习训练通常需要大量的计算。目前，GPU是深度学习最具成本效益的硬件加速器。与CPU相比，GPU更便宜，性能更高，通常超过一个数量级。此外，一台服务器可以支持多个GPU，高端服务器最多支持8个GPU。更典型的数字是工程工作站最多4个GPU，这是因为热量、冷却和电源需求会迅速增加，超出办公楼所能支持的范围。对于更大的部署，云计算（例如亚马逊的P3²¹⁹和G4²²⁰实例）是一个更实用的解决方案。

²¹⁷ <https://aws.amazon.com/ec2/spot/>

²¹⁸ <https://discuss.d2l.ai/t/5733>

²¹⁹ <https://aws.amazon.com/ec2/instance-types/p3/>

²²⁰ <https://aws.amazon.com/blogs/aws/in-the-news-ec2-instances-g4-with-nvidia-t4-gpus/>

16.4.1 选择服务器

通常不需要购买具有多个线程的高端CPU，因为大部分计算都发生在GPU上。这就是说，由于Python中的全局解释器锁（GIL），CPU的单线程性能在有4-8个GPU的情况下可能很重要。所有的条件都是一样的，这意味着核数较少但时钟频率较高的CPU可能是更经济的选择。例如，当在6核4GHz和8核3.5GHz CPU之间进行选择时，前者更可取，即使其聚合速度较低。一个重要的考虑因素是，GPU使用大量的电能，从而释放大量的热量。这需要非常好的冷却和足够大的机箱来容纳GPU。如有可能，请遵循以下指南：

1. **电源。** GPU使用大量的电源。每个设备预计高达350W（检查显卡的峰值需求而不是一般需求，因为高效代码可能会消耗大量能源）。如果电源不能满足需求，系统会变得不稳定。
2. **机箱尺寸。** GPU很大，辅助电源连接器通常需要额外的空间。此外，大型机箱更容易冷却。
3. **GPU散热。**如果有大量的GPU，可能需要投资水冷。此外，即使风扇较少，也应以“公版设计”为目标，因为它们足够薄，可以在设备之间进气。当使用多风扇GPU，安装多个GPU时，它可能太厚而无法获得足够的空气。
4. **PCIe插槽。**在GPU之间来回移动数据（以及在GPU之间交换数据）需要大量带宽。建议使用16通道的PCIe 3.0插槽。当安装了多个GPU时，请务必仔细阅读主板说明，以确保在同时使用多个GPU时 $16\times$ 带宽仍然可用，并且使用的是PCIe3.0，而不是用于附加插槽的PCIe2.0。在安装多个GPU的情况下，一些主板的带宽降级到 $8\times$ 甚至 $4\times$ 。这部分是由于CPU提供的PCIe通道数量限制。

简而言之，以下是构建深度学习服务器的一些建议。

- **初学者。**购买低功耗的低端GPU（适合深度学习的廉价游戏GPU，功耗150-200W）。如果幸运的话，大家现在常用的计算机将支持它。
- **1个GPU。**一个4核的低端CPU就足够了，大多数主板也足够了。以至少32 GB的DRAM为目标，投资SSD进行本地数据访问。600W的电源应足够。买一个有很多风扇的GPU。
- **2个GPU。**一个4-6核的低端CPU就足够了。可以考虑64 GB的DRAM并投资于SSD。两个高端GPU将需要1000瓦的功率。对于主板，请确保它们具有两个PCIe 3.0 x16插槽。如果可以，请使用PCIe 3.0 x16插槽之间有两个可用空间（60毫米间距）的主板，以提供额外的空气。在这种情况下，购买两个具有大量风扇的GPU。
- **4个GPU。**确保购买的CPU具有相对较快的单线程速度（即较高的时钟频率）。可能需要具有更多PCIe通道的CPU，例如AMD Threadripper。可能需要相对昂贵的主板才能获得4个PCIe 3.0 x16插槽，因为它们可能需要一个PLX来多路复用PCIe通道。购买带有公版设计的GPU，这些GPU很窄，并且让空气进入GPU之间。需要一个1600-2000W的电源，而办公室的插座可能不支持。此服务器可能在运行时声音很大，很热。不想把它放在桌子下面。建议使用128 GB的DRAM。获取一个用于本地存储的SSD（1-2 TB NVMe）和RAID配置的硬盘来存储数据。
- **8 GPU。**需要购买带有多个冗余电源的专用多GPU服务器机箱（例如，每个电源为1600W时为2+1）。这将需要双插槽服务器CPU、256 GB ECC DRAM、快速网卡（建议使用10 GBE），并且需要检查服务器是否支持GPU的物理外形。用户GPU和服务器GPU之间的气流和布线位置存在显著差异（例如RTX 2080和Tesla V100）。这意味着可能无法在服务器中安装消费级GPU，因为电源线间隙不足或缺少合适的接线（本书一位合著者痛苦地发现了这一点）。

16.4.2 选择GPU

目前,AMD和NVIDIA是专用GPU的两大主要制造商。NVIDIA是第一个进入深度学习领域的公司,通过CUDA为深度学习框架提供更好的支持。因此,大多数买家选择NVIDIA GPU。

NVIDIA提供两种类型的GPU,针对个人用户(例如,通过GTX和RTX系列)和企业用户(通过其Tesla系列)。这两种类型的GPU提供了相当的计算能力。但是,企业用户GPU通常使用强制(被动)冷却、更多内存和ECC(纠错)内存。这些GPU更适用于数据中心,通常成本是消费者GPU的十倍。

如果是一个拥有100个服务器的大公司,则应该考虑英伟达Tesla系列,或者在云中使用GPU服务器。对于实验室或10+服务器的中小型公司,英伟达RTX系列可能是最具成本效益的,可以购买超微或华硕机箱的预配置服务器,这些服务器可以有效地容纳4-8个GPU。

GPU供应商通常每一到两年发布一代,例如2017年发布的GTX 1000(Pascal)系列和2019年发布的RTX 2000(Turing)系列。每个系列都提供几种不同的型号,提供不同的性能级别。GPU性能主要是以下三个参数的组合:

1. **计算能力**。通常大家会追求32位浮点计算能力。16位浮点训练(FP16)也进入主流。如果只对预测感兴趣,还可以使用8位整数。最新一代图灵GPU提供4-bit加速。不幸的是,目前训练低精度网络的算法还没有普及;
2. **内存大小**。随着模型变大或训练期间使用的批量变大,将需要更多的GPU内存。检查HBM2(高带宽内存)与GDDR6(图形DDR)内存。HBM2速度更快,但成本更高;
3. **内存带宽**。当有足够的内存带宽时,才能最大限度地利用计算能力。如果使用GDDR6,请追求宽内存总线。

对于大多数用户,只需看看计算能力就足够了。请注意,许多GPU提供不同类型的加速。例如,NVIDIA的Tensor Cores将操作符集的速度提高了5×。确保所使用的库支持这一点。GPU内存应不小于4GB(8GB更好)。尽量避免将GPU也用于显示GUI(改用内置显卡)。如果无法避免,请添加额外的2GB RAM以确保安全。

图16.4.1比较了各种GTX 900、GTX 1000和RTX 2000系列的(GFlops)和价格(Price)。价格是维基百科上的建议价格。



图16.4.1: 浮点计算能力和价格比较

由上图，可以看出很多事情：

1. 在每个系列中，价格和性能大致成比例。Titan因拥有大GPU内存而有相当的溢价。然而，通过比较980 Ti和1080 Ti可以看出，较新型号具有更好的成本效益。RTX 2000系列的价格似乎没有多大提高。然而，它们提供了更优秀的低精度性能（FP16、INT8和INT4）；
2. GTX 1000系列的性价比大约是900系列的两倍；
3. 对于RTX 2000系列，浮点计算能力是价格的“仿射”函数。

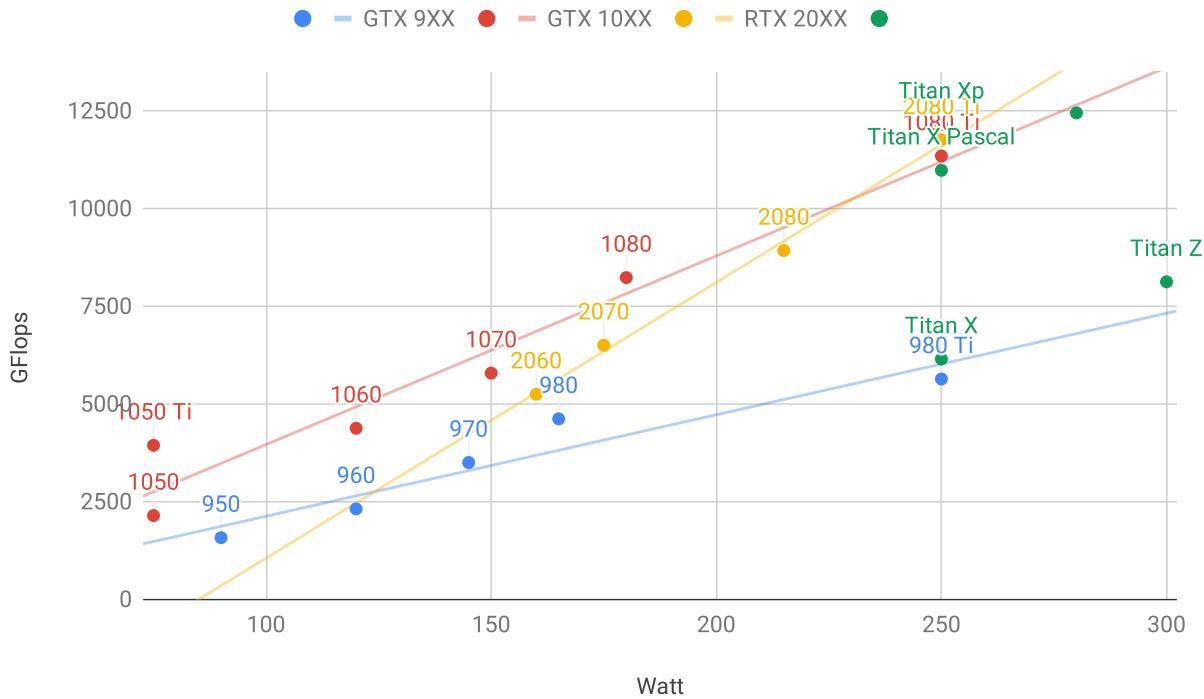


图16.4.2: 浮点计算能力和能耗

图16.4.2显示了能耗与计算量基本成线性关系。其次，后一代更有效率。这似乎与对应于RTX 2000系列的图表相矛盾。然而，这是TensorCore不成比例的大能耗的结果。

小结

- 在构建服务器时，请注意电源、PCIe总线通道、CPU单线程速度和散热。
- 如果可能，应该购买最新一代的GPU。
- 使用云进行大型部署。
- 高密度服务器可能不与所有GPU兼容。在购买之前，请检查一下机械和散热规格。
- 为提高效率，请使用FP16或更低的精度。

16.5 为本书做贡献

读者们的投稿大大帮助我们改进了本书的质量。如果你发现笔误、无效的链接、一些你认为我们遗漏了引文的地方，代码看起来不优雅，或者解释不清楚的地方，请回复我们以帮助读者。在常规书籍中，两次印刷之间的间隔（即修订笔误的间隔）常常需要几年，但这本书的改进通常需要几小时到几天的时间。由于版本控制和持续自动集成（CI）测试，这一切颇为高效。为此，你需要向github存储库提交一个 pull request²²¹。当你的pull请求被作者合并到代码库中时，你将成为贡献者²²²。

16.5.1 提交微小更改

最常见的贡献是编辑一句话或修正笔误。我们建议你在GitHub存储库²²³ 中查找源文件，以定位源文件（一个markdown文件）。然后单击右上角的“Edit this file”按钮，在markdown文件中进行更改。

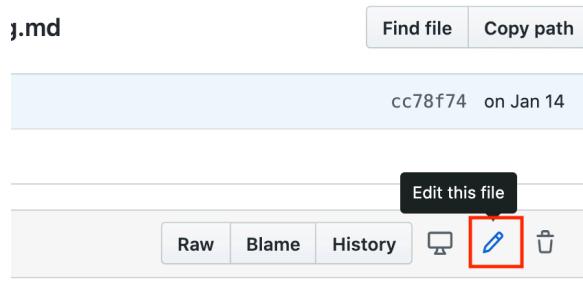


图16.5.1: 在Github上编辑文件

完成后，在页面底部的“Propose file change”（“提交文件修改”）面板中填写更改说明，然后单击“Propose file change”按钮。它会重定向到新页面以查看你的更改（图16.5.7）。如果一切正常，你可以通过点击“Create pull request”按钮提交pull请求。

16.5.2 大量文本或代码修改

如果你计划修改大量文本或代码，那么你需要更多地了解本书使用的格式。源文件基于markdown格式²²⁴，并通过d2lbook²²⁵包提供了一组扩展，例如引用公式、图像、章节和引文。你可以使用任何markdown编辑器打开这些文件并进行更改。

如果你想要更改代码，我们建议你使用Jupyter Notebook打开这些标记文件，如 16.1节中所述。这样你就可以在运行并测试你的更改。请记住在提交更改之前清除所有输出，我们的CI系统将执行你更新的部分以生成输出。

²²¹ <https://github.com/d2l-ai/d2l-en/pulls>

²²² <https://github.com/d2l-ai/d2l-en/graphs/contributors>

²²³ <https://github.com/d2l-ai/d2l-en>

²²⁴ <https://daringfireball.net/projects/markdown/syntax>

²²⁵ <http://book.d2l.ai/user/markdown.html>

某些部分可能支持多个框架实现。如果你添加的新代码块不是使用mxnet，请使用`#@tab`来标记代码块的起始行。例如`#@tab pytorch`用于一个PyTorch代码块，`#@tab tensorflow`用于一个TensorFlow代码块，`#@tab paddle`用于一个PaddlePaddle代码块，或者`#@tab all`是所有实现的共享代码块。你可以参考[d2lbook](#)²²⁶包了解更多信息。

16.5.3 提交主要更改

我们建议你使用标准的Git流程提交大量修改。简而言之，该过程的工作方式如图16.5.2中所述。

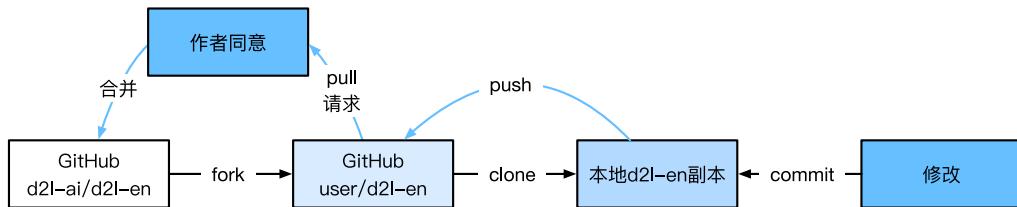


图16.5.2: 为这本书作贡献

我们将向你详细介绍这些步骤。如果你已经熟悉Git，可以跳过本部分。在介绍时，我们假设贡献者的用户名为“astonzhang”。

安装Git

Git开源书籍描述了[如何安装git](#)²²⁷。这通常通过Ubuntu Linux上的`apt install git`，在MacOS上安装Xcode开发人员工具或使用github的[桌面客户端](#)²²⁸来实现。如果你没有GitHub帐户，则需要注册一个帐户。

登录GitHub

在浏览器中输入本书代码存储库的[地址](#)²²⁹。单击图16.5.3右上角红色框中的Fork按钮，以复制本书的存储库。这将是你的副本，你可以随心所欲地更改它。

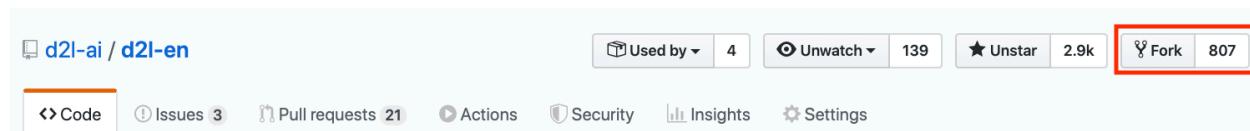


图16.5.3: 代码存储库页面

现在，本书的代码库将被分叉（即复制）到你的用户名，例如`astonzhang/d2l-en`显示在图16.5.4的左上角。

²²⁶ http://book.d2l.ai/user/code_tabs.html

²²⁷ <https://git-scm.com/book/en/v2>

²²⁸ <https://desktop.github.com>

²²⁹ <https://github.com/d2l-ai/d2l-en/>

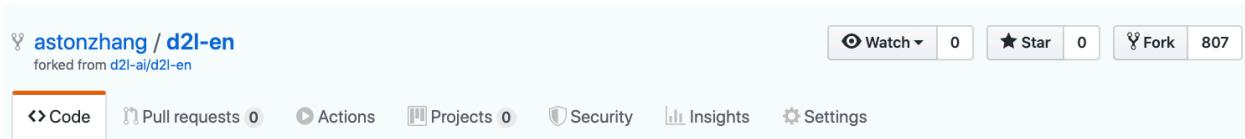


图16.5.4: 分叉代码存储库

克隆存储库

要克隆存储库（即制作本地副本），我们需要获取其存储库地址。点击 图16.5.5中的绿色按钮显示此信息。如果你决定将此分支保留更长时间，请确保你的本地副本与主存储库保持最新。现在，只需按照安装 (page 9) 中的说明开始。主要区别在于，你现在下载的是你自己的存储库分支。



图16.5.5: 克隆存储库

```
# 将your.github.username替换为你的github用户名  
git clone https://github.com/your.github.username/d2l-en.git
```

编辑和推送

现在是编辑这本书的时候了。最好按照 16.1 节 中的说明在 Jupyter Notebook 中编辑它。进行更改并检查它们是否正常。假设我们已经修改了文件 ~/d2l-en/chapter_appendix_tools/how-to-contribute.md 中的一个拼写错误。你可以检查你更改了哪些文件。

此时，Git 将提示 chapter_appendix_tools/how-to-contribute.md 文件已被修改。

```
mylaptop:d2l-en me$ git status  
On branch master  
Your branch is up-to-date with 'origin/master'.  
  
Changes not staged for commit:  
(use "git add <file>..." to update what will be committed)  
(use "git checkout -- <file>..." to discard changes in working directory)  
  
modified:   chapter_appendix_tools/how-to-contribute.md
```

在确认这是你想要的之后，执行以下命令：

```
git add chapter_appendix_tools/how-to-contribute.md  
git commit -m 'fix typo in git documentation'  
git push
```

然后，更改后的代码将位于存储库的个人分支中。要请求添加更改，你必须为本书的官方存储库创建一个Pull请求。

提交Pull请求

如 图16.5.6所示，进入github上的存储库分支，选择“New pull request”。这将打开一个页面，显示你的编辑与本书主存储库中的当前内容之间的更改。

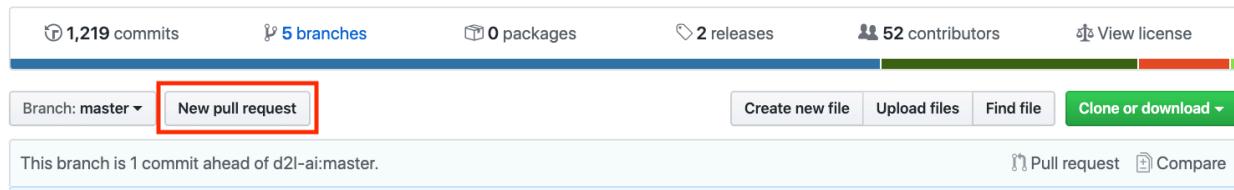


图16.5.6: 新的Pull请求

最后，单击按钮提交Pull请求，如 图16.5.7所示。请务必描述你在Pull请求中所做的更改。这将使作者更容易审阅它，并将其与本书合并。根据更改的不同，这可能会立即被接受，也可能会被拒绝，或者更有可能的是，你会收到一些关于更改的反馈。一旦你把它们合并了，你就做完了。

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

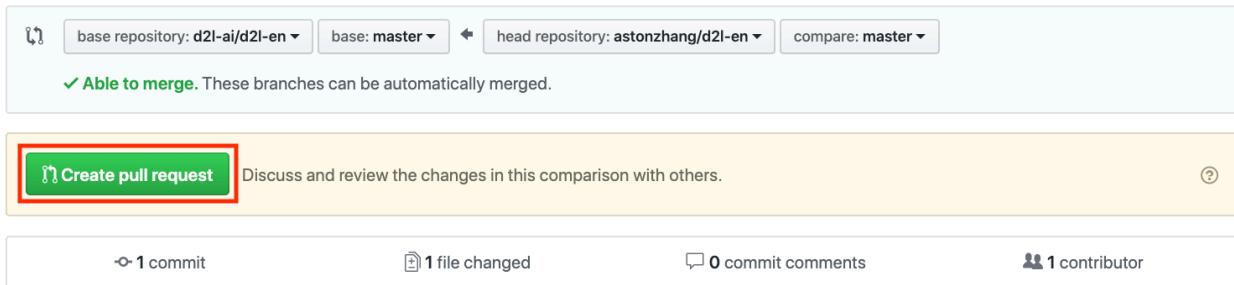


图16.5.7: 创建Pull请求

小结

- 你可以使用GitHub为本书做贡献。
- 你可以直接在GitHub上编辑文件以进行微小更改。
- 要进行重大更改, 请分叉存储库, 在本地编辑内容, 并在准备好后再做出贡献。
- 尽量不要提交巨大的Pull请求, 因为这会使它们难以理解和合并。最好拆分为几个小一点的。

练习

1. 启动并分叉d2l-ai/d2l-en存储库。
2. 如果发现任何需要改进的地方(例如, 缺少引用), 请提交Pull请求。
3. 通常更好的做法是使用新分支创建Pull请求。学习如何用[Git分支²³⁰](#)来做这件事。

Discussions²³¹

16.6 d2l API 文档

d2l包以下成员的实现及其定义和解释部分可在[源文件²³²](#)中找到。

16.6.1 模型

16.6.2 数据

16.6.3 训练

16.6.4 公用

²³⁰ <https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell>

²³¹ <https://discuss.d2l.ai/t/5730>

²³² <https://github.com/d2l-ai/d2l-en/tree/master/d2l>

Bibliography

- [Ahmed et al., 2012] Ahmed, A., Aly, M., Gonzalez, J., Narayananurthy, S., & Smola, A. J. (2012). Scalable inference in latent variable models. *Proceedings of the fifth ACM international conference on Web search and data mining* (pp. 123–132).
- [Aji & McEliece, 2000] Aji, S. M., & McEliece, R. J. (2000). The generalized distributive law. *IEEE transactions on Information Theory*, 46(2), 325–343.
- [Ba et al., 2016] Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.
- [Bahdanau et al., 2014] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.
- [Bay et al., 2006] Bay, H., Tuytelaars, T., & Van Gool, L. (2006). Surf: speeded up robust features. *European conference on computer vision* (pp. 404–417).
- [Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb), 1137–1155.
- [Bishop, 1995] Bishop, C. M. (1995). Training with noise is equivalent to tikhonov regularization. *Neural computation*, 7(1), 108–116.
- [Bishop, 2006] Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.
- [Bodla et al., 2017] Bodla, N., Singh, B., Chellappa, R., & Davis, L. S. (2017). Soft-nms—improving object detection with one line of code. *Proceedings of the IEEE international conference on computer vision* (pp. 5561–5569).
- [Bojanowski et al., 2017] Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5, 135–146.

- [Bollobas, 1999] Bollobás, B. (1999). *Linear analysis*. Cambridge University Press, Cambridge.
- [Bowman et al., 2015] Bowman, S. R., Angeli, G., Potts, C., & Manning, C. D. (2015). A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*.
- [Boyd & Vandenberghe, 2004] Boyd, S., & Vandenberghe, L. (2004). *Convex Optimization*. Cambridge, England: Cambridge University Press.
- [Brown & Sandholm, 2017] Brown, N., & Sandholm, T. (2017). Libratus: the superhuman ai for no-limit poker. *IJCAI* (pp. 5226–5228).
- [Brown et al., 1990] Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Lafferty, J., … Roossin, P. S. (1990). A statistical approach to machine translation. *Computational linguistics*, 16(2), 79–85.
- [Brown et al., 1988] Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Mercer, R. L., & Roossin, P. (1988). A statistical approach to language translation. *Coling Budapest 1988 Volume 1: International Conference on Computational Linguistics*.
- [Campbell et al., 2002] Campbell, M., Hoane Jr, A. J., & Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, 134(1-2), 57–83.
- [Canny, 1987] Canny, J. (1987). A computational approach to edge detection. *Readings in computer vision* (pp. 184–203). Elsevier.
- [Cer et al., 2017] Cer, D., Diab, M., Agirre, E., Lopez-Gazpio, I., & Specia, L. (2017). Semeval-2017 task 1: semantic textual similarity multilingual and crosslingual focused evaluation. *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)* (pp. 1–14).
- [Cheng et al., 2016] Cheng, J., Dong, L., & Lapata, M. (2016). Long short-term memory-networks for machine reading. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing* (pp. 551–561).
- [Cho et al., 2014a] Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.
- [Cho et al., 2014b] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.
- [Chung et al., 2014] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.
- [Collobert et al., 2011] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(ARTICLE), 2493–2537.
- [Dalal & Triggs, 2005] Dalal, N., & Triggs, B. (2005). Histograms of oriented gradients for human detection. *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)* (pp. 886–893).

- [DeCock, 2011] De Cock, D. (2011). Ames, iowa: alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education*, 19(3).
- [DeCandia et al., 2007] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ··· Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* (pp. 205–220).
- [Devlin et al., 2018] Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.
- [Dosovitskiy et al., 2021] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ··· others. (2021). An image is worth 16x16 words: transformers for image recognition at scale. *International Conference on Learning Representations*.
- [Doucet et al., 2001] Doucet, A., De Freitas, N., & Gordon, N. (2001). An introduction to sequential monte carlo methods. *Sequential Monte Carlo methods in practice* (pp. 3–14). Springer.
- [Duchi et al., 2011] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul), 2121–2159.
- [Dumoulin & Visin, 2016] Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.
- [Flammarion & Bach, 2015] Flammarion, N., & Bach, F. (2015). From averaging to acceleration, there is only a step-size. *Conference on Learning Theory* (pp. 658–695).
- [Gatys et al., 2016] Gatys, L. A., Ecker, A. S., & Bethge, M. (2016). Image style transfer using convolutional neural networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2414–2423).
- [Girshick, 2015] Girshick, R. (2015). Fast r-cnn. *Proceedings of the IEEE international conference on computer vision* (pp. 1440–1448).
- [Girshick et al., 2014] Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 580–587).
- [Glorot & Bengio, 2010] Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feed-forward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).
- [Goh, 2017] Goh, G. (2017). Why momentum really works. *Distill*. URL: <http://distill.pub/2017/momentum>, doi:10.23915/distill.00006²³³
- [Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. <http://www.deeplearningbook.org>.

²³³ <https://doi.org/10.23915/distill.00006>

- [Goodfellow et al., 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems* (pp. 2672–2680).
- [Gotmare et al., 2018] Gotmare, A., Keskar, N. S., Xiong, C., & Socher, R. (2018). A closer look at deep learning heuristics: learning rate restarts, warmup and distillation. *arXiv preprint arXiv:1810.13243*.
- [Graves, 2013] Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.
- [Graves & Schmidhuber, 2005] Graves, A., & Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural networks*, 18(5-6), 602–610.
- [Hadjis et al., 2016] Hadjis, S., Zhang, C., Mitliagkas, I., Iter, D., & Ré, C. (2016). Omnivore: an optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487*.
- [He et al., 2017] He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask r-cnn. *Proceedings of the IEEE international conference on computer vision* (pp. 2961–2969).
- [He et al., 2015] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: surpassing human-level performance on imagenet classification. *Proceedings of the IEEE international conference on computer vision* (pp. 1026–1034).
- [He et al., 2016a] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).
- [He et al., 2016b] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Identity mappings in deep residual networks. *European conference on computer vision* (pp. 630–645).
- [Hebb & Hebb, 1949] Hebb, D. O., & Hebb, D. (1949). *The organization of behavior*. Vol. 65. Wiley New York.
- [Hendrycks & Gimpel, 2016] Hendrycks, D., & Gimpel, K. (2016). Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.
- [Hennessy & Patterson, 2011] Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.
- [Hochreiter et al., 2001] Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., & others (2001). *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*.
- [Hochreiter & Schmidhuber, 1997] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735–1780.
- [Hoyer et al., 2009] Hoyer, P. O., Janzing, D., Mooij, J. M., Peters, J., & Schölkopf, B. (2009). Nonlinear causal discovery with additive noise models. *Advances in neural information processing systems* (pp. 689–696).
- [Hu et al., 2018] Hu, J., Shen, L., & Sun, G. (2018). Squeeze-and-excitation networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 7132–7141).

- [Hu et al., 2020] Hu, Z., Lee, R. K.-W., Aggarwal, C. C., & Zhang, A. (2020). Text style transfer: a review and experimental evaluation. *arXiv preprint arXiv:2010.12742*.
- [Huang et al., 2017] Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4700–4708).
- [Ioffe, 2017] Ioffe, S. (2017). Batch renormalization: towards reducing minibatch dependence in batch-normalized models. *Advances in neural information processing systems* (pp. 1945–1953).
- [Ioffe & Szegedy, 2015] Ioffe, S., & Szegedy, C. (2015). Batch normalization: accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.
- [Izmailov et al., 2018] Izmailov, P., Podoprikin, D., Garipov, T., Vetrov, D., & Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*.
- [Jaeger, 2002] Jaeger, H. (2002). *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach*. Vol. 5. GMD-Forschungszentrum Informationstechnik Bonn.
- [James, 2007] James, W. (2007). *The principles of psychology*. Vol. 1. Cosimo, Inc.
- [Jia et al., 2018] Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., ··· others. (2018). Highly scalable deep learning training system with mixed-precision: training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*.
- [Jouppi et al., 2017] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., ··· others. (2017). In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (pp. 1–12).
- [Karras et al., 2017] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*.
- [Kim, 2014] Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.
- [Kingma & Ba, 2014] Kingma, D. P., & Ba, J. (2014). Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
- [Kolter, 2008] Kolter, Z. (2008). Linear algebra review and reference. Available online: <http://>.
- [Koren, 2009] Koren, Y. (2009). Collaborative filtering with temporal dynamics. *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 447–456).
- [Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* (pp. 1097–1105).
- [Kung, 1988] Kung, S. Y. (1988). Vlsi array processors. *Englewood Cliffs, NJ, Prentice Hall, 1988, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy*.

- [LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., & others. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278–2324.
- [Li, 2017] Li, M. (2017). *Scaling Distributed Machine Learning with System and Algorithm Co-design* (Doctoral dissertation). PhD Thesis, CMU.
- [Li et al., 2014] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., ··· Su, B.-Y. (2014). Scaling distributed machine learning with the parameter server. *11th USENIX OSDI 14* (pp. 583–598).
- [Lin et al., 2013] Lin, M., Chen, Q., & Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.
- [Lin et al., 2017a] Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal loss for dense object detection. *Proceedings of the IEEE international conference on computer vision* (pp. 2980–2988).
- [Lin et al., 2010] Lin, Y., Lv, F., Zhu, S., Yang, M., Cour, T., Yu, K., ··· others. (2010). Imagenet classification: fast descriptor coding and large-scale svm training. *Large scale visual recognition challenge*.
- [Lin et al., 2017b] Lin, Z., Feng, M., Santos, C. N. d., Yu, M., Xiang, B., Zhou, B., & Bengio, Y. (2017). A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130*.
- [Lipton & Steinhardt, 2018] Lipton, Z. C., & Steinhardt, J. (2018). Troubling trends in machine learning scholarship. *arXiv preprint arXiv:1807.03341*.
- [Liu et al., 2016] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). Ssd: single shot multibox detector. *European conference on computer vision* (pp. 21–37).
- [Liu et al., 2019] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ··· Stoyanov, V. (2019). Roberta: a robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.
- [Long et al., 2015] Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3431–3440).
- [Loshchilov & Hutter, 2016] Loshchilov, I., & Hutter, F. (2016). Sgdr: stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*.
- [Lowe, 2004] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2), 91–110.
- [Luo et al., 2018] Luo, P., Wang, X., Shao, W., & Peng, Z. (2018). Towards understanding regularization in batch normalization. *arXiv preprint*.
- [Maas et al., 2011] Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1* (pp. 142–150).
- [McCann et al., 2017] McCann, B., Bradbury, J., Xiong, C., & Socher, R. (2017). Learned in translation: contextualized word vectors. *Advances in Neural Information Processing Systems* (pp. 6294–6305).

- [McCulloch & Pitts, 1943] McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4), 115–133.
- [Merity et al., 2016] Merity, S., Xiong, C., Bradbury, J., & Socher, R. (2016). Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*.
- [Mikolov et al., 2013a] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.
- [Mikolov et al., 2013b] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* (pp. 3111–3119).
- [Mirhoseini et al., 2017] Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., ··· Dean, J. (2017). Device placement optimization with reinforcement learning. *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (pp. 2430–2439).
- [Mnih et al., 2014] Mnih, V., Heess, N., Graves, A., & others. (2014). Recurrent models of visual attention. *Advances in neural information processing systems* (pp. 2204–2212).
- [Nadaraya, 1964] Nadaraya, E. A. (1964). On estimating regression. *Theory of Probability & Its Applications*, 9(1), 141–142.
- [Nesterov & Vial, 2000] Nesterov, Y., & Vial, J.-P. (2000). *Confidence level solutions for stochastic programming, Stochastic Programming E-Print Series*.
- [Nesterov, 2018] Nesterov, Y. (2018). *Lectures on convex optimization*. Vol. 137. Springer.
- [Papineni et al., 2002] Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. *Proceedings of the 40th annual meeting of the Association for Computational Linguistics* (pp. 311–318).
- [Parikh et al., 2016] Parikh, A. P., Täckström, O., Das, D., & Uszkoreit, J. (2016). A decomposable attention model for natural language inference. *arXiv preprint arXiv:1606.01933*.
- [Park et al., 2019] Park, T., Liu, M.-Y., Wang, T.-C., & Zhu, J.-Y. (2019). Semantic image synthesis with spatially-adaptive normalization. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2337–2346).
- [Paulus et al., 2017] Paulus, R., Xiong, C., & Socher, R. (2017). A deep reinforced model for abstractive summarization. *arXiv preprint arXiv:1705.04304*.
- [Pennington et al., 2014] Pennington, J., Socher, R., & Manning, C. (2014). Glove: global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1532–1543).
- [Peters et al., 2017a] Peters, J., Janzing, D., & Schölkopf, B. (2017). *Elements of causal inference: foundations and learning algorithms*. MIT press.

- [Peters et al., 2017b] Peters, M., Ammar, W., Bhagavatula, C., & Power, R. (2017). Semi-supervised sequence tagging with bidirectional language models. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 1756–1765).
- [Peters et al., 2018] Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)* (pp. 2227–2237).
- [Petersen et al., 2008] Petersen, K. B., Pedersen, M. S., & others. (2008). The matrix cookbook. *Technical University of Denmark*, 7(15), 510.
- [Polyak, 1964] Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5), 1–17.
- [Radford et al., 2018] Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training. *OpenAI*.
- [Radford et al., 2019] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 9.
- [Rajpurkar et al., 2016] Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016). Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*.
- [Reddi et al., 2019] Reddi, S. J., Kale, S., & Kumar, S. (2019). On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*.
- [Redmon et al., 2016] Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: unified, real-time object detection. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 779–788).
- [Reed & DeFreitas, 2015] Reed, S., & De Freitas, N. (2015). Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*.
- [Ren et al., 2015] Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: towards real-time object detection with region proposal networks. *Advances in neural information processing systems* (pp. 91–99).
- [Russell & Norvig, 2016] Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.
- [Santurkar et al., 2018] Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization? *Advances in Neural Information Processing Systems* (pp. 2483–2493).
- [Schuster & Paliwal, 1997] Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11), 2673–2681.
- [Sennrich et al., 2015] Sennrich, R., Haddow, B., & Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

- [Sergeev & DelBalso, 2018] Sergeev, A., & Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*.
- [Shao et al., 2020] Shao, H., Yao, S., Sun, D., Zhang, A., Liu, S., Liu, D., ⋯ Abdelzaher, T. (2020). Controlvae: controllable variational autoencoder. *Proceedings of the 37th International Conference on Machine Learning*.
- [Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ⋯ others. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587), 484.
- [Simonyan & Zisserman, 2014] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.
- [Smola & Narayananamurthy, 2010] Smola, A., & Narayananamurthy, S. (2010). An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2), 703–710.
- [Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 1929–1958.
- [Strang, 1993] Strang, G. (1993). *Introduction to linear algebra*. Vol. 3. Wellesley-Cambridge Press Wellesley, MA.
- [Sukhbaatar et al., 2015] Sukhbaatar, S., Weston, J., Fergus, R., & others. (2015). End-to-end memory networks. *Advances in neural information processing systems* (pp. 2440–2448).
- [Sutskever et al., 2013] Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *International conference on machine learning* (pp. 1139–1147).
- [Sutskever et al., 2014] Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in neural information processing systems* (pp. 3104–3112).
- [Szegedy et al., 2017] Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. *Thirty-First AAAI Conference on Artificial Intelligence*.
- [Szegedy et al., 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ⋯ Rabinovich, A. (2015). Going deeper with convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1–9).
- [Szegedy et al., 2016] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2818–2826).
- [Tallec & Ollivier, 2017] Tallec, C., & Ollivier, Y. (2017). Unbiasing truncated backpropagation through time. *arXiv preprint arXiv:1705.08209*.

- [Tay et al., 2020] Tay, Y., Dehghani, M., Bahri, D., & Metzler, D. (2020). Efficient transformers: a survey. *arXiv preprint arXiv:2009.06732*.
- [Teye et al., 2018] Teye, M., Azizpour, H., & Smith, K. (2018). Bayesian uncertainty estimation for batch normalized deep networks. *arXiv preprint arXiv:1802.06455*.
- [Tieleman & Hinton, 2012] Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2), 26–31.
- [Turing, 1950] Turing, A. (1950). Computing machinery and intelligence. *Mind*, 59(236), 433.
- [Uijlings et al., 2013] Uijlings, J. R., Van De Sande, K. E., Gevers, T., & Smeulders, A. W. (2013). Selective search for object recognition. *International journal of computer vision*, 104(2), 154–171.
- [Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems* (pp. 5998–6008).
- [Wang et al., 2018] Wang, L., Li, M., Liberty, E., & Smola, A. J. (2018). Optimal message scheduling for aggregation. *NETWORKS*, 2(3), 2–3.
- [Wang et al., 2016] Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., & Owens, J. D. (2016). Gunrock: a high-performance graph processing library on the gpu. *ACM SIGPLAN Notices* (p. 11).
- [Warstadt et al., 2019] Warstadt, A., Singh, A., & Bowman, S. R. (2019). Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, 7, 625–641.
- [Wasserman, 2013] Wasserman, L. (2013). *All of statistics: a concise course in statistical inference*. Springer Science & Business Media.
- [Watkins & Dayan, 1992] Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4), 279–292.
- [Watson, 1964] Watson, G. S. (1964). Smooth regression analysis. *Sankhyā: The Indian Journal of Statistics, Series A*, pp. 359–372.
- [Welling & Teh, 2011] Welling, M., & Teh, Y. W. (2011). Bayesian learning via stochastic gradient langevin dynamics. *Proceedings of the 28th international conference on machine learning (ICML-11)* (pp. 681–688).
- [Werbos, 1990] Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10), 1550–1560.
- [Wigner, 1958] Wigner, E. P. (1958). On the distribution of the roots of certain symmetric matrices. *Ann. Math* (pp. 325–327).
- [Wood et al., 2011] Wood, F., Gasthaus, J., Archambeau, C., James, L., & Teh, Y. W. (2011). The sequence memoizer. *Communications of the ACM*, 54(2), 91–98.

- [Wu et al., 2017] Wu, C.-Y., Ahmed, A., Beutel, A., Smola, A. J., & Jing, H. (2017). Recurrent recommender networks. *Proceedings of the tenth ACM international conference on web search and data mining* (pp. 495–503).
- [Wu et al., 2016] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., ⋯others. (2016). Google's neural machine translation system: bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.
- [Xiao et al., 2017] Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*.
- [Xiao et al., 2018] Xiao, L., Bahri, Y., Sohl-Dickstein, J., Schoenholz, S., & Pennington, J. (2018). Dynamical isometry and a mean field theory of cnns: how to train 10,000-layer vanilla convolutional neural networks. *International Conference on Machine Learning* (pp. 5393–5402).
- [Xiong et al., 2018] Xiong, W., Wu, L., Alleva, F., Droppo, J., Huang, X., & Stolcke, A. (2018). The microsoft 2017 conversational speech recognition system. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5934–5938).
- [You et al., 2017] You, Y., Gitman, I., & Ginsburg, B. (2017). Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*.
- [Zaheer et al., 2018] Zaheer, M., Reddi, S., Sachan, D., Kale, S., & Kumar, S. (2018). Adaptive methods for nonconvex optimization. *Advances in Neural Information Processing Systems* (pp. 9793–9803).
- [Zeiler, 2012] Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.
- [Zhang et al., 2021] Zhang, A., Tay, Y., Zhang, S., Chan, A., Luu, A. T., Hui, S. C., & Fu, J. (2021). Beyond fully-connected layers with quaternions: parameterization of hypercomplex multiplications with 1/n parameters. *International Conference on Learning Representations*.
- [Zhao et al., 2019] Zhao, Z.-Q., Zheng, P., Xu, S.-t., & Wu, X. (2019). Object detection with deep learning: a review. *IEEE transactions on neural networks and learning systems*, 30(11), 3212–3232.
- [Zhu et al., 2017] Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. *Proceedings of the IEEE international conference on computer vision* (pp. 2223–2232).
- [Zhu et al., 2015] Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., & Fidler, S. (2015). Aligning books and movies: towards story-like visual explanations by watching movies and reading books. *Proceedings of the IEEE international conference on computer vision* (pp. 19–27).