动手学深度学习

Release 2.0.0-alpha0

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola

Mar 19, 2021

目录

安装 **1**

符号 **5**

前言 **9**

1 预备知识 **31**

2 线性神经网络 **77**

2.1 线性回归 . . . 77

2.1.1 线性回归的基本元素 . . . 78

2.1.2 矢量化加速 . . . 81

2.1.3 正态分布与平方损失 . . . 83

2.1.4 从线性回归到深度网络 . . . 84

2.1.5 小结 . . . 86

2.1.6 练习 . . . 86

2.2 线性回归的从零开始实现 . . . 86

2.2.1 生成数据集 . . . 87

2.2.2 读取数据集 . . . 88

2.2.3 初始化模型参数 . . . 89

2.2.4 定义模型 . . . 90

2.2.5 定义损失函数 . . . 90

2.2.6 定义优化算法 . . . 90

2.2.7 训练 . . . 91

2.2.8 小结 . . . 92

2.2.9 练习 . . . 92

2.3 线性回归的简洁实现 . . . 92

2.3.1 生成数据集 . . . 93

2.3.2 读取数据集 . . . 93
2.3.3 定义模型 . . . 94
2.3.4 初始化模型参数 . . . 95
2.3.5 定义损失函数 . . . 95
2.3.6 定义优化算法 . . . 95
2.3.7 训练 . . . 96
2.3.8 小结 . . . 97
2.3.9 练习 . . . 97
2.4 softmax回归 . . . 97
2.4.1 分类问题 . . . 98
2.4.2 网络结构 . . . 98
2.4.3 全连接层的参数开销 . . . 99
2.4.4 softmax操作 . . . 99
2.4.5 小批量样本的矢量化 . . . 100
2.4.6 损失函数 . . . 100
2.4.7 信息论基础 . . . 101
2.4.8 模型预测和评估 . . . 102
2.4.9 小结 . . . 102
2.4.10 练习 . . . 102
2.5 图像分类数据集 . . . 103
2.5.1 读取数据集 . . . 103
2.5.2 读取小批量 . . . 105
2.5.3 整合所有组件 . . . 106
2.5.4 小结 . . . 106
2.5.5 练习 . . . 107
2.6 softmax回归的从零开始实现 . . . 107
2.6.1 初始化模型参数 . . . 107
2.6.2 定义softmax操作 . . . 108
2.6.3 定义模型 . . . 109
2.6.4 定义损失函数 . . . 109
2.6.5 分类准确率 . . . 109
2.6.6 训练 . . . 111
2.6.7 预测 . . . 114
2.6.8 小结 . . . 114
2.6.9 练习 . . . 114
2.7 softmax回归的简洁实现 . . . 115
2.7.1 初始化模型参数 . . . 115
2.7.2 重新审视Softmax的实现 . . . 115
2.7.3 优化算法 . . . 116
2.7.4 训练 . . . 116
2.7.5 小结 . . . 117
2.7.6 练习 . . . 117

3 多层感知机 119

3.1 多层感知机 . . . 119
3.1.1 隐藏层 . . . 120
3.1.2 激活函数 . . . 122
3.1.3 小结 . . . 126
3.1.4 练习 . . . 126
3.2 多层感知机的从零开始实现 . . . 127
3.2.1 初始化模型参数 . . . 127
3.2.2 激活函数 . . . 128
3.2.3 模型 . . . 128
3.2.4 损失函数 . . . 128
3.2.5 训练 . . . 128
3.2.6 小结 . . . 129
3.2.7 练习 . . . 129
3.3 多层感知机的简洁实现 . . . 130
3.3.1 模型 . . . 130
3.3.2 小结 . . . 131
3.3.3 练习 . . . 131
3.4 模型选择、欠拟合和过拟合 . . . 131
3.4.1 训练误差和泛化误差 . . . 132
3.4.2 模型选择 . . . 134
3.4.3 欠拟合还是过拟合? . . . 135
3.4.4 多项式回归 . . . 136
3.4.5 小结 . . . 141
3.4.6 练习 . . . 141
3.5 权重衰减 . . . 141
3.5.1 范数与权重衰减 . . . 142
3.5.2 高维线性回归 . . . 143
3.5.3 从零开始实现 . . . 143
3.5.4 简洁实现 . . . 146
3.5.5 小结 . . . 148
3.5.6 练习 . . . 148
3.6 Dropout . . . 148
3.6.1 重新审视过拟合 . . . 148
3.6.2 扰动的鲁棒性 . . . 149
3.6.3 实践中的dropout . . . 150
3.6.4 从零开始实现 . . . 150
3.6.5 简洁实现 . . . 153
3.6.6 小结 . . . 154
3.6.7 练习 . . . 154
3.7 正向传播、反向传播和计算图 . . . 155
3.7.1 正向传播 . . . 155

3.7.2 正向传播计算图 156
3.7.3 反向传播 156
3.7.4 训练神经网络 157
3.7.5 小结 157
3.7.6 练习 158
3.8 数值稳定性和模型初始化 158
3.8.1 梯度消失和梯度爆炸 158
3.8.2 参数初始化 161
3.8.3 小结 162
3.8.4 练习 162
3.9 环境和分布偏移 163
3.9.1 分布偏移的类型 163
3.9.2 分布偏移示例 166
3.9.3 分布偏移纠正 167
3.9.4 学习问题的分类法 170
3.9.5 机器学习中的公平、责任和透明度 171
3.9.6 小结 172
3.9.7 练习 172
3.10 实战 Kaggle 比赛：预测房价 172
3.10.1 下载和缓存数据集 173
3.10.2 Kaggle 174
3.10.3 访问和读取数据集 175
3.10.4 数据预处理 177
3.10.5 训练 178
3.10.6 K折交叉验证 179
3.10.7 模型选择 180
3.10.8 提交Kaggle的预测 181
3.10.9 小结 183
3.10.10 练习 183

4 深度学习计算 185
4.1 层和块 185
4.1.1 自定义块 187
4.1.2 顺序块 188
4.1.3 在正向传播函数中执行代码 189
4.1.4 效率 191
4.1.5 小结 191
4.1.6 练习 192
4.2 参数管理 192
4.2.1 参数访问 193
4.2.2 参数初始化 196
4.2.3 参数绑定 198

4.2.4 小结 . . . 199
4.2.5 练习 . . . 199
4.3 延后初始化 . . . 199
4.3.1 实例化网络 . . . 200
4.3.2 小结 . . . 201
4.3.3 练习 . . . 201
4.4 自定义层 . . . 202
4.4.1 不带参数的层 . . . 202
4.4.2 带参数的图层 . . . 203
4.4.3 小结 . . . 204
4.4.4 练习 . . . 204
4.5 读写文件 . . . 204
4.5.1 加载和保存张量 . . . 204
4.5.2 加载和保存模型参数 . . . 205
4.5.3 小结 . . . 206
4.5.4 练习 . . . 207
4.6 GPU . . . 207
4.6.1 计算设备 . . . 208
4.6.2 张量与gpu . . . 209
4.6.3 神经网络与GPU . . . 211
4.6.4 小结 . . . 212
4.6.5 练习 . . . 212

5 卷积神经网络 **213**
5.1 从全连接层到卷积 . . . 214
5.1.1 不变性 . . . 214
5.1.2 限制多层感知机 . . . 215
5.1.3 卷积 . . . 216
5.1.4 “沃尔多在哪里”回顾 . . . 217
5.1.5 小结 . . . 218
5.1.6 练习 . . . 218
5.2 图像卷积 . . . 218
5.2.1 互相关运算 . . . 218
5.2.2 卷积层 . . . 220
5.2.3 图像中目标的边缘检测 . . . 220
5.2.4 学习卷积核 . . . 222
5.2.5 互相关和卷积 . . . 223
5.2.6 特征映射和感受野 . . . 223
5.2.7 小结 . . . 223
5.2.8 练习 . . . 224
5.3 填充和步幅 . . . 224
5.3.1 填充 . . . 224

5.3.2 步幅 . . . 226
5.3.3 小结 . . . 227
5.3.4 练习 . . . 227
5.4 多输入多输出通道 . . . 228
5.4.1 多输入通道 . . . 228
5.4.2 多输出通道 . . . 229
5.4.3 1×1 卷积层 . . . 230
5.4.4 小结 . . . 231
5.4.5 练习 . . . 231
5.5 池化层 . . . 232
5.5.1 最大池化层和平均池化层 . . . 232
5.5.2 填充和步幅 . . . 234
5.5.3 多个通道 . . . 235
5.5.4 小结 . . . 235
5.5.5 练习 . . . 236
5.6 卷积神经网络（LeNet） . . . 236
5.6.1 LeNet . . . 237
5.6.2 模型训练 . . . 239
5.6.3 小结 . . . 241
5.6.4 练习 . . . 241

6 现代卷积神经网络 243
6.1 深度卷积神经网络（AlexNet） . . . 244
6.1.1 学习表征 . . . 244
6.1.2 AlexNet . . . 246
6.1.3 读取数据集 . . . 249
6.1.4 训练AlexNet . . . 249
6.1.5 小结 . . . 250
6.1.6 练习 . . . 250
6.2 使用块的网络（VGG） . . . 251
6.2.1 VGG 块 . . . 251
6.2.2 VGG 网络 . . . 252
6.2.3 训练 . . . 253
6.2.4 小结 . . . 254
6.2.5 练习 . . . 254
6.3 网络中的网络（NiN） . . . 255
6.3.1 NiN 块 . . . 255
6.3.2 NiN 模型 . . . 257
6.3.3 训练 . . . 258
6.3.4 小结 . . . 258
6.3.5 练习 . . . 259
6.4 含并行连结的网络（GoogLeNet） . . . 259

6.4.1 Inception块 . . . 259
6.4.2 GoogLeNet 模型 . . . 261
6.4.3 训练 . . . 263
6.4.4 小结 . . . 264
6.4.5 练习 . . . 264
6.5 批量归一化 . . . 264
6.5.1 训练深层网络 . . . 264
6.5.2 批量归一化层 . . . 266
6.5.3 从零实现 . . . 266
6.5.4 使用批量归一化层的 LeNet . . . 268
6.5.5 简明实现 . . . 269
6.5.6 争议 . . . 270
6.5.7 小结 . . . 271
6.5.8 练习 . . . 271
6.6 残差网络（ResNet） . . . 271
6.6.1 函数类 . . . 272
6.6.2 残差块 . . . 273
6.6.3 ResNet模型 . . . 275
6.6.4 训练 ResNet . . . 278
6.6.5 小结 . . . 279
6.6.6 练习 . . . 279
6.7 稠密连接网络（DenseNet） . . . 279
6.7.1 从ResNet到DenseNet . . . 279
6.7.2 稠密块体 . . . 280
6.7.3 过渡层 . . . 282
6.7.4 DenseNet模型 . . . 282
6.7.5 训练模型 . . . 283
6.7.6 小结 . . . 284
6.7.7 练习 . . . 284
7 循环神经网络 285
7.1 序列模型 . . . 286
7.1.1 统计工具 . . . 286
7.1.2 训练 . . . 289
7.1.3 预测 . . . 291
7.1.4 小结 . . . 293
7.1.5 练习 . . . 294
7.2 文本预处理 . . . 294
7.2.1 读取数据集 . . . 295
7.2.2 标记化 . . . 295
7.2.3 词汇 . . . 296
7.2.4 把所有的东西放在一起 . . . 298

7.2.5 小结 298
7.2.6 练习 299
7.3 语言模型和数据集 299
7.3.1 学习语言模型 299
7.3.2 马尔可夫模型与n元语法 300
7.3.3 自然语言统计 301
7.3.4 读取长序列数据 304
7.3.5 小结 307
7.3.6 练习 307
7.4 循环神经网络 308
7.4.1 无隐藏状态的神经网络 308
7.4.2 具有隐藏状态的循环神经网络 309
7.4.3 基于循环神经网络的字符级语言模型 311
7.4.4 困惑度（Perplexity） 311
7.4.5 小结 312
7.4.6 练习 313
7.5 循环神经网络的从零开始实现 313
7.5.1 独热编码 313
7.5.2 初始化模型参数 314
7.5.3 循环神经网络模型 314
7.5.4 预测 316
7.5.5 梯度裁剪 316
7.5.6 训练 317
7.5.7 小结 320
7.5.8 练习 320
7.6 循环神经网络的简洁实现 321
7.6.1 定义模型 321
7.6.2 训练与预测 322
7.6.3 小结 323
7.6.4 练习 324
7.7 通过时间反向传播 324
7.7.1 循环神经网络的梯度分析 324
7.7.2 通过时间反向传播细节 327
7.7.3 小结 329
7.7.4 练习 329

Bibliography **331**

安装

我们需要配置一个环境来运行 Python、Jupyter Notebook、相关库以及运行本书所需的代码，以快速入门并获得动手学习经验。

安装 Miniconda

最简单的方法就是安装依赖 Python 3.x 的 Miniconda[1]。如果已安装 conda，则可以跳过以下步骤。从网站下载相应的 Miniconda sh 文件，然后使用 `sh <FILENAME> -b` 从命令行执行安装。

对于 macOS 用户：

```
# 文件名可能会更改
sh Miniconda3-latest-MacOSX-x86_64.sh -b
```

对于 Linux 用户：

```
# 文件名可能会更改
sh Miniconda3-latest-Linux-x86_64.sh -b
```

接下来，初始化终端 Shell，以便我们可以直接运行 `conda`。

```
~/miniconda3/bin/conda init
```

现在关闭并重新打开当前的 shell。你应该能用下面的命令创建一个新的环境：

```
conda create --name d2l python=3.8 -y
```

[1] https://conda.io/en/latest/miniconda.html

下载 D2L Notebook

接下来，需要下载这本书的代码。你可以点击任何 HTML 页面顶部的“Jupyter 记事本文件”选项下载后解压代码。或者可以按照如下方式进行下载：

```
mkdir d2l-zh && cd d2l-zh
curl https://zh-v2.d2l.ai/d2l-zh.zip -o d2l-zh.zip
unzip d2l-zh.zip && rm d2l-zh.zip
```

注意：如果没有安装 `unzip`，则可以通过运行 `sudo apt install unzip` 进行安装。

现在我们要激活 `d2l` 环境。

```
conda activate d2l
```

安装框架和 `d2l` 软件包

在安装深度学习框架之前，请先检查你的计算机上是否有可用的 GPU（在笔记本电脑上为显示器提供输出的GPU不算）。如果要在 GPU 机器上安装，请继续在*GPU 支持* (page 3) 获取有关安装GPU支持版本的说明。

或者，你可以按照如下方法安装CPU版本。这将足够帮助你完成前几章，但你需要在运行更大模型之前获取GPU。

```
pip install mxnet==1.7.0.post1
```

你还需要安装 `d2l` 软件包，它封装了本书中常用的函数和类。

```
# -U：将所有包升级到最新的可用版本
pip install -U d2l
```

安装完成后，我们通过运行以下命令打开 Jupyter 笔记本：

```
jupyter notebook
```

现在，你可以在 Web 浏览器中打开 http://localhost:8888（通常会自动打开）。然后我们可以运行这本书中每个部分的代码。在运行书籍代码、更新深度学习框架或 `d2l` 软件包之前，请始终执行 `conda activate d2l` 以激活运行时环境。要退出环境，请运行 `conda deactivate`。

GPU 支持

默认情况下，安装的MXNet不支持GPU。这可以确保它在任何计算机（包括大多数笔记本电脑）上运行。本书的部分内容建议或要求使用 GPU 运行。如果你的计算机带有 NVIDIA 显卡并且已安装 CUDA[2]，则应安装支持 GPU 的版本。如果你已经安装了仅支持 CPU 版本，则可能需要先通过运行以下命令将其删除：

```
pip uninstall mxnet
```

然后，我们需要找到安装的 CUDA 版本。你可以通过 `nvcc --version` 或 `cat /usr/local/cuda/version.txt` 查看。假设你已安装 CUDA 10.1，则可以使用以下命令进行安装：

```
# 对于 Windows 用户：
pip install mxnet-cu101==1.7.0 -f https://dist.mxnet.io/python

# 对于 Linux 和 macOS 用户：
pip install mxnet-cu101==1.7.0
```

你可以根据你的 CUDA 版本更改最后一位数字，例如：CUDA 10.0 是 `cu100`，CUDA 9.0 是 `cu90`。

练习

1. 下载该书的代码并安装运行时环境。

Discussions[3]

[2] https://developer.nvidia.com/cuda-downloads
[3] https://discuss.d2l.ai/t/2082

符号

本书中使用的符号概述如下。

数字

- x：标量
- $\mathbf{x}$：向量
- $\mathbf{X}$：矩阵
- X：张量
- $\mathbf{I}$：单位矩阵
- x_i, $[\mathbf{x}]_i$：向量$\mathbf{x}$第i个元素
- x_{ij}, $[\mathbf{X}]_{ij}$：矩阵$\mathbf{X}$第i行第j列的元素

集合论

- $\mathcal{X}$: 集合
- $\mathbb{Z}$: 整数集合
- $\mathbb{R}$ 实数集合
- $\mathbb{R}^n$: n维实数向量
- $\mathbb{R}^{a\times b}$: 包含a行和b列的实数矩阵
- $\mathcal{A}\cup\mathcal{B}$: 集合$\mathcal{A}$和$\mathcal{B}$的并集

- $\mathcal{A} \cap \mathcal{B}$：集合$\mathcal{A}$和$\mathcal{B}$的交集
- $\mathcal{A} \setminus \mathcal{B}$：集合$\mathcal{B}$与集合$\mathcal{A}$相减

函数和运算符

- $f(\cdot)$：函数
- $\log(\cdot)$：自然对数
- $\exp(\cdot)$: 指数函数
- $\mathbf{1}_\mathcal{X}$: 指示函数
- $(\cdot)^\top$: 向量或矩阵的转置
- $\mathbf{X}^{-1}$: 矩阵的逆
- $\odot$: 按元素相乘
- $[\cdot, \cdot]$：连结
- $|\mathcal{X}|$：集合的基数
- $\|\cdot\|_p$:：L_p 正则
- $\|\cdot\|$: L_2 正则
- $\langle \mathbf{x}, \mathbf{y} \rangle$：向量$\mathbf{x}$和$\mathbf{y}$的点积
- $\sum$: 连加
- $\prod$: 连乘
- $\stackrel{\mathrm{def}}{=}$：定义

微积分

- $\frac{dy}{dx}$：y关于x的导数
- $\frac{\partial y}{\partial x}$：$y$关于$x$的偏导数
- $\nabla_{\mathbf{x}} y$：y关于$\mathbf{x}$的梯度
- $\int_a^b f(x) \; dx$: f在a到b区间上关于x的定积分
- $\int f(x) \; dx$: f关于x的不定积分

概率与信息论

- $P(\cdot)$：概率分布
- $z \sim P$: 随机变量z具有概率分布P
- $P(X \mid Y)$：$X \mid Y$的条件概率
- $p(x)$: 概率密度函数
- $E_x[f(x)]$: 函数f对x的数学期望
- $X \perp Y$: 随机变量X和Y是独立的
- $X \perp Y \mid Z$: 随机变量X和Y在给定随机变量Z的条件下是独立的
- $\mathrm{Var}(X)$: 随机变量X的方差
- σ_X: 随机变量X的标准差
- $\mathrm{Cov}(X, Y)$: 随机变量X和Y的协方差
- $\rho(X, Y)$: 随机变量X和Y的相关性
- $H(X)$: 随机变量X的熵
- $D_{\mathrm{KL}}(P\|Q)$: P和Q的KL-散度

复杂度

- $\mathcal{O}$：大O标记

Discussions[4]

[4] https://discuss.d2l.ai/t/2089

前言

时至今日，我们常用的计算机程序几乎都是软件开发人员从零编写的。比如，现在我们要编写一个程序来管理网上商城。经过思考，我们可能提出如下一个解决方案：首先，用户通过Web浏览器（或移动应用程序）与应用程序进行交互。紧接着，应用程序与数据库引擎进行交互，以保存交易历史记录并跟踪每个用户的动态。其中，这个程序的核心——"业务逻辑"，详细说明了程序在各种情况下进行的操作。

为了完善业务逻辑，我们必须细致地考虑应用程序所有可能遇到的边界情况，并为这些边界情况设计合适的规则。当买家单击将商品添加到购物车时，我们会向购物车数据库表中添加一个条目，将该用户ID与商品ID关联起来。虽然一次编写出完美应用程序的可能性微乎其微，但在大多数情况下，我们可以从基本原理出发编写这样的程序，并不断测试直到满足用户的需求。我们能够根据第一原则设计自动化系统，驱动正常运行的产品和系统，是一个人类认知上的非凡壮举。

幸运的是，对于日益壮大机器学习科学家群体来说，实现很多任务的自动化并不再屈从于人类的聪明才智。想象一下，你正和你最聪明一群朋友围绕着白板，试图解决以下问题之一：

- 编写一个程序，给出地理信息、卫星图像和一些历史天气信息，来预测明天的天气。
- 编写一个程序，给出自然文本表示的问题，并正确回答该问题。
- 编写一个程序，给出一张图像，识别出图像所包含的人，并在每个人周围绘制轮廓。
- 编写一个程序，向用户推荐他们可能喜欢但在自然浏览过程中不太可能遇到的产品。

在这些情况下，即使是顶级程序员也无法从零开始，交上完美的解决方案。原因可能各不相同：有时，我们的任务可能遵循一种随着时间推移而变化的模式，我们需要程序来自动调整。有时，任务内的关系（比如像素和抽象类别之间的关系）可能太复杂，需要数千或数百万次的计算。即使我们的眼睛能毫不费力地完成任务，这些计算也超出了我们的意识理解。机器学习（machine learning，ML）是强大的可以从经验中学习的技术。通常采用观测数据或与环境交互的形式，机器学习算法会积累更多的经验，其性能也会逐步提高。相反，对比刚刚所说的电子商务平台，一直执行相同的业务逻辑，无论积累多少经验，都不会自动提高（直到开发人员认识到并更新软件）。在这本书中，我们将带你开启机器学习之旅，并特别关注深度学习（deep learning）的基础知识。这是一套强大的技术，它可以推动计算机视觉、自然语言处理、医疗保健和基因组学等不同领

域的创新。

日常生活中的机器学习

假设你正和本书的作者们一起，驱车去咖啡店。亚历山大拿起一部iPhone，对它说道“Hey Siri”–手机的语音识别系统主动唤醒了。接着，李沐对Siri说道“去星巴克咖啡店”–语音识别系统自动触发语音转文字功能，并启动地图应用程序来满足我们的请求。地图应用程序在启动后确定了若干条路线：每条路线都显示了预计的通行时间……由此可见，机器学习渗透在生活中的方方面面，在短短几秒钟的时间里，我们与智能手机的日常互动就可以涉及几种机器学习模型。

现在，请你从基本原则出发，编写一个程序来响应一个“唤醒词“（比如“Alexa”、“小爱同学”和“Hey Siri”）。试着用一台计算机和一个代码编辑器自己编写代码，如 图1 中所示。问题看似很难解决：麦克风每秒钟将收集大约44000个样本，每个样本都是声波振幅的测量值。如何编写程序，令其输入原始音频片段，输出{是, 否}（表示该片段是否包含唤醒词）的可靠预测呢？如果你毫无头绪，别担心，我们也不知道如何从头开始编写这个程序。这就是我们需要机器学习的原因。

图1: 识别唤醒词

显然，即使我们不知道如何编程从输入到输出的映射，人类大脑认知仍然能够执行这个任务。换句话说，即使你不知道如何编写程序来识别单词“Alexa”，你的大脑却能够轻易识别它。有了这一能力，我们就可以收集一个包含音频样本的巨大的数据集（dataset），并对包含和不包含唤醒词的样本进行标记。通过机器学习算法，我们不需要设计一个“明确地”识别唤醒词的系统。相反，我们定义一个灵活的程序算法，其输出由许多参数（parameter）决定。然后我们使用数据集来确定当下的“最佳参数集”，这些参数通过某种性能度量来获取完成任务的最佳性能。

那么到底什么是参数呢？你可以把参数看作是旋钮，我们可以转动旋钮来调整程序的行为。任一调整参数的程序后，我们称为模型（model）。通过操作参数而生成的所有不同程序（输入-输出映射）的集合称为“模型族”。使用数据集来选择参数的元程序被称为学习算法（learning algorithm）。

在我们开始用机器学习算法解决问题之前，我们必须精确地定义问题，确定输入和输出的性质，并选择合适的模型族。在本例中，我们的模型接收一段音频作为输入（input），然后模型生成{是, 否}中的输出（output）。如果一切顺利，经过一番训练，模型对于“片段是否包含唤醒词“的预测通常是正确的。

现在我们的模型每次听到“Alexa”这个词时都会发出“是”的声音。由于这里的唤醒词是任意选择的自然语言，因此我们可能需要一个足够丰富的模型族，使模型多元化，。比如，模型族的另一个模型只在听到“Hey Siri”这个词时发出“是”。理想情况下，同一个模型族应该适合于“Alexa”识别和“Hey Siri”识别，因为它们似乎是相似的任务。相反，如果我们想处理完全不同的输入或输出，比如从图像映射到字幕，或从英语映射到中文，我们可能需要一个完全不同的模型族。

正如你可能猜到的，如果我们只是随机设置模型参数，所以这个模型不太可能识别出“Alexa”、“Hey Siri”或任何其他单词。在机器学习中，学习（learning）是一个模型的训练过程。通过这个过程，我们可以发现正确的参数集，从而从使模型强制执行所需的行为。换句话说，我们用数据训练（train）我们的模型。如 图2所示，训练过程通常包含如下步骤：

1. 从一个随机初始化参数的模型开始，这个模型基本毫不”智能“。
2. 获取一些数据样本（例如，音频片段以及对应的{是, 否}标签）。
3. 调整参数，使模型在这些样本中表现得更好。
4. 重复第2步和第3步，直到模型在任务中的表现令你满意。

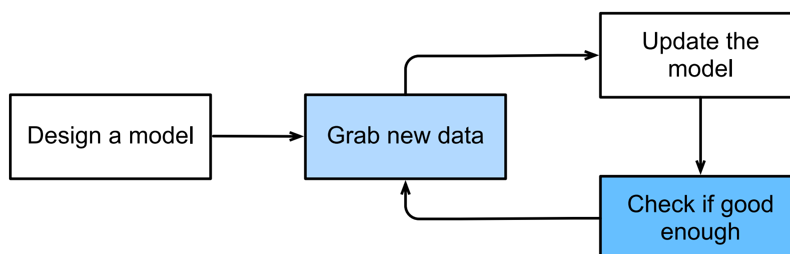


图2: 一个典型的训练过程

总而言之，我们没有编写唤醒词识别器，而是编写了一个“学习”程序。如果我们用一个巨大的带标签的数据集，它很可能可以“学习”识别唤醒词。你可以将这种”通过用数据集来确定程序行为”的方法看作是“用数据编程”（programming with data）。比如，我们可以通过向机器学习系统提供许多猫和狗的图片来设计一个“猫图检测器”。通过这种方式，检测器最终可以学会：如果输入是猫的图片就输出一个非常大的正数，如果输入是狗的图片就会得出一个非常大的负数。如果检测器不确定，它会输出接近于零的数……这个例子仅仅是机器学习常见应用的冰山一角。而深度学习是机器学习的一个主要分之，我们稍后将对其进行更详细的解析。

关键组件

首先，我们想让大家更清楚地了解一些核心组件。无论我们遇到什么类型的机器学习问题，这些组件都将伴随我们左右：

1. 我们可以学习的数据（data）。
2. 如何转换数据的模型（model）。
3. 一个目标函数（objective function），用来量化模型的有效性。
4. 调整模型参数以优化目标函数的算法。

数据

毋庸置疑，如果没有数据，那么数据科学毫无用武之地。每个数据集由一个个样本（example）组成，大多时候，它们遵循独立同分布(idependently and identically distributed, i.i.d.)。样本有时也叫做数据点（data point）或者数据实例（data instance），通常每个样本由一组称为特征（features，或协变量（covariates））的属性组成。机器学习模型会根据这些属性进行预测。在上面的监督学习问题中，要预测的是一个特殊的属性，它被称为标签（label，或目标（target））。

假设我们处理的是图像数据，每一张单独的照片即为一个样本，它的特征由每个像素数值的有序列表表示。比如，200×200彩色照片由$200 \times 200 \times 3 = 120000$个数值组成，其中的"3"对应于每个空间位置的红、绿、蓝通道的强度。再比如，对于一组医疗数据，给定一组标准的特征(如年龄、生命体征和诊断)，我们可能用此数据尝试预测患者是否会存活。

当每个样本的特征类别数量都是相同的，所以其特征向量是固定长度的，这个长度被称为数据的维数（dimensionality）。固定长度的特征向量是一个方便的属性，它有助于我们量化学习大量样本。

然而，并不是所有的数据都可以用"固定长度"的向量表示。以图像数据为例，如果它们全部来自标准显微镜设备，那么"固定长度"是可取的；但是如果我们的图像数据来自互联网，我们不能天真的假想它们都有相同的分辨率或形状。这时，我们可以考虑将图像裁剪成标准尺寸，但这种办法很局限，数据有丢失信息的风险。此外，文本数据更不符合"固定长度"的要求。考虑一下亚马逊等电子商务网站上的客户评论：有些文本数据是简短的（比如"好极了"）；有些则长篇大论。与传统机器学习方法相比，深度学习的一个主要优势是可以处理不同长度的数据。

一般来说，我们拥有的数据越多，我们的工作就越容易。当我们有了更多的数据，我们通常可以训练出更强大的模型，从而减少对预先设想假设的依赖。数据集的由小变大为现代深度学习的成功奠定基础。在没有大数据集的情况下，许多令人兴奋的深度学习模型黯然失色。就算一些深度学习模型在小数据集上能够工作，但其效能并不比不上传统方法。

请注意，仅仅拥有海量的数据是不够的，我们还需要正确的数据。如果数据中充满了错误，或者如果数据的特征不能预测任务目标，那么模型很可能无效。有一句古语很好地反映了这个现象："输入的是垃圾,输出的也是垃圾。"（"Garbage in, garbage out."）此外，糟糕的预测性能甚至会加倍放大事态的严重性。在一些敏感应用中，如预测性监管、简历筛选和用于贷款的风险模型，我们必须特别警惕垃圾数据的后果。一种常见的问题来着不均衡的数据集，比如在一个有关医疗的训练数据集中，某些人群没有样本表示。想象一下，假设你要训练一个皮肤癌识别模型，但它（在训练数据集）从未见过的黑色皮肤的人群，就会顿时束手无策。

再比如，如果用"过去的招聘决策数据"来训练一个筛选简历的模型，那么机器学习模型可能会无意中捕捉到历史残留的不公正，并将其自动化。然而，这一切都可能在不知情的情况下发生。因此，当数据不具有充分代表性，甚至包含了一些社会偏见时，模型就很有可能失败。

模型

大多数机器学习会涉及到数据的转换。比如，我们建立一个"摄取照片并预测笑脸"的系统。再比如，我们摄取一组传感器读数，并预测读数的正常与异常程度。虽然简单的模型能够解决如上简单的问题，但本书中关注的问题超出了经典方法的极限。深度学习与经典方法的区别主要在于：前者关注的功能强大的模型，这些模型由神经网络错综复杂的交织在一起，包含层层数据转换，因此被称为深度学习（deep learning）。在讨论深度模型的过程中，我们也将提及一些传统方法。

目标函数

前面，我们将机器学习介绍为"从经验中学习"。这里所说的"学习"，是指自主提高模型完成某些任务的效能。但是，什么才算真正的提高呢？在机器学习中，我们需要定义模型的优劣程度的度量，这个度量在大多数情况是"可优化"的，我们称之为目标函数（objective function）。我们通常定义一个目标函数，并希望优化它到最低点。因为越低越好，所以这些函数有时被称为损失函数（loss function, 或cost function）。但这只是一个惯例，你也可以取一个新的函数，优化到它的最高点。这两个函数本质上是相同的，只是翻转一下符号。

当任务为试图预测数值时，最常见的损失函数是平方误差（squared error），即预测值与实际值之差的平方。当试图解决分类问题时，最常见的目标函数是最小化错误率，即预测与实际情况不符的样本比例。有些目标函数（如平方误差）很容易被优化，有些目标（如错误率）由于不可微性或其他复杂性难以直接优化。在这些情况下，通常会优化替代目标。

通常，损失函数是根据模型参数定义的，并取决于数据集。在一个数据集上，我们通过最小化总损失来学习模型参数的最佳值。该数据集由一些为训练而收集的样本组成，称为训练数据集（training dataset，或称为训练集（training set））。然而，在训练数据上表现良好的模型，并不一定在"新数据集"上有同样的效能，这里的"新数据集"通常称为测试数据集（test dataset，或称为测试集（test set））。

综上所述，我们通常将可用数据集分成两部分：训练数据集用于拟合模型参数，测试数据集用于评估拟合的模型。然后我们观察模型在这两部分数据集的效能。你可以把"一个模型在训练数据集上的效能"想象成"一个学生在模拟考试中的分数"。这个分数用来为一些真正的期末考试做参考，即使成绩令人鼓舞，也不能保证期末考试成功。换言之，测试性能可能会显著偏离训练性能。当一个模型在训练集上表现良好，但不能推广到测试集时，我们说这个模型是"过拟合"（overfitting）的。就像在现实生活中，尽管模拟考试考得很好，真正的考试不一定百发百中。

优化算法

一旦我们获得了一些数据源及其表示、一个模型和一个合适的损失函数，我们接下来就需要一种算法，它能够搜索出最佳参数，以最小化损失函数。深度学习中，大多流行的优化算法通常基于一种基本方法–梯度下降（gradient descent）。简而言之，在每个步骤中，梯度下降法都会检查每个参数，看看如果你仅对该参数进行少量变动，训练集损失会朝哪个方向移动。然后，它在可以减少损失的方向上优化参数。

各种机器学习问题

在机器学习的广泛应用中，唤醒词问题只是冰山一角。在前面的例子中，只是机器学习可以解决的众多问题中的一个。下面，我们将列出一些常见机器学习问题和应用，为之后本书的讨论做铺垫。我们将不断引用前面提到的概念，如数据、模型和训练技术。

监督学习

监督学习（supervised learning）擅长在"给定输入特征"的情况下预测标签。每个"特征-标签"对都称为一个样本（example）。有时，即使标签是未知的，样本也可以指代输入特征。我们的目标是生成一个模型，能够将任何输入特征映射到标签，即预测。

举一个具体的例子。假设我们需要预测患者是否会心脏病发作，那么观察结果"心脏病发作"或"心脏病没有发作"将是我们的标签。输入特征可能是生命体征，如心率、舒张压和收缩压。

监督学习之所以发挥作用，是因为在训练参数时，我们为模型提供了一个数据集，其中每个样本都有真实的标签。用概率论术语来说，我们希望预测"估计给定输入特征的标签"的条件概率。虽然监督学习只是几大类机器学习问题之一，但是在工业中，大部分机器学习的成功应用都是监督学习。这是因为在一定程度上，许多重要的任务可以清晰地描述为：在给定一组特定的可用数据的情况下，估计未知事物的概率。比如：

- 根据计算机断层扫描（CT）图像预测是否为癌症。
- 给出一个英语句子，预测正确的法语翻译。
- 根据本月的财务报告数据预测下个月股票的价格。

非正式地说，监督学习的学习过程如下所示。首先，从已知大量数据样本中随机选取一个子集，为每个样本获取基本的真实标签。有时，这些样本已有标签（例如，患者是否在下一年内康复?）；有时，我们可能需要人工标记数据（例如，将图像分类）。这些输入和相应的标签一起构成了训练数据集。随后，我们选择有监督的学习算法，它将训练数据集作为输入，并输出一个"完成学习模型"。最后，我们将之前没见过的样本特征放到这个"完成学习模型"中，使用模型的输出作为相应标签的预测。整个监督学习过程在 图3 中绘制。

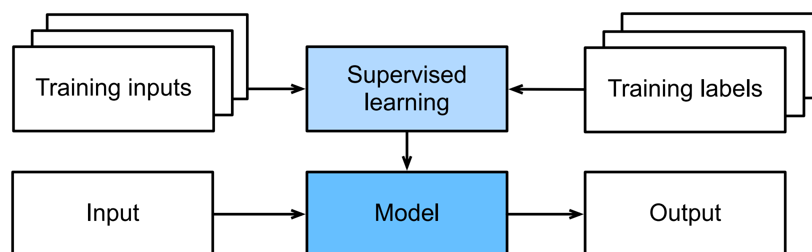


图3: 监督学习

综上所述，即使使用简单的描述"给定输入特征的预测标签"，监督学习也可以采取多种形式的模型，并且需要大量不同的建模决策，这取决于输入和输出的类型、大小和数量。例如，在处理"任意长度的序列"或者"固定长度的向量表示"时，我们可以使用不同的模型。我们将在本书中深入探讨这些问题。

回归

回归（regression）是最简单的监督学习任务之一。比方说，假设我们有一组房屋销售数据表格，其中每行对应于每个房子，每列对应于一些相关的属性，例如房屋的面积、卧室的数量、浴室的数量以及到镇中心的步行分钟数等等。对机器学习来说，每个样本即为一个特定的房屋，相应的特征向量将是表中的一行。如果你住在纽约或旧金山，而且你不是亚马逊、谷歌、微软或Facebook的首席执行官，那么你家中的（平方英尺、卧室数量、浴室数量、步行距离）特征向量可能类似于：$[600, 1, 1, 60]$。然而，如果你住在匹兹堡，这个特征向量可能看起来更像$[3000, 4, 3, 10]$……为什么这个任务可以归类于回归问题呢？本质上是输出决定的。假设你在市场上寻找新房子，你可能需要估计一栋房子的公平市场价值。销售价格，即标签，是一个数值。当标签取任意数值时，我们称之为回归问题。我们的目标是生成一个模型，它的预测非常接近实际标签值。

生活中的许多问题都可归类于回归问题。比如，预测用户对一部电影的评分可以被认为是一个回归问题。这里一个小插曲，如果你在2009年设计了一个很棒的算法来预测电影评分，你可能会赢得100万美元的奈飞奖[5]。再比如，预测病人在医院的住院时间也是一个回归问题。总而言之，判断回归问题的一个很好的经验法则是，任何有关“多少”的问题很可能就是回归问题。比如：

- 这个手术需要多少小时？
- 在未来六小时，这个镇会有多少降雨量？

你可能发现，即使你以前从未使用过机器学习，可能在不经意间，你已经解决了一些回归问题。例如，你让人修理了排水管，你的承包商花了3个小时清除污水管道中的污物，然后他寄给你一张350美元的账单。而你的朋友雇了同一个承包商两个小时，他收到了250美元的账单。如果有人请你估算的清理污物的费用，你可以假设承包商有一些基本费用，然后按小时收费。如果这些假设成立，那么给出这两个数据样本，你就已经可以确定承包商的定价结构：每小时100美元，外加50美元上门服务费。你看，在不经意间，你就已经理解并应用了线性回归的本质。

以上假设有时这是不可取。例如，如果一些差异是由于两个特征之外的几个因素造成的。在这些情况下，我们将尝试学习最小化”预测值和实际标签值的差异“的模型。在本书大部分章节中，我们将关注最小化平方误差损失函数。正如我们稍后将看到的，这种损失对应于我们的数据被高斯噪声破坏的假设。

分类

虽然回归模型可以很好地解决“有多少？“的问题，但是很多问题并非如此。例如，一家银行希望在其移动应用程序中添加支票扫描功能。具体地说，这款应用程序需要能够自动理解照片图像中看到的文本，并将手写字符映射到已知字符之一。这种“哪一个？”的问题叫做分类（classification）问题。在分类问题中，我们希望模型能够预测样本属于哪个类别（category，正式称为类（class））。例如，对于手写数字，我们可能有10类，分别数字0到9。最简单的分类问题是只有两类，我们称之为“二元分类”。例如，数据集可能由动物图像组成，标签可能是{⊠, ⊠}⊠⊠。在回归中，我们训练一个回归函数来输出一个数值；而在分类中，我们训练一个分类器，它的输出即为预测的类别。

然而模型怎么判断得出这种“是”或“不是”的硬分类预测呢？我们可以试着用概率语言来理解模型。给定一个样本特征，我们的模型为每个可能的类分配一个概率。比如，之前的猫狗分类例子中，分类器可能会输出图像是猫的概率为0.9。0.9这个数字表达什么意思呢？我们可以这样解释：分类器90%确定图像描绘的是一

[5] https://en.wikipedia.org/wiki/Netflix_Prize

只猫。预测类别的概率的大小传达了一种模型的不确定性，我们将在后面章节中讨论其他运用不确定性概念的算法。

当我们有两个以上的类别时，我们把这个问题称为多类分类（multiclass classification）问题。常见的例子包括手写字符识别 $\{0, 1, 2, ...9, a, b, c, ...\}$。与解决回归问题不同，分类问题的常见损失函数被称为交叉熵（cross-entropy），我们将在后面的章节中详细阐述。

请注意，最常见的类别不一定是你将用于决策的类别。举个例子，假设你在后院发现了一个美丽的蘑菇，如图4 所示。

图4: 死帽蕈——不能吃!!

现在，请你训练一个毒蘑菇检测分类器，根据照片预测蘑菇是否有毒。假设这个分类器输出 图4 包含死帽蕈的概率是0.2。换句话说，分类器80%确定我们的蘑菇不是死帽蕈。尽管如此，我们也不会吃它，因为我们不值得冒20%的死亡风险。换句话说，不确定风险的影响远远大于收益。因此，我们需要将“预期风险”作为损失函数。也就是说，我们需要将结果的概率乘以与之相关的收益（或伤害）。在这种情况下，食用蘑菇造成的损失为 $0.2 \times \infty + 0.8 \times 0 = \infty$，而丢弃蘑菇的损失为$0.2 \times 0 + 0.8 \times 1 = 0.8$。我们的谨慎是有道理的：正如任何真菌学家都会告诉我们的那样，图4 中的蘑菇实际上是一个死帽蕈。

分类可能变得比二元分类、多类分类复杂得多。例如，有一些分类任务的变体可以用于寻找层次结构，层次结构假定在许多类之间存在某种关系。因此，并不是所有的错误都是均等的。我们宁愿错误地分入一个相关的类别，也不愿错误地分入一个遥远的类别，这通常被称为层次分类(hierarchical classification)。早期的一个例子是卡尔·林耐[6]人，他们把动物组织分类成等级制。

在动物分类的应用中，把一只狮子狗（一种狗的品种）误认为雪纳瑞（另一种狗的品种）可能不会太糟糕。但如果我们的模型将狮子狗与恐龙混淆，就滑稽至极了。层次结构相关性可能取决于你计划如何使用模型。例如，响尾蛇和乌梢蛇血缘上可能很接近，但如果把响尾蛇误认为是乌梢蛇可能会是致命的。因为响尾蛇是有毒的，而乌梢蛇是无毒的。

[6] https://en.wikipedia.org/wiki/Carl_Linnaeus

标记问题

有些分类问题很适合于二元分类或多类分类。例如，我们可以训练一个普通的二元分类器来区分猫和狗。运用最前沿的计算机视觉的算法，我们可以轻松地训练这个模型。尽管如此，无论我们的模型有多精确，当分类器遇到新的动物时可能会束手无策。比如这张“不来梅的城市音乐家”的图像 图5 （这是一个流行的德国童话故事），图中有一只猫，一只公鸡，一只狗，一头驴，背景是一些树。取决于我们最终想用我们的模型做什么，将其视为二元分类问题可能没有多大意义。取而代之，我们可能想让模型描绘输入图像的内容，一只猫、一只狗、一头驴，还有一只公鸡。

图5: 一头驴，一只狗，一只猫和一只公鸡

学习预测不相互排斥的类别的问题称为多标签分类（multilabel classification）。举个例子，人们在技术博客上贴的标签，比如“机器学习”、“技术”、“小工具”、“编程语言”、“Linux”、“云计算”、“AWS”。一篇典型的文章可能会用5-10个标签，因为这些概念是相互关联的。关于“云计算”的帖子可能会提到“AWS”，而关于“机器学习”的帖子也可能涉及“编程语言”。

此外，在处理生物医学文献时，我们也会遇到这类问题。正确地标记文献很重要，有利于研究人员对文献进行详尽的审查。在国家医学图书馆，一些专业的注释员会检查每一篇在PubMed中被索引的文章，以便将其与Mesh中的相关术语相关联（Mesh是一个大约有28000个标签的集合）。这是一个十分耗时的过程，注释器通常在归档和标记之间有一年的延迟。这里，机器学习算法可以提供临时标签，直到每一篇文章都有严格的人工审核。事实上，近几年来，BioASQ组织已经举办比赛[7]来完成这项工作。

[7] http://bioasq.org/

搜索

有时，我们不仅仅希望输出为一个类别或一个实值。在信息检索领域，我们希望对一组项目进行排序。以网络搜索为例，我们的目标不是简单的“查询（query）-网页（page）”分类，而是在海量搜索结果中找到用户最需要的那部分。搜索结果的排序也十分重要，我们的学习算法需要输出有序的元素子集。换句话说，如果要求我们输出字母表中的前5个字母，返回“A、B、C、D、E”和“C、A、B、E、D”是不同的。即使结果集是相同的，集内的顺序有时却很重要。

该问题的一种可能的解决方案：首先为集合中的每个元素分配相应的相关性分数，然后检索评级最高的元素。PageRank[8]，谷歌搜索引擎背后最初的秘密武器就是这种评分系统的早期例子，但它的奇特之处在于它不依赖于实际的查询。在这里，他们依靠一个简单的相关性过滤来识别一组相关条目，然后根据PageRank对包含查询条件的结果进行排序。如今，搜索引擎使用机器学习和用户行为模型来获取网页相关性得分，很多学术会议也致力于这一主题。

推荐系统

另一类与搜索和排名相关的问题是推荐系统（recommender system），它的目标是向给特定用户进行“个性化”推荐。例如，对于电影推荐，科幻迷和喜剧爱好者的推荐结果页面可能会有很大不同。类似的应用也会出现在零售产品、音乐和新闻推荐等等。

在某些应用中，客户会提供明确反馈，表达他们对特定产品的喜爱程度。例如，亚马逊上的产品评级和评论。在其他一些情况下，客户会提供隐性反馈。例如，某用户跳过播放列表中的某些歌曲，这可能说明歌曲对此用户不大合适。总的来说，推荐系统会为“给定用户和物品”的匹配性打分，这个“分数”可能是估计的评级或购买的概率。由此，对于任何给定的用户，推荐系统都可以检索得分最高的对象集，然后将其推荐给用户。以上只是简单的算法，而工艺生产的推荐系统要先进得多，它会将详细的用户活动和项目特征考虑在内。推荐系统算法经过调整，可以捕捉一个人的偏好。比如，图6 是亚马逊基于个性化算法推荐的深度学习书籍，成功的捕捉了作者的喜好。

[8] https://en.wikipedia.org/wiki/PageRank

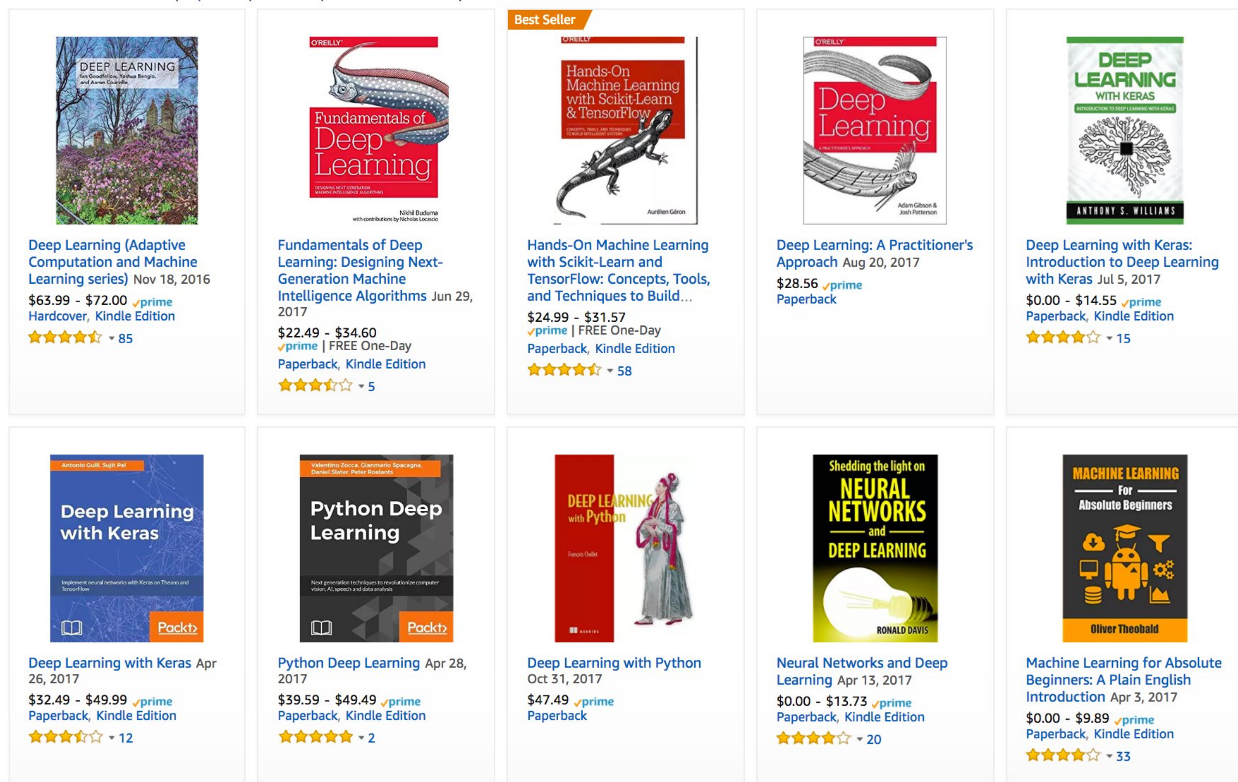


图6: 亚马逊推荐的深度学习书籍

尽管推荐系统具有巨大的应用价值，但单纯用它作为预测模型仍存在一些缺陷。首先，我们的数据只包含“审查后的反馈”：用户更倾向于给他们感觉强烈的事物打分。例如，在五分制电影评分中，会有许多五星级和一星级评分，但三星级却明显很少。此外，推荐系统有可能形成反馈循环：推荐系统首先会优先推送一个购买量较大（可能被认为更好）的商品，然而目前用户的购买习惯往往是遵循推荐算法，但学习算法并不总是考虑到这一细节，进而更频繁地被推荐。综上所述，关于如何处理审查、激励和反馈循环的许多问题，都是重要的开放性研究问题。

序列学习

以上大多问题都具有固定大小的输入和产生固定大小的输出。例如，在预测房价的问题中，我们考虑从一组固定的特征：平方英尺、卧室数量、浴室数量、步行到市中心的时间；图像分类问题中，输入为固定尺寸的图像，输出则为固定数量（有关每一个类别）的预测概率；在这些情况下，模型只会将输入作为生成输出的“原料”，而不会“记住”输入的具体内容。

如果输入的样本之间没有任何关系，以上模型可能完美无缺。但是如果输入是连续的，我们的模型可能就需要拥有“记忆”功能了。比如，我们该如何处理视频片段呢？在这种情况下，每个代码段可能由不同数量的帧组成。通过前一帧的图像，我们可能对后一帧中发生的事情的更有把握。语言也是如此，机器翻译的输入和输出都为文字序列。

再比如，在医学上序列输入和输出就更为重要。设想一下，假设我们用一个模型来监控重症监护病人，如果

他们在未来24小时内死亡的风险超过某个阈值，这个模型就会发出警报。我们绝不希望抛弃过去每小时有关病人病史的所有信息，而仅根据最近的测量结果做出预测。

这些问题是序列学习的实例，是机器学习最令人兴奋的应用之一。序列学习需要摄取输入序列或预测输出序列，或两者兼而有之。具体来说，输入和输出都是可变长度的序列，例如机器翻译和从语音中转录文本。虽然不可能考虑所有类型的序列转换，但以下特殊情况值得一提。

标记和解析。这涉及到用属性注释文本序列。换句话说，输入和输出的数量基本上是相同的。例如，我们可能想知道动词和主语在哪里，或者，我们可能想知道哪些单词是命名实体。通常，目标是基于结构和语法假设对文本进行分解和注释，以获得一些注释。这听起来比实际情况要复杂得多。下面是一个非常简单的示例，它使用标记来注释一个句子，该标记指示哪些单词引用命名实体(标记为“Ent”，是实体（entity）的简写)。

```
Tom has dinner in Washington with Sally
Ent  -    -     -     Ent      -    Ent
```

自动语音识别。在语音识别中，输入序列是说话人的录音（如 图7 所示），输出序列是说话人所说内容的文本记录。它的挑战在于，与文本相比，音频帧多得多（声音通常以8kHz或16kHz采样）。也就是说，音频和文本之间没有1:1的对应关系，因为数千个样本可能对应于一个单独的单词。这也是“序列到序列”的学习问题，其中输出比输入短得多。

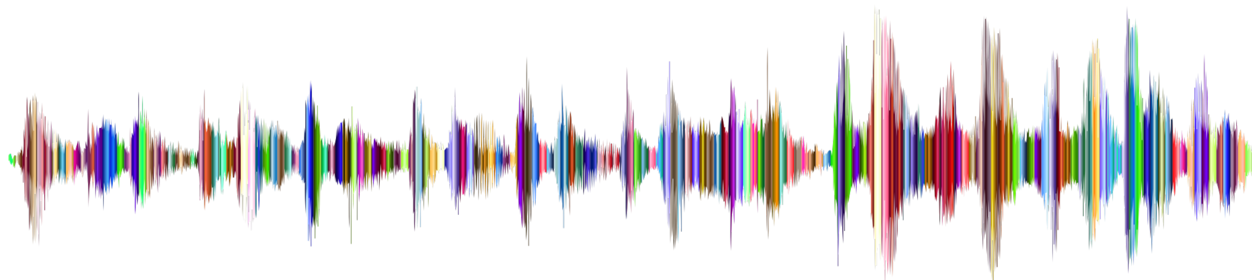


图7: -D-e-e-p- L-ea-r-ni-ng- 在录音中。

文本到语音。这与自动语音识别相反。换句话说，输入是文本，输出是音频文件。在这种情况下，输出比输入长得多。虽然人类很容易识判断发音别扭的音频文件，但这对计算机来说并不是那么简单。

机器翻译。在语音识别中，输入和输出的出现顺序基本相同。而在机器翻译中，颠倒输入和输出的顺序非常重要。换句话说，虽然我们仍将一个序列转换成另一个序列，但是输入和输出的数量以及相应序列的顺序大都不会相同。比如下面这个例子，“错误的对齐”反应了德国人喜欢把动词放在句尾的特殊倾向。

```
德语:          Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
英语:          Did you already check out this excellent tutorial?
错误的对齐:  Did you yourself already this excellent tutorial looked-at?
```

其他学习任务也有序列学习的应用。例如，确定“用户阅读网页的顺序”是二维布局分析问题。再比如，对话问题对序列的学习更为复杂：确定下一轮对话，需要考虑对话历史状态以及现实世界的知识……如上这些都是热门的序列学习研究领域。

无监督学习

到目前为止，所有的例子都与监督学习有关，即我们向模型提供巨大数据集：每个样本包含特征和相应标签值。打趣一下，“监督学习”模型像一个打工仔，有一份极其专业的工作和一位极其平庸的老板。老板站在身后，准确地告诉模型在每种情况下应该做什么，直到模型学会从情况到行动的映射。取悦这位老板很容易，只需尽快识别出模式并模仿他们的行为即可。

相反，如果你的工作没有十分具体的目标，你就需要“自发”地去学习了。（如果你打算成为一名数据科学家，你最好培养这个习惯。）比如，你的老板可能会给你一大堆数据，然后让你用它做一些数据科学研究，却没有对结果要求。我们称这类数据中不含有“目标”的机器学习问题为无监督学习（unsupervised learning），我们将在后面的章节中讨论无监督学习技术。那么无监督学习可以回答什么样的问题呢？我们来看看下面的例子：

- 聚类（clustering）问题：没有标签的情况下，我们是否能给数据分类呢？比如，给定一组照片，我们能把它们分成风景照片、狗、婴儿、猫和山峰的照片吗？同样，给定一组用户的网页浏览记录，我们能否将具有相似行为的用户聚类吗？
- 主成分分析（principal component analysis）问题：我们能否找到少量的参数来准确地捕捉数据的线性相关属性?比如，一个球的运动轨迹可以用球的速度、直径和质量来描述。再比如，裁缝们已经开发出了一小部分参数，这些参数相当准确地描述了人体的形状，以适应衣服的需要。另一个例子：在欧几里得空间中是否存在一种(任意结构的)对象的表示，使其符号属性能够很好地匹配?这可以用来描述实体及其关系，例如“罗马”－“意大利”＋“法国”＝“巴黎”。
- 因果关系（causality）和概率图模型（probabilistic graphical models）问题：我们能否描述观察到的许多数据的根因?例如，如果我们有关于房价、污染、犯罪、地理位置、教育和工资的人口统计数据，我们能否简单地根据经验数据发现它们之间的关系？
- 生成对抗性网络（generative adversarial networks）：为我们提供一种合成数据的方法，甚至像图像和音频这样复杂的结构化数据。潜在的统计机制是检查真实和虚假数据是否相同的测试，它是无监督学习的另一个重要而令人兴奋的领域。

与环境互动

你可能一直心存疑虑：机器学习的输入（数据）来自哪里？机器学习的输出又将去往何方？到目前为止，不管是监督学习还是无监督学习，我们都会预先获取大量数据，然后启动模型，不再与环境交互。这里所有学习都是在算法与环境断开后进行的，被称为离线学习（offline learning）。对于监督学习，从环境中收集数据的过程类似于 图8。

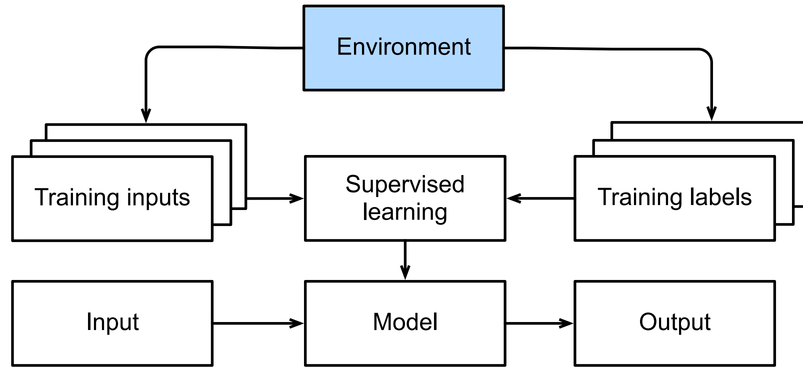


图8: 从环境中为监督学习收集数据。

这种简单的离线学习有它的魅力。好的一面是，我们可以孤立地进行模式识别，而不必分心于其他问题。但缺点是，解决的问题相当有限。如果你更有雄心壮志，那么你可能会期望人工智能不仅能够做出预测，而且能够与真实环境互动。与预测不同，“与真实环境互动”实际上会影响环境。这里的人工智能是“智能代理”，而不仅是“预测模型”。因此，我们必须考虑到它的行为可能会影响未来的观察结果。

考虑“与真实环境互动”将打开一整套新的建模问题。以下只是几个例子：

- 环境还记得我们以前做过什么吗？
- 环境是否有助于我们建模？例如，用户将文本读入语音识别器。
- 环境是否想要打败模型？例如，一个对抗性的设置，如垃圾邮件过滤或玩游戏？
- 环境是否重要？
- 环境是否变化？例如，未来的数据是否总是与过去相似，还是随着时间的推移会发生变化？是自然变化还是响应我们的自动化工具而发生变化？

最后一个问题提出了当训练和测试数据不同时数据分布偏移（distribution shift）的问题。接下来，我们将简要描述强化学习问题，这是一类明确考虑与环境交互的问题。

强化学习

如果你对使用机器学习开发与环境交互并采取行动感兴趣，那么你最终可能会专注于强化学习（reinforcement learning）。这可能包括应用到机器人、对话系统，甚至开发视频游戏的人工智能(AI)。深度强化学习（deep reinforcement learning）将深度学习应用于强化学习的问题，是非常热门的研究领域。突破性的深度Q网络（Q-network）在雅达利游戏中仅使用视觉输入就击败了人类，以及 AlphaGo 程序在棋盘游戏围棋中击败了世界冠军，是两个突出强化学习的例子。

在强化学习问题中，agent 在一系列的时间步骤上与环境交互。在每个特定时间点，agent 从环境接收一些观察（observation），并且必须选择一个动作（action），然后通过某种机制（有时称为执行器）将其传输回环境，最后 agent 从环境中获得 奖励（reward）。此后新一轮循环开始，agent 接收后续观察，并选择后续操作，依此类推。强化学习的过程在 图9 中进行了说明。请注意，强化学习的目标是产生一个好的策略（policy）。强化学习 agent 的选择的”动作“受策略控制，即一个从环境观察映射到行动的功能。

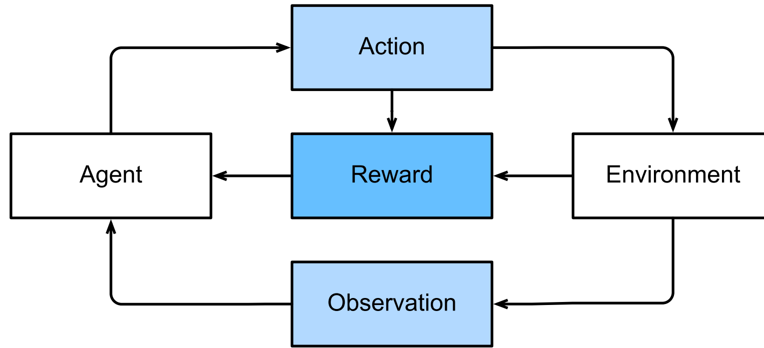


图9: 强化学习和环境之间的相互作用

强化学习框架的通用性十分强大。例如，我们可以将任何监督学习问题转化为强化学习问题。假设我们有一个分类问题，我们可以创建一个强化学习agent，每个分类对应一个“动作”。然后，我们可以创建一个环境，该环境给予agent的奖励。这个奖励与原始监督学习问题的损失函数是一致的。

当然，强化学习还可以解决许多监督学习无法解决的问题。例如，在监督学习中，我们总是希望输入与正确的标签相关联。但在强化学习中，我们并不假设环境告诉agent每个观测的最优动作。一般来说，agent只是得到一些奖励。此外，环境甚至可能不会告诉我们是哪些行为导致了奖励。

以强化学习在国际象棋的应用为例。唯一真正的奖励信号出现在游戏结束时：当agent获胜时，agent可以得到奖励1；当agent失败时，agent将得到奖励-1。因此，强化学习者必须处理学分分配（credit assignment）问题：决定哪些行为是值得奖励的，哪些行为是需要惩罚的。就像一个员工升职一样，这次升职很可能反映了前一年的大量的行动。要想在未来获得更多的晋升，就需要弄清楚这一过程中哪些行为导致了晋升。

强化学习可能还必须处理部分可观测性问题。也就是说，当前的观察结果可能无法阐述有关当前状态的所有信息。比方说，一个清洁机器人发现自己被困在一个许多相同的壁橱的房子里。推断机器人的精确位置(从而推断其状态)，需要在进入壁橱之前考虑它之前的观察结果。

最后，在任何时间点上，强化学习agent可能知道一个好的策略，但可能有许多更好的策略从未尝试过的。强化学习agent必须不断地做出选择：是应该利用当前最好的策略，还是探索新的策略空间（放弃一些短期回报来换取知识）。

一般的强化学习问题是一个非常普遍的问题。agent的动作会影响后续的观察，而奖励只与所选的动作相对应。环境可以是完整观察到的，也可以是部分观察到的,解释所有这些复杂性可能会对研究人员要求太高。此外，并不是每个实际问题都表现出所有这些复杂性。因此，学者们研究了一些特殊情况下的强化学习问题。

当环境可被完全观察到时，我们将强化学习问题称为马尔可夫决策过程（markov decision process）。当状态不依赖于之前的操作时，我们称该问题为上下文赌博机（contextual bandit problem）。当没有状态，只有一组最初未知回报的可用动作时，这个问题就是经典的多臂赌博机（multi-armed bandit problem）。

起源

为解决各式机器学习问题，深度学习提供了强大的工具。虽然许多深度学习方法都是最近的才有重大突破，但使用数据和神经网络编程的核心思想已经研究了几个世纪。事实上，人类长期以来就有分析数据和预测未来结果的愿望，而自然科学的大部分都植根于此。例如，伯努利分布是以雅各布•贝努利（1655–1705）[9]命名的。而高斯分布是由卡尔•弗里德里希•高斯（1777—1855）[10]发现的，他发明了最小均方算法，至今仍用于解决从保险计算到医疗诊断的许多问题。这些工具算法在自然科学中产生了一种实验方法——例如，电阻中电流和电压的欧姆定律可以用线性模型完美地描述。

即使在中世纪，数学家对估计（estimation）也有敏锐的直觉。例如，雅各布·克贝尔 (1460–1533)[11]的几何学书籍举例说明，通过平均16名成年男性的脚的长度，可以得出一英尺的长度。

图10: 估计一英尺的长度。

图10 说明了这个估计器是如何工作的。16名成年男子被要求脚连脚排成一行。然后将它们的总长度除以16，得到现在等于1英尺的估计值。这个算法后来被改进以处理畸形的脚——将拥有最短和最长脚的两个人送走，对其余的人取平均值。这是最早的修剪均值估计的例子之一。

随着数据的收集和可获得性，统计数据真正实现了腾飞。罗纳德·费舍尔（1890-1962）[12]对统计理论和在遗传

[9] https://en.wikipedia.org/wiki/JacobuBernoulli

[10] https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss

[11] https://www.maa.org/press/periodicals/convergence/mathematical-treasures-jacob-kobels-geometry

[12] https://en.wikipedia.org/wiki/Ronald_-Fisher

学中的应用做出了重大贡献。他的许多算法（如线性判别分析）和公式（如费舍尔信息矩阵）至今仍被频繁使用。甚至，费舍尔在1936年发布的虹膜数据集，有时仍然被用来解读机器学习算法。他也是优生学的倡导者，这提醒我们：使用数据科学虽然在道德上存在疑问，但是与数据科学在工业和自然科学中的生产性使用一样，有着悠久的历史。

机器学习的第二个影响来自克劳德·香农(1916–2001)[13]的信息论和艾伦·图灵（1912-1954）[14]的计算理论。图灵在他著名的论文《计算机器与智能》[Turing, 1950] 中提出了“机器能思考吗?”的问题。在他所描述的图灵测试中，如果人类评估者很难根据文本互动区分机器和人类的回答，那么机器就可以被认为是“智能的”。

另一个影响可以在神经科学和心理学中找到。其中，最古老的算法之一是唐纳德·赫布 (1904–1985)[15]开创性的著作《行为的组织》[Hebb & Hebb, 1949] 。他提出神经元通过积极强化学习，是Rosenblatt感知器学习算法的原型，被称为“赫布学习”。这个算法也为当今深度学习的许多随机梯度下降算法奠定了基础：强化期望行为和减少不良行为，以获得神经网络中参数的良好设置。

神经网络（neural networks）得名的原因是生物灵感。一个多世纪以来（追溯到1873年亚历山大·贝恩和1890年詹姆斯·谢林顿的模型），研究人员一直试图组装类似于相互作用的神经元网络的计算电路。随着时间的推移，对生物学的解释变得不再肤浅，但这个名字仍然存在。其核心是当今大多数网络中都可以找到的几个关键原则：

- 线性和非线性处理单元的交替，通常称为层(layers)。
- 使用链式规则(也称为反向传播(backpropagation))一次性调整网络中的全部参数。

在最初的快速发展之后，神经网络的研究从1995年左右一直开始停滞不前，直到到2005年才稍有起色。这主要是因为两个原因。首先，训练网络（在计算上）非常昂贵。在上个世纪末，随机存取存储器（RAM）非常强大，而计算能力却很弱。其次，数据集相对较小。事实上，费舍尔1932年的虹膜数据集是测试算法有效性的流行工具，而MNIST数据集的60000个手写数字的数据集被认为是巨大的。考虑到数据和计算的稀缺性，核方法（kernel method）、决策树（decision tree）和图模型（graph models）等强大的统计工具（在经验上）证明是更为优越的。与神经网络不同的是，这些算法不需要数周的训练，而且有很强的理论依据，可以提供可预测的结果。

深度学习之路

大约2010年开始，那些在计算上看起来不可行的神经网络算法变得热门起来，实际上是以下两点导致的。其一，随着互联网的公司的出现，为数亿在线用户提供服务，大规模数据集变得触手可及。另外，廉价又高质量的传感器、廉价的数据存储（克里德定律）以及廉价计算（摩尔定律）的普及，特别是GPU的普及，使大规模算力唾手可得。

这一点在 表1 中得到了说明。

[13] https://en.wikipedia.org/wiki/Claude_Shannon
[14] https://en.wikipedia.org/wiki/Alan_Turing
[15] https://en.wikipedia.org/wiki/Donald_O._Hebb

表1: 数据集vs计算机内存和计算能力

年代	数据规模	内存	每秒浮点运算
1970	100 （虹膜）	1 KB	100 KF (Intel 8080)
1980	1 K （波士顿房价）	100 KB	1 MF (Intel 80186)
1990	10 K （光学字符识别）	10 MB	10 MF (Intel 80486)
2000	10 M （网页）	100 MB	1 GF (Intel Core)
2010	10 G （广告）	1 GB	1 TF (Nvidia C2050)
2020	1 T （社交网络）	100 GB	1 PF (Nvidia DGX-2)

很明显，随机存取存储器没有跟上数据增长的步伐。与此同时，算力的增长速度已经超过了现有数据的增长速度。这意味着统计模型需要提高内存效率（这通常是通过添加非线性来实现的），同时由于计算预算的增加，能够花费更多时间来优化这些参数。因此，机器学习和统计的关注点从（广义的）线性模型和核方法转移到了深度神经网络。这也是为什么许多深度学习的中流砥柱，如多层感知机 [McCulloch & Pitts, 1943] 、卷积神经网络 [LeCun et al., 1998] 、长短期记忆网络 [Watkins & Dayan, 1992] 和Q学习 [Watkins & Dayan, 1992] ，在相当长一段时间处于相对休眠状态之后，在过去十年中被“重新发现”的原因之一。

最近十年，在统计模型、应用和算法方面的进展就像寒武纪大爆发。事实上，最先进的技术不仅仅是应用于几十年前的算法的可用资源的结果。下面列举了帮助研究人员在过去十年中取得巨大进步的想法（虽然只是触及了的皮毛）。

- 新的容量控制方法，如*dropout* [Srivastava et al., 2014]，有助于减轻过拟合的危险。这是通过在整个神经网络中应用噪声注入 [Bishop, 1995] 来实现的，出于训练目的，用随机变量来代替权重。
- 注意力机制解决了困扰统计学一个多世纪的问题：如何在不增加可学习参数的情况下增加系统的记忆和复杂性。研究人员通过使用只能被视为可学习的指针结构 [Bahdanau et al., 2014] 找到了一个优雅的解决方案。不需要记住整个文本序列(例如用于固定维度表示中的机器翻译)，所有需要存储的都是指向翻译过程的中间状态的指针。这大大提高了长序列的准确性，因为模型在开始生成新序列之前不再需要记住整个序列。
- 多阶段设计。例如，存储器网络 [Sukhbaatar et al., 2015] 和神经编程器-解释器 [Reed & DeFreitas, 2015]。它们允许统计建模者描述用于推理的迭代方法。这些工具允许重复修改深度神经网络的内部状态，从而执行推理链中的后续步骤，类似于处理器如何修改用于计算的存储器。
- 另一个关键的发展是生成对抗网络 [Goodfellow et al., 2014] 的发明。传统模型中，密度估计和生成模型的统计方法侧重于找到合适的概率分布和（通常是近似的）抽样算法。因此，这些算法在很大程度上受到统计模型固有灵活性的限制。生成式对抗性网络的关键创新是用具有可微参数的任意算法代替采样器。然后对这些数据进行调整，使得鉴别器（实际上是对两个样本的测试）不能区分假数据和真实数据。通过使用任意算法生成数据的能力，它为各种技术打开了密度估计的大门。驰骋的斑马 [Zhu et al., 2017] 和假名人脸 [Karras et al., 2017] 的例子都证明了这一进展。即使是业余的涂鸦者也可以根据描述场景布局的草图生成照片级真实图像（[Park et al., 2019] ）。
- 在许多情况下，单个GPU不足以处理可用于训练的大量数据。在过去的十年中，构建并行和分布式训练算法的能力有了显着提高。设计可伸缩算法的关键挑战之一是深度学习优化的主力——随机梯度下降，它依赖于相对较小的小批量数据来处理。同时，小批量限制了GPU的效率。因此，在1024个GPU上进

行训练，例如每批32个图像的小批量大小相当于总计约32000个图像的小批量。最近的工作，首先是由 [Li, 2017] 完成的，随后是 [You et al., 2017] 和 [Jia et al., 2018]，将观察大小提高到64000个，将ResNet-50模型在Imagenet数据集上的训练时间减少到不到7分钟。作为比较——最初的训练时间是按天为单位的。

- 并行计算的能力也对强化学习的进步做出了相当关键的贡献。这导致了计算机在围棋、雅达里游戏、星际争霸和物理模拟（例如，使用MuJoCo）中实现超人性能的重大进步。有关如何在AlphaGo中实现这一点的说明，请参见如 [Silver et al., 2016]。简而言之，如果有大量的（状态、动作、奖励）三元组可用，即只要有可能尝试很多东西来了解它们之间的关系，强化学习就会发挥最好的作用。仿真提供了这样一条途径。
- 深度学习框架在传播思想方面发挥了至关重要的作用。允许轻松建模的第一代框架包括Caffe[16]、Torch[17]和Theano[18]。许多开创性的论文都是用这些工具写的。到目前为止，它们已经被TensorFlow[19]（通常通过其高级API Keras[20]使用）、CNTK[21]、Caffe 2[22]和Apache MXNet[23]所取代。第三代工具，即用于深度学习的命令式工具，可以说是由Chainer[24]率先推出的，它使用类似于Python NumPy的语法来描述模型。这个想法被PyTorch[25]、MXNet的Gluon API[26]和Jax[27]都采纳了。

"系统研究人员构建更好的工具"和"统计建模人员构建更好的神经网络"之间的分工大大简化了事情。例如，在2014年，对于卡内基梅隆大学机器学习博士生来说，训练线性回归模型曾经是一个不容易的作业问题。而现在，这项任务只需不到10行代码就能完成，这让每个程序员轻易掌握了它。

成功案例

人工智能已经有很长的历史了，它能带来用其他方法很难实现的结果。例如，使用光学字符识别的邮件分拣系统从20世纪90年代开始部署，毕竟，这是著名的手写数字MNIST数据集的来源。这同样适用于阅读银行存款支票和对申请者的信用进行评分。系统会自动检查金融交易是否存在欺诈。这构成了许多电子商务支付系统的支柱，如PayPal、Stripe、支付宝、微信、苹果、Visa和万事达卡。国际象棋的计算机程序已经竞争了几十年。机器学习在互联网上提供搜索、推荐、个性化和排名。换句话说，机器学习是无处不在的，尽管它经常隐藏在视线之外。

直到最近，人工智能才成为人们关注的焦点，主要是因为解决了以前被认为难以解决的问题，这些问题与消费者直接相关。许多这样的进步都归功于深度学习。

[16] https://github.com/BVLC/caffe
[17] https://github.com/torch
[18] https://github.com/Theano/Theano
[19] https://github.com/tensorflow/tensorflow
[20] https://github.com/keras-team/keras
[21] https://github.com/Microsoft/CNTK
[22] https://github.com/caffe2/caffe2
[23] https://github.com/apache/incubator-mxnet
[24] https://github.com/chainer/chainer
[25] https://github.com/pytorch/pytorch
[26] https://github.com/apache/incubator-mxnet
[27] https://github.com/google/jax

- 智能助理，如苹果的Siri、亚马逊的Alexa和谷歌助手，都能够相当准确地回答口头问题。这包括一些琐碎的工作，比如打开电灯开关（对残疾人来说是个福音）到预约理发师和提供电话支持对话。这可能是人工智能正在影响我们生活的最明显的迹象。
- 数字助理的一个关键因素是准确识别语音的能力。逐渐地，这样的系统的精确度已经增加到在某些应用中达到人类平等的程度 [Xiong et al., 2018]。
- 物体识别同样也取得了长足的进步。估计图片中的物体在2010年是一项相当具有挑战性的任务。在ImageNet基准上，来自NEC实验室和伊利诺伊大学香槟分校的研究人员获得了28%的Top-5错误率 [Lin et al., 2010] 。到2017年，这一错误率降低到2.25% [Hu et al., 2018] 。同样，在鉴别鸟类或诊断皮肤癌方面也取得了惊人的成果。
- 游戏曾经是人类智慧的堡垒。从TD-Gammon开始，一个使用时差强化学习的五子棋游戏程序，算法和计算的进展导致了广泛应用的算法。与五子棋不同的是，国际象棋有一个复杂得多的状态空间和一组动作。深蓝公司利用大规模并行性、专用硬件和高效搜索游戏树 [Campbell et al., 2002] 击败了加里·卡斯帕罗夫(Garry Kasparov)。围棋由于其巨大的状态空间，难度更大。AlphaGo在2015年达到了人类平等，使用深度学习和蒙特卡洛树抽样 [Silver et al., 2016] 相结合。扑克中的挑战是状态空间很大，而且没有完全观察到（我们不知道对手的牌）。在扑克游戏中，库图斯使用有效的结构化策略超过了人类的表现 [Brown & Sandholm, 2017] 。这说明了游戏中令人印象深刻的进步，以及先进的算法在其中发挥了关键作用的事实。
- 人工智能进步的另一个迹象是自动驾驶汽车和卡车的出现。虽然完全自主还没有完全触手可及，但在这个方向上已经取得了很好的进展，特斯拉(Tesla)、NVIDIA和Waymo等公司的产品至少实现了部分自主。让完全自主如此具有挑战性的是，正确的驾驶需要感知、推理和将规则纳入系统的能力。目前，深度学习主要应用于这些问题的计算机视觉方面。其余部分则由工程师进行大量调整。

同样，上面的列表仅仅触及了机器学习对实际应用的影响之处的皮毛。例如，机器人学、物流、计算生物学、粒子物理学和天文学最近取得的一些突破性进展至少部分归功于机器学习。因此，机器学习正在成为工程师和科学家必备的工具。

关于人工智能的非技术性文章中，经常提到人工智能奇点的问题：机器学习系统会变得有知觉，并独立于主人来决定那些直接影响人类生计的事情。在某种程度上，人工智能已经直接影响到人类的生计：信誉度的自动评估，车辆的自动驾驶，保释决定的自动准予等等。甚至，我们可以让Alexa打开咖啡机。

幸运的是，现在的人工智能系统，还远远没有知觉，以至于操纵它的人类创造者。首先，人工智能系统是以一种特定的、面向目标的方式设计、训练和部署的。虽然他们的行为可能会给人一种通用智能的错觉，但设计的基础是规则、启发式和统计模型的结合。其次，目前还不存在能够自我改进、自我推理、能够在试图解决一般任务的同时，修改、扩展和改进自己的架构的“人工通用智能”工具。

一个更紧迫的问题是人工智能在我们日常生活中的应用：卡车司机和店员完成的许多琐碎的工作很可能也将是自动化的。农业机器人可能会降低有机农业的成本，但它们也将使收获操作自动化。工业革命的这一阶段可能对社会的大部分地区产生深远的影响，因为卡车司机和店员是许多国家最常见的工作之一。此外，统计模型在不加注意地应用时，可能会导致种族、性别或年龄偏见，如果自动驱动相应的决策，则会引起对程序公平性的合理关注。重要的是要确保小心使用这些算法。就我们今天所知，这比恶意的超级智能毁灭人类的潜力更为紧迫。

特点

到目前为止，我们已经广泛地讨论了机器学习，它既是人工智能的一个分支，也是人工智能的一种方法。虽然深度学习是机器学习的一个子集，但令人眼花缭乱的算法和应用程序集让人很难评估深度学习的具体成分是什么。这就像试图确定披萨所需的配料一样困难，因为几乎每种成分都是可以替代的。

如前所述，机器学习可以使用数据来学习输入和输出之间的转换，例如在语音识别中将音频转换为文本。在这样做时，通常需要以适合算法的方式表示数据，以便将这种表示转换为输出。深度学习是"深度"的，模型学习许多转换的"层"，每一层提供一个层次的表示。例如，靠近输入的层可以表示数据的低级细节，而接近分类输出的层可以表示用于区分的更抽象的概念。由于表示学习的目的是寻找表示本身，因此深度学习可以称为"多级表示学习"。

到目前为止，我们讨论的问题，例如从原始音频信号中学习，图像的原始像素值，或者任意长度的句子与外语中的对应句子之间的映射，都是深度学习优于传统机器学习方法的问题。事实证明，这些多层模型能够以以前的工具所不能的方式处理低级的感知数据。毋庸置疑，深度学习方法中最显著的共同点是使用端到端训练。也就是说，与其基于单独调整的组件组装系统，不如构建系统，然后联合调整它们的性能。例如，在计算机视觉中，科学家们习惯于将特征工程的过程与建立机器学习模型的过程分开。Canny边缘检测器 [Canny, 1987] 和SIFT特征提取器 [Lowe, 2004] 作为将图像映射到特征向量的算法，在过去的十年里占据了至高无上的地位。在过去的日子里，将机器学习应用于这些问题的关键部分是提出人工设计的特征工程方法，将数据转换为某种适合于浅层模型的形式。然而，与一个算法自动执行的数百万个选择相比，人类通过特征工程所能完成的事情很少。当深度学习开始时，这些特征抽取器被自动调整的滤波器所取代，产生了更高的精确度。

因此，深度学习的一个关键优势是它不仅取代了传统学习管道末端的浅层模型，而且还取代了劳动密集型的特征工程过程。此外，通过取代大部分特定领域的预处理，深度学习消除了以前分隔计算机视觉、语音识别、自然语言处理、医学信息学和其他应用领域的许多界限，为解决各种问题提供了一套统一的工具。

除了端到端的训练，我们正在经历从参数统计描述到完全非参数模型的转变。当数据稀缺时，人们需要依靠简化对现实的假设来获得有用的模型。当数据丰富时，可以用更准确地拟合实际情况的非参数模型来代替。在某种程度上，这反映了物理学在上个世纪中叶随着计算机的出现所经历的进步。现在人们可以求助于相关偏微分方程的数值模拟，而不是用手来求解电子行为的参数近似。这导致了更精确的模型，尽管常常以牺牲可解释性为代价。

与以前工作的另一个不同之处是接受次优解，处理非凸非线性优化问题，并且愿意在证明之前尝试。这种在处理统计问题上新发现的经验主义，加上人才的迅速涌入，导致了实用算法的快速进步。尽管在许多情况下，这是以修改和重新发明存在了数十年的工具为代价的。

最后，深度学习社区引以为豪的是，他们跨越学术界和企业界共享工具，发布了许多优秀的算法库、统计模型和经过训练的开源神经网络。正是本着这种精神，本书免费分发和使用。我们努力降低每个人了解深度学习的门槛，我们希望我们的读者能从中受益。

小结

- 机器学习研究计算机系统如何利用经验（通常是数据）来提高特定任务的性能。它结合了统计学、数据挖掘和优化的思想。通常，它被用作实现人工智能解决方案的一种手段。
- 表示学习作为机器学习的一类，其研究的重点是如何自动找到合适的数据表示方式。深度学习是通过学习多层次的转换来进行的多层次的表示学习。
- 深度学习不仅取代了传统机器学习的浅层模型，而且取代了劳动密集型的特征工程。
- 最近在深度学习方面取得的许多进展，大都是由廉价传感器和互联网规模应用所产生的大量数据，以及（通过GPU）算力的突破来触发的。
- 整个系统优化是获得高性能的关键环节。有效的深度学习框架的开源使得这一点的设计和实现变得非常容易。

练习

1. 你当前正在编写的代码的哪些部分可以"学习"，即通过学习和自动确定代码中所做的设计选择来改进？你的代码是否包含启发式设计选择？
2. 你遇到的哪些问题有许多解决它们的样本，但没有具体的自动化方法？这些可能是使用深度学习的主要候选者。
3. 如果把人工智能的发展看作一场新的工业革命，那么算法和数据之间的关系是什么？它类似于蒸汽机和煤吗？根本区别是什么？
4. 你还可以在哪里应用端到端的训练方法，比如 图2 、物理、工程和计量经济学？

Discussions[28]

[28] https://discuss.d2l.ai/t/2088

1

预备知识

要开始深度学习课程的学习，我们需要掌握一些基本技能。所有的机器学习方法都涉及从数据中提取信息。因此，我们首先将学习一些实用技能，包括存储、操作和预处理数据。

机器学习通常需要处理大型数据集。我们可以将数据集视为表，其中表的行对应于样本，列对应于属性。线性代数为我们提供了一些用来处理表格数据的技术。我们不会太深入细节，而是将重点放在矩阵运算的基本原理及其实现上。

深度学习是关于优化的。我们有一个带有参数的模型，我们想要找到那些能拟合数据的最好模型。在算法的每个步骤中，决定以何种方式调整参数需要一点微积分知识。在本节中将简要介绍这些知识。幸运的是，`autograd`包会自动为我们计算微分，在本节中我们也将介绍它。

接下来，机器学习涉及如何做出预测：给定我们观察到的信息，某些未知属性的可能值是多少？要在不确定的情况下进行严格的推理，我们需要引用概率语言。

最后，官方文档提供了本书之外的大量描述和示例。在本章的结尾，我们将向你展示如何在文档中查找所需信息。

这本书将对数学的要求保持在正确理解深度学习所需的最低限度。然而，这并不意味着这本书是没有数学的。因此，本章提供了基本且常用的数学知识的快速介绍，使任何人能够至少理解书中的大部分数学内容。如果你希望理解全部的数学内容，进一步学习数学的在线附录[29]就足够了。

[29] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/index.html

1.1 数据操作

为了能够完成各种操作，我们需要某种方法来存储和操作数据。一般来说，我们需要做两件重要的事情：（1）获取数据；（2）在数据读入计算机后对其进行处理。如果没有某种方法来存储数据，那么获取数据是没有意义的。我们先尝试一个合成数据。首先，我们介绍n维数组，也称为 张量（tensor）。

如果你使用过 Python 中最广泛使用的科学计算包 NumPy，那么你会感觉怼本部分很熟悉。无论你使用哪个框架，它的 张量类（在 MXNet 中为 `ndarray`，在 PyTorch 和TensorFlow中为 `Tensor`）与 Numpy 的 `ndarray` 类似，但都比Numpy 的 `ndarray`多一些重要功能。首先，GPU 很好地支持加速计算，而 NumPy 仅支持 CPU 计算。其次，张量类支持自动微分。这些功能使得张量类更适合深度学习。除非另有说明，在整本书中所说的张量指的是张量类的实例。

1.1.1 入门

在本节中，我们的目标是帮助你开始了解并运行一些基本数值计算工具。在你阅读本书的过程中，将用到这些工具。如果你很难理解一些数学概念或库函数，请不要担心。在后面的章节将通过一些实际的例子来回顾这些内容。如果你已经有了一些背景知识，想要深入学习数学内容，可以就跳过这一节。

首先，我们从MXNet导入 `np`（`numpy`）模块和 `npx`（`numpy_extension`）模块。`np` 模块包含了 NumPy 支持的函数。而 `npx` 模块包含了一组扩展函数，用来在类似 NumPy 的环境中实现深度学习开发。当使用张量时，我们几乎总是会调用 `set_np` 函数：这是为了兼容 MXNet 其他的张量处理组件。

```
from mxnet import np, npx

npx.set_np()
```

张量表示一个数值组成的数组，这个数组可能有多个维度。具有一个轴的张量对应于数学上的 向量（vector）。具有两个轴的张量对应于数学上的 矩阵（matrix）。具有两个轴以上的张量没有特殊的数学名称。

首先，我们可以使用 `arange` 创建一个行向量 `x`。这个行向量包含以0开始的前12个整数，它们默认创建为浮点数。张量中的每个值都称为张量的 元素（element）。例如，张量 `x` 中有 12 个元素。除非额外指定，新的张量将存储在内存中，并采用基于CPU的计算。

```
x = np.arange(12)
x
```

```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

我们可以通过张量的 `shape` 属性来访问张量的 形状（沿每个轴的长度）。

```
x.shape
```

```
(12,)
```

如果我们只想知道张量中元素的总数，即形状的所有元素乘积，我们可以检查它的大小（size）。因为这里在处理的是一个向量，所以它的 `shape` 与它的 `size` 相同。

```
x.size
```

```
12
```

要改变一个张量的形状而不改变元素数量和元素值，我们可以调用 `reshape` 函数。例如，我们可以把张量 `x` 从形状为 (12,) 的行向量转换为形状 (3, 4) 的矩阵。这个新的张量包含与转换前相同的值，但是把它们看成一个三行四列的矩阵。要重点说明一下，虽然形状发生了改变，但元素值没有变。注意，通过改变张量的形状，张量的大小不会改变。

```
X = x.reshape(3, 4)
X
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

不需要通过手动指定每个维度来改变形状。如果我们的目标形状是 (高度, 宽度)，那么在知道宽度后，高度应当会隐式得出，我们不必自己做除法。在上面的例子中，要获得一个有3行的矩阵，我们手动指定了它有3行和4列。幸运的是，张量在给出其他部分后可以自动计算出一个维度。我们可以通过将希望张量自动推断的维度放置 `-1` 来调用此功能。在上面的例子中，我们可以用 `x.reshape(-1, 4)` 或 `x.reshape(3, -1)`来取代`x.reshape(3, 4)`。

有时，我们希望使用全0、全1、其他常量或者从特定分布中随机采样的数字，来初始化矩阵。我们可以创建一个形状为 (2, 3, 4) 的张量，其中所有元素都设置为0。代码如下：

```
np.zeros((2, 3, 4))
```

```
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]])
```

同样的，我们可以创建一个张量，其中所有元素都设置为1。代码如下：

```
np.ones((2, 3, 4))
```

```
array([[[1., 1., 1., 1.],
        [1., 1., 1., 1.],
```

(continues on next page)

(continued from previous page)

```
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]])
```

有时我们想从某个概率分布中随机采样来得到张量中每个元素的值。例如，当我们构造数组来作为神经网络中的参数时，我们通常会随机初始化参数的值。以下代码创建一个形状为 (3, 4) 的张量。其中的每个元素都从均值为0、标准差为1的标准高斯（正态）分布中随机采样。

```
np.random.normal(0, 1, size=(3, 4))
```

```
array([[ 2.2122064 ,  1.1630787 ,  0.7740038 ,  0.4838046 ],
       [ 1.0434405 ,  0.29956347,  1.1839255 ,  0.15302546],
       [ 1.8917114 , -1.1688148 , -1.2347414 ,  1.5580711 ]])
```

我们还可以通过提供包含数值的 Python 列表（或嵌套列表）来为所需张量中的每个元素赋予确定值。在这里，最外层的列表对应于轴 0，内层的列表对应于轴 1。

```
np.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
array([[2., 1., 4., 3.],
       [1., 2., 3., 4.],
       [4., 3., 2., 1.]])
```

1.1.2 运算

这本书不是关于软件工程的。我们的兴趣不仅仅限于从数组读取和写入数据。我们想在这些数组上执行数学运算。一些最简单且最有用的操作是 按元素（elementwise）操作。它们将标准标量运算符应用于数组的每个元素。对于将两个数组作为输入的函数，按元素运算将二元运算符应用于两个数组中的每对位置对应的元素。我们可以基于任何从标量到标量的函数来创建按元素函数。

在数学表示法中，我们将通过符号 $f: \mathbb{R} \rightarrow \mathbb{R}$ 来表示 一元标量运算符（只接收一个输入）。这意味着该函数从任何实数（$\mathbb{R}$）映射到另一个实数。同样，我们通过符号 $f: \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$ 表示 二元标量运算符，这意味着该函数接收两个输入，并产生一个输出。给定同一形状的任意两个向量$\mathbf{u}$和$\mathbf{v}$ 和二元运算符 f，我们可以得到向量$\mathbf{c} = F(\mathbf{u},\mathbf{v})$。具体计算方法是$c_i \gets f(u_i, v_i)$ ，其中 c_i、u_i$ 和 v_i 分别是向量$\mathbf{c}$、$\mathbf{u}$ 和 $\mathbf{v}$中的元素。在这里，我们通过将标量函数升级为按元素向量运算来生成向量值 $F: \mathbb{R}^d, \mathbb{R}^d \rightarrow \mathbb{R}^d$。

对于任意具有相同形状的张量，常见的标准算术运算符（+、-、*、/ 和 **）都可以被升级为按元素运算。我们可以在同一形状的任意两个张量上调用按元素操作。在下面的例子中，我们使用逗号来表示一个具有5个元素的元组，其中每个元素都是按元素操作的结果。

```
x = np.array([1, 2, 4, 8])
y = np.array([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x**y  # **运算符是求幂运算
```

```
(array([ 3.,  4.,  6., 10.]),
 array([-1.,  0.,  2.,  6.]),
 array([ 2.,  4.,  8., 16.]),
 array([0.5, 1. , 2. , 4. ]),
 array([ 1.,  4., 16., 64.]))
```

可以按按元素方式应用更多的计算，包括像求幂这样的一元运算符。

```
np.exp(x)
```

```
array([2.7182817e+00, 7.3890562e+00, 5.4598148e+01, 2.9809580e+03])
```

除了按元素计算外，我们还可以执行线性代数运算，包括向量点积和矩阵乘法。我们将在 1.3节 中解释线性代数的重点内容（不需要先修知识）。

我们也可以把多个张量连结在一起，把它们端对端地叠起来形成一个更大的张量。我们也可以 连结（concatenate）多个张量在一起，将它们端到端堆叠以形成更大的张量。我们只需要提供张量列表，并给出沿哪个轴连结。下面的例子分别演示了当我们沿行（轴-0，形状的第一个元素）和按列（轴-1，形状的第二个元素）连结两个矩阵时会发生什么情况。我们可以看到，第一个输出张量的轴-0长度 (6) 是两个输入张量轴-0长度的总和 $(3 + 3)$；第二个输出张量的轴-1长度 (8) 是两个输入张量轴-1长度的总和 $(4 + 4)$。

```
X = np.arange(12).reshape(3, 4)
Y = np.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
np.concatenate([X, Y], axis=0), np.concatenate([X, Y], axis=1)
```

```
(array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [ 2.,  1.,  4.,  3.],
        [ 1.,  2.,  3.,  4.],
        [ 4.,  3.,  2.,  1.]]),
 array([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
        [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
        [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

有时，我们想通过 逻辑运算符构建二元张量。以 `X == Y` 为例子。对于每个位置，如果 `X` 和 `Y` 在该位置相等，则新张量中相应项的值为1，这意味着逻辑语句 `X == Y` 在该位置处为真，否则该位置为 0。

```
X == Y
```

```
array([[False,  True, False,  True],
       [False, False, False, False],
       [False, False, False, False]])
```

对张量中的所有元素进行求和会产生一个只有一个元素的张量。

```
X.sum()
```

```
array(66.)
```

1.1.3 广播机制

在上面的部分中，我们看到了如何在相同形状的两个张量上执行按元素操作。在某些情况下，即使形状不同，我们仍然可以通过调用 广播机制（broadcasting mechanism）来执行按元素操作。这种机制的工作方式如下：首先，通过适当复制元素来扩展一个或两个数组，以便在转换之后，两个张量具有相同的形状。其次，对生成的数组执行按元素操作。

在大多数情况下，我们将沿着数组中长度为1的轴进行广播，如下例子：

```
a = np.arange(3).reshape(3, 1)
b = np.arange(2).reshape(1, 2)
a, b
```

```
(array([[0.],
        [1.],
        [2.]]),
 array([[0., 1.]]))
```

由于 a 和 b 分别是 3×1 和 1×2 矩阵，如果我们让它们相加，它们的形状不匹配。我们将两个矩阵广播为一个更大的 3×2 矩阵，如下所示：矩阵 a将复制列，矩阵 b将复制行，然后再按元素相加。

```
a + b
```

```
array([[0., 1.],
       [1., 2.],
       [2., 3.]])
```

1.1.4 索引和切片

就像在任何其他 Python 数组中一样，张量中的元素可以通过索引访问。与任何 Python 数组一样：第一个元素的索引是 0；可以指定范围以包含第一个元素和最后一个之前的元素。与标准 Python 列表一样，我们可以通过使用负索引根据元素到列表尾部的相对位置访问元素。

因此，我们可以用 `[-1]` 选择最后一个元素，可以用 `[1:3]` 选择第二个和第三个元素，如下所示：

```
X[-1], X[1:3]
```

```
(array([ 8.,  9., 10., 11.]),
 array([[ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]]))
```

除读取外，我们还可以通过指定索引来将元素写入矩阵。

```
X[1, 2] = 9
X
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  9.,  7.],
       [ 8.,  9., 10., 11.]])
```

如果我们想为多个元素赋值相同的值，我们只需要索引所有元素，然后为它们赋值。例如，`[0:2, :]` 访问第1行和第2行，其中“:”代表沿轴 1（列）的所有元素。虽然我们讨论的是矩阵的索引，但这也适用于向量和超过2个维度的张量。

```
X[0:2, :] = 12
X
```

```
array([[12., 12., 12., 12.],
       [12., 12., 12., 12.],
       [ 8.,  9., 10., 11.]])
```

1.1.5 节省内存

运行一些操作可能会导致为新结果分配内存。例如，如果我们用 `Y = X + Y`，我们将取消引用 `Y` 指向的张量，而是指向新分配的内存处的张量。

在下面的例子中，我们用 Python 的 `id()` 函数演示了这一点，它给我们提供了内存中引用对象的确切地址。运行 `Y = Y + X` 后，我们会发现 `id(Y)` 指向另一个位置。这是因为 Python 首先计算 `Y + X`，为结果分配新的内存，然后使 `Y` 指向内存中的这个新位置。

```
before = id(Y)
Y = Y + X
id(Y) == before
```

```
False
```

这可能是不可取的，原因有两个：首先，我们不想总是不必要地分配内存。在机器学习中，我们可能有数百兆的参数，并且在一秒内多次更新所有参数。通常情况下，我们希望原地执行这些更新。其次，我们可能通过多个变量指向相同参数。如果我们不原地更新，其他引用仍然会指向旧的内存位置，这样我们的某些代码可能会无意中引用旧的参数。

幸运的是，执行原地操作非常简单。我们可以使用切片表示法将操作的结果分配给先前分配的数组，例如 `Y[:] = <expression>`。为了说明这一点，我们首先创建一个新的矩阵 `Z`，其形状与另一个 `Y` 相同，使用 `zeros_like` 来分配一个全0的块。

```
Z = np.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 140587942287104
id(Z): 140587942287104
```

如果在后续计算中没有重复使用 `X`，我们也可以使用 `X[:] = X + Y` 或 `X += Y` 来减少操作的内存开销。

```
before = id(X)
X += Y
id(X) == before
```

```
True
```

1.1.6 转换为其他 Python 对象

转换为 NumPy 张量很容易，反之也很容易。转换后的结果不共享内存。这个小的不便实际上是非常重要的：当你在 CPU 或 GPU 上执行操作的时候，此时Python的NumPy包也希望使用相同的内存块执行其他操作时，你不希望停止计算。

```
A = X.asnumpy()
B = np.array(A)
type(A), type(B)
```

```
(numpy.ndarray, mxnet.numpy.ndarray)
```

要将大小为1的张量转换为 Python 标量，我们可以调用 item 函数或 Python 的内置函数。

```
a = np.array([3.5])
a, a.item(), float(a), int(a)
```

```
(array([3.5]), 3.5, 3.5, 3)
```

1.1.7 小结

- 深度学习存储和操作数据的主要接口是张量（n维数组）。它提供了各种功能，包括基本数学运算、广播、索引、切片、内存节省和转换其他 Python 对象。

1.1.8 练习

1. 运行本节中的代码。将本节中的条件语句 `X == Y` 更改为 `X < Y` 或 `X > Y`，然后看看你可以得到什么样的张量。
2. 用其他形状（例如三维张量）替换广播机制中按元素操作的两个张量。结果是否与预期相同？

Discussions[30]

1.2 数据预处理

到目前为止，我们已经介绍了处理存储在张量中数据的各种技术。为了能用深度学习来解决现实世界的问题，我们经常从预处理原始数据开始，而不是从那些准备好的张量格式数据开始。在Python中常用的数据分析工具中，通常使用 pandas 软件包。像庞大的 Python 生态系统中的许多其他扩展包一样，pandas 可以与张量兼容。因此，我们将简要介绍使用 pandas 预处理原始数据并将原始数据转换为张量格式的步骤。我们将在后面的章节中介绍更多的数据预处理技术。

1.2.1 读取数据集

举一个例子，我们首先创建一个人工数据集，并存储在csv（逗号分隔值）文件 ../data/house_tiny.csv 中。以其他格式存储的数据也可以通过类似的方式进行处理。下面的mkdir_if_not_exist 函数可确保目录 ../data 存在。注意，注释 #@save是一个特殊的标记，该标记下方的函数、类或语句将保存在 d2l 软件包中，以便以后可以直接调用它们（例如 d2l.mkdir_if_not_exist(path)）而无需重新定义。

下面我们将数据集按行写入 csv 文件中。

[30] https://discuss.d2l.ai/t/1745

```
import os

os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('NumRooms,Alley,Price\n')  # 列名
    f.write('NA,Pave,127500\n')  # 每行表示一个数据样本
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')
```

要从创建的csv文件中加载原始数据集，我们导入pandas包并调用read_csv函数。该数据集有四行三列。其中每行描述了房间数量（“NumRooms”）、巷子类型（“Alley”）和房屋价格（“Price”）。

```
# 如果没有安装pandas，只需取消对以下行的注释：
# !pip install pandas
import pandas as pd

data = pd.read_csv(data_file)
print(data)
```

```
   NumRooms Alley   Price
0       NaN  Pave  127500
1       2.0   NaN  106000
2       4.0   NaN  178100
3       NaN   NaN  140000
```

1.2.2 处理缺失值

注意，“NaN”项代表缺失值。为了处理缺失的数据，典型的方法包括 插值和 删除，其中插值用替代值代替缺失值。而删除则忽略缺失值。在这里，我们将考虑插值。

通过位置索引iloc，我们将data分成inputs和outputs，其中前者为data的前两列，而后者为data的最后一列。对于inputs中缺少的的数值，我们用同一列的均值替换“NaN”项。

```
inputs, outputs = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

```
   NumRooms Alley
0       3.0  Pave
1       2.0   NaN
```

(continues on next page)

(continued from previous page)

```
2       4.0   NaN
3       3.0   NaN
```

对于 inputs 中的类别值或离散值，我们将“NaN”视为一个类别。由于“巷子”(“Alley”)列只接受两种类型的类别值“Alley”和“NaN”，pandas 可以自动将此列转换为两列“Alley_Pave”和“Alley_nan”。巷子类型为“Pave”的行会将“Alley_Pave”的值设置为1，“Alley_nan”的值设置为0。缺少巷子类型的行会将“Alley_Pave”和“Alley_nan”分别设置为0和1。

```
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

```
   NumRooms  Alley_Pave  Alley_nan
0       3.0           1          0
1       2.0           0          1
2       4.0           0          1
3       3.0           0          1
```

1.2.3 转换为张量格式

现在 inputs 和 outputs 中的所有条目都是数值类型，它们可以转换为张量格式。当数据采用张量格式后，可以通过在 1.1节 中引入的那些张量函数来进一步操作。

```
from mxnet import np

X, y = np.array(inputs.values), np.array(outputs.values)
X, y
```

```
(array([[3., 1., 0.],
        [2., 0., 1.],
        [4., 0., 1.],
        [3., 0., 1.]], dtype=float64),
 array([127500, 106000, 178100, 140000], dtype=int64))
```

1.2.4 小结

- 像庞大的 Python 生态系统中的许多其他扩展包一样，pandas 可以与张量兼容。
- 插值和删除可用于处理缺失的数据。

1.2.5 练习

创建包含更多行和列的原始数据集。

1. 删除缺失值最多的列。
2. 将预处理后的数据集转换为张量格式。

Discussions[31]

1.3 线性代数

在你已经可以存储和操作数据后，让我们简要地回顾一下基本线性代数的部分内容。这些内容能够帮助你了解和实现本书中介绍的大多数模型。下面我们将介绍线性代数中的基本数学对象、算术和运算，并用数学符号和相应的代码实现来表示它们。

1.3.1 标量

如果你从来没有学过线性代数或机器学习，那么你过去的数学经历可能是一次只想一个数字。如果你曾经报销过发票，或者在餐厅支付餐费，那么你已经知道如何做一些基本的事情，比如在数字间相加或相乘。例如，北京的温度为 52 华氏度（除了摄氏度外，另一种温度刻度）。严格来说，我们称仅包含一个数值的叫 标量（scalar）。如果要将此华氏度值转换为更常用的摄氏度，则可以计算表达式 $c = \frac{5}{9}(f - 32)$，并将 f 赋为 52。在此等式中，每一项（5、9 和 32）都是标量值。符号 c 和 f 称为 变量（variables），它们表示未知的标量值。

在本书中，我们采用了数学表示法，其中标量变量由普通小写字母表示（例如，x、y 和 z）。我们用 $\mathbb{R}$ 表示所有（连续）实数标量的空间。为了方便，我们之后将严格定义 空间（space）是什么，但现在只要记住，表达式 $x \in \mathbb{R}$ 是表示x是一个实值标量的正式形式。符号 $\in$ 称为“属于”，它表示“是集合中的成员”。我们可以用 $x, y \in \{0, 1\}$ 来表明 x 和 y 是值只能为 0 或 1的数字。

标量由只有一个元素的张量表示。在下面的代码中，我们实例化两个标量，并使用它们执行一些熟悉的算术运算，即加法，乘法，除法和指数。

```
from mxnet import np, npx

npx.set_np()

x = np.array(3.0)
y = np.array(2.0)

x + y, x * y, x / y, x**y
```

```
(array(5.), array(6.), array(1.5), array(9.))
```

[31] https://discuss.d2l.ai/t/1749

1.3.2 向量

你可以将向量视为标量值组成的列表。我们将这些标量值称为向量的 元素（elements）或分量（components）。当我们的向量表示数据集中的样本时，它们的值具有一定的现实意义。例如，如果我们正在训练一个模型来预测贷款违约风险，我们可能会将每个申请人与一个向量相关联，其分量与其收入、工作年限、过往违约次数和其他因素相对应。如果我们正在研究医院患者可能面临的心脏病发作风险，我们可能会用一个向量来表示每个患者，其分量为最近的生命体征、胆固醇水平、每天运动时间等。在数学表示法中，我们通常将向量记为粗体、小写的符号（例如，$\mathbf{x}$、$\mathbf{y}$和$\mathbf{z}$)）。

我们通过一维张量处理向量。一般来说，张量可以具有任意长度，取决于机器的内存限制。

```
x = np.arange(4)
x
```

```
array([0., 1., 2., 3.])
```

我们可以使用下标来引用向量的任一元素。例如，我们可以通过 x_i 来引用第 i 个元素。注意，元素 x_i 是一个标量，所以我们在引用它时不会加粗。大量文献认为列向量是向量的默认方向，在本书中也是如此。在数学中，向量 $\mathbf{x}$ 可以写为：

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \tag{1.3.1}$$

其中 $x_1, \ldots, x_n$ 是向量的元素。在代码中，我们通过张量的索引来访问任一元素。

```
x[3]
```

```
array(3.)
```

长度、维度和形状

让我们回顾一下 1.1节 中的一些概念。向量只是一个数字数组。就像每个数组都有一个长度一样，每个向量也是如此。在数学表示法中，如果我们想说一个向量 $\mathbf{x}$ 由 n 个实值标量组成，我们可以将其表示为 $\mathbf{x} \in \mathbb{R}^n$。向量的长度通常称为向量的 维度（dimension）。

与普通的 Python 数组一样，我们可以通过调用 Python 的内置 `len()` 函数来访问张量的长度。

```
len(x)
```

```
4
```

当用张量表示一个向量（只有一个轴）时，我们也可以通过 `.shape` 属性访问向量的长度。形状（shape）是一个元组，列出了张量沿每个轴的长度（维数）。对于只有一个轴的张量，形状只有一个元素。

```
x.shape
```

```
(4,)
```

请注意，维度（dimension）这个词在不同上下文时往往会有不同的含义，这经常会使人感到困惑。为了清楚起见，我们在此明确一下。向量或轴的维度被用来表示向量或轴的长度，即向量或轴的元素数量。然而，张量的维度用来表示张量具有的轴数。在这个意义上，张量的某个轴的维数就是这个轴的长度。

1.3.3 矩阵

正如向量将标量从零阶推广到一阶，矩阵将向量从一阶推广到二阶。矩阵，我们通常用粗体、大写字母来表示（例如，$\mathbf{X}$、$\mathbf{Y}$ 和 $\mathbf{Z}$），在代码中表示为具有两个轴的张量。

在数学表示法中，我们使用 $\mathbf{A} \in \mathbb{R}^{m\times n}$ 来表示矩阵 $\mathbf{A}$，其由m 行和 n 列的实值标量组成。直观地，我们可以将任意矩阵 $\mathbf{A} \in \mathbb{R}^{m\times n}$ 视为一个表格，其中每个元素 a_{ij} 属于第 i 行第j 列：

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{1.3.2}$$

对于任意$\mathbf{A} \in \mathbb{R}^{m\times n}$,$\mathbf{A}$的形状是$(m, n)$或$m \times n$。当矩阵具有相同数量的行和列时，其形状将变为正方形；因此，它被称为 方矩阵（square matrix）。

当调用函数来实例化张量时，我们可以通过指定两个分量m 和 n来创建一个形状为$m \times n$ 的矩阵。

```
A = np.arange(20).reshape(5, 4)
A
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.],
       [16., 17., 18., 19.]])
```

我们可以通过行索引（i）和列索引（j）来访问矩阵中的标量元素 a_{ij}，例如 $[\mathbf{A}]_{ij}$。如果没有给出矩阵 $\mathbf{A}$ 的标量元素，如在 (1.3.2)那样，我们可以简单地使用矩阵 $\mathbf{A}$ 的小写字母索引下标 a_{ij}来引用$[\mathbf{A}]_{ij}$。为了表示起来简单，只有在必要时才会将逗号插入到单独的索引中，例如 $a_{2,3j}$ 和 $[\mathbf{A}]_{2i-1,3}$。

有时候，我们想翻转轴。当我们交换矩阵的行和列时，结果称为矩阵的 转置（transpose）。我们用$\mathbf{a}^\top$来表示矩阵的转置，如果$\mathbf{B} = \mathbf{A}^\top$，则对于任意$i$和$j$，都有$b_{ij} = a_{ji}$。因此，在 (1.3.2) 中的转置是一个形状为$n \times m$的

矩阵：

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}. \tag{1.3.3}$$

现在我们在代码中访问矩阵的转置。

```
A.T
```

```
array([[ 0.,  4.,  8., 12., 16.],
       [ 1.,  5.,  9., 13., 17.],
       [ 2.,  6., 10., 14., 18.],
       [ 3.,  7., 11., 15., 19.]])
```

作为方矩阵的一种特殊类型，对称矩阵（symmetric matrix）$\mathbf{A}$ 等于其转置：$\mathbf{A} = \mathbf{A}^\top$。这里我们定义一个对称矩阵 B：

```
B = np.array([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
B
```

```
array([[1., 2., 3.],
       [2., 0., 4.],
       [3., 4., 5.]])
```

现在我们将 B 与它的转置进行比较。

```
B == B.T
```

```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

矩阵是有用的数据结构：它们允许我们组织具有不同变化模式的数据。例如，我们矩阵中的行可能对应于不同的房屋（数据样本），而列可能对应于不同的属性。如果你曾经使用过电子表格软件或已阅读过 1.2节，这应该听起来很熟悉。因此，尽管单个向量的默认方向是列向量，但在表示表格数据集的矩阵中，将每个数据样本作为矩阵中的行向量更为常见。我们将在后面的章节中讲到这点。这种约定将支持常见的深度学习实践。例如，沿着张量的最外轴，我们可以访问或遍历小批量的数据样本。如果不存在小批量，我们也可以只访问数据样本。

1.3.4 张量

就像向量是标量的推广，矩阵是向量的推广一样，我们可以构建具有更多轴的数据结构。张量（本小节中的“张量”指代数对象）为我们提供了描述具有任意数量轴的n维数组的通用方法。例如，向量是一阶张量，矩阵是二阶张量。张量用特殊字体的大写字母（例如，X、Y 和 Z）表示，它们的索引机制（例如 x_{ijk} 和 $[\mathsf{X}]_{1,2i-1,3}$）与矩阵类似。

当我们开始处理图像时，张量将变得更加重要，图像以n维数组形式出现，其中3个轴对应于高度、宽度，以及一个通道（channel）轴，用于堆叠颜色通道（红色、绿色和蓝色）。现在，我们将跳过高阶张量，集中在基础知识上。

```
X = np.arange(24).reshape(2, 3, 4)
X
```

```
array([[[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]],

       [[12., 13., 14., 15.],
        [16., 17., 18., 19.],
        [20., 21., 22., 23.]]])
```

1.3.5 张量算法的基本性质

标量、向量、矩阵和任意数量轴的张量（本小节中的“张量”指代数对象）有一些很好的属性，通常会派上用场。例如，你可能已经从按元素操作的定义中注意到，任何按元素的一元运算都不会改变其操作数的形状。同样，给定具有相同形状的任何两个张量，任何按元素二元运算的结果都将是相同形状的张量。例如，将两个相同形状的矩阵相加会在这两个矩阵上执行元素加法。

```
A = np.arange(20).reshape(5, 4)
B = A.copy()  # 通过分配新内存，将A的一个副本分配给B
A, A + B
```

```
(array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]]),
 array([[ 0.,  2.,  4.,  6.],
        [ 8., 10., 12., 14.],
        [16., 18., 20., 22.],
        [24., 26., 28., 30.],
        [32., 34., 36., 38.]]))
```

具体而言，两个矩阵的按元素乘法称为 哈达玛积（Hadamard product）（数学符号 $\odot$）。对于矩阵 $\mathbf{B} \in \mathbb{R}^{m\times n}$，其中第 i 行和第 j 列的元素是 b_{ij}。矩阵$\mathbf{A}$（在 (1.3.2) 中定义）和 $\mathbf{B}$的哈达玛积为：

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}. \tag{1.3.4}$$

```
A * B
```

```
array([[  0.,   1.,   4.,   9.],
       [ 16.,  25.,  36.,  49.],
       [ 64.,  81., 100., 121.],
       [144., 169., 196., 225.],
       [256., 289., 324., 361.]])
```

将张量乘以或加上一个标量不会改变张量的形状，其中张量的每个元素都将与标量相加或相乘。

```
a = 2
X = np.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

```
(array([[[ 2.,  3.,  4.,  5.],
         [ 6.,  7.,  8.,  9.],
         [10., 11., 12., 13.]],

        [[14., 15., 16., 17.],
         [18., 19., 20., 21.],
         [22., 23., 24., 25.]]]),
 (2, 3, 4))
```

1.3.6 汇总

我们可以对任意张量进行的一个有用的操作是计算其元素的和。在数学表示法中，我们使用 $\sum$ 符号表示求和。为了表示长度为d的向量中元素的总和，可以记为 $\sum_{i=1}^{d} x_i$。在代码中，我们可以调用计算求和的函数：

```
x = np.arange(4)
x, x.sum()
```

```
(array([0., 1., 2., 3.]), array(6.))
```

我们可以表示任意形状张量的元素和。例如，矩阵 $\mathbf{A}$ 中元素的和可以记为$\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$。

```
A.shape, A.sum()
```

```
((5, 4), array(190.))
```

默认情况下，调用求和函数会将一个张量在所有轴上汇总为一个标量。我们还可以指定求和汇总张量的轴。以矩阵为例。为了通过求和所有行的元素来汇总行维度（轴0），我们可以在调用函数时指定`axis=0`。由于输入矩阵沿0轴汇总以生成输出向量，因此输入的轴0的维数在输出形状中丢失。

```
A_sum_axis0 = A.sum(axis=0)
A_sum_axis0, A_sum_axis0.shape
```

```
(array([40., 45., 50., 55.]), (4,))
```

指定 `axis=1` 将通过汇总所有列的元素来汇总列维度（轴 1）。因此，输入的轴1的维数在输出形状中丢失。

```
A_sum_axis1 = A.sum(axis=1)
A_sum_axis1, A_sum_axis1.shape
```

```
(array([ 6., 22., 38., 54., 70.]), (5,))
```

沿着行和列对矩阵求和，等价于对矩阵的所有元素进行求和。

```
A.sum(axis=[0, 1])  # Same as `A.sum()`
```

```
array(190.)
```

一个与求和相关的量是 平均值（mean或average）。我们通过将总和除以元素总数来计算平均值。在代码中，我们可以调用函数来计算任意形状张量的平均值。

```
A.mean(), A.sum() / A.size
```

```
(array(9.5), array(9.5))
```

同样，计算平均值的函数也可以沿指定轴汇总张量。

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
(array([ 8.,  9., 10., 11.]), array([ 8.,  9., 10., 11.]))
```

非汇总求和

但是，有时在调用函数来计算总和或均值时保持轴数不变会很有用。

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A
```

```
array([[ 6.],
       [22.],
       [38.],
       [54.],
       [70.]])
```

例如，由于 `sum_A` 在对每行进行求和后仍保持两个轴，我们可以通过广播将 `A` 除以 `sum_A`。

```
A / sum_A
```

```
array([[0.        , 0.16666667, 0.33333334, 0.5       ],
       [0.18181819, 0.22727273, 0.27272728, 0.3181818 ],
       [0.21052632, 0.23684211, 0.2631579 , 0.28947368],
       [0.22222222, 0.24074075, 0.25925925, 0.2777778 ],
       [0.22857143, 0.24285714, 0.25714287, 0.27142859]])
```

如果我们想沿某个轴计算 `A` 元素的累积总和，比如 `axis=0`（按行计算），我们可以调用 `cumsum` 函数。此函数不会沿任何轴汇总输入张量。

```
A.cumsum(axis=0)
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  6.,  8., 10.],
       [12., 15., 18., 21.],
       [24., 28., 32., 36.],
       [40., 45., 50., 55.]])
```

1.3.7 点积（Dot Product）

到目前为止，我们只执行了按元素操作、求和及平均值。如果这就是我们所能做的，那么线性代数可能就不需要单独一节了。但是，最基本的操作之一是点积。给定两个向量 $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$，它们的 点积（dot product）$\mathbf{x}^\top \mathbf{y}$（或 $\langle \mathbf{x}, \mathbf{y} \rangle$）是相同位置的按元素乘积的和：$\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$。

```
y = np.ones(4)
x, y, np.dot(x, y)
```

```
(array([0., 1., 2., 3.]), array([1., 1., 1., 1.]), array(6.))
```

注意，我们可以通过执行按元素乘法，然后进行求和来表示两个向量的点积：

```
np.sum(x * y)
```

```
array(6.)
```

点积在很多场合都很有用。例如，给定一组由向量$\mathbf{x} \in \mathbb{R}^d$ 表示的值，和一组由 $\mathbf{w} \in \mathbb{R}^d$ 表示的权重。$\mathbf{x}$ 中的值根据权重 $\mathbf{w}$ 的加权和可以表示为点积 $\mathbf{x}^\top \mathbf{w}$。当权重为非负数且和为1（即 $\left(\sum_{i=1}^{d} w_i = 1\right)$）时，点积表示加权平均（weighted average）。将两个向量归一化得到单位长度后，点积表示它们夹角的余弦。我们将在本节的后面正式介绍长度（length）的概念。

1.3.8 矩阵-向量积

现在我们知道如何计算点积，我们可以开始理解 矩阵-向量积（matrix-vector products）。回顾分别在 (1.3.2) 和 (1.3.1) 中定义和可视化的矩阵 $\mathbf{A} \in \mathbb{R}^{m \times n}$ 和向量 $\mathbf{x} \in \mathbb{R}^n$。让我们从可视化矩阵$\mathbf{A}$开始，用它的行向量表示

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_m \end{bmatrix}, \tag{1.3.5}$$

其中每个$\mathbf{a}^\top_{i} \in \mathbb{R}^n$ 都是行向量，表示矩阵的 i^th 行。矩阵向量积 $\mathbf{A}\mathbf{x}$ 是一个长度为 m 的列向量，其 i^th 元素是点积 $\mathbf{a}^\top_i \mathbf{x}$：

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_m \end{bmatrix}\mathbf{x} = \begin{bmatrix} \mathbf{a}^\top_{1} \mathbf{x} \\ \mathbf{a}^\top_{2} \mathbf{x} \\ \vdots\\ \mathbf{a}^\top_{m} \mathbf{x}\end{bmatrix}. \tag{1.3.6}$$

我们可以把一个矩阵 $\mathbf{A} \in \mathbb{R}^{m \times n}$ 乘法看作是一个从 $\mathbb{R}^{n}$ 到 $\mathbb{R}^{m}$ 向量的转换。这些转换证明是非常有用的。例如，我们可以用方阵的乘法来表示旋转。我们将在后续章节中讲到，我们也可以使用矩阵向量乘积来描述在给定前一层的值时计算神经网络的每一层所需要的计算。

在代码中使用张量表示矩阵向量积，我们使用与点积相同的 `dot` 函数。当我们为矩阵 `A` 和向量 `x` 调用 `np.dot(A, x)`时，会执行矩阵向量积。注意，`A` 的列维数（沿轴1的长度）必须与 `x` 的维数（其长度）相同。

```
A.shape, x.shape, np.dot(A, x)
```

```
((5, 4), (4,), array([ 14., 38., 62., 86., 110.]))
```

1.3.9 矩阵-矩阵乘法

如果你已经掌握了点积和矩阵-向量积的知识，那么 **矩阵-矩阵乘法**（matrix-matrix multiplication）应该很简单。

假设我们有两个矩阵 $\mathbf{A} \in \mathbb{R}^{n \times k}$ 和 $\mathbf{B} \in \mathbb{R}^{k \times m}$：

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix},\quad \mathbf{B}=\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}. \tag{1.3.7}$$

用行向量$\mathbf{a}^\top_{i} \in \mathbb{R}^k$ 表示矩阵$\mathbf{A}$的 i^th 行，并让列向量$\mathbf{b}_{j} \in \mathbb{R}^k$ 作为矩阵$\mathbf{B}$的 j^th 列。要生成矩阵积 $\mathbf{C} = \mathbf{A}\mathbf{B}$，最简单的方法是考虑$\mathbf{A}$的行向量和$\mathbf{B}$的列向量:

$$\mathbf{A}=\begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_n \end{bmatrix},\quad \mathbf{B}=\begin{bmatrix} \mathbf{b}_{1} & \mathbf{b}_{2} & \cdots & \mathbf{b}_{m} \end{bmatrix}. \tag{1.3.8}$$

当我们简单地将每个元素c_{ij}计算为点积$\mathbf{a}^\top_i \mathbf{b}_j$:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_n \end{bmatrix} \begin{bmatrix} \mathbf{b}_{1} & \mathbf{b}_{2} & \cdots & \mathbf{b}_{m} \end{bmatrix} = \begin{bmatrix} \mathbf{a}^\top_{1} \mathbf{b}_1 & \mathbf{a}^\top_{1}\mathbf{b}_2& \cdots & \mathbf{a}^\top_{1} \mathbf{b}_m \\ \mathbf{a}^\top_{2}\mathbf{b}_1 & \mathbf{a}^\top_{2} \mathbf{b}_2 & \cdots & \mathbf{a}^\top_{2} \mathbf{b}_m \\ \vdots & \vdots & \ddots &\vdots\\ \mathbf{a}^\top_{n} \mathbf{b}_1 & \mathbf{a}^\top_{n}\mathbf{b}_2& \cdots& \mathbf{a}^\top_{n} \mathbf{b}_m \end{bmatrix}. \tag{1.3.9}$$

我们可以将矩阵-矩阵乘法 $\mathbf{AB}$ 看作是简单地执行 m次矩阵-向量积，并将结果拼接在一起，形成一个 $n \times m$ 矩阵。在下面的代码中，我们在 A 和 B 上执行矩阵乘法。这里的A 是一个5行4列的矩阵，B是一个4行3列的矩阵。相乘后，我们得到了一个5行3列的矩阵。

```
B = np.ones(shape=(4, 3))
np.dot(A, B)
```

```
array([[ 6.,  6.,  6.],
       [22., 22., 22.],
       [38., 38., 38.],
       [54., 54., 54.],
       [70., 70., 70.]])
```

矩阵矩阵乘法可以简单地称为 **矩阵乘法**，不应与哈达玛积混淆。

1.3.10 范数

线性代数中最有用的一些运算符是 范数（norms）。非正式地说，一个向量的范数告诉我们一个向量有多大。这里考虑的 大小（size）概念不涉及维度，而是分量的大小。

在线性代数中，向量范数是将向量映射到标量的函数 f。向量范数要满足一些属性。给定任意向量 $\mathbf{x}$，第一个性质说，如果我们按常数因子 α 缩放向量的所有元素，其范数也会按相同常数因子的 绝对值缩放：

$$f(\alpha \mathbf{x}) = |\alpha| f(\mathbf{x}). \tag{1.3.10}$$

第二个性质是我们熟悉的三角不等式:

$$f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y}). \tag{1.3.11}$$

第三个性质简单地说范数必须是非负的:

$$f(\mathbf{x}) \geq 0. \tag{1.3.12}$$

这是有道理的，因为在大多数情况下，任何东西的最小的大小是0。最后一个性质要求最小范数，并且只有由所有零组成的向量才能达到最小范数。

$$\forall i, [\mathbf{x}]_i = 0 \Leftrightarrow f(\mathbf{x}) = 0. \tag{1.3.13}$$

你可能会注意到，范数听起来很像距离的度量。如果你还记得小学时的欧几里得距离(想想毕达哥拉斯定理)，那么非负性的概念和三角不等式可能会给你一些启发。事实上，欧几里得距离是一个范数：具体而言，它是 L_2 范数。假设n-维向量:math:*mathbf{x}*中的元素是$x_1, \ldots, x_n$ 的 L_2 范数是向量元素平方和的平方根:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}, \tag{1.3.14}$$

其中，在 L_2 范数中常常省略下标 2，也就是说，$\|\mathbf{x}\|$ 等同于 $\|\mathbf{x}\|_2$。在代码中，我们可以按如下方式计算向量的 L_2 范数。

```
u = np.array([3, -4])
np.linalg.norm(u)
```

```
array(5.)
```

在深度学习中，我们更经常地使用平方 L_2 范数。你还会经常遇到 L_1 范数，它表示为向量元素的绝对值之和:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n \left|x_i \right|. \tag{1.3.15}$$

与 L_2 范数相比，L_1 范数受异常值的影响较小。为了计算 L_1 范数，我们将绝对值函数和按元素求和组合起来。

```
np.abs(u).sum()
```

```
array(7.)
```

L_2 范数和 L_1 范数都是更一般的L_p范数的特例：

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^n |x_i|^p\right)^{1/p}. \tag{1.3.16}$$

类似于向量的L_2 范数，矩阵 $\mathbf{X} \in \mathbb{R}^{m \times n}$ 的 弗罗贝尼乌斯范数（Frobenius norm）是矩阵元素的平方和的平方根：

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n x_{ij}^2}. \tag{1.3.17}$$

弗罗贝尼乌斯范数满足向量范数的所有性质。它的行为就好像它是矩阵形向量的 L_2 范数。调用以下函数将计算矩阵的弗罗贝尼乌斯范数。

```
np.linalg.norm(np.ones((4, 9)))
```

```
array(6.)
```

范数和目标

虽然我们不想走得太远，但我们可以对这些概念为什么有用有一些直觉。在深度学习中，我们经常试图解决优化问题：最大化分配给观测数据的概率; 最小化预测和真实观测之间的距离。为物品(如单词、产品或新闻文章)分配向量表示，以便最小化相似项目之间的距离，最大化不同项目之间的距离。通常，目标，或许是深度学习算法最重要的组成部分(除了数据)，被表达为范数。

1.3.11 关于线性代数的更多信息

就在这一部分，我们已经教会了你们所有的线性代数，你们将需要这些线性代数来理解大量的现代深度学习。线性代数还有很多，其中很多数学对于机器学习非常有用。例如，矩阵可以分解为因子，这些分解可以显示真实世界数据集中的低维结构。机器学习的整个子领域都侧重于使用矩阵分解及其向高阶张量的泛化来发现数据集中的结构并解决预测问题。但这本书的重点是深度学习。我们相信，一旦你开始动手尝试在真实数据集上应用了有效的机器学习模型，你会更倾向于学习更多数学。因此，虽然我们保留在后面介绍更多数学知识的权利，但我们将在这里结束这一部分。

如果你渴望了解有关线性代数的更多信息，你可以参考 线性代数运算的在线附录[32] 或其他优秀资源 [Strang, 1993, Kolter, 2008, Petersen et al., 2008]。

[32] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/geometry-linear-algebraic-ops.html

1.3.12 小结

- 标量、向量、矩阵和张量是线性代数中的基本数学对象。
- 向量泛化自标量，矩阵泛化自向量。
- 标量、向量、矩阵和张量分别具有零、一、二和任意数量的轴。
- 一个张量可以通过`sum` 和 `mean`沿指定的轴汇总。
- 两个矩阵的按元素乘法被称为他们的哈达玛积。它与矩阵乘法不同。
- 在深度学习中，我们经常使用范数，如 L_1范数、L_2范数和弗罗贝尼乌斯范数。
- 我们可以对标量、向量、矩阵和张量执行各种操作。

1.3.13 练习

1. 证明一个矩阵 $\mathbf{A}$ 的转置的转置是 $\mathbf{A}$：$(\mathbf{A}^\top)^\top = \mathbf{A}$。
2. 给出两个矩阵 $\mathbf{A}$ 和 $\mathbf{B}$, 显示转置的和等于和的转置：$\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$.
3. 给定任意方矩阵$\mathbf{A}$，$\mathbf{A} + \mathbf{A}^\top$总是对称的吗?为什么?
4. 我们在本节中定义了形状（$2, 3, 4$）的张量 `X`。`len(X)`的输出结果是什么？
5. 对于任意形状的张量`X`, `len(X)`是否总是对应于`X`特定轴的长度?这个轴是什么?
6. 运行 `A / A.sum(axis=1)`，看看会发生什么。你能分析原因吗？
7. 当你在曼哈顿的两点之间旅行时，你需要在坐标上走多远，也就是说，就大街和街道而言？你能斜着走吗？
8. 考虑一个具有形状（$2, 3, 4$）的张量，在轴 0,1,2 上的求和输出是什么形状?
9. 向 `linalg.norm` 函数提供 3 个或更多轴的张量，并观察其输出。对于任意形状的张量这个函数计算得到什么?

Discussions[33]

1.4 微分

在2500年前，古希腊人把一个多边形分成三角形，并把它们的面积相加，才找到计算多边形面积的方法。为了求出曲线形状（比如圆）的面积，古希腊人在这样的形状上刻内接多边形。如 图1.4.1 所示，内接多边形的等长边越多，就越接近圆。这个过程也被称为 逼近法（method of exhaustion）。

[33] https://discuss.d2l.ai/t/1752

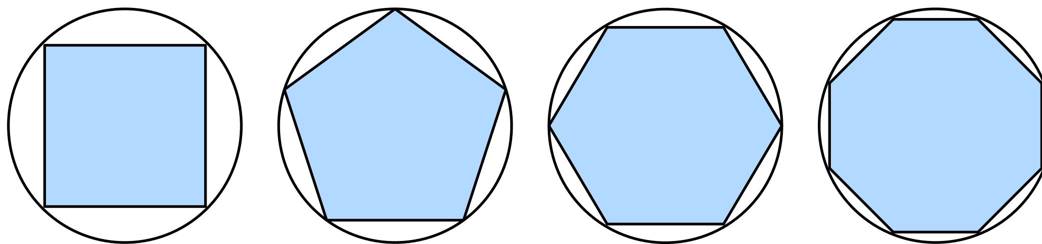


图1.4.1: 用逼近法求圆的面积。

事实上，逼近法就是 积分（integral calculus）的起源，我们将在 sec_integral_calculus 中详细描述。2000 多年后，微积分的另一支，微分（differential calculus），被发明出来。在微分学最重要的应用是优化问题，即考虑如何把事情做到最好。正如在 1.3.10节 中讨论的那样，这种问题在深度学习中是无处不在的。

在深度学习中，我们"训练"模型，不断更新它们，使它们在看到越来越多的数据时变得越来越好。通常情况下，变得更好意味着最小化一个 损失函数（loss function），即一个衡量"我们的模型有多糟糕"这个问题的分数。这个问题比看上去要微妙得多。最终，我们真正关心的是生成一个能够在我们从未见过的数据上表现良好的模型。但我们只能将模型与我们实际能看到的数据相拟合。因此，我们可以将拟合模型的任务分解为两个关键问题：（1）优化（optimization）：用模型拟合观测数据的过程；（2）泛化（generalization）：数学原理和实践者的智慧，能够指导我们生成出有效性超出用于训练的数据集本身的模型。

为了帮助你在后面的章节中更好地理解优化问题和方法，这里我们对深度学习中常用的微分知识提供了一个非常简短的入门教程。

1.4.1 导数和微分

我们首先讨论导数的计算，这是几乎所有深度学习优化算法的关键步骤。在深度学习中，我们通常选择对于模型参数可微的损失函数。简而言之，这意味着，对于每个参数，如果我们把这个参数增加或减少一个无穷小的量，我们可以知道损失会以多快的速度增加或减少，

假设我们有一个函数 $f: \mathbb{R} \rightarrow \mathbb{R}$，其输入和输出都是标量。$f$ 的 导数被定义为

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}, \tag{1.4.1}$$

如果这个极限存在。如果$f'(a)$存在，则称f在a处是可微（differentiable）的。如果 f 在一个区间内的每个数上都是可微的，则此函数在此区间中是可微的。我们可以将 (1.4.1) 中的导数 $f'(x)$ 解释为$f(x)$相对于 x 的 瞬时（instantaneous）变化率。所谓的瞬时变化率是基于x中的变化h，且h接近0。

为了更好地解释导数，让我们用一个例子来做实验。定义$u = f(x) = 3x^2 - 4x$.

```
%matplotlib inline
from IPython import display
from mxnet import np, npx
from d2l import mxnet as d2l

npx.set_np()
```

(continues on next page)

(continued from previous page)

```
def f(x):
    return 3 * x ** 2 - 4 * x
```

通过令 $x=1$ 并让 h 接近 0，(1.4.1) 中 $\frac{f(x+h)-f(x)}{h}$ 的数值结果接近 2。虽然这个实验不是一个数学证明，但我们稍后会看到，当 $x=1$时，导数 u'是 2 。

```
def numerical_lim(f, x, h):
    return (f(x + h) - f(x)) / h

h = 0.1
for i in range(5):
    print(f'h={h:.5f}, numerical limit={numerical_lim(f, 1, h):.5f}')
    h *= 0.1
```

```
h=0.10000, numerical limit=2.30000
h=0.01000, numerical limit=2.03000
h=0.00100, numerical limit=2.00300
h=0.00010, numerical limit=2.00030
h=0.00001, numerical limit=2.00003
```

让我们熟悉一下导数的几个等价符号。给定 $y=f(x)$，其中 x 和 y 分别是函数 f 的自变量和因变量。以下表达式是等价的：

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx}f(x) = Df(x) = D_x f(x), \tag{1.4.2}$$

其中符号 $\frac{d}{dx}$ 和 D 是微分运算符，表示微分操作。我们可以使用以下规则来对常见函数求微分：

- $DC=0$ （C 是一个常数）
- $Dx^n = nx^{n-1}$ （幂律（power rule），n是任意实数）
- $De^x = e^x$
- $D\ln(x) = 1/x$

为了微分一个由一些简单函数(如上面的常见函数)组成的函数,下面的法则使用起来很方便。假设函数f和g都是可微的，C是一个常数，我们有：

常数相乘法则

$$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx}f(x), \tag{1.4.3}$$

加法法则

$$\frac{d}{dx}[f(x)+g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \tag{1.4.4}$$

乘法法则

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \tag{1.4.5}$$

除法法则

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}. \tag{1.4.6}$$

现在我们可以应用上述几个法则来计算 $u' = f'(x) = 3\frac{d}{dx}x^2 - 4\frac{d}{dx}x = 6x - 4$。因此，通过令 $x = 1$，我们有 $u' = 2$：这一点得到了我们在本节前面的实验的支持，在这个实验中，数值结果接近2。当 $x = 1$ 时，此导数也是曲线 $u = f(x)$ 切线的斜率。

为了对导数的这种解释进行可视化，我们将使用 `matplotlib`，这是一个Python中流行的绘图库。要配置`matplotlib`生成图形的属性，我们需要定义几个函数。在下面，`use_svg_display` 函数指定 `matplotlib` 软件包输出svg图表以获得更清晰的图像。

注意，注释`#@save`是一个特殊的标记，会将对应的函数、类或语句保存在`d2l`包中因此，以后无需重新定义就可以直接调用它们（例如，`d2l.use_svg_display()`）。

```python
def use_svg_display():  #@save
    """使用svg格式在Jupyter中显示绘图。"""
    display.set_matplotlib_formats('svg')
```

我们定义 `set_figsize` 函数来设置图表大小。注意，这里我们直接使用 `d2l.plt`，因为导入语句 `from matplotlib import pyplot as plt` 已在前言中标记为保存到`d2l` 包中。

```python
def set_figsize(figsize=(3.5, 2.5)):  #@save
    """设置matplotlib的图表大小。"""
    use_svg_display()
    d2l.plt.rcParams['figure.figsize'] = figsize
```

下面的`set_axes`函数用于设置由`matplotlib`生成图表的轴的属性。

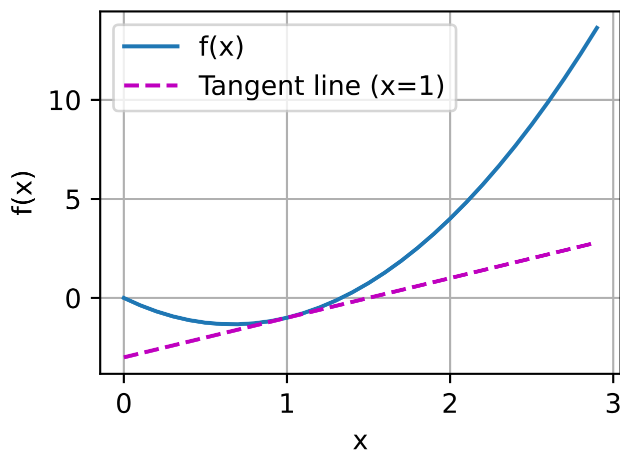
```python
#@save
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """设置matplotlib的轴。"""
    axes.set_xlabel(xlabel)
    axes.set_ylabel(ylabel)
    axes.set_xscale(xscale)
    axes.set_yscale(yscale)
    axes.set_xlim(xlim)
    axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()
```

通过这三个用于图形配置的函数，我们定义了 `plot` 函数来简洁地绘制多条曲线，因为我们需要在整个书中可视化许多曲线。

```python
#@save
def plot(X, Y=None, xlabel=None, ylabel=None, legend=None, xlim=None,
         ylim=None, xscale='linear', yscale='linear',
         fmts=('-', 'm--', 'g-.', 'r:'), figsize=(3.5, 2.5), axes=None):
    """绘制数据点。"""
    if legend is None:
        legend = []

    set_figsize(figsize)
    axes = axes if axes else d2l.plt.gca()

    # 如果 `X` 有一个轴，输出True
    def has_one_axis(X):
        return (hasattr(X, "ndim") and X.ndim == 1 or
                isinstance(X, list) and not hasattr(X[0], "__len__"))

    if has_one_axis(X):
        X = [X]
    if Y is None:
        X, Y = [[]] * len(X), X
    elif has_one_axis(Y):
        Y = [Y]
    if len(X) != len(Y):
        X = X * len(Y)
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        if len(x):
            axes.plot(x, y, fmt)
        else:
            axes.plot(y, fmt)
    set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
```

现在我们可以绘制函数 $u = f(x)$ 及其在 $x = 1$ 处的切线 $y = 2x - 3$，其中系数2是切线的斜率。

```python
x = np.arange(0, 3, 0.1)
plot(x, [f(x), 2 * x - 3], 'x', 'f(x)', legend=['f(x)', 'Tangent line (x=1)'])
```



1.4.2 偏导数

到目前为止，我们只讨论了仅含一个变量的函数的微分。在深度学习中，函数通常依赖于许多变量。因此，我们需要将微分的思想推广到这些 多元函数（multivariate function）上。

设 $y = f(x_1, x_2, \ldots, x_n)$ 是一个具有 n 个变量的函数。y 关于第i 个参数x_i的偏导数（partial derivative）为：

$$\frac{\partial y}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \ldots, x_{i-1}, x_i+h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}. \tag{1.4.7}$$

为了计算 $\frac{\partial y}{\partial x_i}$，我们可以简单地将 $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ 看作常数，并计算 y关于x_i 的导数。对于偏导数的表示，以下是等价的：

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f. \tag{1.4.8}$$

1.4.3 梯度

我们可以连结一个多元函数对其所有变量的偏导数，以得到该函数的梯度（gradient）向量。设函数 $f: \mathbb{R}^n \rightarrow \mathbb{R}$ 的输入是一个 n 维向量 $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\top$，并且输出是一个标量。函数$f(\mathbf{x})$相对于$\mathbf{x}$的梯度是一个包含$n$个偏导数的向量：

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \bigg[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\bigg]^\top, \tag{1.4.9}$$

其中 $\nabla_{\mathbf{x}} f(\mathbf{x})$ 通常在没有歧义时被 $\nabla f(\mathbf{x})$ 取代。

假设$\mathbf{x}$为n维向量，在微分多元函数时经常使用以下规则：

- 对于所有$\mathbf{A} \in \mathbb{R}^{m \times n}$，都有 $\nabla_{\mathbf{x}} \mathbf{A} \mathbf{x} = \mathbf{A}^\top$
- 对于所有$\mathbf{A} \in \mathbb{R}^{n \times m}$，都有 $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$
- 对于所有$\mathbf{A} \in \mathbb{R}^{n \times n}$，都有 $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} \mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$
- $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$

同样，对于任何矩阵 $\mathbf{X}$，我们都有 $\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$。正如我们之后将看到的，梯度对于设计深度学习中的优化算法有很大用处。

1.4.4 链式法则

然而，上面方法可能很难找到梯度。这是因为在深度学习中，多元函数通常是 复合（composite）的，所以我们可能没法应用上述任何规则来微分这些函数。幸运的是，链式法则使我们能够微分复合函数。

让我们先考虑单变量函数。假设函数 $y = f(u)$ 和 $u = g(x)$ 都是可微的，根据链式法则：

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}. \tag{1.4.10}$$

现在让我们把注意力转向一个更一般的场景，即函数具有任意数量的变量的情况。假设可微分函数 y 有变量 $u_1, u_2, \ldots, u_m$，其中每个可微分函数 u_i 都有变量 $x_1, x_2, \ldots, x_n$。注意，y是 x_1, x_2⊠$\ldots, x_n$ 的函数。对于任意 $i = 1, 2, \ldots, n$，链式法则给出：

$$\frac{dy}{dx_i} = \frac{dy}{du_1}\frac{du_1}{dx_i} + \frac{dy}{du_2}\frac{du_2}{dx_i} + \cdots + \frac{dy}{du_m}\frac{du_m}{dx_i} \tag{1.4.11}$$

1.4.5 小结

- 微分和积分是微积分的两个分支，其中前者可以应用于深度学习中无处不在的优化问题。
- 导数可以被解释为函数相对于其变量的瞬时变化率。它也是函数曲线的切线的斜率。
- 梯度是一个向量，其分量是多变量函数相对于其所有变量的偏导数。
- 链式法则使我们能够微分复合函数。

1.4.6 练习

1. 绘制函数$y = f(x) = x^3 - \frac{1}{x}$和其在$x = 1处切线的图像。
2. 求函数$f(\mathbf{x}) = 3x_1^2 + 5e^{x_2}$的梯度。
3. 函数$f(\mathbf{x}) = \|\mathbf{x}\|_2$的梯度是什么？
4. 你可以写出函数$u = f(x, y, z)$，其中$x = x(a, b)$，$y = y(a, b)$，$z = z(a, b)$的链式法则吗?

Discussions[34]

[34] https://discuss.d2l.ai/t/1755

1.5 自动求导

正如我们在 1.4节 中所说的那样，求导是几乎所有深度学习优化算法的关键步骤。虽然求导的计算很简单，只需要一些基本的微积分，但对于复杂的模型，手工进行更新是一件很痛苦的事情（而且经常容易出错）。

深度学习框架通过自动计算导数，即 自动求导（automatic differentiation），来加快这项工作。实际中，根据我们设计的模型，系统会构建一个 计算图（computational graph），来跟踪数据通过若干操作组合起来产生输出。自动求导使系统能够随后反向传播梯度。这里，反向传播（backpropagate）只是意味着跟踪整个计算图，填充关于每个参数的偏导数。

1.5.1 一个简单的例子

作为一个演示例子，假设我们想对函数 $y = 2\mathbf{x}^{\top}\mathbf{x}$关于列向量 $\mathbf{x}$求导。首先，我们创建变量 x 并为其分配一个初始值。

```
from mxnet import autograd, np, npx

npx.set_np()

x = np.arange(4.0)
x
```

```
array([0., 1., 2., 3.])
```

在我们计算y关于$\mathbf{x}$的梯度之前，我们需要一个地方来存储梯度。重要的是，我们不会在每次对一个参数求导时都分配新的内存。因为我们经常会成千上万次地更新相同的参数，每次都分配新的内存可能很快就会将内存耗尽。注意，标量函数关于向量$\mathbf{x}$的梯度是向量，并且与$\mathbf{x}$具有相同的形状。

```
# 我们通过调用`attach_grad`来为一个张量的梯度分配内存
x.attach_grad()
# 在我们计算关于`x`的梯度后，我们将能够通过'grad'属性访问它，它的值被初始化为0
x.grad
```

```
array([0., 0., 0., 0.])
```

现在让我们计算 y。

```
# 把代码放到`autograd.record`内，以建立计算图
with autograd.record():
    y = 2 * np.dot(x, x)
y
```

```
array(28.)
```

x 是一个长度为 4 的向量，计算 x 和 x 的内积，得到了我们赋值给 y 的标量输出。接下来，我们可以通过调用反向传播函数来自动计算y关于x 每个分量的梯度，并打印这些梯度。

```
y.backward()
x.grad
```

```
array([ 0.,  4.,  8., 12.])
```

函数 $y = 2\mathbf{x}^\top\mathbf{x}$ 关于$\mathbf{x}$ 的梯度应为$4\mathbf{x}$。让我们快速验证我们想要的梯度是否正确计算。

```
x.grad == 4 * x
```

```
array([ True,  True,  True,  True])
```

现在让我们计算 x 的另一个函数。

```
with autograd.record():
    y = x.sum()
y.backward()
x.grad  # 被新计算的梯度覆盖
```

```
array([1., 1., 1., 1.])
```

1.5.2 非标量变量的反向传播

当 y 不是标量时，向量y关于向量x的导数的最自然解释是一个矩阵。对于高阶和高维的 y 和 x，求导的结果可以是一个高阶张量。

然而，虽然这些更奇特的对象确实出现在高级机器学习中（包括深度学习中），但当我们调用向量的反向计算时，我们通常会试图计算一批训练样本中每个组成部分的损失函数的导数。这里，我们的目的不是计算微分矩阵，而是批量中每个样本单独计算的偏导数之和。

```
# 当我们对向量值变量`y`（关于`x`的函数）调用`backward`时，
# 将通过对`y`中的元素求和来创建一个新的标量变量。然后计算这个标量变量相对于`x`的梯度
with autograd.record():
    y = x * x  # `y`是一个向量
y.backward()
x.grad  # 等价于y = sum(x * x)
```

```
array([0., 2., 4., 6.])
```

1.5.3 分离计算

有时,我们希望将某些计算移动到记录的计算图之外。例如,假设y是作为x的函数计算的,而z则是作为y和x的函数计算的。现在,想象一下,我们想计算 z 关于 x 的梯度,但由于某种原因,我们希望将 y 视为一个常数,并且只考虑到 x 在y被计算后发挥的作用。

在这里,我们可以分离 y 来返回一个新变量 u,该变量与 y 具有相同的值,但丢弃计算图中如何计算 y 的任何信息。换句话说,梯度不会向后流经 u 到 x。因此,下面的反向传播函数计算 `z = u * x` 关于 x 的偏导数,同时将 u 作为常数处理,而不是`z = x * x * x`关于 x 的偏导数。

```
with autograd.record():
    y = x * x
    u = y.detach()
    z = u * x
z.backward()
x.grad == u
```

```
array([ True,  True,  True,  True])
```

由于记录了 y 的计算结果,我们可以随后在 y 上调用反向传播,得到 `y = x * x` 关于的x的导数,这里是 `2 * x`。

```
y.backward()
x.grad == 2 * x
```

```
array([ True,  True,  True,  True])
```

1.5.4 Python控制流的梯度计算

使用自动求导的一个好处是,即使构建函数的计算图需要通过 Python控制流(例如,条件、循环或任意函数调用),我们仍然可以计算得到的变量的梯度。在下面的代码中,`while` 循环的迭代次数和 `if` 语句的结果都取决于输入 a 的值。

```
def f(a):
    b = a * 2
    while np.linalg.norm(b) < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

让我们计算梯度。

```python
a = np.random.normal()
a.attach_grad()
with autograd.record():
    d = f(a)
d.backward()
```

我们现在可以分析上面定义的 f 函数。请注意，它在其输入 a 中是分段线性的。换言之，对于任何 a，存在某个常量标量 k，使得 f(a) = k * a，其中 k 的值取决于输入 a。因此，d / a 允许我们验证梯度是否正确。

```python
a.grad == d / a
```

```
array(True)
```

1.5.5 小结

- 深度学习框架可以自动计算导数。为了使用它，我们首先将梯度附加到想要对其计算偏导数的变量上。然后我们记录目标值的计算，执行它的反向传播函数，并访问得到的梯度。

1.5.6 练习

1. 为什么计算二阶导数比一阶导数的开销要更大？
2. 在运行反向传播函数之后，立即再次运行它，看看会发生什么。
3. 在控制流的例子中，我们计算 d 关于 a 的导数，如果我们将变量 a 更改为随机向量或矩阵，会发生什么？此时，计算结果 f(a) 不再是标量。结果会发生什么？我们如何分析这个结果？
4. 重新设计一个求控制流梯度的例子。运行并分析结果。
5. 使 $f(x) = \sin(x)$，绘制 $f(x)$ 和 $\frac{df(x)}{dx}$ 的图像，其中后者不使用 $f'(x) = \cos(x)$。

Discussions[35]

1.6 概率

在某种形式上，机器学习就是做出预测。

根据病人的临床病史，我们可能想预测他们在下一年心脏病发作的概率。在异常检测中，我们可能想要评估飞机喷气发动机的一组读数是正常运行情况的可能性有多大。在强化学习中，我们希望智能体（agent）能在一个环境中智能地行动。这意味着我们需要考虑在每种可行的行为下获得高奖励的概率。当我们建立推荐系统时，我们也需要考虑概率。例如，假设我们为一家大型在线书店工作。我们可能希望估计特定用户购买特定图书的概率。为此，我们需要使用概率学。有完整的课程、专业、论文、职业、甚至院系，都致力于概率

[35] https://discuss.d2l.ai/t/1758

学的工作。所以很自然地，我们在这部分的目标不是教授整个科目。相反，我们希望让你起步，教给你足够的知识，使你能够开始构建你的第一个深度学习模型，并让你对该主题有足够的了解，以便你可以开始自己探索它。

在前面的章节中，我们已经提到了概率，但没有明确说明它们是什么，也没有给出具体的例子。现在让我们更认真地考虑第一个例子：根据照片区分猫和狗。这听起来可能很简单，但实际上是一个艰巨的挑战。首先，问题的难度可能取决于图像的分辨率。

图1.6.1: 不同分辨率的图像 (10×10, 20×20, 40×40, 80×80, 和 160×160 pixels).

如 图1.6.1 所示，虽然人类很容易以 160×160 像素的分辨率识别猫和狗，但它在 40×40 像素上变得具有挑战性，而且在 10×10 像素下几乎是不可能的。换句话说，我们在很远的距离（从而降低分辨率）区分猫和狗的能力可能会接近不知情的猜测。概率给了我们一种正式的途径来说明我们的确定性水平。如果我们完全肯定图像是一只猫，我们说标签y是“猫”的概率，表示为$P(y=$“猫”$)$ 等于 1。如果我们没有证据表明 $y=$“猫”或 $y=$“狗”，那么我们可以说这两种可能性是等可能的，把它表示为$P(y=$“猫”$)=P(y=$“狗”$)=0.5$。如果我们有足够的信心，但不确定图像描绘的是一只猫，我们可以将概率赋值为$0.5<P(y=$“猫”$)<1$。

现在考虑第二个例子：给出一些天气监测数据，我们想预测明天北京下雨的概率。如果是夏天，下雨的概率是0.5。

在这两种情况下，我们都不确定结果。但这两种情况之间有一个关键区别。在第一种情况中，图像实际上是狗或猫，我们只是不知道哪个。在第二种情况下，结果实际上可能是一个随机的事件（如果你相信这些东西。大多数物理学家都相信）。因此，概率是一种灵活的语言，用于说明我们的确定程度，并且它可以有效地应用于广泛的上下文中。

1.6.1 基本概率论

假设我们掷骰子，想知道看到1的几率有多大，而不是看到另一个数字。如果骰子是公平的，那么所有六个结果$\{1, \ldots, 6\}$都有相同的可能发生，因此我们将在每六次中看到一个1。我们可以说1发生的概率为$\frac{1}{6}$。

对于我们从工厂收到的真实骰子，我们可能不知道那些比例，我们需要检查它是否有污染。调查骰子的唯一方法是多次投掷并记录结果。对于每个骰子，我们将观察到 $\{1, \ldots, 6\}$ 中的一个值。给定这些结果，我们想调查每个结果的概率。

对于每个值，一种自然的方法是将单个计数的值除以投掷的总次数。这给了我们一个给定事件的概率的估计值。大数定律（law of large numbers）告诉我们，随着投掷次数的增加，这个估计值会越来越接近真实的潜在概率。在深入了解这里的细节之前，让我们先试一试。

首先，让我们导入必要的软件包。

```
%matplotlib inline
import random
from mxnet import np, npx
from d2l import mxnet as d2l

npx.set_np()
```

接下来，我们将希望能够投掷骰子。在统计学中，我们把从概率分布中抽取样本的过程称为抽样（sampling）。将概率分配给一些离散选择的分布称为多项分布（multinomial distribution）。稍后我们将给出分布（distribution）的更正式定义。但笼统来说，可以把它看作是对事件的概率分配。

为了抽取一个样本，我们只需传入一个概率向量。输出是另一个相同长度的向量：它在索引i处的值是采样结果对应于i的次数。

```
fair_probs = [1.0 / 6] * 6
np.random.multinomial(1, fair_probs)
```

```
array([0, 0, 0, 1, 0, 0], dtype=int64)
```

如果你运行采样器很多次，你会发现每次你都得到随机的值。在估计一个骰子的公平性时，我们经常希望从同一分布中生成多个样本。如果用Python的for循环来完成这个任务，速度会慢得令人难以忍受，因此我们使用的函数支持同时抽取多个样本，返回我们想要的任意形状的独立样本数组。

```
np.random.multinomial(10, fair_probs)
```

```
array([1, 1, 5, 1, 1, 1], dtype=int64)
```

现在我们知道如何对骰子进行采样，我们可以模拟1000次投掷。然后，我们可以统计1000次投掷后，每个数字被投中了多少次。具体来说，我们计算相对频率作为真实概率的估计。

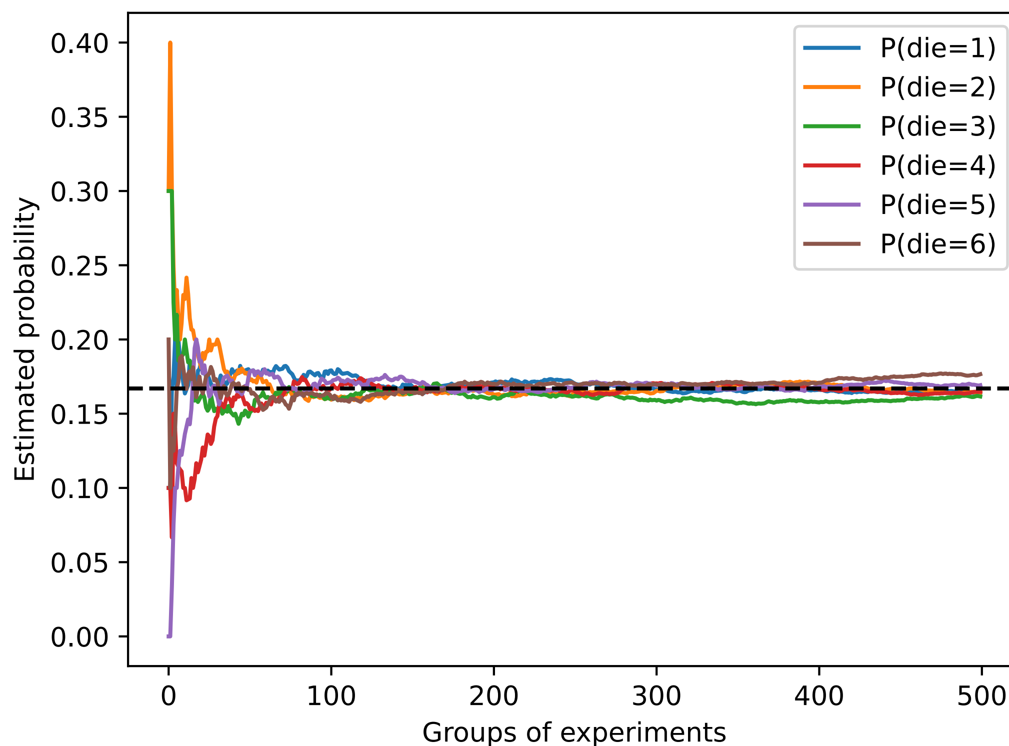
```
counts = np.random.multinomial(1000, fair_probs).astype(np.float32)
counts / 1000
```

```
array([0.162, 0.149, 0.178, 0.17 , 0.166, 0.175])
```

因为我们是从一个公平的骰子中生成的数据，我们知道每个结果都有真实的概率$\frac{1}{6}$，大约是0.167，所以上面输出的估计值看起来不错。

我们也可以看到这些概率如何随着时间的推移收敛到真实概率。让我们进行500组实验，每组抽取10个样本。

```
counts = np.random.multinomial(10, fair_probs, size=500)
cum_counts = counts.astype(np.float32).cumsum(axis=0)
estimates = cum_counts / cum_counts.sum(axis=1, keepdims=True)

d2l.set_figsize((6, 4.5))
for i in range(6):
    d2l.plt.plot(estimates[:, i].asnumpy(),
                 label=("P(die=" + str(i + 1) + ")"))
d2l.plt.axhline(y=0.167, color='black', linestyle='dashed')
d2l.plt.gca().set_xlabel('Groups of experiments')
d2l.plt.gca().set_ylabel('Estimated probability')
d2l.plt.legend();
```



每条实线对应于骰子的6个值中的一个，并给出骰子在每组实验后出现值的估计概率。当我们通过更多的实验获得更多的数据时，这6条实体曲线向真实概率收敛。

概率论公理

在处理骰子掷出时，我们将集合 $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ 称为 样本空间（sample space）或 结果空间（outcome space），其中每个元素都是 结果（outcome）。事件（event）是来自给定样本空间的一组结果。例如，“看到5”（$\{5\}$）和“看到奇数”（$\{1, 3, 5\}$）都是掷出骰子的有效事件。注意，如果随机实验的结果在事件 $\mathcal{A}$ 中，则事件 $\mathcal{A}$ 已经发生。也就是说，如果投掷出3点，因为 $3 \in \{1, 3, 5\}$，我们可以说，“看到奇数”的事件发生了。

形式上，概率（probability）可以被认为是将集合映射到真实值的函数。在给定的样本空间 $\mathcal{S}$ 中，事件$\mathcal{A}$的概率，表示为 $P(\mathcal{A})$，满足以下属性:

- 对于任意事件 $\mathcal{A}$，其概率从不会是负数，即 $P(\mathcal{A}) \geq 0$;
- 整个样本空间的概率为 1，即 $P(\mathcal{S}) = 1$;
- 对于任意事件 $\mathcal{A}_1, \mathcal{A}_2, \ldots$ 的可数序列，这些事件互斥（mutually exclusive）（对于所有 $i \neq j$ 都有 $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$），任何事件发生的概率等于它们各自发生的概率之和，即 $P(\bigcup_{i=1}^{\infty} \mathcal{A}_i) = \sum_{i=1}^{\infty} P(\mathcal{A}_i)$。

这些也是概率论的公理，由科尔莫戈罗夫于 1933 年提出。有了这个公理系统，我们可以避免任何关于随机性的哲学争论；相反，我们可以用数学语言严格地推理。例如，让事件 $\mathcal{A}_1$ 为整个样本空间，且当所有$i > 1$时的$\mathcal{A}_i = \emptyset$，我们可以证明 $P(\emptyset) = 0$，即不可能发生事件的概率是 0。

随机变量

在我们掷骰子的随机实验中，我们引入了 随机变量（random variable）的概念。随机变量几乎可以是任何数量，并且不是确定性的。它可以在随机实验的一组可能性中取一个值。考虑一个随机变量 X，其值在掷骰子的样本空间 $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ 中。我们可以将事件“看到一个 5”表示为 $\{X = 5\}$ 或 $X = 5$，其概率表示为 $P(\{X = 5\})$ 或 $P(X = 5)$。通过 $P(X = a)$，我们区分了随机变量 X 和 X 可以采取的值（例如 a）。然而，这可能会导致繁琐的表示。为了简化符号，一方面，我们可以将 $P(X)$ 表示为随机变量 X 上的 分布（distribution）：分布告诉我们 X 获得任意值的概率。另一方面，我们可以简单用 $P(a)$ 表示随机变量取值 a 的概率。由于概率论中的事件是来自样本空间的一组结果，因此我们可以为随机变量指定值的可取范围。例如，$P(1 \leq X \leq 3)$ 表示事件的概率 $\{1 \leq X \leq 3\}$，这意味着 $\{X = 1, 2, \text{or}, 3\}$。等价地，$P(1 \leq X \leq 3)$ 表示随机变量 X 从 $\{1, 2, 3\}$ 中取值的概率。

请注意，离散 (discrete) 随机变量（如骰子的侧面）和 连续 (continuous) 变量（如人的体重和身高）之间存在微妙的区别。问两个人是否具有完全相同的身高没有什么意义。如果我们进行足够精确的测量，你会发现这个星球上没有两个人具有完全相同的身高。事实上，如果我们采取足够精细的测量，在你起床和去睡觉时都不会得到相同的身高。因此，问一个人身高为 1.80139278297192196202 米高的概率是没有任何意义的。考虑到世界上的人口数量，这个概率几乎是 0。在这种情况下，询问某人的身高是否落入给定的区间，比如是否在 1.79 米和 1.81 米之间更有意义。在这些情况下，我们将这个看到某个数值的可能性量化为 密度 (density)。恰好 1.80 米的高度上没有概率，但密度不是 0。在任何两个不同高度之间的区间，我们都有非零的概率。在本节的其余部分中，我们将考虑离散空间中的概率。对于连续随机变量的概率，您可以参考 `sec_random_variables`。

1.6.2 处理多个随机变量

很多时候，我们会希望一次考虑多个随机变量。比如，我们可能需要对疾病和症状之间的关系进行建模。给定一个疾病和一个症状，比如“流感”和“咳嗽”，会在某个患者身上，以某个概率存在或不存在关系。虽然我们可能希望这两者发生的概率都接近于零，但我们可能需要估计这些概率和它们之间的关系，以便我们可以运用我们的推断来实现更好的医疗服务。

再举一个更复杂的例子：图像包含数百万像素，因此有数百万个随机变量。在许多情况下，图像会附带一个标签，标识图像中的对象。我们也可以将标签视为一个随机变量。我们甚至可以将所有元数据视为随机变量，例如位置、时间、光圈、焦距、ISO、对焦距离和相机类型。所有这些都是联合发生的随机变量。当我们处理多个随机变量时，会有若干个变量是我们感兴趣的。

联合概率

第一个被称为 联合概率 (joint probability) $P(A=a,B=b)$。给定任何值 a 和 b, 联合概率可以回答, $A=a$ 和 $B=b$ 同时满足的概率是多少? 请注意，对于任何值，对于任何 a 和 b 的取值，$P(A=a,B=b)\leq P(A=a)$。这点是确定的，因为要同时发生 $A=a$ 和 $B=b$，$A=a$就必须发生，$B=b$也必须发生（反之亦然）。因此，$A=a$ 和 $B=b$ 同时发生的可能性不大于 $A=a$ 或是 $B=b$ 的可能性。

条件概率

这给我们带来了一个有趣的比率：$0\leq\frac{P(A=a,B=b)}{P(A=a)}\leq 1$。我们称这个比率为 条件概率 (conditional probability), 并用 $P(B=b\mid A=a)$ 表示它：它是 $B=b$ 的概率，前提是发生了 $A=a$。

贝叶斯定理

使用条件概率的定义，我们可以得出统计数据中最有用和最著名的方程之一：*Bayes* 定理 (Bayes’theorem)。它如下所示。通过构造，我们有 乘法规则，$P(A,B)=P(B\mid A)P(A)$。根据对称性，这也适用于 $P(A,B)=P(A\mid B)P(B)$。假设 $P(B)>0$，求解其中一个条件变量，我们得到

$$P(A\mid B)=\frac{P(B\mid A)P(A)}{P(B)}. \tag{1.6.1}$$

请注意，在这里我们使用更紧凑的表示法，其中 $P(A,B)$ 是一个 联合分布，$P(A\mid B)$ 是一个 条件分布。这种分布可以在给定值 $A=a,B=b$ 上进行求值。

边际化

如果我们想从另一件事中推断一件事，但我们只知道相反方向的属性，比如因和果的时候，Bayes 定理是非常有用的，正如我们将在本节后面看到的那样。为了能进行这项工作，我们需要的一个重要操作是 边际化。这项操作是从 $P(A,B)$ 中确定 $P(B)$ 的操作。我们可以看到，B 的概率相当于计算 A 的所有可能选择，并将所有选择的联合概率聚合在一起：

$$P(B)=\sum_{A}P(A,B), \tag{1.6.2}$$

这也称为 求和规则。边际化结果的概率或分布称为 边际概率或 边际分布。

独立性

另一个要检查的有用属性是 依赖与 独立。两个随机变量 A 和 B 是独立的，意味着事件 A 的发生不会透露有关 B 事件的发生情况的任何信息。在这种情况下，统计学家通常将这一点表述为 $A \perp B$。根据贝叶斯定理，马上就能同样得到 $P(A \mid B) = P(A)$。在所有其他情况下，我们称 A 和 B 依赖。比如，一个骰子的两次连续抛出是独立的。相比之下，灯开关的位置和房间的亮度并不是（尽管它们不是具有确定性的，因为总是可能存在灯泡坏掉，电源故障，或者开关故障）。

由于 $P(A \mid B) = \frac{P(A,B)}{P(B)} = P(A)$ 等价于 $P(A, B) = P(A)P(B)$，因此两个随机变量是独立的当且仅当两个随机变量的联合分布是其各自分布的乘积。同样地，给定另一个随机变量 C时，两个随机变量 A 和 B 是 条件独立的，当且仅当 $P(A, B \mid C) = P(A \mid C)P(B \mid C)$ 。这个情况表示为 $A \perp B \mid C$。

应用

让我们用实战考验一下我们的技能。假设一个医生对患者进行艾滋病病毒（HIV）测试。这个测试是相当准确的，如果患者健康但测试显示他患病，这样的失败概率只有 1% 。此外，如果患者真正感染HIV，它永远不会检测不出。我们使用 D_1 来表示诊断结果（如果阳性，则为 1，如果阴性，则为 0），H 来表示感染艾滋病病毒的状态（如果阳性，则为 1，如果阴性，则为0）。在 表1.6.1 中列出了这样的条件概率。

表1.6.1: 条件概率为$P(D_1 \mid H)$。

条件概率	$H = 1$	$H = 0$
$P(D_1 = 1 \mid H)$	1	0.01
$P(D_1 = 0 \mid H)$	0	0.99

请注意，每列的加和都是 1（但每行的加和不是），因为条件概率需要总和为1，就像概率一样。让我们计算如果测试出来呈阳性，患者感染HIV的概率，即 $P(H = 1 \mid D_1 = 1)$。显然，这将取决于疾病有多常见，因为它会影响错误警报的数量。假设人口总体是相当健康的，例如，$P(H = 1) = 0.0015$。为了应用贝叶斯定理，我们需要运用边际化和乘法规则来确定

$$\begin{aligned} & P(D_1 = 1) \\ =& P(D_1 = 1, H = 0) + P(D_1 = 1, H = 1) \\ =& P(D_1 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1 \mid H = 1)P(H = 1) \\ =& 0.011485. \end{aligned} \tag{1.6.3}$$

因此，我们得到

$$\begin{aligned} & P(H = 1 \mid D_1 = 1) \\ =& \frac{P(D_1 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1)}. \\ =& 0.1306 \end{aligned} \tag{1.6.4}$$

换句话说，尽管使用了非常准确的测试，患者实际上患有艾滋病的几率只有 13.06%。正如我们所看到的，概率可能是违反直觉的。

患者在收到这样可怕的消息后应该怎么办？很可能，患者会要求医生进行另一次测试来了解清楚。第二个测试具有不同的特性，它不如第一个测试那么好，如 表1.6.2 所示。

表1.6.2: 条件概率为$P(D_2 \mid H)$。

条件概率	$H=1$	$H=0$
$P(D_2=1 \mid H)$	0.98	0.03
$P(D_2=0 \mid H)$	0.02	0.97

不幸的是，第二次测试也显示阳性。让我们通过假设条件独立性来计算出应用 Bayes 定理的必要概率：

$$\begin{aligned}&P(D_1=1, D_2=1 \mid H=0)\\=&P(D_1=1 \mid H=0)P(D_2=1 \mid H=0)\\=&0.0003,\end{aligned} \tag{1.6.5}$$

$$\begin{aligned}&P(D_1=1, D_2=1 \mid H=1)\\=&P(D_1=1 \mid H=1)P(D_2=1 \mid H=1)\\=&0.98.\end{aligned} \tag{1.6.6}$$

现在我们可以应用边际化和乘法规则：

$$\begin{aligned}&P(D_1=1, D_2=1)\\=&P(D_1=1, D_2=1, H=0)+P(D_1=1, D_2=1, H=1)\\=&P(D_1=1, D_2=1 \mid H=0)P(H=0)+P(D_1=1, D_2=1 \mid H=1)P(H=1)\\=&0.00176955.\end{aligned} \tag{1.6.7}$$

最后，鉴于存在两次阳性检测，患者患有艾滋病的概率为

$$\begin{aligned}&P(H=1 \mid D_1=1, D_2=1)\\=&\frac{P(D_1=1, D_2=1 \mid H=1)P(H=1)}{P(D_1=1, D_2=1)}\\=&0.8307.\end{aligned} \tag{1.6.8}$$

也就是说，第二次测试使我们能够对患病的情况获得更高的信心。尽管第二次检验比第一次检验的准确性要低得多，但它仍然显著改善了我们的估计。

1.6.3 期望和差异

为了概括概率分布的关键特征，我们需要一些测量方法。随机变量 X 的 期望（或平均值）表示为

$$E[X]=\sum_x xP(X=x). \tag{1.6.9}$$

当函数 $f(x)$ 的输入是从分布 P 中抽取的随机变量时，$f(x)$ 的期望值为

$$E_{x\sim P}[f(x)] = \sum_x f(x)P(x). \tag{1.6.10}$$

在许多情况下，我们希望衡量随机变量 X 与其期望值的偏置。这可以通过方差来量化

$$\mathrm{Var}[X] = E\left[(X - E[X])^2\right] = E[X^2] - E[X]^2. \tag{1.6.11}$$

它的平方根被称为 标准差 (standared deviation)。随机变量函数的方差衡量的是，当从该随机变量分布中采样不同值 x 时，函数值偏离该函数的期望的程度：

$$\mathrm{Var}[f(x)] = E\left[\left(f(x) - E[f(x)]\right)^2\right]. \tag{1.6.12}$$

1.6.4 小结

- 我们可以从概率分布中采样。
- 我们可以使用联合分布、条件分布、Bayes 定理、边缘化和独立性假设来分析多个随机变量。
- 期望和方差为概率分布的关键特征的概括提供了实用的度量形式。

1.6.5 练习

1. 我们进行了 $m = 500$ 组实验，每组抽取 $n = 10$ 个样本。变化 m 和 n，观察和分析实验结果。
2. 给定两个概率为 $P(\mathcal{A})$ 和 $P(\mathcal{B})$ 的事件，计算 $P(\mathcal{A}\cup\mathcal{B})$ 和 $P(\mathcal{A}\cap\mathcal{B})$ 的上限和下限。（提示：使用 友元图[36] 来展示这些情况。）
3. 假设我们有一系列随机变量，例如 A，B 和 C，其中 B 只依赖于 A，而 C 只依赖于 B，你能简化联合概率 $P(A, B, C)$ 吗？（提示：这是一个 马尔可夫链[37]。）
4. 在 1.6.2节 中，第一个测试更准确。为什么不运行第一个测试两次，而是同时运行第一个和第二个测试?

Discussions[38]

1.7 查阅文档

由于这本书篇幅的限制，我们不可能介绍每一个 MXNet 函数和类（你可能也不希望我们这样做）。API文档、其他教程和示例提供了本书之外的大量文档。在本节中，我们为你提供了一些查看MXNet API 的指导。

[36] https://en.wikipedia.org/wiki/Venn_diagram
[37] https://en.wikipedia.org/wiki/Markov_chain
[38] https://discuss.d2l.ai/t/1761

1.7.1 查找模块中的所有函数和类

为了知道模块中可以调用哪些函数和类，我们调用 dir 函数。例如，我们可以查询随机数生成模块中的所有属性：

```
from mxnet import np

print(dir(np.random))
```

```
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '_
↪_name__', '__package__', '__spec__', '_mx_nd_np', 'beta', 'chisquare', 'choice
↪', 'exponential', 'gamma', 'gumbel', 'logistic', 'lognormal', 'multinomial',
↪'multivariate_normal', 'normal', 'pareto', 'power', 'rand', 'randint', 'randn',
↪'rayleigh', 'shuffle', 'uniform', 'weibull']
```

通常，我们可以忽略以 __ 开始和结束的函数（Python 中的特殊对象）或以单个 _ 开始的函数（通常是内部函数）。根据剩余的函数名或属性名，我们可能会猜测这个模块提供了各种生成随机数的方法，包括从均匀分布（uniform）、正态分布（normal）和多项分布（multinomial）中采样。

1.7.2 查找特定函数和类的用法

有关如何使用给定函数或类的更具体说明，我们可以调用 help 函数。例如，我们来查看张量 ones 函数的用法。

```
help(np.ones)
```

```
Help on function ones in module mxnet.numpy:

ones(shape, dtype=<class 'numpy.float32'>, order='C', ctx=None)
    Return a new array of given shape and type, filled with ones.
    This function currently only supports storing multi-dimensional data
    in row-major (C-style).

    Parameters
    ----------
    shape : int or tuple of int
        The shape of the empty array.
    dtype : str or numpy.dtype, optional
        An optional value type. Default is `numpy.float32`. Note that this
        behavior is different from NumPy's `ones` function where `float64`
        is the default value, because `float32` is considered as the default
        data type in deep learning.
    order : {'C'}, optional, default: 'C'
```

```
        How to store multi-dimensional data in memory, currently only row-
major
        (C-style) is supported.
    ctx : Context, optional
        An optional device context (default is the current default context).

    Returns
    -------
    out : ndarray
        Array of ones with the given shape, dtype, and ctx.

    Examples
    --------
    >>> np.ones(5)
    array([1., 1., 1., 1., 1.])

    >>> np.ones((5,), dtype=int)
    array([1, 1, 1, 1, 1], dtype=int64)

    >>> np.ones((2, 1))
    array([[1.],
           [1.]])

    >>> s = (2,2)
    >>> np.ones(s)
    array([[1., 1.],
           [1., 1.]])
```

从文档中，我们可以看到 ones 函数创建一个具有指定形状的新张量，并将所有元素值设置为 1。让我们来运行一个快速测试来确认这一解释：

```
np.ones(4)
```

```
array([1., 1., 1., 1.])
```

在Jupyter记事本中，我们可以使用？在另一个窗口中显示文档。例如，list?将创建与help(list) 几乎相同的内容，并在新的浏览器窗口中显示它。此外，如果我们使用两个问号，如 list??，将显示实现该函数的 Python 代码。

1.7.3 小结

- 官方文档提供了本书之外的大量描述和示例。
- 我们可以通过调用 `dir` 和 `help` 函数或在Jupyter记事本中使用 `?` 和 `??`查看API的用法文档。

1.7.4 练习

1. 在深度学习框架中查找任何函数或类的文档。你能在这个框架的官方网站上找到文档吗?

Discussions[39]

[39] https://discuss.d2l.ai/t/1764

2

线性神经网络

在介绍深度神经网络之前，我们需要了解神经网络训练的基础知识。在本章中，我们将介绍神经网络的整个训练过程，包括：定义简单的神经网络架构、数据处理、指定损失函数和如何训练模型。经典统计学习技术中的线性回归和softmax回归可以视为 线性神经网络。为了更容易学习，我们将从这些经典算法开始，向你介绍神经网络的基础知识。这些知识将为本书其他部分中更复杂的技术奠定基础。

2.1 线性回归

回归（regression）是指一类为一个或多个自变量与因变量之间关系建模的方法。在自然科学和社会科学领域，回归经常用来表示输入和输出之间的关系。

在机器学习领域中的大多数任务通常都与预测（prediction）有关。当我们想预测一个数值时，就会涉及到回归问题。常见的例子包括：预测价格（房屋、股票等）、预测住院时间（针对住院病人）、预测需求（零售销量）等。但不是所有的预测都是回归问题。在后面的章节中，我们将介绍分类问题。分类问题的目标是预测数据属于一组类别中的哪一个。

2.1.1 线性回归的基本元素

线性回归（linear regression）在回归的各种标准工具中最简单而且最流行。它可以追溯到19世纪初。线性回归基于几个简单的假设：首先，假设自变量 $\mathbf{x}$ 和因变量 y 之间的关系是线性的，即y可以表示为 $\mathbf{x}$ 中元素的加权和，这里通常允许包含观测值的一些噪声；其次，我们假设任何噪声都比较正常，如噪声遵循正态分布。

为了解释线性回归，我们举一个实际的例子：我们希望根据房屋的面积（平方英尺）和房龄（年）来估算房屋价格（美元）。为了开发一个能预测房价的模型，我们需要收集一个真实的数据集。这个数据集包括了房屋的销售价格、面积和房龄。在机器学习的术语中，该数据集称为 训练数据集（training data set）或训练集（training set），每行数据（在这个例子中是与一次房屋交易相对应的数据）称为样本（sample），也可以称为数据点（data point）或数据样本（data instance）。我们要试图预测的目标（在这个例子中是房屋价格）称为 标签（label）或 目标（target）。预测所依据的自变量（面积和房龄）称为 特征（features）或 协变量（covariates）。

通常，我们使用 n 来表示数据集中的样本数。对索引为 i 的样本，其输入表示为 $\mathbf{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}]^\top$，其对应的标签是 $y^{(i)}$。

线性模型

线性假设是指目标（房屋价格）可以表示为特征（面积和房龄）的加权和，如下面的式子：

$$\mathrm{price} = w_{\mathrm{area}} \cdot \mathrm{area} + w_{\mathrm{age}} \cdot \mathrm{age} + b. \tag{2.1.1}$$

(2.1.1) 中的w_{area} 和 w_{age} 称为 权重（weight），b 称为 偏置（bias），或称为偏移量（offset）、截距（intercept）。权重决定了每个特征对我们预测值的影响。偏置是指当所有特征都取值为0时，预测值应该为多少。即使现实中不会有任何房子的面积是0或房龄正好是0年，我们仍然需要偏置项。如果没有偏置项，我们模型的表达能力将受到限制。严格来说，(2.1.1) 是输入特征的一个仿射变换（affine transformation）。仿射变换的特点是通过加权和对特征进行线性变换（linear transformation），并通过偏置项来进行平移（translation）。

给定一个数据集，我们的目标是寻找模型的权重 $\mathbf{w}$ 和偏置 b，使得根据模型做出的预测大体符合数据里的真实价格。输出的预测值由输入特征通过线性模型的仿射变换决定，仿射变换由所选权重和偏置确定。

有些学科往往只关注有少量特征的数据集。在这些学科中，建模时经常过通过显式地长形式表达。而在机器学习领域，我们通常使用的是高维数据集，建模时采用线性代数表示法会比较方便。当我们的输入包含 d 个特征时，我们将预测结果 $\hat{y}$（通常使用“尖角”符号表示估计值）表示为：

$$\hat{y} = w_1 x_1 + ... + w_d x_d + b. \tag{2.1.2}$$

将所有特征放到向量 $\mathbf{x} \in \mathbb{R}^d$ 中，并将所有权重放到向量 $\mathbf{w} \in \mathbb{R}^d$ 中，我们可以用点积形式来简洁地表达模型：

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b. \tag{2.1.3}$$

在 (2.1.3) 中，向量 $\mathbf{x}$ 对应于单个数据样本的特征。用符号表示的矩阵 $\mathbf{X} \in \mathbb{R}^{n \times d}$ 可以很方便地引用我们整个数据集的 n 个样本。其中，$\mathbf{X}$ 的每一行是一个样本，每一列是一种特征。

对于特征集合 $\mathbf{X}$ 和预测值 $\hat{\mathbf{y}} \in \mathbb{R}^n$ 可以通过矩阵-向量乘法表示为：

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b \tag{2.1.4}$$

这个过程的求和过程将使用广播机制。广播机制会在 1.1.3节)详细介绍。给定训练数据特征 $\mathbf{X}$ 和对应的已知标签 $\mathbf{y}$，线性回归的目标是找到一组权重向量 $\mathbf{w}$ 和偏置 b。当给定从$\mathbf{X}$的同分布中取样的新样本特征时，找到的权重向量和偏置能够使得新样本预测标签的误差尽可能小。

虽然我们相信给定 $\mathbf{x}$ 预测 y 的最佳模型会是线性的。但我们很难找到找到一个有n个样本的真实数据集，其中 $y^{(i)}$ 完全等于 $\mathbf{w}^\top\mathbf{x}^{(i)}+b$。无论我们使用什么手段来观察特征 $\mathbf{X}$ 和标签 $\mathbf{y}$，都可能会出现少量的观测误差。因此，即使确信特征与标签的潜在关系是线性的，我们也会加入一个噪声项来考虑观测误差带来的影响。

在我们开始寻找最好的 模型参数（model parameters）$\mathbf{w}$ 和 b 之前，我们还需要两个东西：（1）一种模型质量的度量方式；（2）一种能够更新模型以提高模型预测质量的方法。

损失函数

在我们开始考虑如何用模型拟合（fit）数据之前，我们需要确定一个拟合程度的度量。损失函数能够量化目标的实际值与预测值之间的差距。通常我们会选择非负数作为损失，且数值越小表示损失越小，完美预测时的损失为0。回归问题中最常用的损失函数是平方误差函数。当样本 i 的预测值为 $\hat{y}^{(i)}$，其相应的真实标签为 $y^{(i)}$ 时，平方误差可以定义为以下公式：

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2} \left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{2.1.5}$$

常数$\frac{1}{2}$不会带来本质的差别，但这样在形式上稍微简单一些。当我们对损失函数求导后常数系数为1。由于训练数据集并不受我们控制，所以经验误差只是关于模型参数的函数。为了进一步说明，来看下面的例子。我们为一维情况下的回归问题绘制图像，如 图2.1.1 所示。

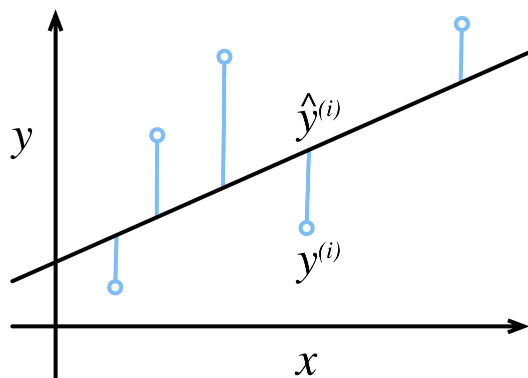


图2.1.1: 用线性模型拟合数据。

由于平方误差函数中的二次方项，估计值 $\hat{y}^{(i)}$ 和观测值 $y^{(i)}$ 之间较大的差异将贡献更大的损失。为了度量模型在整个数据集上的质量，我们需计算在训练集n个样本上的损失均值（也等价于求和）。

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^n l^{(i)}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{2.1.6}$$

在训练模型时，我们希望寻找一组参数 $(\mathbf{w}^*, b^*)$，这组参数能最小化在所有训练样本上的总损失。如下式：

$$\mathbf{w}^*, b^* = \operatorname*{argmin}_{\mathbf{w}, b}\ L(\mathbf{w}, b). \tag{2.1.7}$$

解析解

线性回归刚好是一个很简单的优化问题。与我们将在本书中所讲到的其他大部分模型不同，线性回归的解可以用一个公式简单地表达出来，这类解叫作解析解（analytical solution）。首先，我们将偏置 b 合并到参数 $\mathbf{w}$ 中。合并方法是在包含所有参数的矩阵中附加一列。我们的预测问题是最小化 $\|\mathbf{y}-\mathbf{X}\mathbf{w}\|^2$。这在损失平面上只有一个临界点，这个临界点对应于整个区域的损失最小值。将损失关于$\mathbf{w}$的导数设为0，得到解析解（闭合形式）：

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}. \tag{2.1.8}$$

像线性回归这样的简单问题存在解析解，但并不是所有的问题都存在解析解。解析解可以进行很好的数学分析，但解析解的限制很严格，导致它无法应用在深度学习里。

小批量随机梯度下降

即使在我们无法得到解析解的情况下，我们仍然可以有效地训练模型。在许多任务上，那些难以优化的模型效果要更好。因此，弄清楚如何训练这些难以优化的模型是非常重要的。

本书中我们用到一种名为梯度下降（gradient descent）的方法，这种方法几乎可以优化所有深度学习模型。它通过不断地在降低损失的方向上更新参数来降低误差。

梯度下降最简单的用法是计算数据集中所有样本的损失关于模型参数的导数（在这里也可以称为梯度）。但实际中的执行可能会比较慢：因为在进行一次更新之前，我们必须遍历整个数据集。因此，我们通常会在每次需要计算更新的时候随机抽取一小批样本，这种变体叫做小批量随机梯度下降（minibatch stochastic gradient descent）。

在每次迭代中，我们首先随机抽样一个小批量$\mathcal{B}$，它是由固定数量的训练样本组成的。然后，我们计算小批量的平均损失关于模型参数的导数（也可以称为梯度）。最后，我们将梯度乘以一个预先确定的正数η，并从当前参数的值中减掉。

我们用下面的数学公式来表示这一更新过程（∂ 表示偏导数）：

$$(\mathbf{w},b) \leftarrow (\mathbf{w},b) - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w},b). \tag{2.1.9}$$

总结一下，算法的步骤如下：（1）初始化模型参数的值，如随机初始化；（2）从数据中迭代抽取随机的小批量样本。然后在负梯度的方向上更新参数。对于平方损失和仿射变换，我们可以明确地写成如下形式:

$$\begin{aligned}
\mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w},b) = \mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \mathbf{x}^{(i)}\left(\mathbf{w}^\top\mathbf{x}^{(i)} + b - y^{(i)}\right),\\
b &\leftarrow b - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \partial_b l^{(i)}(\mathbf{w},b) = b - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \left(\mathbf{w}^\top\mathbf{x}^{(i)} + b - y^{(i)}\right).
\end{aligned} \tag{2.1.10}$$

公式 (2.1.10) 中的$\mathbf{w}$ 和 $\mathbf{x}$ 都是向量。在这里，更优雅的向量表示法比系数表示法（如 $w_1, w_2, \ldots, w_d$）更具可读性。$|\mathcal{B}|$ 表示每个小批量中的样本数，这也称为批量大小（batch size）。η 表示 学习率（learning rate）。批量大小和学习率的值通常是手动预先指定，而不是通过模型训练得到的。这些可以调整但不在训练过程中更新的参数称为 超参数（hyperparameter）。调参（hyperparameter tuning）是选择超参数的过程。超参数通

常是我们根据训练迭代结果来调整的，而训练迭代结果是在独立的验证数据集（validation dataset）上评估得到的。

在训练了预先确定的若干迭代次数后（或者直到满足某些其他停止条件后），我们记录估计的模型参数，表示为$\hat{\mathbf{w}}, \hat{b}$。但是，即使我们的函数真是线性的且无噪声。我们估计得到的参数也不会是损失的精确最小值。因为算法会使得损失向最小值缓慢收敛，但不能在有限的步数内非常精确地达到最小值。

线性回归恰好是一个在整个域中只有一个最小值的学习问题。但是对于像深度神经网络这样复杂的模型来说，损失平面上通常包含许多个最小值。幸运的是，深度学习实践者很少努力寻找能够将 训练集损失最小化的参数，虽然这么做原因尚未被完全理解。事实上，更难做到的是找到一组参数，这组参数能够在我们从未见过的数据上实现低的损失，这一挑战被称为 泛化（generalization）。

用学习到的模型进行预测

给定学习到的线性回归模型 $\hat{\mathbf{w}}^\top \mathbf{x} + \hat{b}$，现在我们可以通过给定的房屋面积 x_1 和房龄 x_2来估计一个未包含在训练数据中的新房屋价格。给定特征估计目标的过程通常称为预测（prediction）或推断（inference）。

我们将尝试坚持使用预测这个词。虽然推断这个词已经成为深度学习的标准术语，但其实推断这个词有些用词不当。在统计学中，推断更多地表示基于数据集估计参数。当深度学习从业者与统计学家交谈时，术语的误用经常导致一些误解。

2.1.2 矢量化加速

在训练我们的模型时，我们经常希望能够同时处理整个小批量的样本。为了实现这一点，需要我们对计算进行矢量化，从而利用好快速线性代数库，而不是在Python中编写开销高昂的for循环。

```
%matplotlib inline
import math
import time
from mxnet import np
from d2l import mxnet as d2l
```

为了说明矢量化为什么如此重要，我们考虑对两个向量相加的两种方法。我们实例化两个全1的1000维向量。在一种方法中，我们将使用Python的for循环遍历向量。在另一种方法中，我们将依赖对 + 的调用。

```
n = 10000
a = np.ones(n)
b = np.ones(n)
```

由于在本书中我们将频繁地进行运行时间的基准测试，所以让我们定义一个计时器。

```
class Timer:  #@save
    """记录多次运行时间。"""
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        self.times = []
        self.start()

    def start(self):
        """启动计时器。"""
        self.tik = time.time()

    def stop(self):
        """停止计时器并将时间记录在列表中。"""
        self.times.append(time.time() - self.tik)
        return self.times[-1]

    def avg(self):
        """返回平均时间。"""
        return sum(self.times) / len(self.times)

    def sum(self):
        """返回时间总和。"""
        return sum(self.times)

    def cumsum(self):
        """返回累计时间。"""
        return np.array(self.times).cumsum().tolist()
```

现在我们可以对工作负载进行基准测试。

首先，我们使用for循环，每次执行一位的加法。

```
c = np.zeros(n)
timer = Timer()
for i in range(n):
    c[i] = a[i] + b[i]
f'{timer.stop():.5f} sec'
```

```
'4.44498 sec'
```

然后，我们使用重载的 + 运算符来计算按元素的和。

```
timer.start()
d = a + b
f'{timer.stop():.5f} sec'
```

```
'0.00036 sec'
```

结果很明显，第二种方法比第一种方法快得多。矢量化代码通常会带来数量级的加速。另外，我们将更多的数学运算放到库中，而无需自己编写那么多的计算，从而减少了出错的可能性。

2.1.3 正态分布与平方损失

接下来，我们通过对噪声分布的假设来解读平方损失目标。

正态分布（normal distribution），也称为 高斯分布（Gaussian distribution），最早由德国数学家高斯（Gauss）应用于天文学研究。正态分布和线性回归之间的关系很密切。简单的说，若随机变量 x 具有均值 μ 和方差 σ^2（标准差 σ），其正态分布概率密度函数如下：

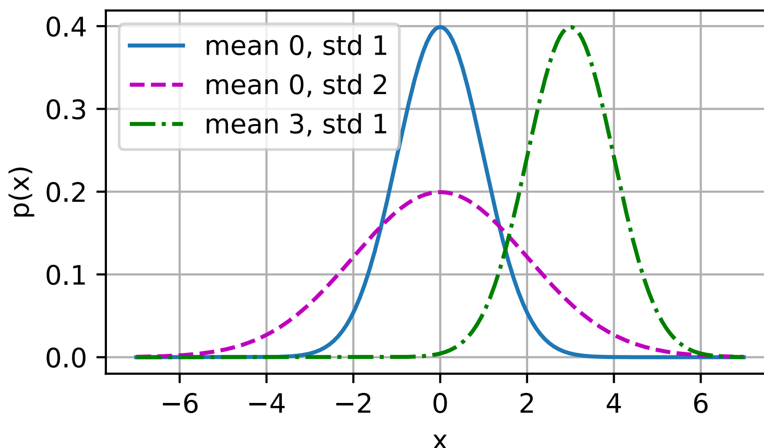
$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right). \tag{2.1.11}$$

下面我们定义一个Python函数来计算正态分布。

```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 / sigma**2 * (x - mu)**2)
```

我们现在可视化正态分布。

```
# 再次使用numpy进行可视化
x = np.arange(-7, 7, 0.01)

# 均值和标准差对
params = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
         ylabel='p(x)', figsize=(4.5, 2.5),
         legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```



就像我们所看到的，改变均值会产生沿 x 轴的偏移，增加方差将会分散分布、降低其峰值。

利用均方误差损失函数（简称均方损失）可以用于线性回归的一个原因是：假设观测中包含噪声，其中噪声服从正态分布。噪声正态分布如下式:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{2.1.12}$$

因此，我们现在可以写出通过给定的$\mathbf{x}$观测到特定y的可能性（likelihood）:

$$P(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \tag{2.1.13}$$

现在，根据最大似然估计法，参数 $\mathbf{w}$ 和 b 的最优值是使整个数据集的可能性最大的值:

$$P(\mathbf{y} \mid \mathbf{X}) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)}). \tag{2.1.14}$$

根据最大似然估计法选择的估计量称为最大似然估计量。虽然使许多指数函数的乘积最大化看起来很困难，但是我们可以在不改变目标的前提下，通过最大化似然对数来简化。由于历史原因，优化通常是说最小化而不是最大化。我们可以改为 最小化负对数似然 $-\log P(\mathbf{y} \mid \mathbf{X})$。由此可以得到的数学公式是:

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2. \tag{2.1.15}$$

现在我们只需要假设σ是某个固定常数就可以忽略第一项，因为第一项不依赖于 $\mathbf{w}$和b。现在第二项除了常数$\frac{1}{\sigma^2}$外，其余部分和前面介绍的平方误差损失是一样的。幸运的是，上面式子的解并不依赖于 σ。因此，在高斯噪声的假设下，最小化均方误差等价于对线性模型的最大似然估计。

2.1.4 从线性回归到深度网络

到目前为止，我们只谈论了线性模型。当神经网络涵盖了更多更为丰富的模型时，我们可以通过用神经网络的语言来表达线性模型，从而开始把线性模型看作一个神经网络。首先，让我们用"层"符号来重写这个模型。

神经网络图

深度学习从业者喜欢绘制图表来可视化模型中正在发生的事情。在 图2.1.2 中，我们将线性回归模型描述为一个神经网络。需要注意的是，该图只显示连接模式，即只显示每个输入如何连接到输出，隐去了权重和偏置的值。

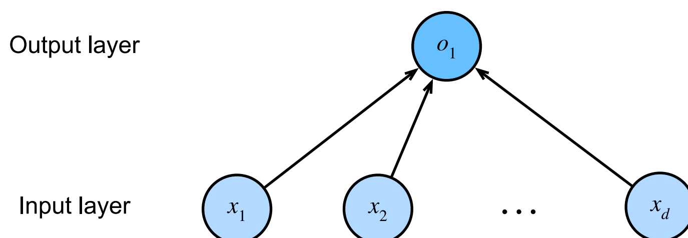


图2.1.2: 线性回归是一个单层神经网络。

在 图2.1.2 所示的神经网络中，输入为 $x_1, \ldots, x_d$，因此输入层中的 输入数（或称为 特征维度 feature dimensionality）为 d。网络的输出为o_1，因此输出层中的 输出数是 1。需要注意的是，输入值都是已经给定的，并且只有一个计算神经元。由于模型重点在发生计算的地方，所以通常我们在计算层数时不考虑输入层。也就是说，图2.1.2 中神经网络的 层数为1。我们可以将线性回归模型视为仅由单个人工神经元组成的神经网络，或称为单层神经网络。

对于线性回归，每个输入都与每个输出（在本例中只有一个输出）相连，我们将这种变换（图2.1.2 中的输出层）称为 全连接层（fully-connected layer）（或称为 稠密层 dense layer）。下一章将详细讨论由这些层组成的网络。

生物学

线性回归发明的时间（1795年）早于计算神经科学，所以将线性回归描述为神经网络似乎不合适。当控制学家、神经生物学家沃伦·麦库洛奇和沃尔特·皮茨开始开发人工神经元模型时，他们为什么将线性模型作为一个起点呢？我们来看一张图片 图2.1.3，这是一张由树突（dendrites，输入终端）、细胞核（nucleus，CPU）组成的生物神经元图片。轴突（axon，输出线）和轴突端子（axon terminals，输出端子）通过突触（synapses）与其他神经元连接。

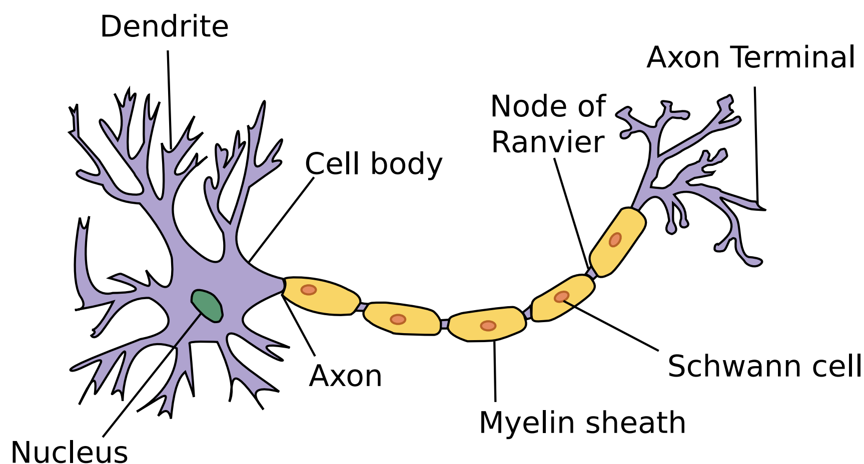


图2.1.3: 真实的神经元。

树突中接收到来自其他神经元（或视网膜等环境传感器）的信息x_i。该信息通过突触权重 w_i来加权，以确定输入的影响（即，通过$x_i w_i$相乘来激活或抑制）。来自多个源的加权输入以加权和$y = \sum_i x_i w_i + b$的形式汇聚在细胞核中，然后将这些信息发送到轴突 y 中进一步处理，通常会通过 $\sigma(y)$ 进行一些非线性处理。之后，它要么到达目的地（例如肌肉），要么通过树突进入另一个神经元。

当然，许多这样的单元可以通过正确连接和正确的学习算法拼凑在一起，从而产生的行为会比单独一个神经元所产生的行为更有趣、更复杂，这种想法归功于我们对真实生物神经系统的研究。

当今大多数深度学习的研究几乎没有直接从神经科学中获得灵感。我们援引斯图尔特·罗素和彼得·诺维格谁，在他们的经典人工智能教科书 *Artificial Intelligence: A Modern Approach* [Russell & Norvig, 2016], 中所说：虽然飞机可能受到鸟类的启发。但几个世纪以来，鸟类学并不是航空创新的主要驱动力。同样地，如今在深度学习中的灵感同样或更多地来自数学、统计学和计算机科学。

2.1.5 小结

- 机器学习模型中的关键要素是训练数据，损失函数，优化算法，还有模型本身。
- 矢量化使数学表达上更简洁，同时运行的更快。
- 最小化目标函数和执行最大似然估计等价。
- 线性回归模型也是神经网络。

2.1.6 练习

1. 假设我们有一些数据 $x_1, \ldots, x_n \in \mathbb{R}$。我们的目标是找到一个常数$b$，使得最小化 $\sum_i (x_i - b)^2$。
    1. 找到最优值 b 的解析解。
    2. 这个问题及其解与正态分布有什么关系?
2. 推导出使用平方误差的线性回归优化问题的解析解。为了简化问题，可以忽略偏置b(我们可以通过向 $\mathbf{X}$ 添加所有值为1的一列来做到这一点)。
    1. 用矩阵和向量表示法写出优化问题(将所有数据视为单个矩阵，将所有目标值视为单个向量)。
    2. 计算损失对w的梯度。
    3. 通过将梯度设为0、求解矩阵方程来找到解析解。
    4. 什么时候可能比使用随机梯度下降更好？这种方法何时会失效？
3. 假定控制附加噪声 ϵ 的噪声模型是指数分布。也就是说，$p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$
    1. 写出模型 $-\log P(\mathbf{y} \mid \mathbf{X})$ 下数据的负对数似然。
    2. 你能写出解析解吗?
    3. 提出一种随机梯度下降算法来解决这个问题。哪里可能出错？（提示：当我们不断更新参数时，在驻点附近会发生什么情况）你能解决这个问题吗?

Discussions[40]

2.2 线性回归的从零开始实现

在了解线性回归的关键思想之后，我们可以开始通过代码来动手实现线性回归了。在这一节中，我们将从零开始实现整个方法，包括数据流水线、模型、损失函数和小批量随机梯度下降优化器。虽然现代的深度学习框架几乎可以自动化地进行所有这些工作，但从零开始实现可以确保你真正知道自己在做什么。同时，了解更细致的工作原理将方便我们自定义模型、自定义层或自定义损失函数。在这一节中，我们将只使用张量和自动求导。在之后的章节中，我们会充分利用深度学习框架的优势，介绍更简洁的实现方式。

[40] https://discuss.d2l.ai/t/1774

```python
%matplotlib inline
import random
from mxnet import autograd, np, npx
from d2l import mxnet as d2l

npx.set_np()
```

2.2.1 生成数据集

为了简单起见，我们将根据带有噪声的线性模型构造一个人造数据集。我们的任务是使用这个有限样本的数据集来恢复这个模型的参数。我们将使用低维数据，这样可以很容易地将其可视化。在下面的代码中，我们生成一个包含1000个样本的数据集，每个样本包含从标准正态分布中采样的2个特征。我们的合成数据集是一个矩阵 $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$。

我们使用线性模型参数$\mathbf{w} = [2, -3.4]^\top$、$b = 4.2$和噪声项$\epsilon$生成数据集及其标签：

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon. \tag{2.2.1}$$

你可以将 ϵ 视为捕获特征和标签时的潜在观测误差。在这里我们认为标准假设成立，即ϵ服从均值为0的正态分布。为了简化问题，我们将标准差设为0.01。下面的代码生成合成数据集。

```python
def synthetic_data(w, b, num_examples):  #@save
    """生成 y = Xw + b + 噪声。"""
    X = np.random.normal(0, 1, (num_examples, len(w)))
    y = np.dot(X, w) + b
    y += np.random.normal(0, 0.01, y.shape)
    return X, y.reshape((-1, 1))
```

```python
true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

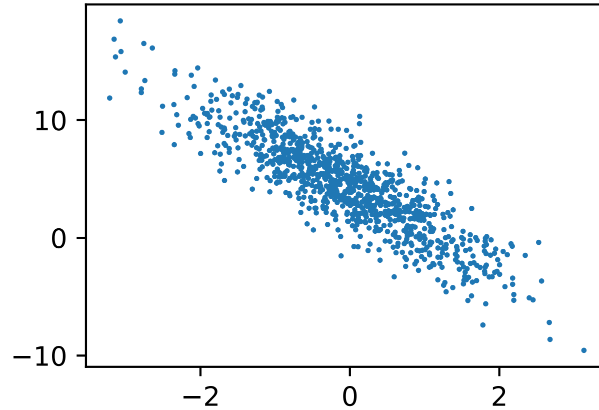
注意，`features` 中的每一行都包含一个二维数据样本，`labels` 中的每一行都包含一维标签值（一个标量）。

```python
print('features:', features[0], '\nlabel:', labels[0])
```

```
features: [2.2122064 1.1630787]
label: [4.662078]
```

通过生成第二个特征 `features[:, 1]` 和 `labels` 的散点图，可以直观地观察到两者之间的线性关系。

```python
d2l.set_figsize()
d2l.plt.scatter(features[:, (1)].asnumpy(), labels.asnumpy(), 1);
```



2.2.2 读取数据集

回想一下，训练模型时要对数据集进行遍历，每次抽取一小批量样本，并使用它们来更新我们的模型。由于这个过程是训练机器学习算法的基础，所以有必要定义一个函数，该函数能打乱数据集中的样本并以小批量方式获取数据。

在下面的代码中，我们定义一个data_iter 函数实现这一功能。该函数接收批量大小、特征矩阵和标签向量作为输入，生成大小为batch_size的小批量。每个小批量包含一组特征和标签。

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # 这些样本是随机读取的，没有特定的顺序
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = np.array(indices[i:min(i + batch_size, num_examples)])
        yield features[batch_indices], labels[batch_indices]
```

通常，我们使用合理大小的小批量来利用GPU硬件的优势，因为GPU在并行处理方面表现出色。每个样本都可以并行地进行模型计算，且每个样本损失函数的梯度也可以被并行地计算，GPU可以在处理几百个样本时，所花费的时间不比处理一个样本时多太多。

让我们直观感受一下。读取第一个小批量数据样本并打印。每个批量的特征维度说明了批量大小和输入特征数。同样的，批量的标签形状与 batch_size 相等。

```
batch_size = 10

for X, y in data_iter(batch_size, features, labels):
    print(X, '\n', y)
    break
```

```
[[-0.7774003   2.193714  ]
 [-0.2535011  -0.9285665 ]
 [-1.4548141  -0.02985478]
 [-1.0636146  -1.6143361 ]
 [-0.08507421 -0.157554  ]
 [ 0.00955429 -0.35396427]
 [-2.2268252  -1.7408477 ]
 [-0.37872985 -0.44169798]
 [-0.40910295  0.93317765]
 [ 0.77901214 -0.84304506]]
 [[-4.8152013 ]
 [ 6.8496747 ]
 [ 1.4094979 ]
 [ 7.5766845 ]
 [ 4.5783463 ]
 [ 5.4105372 ]
 [ 5.67558   ]
 [ 4.9423246 ]
 [ 0.22510855]
 [ 8.63009   ]]
```

当我们运行迭代时，我们会连续地获得不同的小批量，直至遍历完整个数据集。上面实现的迭代对于教学来说很好，但它的执行效率很低，可能会在实际问题上陷入麻烦。例如，它要求我们将所有数据加载到内存中，并执行大量的随机内存访问。在深度学习框架中实现的内置迭代器效率要高得多，它可以处理存储在文件中的数据和通过数据流提供的数据。

2.2.3 初始化模型参数

在我们开始用小批量随机梯度下降优化我们的模型参数之前，我们需要先有一些参数。在下面的代码中，我们通过从均值为0、标准差为0.01的正态分布中采样随机数来初始化权重，并将偏置初始化为0。

```
w = np.random.normal(0, 0.01, (2, 1))
b = np.zeros(1)
w.attach_grad()
b.attach_grad()
```

在初始化参数之后，我们的任务是更新这些参数，直到这些参数足够拟合我们的数据。每次更新都需要计算损失函数关于模型参数的梯度。有了这个梯度，我们就可以向减小损失的方向更新每个参数。因为手动计算梯度很枯燥而且容易出错，所以没有人会手动计算梯度。我们使用 1.5节 中引入的自动微分来计算梯度。

2.2.4 定义模型

接下来，我们必须定义模型，将模型的输入和参数同模型的输出关联起来。回想一下，要计算线性模型的输出，我们只需计算输入特征 $\mathbf{X}$ 和模型权重$\mathbf{w}$的矩阵-向量乘法后加上偏置b。注意，上面的$\mathbf{Xw}$ 是一个向量，而b是一个标量。回想一下 1.1.3节 中描述的广播机制。当我们用一个向量加一个标量时，标量会被加到向量的每个分量上。

```
def linreg(X, w, b):  #@save
    """线性回归模型。"""
    return np.dot(X, w) + b
```

2.2.5 定义损失函数

因为要更新模型。需要计算损失函数的梯度，所以我们应该先定义损失函数。这里我们使用 2.1节 中描述的平方损失函数。在实现中，我们需要将真实值`y`的形状转换为和预测值`y_hat`的形状相同。

```
def squared_loss(y_hat, y):  #@save
    """均方损失。"""
    return (y_hat - y.reshape(y_hat.shape))**2 / 2
```

2.2.6 定义优化算法

正如我们在 2.1节 中讨论的，线性回归有解析解。然而，这是一本关于深度学习的书，而不是一本关于线性回归的书。由于这本书介绍的其他模型都没有解析解，下面我们将在这里介绍小批量随机梯度下降的工作示例。

在每一步中，使用从数据集中随机抽取的一个小批量，然后根据参数计算损失的梯度。接下来，朝着减少损失的方向更新我们的参数。下面的函数实现小批量随机梯度下降更新。该函数接受模型参数集合、学习速率和批量大小作为输入。每一步更新的大小由学习速率`lr`决定。因为我们计算的损失是一个批量样本的总和，所以我们用批量大小（`batch_size`）来归一化步长，这样步长大小就不会取决于我们对批量大小的选择。

```
def sgd(params, lr, batch_size):  #@save
    """小批量随机梯度下降。"""
    for param in params:
        param[:] = param - lr * param.grad / batch_size
```

2.2.7 训练

现在我们已经准备好了模型训练所有需要的要素，可以实现主要的训练过程部分了。理解这段代码至关重要，因为在整个深度学习的职业生涯中，你会看到一遍又一遍几乎相同的训练过程。在每次迭代中，我们读取一小批量训练样本，并通过我们的模型来获得一组预测。计算完损失后，我们开始反向传播，存储每个参数的梯度。最后，我们调用优化算法 sgd 来更新模型参数。

概括一下，我们将执行以下循环：

- 初始化参数
- 重复，直到完成
  - 计算梯度 $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$
  - 更新参数 $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

在每个迭代周期（epoch）中，我们使用 data_iter 函数遍历整个数据集，并将训练数据集中所有样本都使用一次（假设样本数能够被批量大小整除）。这里的迭代周期个数num_epochs和学习率lr都是超参数，分别设为3和0.03。设置超参数很棘手，需要通过反复试验进行调整。我们现在忽略这些细节，以后会在 chap_optimization 中详细介绍。

```
lr = 0.03
num_epochs = 3
net = linreg
loss = squared_loss
```

```
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        with autograd.record():
            l = loss(net(X, w, b), y)  # `X`和`y`的小批量损失
        # 计算l关于[`w`, `b`]的梯度
        l.backward()
        sgd([w, b], lr, batch_size)  # 使用参数的梯度更新参数
    train_l = loss(net(features, w, b), labels)
    print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')
```

```
epoch 1, loss 0.024930
epoch 2, loss 0.000091
epoch 3, loss 0.000051
```

因为我们使用的是自己合成的数据集，所以我们知道真正的参数是什么。因此，我们可以通过比较真实参数和通过训练学到的参数来评估训练的成功程度。事实上，真实参数和通过训练学到的参数确实非常接近。

```
print(f'w的估计误差: {true_w - w.reshape(true_w.shape)}')
print(f'b的估计误差: {true_b - b}')
```

```
w的估计误差: [ 5.916357e-04 -4.696846e-05]
b的估计误差: [0.00030231]
```

注意，我们不应该想当然地认为我们能够完美地恢复参数。在机器学习中，我们通常不太关心恢复真正的参数，而更关心那些能高度准确预测的参数。幸运的是，即使是在复杂的优化问题上，随机梯度下降通常也能找到非常好的解。其中一个原因是，在深度网络中存在许多参数组合能够实现高度精确的预测。

2.2.8 小结

- 我们学习了深度网络是如何实现和优化的。在这一过程中只使用张量和自动微分，不需要定义层或复杂的优化器。
- 这一节只触及到了表面知识。在下面的部分中，我们将基于刚刚介绍的概念描述其他模型，并学习如何更简洁地实现其他模型。

2.2.9 练习

1. 如果我们将权重初始化为零，会发生什么。算法仍然有效吗？
2. 假设你是 乔治·西蒙·欧姆[41]，试图为电压和电流的关系建立一个模型。你能使用自动微分来学习模型的参数吗?
3. 您能基于 普朗克定律[42] 使用光谱能量密度来确定物体的温度吗？
4. 如果你想计算二阶导数可能会遇到什么问题？你会如何解决这些问题？
5. 为什么在 `squared_loss` 函数中需要使用 `reshape` 函数？
6. 尝试使用不同的学习率，观察损失函数值下降的快慢。
7. 如果样本个数不能被批量大小整除，`data_iter`函数的行为会有什么变化？

Discussions[43]

2.3 线性回归的简洁实现

在过去的几年里，出于对深度学习强烈的兴趣，许多公司、学者和业余爱好者开发了各种成熟的开源框架。通过这些框架可以自动化实现基于梯度的学习算法中重复性的工作。在 2.2节 中，我们只依赖了：（1）通过张量来进行数据存储和线性代数；（2）通过自动微分来计算梯度。实际上，由于数据迭代器、损失函数、优化器和神经网络层很常用，现代深度学习库也为我们实现了这些组件。

在本节中，我们将介绍如何通过使用深度学习框架的高级API来简洁地实现 2.2节 中的线性回归模型。

[41] https://en.wikipedia.org/wiki/Georg_Ohm
[42] https://en.wikipedia.org/wiki/Planck%27s_law
[43] https://discuss.d2l.ai/t/1779

2.3.1 生成数据集

首先，我们生成与2.2节中相同的数据集。

```
from mxnet import autograd, gluon, np, npx
from d2l import mxnet as d2l

npx.set_np()
```

```
true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1000)
```

2.3.2 读取数据集

我们可以调用框架中现有的API来读取数据，而不使用我们自己定义的迭代器。我们将 features 和 labels 作为API的参数传递，并在实例化数据迭代器对象时指定 batch_size。此外，布尔值 is_train 表示是否希望数据迭代器对象在每个迭代周期内打乱数据。

```
def load_array(data_arrays, batch_size, is_train=True):  #@save
    """构造一个Gluon数据迭代器。"""
    dataset = gluon.data.ArrayDataset(*data_arrays)
    return gluon.data.DataLoader(dataset, batch_size, shuffle=is_train)
```

```
batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

使用 data_iter 的方式与我们在2.2节中使用 data_iter 函数的方式相同。为了验证是否正常工作，让我们读取并打印第一个小批量样本。与2.2节不同，这里我们使用 iter 构造Python迭代器，并使用 next 从迭代器中获取第一项。

```
next(iter(data_iter))
```

```
[array([[-0.0643351 ,  1.6473899 ],
        [ 0.8486986 ,  0.04734707],
        [ 1.4824563 , -2.6438675 ],
        [-0.35430676, -0.87738055],
        [ 0.40236598,  0.5925345 ],
        [-1.1642394 ,  0.1445754 ],
        [ 0.6397144 , -0.90252906],
        [-0.19628292,  0.32660788],
        [ 1.9984016 ,  0.30198845],
```

(continues on next page)

(continued from previous page)

```
        [ 0.7615899 ,  1.6106696 ]]),
 array([[-1.5441506],
        [ 5.7442694],
        [16.123064 ],
        [ 6.4599323],
        [ 3.0004385],
        [ 1.3729192],
        [ 8.548329 ],
        [ 2.6886718],
        [ 7.185257 ],
        [ 0.2617909]])]
```

2.3.3 定义模型

当我们在 2.2节 中实现线性回归时，我们明确定义了模型参数变量，并编写了计算的代码，这样通过基本的线性代数运算得到输出。但是，如果模型变得更加复杂，而且当你几乎每天都需要实现模型时，你会想简化这个过程。这种情况类似于从头开始编写自己的博客。做一两次是有益的、有启发性的，但如果每次你每需要一个博客就花一个月的时间重新发明轮子，那你将是一个糟糕的网页开发者。

对于标准操作，我们可以使用框架的预定义好的层。这使我们只需关注使用哪些层来构造模型，而不必关注层的实现细节。我们首先定义一个模型变量net，它是一个 Sequential 类的实例。Sequential 类为串联在一起的多个层定义了一个容器。当给定输入数据，Sequential 实例将数据传入到第一层，然后将第一层的输出作为第二层的输入，依此类推。在下面的例子中，我们的模型只包含一个层，因此实际上不需要Sequential。但是由于以后几乎所有的模型都是多层的，在这里使用Sequential会让你熟悉标准的流水线。

回顾 图2.1.2 中的单层网络架构，这一单层被称为 全连接层（fully-connected layer），因为它的每一个输入都通过矩阵-向量乘法连接到它的每个输出。

在 Gluon 中，全连接层在 Dense 类中定义。由于我们只想得到一个标量输出，所以我们将该数字设置为 1。

值得注意的是，为了方便使用，Gluon 并不要求我们为每个层指定输入的形状。所以在这里，我们不需要告诉 Gluon 有多少输入进入这一层。当我们第一次尝试通过我们的模型传递数据时，例如，当后面执行 net(X) 时，Gluon 会自动推断每个层输入的形状。我们稍后将详细介绍这种工作机制。

```
# `nn` 是神经网络的缩写
from mxnet.gluon import nn

net = nn.Sequential()
net.add(nn.Dense(1))
```

2.3.4 初始化模型参数

在使用net之前，我们需要初始化模型参数。如在线性回归模型中的权重和偏置。深度学习框架通常有预定义的方法来初始化参数。在这里，我们指定每个权重参数应该从均值为0、标准差为0.01的正态分布中随机采样，偏置参数将初始化为零。

我们从 MXNet 导入 initializer 模块。这个模块提供了各种模型参数初始化方法。Gluon将 init 作为访问 initializer 包的快捷方式。我们可以通过调用 init.Normal(sigma=0.01) 来指定初始化权重的方法。默认情况下，偏置参数初始化为零。

```
from mxnet import init

net.initialize(init.Normal(sigma=0.01))
```

上面的代码可能看起来很简单，但是你应该注意到这里的一个细节：我们正在为网络初始化参数，而Gluon还不知道输入将有多少维! 网络的输入可能有2维，也可能有2000维。Gluon让我们避免了这个问题，在后端执行时，初始化实际上是 推迟（deferred）执行的。只有在我们第一次尝试通过网络传递数据时才会进行真正的初始化。只是要记住，因为参数还没有初始化，所以我们不能访问或操作它们。

2.3.5 定义损失函数

在 Gluon 中，loss 模块定义了各种损失函数。在这个例子中，我们将使用Gluon中的平方损失 (L2Loss)。

```
loss = gluon.loss.L2Loss()
```

2.3.6 定义优化算法

小批量随机梯度下降算法是一种优化神经网络的标准工具，Gluon 通过 Trainer 类支持该算法的许多变种。当我们实例化 Trainer 时，我们要指定优化的参数（可通过 net.collect_params() 从我们的模型 net 中获得)、我们希望使用的优化算法（sgd）以及优化算法所需的超参数字典。小批量随机梯度下降只需要设置 learning_rate值，这里设置为 0.03。

```
from mxnet import gluon

trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.03})
```

2.3.7 训练

通过深度学习框架的高级API来实现我们的模型只需要相对较少的代码。我们不必单独分配参数、不必定义我们的损失函数，也不必手动实现小批量随机梯度下降。当我们需要更复杂的模型时，高级API的优势将大大增加。当我们有了所有的基本组件，训练过程代码与我们从零开始实现所有东西时所做的非常相似。

回顾一下：在每个迭代周期里，我们将完整遍历一次数据集（`train_data`），不停地从中获取一个小批量的输入和相应的标签。对于每一个小批量，我们会进行以下步骤:

- 通过调用 `net(X)` 生成预测并计算损失 `l`（正向传播）。
- 通过进行反向传播来计算梯度。
- 通过调用优化器来更新模型参数。

为了更好的衡量训练效果，我们计算每个迭代周期后的损失，并打印它来监控训练过程。

```
num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
        with autograd.record():
            l = loss(net(X), y)
        l.backward()
        trainer.step(batch_size)
    l = loss(net(features), labels)
    print(f'epoch {epoch + 1}, loss {l.mean().asnumpy():f}')
```

```
epoch 1, loss 0.025140
epoch 2, loss 0.000089
epoch 3, loss 0.000051
```

下面我们比较生成数据集的真实参数和通过有限数据训练获得的模型参数。要访问参数，我们首先从 `net` 访问所需的层，然后读取该层的权重和偏置。正如在从零开始实现中一样，我们估计得到的参数与生成数据的真实参数非常接近。

```
w = net[0].weight.data()
print(f'w的估计误差： {true_w - w.reshape(true_w.shape)}')
b = net[0].bias.data()
print(f'b的估计误差： {true_b - b}')
```

```
w的估计误差： [ 0.00055146 -0.0003221 ]
b的估计误差： [0.00057793]
```

2.3.8 小结

- 我们可以使用Gluon更简洁地实现模型。
- 在 Gluon 中，`data` 模块提供了数据处理工具，`nn` 模块定义了大量的神经网络层，`loss` 模块定义了许多常见的损失函数。
- MXNet 的 `initializer` 模块提供了各种模型参数初始化方法。
- 维度和存储可以自动推断，但注意不要在初始化参数之前尝试访问参数。

2.3.9 练习

1. 如果我们用 `l = loss(output, y).mean()` 替换 `l = loss(output, y)`。为了使代码的行为相同，需要将 `trainer.step(batch_size)` 更改为 `trainer.step(1)`，这是为什么？
2. 查看 MXNet 文档，了解模块 `gluon.loss` 和 `init` 中提供了哪些损失函数和初始化方法。用Huber损失来代替。
3. 你如何访问 `dense.weight` 的梯度？

Discussions[44]

2.4 softmax回归

在 2.1节 中我们介绍了线性回归。随后，在 2.2节 中我们从头实现了线性回归。然后在 2.3节 中我们使用深度学习框架的高级API来完成繁重的工作。

回归可以用于预测 多少的问题。比如预测房屋被售出价格，或者棒球队可能获得的胜利数，又或者患者住院的天数。

事实上，我们经常对 分类感兴趣：不是问“多少”，而是问“哪一个”：

- 该电子邮件是否属于垃圾邮件文件夹？
- 该用户可能 注册或 不注册订阅服务？
- 该图像描绘的是驴、狗、猫、还是鸡？
- 韩梅梅接下来最有可能看哪部电影？

通常，机器学习实践者用分类这个词来描述两个有微妙差别的问题：（1）我们只对样本的硬性类别感兴趣，即属于哪个类别；（2）我们希望得到软性类别，即得到属于每个类别的概率。这两者的界限往往很模糊。其中的一个原因是，即使我们只关心硬任务，我们仍然使用软任务的模型。

[44] https://discuss.d2l.ai/t/1782

2.4.1 分类问题

让我们从一个图像分类问题开始简单尝试一下。每次输入是一个 2×2 的灰度图像。我们可以用一个标量表示每个像素值，每个图像对应四个特征 x_1, x_2, x_3, x_4。此外，让我们假设每个图像属于类别“猫”，“鸡”和“狗”中的一个。

接下来，我们要选择如何表示标签。我们有两个明显的选择。也许最直接的想法是选择 $y \in \{1, 2, 3\}$，其中整数分别代表 {狗, 猫, 鸡}。这是在计算机上存储此类信息的好方法。如果类别间有一些自然的排序，比如说我们试图预测 {婴儿, 儿童, 青少年, 青年人, 中年人, 老年人}，那么将这个问题转变为回归问题并保留这种格式是有意义的。

但是，一般的分类问题并不与类别之间的自然排序有关。幸运的是，统计学家很早以前就发明了一种表示分类数据的简单方法：独热编码（one-hot encoding）。独热编码是一个向量，它的分量和类别一样多。类别对应的分量设置为1，其他所有分量设置为0。在我们的例子中，标签 y 将是一个三维向量，其中 $(1, 0, 0)$ 对应于“猫”、$(0, 1, 0)$ 对应于“鸡”、$(0, 0, 1)$ 对应于“狗”：

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}. \tag{2.4.1}$$

2.4.2 网络结构

为了估计所有可能类别的条件概率，我们需要一个有多个输出的模型，每个类别对应一个输出。为了解决线性模型的分类问题，我们需要和输出一样多的仿射函数（affine function）。每个输出对应于它自己的仿射函数。在我们的例子中，由于我们有4个特征和3个可能的输出类别，我们将需要12个标量来表示权重（带下标的w），3个标量来表示偏置（带下标的b）。下面我们为每个输入计算三个未归一化的预测（logits）：o_1、o_2和o_3。

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1,\\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2,\\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned} \tag{2.4.2}$$

我们可以用神经网络图 图2.4.1 来描述这个计算过程。与线性回归一样，softmax回归也是一个单层神经网络。由于计算每个输出o_1、o_2和o_3取决于所有输入x_1、x_2、x_3和x_4，所以softmax回归的输出层也是全连接层。

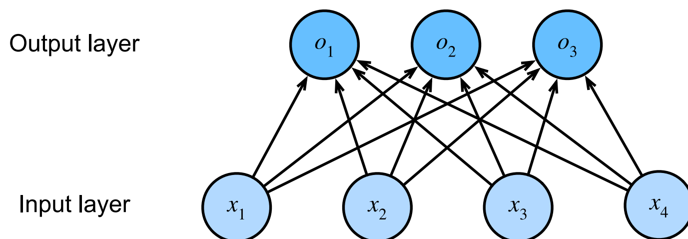


图2.4.1: softmax回归是一种单层神经网络。

为了更简洁地表达模型，我们仍然使用线性代数符号。通过向量形式表达为 $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$，这是一种更适合数学和编写代码的形式。我们已经将所有权重放到一个 3×4 矩阵中。对于给定数据样本的特征 $\mathbf{x}$，我们的输出是由权重与输入特征进行矩阵-向量乘法加上偏置$\mathbf{b}$得到的。

2.4.3 全连接层的参数开销

正如我们将在后续章节中看到的，在深度学习中，全连接层无处不在。然而，顾名思义，全连接层是“完全”连接的。这可能有很多可学习的参数。具体来说，对于任何具有d个输入和q个输出的全连接层，参数开销为$\mathcal{O}(dq)$，在实践中可能高得令人望而却步。幸运的是，将d个输入转换为q个输出的成本可以减少到$\mathcal{O}(\frac{dq}{n})$，其中超参数$n$可以由我们灵活指定，以在实际应用中平衡节省参数和模型有效性 [Zhang et al., 2021] 。

2.4.4 softmax操作

在这里要采取的主要方法是将模型的输出视作为概率。我们将优化参数以最大化观测数据的概率。为了得到预测结果，我们将设置一个阈值，如选择具有最大概率的标签。

我们希望模型的输出 $\hat{y}_j$ 可以视为属于类 j 的概率。然后我们可以选择具有最大输出值的类别$\operatorname{argmax}_j y_j$作为我们的预测。例如，如果 $\hat{y}_1$、$\hat{y}_2$ 和 $\hat{y}_3$ 分别为 0.1、0.8 和 0.1，那么我们预测的类别是2，在我们的例子中代表“鸡”。

你可能会想是否可以将未归一化的预测 o 直接视作我们感兴趣的输出。但是，将线性层的输出直接视为概率时存在一些问题：一方面，没有限制这些数字的总和为1。另一方面，根据输入的不同，它们可以为负值。这些违反了 1.6节 中所说的概率基本公理。

要将输出视为概率，我们必须保证在任何数据上的输出都是非负的且总和为1。此外，我们需要一个训练目标，来鼓励模型估计概率。在分类器输出0.5的所有样本中，我们希望这些样本有一半实际上属于预测的类。这个属性叫做校准（calibration）。

社会科学家邓肯·卢斯于1959年在选择模型（choice models）的背景下发明的*softmax*函数正是这样做的。为了将未归一化的预测变换为非负并且总和为1，同时要求模型保持可导。我们首先对每个未归一化的预测求幂，这样可以确保输出非负数。为了确保最终输出的总和为1，我们再对每个求幂后的结果除以它们的总和。如下式：

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{其中} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)} \tag{2.4.3}$$

容易看出对于所有的 j 总有 $0 \leq \hat{y}_j \leq 1$。因此，$\hat{\mathbf{y}}$ 可以视为一个正确的概率分布。softmax 运算不会改变未归一化的预测 $\mathbf{o}$ 之间的顺序，只会确定分配给每个类别的概率。因此，在预测过程中，我们仍然可以用下式来选择最有可能的类别。

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j. \tag{2.4.4}$$

尽管softmax是一个非线性函数，但softmax回归的输出仍然由输入特征的仿射变换决定。因此，softmax回归是一个线性模型。

2.4.5 小批量样本的矢量化

为了提高计算效率并且充分利用GPU，我们通常会针对小批量数据执行矢量计算。假设我们读取了一个批量的样本 $\mathbf{X}$，其中特征维度（输入数量）为d，批量大小为n。此外，假设我们在输出中有 q 个类别。设小批量特征为 $\mathbf{X} \in \mathbb{R}^{n \times d}$，权重为 $\mathbf{W} \in \mathbb{R}^{d \times q}$，偏置为 $\mathbf{b} \in \mathbb{R}^{1 \times q}$。softmax回归的矢量计算表达式为：

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \mathrm{softmax}(\mathbf{O}). \end{aligned} \tag{2.4.5}$$

相对于一次处理一个样本，小批量样本的矢量化加快了 $\mathbf{X}\mathbf{W}$ 的矩阵-向量乘法运算。由于$\mathbf{X}$ 中的每一行代表一个数据样本，且softmax操作本身可以按行（rowwise）执行。所以，对于$\mathbf{O}$的每一行，我们首先对所有项进行幂运算，然后通过求和对它们进行标准化。在 (2.4.5) 中$\mathbf{X}\mathbf{W} + \mathbf{b}$的求和时会使用广播，小批量的未归一化预测 $\mathbf{O}$ 和输出概率 $\hat{\mathbf{Y}}$ 都是形状为 $n \times q$ 的矩阵。

2.4.6 损失函数

接下来，我们需要一个损失函数来度量预测概率的效果。我们将依赖最大似然估计，这与我们在为线性回归（2.1.3节 ）中的均方误差目标提供概率证明时遇到的概念完全相同。

对数似然

softmax函数给出了一个向量 $\hat{\mathbf{y}}$，我们可以将其视为给定任意输入 $\mathbf{x}$的每个类的估计条件概率。例如，$\hat{y}_1 = P(y = \text{猫} \mid \mathbf{x})$。假设整个数据集 $\{\mathbf{X}, \mathbf{Y}\}$ 具有 n 个样本，其中索引 i 的样本由特征向量 $\mathbf{x}^{(i)}$ 和独热标签向量 $\mathbf{y}^{(i)}$ 组成。我们可以将估计值与实际值进行比较：

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}). \tag{2.4.6}$$

根据最大似然估计，我们最大化 $P(\mathbf{Y} \mid \mathbf{X})$，相当于最小化负对数似然：

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \tag{2.4.7}$$

其中，对于任何标签 $\mathbf{y}$ 和模型预测 $\hat{\mathbf{y}}$，损失函数为：

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j. \tag{2.4.8}$$

在本节稍后的内容会讲到，(2.4.8) 中的损失函数通常被称为 交叉熵损失（cross-entropy loss）。由于 $\mathbf{y}$ 是一个长度为 q 的独热编码向量，所以除了一个项以外的所有项 j 都消失了。由于所有 $\hat{y}_j$ 都是预测的概率，所以它们的对数永远不会大于 0。因此，如果正确地预测实际标签，即，如果实际标签 $P(\mathbf{y} \mid \mathbf{x}) = 1$，则损失函数不能进一步最小化。注意，这往往是不可能的。例如，数据集中可能存在标签噪声（某些样本可能被误标），或输入特征没有足够的信息来完美地对每一个样本分类。

softmax及其导数

由于softmax和相关的损失函数很常见，因此值得更好地理解它的计算方式。将 (2.4.3) 代入损失 (2.4.8) 中。利用softmax的定义，我们得到：

$$\begin{aligned}
l(\mathbf{y}, \hat{\mathbf{y}}) &= - \sum_{j=1}^q y_j \log \frac{\exp(o_j)}{\sum_{k=1}^q \exp(o_k)} \\
&= \sum_{j=1}^q y_j \log \sum_{k=1}^q \exp(o_k) - \sum_{j=1}^q y_j o_j \\
&= \log \sum_{k=1}^q \exp(o_k) - \sum_{j=1}^q y_j o_j.
\end{aligned} \tag{2.4.9}$$

为了更好地理解发生了什么，考虑相对于任何未归一化的预测 o_j 的导数。我们得到：

$$\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(o_j)}{\sum_{k=1}^q \exp(o_k)} - y_j = \mathrm{softmax}(\mathbf{o})_j - y_j. \tag{2.4.10}$$

换句话说，导数是我们模型分配的概率（由softmax得到）与实际发生的情况（由独热标签向量表示）之间的差异。从这个意义上讲，与我们在回归中看到的非常相似，其中梯度是观测值y和估计值$\hat{y}$之间的差异。这不是巧合，在任何指数族分布（参见 关于分布的在线附录[45]）模型中，对数似然的梯度正是由这给出的。这使计算梯度在实践中变得容易。

交叉熵损失

现在考虑这样一个例子：我们观察到的不仅仅是一个结果，而是整个结果分布。对于标签 $\mathbf{y}$，我们可以使用与以前相同的表示形式。唯一的区别是，我们现在用一个概率向量表示，如$(0.1, 0.2, 0.7)$，而不是仅包含二元项的向量$(0, 0, 1)$。我们使用 (2.4.8) 来定义损失 l。它是所有标签分布的预期损失值。此损失称为 交叉熵损失（cross-entropy loss），它是分类问题最常用的损失之一。我们将通过介绍信息论的基础来理解这个名字。如果你想了解更多信息论细节，你可以进一步参考 信息论的在线附录[46]。

2.4.7 信息论基础

信息论涉及编码、解码、发送以及尽可能简洁地处理信息或数据。

熵

信息论的核心思想是量化数据中的信息内容，在信息论中，该数值被称为分布P 的 熵（entropy）。可以通过以下方程得到：

$$H[P] = \sum_j - P(j) \log P(j). \tag{2.4.11}$$

信息论的基本定理之一指出，为了对从分布 p 中随机抽取的数据进行编码，我们至少需要 $H[P]$ "纳特（nat）"对其进行编码。"纳特"相当于位，但是对数底为e而不是2。因此，一个纳特是 $\frac{1}{\log(2)} \approx 1.44$ 位。

[45] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/distributions.html
[46] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/information-theory.html

惊讶

你可能想知道压缩与预测有什么关系。想象一下，我们有一个要压缩的数据流。如果我们总是很容易预测下一个数据，那么这个数据很容易压缩！举一个极端的例子，数据流中的每个数据总是采用相同的值。这是一个非常无聊的数据流！由于它们总是相同的，所以很容易被预测，所以我们为了传递数据流的内容不必传输任何信息。当数据易于预测，也就易于压缩。

但是，如果我们不能完全预测每一个事件，那么我们有时可能会感到惊异。当我们赋予一个事件较低的概率时，我们的惊异会更大。克劳德·香农决定用 $\log \frac{1}{P(j)} = -\log P(j)$来量化一个人的 惊异（surprisal）。在观察一个事件 j，并赋予它（主观）概率 $P(j)$。在 (2.4.11) 中定义的熵是当分配的概率真正匹配数据生成过程时的预期惊异（expected surprisal）。

重新审视交叉熵

所以，如果熵是知道真实概率的人所经历的惊异程度，那么你可能会想知道，什么是交叉熵？交叉熵 从 P 到 Q，记为 $H(P, Q)$，是主观概率为Q的观察者在看到根据概率P实际生成的数据时的预期惊异。当$P = Q$时，交叉熵达到最低。在这种情况下，从 P到Q 的交叉熵是 $H(P, P) = H(P)$。

简而言之，我们可以从两方面来考虑交叉熵分类目标：（i）最大化观测数据的似然；（ii）尽量减少我们的惊异所需的通讯量。

2.4.8 模型预测和评估

在训练softmax回归模型后，给出任何样本特征，我们可以预测每个输出类别的概率。通常我们使用预测概率最高的类别作为输出类别。如果预测与实际类别（标签）一致，则预测是正确的。在接下来的实验中，我们将使用 准确率来评估模型的性能。准确率等于正确预测数与预测的总数之间的比率。

2.4.9 小结

- softmax运算获取一个向量并将其映射为概率。
- softmax回归适用于分类问题。它使用了softmax运算中输出类别的概率分布。
- 交叉熵是一个衡量两个概率分布之间差异的很好的度量。它测量给定模型编码数据所需的比特数。

2.4.10 练习

1. 我们可以更深入地探讨指数族与 softmax 之间的联系。
    1. 计算softmax交叉熵损失 $l(\mathbf{y}, \hat{\mathbf{y}})$ 的二阶导数。
    2. 计算 softmax$(\mathbf{o})$ 给出的分布方差，并与上面计算的二阶导数匹配。
2. 假设我们有三个类发生的概率相等，即概率向量是 $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$。
    1. 如果我们尝试为它设计二进制代码，有什么问题?

2. 你能设计一个更好的代码吗？提示：如果我们尝试编码两个独立的观察结果会发生什么？如果我们联合编码 n 个观测值怎么办？

3. softmax是对上面介绍的映射的误用(但深度学习中的每个人都使用它)。真正的softmax被定义为 $\mathrm{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$。

   1. 证明 $\mathrm{RealSoftMax}(a, b) > \max(a, b)$。
   2. 证明 $\lambda^{-1}\mathrm{RealSoftMax}(\lambda a, \lambda b)$成立，前提是 $\lambda > 0$。
   3. 证明对于 $\lambda \to \infty$ ，有 $\lambda^{-1}\mathrm{RealSoftMax}(\lambda a, \lambda b) \to \max(a, b)$。
   4. soft-min会是什么样子？
   5. 将其扩展到两个以上的数字。

Discussions[47]

2.5 图像分类数据集

目前广泛使用的图像分类数据集之一是 MNIST 数据集 [LeCun et al., 1998]。虽然它是很不错的基准数据集，但按今天的标准，即使是简单的模型也能达到95%以上的分类准确率，因此不适合区分强模型和弱模型。如今，MNIST更像是一个健全检查，而不是一个基准。为了提高难度，我们将在接下来的章节中讨论在2017年发布的性质相似但相对复杂的Fashion-MNIST数据集 [Xiao et al., 2017]。

```
%matplotlib inline
import sys
from mxnet import gluon
from d2l import mxnet as d2l

d2l.use_svg_display()
```

2.5.1 读取数据集

我们可以通过框架中的内置函数将 Fashion-MNIST 数据集下载并读取到内存中。

```
mnist_train = gluon.data.vision.FashionMNIST(train=True)
mnist_test = gluon.data.vision.FashionMNIST(train=False)
```

Fashion-MNIST 由 10 个类别的图像组成，每个类别由训练数据集中的 6000 张图像和测试数据集中的 1000 张图像组成。测试数据集（test dataset）不会用于训练，只用于评估模型性能。训练集和测试集分别包含 60000 和 10000 张图像。

[47] https://discuss.d2l.ai/t/1785

```
len(mnist_train), len(mnist_test)
```

```
(60000, 10000)
```

每个输入图像的高度和宽度均为 28 像素。数据集由灰度图像组成，其通道数为1。为了简洁起见，在这本书中，我们将高度h像素，宽度w像素图像的形状记为$h \times w$或(h, w)。

```
mnist_train[0][0].shape
```

```
(28, 28, 1)
```

Fashion-MNIST中包含的10个类别分别为t-shirt（T恤）、trouser（裤子）、pullover（套衫）、dress（连衣裙）、coat（外套）、sandal（凉鞋）、shirt（衬衫）、sneaker（运动鞋）、bag（包）和ankle boot（短靴）。以下函数用于在数字标签索引及其文本名称之间进行转换。

```
def get_fashion_mnist_labels(labels):  #@save
    """返回Fashion-MNIST数据集的文本标签。"""
    text_labels = [
        't-shirt', 'trouser', 'pullover', 'dress', 'coat', 'sandal', 'shirt',
        'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]
```

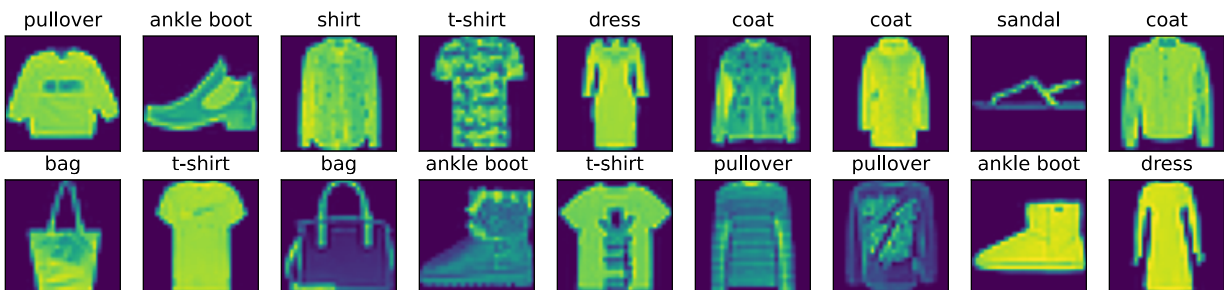
我们现在可以创建一个函数来可视化这些样本。

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """绘制图像列表。"""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        ax.imshow(img.asnumpy())
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes
```

以下是训练数据集中前几个样本的图像及其相应的标签（文本形式）。

```
X, y = mnist_train[:18]

print(X.shape)
show_images(X.squeeze(axis=-1), 2, 9, titles=get_fashion_mnist_labels(y));
```

```
(18, 28, 28, 1)
```



2.5.2 读取小批量

为了使我们在读取训练集和测试集时更容易，我们使用内置的数据迭代器，而不是从零开始创建一个。回顾一下，在每次迭代中，数据加载器每次都会读取一小批量数据，大小为batch_size。我们在训练数据迭代器中还随机打乱了所有样本。

```
batch_size = 256

def get_dataloader_workers():  #@save
    """在非Windows的平台上，使用4个进程来读取的数据。"""
    return 0 if sys.platform.startswith('win') else 4

# 通过ToTensor实例将图像数据从uint8格式变换成32位浮点数格式，并除以255使得所有像素
# 的数值均在0到1之间
transformer = gluon.data.vision.transforms.ToTensor()
train_iter = gluon.data.DataLoader(mnist_train.transform_first(transformer),
                                   batch_size, shuffle=True,
                                   num_workers=get_dataloader_workers())
```

让我们看一下读取训练数据所需的时间。

```
timer = d2l.Timer()
for X, y in train_iter:
    continue
f'{timer.stop():.2f} sec'
```

```
'2.42 sec'
```

2.5.3 整合所有组件

现在我们定义了 `load_data_fashion_mnist` 函数，用于获取和读取Fashion-MNIST数据集。它返回训练集和验证集的数据迭代器。此外，它还接受一个可选参数，用来将图像大小调整为另一种形状。

```python
def load_data_fashion_mnist(batch_size, resize=None):  #@save
    """下载Fashion-MNIST数据集，然后将其加载到内存中。"""
    dataset = gluon.data.vision
    trans = [dataset.transforms.ToTensor()]
    if resize:
        trans.insert(0, dataset.transforms.Resize(resize))
    trans = dataset.transforms.Compose(trans)
    mnist_train = dataset.FashionMNIST(train=True).transform_first(trans)
    mnist_test = dataset.FashionMNIST(train=False).transform_first(trans)
    return (gluon.data.DataLoader(mnist_train, batch_size, shuffle=True,
                                  num_workers=get_dataloader_workers()),
            gluon.data.DataLoader(mnist_test, batch_size, shuffle=False,
                                  num_workers=get_dataloader_workers()))
```

下面，我们通过指定`resize`参数来测试`load_data_fashion_mnist`函数的图像大小调整功能。

```python
train_iter, test_iter = load_data_fashion_mnist(32, resize=64)
for X, y in train_iter:
    print(X.shape, X.dtype, y.shape, y.dtype)
    break
```

```
(32, 1, 64, 64) <class 'numpy.float32'> (32,) <class 'numpy.int32'>
```

我们现在已经准备好在下面的章节中使用Fashion-MNIST数据集。

2.5.4 小结

- Fashion-MNIST是一个服装分类数据集，由10个类别的图像组成。我们将在后续章节中使用此数据集来评估各种分类算法。
- 我们将高度h像素，宽度w像素图像的形状记为$h \times w$或(h, w)。
- 数据迭代器是获得更高性能的关键组件。依靠实现良好的数据迭代器，利用高性能计算来避免减慢训练过程。

2.5.5 练习

1. 减少 `batch_size`（如减少到 1）是否会影响读取性能？
2. 数据迭代器的性能非常重要。你认为当前的实现足够快吗？探索各种选择来改进它。
3. 查阅框架的在线API文档。还有哪些其他数据集可用？

Discussions[48]

2.6 softmax回归的从零开始实现

就像我们从零开始实现线性回归一样，我们认为softmax回归也是重要的基础，因此你应该知道如何自己实现它的细节。我们使用刚刚在 2.5节 中引入的Fashion-MNIST数据集，并设置数据迭代器的批量大小为256。

```
from IPython import display
from mxnet import autograd, gluon, np, npx
from d2l import mxnet as d2l

npx.set_np()
```

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

2.6.1 初始化模型参数

这里的每个样本都用固定长度向量表示。原始数据集中的每个样本都是 28×28 的图像。在本节中，我们将展平每个图像，将它们视为长度为784的向量。在以后的章节中，我们将讨论能够利用图像空间结构的复杂策略，但现在我仅将每个像素位置视为一个特征。

回想一下，在softmax回归中，我们的输出与类别一样多。因为我们的数据集有10个类别，所以网络输出维度为 10。因此，权重将构成一个 784×10 的矩阵，偏置将构成一个 1×10 的行向量。与线性回归一样，我们将使用正态分布初始化我们的权重 `W`，偏置初始化为0。

```
num_inputs = 784
num_outputs = 10

W = np.random.normal(0, 0.01, (num_inputs, num_outputs))
b = np.zeros(num_outputs)
W.attach_grad()
b.attach_grad()
```

[48] https://discuss.d2l.ai/t/1788

2.6.2 定义softmax操作

在实现softmax回归模型之前，让我们简要地回顾一下sum运算符如何沿着张量中的特定维度工作，如 1.3.6节 和 1.3.6节 所述。给定一个矩阵X，我们可以对所有元素求和（默认情况下），也可以只求同一个轴上的元素，即同一列（轴0）或同一行（轴1）。如果 X 是一个形状为 (2, 3) 的张量，我们对列进行求和，则结果将是一个具有形状 (3,) 的向量。当调用sum运算符时，我们可以指定保持在原始张量的轴数，而不折叠求和的维度。这将产生一个具有形状 (1, 3) 的二维张量。

```
X = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
X.sum(0, keepdims=True), X.sum(1, keepdims=True)
```

```
(array([[5., 7., 9.]]),
 array([[ 6.],
        [15.]]))
```

我们现在已经准备好实现softmax操作了。回想一下，softmax 由三个步骤组成：(1) 对每个项求幂（使用exp）；(2) 对每一行求和（小批量中每个样本是一行），得到每个样本的归一化常数；(3) 将每一行除以其归一化常数，确保结果的和为1。在查看代码之前，让我们回顾一下这个表达式：

$$\mathrm{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}. \tag{2.6.1}$$

分母或归一化常数，有时也称为配分函数（其对数称为对数-配分函数）。该名称的起源来自 统计物理学[49]中一个模拟粒子群分布的方程。

```
def softmax(X):
    X_exp = np.exp(X)
    partition = X_exp.sum(1, keepdims=True)
    return X_exp / partition  # 这里应用了广播机制
```

正如你所看到的，对于任何随机输入，我们将每个元素变成一个非负数。此外，因为概率的要求，每行总和为1。

```
X = np.random.normal(0, 1, (2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

```
(array([[0.22376052, 0.06659239, 0.06583703, 0.29964197, 0.3441681 ],
        [0.63209665, 0.03179282, 0.194987  , 0.09209415, 0.04902935]]),
 array([1.        , 0.99999994]))
```

注意，虽然这在数学上看起来是正确的，但我们在代码实现中有点草率。矩阵中的非常大或非常小的元素可能造成数值上溢或下溢，但我们没有采取措施来防止这点。

[49] https://en.wikipedia.org/wiki/Partition_function_(statistical_mechanics)

2.6.3 定义模型

现在我们已经定义了softmax操作，我们可以实现softmax回归模型。下面的代码定义了输入如何通过网络映射到输出。注意，在将数据传递到我们的模型之前，我们使用 reshape 函数将每张原始图像展平为向量。

```
def net(X):
    return softmax(np.dot(X.reshape((-1, W.shape[0])), W) + b)
```

2.6.4 定义损失函数

接下来，我们需要实现 2.4节 中引入的交叉熵损失函数。这可能是深度学习中最常见的损失函数，因为目前分类问题的数量远远超过回归问题。

回顾一下，交叉熵采用真实标签的预测概率的负对数似然。我们不需要使用Python的for循环迭代预测（这往往是低效的）。我们可以通过一个运算符选择所有元素。下面，我们一个演示数据，其中包含2个样本在3个类别的预测概率y_hat。以及它们对应的标签y。有了y，我们知道在第一个样本中，第一类是正确的预测，而在第二个样本中，第三类是正确的预测。然后使用y作为y_hat中概率的索引，我们选择第一个样本中第一个类的概率和第二个样本中第三个类的概率。

```
y = np.array([0, 2])
y_hat = np.array([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
array([0.1, 0.5])
```

现在我们只需一行代码就可以实现交叉熵损失函数。

```
def cross_entropy(y_hat, y):
    return -np.log(y_hat[range(len(y_hat)), y])

cross_entropy(y_hat, y)
```

```
array([2.3025851, 0.6931472])
```

2.6.5 分类准确率

在给定预测概率分布 y_hat时，当我们必须输出硬预测（hard prediction）时，我们通常选择预测概率最高的类。许多应用都要求我们做出选择。如Gmail必须将电子邮件分为“Primary（主要）”、“Social（社交）”、“Updates（更新）”或“Forums（论坛）”。它可能在内部估计概率，但最终它必须在类中选择一个。

当预测与标签分类 y 一致时，它们是正确的。分类准确率即正确预测数量与总预测数量之比。虽然直接优化准确率可能很困难（因为准确率的计算不可导），但准确率通常是我们最关心的性能衡量标准，我们在训练分类器时几乎总是会报告它。

为了计算准确率，我们执行以下操作。首先，如果 y_hat 是矩阵，第二个维度存储每个类的预测分数。我们使用 argmax 获得每行中最大元素的索引来获得预测类别。然后我们将预测类别与真实 y 元素进行比较。由于等式运算符 == 对数据类型很敏感，因此我们将 y_hat 的数据类型转换为与 y 的数据类型一致。结果是一个包含 0（错）和 1（对）的张量。进行求和会得到正确预测的数量。

```
def accuracy(y_hat, y):  #@save
    """计算预测正确的数量。"""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = y_hat.argmax(axis=1)
    cmp = y_hat.astype(y.dtype) == y
    return float(cmp.astype(y.dtype).sum())
```

我们将继续使用之前定义的变量 y_hat 和 y 分别作为预测的概率分布和标签。我们可以看到，第一个样本的预测类别是2（该行的最大元素为0.6，索引为2），这与实际标签0不一致。第二个样本的预测类别是2（该行的最大元素为0.5，索引为 2），这与实际标签2一致。因此，这两个样本的分类准确率率为0.5。

```
accuracy(y_hat, y) / len(y)
```

```
0.5
```

同样，我们可以评估数据迭代器 data_iter 访问的数据集在任意模型 net 上的准确率。

```
def evaluate_accuracy(net, data_iter):  #@save
    """计算在指定数据集上模型的精度。"""
    metric = Accumulator(2)  # 正确预测数、预测总数
    for X, y in data_iter:
        metric.add(accuracy(net(X), y), y.size)
    return metric[0] / metric[1]
```

这里 Accumulator 是一个实用程序类，用于对多个变量进行累加。在上面的 evaluate_accuracy 函数中，我们在 Accumulator 实例中创建了 2 个变量，用于分别存储正确预测的数量和预测的总数量。当我们遍历数据集时，两者都将随着时间的推移而累加。

```
class Accumulator:  #@save
    """在`n`个变量上累加。"""
    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a + float(b) for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0.0] * len(self.data)
```

(continues on next page)

(continued from previous page)

```
    def __getitem__(self, idx):
        return self.data[idx]
```

由于我们使用随机权重初始化 net 模型，因此该模型的准确率应接近于随机猜测。例如在有10个类别情况下的准确率为0.1。

```
evaluate_accuracy(net, test_iter)
```

```
0.0811
```

2.6.6 训练

如果你看过 2.2节 中的线性回归实现，softmax回归的训练过程代码应该看起来非常熟悉。在这里，我们重构训练过程的实现以使其可重复使用。首先，我们定义一个函数来训练一个迭代周期。请注意，updater 是更新模型参数的常用函数，它接受批量大小作为参数。它可以是封装的d2l.sgd函数，也可以是框架的内置优化函数。

```
def train_epoch_ch3(net, train_iter, loss, updater):  #@save
    """训练模型一个迭代周期（定义见第3章）。"""
    # 训练损失总和、训练准确度总和、样本数
    metric = Accumulator(3)
    if isinstance(updater, gluon.Trainer):
        updater = updater.step
    for X, y in train_iter:
        # 计算梯度并更新参数
        with autograd.record():
            y_hat = net(X)
            l = loss(y_hat, y)
        l.backward()
        updater(X.shape[0])
        metric.add(float(l.sum()), accuracy(y_hat, y), y.size)
    # 返回训练损失和训练准确率
    return metric[0] / metric[2], metric[1] / metric[2]
```

在展示训练函数的实现之前，我们定义了一个在动画中绘制数据的实用程序类。它能够简化本书其余部分的代码。

```
class Animator:  #@save
    """在动画中绘制数据。"""
    def __init__(self, xlabel=None, ylabel=None, legend=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,
```

(continues on next page)

(continued from previous page)

```
                 figsize=(3.5, 2.5)):
        # 增量地绘制多条线
        if legend is None:
            legend = []
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes,]
        # 使用lambda函数捕获参数
        self.config_axes = lambda: d2l.set_axes(self.axes[
            0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts

    def add(self, x, y):
        # 向图表中添加多个数据点
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
        self.config_axes()
        display.display(self.fig)
        display.clear_output(wait=True)
```

接下来我们实现一个训练函数，它会在train_iter 访问到的训练数据集上训练一个模型net。该训练函数将会运行多个迭代周期（由num_epochs指定）。在每个迭代周期结束时，利用 test_iter 访问到的测试数据集对模型进行评估。我们将利用 Animator 类来可视化训练进度。

```
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):  #@save
    """训练模型（定义见第3章）。"""
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
                        legend=['train loss', 'train acc', 'test acc'])
```

(continues on next page)

(continued from previous page)

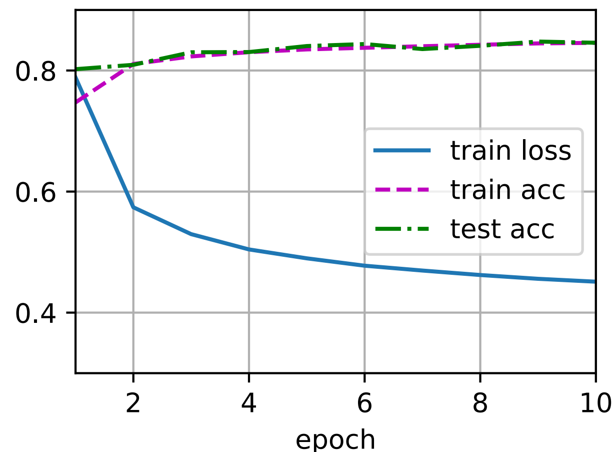
```
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
    train_loss, train_acc = train_metrics
    assert train_loss < 0.5, train_loss
    assert train_acc <= 1 and train_acc > 0.7, train_acc
    assert test_acc <= 1 and test_acc > 0.7, test_acc
```

作为一个从零开始的实现，我们使用 2.2节 中定义的小批量随机梯度下降来优化模型的损失函数，设置学习率为0.1。

```
lr = 0.1

def updater(batch_size):
    return d2l.sgd([W, b], lr, batch_size)
```

现在，我们训练模型10个迭代周期。请注意，迭代周期（num_epochs）和学习率（lr）都是可调节的超参数。通过更改它们的值，我们可以提高模型的分类准确率。

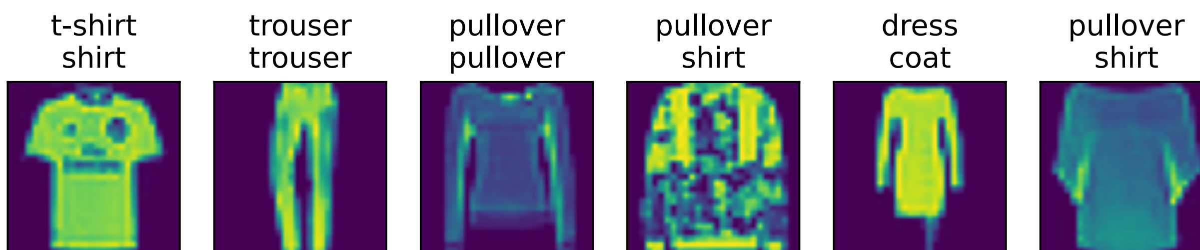
```
num_epochs = 10
train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, updater)
```



2.6.7 预测

现在训练已经完成，我们的模型已经准备好对图像进行分类。给定一系列图像，我们将比较它们的实际标签（文本输出的第一行）和模型预测（文本输出的第二行）。

```
def predict_ch3(net, test_iter, n=6):  #@save
    """预测标签（定义见第3章）。"""
    for X, y in test_iter:
        break
    trues = d2l.get_fashion_mnist_labels(y)
    preds = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1))
    titles = [true + '\n' + pred for true, pred in zip(trues, preds)]
    d2l.show_images(X[0:n].reshape((n, 28, 28)), 1, n, titles=titles[0:n])

predict_ch3(net, test_iter)
```



2.6.8 小结

- 借助 softmax 回归，我们可以训练多分类的模型。
- softmax 回归的训练循环与线性回归中的训练循环非常相似：读取数据、定义模型和损失函数，然后使用优化算法训练模型。正如你很快就会发现的那样，大多数常见的深度学习模型都有类似的训练过程。

2.6.9 练习

1. 在本节中，我们直接实现了基于数学定义softmax运算的`softmax`函数。这可能会导致什么问题？提示：尝试计算 $\exp(50)$ 的大小。
2. 本节中的函数 `cross_entropy` 是根据交叉熵损失函数的定义实现的。这个实现可能有什么问题？提示：考虑对数的值域。
3. 你可以想到什么解决方案来解决上述两个问题？
4. 返回概率最大的标签总是一个好主意吗？例如，医疗诊断场景下你会这样做吗？
5. 假设我们希望使用softmax回归来基于某些特征预测下一个单词。词汇量大可能会带来哪些问题?

Discussions[50]

2.7 softmax回归的简洁实现

在 2.3节 中，我们可以发现通过深度学习框架的高级API能够使实现线性回归变得更加容易。同样地，通过深度学习框架的高级API也能更方便地实现分类模型。让我们继续使用Fashion-MNIST数据集，并保持批量大小为256，就像在 2.6节 中一样。

```
from mxnet import gluon, init, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

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

2.7.1 初始化模型参数

如我们在 2.4节 所述，softmax 回归的输出层是一个全连接层。因此，为了实现我们的模型，我们只需在 `Sequential` 中添加一个带有10个输出的全连接层。同样，在这里，`Sequential` 并不是必要的，但我们可能会形成这种习惯。因为在实现深度模型时，`Sequential`将无处不在。我们仍然以均值0和标准差0.01随机初始化权重。

```
net = nn.Sequential()
net.add(nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

2.7.2 重新审视Softmax的实现

在前面 2.6节 的例子中，我们计算了模型的输出，然后将此输出送入交叉熵损失。从数学上讲，这是一件完全合理的事情。然而，从计算角度来看，指数可能会造成数值稳定性问题。

回想一下，softmax函数 $\hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}$，其中$\hat{y}_j$是预测的概率分布。$o_j$是未归一化的预测$\mathbf{o}$的第$j$个元素。如果$o_k$中的一些数值非常大，那么 $\exp(o_k)$ 可能大于数据类型容许的最大数字（即 上溢（overflow））。这将使分母或分子变为`inf`（无穷大），我们最后遇到的是0、`inf` 或 `nan`（不是数字）的 $\hat{y}_j$。在这些情况下，我们不能得到一个明确定义的交叉熵的返回值。

解决这个问题的一个技巧是，在继续softmax计算之前，先从所有o_k中减去$\max(o_k)$。你可以证明每个 o_k 按常数进行的移动不会改变softmax的返回值。在减法和归一化步骤之后，可能有些 o_j 具有较大的负值。由于

[50] https://discuss.d2l.ai/t/1791

精度受限，$\exp(o_j)$ 将有接近零的值，即 下溢（underflow）。这些值可能会四舍五入为零，使 $\hat{y}_j$ 为零，并且使得 $\log(\hat{y}_j)$ 的值为 -inf。反向传播几步后，我们可能会发现自己面对一屏幕可怕的nan结果。

尽管我们要计算指数函数，但我们最终在计算交叉熵损失时会取它们的对数。通过将softmax和交叉熵结合在一起，可以避免反向传播过程中可能会困扰我们的数值稳定性问题。如下面的等式所示，我们避免计算$\exp(o_j)$，而可以直接使用o_j。因为$\log(\exp(\cdot))$被抵消了。

$$\begin{aligned}
\log\left(\hat{y}_j\right) &= \log\left(\frac{\exp(o_j)}{\sum_k \exp(o_k)}\right) \\
&= \log\left(\exp(o_j)\right) - \log\left(\sum_k \exp(o_k)\right) \\
&= o_j - \log\left(\sum_k \exp(o_k)\right).
\end{aligned} \tag{2.7.1}$$

我们也希望保留传统的softmax函数，以备我们需要评估通过模型输出的概率。但是，我们没有将softmax概率传递到损失函数中，而是在交叉熵损失函数中传递未归一化的预测并同时计算softmax及其对数，这是一件聪明的事情“LogSumExp技巧”[51]。

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

2.7.3 优化算法

在这里，我们使用学习率为0.1的小批量随机梯度下降作为优化算法。这与我们在线性回归例子中的相同，这说明了优化器的普适性。

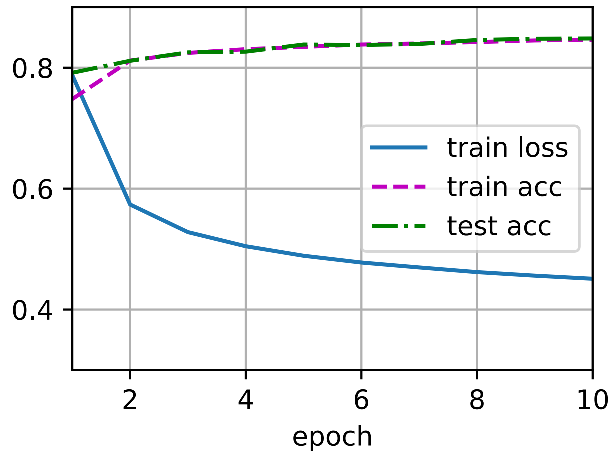
```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

2.7.4 训练

接下来我们调用 2.6节 中定义的训练函数来训练模型。

```
num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

[51] https://en.wikipedia.org/wiki/LogSumExp



和以前一样，这个算法收敛到一个相当高的精度，而且这次的代码行比以前少了。

2.7.5 小结

- 使用高级 API，我们可以更简洁地实现 softmax 回归。
- 从计算的角度来看，实现softmax回归比较复杂。在许多情况下，深度学习框架在这些著名的技巧之外采取了额外的预防措施，来确保数值的稳定性。这使我们避免了在实践中从零开始编写模型时可能遇到的陷阱。

2.7.6 练习

1. 尝试调整超参数，例如批量大小、迭代周期数和学习率，并查看结果。
2. 增加迭代周期的数量。为什么测试准确率会在一段时间后降低？我们怎么解决这个问题？

Discussions[52]

[52] https://discuss.d2l.ai/t/1794

3

多层感知机

在本章中，我们将介绍你的第一个真正的深度网络。最简单的深度网络称为多层感知机，它们由多层神经元组成，每一层都与下面一层（从中接收输入）和上面一层（反过来影响当前层的神经元）完全相连。当我们训练大容量模型时，我们面临着过拟合的风险。因此，我们需要为你提供第一次严格的概念介绍，包括过拟合、欠拟合和模型选择。为了帮助你解决这些问题，我们将介绍权重衰减和dropout等正则化技术。我们还将讨论数值稳定性和参数初始化相关的问题，这些问题是成功训练深度网络的关键。在整个过程中，我们的目标不仅是让你掌握概念，还希望让你掌握深度网络的实践方法。在本章的最后，我们将把到目前为止所介绍的内容应用到一个真实的案例：房价预测。我们将有关于模型计算性能、可伸缩性和效率相关的问题放在后面的章节中讨论。

3.1 多层感知机

在 2节 中，我们介绍了softmax回归（2.4节 ），然后我们从零开始实现softmax回归（2.6节 ），接着使用高级API实现了算法（2.7节 ），并训练分类器从低分辨率图像中识别10类服装。在这个过程中，我们学习了如何处理数据，将输出转换为有效的概率分布，并应用适当的损失函数，根据模型参数最小化损失。我们已经在简单的线性模型背景下掌握了这些知识，现在我们可以开始对深度神经网络的探索，这也是本书主要涉及的比较丰富的一类模型。

3.1.1 隐藏层

我们在 2.1.1节中描述了仿射变换，它是一个带有偏置项的线性变换。首先，回想一下如 图2.4.1中所示的softmax回归的模型结构。该模型通过单个仿射变换将我们的输入直接映射到我们的输出，然后进行softmax操作。如果我们的标签确实通过仿射变换后与我们的输入数据相关，那么这种方法就足够了。但是，仿射变换中的线性是一个很强的假设。

线性模型可能会出错

例如，线性意味着单调假设：特征的任何增大都会导致模型输出增大（如果对应的权重为正），或者导致模型输出减少（如果对应的权重为负）。有时这是有道理的。例如，如果我们试图预测一个人是否会偿还贷款。我们可以认为，在其他条件不变的情况下，收入较高的申请人总是比收入较低的申请人更有可能偿还贷款。但是，虽然收入与还款概率存在单调性，但它们不是线性相关的。收入从0增加到5万，可能比从100万增加到105万带来更大的还款可能性。处理这一问题的一种方法是对我们的数据进行预处理，使线性变得更合理，如，使用收入的对数作为我们的特征。

我们可以很容易地找出违反单调性的例子。例如，我们想要根据体温预测死亡率。对于体温高于37摄氏度的人来说，温度越高风险越大。然而，对于体温低于37摄氏度的人来说，温度越高风险就越低。在这种情况下，我们也可以通过一些巧妙的预处理来解决问题。例如，我们可以使用与37摄氏度的距离作为特征。

但是，如何对猫和狗的图像进行分类呢？增加位置(13, 17)处像素的强度是否总是增加(或总是降低)图像描绘狗的可能性？对线性模型的依赖对应于一个隐含的假设，即区分猫和狗的唯一要求是评估单个像素的强度。这种方法注定会失败，因为在这样一个世界里，把一幅图像倒过来就保留了类别。

与我们前面的例子相比，这里的线性很荒谬，而且我们难以通过简单的预处理来解决这个问题。这是因为任何像素的重要性都以复杂的方式取决于该像素的上下文（周围像素的值）。我们的数据可能会有一种表示，这种表示会考虑到我们的特征之间的相关交互作用。在此表示的基础上建立一个线性模型可能会是合适的，但我们不知道如何手动计算这么一种表示。对于深度神经网络，我们使用观测数据来联合学习隐藏层表示和应用于该表示的线性预测器。

合并隐藏层

我们可以通过合并一个或多个隐藏层来克服线性模型的限制，用来处理更一般化的函数。要做到这一点，最简单的方法是将许多全连接层堆叠在一起。每一层都输出到上面的层，直到生成最后的输出。我们可以把前$L-1$层看作表示，把最后一层看作线性预测器。这种架构通常称为多层感知机（multilayer perceptron），通常缩写为*MLP*。下面，我们以图的方式描述了多层感知机（图3.1.1）。

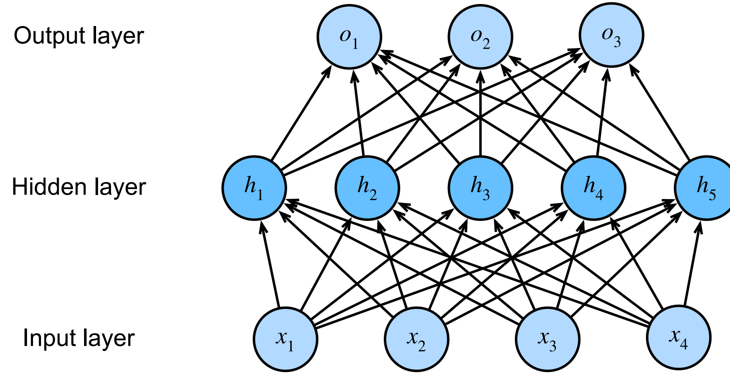


图3.1.1: 一个单隐藏层的多层感知机，具有5个隐藏单元

这个多层感知机有4个输入，3个输出，其隐藏层包含5个隐藏单元。输入层不涉及任何计算，因此使用此网络产生输出只需要实现隐藏层和输出层的计算；因此，这个多层感知机中的层数为2。注意，这两个层都是全连接的。每次输入都会影响隐藏层中的每个神经元，而隐藏层中的每个神经元又会影响输出层中的每个神经元。

然而，正如 2.4.3节 所说，具有全连接层的多层感知机的参数开销可能会高得令人望而却步，即使在不改变输入或输出大小的情况下，也可能促使在节省参数和模型效果之间进行权衡 [Zhang et al., 2021]。

从线性到非线性

跟之前的章节一样，我们通过矩阵$\mathbf{X} \in \mathbb{R}^{n \times d}$来表示$n$个样本的小批量，其中每个样本具有$d$个输入(特征)。对于具有$h$个隐藏单元的单隐藏层多层感知机，用$\mathbf{H} \in \mathbb{R}^{n \times h}$表示隐藏层的输出，称为隐藏表示（hidden representations）。在数学或代码中，$\mathbf{H}$也被称为隐藏层变量（hidden-layer variable）或隐藏变量（hidden variable）。因为隐藏层和输出层都是全连接的，所以我们具有隐藏层权重$\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$和隐藏层偏置$\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$以及输出层权重$\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$和输出层偏置$\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$。形式上，我们按如下方式计算单隐藏层多层感知机的输出$\mathbf{O} \in \mathbb{R}^{n \times q}$：

$$\begin{aligned} \mathbf{H} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{3.1.1}$$

注意，在添加隐藏层之后，模型现在需要跟踪和更新额外的参数。可我们能从中得到什么好处呢？你可能会惊讶地发现：在上面定义的模型里，我们没有好处！原因很简单。上面的隐藏单元由输入的仿射函数给出，而输出（softmax操作前）只是隐藏单元的仿射函数。仿射函数的仿射函数本身就是仿射函数。但是线性模型已经能够表示任何仿射函数。

我们可以正式地确定等价性，对于任意权重值，我们只需合并隐藏层，即可产生具有参数$\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$和$\mathbf{b} = \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$的等价单层模型：

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}. \tag{3.1.2}$$

为了发挥多层结构的潜力，我们还需要一个额外的关键要素：在仿射变换之后对每个隐藏单元应用非线性的激活函数（activation function）σ。激活函数的输出（例如，$\sigma(\cdot)$）被称为激活值（activations）。一般来说，

有了激活函数，就不可能再将我们的多层感知机退化成线性模型：

$$\begin{aligned} \mathbf{H} & = \sigma(\mathbf{X} \mathbf{W}^{(1)} + \mathbf{b}^{(1)}), \\ \mathbf{O} & = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{3.1.3}$$

由于$\mathbf{X}$中的每一行对应于小批量中的一个样本，即我们定义非线性函数σ以按行的方式应用于其输入，即，一次计算一个样本。我们在 2.4.5节 中以相同的方式使用了softmax符号来表示按行操作。但是在本节中，我们应用于隐藏层的激活函数通常不仅仅是按行的，而且也是按元素。这意味着在计算每一层的线性部分之后，我们可以计算每个激活值，而不需要查看其他隐藏单元所取的值。对于大多数激活函数都是这样。

为了构建更通用的多层感知机，我们可以继续堆叠这样的隐藏层，例如，$\mathbf{H}^{(1)} = \sigma_1(\mathbf{X} \mathbf{W}^{(1)} + \mathbf{b}^{(1)})$和$\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)} \mathbf{W}^{(2)} + \mathbf{b}^{(2)})$，一层叠一层，从而产生更有表达能力的模型。

通用近似定理

多层感知机可以通过隐藏神经元捕捉到我们输入之间的复杂相互作用，这些神经元依赖于每个输入的值。我们可以很容易地设计隐藏节点来执行任意计算。例如，在一对输入上进行基本逻辑操作。多层感知机是通用近似器。即使是网络只有一个隐藏层，给定足够的神经元（可能非常多）和正确的权重，我们可以对任意函数建模，尽管实际中学习该函数是很困难的。你可能认为神经网络有点像C语言。C语言和任何其他现代编程语言一样，能够表达任何可计算的程序。但实际上，想出一个符合规范的程序才是最困难的部分。

而且，虽然一个单隐层网络能学习任何函数，但并不意味着应该尝试使用单隐藏层网络来解决所有问题。事实上，通过使用更深（而不是更广）的网络，我们可以更容易地逼近许多函数。我们将在后面的章节中进行更细致的讨论。

```
%matplotlib inline
from mxnet import autograd, np, npx
from d2l import mxnet as d2l

npx.set_np()
```

3.1.2 激活函数

激活函数通过计算加权和并加上偏置来确定神经元是否应该被激活。它们是将输入信号转换为输出的可微运算。大多数激活函数都是非线性的。由于激活函数是深度学习的基础，下面简要介绍一些常见的激活函数。

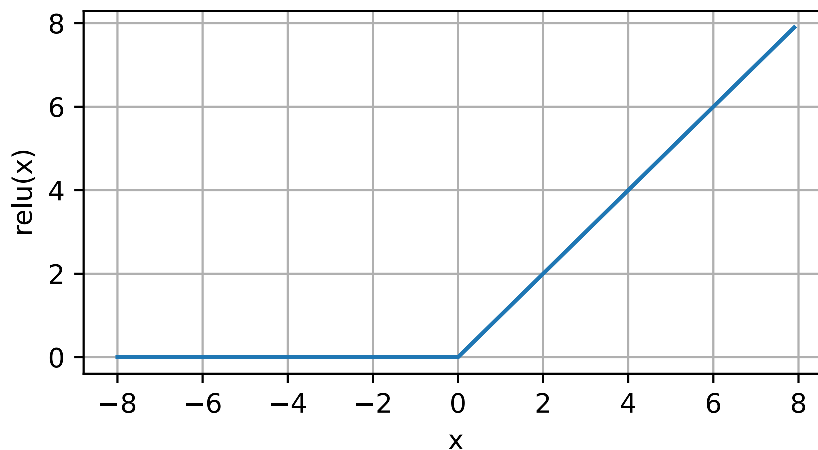
ReLU函数

最受欢迎的选择是线性整流单元（Rectified linear unit，*ReLU*），因为它实现简单，同时在各种预测任务中表现良好。ReLU提供了一种非常简单的非线性变换。给定元素x，ReLU函数被定义为该元素与0的最大值：

$$\operatorname{ReLU}(x) = \max(x, 0). \tag{3.1.4}$$

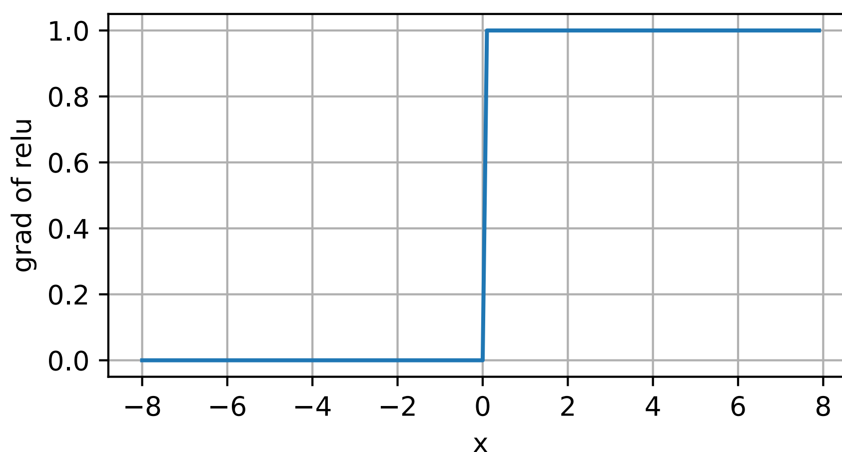
通俗地说，ReLU函数通过将相应的激活值设为0来仅保留正元素并丢弃所有负元素。为了直观感受一下，我们可以画出函数的曲线图。正如从图中所看到，激活函数是分段线性的。

```
x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.relu(x)
d2l.plot(x, y, 'x', 'relu(x)', figsize=(5, 2.5))
```



当输入为负时，ReLU函数的导数为0，而当输入为正时，ReLU函数的导数为1。注意，当输入值精确等于0时，ReLU函数不可导。在此时，我们默认使用左侧的导数，即当输入为0时导数为0。我们可以忽略这种情况，因为输入可能永远都不会是0。这里用上一句古老的谚语，“如果微妙的边界条件很重要，我们很可能是在做数学而非工程”，这个观点正好适用于这里。下面我们绘制ReLU函数的导数。

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



使用ReLU的原因是，它求导表现得特别好：要么让参数消失，要么让参数通过。这使得优化表现得更好，并且ReLU减轻了困扰以往神经网络的梯度消失问题（稍后将详细介绍）。

注意，ReLU函数有许多变体，包括参数化*ReLU*（Parameterized ReLU，*pReLU*）函数 [He et al., 2015]。该变体是为ReLU添加了一个线性项，因此即使参数是负的，某些信息仍然可以通过：

$$\operatorname{pReLU}(x) = \max(0, x) + \alpha \min(0, x). \tag{3.1.5}$$

sigmoid函数

*sigmoid*函数将定义域在$\mathbb{R}$中的输入变换为区间(0, 1)上的输出。因此，sigmoid通常称为挤压函数（Squashing function）：它将范围(-inf, inf)中的任意输入压缩到区间(0, 1)中的某个值：

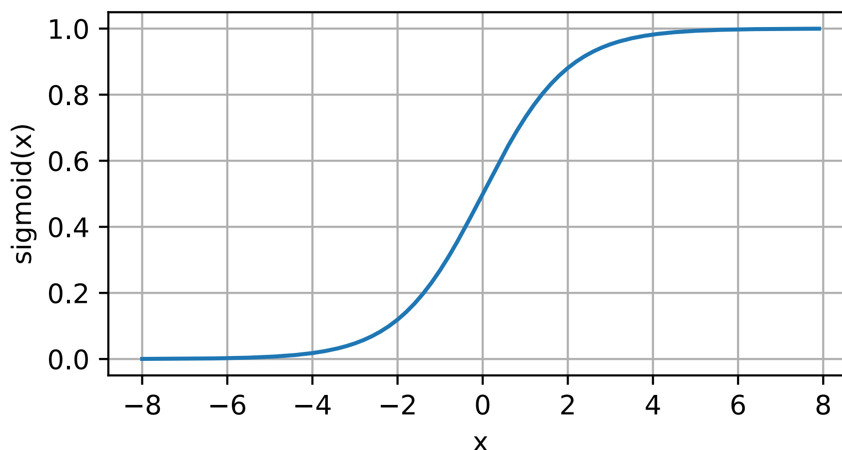
$$\operatorname{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{3.1.6}$$

在最早的神经网络中，科学家们感兴趣的是对"激发"或"不激发"的生物神经元进行建模。因此，这一领域的先驱，如人工神经元的发明者麦卡洛克和皮茨。从他们开始就专注于阈值单元。阈值单元在其输入低于某个阈值时取值0，当输入超过阈值时取值1。

当人们的注意力逐渐转移到基于梯度的学习时，sigmoid函数是一个自然的选择，因为它是一个平滑的、可微的阈值单元近似。当我们想要将输出视作二分类问题的概率时，sigmoid仍然被广泛用作输出单元上的激活函数（你可以将sigmoid视为softmax的特例）。然而，sigmoid在隐藏层中已经较少使用，它在大部分时候已经被更简单、更容易训练的ReLU所取代。在后面关于循环神经网络的章节中，我们将描述利用sigmoid单元来控制时序信息流动的结构。

下面，我们绘制sigmoid函数。注意，当输入接近0时，sigmoid函数接近线性变换。

```python
with autograd.record():
    y = npx.sigmoid(x)
d2l.plot(x, y, 'x', 'sigmoid(x)', figsize=(5, 2.5))
```

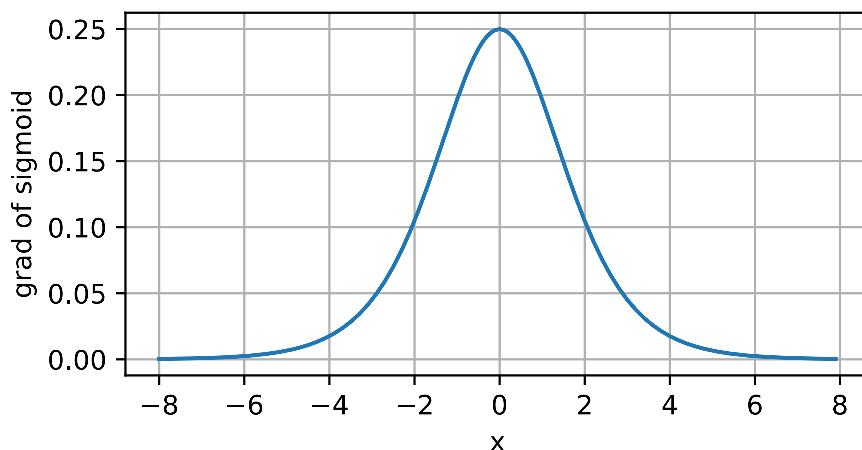


sigmoid函数的导数为下面的公式：

$$\frac{d}{dx} \operatorname{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \operatorname{sigmoid}(x)\left(1 - \operatorname{sigmoid}(x)\right). \tag{3.1.7}$$

sigmoid函数的导数图像如下所示。注意，当输入为0时，sigmoid函数的导数达到最大值0.25。当输入在任一方向上远离0点时，导数会更靠近0。

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



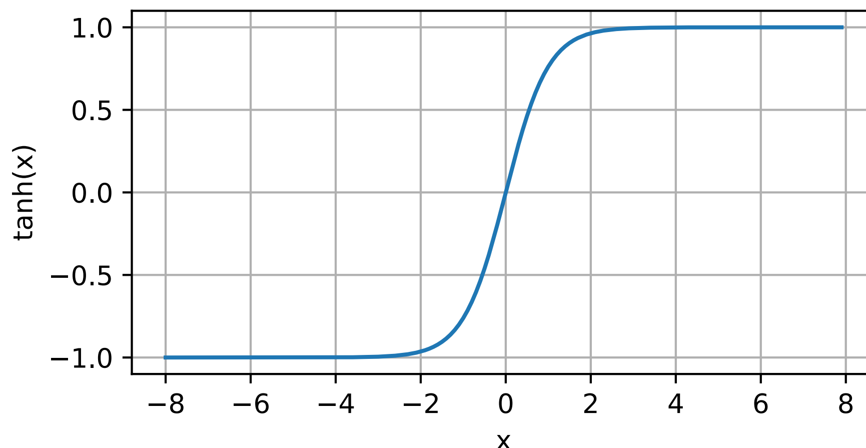
tanh函数

与sigmoid函数类似，tanh(双曲正切)函数也能将其输入压缩转换为区间(-1, 1)上。tanh函数的公式如下：

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}. \tag{3.1.8}$$

下面我们绘制tanh函数。注意，当输入在0附近时，tanh函数接近线性变换。函数的形状类似于sigmoid函数，不同的是tanh函数关于坐标系原点中心对称。

```
with autograd.record():
    y = np.tanh(x)
d2l.plot(x, y, 'x', 'tanh(x)', figsize=(5, 2.5))
```

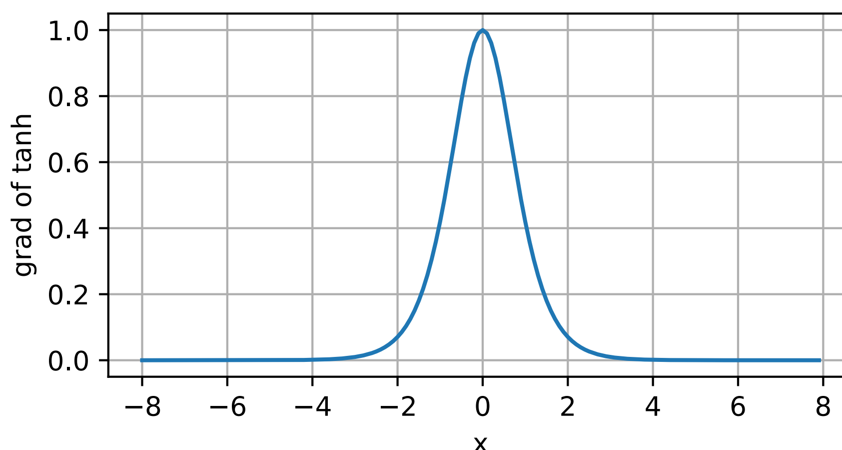


tanh函数的导数是：

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x). \tag{3.1.9}$$

tanh函数的导数图像如下所示。当输入接近0时，tanh函数的导数接近最大值1。与我们在sigmoid函数图像中看到的类似，当输入在任一方向上远离0点时，导数会更靠近0。

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



总结一下，我们现在知道如何结合非线性函数来构建更强表达能力的多层神经网络结构。顺便说一句，你的知识已经让你掌握了一个类似于1990年左右深度学习从业者的工具。在某些方面，你比在20世纪90年代工作的任何人都有优势。这是因为你可以利用功能强大的开源深度学习框架。你只需使用几行代码就可以快速构建模型。在以前，训练这些网络需要研究人员编写数千行的C或Fortran代码。

3.1.3 小结

- 多层感知机在输出层和输入层之间增加一个或多个全连接的隐藏层，并通过激活函数转换隐藏层的输出。
- 常用的激活函数包括ReLU函数、sigmoid函数和tanh函数。

3.1.4 练习

1. 计算pReLU激活函数的导数。
2. 证明一个仅使用ReLU（或pReLU）的多层感知机构造了一个连续的分段线性函数。
3. 证明$\tanh(x) + 1 = 2\,\mathrm{sigmoid}(2x)$。
4. 假设我们有一个非线性单元，将它一次应用于一个小批量的数据。你认为这会导致什么样的问题？

Discussions[53]

3.2 多层感知机的从零开始实现

我们已经在数学上描述了多层感知机（MLP），现在让我们尝试自己实现一个多层感知机。为了与我们之前使用softmax回归（2.6节）获得的结果进行比较，我们将继续使用Fashion-MNIST图像分类数据集（2.5节）。

```
from mxnet import gluon, np, npx
from d2l import mxnet as d2l

npx.set_np()
```

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

3.2.1 初始化模型参数

回想一下，Fashion-MNIST中的每个图像由$28 \times 28 = 784$个灰度像素值组成。所有图像共分为10个类别。忽略像素之间的空间结构，我们可以将每个图像视为具有784个输入特征和10个类的简单分类数据集。首先，我们将实现一个具有1个隐藏层的多层感知机，其中包含256个隐藏单元。注意，我们可以将这两个量都视为超参数。通常，我们选择2的幂次方作为层的宽度。因为内存在硬件中的分配和寻址方式，这么做往往可以在计算上更高效。

我们用几个张量来表示我们的参数。注意，对于每一层我们都要记录一个权重矩阵和一个偏置向量。跟以前一样，我们要为这些参数的损失的梯度分配内存。

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = np.random.normal(scale=0.01, size=(num_inputs, num_hiddens))
b1 = np.zeros(num_hiddens)
W2 = np.random.normal(scale=0.01, size=(num_hiddens, num_outputs))
b2 = np.zeros(num_outputs)
params = [W1, b1, W2, b2]

for param in params:
    param.attach_grad()
```

[53] https://discuss.d2l.ai/t/1797

3.2.2 激活函数

为了确保我们知道一切是如何工作的，我们将使用最大值函数自己实现ReLU激活函数，而不是直接调用内置的relu函数。

```
def relu(X):
    return np.maximum(X, 0)
```

3.2.3 模型

因为我们忽略了空间结构，所以我们使用reshape将每个二维图像转换为一个长度为num_inputs的向量。我们只需几行代码就可以实现我们的模型。

```
def net(X):
    X = X.reshape((-1, num_inputs))
    H = relu(np.dot(X, W1) + b1)
    return np.dot(H, W2) + b2
```

3.2.4 损失函数

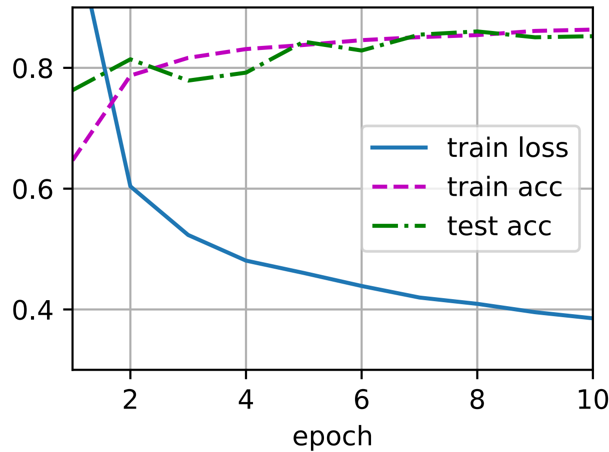
为了确保数值稳定性，同时由于我们已经从零实现过softmax函数（2.6节），因此在这里我们直接使用高级API中的内置函数来计算softmax和交叉熵损失。回想一下我们之前在 2.7.2节 中对这些复杂问题的讨论。我们鼓励感兴趣的读者查看损失函数的源代码，以加深对实现细节的了解。

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

3.2.5 训练

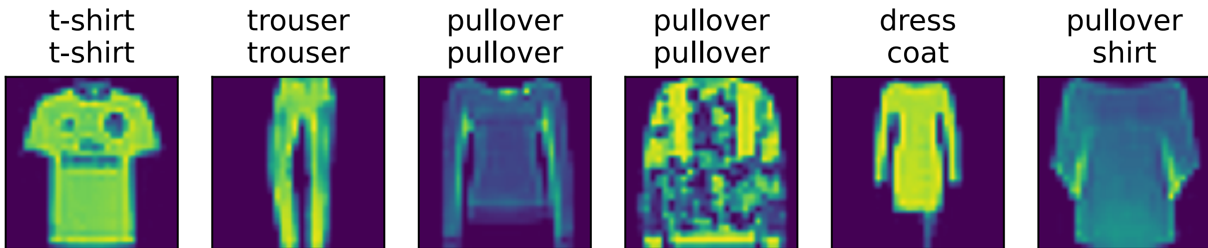
幸运的是，多层感知机的训练过程实现与softmax回归的训练过程实现完全相同。可以直接调用d2l包的train_ch3函数（参见 2.6节），将迭代周期数设置为10，并将学习率设置为0.1.

```
num_epochs, lr = 10, 0.1
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
              lambda batch_size: d2l.sgd(params, lr, batch_size))
```



为了对学习到的模型进行评估，我们将在一些测试数据上应用这个模型。

```
d2l.predict_ch3(net, test_iter)
```



3.2.6 小结

- 我们看到即使手动实现一个简单的多层感知机也是很容易的。
- 然而，如果有大量的层，从零开始实现多层感知机会变得很麻烦（例如，要命名和记录模型的参数）。

3.2.7 练习

1. 在所有其他参数保持不变的情况下，更改超参数`num_hiddens`的值，并查看此超参数的变化对结果有何影响。确定此超参数的最佳值。
2. 尝试添加更多的隐藏层，并查看它对结果有何影响。
3. 改变学习速率会如何影响结果？保持模型结构和其他超参数(包括迭代周期数)不变，学习率设置为多少会带来最好的结果？
4. 通过对所有超参数(学习率、迭代周期数、隐藏层数、每层的隐藏单元数)进行联合优化，可以得到的最佳结果是什么？

5. 描述为什么涉及多个超参数更具挑战性。
6. 如果要构建多个超参数的搜索方法，你能想到的最聪明的策略是什么？

Discussions[54]

3.3 多层感知机的简洁实现

正如你所期待的，我们可以通过高级API更简洁地实现多层感知机。

```
from mxnet import gluon, init, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

3.3.1 模型

与softmax回归的简洁实现（2.7节）相比，唯一的区别是我们添加了2个全连接层（之前我们只添加了1个全连接层）。第一层是隐藏层，它包含256个隐藏单元并使用了ReLU激活函数。第二层是输出层。

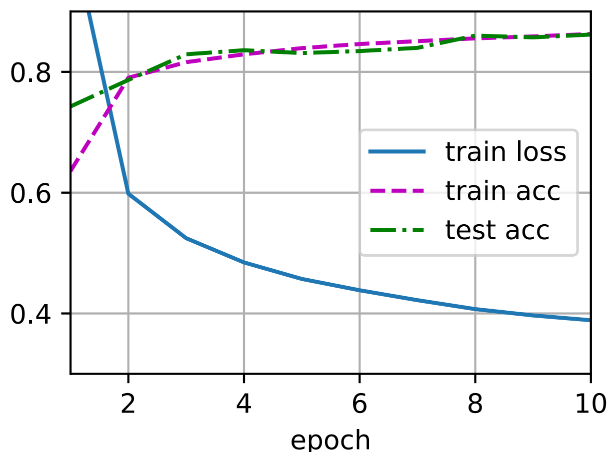
```
net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'), nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

训练过程实现与我们实现softmax回归时完全相同。这种模块化设计使我们能够将与和模型架构有关的内容独立出来。

```
batch_size, lr, num_epochs = 256, 0.1, 10
loss = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
```

```
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

[54] https://discuss.d2l.ai/t/1800



3.3.2 小结

- 我们可以使用高级API更简洁地实现多层感知机。
- 对于相同的分类问题，多层感知机的实现与softmax回归的实现相同，只是多层感知机的实现里增加了带有激活函数的隐藏层。

3.3.3 练习

1. 尝试添加不同数量的隐藏层（也可以修改学习率）。怎么样设置效果最好？
2. 尝试不同的激活函数。哪个效果最好？
3. 尝试不同的方案来初始化权重。什么方法效果最好？

Discussions[55]

3.4 模型选择、欠拟合和过拟合

作为机器学习科学家，我们的目标是发现模式（pattern）。但是，我们如何才能确定模型是真正发现了一种泛化的模式，而不是简单地记住了数据呢？例如，我们想要在患者的基因数据与痴呆状态之间寻找模式，其中标签是从集合{痴呆, 轻度认知障碍, 健康}中提取的。因为基因可以唯一确定每个个体（不考虑双胞胎），所以在这个任务中是有可能记住整个数据集的。

我们不想让模型只会做这样的事情："那是鲍勃！我记得他！他有痴呆症！"。原因很简单。当我们将来部署该模型时，模型会遇到从未见过的患者。只有当我们的模型真正发现了一种泛化模式时，才会作出有效的预测。

更正式地来说，我们的目标是发现模式，这些模式捕捉到了我们训练集所来自的潜在总体的规律。如果成功做到了这点，即使是对我们以前从未遇到过的个体，我们也可以成功地评估风险。如何发现可以泛化的模式是机器学习的根本问题。

[55] https://discuss.d2l.ai/t/1803

困难在于，当我们训练模型时，我们只能访问数据中的小部分样本。最大的公开图像数据集包含大约一百万张图像。而在大部分时候，我们只能从数千或数万个数据样本中学习。在大型医院系统中，我们可能会访问数十万份医疗记录。当我们使用有限的样本时，可能会遇到这样的问题：当收集到更多的数据时，会发现之前找到的明显关系并不成立。

将模型在训练数据上拟合得比在潜在分布中更接近的现象称为过拟合，用于对抗过拟合的技术称为正则化。在前面的章节中，你可能在用Fashion-MNIST数据集做实验时已经观察到了这种现象。在实验中调整模型结构或超参数时，如果有足够多的神经元、层数和训练迭代周期，你会观察到模型可以在训练集上达到完美的精度。虽然此时测试集的准确性会下降。

3.4.1 训练误差和泛化误差

为了进一步讨论这一现象，我们需要了解训练误差和泛化误差。训练误差（training error）是指，我们的模型在训练数据集上计算得到的误差。泛化误差（generalization error）是指，当我们将模型应用在同样从原始样本的分布中抽取的无限多的数据样本时，我们模型误差的期望。

问题是，我们永远不能准确地计算出泛化误差。这是因为无限多的数据样本是一个虚构的对象。在实际中，我们只能通过将模型应用于一个独立的测试集来估计泛化误差，该测试集由从训练集中随机选择并保留的数据样本组成。

下面的三个思考实验将有助于更好地说明这种情况。假设一个大学生正在努力准备期末考试。一个勤奋的学生会努力做好练习，并利用往年的考试题目来测试自己的能力。尽管如此，在过去的考试题目上取得好成绩并不能保证他会在真正考试时发挥出色。例如，学生可能试图通过死记硬背考题的答案来做准备。他甚至可以完全记住过去考试的答案。另一名学生可能会通过试图理解给出某些答案的原因来做准备。在大多数情况下，后一个学生会考得更好。

类似地，考虑一个简单地使用查表法来回答问题的模型。如果允许的输入集合是离散的并且相当小，那么也许在查看许多训练样本后，该方法将执行得很好。但当面对这个模型从未见过的例子时，它表现的可能比随机猜测好不到哪去。这是因为输入空间太大了，远远不可能记住每一个可能输入的对应答案。例如，考虑28×28的灰度图像。如果每个像素可以取256个灰度值中的一个，则有256^{784}个可能的图像。这意味着指甲大小的低分辨率灰度图像的数量比宇宙中的原子要多得多。即使我们可能遇到这样的数据，我们也不可能存储整个查找表。

最后，考虑尝试根据一些可用的上下文特征对掷硬币的结果（类别0：正面，类别1：反面）进行分类的问题。假设硬币投掷是公平的。无论我们想出什么算法，泛化误差始终是$\frac{1}{2}$。然而，对于大多数算法，我们应该期望训练误差会更低，这取决于运气。考虑数据集{0，1，1，1，0，1}。我们的算法不需要额外的特征，将倾向于总是预测多数类，从我们有限的样本来看，它似乎是*1*。在这种情况下，总是预测类1的模型将产生$\frac{1}{3}$的误差，这比我们的泛化误差要好得多。当我们逐渐增加数据量，正面比例明显偏离$\frac{1}{2}$的可能性将会降低，我们的训练误差将与泛化误差相匹配。

统计学习理论

由于泛化是机器学习中的基本问题，许多数学家和理论家毕生致力于研究描述这一现象的形式理论。在同名定理（eponymous theorem）[56]]中，格里文科和坎特利推导出了训练误差收敛到泛化误差的速率。在一系列开创性的论文中，Vapnik和Chervonenkis[57]将这一理论扩展到更一般种类的函数。这项工作为统计学习理论奠定了基础。

在有监督中，我们到目前为止都基于一个假设，并且这个假设将贯穿本书的大部分内容。即我们假设训练数据和测试数据都是从相同的分布中独立提取的。这通常被称为独立同分布假设（i.i.d. assumption），这意味着对数据进行采样的过程没有进行"记忆"。换句话说，抽取的第2个样本和第3个样本的相关性并不比抽取的第2个样本和第200万个样本的相关性更强。

要成为一名优秀的机器学习科学家需要有批判性的思考，而且你应该已经从这个假设中找出漏洞，即很容易找出假设失效的情况。如果我们根据从加州大学旧金山分校医学中心的患者数据训练死亡风险预测模型，并将其应用于马萨诸塞州综合医院的患者数据，结果会怎么样？这两个数据的分布可能不完全一样。此外，抽样过程可能与时间有关。比如当我们对微博的主题进行分类时，新闻周期会使得正在讨论的话题产生时间依赖性，这违反了独立性的假设。

有时候我们即使轻微违背独立同分布假设，模型仍将继续运行得非常好。毕竟，几乎所有现实的应用都至少涉及到一些违背独立同分布假设的情况。然而，我们仍然有许多有用的工具已经应用于现实，如人脸识别、语音识别和语言翻译。

有些违背独立同分布假设的行为肯定会带来麻烦。比如，如果我们试图训练一个人脸识别系统，只用来自大学生的人脸数据进行训练，然后想要将其部署为一种工具，用于监测疗养院人群中的老人。这不太可能有效，因为大学生看起来往往与老年人有很大的不同。

在接下来的章节中，我们将讨论因违背独立同分布假设而引起的问题。目前，即使认为独立同分布假设是理所当然的，理解泛化也是一个困难的问题。此外，能够解释深层神经网络泛化性能的理论基础，也仍在继续困扰着学习理论领域最伟大的学者们。

当我们训练模型时，我们试图找到一个能够尽可能拟合训练数据的函数。如果该函数灵活到可以像捕捉真实模式一样容易地捕捉到干扰的模式，那么它可能执行得"太好了"，而不会产生一个对看不见的数据进行很好概括的模型。但我们想要避免这样，或者想要至少能够控制这种现象的出现。深度学习中有许多启发式的技术旨在防止过拟合。

模型复杂性

当我们有简单的模型和大量的数据时，我们期望泛化误差与训练误差相近。当我们有更复杂的模型和更少的样本时，我们预计训练误差会下降，但泛化误差会增大。模型复杂性由什么构成是一个复杂的问题。一个模型是否能很好地泛化取决于很多因素。例如，具有更多参数的模型可能被认为更复杂。其参数有更大取值范围的模型可能更为复杂。通常，对于神经网络，我们认为需要更多训练迭代的模型比较复杂，而需要"提前停止"（early stopping）的模型（意味着具有较少训练迭代周期）就不那么复杂。

[56] https://en.wikipedia.org/wiki/Glivenko%E2%80%93Cantelli_theorem

[57] https://en.wikipedia.org/wiki/Vapnik%E2%80%93Chervonenkis_theory

很难比较本质上不同大类的模型之间（例如，决策树与神经网络）的复杂性。就目前而言，一条简单的经验法则相当有用：统计学家认为，能够轻松解释任意事实的模型是复杂的，而表达能力有限但仍能很好地解释数据的模型可能更有现实用途。在理论上，这与波普尔的科学理论的可证伪性标准密切相关：如果一个理论能拟合数据，且有具体的测试可以用来证明它是错误的，那么它就是好的。这一点很重要，因为所有的统计估计都是事后归纳，也就是说，我们在观察事实之后进行估计，因此容易受到相关谬误的影响。目前，我们将把理论放在一边，坚持更切实的问题。

在本节中，为了给你一些直观的印象，我们将重点介绍几个倾向于影响模型泛化的因素：

1. 可调整参数的数量。当可调整参数（有时称为自由度）的数量很大时，模型往往更容易过拟合。
2. 参数采用的值。当权重的取值范围较大时，模型可能更容易过拟合。
3. 训练样本的数量。即使你的模型很简单，也很容易过拟合只包含一个或两个样本的数据集。但是，过拟合一个数百万个样本数据集需要一个极其灵活的模型。

3.4.2 模型选择

在机器学习中，我们通常在评估几个候选模型后选择最终的模型。这个过程叫做模型选择。有时，需要进行比较的模型在本质上是完全不同的（比如，决策树与线性模型）。又有时，我们需要比较不同的超参数设置下的同一类模型。

例如，我们要训练多层感知机模型，我们可能希望比较具有不同数量的隐藏层、不同数量的隐藏单元以及不同的的激活函数组合的模型。为了确定候选模型中的最佳模型，我们通常会使用验证集。

验证集

原则上，在我们确定所有的超参数之前，我们不应该用到测试集。如果我们在模型选择过程中使用测试数据，可能会有过拟合测试数据的风险。那我们就麻烦大了。如果我们过拟合了我们的训练数据，还有在测试数据上的评估来判断过拟合。但是如果我们过拟合了测试数据，我们又该怎么知道呢？

因此，我们决不能依靠测试数据进行模型选择。然而，我们也不能仅仅依靠训练数据来选择模型，因为我们无法估计训练数据的泛化误差。

在实际应用中，情况变得更加复杂。虽然理想情况下我们只会使用测试数据一次，以评估最好的模型或比较一些模型效果，但现实是，测试数据很少在使用一次后被丢弃。我们很少能有充足的数据来对每一轮实验采用全新测试集。

解决此问题的常见做法是将我们的数据分成三份，除了训练和测试数据集之外，还增加一个验证数据集（validation dataset），也叫验证集（validation set）。但现实是验证数据和测试数据之间的边界模糊得令人担忧。除非另有明确说明，否则在这本书的实验中，我们实际上是在使用应该被正确地称为训练数据和验证数据的东西，并没有真正的测试数据集。因此，书中每次实验报告的准确度都是验证集准确度，而不是测试集准确度。

K折交叉验证

当训练数据稀缺时，我们甚至可能无法提供足够的数据来构成一个合适的验证集。这个问题的一个流行的解决方案是采用K折交叉验证*。这里，原始训练数据被分成K个不重叠的子集。然后执行K次模型训练和验证，每次在$K-1$个子集上进行训练，并在剩余的一个子集(在该轮中没有用于训练的子集)上进行验证。最后，通过对K次实验的结果取平均来估计训练和验证误差。

3.4.3 欠拟合还是过拟合？

当我们比较训练和验证误差时，我们要注意两种常见的情况。首先，我们要注意这样的情况：训练误差和验证误差都很严重，但它们之间仅有一点差距。如果模型不能降低训练误差，这可能意味着我们的模型过于简单(即，表达能力不足)，无法捕获我们试图学习的模式。此外，由于我们的训练和验证误差之间的泛化误差很小，我们有理由相信可以用一个更复杂的模型降低训练误差。这种现象被称为欠拟合（underfitting）。

另一方面，当我们的训练误差明显低于验证误差，表明了严重的过拟合（overfitting）。注意，过拟合并不总是一件坏事。特别是在深度学习领域，众所周知，最好的预测模型在训练数据上的表现往往比在保留数据上好得多。最终，我们通常更关心验证误差，而不是训练误差和验证误差之间的差距。

我们是否过拟合可能取决于模型复杂性和可用训练数据集的大小，这两个点将在下面进行讨论。

模型复杂性

为了说明一些关于过拟合和模型复杂性的经典的直觉，我们给出一个多项式的例子。给定由单个特征x和对应实数标签y组成的训练数据，我们试图找到下面的d阶多项式来估计标签y。

$$\hat{y} = \sum_{i=0}^{d} x^i w_i \tag{3.4.1}$$

这只是一个线性回归问题，我们的特征是x的幂给出的，模型的权重是w_i给出的，偏置是w_0给出的（因为对于所有的x都有$x^0=1$）。由于这只是一个线性回归问题，我们可以使用平方误差作为我们的损失函数。

高阶多项式函数比低阶多项式函数复杂得多。高阶多项式的参数较多，模型函数的选择范围较广。因此在固定训练数据集的情况下，高阶多项式函数相对于低阶多项式的训练误差应该始终更低(最坏情况下是相等的)。事实上，当数据样本包含了x的不同值时，函数阶数等于数据样本数量的多项式函数就可以很好地拟合训练集。在 图3.4.1 中，我们直观地描述了多项式的阶数和欠拟合与过拟合之间的关系。

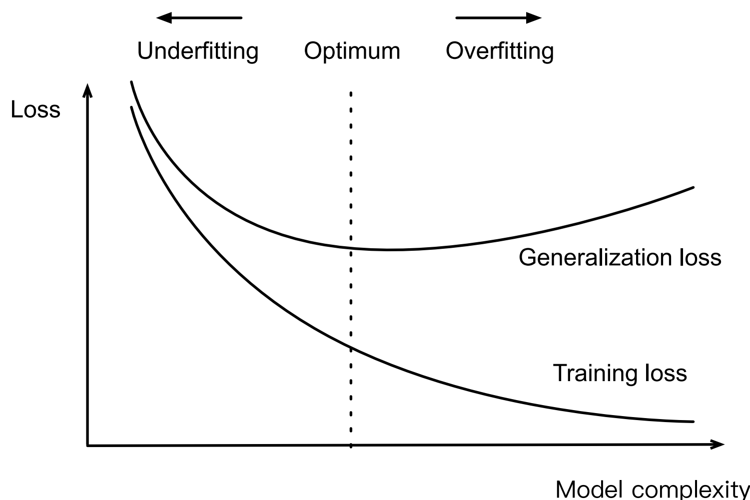


图3.4.1: 模型复杂度对欠拟合和过拟合的影响

数据集大小

另一个需要牢记的重要因素是数据集的大小。训练数据集中的样本越少，我们就越有可能（而且更严重）遇到过拟合。随着训练数据量的增加，泛化误差通常会减小。此外，一般来说，更多的数据不会有什么坏处。对于固定的任务和数据分布，通常在模型复杂性和数据集大小之间存在关系。给出更多的数据，我们可能会尝试拟合一个更复杂的模型。能够拟合更复杂的模型可能是有益的。如果没有足够的数据，简单的模型可能更有用。对于许多任务，深度学习只有在有数千个训练样本时才优于线性模型。从一定程度上来说，深度学习目前的成功要归功于互联网公司、廉价存储、互联设备以及经济数字化带来的海量数据集。

3.4.4 多项式回归

我们现在可以通过将对数据进行多项式拟合来交互地探索这些概念。

```
import math
from mxnet import gluon, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

生成数据集

首先，我们需要数据。给定x，我们将使用以下三阶多项式来生成训练和测试数据的标签：

$$y = 5 + 1.2x - 3.4\frac{x^2}{2!} + 5.6\frac{x^3}{3!} + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.1^2). \tag{3.4.2}$$

噪声项ϵ服从均值为0且标准差为0.1的正态分布。在优化的过程中，我们通常希望避免非常大的梯度值或损失值。这就是我们将特征从x^i调整为$\frac{x^i}{i!}$的原因，这样可以避免对于很大的i得到特别大的指数值。我们将为训练集和测试集各合成100个样本。

```
max_degree = 20  # 多项式的最大阶数
n_train, n_test = 100, 100  # 训练和测试数据集大小
true_w = np.zeros(max_degree)  # 分配大量的空间
true_w[0:4] = np.array([5, 1.2, -3.4, 5.6])

features = np.random.normal(size=(n_train + n_test, 1))
np.random.shuffle(features)
poly_features = np.power(features, np.arange(max_degree).reshape(1, -1))
for i in range(max_degree):
    poly_features[:, i] /= math.gamma(i + 1)  # `gamma(n)` = (n-1)!
# `labels`的维度: (`n_train` + `n_test`,)
labels = np.dot(poly_features, true_w)
labels += np.random.normal(scale=0.1, size=labels.shape)
```

同样，存储在poly_features中的单项式由gamma函数重新缩放，其中$\Gamma(n) = (n-1)!$。从生成的数据集中查看前2个样本。值1是与偏置相对应的常量特征。

```
features[:2], poly_features[:2, :], labels[:2]
```

```
(array([[-0.03716067],
        [-1.1468065 ]]),
 array([[ 1.0000000e+00, -3.7160669e-02,  6.9045764e-04, -8.5526226e-06,
          7.9455290e-08, -5.9052235e-10,  3.6573678e-12, -1.9415747e-14,
          9.0187767e-17, -3.7238198e-19,  1.3837962e-21, -4.6747992e-24,
          1.4476556e-26, -4.1381425e-29,  1.0984010e-31, -2.7211542e-34,
          6.3199942e-37, -1.3815009e-39,  2.8516424e-42, -5.6051939e-45],
        [ 1.0000000e+00, -1.1468065e+00,  6.5758252e-01, -2.5137332e-01,
          7.2069131e-02, -1.6529869e-02,  3.1594271e-03, -5.1760738e-04,
          7.4199430e-05, -9.4547095e-06,  1.0842722e-06, -1.1304095e-07,
          1.0803007e-08, -9.5299690e-10,  7.8064499e-11, -5.9683248e-12,
          4.2778208e-13, -2.8857840e-14,  1.8385756e-15, -1.1097316e-16]]),
 array([ 5.1432443 , -0.06415121]))
```

对模型进行训练和测试

首先让我们实现一个函数来评估模型在给定数据集上的损失。

```
def evaluate_loss(net, data_iter, loss):  #@save
    """评估给定数据集上模型的损失。"""
    metric = d2l.Accumulator(2)  # 损失的总和, 样本数量
    for X, y in data_iter:
        l = loss(net(X), y)
        metric.add(l.sum(), l.size)
    return metric[0] / metric[1]
```

现在定义训练函数。

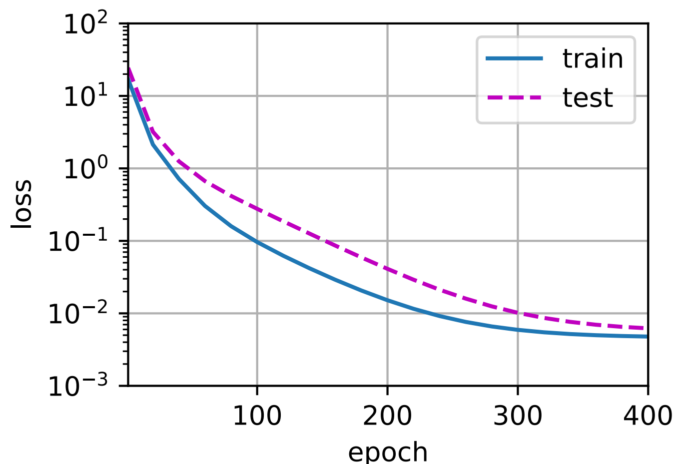
```
def train(train_features, test_features, train_labels, test_labels,
          num_epochs=400):
    loss = gluon.loss.L2Loss()
    net = nn.Sequential()
    # 不设置偏置，因为我们已经在多项式特征中实现了它
    net.add(nn.Dense(1, use_bias=False))
    net.initialize()
    batch_size = min(10, train_labels.shape[0])
    train_iter = d2l.load_array((train_features, train_labels), batch_size)
    test_iter = d2l.load_array((test_features, test_labels), batch_size,
                               is_train=False)
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': 0.01})
    animator = d2l.Animator(xlabel='epoch', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], ylim=[1e-3, 1e2],
                            legend=['train', 'test'])
    for epoch in range(num_epochs):
        d2l.train_epoch_ch3(net, train_iter, loss, trainer)
        if epoch == 0 or (epoch + 1) % 20 == 0:
            animator.add(epoch + 1, (evaluate_loss(
                net, train_iter, loss), evaluate_loss(net, test_iter, loss)))
    print('weight:', net[0].weight.data().asnumpy())
```

三阶多项式函数拟合(正态)

我们将首先使用三阶多项式函数，它与数据生成函数的阶数相同。结果表明，该模型能有效降低训练损失和测试损失。学习到的模型参数也接近真实值$w = [5, 1.2, -3.4, 5.6]$。

```
# 从多项式特征中选择前4个维度，即 1, x, x^2/2!, x^3/3!
train(poly_features[:n_train, :4], poly_features[n_train:, :4],
      labels[:n_train], labels[n_train:])
```

```
weight: [[ 5.0191116  1.2219777 -3.423689   5.571651 ]]
```

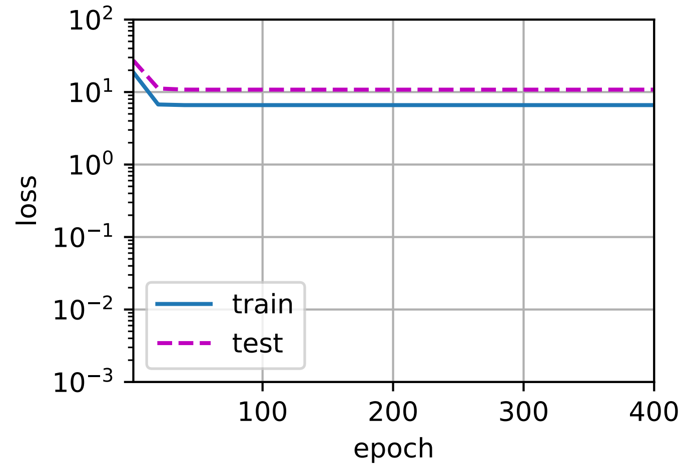


线性函数拟合(欠拟合)

让我们再看看线性函数拟合。在经历了早期的下降之后，进一步减少该模型的训练损失变得困难。在最后一个迭代周期完成后，训练损失仍然很高。当用来拟合非线性模式（如这里的三阶多项式函数）时，线性模型容易欠拟合。

```
# 从多项式特征中选择前2个维度，即 1, x
train(poly_features[:n_train, :2], poly_features[n_train:, :2],
      labels[:n_train], labels[n_train:])
```

```
weight: [[2.7092137 4.213667 ]]
```

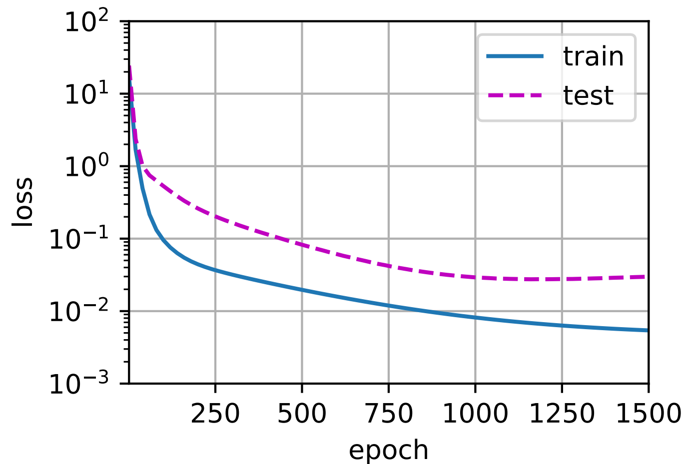


高阶多项式函数拟合(过拟合)

现在，让我们尝试使用一个过于高阶的多项式来训练模型。在这种情况下，没有足够的数据用于学到高阶系数应该具有接近于零的值。因此，这个过于复杂的模型会轻易受到训练数据中噪声的影响。虽然训练损失可以有效地降低，但测试损失仍然很高。结果表明，复杂模型对数据造成了过拟合。

```
# Pick all the dimensions from the polynomial features
train(poly_features[:n_train, :], poly_features[n_train:, :],
      labels[:n_train], labels[n_train:], num_epochs=1500)
```

```
weight: [[ 4.99207     1.3059171  -3.3530948   5.116551   -0.11148077  1.3030204
   0.12671319  0.16651939  0.05128141 -0.02275307  0.00806017 -0.05167844
  -0.02426324 -0.01502205 -0.04941358  0.06389865 -0.04761846 -0.04380166
  -0.05188227  0.05655775]]
```



在接下来的章节中，我们将继续讨论过拟合问题和处理这些问题的方法，例如权重衰减和dropout。

3.4.5 小结

- 由于不能基于训练误差来估计泛化误差，因此简单地最小化训练误差并不一定意味着泛化误差的减小。机器学习模型需要注意防止过拟合，来使得泛化误差最小。
- 验证集可以用于模型选择，但不能过于随意地使用它。
- 欠拟合是指模型无法继续减少训练误差。过拟合是指训练误差远小于验证误差。
- 我们应该选择一个复杂度适当的模型，避免使用数量不足的训练样本。

3.4.6 练习

1. 你能准确地解出这个多项式回归问题吗？提示：使用线性代数。
2. 考虑多项式的模型选择：
    1. 绘制训练损失与模型复杂度（多项式的阶数）的关系图。你观察到了什么？需要多少阶的多项式才能将训练损失减少到0?
    2. 在这种情况下绘制测试的损失图。
    3. 生成同样的图，作为数据量的函数。
3. 如果你不对多项式特征x^i进行标准化($1/i!$)，会发生什么事情？你能用其他方法解决这个问题吗？
4. 你能期待看到泛化误差为零吗?

Discussions[58]

3.5 权重衰减

我们已经描述了过拟合的问题，现在我们可以介绍一些正则化模型的技术。我们总是可以通过去收集更多的训练数据来缓解过拟合。但这可能成本很高而且耗时.或者完全超出我们的控制，在短期内不可能做到。假设已经拥有尽可能多的高质量数据，现在我们将重点放在正则化技术上。

回想一下，在多项式回归的例子（3.4节）中，我们可以通过调整拟合多项式的阶数来限制模型的容量。实际上，限制特征的数量是缓解过拟合的一种常用技术。然而，简单地丢弃特征对于这项工作来说可能过于生硬。我们继续思考多项式回归的例子，考虑高维输入可能发生的情况。多项式对多变量数据的自然扩展称为单项式（monomials)，也可以说是变量幂的乘积。单项式的阶数是幂的和。例如，$x_1^2x_2$和$x_3x_5^2$都是3次单项式。

注意，随着阶数d的增长，带有阶数d的项数迅速增加。给定k个变量，阶数d（即k多选d）的个数为$\binom{k-1+d}{k-1}$。即使是阶数上的微小变化，比如从2到3，也会显著增加我们模型的复杂性。因此，我们经常需要一个更细粒度的工具来调整函数的复杂性。

[58] https://discuss.d2l.ai/t/1807

3.5.1 范数与权重衰减

在之前的章节，我们已经描述了L_2范数和L_1范数，它们是L_p范数的特殊情况。权重衰减（通常称为L_2正则化），可能是最广泛使用的对参数化机器学习模型进行正则化的技术。这项技术是基于一个基本直觉，即在所有函数f中，函数$f = 0$（所有输入都得到值0）在某种意义上是最简单的，我们可以通过函数与零的距离来衡量函数的复杂度。但是我们应该如何精确地测量一个函数和零之间的距离呢？没有一个正确的答案。事实上，整个数学分支，包括函数分析和巴拿赫空间理论，都在致力于回答这个问题。

一种简单的方法是通过线性函数$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$中的权重向量的某个范数来度量其复杂性，例如$\|\mathbf{w}\|^2$。要保证权重向量比较小，最常用方法是将其范数作为惩罚项加到最小化损失的问题中。将原来的训练目标最小化训练标签上的预测损失，调整为最小化预测损失和惩罚项之和。现在，如果我们的权重向量增长的太大，我们的学习算法可能会更集中于最小化权重范数$\|\mathbf{w}\|^2$。这正是我们想要的。让我们回顾一下 2.1节 中的线性回归例子。我们的损失由下式给出：

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^n \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{3.5.1}$$

回想一下，$\mathbf{x}^{(i)}$是样本i的特征，$y^{(i)}$是样本i的标签。$(\mathbf{w}, b)$是权重和偏置参数。为了惩罚权重向量的大小，我们必须以某种方式在损失函数中添加$\|\mathbf{w}\|^2$，但是模型应该如何平衡这个新的额外惩罚的损失？实际上，我们通过正则化常数λ来描述这种权衡，这是一个非负超参数，我们使用验证数据拟合：

$$L(\mathbf{w}, b) + \frac{\lambda}{2} \|\mathbf{w}\|^2, \tag{3.5.2}$$

对于$\lambda = 0$，我们恢复了原来的损失函数。对于$\lambda > 0$，我们限制$\|\mathbf{w}\|$的大小。我们仍然除以2：当我们取一个二次函数的导数时，2和1/2会抵消，以确保更新表达式看起来既漂亮又简单。聪明的读者可能会想知道为什么我们使用平方范数而不是标准范数（即欧几里得距离）。我们这样做是为了便于计算。通过平方L_2范数，我们去掉平方根，留下权重向量每个分量的平方和。这使得惩罚的导数很容易计算：导数的和等于和的导数。

此外，你可能会问为什么我们首先使用L_2范数，而不是L_1范数。事实上，这些选择在整个统计领域中都是有效的和受欢迎的。L_2正则化线性模型构成经典的岭回归（ridge regression）算法，L_1正则化线性回归是统计学中类似的基本模型，通常被称为套索回归（lasso regression）。

使用L_2范数的一个原因是它对权重向量的大分量施加了巨大的惩罚。这使得我们的学习算法偏向于在大量特征上均匀分布权重的模型。在实践中，这可能使它们对单个变量中的观测误差更为鲁棒。相比之下，L_1惩罚会导致模型将其他权重清除为零而将权重集中在一小部分特征上。这称为特征选择（feature selection），这可能是其他场景下需要的。

使用与 (2.1.10) 中的相同符号，L_2正则化回归的小批量随机梯度下降更新如下式：

$$\mathbf{w} \leftarrow (1 - \eta\lambda)\,\mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right). \tag{3.5.3}$$

根据之前章节所讲的，我们根据估计值与观测值之间的差异来更新$\mathbf{w}$。然而，我们同时也在试图将$\mathbf{w}$的大小缩小到零。这就是为什么这种方法有时被称为权重衰减。我们仅考虑惩罚项，优化算法在训练的每一步衰减权重。与特征选择相比，权重衰减为我们提供了一种连续的机制来调整函数的复杂度。较小的λ值对应较少约束的$\mathbf{w}$，而较大的λ值对$\mathbf{w}$的约束更大。

是否对相应的偏置b^2进行惩罚在不同的实现中会有所不同。在神经网络的不同层中也会有所不同。通常，我们不正则化网络输出层的偏置项。

3.5.2 高维线性回归

我们通过一个简单的例子来说明演示权重衰减。

```
%matplotlib inline
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

首先，我们像以前一样生成一些数据，生成公式如下：

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01^2). \tag{3.5.4}$$

我们选择标签是关于输入的线性函数。标签同时被均值为0，标准差为0.01高斯噪声破坏。为了使过拟合的效果更加明显，我们可以将问题的维数增加到$d = 200$，并使用一个只包含20个样本的小训练集。

```
n_train, n_test, num_inputs, batch_size = 20, 100, 200, 5
true_w, true_b = np.ones((num_inputs, 1)) * 0.01, 0.05
train_data = d2l.synthetic_data(true_w, true_b, n_train)
train_iter = d2l.load_array(train_data, batch_size)
test_data = d2l.synthetic_data(true_w, true_b, n_test)
test_iter = d2l.load_array(test_data, batch_size, is_train=False)
```

3.5.3 从零开始实现

在下面，我们将从头开始实现权重衰减，只需将L_2的平方惩罚添加到原始目标函数中。

初始化模型参数

首先，我们将定义一个函数来随机初始化我们的模型参数。

```
def init_params():
    w = np.random.normal(scale=1, size=(num_inputs, 1))
    b = np.zeros(1)
    w.attach_grad()
    b.attach_grad()
    return [w, b]
```

定义L_2范数惩罚

实现这一惩罚最方便的方法是对所有项求平方后并将它们求和。

```
def l2_penalty(w):
    return (w**2).sum() / 2
```

定义训练代码实现

下面的代码将模型拟合训练数据集，并在测试数据集上进行评估。从 2节 以来，线性网络和平方损失没有变化，所以我们通过`d2l.linreg`和`d2l.squared_loss`导入它们。唯一的变化是损失现在包括了惩罚项。

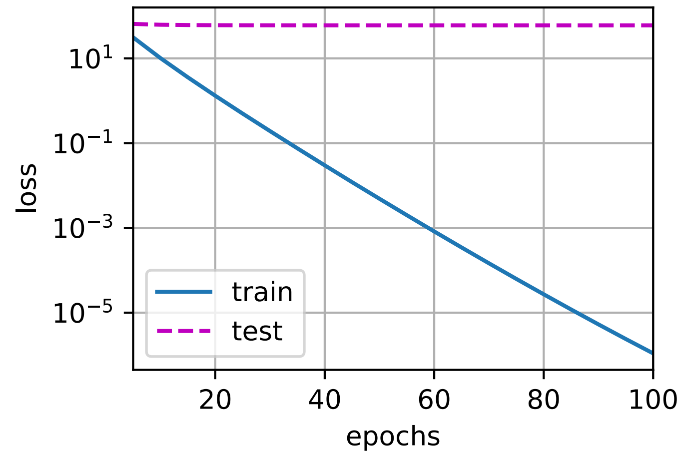
```
def train(lambd):
    w, b = init_params()
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    num_epochs, lr = 100, 0.003
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                # 增加了L2范数惩罚项，广播机制使l2_penalty(w)成为一个长度为`batch_size`的向量。
                l = loss(net(X), y) + lambd * l2_penalty(w)
            l.backward()
            d2l.sgd([w, b], lr, batch_size)
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('w的L2范数是：', np.linalg.norm(w))
```

不使用正则化进行训练

我们现在用`lambd = 0`禁用权重衰减后运行这个代码。注意，这里训练误差有了减少，但测试误差没有减少。这意味着出现了严重的过拟合。这是过拟合的一个典型例子。

```
train(lambd=0)
```

```
w的L2范数是： 13.259393
```

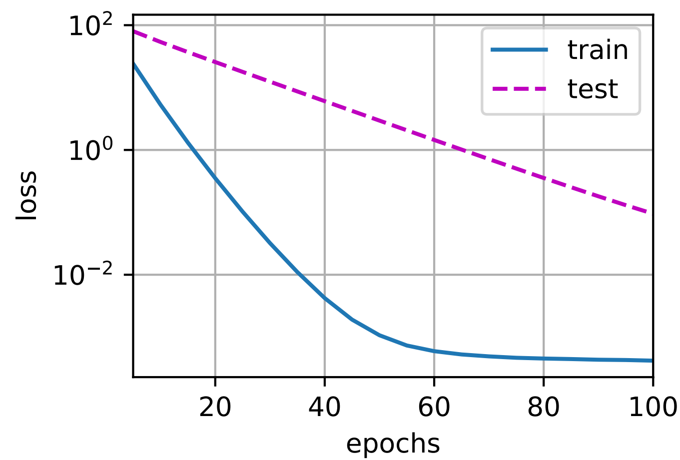


使用权重衰减

下面，我们使用权重衰减来运行代码。注意，在这里训练误差增大，但测试误差减小。这正是我们期望从正则化中得到的效果。

```
train(lambd=3)
```

```
w的L2范数是： 0.38248765
```



3.5.4 简洁实现

由于权重衰减在神经网络优化中很常用，深度学习框架为了便于使用权重衰减，便将权重衰减集成到优化算法中，以便与任何损失函数结合使用。此外，这种集成还有计算上的好处，允许在不增加任何额外的计算开销的情况下向算法中添加权重衰减。由于更新的权重衰减部分仅依赖于每个参数的当前值，因此优化器必须至少接触每个参数一次。

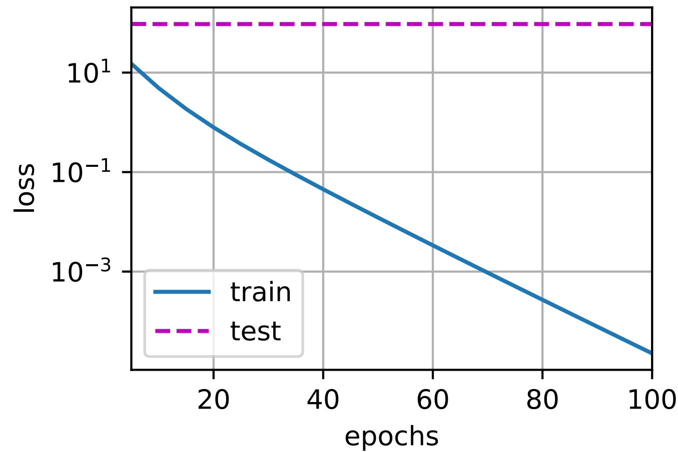
在下面的代码中，我们在实例化`Trainer`时直接通过wd指定weight decay超参数。默认情况下，Gluon同时衰减权重和偏置。注意，更新模型参数时，超参数wd将乘以`wd_mult`。因此，如果我们将`wd_mult`设置为零，则偏置参数b将不会被衰减。

```
def train_concise(wd):
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize(init.Normal(sigma=1))
    loss = gluon.loss.L2Loss()
    num_epochs, lr = 100, 0.003
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {
        'learning_rate': lr,
        'wd': wd})
    # 偏置参数没有衰减。偏置名称通常以“Bias”结尾
    net.collect_params('.*bias').setattr('wd_mult', 0)
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('w的L2范数：', np.linalg.norm(net[0].weight.data()))
```

这些图看起来和我们从零开始实现权重衰减时的图相同。然而，它们运行得更快，更容易实现，对于更复杂的问题，这一好处将变得更加明显。

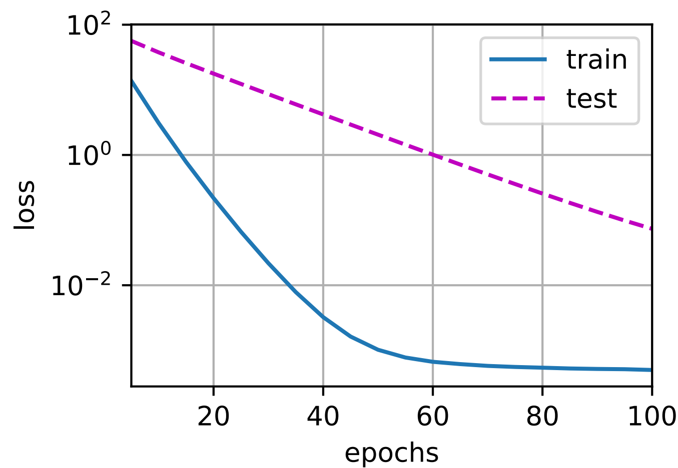
```
train_concise(0)
```

```
w的L2范数： 15.014068
```




```
train_concise(3)
```

```
w的L2范数:  0.3399133
```



到目前为止，我们只接触到一个简单线性函数的概念。此外，由什么构成一个简单的非线性函数可能是一个更复杂的问题。例如，再生核希尔伯特空间（RKHS）[59]允许在非线性环境中应用为线性函数引入的工具。不幸的是，基于RKHS的算法往往难以扩展到大型、高维的数据。在这本书中，我们将默认使用简单的启发式方法，即在深层网络的所有层上应用权重衰减。

[59] https://en.wikipedia.org/wiki/Reproducing_kernel_Hilbert_space

3.5.5 小结

- 正则化是处理过拟合的常用方法。在训练集的损失函数中加入惩罚项，以降低学习到的模型的复杂度。
- 保持模型简单的一个特别的选择是使用L_2惩罚的权重衰减。这会导致学习算法更新步骤中的权重衰减。
- 权重衰减功能在深度学习框架的优化器中提供。
- 在同一训练代码实现中，不同的参数集可以有不同的更新行为。

3.5.6 练习

1. 在本节的估计问题中使用λ的值进行实验。绘制训练和测试准确率关于λ的函数。你观察到了什么？
2. 使用验证集来找到最佳值λ。它真的是最优值吗？这有关系吗？
3. 如果我们使用$\sum_i |w_i|$作为我们选择的惩罚（L_1正则化），那么更新方程会是什么样子？
4. 我们知道$\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$。你能找到类似的矩阵方程吗（见 1.3.10节 中的弗罗贝尼乌斯范数）？
5. 回顾训练误差和泛化误差之间的关系。除了权重衰减、增加训练数据、使用适当复杂度的模型之外，你还能想出其他什么方法来处理过拟合？
6. 在贝叶斯统计中，我们使用先验和似然的乘积，通过公式$P(w \mid x) \propto P(x \mid w)P(w)$得到后验。如何得到带正则化的$P(w)$？

Discussions[60]

3.6 Dropout

在 3.5节 中，我们介绍了通过惩罚权重的L_2范数来正则化统计模型的经典方法。在概率角度看，我们可以通过以下论证来证明这一技术的合理性：我们已经假设了一个先验，即权重的值取自均值为0的高斯分布。更直观的是，我们可能会说，我们鼓励模型将其权重分散到许多特征中，而不是过于依赖少数潜在的虚假关联。

3.6.1 重新审视过拟合

当面对更多的特征而样本不足时，线性模型往往会过度拟合。当给出更多样本而不是特征，我们通常可以指望线性模型不会过拟合。不幸的是，线性模型泛化的可靠性是有代价的。简单地说，线性模型没有考虑到特征之间的交互作用。对于每个特征，线性模型必须指定正的或负的权重，而忽略上下文。

在传统说法中，泛化性和灵活性之间的这种基本权衡被描述为偏差-方差权衡（bias-variance tradeoff）。线性模型有很高的偏差：它们只能表示一小类函数。然而，这些模型的方差很低：它们在不同的随机数据样本上给出了相似的结果。

[60] https://discuss.d2l.ai/t/1810

深度神经网络位于偏差-方差谱的另一端。与线性模型不同，神经网络并不局限于单独查看每个特征。它们可以学习特征之间的交互。例如，它们可能推断“尼日利亚”和“西联汇款”一起出现在电子邮件中表示垃圾邮件，但单独出现则不表示垃圾邮件。

即使我们有比特征多得多的样本，深度神经网络也有可能过拟合。2017年，一组研究人员通过在随机标记的图像上训练深度网络。这展示了神经网络的极大灵活性。因为没有任何真实的模式将输入和输出联系起来，但他们发现，通过随机梯度下降优化的神经网络可以完美地标记训练集中的每一幅图像。想一想这意味着什么。如果标签是随机均匀分配的，并且有10个类别，那么在保留数据上没有分类器会取得高于10%的准确率。这里的泛化差距高达90%。如果我们的模型具有这么强的表达能力，以至于它们可以如此严重地过拟合，那么我们指望在什么时候它们不会过拟合呢？

深度网络有着令人费解的泛化性质，而这种泛化性质的数学基础仍然是悬而未决的研究问题，我们鼓励面向理论的读者更深入地研究这个主题。目前，我们转向对实际工具的探究，这些工具倾向于经验上改进深层网络的泛化性。

3.6.2 扰动的鲁棒性

让我们简单地思考一下我们对一个好的预测模型的期待。我们期待它能在看不见的数据上有很好的表现。经典泛化理论认为，为了缩小训练和测试性能之间的差距，我们应该以简单的模型为目标。简单性以较小维度的形式出现。我们在 3.4节 讨论线性模型的单项式函数时探讨了这一点。此外，正如我们在 3.5节 中讨论权重衰减（L_2正则化）时看到的那样，参数的范数也代表了一种有用的简单性度量。简单性的另一个有用角度是平滑性，即函数不应该对其输入的微小变化敏感。例如，当我们对图像进行分类时，我们预计向像素添加一些随机噪声应该是基本无影响的。

1995年，克里斯托弗·毕晓普证明了具有输入噪声的训练等价于Tikhonov正则化 [Bishop, 1995]，从而将这一观点正式化。这项工作在要求函数光滑(因而简单)和要求它对输入中的扰动具有适应性之间有了明确的数学联系。

然后，在2014年，斯里瓦斯塔瓦等人 [Srivastava et al., 2014] 还就如何将毕晓普的想法应用于网络的内部层提出了一个聪明的想法。在训练过程中，他们建议在计算后续层之前向网络的每一层注入噪声。他们意识到，当训练一个有多层的深层网络时，注入噪声只会在输入-输出映射上增强平滑性。

他们的想法被称为丢弃法（dropout），dropout在正向传播过程中，计算每一内部层的同时注入噪声，这已经成为训练神经网络的标准技术。这种方法之所以被称为 *dropout* ，因为我们从表面上看是在训练过程中丢弃（drop out）一些神经元。在整个训练过程的每一次迭代中，dropout包括在计算下一层之前将当前层中的一些节点置零。

需要说明的是，我们将自己的叙述与毕晓普联系起来。关于dropout的原始论文出人意料地通过一个有性繁殖类比提供了直觉。作者认为，神经网络过拟合的特征是每一层都依赖于前一层激活值的特定模式，称这种情况为“共适应性”。他们声称，dropout会破坏共适应性，就像有性生殖会破坏共适应的基因一样。

那么关键的挑战就是如何注入这种噪声。一种想法是以一种无偏的方式注入噪声。这样在固定住其他层时，每一层的期望值等于没有噪音时的值。

在毕晓普的工作中，他将高斯噪声添加到线性模型的输入中。在每次训练迭代中，他将从均值为零的分布$\epsilon \sim \mathcal{N}(0,\sigma^2)$采样噪声添加到输入$\mathbf{x}$，从而产生扰动点$\mathbf{x}' = \mathbf{x} + \epsilon$。预期是$E[\mathbf{x}'] = \mathbf{x}$。

在标准dropout正则化中，通过按保留（未丢弃）的节点的分数进行归一化来消除每一层的偏差。换言之，每个中间激活值h以丢弃概率p由随机变量h'替换，如下所示：

$$h' = \begin{cases} 0 & \text{概率为} p \\ \frac{h}{1-p} & \text{其他情况} \end{cases} \tag{3.6.1}$$

根据设计，期望值保持不变，即$E[h'] = h$。

3.6.3 实践中的dropout

回想一下 图3.1.1 中带有一个隐藏层和5个隐藏单元的多层感知机。当我们将dropout应用到隐藏层，以p的概率将隐藏单元置为零时，结果可以看作是一个只包含原始神经元子集的网络。在 图3.6.1 中，删除了h_2和h_5。因此，输出的计算不再依赖于h_2或h_5，并且它们各自的梯度在执行反向传播时也会消失。这样，输出层的计算不能过度依赖于$h_1, \ldots, h_5$的任何一个元素。

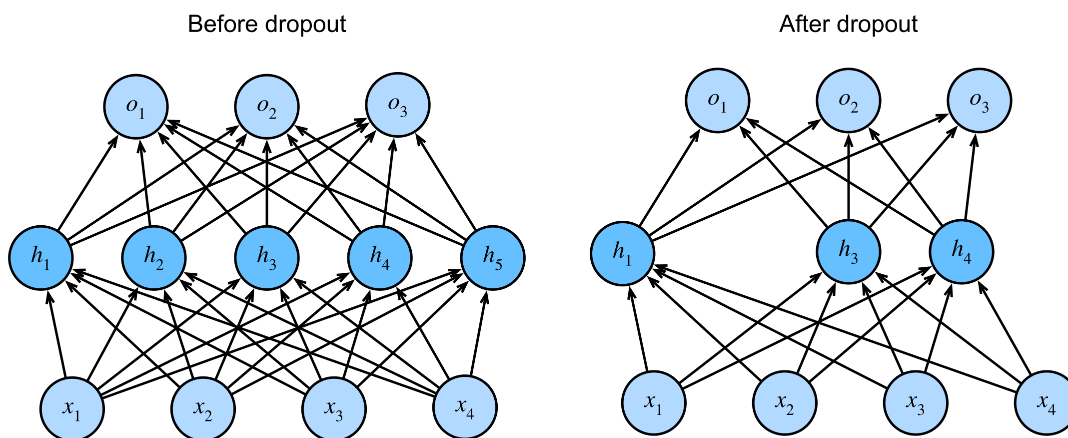


图3.6.1: dropout前后的多层感知机。

通常，我们在测试时禁用dropout。给定一个训练好的模型和一个新的样本，我们不会丢弃任何节点，因此不需要标准化。然而，也有一些例外：一些研究人员使用测试时的dropout作为估计神经网络预测的"不确定性"的启发式方法：如果预测在许多不同的dropout掩码上都是一致的，那么我们可以说网络更有自信心。

3.6.4 从零开始实现

要实现单层的dropout函数，我们必须从伯努利（二元）随机变量中提取与我们的层的维度一样多的样本，其中随机变量以概率$1-p$取值1（保持），以概率p取值0（丢弃）。实现这一点的一种简单方式是首先从均匀分布$U[0, 1]$中抽取样本。那么我们可以保留那些对应样本大于p的节点，把剩下的丢弃。

在下面的代码中，我们实现了一个`dropout_layer`函数，该函数以`dropout`的概率丢弃张量输入`X`中的元素，如上所述重新缩放剩余部分：将剩余部分除以`1.0-dropout`。

```python
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()

def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    # 在本情况中，所有元素都被丢弃。
    if dropout == 1:
        return np.zeros_like(X)
    # 在本情况中，所有元素都被保留。
    if dropout == 0:
        return X
    mask = np.random.uniform(0, 1, X.shape) > dropout
    return mask.astype(np.float32) * X / (1.0 - dropout)
```

我们可以通过几个例子来测试dropout_layer函数。在下面的代码行中，我们将输入X通过dropout操作，丢弃概率分别为0、0.5和1。

```python
X = np.arange(16).reshape(2, 8)
print(dropout_layer(X, 0))
print(dropout_layer(X, 0.5))
print(dropout_layer(X, 1))
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.]
 [ 8.  9. 10. 11. 12. 13. 14. 15.]]
[[ 0.  2.  4.  6.  8. 10. 12. 14.]
 [ 0. 18. 20.  0.  0.  0. 28.  0.]]
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

定义模型参数

同样，我们使用 2.5节 中引入的Fashion-MNIST数据集。我们定义具有两个隐藏层的多层感知机，每个隐藏层包含256个单元。

```python
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256

W1 = np.random.normal(scale=0.01, size=(num_inputs, num_hiddens1))
b1 = np.zeros(num_hiddens1)
W2 = np.random.normal(scale=0.01, size=(num_hiddens1, num_hiddens2))
b2 = np.zeros(num_hiddens2)
```

(continues on next page)

(continued from previous page)

```
W3 = np.random.normal(scale=0.01, size=(num_hiddens2, num_outputs))
b3 = np.zeros(num_outputs)

params = [W1, b1, W2, b2, W3, b3]
for param in params:
    param.attach_grad()
```

定义模型

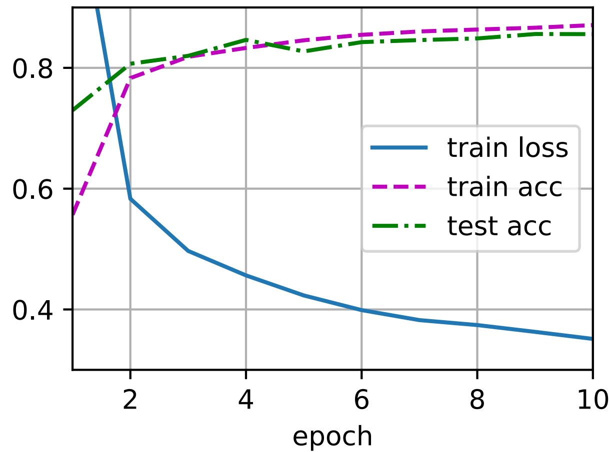
下面的模型将dropout应用于每个隐藏层的输出（在激活函数之后）。我们可以分别为每一层设置丢弃概率。一种常见的技巧是在靠近输入层的地方设置较低的丢弃概率。下面，我们将第一个和第二个隐藏层的丢弃概率分别设置为0.2和0.5。我们确保dropout只在训练期间有效。

```
dropout1, dropout2 = 0.2, 0.5

def net(X):
    X = X.reshape(-1, num_inputs)
    H1 = npx.relu(np.dot(X, W1) + b1)
    # 只有在训练模型时才使用dropout
    if autograd.is_training():
        # 在第一个全连接层之后添加一个dropout层
        H1 = dropout_layer(H1, dropout1)
    H2 = npx.relu(np.dot(H1, W2) + b2)
    if autograd.is_training():
        # 在第二个全连接层之后添加一个dropout层
        H2 = dropout_layer(H2, dropout2)
    return np.dot(H2, W3) + b3
```

训练和测试

这类似于前面描述的多层感知机训练和测试。

```
num_epochs, lr, batch_size = 10, 0.5, 256
loss = gluon.loss.SoftmaxCrossEntropyLoss()
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
              lambda batch_size: d2l.sgd(params, lr, batch_size))
```



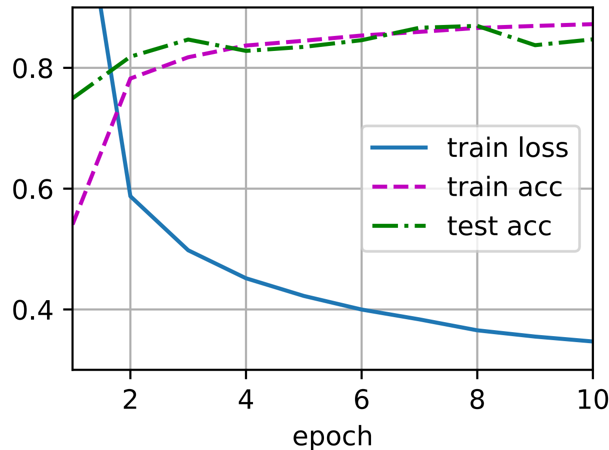
3.6.5 简洁实现

对于高级API，我们所需要做的就是在每个全连接层之后添加一个Dropout层，将丢弃概率作为唯一的参数传递给它的构造函数。在训练过程中，Dropout层将根据指定的丢弃概率随机丢弃上一层的输出（相当于下一层的输入）。当不处于训练模式时，Dropout层仅在测试时传递数据。

```
net = nn.Sequential()
net.add(nn.Dense(256, activation="relu"),
        # 在第一个全连接层之后添加一个dropout层
        nn.Dropout(dropout1), nn.Dense(256, activation="relu"),
        # 在第二个全连接层之后添加一个dropout层
        nn.Dropout(dropout2), nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

接下来，我们对模型进行训练和测试。

```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



3.6.6 小结

- 除了控制权重向量的维数和大小之外，dropout也是避免过拟合的另一种工具。它们通常是联合使用的。
- dropout将激活值h替换为具有期望值h的随机变量。
- dropout仅在训练期间使用。

3.6.7 练习

1. 如果更改第一层和第二层的dropout概率，会发生什么情况？具体地说，如果交换这两个层，会发生什么情况？设计一个实验来回答这些问题，定量描述你的结果，并总结定性的结论。
2. 增加迭代周期数，并将使用dropout和不使用dropout时获得的结果进行比较。
3. 当应用或不应用dropout时，每个隐藏层中激活值的方差是多少？绘制一个曲线图，以显示这两个模型的每个隐藏层中激活值的方差是如何随时间变化的。
4. 为什么在测试时通常不使用dropout？
5. 以本节中的模型为例，比较使用dropout和权重衰减的效果。如果同时使用dropout和权重衰减，会发生什么情况？结果是累加的吗？收益是否减少（或者说更糟）？它们互相抵消了吗？
6. 如果我们将dropout应用到权重矩阵的各个权重，而不是激活值，会发生什么？
7. 发明另一种用于在每一层注入随机噪声的技术，该技术不同于标准的dropout技术。你能否开发一种在Fashion-MNIST数据集(对于固定结构)上性能优于dropout的方法？

Discussions[61]

[61] https://discuss.d2l.ai/t/1812

3.7 正向传播、反向传播和计算图

到目前为止，我们已经用小批量随机梯度下降训练了我们的模型。然而，当我们实现该算法时，我们只考虑了通过模型 正向传播（forward propagation）所涉及的计算。在计算梯度时，我们只调用了深度学习框架提供的反向传播函数。

梯度的自动计算（自动微分）大大简化了深度学习算法的实现。在自动微分之前，即使是对复杂模型的微小调整也需要手工重新计算复杂的导数。学术论文也不得不分配大量页面来推导更新规则。我们必须继续依赖于自动微分，这样我们就可以专注于有趣的部分，但是如果你想超过对深度学习的浅薄理解，你应当知道这些梯度是如何计算出来的。

在本节中，我们将深入探讨 反向传播（backward propagation 或 backpropagation）的细节。为了传达对这些技术及其实现的一些见解，我们依赖一些基本的数学和计算图。首先，我们将重点放在带权重衰减（L_2正则化）的单隐藏层多层感知机上。

3.7.1 正向传播

正向传播（forward propagation或forward pass）指的是：按顺序（从输入层到输出层）计算和存储神经网络中每层的结果。

我们将一步步研究单隐藏层神经网络的机制，为了简单起见，我们假设输入样本是 $\mathbf{x} \in \mathbb{R}^d$，并且我们的隐藏层不包括偏置项。这里的中间变量是：

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}, \tag{3.7.1}$$

其中$\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$是隐藏层的权重参数。然后将中间变量$\mathbf{z} \in \mathbb{R}^h$通过激活函数$\phi$后，我们得到长度为$h$的隐藏激活向量：

$$\mathbf{h} = \phi(\mathbf{z}). \tag{3.7.2}$$

隐藏变量$\mathbf{h}$也是一个中间变量。假设输出层的参数只有权重$\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$，我们可以得到输出层变量，它是一个长度为$q$的向量：

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \tag{3.7.3}$$

假设损失函数为l，样本标签为y，我们可以计算单个数据样本的损失项，

$$L = l(\mathbf{o}, y). \tag{3.7.4}$$

根据L_2正则化的定义，给定超参数λ，正则化项为

$$s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right), \tag{3.7.5}$$

其中，矩阵的弗罗贝尼乌斯范数是将矩阵展平为向量后应用的L_2范数。最后，模型在给定数据样本上的正则化损失为：

$$J = L + s. \tag{3.7.6}$$

在下面的讨论中，我们将J称为目标函数。

3.7.2 正向传播计算图

绘制计算图有助于我们可视化计算中操作符和变量的依赖关系。图3.7.1 是与上述简单网络相对应的计算图，其中正方形表示变量，圆圈表示操作符。左下角表示输入，右上角表示输出。注意显示数据流的箭头方向主要是向右和向上的。

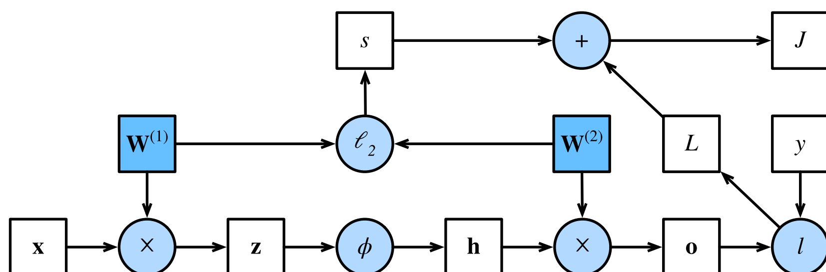


图3.7.1: 正向传播的计算图

3.7.3 反向传播

反向传播指的是计算神经网络参数梯度的方法。简言之，该方法根据微积分中的链式规则，按相反的顺序从输出层到输入层遍历网络。该算法存储了计算某些参数梯度时所需的任何中间变量（偏导数）。假设我们有函数$\mathsf{Y} = f(\mathsf{X})$和$\mathsf{Z} = g(\mathsf{Y})$，其中输入和输出$\mathsf{X}, \mathsf{Y}, \mathsf{Z}$是任意形状的张量。利用链式法则，我们可以计算$\mathsf{Z}$关于$\mathsf{X}$的导数

$$\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \text{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}}\right). \tag{3.7.7}$$

在这里，我们使用prod运算符在执行必要的操作（如换位和交换输入位置）后将其参数相乘。对于向量，这很简单：它只是矩阵-矩阵乘法。对于高维张量，我们使用适当的对应项。运算符prod指代了所有的这些符号。

回想一下，在计算图 图3.7.1 中的单隐藏层简单网络的参数是$\mathbf{W}^{(1)}$和$\mathbf{W}^{(2)}$。反向传播的目的是计算梯度$\partial J/\partial \mathbf{W}^{(1)}$和$\partial J/\partial \mathbf{W}^{(2)}$。为此，我们应用链式法则，依次计算每个中间变量和参数的梯度。计算的顺序与正向传播中执行的顺序相反，因为我们需要从计算图的结果开始，并朝着参数的方向努力。第一步是计算目标函数$J = L + s$相对于损失项L和正则项s的梯度。

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1. \tag{3.7.8}$$

接下来，我们根据链式法则计算目标函数关于输出层变量$\mathbf{o}$的梯度：

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q. \tag{3.7.9}$$

接下来，我们计算正则化项相对于两个参数的梯度：

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}. \tag{3.7.10}$$

现在我们可以计算最接近输出层的模型参数的梯度$\partial J/\partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$。使用链式法则得出：

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}. \tag{3.7.11}$$

为了获得关于$\mathbf{W}^{(1)}$的梯度，我们需要继续沿着输出层到隐藏层反向传播。关于隐藏层输出的梯度$\partial J/\partial \mathbf{h} \in \mathbb{R}^h$由下式给出：

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = {\mathbf{W}^{(2)}}^\top \frac{\partial J}{\partial \mathbf{o}}. \tag{3.7.12}$$

由于激活函数ϕ是按元素计算的，计算中间变量$\mathbf{z}$的梯度$\partial J/\partial \mathbf{z} \in \mathbb{R}^h$需要使用按元素乘法运算符，我们用$\odot$表示：

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'\left(\mathbf{z}\right). \tag{3.7.13}$$

最后，我们可以得到最接近输入层的模型参数的梯度$\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$。根据链式法则，我们得到：

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}. \tag{3.7.14}$$

3.7.4 训练神经网络

在训练神经网络时，正向传播和后向传播相互依赖。对于正向传播，我们沿着依赖的方向遍历计算图并计算其路径上的所有变量。然后将这些用于反向传播，其中计算顺序与计算图的相反。

以上述简单网络为例进行说明。一方面，在正向传播期间计算正则项 (3.7.5) 取决于模型参数$\mathbf{W}^{(1)}$和$\mathbf{W}^{(2)}$的当前值。它们是由优化算法根据最近迭代的反向传播给出的。另一方面，反向传播期间参数 (3.7.11) 的梯度计算取决于由正向传播给出的隐藏变量$\mathbf{h}$的当前值。

因此，在训练神经网络时，在初始化模型参数后，我们交替使用正向传播和反向传播，利用反向传播给出的梯度来更新模型参数。注意，反向传播复用正向传播中存储的中间值，以避免重复计算。带来的影响之一是我们需要保留中间值，直到反向传播完成。这也是为什么训练比单纯的预测需要更多的内存（显存）的原因之一。此外，这些中间值的大小与网络层的数量和批量的大小大致成正比。因此，使用更大的批量来训练更深层次的网络更容易导致内存（显存）不足（out of memory）错误。

3.7.5 小结

- 正向传播在神经网络定义的计算图中按顺序计算和存储中间变量。它的顺序是从输入层到输出层。
- 反向传播按相反的顺序计算和存储神经网络的中间变量和参数的梯度。
- 在训练深度学习模型时，正向传播和反向传播是相互依赖的。
- 训练比预测需要更多的内存（显存）。

3.7.6 练习

1. 假设一些标量函数$\mathbf{X}$的输入$\mathbf{X}$是$n\times m$矩阵。f相对于$\mathbf{X}$的梯度维数是多少?
2. 向本节中描述的模型的隐藏层添加偏置项（不需要在正则化项中包含偏置项)。
    1. 画出相应的计算图。
    2. 推导正向和后向传播方程。
3. 计算本节所描述的模型，用于训练和预测的内存占用。
4. 假设你想计算二阶导数。计算图发生了什么？你预计计算需要多长时间?
5. 假设计算图对于你的GPU来说太大了。
    1. 你能把它划分到多个GPU上吗?
    2. 与小批量训练相比，有哪些优点和缺点?

Discussions[62]

3.8 数值稳定性和模型初始化

到目前为止，我们实现的每个模型都是根据某个预先指定的分布来初始化模型的参数。直到现在，我们认为初始化方案是理所当然的，忽略了如何做出这些选择的细节。你甚至可能会觉得，初始化方案的选择并不是特别重要。相反，初始化方案的选择在神经网络学习中起着非常重要的作用，它对保持数值稳定性至关重要。此外，这些选择可以与非线性激活函数的选择以有趣的方式结合在一起。我们选择哪个函数以及如何初始化参数可以决定优化算法收敛的速度有多快。糟糕选择可能会导致我们在训练时遇到梯度爆炸或梯度消失。在本节中，我们将更详细地探讨这些主题，并讨论一些有用的启发式方法。你会发现这些启发式方法在你的整个深度学习生涯中都很有用。

3.8.1 梯度消失和梯度爆炸

考虑一个具有L层、输入$\mathbf{x}$和输出$\mathbf{o}$的深层网络。每一层l由变换f_l定义，该变换的参数为权重$\mathbf{W}^{(l)}$，其隐藏变量是$\mathbf{h}^{(l)}$（令 $\mathbf{h}^{(0)}=\mathbf{x}$)。我们的网络可以表示为:

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ 因此} \mathbf{o} = f_L \circ \ldots \circ f_1(\mathbf{x}). \tag{3.8.1}$$

如果所有隐藏变量和输入都是向量，我们可以将$\mathbf{o}$关于任何一组参数$\mathbf{W}^{(l)}$的梯度写为下式:

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{ \mathbf{M}^{(L)} \stackrel{\mathrm{def}}{=}} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{ \mathbf{M}^{(l+1)} \stackrel{\mathrm{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{ \mathbf{v}^{(l)} \stackrel{\mathrm{def}}{=}}. \tag{3.8.2}$$

换言之，该梯度是$L-l$个矩阵$\mathbf{M}^{(L)}\cdot\ldots\cdot\mathbf{M}^{(l+1)}$与梯度向量 $\mathbf{v}^{(l)}$的乘积。因此，我们容易受到数值下溢问题的影响，当将太多的概率乘在一起时，这些问题经常会出现。在处理概率时，一个常见的技巧是切换到对数空

[62] https://discuss.d2l.ai/t/1816

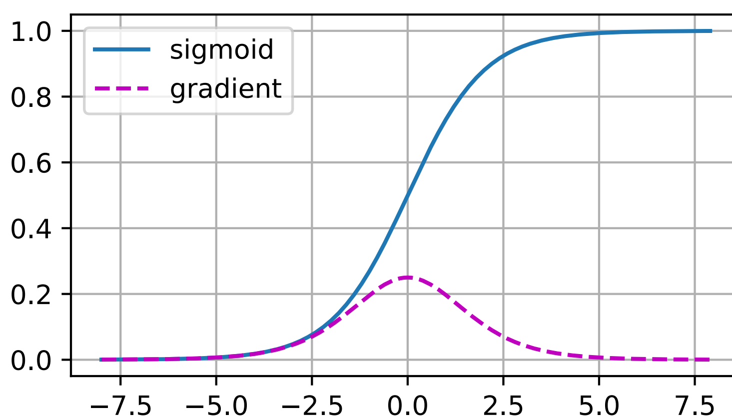
间，即将数值表示的压力从尾数转移到指数。不幸的是，我们上面的问题更为严重：最初，矩阵 $\mathbf{M}^{(l)}$ 可能具有各种各样的特征值。他们可能很小，也可能很大，他们的乘积可能非常大，也可能非常小。

不稳定梯度带来的风险不止在于数值表示。不稳定梯度也威胁到我们优化算法的稳定性。我们可能面临一些问题。要么是 梯度爆炸（gradient exploding）问题：参数更新过大，破坏了模型的稳定收敛；要么是 梯度消失（gradient vanishing）问题：参数更新过小，在每次更新时几乎不会移动，导致无法学习。

梯度消失

导致梯度消失问题的一个常见的原因是跟在每层的线性运算之后的激活函数σ。从历史上看，sigmoid函数$1/(1+\exp(-x))$（3.1节 提到过）很流行，因为它类似于阈值函数。由于早期的人工神经网络受到生物神经网络的启发，神经元要么完全激活要么完全不激活（就像生物神经元）的想法很有吸引力。让我们仔细看看sigmoid函数为什么会导致梯度消失。

```
%matplotlib inline
from mxnet import autograd, np, npx
from d2l import mxnet as d2l

npx.set_np()

x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.sigmoid(x)
y.backward()

d2l.plot(x, [y, x.grad], legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



正如你所看到的，当它的输入很大或是很小时，sigmoid函数的梯度都会消失。此外，当反向传播通过许多层时，除非我们在刚刚好的地方，这些地方sigmoid函数的输入接近于零，否则整个乘积的梯度可能会消失。当我们的网络有很多层时，除非我们很小心，否则在某一层可能会切断梯度。事实上，这个问题曾经困扰着

深度网络的训练。因此，更稳定（但在神经科学的角度看起来不太合理）的ReLU系列函数已经成为从业者的默认选择。

梯度爆炸

相反的问题，当梯度爆炸时，可能同样令人烦恼。为了更好地说明这一点，我们生成100个高斯随机矩阵，并将它们与某个初始矩阵相乘。对于我们选择的尺度（方差$\sigma^2 = 1$），矩阵乘积发生爆炸。当这种情况是由于深度网络的初始化所导致时，我们没有机会让梯度下降优化器收敛。

```
M = np.random.normal(size=(4, 4))
print('一个矩阵 \n', M)
for i in range(100):
    M = np.dot(M, np.random.normal(size=(4, 4)))

print('乘以100个矩阵后\n', M)
```

```
一个矩阵
 [[ 2.2122064   1.1630787   0.7740038   0.4838046 ]
 [ 1.0434405   0.29956347  1.1839255   0.15302546]
 [ 1.8917114  -1.1688148  -1.2347414   1.5580711 ]
 [-1.771029   -0.5459446  -0.45138445 -2.3556297 ]]
乘以100个矩阵后
 [[ 3.4459714e+23 -7.8040680e+23  5.9973287e+23  4.5229990e+23]
 [ 2.5275089e+23 -5.7240326e+23  4.3988473e+23  3.3174740e+23]
 [ 1.3731286e+24 -3.1097155e+24  2.3897773e+24  1.8022959e+24]
 [-4.4951040e+23  1.0180033e+24 -7.8232281e+23 -5.9000354e+23]]
```

打破对称性

神经网络设计中的另一个问题是其参数化所固有的对称性。假设我们有一个简单的多层感知机，它有一个隐藏层和两个隐藏单元。在这种情况下，我们可以对第一层的权重$\mathbf{W}^{(1)}$进行重排列，并且同样对输出层的权重进行重排列，可以获得相同的函数。第一个隐藏单元与第二个隐藏单元没有什么特别的区别。换句话说，我们在每一层的隐藏单元之间具有排列对称性。

这不仅仅是理论上的麻烦。考虑前述具有两个隐藏单元的单隐藏层多层感知机。为便于说明，假设输出层将两个隐藏单元转换为仅一个输出单元。想象一下，如果我们将隐藏层的所有参数初始化为$\mathbf{W}^{(1)} = c$，c为常量，会发生什么情况。在这种情况下，在正向传播期间，两个隐藏单元采用相同的输入和参数，产生相同的激活，该激活被送到输出单元。在反向传播期间，根据参数$\mathbf{W}^{(1)}$对输出单元进行微分，得到一个梯度，其元素都取相同的值。因此，在基于梯度的迭代(例如，小批量随机梯度下降)之后，$\mathbf{W}^{(1)}$的所有元素仍然采用相同的值。这样的迭代永远不会打破对称性，我们可能永远也无法实现网络的表达能力。隐藏层的行为就好像只有一个单元。请注意，虽然小批量随机梯度下降不会打破这种对称性，但dropout正则化可以。

3.8.2 参数初始化

解决（或至少减轻）上述问题的一种方法是仔细地进行初始化。优化期间的注意和适当的正则化可以进一步提高稳定性。

默认初始化

在前面的部分中，例如在 2.3节 中，我们使用正态分布来初始化权重值。如果我们不指定初始化方法，框架将使用默认的随机初始化方法，对于中等规模的问题，这种方法通常很有效。

Xavier初始化

让我们看看某些没有非线性的全连接层输出(例如，隐藏变量)o_i的尺度分布。对于该层n_in输入x_j及其相关权重w_{ij}，输出由下式给出

$$o_i = \sum_{j=1}^{n_\mathrm{in}} w_{ij} x_j. \tag{3.8.3}$$

权重w_{ij}都是从同一分布中独立抽取的。此外，让我们假设该分布具有零均值和方差σ^2。请注意，这并不意味着分布必须是高斯的，只是均值和方差需要存在。现在，让我们假设层x_j的输入也具有零均值和方差γ^2，并且它们独立于w_{ij}并且彼此独立。在这种情况下，我们可以按如下方式计算o_i的平均值和方差：

$$\begin{aligned}
E[o_i] & = \sum_{j=1}^{n_\mathrm{in}} E[w_{ij} x_j] \\
& = \sum_{j=1}^{n_\mathrm{in}} E[w_{ij}] E[x_j] \\
& = 0, \\
\mathrm{Var}[o_i] & = E[o_i^2] - (E[o_i])^2 \\
& = \sum_{j=1}^{n_\mathrm{in}} E[w^2_{ij} x^2_j] - 0 \\
& = \sum_{j=1}^{n_\mathrm{in}} E[w^2_{ij}] E[x^2_j] \\
& = n_\mathrm{in} \sigma^2 \gamma^2.
\end{aligned} \tag{3.8.4}$$

保持方差不变的一种方法是设置$n_\mathrm{in} \sigma^2 = 1$。现在考虑反向传播过程，我们面临着类似的问题，尽管梯度是从更靠近输出的层传播的。使用与正向传播相同的推理，我们可以看到，除非$n_\mathrm{out} \sigma^2 = 1$，否则梯度的方差可能会增大，其中$n_\mathrm{out}$是该层的输出的数量。这使我们进退两难：我们不可能同时满足这两个条件。相反，我们只需满足：

$$\frac{1}{2} (n_\mathrm{in} + n_\mathrm{out}) \sigma^2 = 1 \text{ 或等价于 } \sigma = \sqrt{\frac{2}{n_\mathrm{in} + n_\mathrm{out}}}. \tag{3.8.5}$$

这就是现在标准且实用的*Xavier*初始化的基础，它以其提出者 [Glorot & Bengio, 2010] 第一作者的名字命名。通常，Xavier初始化从均值为零，方差$\sigma^2 = \frac{2}{n_\mathrm{in} + n_\mathrm{out}}$的高斯分布中采样权重。我们也可以利用Xavier的直觉来

选择从均匀分布中抽取权重时的方差。注意均匀分布$U(-a, a)$的方差为$\frac{a^2}{3}$。将$\frac{a^2}{3}$代入到σ^2的条件中，将得到初始化的建议：

$$U\left(-\sqrt{\frac{6}{n_\mathrm{in} + n_\mathrm{out}}}, \sqrt{\frac{6}{n_\mathrm{in} + n_\mathrm{out}}}\right). \tag{3.8.6}$$

尽管上述数学推理中，不存在非线性的假设在神经网络中很容易被违反，但Xavier初始化方法在实践中被证明是有效的。

额外内容

上面的推理仅仅触及了现代参数初始化方法的皮毛。深度学习框架通常实现十几种不同的启发式方法。此外，参数初始化一直是深度学习基础研究的热点领域。其中包括专门用于参数绑定(共享)、超分辨率、序列模型和其他情况的启发式算法。例如，Xiao等人演示了通过使用精心设计的初始化方法 [Xiao et al., 2018] ，可以无需结构上的技巧而训练10000层神经网络的可能性。

如果你对该主题感兴趣，我们建议你深入研究本模块的内容，阅读提出并分析每种启发式方法的论文，然后探索有关该主题的最新出版物。也许你会偶然发现甚至发明一个聪明的想法，并为深度学习框架提供一个实现。

3.8.3 小结

- 梯度消失和爆炸是深度网络中常见的问题。在参数初始化时需要非常小心，以确保梯度和参数可以得到很好的控制。
- 需要用启发式的初始化方法来确保初始梯度既不太大也不太小。
- ReLU激活函数缓解了梯度消失问题，这样可以加速收敛。
- 随机初始化是保证在进行优化前打破对称性的关键。
- Xavier初始化表明，对于每一层，输出的方差不受输入数量的影响，任何梯度的方差不受输出数量的影响。

3.8.4 练习

1. 除了多层感知机的排列对称性之外，你能设计出其他神经网络可能会表现出对称性且需要被打破的情况吗？
2. 我们是否可以将线性回归或softmax回归中的所有权重参数初始化为相同的值？
3. 在相关资料中查找两个矩阵乘积特征值的解析界。这对确保梯度条件合适有什么启示？
4. 如果我们知道某些项是发散的，我们能在事后修正吗？看看关于分层自适应速率缩放的论文 [You et al., 2017] 。

Discussions[63]

[63] https://discuss.d2l.ai/t/1819

3.9 环境和分布偏移

在前面的部分中，我们学习了机器学习的许多实际应用，将模型拟合各种数据集。然而，我们从来没有思考数据最初从哪里来，或者我们计划最终如何处理模型的输出。通常情况下，拥有数据的机器学习开发人员急于开发模型，而不停下来考虑这些基本问题。

许多失败的机器学习部署都可以追溯到这种方式。有时，根据测试集的准确度衡量，模型表现得非常出色，但是当数据分布突然改变时，模型在部署中会出现灾难性的失败。更隐蔽的是，有时模型的部署本身就是扰乱数据分布的催化剂。例如，我们训练了一个模型来预测谁将偿还贷款或违约，发现申请人选择的鞋子与违约风险相关（牛津鞋表示偿还，运动鞋表示违约）。此后，我们可能倾向于向所有穿着牛津鞋的申请人发放贷款，并拒绝所有穿着运动鞋的申请人。

在这种情况下，在这种情况下，我们从模式识别到决策的未经深思熟虑地跳跃，以及我们未能批判性地考虑环境可能会带来灾难性的后果。首先，一旦我们开始根据鞋类做出决定，顾客就会理解并改变他们的行为。不久，所有的申请者都会穿牛津鞋，而信用度却没有相应的提高。花点时间来理解这一点，因为机器学习的许多应用中都存在类似的问题：通过将基于模型的决策引入环境，我们可能会破坏模型。

虽然我们不可能在一节中完整地讨论这些主题，但我们的目的是揭示一些常见的问题，并激发必要的批判性思考，以便及早发现这些情况，减轻损害，并负责任地使用机器学习。有些解决方案很简单（要求"正确"的数据），有些在技术上很困难（实施强化学习系统），还有一些解决方案要求我们完全跳出统计预测的领域，努力解决与算法的伦理应用有关的棘手哲学问题。

3.9.1 分布偏移的类型

首先，我们坚持使用被动预测设置，考虑到数据分布可能发生变化的各种方式，以及为挽救模型性能可能采取的措施。在一个经典的设置中，我们假设我们的训练数据是从某个分布$p_S(\mathbf{x}, y)$中采样的，但是我们的测试数据将包含从不同分布$p_T(\mathbf{x}, y)$中抽取的未标记样本。我们必须面对一个清醒的现实。如果没有任何关于p_S和p_T之间相互关系的假设，学习到一个鲁棒的分类器是不可能的。

考虑一个二元分类问题，我们希望区分狗和猫。如果分布可以以任意方式偏移，那么我们的设置允许病态的情况，即输入的分布保持不变：$p_S(\mathbf{x}) = p_T(\mathbf{x})$，但标签全部翻转：$p_S(y|\mathbf{x}) = 1 - p_T(y|\mathbf{x})$。换言之，如果上帝能突然决定，将来所有的"猫"现在都是狗，而我们以前所说的"狗"现在是猫。而此时输入$p(\mathbf{x})$的分布没有任何改变，那么我们就不可能将这种设置与分布完全没有变化的设置区分开。

幸运的是，在对未来我们的数据可能发生变化的一些限制性假设下，有原则的算法可以检测这种偏移，有时甚至动态调整，提高原始分类器的准确性。

协变量偏移

在分布偏移的分类中，协变量偏移可能是研究的最广泛的。这里我们假设，虽然输入的分布可能随时间而改变，但标签函数（即条件分布$P(y \mid \mathbf{x})$）没有改变。统计学家称之为协变量偏移（covariate shift），因为这个问题是由于协变量（特征）分布的变化而产生的。虽然有时我们可以在不引用因果关系的情况下对分布偏移进行推理，但我们注意到，在我们认为$\mathbf{x}$导致y的情况下，协变量偏移是一种自然假设。

考虑一下区分猫和狗的挑战。我们的训练数据包括 图3.9.1 中的图像。

图3.9.1: 区分猫和狗的训练数据。

在测试时，我们被要求对 图3.9.2 中的图像进行分类。

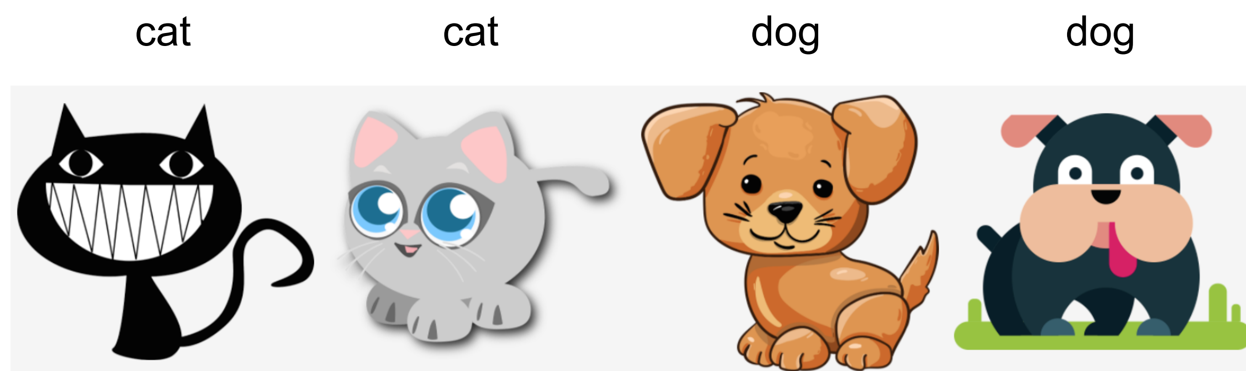


图3.9.2: 区分猫和狗的测试数据。

训练集由真实照片组成，而测试集只包含卡通图片。在一个与测试集的特征有着本质不同的数据集上进行训练，如果没有一个计划来适应新的领域，可能会带来麻烦。

标签偏移

标签偏移描述了与协变量偏移相反的问题。这里，我们假设标签边缘概率$P(y)$可以改变，但是类别条件分布$P(\mathbf{x} \mid y)$在不同的领域之间保持不变。当我们认为y导致$\mathbf{x}$时，标签偏移是一个合理的假设。例如，我们可能希望根据症状（或其他表现）来预测疾病，即使疾病的相对流行率随着时间的推移而变化。标签偏移在这里是恰当的假设，因为疾病会引起症状。在一些退化的情况下，标签偏移和协变量偏移假设可以同时成立。例如，当标签是确定的，即使y导致$\mathbf{x}$，协变量偏移假设也会得到满足。有趣的是，在这些情况下，使用基于标签偏移假设的方法通常是有利的。这是因为这些方法倾向于包含看起来像标签（通常是低维）的对象，而不是像输入的对象，后者在深度学习中往往是高维的。

概念偏移

我们也可能会遇到概念偏移的相关问题，当标签的定义发生变化时，就会出现这种问题。这听起来很奇怪——一只猫就是一只猫，不是吗？然而，其他类别会随着不同时间的用法而发生变化。精神疾病的诊断标准，所谓的时髦，以及工作头衔，都受到相当大的概念偏移的影响。事实证明，如果我们环游美国，根据所在的地理位置改变我们的数据来源，我们会发现关于"软饮"名称的分布发生了相当大的概念偏移，如 图3.9.3 所示。

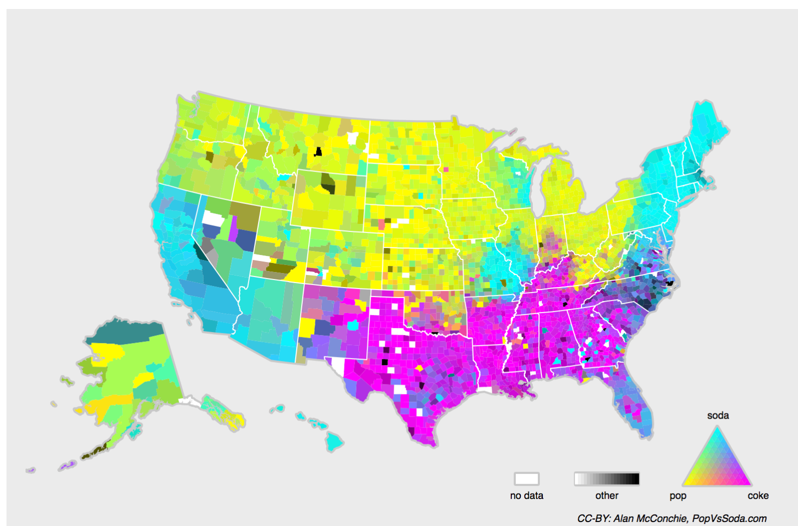


图3.9.3: 美国软饮名称的概念偏移。

如果我们要建立一个机器翻译系统，$P(y \mid \mathbf{x})$的分布可能会因我们的位置不同而有所不同。这个问题可能很难发现。我们希望利用在时间或空间上仅仅逐渐发生偏移的知识。

3.9.2 分布偏移示例

在深入研究形式体系和算法之前，我们可以讨论一些协变量偏移或概念偏移可能并不明显的具体情况。

医学诊断

假设你想设计一个检测癌症的算法。你从健康人和病人那里收集数据，然后训练你的算法。它工作得很好，给你很高的准确性，然后你得出了你已经准备好在医疗诊断事业上取得成功的结论。请先别着急。

产生训练数据的分布和你在实际中遇到的分布可能有很大的不同。这件事在一个不幸的初创公司身上发生过，我们中的一些作者几年前和他们合作过。他们正在研究一种血液检测方法，主要针对一种影响老年男性的疾病，并希望利用他们从病人身上采集的血液样本进行研究。然而，从健康男性身上获取血样比中系统中已有的病人身上获取要困难得多。作为补偿，这家初创公司向一所大学校园内的学生征集献血，作为开发测试的健康对照样本。然后这家初创公司问我们是否可以帮助他们建立一个用于检测疾病的分类器。

正如我们向他们解释的那样，用近乎完美的准确度来区分健康和患病人群确实很容易。然而，这是因为受试者在年龄、激素水平、体力活动、饮食、饮酒以及其他许多与疾病无关的因素上存在差异。这对真正的病人可能并不适用。从他们的抽样程序出发，我们可能会遇到极端的协变量偏移。此外，这种情况不太可能通过常规方法加以纠正。简言之，他们浪费了一大笔钱。

自动驾驶汽车

比如一家公司想利用机器学习来开发自动驾驶汽车。这里的一个关键部件是路沿检测器。由于真实的注释数据获取成本很高，他们想出了一个（聪明却有问题的）想法，将游戏渲染引擎中的合成数据用作额外的训练数据。这对从渲染引擎中抽取的"测试数据"非常有效。但应用在一辆真正的汽车里真是一场灾难。正如事实证明的那样，路沿被渲染成一种非常简单的纹理。更重要的是，所有的路沿都被渲染成了相同的纹理，路沿检测器很快就学习到了这个"特征"。

当美军第一次试图在森林中探测坦克时，也发生了类似的事情。他们在没有坦克的情况下拍摄了森林的航拍照片，然后把坦克开进森林，拍摄了另一组照片。分类器似乎工作得很好。不幸的是，它仅仅学会了如何区分有阴影的树和没有阴影的树——第一组照片是在清晨拍摄的，第二组是在中午拍摄的。

非平稳分布

当分布变化缓慢并且模型没有得到充分更新时，就会出现更微妙的情况：非平稳分布（nonstationary distribution）。以下是一些典型的情况。

- 我们训练了一个计算广告模型，但却没有经常更新（例如，我们忘记了一个叫iPad的不知名新设备刚刚上市）。
- 我们建立了一个垃圾邮件过滤器。它能很好地检测到我们目前看到的所有垃圾邮件。但是，垃圾邮件发送者们变得聪明起来，制造出新的信息，看起来不像我们以前见过的任何垃圾邮件。
- 我们建立了一个产品推荐系统。它在整个冬天都有效，但圣诞节过后很久还会继续推荐圣诞帽。

更多轶事

- 我们建立了一个人脸检测器。它在所有基准测试中都能很好地工作。不幸的是，它在测试数据上失败了——有问题的例子是人脸充满了整个图像的特写镜头（训练集中没有这样的数据）。
- 我们为美国市场建立了一个网络搜索引擎，并希望将其部署到英国。
- 我们通过在一个大的数据集来训练图像分类器，其中每一个大类的数量在数据集近乎是平均的，比如1000个类别，每个类别由1000个图像表示。然后我们将该系统部署到真实世界中，照片的实际标签分布显然是不均匀的。

3.9.3 分布偏移纠正

正如我们所讨论的，在许多情况下，训练和测试分布$P(\mathbf{x}, y)$是不同的。在某些情况下，我们很幸运，不管协变量、标签或概念发生偏移，模型都能正常工作。在其他情况下，我们可以通过运用有原则的策略来应对这种偏移，从而做得更好。本节的其余部分将变得更加技术化。不耐烦的读者可以继续下一节，因为这些内容不是后续概念的先修内容。

经验风险与真实风险

让我们首先反思一下在模型训练期间到底发生了什么：我们迭代训练数据$\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$的特征和相关的标签，并在每一个小批量之后更新模型$f$的参数。为了简单起见，我们不考虑正则化，因此我们在极大地降低了训练损失：

$$\mathop{\mathrm{minimize}}_f \frac{1}{n} \sum_{i=1}^n l(f(\mathbf{x}_i), y_i), \tag{3.9.1}$$

其中l是损失函数，用来度量给定响应标签y_i，预测$f(\mathbf{x}_i)$的“糟糕程度”。统计学家称 (3.9.1) 中的这一项为经验风险。经验风险（empirical risk）是为了近似 真实风险（true risk），整个训练数据上的平均损失，即从其真实分布$p(\mathbf{x},y)$中抽取的所有数据的总体损失的期望值：

$$E_{p(\mathbf{x}, y)} [l(f(\mathbf{x}), y)] = \int\int l(f(\mathbf{x}), y) p(\mathbf{x}, y) \;d\mathbf{x}dy. \tag{3.9.2}$$

然而，在实践中，我们通常无法获得数据的总体。因此，经验风险最小化即在 (3.9.1) 中最小化经验风险，是一种实用的机器学习策略，希望能近似最小化真实风险。

协变量偏移纠正

假设我们要估计一些依赖关系$P(y \mid \mathbf{x})$，我们有了带有标签的数据$(\mathbf{x}_i, y_i)$。不幸的是，观测值$\mathbf{x}_i$是从某些源分布$q(\mathbf{x})$中得出的，而不是从目标分布$p(\mathbf{x})$中得出的。幸运的是，依赖性假设意味着条件分布保持不变：$p(y \mid \mathbf{x}) = q(y \mid \mathbf{x})$。如果源分布$q(\mathbf{x})$是“错误的”，我们可以通过在真实风险的计算中使用以下简单的恒等式来进行纠正：

$$\int\int l(f(\mathbf{x}), y) p(y \mid \mathbf{x})p(\mathbf{x}) \;d\mathbf{x}dy = \int\int l(f(\mathbf{x}), y) q(y \mid \mathbf{x})q(\mathbf{x})\frac{p(\mathbf{x})}{q(\mathbf{x})} \;d\mathbf{x}dy. \tag{3.9.3}$$

换句话说，我们需要根据数据来自正确分布与来自错误分布的概率之比来重新衡量每个数据样本的权重：

$$\beta_i \stackrel{\mathrm{def}}{=} \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)}. \tag{3.9.4}$$

将权重β_i代入到每个数据样本$(\mathbf{x}_i, y_i)$中，我们可以使用"加权经验风险最小化"来训练模型：

$$\underset{f}{\mathrm{minimize}} \frac{1}{n} \sum_{i=1}^{n} \beta_i l(f(\mathbf{x}_i), y_i). \tag{3.9.5}$$

同样，我们不知道这个比率，所以在我们可以做任何有用的事情之前，我们需要估计它。有许多方法都可以用，包括一些花哨的算子理论方法，试图直接使用最小范数或最大熵原理重新校准期望算子。需要注意的是，对于任意一种这样的方法，我们都需要从全部两个分布中抽取样本："真实"的分布p，通过访问测试数据获取，和用于生成训练集q的分布（后者很容易获得）。但是请注意，我们只需要特征$\mathbf{x} \sim p(\mathbf{x})$；我们不需要访问标签$y \sim p(y)$。

在这种情况下，有一种非常有效的方法可以得到几乎与原始方法一样好的结果：logistic回归，这是用于二元分类的softmax回归（见 2.4节 ）的一个特例。这就是计算估计的概率比所需的全部内容。我们学习了一个分类器来区分从$p(\mathbf{x})$抽取的数据和从$q(\mathbf{x})$抽取的数据。如果无法区分这两个分布，则意味着想相关的样本同样可能来自这两个分布中的任何一个。另一方面，任何可以很好区分的样本都应该相应地显著增加或减少权重。

为了简单起见，假设我们分别从$p(\mathbf{x})$和$q(\mathbf{x})$两个分布中拥有相同数量的样本。现在用z标签表示从p抽取的数据为1，从q抽取的数据为-1。然后，混合数据集中的概率由下式给出

$$P(z=1 \mid \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x})+q(\mathbf{x})} \text{ and hence } \frac{P(z=1 \mid \mathbf{x})}{P(z=-1 \mid \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}. \tag{3.9.6}$$

因此，如果我们使用logistic回归方法，其中$P(z=1 \mid \mathbf{x}) = \frac{1}{1+\exp(-h(\mathbf{x}))}$（$h$是一个参数化函数），则很自然有：

$$\beta_i = \frac{1/(1+\exp(-h(\mathbf{x}_i)))}{\exp(-h(\mathbf{x}_i))/(1+\exp(-h(\mathbf{x}_i)))} = \exp(h(\mathbf{x}_i)). \tag{3.9.7}$$

因此，我们需要解决两个问题：第一个问题是区分来自两个分布的数据，然后是 (3.9.5) 中的加权经验风险最小化问题，在这个问题中，我们将对其中的项加权β_i。

现在我们准备描述一个纠正算法。假设我们有一个训练集$\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$和一个未标记的测试集$\{\mathbf{u}_1, \ldots, \mathbf{u}_m\}$。对于协变量偏移，我们假设$1 \leq i \leq n$的$\mathbf{x}_i$来自某个源分布，$\mathbf{u}_i$来自目标分布。以下是纠正协变量偏移的典型算法：

1. 生成一个二元分类训练集：$\{(\mathbf{x}_1, -1), \ldots, (\mathbf{x}_n, -1), (\mathbf{u}_1, 1), \ldots, (\mathbf{u}_m, 1)\}$。
2. 用logistic回归训练二元分类器得到函数h。
3. 使用$\beta_i = \exp(h(\mathbf{x}_i))$或更好的$\beta_i = \min(\exp(h(\mathbf{x}_i)), c)$（$c$为常量）对训练数据进行加权。
4. 使用权重β_i进行 (3.9.5) 中$\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$的训练。

请注意，上述算法依赖于一个重要的假设。为了使该方案起作用，我们需要目标分布(例如，测试分布)中的每个数据样本在训练时出现的概率非零。如果我们找到$p(\mathbf{x}) > 0$但$q(\mathbf{x}) = 0$的点，那么相应的重要性权重应该是无穷大。

标签偏移纠正

假设我们处理的是k个类别的分类任务。使用 3.9.3节 中相同符号，q和p中分别是源分布（例如训练时）和目标分布（例如测试时的分布）。假设标签的分布随时间变化：$q(y) \neq p(y)$，但类别条件分布保持不变：$q(\mathbf{x} \mid y) = p(\mathbf{x} \mid y)$。如果源分布$q(y)$是"错误的"，我们可以根据 (3.9.2) 中定义的真实风险中的如下恒等式进行更正：

$$\int \int l(f(\mathbf{x}), y) p(\mathbf{x} \mid y) p(y) \; d\mathbf{x} dy = \int \int l(f(\mathbf{x}), y) q(\mathbf{x} \mid y) q(y) \frac{p(y)}{q(y)} \; d\mathbf{x} dy. \tag{3.9.8}$$

这里，我们的重要性权重将对应于标签似然比率

$$\beta_i \stackrel{\text{def}}{=} \frac{p(y_i)}{q(y_i)}. \tag{3.9.9}$$

标签偏移的一个好处是，如果我们在源分布上有一个相当好的模型，那么我们可以得到对这些权重的一致估计，而不需要处理周边的其他维度。在深度学习中，输入往往是高维对象，如图像，而标签通常是更简单的对象，如类别。

为了估计目标标签分布，我们首先采用性能相当好的现成分类器(通常基于训练数据进行训练)，并使用验证集(也来自训练分布)计算其混淆矩阵。混淆矩阵$\mathbf{C}$是一个$k \times k$矩阵，其中每列对应于标签类别，每行对应于我们模型的预测类别。每个单元格的值c_{ij}是验证集中，真实标签为j，而我们的模型预测为i的样本数量所占的比例。

现在，我们不能直接计算目标数据上的混淆矩阵，因为我们无法看到真实环境下的样本的标签，除非我们再搭建一个复杂的实时标注流程。然而，我们所能做的是将所有模型在测试时的预测平均起来，得到平均模型输出$\mu(\hat{\mathbf{y}}) \in \mathbb{R}^k$，其中第$i$个元素$\mu(\hat{y}_i)$是我们模型预测测试集中$i$的总预测分数。

结果表明，在一些温和的条件下——如果我们的分类器一开始就相当准确，如果目标数据只包含我们以前见过的类别，以及如果标签偏移假设成立（这是这里最强的假设），然后我们可以通过求解一个简单的线性系统来估计测试集的标签分布

$$\mathbf{C} p(\mathbf{y}) = \mu(\hat{\mathbf{y}}), \tag{3.9.10}$$

因为作为一个估计，$\sum_{j=1}^k c_{ij} p(y_j) = \mu(\hat{y}_i)$对所有$1 \leq i \leq k$成立，其中$p(y_j)$是$k$维标签分布向量$p(\mathbf{y})$的第$j^\mathrm{th}$元素。如果我们的分类器一开始就足够精确，那么混淆矩阵$\mathbf{C}$将是可逆的，进而我们可以得到一个解$p(\mathbf{y}) = \mathbf{C}^{-1} \mu(\hat{\mathbf{y}})$。

因为我们观测源数据上的标签，所以很容易估计分布$q(y)$。那么对于标签为y_i的任何训练样本i，我们可以使用我们估计的$p(y_i)/q(y_i)$比率来计算权重β_i，并将其代入 (3.9.5) 中的加权经验风险最小化中。

概念偏移纠正

概念偏移很难用原则性的方式解决。例如，在一个问题突然从区分猫和狗偏移为区分白色和黑色动物的情况下，再假设我们可以做得比从零开始收集新标签和训练要好得多就是不合理的。幸运的是，在实践中，这种极端的偏移是罕见的。相反，通常情况下，任务的变化总是缓慢的。为了更具体说明，下面是一些例子：

- 在计算广告中，新产品推出，旧产品变得不那么受欢迎了。这意味着广告的分布和受欢迎程度是逐渐变化的，任何点击率预测器都需要随之逐渐变化。

- 由于环境的磨损，交通摄像头的镜头会逐渐退化，影响摄像头的图像质量。
- 新闻内容逐渐变化（即大部分新闻保持不变，但出现新的新闻）。

在这种情况下，我们可以使用与训练网络相同的方法，使其适应数据的变化。换言之，我们使用现有的网络权值，简单地用新数据执行一些更新步骤，而不是从头开始训练。

3.9.4 学习问题的分类法

有了如何处理分布变化的知识，我们现在可以考虑机器学习问题形式化的其他方面。

批量学习

在批量学习中，我们可以访问训练特征和标签$\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$，我们使用这些特性和标签训练$f(\mathbf{x})$。然后，我们部署此模型来对来自同一分布的新数据$(\mathbf{x}, y)$进行评分。这个假设是我们这里讨论的任何问题的默认假设。例如，我们可以根据猫和狗的大量图片训练猫检测器。一旦我们训练了它，我们就把它作为智能猫门计算视觉系统的一部分，来控制只允许猫进入。然后这个系统会被安装在客户家中，再也不会更新（除非极端情况下）。

在线学习

现在假设数据$(\mathbf{x}_i, y_i)$每次到达一个样本。更具体地说，假设我们首先观测到$\mathbf{x}_i$，然后我们需要得出一个估计值$f(\mathbf{x}_i)$，只有当我们做到这一点后，我们才观测到y_i，然后根据我们的决定，我们会得到奖励或遇到损失。许多实际问题都属于这一类。例如，我们需要预测明天的股票价格，这样我们就可以根据这一估计进行交易，在一天结束时，我们会发现我们的估计是否会使我们盈利。换句话说，在在线学习中，我们有以下的循环，在这个循环中，给定新的观测结果，我们会不断地改进我们的模型。

$$\mathrm{model}\ f_t \longrightarrow \mathrm{data}\ \mathbf{x}_t \longrightarrow \mathrm{estimate}\ f_t(\mathbf{x}_t) \longrightarrow \mathrm{observation}\ y_t \longrightarrow \mathrm{loss}\ l(y_t, f_t(\mathbf{x}_t)) \longrightarrow \mathrm{model}\ f_{t+1} \tag{3.9.11}$$

老虎机

老虎机（Bandits）是上述问题的一个特例。虽然在大多数学习问题中，我们有一个连续参数化的函数f，我们想学习它的参数（例如，一个深度网络），但在一个老虎机问题中，我们只有有限数量的手臂可以拉动，也就是说，我们可以采取的行动是有限的。对于这个更简单的问题，可以获得更强的最优性理论保证，这并不令人惊讶。我们之所以列出它，主要是因为这个问题经常被视为一个单独的学习问题的设定。

控制

在很多情况下，环境会记住我们所做的。不一定是以一种对抗的方式，但它会记住，而且它的反应将取决于之前发生的事情。例如，咖啡锅炉控制器将根据之前是否加热锅炉来观测到不同的温度。PID（比例—积分—微分）控制器算法是一个流行的选择。同样，用户在新闻网站上的行为也将取决于我们之前向他展示的内容（例如，大多数新闻他只阅读一次）。许多这样的算法形成了一个环境模型，在这个模型中，他们的行为使得他们的决策看起来不那么随机。近年来，控制理论（如PID的变体）也被用于自动调整超参数，以获得更好的解构和重建质量，提高生成文本的多样性和生成图像的重建质量 [Shao et al., 2020] 。

强化学习

强化学习（reinforcement learning）强调如何基于环境而行动，以取得最大化的预期利益。国际象棋、围棋、西洋双陆棋或星际争霸都是强化学习的一些例子。再比如，为自动驾驶汽车制造一个好的控制器，或者以其他方式对自动驾驶汽车的驾驶方式做出反应，例如，试图避开它，试图造成事故，并试图与其合作。

考虑到环境

上述不同情况之间的一个关键区别是，在静止环境中可能一直有效的相同策略，在环境能够改变的情况下可能不会始终有效。例如，一个交易者发现的套利机会很可能在他开始利用它时就消失了。环境变化的速度和方式在很大程度上决定了我们可以采用的算法类型。例如，如果我们知道事情只会缓慢地变化，我们就可以迫使任何估计也只能缓慢地发生改变。如果我们知道环境可能会瞬间发生变化，但这种变化非常罕见，我们就可以在使用算法时考虑到这一点。这些类型的知识对于有追求的数据科学家处理概念偏移时至关重要，也就是说，当他试图解决的问题会随着时间的推移而发生变化时。

3.9.5 机器学习中的公平、责任和透明度

最后，重要的是要记住，当你部署机器学习系统时，你不仅仅是在优化一个预测模型——你通常是在提供一个会被用来（部分或完全）进行自动化决策的工具。这些技术系统可能会通过其进行的决定而影响到个人的生活。从考虑预测到决策的飞跃不仅提出了新的技术问题，而且还提出了一系列必须仔细考虑的伦理问题。如果我们正在部署一个医疗诊断系统，我们需要知道它可能适用于哪些人群，哪些人群可能无效。忽视对一个亚群体的幸福的可预见风险可能会导致我们执行劣质的护理水平。此外，一旦我们规划整个决策系统，我们必须退后一步，重新考虑如何评估我们的技术。在这个视野变化所导致的结果中，我们会发现准确率很少成为合适的衡量标准。例如，当我们将预测转化为行动时，我们通常会考虑到各种方式犯错的潜在成本敏感性。如果对图像错误地分到某一类别可能被视为一种种族伎俩，而错误分到另一个类别是无害的，那么我们可能需要相应地调整我们的阈值，在设计决策方式时考虑到这些社会价值。我们还需要注意预测系统如何导致反馈循环。例如，考虑预测性警务系统，它将巡逻人员分配到预测犯罪率较高的地区。很容易看出一种令人担忧的模式是如何出现的：

1. 犯罪率高的社区会得到更多的巡逻。
2. 因此，在这些社区中会发现更多的犯罪行为，输入可用于未来迭代的训练数据。
3. 面对更多的积极因素，该模型预测这些社区还会有更多的犯罪。

4. 在下一次迭代中，更新后的模型会更加倾向于针对同一个地区，这会导致更多的犯罪行为被发现等等。

通常，在建模过纠正程中，模型的预测与训练数据耦合的各种机制都没有得到解释。这可能导致研究人员称之为“失控反馈循环”的现象。此外，我们首先要注意我们是否解决了正确的问题。预测算法现在在信息传播中起着巨大的中介作用。个人遭遇的新闻应该由他们喜欢的Facebook页面决定吗？这些只是你在机器学习职业生涯中可能遇到的许多令人感到压力山大的道德困境中的一小部分。

3.9.6 小结

- 在许多情况下，训练集和测试集并不来自同一个分布。这就是所谓的分布偏移。
- 真实风险是从真实分布中抽取的所有数据的总体损失的预期。然而，这个数据总体通常是无法获得的。经验风险是训练数据的平均损失，用于近似真实风险。在实践中，我们进行经验风险最小化。
- 在相应的假设条件下，可以在测试时检测并纠正协变量偏移和标签偏移。在测试时，不考虑这种偏移可能会成为问题。
- 在某些情况下，环境可能会记住自动操作并以令人惊讶的方式做出响应。在构建模型时，我们必须考虑到这种可能性，并继续监控实时系统，并对我们的模型和环境以意想不到的方式纠缠在一起的可能性持开放态度。

3.9.7 练习

1. 当我们改变搜索引擎的行为时会发生什么?用户可能会做什么?那广告商呢?
2. 实现一个协变量偏移检测器。提示:构建一个分类器。
3. 实现协变量偏移纠正。
4. 除了分布偏移，还有什么会影响经验风险接近真实风险的程度?

Discussions[64]

3.10 实战 Kaggle 比赛：预测房价

现在我们已经介绍了一些建立和训练深度网络的基本工具,和网络正则化的技术(如权重衰减、Dropout等)。我们准备通过参加Kaggle比赛来将所有这些知识付诸实践。房价预测比赛是一个很好的起点。这个数据是相当通用的,不会需要使用带特殊结构的模型(就像音频或视频可能需要的那样)。此数据集由Bart de Cock于2011年收集 [DeCock, 2011] ,涵盖了2006-2010年期间亚利桑那州埃姆斯市的房价。它比哈里森和鲁宾菲尔德(1978年)的波士顿房价[65]数据集要大得多，也有更多的特征。

在本节中，我们将详细介绍数据预处理、模型设计和超参数选择。我们希望通过亲身实践的方式，你将获得一些直觉。这些直觉将指导你数据科学家职业生涯。

[64] https://discuss.d2l.ai/t/1822

[65] https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names

3.10.1 下载和缓存数据集

在整本书中，我们将在各种下载的数据集上训练和测试模型。在这里，我们实现了几个实用函数来方便下载数据。首先，我们维护字典DATA_HUB，其将数据集名称的字符串映射到数据集相关的二元组上，这个二元组包含数据集的url和验证文件完整性的sha-1密钥。所有这样的数据集都托管在地址为DATA_URL的站点上。

```
import hashlib
import os
import tarfile
import zipfile
import requests

#@save
DATA_HUB = dict()
DATA_URL = 'http://d2l-data.s3-accelerate.amazonaws.com/'
```

下面的download函数用来下载数据集，将数据集缓存在本地目录（默认情况下为../data）中，并返回下载文件的名称。如果缓存目录中已经存在与此数据集文件，并且其sha-1与存储在DATA_HUB中的相匹配，我们将使用缓存的文件，以避免重复的下载。

```
def download(name, cache_dir=os.path.join('..', 'data')):  #@save
    """下载一个DATA_HUB中的文件，返回本地文件名。"""
    assert name in DATA_HUB, f"{name} 不存在于 {DATA_HUB}."
    url, sha1_hash = DATA_HUB[name]
    os.makedirs(cache_dir, exist_ok=True)
    fname = os.path.join(cache_dir, url.split('/')[-1])
    if os.path.exists(fname):
        sha1 = hashlib.sha1()
        with open(fname, 'rb') as f:
            while True:
                data = f.read(1048576)
                if not data:
                    break
                sha1.update(data)
        if sha1.hexdigest() == sha1_hash:
            return fname  # Hit cache
    print(f'正在从{url}下载{fname}...')
    r = requests.get(url, stream=True, verify=True)
    with open(fname, 'wb') as f:
        f.write(r.content)
    return fname
```

我们还实现了两个额外的实用函数：一个是下载并解压缩一个zip或tar文件，另一个是将本书中使用的所有数据集从DATA_HUB下载到缓存目录中。

```python
def download_extract(name, folder=None):  #@save
    """下载并解压zip/tar文件。"""
    fname = download(name)
    base_dir = os.path.dirname(fname)
    data_dir, ext = os.path.splitext(fname)
    if ext == '.zip':
        fp = zipfile.ZipFile(fname, 'r')
    elif ext in ('.tar', '.gz'):
        fp = tarfile.open(fname, 'r')
    else:
        assert False, '只有zip/tar文件可以被解压缩。'
    fp.extractall(base_dir)
    return os.path.join(base_dir, folder) if folder else data_dir

def download_all():  #@save
    """下载DATA_HUB中的所有文件。"""
    for name in DATA_HUB:
        download(name)
```

3.10.2 Kaggle

Kaggle[66]是一个现在流行的举办机器学习比赛的平台。每场比赛都以一个数据集为中心。许多比赛都是由利益相关者赞助的，他们为获胜的解决方案提供奖金。该平台帮助用户通过论坛和共享代码进行互动，促进协作和竞争。虽然排行榜的追逐往往失控，研究人员短视地专注于预处理步骤，而不是考虑基础性问题，但一个客观的平台有巨大的价值。该平台促进了竞争方法之间的直接定量比较，以及代码共享。这便于每个人都可以了解哪些方法起作用，哪些没有起作用。如果你想参加Kaggle比赛，你首先需要注册一个账户（见 图3.10.1 ）。

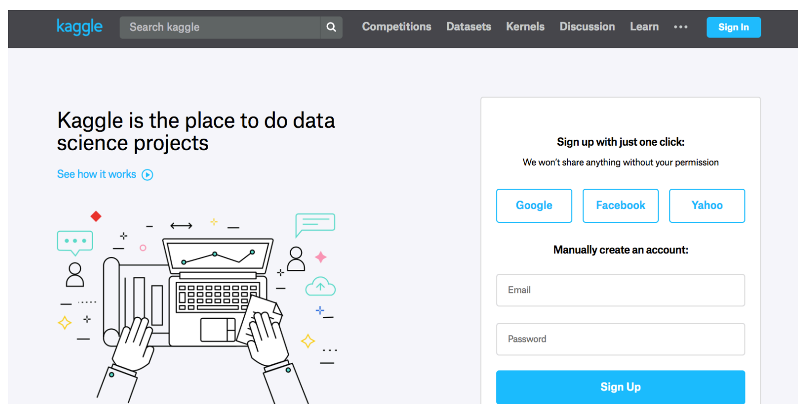


图3.10.1: Kaggle网站

[66] https://www.kaggle.com

在房价预测比赛页面（如 图3.10.2 所示），你在“Data”选项卡下可以找到数据集。你可以通过下面的网址提交预测，并查看排名：

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

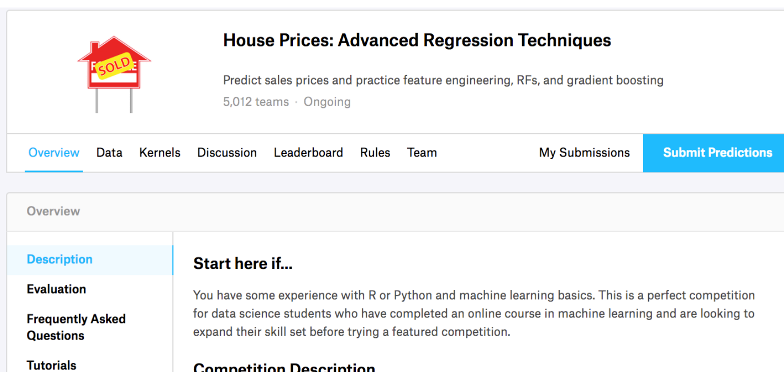


图3.10.2: 房价预测比赛页面

3.10.3 访问和读取数据集

注意，竞赛数据分为训练集和测试集。每条记录都包括房屋的属性值和属性，如街道类型、施工年份、屋顶类型、地下室状况等。这些特征由各种数据类型组成。例如，建筑年份由整数表示，屋顶类型由离散类别表示，其他特征由浮点数表示。这就是现实让事情变得复杂的地方：例如，一些数据完全丢失了，缺失值被简单地标记为“NA”。每套房子的价格只出现在训练集中（毕竟这是一场比赛）。我们将希望划分训练集以创建验证集，但是在将预测结果上传到Kaggle之后，我们只能在官方测试集中评估我们的模型。在 图3.10.2 中，“Data”选项卡有下载数据的链接。

开始之前，我们将使用pandas读入并处理数据，这是我们在 1.2节 中引入的。因此，在继续操作之前，您需要确保已安装pandas。幸运的是，如果你正在用Jupyter阅读该书，你可以在不离开笔记本的情况下安装pandas。

```
# 如果pandas没有被安装，请取消下一句的注释。
# !pip install pandas

%matplotlib inline
import pandas as pd
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

为方便起见，我们可以使用上面定义的脚本下载并缓存Kaggle房屋数据集。

```
DATA_HUB['kaggle_house_train'] = (  #@save
    DATA_URL + 'kaggle_house_pred_train.csv',
```

(continues on next page)

(continued from previous page)

```
    '585e9cc93e70b39160e7921475f9bcd7d31219ce')

DATA_HUB['kaggle_house_test'] = (  #@save
    DATA_URL + 'kaggle_house_pred_test.csv',
    'fa19780a7b011d9b009e8bff8e99922a8ee2eb90')
```

我们使用pandas分别加载包含训练数据和测试数据的两个csv文件。

```
train_data = pd.read_csv(download('kaggle_house_train'))
test_data = pd.read_csv(download('kaggle_house_test'))
```

```
正 在 从http://d2l-data.s3-accelerate.amazonaws.com/kaggle_house_pred_train.csv下 载../
↪data/kaggle_house_pred_train.csv...
正在从http://d2l-data.s3-accelerate.amazonaws.com/kaggle_house_pred_test.csv下载../data/
↪kaggle_house_pred_test.csv...
```

训练数据集包括1460个样本，每个样本80个特征和1个标签，而测试数据包含1459个样本，每个样本80个特征。

```
print(train_data.shape)
print(test_data.shape)
```

```
(1460, 81)
(1459, 80)
```

让我们看看前四个和最后两个特征，以及前四个样本的标签（房价）。

```
print(train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]])
```

```
   Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SalePrice
0   1          60       RL         65.0       WD        Normal     208500
1   2          20       RL         80.0       WD        Normal     181500
2   3          60       RL         68.0       WD        Normal     223500
3   4          70       RL         60.0       WD       Abnorml     140000
```

我们可以看到，在每个样本中，第一个特征是ID，这有助于模型识别每个训练样本。虽然这很方便，但它不携带任何用于预测的信息。因此，在将数据提供给模型之前，我们将其从数据集中删除。

```
all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

3.10.4 数据预处理

如上所述，我们有各种各样的数据类型。在开始建模之前，我们需要对数据进行预处理。让我们从数字特征开始。首先，我们应用启发式方法，将所有缺失的值替换为相应特征的平均值。然后，为了将所有特征放在一个共同的尺度上，我们通过将特征重新缩放到零均值和单位方差来标准化数据：

$$x \leftarrow \frac{x - \mu}{\sigma}. \tag{3.10.1}$$

要验证这确实转换了我们的特征(变量)，使特征具有零均值和单位方差，即 $E[\frac{x-\mu}{\sigma}] = \frac{\mu-\mu}{\sigma} = 0$和$E[(x-\mu)^2] = (\sigma^2 + \mu^2) - 2\mu^2 + \mu^2 = \sigma^2$。直观地说，我们标准化数据有两个原因。首先，它方便优化。其次，因为我们不知道哪些特征是相关的，所以我们不想让惩罚分配给一个特征的系数比分配给其他任何特征的系数更大。

```
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# 在标准化数据之后，所有数据都意味着消失，因此我们可以将缺失值设置为0
all_features[numeric_features] = all_features[numeric_features].fillna(0)
```

接下来，我们处理离散值。这包括诸如"MSZoning"之类的特征。我们用一次独热编码替换它们，方法与前面将多类别标签转换为向量的方式相同(请参见 2.4.1节)。例如，"MSZoning"包含值"RL"和"Rm"。将创建两个新的指示器特征"MSZoning_RL"和"MSZoning_RM"，其值为0或1。根据独热编码，如果"MSZoning"的原始值为"RL"，则："MSZoning_RL"为1，"MSZoning_RM"为0。pandas软件包会自动为我们实现这一点。

```
# `Dummy_na=True` 将"na"（缺失值）视为有效的特征值，并为其创建指示符特征。
all_features = pd.get_dummies(all_features, dummy_na=True)
all_features.shape
```

```
(2919, 331)
```

你可以看到，此转换会将特征的数量从79个增加到331个。最后，通过values属性，我们可以从pandas格式中提取NumPy格式，并将其转换为张量表示用于训练。

```
n_train = train_data.shape[0]
train_features = np.array(all_features[:n_train].values, dtype=np.float32)
test_features = np.array(all_features[n_train:].values, dtype=np.float32)
train_labels = np.array(train_data.SalePrice.values.reshape(-1, 1),
                        dtype=np.float32)
```

3.10.5 训练

首先，我们训练一个带有损失平方的线性模型。毫不奇怪，我们的线性模型不会让我们在竞赛中获胜，但线性模型提供了一种健全性检查，以查看数据中是否存在有意义的信息。如果我们在这里不能做得比随机猜测更好，那么我们很可能存在数据处理错误。如果一切顺利，线性模型将作为基线模型，让我们直观地知道简单的模型离报告最好的模型有多近，让我们感觉到我们应该从更酷炫的模型中获得多少收益。

```
loss = gluon.loss.L2Loss()

def get_net():
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize()
    return net
```

对于房价，就像股票价格一样，我们关心的是相对数量，而不是绝对数量。因此，我们更关心相对误差$\frac{y-\hat{y}}{y}$，而不是绝对误差$y-\hat{y}$。例如，如果我们在俄亥俄州农村地区估计一栋房子的价格时，我们的预测偏差了10万美元，在那里一栋典型的房子的价值是12.5万美元，那么我们可能做得很糟糕。另一方面，如果我们在加州豪宅区的预测出现了这个数字的偏差，这可能是一个惊人的准确预测（在那里，房价均值超过400万美元）。

解决这个问题的一种方法是用价格预测的对数来衡量差异。事实上，这也是比赛中官方用来评价提交质量的误差指标。即将$\delta for $|\log y - \log \hat{y}| \leq \delta$转换为$e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^\delta$。这使得预测价格的对数与真实标签价格的对数之间出现以下均方根误差：

$$\sqrt{\frac{1}{n}\sum_{i=1}^n\left(\log y_i -\log \hat{y}_i\right)^2}. \tag{3.10.2}$$

```
def log_rmse(net, features, labels):
    # 为了在取对数时进一步稳定该值，将小于1的值设置为1
    clipped_preds = np.clip(net(features), 1, float('inf'))
    return np.sqrt(2 * loss(np.log(clipped_preds), np.log(labels)).mean())
```

与前面的部分不同，我们的训练函数将依赖于Adam优化器（我们将在后面更详细地描述它）。这个优化器的主要吸引力在于，尽管在提供无限资源进行超参数优化方面没有做得更好（有时更差），但人们发现它对初始学习率不那么敏感。

```
def train(net, train_features, train_labels, test_features, test_labels,
          num_epochs, learning_rate, weight_decay, batch_size):
    train_ls, test_ls = [], []
    train_iter = d2l.load_array((train_features, train_labels), batch_size)
    # 这里使用的是Adam优化算法
    trainer = gluon.Trainer(net.collect_params(), 'adam', {
        'learning_rate': learning_rate,
        'wd': weight_decay})
```

(continues on next page)

(continued from previous page)

```
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        train_ls.append(log_rmse(net, train_features, train_labels))
        if test_labels is not None:
            test_ls.append(log_rmse(net, test_features, test_labels))
    return train_ls, test_ls
```

3.10.6 K折交叉验证

你可能还记得，我们在讨论模型选择的部分（3.4节）中介绍了K折交叉验证。这有助于模型选择和超参数调整。我们首先需要一个函数，在K折交叉验证过程中返回第i折的数据。它选择第i个切片作为验证数据，其余部分作为训练数据。注意，这并不是处理数据的最有效方法，如果我们的数据集大得多，我们肯定会做一些更聪明的改变。但是这种改变所增加的复杂性可能会使代码看起来更乱。在这里可以忽略这些改变，因为我们的问题很简单。

```
def get_k_fold_data(k, i, X, y):
    assert k > 1
    fold_size = X.shape[0] // k
    X_train, y_train = None, None
    for j in range(k):
        idx = slice(j * fold_size, (j + 1) * fold_size)
        X_part, y_part = X[idx, :], y[idx]
        if j == i:
            X_valid, y_valid = X_part, y_part
        elif X_train is None:
            X_train, y_train = X_part, y_part
        else:
            X_train = np.concatenate([X_train, X_part], 0)
            y_train = np.concatenate([y_train, y_part], 0)
    return X_train, y_train, X_valid, y_valid
```

当我们在K折交叉验证中训练K次后，返回训练和验证误差的平均值。

```
def k_fold(k, X_train, y_train, num_epochs, learning_rate, weight_decay,
           batch_size):
    train_l_sum, valid_l_sum = 0, 0
    for i in range(k):
        data = get_k_fold_data(k, i, X_train, y_train)
```

(continues on next page)

(continued from previous page)

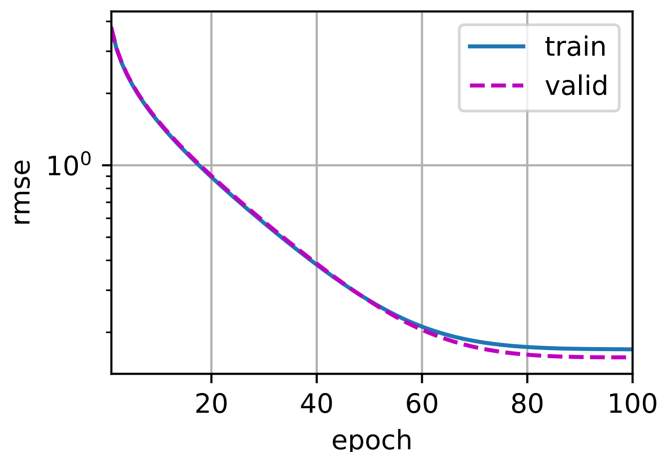
```
        net = get_net()
        train_ls, valid_ls = train(net, *data, num_epochs, learning_rate,
                                   weight_decay, batch_size)
        train_l_sum += train_ls[-1]
        valid_l_sum += valid_ls[-1]
        if i == 0:
            d2l.plot(list(range(1, num_epochs + 1)), [train_ls, valid_ls],
                     xlabel='epoch', ylabel='rmse', xlim=[1, num_epochs],
                     legend=['train', 'valid'], yscale='log')
        print(f'fold {i + 1}, train log rmse {float(train_ls[-1]):f}, '
              f'valid log rmse {float(valid_ls[-1]):f}')
    return train_l_sum / k, valid_l_sum / k
```

3.10.7 模型选择

在本例中，我们选择了一组未调优的超参数，并将其留给读者来改进模型。找到一个好的选择可能需要时间，这取决于一个人优化了多少变量。有了足够大的数据集和合适设置，K折交叉验证往往对多次测试具有相当的适应性。然而，如果我们尝试了不合理的大量选项，我们可能会发现验证效果不再代表真正的误差。

```
k, num_epochs, lr, weight_decay, batch_size = 5, 100, 5, 0, 64
train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr,
                          weight_decay, batch_size)
print(f'{k}-折验证: 平均训练log rmse: {float(train_l):f}, '
      f'平均验证log rmse: {float(valid_l):f}')
```

```
fold 1, train log rmse 0.169752, valid log rmse 0.157111
fold 2, train log rmse 0.162290, valid log rmse 0.189695
fold 3, train log rmse 0.163672, valid log rmse 0.168015
fold 4, train log rmse 0.167820, valid log rmse 0.154533
fold 5, train log rmse 0.163145, valid log rmse 0.182941
5-折验证: 平均训练log rmse: 0.165336, 平均验证log rmse: 0.170459
```



请注意，有时一组超参数的训练误差可能非常低，但K折交叉验证的误差要高得多。这表明我们过拟合了。在整个训练过程中，你将希望监控这训练误差和验证误差两个数字。较少的过拟合可能表明现有数据可以支撑一个更强大的模型。较大的过拟合可能意味着我们可以通过正则化技术来获益。

3.10.8 提交Kaggle的预测

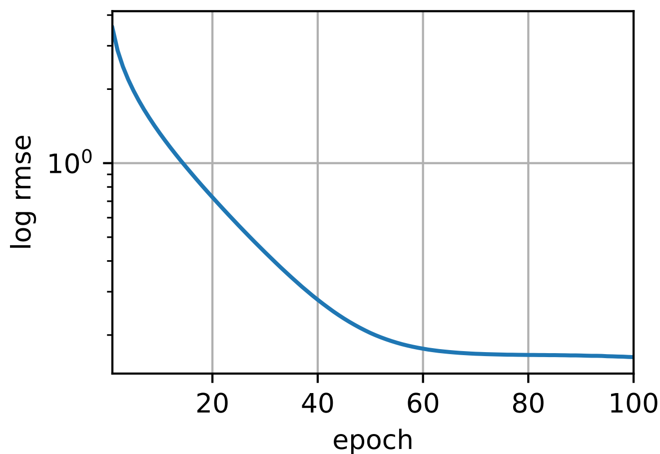
既然我们知道应该选择什么样的超参数，我们不妨使用所有数据对其进行训练（而不是仅使用交叉验证中使用的$1-1/K$的数据）。然后，我们通过这种方式获得的模型可以应用于测试集。将预测保存在csv文件中可以简化将结果上传到Kaggle的过程。

```
def train_and_pred(train_features, test_feature, train_labels, test_data,
                   num_epochs, lr, weight_decay, batch_size):
    net = get_net()
    train_ls, _ = train(net, train_features, train_labels, None, None,
                        num_epochs, lr, weight_decay, batch_size)
    d2l.plot(np.arange(1, num_epochs + 1), [train_ls], xlabel='epoch',
             ylabel='log rmse', xlim=[1, num_epochs], yscale='log')
    print(f'train log rmse {float(train_ls[-1]):f}')
    # 将网络应用于测试集。
    preds = net(test_features).asnumpy()
    # 将其重新格式化以导出到Kaggle
    test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
    submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
    submission.to_csv('submission.csv', index=False)
```

一种良好的完整性检查是查看测试集上的预测是否与K倍交叉验证过程中的预测相似。如果是，那就是时候把它们上传到Kaggle了。下面的代码将生成一个名为`submission.csv`的文件。

```
train_and_pred(train_features, test_features, train_labels, test_data,
               num_epochs, lr, weight_decay, batch_size)
```

```
train log rmse 0.162663
```



接下来，如 图3.10.3 中所示，我们可以提交预测到Kaggle上，并查看预测在测试集上与实际房价（标签）的比较情况。步骤非常简单：

- 登录Kaggle网站，访问房价预测竞赛页面。
- 点击“Submit Predictions”或“Late Submission”按钮(在撰写本文时，该按钮位于右侧)。
- 点击页面底部虚线框中的“Upload Submission File”按钮，选择你要上传的预测文件。
- 点击页面底部的“Make Submission”按钮，即可查看您的结果。

图3.10.3: 向Kaggle提交数据

3.10.9 小结

- 真实数据通常混合了不同的数据类型，需要进行预处理。
- 将实值数据重新缩放为零均值和单位方差是一个很好的默认设置。用它们的平均值替换缺失的值也是一个很好的默认设置。
- 将类别特征转化为指标特征，可以使我们把它们当作一个独热向量来对待。
- 我们可以使用K折交叉验证来选择模型并调整超参数。
- 对数对于相对误差很有用。

3.10.10 练习

1. 把你的预测提交给Kaggle。你的预测有多好？
2. 你能通过直接最小化价格的对数来改进你的模型吗？如果你试图预测价格的对数而不是价格，会发生什么？
3. 用平均值替换缺失值总是好主意吗？提示：你能构造一个不随机丢失值的情况吗？
4. 通过K折交叉验证调整超参数，从而提高Kaggle的得分。
5. 通过改进模型（例如，层、权重衰减和dropout）来提高分数。
6. 如果我们没有像本节所做的那样标准化连续的数值特征，会发生什么？

Discussions[67]

[67] https://discuss.d2l.ai/t/1823

4

深度学习计算

除了庞大的数据集和强大的硬件，优秀的软件工具在深度学习的快速发展中发挥了不可或缺的作用。从2007年发布的开创性的Theano库开始，灵活的开源工具使研究人员能够快速开发模型原型，避免了使用标准组件时的重复工作，同时仍然保持了进行底层修改的能力。随着时间的推移，深度学习库已经演变成提供越来越粗糙的抽象。就像半导体设计师从指定晶体管到逻辑电路再到编写代码一样，神经网络研究人员已经从考虑单个人工神经元的行为转变为从层的角度构思网络，现在通常在设计结构时考虑的是更粗糙的块（block）。

到目前为止，我们已经介绍了一些基本的机器学习概念，并慢慢介绍了功能齐全的深度学习模型。在上一章中，我们从零开始实现了多层感知机的每个组件，然后展示了如何利用高级API轻松地实现相同的模型。为了易于学习，我们调用了深度学习库，但是跳过了它们工作的细节。在本章中，我们开始深入探索深度学习计算的关键组件，即模型构建、参数访问与初始化、设计自定义层和块、将模型读写到磁盘，以及利用GPU实现显著的加速。这些知识将使你从基础用户变为高级用户。虽然本章不介绍任何新的模型或数据集，但后面的高级模型章节在很大程度上依赖于本章的知识。

4.1 层和块

当我们第一次介绍神经网络时，我们关注的是具有单一输出的线性模型。在这里，整个模型只由一个神经元组成。注意，单个神经元（1）接受一些输入；（2）生成相应的标量输出；（3）具有一组相关 参数（parameters），这些参数可以更新以优化某些感兴趣的目标函数。然后，当我们开始考虑具有多个输出的网络，我们就利用矢量化算法来描述整层神经元。像单个神经元一样，层（1）接受一组输入，（2）生成相应的输出，（3）由一组可调整参数描述。当我们使用softmax回归时，一个单层本身就是模型。然而，即使我们随后引入了多层感知机，我们仍然可以认为该模型保留了上面所说的基本结构。

有趣的是，对于多层感知机而言，整个模型及其组成层都是这种结构。整个模型接受原始输入（特征），生成

输出（预测），并包含一些参数（所有组成层的参数集合）。同样，每个单独的层接收输入（由前一层提供）生成输出（到下一层的输入），并且具有一组可调参数，这些参数根据从下一层反向传播的信号进行更新。

虽然你可能认为神经元、层和模型为我们的业务提供了足够的抽象，但事实证明，我们经常发现谈论比单个层大但比整个模型小的组件更方便。例如，在计算机视觉中广泛流行的ResNet-152结构就有数百层。这些层是由层组的重复模式组成。一次只实现一层这样的网络会变得很乏味。这种问题不是我们幻想出来的，这种设计模式在实践中很常见。上面提到的ResNet结构赢得了2015年ImageNet和COCO计算机视觉比赛的识别和检测任务 [He et al., 2016a]，目前ResNet结构仍然是许多视觉任务的首选结构。在其他的领域，如自然语言处理和语音，层以各种重复模式排列的类似结构现在也是普遍存在。

为了实现这些复杂的网络，我们引入了神经网络块的概念。块可以描述单个层、由多个层组成的组件或整个模型本身。使用块进行抽象的一个好处是可以将一些块组合成更大的组件，这一过程通常是递归的。这一点在 图4.1.1 中进行了说明。通过定义代码来按需生成任意复杂度的块，我们可以通过简洁的代码实现复杂的神经网络。

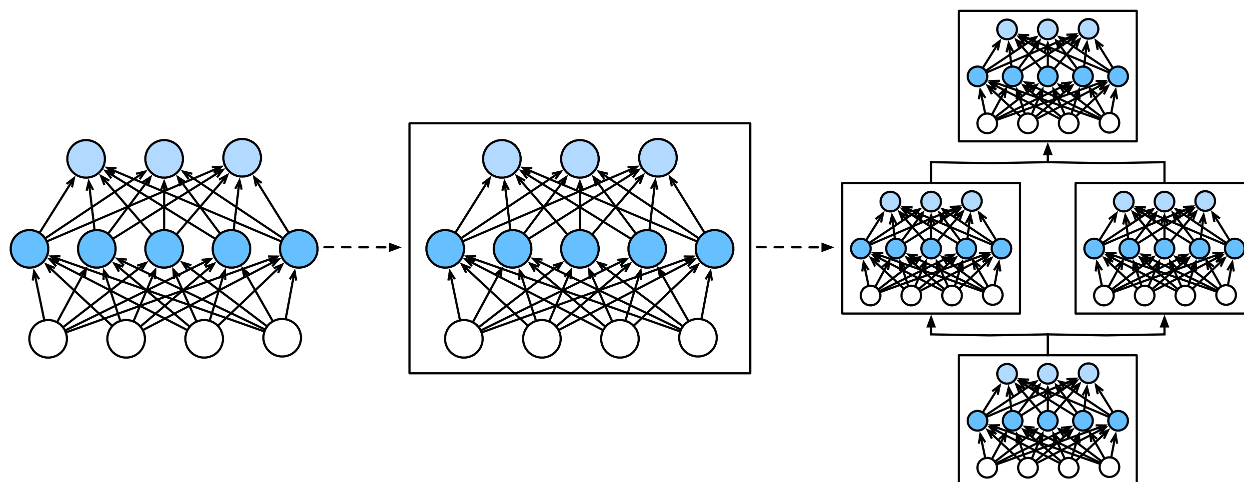


图4.1.1: 多个层被组合成块，形成更大的模型。

从编程的角度来看，块由类（class）表示。它的任何子类都必须定义一个将其输入转换为输出的正向传播函数，并且必须存储任何必需的参数。注意，有些块不需要任何参数。最后，为了计算梯度，块必须具有反向传播函数。幸运的是，在定义我们自己的块时，由于自动微分（在 1.5节 中引入）提供了一些后端实现，我们只需要考虑正向传播函数和必需的参数。

首先，我们回顾一下多层感知机（3.3节 ）的代码。下面的代码生成一个网络，其中包含一个具有256个单元和ReLU激活函数的全连接的隐藏层，然后是一个具有10个隐藏单元且不带激活函数的全连接的输出层。

```
from mxnet import np, npx
from mxnet.gluon import nn

npx.set_np()

net = nn.Sequential()
```

(continues on next page)

(continued from previous page)

```
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()

X = np.random.uniform(size=(2, 20))
net(X)
```

```
array([[ 0.06240274, -0.03268593,  0.02582653,  0.02254181, -0.03728798,
        -0.04253785,  0.00540612, -0.01364185, -0.09915454, -0.02272737],
       [ 0.02816679, -0.03341204,  0.03565665,  0.02506384, -0.04136416,
        -0.04941844,  0.01738529,  0.01081963, -0.09932579, -0.01176296]])
```

在这个例子中，我们通过实例化`nn.Sequential`来构建我们的模型，返回的对象赋给net变量。接下来，我们反复调用net变量的add函数，按照想要执行的顺序添加层。简而言之，`nn.Sequential`定义了一种特殊类型的`Block`，即在Gluon中表示块的类。它维护`Block`的有序列表。add函数方便将每个连续的`Block`添加到列表中。请注意，每层都是`Dense`类的一个实例，`Dense`类本身就是`Block`的子类。正向传播（`forward`）函数也非常简单：它将列表中的每个`Block`连接在一起，将每个`Block`的输出作为输入传递给下一层。注意，到目前为止，我们一直在通过`net(X)`调用我们的模型来获得模型的输出。这实际上是`net.forward(X)`的简写，这是通过`Block`类的`__call__`函数实现的一个Python技巧。

4.1.1 自定义块

要想直观地了解块是如何工作的，最简单的方法可能就是自己实现一个。在实现我们自定义块之前，我们简要总结一下每个块必须提供的基本功能：

1. 将输入数据作为其正向传播函数的参数。
2. 通过正向传播函数来生成输出。请注意，输出的形状可能与输入的形状不同。例如，我们上面模型中的第一个全连接的层接收任意维的输入，但是返回一个维度256的输出。
3. 计算其输出关于输入的梯度，可通过其反向传播函数进行访问。通常这是自动发生的。
4. 存储和访问正向传播计算所需的参数。
5. 根据需要初始化模型参数。

在下面的代码片段中，我们从零开始编写一个块。它包含一个多层感知机，其具有256个隐藏单元的隐藏层和一个10维输出层。注意，下面的MLP类继承了表示块的类。我们的实现将严重依赖父类，只需要提供我们自己的构造函数（Python中的`__init__`函数）和正向传播函数。

```
class MLP(nn.Block):
    # 用模型参数声明层。这里，我们声明两个全连接的层
    def __init__(self, **kwargs):
        # 调用`MLP`的父类`Block`的构造函数来执行必要的初始化。
        # 这样，在类实例化时也可以指定其他函数参数，例如模型参数`params`（稍后将介绍）
```

(continues on next page)

(continued from previous page)

```
        super().__init__(**kwargs)
        self.hidden = nn.Dense(256, activation='relu')  # 隐藏层
        self.out = nn.Dense(10)  # 输出层

    # 定义模型的正向传播，即如何根据输入`X`返回所需的模型输出
    def forward(self, X):
        return self.out(self.hidden(X))
```

让我们首先关注正向传播函数。注意，它以X作为输入，计算带有激活函数的隐藏表示，并输出其未归一化的输出值。在这个MLP实现中，两个层都是实例变量。要了解这为什么是合理的，可以想象实例化两个多层感知机（net1和net2），并根据不同的数据对它们进行训练。当然，我们希望它们学到两种不同的模型。

我们在构造函数中实例化多层感知机的层，然后在每次调用正向传播函数时调用这些层。注意一些关键细节。首先，我们定制的__init__函数通过super().__init__()调用父类的__init__函数，省去了重复编写适用于大多数块的模版代码的痛苦。然后我们实例化两个全连接层，分别为self.hidden和self.out。注意，除非我们实现一个新的运算符，否则我们不必担心反向传播函数或参数初始化，系统将自动生成这些。让我们试一下。

```
net = MLP()
net.initialize()
net(X)
```

```
array([[-0.03989595, -0.10414709,  0.06799038,  0.05245074,  0.0252606 ,
        -0.00640342,  0.04182098, -0.01665318, -0.02067345, -0.07863816],
       [-0.03612847, -0.07210435,  0.09159479,  0.07890773,  0.02494171,
        -0.01028665,  0.01732427, -0.02843244,  0.03772651, -0.06671703]])
```

块抽象的一个主要优点是它的多功能性。我们可以子类化块以创建层（如全连接层的类）、整个模型（如上面的MLP类）或具有中等复杂度的各种组件。我们在接下来的章节中充分利用了这种多功能性，比如在处理卷积神经网络时。

4.1.2 顺序块

现在我们可以更仔细地看看Sequential类是如何工作的。回想一下Sequential的设计是为了把其他模块串起来。为了构建我们自己的简化的MySequential，我们只需要定义两个关键函数：1. 一种将块逐个追加到列表中的函数。2. 一种正向传播函数，用于将输入按追加块的顺序传递给块组成的“链条”。

下面的MySequential类提供了与默认Sequential类相同的功能。

```
class MySequential(nn.Block):
    def add(self, block):
        # 这里，`block`是`Block`子类的一个实例，我们假设它有一个唯一的名称。我们把它保存在'Block'类
的成# 员变量`_children` 中。`block`的类型是OrderedDict。当`MySequential`实例调用`initialize`
```

(continues on next page)

(continued from previous page)

```
        # 函数时，系统会自动初始化`_children`的所有成员
        self._children[block.name] = block

    def forward(self, X):
        # OrderedDict保证了按照成员添加的顺序遍历它们
        for block in self._children.values():
            X = block(X)
        return X
```

add函数向有序字典_children添加一个块。你可能会想知道为什么每个Gluon中的Block都有一个_children属性，以及为什么我们使用它而不是自己定义一个Python列表。简而言之，_children的主要优点是，在块的参数初始化过程中，Gluon知道在_children字典中查找需要初始化参数的子块。

当MySequential的正向传播函数被调用时，每个添加的块都按照它们被添加的顺序执行。现在可以使用我们的MySequential类重新实现多层感知机。

```
net = MySequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(X)
```

```
array([[-0.0764568 , -0.01130233,  0.04952145, -0.04651389, -0.04131571,
        -0.05884131, -0.06213811,  0.01311471, -0.01379425, -0.02514282],
       [-0.05124623,  0.00711232, -0.00155933, -0.07555379, -0.06675334,
        -0.01762914,  0.00589085,  0.0144719 , -0.04330775,  0.03317727]])
```

注意，MySequential的用法与之前为Sequential类编写的代码相同（如 3.3节 中所述）。

4.1.3 在正向传播函数中执行代码

Sequential类使模型构造变得简单，允许我们组合新的结构，而不必定义自己的类。然而，并不是所有的架构都是简单的顺序结构。当需要更大的灵活性时，我们需要定义自己的块。例如，我们可能希望在正向传播函数中执行Python的控制流。此外，我们可能希望执行任意的数学运算，而不是简单地依赖预定义的神经网络层。

你可能已经注意到，到目前为止，我们网络中的所有操作都对网络的激活值及网络的参数起作用。然而，有时我们可能希望合并既不是上一层的结果也不是可更新参数的项。我们称之为常数参数（constant parameters）。例如，我们需要一个计算函数$f(\mathbf{x},\mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}$的层，其中$\mathbf{x}$是输入，$\mathbf{w}$是我们的参数，$c$是某个在优化过程中没有更新的指定常量。因此我们实现了一个FixedHiddenMLP类，如下所示。

```
class FixedHiddenMLP(nn.Block):
    def __init__(self, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        super().__init__(**kwargs)
        # 使用`get_constant`函数创建的随机权重参数在训练期间不会更新（即为常量参数）。
        self.rand_weight = self.params.get_constant(
            'rand_weight', np.random.uniform(size=(20, 20)))
        self.dense = nn.Dense(20, activation='relu')

    def forward(self, X):
        X = self.dense(X)
        # 使用创建的常量参数以及`relu`和`dot`函数。
        X = npx.relu(np.dot(X, self.rand_weight.data()) + 1)
        # 复用全连接层。这相当于两个全连接层共享参数。
        X = self.dense(X)
        # 控制流
        while np.abs(X).sum() > 1:
            X /= 2
        return X.sum()
```

在这个`FixedHiddenMLP`模型中，我们实现了一个隐藏层，其权重（`self.rand_weight`）在实例化时被随机初始化，之后为常量。这个权重不是一个模型参数，因此它永远不会被反向传播更新。然后，网络将这个固定层的输出通过一个全连接层。

注意，在返回输出之前，我们的模型做了一些不寻常的事情。我们运行了一个while循环，在L_1范数大于1的条件下，将输出向量除以2，直到它满足条件为止。最后，我们返回了`X`中所有项的和。据我们所知，没有标准的神经网络执行这种操作。注意，此特定操作在任何实际任务中可能都没有用处。我们的重点只是向你展示如何将任意代码集成到神经网络计算的流程中。

```
net = FixedHiddenMLP()
net.initialize()
net(X)
```

```
array(0.52637565)
```

我们可以混合搭配各种组合块的方法。在下面的例子中，我们以一些想到的的方法嵌套块。

```
class NestMLP(nn.Block):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.net = nn.Sequential()
        self.net.add(nn.Dense(64, activation='relu'),
                     nn.Dense(32, activation='relu'))
        self.dense = nn.Dense(16, activation='relu')

    def forward(self, X):
```

(continues on next page)

(continued from previous page)

```
        return self.dense(self.net(X))

chimera = nn.Sequential()
chimera.add(NestMLP(), nn.Dense(20), FixedHiddenMLP())
chimera.initialize()
chimera(X)
```

```
array(0.9772054)
```

4.1.4 效率

热心的读者可能会开始担心其中一些操作的效率。毕竟，我们在一个应该是高性能的深度学习库中进行了大量的字典查找、代码执行和许多其他的Python代码。Python的问题全局解释器锁[68]是众所周知的。在深度学习环境中，我们担心速度极快的GPU可能要等到CPU运行Python代码后才能运行另一个作业。提高Python速度的最好方法是完全避免使用Python。

Gluon这样做的一个方法是允许混合式编程（hybridization），这将在后面描述。Python解释器在第一次调用块时执行它。Gluon运行时记录正在发生的事情，以及下一次它将对Python调用加速。在某些情况下，这可以大大加快速度，但当控制流（如上所述）在不同的网络通路上引导不同的分支时，需要格外小心。我们建议感兴趣的读者在读完本章后，阅读混合式编程部分（sec_hybridize）来了解编译。

4.1.5 小结

- 层也是块。
- 一个块可以由许多层组成。
- 一个块可以由许多块组成。
- 块可以包含代码。
- 块负责大量的内部处理，包括参数初始化和反向传播。
- 层和块的顺序连接由Sequential块处理。

[68] https://wiki.python.org/moin/GlobalInterpreterLock

4.1.6 练习

1. 如果将`MySequential`中存储块的方式更改为Python列表，会出现什么样的问题？
2. 实现一个块，它以两个块为参数，例如`net1`和`net2`，并返回正向传播中两个网络的串联输出。这也被称为平行块。
3. 假设你想要连接同一网络的多个实例。实现一个工厂函数，该函数生成同一个块的多个实例，并在此基础上构建更大的网络。

Discussions[69]

4.2 参数管理

一旦我们选择了架构并设置了超参数，我们就进入了训练阶段。此时，我们的目标是找到使损失函数最小化的参数值。经过训练后，我们将需要使用这些参数来做出未来的预测。此外，有时我们希望提取参数，以便在其他环境中复用它们，将模型保存到磁盘，以便它可以在其他软件中执行，或者为了获得科学的理解而进行检查。

大多数情况下，我们可以忽略声明和操作参数的具体细节，而只依靠深度学习框架来完成繁重的工作。然而，当我们离开具有标准层的层叠架构时，我们有时会陷入声明和操作参数的麻烦中。在本节中，我们将介绍以下内容：

- 访问参数，用于调试、诊断和可视化。
- 参数初始化。
- 在不同模型组件间共享参数。

我们首先关注具有单隐藏层的多层感知机。

```
from mxnet import init, np, npx
from mxnet.gluon import nn

npx.set_np()

net = nn.Sequential()
net.add(nn.Dense(8, activation='relu'))
net.add(nn.Dense(1))
net.initialize()  # 使用默认初始化方法

X = np.random.uniform(size=(2, 4))
net(X)  # 正向计算
```

[69] https://discuss.d2l.ai/t/1828

```
array([[0.0054572 ],
       [0.00488594]])
```

4.2.1 参数访问

我们从已有模型中访问参数。当通过Sequential类定义模型时，我们可以通过索引来访问模型的任意层。这就像模型是一个列表一样。每层的参数都在其属性中。如下所示，我们可以检查第二个全连接层的参数。

```
print(net[1].params)
```

```
dense1_ (
  Parameter dense1_weight (shape=(1, 8), dtype=float32)
  Parameter dense1_bias (shape=(1,), dtype=float32)
)
```

输出的结果告诉我们一些重要的事情。首先，这个全连接层包含两个参数，分别是该层的权重和偏置。两者都存储为单精度浮点数（float32）。注意，参数名称允许我们唯一地标识每个参数，即使在包含数百个层的网络中也是如此。

目标参数

注意，每个参数都表示为参数（parameter）类的一个实例。要对参数执行任何操作，首先我们需要访问底层的数值。有几种方法可以做到这一点。有些比较简单，而另一些则比较通用。下面的代码从第二个神经网络层提取偏置，提取后返回的是一个参数类实例，并进一步访问该参数的值。

```
print(type(net[1].bias))
print(net[1].bias)
print(net[1].bias.data())
```

```
<class 'mxnet.gluon.parameter.Parameter'>
Parameter dense1_bias (shape=(1,), dtype=float32)
[0.]
```

参数是复合的对象，包含值、梯度和额外信息。这就是为什么我们需要显式请求值的原因。

除了值之外，我们还可以访问每个参数的梯度。由于我们还没有调用这个网络的反向传播，所以参数的梯度处于初始状态。

```
net[1].weight.grad()
```

```
array([[0., 0., 0., 0., 0., 0., 0., 0.]])
```

一次性访问所有参数

当我们需要对所有参数执行操作时，逐个访问它们可能会很麻烦。当我们处理更复杂的块（例如，嵌套块）时，情况可能会变得特别复杂，因为我们需要递归整个树来提取每个子块的参数。下面，我们将通过演示来比较访问第一个全连接层的参数和访问所有层。

```
print(net[0].collect_params())
print(net.collect_params())
```

```
dense0_ (
  Parameter dense0_weight (shape=(8, 4), dtype=float32)
  Parameter dense0_bias (shape=(8,), dtype=float32)
)
sequential0_ (
  Parameter dense0_weight (shape=(8, 4), dtype=float32)
  Parameter dense0_bias (shape=(8,), dtype=float32)
  Parameter dense1_weight (shape=(1, 8), dtype=float32)
  Parameter dense1_bias (shape=(1,), dtype=float32)
)
```

这为我们提供了另一种访问网络参数的方式，如下所示。

```
net.collect_params()['dense1_bias'].data()
```

```
array([0.])
```

从嵌套块收集参数

让我们看看，如果我们将多个块相互嵌套，参数命名约定是如何工作的。为此，我们首先定义一个生成块的函数（可以说是块工厂），然后将这些块组合到更大的块中。

```
def block1():
    net = nn.Sequential()
    net.add(nn.Dense(32, activation='relu'))
    net.add(nn.Dense(16, activation='relu'))
    return net

def block2():
    net = nn.Sequential()
    for _ in range(4):
        # Nested here
        net.add(block1())
    return net
```

(continues on next page)

(continued from previous page)

```
rgnet = nn.Sequential()
rgnet.add(block2())
rgnet.add(nn.Dense(10))
rgnet.initialize()
rgnet(X)
```

```
array([[-6.3465846e-09, -1.1096752e-09,  6.4161787e-09,  6.6354140e-09,
        -1.1265507e-09,  1.3284951e-10,  9.3619388e-09,  3.2229084e-09,
         5.9429879e-09,  8.8181435e-09],
       [-8.6219423e-09, -7.5150686e-10,  8.3133251e-09,  8.9321128e-09,
        -1.6740003e-09,  3.2405989e-10,  1.2115976e-08,  4.4926449e-09,
         8.0741742e-09,  1.2075874e-08]])
```

现在我们已经设计了网络，让我们看看它是如何组织的。

```
print(rgnet.collect_params)
print(rgnet.collect_params())
```

```
<bound method Block.collect_params of Sequential(
  (0): Sequential(
    (0): Sequential(
      (0): Dense(4 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (1): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (2): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (3): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
  )
  (1): Dense(16 -> 10, linear)
)>
sequential1_ (
  Parameter dense2_weight (shape=(32, 4), dtype=float32)
  Parameter dense2_bias (shape=(32,), dtype=float32)
```

(continues on next page)

(continued from previous page)

```
  Parameter dense3_weight (shape=(16, 32), dtype=float32)
  Parameter dense3_bias (shape=(16,), dtype=float32)
  Parameter dense4_weight (shape=(32, 16), dtype=float32)
  Parameter dense4_bias (shape=(32,), dtype=float32)
  Parameter dense5_weight (shape=(16, 32), dtype=float32)
  Parameter dense5_bias (shape=(16,), dtype=float32)
  Parameter dense6_weight (shape=(32, 16), dtype=float32)
  Parameter dense6_bias (shape=(32,), dtype=float32)
  Parameter dense7_weight (shape=(16, 32), dtype=float32)
  Parameter dense7_bias (shape=(16,), dtype=float32)
  Parameter dense8_weight (shape=(32, 16), dtype=float32)
  Parameter dense8_bias (shape=(32,), dtype=float32)
  Parameter dense9_weight (shape=(16, 32), dtype=float32)
  Parameter dense9_bias (shape=(16,), dtype=float32)
  Parameter dense10_weight (shape=(10, 16), dtype=float32)
  Parameter dense10_bias (shape=(10,), dtype=float32)
)
```

因为层是分层嵌套的，所以我们也可以像通过嵌套列表索引一样访问它们。例如，我们下面访问第一个主要的块，其中第二个子块的第一层的偏置项。

```
rgnet[0][1][0].bias.data()
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

4.2.2 参数初始化

我们知道了如何访问参数，现在让我们看看如何正确地初始化参数。我们在 3.8节 中讨论了良好初始化的必要性。深度学习框架提供默认随机初始化。然而，我们经常希望根据其他规则初始化权重。深度学习框架提供了最常用的规则，也允许创建自定义初始化方法。

默认情况下，MXNet通过初始化权重参数的方法是从均匀分布$U(-0.07, 0.07)$中随机采样权重，并将偏置参数设置为0。MXNet的`init`模块提供了多种预置初始化方法。

内置初始化

让我们首先调用内置的初始化器。下面的代码将所有权重参数初始化为标准差为0.01的高斯随机变量，且将偏置参数设置为0。

```
# 这里的`force_reinit`确保参数始终会被重新初始化，而不论之前是否已经初始化
net.initialize(init=init.Normal(sigma=0.01), force_reinit=True)
net[0].weight.data()[0]
```

```
array([-0.00324057, -0.00895028, -0.00698632,  0.01030831])
```

我们还可以将所有参数初始化为给定的常数（比如1）。

```
net.initialize(init=init.Constant(1), force_reinit=True)
net[0].weight.data()[0]
```

```
array([1., 1., 1., 1.])
```

我们还可以对某些块应用不同的初始化方法。例如，下面我们使用Xavier初始化方法初始化第一层，然后第二层初始化为常量值42。

```
net[0].weight.initialize(init=init.Xavier(), force_reinit=True)
net[1].initialize(init=init.Constant(42), force_reinit=True)
print(net[0].weight.data()[0])
print(net[1].weight.data())
```

```
[-0.17594433  0.02314097 -0.1992535   0.09509248]
[[42. 42. 42. 42. 42. 42. 42. 42.]]
```

自定义初始化

有时，深度学习框架没有提供我们需要的初始化方法。在下面的例子中，我们使用以下的分布为任意权重参数w定义初始化方法：

$$w \sim \begin{cases} U(5, 10) & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{1}{2} \\ U(-10, -5) & \text{with probability } \frac{1}{4} \end{cases} \tag{4.2.1}$$

在这里，我们定义了`Initializer`类的子类。通常，我们只需要实现`_init_weight`函数，该函数接受张量参数（`data`）并为其分配所需的初始化值。

```
class MyInit(init.Initializer):
    def _init_weight(self, name, data):
```

(continues on next page)

(continued from previous page)

```
        print('Init', name, data.shape)
        data[:] = np.random.uniform(-10, 10, data.shape)
        data *= np.abs(data) >= 5

net.initialize(MyInit(), force_reinit=True)
net[0].weight.data()[:2]
```

```
Init dense0_weight (8, 4)
Init dense1_weight (1, 8)
```

```
array([[ 0.        , -0.        , -0.        ,  8.522827 ],
       [ 0.        , -8.828651  , -0.        , -5.6012006]])
```

注意，我们始终可以直接设置参数。

```
net[0].weight.data()[:] += 1
net[0].weight.data()[0, 0] = 42
net[0].weight.data()[0]
```

```
array([42.        ,  1.        ,  1.        ,  9.522827])
```

高级用户请注意：如果要在autograd范围内调整参数，则需要使用set_data，以避免误导自动微分机制。

4.2.3 参数绑定

有时我们希望在多个层间共享参数。让我们看看如何优雅地做这件事。在下面，我们定义一个稠密层，然后使用它的参数来设置另一个层的参数。

```
net = nn.Sequential()
# 我们需要给共享层一个名称，以便可以引用它的参数。
shared = nn.Dense(8, activation='relu')
net.add(nn.Dense(8, activation='relu'), shared,
        nn.Dense(8, activation='relu', params=shared.params), nn.Dense(10))
net.initialize()

X = np.random.uniform(size=(2, 20))
net(X)

# 检查参数是否相同
print(net[1].weight.data()[0] == net[2].weight.data()[0])
net[1].weight.data()[0, 0] = 100
```

(continues on next page)

(continued from previous page)

```
# 确保它们实际上是同一个对象，而不只是有相同的值。
print(net[1].weight.data()[0] == net[2].weight.data()[0])
```

```
[ True  True  True  True  True  True  True  True]
[ True  True  True  True  True  True  True  True]
```

这个例子表明第二层和第三层的参数是绑定的。它们不仅值相等，而且由相同的张量表示。因此，如果我们改变其中一个参数，另一个参数也会改变。你可能会想，当参数绑定时，梯度会发生什么情况？答案是由于模型参数包含梯度，因此在反向传播期间第二个隐藏层和第三个隐藏层的梯度会加在一起。

4.2.4 小结

- 我们有几种方法可以访问、初始化和绑定模型参数。
- 我们可以使用自定义初始化方法。

4.2.5 练习

1. 使用 4.1节 中定义的FancyMLP模型，访问各个层的参数。
2. 查看初始化模块文档以了解不同的初始化方法。
3. 构建包含共享参数层的多层感知机并对其进行训练。在训练过程中，观察模型各层的参数和梯度。
4. 为什么共享参数是个好主意？

Discussions[70]

4.3 延后初始化

到目前为止，似乎我们在建立网络时表现得很简单。具体来说，我们忘记了做以下这些事情，似乎无法成功建立网络：

- 我们定义了网络架构，但没有指定输入维度。
- 我们添加层时没有指定前一层的输出维度。
- 我们在初始化参数时，甚至没有足够的信息来确定模型应该包含多少参数。

你可能会对我们的代码能运行感到惊讶。毕竟，深度学习框架无法判断网络的输入维度是什么。这里的诀窍是框架的延后初始化（defers initialization），即等到我们第一次将数据通过模型传递时，才会动态地推断出每个层的大小。

[70] https://discuss.d2l.ai/t/1831

在以后，当使用卷积神经网络时，由于输入维度（即图像的分辨率）将影响每个后续层的维数，因此该技术将变得更加方便。现在我们在编写代码时无需知道维度是什么就可以设置参数，这种能力可以大大简化定义和修改模型的任务。接下来，我们将更深入地研究初始化机制。

4.3.1 实例化网络

首先，让我们实例化一个多层感知机。

```
from mxnet import np, npx
from mxnet.gluon import nn

npx.set_np()

def get_net():
    net = nn.Sequential()
    net.add(nn.Dense(256, activation='relu'))
    net.add(nn.Dense(10))
    return net

net = get_net()
```

此时，因为输入维数是未知的，所以网络不可能知道输入层权重的维数。因此，框架尚未初始化任何参数。我们通过尝试访问以下参数进行确认。

```
print(net.collect_params)
print(net.collect_params())
```

```
<bound method Block.collect_params of Sequential(
  (0): Dense(-1 -> 256, Activation(relu))
  (1): Dense(-1 -> 10, linear)
)>
sequential0_ (
  Parameter dense0_weight (shape=(256, -1), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, -1), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

注意，当参数对象存在时，每个层的输入维度为-1。MXNet使用特殊值-1表示参数维度仍然未知。此时，尝试访问`net[0].weight.data()`将触发运行时错误，提示必须先初始化网络，然后才能访问参数。现在让我们看看当我们试图通过`initialize`函数初始化参数时会发生什么。

```
net.initialize()
net.collect_params()
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, -1), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, -1), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

如我们所见，一切都没有改变。当输入维度未知时，调用`initialize`不会真正初始化参数。而是会在MXNet内部声明希望初始化参数，并且可以选择初始化分布。

接下来让我们将数据通过网络，最终使框架初始化参数。

```
X = np.random.uniform(size=(2, 20))
net(X)

net.collect_params()
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

一旦我们知道输入维数是20，框架可以通过代入值20来识别第一层权重矩阵的形状。识别出第一层的形状后，框架处理第二层，依此类推，直到所有形状都已知为止。注意，在这种情况下，只有第一层需要延迟初始化，但是框架仍是按顺序初始化的。等到知道了所有的参数形状，框架就可以初始化参数。

4.3.2 小结

- 延后初始化可以很方便地通过框架自动推断参数形状，使修改模型结构变得容易，避免了一类常见的错误。
- 我们可以通过模型传递数据，使框架最终初始化参数。

4.3.3 练习

1. 如果您指定了第一层的输入尺寸，但没有指定后续层的尺寸，会发生什么？是否立即进行初始化？
2. 如果指定了不匹配的维度会发生什么？
3. 如果输入具有不同的维度，你需要做什么？提示：查看参数绑定的相关内容。

Discussions[71]

[71] https://discuss.d2l.ai/t/1834

4.4 自定义层

深度学习成功背后的一个因素是，可以用创造性的方式组合广泛的层，从而设计出适用于各种任务的结构。例如，研究人员发明了专门用于处理图像、文本、序列数据和执行动态编程的层。早晚有一天，你会遇到或要自己发明一个在深度学习框架中还不存在的层。在这些情况下，你必须构建自定义层。在本节中，我们将向你展示如何操作。

4.4.1 不带参数的层

首先，我们构造一个没有任何参数的自定义层。如果你还记得我们在4.1节 对块的介绍，这应该看起来很眼熟。下面的CenteredLayer类要从其输入中减去均值。要构建它，我们只需继承基础层类并实现正向传播功能。

```
from mxnet import np, npx
from mxnet.gluon import nn

npx.set_np()

class CenteredLayer(nn.Block):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def forward(self, X):
        return X - X.mean()
```

让我们通过向其提供一些数据来验证该层是否按预期工作。

```
layer = CenteredLayer()
layer(np.array([1, 2, 3, 4, 5]))
```

```
array([-2., -1.,  0.,  1.,  2.])
```

现在，我们可以将层作为组件合并到构建更复杂的模型中。

```
net = nn.Sequential()
net.add(nn.Dense(128), CenteredLayer())
net.initialize()
```

作为额外的健全性检查，我们可以向网络发送随机数据后，检查均值是否为0。由于我们处理的是浮点数，因为存储精度的原因，我们仍然可能会看到一个非常小的非零数。

```
Y = net(np.random.uniform(size=(4, 8)))
Y.mean()
```

```
array(3.783498e-10)
```

4.4.2 带参数的图层

既然我们知道了如何定义简单的层，那么让我们继续定义具有参数的层，这些参数可以通过训练进行调整。我们可以使用内置函数来创建参数，这些参数提供一些基本的管理功能。比如管理访问、初始化、共享、保存和加载模型参数。这样做的好处之一是，我们不需要为每个自定义层编写自定义序列化程序。

现在，让我们实现自定义版本的全连接层。回想一下，该层需要两个参数，一个用于表示权重，另一个用于表示偏置项。在此实现中，我们使用ReLU作为激活函数。该层需要输入参数：in_units和units，分别表示输入和输出的数量。

```
class MyDense(nn.Block):
    def __init__(self, units, in_units, **kwargs):
        super().__init__(**kwargs)
        self.weight = self.params.get('weight', shape=(in_units, units))
        self.bias = self.params.get('bias', shape=(units,))

    def forward(self, x):
        linear = np.dot(
            x, self.weight.data(ctx=x.ctx)) + self.bias.data(ctx=x.ctx)
        return npx.relu(linear)
```

接下来，我们实例化MyDense类并访问其模型参数。

```
dense = MyDense(units=3, in_units=5)
dense.params
```

```
mydense0_ (
  Parameter mydense0_weight (shape=(5, 3), dtype=<class 'numpy.float32'>)
  Parameter mydense0_bias (shape=(3,), dtype=<class 'numpy.float32'>)
)
```

我们可以使用自定义层直接执行正向传播计算。

```
dense.initialize()
dense(np.random.uniform(size=(2, 5)))
```

```
array([[0.        , 0.01633355, 0.        ],
       [0.        , 0.01581812, 0.        ]])
```

我们还可以使用自定义层构建模型。我们可以像使用内置的全连接层一样使用自定义层。

```
net = nn.Sequential()
net.add(MyDense(8, in_units=64), MyDense(1, in_units=8))
net.initialize()
net(np.random.uniform(size=(2, 64)))
```

```
array([[0.06508517],
       [0.0615553 ]])
```

4.4.3 小结

- 我们可以通过基本层类设计自定义层。这允许我们定义灵活的新层，其行为与库中的任何现有层不同。
- 在自定义层定义完成后，就可以在任意环境和网络结构中调用该自定义层。
- 层可以有局部参数，这些参数可以通过内置函数创建。

4.4.4 练习

1. 设计一个接受输入并计算张量汇总的层，它返回$y_k = \sum_{i,j} W_{ijk} x_i x_j$。
2. 设计一个返回输入数据的傅立叶系数前半部分的层。

Discussions[72]

4.5 读写文件

到目前为止，我们讨论了如何处理数据，以及如何构建、训练和测试深度学习模型。然而，有时我们对所学的模型足够满意，我们希望保存训练的模型以备将来在各种环境中使用（可能部署进行预测）。此外，当运行一个耗时较长的训练过程时，最佳实践是定期保存中间结果（检查点），以确保在服务器电源被不小心断掉时不会损失几天的计算结果。因此，现在是时候学习如何加载和存储权重向量和整个模型。本节将讨论这些问题。

4.5.1 加载和保存张量

对于单个张量，我们可以直接调用`load`和`save`函数分别读写它们。这两个函数都要求我们提供一个名称，`save`要求将要保存的变量作为输入。

```
from mxnet import np, npx
from mxnet.gluon import nn

npx.set_np()
```

(continues on next page)

[72] https://discuss.d2l.ai/t/1837

(continued from previous page)

```
x = np.arange(4)
npx.save('x-file', x)
```

我们现在可以将存储在文件中的数据读回内存。

```
x2 = npx.load('x-file')
x2
```

```
[array([0., 1., 2., 3.])]
```

我们可以存储一个张量列表，然后把它们读回内存。

```
y = np.zeros(4)
npx.save('x-files', [x, y])
x2, y2 = npx.load('x-files')
(x2, y2)
```

```
(array([0., 1., 2., 3.]), array([0., 0., 0., 0.]))
```

我们甚至可以写入或读取从字符串映射到张量的字典。当我们要读取或写入模型中的所有权重时，这很方便。

```
mydict = {'x': x, 'y': y}
npx.save('mydict', mydict)
mydict2 = npx.load('mydict')
mydict2
```

```
{'x': array([0., 1., 2., 3.]), 'y': array([0., 0., 0., 0.])}
```

4.5.2 加载和保存模型参数

保存单个权重向量（或其他张量）确实是有用的，但是如果我们想保存整个模型，并在以后加载它们。单独保存每个向量则会变得很麻烦。毕竟，我们可能有数百个参数散布在各处。因此，深度学习框架提供了内置函数来保存和加载整个网络。需要注意的一个重要细节是，这将保存模型的参数而不是保存整个模型。例如，如果我们有一个3层多层感知机，我们需要单独指定结构。因为模型本身可以包含任意代码，所以模型本身难以序列化。因此，为了恢复模型，我们需要用代码生成结构，然后从磁盘加载参数。让我们从熟悉的多层感知机开始尝试一下。

```
class MLP(nn.Block):
    def __init__(self, **kwargs):
        super(MLP, self).__init__(**kwargs)
```

(continues on next page)

(continued from previous page)

```
        self.hidden = nn.Dense(256, activation='relu')
        self.output = nn.Dense(10)

    def forward(self, x):
        return self.output(self.hidden(x))

net = MLP()
net.initialize()
X = np.random.uniform(size=(2, 20))
Y = net(X)
```

接下来，我们将模型的参数存储为一个叫做“mlp.params”的文件。

```
net.save_parameters('mlp.params')
```

为了恢复模型，我们实例化了原始多层感知机模型的一个备份。我们没有随机初始化模型参数，而是直接读取文件中存储的参数。

```
clone = MLP()
clone.load_parameters('mlp.params')
```

由于两个实例具有相同的模型参数，在输入相同的X时，两个实例的计算结果应该相同。让我们来验证一下。

```
Y_clone = clone(X)
Y_clone == Y
```

```
array([[ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True]])
```

4.5.3 小结

- save和load函数可用于张量对象的文件读写。
- 我们可以通过参数字典保存和加载网络的全部参数。
- 保存结构必须在代码中完成，而不是在参数中完成。

4.5.4 练习

1. 即使不需要将经过训练的模型部署到不同的设备上，存储模型参数还有什么实际的好处？
2. 假设我们只想复用网络的一部分，以将其合并到不同的网络结构中。比如说，如果你想在一个新的网络中使用之前网络的前两层，你该怎么做？
3. 如何同时保存网络结构和参数？你会对结构加上什么限制？

Discussions[73]

4.6 GPU

在 表1 中，我们讨论了过去20年中计算能力的快速增长。简而言之，自2000年以来，GPU性能每十年增长1000倍。这提供了巨大的机会，但也表明需要提供这样的性能。

在本节中，我们开始讨论如何利用这种计算性能进行研究。首先是使用单个GPU，然后是如何使用多个GPU和多个服务器（具有多个GPU）。

具体来说，我们将讨论如何使用单个NVIDIA GPU进行计算。首先，确保至少安装了一个NVIDIA GPU。然后，下载NVIDIA驱动和CUDA[74]并按照提示设置适当的路径。当这些准备工作完成，就可以使用nvidia-smi命令来查看显卡信息。

```
!nvidia-smi
```

```
Sun Mar  7 22:31:10 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:1B.0 Off |                    0 |
| N/A   49C    P0    52W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:00:1C.0 Off |                    0 |
| N/A   44C    P0    60W / 300W |   1336MiB / 16130MiB |     36%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:00:1D.0 Off |                    0 |
| N/A   59C    P0    61W / 300W |   1314MiB / 16130MiB |     15%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   50C    P0    57W / 300W |      0MiB / 16130MiB |      4%      Default |
```

(continues on next page)

[73] https://discuss.d2l.ai/t/1840

[74] https://developer.nvidia.com/cuda-downloads

(continued from previous page)

```
+-------------------------------+----------------------+----------------------+


+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    1     71148      C   ...conda3/envs/d2l-zh-release-1/bin/python  1325MiB |
|    2     70686      C   ...conda3/envs/d2l-zh-release-1/bin/python  1303MiB |
+-----------------------------------------------------------------------------+
```

你可能已经注意到MXNet张量看起来与NumPy的ndarray几乎相同。但有一些关键区别，其中之一是MXNet支持不同的硬件设备。

在MXNet中，每个数组都有一个上下文（context）。到目前为止，默认情况下，所有变量和相关的计算都分配给CPU。有时上下文可能是GPU。当我们跨多个服务器部署作业时，事情会变得更加棘手。通过智能地将数组分配给上下文，我们可以最大限度地减少在设备之间传输数据的时间。例如，当在带有GPU的服务器上训练神经网络时，我们通常希望模型的参数在GPU上。

接下来，我们需要确认是否安装了MXNet的GPU版本。如果已经安装了MXNet的CPU版本，我们需要先卸载它。例如，使用`pip uninstall mxnet`命令，然后根据你的CUDA版本安装相应的MXNet版本。假设你已经安装了CUDA10.0，你可以通过`pip install mxnet-cu100`安装支持CUDA10.0的MXNet版本。

要运行此部分中的程序，至少需要两个GPU。注意，对于大多数桌面计算机来说，这可能是奢侈的，但在云中很容易获得，例如，通过使用AWS EC2的多GPU实例。本节几乎所有的其他部分都不需要多个GPU。本节只是为了说明数据如何在不同的设备之间传递。

4.6.1 计算设备

我们可以指定用于存储和计算的设备，如CPU和GPU。默认情况下，张量是在内存中创建的，然后使用CPU计算它。

在MXNet中，CPU和GPU可以用`cpu()`和`gpu()`表示。需要注意的是，`cpu()`（或括号中的任意整数）表示所有物理CPU和内存。这意味着MXNet的计算将尝试使用所有CPU核心。然而，`gpu()`只代表一个卡和相应的显存。如果有多个GPU，我们使用`gpu(i)`表示第i块GPU（i从0开始）。另外，`gpu(0)`和`gpu()`是等价的。

```python
from mxnet import np, npx
from mxnet.gluon import nn

npx.set_np()

npx.cpu(), npx.gpu(), npx.gpu(1)
```

```
(cpu(0), gpu(0), gpu(1))
```

我们可以查询可用gpu的数量。

```
npx.num_gpus()
```

```
2
```

现在我们定义了两个方便的函数，这两个函数允许我们在请求的GPU不存在的情况下运行代码。

```
def try_gpu(i=0):  #@save
    """如果存在，则返回gpu(i)，否则返回cpu()。"""
    return npx.gpu(i) if npx.num_gpus() >= i + 1 else npx.cpu()

def try_all_gpus():  #@save
    """返回所有可用的GPU，如果没有GPU，则返回[cpu()]。"""
    devices = [npx.gpu(i) for i in range(npx.num_gpus())]
    return devices if devices else [npx.cpu()]

try_gpu(), try_gpu(10), try_all_gpus()
```

```
(gpu(0), cpu(0), [gpu(0), gpu(1)])
```

4.6.2 张量与gpu

默认情况下，张量是在CPU上创建的。我们可以查询张量所在的设备。

```
x = np.array([1, 2, 3])
x.ctx
```

```
cpu(0)
```

需要注意的是，无论何时我们要对多个项进行操作，它们都必须在同一个设备上。例如，如果我们对两个张量求和，我们需要确保两个张量都位于同一个设备上，否则框架将不知道在哪里存储结果，甚至不知道在哪里执行计算。

存储在GPU上

有几种方法可以在GPU上存储张量。例如，我们可以在创建张量时指定存储设备。接下来，我们在第一个gpu上创建张量变量X。在GPU上创建的张量只消耗这个GPU的显存。我们可以使用`nvidia-smi`命令查看显存使用情况。一般来说，我们需要确保不创建超过GPU显存限制的数据。

```
X = np.ones((2, 3), ctx=try_gpu())
X
```

```
array([[1., 1., 1.],
       [1., 1., 1.]], ctx=gpu(0))
```

假设你至少有两个GPU，下面的代码将在第二个GPU上创建一个随机张量。

```
Y = np.random.uniform(size=(2, 3), ctx=try_gpu(1))
Y
```

```
array([[0.67478997, 0.07540122, 0.9956977 ],
       [0.09488854, 0.415456  , 0.11231736]], ctx=gpu(1))
```

复制

如果我们要计算X + Y，我们需要决定在哪里执行这个操作。例如，如 图4.6.1 所示，我们可以将X传输到第二个GPU并在那里执行操作。不要简单地X加上Y，因为这会导致异常。运行时引擎不知道该怎么做：它在同一设备上找不到数据会导致失败。由于Y位于第二个GPU上，所以我们需要将X移到那里，然后才能添加这两个GPU。

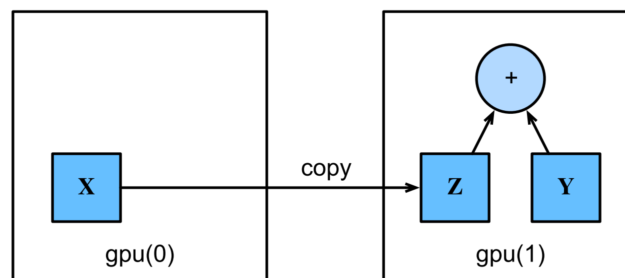


图4.6.1: 复制数据以在同一设备上执行操作。

```
Z = X.copyto(try_gpu(1))
print(X)
print(Z)
```

```
[[1. 1. 1.]
 [1. 1. 1.]] @gpu(0)
[[1. 1. 1.]
 [1. 1. 1.]] @gpu(1)
```

现在数据在同一个GPU上（Z和Y都在），我们可以将它们相加。

```
Y + Z
```

```
array([[1.6747899, 1.0754012, 1.9956977],
       [1.0948886, 1.415456 , 1.1123173]], ctx=gpu(1))
```

假设变量Z已经存在于第二个GPU上。如果我们还是调用`Z.copyto(gpu(1))`怎么办？它将复制并分配新的显存，即使该变量已经存在于所需的设备上。有时，根据代码运行的环境不同，两个变量可能已经存在于同一设备上。因此，我们只想在变量存在于不同设备中时进行复制。在这种情况下，我们可以调用`as_in_ctx`。如果变量已经存在于指定的设备中，则这不会进行任何操作。除非你特别想创建一个复制，否则选择`as_in_ctx`方法。

```
Z.as_in_ctx(try_gpu(1)) is Z
```

```
True
```

旁注

人们使用GPU来进行机器学习，因为他们希望运行速度快。但是在设备之间传输变量是缓慢的。所以我们希望你百分之百确定你想做一些缓慢的事情。如果深度学习框架只是自动复制而没有崩溃，那么你可能不会意识到你已经编写了一些缓慢的代码。

此外，在设备（CPU、GPU和其他机器）之间传输数据比计算慢得多。这也使得并行化变得更加困难，因为我们必须等待数据被发送（或者接收），然后才能继续进行更多的操作。这就是为什么拷贝操作要格外小心。根据经验，许多小手术比一个大手术糟糕得多。此外，除非你知道自己在做什么。否则，一次执行几个操作比代码中散布的许多单个操作要好得多。如果一个设备必须等待另一个设备才能执行其他操作，那么这样的操作可能会阻塞。这有点像排队订购咖啡，而不像通过电话预先订购时，当你在的时候发现咖啡已经准备好了。

最后，当我们打印张量或将张量转换为NumPy格式时。如果数据不在内存中，框架会首先将其复制到内存中，这会导致额外的传输开销。更糟糕的是，它现在受制于可怕的全局解释器锁，这使得一切都得等待Python完成。

4.6.3 神经网络与GPU

类似地，神经网络模型可以指定设备。下面的代码将模型参数放在GPU上。

```
net = nn.Sequential()
net.add(nn.Dense(1))
net.initialize(ctx=try_gpu())
```

在接下来的几章中，我们将看到更多关于如何在GPU上运行模型的例子，因为它们将变得更加计算密集。

当输入为GPU上的张量时，模型将在同一GPU上计算结果。

```
net(X)
```

```
array([[0.04995865],
       [0.04995865]], ctx=gpu(0))
```

让我们确认模型参数存储在同一个GPU上。

```
net[0].weight.data().ctx
```

```
gpu(0)
```

总之，只要所有的数据和参数都在同一个设备上，我们就可以有效地学习模型。在下面的章节中，我们将看到几个这样的例子。

4.6.4 小结

- 我们可以指定用于存储和计算的设备，例如CPU或GPU。默认情况下，数据在主内存中创建，然后使用CPU进行计算。
- 深度学习框架要求计算的所有输入数据都在同一设备上，无论是CPU还是GPU。
- 不经意地移动数据可能会显著降低性能。一个典型的错误如下：计算GPU上每个小批量的损失，并在命令行中将其报告给用户（或将其记录在NumPy `ndarray`中）时，将触发全局解释器锁，从而使所有GPU阻塞。最好是为GPU内部的日志分配内存，并且只移动较大的日志。

4.6.5 练习

1. 尝试一个更大的计算任务，比如大矩阵的乘法，看看CPU和GPU之间的速度差异。一个计算量很小的任务呢？
2. 我们应该如何在GPU上读写模型参数？
3. 测量计算1000个100×100矩阵的矩阵乘法所需的时间，并记录输出矩阵的弗罗贝尼乌斯范数，一次记录一个结果，而不是在GPU上保存日志并仅传输最终结果。
4. 测量同时在两个GPU上执行两个矩阵乘法与在一个GPU上按顺序执行两个矩阵乘法所需的时间。提示：你应该看到近乎线性的缩放。

Discussions[75]

[75] https://discuss.d2l.ai/t/1843

5

卷积神经网络

在前面的章节中，我们曾讨论过图像数据。对于图像数据，每个样本都由一个二维像素网格组成。取决于我们处理的是黑白图像还是彩色图像，每个像素可能与一个或多个数值相关。到目前为止，我们处理二维结构的方式还十分有限：先将每个图像的空间结构展平成一维向量，再放入一个多层感知机中。由于这些网络相对于特征元素的顺序不变，所以无论我们保持像素在二维空间的顺序，或是在拟合多层感知机的参数之前对二维矩阵的列进行交换，我们都可以训练相似的神经网络。而最优情况是利用先验知识，即利用相近像素之间的相互关联性，建立图像数据的有效模型。

本章介绍的卷积神经网络（convolutional neural network，CNN）是一类强大的神经网络，正是为处理图像数据而设计的。基于卷积神经网络结构的模型在计算机视觉领域中已经占主导地位，当今几乎所有的图像识别、对象检测或语义分割相关的学术竞赛、商业应用都以这种方法为基础。

现代卷积神经网络的设计得益于生物学、群论和大量的实验研究。除了在获得精确模型的采样效率外，卷积神经网络在计算上也是极其高效的。这是因为卷积神经网络需要的参数比多层感知机少，而且卷积神经网络很容易用GPU并行计算。因此，实践者经常尽可能多地应用卷积神经网络。即使在一维序列结构的任务上（例如音频、文本和时间序列分析），通常大家使用的是循环神经网络，而实践者也会经常使用到卷积神经网络。通过对卷积神经网络一些巧妙的调整，也使它们在图结构数据和推荐系统中发挥作用。

在本章的开始，我们将介绍构成所有卷积网络主干的基本元素。这包括卷积层本身、填充（padding）和步幅（stride）、用于在相邻空间区域聚集信息的池化层（pooling）、每层中多通道（channel）的使用以及有关现代卷积网络架构的全面介绍。在本章的最后，我们将介绍一个完整的、可运行的LeNet模型：这是第一个卷积神经网络，早在现代深度学习兴起之前就已经得到成功应用。在下一章中，我们将深入研究一些流行的、相对较新并具有一定代表性的卷积网络架构。

5.1 从全连接层到卷积

我们之前讨论的多层感知机十分适合处理表格数据。表格数据中的每行对应每个样本，每列分别对应每个特征。这些特征之间的交互可能产生影响，但我们没有考虑特征交互结构上的先验假设。

有时我们缺乏足够的知识来指导更巧妙的模型结构设计，此时多层感知机可能是最好的选择。然而，对于高维感知数据，这种无结构网络可能会变得笨拙。

例如，在之前区分猫和狗的例子中。假设我们收集了一个照片数据集，每张照片具有百万级像素，这意味着多层感知机的每次输入都有一百万个维度。根据我们在 2.4.3节 中对全连接层参数开销的讨论。即使将隐藏层维度降低到 1000 ，这个神经网络也将有 $10^6 \times 10^3 = 10^9$ 个参数。想要训练这个模型很难，需要有大量的GPU、分布式优化训练的经验和超乎常人的耐心。

细心的读者可能会反对这一论点，认为要求百万像素的分辨率可能不是必要的。然而，即使减小为十万像素，1000 个隐藏单元的隐藏层也可能不足以学习到良好的图像特征，所以我们仍然需要数十亿个参数。此外，拟合如此多的参数还需要收集大量的数据。然而，如今人类视觉和传统机器学习模型都能很好地区分猫和狗。这是因为图像中有丰富的结构，人类和机器学习模型都可以利用这些结构。卷积神经网络（convolutional neural networks，CNN）是机器学习利用自然图像中一些已知结构的创造性方法。

5.1.1 不变性

想象一下，在一个目标检测任务中，我们不应过分在意图像中物体的确切位置，比如猪通常不在天上飞，飞机通常不游泳。我们可以从儿童游戏”沃尔多在哪里”（图5.1.1 ）中汲取一些灵感。这个游戏包括一些混乱的场景，游戏玩家的目标是找到沃尔多，而沃尔多通常潜伏在一些不太可能的位置。所以尽管沃尔多的样子很有特点，在眼花缭乱的场景中找到他也如大海捞针。

由于沃尔多潜藏的地方并不取决于它的样子。我们可以使用一个“沃尔多检测器”扫描图像，该检测器将图像分成数个小方片，并为每个方片包含沃尔多的可能性打分。而卷积神经网络将“空间不变性”的概念系统化，用较少参数来学习有用的特征。

图5.1.1: 沃尔多游戏示例图。

现在，我们将上面想法总结一下，从而帮助我们设计适合于计算机视觉的神经网络结构：

1. 平移不变性：不管出现在图像中的哪个位置，神经网络的底层应该对相同的图像区域做出类似的响应。这个原理即为“平移不变性”。
2. 局部性：神经网络的底层应该只探索输入图像中的局部区域，而不考虑图像远处区域的内容，这就是“局部性”原则。最终，这些局部特征可以融会贯通，在整个图像级别上做出预测。

让我们看看这是如何转化为数学表示的。

5.1.2 限制多层感知机

首先，假设以二维图像 $\mathbf{X}$ 作为输入，那么我们多层感知机的隐藏表示 $\mathbf{H}$ 在数学上是一个矩阵，在代码中表示为二维张量。其中 $\mathbf{X}$ 和 $\mathbf{H}$ 具有相同的形状。我们可以认为，不仅输入有空间结构，隐藏表示也应该有空间结构。我们用 $[\mathbf{X}]_{i,j}$ 和 $[\mathbf{H}]_{i,j}$ 分别表示输入图像和隐藏表示中的位置(i, j)处的像素。

为了使每个输入像素都有神经元处理，我们将参数从权重矩阵（如同我们先前在多层感知机中所做的那样）替换为四阶权重张量 W。假设 $\mathbf{U}$ 包含偏置参数，我们可以将全连接层表示为

$$\begin{aligned}[\mathbf{H}]_{i,j} &= [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l} [\mathbf{X}]_{k,l}\\ &= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}.\end{aligned} \tag{5.1.1}$$

其中，从 W 到 V 的转换只是形式的转换，因为在两个四阶张量中，系数之间存在一对一的对应关系。我们只需重新索引下标 (k,l)，使 $k=i+a$、$l=j+b$，由此 $[\mathsf{V}]_{i,j,a,b}=[\mathsf{W}]_{i,j,i+a,j+b}$。这里的索引 a 和 b 覆盖了正偏移和负偏移。对于隐藏表示$[\mathbf{H}]_{i,j}$中的任何给定位置（i, j），我们通过对 x 中以 (i,j) 为中心的像素，以 $[\mathsf{V}]i,j,a,b$ 作为权重进行加权求和。

平移不变性

现在让我们用上面的第一个原则：平移不变性。这意味着输入 $\mathbf{X}$ 中的移位，应该仅与隐藏表示 $\mathbf{H}$ 中的移位相关。也就是说，V 和 $\mathbf{U}$ 实际上不依赖于 (i,j) 的值，即 $[\mathsf{V}]_{i,j,a,b} = [\mathbf{V}]_{a,b}$。并且 $\mathbf{U}$ 是一个常数，比如 u。因此，我们可以简化 $\mathbf{H}$ 定义为：

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b}[\mathbf{X}]_{i+a,j+b}. \tag{5.1.2}$$

这就是卷积。我们使用系数 $[\mathbf{V}]_{a,b}$ 对位置 (i,j) 附近的像素 $(i+a,j+b)$ 进行加权。注意，$[\mathbf{V}]_{a,b}$ 的参数比 $[\mathsf{V}]_{i,j,a,b}$ 少很多，因为前者不再依赖于图像中的位置。

局部性

现在引用上述的第二个原则：局部性。如上所述，为了收集用来训练参数 $[\mathbf{H}]_{i,j}$ 的相关信息，我们不应偏离到距 (i,j) 很远的地方。这意味着在 $|a| > \Delta$ 或 $|b| > \Delta$ 的范围之外，我们可以设置 $[\mathbf{V}]_{a,b} = 0$。由此，我们可以将参数 $[\mathbf{H}]_{i,j}$ 重写为

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b}[\mathbf{X}]_{i+a,j+b}. \tag{5.1.3}$$

简而言之，(5.1.3) 是一个卷积层，而卷积神经网络是包含卷积层的一类特殊的神经网络。在深度学习研究社区中，$\mathbf{V}$ 被称为 卷积核（convolution kernel）或者 滤波器（filter），是可学习的权重。当图像处理的局部区域很小时，卷积神经网络与多层感知机的训练差异可能是巨大的：以前，多层感知机可能需要数十亿个参数来表示，而现在卷积神经网络通常只需要几百个参数，而且不需要改变输入或隐藏表示的维数。以上所有的权重学习都依赖于归纳偏置，当这种偏置与实际情况相符时，我们就可以得到有效的模型，这些模型能很好地推广到不可见的数据中。但如果这些假设与实际情况不符，比如当图像不满足平移不变时，我们的模型可能难以拟合。

5.1.3 卷积

在进一步讨论之前，我们先简要回顾一下为什么上面的操作被称为卷积。在数学中，两个函数（比如 $f,g : \mathbb{R}^d \to \mathbb{R}$）之间的卷积被定义为

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z})g(\mathbf{x}-\mathbf{z})d\mathbf{z}. \tag{5.1.4}$$

也就是说，卷积是测量 f 和 g 之间（把函数"翻转"并移位 $\mathbf{x}$ 时）的重叠。当我们有离散对象时（即定义域为 $\mathbb{Z}$），积分就变成求和，我们得到以下定义：

$$(f * g)(i) = \sum_a f(a)g(i-a). \tag{5.1.5}$$

对于二维张量，则为 f 在 (a,b) 和 g 在 $(i-a,j-b)$ 上的对应和：

$$(f * g)(i,j) = \sum_a \sum_b f(a,b)g(i-a,j-b). \tag{5.1.6}$$

这看起来类似于 (5.1.3)，但有一个主要区别：这里不是使用 $(i+a,j+b)$，而是使用差值。然而，这种区别是可以化简的，因为我们总是可以匹配 (5.1.3) 和 (5.1.6) 之间的符号。我们在 (5.1.3) 中的原始定义更正确地描述了互相关。我们将在下一节中讨论这一问题。

5.1.4 “沃尔多在哪里”回顾

回到上面的“沃尔多在哪里”游戏，让我们看看它到底是什么样子。卷积层根据滤波器 V 选取给定大小的窗口，并加权处理图片，如 图5.1.2 中所示。我们的目标是学习一个模型，以便探测出在“沃尔多”最可能出现的地方。

图5.1.2: 发现沃尔多。

通道

然而这种方法有一个问题：我们忽略了图像是由三原色（红色、绿色和蓝色）的组成。实际上，图像不是二维张量，而是一个由高度、宽度和颜色组成的三维张量，例如形状为 $1024 \times 1024 \times 3$ 的像素。因此，我们将 X 索引为 $[\mathsf{X}]_{i,j,k}$ 。由此卷积相应地调整为 $[\mathsf{V}]_{a,b,c}$ ，而不是 $[\mathbf{V}]_{a,b}$ 。

此外，由于输入图像是三维的，我们的隐藏表示 H 也是一个三维张量。因此，我们可以把隐藏表示想象为一系列具有二维张量的 通道（channel）。这些通道有时也被称为 特征映射（feature maps），因为每一层都向后续层提供一组空间化的学习特征。在靠近输入的底层，一些通道专门识别边，而其他通道专门识别纹理。

为了支持输入 X 和隐藏表示 H 中的多个通道，我们可以在 V 中添加第四个坐标，即 $[\mathsf{V}]_{a,b,c,d}$ 。综上所述，

$$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_c [\mathsf{V}]_{a,b,c,d} [\mathsf{X}]_{i+a,j+b,c}, \tag{5.1.7}$$

其中隐藏表示 H 中的 d 索引表示输出通道，而随后的输出将继续以三维张量 H 作为输入进入下一个卷积层。所以，(5.1.7) 可以定义具有多个通道的卷积层，而其中 V 是该卷积层的权重。

然而，仍有许多问题亟待解决。例如，图像中是否到处都有存在沃尔多的可能？如何有效地计算输出层？如何选择适当的激活函数？为了训练有效的网络，如何做出合理的网络设计选择？我们将在本章的其它部分讨论这些问题。

5.1.5 小结

- 图像的平移不变性使我们可以以相同的方式处理局部图像。
- 局部性意味着计算相应的隐藏表示只需一小部分局部图像像素。
- 在图像处理中，卷积层通常比全连接层需要更少的参数。
- 卷积神经网络（CNN）是一类特殊的神经网络，它可以包含多个卷积层。
- 多个输入和输出通道使模型在每个空间位置可以获取图像的多方面特征。

5.1.6 练习

1. 假设卷积层 (5.1.3) 覆盖的局部区域 $\Delta = 0$ 。在这种情况下，证明卷积内核为每组通道独立地实现一个全连接层。
2. 为什么平移不变性可能也不是好主意呢？
3. 当从图像边界像素获取隐藏表示时，我们需要思考哪些问题？
4. 描述一个类似的音频卷积层的架构。
5. 卷积层也适合于文本数据吗？为什么？
6. 证明在 (5.1.6) 中，$f * g = g * f$ 。

Discussions[76]

5.2 图像卷积

上节我们解析了卷积层的原理，现在我们看看它的实际应用。由于卷积神经网络的设计是用于探索图像数据，本节我们将以图像为例。

5.2.1 互相关运算

严格地说，卷积层所表达的运算可以被更准确地描述为 互相关运算 (cross-correlation)。根据 5.1节 中的描述，在卷积层中，输入张量和核张量通过互相关运算产生输出张量。

首先，我们暂时忽略通道（第三维）这一情况，看看如何处理二维图像数据和隐藏表示。在 图5.2.1 中，输入是高度为 3 、宽度为 3 的二维张量（即形状为 3×3 ）。卷积核的高度和宽度都是 2 ，而卷积核窗口（或卷积窗口）的形状由内核的高度和宽度决定（即 2×2 ）。

[76] https://discuss.d2l.ai/t/1846

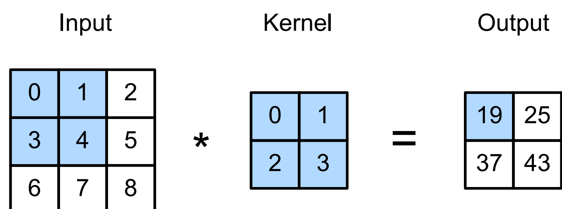


图5.2.1: 二维互相关运算。阴影部分是第一个输出元素，以及用于计算这个输出的输入和核张量元素：$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

在二维互相关运算中，卷积窗口从输入张量的左上角开始，从左到右、从上到下滑动。当卷积窗口滑动到新一个位置时，包含在该窗口中的部分张量与卷积核张量进行按元素相乘，得到的张量再求和得到一个单一的标量值，由此我们得出了这一位置的输出张量值。在如上例子中，输出张量的四个元素由二维互相关运算得到，这个输出高度为 2 、宽度为 2 ，如下所示：

$$\begin{aligned} 0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19, \\ 1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25, \\ 3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37, \\ 4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43. \end{aligned} \tag{5.2.1}$$

注意，输出大小略小于输入大小。这是因为卷积核的宽度和高度大于1，而卷积核只与图像中每个大小完全适合的位置进行互相关运算。所以，输出大小等于输入大小 $n_h \times n_w$ 减去卷积核大小 $k_h \times k_w$，即：

$$(n_h - k_h + 1) \times (n_w - k_w + 1). \tag{5.2.2}$$

这是因为我们需要足够的空间在图像上“移动”卷积核。稍后，我们将看到如何通过在图像边界周围填充零来有保证足够的空间来移动内核，从而保持输出大小不变。接下来，我们在 `corr2d` 函数中实现如上过程，该函数接受输入张量 `X` 和卷积核张量 `K` ，并返回输出张量 `Y` 。

```
from mxnet import autograd, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

```
def corr2d(X, K):  #@save
    """计算二维互相关运算。"""
    h, w = K.shape
    Y = np.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
    return Y
```

通过 图5.2.1 的输入张量 `X` 和卷积核张量 `K` ，我们来验证一下上述二维互相关运算的输出。

```
X = np.array([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = np.array([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

```
array([[19., 25.],
       [37., 43.]])
```

5.2.2 卷积层

卷积层对输入和卷积核权重进行互相关运算，并在添加标量偏置之后产生输出。所以，卷积层中的两个被训练的参数是卷积核权重和标量偏置。就像我们之前随机初始化全连接层一样，在训练基于卷积层的模型时，我们也随机初始化卷积核权重

基于上面定义的 `corr2d` 函数实现二维卷积层。在 `__init__` 构造函数中，将 `weight` 和 `bias` 声明为两个模型参数。前向传播函数调用 `corr2d` 函数并添加偏置。

```
class Conv2D(nn.Block):
    def __init__(self, kernel_size, **kwargs):
        super().__init__(**kwargs)
        self.weight = self.params.get('weight', shape=kernel_size)
        self.bias = self.params.get('bias', shape=(1,))

    def forward(self, x):
        return corr2d(x, self.weight.data()) + self.bias.data()
```

高度和宽度分别为 h 和 w的卷积核可以被称为 $h \times w$ 卷积或 $h \times w$ 卷积核。我们也将带有 $h \times w$ 卷积核的卷积层称为 $h \times w$ 卷积层

5.2.3 图像中目标的边缘检测

如下是卷积层的一个简单应用：通过找到像素变化的位置来检测图像中不同颜色的边缘。首先，我们构造一个 6×8 像素的黑白图像。中间四列为黑色（0），其余像素为白色（1）。

```
X = np.ones((6, 8))
X[:, 2:6] = 0
X
```

```
array([[1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
```

(continues on next page)

(continued from previous page)

```
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.]])
```

接下来，我们构造一个高度为 1 、宽度为 2 的卷积核 K 。当进行互相关运算时，如果水平相邻的两元素相同，则输出为零，否则输出为非零。

```
K = np.array([[1.0, -1.0]])
```

现在，我们对参数 X （输入）和 K （卷积核）执行互相关运算。如下所示，输出 Y 中的 1 代表从白色到黑色的边缘，-1 代表从黑色到白色的边缘，其他情况的输出为 0

```
Y = corr2d(X, K)
Y
```

```
array([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
       [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
       [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
       [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
       [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
       [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

现在我们将输入的二维图像转置，再进行如上的互相关运算。其输出如下，之前检测到的垂直边缘消失了。不出所料，这个卷积核 K 只可以检测垂直边缘，无法检测水平边缘。

```
corr2d(X.T, K)
```

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

5.2.4 学习卷积核

如果我们只需寻找黑白边缘，那么以上 `[1, -1]` 的边缘检测器足以。然而，当有了更复杂数值的卷积核，或者连续的卷积层时，我们不可能手动设计过滤器。那么我们是否可以学习由 `X` 生成 `Y` 的卷积核呢？

现在让我们看看是否可以通过仅查看"输入-输出"对来了解由 `X` 生成 `Y` 的卷积核。我们先构造一个卷积层，并将其卷积核初始化为随机张量。接下来，在每次迭代中，我们比较 `Y` 与卷积层输出的平方误差，然后计算梯度来更新卷积核。为了简单起见，我们在此使用内置的二维卷积层，并忽略偏置。

```
# 构造一个二维卷积层，它具有1个输出通道和形状为（1，2）的卷积核
conv2d = nn.Conv2D(1, kernel_size=(1, 2), use_bias=False)
conv2d.initialize()

# 这个二维卷积层使用四维输入和输出格式（批量大小、通道、高度、宽度），
# 其中批量大小和通道数都为1

X = X.reshape(1, 1, 6, 8)
Y = Y.reshape(1, 1, 6, 7)

for i in range(10):
    with autograd.record():
        Y_hat = conv2d(X)
        l = (Y_hat - Y)**2
    l.backward()
    # 迭代卷积核
    conv2d.weight.data()[:] -= 3e-2 * conv2d.weight.grad()
    if (i + 1) % 2 == 0:
        print(f'batch {i+1}, loss {float(l.sum()):.3f}')
```

```
batch 2, loss 4.949
batch 4, loss 0.831
batch 6, loss 0.140
batch 8, loss 0.024
batch 10, loss 0.004
```

在 10 次迭代之后，误差已经降到足够低。现在我们来看看我们所学的卷积核的权重张量。

```
conv2d.weight.data().reshape((1, 2))
```

```
array([[ 0.9895    , -0.9873705]])
```

细心的你一定会发现，我们学习到的卷积核权重非常接近我们之前定义的卷积核 `K`。

5.2.5 互相关和卷积

回想一下我们在 5.1节 中观察到的互相关和卷积运算之间的对应关系。为了得到严格卷积运算输出，我们需要执行 (5.1.6) 中定义的严格卷积运算，而不是互相关运算。幸运的是，它们差别不大，我们只需水平和垂直翻转二维卷积核张量，然后对输入张量执行互相关运算。

值得注意的是，由于卷积核是从数据中学习到的，因此无论这些层执行严格的卷积运算还是互相关运算，卷积层的输出都不会受到影响。为了说明这一点，假设卷积层执行互相关运算并学习 图5.2.1 中的卷积核，该卷积核在这里由矩阵 $\mathbf{K}$ 表示。假设其他条件不变，当这个层执行严格的卷积时，学习的卷积核 $\mathbf{K}'$ 在水平和垂直翻转之后将与 $\mathbf{K}$ 相同。也就是说，当卷积层对 图5.2.1 中的输入和 $\mathbf{K}'$ 执行严格卷积运算时，将得到与互相关运算 图5.2.1 中相同的输出。

为了与深度学习文献中的标准术语保持一致，我们将继续把"互相关运算"称为卷积运算，尽管严格地说，它们略有不同。此外，对于卷积核张量上的权重，我们称其为元素。

5.2.6 特征映射和感受野

如在 5.1.4节 中所述，图5.2.1 中输出的卷积层有时被称为 特征映射（Feature Map），因为它可以被视为一个输入映射到下一层的空间维度的转换器。在CNN中，对于某一层的任意元素 x ，其 感受野（Receptive Field）是指在前向传播期间可能影响 x 计算的所有元素（来自所有先前层）。

注意，感受野的覆盖率可能大于某层输入的实际区域大小。让我们用 图5.2.1 为例来解释感受野：给定 2×2 卷积核，阴影输出元素值 19 的接收域是输入阴影部分的四个元素。假设之前输出为 $\mathbf{Y}$ ，其大小为 2×2 ，现在我们在其后附加一个卷积层，该卷积层以 $\mathbf{Y}$ 为输入，输出单个元素 z。在这种情况下，$\mathbf{Y}$ 上的 z 的接收字段包括 $\mathbf{Y}$ 的所有四个元素，而输入的感受野包括最初所有九个输入元素。因此，当一个特征图中的任意元素需要检测更广区域的输入特征时，我们可以构建一个更深的网络。

5.2.7 小结

- 二维卷积层的核心计算是二维互相关运算。最简单的形式是，对二维输入数据和卷积核执行互相关操作，然后添加一个偏置。
- 我们可以设计一个卷积核来检测图像的边缘。
- 我们可以从数据中学习卷积核的参数。
- 学习卷积核时，无论用严格卷积运算或互相关运算，卷积层的输出不会受太大影响。
- 当需要检测输入特征中更广区域时，我们可以构建一个更深的卷积网络。

5.2.8 练习

1. 构建一个具有对角线边缘的图像 X。
    1. 如果将本节中举例的卷积核 K 应用于 X，会发生什么情况？
    2. 如果转置 X 会发生什么？
    3. 如果转置 K 会发生什么？
2. 在我们创建的 Conv2D 自动求导时，有什么错误消息？
3. 如何通过改变输入张量和卷积核张量，将互相关运算表示为矩阵乘法？
4. 手工设计一些卷积核：
    1. 二阶导数的核形式是什么？
    2. 积分的核形式是什么？
    3. 得到 d 次导数的最小核大小是多少？

Discussions[77]

5.3 填充和步幅

在前面的例子 图5.2.1 中，输入的高度和宽度都为3，卷积核的高度和宽度都为 2，生成的输出表征的维数为 2×2。正如我们在 5.2节 中所概括的那样，假设输入形状为 $n_h \times n_w$，卷积核形状为 $k_h \times k_w$，那么输出形状将是 $(n_h - k_h + 1) \times (n_w - k_w + 1)$。因此，卷积的输出形状取决于输入形状和卷积核的形状。

还有什么因素会影响输出的大小呢？本节我们将介绍 填充（padding）和 步幅 (stride)。假设以下情景：- 有时，在应用了连续的卷积之后，我们最终得到的输出远小于输入大小。这是由于卷积核的宽度和高度通常大于 1 所导致的。比如，一个 240×240 像素的图像，经过 10 层 5×5 的卷积后，将减少到 200×200 像素。如此一来，原始图像的边界丢失了许多有用信息。而填充是解决此问题最有效的方法。- 有时，我们可能希望大幅降低图像的宽度和高度。例如，如果我们发现原始的输入分辨率十分冗余。步幅则可以在这类情况下提供帮助。

5.3.1 填充

如上所述，在应用多层卷积时，我们常常丢失边缘像素。由于我们通常使用小卷积核，因此对于任何单个卷积，我们可能只会丢失几个像素。但随着我们应用许多连续卷积层，累积丢失的像素数就多了。解决这个问题的简单方法即为填充（padding）：在输入图像的边界填充元素（通常填充元素是 0）。例如，在 图5.3.1 中，我们将 3×3 输入填充到 5×5，那么它的输出就增加为 4×4。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：$0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$。

[77] https://discuss.d2l.ai/t/1849

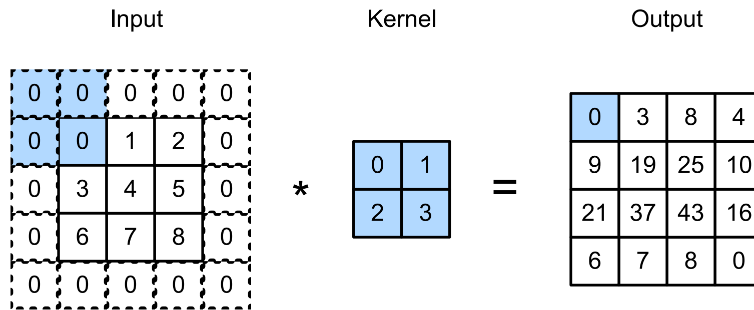


图5.3.1: 带填充的二维互相关。

通常，如果我们添加 p_h 行填充（大约一半在顶部，一半在底部）和 p_w 列填充（左侧大约一半，右侧半），则输出形状将为

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)。 \tag{5.3.1}$$

这意味着输出的高度和宽度将分别增加 p_h 和 p_w。

在许多情况下，我们需要设置 $p_h = k_h - 1$ 和 $p_w = k_w - 1$，使输入和输出具有相同的高度和宽度。这样可以在构建网络时更容易地预测每个图层的输出形状。假设 k_h 是奇数，我们将在高度的两侧填充 $p_h/2$ 行。如果 k_h 是偶数，则一种可能性是在输入顶部填充 $\lceil p_h/2 \rceil$ 行，在底部填充 $\lfloor p_h/2 \rfloor$ 行。同理，我们填充宽度的两侧。

卷积神经网络中卷积核的高度和宽度通常为奇数，例如 1、3、5 或 7。选择奇数的好处是，保持空间维度的同时，我们可以在顶部和底部填充相同数量的行，在左侧和右侧填充相同数量的列。

此外，使用奇数核和填充也提供了书写上的便利。对于任何二维张量 `X`，当满足：1. 内核的大小是奇数；2. 所有边的填充行数和列数相同；3. 输出与输入具有相同高度和宽度则可以得出：输出 `Y[i, j]` 是通过以输入 `X[i, j]` 为中心，与卷积核进行互相关计算。

比如，在下面的例子中，我们创建一个高度和宽度为3的二维卷积层，并在所有侧边填充 1 个像素。给定高度和宽度为 8 的输入，则输出的高度和宽度也是 8。

```
from mxnet import np, npx
from mxnet.gluon import nn

npx.set_np()

# 为了方便起见，我们定义了一个计算卷积层的函数。
# 此函数初始化卷积层权重，并对输入和输出提高和缩减相应的维数
def comp_conv2d(conv2d, X):
    conv2d.initialize()
    # 这里的（1，1）表示批量大小和通道数都是1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # 省略前两个维度：批量大小和通道
    return Y.reshape(Y.shape[2:])
```

(continues on next page)

(continued from previous page)

```
# 请注意，这里每边都填充了1行或1列，因此总共添加了2行或2列
conv2d = nn.Conv2D(1, kernel_size=3, padding=1)
X = np.random.uniform(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

当卷积内核的高度和宽度不同时，我们可以填充不同的高度和宽度，使输出和输入具有相同的高度和宽度。在如下示例中，我们使用高度为 5，宽度为 3 的卷积核，高度和宽度两边的填充分别为 2 和 1。

```
conv2d = nn.Conv2D(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

5.3.2 步幅

在计算互相关时，卷积窗口从输入张量的左上角开始，向下和向右滑动。在前面的例子中，我们默认每次滑动一个元素。但是，有时候为了高效计算或是缩减采样次数，卷积窗口可以跳过中间位置，每次滑动多个元素。

我们将每次滑动元素的数量称为 步幅（stride）。到目前为止，我们只使用过高度或宽度为 1 的步幅，那么如何使用较大的步幅呢？图5.3.2 是垂直步幅为 3，水平步幅为 2 的二维互相关运算。着色部分是输出元素以及用于输出计算的输入和内核张量元素：$0\times0+0\times1+1\times2+2\times3=8$、$0\times0+6\times1+0\times2+0\times3=6$。

如何计算输出中第一列的第二个元素呢？如图所示，卷积窗口向下滑动三行、向右滑动两列。但是，当卷积窗口继续向右滑动两列时，没有输出，因为输入元素无法填充窗口（除非我们添加另一列填充）。

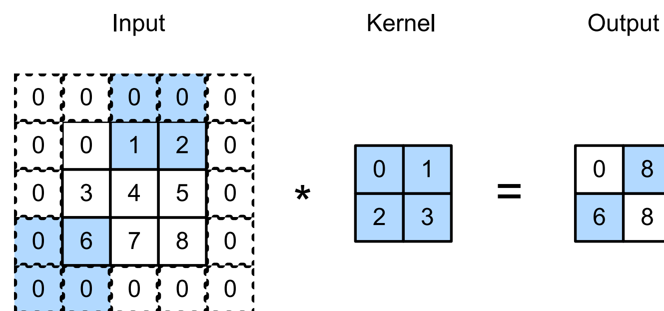


图5.3.2: 垂直步幅为 3，水平步幅为 2 的二维互相关运算。

通常，当垂直步幅为 s_h 、水平步幅为 s_w 时，输出形状为

$$\lfloor(n_h-k_h+p_h+s_h)/s_h\rfloor \times \lfloor(n_w-k_w+p_w+s_w)/s_w\rfloor. \tag{5.3.2}$$

如果我们设置了 $p_h = k_h - 1$ 和 $p_w = k_w - 1$，则输出形状将简化为 $\lfloor(n_h + s_h - 1)/s_h\rfloor \times \lfloor(n_w + s_w - 1)/s_w\rfloor$。更进一步，如果输入的高度和宽度可以被垂直和水平步幅整除，则输出形状将为 $(n_h/s_h) \times (n_w/s_w)$。

下面，我们将高度和宽度的步幅设置为 2，从而将输入的高度和宽度减半。

```
conv2d = nn.Conv2D(1, kernel_size=3, padding=1, strides=2)
comp_conv2d(conv2d, X).shape
```

```
(4, 4)
```

接下来，看一个稍微复杂的例子。

```
conv2d = nn.Conv2D(1, kernel_size=(3, 5), padding=(0, 1), strides=(3, 4))
comp_conv2d(conv2d, X).shape
```

```
(2, 2)
```

为了简洁起见,当输入高度和宽度两侧的填充数量分别为 p_h 和 p_w 时,我们称之为填充 (p_h, p_w)。当 $p_h = p_w = p$ 时，填充是 p。同理，当高度和宽度上的步幅分别为 s_h 和 s_w 时，我们称之为步幅 (s_h, s_w)。当时的步幅为 $s_h = s_w = s$ 时，步幅为 s。默认情况下，填充为 0，步幅为 1。在实践中，我们很少使用不一致的步幅或填充，也就是说，我们通常有 $p_h = p_w$ 和 $s_h = s_w$。

5.3.3 小结

- 填充可以增加输出的高度和宽度。这常用来使输出与输入具有相同的高和宽。
- 步幅可以减小输出的高和宽，例如输出的高和宽仅为输入的高和宽的 $1/n$（n 是一个大于 1 的整数)。
- 填充和步幅可用于有效地调整数据的维度。

5.3.4 练习

1. 对于本节中的最后一个示例，计算其输出形状，以查看它是否与实验结果一致。
2. 在本节中的实验中，试一试其他填充和步幅组合。
3. 对于音频信号，步幅 2 说明什么？
4. 步幅大于 1 的计算优势是什么？

Discussions[78]

[78] https://discuss.d2l.ai/t/1852

5.4 多输入多输出通道

虽然我们在 5.1.4节 中描述了构成每个图像的多个通道和多层卷积层。例如彩色图像具有标准的 RGB 通道来指示红、绿和蓝。但是到目前为止，我们仅展示了单个输入和单个输出通道的简化例子。这使得我们可以将输入、卷积核和输出看作二维张量。

当我们添加通道时，我们的输入和隐藏的表示都变成了三维张量。例如，每个RGB输入图像具有 $3 \times h \times w$ 的形状。我们将这个大小为 3 的轴称为 通道（channel）维度。在本节中，我们将更深入地研究具有多输入和多输出通道的卷积核。

5.4.1 多输入通道

当输入包含多个通道时，需要构造一个与输入数据具有相同输入通道数目的卷积核，以便与输入数据进行互相关运算。假设输入的通道数为 c_i，那么卷积核的输入通道数也需要为 c_i 。如果卷积核的窗口形状是 $k_h \times k_w$，那么当 $c_i = 1$ 时，我们可以把卷积核看作形状为 $k_h \times k_w$ 的二维张量。

然而，当 $c_i > 1$ 时，我们卷积核的每个输入通道将包含形状为 $k_h \times k_w$ 的张量。将这些张量 c_i 连结在一起可以得到形状为 $c_i \times k_h \times k_w$ 的卷积核。由于输入和卷积核都有 c_i 个通道，我们可以对每个通道输入的二维张量和卷积核的二维张量进行互相关运算，再对通道求和（将 c_i 的结果相加）得到二维张量。这是多通道输入和多输入通道卷积核之间进行二维互相关运算的结果。

在 图5.4.1 中，我们演示了一个具有两个输入通道的二维互相关运算的示例。阴影部分是第一个输出元素以及用于计算这个输出的输入和核张量元素：$(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$。

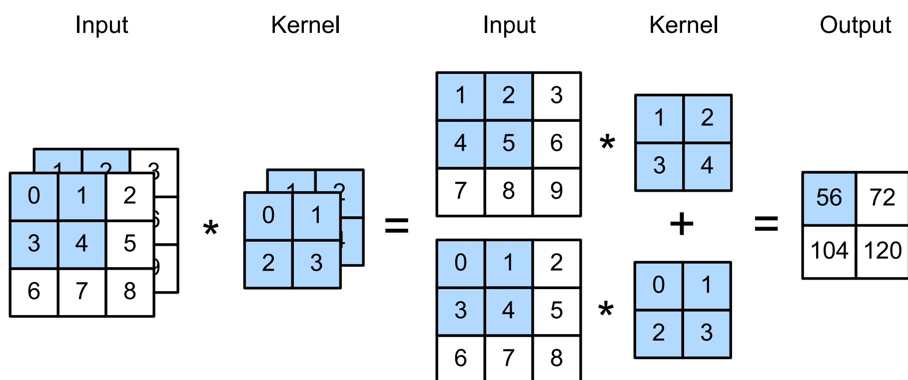


图5.4.1: 两个输入通道的互相关计算。

为了加深理解，我们将多输入通道互相关运算实现一下。简而言之，我们所做的就是对每个通道执行互相关操作，然后将结果相加。

```python
from mxnet import np, npx
from d2l import mxnet as d2l

npx.set_np()
```

```python
def corr2d_multi_in(X, K):
    # 先遍历 “X” 和 “K” 的第0个维度（通道维度），再把它们加在一起
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

我们可以构造与 图5.4.1 中的值相对应的输入张量 X 和核张量 K，以验证互相关运算的输出。

```python
X = np.array([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
              [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = np.array([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K)
```

```
array([[ 56.,  72.],
       [104., 120.]])
```

5.4.2 多输出通道

到目前为止，不论有多少输入通道，我们还只有一个输出通道。然而，正如我们在 5.1.4节 中所讨论的，每一层有多个输出通道是至关重要的。在最流行的神经网络架构中，随着神经网络层数的加深，我们常会增加输出通道的维数，通过减少空间分辨率以获得更大的通道深度。直观地说，我们可以将每个通道看作是对不同特征的响应。而现实可能更为复杂一些，因为每个通道不是独立学习的，而是为了共同使用而优化的。因此，多输出通道并不仅是学习多个单通道的检测器。

用 c_i 和 c_o 分别表示输入和输出通道的数目，并让 k_h 和 k_w 为卷积核的高度和宽度。为了获得多个通道的输出，我们可以为每个输出通道创建一个形状为 $c_i \times k_h \times k_w$ 的卷积核张量，这样卷积核的形状是 $c_o \times c_i \times k_h \times k_w$。在互相关运算中，每个输出通道先获取所有输入通道，再以对应该输出通道的卷积核计算出结果。

如下所示，我们实现一个计算多个通道的输出的互相关函数。

```python
def corr2d_multi_in_out(X, K):
    # 迭代“K”的第0个维度，每次都对输入“X”执行互相关运算。
    # 最后将所有结果都叠加在一起
    return np.stack([corr2d_multi_in(X, k) for k in K], 0)
```

通过将核张量 K 与 K+1 （K 中每个元素加 1 ）和 K+2 连接起来，构造了一个具有3个输出通道的卷积核。

```python
K = np.stack((K, K + 1, K + 2), 0)
K.shape
```

```
(3, 2, 2, 2)
```

下面，我们对输入张量 X 与卷积核张量 K 执行互相关运算。现在的输出包含 3 个通道，第一个通道的结果与先前输入张量 X 和多输入单输出通道的结果一致。

```
corr2d_multi_in_out(X, K)
```

```
array([[[ 56.,  72.],
        [104., 120.]],

       [[ 76., 100.],
        [148., 172.]],

       [[ 96., 128.],
        [192., 224.]]])
```

5.4.3 1×1 卷积层

1×1 卷积，即 $k_h = k_w = 1$，看起来似乎没有多大意义。毕竟，卷积的本质是有效提取相邻像素间的相关特征，而 1×1 卷积显然没有此作用。尽管如此，1×1 仍然十分流行，时常包含在复杂深层网络的设计中。下面，让我们详细地解读一下它的实际作用。

因为使用了最小窗口，1×1 卷积失去了卷积层的特有能力——在高度和宽度维度上，识别相邻元素间相互作用的能力。而 1×1 卷积的唯一计算发生在通道上。

图5.4.2 展示了使用 1×1 卷积核与 3 个输入通道和 2 个输出通道的互相关计算。这里输入和输出具有相同的高度和宽度，输出中的每个元素都是从输入图像中同一位置的元素的线性组合。我们可以将 1×1 卷积层看作是在每个像素位置应用的全连接层，以 c_i 个输入值转换为 c_o 个输出值。因为这仍然是一个卷积层，所以跨像素的权重是一致的。同时，1×1 卷积层需要的权重维度为 $c_o \times c_i$，再额外加上一个偏置。

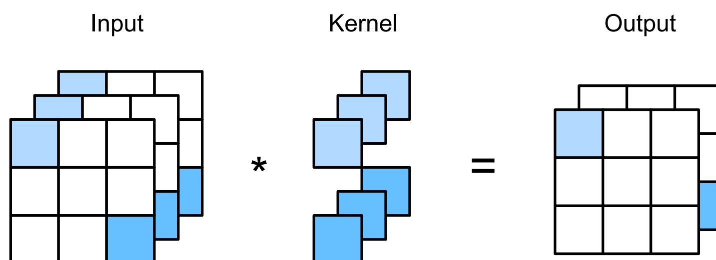


图5.4.2: 互相关计算使用了具有3个输入通道和2个输出通道的 1×1 卷积核。其中，输入和输出具有相同的高度和宽度。

下面，我们使用全连接层实现 1×1 卷积。请注意，我们需要对输入和输出的数据形状进行微调。

```python
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = X.reshape((c_i, h * w))
    K = K.reshape((c_o, c_i))
```

(continues on next page)

(continued from previous page)

```
    Y = np.dot(K, X)  # 全连接层中的矩阵乘法
    return Y.reshape((c_o, h, w))
```

当执行 1×1 卷积运算时，上述函数相当于先前实现的互相关函数corr2d_multi_in_out。让我们用一些样本数据来验证这一点。

```
X = np.random.normal(0, 1, (3, 3, 3))
K = np.random.normal(0, 1, (2, 3, 1, 1))
```

```
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(np.abs(Y1 - Y2).sum()) < 1e-6
```

5.4.4 小结

- 多输入多输出通道可以用来扩展卷积层的模型。
- 当以每像素为基础应用时，1×1 卷积层相当于全连接层。
- 1×1 卷积层通常用于调整网络层的通道数量和控制模型复杂性。

5.4.5 练习

1. 假设我们有两个卷积核，大小分别为 k_1 和 k_2（中间没有非线性激活函数）。
    1. 证明运算可以用单次卷积来表示。
    2. 这个等效的单卷积的维数是多少呢？
    3. 反之亦然吗？
2. 假设输入为 $c_i \times h \times w$，卷积核大小为 $c_o \times c_i \times k_h \times k_w$，填充为 (p_h, p_w)，步幅为 (s_h, s_w)。
    1. 正向传播的计算成本（乘法和加法）是多少？
    2. 内存占用是多少？
    3. 反向传播的内存占用是多少？
    4. 反向传播的计算成本是多少？
3. 如果我们将输入通道 c_i 和输出通道 c_o 的数量加倍，计算数量会增加多少？如果我们把填充数量翻一番会怎么样？
4. 如果卷积核的高度和宽度是 $k_h = k_w = 1$，前向传播的计算复杂度是多少？
5. 本节最后一个示例中的变量 Y1 和 Y2 是否完全相同？为什么？
6. 当卷积窗口不是 1×1 时，如何使用矩阵乘法实现卷积？

Discussions[79]

5.5 池化层

通常当我们处理图像时，我们希望逐渐降低隐藏表示的空间分辨率，聚集信息，这样的随着我们在神经网络中层叠的上升，每个神经元对其敏感的感受野（输入）就越大。

而我们的机器学习任务通常会跟全局图像的问题有关（例如，“图像是否包含一只猫呢?”），所以我们最后一层的神经元应该对整个输入的全局敏感。通过逐渐聚合信息，生成越来越粗糙的映射，最终实现学习全局表示的目标，同时将卷积图层的所有优势保留在中间层。

此外，当检测较底层的特征时（例如 5.2节 中所讨论的边缘），我们通常希望这些特征保持某种程度上的平移不变性。例如，如果我们拍摄黑白之间轮廓清晰的图像 `X`，并将整个图像向右移动一个像素，即 `Z[i, j] = X[i, j + 1]`，则新图像 `Z` 的输出可能大不相同。而在现实中，随着拍摄角度的移动，任何物体几乎不可能发生在同一像素上。即使用三脚架拍摄一个静止的物体，由于快门的移动而引起的相机振动，可能会使所有物体左右移动一个像素（除了高端相机配备了特殊功能来解决这个问题）。

本节将介绍 池化（pooling）层，它具有双重目的：降低卷积层对位置的敏感性，同时降低对空间降采样表示的敏感性。

5.5.1 最大池化层和平均池化层

与卷积层类似，池化层运算符由一个固定形状的窗口组成，该窗口根据其步幅大小在输入的所有区域上滑动，为固定形状窗口（有时称为 池化窗口）遍历的每个位置计算一个输出。然而，不同于卷积层中的输入与卷积核之间的互相关计算，池化层不包含参数。相反，池运算符是确定性的，我们通常计算池化窗口中所有元素的最大值或平均值。这些操作分别称为 最大池化层（maximum pooling）和 平均池化层（average pooling）。

在这两种情况下，与互相关运算符一样，池化窗口从输入张量的左上角开始，从左到右、从上到下的在输入张量内滑动。在池化窗口到达的每个位置，它计算该窗口中输入子张量的最大值或平均值，具体取决于是使用了最大池化层还是平均池化层。

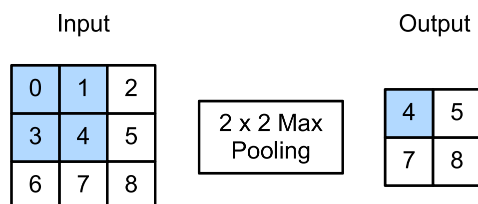


图5.5.1: 池化窗口形状为 2×2 的最大池化层。着色部分是第一个输出元素，以及用于计算这个输出的输入元素: $\max(0, 1, 3, 4) = 4$.

[79] https://discuss.d2l.ai/t/1855

图5.5.1 中的输出张量的高度为 2，宽度为 2。这四个元素为每个池化窗口中的最大值：

$$
\begin{aligned}
\max(0,1,3,4) &= 4,\\
\max(1,2,4,5) &= 5,\\
\max(3,4,6,7) &= 7,\\
\max(4,5,7,8) &= 8.
\end{aligned}
\tag{5.5.1}
$$

池化窗口形状为 $p \times q$ 的池化层称为 $p \times q$ 池化层，池化操作称为 $p \times q$ 池化。

回到本节开头提到的对象边缘检测示例，现在我们将使用卷积层的输出作为 2×2 最大池化的输入。设置卷积层输入为 `X`，池化层输出为 `Y`。无论 `X[i, j]` 和 `X[i, j + 1]` 的值是否不同，或 `X[i, j + 1]` 和 `X[i, j + 2]` 的值是否不同，池化层始终输出 `Y[i, j] = 1`。也就是说，使用 2×2 最大池化层，即使在高度或宽度上移动一个元素，卷积层仍然可以识别到模式。

在下面的代码中的 `pool2d` 函数，实现了池化层的正向传播。此功能类似于 5.2节 中的 `corr2d` 函数。然而，这里我们没有卷积核，输出为输入中每个区域的最大值或平均值。

```
from mxnet import np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = np.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i:i + p_h, j:j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i:i + p_h, j:j + p_w].mean()
    return Y
```

我们可以构建 图5.5.1 中的输入张量 `X`，验证二维最大池化层的输出。

```
X = np.array([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
array([[4., 5.],
       [7., 8.]])
```

此外，我们还可以验证平均池化层。

```
pool2d(X, (2, 2), 'avg')
```

```
array([[2., 3.],
       [5., 6.]])
```

5.5.2 填充和步幅

与卷积层一样，池化层也可以改变输出形状。和以前一样，我们可以通过填充和步幅以获得所需的输出形状。下面，我们用深度学习框架中内置的二维最大池化层，来演示池化层中填充和步幅的使用。我们首先构造了一个输入张量 X，它有四个维度，其中样本数和通道数都是 1。

```
X = np.arange(16, dtype=np.float32).reshape((1, 1, 4, 4))
X
```

```
array([[[[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.]]]])
```

默认情况下，深度学习框架中的步幅与池化窗口的大小相同。因此，如果我们使用形状为 (3, 3) 的池化窗口，那么默认情况下，我们得到的步幅形状为 (3, 3)。

```
pool2d = nn.MaxPool2D(3)
# 由于池化层中没有参数，所以不需要调用初始化函数
pool2d(X)
```

```
array([[[[10.]]]])
```

填充和步幅可以手动设定。

```
pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)
```

```
array([[[[ 5.,  7.],
         [13., 15.]]]])
```

当然，我们可以设定一个任意大小的矩形池化窗口，并分别设定填充和步幅的高度和宽度。

```
pool2d = nn.MaxPool2D((2, 3), padding=(1, 2), strides=(2, 3))
pool2d(X)
```

```
array([[[[ 0.,  3.],
         [ 8., 11.],
         [12., 15.]]]])
```

5.5.3 多个通道

在处理多通道输入数据时，池化层在每个输入通道上单独运算，而不是像卷积层一样在通道上对输入进行汇总。这意味着池化层的输出通道数与输入通道数相同。下面，我们将在通道维度上连结张量 X 和 X + 1，以构建具有 2 个通道的输入。

```
X = np.concatenate((X, X + 1), 1)
X
```

```
array([[[[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.]],

        [[ 1.,  2.,  3.,  4.],
         [ 5.,  6.,  7.,  8.],
         [ 9., 10., 11., 12.],
         [13., 14., 15., 16.]]]])
```

如下所示，池化后输出通道的数量仍然是 2。

```
pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)
```

```
array([[[[ 5.,  7.],
         [13., 15.]],

        [[ 6.,  8.],
         [14., 16.]]]])
```

5.5.4 小结

- 对于给定输入元素，最大池化层会输出该窗口内的最大值，平均池化层会输出该窗口内的平均值。
- 池化层的主要优点之一是减轻卷积层对位置的过度敏感。
- 我们可以指定池化层的填充和步幅。
- 使用最大池化层以及大于 1 的步幅，可减少空间维度（如高度和宽度）。

• 池化层的输出通道数与输入通道数相同。

5.5.5 练习

1. 你能将平均池化层作为卷积层的特殊情况实现吗？
2. 你能将最大池化层作为卷积层的特殊情况实现吗？
3. 假设池化层的输入大小为 $c \times h \times w$，则池化窗口的形状为 $p_h \times p_w$，填充为 (p_h, p_w)，步幅为 (s_h, s_w)。这个池化层的计算成本是多少？
4. 为什么最大池化层和平均池化层的工作方式不同？
5. 我们是否需要最小池化层？可以用已知函数替换它吗？
6. 除了平均池化层和最大池化层，是否有其它函数可以考虑（提示：回忆 `softmax`）？为什么它可能不受欢迎？

Discussions[80]

5.6 卷积神经网络（LeNet）

通过之前几节，我们学习了构建一个完整卷积神经网络的所需组件。回想一下，之前我们将 softmax 回归模型（2.6节）和多层感知机模型（3.2节）应用于 Fashion-MNIST 数据集中的服装图片上。为了能够应用 softmax 回归和多层感知机，我们首先将每个大小为 28×28 的图像展平为一个 784 固定长度的一维向量，然后用全连接层对其进行处理。而现在，我们已经掌握了卷积层的处理方法，我们可以在图像中保留空间结构。同时，用卷积层代替全连接层的另一个好处是：更简洁的模型所需的参数更少。

在本节中，我们将介绍 LeNet，它是最早发布的卷积神经网络之一，因其在计算机视觉任务中的高效性能而受到广泛关注。这个模型是由 AT&T 贝尔实验室的研究员 Yann LeCun 在1989年提出的（并以其命名），目的是识别图像 [LeCun et al., 1998] 中的手写数字。当时，Yann LeCun 发表了第一篇通过反向传播成功训练卷积神经网络的研究，这项工作代表了十多年来神经网络研究开发的成果。

当时，LeNet 取得了与支持向量机（support vector machines）性能相媲美的成果，成为监督学习的主流方法。LeNet 被广泛用于自动取款机（ATM）机中，帮助识别处理支票的数字。时至今日，一些自动取款机仍在运行 Yann LeCun 和他的同事 Leon Bottou 在上世纪90年代写的代码呢！

[80] https://discuss.d2l.ai/t/1858

5.6.1 LeNet

总体来看，LeNet (LeNet-5) 由两个部分组成：

- 卷积编码器：由两个卷积层组成;
- 全连接层密集块：由三个全连接层组成。

该结构在 图5.6.1 中所展示。

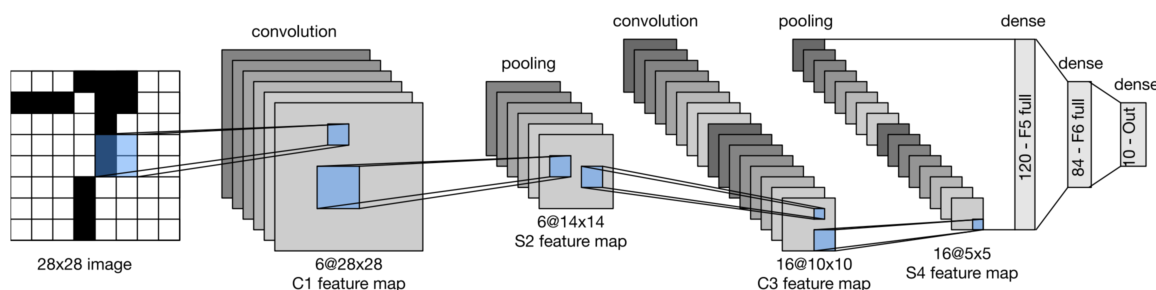


图5.6.1: LeNet中的数据流。输入是手写数字，输出为10种可能结果的概率。

每个卷积块中的基本单元是一个卷积层、一个 sigmoid 激活函数和平均池化层。请注意，虽然 ReLU 和最大池化层更有效，但它们在20世纪90年代还没有出现。每个卷积层使用 5×5 卷积核，这些层将输入映射到多个二维特征输出，通常同时增加通道的数量。第一卷积层有 6 个输出通道，而第二个卷积层有 16 个输出通道。每个 2×2 池操作（步骤2）通过空间下采样将维数减少 4 倍。卷积的输出形状由批量大小、通道数、高度、宽度决定。

为了将卷积块的输出传递给稠密块，我们必须在小批量中展平每个样本。换言之，我们将这个四维输入转换成全连接层所期望的二维输入。这里的二维表示的第一个维度索引小批量中的样本，第二个维度给出每个样本的平面向量表示。LeNet 的稠密块有三个全连接层，分别有 120、84 和 10 个输出。因为我们仍在执行分类，所以输出层的 10 维对应于最后输出结果的数量。

通过下面的 LeNet 代码，你会相信用深度学习框架实现此类模型非常简单。我们只需要实例化一个 Sequential 块并将需要的层连接在一起。

```python
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()

net = nn.Sequential()
```

(continues on next page)

(continued from previous page)

```
net.add(
    nn.Conv2D(channels=6, kernel_size=5, padding=2, activation='sigmoid'),
    nn.AvgPool2D(pool_size=2, strides=2),
    nn.Conv2D(channels=16, kernel_size=5, activation='sigmoid'),
    nn.AvgPool2D(pool_size=2, strides=2),
    # 默认情况下，“Dense” 会自动将形状为（批量大小，通道数，高度，宽度）的输入，
    # 转换为形状为（批量大小，通道数*高度*宽度）的输入
    nn.Dense(120, activation='sigmoid'), nn.Dense(84, activation='sigmoid'),
    nn.Dense(10))
```

我们对原始模型做了一点小改动，去掉了最后一层的高斯激活。除此之外，这个网络与最初的 LeNet-5 一致。

下面，我们将一个大小为 28×28 的单通道（黑白）图像通过 LeNet。通过在每一层打印输出的形状，我们可以检查模型，以确保其操作与我们期望的 图5.6.2 一致。

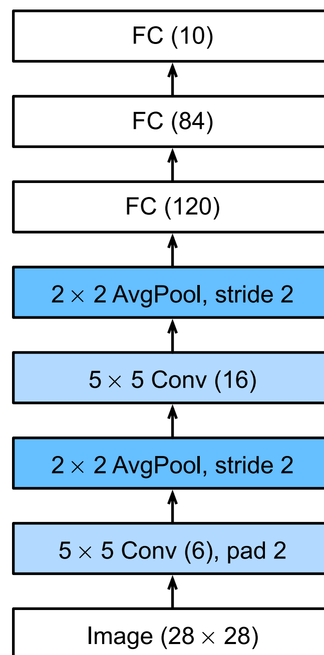


图5.6.2: LeNet 的简化版。

```
X = np.random.uniform(size=(1, 1, 28, 28))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

```
conv0 output shape:	 (1, 6, 28, 28)
pool0 output shape:	 (1, 6, 14, 14)
```

(continues on next page)

(continued from previous page)

```
conv1 output shape:  (1, 16, 10, 10)
pool1 output shape:  (1, 16, 5, 5)
dense0 output shape:         (1, 120)
dense1 output shape:         (1, 84)
dense2 output shape:         (1, 10)
```

请注意，在整个卷积块中，与上一层相比，每一层特征的高度和宽度都减小了。第一个卷积层使用 2 个像素的填充，来补偿 5×5 卷积核导致的特征减少。相反，第二个卷积层没有填充，因此高度和宽度都减少了 4 个像素。随着层叠的上升，通道的数量从输入时的 1 个，增加到第一个卷积层之后的 6 个，再到第二个卷积层之后的 16 个。同时，每个池化层的高度和宽度都减半。最后，每个全连接层减少维数，最终输出一个维数与结果分类数相匹配的输出。

5.6.2 模型训练

现在我们已经实现了 LeNet，让我们看看这个模型在 Fashion-MNIST 数据集上的表现。

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

虽然卷积神经网络的参数较少，但与深度的多层感知机相比，它们的计算成本仍然很高，因为每个参数都参与更多的乘法。如果你有机会使用GPU，可以用它加快训练。

为了进行评估，我们需要对 2.6节 中描述的 `evaluate_accuracy` 函数进行轻微的修改。由于完整的数据集位于内存中，因此在模型使用 GPU 计算数据集之前，我们需要将其复制到显存中。

```
def evaluate_accuracy_gpu(net, data_iter, device=None):  #@save
    """Compute the accuracy for a model on a dataset using a GPU."""
    if not device:  # 查询第一个参数所在的第一个设备
        device = list(net.collect_params().values())[0].list_ctx()[0]
    metric = d2l.Accumulator(2)  # 正确预测的数量，总预测的数量
    for X, y in data_iter:
        X, y = X.as_in_ctx(device), y.as_in_ctx(device)
        metric.add(d2l.accuracy(net(X), y), y.size)
    return metric[0] / metric[1]
```

为了使用 GPU，我们还需要一点小改动。与 2.6节 中定义的 `train_epoch_ch3` 不同，在进行正向和反向传播之前，我们需要将每一小批量数据移动到我们指定的设备（例如 GPU）上。

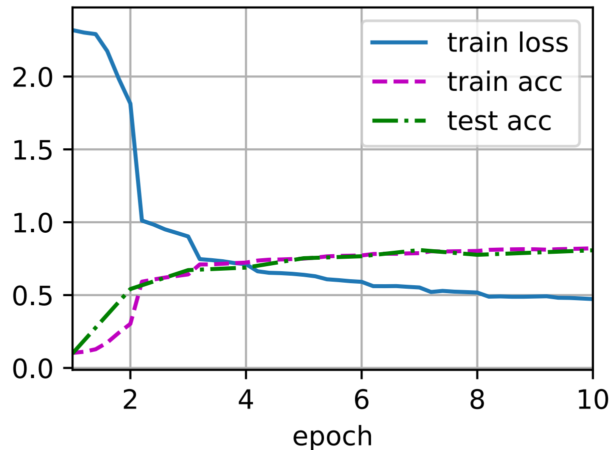
如下所示，训练函数 `train_ch6` 也类似于 2.6节 中定义的 `train_ch3` 。由于我们将实现多层神经网络，因此我们将主要使用高级 API。以下训练函数假定从高级 API 创建的模型作为输入，并进行相应的优化。我们使用在 3.8.2节 中介绍的 Xavier 随机初始化模型参数。与全连接层一样，我们使用交叉熵损失函数和小批量随机梯度下降。

```python
#@save
def train_ch6(net, train_iter, test_iter, num_epochs, lr, device):
    """Train a model with a GPU (defined in Chapter 6)."""
    net.initialize(force_reinit=True, ctx=device, init=init.Xavier())
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer, num_batches = d2l.Timer(), len(train_iter)
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)  # 训练损失之和，训练准确率之和，范例数
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            # 下面是与“d2l.train_epoch_ch3”的主要不同
            X, y = X.as_in_ctx(device), y.as_in_ctx(device)
            with autograd.record():
                y_hat = net(X)
                l = loss(y_hat, y)
            l.backward()
            trainer.step(X.shape[0])
            metric.add(l.sum(), d2l.accuracy(y_hat, y), X.shape[0])
            timer.stop()
            train_l = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (train_l, train_acc, None))
        test_acc = evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(device)}')
```

现在，我们训练和评估 LeNet-5 模型。

```python
lr, num_epochs = 0.9, 10
train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.473, train acc 0.821, test acc 0.807
39406.8 examples/sec on gpu(0)
```



5.6.3 小结

- 卷积神经网络（CNN）是一类使用卷积层的网络。
- 在卷积神经网络中，我们组合使用卷积层、非线性激活函数和池化层。
- 为了构造高性能的卷积神经网络，我们通常对卷积层进行排列，逐渐降低其表示的空间分辨率，同时增加通道数。
- 在传统的卷积神经网络中，卷积块编码得到的表征在输出之前需由一个或多个全连接层进行处理。
- LeNet是最早发布的卷积神经网络之一。

5.6.4 练习

1. 将平均池化层替换为最大池化层，会发生什么？
2. 尝试构建一个基于 LeNet 的更复杂的网络，以提高其准确性。
    1. 调整卷积窗口大小。
    2. 调整输出通道的数量。
    3. 调整激活函数（如 ReLU）。
    4. 调整卷积层的数量。
    5. 调整全连接层的数量。
    6. 调整学习率和其他训练细节（例如，初始化和周期数）。
3. 在 MNIST 数据集上尝试以上改进的网络。
4. 显示不同输入（例如毛衣和外套）时，LeNet 第一层和第二层的激活值。

Discussions[81]

[81] https://discuss.d2l.ai/t/1861

6

现代卷积神经网络

上一章我们介绍了卷积神经网络的基本原理，本章我们将带你了解现代的卷积神经网络结构，许多现代卷积神经网络的研究都是建立在这一章的基础上的。在本章中的每一个模型都曾一度占据主导地位，其中许多模型都是ImageNet竞赛的优胜者。ImageNet竞赛自2010年以来，一直是计算机视觉中监督学习进展的指向标。

这些模型包括：

- AlexNet。它是第一个在大规模视觉竞赛中击败传统计算机视觉模型的大型神经网络；
- 使用重复块的网络（VGG）。它利用许多重复的神经网络块；
- 网络中的网络（NiN）。它重复使用由卷积层和1×1卷积层（用来代替全连接层）来构建深层网络；
- 含并行连结的网络（GoogLeNet）。它使用并行连结的网络，通过不同窗口大小的卷积层和最大池化层来并行抽取信息；
- 残差网络（ResNet）。它通过残差块构建跨层的数据通道，是计算机视觉中最流行的体系结构；
- 稠密连接网络（DenseNet）。它的计算成本很高，但给我们带来了更好的效果。

虽然深度神经网络的概念非常简单——将神经网络堆叠在一起。但由于不同的网络结构和超参数选择，这些神经网络的性能会发生很大变化。本章介绍的神经网络是将人类直觉和相关数学见解结合后，经过大量研究试错后的结晶。我们会按时间顺序介绍这些模型，在追寻历史的脉络的同时，帮助你培养对该领域发展的直觉。这将有助于你研究开发自己的结构。例如，本章介绍的批量归一化（batch normalization）和残差网络（ResNet）为设计和训练深度神经网络提供了重要思想指导。

6.1 深度卷积神经网络（AlexNet）

在LeNet提出后，卷积神经网络在计算机视觉和机器学习领域中很有名气。但卷积神经网络并没有主导这些领域。这是因为虽然 LeNet 在小数据集上取得了很好的效果，但是在更大、更真实的数据集上训练卷积神经网络的性能和可行性还有待研究。事实上，在上世纪90年代初到2012年之间的大部分时间里，神经网络往往被其他机器学习方法超越，如支持向量机（support vector machines）。

在计算机视觉中，直接将神经网络与其他机器学习方法进行比较也许不公平。这是因为，卷积神经网络的输入是由原始像素值或是经过简单预处理（例如居中、缩放）的像素值组成的。但在使用传统机器学习方法时，从业者永远不会将原始像素作为输入。在传统机器学习方法中，计算机视觉流水线是由经过人的手工精心设计的特征流水线组成的。对于这些传统方法，大部分的进展都来自于对特征有了更聪明的想法，并且学习到的算法往往归于事后的解释。

虽然上世纪90年代就有了一些神经网络加速器，但仅靠它们还不足以开发出有大量参数的深层多通道多层卷积神经网络。此外，当时的数据集仍然相对较小。除了这些障碍，训练神经网络的一些关键技巧仍然缺失，包括启发式参数初始化、随机梯度下降的巧妙变体、非挤压激活函数和有效的正则化技术。

因此，与训练端到端（从像素到分类结果）系统不同，经典机器学习的流水线看起来更像下面这样：

1. 获取一个有趣的数据集。在早期，收集这些数据集需要昂贵的传感器（在当时最先进的图像也就100万像素）。
2. 根据光学、几何学、其他知识以及偶然的发现，手工对特征数据集进行预处理。
3. 通过标准的特征提取算法（如SIFT（尺度不变特征变换）[Lowe, 2004] 、SURF（加速鲁棒特征）[Bay et al., 2006] 或其他手动调整的流水线来输入数据。
4. 将提取的特征放到最喜欢的分类器中（例如线性模型或其它核方法），以训练分类器。

如果你和机器学习研究人员交谈，你会发现他们相信机器学习既重要又美丽：优雅的理论去证明各种模型的性质。机器学习是一个正在蓬勃发展、严谨且非常有用的领域。然而，如果你和计算机视觉研究人员交谈，你会听到一个完全不同的故事。他们会告诉你图像识别的诡异事实————推动领域进步的是数据特征，而不是学习算法。计算机视觉研究人员相信，从对最终模型精度的影响来说，更大或更干净的数据集、或是稍微改进的特征提取，比任何学习算法带来的进步要大得多。

6.1.1 学习表征

另一种预测这个领域发展的方法————观察图像特征的提取方法。在2012年前，图像特征都是机械地计算出来的。事实上，设计一套新的特征函数、改进结果，并撰写论文是盛极一时的潮流。SIFT [Lowe, 2004]、SURF [Bay et al., 2006]、HOG（定向梯度直方图）[Dalal & Triggs, 2005] 、bags of visual words[82] 和类似的特征提取方法占据了主导地位。

另一组研究人员，包括Yann LeCun、Geoff Hinton、Yoshua Bengio、Andrew Ng、Shun ichi Amari和Juergen Schmidhuber，想法则与众不同：他们认为特征本身应该被学习。此外，他们还认为，在合理地复杂性前提下，特征应该由多个共同学习的神经网络层组成，每个层都有可学习的参数。在机器视觉中，最底层可能检

[82] https://en.wikipedia.org/wiki/Bag-of-words_model_in_computer_vision

测边缘、颜色和纹理。事实上，Alex Krizhevsky、Ilya Sutskever和Geoff Hinton提出了一种新的卷积神经网络变体*AlexNet*。在2012年ImageNet挑战赛中取得了轰动一时的成绩。AlexNet 以 Alex Krizhevsky 的名字命名，他是论文 [Krizhevsky et al., 2012] 的第一作者。

有趣的是，在网络的最底层，模型学习到了一些类似于传统滤波器的特征抽取器。图6.1.1 是从AlexNet论文[Krizhevsky et al., 2012] 复制的，描述了底层图像特征。

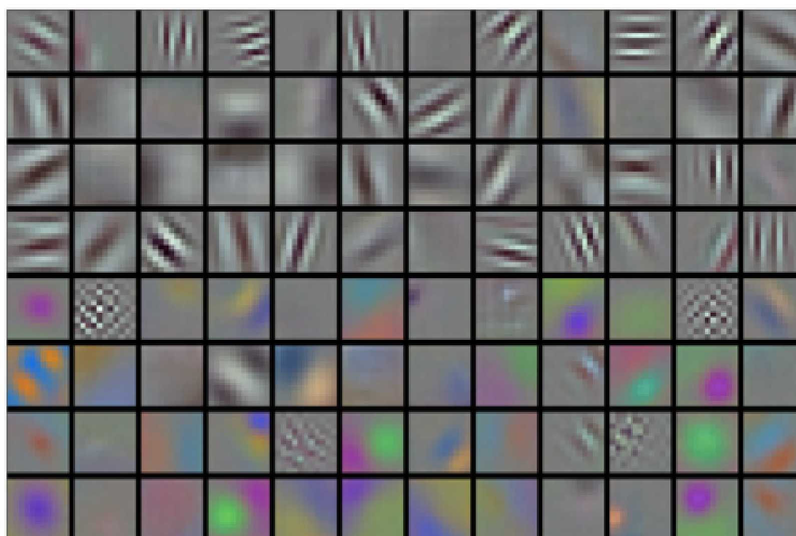


图6.1.1: AlexNet第一层学习到的特征抽取器。

AlexNet的更高层建立在这些底层表示的基础上，以表示更大的特征，如眼睛、鼻子、草叶等等。而更高的层可以检测整个物体，如人、飞机、狗或飞盘。最终的隐藏神经元可以学习图像的综合表示，从而使属于不同类别的数据易于区分。尽管一直有一群执着的研究者不断钻研，试图学习视觉数据的逐级表征，然而很长一段时间里这些尝试都未有突破。深度卷积神经网络的突破出现在2012年。突破可归因于两个关键因素。

缺少的成分：数据

包含许多特征的深度模型需要大量的有标签数据，才能显著优于基于凸优化的传统方法（如线性方法和核方法）。然而，限于早期计算机有限的存储和90年代有限的研究预算，大部分研究只基于小的公开数据集。例如，不少研究论文基于加州大学欧文分校（UCI）提供的若干个公开数据集，其中许多数据集只有几百至几千张在非自然环境下以低分辨率拍摄的图像。这一状况在2010年前后兴起的大数据浪潮中得到改善。2009年，ImageNet数据集发布，并发起ImageNet挑战赛：要求研究人员从100万个样本中训练模型，以区分1000个不同类别的对象。ImageNet数据集由斯坦福教授李飞飞小组的研究人员开发，利用谷歌图像搜索（Google Image Search）对每一类图像进行预筛选，并利用亚马逊众包（Amazon Mechanical Turk）来标注每张图片的相关类别。这种规模是前所未有的。这项被称为ImageNet的挑战赛推动了计算机视觉和机器学习研究的发展，挑战研究人员确定哪些模型能够在更大的数据规模下表现最好。

缺少的成分：硬件

深度学习对计算资源要求很高，训练可能需要数百个迭代周期，每次迭代都需要通过代价高昂的许多线性代数层传递数据。这也是为什么在20世纪90年代至21世纪初，优化凸目标的简单算法是研究人员的首选。然而，用GPU训练神经网络改变了这一格局。图形处理器（Graphics Processing Unit，GPU）早年用来加速图形处理，使电脑游戏玩家受益。GPU可优化高吞吐量的 4×4 矩阵和向量乘法，从而服务于基本的图形任务。幸运的是，这些数学运算与卷积层的计算惊人地相似。由此，英伟达（NVIDIA）和ATI已经开始为通用计算操作优化gpu，甚至把它们作为 通用*GPU*（general-purpose GPUs，GPGPU）来销售。

那么GPU比CPU强在哪里呢？

首先，我们深度理解一下中央处理器（Central Processing Unit，CPU）的核心。CPU的每个核心都拥有高时钟频率的运行能力，和高达数MB的三级缓存(L3 Cache)。它们非常适合执行各种指令，具有分支预测器、深层流水线和其他使CPU能够运行各种程序的功能。然而，这种明显的优势也是它的致命弱点：通用核心的制造成本非常高。它们需要大量的芯片面积、复杂的支持结构（内存接口、内核之间的缓存逻辑、高速互连等等），而且它们在任何单个任务上的性能都相对较差。现代笔记本电脑最多有4核，即使是高端服务器也很少超过64核，因为它们的性价比不高。

相比于CPU，GPU由 $100 \sim 1000$ 个小的处理单元组成（NVIDIA、ATI、ARM和其他芯片供应商之间的细节稍有不同），通常被分成更大的组（NVIDIA称之为warps）。虽然每个GPU核心都相对较弱，有时甚至以低于1GHz的时钟频率运行，但庞大的核心数量使GPU比CPU快几个数量级。例如，NVIDIA最近一代的Ampere GPU架构为每个芯片提供了高达312 TFlops的浮点性能，而CPU的浮点性能到目前为止还没有超过1 TFlops。之所以有如此大的差距，原因其实很简单：首先，功耗往往会随时钟频率呈二次方增长。对于一个CPU核心，假设它的运行速度比GPU快4倍，你可以使用16个GPU内核取代，那么GPU的综合性能就是CPU的 $16 \times 1/4 = 4$ 倍。其次，GPU内核要简单得多，这使得它们更节能。此外，深度学习中的许多操作需要相对较高的内存带宽，而GPU拥有10倍于CPU的带宽。

回到2012年的重大突破，当Alex Krizhevsky和Ilya Sutskever实现了可以在GPU硬件上运行的深度卷积神经网络时，一个重大突破出现了。他们意识到卷积神经网络中的计算瓶颈：卷积和矩阵乘法，都是可以在硬件上并行化的操作。于是，他们使用两个显存为3GB的NVIDIA GTX580 GPU实现了快速卷积运算。他们的创新cuda-convnet[83]几年来它一直是行业标准，并推动了深度学习热潮。

6.1.2 AlexNet

2012年，AlexNet横空出世。它首次证明了学习到的特征可以超越手工设计的特征。它一举了打破计算机视觉研究的现状。AlexNet使用了8层卷积神经网络，并以很大的优势赢得了2012年ImageNet图像识别挑战赛。

AlexNet和LeNet的架构非常相似，如 图6.1.2 所示。注意，这里我们提供了一个稍微精简版本的AlexNet，去除了当年需要两个小型GPU同时运算的设计特点。

[83] https://code.google.com/archive/p/cuda-convnet/

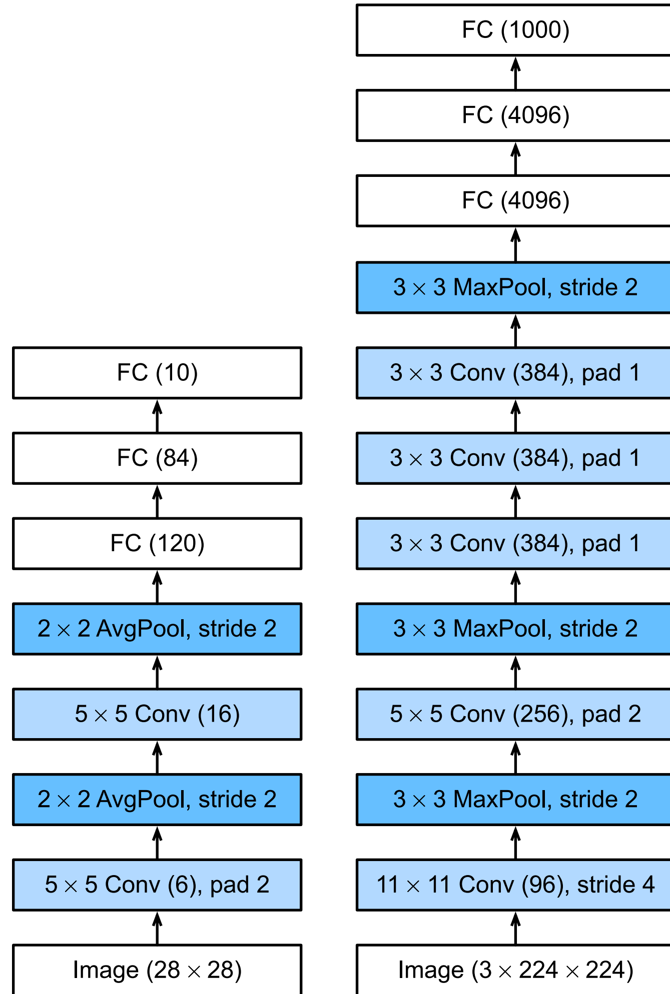


图6.1.2: 从LeNet（左）到AlexNet（right）

AlexNet和LeNet的设计理念非常相似，但也存在显著差异。首先，AlexNet比相对较小的LeNet5要深得多。AlexNet由八层组成：五个卷积层、两个全连接隐藏层和一个全连接输出层。其次，AlexNet使用ReLU而不是sigmoid作为其激活函数。下面，让我们深入研究AlexNet的细节。

模型设计

在AlexNet的第一层,卷积窗口的形状是 11×11。由于大多数ImageNet中图像的宽和高比MNIST图像的多10倍以上，因此，需要一个更大的卷积窗口来捕获目标。第二层中的卷积窗形状被缩减为 5×5，然后是 3×3。此外，在第一层、第二层和第五层之后，加入窗口形状为 3×3、步幅为2的最大池化层。此外，AlexNet的卷积通道是LeNet的10倍。

在最后一个卷积层后有两个全连接层，分别有4096个输出。这两个巨大的全连接层拥有将近1GB的模型参数。由于早期GPU显存有限，原版的AlexNet采用了双数据流设计，使得每个GPU只负责存储和计算模型的一半参数。幸运的是，现在GPU显存相对充裕，所以我们现在很少需要跨GPU分解模型（因此，我们的AlexNet模型在这方面与原始论文稍有不同）。

激活函数

此外，AlexNet将sigmoid激活函数改为更简单的ReLU激活函数。一方面，ReLU激活函数的计算更简单，它不需要如sigmoid激活函数那般复杂的求幂运算。另一方面，当使用不同的参数初始化方法时，ReLU激活函数使训练模型更加容易。当sigmoid激活函数的输出非常接近于0或1时，这些区域的梯度几乎为0，因此反向传播无法继续更新一些模型参数。相反，ReLU激活函数在正区间的梯度总是1。因此，如果模型参数没有正确初始化，sigmoid函数可能在正区间内得到几乎为0的梯度，从而使模型无法得到有效的训练。

容量控制和预处理

AlexNet通过dropout（3.6节）控制全连接层的模型复杂度，而LeNet只使用了权重衰减。为了进一步扩充数据，AlexNet在训练时增加了大量的图像增强数据，如翻转、裁切和变色。这使得模型更健壮，更大的样本量有效地减少了过拟合。我们将在 `sec_image_augmentation` 中更详细地讨论数据扩充。

```
from mxnet import np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()

net = nn.Sequential()

net.add(
    # 这里，我们使用一个11*11的更大窗口来捕捉对象。
    # 同时，步幅为4，以减少输出的高度和宽度。
    # 另外，输出通道的数目远大于LeNet
    nn.Conv2D(96, kernel_size=11, strides=4, activation='relu'),
    nn.MaxPool2D(pool_size=3, strides=2),
    # 减小卷积窗口，使用填充为2来使得输入与输出的高和宽一致，且增大输出通道数
    nn.Conv2D(256, kernel_size=5, padding=2, activation='relu'),
    nn.MaxPool2D(pool_size=3, strides=2),
    # 使用三个连续的卷积层和一个较小的卷积窗口。
    # 除了最后的卷积层，输出通道的数量进一步增加。
    # 前两个卷积层后不使用池化层来减小输入的高和宽
    nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
    nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
    nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'),
    nn.MaxPool2D(pool_size=3, strides=2),
    # 这里，全连接层的输出数量是LeNet中的好几倍。使用dropout层来减轻过度拟合
    nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
    nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
    # 最后是输出层。由于这里使用Fashion-MNIST，所以用类别数为10，而非论文中的1000
    nn.Dense(10))
```

我们构造了一个高度和宽度都为224的单通道数据，来观察每一层输出的形状。它与 图6.1.2 中的AlexNet架构相匹配。

```
X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

```
conv0 output shape:	 (1, 96, 54, 54)
pool0 output shape:	 (1, 96, 26, 26)
conv1 output shape:	 (1, 256, 26, 26)
pool1 output shape:	 (1, 256, 12, 12)
conv2 output shape:	 (1, 384, 12, 12)
conv3 output shape:	 (1, 384, 12, 12)
conv4 output shape:	 (1, 256, 12, 12)
pool2 output shape:	 (1, 256, 5, 5)
dense0 output shape:	 (1, 4096)
dropout0 output shape:	 (1, 4096)
dense1 output shape:	 (1, 4096)
dropout1 output shape:	 (1, 4096)
dense2 output shape:	 (1, 10)
```

6.1.3 读取数据集

尽管本文中AlexNet是在ImageNet上进行训练的，但我们在这里使用的是Fashion-MNIST数据集。因为即使在现代GPU上，训练ImageNet模型，同时使其收敛可能需要数小时或数天的时间。将AlexNet直接应用于Fashion-MNIST的一个问题是，Fashion-MNIST的图像分辨率（28×28像素）低于ImageNet图像。为了解决这个问题，我们将它们增加到 224×224（通常来讲这不是一个明智的做法，但我们在这里这样做是为了有效使用AlexNet结构）。我们使用 `d2l.load_data_fashion_mnist` 函数中的 `resize` 参数执行此调整。

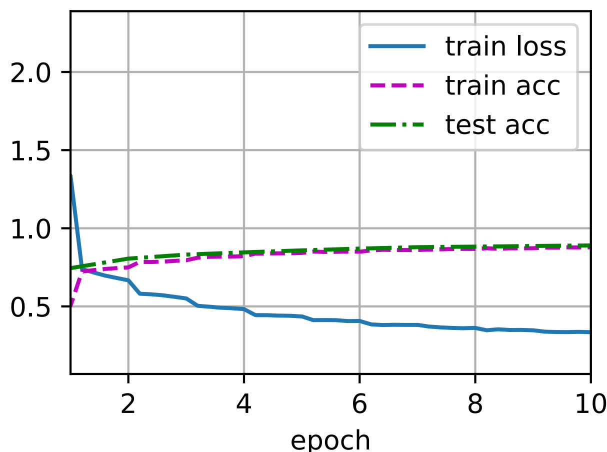
```
batch_size = 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
```

6.1.4 训练AlexNet

现在，我们可以开始训练AlexNet了。与 5.6节 中的LeNet相比，这里的主要变化是使用更小的学习速率训练，这是因为网络更深更广、图像分辨率更高，训练卷积神经网络就更昂贵。

```
lr, num_epochs = 0.01, 10
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.335, train acc 0.878, test acc 0.890
4059.9 examples/sec on gpu(0)
```



6.1.5 小结

- AlexNet的结构与LeNet相似，但使用了更多的卷积层和更多的参数来拟合大规模的ImageNet数据集。
- 今天，AlexNet已经被更有效的结构所超越，但它是从浅层网络到深层网络的关键一步。
- 尽管AlexNet的代码只比LeNet多出几行，但学术界花了很多年才接受深度学习这一概念，并应用其出色的实验结果。这也是由于缺乏有效的计算工具。
- Dropout、ReLU和预处理是提升计算机视觉任务性能的其他关键步骤。

6.1.6 练习

1. 试着增加迭代周期。对比LeNet的结果有什么不同？为什么？
2. AlexNet对于Fashion-MNIST数据集来说可能太复杂了。
    1. 尝试简化模型以加快训练速度，同时确保准确性不会显著下降。
    2. 设计一个更好的模型，可以直接在 28×28 图像上工作。
3. 修改批量大小，并观察模型精度和GPU显存变化。
4. 分析了AlexNet的计算性能。
    1. 在AlexNet中主要是哪部分占用显存？
    2. 在AlexNet中主要是哪部分需要更多的计算？
    3. 计算结果时显存带宽如何？
5. 将dropout和ReLU应用于LeNet-5，效果有提升吗？再试试预处理会怎么样？

Discussions[84]

6.2 使用块的网络（VGG）

虽然 AlexNet 证明深层神经网络卓有成效，但它没有提供一个通用的模板来指导后续的研究人员设计新的网络。在下面的几个章节中，我们将介绍一些常用于设计深层神经网络的启发式概念。

与芯片设计中工程师从放置晶体管到逻辑元件再到逻辑块的过程类似，神经网络结构的设计也逐渐变得更加抽象。研究人员开始从单个神经元的角度思考问题，发展到整个层次，现在又转向模块，重复各层的模式。

使用块的想法首先出现在牛津大学的 视觉几何组（visualgeometry Group）[85] (VGG)的 *VGG*网络中。通过使用循环和子程序，可以很容易地在任何现代深度学习框架的代码中实现这些重复的结构。

6.2.1 VGG 块

经典卷积神经网络的基本组成部分是下面的这个序列：1. 带填充以保持分辨率的卷积层；1. 非线性激活函数，如ReLU；1. 池化层，如最大池化层。

而一个 VGG 块与之类似，由一系列卷积层组成，后面再加上用于空间下采样的最大池化层。在最初的 VGG 论文 [Simonyan & Zisserman, 2014] 中，作者使用了带有 3×3 卷积核、填充为 1（保持高度和宽度）的卷积层，和带有 2×2 池化窗口、步幅为 2（每个块后的分辨率减半）的最大池化层。在下面的代码中，我们定义了一个名为 `vgg_block` 的函数来实现一个 VGG 块。

该函数有两个参数，分别对应于卷积层的数量 `num_convs` 和输出通道的数量 `num_channels`.

```
from mxnet import np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()

def vgg_block(num_convs, num_channels):
    blk = nn.Sequential()
    for _ in range(num_convs):
        blk.add(
            nn.Conv2D(num_channels, kernel_size=3, padding=1,
                      activation='relu'))
    blk.add(nn.MaxPool2D(pool_size=2, strides=2))
    return blk
```

[84] https://discuss.d2l.ai/t/1864

[85] http://www.robots.ox.ac.uk/~vgg/

6.2.2 VGG 网络

与 AlexNet、LeNet 一样，VGG 网络可以分为两部分：第一部分主要由卷积层和池化层组成，第二部分由全连接层组成。如 图6.2.1 中所示。

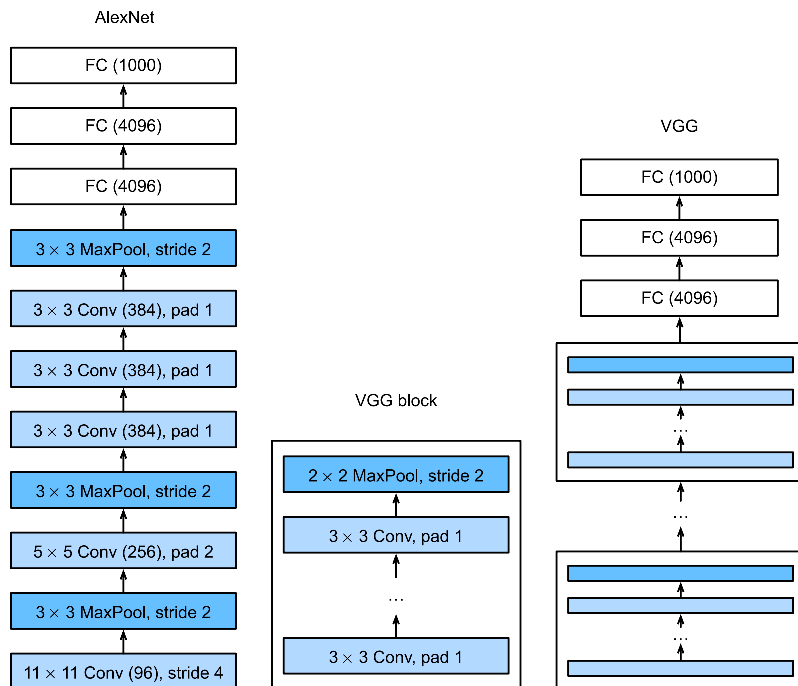


图6.2.1: 从AlexNet到VGG，它们本质上都是块设计。

VGG神经网络连续连接 图6.2.1 的几个 VGG 块（在 vgg_block 函数中定义）。其中有超参数变量 conv_arch 。该变量指定了每个VGG块里卷积层个数和输出通道数。全连接模块则与AlexNet中的相同。

原始 VGG 网络有 5 个卷积块，其中前两个块各有一个卷积层，后三个块各包含两个卷积层。第一个模块有 64 个输出通道，每个后续模块将输出通道数量翻倍，直到该数字达到 512。由于该网络使用 8 个卷积层和 3 个全连接层，因此它通常被称为 VGG-11。

```
conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
```

下面的代码实现了 VGG-11。可以通过在 conv_arch 上执行 for 循环来简单实现。

```
def vgg(conv_arch):
    net = nn.Sequential()
    # 卷积层部分
    for (num_convs, num_channels) in conv_arch:
        net.add(vgg_block(num_convs, num_channels))
    # 全连接层部分
    net.add(nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
```

(continues on next page)

(continued from previous page)

```
            nn.Dense(4096, activation='relu'), nn.Dropout(0.5), nn.Dense(10))
    return net

net = vgg(conv_arch)
```

接下来，我们将构建一个高度和宽度为 224 的单通道数据样本，以观察每个层输出的形状。

```
net.initialize()
X = np.random.uniform(size=(1, 1, 224, 224))
for blk in net:
    X = blk(X)
    print(blk.name, 'output shape:\t', X.shape)
```

```
sequential1 output shape:    (1, 64, 112, 112)
sequential2 output shape:    (1, 128, 56, 56)
sequential3 output shape:    (1, 256, 28, 28)
sequential4 output shape:    (1, 512, 14, 14)
sequential5 output shape:    (1, 512, 7, 7)
dense0 output shape:         (1, 4096)
dropout0 output shape:       (1, 4096)
dense1 output shape:         (1, 4096)
dropout1 output shape:       (1, 4096)
dense2 output shape:         (1, 10)
```

正如你所看到的，我们在每个块的高度和宽度减半，最终高度和宽度都为7。最后再展平表示，送入全连接层处理。

6.2.3 训练

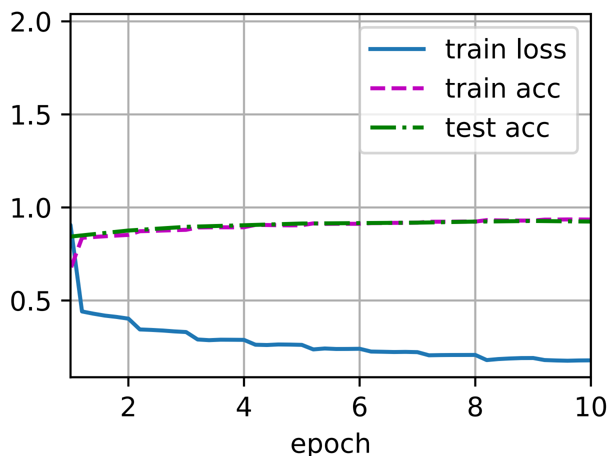
由于 VGG-11 比 AlexNet 计算量更大，因此我们构建了一个通道数较少的网络，足够用于训练Fashion-MNIST数据集。

```
ratio = 4
small_conv_arch = [(pair[0], pair[1] // ratio) for pair in conv_arch]
net = vgg(small_conv_arch)
```

除了使用略高的学习率外，模型训练过程与 6.1节 中的 AlexNet 类似。

```
lr, num_epochs, batch_size = 0.05, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.178, train acc 0.935, test acc 0.924
1792.0 examples/sec on gpu(0)
```



6.2.4 小结

- VGG-11 使用可复用的卷积块构造网络。不同的 VGG 模型可通过每个块中卷积层数量和输出通道数量的差异来定义。
- 块的使用导致网络定义的非常简洁。使用块可以有效地设计复杂的网络。
- 在VGG论文中，Simonyan和Ziserman尝试了各种架构。特别是他们发现深层且窄的卷积（即3×3）比较浅层且宽的卷积更有效。

6.2.5 练习

1. 打印层的尺寸时，我们只看到 8 个结果，而不是 11 个结果。剩余的 3 层信息去哪了？
2. 与 AlexNet 相比，VGG 的计算要慢得多，而且它还需要更多的显存。分析出现这种情况的原因。
3. 尝试将Fashion-MNIST数据集图像的高度和宽度从 224 改为 96。这对实验有什么影响？
4. 请参考 VGG 论文 [Simonyan & Zisserman, 2014] 中的表1构建其他常见模型，如 VGG-16 或 VGG-19。

Discussions[86]

[86] https://discuss.d2l.ai/t/1867

6.3 网络中的网络（NiN）

LeNet、AlexNet 和 VGG 都有一个共同的设计模式：通过一系列的卷积层与池化层来提取空间结构特征；然后通过全连接层对特征的表征进行处理。AlexNet 和 VGG 对 LeNet 的改进主要在于如何扩大和加深这两个模块。或者，可以想象在这个过程的早期使用全连接层。然而，如果使用稠密层了，可能会完全放弃表征的空间结构。网络中的网络 (*NiN*) 提供了一个非常简单的解决方案：在每个像素的通道上分别使用多层感知机 [Lin et al., 2013]

6.3.1 NiN 块

回想一下，卷积层的输入和输出由四维张量组成，张量的每个轴分别对应样本、通道、高度和宽度。另外，全连接层的输入和输出通常是分别对应于样本和特征的二维张量。NiN 的想法是在每个像素位置（针对每个高度和宽度）应用一个全连接层。如果我们将权重连接到每个空间位置，我们可以将其视为 1×1 卷积层（如 5.4节 中所述），或作为在每个像素位置上独立作用的全连接层。从另一个角度看，即将空间维度中的每个像素视为单个样本，将通道维度视为不同特征（feature）。

图6.3.1 说明了 VGG 和 NiN 及它们的块之间主要结构差异。NiN 块以一个普通卷积层开始，后面是两个 1×1 的卷积层。这两个1×1 卷积层充当带有 ReLU 激活函数的逐像素全连接层。第一层的卷积窗口形状通常由用户设置。随后的卷积窗口形状固定为 1×1。

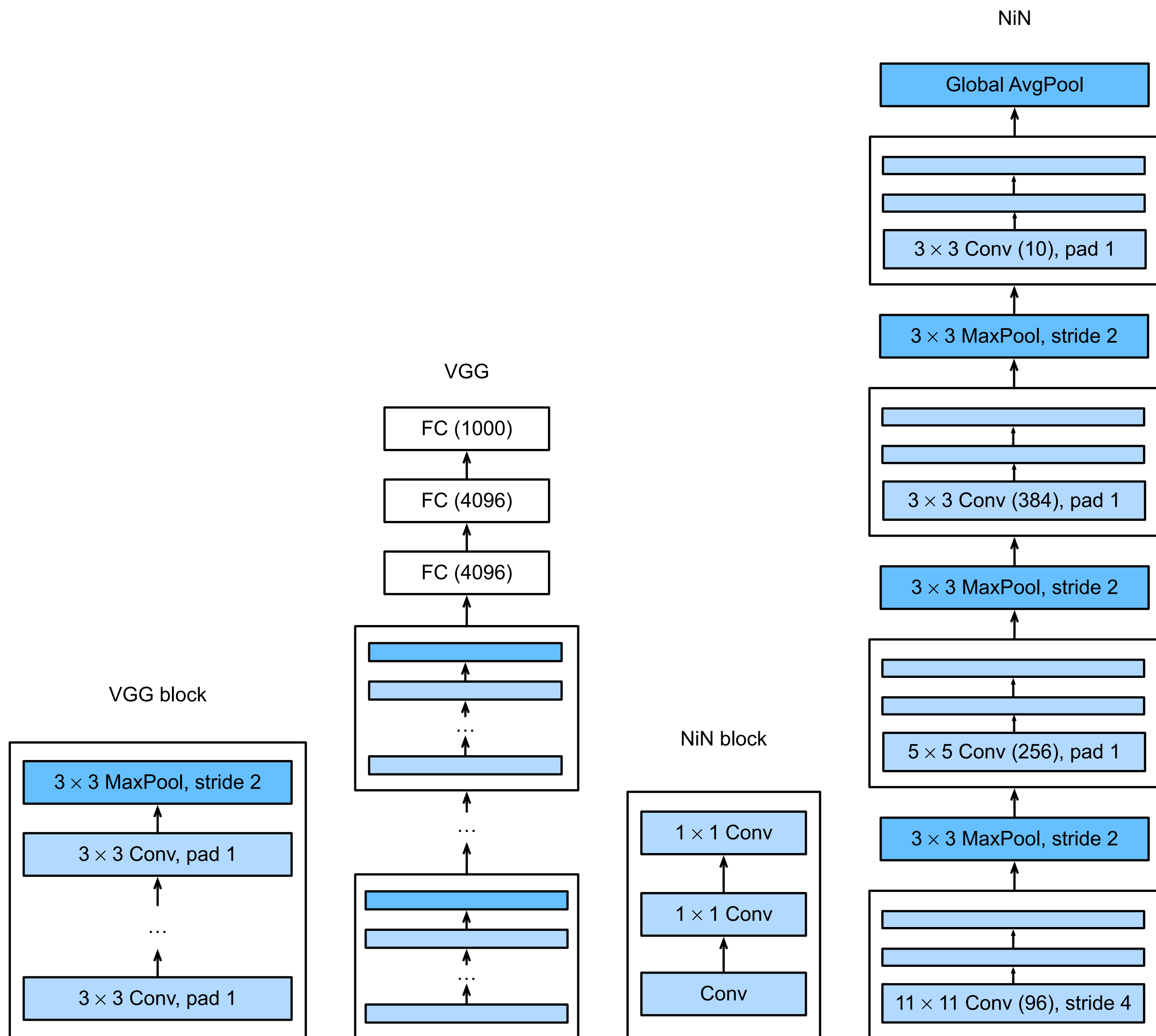


图6.3.1: 对比 VGG 和 NiN 及它们的块之间主要结构差异。

```python
from mxnet import np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()

def nin_block(num_channels, kernel_size, strides, padding):
    blk = nn.Sequential()
    blk.add(
        nn.Conv2D(num_channels, kernel_size, strides, padding,
                  activation='relu'),
        nn.Conv2D(num_channels, kernel_size=1, activation='relu'),
```

(continues on next page)

(continued from previous page)

```
        nn.Conv2D(num_channels, kernel_size=1, activation='relu'))
    return blk
```

6.3.2 NiN 模型

最初的 NiN 网络是在 AlexNet 后不久提出的，显然从中得到了一些启示。NiN使用窗口形状为 11×11、5×5 和 3×3的卷积层，输出通道数量与 AlexNet 中的相同。每个 NiN 块后有一个最大池化层，池化窗口形状为 3×3，步幅为 2。

NiN 和 AlexNet 之间的一个显著区别是 NiN 完全取消了全连接层。相反，NiN 使用一个 NiN块，其输出通道数等于标签类别的数量。最后放一个 全局平均池化层（global average pooling layer），生成一个多元逻辑向量（logits）。NiN 设计的一个优点是，它显著减少了模型所需参数的数量。然而，在实践中，这种设计有时会增加训练模型的时间。

```
net = nn.Sequential()
net.add(nin_block(96, kernel_size=11, strides=4, padding=0),
        nn.MaxPool2D(pool_size=3, strides=2),
        nin_block(256, kernel_size=5, strides=1, padding=2),
        nn.MaxPool2D(pool_size=3, strides=2),
        nin_block(384, kernel_size=3, strides=1, padding=1),
        nn.MaxPool2D(pool_size=3, strides=2), nn.Dropout(0.5),
        # 标签类别数是10
        nin_block(10, kernel_size=3, strides=1, padding=1),
        # 全局平均池化层将窗口形状自动设置成输入的高和宽
        nn.GlobalAvgPool2D(),
        # 将四维的输出转成二维的输出，其形状为(批量大小, 10)
        nn.Flatten())
```

我们创建一个数据样本来查看每个块的输出形状。

```
X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

```
sequential1 output shape:    (1, 96, 54, 54)
pool0 output shape:  (1, 96, 26, 26)
sequential2 output shape:    (1, 256, 26, 26)
pool1 output shape:  (1, 256, 12, 12)
sequential3 output shape:    (1, 384, 12, 12)
pool2 output shape:  (1, 384, 5, 5)
```

(continues on next page)

(continued from previous page)

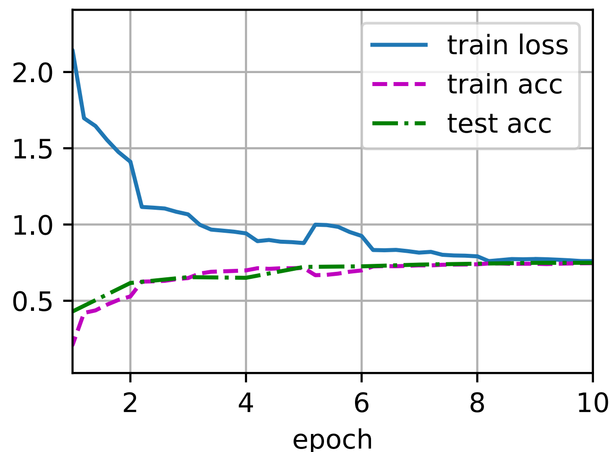
```
dropout0 output shape:       (1, 384, 5, 5)
sequential4 output shape:    (1, 10, 5, 5)
pool3 output shape:  (1, 10, 1, 1)
flatten0 output shape:       (1, 10)
```

6.3.3 训练

和以前一样，我们使用 Fashion-MNIST 来训练模型。训练 NiN 与训练 AlexNet、VGG时相似。

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.760, train acc 0.746, test acc 0.749
2951.2 examples/sec on gpu(0)
```



6.3.4 小结

- NiN使用由一个卷积层和多个 1×1 卷积层组成的块。该块可以在卷积神经网络中使用，以允许更多的每像素非线性。
- NiN去除了容易造成过拟合的全连接层，将它们替换为全局平均池化层（即在所有位置上进行求和）。该池化层通道数量为所需的输出数量（例如，Fashion-MNIST的输出为10）。
- 移除全连接层可减少过拟合，同时显著减少NiN的参数。
- NiN的设计影响了许多后续卷积神经网络的设计。

6.3.5 练习

1. 调整NiN的超参数，以提高分类准确性。
2. 为什么NiN块中有两个 1×1 卷积层？删除其中一个，然后观察和分析实验现象。
3. 计算NiN的资源使用情况。
    1. 参数的数量是多少？
    2. 计算量是多少？
    3. 训练期间需要多少显存？
    4. 预测期间需要多少显存？
4. 一次性直接将 $384 \times 5 \times 5$ 的表示缩减为 $10 \times 5 \times 5$ 的表示，会存在哪些问题？

Discussions[87]

6.4 含并行连结的网络（GoogLeNet）

在2014年的ImageNet图像识别挑战赛中，一个名叫*GoogLeNet* [Szegedy et al., 2015] 的网络结构大放异彩。GoogLeNet吸收了NiN中串联网络的思想，并在此基础上做了改进。这篇论文的一个重点是解决了什么样大小的卷积核最合适的问题。毕竟，以前流行的网络使用小到 1×1 ，大到 11×11 的卷积核。本文的一个观点是，有时使用不同大小的卷积核组合是有利的。在本节中，我们将介绍一个稍微简化的GoogLeNet版本：我们省略了一些为稳定训练而添加的特殊特性，但是现在有了更好的训练算法，这些特性不是必要的。

6.4.1 Inception块

在GoogLeNet中，基本的卷积块被称为*Inception*块（Inception block）。这很可能得名于电影《盗梦空间》（Inception），因为电影中的一句话“我们需要走得更深”（“We need to go deeper”）。

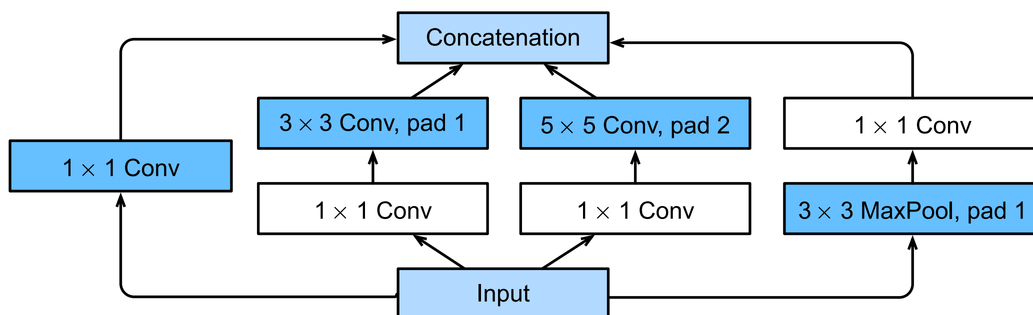


图6.4.1: Inception块的结构。

[87] https://discuss.d2l.ai/t/1870

如 图6.4.1 所示，Inception块由四条并行路径组成。前三条路径使用窗口大小为 1×1、3×3 和 5×5 的卷积层，从不同空间大小中提取信息。中间的两条路径在输入上执行 1×1 卷积，以减少通道数，从而降低模型的复杂性。第四条路径使用 3×3 最大池化层，然后使用 1×1 卷积层来改变通道数。这四条路径都使用合适的填充来使输入与输出的高和宽一致，最后我们将每条线路的输出在通道维度上连结，并构成Inception块的输出。在Inception块中，通常调整的超参数是每层输出通道的数量。

```
from mxnet import np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()

class Inception(nn.Block):
    # `c1`--`c4` 是每条路径的输出通道数
    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # 线路1，单1 x 1卷积层
        self.p1_1 = nn.Conv2D(c1, kernel_size=1, activation='relu')
        # 线路2，1 x 1卷积层后接3 x 3卷积层
        self.p2_1 = nn.Conv2D(c2[0], kernel_size=1, activation='relu')
        self.p2_2 = nn.Conv2D(c2[1], kernel_size=3, padding=1,
                              activation='relu')
        # 线路3，1 x 1卷积层后接5 x 5卷积层
        self.p3_1 = nn.Conv2D(c3[0], kernel_size=1, activation='relu')
        self.p3_2 = nn.Conv2D(c3[1], kernel_size=5, padding=2,
                              activation='relu')
        # 线路4，3 x 3最大池化层后接1 x 1卷积层
        self.p4_1 = nn.MaxPool2D(pool_size=3, strides=1, padding=1)
        self.p4_2 = nn.Conv2D(c4, kernel_size=1, activation='relu')

    def forward(self, x):
        p1 = self.p1_1(x)
        p2 = self.p2_2(self.p2_1(x))
        p3 = self.p3_2(self.p3_1(x))
        p4 = self.p4_2(self.p4_1(x))
        # 在通道维度上连结输出
        return np.concatenate((p1, p2, p3, p4), axis=1)
```

那么为什么GoogLeNet这个网络如此有效呢？首先我们考虑一下滤波器（filter）的组合，它们可以用各种滤波器尺寸探索图像，这意味着不同大小的滤波器可以有效地识别不同范围的图像细节。同时，我们可以为不同的滤波器分配不同数量的参数。

6.4.2 GoogLeNet 模型

如 图6.4.2 所示，GoogLeNet 一共使用 9 个Inception块和全局平均池化层的堆叠来生成其估计值。Inception块之间的最大池化层可降低维度。第一个模块类似于 AlexNet 和 LeNet，Inception块的栈从VGG继承，全局平均池化层避免了在最后使用全连接层。

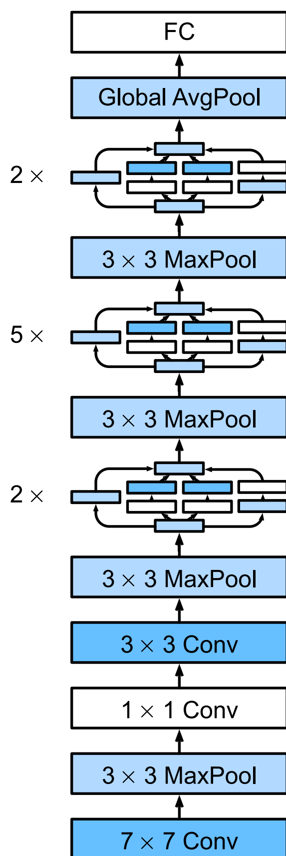


图6.4.2: GoogLeNet结构。

现在，我们逐一实现GoogLeNet的每个模块。第一个模块使用 64 个通道、7×7 卷积层。

```
b1 = nn.Sequential()
b1.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3, activation='relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

第二个模块使用两个卷积层：第一个卷积层是 64个通道、1×1 卷积层；第二个卷积层使用将通道数量增加三倍的 3×3 卷积层。这对应于 Inception 块中的第二条路径。

```
b2 = nn.Sequential()
b2.add(nn.Conv2D(64, kernel_size=1, activation='relu'),
       nn.Conv2D(192, kernel_size=3, padding=1, activation='relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

第三个模块串联两个完整的Inception块。第一个 Inception 块的输出通道数为 $64+128+32+32=256$，四个路径之间的输出通道数量比为 $64:128:32:32=2:4:1:1$。第二个和第三个路径首先将输入通道的数量分别减少到 $96/192=1/2$ 和 $16/192=1/12$，然后连接第二个卷积层。第二个 Inception 块的输出通道数增加到 $128+192+96+64=480$，四个路径之间的输出通道数量比为 $128:192:96:64=4:6:3:2$。第二条和第三条路径首先将输入通道的数量分别减少到 $128/256=1/2$ 和 $32/256=1/8$。

```
b3 = nn.Sequential()
b3.add(Inception(64, (96, 128), (16, 32), 32),
       Inception(128, (128, 192), (32, 96), 64),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

第四模块更加复杂，它串联了5个Inception块，其输出通道数分别是 $192+208+48+64=512$、$160+224+64+64=512$、$128+256+64+64=512$、$112+288+64+64=528$ 和 $256+320+128+128=832$。这些路径的通道数分配和第三模块中的类似，首先是含 3×3 卷积层的第二条路径输出最多通道，其次是仅含 1×1 卷积层的第一条路径，之后是含 5×5 卷积层的第三条路径和含 3×3 最大池化层的第四条路径。其中第二、第三条路径都会先按比例减小通道数。这些比例在各个 Inception 块中都略有不同。

```
b4 = nn.Sequential()
b4.add(Inception(192, (96, 208), (16, 48), 64),
       Inception(160, (112, 224), (24, 64), 64),
       Inception(128, (128, 256), (24, 64), 64),
       Inception(112, (144, 288), (32, 64), 64),
       Inception(256, (160, 320), (32, 128), 128),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

第五模块包含输出通道数为 $256+320+128+128=832$ 和 $384+384+128+128=1024$ 的两个Inception块。其中每条路径通道数的分配思路和第三、第四模块中的一致，只是在具体数值上有所不同。需要注意的是，第五模块的后面紧跟输出层，该模块同 NiN 一样使用全局平均池化层，将每个通道的高和宽变成1。最后我们将输出变成二维数组，再接上一个输出个数为标签类别数的全连接层。

```
b5 = nn.Sequential()
b5.add(Inception(256, (160, 320), (32, 128), 128),
       Inception(384, (192, 384), (48, 128), 128), nn.GlobalAvgPool2D())

net = nn.Sequential()
net.add(b1, b2, b3, b4, b5, nn.Dense(10))
```

GoogLeNet 模型的计算复杂，而且不如 VGG 那样便于修改通道数。为了在Fashion-MNIST上有一个合理的训练时间，我们将输入的高和宽从 224 降到 96，这简化了计算。下面演示各个模块输出的形状变化。

```
X = np.random.uniform(size=(1, 1, 96, 96))
net.initialize()
for layer in net:
    X = layer(X)
```

(continues on next page)

(continued from previous page)

```
    print(layer.name, 'output shape:\t', X.shape)
```

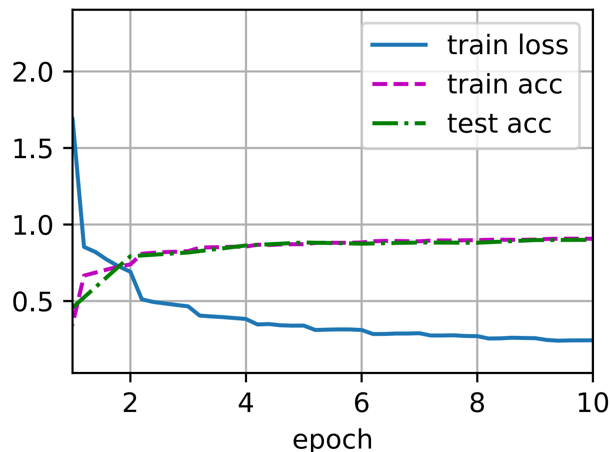
```
sequential0 output shape:    (1, 64, 24, 24)
sequential1 output shape:    (1, 192, 12, 12)
sequential2 output shape:    (1, 480, 6, 6)
sequential3 output shape:    (1, 832, 3, 3)
sequential4 output shape:    (1, 1024, 1, 1)
dense0 output shape:         (1, 10)
```

6.4.3 训练

和以前一样，我们使用 Fashion-MNIST 数据集来训练我们的模型。在训练之前，我们将图片转换为 96×96 分辨率。

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.243, train acc 0.907, test acc 0.900
2197.7 examples/sec on gpu(0)
```



6.4.4 小结

- Inception 块相当于一个有4条路径的子网络。它通过不同窗口形状的卷积层和最大池化层来并行抽取信息，并使用 1×1 卷积层减少每像素级别上的通道维数从而降低模型复杂度。
- GoogLeNet将多个设计精细的Inception块与其他层（卷积层、全连接层）串联起来。其中Inception块的通道数分配之比是在 ImageNet 数据集上通过大量的实验得来的。
- GoogLeNet 和它的后继者们一度是 ImageNet 上最有效的模型之一：它以较低的计算复杂度提供了类似的测试精度。

6.4.5 练习

1. GoogLeNet 有数个后续版本。尝试实现并运行它们，然后观察实验结果。这些后续版本包括：
   - 添加批量归一化层 [Ioffe & Szegedy, 2015]（batch normalization），在 6.5节中将介绍）。
   - 对 Inception 模块进行调整。
   - 使用标签平滑（label smoothing）进行模型正则化 [Szegedy et al., 2016]。
   - 加入残差连接 [Szegedy et al., 2017]，（6.6节 将介绍）。
2. 使用 GoogLeNet 的最小图像大小是多少？
3. 将 AlexNet、VGG 和 NiN 的模型参数大小与 GoogLeNet 进行比较。后两个网络结构是如何显著减少模型参数大小的？

Discussions[88]

6.5 批量归一化

训练深层神经网络是十分困难的，特别是在较短的时间内使他们收敛更加棘手。在本节中，我们将介绍 批量归一化（batch normalization）[Ioffe & Szegedy, 2015]，这是一种流行且有效的技术，可持续加速深层网络的收敛速度。再结合在 6.6节 中将介绍的残差块，批量归一化使得研究人员能够训练 100 层以上的网络。

6.5.1 训练深层网络

为什么需要批量归一化层呢？让我们来回顾一下训练神经网络时出现的一些实际挑战。

首先，数据预处理的方式通常会对最终结果产生巨大影响。回想一下我们应用多层感知机来预测房价的例子（3.10节 ）。使用真实数据时，我们的第一步是标准化输入特征，使其平均值为0，方差为1。直观地说，这种标准化可以很好地与我们的优化器配合使用，因为它可以将参数的量级进行统一。

第二，对于典型的多层感知机或卷积神经网络。当我们训练时，中间层中的变量（例如，多层感知机中的仿射变换输出）可能具有更广的变化范围：不论是沿着从输入到输出的层，跨同一层中的单元，或是随着时间

[88] https://discuss.d2l.ai/t/1873

的推移，模型参数的随着训练更新变幻莫测。批量归一化的发明者非正式地假设，这些变量分布中的这种偏移可能会阻碍网络的收敛。直观地说，我们可能会猜想，如果一个层的可变值是另一层的 100 倍，这可能需要对学习率进行补偿调整。

第三，更深层的网络很复杂，容易过拟合。这意味着正则化变得更加重要。

批量归一化应用于单个可选层（也可以应用到所有层），其原理如下：在每次训练迭代中，我们首先归一化输入，即通过减去其均值并除以其标准差，其中两者均基于当前小批量处理。接下来，我们应用比例系数和比例偏移。正是由于这个基于批量统计的标准化，才有了批量归一化的名称。。

请注意，如果我们尝试使用大小为 1 的小批量应用批量归一化，我们将无法学到任何东西。这是因为在减去均值之后，每个隐藏单元将为 0。所以，只有使用足够大的小批量，批量归一化这种方法才是有效且稳定的。请注意，在应用批量归一化时，批量大小的选择可能比没有批量归一化时更重要。

从形式上来说，用 $\mathbf{x} \in \mathcal{B}$ 表示一个来自小批量 $\mathcal{B}$ 的输入，批量归一化BN 根据以下表达式转换 $\mathbf{x}$:

$$\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_\mathcal{B}}{\hat{\boldsymbol{\sigma}}_\mathcal{B}} + \boldsymbol{\beta}. \tag{6.5.1}$$

在 (6.5.1) 中，$\hat{\boldsymbol{\mu}}_\mathcal{B}$ 是样本均值，$\hat{\boldsymbol{\mu}}_\mathcal{B}$ 是小批量 $\mathcal{B}$ 的样本标准差。应用标准化后，生成的小批量的平均值为 0 和单位方差为 1。由于单位方差（与其他一些魔法数）是一个任意的选择，因此我们通常包含 拉伸参数（scale）$\boldsymbol{\gamma}$ 和 偏移参数（shift）$\boldsymbol{\beta}$，它们的形状与 $\mathbf{x}$ 相同。请注意，$\boldsymbol{\gamma}$ 和 $\boldsymbol{\beta}$ 是需要与其他模型参数一起学习的参数。

由于在训练过程中，中间层的变化幅度不能过于剧烈，而批量归一化将每一层主动居中，并将它们重新调整为给定的平均值和大小（通过 $\hat{\boldsymbol{\mu}}_\mathcal{B}$ 和 $\hat{\boldsymbol{\sigma}}_\mathcal{B}$）。

从形式上来看，我们计算出 (6.5.1) 中的 $\hat{\boldsymbol{\mu}}_\mathcal{B}$ 和 $\hat{\boldsymbol{\sigma}}_\mathcal{B}$，如下所示：

$$\begin{aligned} \hat{\boldsymbol{\mu}}_\mathcal{B} &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x},\\
\hat{\boldsymbol{\sigma}}_\mathcal{B}^2 &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon.\end{aligned} \tag{6.5.2}$$

请注意，我们在方差估计值中添加一个小常量 $\epsilon > 0$，以确保我们永远不会尝试除以零，即使在经验方差估计值可能消失的情况下也是如此。估计值 $\hat{\boldsymbol{\mu}}_\mathcal{B}$ 和 $\hat{\boldsymbol{\sigma}}_\mathcal{B}$ 通过使用平均值和方差的噪声（noise）估计来抵消缩放问题。你可能会认为这种噪声是一个问题，而事实上它是有益的。

事实证明, 这是深度学习中一个反复出现的主题。由于理论上尚未明确表述的原因, 优化中的各种噪声源通常会导致更快的训练和较少的过拟合: 这种变化似乎是正则化的一种形式。在一些初步研究中, [Teye et al., 2018] 和 [Luo et al., 2018] 分别将批量归一化的性质与贝叶斯先验相关联。这些理论揭示了为什么批量归一化最适应 $50 \sim 100$ 范围中的中等小批量尺寸的难题。

另外，批量归一化图层在”训练模式“（通过小批量统计数据归一化）和“预测模式”（通过数据集统计归一化）中的功能不同。在训练过程中，我们无法得知使用整个数据集来估计平均值和方差，所以只能根据每个小批次的平均值和方差不断训练模型。而在预测模式下，可以根据整个数据集精确计算批量归一化所需的平均值和方差。

现在，我们了解一下批量归一化在实践中是如何工作的。

6.5.2 批量归一化层

回想一下，批量归一化和其他图层之间的一个关键区别是，由于批量归一化在完整的小批次上运行，因此我们不能像以前在引入其他图层时那样忽略批处理的尺寸大小。我们在下面讨论这两种情况：全连接层和卷积层，他们的批量归一化实现略有不同。

全连接层

通常，我们将批量归一化层置于全连接层中的仿射变换和激活函数之间。设全连接层的输入为 u ，权重参数和偏置参数分别为 $\mathbf{W}$ 和 $\mathbf{b}$ ，激活函数为 ϕ ，批量归一化的运算符为 BN 。那么，使用批量归一化的全连接层的输出的计算详情如下：

$$\mathbf{h} = \phi(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b})). \tag{6.5.3}$$

回想一下，均值和方差是在应用变换的"相同"小批量上计算的。

卷积层

同样，对于卷积层，我们可以在卷积层之后和非线性激活函数之前应用批量归一化。当卷积有多个输出通道时，我们需要对这些通道的"每个"输出执行批量归一化，每个通道都有自己的拉伸（scale）和偏移（shift）参数，这两个参数都是标量。假设我们的微批次包含 m 个示例，并且对于每个通道，卷积的输出具有高度 p 和宽度 q。那么对于卷积层，我们在每个输出通道的 $m \cdot p \cdot q$ 个元素上同时执行每个批量归一化。因此，在计算平均值和方差时，我们会收集所有空间位置的值，然后在给定通道内应用相同的均值和方差，以便在每个空间位置对值进行归一化。

预测过程中的批量归一化

正如我们前面提到的，批量归一化在训练模式和预测模式下的行为通常不同。首先，将训练好的模型用于预测时，我们不再需要样本均值中的噪声以及在微批次上估计每个小批次产生的样本方差了。其次，例如，我们可能需要使用我们的模型对逐个样本进行预测。一种常用的方法是通过移动平均估算整个训练数据集的样本均值和方差，并在预测时使用它们得到确定的输出。可见，和 dropout 一样，批量归一化层在训练模式和预测模式下的计算结果也是不一样的。

6.5.3 从零实现

下面，我们从头开始实现一个具有张量的批量归一化层。

```
from mxnet import autograd, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l
```

(continues on next page)

(continued from previous page)

```
npx.set_np()

def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # 通过 `autograd` 来判断当前模式是训练模式还是预测模式
    if not autograd.is_training():
        # 如果是在预测模式下，直接使用传入的移动平均所得的均值和方差
        X_hat = (X - moving_mean) / np.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # 使用全连接层的情况，计算特征维上的均值和方差
            mean = X.mean(axis=0)
            var = ((X - mean)**2).mean(axis=0)
        else:
            # 使用二维卷积层的情况，计算通道维上（axis=1）的均值和方差。
            # 这里我们需要保持X的形状以便后面可以做广播运算
            mean = X.mean(axis=(0, 2, 3), keepdims=True)
            var = ((X - mean)**2).mean(axis=(0, 2, 3), keepdims=True)
        # 训练模式下，用当前的均值和方差做标准化
        X_hat = (X - mean) / np.sqrt(var + eps)
        # 更新移动平均的均值和方差
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # 缩放和移位
    return Y, moving_mean, moving_var
```

我们现在可以创建一个正确的 BatchNorm 图层。这个层将保持适当的参数：拉伸 gamma 和偏移 beta, 这两个参数将在训练过程中更新。此外，我们的图层将保存均值和方差的移动平均值，以便在模型预测期间随后使用。

撇开算法细节，注意我们实现图层的基础设计模式。通常情况下，我们用一个单独的函数定义其数学原理，比如说 batch_norm。然后，我们将此功能集成到一个自定义层中，其代码主要处理簿记问题，例如将数据移动到训练设备（如 GPU）、分配和初始化任何必需的变量、跟踪移动平均线（此处为均值和方差）等。为了方便起见，我们并不担心在这里自动推断输入形状，因此我们需要指定整个特征的数量。不用担心，深度学习框架中的批量归一化 API 将为我们解决上述问题，我们稍后将展示这一点。

```
class BatchNorm(nn.Block):
    # `num_features`：完全连接层的输出数量或卷积层的输出通道数。
    # `num_dims`：2表示完全连接层，4表示卷积层
    def __init__(self, num_features, num_dims, **kwargs):
        super().__init__(**kwargs)
        if num_dims == 2:
            shape = (1, num_features)
```

(continues on next page)

(continued from previous page)

```
        else:
            shape = (1, num_features, 1, 1)
        # 参与求梯度和迭代的拉伸和偏移参数，分别初始化成1和0
        self.gamma = self.params.get('gamma', shape=shape, init=init.One())
        self.beta = self.params.get('beta', shape=shape, init=init.Zero())
        # 非模型参数的变量初始化为0和1
        self.moving_mean = np.zeros(shape)
        self.moving_var = np.ones(shape)

    def forward(self, X):
        # 如果 `X` 不在内存上，将 `moving_mean` 和 `moving_var`
        # 复制到 `X` 所在显存上
        if self.moving_mean.ctx != X.ctx:
            self.moving_mean = self.moving_mean.copyto(X.ctx)
            self.moving_var = self.moving_var.copyto(X.ctx)
        # 保存更新过的 `moving_mean` 和 `moving_var`
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma.data(), self.beta.data(), self.moving_mean,
            self.moving_var, eps=1e-12, momentum=0.9)
        return Y
```

6.5.4 使用批量归一化层的 LeNet

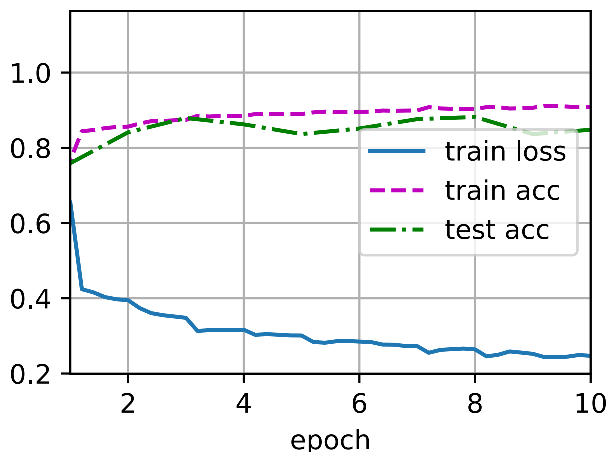
为了更好理解如何应用 `BatchNorm`，下面我们将其应用于 LeNet 模型（5.6节）。回想一下，批量归一化是在卷积层或全连接层之后、相应的激活函数之前应用的。

```
net = nn.Sequential()
net.add(nn.Conv2D(6, kernel_size=5), BatchNorm(6, num_dims=4),
        nn.Activation('sigmoid'), nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(16, kernel_size=5), BatchNorm(16, num_dims=4),
        nn.Activation('sigmoid'), nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120), BatchNorm(120, num_dims=2), nn.Activation('sigmoid'),
        nn.Dense(84), BatchNorm(84, num_dims=2), nn.Activation('sigmoid'),
        nn.Dense(10))
```

和以前一样，我们将在 Fashion-MNIST 数据集训练我们的网络。这个代码与我们第一次训练 LeNet（5.6节）时几乎完全相同，主要区别在于学习率大得多。

```
lr, num_epochs, batch_size = 1.0, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.247, train acc 0.909, test acc 0.848
16340.7 examples/sec on gpu(0)
```



让我们来看看从第一个批量归一化层中学到的拉伸参数 gamma 和偏移参数 beta。

```
net[1].gamma.data().reshape(-1,), net[1].beta.data().reshape(-1,)
```

```
(array([2.3308132, 1.2786148, 2.9026124, 1.6585207, 1.9857398, 1.0288494],␣
↪ctx=gpu(0)),
 array([ 1.4144272 ,  0.0043745 , -3.1820958 , -0.4915184 , -0.19743346,
        -0.43272173], ctx=gpu(0)))
```

6.5.5 简明实现

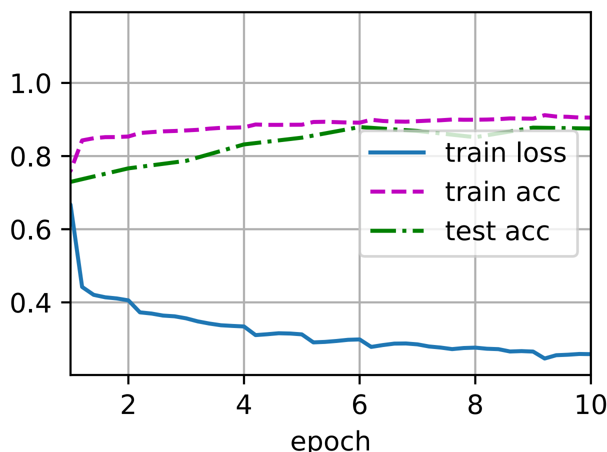
除了使用我们刚刚定义的 BatchNorm，我们也可以直接使用深度学习框架中定义的 BatchNorm。该代码看起来几乎与我们上面的代码相同。

```
net = nn.Sequential()
net.add(nn.Conv2D(6, kernel_size=5), nn.BatchNorm(), nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2), nn.Conv2D(16, kernel_size=5),
        nn.BatchNorm(), nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2), nn.Dense(120), nn.BatchNorm(),
        nn.Activation('sigmoid'), nn.Dense(84), nn.BatchNorm(),
        nn.Activation('sigmoid'), nn.Dense(10))
```

下面，我们使用相同的超参数来训练我们的模型。请注意，通常高级 API 变体运行速度快得多，因为它的代码已编译为 C++ 或 CUDA，而我们的自定义代码由 Python 实现。

```
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.258, train acc 0.905, test acc 0.875
30715.3 examples/sec on gpu(0)
```



6.5.6 争议

直观地说，批量归一化被认为可以使优化更加平滑。然而，我们必须小心区分投机直觉和对我们观察到的现象的真实解释。回想一下，我们甚至不知道为什么简单的神经网络（多层感知机和传统的卷积神经网络）为什么如此有效。即使在 dropout 和权重衰减的情况下，它们仍然非常灵活，因此无法通过传统的学习理论泛化保证来解释它们是否能够概括到看不见的数据。

在提出批量归一化的论文中，作者除了介绍了其应用，还解释了其原理：通过减少 内部协变量偏移（internal covariate shift）。据推测，作者所说的"内部协变量转移"类似于上述的投机直觉，即变量值的分布在训练过程中会发生变化。然而，这种解释有两个问题：i）这种偏移与严格定义的 协变量偏移（covariate shift）非常不同，所以这个名字用词不当。ii）这种解释只提供了一种不明确的直觉，但留下了一个有待后续挖掘的问题：为什么这项技术如此有效?。本书旨在传达实践者用来发展深层神经网络的直觉。然而，重要的是将这些指导性直觉与既定的科学事实区分开来。最终，当你掌握了这些方法，并开始撰写自己的研究论文时，你会希望清楚地区分技术和直觉。

随着批量归一化的普及，"内部协变量偏移"的解释反复出现在技术文献的辩论，特别是关于"如何展示机器学习研究"的更广泛的讨论中。Ali Rahimi 在接受 2017 年 NeurIPS 大会的"接受时间考验奖"（Test of Time Award）时发表了一篇令人难忘的演讲。他将"内部协变量转移"作为焦点，将现代深度学习的实践比作炼金术。他对该示例进行了详细回顾 [Lipton & Steinhardt, 2018] ，概述了机器学习中令人不安的趋势。此外，一些作者对批量归一化的成功提出了另一种解释：在某些方面，批量归一化的表现出与原始论文 [Santurkar et al., 2018] 中声称的行为是相反的。

然而，与技术机器学习文献中成千上万类似模糊的声明相比，内部协变量偏移没有什么更值得批评。很可能，它作为这些辩论的焦点而产生共鸣，要归功于它对目标受众的广泛认可。批量归一化已经证明是一种不可或缺的方法，适用于几乎所有图像分类器，在学术界获得了数万引用。

6.5.7 小结

- 在模型训练过程中，批量归一化利用小批量的均值和标准差，不断调整神经网络的中间输出，使整个神经网络各层的中间输出值更加稳定。
- 批量归一化在全连接层和卷积层的使用略有不同。
- 批量归一化层和 dropout 层一样，在训练模式和预测模式下计算不同。
- 批量归一化有许多有益的副作用，主要是正则化。另一方面，"减少内部协变量偏移"的原始动机似乎不是一个有效的解释。

6.5.8 练习

1. 在使用批量归一化之前，我们是否可以从全连接层或卷积层中删除偏置参数？为什么？
2. 比较LeNet在使用和不使用批量归一化情况下的学习率。
    1. 绘制训练和测试准确度的提高。
    2. 你的学习率有多高？
3. 我们是否需要在每个层中进行批量归一化？尝试一下？
4. 你可以通过批量归一化来替换 dropout 吗？行为如何改变？
5. 确定参数 `beta` 和 `gamma`，并观察和分析结果。
6. 查看高级 API 中有关 `BatchNorm` 的在线文档，以查看其他批量归一化的应用。
7. 研究思路：想想你可以应用的其他"归一化"转换？你可以应用概率积分变换吗？全秩协方差估计如何？

Discussions[89]

6.6 残差网络（ResNet）

随着我们设计越来越深的网络，深刻理解"新添加的层如何提升神经网络的性能"变得至关重要。更重要的是设计网络的能力，在这种网络中，添加层会使网络更具表现力，为了取得质的突破，我们需要一些数学基础知识。

[89] https://discuss.d2l.ai/t/1876

6.6.1 函数类

首先，假设有一类特定的神经网络结构 $\mathcal{F}$，它包括学习速率和其他超参数设置。对于所有 $f \in \mathcal{F}$，存在一些参数集（例如权重和偏置），这些参数可以通过在合适的数据集上进行训练而获得。现在假设 f^* 是我们真正想要找到的函数，如果是 $f^* \in \mathcal{F}$，那我们可以轻而易举的训练得到它，但通常我们不会那么幸运。相反，我们将尝试找到一个函数 $f^*_\mathcal{F}$，这是我们在 $\mathcal{F}$ 中的最佳选择。例如，给定一个具有 $\mathbf{X}$ 特性和 $\mathbf{y}$ 标签的数据集，我们可以尝试通过解决以下优化问题来找到它：

$$f^*_\mathcal{F} := \mathop{\mathrm{argmin}}_f L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}. \tag{6.6.1}$$

那么，怎样得到更近似真正 f^* 的函数呢？唯一合理的可能性是，我们需要设计一个更强大的结构 $\mathcal{F}'$。换句话说，我们预计 $f^*_{\mathcal{F}'}$ 比 $f^*_\mathcal{F}$ “更近似”。然而，如果 $\mathcal{F} \not\subseteq \mathcal{F}'$，则无法保证新的体系“更近似”。事实上，$f^*_{\mathcal{F}'}$ 可能更糟：如 图6.6.1 所示，对于非嵌套函数（non-nested function）类，较复杂的函数类并不总是向“真”函数 f^* 靠拢（复杂度由 $\mathcal{F}_1$ 向 $\mathcal{F}_6$ 递增）。在 图6.6.1 的左边，虽然 $\mathcal{F}_3$ 比 f^* 更接近 f^*，但$\mathcal{F}_6$ 却离的更远了。相反对于 图6.6.1 右侧的嵌套函数（nested function）类 $\mathcal{F}_1 \subseteq \ldots \subseteq \mathcal{F}_6$，我们可以避免上述问题。

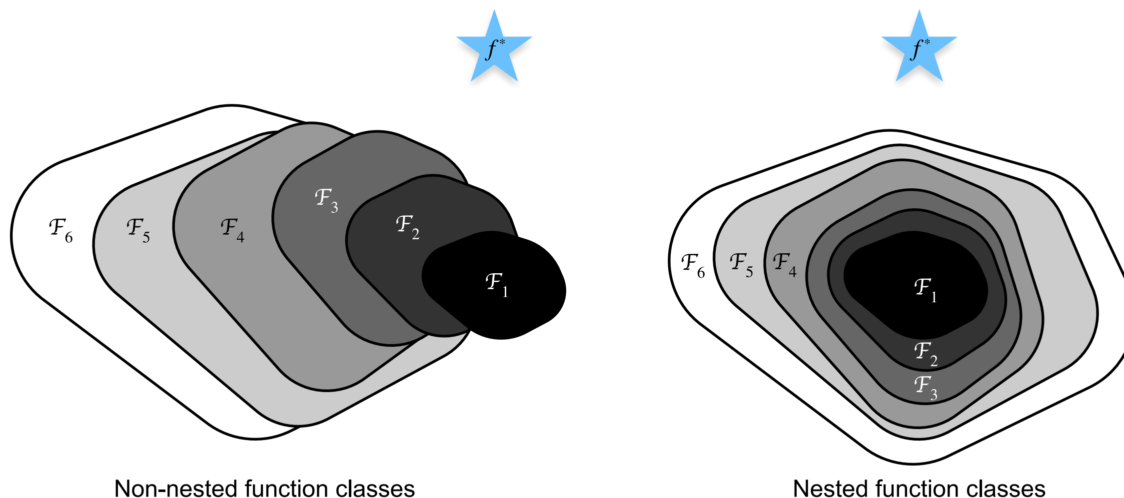


图6.6.1: 对于非嵌套函数类，较复杂（由较大区域表示）的函数类不能保证更接近“真”函数（f^* ）。这种现象在嵌套函数类中不会发生。

因此，只有当较复杂的函数类包含较小的函数类时，我们才能确保提高它们的性能。对于深度神经网络，如果我们能将新添加的层训练成 恒等映射（identity function）$f(\mathbf{x}) = \mathbf{x}$ ，新模型和原模型将同样有效。同时，由于新模型可能得出更优的解来拟合训练数据集，因此添加层似乎更容易降低训练误差。

针对这一问题，何恺明等人提出了残差网络（ResNet）[He et al., 2016a]。它在2015年的ImageNet图像识别挑战赛夺魁，并深刻影响了后来的深度神经网络的设计。残差网络的核心思想是：每个附加层都应该更容易地包含原始函数作为其元素之一。于是，残差块（residual blocks）便诞生了，这个设计对如何建立深层神经网络产生了深远的影响。凭借它，ResNet 赢得了 2015 年 ImageNet 大规模视觉识别挑战赛。。

6.6.2 残差块

让我们聚焦于神经网络局部：如图 图6.6.2 所示，假设我们的原始输入为 x，而希望学出的理想映射为 $f(\mathbf{x})$ (作为 图6.6.2 上方激活函数的输入)。图6.6.2 左图虚线框中的部分需要直接拟合出该映射 $f(\mathbf{x})$，而右图虚线框中的部分则需要拟合出残差映射 $f(\mathbf{x})-\mathbf{x}$。残差映射在现实中往往更容易优化。以本节开头提到的恒等映射作为我们希望学出的理想映射 $f(\mathbf{x})$，我们只需将 图6.6.2 中右图虚线框内上方的加权运算（如仿射）的权重和偏置参数设成 0，那么 $f(\mathbf{x})$ 即为恒等映射。实际中，当理想映射 $f(\mathbf{x})$ 极接近于恒等映射时，残差映射也易于捕捉恒等映射的细微波动。图6.6.2 右图是 ResNet 的基础结构–残差块（residual block）。在残差块中，输入可通过跨层数据线路更快地向前传播。

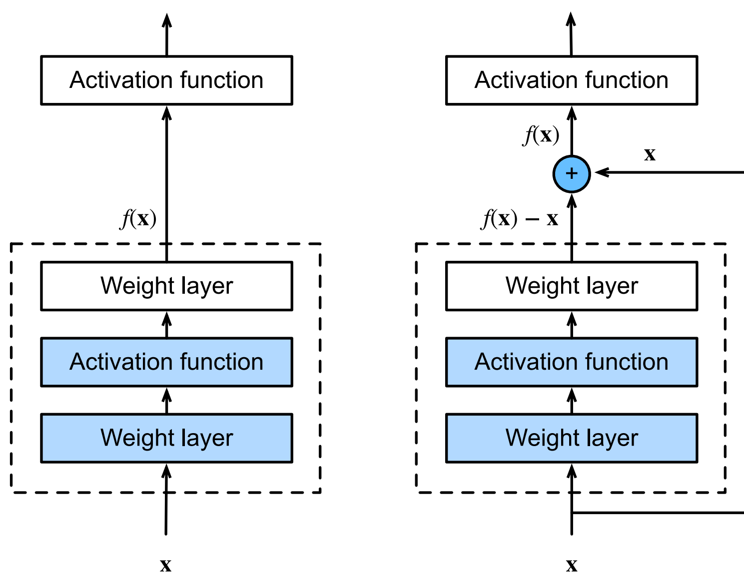


图6.6.2: 一个正常块（左图）和一个残差块（右图）。

ResNet 沿用了 VGG 完整的 3×3 卷积层设计。残差块里首先有 2 个有相同输出通道数的 3×3 卷积层。每个卷积层后接一个批量归一化层和 ReLU 激活函数。然后我们通过跨层数据通路，跳过这 2 个卷积运算，将输入直接加在最后的 ReLU 激活函数前。这样的设计要求 2 个卷积层的输出与输入形状一样，从而可以相加。如果想改变通道数，就需要引入一个额外的 1×1 卷积层来将输入变换成需要的形状后再做相加运算。残差块的实现如下：

```
from mxnet import np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()

class Residual(nn.Block):  #@save
    def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
        super().__init__(**kwargs)
```

(continues on next page)

(continued from previous page)

```
        self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                               strides=strides)
        self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                   strides=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

    def forward(self, X):
        Y = npx.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return npx.relu(Y + X)
```

如图 图6.6.3 所示，此代码生成两种类型的网络：一种是在 `use_1x1conv=False`、应用 ReLU 非线性函数之前，将输入添加到输出。另一种是在 `use_1x1conv=True` 时，添加通过 1×1 卷积调整通道和分辨率。

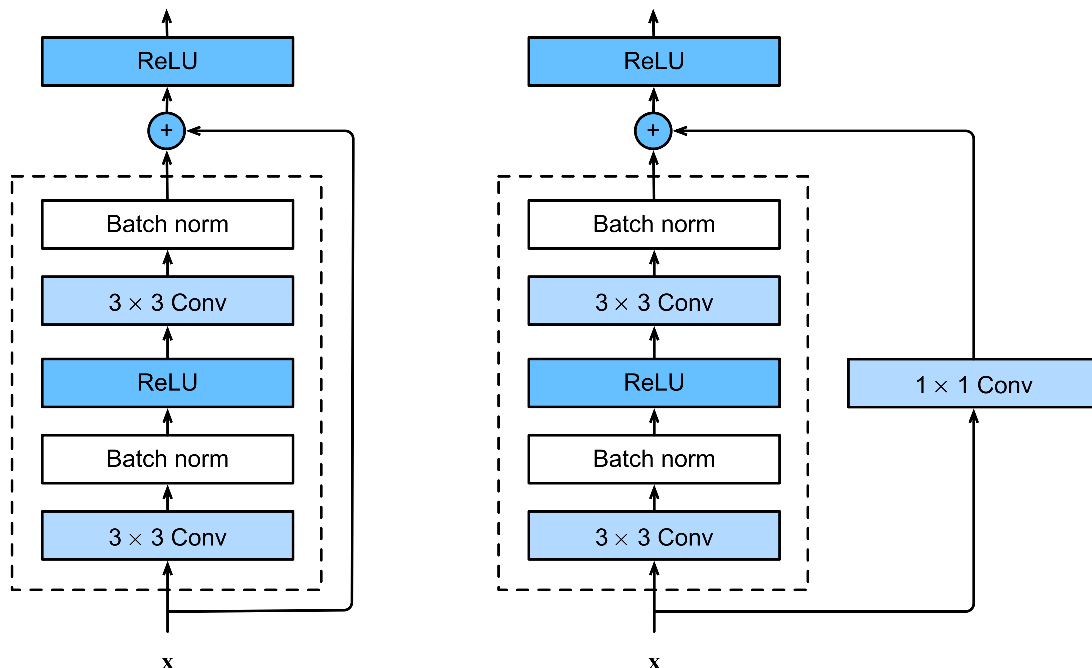


图6.6.3: 包含以及不包含 1×1 卷积层的残差块。

下面我们来查看输入和输出形状一致的情况。

```
blk = Residual(3)
blk.initialize()
X = np.random.uniform(size=(4, 3, 6, 6))
blk(X).shape
```

```
(4, 3, 6, 6)
```

我们也可以在增加输出通道数的同时减半输出的高和宽。

```
blk = Residual(6, use_1x1conv=True, strides=2)
blk.initialize()
blk(X).shape
```

```
(4, 6, 3, 3)
```

6.6.3 ResNet模型

ResNet 的前两层跟之前介绍的 GoogLeNet 中的一样：在输出通道数为 64、步幅为 2 的 7×7 卷积层后，接步幅为 2 的 3×3 的最大池化层。不同之处在于 ResNet 每个卷积层后增加了批量归一化层。

```
net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3), nn.BatchNorm(),
        nn.Activation('relu'), nn.MaxPool2D(pool_size=3, strides=2,
                                            padding=1))
```

GoogLeNet 在后面接了 4 个由Inception块组成的模块。ResNet 则使用 4 个由残差块组成的模块，每个模块使用若干个同样输出通道数的残差块。第一个模块的通道数同输入通道数一致。由于之前已经使用了步幅为 2 的最大池化层，所以无须减小高和宽。之后的每个模块在第一个残差块里将上一个模块的通道数翻倍，并将高和宽减半。

下面我们来实现这个模块。注意，我们对第一个模块做了特别处理。

```
def resnet_block(num_channels, num_residuals, first_block=False):
    blk = nn.Sequential()
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.add(Residual(num_channels))
    return blk
```

接着在 ResNet 加入所有残差块，这里每个模块使用 2 个残差块。

```
net.add(resnet_block(64, 2, first_block=True), resnet_block(128, 2),
        resnet_block(256, 2), resnet_block(512, 2))
```

最后，与 GoogLeNet 一样，在 ResNet 中加入全局平均池化层，以及全连接层输出。

```
net.add(nn.GlobalAvgPool2D(), nn.Dense(10))
```

每个模块有 4 个卷积层（不包括恒等映射的 1×1 卷积层）。加上第一个 7×7 卷积层和最后一个全连接层，共有 18 层。因此，这种模型通常被称为 ResNet-18。通过配置不同的通道数和模块里的残差块数可以得到不同的 ResNet 模型，例如更深的含 152 层的 ResNet-152。虽然 ResNet 的主体结构跟 GoogLeNet类似，但 ResNet 结构更简单，修改也更方便。这些因素都导致了 ResNet 迅速被广泛使用。图6.6.4 描述了完整的 ResNet-18。

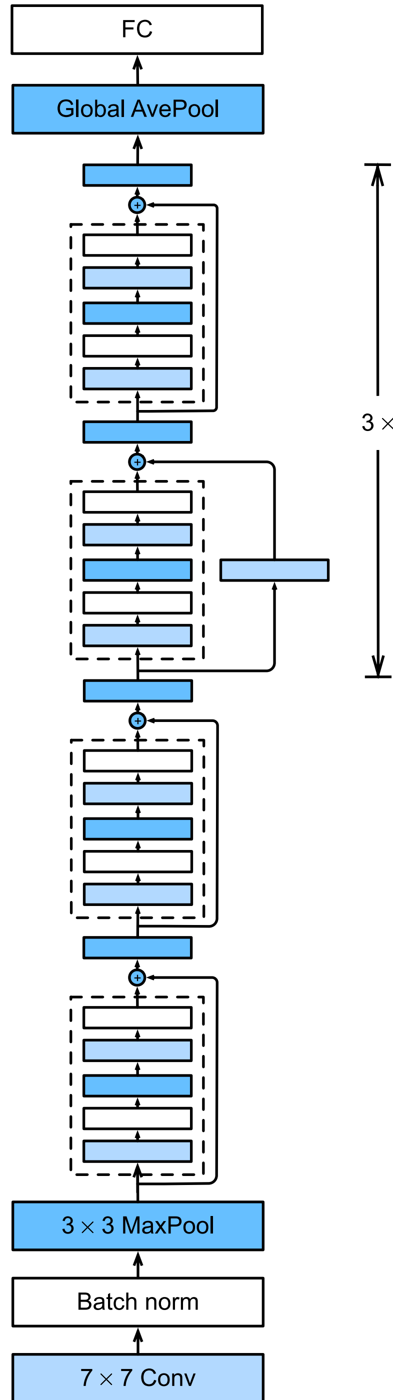


图6.6.4: ResNet-18 架构

在训练 ResNet 之前，让我们观察一下 ResNet 中不同模块的输入形状是如何变化的。在之前所有架构中，分辨率降低，通道数量增加，直到全局平均池化层聚集所有特征。

```
X = np.random.uniform(size=(1, 1, 224, 224))
```

(continues on next page)

(continued from previous page)

```
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

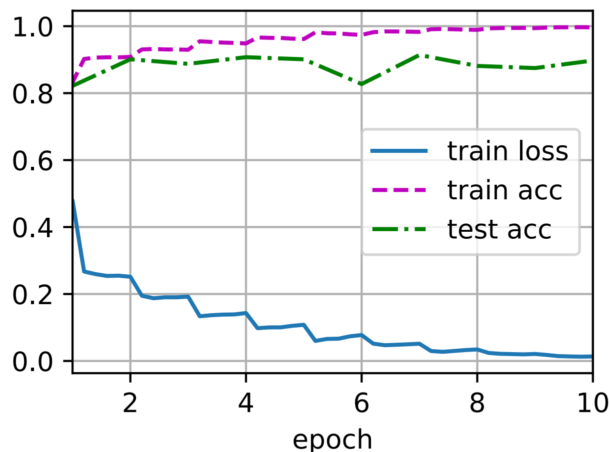
```
conv5 output shape:  (1, 64, 112, 112)
batchnorm4 output shape:     (1, 64, 112, 112)
relu0 output shape:  (1, 64, 112, 112)
pool0 output shape:  (1, 64, 56, 56)
sequential1 output shape:    (1, 64, 56, 56)
sequential2 output shape:    (1, 128, 28, 28)
sequential3 output shape:    (1, 256, 14, 14)
sequential4 output shape:    (1, 512, 7, 7)
pool1 output shape:  (1, 512, 1, 1)
dense0 output shape:         (1, 10)
```

6.6.4 训练 ResNet

同之前一样，我们在 Fashion-MNIST 数据集上训练 ResNet。

```
lr, num_epochs, batch_size = 0.05, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.014, train acc 0.997, test acc 0.897
4772.6 examples/sec on gpu(0)
```



6.6.5 小结

- 学习嵌套函数（nested function）是训练神经网络的理想情况。在深层神经网络中，学习另一层作为恒等映射（identity function）较容易（尽管这是一个极端情况）。
- 残差映射可以更容易地学习同一函数，例如将权重层中的参数近似为零。
- 利用残差块（residual blocks）可以训练出一个有效的深层神经网络：输入可以通过层间的残余连接更快地向前传播。
- 残差网络（ResNet）对随后的深层神经网络设计产生了深远影响，无论是卷积类网络还是全连接类网络。

6.6.6 练习

1. 图6.4.1 中的Inception块与残差块之间的主要区别是什么？在删除了Inception块中的一些路径之后，它们是如何相互关联的？
2. 参考 ResNet 论文 [He et al., 2016a] 中的表 1，以实现不同的变体。
3. 对于更深层次的网络，ResNet 引入了“bottleneck”架构来降低模型复杂性。请你试着去实现它。
4. 在 ResNet 的后续版本中，作者将“卷积层、批量归一化层和激活层”结构更改为“批量归一化层、激活层和卷积层”结构。请你做这个改进。详见 [He et al., 2016b] 中的图 1。
5. 为什么即使函数类是嵌套的，我们仍然要限制增加函数的复杂性呢？

Discussions[90]

6.7 稠密连接网络（DenseNet）

ResNet极大地改变了如何参数化深层网络中函数的观点。稠密连接网络 (DenseNet)［Huang et al., 2017] 在某种程度上是 ResNet 的逻辑扩展。让我们先从数学上了解一下。

6.7.1 从ResNet到DenseNet

回想一下任意函数的泰勒展开式（Taylor expansion），它把这个函数分解成越来越高阶的项。在 x 接近 0 时，

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 + \ldots. \tag{6.7.1}$$

同样，ResNet 将函数展开为

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}). \tag{6.7.2}$$

[90] https://discuss.d2l.ai/t/1879

也就是说，ResNet 将 f 分解为两部分：一个简单的线性项和一个更复杂的非线性项。那么再向前拓展一步，如果我们想将 f 拓展成超过两部分的信息呢？一种方案便是 DenseNet。

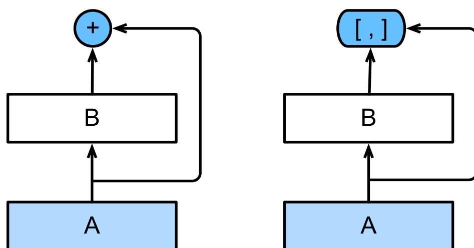


图6.7.1: ResNet（左）与 DenseNet（右）在跨层连接上的主要区别：使用相加和使用连结。

如 图6.7.1 所示，ResNet 和 DenseNet 的关键区别在于，DenseNet 输出是连接（用图中的 $[,]$ 表示）而不是如 ResNet 的简单相加。因此，在应用越来越复杂的函数序列后，我们执行从 $\mathbf{x}$ 到其展开式的映射：

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \ldots] . \tag{6.7.3}$$

最后，将这些展开式结合到多层感知机中，再次减少特征的数量。实现起来非常简单：我们不需要添加术语，而是将它们连接起来。DenseNet 这个名字由变量之间的“稠密连接”而得来，最后一层与之前的所有层紧密相连。稠密连接如 图6.7.2 所示。

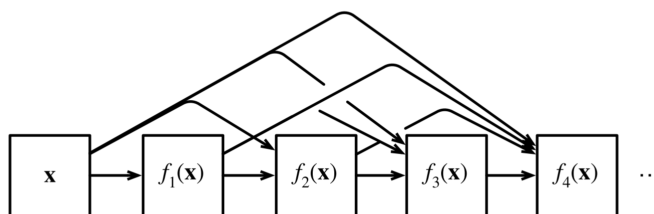


图6.7.2: 稠密连接。

稠密网络主要由 2 部分构成：稠密块（dense block）和 过渡层（transition layer）。前者定义如何连接输入和输出，而后者则控制通道数量，使其不会太复杂。

6.7.2 稠密块体

DenseNet 使用了 ResNet 改良版的“批量归一化、激活和卷积”结构（参见 6.6节 中的练习）。我们首先实现一下这个结构。

```
from mxnet import np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

(continues on next page)

(continued from previous page)

```python
def conv_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(), nn.Activation('relu'),
            nn.Conv2D(num_channels, kernel_size=3, padding=1))
    return blk
```

一个稠密块由多个卷积块组成，每个卷积块使用相同数量的输出信道。然而，在前向传播中，我们将每个卷积块的输入和输出在通道维上连结。

```python
class DenseBlock(nn.Block):
    def __init__(self, num_convs, num_channels, **kwargs):
        super().__init__(**kwargs)
        self.net = nn.Sequential()
        for _ in range(num_convs):
            self.net.add(conv_block(num_channels))

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            # 连接通道维度上每个块的输入和输出
            X = np.concatenate((X, Y), axis=1)
        return X
```

在下面的例子中，我们定义一个有 2 个输出通道数为 10 的 `DenseBlock`。使用通道数为 3 的输入时，我们会得到通道数为 $3 + 2 \times 10 = 23$ 的输出。卷积块的通道数控制了输出通道数相对于输入通道数的增长，因此也被称为增长率（growth rate）。

```python
blk = DenseBlock(2, 10)
blk.initialize()
X = np.random.uniform(size=(4, 3, 8, 8))
Y = blk(X)
Y.shape
```

```
(4, 23, 8, 8)
```

6.7.3 过渡层

由于每个稠密块都会带来通道数的增加，使用过多则会过于复杂化模型。而过渡层可以用来控制模型复杂度。它通过 1×1 卷积层来减小通道数，并使用步幅为 2 的平均池化层减半高和宽，从而进一步降低模型复杂度。

```
def transition_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(), nn.Activation('relu'),
            nn.Conv2D(num_channels, kernel_size=1),
            nn.AvgPool2D(pool_size=2, strides=2))
    return blk
```

对上一个例子中稠密块的输出使用通道数为 10 的过渡层。此时输出的通道数减为 10，高和宽均减半。

```
blk = transition_block(10)
blk.initialize()
blk(Y).shape
```

```
(4, 10, 4, 4)
```

6.7.4 DenseNet模型

我们来构造 DenseNet 模型。DenseNet 首先使用同 ResNet 一样的单卷积层和最大池化层。

```
net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3), nn.BatchNorm(),
        nn.Activation('relu'), nn.MaxPool2D(pool_size=3, strides=2,
                                            padding=1))
```

接下来，类似于 ResNet 使用的 4 个残差块，DenseNet 使用的是 4 个稠密块。与 ResNet 类似，我们可以设置每个稠密块使用多少个卷积层。这里我们设成 4，从而与 6.6节 的 ResNet-18 保持一致。稠密块里的卷积层通道数（即增长率）设为 32，所以每个稠密块将增加 128 个通道。

在每个模块之间，ResNet 通过步幅为 2 的残差块减小高和宽，DenseNet 则使用过渡层来减半高和宽，并减半通道数。

```
# `num_channels`为当前的通道数
num_channels, growth_rate = 64, 32
num_convs_in_dense_blocks = [4, 4, 4, 4]

for i, num_convs in enumerate(num_convs_in_dense_blocks):
    net.add(DenseBlock(num_convs, growth_rate))
    # 上一个稠密块的输出通道数
    num_channels += num_convs * growth_rate
```

(continues on next page)

(continued from previous page)

```
    # 在稠密块之间添加一个转换层，使通道数量减半
    if i != len(num_convs_in_dense_blocks) - 1:
        num_channels //= 2
        net.add(transition_block(num_channels))
```

与 ResNet 类似，最后接上全局池化层和全连接层来输出结果。

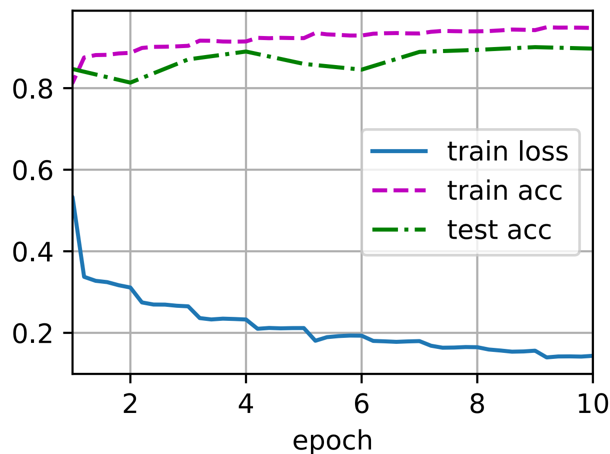
```
net.add(nn.BatchNorm(), nn.Activation('relu'), nn.GlobalAvgPool2D(),
        nn.Dense(10))
```

6.7.5 训练模型

由于这里使用了比较深的网络，本节里我们将输入高和宽从 224 降到 96 来简化计算。

```
lr, num_epochs, batch_size = 0.1, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.144, train acc 0.948, test acc 0.897
5127.8 examples/sec on gpu(0)
```



6.7.6 小结

- 在跨层连接上，不同于 ResNet 中将输入与输出相加，稠密连接网络（DenseNet）在通道维上连结输入与输出。
- DenseNet 的主要构建模块是稠密块和过渡层。
- 在构建 DenseNet 时，我们需要通过添加过渡层来控制网络的维数，从而再次减少信道的数量。

6.7.7 练习

1. 为什么我们在过渡层使用平均池化层而不是最大池化层？
2. DenseNet 的优点之一是其模型参数比 ResNet 小。为什么呢？
3. DenseNet 一个诟病的问题是内存或显存消耗过多。
    1. 真的是这样吗？可以把输入形状换成 224×224，来看看实际的显存消耗。
    2. 你能想出另一种方法来减少显存消耗吗？你需要如何改变框架？
4. 实现 DenseNet 论文 [Huang et al., 2017] 表 1 所示的不同 DenseNet 版本。
5. 应用DenseNet的思想设计一个基于多层感知机的模型。将其应用于 3.10节 中的房价预测任务。

Discussions[91]

[91] https://discuss.d2l.ai/t/1882

7

循环神经网络

到目前为止，我们遇到了两种类型的数据：表格数据和图像数据。对于后者，我们设计了专门的层来利用其中的规律。换句话说，如果我们对图像中的像素进行调换，就很难对其内容进行推理。这些内容看起来很像模拟电视时代的雪花屏。

最重要的是，到目前为止，我们默认我们的数据都来自某种分布，并且所有样本都是独立同分布的（i.i.d.）。不幸的是，大多数的数据并非如此。例如，文章中的单词是按顺序写的，如果打乱它们的顺序，就很难理解它们组成的意思。同样，视频中的图像帧、对话的音频信号以及网站上的浏览行为都是有顺序的。因此，我们可以合理地假设，针对这类数据的专门模型会更好地描述它们。

有时我们希望不仅可以接收一个序列作为输入，而是可以期望继续猜测该序列。例如，任务可以是继续预测$2, 4, 6, 8, 10, \ldots$。这在时间序列分析中是相当常见的，可以用来预测股市、患者的体温曲线或赛车所需的加速度。同样，我们需要能够处理这些数据的模型。

简言之，卷积神经网络可以有效地处理空间信息，循环神经网络（RNN）的设计可以更好地处理序列信息。循环神经网络引入状态变量来存储过去的信息以及当前的输入，以确定当前的输出。

许多使用循环网络的例子都是基于文本数据的。因此，我们将在本章中重点介绍语言模型。在对序列数据进行更正式的回顾之后，我们将介绍文本预处理的实用技术。接下来，我们将讨论语言模型的基本概念，并将此讨论作为循环神经网络设计的灵感。最后，我们描述了循环神经网络的梯度计算方法，以探讨训练此类网络时可能遇到的问题。

7.1 序列模型

想象一下你正在看Netflix（一个国外的视频网站）上的电影。作为一个优秀的Netflix用户，你决定对每一部电影都给出评价。毕竟，一部好电影就是一部好电影，你想看更多的好电影，对吗？事实证明，事情并不是那么简单。随着时间的推移，人们对电影的看法会发生很大的变化。事实上，心理学家甚至对某些效应有了命名：

- 根据别人的意见，有"锚定"。例如，奥斯卡颁奖后，相应电影的评分上升，尽管它仍然是同一部电影。这种影响持续几个月，直到奖项被遗忘。结果表明，这种效应使评分提高了半个百分点以上 [Wu et al., 2017].
- 有一种"享乐适应"，即人类迅速适应，接受一种改善或恶化的情况作为新的常态。例如，在看了很多好电影之后，人们对下一部电影同样好或更好的期望很高。因此，即使是一部普通的电影，在看过许多精彩的电影之后，也可能被认为是糟糕的。
- 有季节性。很少有观众喜欢在八月看圣诞老人的电影。
- 在某些情况下，由于导演或演员在制作中的不当行为，电影变得不受欢迎。
- 有些电影在小圈子内被支持者喜爱及推崇，这是因为它们几乎滑稽可笑。

简而言之，电影评分决不是固定不变的。因此，使用时间动力学可以得到更准确的电影推荐 [Koren, 2009] 。当然，序列数据不仅仅是关于电影评分的。下面给出了更多的场景。

- 许多用户在打开应用程序时都有非常特殊的行为。例如，社交媒体应用在学生放学后更受欢迎。股市交易应用程序在市场开放时更常用。
- 要预测明天的股价要比填补我们昨天错过股价的空白困难得多，尽管两者都只是估计一个数字。毕竟，先见之明比事后诸葛亮难得多。在统计学中，前者（超出已知观测值的预测）称为外推（extrapolation），而后者（在现有观测值之间进行估计）称为内插（interpolation）。
- 音乐、语音、文本和视频在本质上都是连续的。如果我们对它们进行置换，它们就没什么意义了。文本标题"狗咬人"远没有"人咬狗"那么令人惊讶，尽管两句话词的组成完全相同。
- 地震具有很强的相关性，即大地震发生后，很可能会有几次较小的余震，比没有强震的余震要大得多。事实上，地震是时空相关的，也就是说，余震通常发生在很短的时间跨度和很近的距离内。
- 人类之间的互动是连续的，这可以从推特上的争吵和辩论中看出。

7.1.1 统计工具

我们需要统计工具和新的深层神经网络结构来处理序列数据。为了简单起见，我们以 图7.1.1 所示的股票价格（富时100指数）为例。

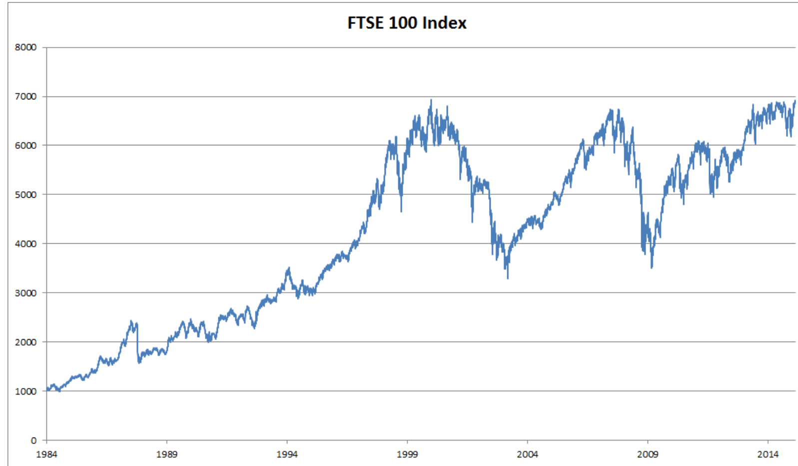


图7.1.1: 近30年的富时100指数。

让我们用x_t表示价格。即在时间步（time step）$t \in \mathbb{Z}^+$时，我们观察到的价格x_t。请注意，对于本文中的序列，t通常是离散的，并随整数或其子集而变化。假设一个想在t日股市表现良好的交易员通过以下途径预测了x_t：

$$x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1). \tag{7.1.1}$$

自回归模型

为了实现这一点，我们的交易员可以使用回归模型，比如我们在 2.3节 中训练的模型。只有一个主要问题：输入$x_{t-1}, \ldots, x_1$的数量因t而异。也就是说，这个数字随着我们遇到的数据量的增加而增加，我们需要一个近似值来使这个计算变得容易处理。本章后面的大部分内容将围绕如何有效估计$P(x_t \mid x_{t-1}, \ldots, x_1)$展开。简单地说，它归结为以下两种策略。

首先，假设相当长的序列$x_{t-1}, \ldots, x_1$实际上不是必需的。在这种情况下，我们可能会满足于长度为τ的一些时间跨度，并且只使用$x_{t-1}, \ldots, x_{t-\tau}$个观测。直接的好处是，现在参数的数量总是相同的，至少对于$t > \tau$。这使我们能够训练一个深层网络，如上所述。这种模型将被称为“自回归模型”（autoregressive models），因为它们实际上是在自己身上执行回归。

第二种策略，如 图7.1.2 所示，是保留一些过去观测的总结h_t，同时除了预测h_t之外还更新$\hat{x}_t$。这就产生了估计x_t和$\hat{x}_t = P(x_t \mid h_t)$的模型，并且更新了$h_t = g(h_{t-1}, x_{t-1})$。由于$h_t$从未被观测到，这类模型也被称为隐变量自回归模型（latent autoregressive models）。

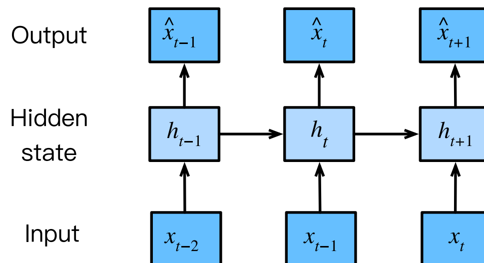


图7.1.2: 潜在自回归模型

这两种情况都有一个显而易见的问题，即如何生成训练数据。一个经典方法是使用历史观测来预测下一次的观测。显然，我们并不指望时间会停滞不前。然而，一个常见的假设是，虽然特定值x_t可能会改变，但至少序列本身的动力学不会改变。统计学家称不变的动力学为“静止的”。因此，无论我们做什么，我们都将通过以下方式获得整个序列的估计值

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}, \ldots, x_1). \tag{7.1.2}$$

注意，如果我们处理离散的对象(如单词)，而不是连续的数字，则上述考虑因素仍然有效。唯一的区别是，在这种情况下，我们需要使用分类器而不是回归模型来估计$P(x_t \mid x_{t-1}, \ldots, x_1)$。

马尔可夫模型

回想一下，在自回归模型中，我们只使用$x_{t-1}, \ldots, x_{t-\tau}$而不是$x_{t-1}, \ldots, x_1$来估计$x_t$。只要这种近似是准确的，我们就说序列满足马尔可夫条件（Markov condition）。特别地，如果$\tau = 1$，我们有一个一阶马尔可夫模型（first-order Markov model），$P(x)$由下式给出：

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}) \text{ where } P(x_1 \mid x_0) = P(x_1). \tag{7.1.3}$$

当x_t只假设离散值时，这样的模型特别好，因为在这种情况下，动态规划可以用来沿着链精确地计算值。例如，我们可以高效地计算$P(x_{t+1} \mid x_{t-1})$：

$$\begin{aligned}
P(x_{t+1} \mid x_{t-1}) &= \frac{\sum_{x_t} P(x_{t+1}, x_t, x_{t-1})}{P(x_{t-1})} \\
&= \frac{\sum_{x_t} P(x_{t+1} \mid x_t, x_{t-1}) P(x_t, x_{t-1})}{P(x_{t-1})} \\
&= \sum_{x_t} P(x_{t+1} \mid x_t) P(x_t \mid x_{t-1})
\end{aligned} \tag{7.1.4}$$

利用这一事实，我们只需要考虑到过去观察到的非常短的历史：$P(x_{t+1} \mid x_t, x_{t-1}) = P(x_{t+1} \mid x_t)$。详细介绍动态规划超出了本节的范围。控制和强化学习算法广泛使用这些工具。

因果关系

原则上，倒序展开$P(x_1, \ldots, x_T)$无可厚非。毕竟，通过条件作用，我们总是可以写出：

$$P(x_1, \ldots, x_T) = \prod_{t=T}^{1} P(x_t \mid x_{t+1}, \ldots, x_T). \tag{7.1.5}$$

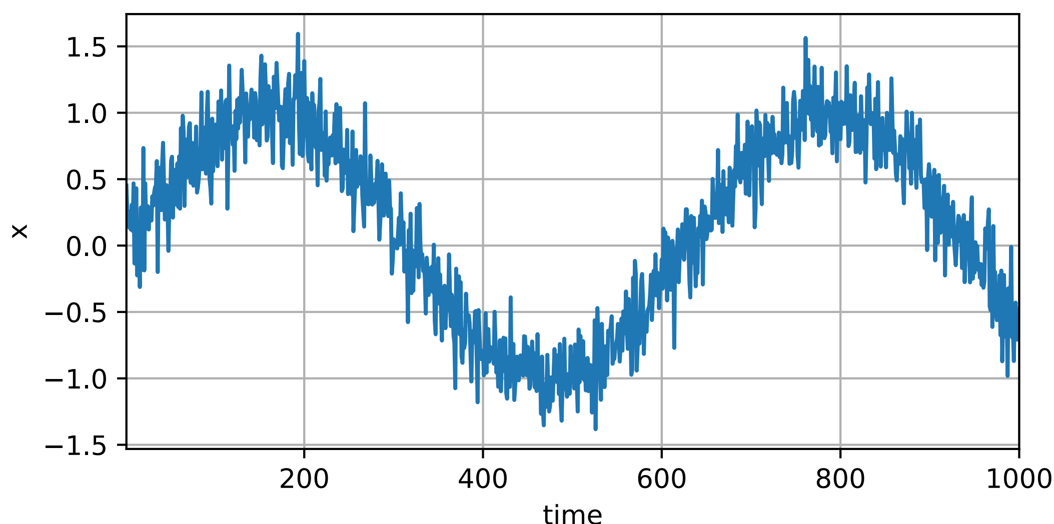
事实上，如果我们有一个马尔可夫模型，我们也可以得到一个反向条件概率分布。然而，在许多情况下，数据存在一个自然的方向，即在时间上前进。很明显，未来的事件不能影响过去。因此，如果我们改变x_t，我们可能能够影响x_{t+1}未来发生的事情，但不能影响相反的情况。也就是说，如果我们改变x_t，过去事件的分布不会改变。因此，解释$P(x_{t+1} \mid x_t)$应该比解释$P(x_t \mid x_{t+1})$更容易。例如，已经表明，在某些情况下，对于某些加性噪声ϵ，我们可以找到$x_{t+1} = f(x_t) + \epsilon$，而反之则不是真的 [Hoyer et al., 2009] 。这是个好消息，因为这通常是我们有兴趣估计的前进方向。彼得斯等人写的这本书。已经解释了关于这个主题的更多内容 [Peters et al., 2017] 。我们仅仅触及了它的皮毛。

7.1.2 训练

在回顾了这么多统计工具之后，让我们在实践中尝试一下。我们首先生成一些数据。为了简单起见，我们使用正弦函数和一些加性噪声来生成序列数据，时间步为$1, 2, \ldots, 1000$。

```
%matplotlib inline
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

```
T = 1000  # 总共产生1000个点
time = np.arange(1, T + 1, dtype=np.float32)
x = np.sin(0.01 * time) + np.random.normal(0, 0.2, (T,))
d2l.plot(time, [x], 'time', 'x', xlim=[1, 1000], figsize=(6, 3))
```



接下来，我们需要将这样的序列转换为我们的模型可以训练的特征和标签。基于嵌入维度τ，我们将数据映射为$y_t = x_t$和$\mathbf{x}_t = [x_{t-\tau}, \ldots, x_{t-1}]$。精明的读者可能已经注意到，这给我们提供的数据样本少了τ个，因为我们没有足够的历史记录来记录前τ个数据样本。一个简单的解决办法，特别是如果序列很长，就是丢弃这几项。或者，我们可以用零填充序列。在这里，我们仅使用前600个特征-标签对进行训练。

```
tau = 4
features = np.zeros((T - tau, tau))
for i in range(tau):
    features[:, i] = x[i:T - tau + i]
labels = x[tau:].reshape((-1, 1))
```

```python
batch_size, n_train = 16, 600
# 只有前`n_train`个样本用于训练
train_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                            batch_size, is_train=True)
```

这里的结构相当简单：只有一个多层感知机，有两个全连接层、ReLU激活函数和平方损失。

```python
# A simple MLP
def get_net():
    net = nn.Sequential()
    net.add(nn.Dense(10, activation='relu'), nn.Dense(1))
    net.initialize(init.Xavier())
    return net

# Square loss
loss = gluon.loss.L2Loss()
```

现在我们准备好训练模型了。下面的代码与前面几节中的训练代码实现基本相同，如 2.3节 。因此，我们不会深入探讨太多细节。

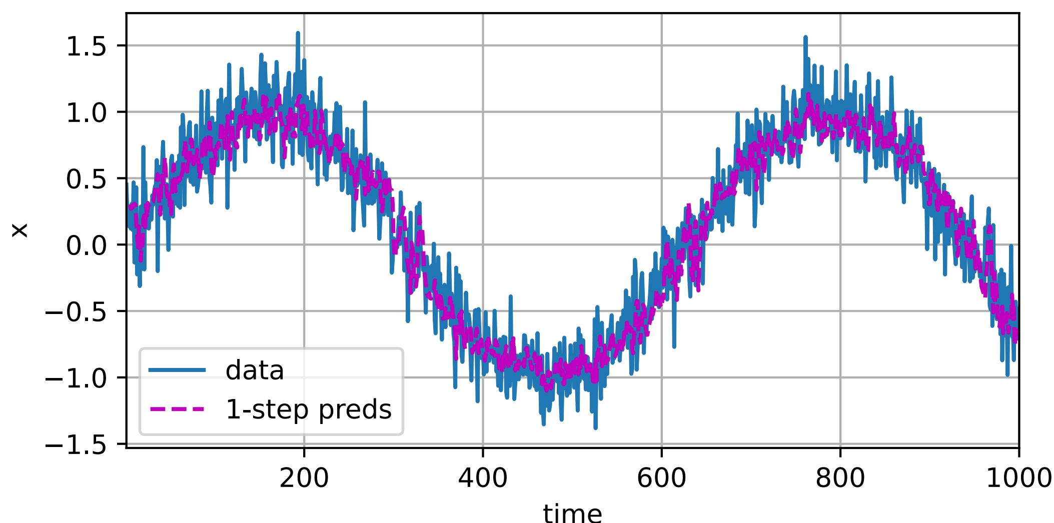
```python
def train(net, train_iter, loss, epochs, lr):
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    for epoch in range(epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        print(f'epoch {epoch + 1}, '
              f'loss: {d2l.evaluate_loss(net, train_iter, loss):f}')

net = get_net()
train(net, train_iter, loss, 5, 0.01)
```

```
epoch 1, loss: 0.043874
epoch 2, loss: 0.030642
epoch 3, loss: 0.028776
epoch 4, loss: 0.028393
epoch 5, loss: 0.026695
```

7.1.3 预测

由于训练损失很小，我们希望我们的模型能够很好地工作。让我们看看这在实践中意味着什么。首先要检查的是模型预测下一时间步发生事情的能力有多好，也就是“单步预测”（one-step-ahead prediction）。

```
onestep_preds = net(features)
d2l.plot([time, time[tau:]],
         [x.asnumpy(), onestep_preds.asnumpy()], 'time', 'x',
         legend=['data', '1-step preds'], xlim=[1, 1000], figsize=(6, 3))
```



单步预测看起来不错，正如我们所料。即使超过了604(`n_train + tau`)的观测，这些预测看起来仍然是可信的。然而，这有一个小问题：如果我们只观察序列数据到时间步604，我们不能期望接收到所有未来提前一步预测的输入。相反，我们需要一步一步向前迈进：

$$
\begin{aligned}
\hat{x}_{605} &= f(x_{601}, x_{602}, x_{603}, x_{604}), \\
\hat{x}_{606} &= f(x_{602}, x_{603}, x_{604}, \hat{x}_{605}), \\
\hat{x}_{607} &= f(x_{603}, x_{604}, \hat{x}_{605}, \hat{x}_{606}), \\
\hat{x}_{608} &= f(x_{604}, \hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}), \\
\hat{x}_{609} &= f(\hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}, \hat{x}_{608}), \\
&\ldots
\end{aligned}
\tag{7.1.6}
$$

通常，对于直到x_t的观测序列，其在时间步长$\hat{x}_{t+k}$处的预测输出$t+k$被称为“k步预测”。由于我们已经观察到了x_{604}，它领先k步的预测是$\hat{x}_{604+k}$。换句话说，我们将不得不使用我们自己的预测来进行多步预测。让我们看看这件事进行得有多顺利。

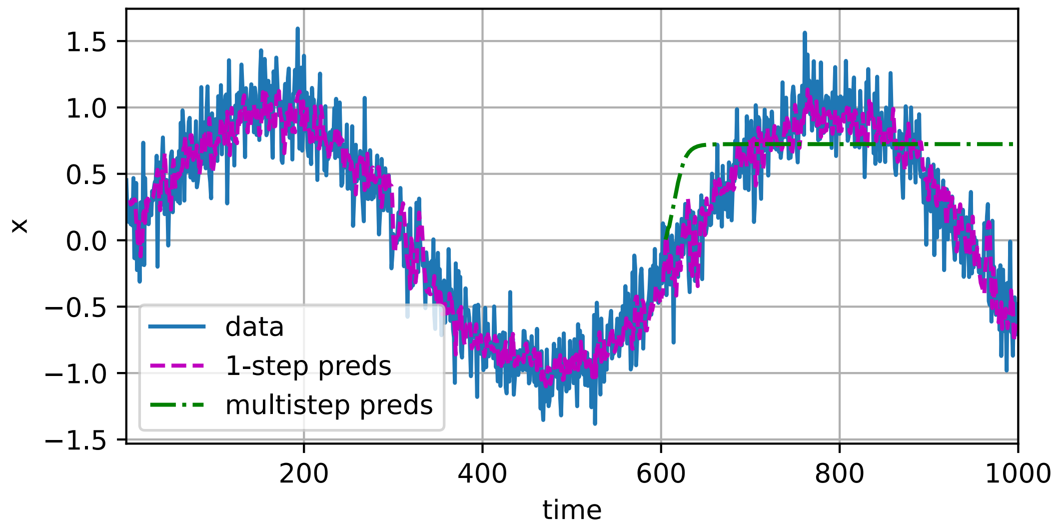
```
multistep_preds = np.zeros(T)
multistep_preds[:n_train + tau] = x[:n_train + tau]
```

(continues on next page)

(continued from previous page)

```
for i in range(n_train + tau, T):
    multistep_preds[i] = net(multistep_preds[i - tau:i].reshape((1, -1)))
```

```
d2l.plot([time, time[tau:], time[n_train + tau:]], [
    x.asnumpy(),
    onestep_preds.asnumpy(), multistep_preds[n_train + tau:].asnumpy()],
        'time', 'x', legend=['data', '1-step preds', 'multistep preds'],
        xlim=[1, 1000], figsize=(6, 3))
```



正如上面的例子所示，这是一个惊人的失败。在几个预测步骤之后，预测很快就会衰减到一个常数。为什么这个算法效果这么差呢？这最终是由于错误累积的事实。假设在步骤1之后，我们有一些错误$\epsilon_1 = \bar\epsilon$。现在，步骤2的*INPUT*被扰动了$\epsilon_1$，因此对于某个常数$\epsilon_2 = \bar\epsilon + c\epsilon_1$，我们会遇到一些大约$c$的误差，依此类推。误差可能会相当快地偏离真实的观测结果。这是一个普遍的现象。例如，未来24小时的天气预报往往相当准确，但超过这一点，准确率会迅速下降。我们将在本章及以后讨论改进这一点的方法。

让我们通过计算$k = 1, 4, 16, 64$的整个序列的预测来更仔细地看一下k步预测的困难。

```
max_steps = 64
```

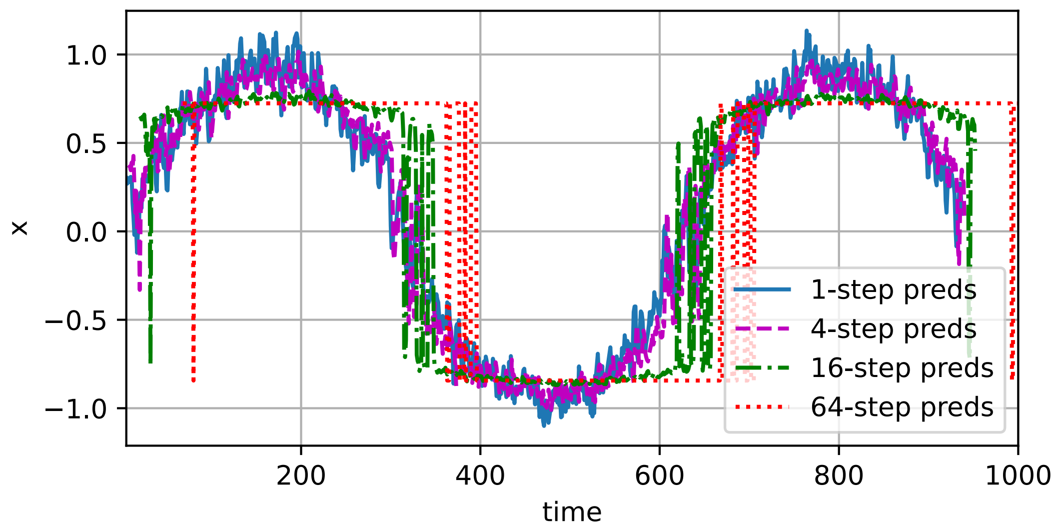
```
features = np.zeros((T - tau - max_steps + 1, tau + max_steps))
# Column `i` (`i` < `tau`) are observations from `x` for time steps from
# `i + 1` to `i + T - tau - max_steps + 1`
for i in range(tau):
    features[:, i] = x[i:i + T - tau - max_steps + 1]

# Column `i` (`i` >= `tau`) are the (`i - tau + 1`)-step-ahead predictions for
# time steps from `i + 1` to `i + T - tau - max_steps + 1`
```

(continues on next page)

(continued from previous page)

```
for i in range(tau, tau + max_steps):
    features[:, i] = net(features[:, i - tau:i]).reshape(-1)
```

```
steps = (1, 4, 16, 64)
d2l.plot([time[tau + i - 1:T - max_steps + i] for i in steps],
         [features[:, (tau + i - 1)].asnumpy() for i in steps], 'time', 'x',
         legend=[f'{i}-step preds'
                 for i in steps], xlim=[5, 1000], figsize=(6, 3))
```



这清楚地说明了当我们试图进一步预测未来时，预测的质量是如何变化的。虽然4步预测看起来仍然不错，但超过这一点的任何预测几乎都是无用的。

7.1.4 小结

- 内插和外推在难度上有很大差别。因此，如果你有一个序列，在训练时始终要尊重数据的时间顺序，即永远不要对未来的数据进行训练。
- 序列模型需要专门的统计工具进行估计。两种流行的选择是自回归模型和隐变量自回归模型。
- 对于因果模型（例如，向前推进的时间），估计正向通常比反向容易得多。
- 对于直到时间步t的观测序列，其在时间步$t+k$的预测输出是“k步预测”。随着我们在时间上进一步预测，增加k，误差会累积，预测的质量会下降。

7.1.5 练习

1. 在本部分的实验中对模型进行改进。
    1. 是否包含过去4个以上观测结果？你真的需要多少？
    2. 如果没有噪音，你需要多少过去的观察？提示：你可以把sin和cos写成微分方程。
    3. 你能在保持特征总数不变的情况下合并旧的观察结果吗？这能提高精确度吗？为什么？
    4. 改变神经网络结构并评估其性能。
2. 一位投资者想要找到一种好的证券来购买。他查看过去的回报，以决定哪一种可能表现良好。这一策略可能会出什么问题呢？
3. 因果关系也适用于文本吗？在多大程度上？
4. 举例说明什么时候可能需要隐变量自回归模型来捕捉数据的动力学模型。

Discussions[92]

7.2 文本预处理

我们回顾和评估了序列数据的统计工具和预测挑战。这些数据可以有多种形式。具体来说，正如我们将在本书的许多章节中重点介绍的那样，文本是序列数据最常见例子。例如，一篇文章可以简单地看作是一个单词序列，甚至是一个字符序列。为了方便我们将来对序列数据的实验，我们将在本节中专门解释文本的常见预处理步骤。通常，这些步骤包括：

1. 将文本作为字符串加载到内存中。
2. 将字符串拆分为标记（如，单词和字符）。
3. 建立一个词汇表，将拆分的标记映射到数字索引。
4. 将文本转换为数字索引序列，以便模型可以轻松地对其进行操作。

```
import collections
import re
from d2l import mxnet as d2l
```

[92] https://discuss.d2l.ai/t/2090

7.2.1 读取数据集

为了开始，我们从H.G.Well的时光机器[93]中加载文本。这是一个相当小的语料库，只有30000多个单词，但对于我们想要说明的目标来说，这足够了。现实中的文档集合可能会包含数十亿个单词。下面的函数将数据集读取到文本行组成的列表中，其中每行都是一个字符串。为简单起见，这里我们忽略标点符号和大写。

```
#@save
d2l.DATA_HUB['time_machine'] = (d2l.DATA_URL + 'timemachine.txt',
                                '090b5e7e70c295757f55df93cb0a180b9691891a')

def read_time_machine():  #@save
    """Load the time machine dataset into a list of text lines."""
    with open(d2l.download('time_machine'), 'r') as f:
        lines = f.readlines()
    return [re.sub('[^A-Za-z]+', ' ', line).strip().lower() for line in lines]

lines = read_time_machine()
print(f'# text lines: {len(lines)}')
print(lines[0])
print(lines[10])
```

```
# text lines: 3221
the time machine by h g wells
twinkled and his usually pale face was flushed and animated the
```

7.2.2 标记化

以下 tokenize 函数将列表作为输入，列表中的每个元素是文本序列（如，文本行）。每个文本序列被拆分成一个标记列表。标记（token）是文本的基本单位。最后返回一个标记列表，其中每个标记都是一个字符串（string）。

```
def tokenize(lines, token='word'):  #@save
    """将文本行拆分为单词或字符标记。"""
    if token == 'word':
        return [line.split() for line in lines]
    elif token == 'char':
        return [list(line) for line in lines]
    else:
        print('错误：未知令牌类型：' + token)

tokens = tokenize(lines)
```

(continues on next page)

[93] http://www.gutenberg.org/ebooks/35

(continued from previous page)

```
for i in range(11):
    print(tokens[i])
```

```
['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
[]
[]
[]
[]
['i']
[]
[]
['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be', 'convenient', 'to',
↪'speak', 'of', 'him']
['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his', 'grey', 'eyes',
↪'shone', 'and']
['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and',
↪'animated', 'the']
```

7.2.3 词汇

标记的字符串类型不方便模型使用，因为模型需要输入数字。现在，让我们构建一个字典，通常也叫做词表（Vocabulary）来将字符串标记映射到从0开始的数字索引中。为此，我们首先统计训练集中所有文档中的唯一标记，即语料（corpus），然后根据每个唯一标记的出现频率为其分配一个数字索引。很少出现的标记通常被移除，这可以降低复杂性。语料库中不存在或已删除的任何标记都将映射到一个特殊的未知标记“<unk>”。我们可以选择添加保留令牌的列表，例如“<pad>”表示填充；“<bos>”表示序列的开始；“<eos>”表示序列的结束。

```
class Vocab:  #@save
    """文本词表"""
    def __init__(self, tokens=None, min_freq=0, reserved_tokens=None):
        if tokens is None:
            tokens = []
        if reserved_tokens is None:
            reserved_tokens = []
        # 按出现频率排序
        counter = count_corpus(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                  reverse=True)
        # 未知标记的索引为0
        self.unk, uniq_tokens = 0, ['<unk>'] + reserved_tokens
        uniq_tokens += [
            token for token, freq in self.token_freqs
```

(continues on next page)

(continued from previous page)

```
            if freq >= min_freq and token not in uniq_tokens]
        self.idx_to_token, self.token_to_idx = [], dict()
        for token in uniq_tokens:
            self.idx_to_token.append(token)
            self.token_to_idx[token] = len(self.idx_to_token) - 1

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):
        if not isinstance(indices, (list, tuple)):
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]

def count_corpus(tokens):  #@save
    """Count token frequencies."""
    # 这里的 `tokens` 是1D列表或2D列表
    if len(tokens) == 0 or isinstance(tokens[0], list):
        # 将令牌列表展平
        tokens = [token for line in tokens for token in line]
    return collections.Counter(tokens)
```

我们使用时光机器数据集作为语料库来构建词汇表。然后，我们打印前几个常见标记及其索引。

```
vocab = Vocab(tokens)
print(list(vocab.token_to_idx.items())[:10])
```

```
[('<unk>', 0), ('the', 1), ('i', 2), ('and', 3), ('of', 4), ('a', 5), ('to', 6), ('was
↪', 7), ('in', 8), ('that', 9)]
```

现在我们可以将每一行文本转换成一个数字索引列表。

```
for i in [0, 10]:
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])
```

```
words: ['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
indices: [1, 19, 50, 40, 2183, 2184, 400]
```

(continues on next page)

(continued from previous page)

```
words: ['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and',
↪'animated', 'the']
indices: [2186, 3, 25, 1044, 362, 113, 7, 1421, 3, 1045, 1]
```

7.2.4 把所有的东西放在一起

使用上述函数，我们将所有内容打包到 load_corpus_time_machine 函数中，该函数返回 corpus（标记索引列表）和 vocab（时光机器语料库的词汇表）。我们在这里所做的修改是：-1、我们将文本标记化为字符，而不是单词，以简化后面部分中的训练；-2、corpus是单个列表，而不是标记列表嵌套，因为时光机器数据集中的每个文本行不一定是句子或段落。

```python
def load_corpus_time_machine(max_tokens=-1):  #@save
    """返回时光机器数据集的令牌索引和词汇表。"""
    lines = read_time_machine()
    tokens = tokenize(lines, 'char')
    vocab = Vocab(tokens)
    # 因为时光机器数据集中的每一个文本行不一定是一个句子或段落，
    # 所以将所有文本行展平到一个列表中
    corpus = [vocab[token] for line in tokens for token in line]
    if max_tokens > 0:
        corpus = corpus[:max_tokens]
    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)
```

```
(170580, 28)
```

7.2.5 小结

- 文本是序列数据的一种重要形式。
- 为了对文本进行预处理，我们通常将文本拆分为标记，构建词汇表将标记字符串映射为数字索引，并将文本数据转换为标记索引以供模型操作。

7.2.6 练习

1. 标记化是一个关键的预处理步骤。它因语言而异。尝试找到另外三种常用的文本标记方法。
2. 在本节的实验中，将文本标记为单词，并更改 `Vocab` 实例的 `min_freq` 参数。这对词汇量有何影响？

Discussions[94]

7.3 语言模型和数据集

在 7.2节 中，我们了解了如何将文本数据映射到标记中，其中这些标记可以被视为一系列离散的观测，例如单词或字符。假设长度为T的文本序列中的标记依次为$x_1, x_2, \ldots, x_T$。然后，在文本序列中，$x_t(1 \leq t \leq T)$可以被认为是时间步t处的观测或标签。给定这样的文本序列，语言模型（language model）的目标是估计序列的联合概率

$$P(x_1, x_2, \ldots, x_T). \tag{7.3.1}$$

语言模型非常有用。例如，一个理想的语言模型能够自己生成自然文本，只需一次给出一个标记$x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1)$。与猴子使用打字机非常不同的是，从这样的模型中出现的所有文本都将作为自然语言来传递，例如英语文本。此外，只需将文本限制在前面的对话片断上，就足以生成一个有意义的对话。显然，我们离设计这样的系统还很远，因为它需要"理解"文本，而不仅是生成在语法上合理的内容。

尽管如此，语言模型即使在有限的形式下也是非常有用的。例如，在文档摘要生成算法中，"狗咬人"比"人咬狗"频繁得多，或者"我想吃奶奶"是一个相当令人不安的语句，而"我想吃，奶奶"要温和得多。

7.3.1 学习语言模型

显而易见的问题是，我们应该如何建模一个文档，或者一串标记。假设我们在单词级别对文本数据进行标记化。我们可以求助于我们在 7.1节 中应用于序列模型的分析。让我们从应用基本概率规则开始：

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^T P(x_t \mid x_1, \ldots, x_{t-1}). \tag{7.3.2}$$

例如，文本序列包含四个单词的概率将被给出：

$$P(\text{deep}, \text{learning}, \text{is}, \text{fun}) = P(\text{deep})P(\text{learning} \mid \text{deep})P(\text{is} \mid \text{deep}, \text{learning})P(\text{fun} \mid \text{deep}, \text{learning}, \text{is}). \tag{7.3.3}$$

为了计算语言模型，我们需要计算单词的概率和给定前面几个单词时出现该单词的条件概率。这样的概率本质上是语言模型参数。

这里，我们假设训练数据集是一个大型文本语料库，比如所有维基百科条目，古登堡计划[95]，以及发布在网络上的所有文本。可以根据训练数据集中给定词的相对词频来计算词的概率。例如，可以将估计值$\hat{P}(\text{deep})$计

[94] https://discuss.d2l.ai/t/2093

[95] https://en.wikipedia.org/wiki/Project_Gutenberg

算为任何以单词“Deep”开头的句子的概率。一种稍微不太准确的方法是统计单词“Deep”的所有出现次数，然后将其除以语料库中的单词总数。这很有效，特别是对于频繁出现的单词。接下来，我们可以尝试估计

$$\hat{P}(\text{learning} \mid \text{deep}) = \frac{n(\text{deep, learning})}{n(\text{deep})}, \tag{7.3.4}$$

其中$n(x)$和$n(x, x')$分别是单个单词和连续单词对的出现次数。不幸的是，由于“深度学习”的出现频率要低得多，所以估计词对的概率要困难得多。特别是，对于一些不寻常的单词组合，可能很难找到足够的出现次数来获得准确的估计。对于三个字的组合和以后的情况，情况变得更糟了。将会有许多可能在数据集中看不到的，但又看似合理的三字组合。除非我们提供一些解决方案来将这些单词组合指定为非零计数，否则我们将无法在语言模型中使用它们。如果数据集很小，或者如果单词非常罕见，我们可能甚至找不到一次出现。

一种常见的策略是执行某种形式的拉普拉斯平滑（Laplace smoothing）。解决方案是在所有计数中添加一个小常量。用n表示训练集中的单词总数，用m表示唯一单词的数量。此解决方案有助于处理个例问题，例如通过：

$$\begin{aligned}
\hat{P}(x) & = \frac{n(x) + \epsilon_1/m}{n + \epsilon_1}, \\
\hat{P}(x' \mid x) & = \frac{n(x, x') + \epsilon_2 \hat{P}(x')}{n(x) + \epsilon_2}, \\
\hat{P}(x'' \mid x,x') & = \frac{n(x, x',x'') + \epsilon_3 \hat{P}(x'')}{n(x, x') + \epsilon_3}.
\end{aligned} \tag{7.3.5}$$

其中，ϵ_1,ϵ_2和ϵ_3是超参数。以ϵ_1为例：当为$\epsilon_1 = 0$时，不应用平滑；当ϵ_1接近正无穷大时，$\hat{P}(x)$接近均匀概率$1/m$。以上是其他技术可以实现的 [Wood et al., 2011] 的一个相当原始的变体。

不幸的是，像这样的模型很快就会变得笨拙，原因如下：首先，我们需要存储所有计数。第二，这完全忽略了单词的意思。例如，“猫”和“猫科动物”应该出现在相关的上下文中。很难将这些模型调整到额外的上下文中，而基于深度学习的语言模型很适合考虑到这一点。最后，长单词序列几乎肯定是新出现的，因此简单地统计过往看到单词序列频率的模型肯定表现不佳。

7.3.2 马尔可夫模型与n元语法

在我们讨论基于深度学习的解决方案之前，我们需要更多的术语和概念。回想一下我们在 7.1节 中对马尔可夫模型的讨论。让我们将其应用于语言建模。序列上的分布满足一阶马尔可夫性质$P(x_{t+1} \mid x_t, \ldots, x_1) = P(x_{t+1} \mid x_t)$的。阶数越高，对应的依赖关系就越长。这导致了我们可以应用于序列建模的许多近似：

$$\begin{aligned}
P(x_1, x_2, x_3, x_4) &= P(x_1) P(x_2) P(x_3) P(x_4),\\
P(x_1, x_2, x_3, x_4) &= P(x_1) P(x_2 \mid x_1) P(x_3 \mid x_2) P(x_4 \mid x_3),\\
P(x_1, x_2, x_3, x_4) &= P(x_1) P(x_2 \mid x_1) P(x_3 \mid x_1, x_2) P(x_4 \mid x_2, x_3).
\end{aligned} \tag{7.3.6}$$

涉及一个、两个和三个变量的概率公式通常分别称为“单变量模型”（unigram）、“双变量模型”（bigram）和“三变量模型”（trigram）。在下面，我们将学习如何设计更好的模型。

7.3.3 自然语言统计

让我们看看这是如何对真实数据起作用的。我们根据 7.2节 中介绍的时光机器数据集构建词汇表，并打印最常用的10个单词。

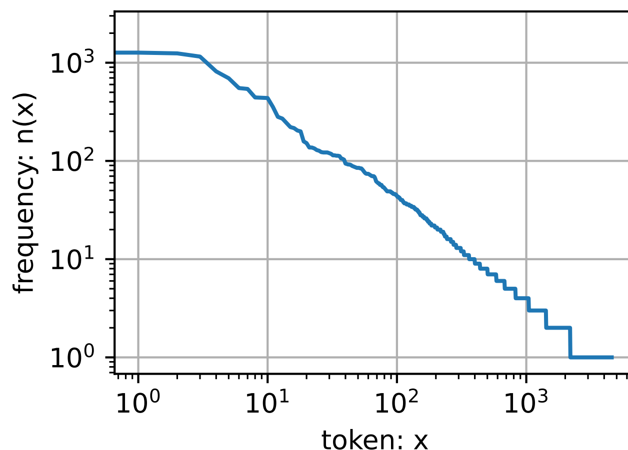
```
import random
from mxnet import np, npx
from d2l import mxnet as d2l

npx.set_np()
```

```
tokens = d2l.tokenize(d2l.read_time_machine())
# 因为每个文本行不一定是一个句子或一个段落，
# 我们连接所有文本行
corpus = [token for line in tokens for token in line]
vocab = d2l.Vocab(corpus)
vocab.token_freqs[:10]
```

```
[('the', 2261),
 ('i', 1267),
 ('and', 1245),
 ('of', 1155),
 ('a', 816),
 ('to', 695),
 ('was', 552),
 ('in', 541),
 ('that', 443),
 ('my', 440)]
```

正如我们所看到的，最流行的词实际上看起来很无聊。它们通常被称为"停用词"（stop words），因此可以被过滤掉。尽管如此，它们仍然有意义，我们仍然会使用它们。此外，很明显，词频衰减得相当快。第10个最常用单词的词频还不到最流行单词词频的1/5。为了得到一个更好的概念，我们画出了词频图表。

```
freqs = [freq for token, freq in vocab.token_freqs]
d2l.plot(freqs, xlabel='token: x', ylabel='frequency: n(x)', xscale='log',
         yscale='log')
```



我们在这里看到了一些非常基本的东西：词频以一种明确的方式迅速衰减。将前几个单词作为例外处理后，所有剩余的单词大致沿着对数曲线上的一条直。这意味着单词符合齐普夫定律（Zipf’s law），即第i个最常用单词的频率n_i为：

$$n_i \propto \frac{1}{i^\alpha}, \tag{7.3.7}$$

这相当于

$$\log n_i = -\alpha \log i + c, \tag{7.3.8}$$

其中α是表征分布的指数，c是常数。如果我们想要通过计数统计和平滑来建模单词，这应该已经让我们停下来了。毕竟，我们会大大高估尾部的频率，也就是所谓的不常用单词。但是其他的单词组合呢，比如二元语法、三元语法等等呢？让我们看看双字频率是否与单字频率的行为方式相同。

```
bigram_tokens = [pair for pair in zip(corpus[:-1], corpus[1:])]
bigram_vocab = d2l.Vocab(bigram_tokens)
bigram_vocab.token_freqs[:10]
```

```
[(('of', 'the'), 309),
 (('in', 'the'), 169),
 (('i', 'had'), 130),
 (('i', 'was'), 112),
 (('and', 'the'), 109),
 (('the', 'time'), 102),
 (('it', 'was'), 99),
 (('to', 'the'), 85),
 (('as', 'i'), 78),
 (('of', 'a'), 73)]
```

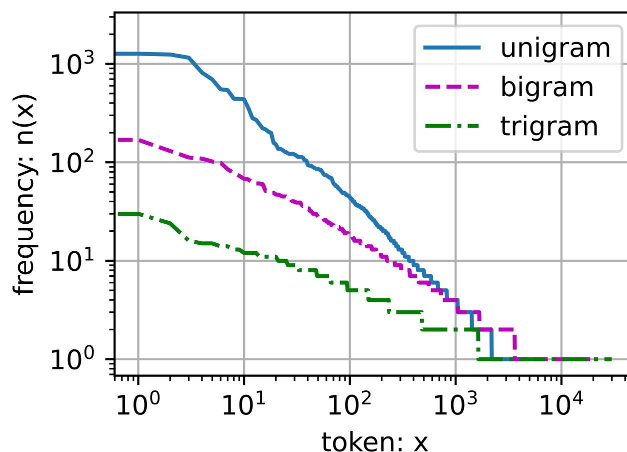
这里有一件事值得注意。在十个最频繁的词对中，有九个是由两个停用词组成的，只有一个与实际的书——《时间》有关。此外，让我们看看三元频率是否以相同的方式运行。

```
trigram_tokens = [
    triple for triple in zip(corpus[:-2], corpus[1:-1], corpus[2:])]
trigram_vocab = d2l.Vocab(trigram_tokens)
trigram_vocab.token_freqs[:10]
```

```
[(('the', 'time', 'traveller'), 59),
 (('the', 'time', 'machine'), 30),
 (('the', 'medical', 'man'), 24),
 (('it', 'seemed', 'to'), 16),
 (('here', 'and', 'there'), 15),
 (('it', 'was', 'a'), 15),
 (('i', 'did', 'not'), 14),
 (('seemed', 'to', 'me'), 14),
 (('i', 'began', 'to'), 13),
 (('i', 'saw', 'the'), 13)]
```

最后，让我们直观地看一下这三种模型中的标记频率：单字、双字和三字。

```
bigram_freqs = [freq for token, freq in bigram_vocab.token_freqs]
trigram_freqs = [freq for token, freq in trigram_vocab.token_freqs]
d2l.plot([freqs, bigram_freqs, trigram_freqs], xlabel='token: x',
         ylabel='frequency: n(x)', xscale='log', yscale='log',
         legend=['unigram', 'bigram', 'trigram'])
```



这个数字相当令人兴奋，原因有很多。首先，除了单字词，单词序列似乎也遵循齐普夫定律，尽管 (7.3.7) 中的指数α更小，这取决于序列长度。其次，n元组的数量并没有那么大。这给了我们希望，语言中有相当多的结构。第三，很多n元组很少出现，这使得拉普拉斯平滑非常不适合语言建模。相反，我们将使用基于深度学习的模型。

7.3.4 读取长序列数据

由于序列数据本质上是连续的，我们需要解决处理这带来的问题。我们在 7.1节 以一种相当特别的方式做到了这一点。当序列变得太长而不能被模型一次全部处理时，我们可能希望拆分这样的序列以供阅读。现在让我们描述一下总体策略。在介绍该模型之前，假设我们将使用神经网络来训练语言模型，其中该网络一次处理具有预定义长度的一小批序列，例如n个时间步。现在的问题是如何随机读取小批量的特征和标签。

首先，由于文本序列可以是任意长的，例如整个“时光机器”书，我们可以将这样长的序列划分为具有相同时间步数的子序列。当训练我们的神经网络时，子序列的小批量将被输入到模型中。假设网络一次处理n个时间步的子序列。图7.3.1 画出了从原始文本序列获得子序列的所有不同方式，其中$n = 5$和每个时间步的标记对应于一个字符。请注意，我们有相当大的自由度，因为我们可以选择指示初始位置的任意偏移量。

the time machine by h g wells
the time machine by h g wells
the time machine by h g wells
the time machine by h g wells
the time machine by h g wells
the time machine by h g wells

图7.3.1: 分割文本时，不同的偏移量会导致不同的子序列。

因此，我们应该从 图7.3.1 中选择哪一个呢？其实，他们都一样好。然而，如果我们只选择一个偏移量，那么用于训练网络的所有可能子序列的覆盖范围都是有限的。因此，我们可以从随机偏移量开始划分序列，以获得覆盖（coverage）和随机性（randomness）。在下面，我们将描述如何实现随机采样和顺序分区策略。

随机采样

在随机采样中，每个样本都是在原始长序列上任意捕获的子序列。迭代期间来自两个相邻随机小批量的子序列不一定在原始序列上相邻。对于语言建模，目标是根据我们到目前为止看到的标记来预测下一个标记，因此标签是原始序列移位了一个标记。

下面的代码每次从数据随机生成一个小批量。这里，参数 `batch_size` 指定每个小批量中的子序列样本数目，`num_steps` 是每个子序列中预定义的时间步数。

```python
def seq_data_iter_random(corpus, batch_size, num_steps):  #@save
    """使用随机抽样生成一个小批子序列。"""
    # 从随机偏移量（包括`num_steps - 1`）开始对序列进行分区
    corpus = corpus[random.randint(0, num_steps - 1):]
    # 减去1，因为我们需要考虑标签
    num_subseqs = (len(corpus) - 1) // num_steps
```

(continues on next page)

(continued from previous page)

```
    # 长度为`num_steps`的子序列的起始索引
    initial_indices = list(range(0, num_subseqs * num_steps, num_steps))
    # 在随机抽样中，迭代过程中两个相邻随机小批量的子序列不一定在原始序列上相邻
    random.shuffle(initial_indices)

    def data(pos):
        # 返回从`pos`开始的长度为`num_steps`的序列
        return corpus[pos:pos + num_steps]

    num_batches = num_subseqs // batch_size
    for i in range(0, batch_size * num_batches, batch_size):
        # 这里，`initial_indices`包含子序列的随机起始索引
        initial_indices_per_batch = initial_indices[i:i + batch_size]
        X = [data(j) for j in initial_indices_per_batch]
        Y = [data(j + 1) for j in initial_indices_per_batch]
        yield np.array(X), np.array(Y)
```

让我们手动生成一个从0到34的序列。我们设批量大小和时间步数分别为2和5。这意味着我们可以生成$\lfloor (35-1)/5 \rfloor = 6$个特征标签子序列对。小批量大小为2时，我们只能得到3个小批量。

```
my_seq = list(range(35))
for X, Y in seq_data_iter_random(my_seq, batch_size=2, num_steps=5):
    print('X: ', X, '\nY:', Y)
```

```
X:  [[ 9. 10. 11. 12. 13.]
 [19. 20. 21. 22. 23.]]
Y: [[10. 11. 12. 13. 14.]
 [20. 21. 22. 23. 24.]]
X:  [[ 4.  5.  6.  7.  8.]
 [24. 25. 26. 27. 28.]]
Y: [[ 5.  6.  7.  8.  9.]
 [25. 26. 27. 28. 29.]]
X:  [[14. 15. 16. 17. 18.]
 [29. 30. 31. 32. 33.]]
Y: [[15. 16. 17. 18. 19.]
 [30. 31. 32. 33. 34.]]
```

顺序分区

除了对原始序列进行随机抽样外，我们还可以保证迭代过程中两个相邻小批量的子序列在原始序列上是相邻的。这种策略在对小批进行迭代时保留了拆分子序列的顺序，因此称为顺序分区。

```
def seq_data_iter_sequential(corpus, batch_size, num_steps):  #@save
    """使用顺序分区生成一小批子序列。"""
    # 从随机偏移量开始划分序列
    offset = random.randint(0, num_steps)
    num_tokens = ((len(corpus) - offset - 1) // batch_size) * batch_size
    Xs = np.array(corpus[offset:offset + num_tokens])
    Ys = np.array(corpus[offset + 1:offset + 1 + num_tokens])
    Xs, Ys = Xs.reshape(batch_size, -1), Ys.reshape(batch_size, -1)
    num_batches = Xs.shape[1] // num_steps
    for i in range(0, num_steps * num_batches, num_steps):
        X = Xs[:, i:i + num_steps]
        Y = Ys[:, i:i + num_steps]
        yield X, Y
```

使用相同的设置，让我们为通过顺序分区读取的每个小批量的子序列打印特征X和标签Y。请注意，迭代期间来自两个相邻小批量的子序列实际上在原始序列上是相邻的。

```
for X, Y in seq_data_iter_sequential(my_seq, batch_size=2, num_steps=5):
    print('X: ', X, '\nY:', Y)
```

```
X:  [[ 4.  5.  6.  7.  8.]
 [19. 20. 21. 22. 23.]]
Y: [[ 5.  6.  7.  8.  9.]
 [20. 21. 22. 23. 24.]]
X:  [[ 9. 10. 11. 12. 13.]
 [24. 25. 26. 27. 28.]]
Y: [[10. 11. 12. 13. 14.]
 [25. 26. 27. 28. 29.]]
X:  [[14. 15. 16. 17. 18.]
 [29. 30. 31. 32. 33.]]
Y: [[15. 16. 17. 18. 19.]
 [30. 31. 32. 33. 34.]]
```

现在，我们将上述两个采样函数包装到一个类中，以便稍后可以将其用作数据迭代器。

```
class SeqDataLoader:  #@save
    """加载序列数据的迭代器。"""
    def __init__(self, batch_size, num_steps, use_random_iter, max_tokens):
        if use_random_iter:
            self.data_iter_fn = d2l.seq_data_iter_random
```

(continues on next page)

(continued from previous page)

```
        else:
            self.data_iter_fn = d2l.seq_data_iter_sequential
        self.corpus, self.vocab = d2l.load_corpus_time_machine(max_tokens)
        self.batch_size, self.num_steps = batch_size, num_steps

    def __iter__(self):
        return self.data_iter_fn(self.corpus, self.batch_size, self.num_steps)
```

最后，我们定义了一个函数 `load_data_time_machine`，它同时返回数据迭代器和词表，因此我们可以与其他带有 `load_data` 前缀的函数（如 2.5节 中定义的 `d2l.load_data_fashion_mnist`）类似地使用它。

```
def load_data_time_machine(batch_size, num_steps,  #@save
                           use_random_iter=False, max_tokens=10000):
    """返回时光机器数据集的迭代器和词表。"""
    data_iter = SeqDataLoader(batch_size, num_steps, use_random_iter,
                              max_tokens)
    return data_iter, data_iter.vocab
```

7.3.5 小结

- 语言模型是自然语言处理的关键。
- n元语法通过截断相关性，为处理长序列提供了一种方便的模型。
- 长序列有一个问题，那就是它们很少出现或从不出现。
- 齐普夫定律不仅规定了单字的单词分布，而且还规定了其他n元语法的单词分布。
- 通过拉普拉斯平滑法可以有效地处理不常见的、结构复杂且频率不够词组。
- 读取长序列的主要选择是随机采样和顺序分区。后者可以保证迭代过程中来自两个相邻小批量的子序列在原始序列上是相邻的。

7.3.6 练习

1. 假设训练数据集中有$100,000$个单词。四元语法需要存储多少词频和多词相邻频率？
2. 你将如何模拟对话？
3. 估计“单变量”（unigram）、“双变量”（bigram）和“三变量”（trigram）的齐普夫定律指数。
4. 您还能想到哪些其他的读取长序列数据的方法？
5. 考虑一下我们用于读取长序列的随机偏移量。
    1. 为什么随机偏移量是个好主意？

   2. 它真的会在文档上的序列上实现完美均匀的分布吗?
   3. 你要怎么做才能让事情变得更加统一呢?
6. 如果我们希望一个序列样本是一个完整的句子，那么这在小批量抽样中会带来什么样的问题呢？我们怎样才能解决这个问题呢?

Discussions[96]

7.4 循环神经网络

在 7.3节 中，我们介绍了n元语法模型，其中单词x_t在时间步t的条件概率仅取决于前面$n-1$个单词。如果我们想将时间步$t-(n-1)$之前的单词的可能影响合并到x_t上，我们需要增加n。但是，模型参数的数量也会随之呈指数增长，因为我们需要为词表$\mathcal{V}$存储$|\mathcal{V}|^n$个数字。因此，与其建模$P(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$，不如使用隐变量模型:

$$P(x_t \mid x_{t-1}, \ldots, x_1) \approx P(x_t \mid h_{t-1}), \tag{7.4.1}$$

其中h_{t-1}是隐藏状态（也称为隐藏变量），其存储了到时间步$t-1$的序列信息。通常，可以基于当前输入x_t和先前隐藏状态h_{t-1}来计算时间步t处的任何时间的隐藏状态:

$$h_t = f(x_t, h_{t-1}). \tag{7.4.2}$$

对于足够强大的函数f（(7.4.2)），隐变量模型不是近似值。毕竟，h_t可能只是存储到目前为止观察到的所有数据。然而，它可能会使计算和存储都变得昂贵。

回想一下，我们在 3节 中讨论过具有隐藏单元的隐藏层。值得注意的是，隐藏层和隐藏状态指的是两个截然不同的概念。如上所述，隐藏层是在从输入到输出的路径上从视图中隐藏的层。从技术上讲，隐藏状态是我们在给定步骤所做的任何事情的“输入”。隐藏状态只能通过查看先前时间点的数据来计算。

循环神经网络（Recurrent neural networks，RNNs）是具有隐藏状态的神经网络。在介绍RNN模型之前，我们首先回顾 3.1节 中介绍的多层感知机模型。

7.4.1 无隐藏状态的神经网络

让我们来看一看只有单隐藏层的多层感知机。设隐藏层的激活函数为ϕ。给定小批量样本$\mathbf{X} \in \mathbb{R}^{n\times d}$，其中批量大小为$n$，输入为$d$维。隐藏层的输出$\mathbf{H} \in \mathbb{R}^{n\times h}$通过下式计算:

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h). \tag{7.4.3}$$

在 (7.4.3) 中，我们有用于隐藏层的权重参数$\mathbf{W}_{xh} \in \mathbb{R}^{d\times h}$、偏置参数$\mathbf{b}_h \in \mathbb{R}^{1\times h}$，其中隐藏单元的数目为$h$。因此，在求和期间应用广播机制（见 1.1.3节 ）。接下来，将隐藏变量$\mathbf{H}$用作输出层的输入。输出层由下式给出:

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q, \tag{7.4.4}$$

[96] https://discuss.d2l.ai/t/2096

其中，$\mathbf{O} \in \mathbb{R}^{n \times q}$是输出变量，$\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$是权重参数，$\mathbf{b}_q \in \mathbb{R}^{1 \times q}$是输出层的偏置参数。如果是分类问题，我们可以用$\text{softmax}(\mathbf{O})$来计算输出类别的概率分布。

这完全类似于我们之前在 7.1节 中解决的回归问题，因此我们省略了细节。可以说，我们可以随机选择特征-标签对，并通过自动微分和随机梯度下降来学习网络参数。

7.4.2 具有隐藏状态的循环神经网络

当我们有隐藏状态时，情况就完全不同了。让我们更详细地看看这个结构。

假设我们在时间步t有小批量输入$\mathbf{X}_t \in \mathbb{R}^{n \times d}$。换言之，对于$n$个序列样本的小批量，$\mathbf{X}_t$的每行对应于来自该序列的时间步$t$处的一个样本。接下来，用$\mathbf{H}_t \in \mathbb{R}^{n \times h}$表示时间步$t$的隐藏变量。与最大似然算法不同的是，这里我们保存了前一个时间步的隐藏变量$\mathbf{H}_{t-1}$，并引入了一个新的权重参数$\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$来描述如何在当前时间步中使用前一个时间步的隐藏变量。具体地，当前时间步的隐藏变量计算由当前时间步的输入与前一个时间步的隐藏变量一起确定：

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h). \tag{7.4.5}$$

与 (7.4.3) 相比，(7.4.5) 多添加了一项$\mathbf{H}_{t-1} \mathbf{W}_{hh}$，从而实例化了 (7.4.2)。从相邻时间步的隐藏变量$\mathbf{H}_t$和$\mathbf{H}_{t-1}$之间的关系可知，这些变量捕获并保留了序列直到其当前时间步的历史信息，就像神经网络的当前时间步的状态或记忆一样。因此，这样的隐藏变量被称为"隐藏状态"（hidden state）。由于隐藏状态使用与当前时间步中的前一个时间步相同的定义，因此 (7.4.5) 的计算是循环的（recurrent）。因此，基于循环计算的隐状态神经网络被命名为循环神经网络（recurrent neural networks）。在循环神经网络中执行 (7.4.5) 计算的层称为"循环层"（recurrent layers）。

构建循环神经网络有许多不同的方法。具有由 (7.4.5) 定义的隐藏状态的循环神经网络非常常见。对于时间步t，输出层的输出类似于多层感知机中的计算：

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \tag{7.4.6}$$

循环神经网络的参数包括隐藏层的权重$\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}, \mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$和偏置$\mathbf{b}_h \in \mathbb{R}^{1 \times h}$，以及输出层的权重$\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$和偏置$\mathbf{b}_q \in \mathbb{R}^{1 \times q}$。值得一提的是，即使在不同的时间步，循环神经网络也总是使用这些模型参数。因此，循环神经网络的参数开销不会随着时间步的增加而增加。

图7.4.1展示了在三个相邻时间步的循环神经网络计算逻辑。在任意时间步t，隐藏状态的计算可以被视为：1、将当前时间步t的输入$\mathbf{X}_t$和前一时间步$t-1$的隐藏状态$\mathbf{H}_{t-1}$连结；2、将连结结果送入带有激活函数ϕ的全连接层。全连接层的输出是当前时间步t的隐藏状态$\mathbf{H}_t$。在本例中，模型参数是$\mathbf{W}_{xh}$和$\mathbf{W}_{hh}$的连结，以及$\mathbf{b}_h$的偏置，所有这些参数都来自 (7.4.5)。当前时间步t、$\mathbf{H}_t$的隐藏状态将参与计算下一时间步$t+1$的隐藏状态$\mathbf{H}_{t+1}$。此外，还将$\mathbf{H}_t$送入全连接输出层，以计算当前时间步t的输出$\mathbf{O}_t$。

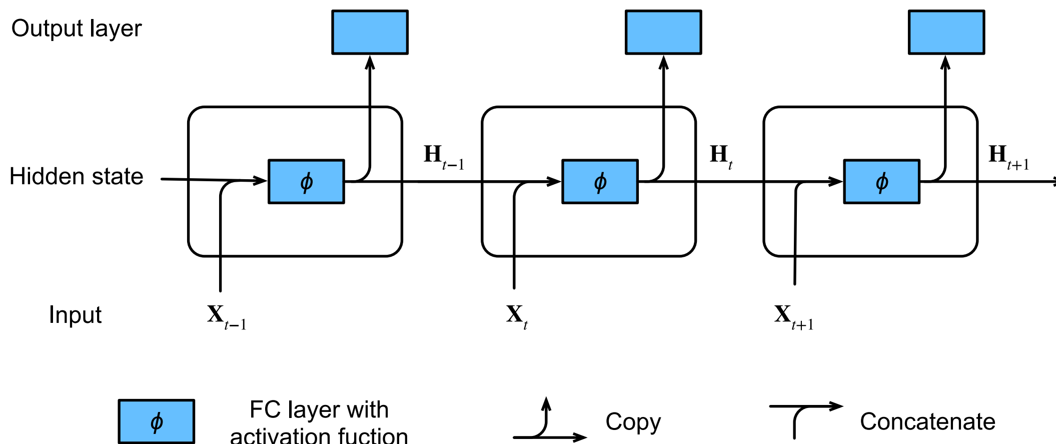


图7.4.1: 具有隐藏状态的循环神经网络。

我们刚才提到，隐藏状态的计算$\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh}$，相当于$\mathbf{X}_t$和$\mathbf{H}_{t-1}$连结和$\mathbf{W}_{xh}$和$\mathbf{W}_{hh}$连结的矩阵乘法。虽然可以在数学上证明这一点，但在下面我们只使用一个简单的代码片段来说明这一点。首先，我们定义矩阵X、W_xh、H和W_hh，它们的形状分别为(3，1)、(1，4)、(3，4)和(4，4)。分别将X乘以W_xh，将H乘以W_hh，然后将这两个乘法相加，我们得到一个形状为(3，4)的矩阵。

```
from mxnet import np, npx
from d2l import mxnet as d2l

npx.set_np()
```

```
X, W_xh = np.random.normal(0, 1, (3, 1)), np.random.normal(0, 1, (1, 4))
H, W_hh = np.random.normal(0, 1, (3, 4)), np.random.normal(0, 1, (4, 4))
np.dot(X, W_xh) + np.dot(H, W_hh)
```

```
array([[-0.21952868,  4.256434  ,  4.5812645 , -5.344988  ],
       [ 3.4478583 , -3.0177274 , -1.6777471 ,  7.535347  ],
       [ 2.239007  ,  1.4199957 ,  4.744728  , -8.421293  ]])
```

现在，我们沿列（轴1）连结矩阵X和H，沿行（轴0）连结矩阵W_xh和W_hh。这两个连结分别产生形状(3, 5)和形状(5, 4)的矩阵。将这两个连结的矩阵相乘，我们得到与上面相同形状(3, 4)的输出矩阵。

```
np.dot(np.concatenate((X, H), 1), np.concatenate((W_xh, W_hh), 0))
```

```
array([[-0.2195287,  4.256434 ,  4.5812645, -5.344988 ],
       [ 3.4478583, -3.0177271, -1.677747 ,  7.535347 ],
       [ 2.2390068,  1.4199957,  4.744728 , -8.421294 ]])
```

7.4.3 基于循环神经网络的字符级语言模型

回想一下，对于 7.3节 中的语言模型。我们的目标是根据当前和过去的标记预测下一个标记，因此我们将原始序列移位一个标记作为标签。Bengio等人 [Bengio et al., 2003] 首先提出使用神经网络进行语言建模。接下来，我们将说明如何使用循环神经网络来构建语言模型。设小批量大小为1，文本序列为“machine”。为了简化后续部分的训练，我们将文本标记化为字符而不是单词，并考虑使用字符级语言模型（character-level language model）。图7.4.2 演示了如何通过用于字符级语言建模的循环神经网络，基于当前字符和先前字符预测下一个字符。

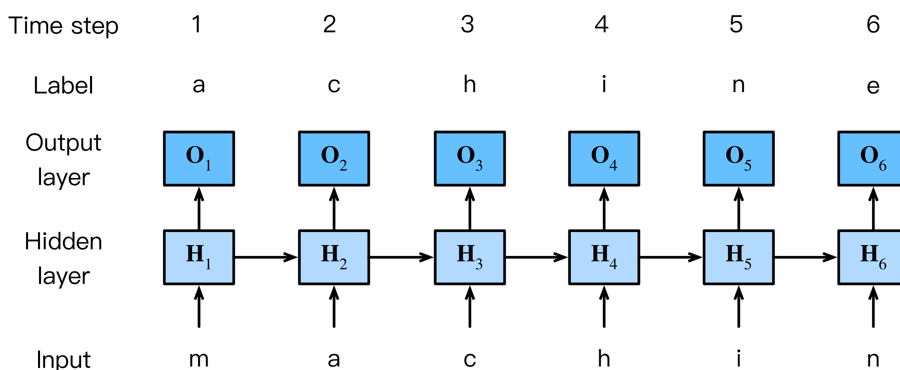


图7.4.2: 基于循环神经网络的字符级语言模型。输入序列和标签序列分别为“machin”和“achine”。

在训练过程中，我们对每个时间步长的输出层的输出进行softmax操作，然后利用交叉熵损失计算模型输出和标签之间的误差。由于隐藏层中隐藏状态的循环计算，图7.4.2 中的时间步骤3的输出$\mathbf{O}_3$由文本序列“m”、“a”和“c”确定。由于训练数据中序列的下一个字符是“h”，因此时间步3的损失将取决于基于该时间步的特征序列“m”、“a”、“c”生成的下一个字符概率分布和标签“h”。

实际上，每个标记都由一个d维向量表示，我们使用批量大小$n > 1$。因此，输入$\mathbf{X}_t$在时间步t将是$n \times d$矩阵，这与我们在 7.4.2节 中讨论的相同。

7.4.4 困惑度（Perplexity）

最后，让我们讨论如何度量语言模型质量，这将在后续部分中用于评估我们基于循环神经网络的模型。一种方法是检查文本有多令人惊讶。一个好的语言模型能够用高精度的标记来预测我们接下来会看到什么。考虑一下不同语言模型对短语“It is raining”提出以下续写：

1. “It is raining outside”
2. “It is raining banana tree”
3. “It is raining piouw;kcj pwepoiut”

就质量而言，例1显然是最好的。这些词是明智的，逻辑上是连贯的。虽然这个模型可能不能很准确地反映出哪个词在语义上跟在后面（“in San Francisco”和“in winter”可能是完全合理的扩展），但该模型能够捕捉

到跟在后面的是哪种单词。例2产生了一个无意义的续写，这要糟糕得多。尽管如此，至少该模型已经学会了如何拼写单词以及单词之间的某种程度的相关性。最后，例3指出训练不足的模型不能很好地拟合数据。

我们可以通过计算序列的似然概率来衡量模型的质量。不幸的是，这是一个很难理解和难以比较的数字。毕竟，较短的序列比较长的序列更有可能出现，因此在托尔斯泰的巨著《战争与和平》上对该模型进行评估不可避免地会比圣埃克苏佩里的中篇小说《小王子》产生的可能性要小得多。缺少的相当于平均数。

信息论在这里派上了用场。。我们在引入softmax回归（2.4.7节 ）时定义了熵、奇异熵和交叉熵，并在信息论的在线附录[97]中讨论了更多的信息论。如果我们想压缩文本，我们可以询问在给定当前标记集的情况下预测下一个标记。一个更好的语言模型应该能让我们更准确地预测下一个标记。因此，它应该允许我们在压缩序列时花费更少的比特。所以我们可以通过一个序列中所有n个标记的平均交叉熵损失来衡量：

$$\frac{1}{n}\sum_{t=1}^{n} -\log P(x_t \mid x_{t-1}, \ldots, x_1), \tag{7.4.7}$$

其中P由语言模型给出，x_t是在时间步t从该序列观察到的实际标记。这使得在不同长度的文档上的性能具有可比性。由于历史原因，自然语言处理的科学家更喜欢使用一个叫做“困惑度”（perplexity）的量。简而言之，它是 (7.4.7) 的指数：

$$\exp\left(-\frac{1}{n}\sum_{t=1}^{n} \log P(x_t \mid x_{t-1}, \ldots, x_1)\right). \tag{7.4.8}$$

困惑度可以最好地理解为当我们决定下一个选择哪个标记时，实际选择数的调和平均数。让我们看看一些案例：

- 在最好的情况下，模型总是完美地估计标签标记的概率为1。在这种情况下，模型的困惑度为1。
- 在最坏的情况下，模型总是预测标签标记的概率为0。在这种情况下，困惑度是正无穷大。
- 在基线上，该模型预测词汇表的所有可用标记上的均匀分布。在这种情况下，困惑程度等于词表中唯一标记的数量。事实上，如果我们在没有任何压缩的情况下存储序列，这将是我们能做的最好的编码。因此，这提供了一个重要的上限，任何实际模型都必须超越这个上限。

在接下来的几节中，我们将为字符级语言模型实现循环神经网络，并使用困惑度来评估这些模型。

7.4.5 小结

- 对隐藏状态使用循环计算的神经网络称为循环神经网络（RNN）。
- 循环神经网络的隐藏状态可以捕获直到当前时间步的序列的历史信息。
- 循环神经网络模型的参数数量不会随着时间步的增加而增加。
- 我们可以使用循环神经网络创建字符级语言模型。
- 我们可以用困惑度来评价语言模型的质量。

[97] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/information-theory.html

7.4.6 练习

1. 如果我们使用循环神经网络来预测文本序列中的下一个字符，那么任意输出所需的维度是什么？
2. 为什么循环神经网络可以基于文本序列中所有先前的标记在某个时间步表示标记的条件概率？
3. 如果你反向传播一个长序列，梯度会发生什么？
4. 与本节中描述的语言模型相关的问题有哪些？

Discussions[98]

7.5 循环神经网络的从零开始实现

在本节中，我们将根据 7.4节 中的描述，从头开始为字符级语言模型实现循环神经网络。这样的模型将在时光机器数据集上训练。和前面一样，我们首先读取数据集，它在 7.3节 中介绍过。

```
%matplotlib inline
import math
from mxnet import autograd, gluon, np, npx
from d2l import mxnet as d2l

npx.set_np()
```

```
batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

7.5.1 独热编码

回想一下，在`train_iter`中，每个标记都表示为一个数字索引。将这些数字直接输入神经网络可能会使学习变得困难。我们通常将每个标记表示为更具表现力的特征向量。最简单的表示称为"独热编码"（One-Hot Encoding），它在 2.4.1节 中介绍过。

简言之，我们将每个索引映射到一个不同的单位向量：假设词表中不同的标记数为N（`len(vocab)`），标记索引的范围为0到$N-1$。如果标记的索引是整数i，那么我们创建一个长度为N的全0向量，并将i处的元素设置为1。此向量是原始标记的一个独热向量。索引为0和2的独热向量如下所示。

```
npx.one_hot(np.array([0, 2]), len(vocab))
```

```
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

[98] https://discuss.d2l.ai/t/2099

我们每次采样的小批量形状是(批量大小, 时间步数)。one_hot函数将这样一个小批量转换成三维张量，最后一个维度等于词表大小（len(vocab)）。我们经常置换输入的维度，以便获得形状(时间步数, 批量大小, 词汇表大小)的输出。这将使我们能够更方便地通过最外层的维度，一步一步地更新小批量的隐藏状态。

```
X = np.arange(10).reshape((2, 5))
npx.one_hot(X.T, 28).shape
```

```
(5, 2, 28)
```

7.5.2 初始化模型参数

接下来，我们初始化循环神经网络模型的模型参数。隐藏单元数num_hiddens是一个可调的超参数。当训练语言模型时，输入和输出来自相同的词表。因此，它们具有相同的维度，即等于词表的大小。

```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return np.random.normal(scale=0.01, size=shape, ctx=device)

    # 隐藏层参数
    W_xh = normal((num_inputs, num_hiddens))
    W_hh = normal((num_hiddens, num_hiddens))
    b_h = np.zeros(num_hiddens, ctx=device)
    # 输出层参数
    W_hq = normal((num_hiddens, num_outputs))
    b_q = np.zeros(num_outputs, ctx=device)
    # 附加梯度
    params = [W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.attach_grad()
    return params
```

7.5.3 循环神经网络模型

为了定义循环神经网络模型，我们首先需要一个init_rnn_state函数在初始化时返回隐藏状态。它返回一个张量，全用0填充，形状为(批量大小, 隐藏单元数)。使用元组可以更容易地处理隐藏状态包含多个变量的情况，我们将在后面的部分中遇到这些情况。

```
def init_rnn_state(batch_size, num_hiddens, device):
    return (np.zeros((batch_size, num_hiddens), ctx=device),)
```

下面的rnn函数定义了如何在一个时间步计算隐藏状态和输出。请注意，循环神经网络模型通过最外层维度inputs循环，以便逐时间步更新小批量的隐藏状态H。此外，这里的激活函数使用tanh函数。如 3.1节 所述，当元素在实数上均匀分布时，tanh函数的平均值为0。

```
def rnn(inputs, state, params):
    # `inputs`的形状：(`时间步数量`，`批量大小`，`词表大小`)
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    # `X`的形状：(`批量大小`，`词表大小`)
    for X in inputs:
        H = np.tanh(np.dot(X, W_xh) + np.dot(H, W_hh) + b_h)
        Y = np.dot(H, W_hq) + b_q
        outputs.append(Y)
    return np.concatenate(outputs, axis=0), (H,)
```

定义了所有需要的函数之后，接下来我们创建一个类来包装这些函数，并存储从零开始实现的循环神经网络模型的参数。

```
class RNNModelScratch:  #@save
    """从零开始实现的循环神经网络模型"""
    def __init__(self, vocab_size, num_hiddens, device, get_params,
                 init_state, forward_fn):
        self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
        self.params = get_params(vocab_size, num_hiddens, device)
        self.init_state, self.forward_fn = init_state, forward_fn

    def __call__(self, X, state):
        X = npx.one_hot(X.T, self.vocab_size)
        return self.forward_fn(X, state, self.params)

    def begin_state(self, batch_size, ctx):
        return self.init_state(batch_size, self.num_hiddens, ctx)
```

让我们检查输出是否具有正确的形状，例如，确保隐藏状态的维数保持不变。

```
num_hiddens = 512
net = RNNModelScratch(len(vocab), num_hiddens, d2l.try_gpu(), get_params,
                      init_rnn_state, rnn)
state = net.begin_state(X.shape[0], d2l.try_gpu())
Y, new_state = net(X.as_in_context(d2l.try_gpu()), state)
Y.shape, len(new_state), new_state[0].shape
```

```
((10, 28), 1, (2, 512))
```

我们可以看到输出形状是(时间步数×批量大小, 词汇表大小)，而隐藏状态形状保持不变，即(批量大小, 隐藏单元数)。

7.5.4 预测

让我们首先定义预测函数来生成用户提供的prefix之后的新字符，prefix是一个包含多个字符的字符串。在prefix中循环遍历这些开始字符时，我们不断地将隐藏状态传递到下一个时间步，而不生成任何输出。这被称为"预热"（Warm-up）期，在此期间模型会自我更新（例如，更新隐藏状态），但不会进行预测。预热期过后，隐藏状态通常比开始时的初始值好。

```
def predict_ch8(prefix, num_preds, net, vocab, device):  #@save
    """在`prefix`后面生成新字符。"""
    state = net.begin_state(batch_size=1, ctx=device)
    outputs = [vocab[prefix[0]]]
    get_input = lambda: np.array([outputs[-1]], ctx=device).reshape((1, 1))
    for y in prefix[1:]:  # 预热期
        _, state = net(get_input(), state)
        outputs.append(vocab[y])
    for _ in range(num_preds):  # 预测`num_preds`步
        y, state = net(get_input(), state)
        outputs.append(int(y.argmax(axis=1).reshape(1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

现在我们可以测试predict_ch8函数。我们将前缀指定为time traveller，并让它生成10个后续字符。鉴于我们没有训练网络，它会产生荒谬的预测。

```
predict_ch8('time traveller ', 10, net, vocab, d2l.try_gpu())
```

```
'time traveller iiiiiiiiii'
```

7.5.5 梯度裁剪

对于长度为T的序列，我们在迭代中计算这些T个时间步上的梯度，从而在反向传播过程中产生长度为$\mathcal{O}(T)$的矩阵乘法链。如 3.8节 所述，当T较大时，它可能导致数值不稳定，例如可能梯度爆炸或梯度消失。因此，循环神经网络模型往往需要额外的帮助来稳定训练。

一般来说，在解决优化问题时，我们对模型参数采取更新步骤，例如在向量形式的$\mathbf{x}$中，在小批量的负梯度$\mathbf{g}$方向上。例如，使用$\eta > 0$作为学习率，在一次迭代中，我们将$\mathbf{x}$更新为$\mathbf{x} - \eta \mathbf{g}$。让我们进一步假设目标函数$f$表现良好，例如，李卜希兹连续（Lipschitz continuous）常数L。也就是说，对于任意$\mathbf{x}$和$\mathbf{y}$我们有：

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L \|\mathbf{x} - \mathbf{y}\|. \tag{7.5.1}$$

在这种情况下，我们可以合理地假设，如果我们将参数向量通过$\eta \mathbf{g}$更新，那么：

$$|f(\mathbf{x}) - f(\mathbf{x} - \eta\mathbf{g})| \leq L \eta\|\mathbf{g}\|, \tag{7.5.2}$$

这意味着我们不会观察到超过$L\eta\|\mathbf{g}\|$的变化。这既是诅咒也是祝福。在诅咒的一面，它限制了进步的速度；而在祝福的一面，它限制了如果我们朝着错误的方向前进，事情会出错的程度。

有时梯度可能很大，优化算法可能无法收敛。我们可以通过降低η的学习率来解决这个问题。但是如果我们很少得到大的梯度呢？在这种情况下，这种做法似乎完全没有根据。一个流行的替代方法是通过将梯度$\mathbf{g}$投影回给定半径的球（例如θ）来裁剪梯度$\mathbf{g}$。如下式：

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right) \mathbf{g}. \tag{7.5.3}$$

通过这样做，我们知道梯度范数永远不会超过θ，并且更新后的梯度完全与$\mathbf{g}$的原始方向对齐。它还有一个理想的副作用，即限制任何给定的小批量（以及其中任何给定的样本）对参数向量的影响。这赋予了模型一定程度的健壮性。梯度裁剪提供了一个快速修复梯度爆炸的方法。虽然它并不能完全解决问题，但它是众多缓解问题的技术之一。

下面我们定义一个函数来裁剪从零开始实现的模型或由高级API构建的模型的梯度。还要注意，我们计算了所有模型参数的梯度范数。

```python
def grad_clipping(net, theta):  #@save
    """裁剪梯度。"""
    if isinstance(net, gluon.Block):
        params = [p.data() for p in net.collect_params().values()]
    else:
        params = net.params
    norm = math.sqrt(sum((p.grad**2).sum() for p in params))
    if norm > theta:
        for param in params:
            param.grad[:] *= theta / norm
```

7.5.6 训练

在训练模型之前，让我们定义一个函数来训练只有一个迭代周期的模型。它与我们训练 2.6节 模型的方式有三个不同之处：

1. 顺序数据的不同采样方法（随机采样和顺序分区）将导致隐藏状态初始化的差异。
2. 我们在更新模型参数之前裁剪梯度。这确保了即使在训练过程中的某个点上梯度爆炸，模型也不会发散。
3. 我们用困惑度来评价模型。如 7.4.4节 所述，这确保了不同长度的序列具有可比性。

具体地说，当使用顺序分区时，我们只在每个迭代周期的开始处初始化隐藏状态。由于下一个小批量中的i^th子序列样本与当前i^th子序列样本相邻，因此当前小批量末尾的隐藏状态将用于初始化下一个小批量开头的隐藏状态。这样，存储在隐藏状态中的序列历史信息可以在一个迭代周期内流过相邻的子序列。然而，任何一点隐藏状态计算都依赖于同一迭代周期中所有的前一个小批量，这使得梯度计算变得复杂。为了降低计算量，我们在处理任何一个小批量之前先分离梯度，使得隐藏状态的梯度计算总是限制在一个小批量的时间步内。

当使用随机抽样时，我们需要为每个迭代周期重新初始化隐藏状态，因为每个样本都是在一个随机位置抽样的。与 2.6节 中的 train_epoch_ch3 函数相同，updater是更新模型参数的常用函数。它既可以是从头开始实现的 d2l.sgd 函数，也可以是深度学习框架中的内置优化函数。

```python
#@save
def train_epoch_ch8(net, train_iter, loss, updater, device, use_random_iter):
    """训练模型一个迭代周期（定义见第8章）。"""
    state, timer = None, d2l.Timer()
    metric = d2l.Accumulator(2)  # 训练损失之和, 标记数量
    for X, Y in train_iter:
        if state is None or use_random_iter:
            # 在第一次迭代或使用随机抽样时初始化`state`
            state = net.begin_state(batch_size=X.shape[0], ctx=device)
        else:
            for s in state:
                s.detach()
        y = Y.T.reshape(-1)
        X, y = X.as_in_ctx(device), y.as_in_ctx(device)
        with autograd.record():
            y_hat, state = net(X, state)
            l = loss(y_hat, y).mean()
        l.backward()
        grad_clipping(net, 1)
        updater(batch_size=1)  # 因为已经调用了`mean`函数
        metric.add(l * y.size, y.size)
    return math.exp(metric[0] / metric[1]), metric[1] / timer.stop()
```

训练函数支持从零开始或使用高级API实现的循环神经网络模型。

```python
def train_ch8(net, train_iter, vocab, lr, num_epochs, device,  #@save
              use_random_iter=False):
    """训练模型（定义见第8章）。"""
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='perplexity',
                            legend=['train'], xlim=[10, num_epochs])
    # 初始化
    if isinstance(net, gluon.Block):
        net.initialize(ctx=device, force_reinit=True, init=init.Normal(0.01))
        trainer = gluon.Trainer(net.collect_params(), 'sgd',
                                {'learning_rate': lr})
        updater = lambda batch_size: trainer.step(batch_size)
    else:
        updater = lambda batch_size: d2l.sgd(net.params, lr, batch_size)
    predict = lambda prefix: predict_ch8(prefix, 50, net, vocab, device)
    # 训练和预测
```

(continues on next page)

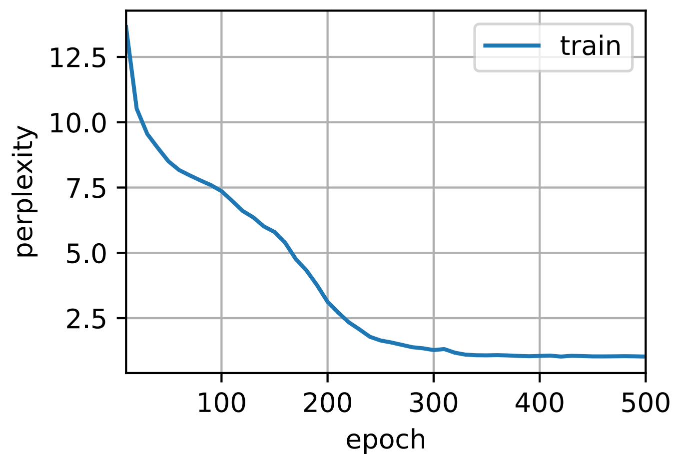
(continued from previous page)

```
    for epoch in range(num_epochs):
        ppl, speed = train_epoch_ch8(net, train_iter, loss, updater, device,
                                     use_random_iter)
        if (epoch + 1) % 10 == 0:
            animator.add(epoch + 1, [ppl])
    print(f'困惑度 {ppl:.1f}, {speed:.1f} 标记/秒 {str(device)}')
    print(predict('time traveller'))
    print(predict('traveller'))
```

现在我们可以训练循环神经网络模型。因为我们在数据集中只使用10000个标记，所以模型需要更多的迭代周期来更好地收敛。

```
num_epochs, lr = 500, 1
train_ch8(net, train_iter, vocab, lr, num_epochs, d2l.try_gpu())
```

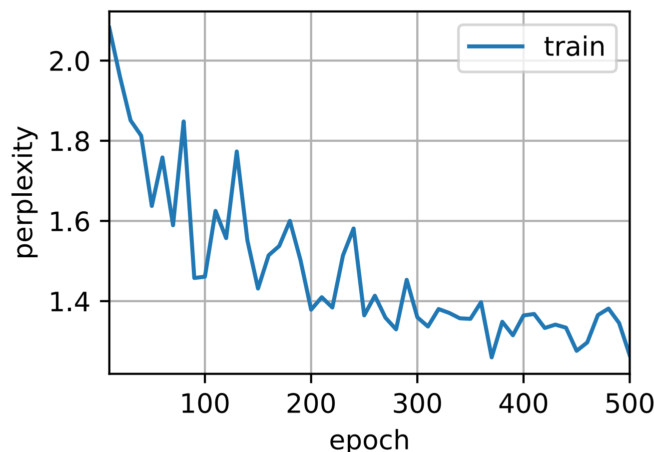
```
困惑度 1.0, 31066.0 标记/秒 gpu(0)
time traveller with a slight accession ofcheerfulness really thi
travelleryou can show black is white by argument said filby
```



最后，让我们检查一下使用随机抽样方法的结果。

```
train_ch8(net, train_iter, vocab, lr, num_epochs, d2l.try_gpu(),
          use_random_iter=True)
```

```
困惑度 1.3, 28290.3 标记/秒 gpu(0)
time traveller held in his hand was a glitteringmetallic framewo
traveller held in his hand was a glitteringmetallic framewo
```



虽然从零开始实现上述循环神经网络模型是有指导意义的，但并不方便。在下一节中，我们将看到如何改进循环神经网络模型，例如，如何使其更易于实现并运行得更快。

7.5.7 小结

- 我们可以训练一个基于循环神经网络的字符级语言模型，根据用户提供的文本前缀生成文本。
- 一个简单的循环神经网络语言模型包括输入编码、循环神经网络模型和输出生成。
- 循环神经网络模型需要状态初始化来训练，尽管随机抽样和顺序划分使用不同的方法。
- 当使用顺序划分时，我们需要分离梯度以减少计算量。
- 预热期允许模型在进行任何预测之前进行自我更新（例如，获得比初始值更好的隐藏状态）。
- 梯度裁剪可以防止渐变爆炸，但不能应对梯度消失。

7.5.8 练习

1. 说明独热编码等同于为每个对象选择不同的嵌入。
2. 通过调整超参数（如迭代周期数、隐藏单元数、小批量的时间步数、学习率等）来改善困惑度。
   - 你能降到多低？
   - 用可学习的嵌入替换独热编码。这会带来更好的表现吗？
   - 它在其他书上的效果如何，例如星球大战[99]？
3. 修改预测函数，例如使用采样，而不是选择最有可能的下一个字符。
   - 会发生什么？
   - 使模型偏向更可能的输出，例如，从$q(x_t \mid x_{t-1}, \ldots, x_1) \propto P(x_t \mid x_{t-1}, \ldots, x_1)^\alpha$中抽取$\alpha > 1$。

[99] http://www.gutenberg.org/ebooks/36

4. 在不裁剪梯度的情况下运行本节中的代码。会发生什么事?
5. 更改顺序划分，使其不会从计算图中分离隐藏状态。运行时间有变化吗？困惑度呢?
6. 用ReLU替换本节中使用的激活函数，并重复本节中的实验。我们还需要梯度裁剪吗？为什么?

Discussions[100]

7.6 循环神经网络的简洁实现

虽然 7.5节 对了解循环神经网络是如何实现的很有指导意义，但这并不便捷。本节将展示如何使用深度学习框架的高级API提供的函数更有效地实现相同的语言模型。我们从读取时光机器数据集开始。

```
from mxnet import np, npx
from mxnet.gluon import nn, rnn
from d2l import mxnet as d2l

npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

7.6.1 定义模型

高级API提供了循环神经网络的实现。我们构造了一个具有256隐藏单元的单隐藏层的循环神经网络层rnn_layer。事实上，我们还没有讨论多层的含义——这将在 sec_deep_rnn介绍。现在，只要说多层仅仅相当于一层循环神经网络的输出被用作下一层循环神经网络的输入就足够了。

```
num_hiddens = 256
rnn_layer = rnn.RNN(num_hiddens)
rnn_layer.initialize()
```

初始化隐藏状态很简单。我们调用成员函数begin_state。这将返回一个列表（state），其中包含小批量中每个样本的初始隐藏状态，其形状为(隐藏层数, 批量大小, 隐藏单元数)。对于以后要介绍的一些模型（例如长-短期记忆网络），这样的列表还包含其他信息。

```
state = rnn_layer.begin_state(batch_size=batch_size)
len(state), state[0].shape
```

```
(1, (1, 32, 256))
```

通过一个隐藏状态和一个输入，我们可以用更新后的隐藏状态计算输出。需要强调的是，rnn_layer的“输出”（Y）不涉及输出层的计算：它是指每个时间步的隐藏状态，它们可以用作后续输出层的输入。

[100] https://discuss.d2l.ai/t/2102

此外，rnn_layer返回的更新后的隐藏状态（state_new）是指小批量的最后时间步的隐藏状态。它可以用来初始化顺序分区中一个迭代周期内下一个小批量的隐藏状态。对于多个隐藏层，每个层的隐藏状态将存储在此变量（state_new）中。对于稍后要介绍的某些模型（例如，长-短期记忆），此变量还包含其他信息。

```
X = np.random.uniform(size=(num_steps, batch_size, len(vocab)))
Y, state_new = rnn_layer(X, state)
Y.shape, len(state_new), state_new[0].shape
```

```
((35, 32, 256), 1, (1, 32, 256))
```

与 7.5节 类似，我们为一个完整的循环神经网络模型定义了一个 RNNModel 类。注意 rnn_layer 只包含隐藏循环层，我们需要创建一个单独的输出层。

```
#@save
class RNNModel(nn.Block):
    """循环神经网络模型。"""
    def __init__(self, rnn_layer, vocab_size, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        self.rnn = rnn_layer
        self.vocab_size = vocab_size
        self.dense = nn.Dense(vocab_size)

    def forward(self, inputs, state):
        X = npx.one_hot(inputs.T, self.vocab_size)
        Y, state = self.rnn(X, state)
        # 全连接层首先将`Y`的形状改为(`时间步数` * `批量大小`, `隐藏单元数`)。
        # 它的输出形状是 (`时间步数` * `批量大小`, `词表大小`)。
        output = self.dense(Y.reshape(-1, Y.shape[-1]))
        return output, state

    def begin_state(self, *args, **kwargs):
        return self.rnn.begin_state(*args, **kwargs)
```

7.6.2 训练与预测

在训练模型之前，让我们用一个具有随机权重的模型进行预测。

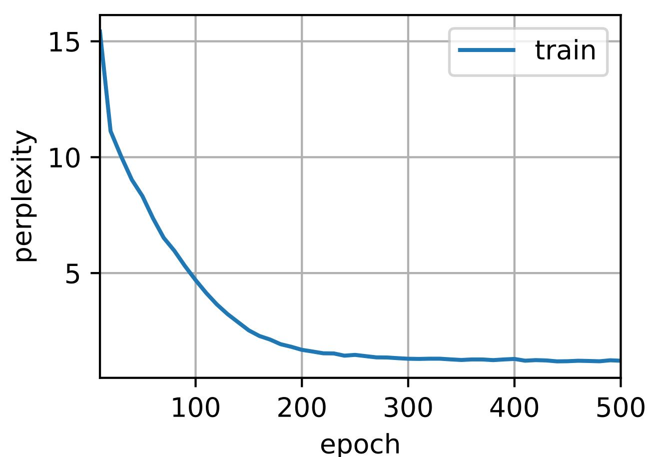
```
device = d2l.try_gpu()
net = RNNModel(rnn_layer, len(vocab))
net.initialize(force_reinit=True, ctx=device)
d2l.predict_ch8('time traveller', 10, net, vocab, device)
```

```
'time travellervmoopwrrrr'
```

很明显，这种模型根本不起作用。接下来，我们使用7.5节中定义的超参数调用train_ch8，并使用高级API训练模型。

```
num_epochs, lr = 500, 1
d2l.train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.2, 139514.8 tokens/sec on gpu(0)
time traveller procebthe fleshthis is so extersome than a savage
traveller came back andfilby s anout an there is the futurd
```



与上一节相比，由于深度学习框架的高级API对代码进行了更多的优化，该模型在较短的时间内实现了类似的困惑度。

7.6.3 小结

- 深度学习框架的高级API提供了循环神经网络层的实现。
- 高级API的循环神经网络层返回输出和更新的隐藏状态，其中输出不涉及输出层计算。
- 与从零开始实现相比，使用高级API会使循环神经网络训练的更快。

7.6.4 练习

1. 你能使用高级API使循环神经网络模型过拟合吗?
2. 如果在循环神经网络模型中增加隐藏层的数量会发生什么?你能使模型工作吗?
3. 使用循环神经网络实现 7.1节 的自回归模型。

Discussions[101]

7.7 通过时间反向传播

到目前为止,我们已经反复提到像梯度爆炸、梯度消失,以及需要对循环神经网络分离梯度。例如,在 7.5节 中,我们在序列上调用了 `detach` 函数。为了能够快速构建模型并了解其工作原理,上面所说的这些都没有得到充分的解释。在本节中,我们将更深入地探讨序列模型反向传播的细节,以及相关数学原理。

当我们首次实现循环神经网络(7.5节)时,我们遇到了梯度爆炸的一些影响。特别是,如果你做了练习题,你会看到梯度裁剪对于确保收敛至关重要。为了更好地理解此问题,本节将回顾如何计算序列模型的梯度。请注意,它的工作原理上没有什么新概念。毕竟,我们仍然只是应用链式法则来计算梯度。尽管如此,还是值得重新思考反向传播(3.7节)。

我们在 3.7节 中描述了多层感知机中的前向和反向传播和计算图。循环神经网络中的前向传播相对简单。通过时间反向传播(Backpropagation through time, BPTT)[Werbos, 1990] 实际上是在循环神经网络中应用的一个特定的反向传播技术。它要求我们将循环神经网络的计算图一次展开一个时间步,以获得模型变量和参数之间的依赖关系。然后,根据链式法则,我们应用反向传播来计算和存储梯度。由于序列可能相当长,因此依赖关系可能相当冗长。例如,对于 1000 个字符的序列,第一个标记可能会对最后位置的标记产生重大影响。这在计算上并不是真正可行的(它需要太长时间,需要太多的内存),并且它需要超过1000个矩阵乘积才能得到非常难以捉摸的梯度。这是一个充满计算与统计不确定性的过程。在下文中,我们将阐明会发生什么情况以及如何在实践中解决这一问题。

7.7.1 循环神经网络的梯度分析

我们从循环神经网络工作原理的简化模型开始。此模型忽略有关隐藏状态的细节及其更新方式的细节。这里的数学表示没有像过去那样明确区分标量,向量和矩阵。因为这些细节对于分析并不重要,只会使本小节中的符号变得混乱。

在此简化模型中,我们将时间步t的隐藏状态表示为h_t,输入表示为x_t,输出表示为o_t。回想一下我们在 7.4.2节 中的讨论,即输入和隐藏状态可以连结为隐藏图层中的一个权重变量。因此,我们分别使用 w_h 和 w_o 来表示隐藏层和输出层的权重。因此,每个时间步的隐藏状态和输出可以写为:

$$\begin{aligned}h_t &= f(x_t, h_{t-1}, w_h),\\ o_t &= g(h_t, w_o),\end{aligned} \tag{7.7.1}$$

[101] https://discuss.d2l.ai/t/2105

其中 f 和 g 分别是隐藏层和输出层的变换。因此，我们有一个链 $\{\ldots,(x_{t-1},h_{t-1},o_{t-1}),(x_t,h_t,o_t),\ldots\}$，它们通过循环计算彼此依赖。正向传播相当简单。一次一个时间步的遍历三元组(x_t,h_t,o_t)。然后通过一个目标函数在所有 T个时间步中评估输出 o_t 和所对应标签 y_t 之间的差异:

$$L(x_1,\ldots,x_T,y_1,\ldots,y_T,w_h,w_o)=\frac{1}{T}\sum_{t=1}^{T}l(y_t,o_t). \tag{7.7.2}$$

对于反向传播，问题有点棘手，特别是当我们计算参数w_h关于目标函数L的梯度时。具体来说，按照链式法则:

$$\begin{aligned}\frac{\partial L}{\partial w_h}&=\frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t,o_t)}{\partial w_h}\\&=\frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t,o_t)}{\partial o_t}\frac{\partial g(h_t,w_o)}{\partial h_t}\frac{\partial h_t}{\partial w_h}.\end{aligned} \tag{7.7.3}$$

(7.7.3) 中乘积的第一项和第二个项很容易计算。第三个项 $\partial h_t/\partial w_h$ 是事情变得棘手的地方，因为我们需要重复计算参数 w_h 对 h_t 的影响。根据 (7.7.1) 中的递归计算，h_t既依赖于h_{t-1}又依赖于w_h，其中h_{t-1}的计算也依赖于w_h。因此，使用链式法则产生:

$$\frac{\partial h_t}{\partial w_h}=\frac{\partial f(x_t,h_{t-1},w_h)}{\partial w_h}+\frac{\partial f(x_t,h_{t-1},w_h)}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial w_h}. \tag{7.7.4}$$

为了得出上述梯度，假设我们有三个序列$\{a_t\},\{b_t\},\{c_t\}$满足$a_0=0$ 且 $a_t=b_t+c_ta_{t-1}$。然后对于 $t\geq 1$，它很容易写出:

$$a_t=b_t+\sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t}c_j\right)b_i. \tag{7.7.5}$$

通过将a_t、b_t和c_t替换:

$$\begin{aligned}a_t&=\frac{\partial h_t}{\partial w_h},\\b_t&=\frac{\partial f(x_t,h_{t-1},w_h)}{\partial w_h},\\c_t&=\frac{\partial f(x_t,h_{t-1},w_h)}{\partial h_{t-1}},\end{aligned} \tag{7.7.6}$$

(7.7.4)中的梯度计算满足$a_t=b_t+c_ta_{t-1}$。因此，对于每个 (7.7.5)，我们可以使用以下式子去掉 (7.7.4) 中的递归计算

$$\frac{\partial h_t}{\partial w_h}=\frac{\partial f(x_t,h_{t-1},w_h)}{\partial w_h}+\sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t}\frac{\partial f(x_j,h_{j-1},w_h)}{\partial h_{j-1}}\right)\frac{\partial f(x_i,h_{i-1},w_h)}{\partial w_h}. \tag{7.7.7}$$

虽然我们可以使用链式法则递归地计算 $\partial h_t/\partial w_h$，但当 t 很大，这个链就会变得很长。让我们讨论处理这一问题的若干方法.

完整计算

显然，我们可以计算 (7.7.7) 中的全部总和。然而，这非常缓慢，梯度可能会爆炸，因为初始条件的微妙变化可能会对结果产生很大影响。也就是说，我们可以看到类似于蝴蝶效应的东西，初始条件的很小变化导致结果的不成比例变化。就我们要估计的模型而言，这实际上是相当不可取的。毕竟，我们正在寻找能够很好地概括出来的可靠估计数。因此，这种方法几乎从未在实践中使用过。

截断时间步

或者，我们可以在 τ 步后截断求和。这就是我们到目前为止一直在讨论的内容，例如当我们在 7.5节 中分离梯度时。这会带来真实梯度的近似，只需将求和终止为$\partial h_{t-\tau}/\partial w_h$。在实践中，这工作得很好。它通常被称为通过时间截断反向传播 [Jaeger, 2002]。这样做的后果之一是，该模型主要侧重于短期影响，而不是长期影响。这实际上是可取的，因为它会将估计值偏向更简单和更稳定的模型。

随机截断

最后，我们可以用一个随机变量替换$\partial h_t/\partial w_h$，该随机变量在预期中是正确的，但是会截断序列。这是通过使用预定义的$0 \leq \pi_t \leq 1$序列ξ_t来实现的，其中$P(\xi_t = 0) = 1 - \pi_t$ 且$P(\xi_t = \pi_t^{-1}) = \pi_t$，因此 $E[\xi_t] = 1$。我们使用它来替换 (7.7.4) 中的梯度 $\partial h_t/\partial w_h$：

$$z_t = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \xi_t \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_h}. \tag{7.7.8}$$

它是从ξ_t的定义推导出来的，那就是$E[z_t] = \partial h_t/\partial w_h$。每当$\xi_t = 0$递归计算在该时间步$t$终止时。这导致不同长度序列的加权和，其中长序列很少但适当地加大权重。这个想法是由塔莱克和奥利维尔 [Tallec & Ollivier, 2017]提出的。

比较策略

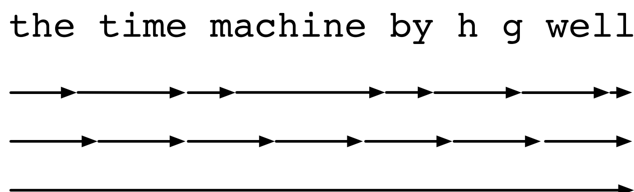


图7.7.1: 比较RNN中计算梯度的策略。自上而下：随机截断、常规截断、完整计算。

图7.7.1 说明了使用循环神经网络的通过时间反向传播的三种策略：

- 第一行是将文本划分为不同长度的段的随机截断。
- 第二行是将文本分解为相同长度的子序列的常规截断。这就是我们在 RNN 实验中一直在做的。
- 第三行是通过时间的完全反向传播，导致计算上不可行的表达式。

遗憾的是，虽然理论上具有吸引力，但随机截断并不比常规截断更好，很可能是由于多种因素。首先，经过一系列反向传播步后的观测结果足以捕获实际依赖关系。其次，增加的方差抵消了步长越多梯度越精确的事实。第三，我们实际上想要只有短范围交互的模型。因此，通过时间的规则截断的反向传播具有轻微的正则化效果。

7.7.2 通过时间反向传播细节

在讨论一般原则之后，让我们详细讨论反向传播问题。与 7.7.1节 中的分析不同，下面我们将展示如何计算目标函数相对于所有分解模型参数的梯度。为了保持简单，我们考虑一个没有偏置参数的循环神经网络，其在隐藏层中的激活函数使用恒等映射（$\phi(x)=x$）。对于时间步t，设单个样本输入和标签分别为$\mathbf{x}_t \in \mathbb{R}^d$和$y_t$。隐藏状态$\mathbf{h}_t \in \mathbb{R}^h$和输出$\mathbf{o}_t \in \mathbb{R}^q$被计算为：

$$\begin{aligned}\mathbf{h}_t &= \mathbf{W}_{hx} \mathbf{x}_t + \mathbf{W}_{hh} \mathbf{h}_{t-1},\\ \mathbf{o}_t &= \mathbf{W}_{qh} \mathbf{h}_{t},\end{aligned} \tag{7.7.9}$$

其中权重参数为 $\mathbf{W}_{hx} \in \mathbb{R}^{h \times d}$、$\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$和 $\mathbf{W}_{qh} \in \mathbb{R}^{q \times h}$。用$l(\mathbf{o}_t, y_t)$表示时间步$t$处的损失。我们的目标函数，从序列开始起的超过$T$个时间步的损失是这样的

$$L = \frac{1}{T} \sum_{t=1}^T l(\mathbf{o}_t, y_t). \tag{7.7.10}$$

为了在循环神经网络计算过程中可视化模型变量和参数之间的依赖关系，我们可以为模型绘制一个计算图，如 图7.7.2 所示。例如，时间步3的隐藏状态$\mathbf{h}_3$的计算取决于模型参数$\mathbf{W}_{hx}$和$\mathbf{W}_{hh}$，以及最终时间步的隐藏状态$\mathbf{h}_2$以及当前时间步的输入$\mathbf{x}_3$。

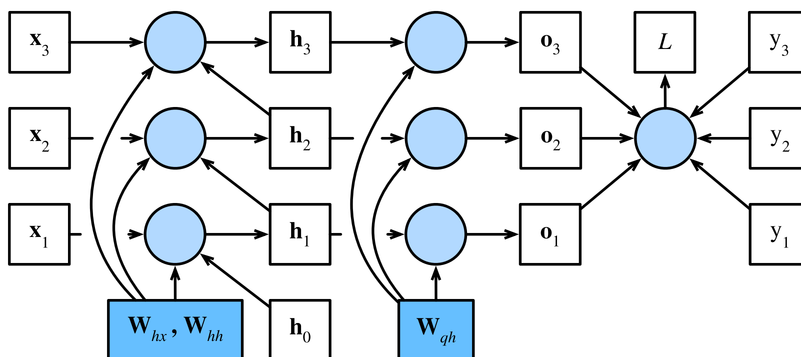


图7.7.2: 显示具有三个时间步的循环神经网络模型依赖关系的计算图。框表示变量（未着色）或参数（着色），圆表示运算符。

正如刚才提到的，图7.7.2 中的模型参数是$\mathbf{W}_{hx}$、$\mathbf{W}_{hh}$和$\mathbf{W}_{qh}$。通常，训练该模型需要对这些参数 $\partial L/\partial \mathbf{W}_{hx}$、$\partial L/\partial \mathbf{W}_{hh}$ 和 $\partial L/\partial \mathbf{W}_{qh}$ 进行梯度计算。根据 图7.7.2 中的依赖关系，我们可以沿箭头的相反方向遍历，依次计算和存储梯度。为了灵活地表示链式法则中不同形状的矩阵、向量和标量的乘法，我们继续使用 prod 运算符，如 3.7节 中所述。

首先，目标函数有关任意时间步t的模型输出的梯度很容易计算：

$$\frac{\partial L}{\partial \mathbf{o}_t} = \frac{\partial l (\mathbf{o}_t, y_t)}{T \cdot \partial \mathbf{o}_t} \in \mathbb{R}^q. \tag{7.7.11}$$

现在，我们可以计算目标函数有关输出层中的参数$\mathbf{W}_{qh}$的梯度：$\partial L/\partial \mathbf{W}_{qh} \in \mathbb{R}^{q\times h}$。根据 图7.7.2 ，目标函数$L$通过$\mathbf{o}_1, \ldots, \mathbf{o}_T$依赖于$\mathbf{W}_{qh}$。依据链式法则，

$$\frac{\partial L}{\partial \mathbf{W}_{qh}} = \sum_{t=1}^T \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{qh}}\right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{o}_t} \mathbf{h}_t^\top, \tag{7.7.12}$$

其中$\partial L/\partial \mathbf{o}_t$是由 (7.7.11) 给出的。

接下来，如 图7.7.2 所示，在最后的时间步T，目标函数L仅通过$\mathbf{o}_T$依赖隐藏状态$\mathbf{h}_T$。因此，我们可以使用链式法则容易地得到梯度$\partial L/\partial \mathbf{h}_T \in \mathbb{R}^h$：

$$\frac{\partial L}{\partial \mathbf{h}_T} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_T}, \frac{\partial \mathbf{o}_T}{\partial \mathbf{h}_T}\right) = \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_T}. \tag{7.7.13}$$

对于任意时间步$t < T$来说都变得更加棘手，其中目标函数L通过$\mathbf{h}_{t+1}$和$\mathbf{o}_t$依赖$\mathbf{h}_t$。根据链式法则，隐藏状态在任何时间步骤$t < T$的梯度$\partial L/\partial \mathbf{h}_t \in \mathbb{R}^h$可以递归地计算为：

$$\frac{\partial L}{\partial \mathbf{h}_t} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_{t+1}}, \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}\right) + \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\right) = \mathbf{W}_{hh}^\top \frac{\partial L}{\partial \mathbf{h}_{t+1}} + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_t}. \tag{7.7.14}$$

为了进行分析，展开任何时间步 $1 \leq t \leq T$ 的递归计算

$$\frac{\partial L}{\partial \mathbf{h}_t}= \sum_{i=t}^T {\left(\mathbf{W}_{hh}^\top\right)}^{T-i} \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{T+t-i}}. \tag{7.7.15}$$

我们可以从 (7.7.15) 中看到，这个简单的线性例子已经展现了长序列模型的一些关键问题：它涉及到 $\mathbf{W}_{hh}^\top$ 的潜在非常大的指数。其中，小于1的特征值消失，大于1的特征值发散。这在数值上是不稳定的，表现为梯度消失或梯度爆炸。解决此问题的一种方法是按照计算方便的大小截断时间步长，如 7.7.1节 中所述。实际上，这种截断是通过在给定数量的时间步长之后分离梯度来实现的。稍后，我们将看到更复杂的序列模型（如长短期记忆）如何进一步缓解这一问题。

最后，图7.7.2 表明了，目标函数L通过隐藏状态$\mathbf{h}_1, \ldots, \mathbf{h}_T$依赖隐藏层中的模型参数$\mathbf{W}_{hx}$和$\mathbf{W}_{hh}$。为了计算有关这些参数的梯度$\partial L / \partial \mathbf{W}_{hx} \in \mathbb{R}^{h \times d}$和$\partial L / \partial \mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$，我们应用链式规则：

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{W}_{hx}}
&= \sum_{t=1}^T \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hx}}\right)
= \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{x}_t^\top,\\
\frac{\partial L}{\partial \mathbf{W}_{hh}}
&= \sum_{t=1}^T \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}\right)
= \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{h}_{t-1}^\top,
\end{aligned} \tag{7.7.16}$$

其中$\partial L/\partial \mathbf{h}_t$是由 (7.7.13) 和 (7.7.14) 递归计算的，是影响数值稳定性的关键量。

正如我们在 3.7节 中所解释的那样，通过时间反向传播是反向传播在循环神经网络中的应用，训练循环神经网络交替使用通过时间前向传播和反向传播。通过时间的反向传播依次计算并存储上述梯度。具体而言，存储的中间值会被重复使用，以避免重复计算，例如存储 $\partial L/\partial \mathbf{h}_t$，以便在计算 $\partial L / \partial \mathbf{W}_{hx}$ 和 $\partial L / \partial \mathbf{W}_{hh}$ 时使用。

7.7.3 小结

- 通过时间反向传播仅仅是反向传播对具有隐藏状态的序列模型的应用。
- 为了计算方便和数值稳定，需要截断，如规则截断和随机截断。
- 矩阵的高次方可能导致特征值发散或消失。这以梯度爆炸或梯度消失的形式表现出来。
- 为了高效计算，在通过时间反向传播期间缓存中间值。

7.7.4 练习

1. 假设我们具有对称矩阵$\mathbf{M} \in \mathbb{R}^{n \times n}$，该对称矩阵具有特征值$\lambda_i$，其对应的特征向量是$\mathbf{v}_i (i = 1, \ldots, n)$。在不丧失泛化性的情况下，假设它们是按顺序$|\lambda_i| \geq |\lambda_{i+1}|$排序的。
   1. 表明$\mathbf{M}^k$具有特征值λ_i^k。
   2. 证明对于随机向量$\mathbf{x} \in \mathbb{R}^n$，有较高概率$\mathbf{M}^k \mathbf{x}$将与$\mathbf{M}$的特征向量$\mathbf{v}_1$对应。
   3. 上述结果对于循环神经网络中的梯度意味着什么？
2. 除了梯度裁剪，你还能想到其他方法来应对循环神经网络中的梯度爆炸吗？

Discussions[102]

[102] https://discuss.d2l.ai/t/2107

Bibliography

[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.

[Bollobas, 1999] Bollobás, B. (1999). *Linear analysis*. Cambridge University Press, Cambridge.

[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.

[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*.

[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).

[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.

[Glorot & Bengio, 2010] Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).

[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).

[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.

[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.

[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).

[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 & Szegedy, 2015] Ioffe, S., & Szegedy, C. (2015). Batch normalization: accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

[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*.

[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*.

[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).

[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.

[Lin et al., 2013] Lin, M., Chen, Q., & Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.

[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., 2017] 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*.

[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*.

[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.

[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.

[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*.

[Peters et al., 2017] Peters, J., Janzing, D., & Schölkopf, B. (2017). *Elements of causal inference: foundations and learning algorithms*. MIT press.

[Petersen et al., 2008] Petersen, K. B., Pedersen, M. S., & others. (2008). The matrix cookbook. *Technical University of Denmark, 7*(15), 510.

[Reed & DeFreitas, 2015] Reed, S., & De Freitas, N. (2015). Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*.

[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.

[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*.

[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., 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*.

[Turing, 1950] Turing, A. (1950). Computing machinery and intelligence. *Mind, 59*(236), 433.

[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).

[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.

[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).

[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*.

[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*.

[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).