

# 深度学习

2017 年 9 月 4 日



# 目录

致谢	xvi
网站	xxii
数学符号	xxiii
<b>第一章 引言</b>	<b>1</b>
1.1 本书面向的读者 . . . . .	10
1.2 深度学习的历史趋势 . . . . .	11
1.2.1 神经网络的众多名称和命运变迁 . . . . .	12
1.2.2 与日俱增的数据量 . . . . .	17
1.2.3 与日俱增的模型规模 . . . . .	19
1.2.4 与日俱增的精度、复杂度和对现实世界的冲击 . . . . .	22
<b>第一部分 应用数学与机器学习基础</b>	<b>25</b>
<b>第二章 线性代数</b>	<b>27</b>
2.1 标量、向量、矩阵和张量 . . . . .	27
2.2 矩阵和向量相乘 . . . . .	29
2.3 单位矩阵和逆矩阵 . . . . .	31
2.4 线性相关和生成子空间 . . . . .	32
2.5 范数 . . . . .	34
2.6 特殊类型的矩阵和向量 . . . . .	36
2.7 特征分解 . . . . .	37

2.8	奇异值分解 . . . . .	39
2.9	Moore-Penrose 伪逆 . . . . .	40
2.10	迹运算 . . . . .	41
2.11	行列式 . . . . .	42
2.12	实例：主成分分析 . . . . .	42
<b>第三章</b>	<b>概率与信息论</b>	<b>47</b>
3.1	为什么要使用概率? . . . . .	47
3.2	随机变量 . . . . .	49
3.3	概率分布 . . . . .	50
3.3.1	离散型变量和概率质量函数 . . . . .	50
3.3.2	连续型变量和概率密度函数 . . . . .	51
3.4	边缘概率 . . . . .	52
3.5	条件概率 . . . . .	52
3.6	条件概率的链式法则 . . . . .	53
3.7	独立性和条件独立性 . . . . .	53
3.8	期望、方差和协方差 . . . . .	54
3.9	常用概率分布 . . . . .	55
3.9.1	Bernoulli 分布 . . . . .	56
3.9.2	Multinoulli 分布 . . . . .	56
3.9.3	高斯分布 . . . . .	57
3.9.4	指数分布和 Laplace 分布 . . . . .	58
3.9.5	Dirac 分布和经验分布 . . . . .	59
3.9.6	分布的混合 . . . . .	59
3.10	常用函数的有用性质 . . . . .	61
3.11	贝叶斯规则 . . . . .	63
3.12	连续型变量的技术细节 . . . . .	64
3.13	信息论 . . . . .	65
3.14	结构化概率模型 . . . . .	69
<b>第四章</b>	<b>数值计算</b>	<b>72</b>
4.1	上溢和下溢 . . . . .	72
4.2	病态条件 . . . . .	73

4.3	基于梯度的优化方法 . . . . .	74
4.3.1	梯度之上: Jacobian 和 Hessian 矩阵 . . . . .	77
4.4	约束优化 . . . . .	82
4.5	实例: 线性最小二乘 . . . . .	85
<b>第五章</b>	<b>机器学习基础</b>	<b>87</b>
5.1	学习算法 . . . . .	87
5.1.1	任务 $T$ . . . . .	88
5.1.2	性能度量 $P$ . . . . .	91
5.1.3	经验 $E$ . . . . .	92
5.1.4	示例: 线性回归 . . . . .	94
5.2	容量、过拟合和欠拟合 . . . . .	97
5.2.1	没有免费午餐定理 . . . . .	102
5.2.2	正则化 . . . . .	104
5.3	超参数和验证集 . . . . .	105
5.3.1	交叉验证 . . . . .	106
5.4	估计、偏差和方差 . . . . .	108
5.4.1	点估计 . . . . .	108
5.4.2	偏差 . . . . .	109
5.4.3	方差和标准差 . . . . .	111
5.4.4	权衡偏差和方差以最小化均方误差 . . . . .	113
5.4.5	一致性 . . . . .	114
5.5	最大似然估计 . . . . .	115
5.5.1	条件对数似然和均方误差 . . . . .	116
5.5.2	最大似然的性质 . . . . .	117
5.6	贝叶斯统计 . . . . .	118
5.6.1	最大后验 (MAP) 估计 . . . . .	121
5.7	监督学习算法 . . . . .	122
5.7.1	概率监督学习 . . . . .	122
5.7.2	支持向量机 . . . . .	123
5.7.3	其他简单的监督学习算法 . . . . .	125
5.8	无监督学习算法 . . . . .	128
5.8.1	主成分分析 . . . . .	128

5.8.2	<i>k</i> -均值聚类	131
5.9	随机梯度下降	132
5.10	构建机器学习算法	133
5.11	促使深度学习发展的挑战	134
5.11.1	维数灾难	135
5.11.2	局部不变性和平滑正则化	135
5.11.3	流形学习	139
<b>第二部分 深度网络：现代实践</b>		<b>143</b>
<b>第六章 深度前馈网络</b>		<b>145</b>
6.1	实例：学习 XOR	148
6.2	基于梯度的学习	152
6.2.1	代价函数	153
6.2.1.1	使用最大似然学习条件分布	154
6.2.1.2	学习条件统计量	155
6.2.2	输出单元	156
6.2.2.1	用于高斯输出分布的线性单元	156
6.2.2.2	用于 Bernoulli 输出分布的 sigmoid 单元	157
6.2.2.3	用于 Multinoulli 输出分布的 softmax 单元	159
6.2.2.4	其他的输出类型	162
6.3	隐藏单元	165
6.3.1	整流线性单元及其扩展	166
6.3.2	logistic sigmoid 与双曲正切函数	168
6.3.3	其他隐藏单元	169
6.4	架构设计	170
6.4.1	万能近似性质和深度	171
6.4.2	其他架构上的考虑	174
6.5	反向传播和其他的微分算法	175
6.5.1	计算图	176
6.5.2	微积分中的链式法则	178
6.5.3	递归地使用链式法则来实现反向传播	179

6.5.4	全连接 MLP 中的反向传播计算 . . . . .	181
6.5.5	符号到符号的导数 . . . . .	182
6.5.6	一般化的反向传播 . . . . .	185
6.5.7	实例：用于 MLP 训练的反向传播 . . . . .	188
6.5.8	复杂化 . . . . .	190
6.5.9	深度学习界以外的微分 . . . . .	191
6.5.10	高阶微分 . . . . .	193
6.6	历史小记 . . . . .	193
<b>第七章</b>	<b>深度学习中的正则化</b>	<b>197</b>
7.1	参数范数惩罚 . . . . .	198
7.1.1	$L^2$ 参数正则化 . . . . .	199
7.1.2	$L^1$ 参数正则化 . . . . .	202
7.2	作为约束的范数惩罚 . . . . .	204
7.3	正则化和欠约束问题 . . . . .	206
7.4	数据集增强 . . . . .	207
7.5	噪声鲁棒性 . . . . .	208
7.5.1	向输出目标注入噪声 . . . . .	209
7.6	半监督学习 . . . . .	209
7.7	多任务学习 . . . . .	210
7.8	提前终止 . . . . .	211
7.9	参数绑定和参数共享 . . . . .	217
7.9.1	卷积神经网络 . . . . .	218
7.10	稀疏表示 . . . . .	218
7.11	Bagging 和其他集成方法 . . . . .	220
7.12	Dropout . . . . .	222
7.13	对抗训练 . . . . .	230
7.14	切面距离、正切传播和流形正切分类器 . . . . .	232
<b>第八章</b>	<b>深度模型中的优化</b>	<b>235</b>
8.1	学习和纯优化有什么不同 . . . . .	235
8.1.1	经验风险最小化 . . . . .	236
8.1.2	代理损失函数和提前终止 . . . . .	237

8.1.3	批量算法和小批量算法 . . . . .	237
8.2	神经网络优化中的挑战 . . . . .	241
8.2.1	病态 . . . . .	242
8.2.2	局部极小值 . . . . .	243
8.2.3	高原、鞍点和其他平坦区域 . . . . .	244
8.2.4	悬崖和梯度爆炸 . . . . .	246
8.2.5	长期依赖 . . . . .	247
8.2.6	非精确梯度 . . . . .	248
8.2.7	局部和全局结构间的弱对应 . . . . .	248
8.2.8	优化的理论限制 . . . . .	250
8.3	基本算法 . . . . .	251
8.3.1	随机梯度下降 . . . . .	251
8.3.2	动量 . . . . .	253
8.3.3	Nesterov 动量 . . . . .	256
8.4	参数初始化策略 . . . . .	256
8.5	自适应学习率算法 . . . . .	261
8.5.1	AdaGrad . . . . .	261
8.5.2	RMSProp . . . . .	262
8.5.3	Adam . . . . .	262
8.5.4	选择正确的优化算法 . . . . .	263
8.6	二阶近似方法 . . . . .	265
8.6.1	牛顿法 . . . . .	266
8.6.2	共轭梯度 . . . . .	267
8.6.3	BFGS . . . . .	270
8.7	优化策略和元算法 . . . . .	271
8.7.1	批标准化 . . . . .	271
8.7.2	坐标下降 . . . . .	274
8.7.3	Polyak 平均 . . . . .	274
8.7.4	监督预训练 . . . . .	275
8.7.5	设计有助于优化的模型 . . . . .	277
8.7.6	延拓法和课程学习 . . . . .	278

<b>第九章</b>	<b>卷积网络</b>	<b>281</b>
9.1	卷积运算	282
9.2	动机	285
9.3	池化	290
9.4	卷积与池化作为一种无限强的先验	295
9.5	基本卷积函数的变体	296
9.6	结构化输出	306
9.7	数据类型	307
9.8	高效的卷积算法	309
9.9	随机或无监督的特征	310
9.10	卷积网络的神经科学基础	311
9.11	卷积网络与深度学习的历史	317
<b>第十章</b>	<b>序列建模：循环和递归网络</b>	<b>319</b>
10.1	展开计算图	320
10.2	循环神经网络	323
10.2.1	导师驱动过程和输出循环网络	326
10.2.2	计算循环神经网络的梯度	328
10.2.3	作为有向图模型的循环网络	330
10.2.4	基于上下文的 RNN 序列建模	334
10.3	双向 RNN	336
10.4	基于编码-解码的序列到序列架构	338
10.5	深度循环网络	340
10.6	递归神经网络	341
10.7	长期依赖的挑战	343
10.8	回声状态网络	345
10.9	渗漏单元和其他多时间尺度的策略	347
10.9.1	时间维度的跳跃连接	347
10.9.2	渗漏单元和一系列不同时间尺度	348
10.9.3	删除连接	348
10.10	长短期记忆和其他门控 RNN	349
10.10.1	LSTM	349
10.10.2	其他门控 RNN	351

10.11 优化长期依赖 . . . . .	352
10.11.1 截断梯度 . . . . .	353
10.11.2 引导信息流的正则化 . . . . .	355
10.12 外显记忆 . . . . .	355
<b>第十一章 实践方法论</b>	<b>359</b>
11.1 性能度量 . . . . .	360
11.2 默认的基准模型 . . . . .	362
11.3 决定是否收集更多数据 . . . . .	363
11.4 选择超参数 . . . . .	364
11.4.1 手动调整超参数 . . . . .	364
11.4.2 自动超参数优化算法 . . . . .	367
11.4.3 网格搜索 . . . . .	368
11.4.4 随机搜索 . . . . .	369
11.4.5 基于模型的超参数优化 . . . . .	370
11.5 调试策略 . . . . .	371
11.6 示例：多位数字识别 . . . . .	374
<b>第十二章 应用</b>	<b>377</b>
12.1 大规模深度学习 . . . . .	377
12.1.1 快速的 CPU 实现 . . . . .	378
12.1.2 GPU 实现 . . . . .	378
12.1.3 大规模的分布式实现 . . . . .	380
12.1.4 模型压缩 . . . . .	381
12.1.5 动态结构 . . . . .	382
12.1.6 深度网络的专用硬件实现 . . . . .	384
12.2 计算机视觉 . . . . .	385
12.2.1 预处理 . . . . .	385
12.2.1.1 对比度归一化 . . . . .	386
12.2.2 数据集增强 . . . . .	389
12.3 语音识别 . . . . .	390
12.4 自然语言处理 . . . . .	392
12.4.1 $n$ -gram . . . . .	392

12.4.2 神经语言模型 . . . . .	394
12.4.3 高维输出 . . . . .	396
12.4.3.1 使用短列表 . . . . .	396
12.4.3.2 分层 Softmax . . . . .	397
12.4.3.3 重要采样 . . . . .	399
12.4.3.4 噪声对比估计和排名损失 . . . . .	401
12.4.4 结合 $n$ -gram 和神经语言模型 . . . . .	401
12.4.5 神经机器翻译 . . . . .	402
12.4.5.1 使用注意力机制并对齐数据片段 . . . . .	403
12.4.6 历史展望 . . . . .	406
12.5 其他应用 . . . . .	407
12.5.1 推荐系统 . . . . .	407
12.5.1.1 探索与利用 . . . . .	409
12.5.2 知识表示、推理和回答 . . . . .	410
12.5.2.1 知识、联系和回答 . . . . .	410
<b>第三部分 深度学习研究</b>	<b>414</b>
<b>第十三章 线性因子模型</b>	<b>417</b>
13.1 概率 PCA 和因子分析 . . . . .	418
13.2 独立成分分析 . . . . .	419
13.3 慢特征分析 . . . . .	421
13.4 稀疏编码 . . . . .	423
13.5 PCA 的流形解释 . . . . .	426
<b>第十四章 自编码器</b>	<b>429</b>
14.1 欠完备自编码器 . . . . .	430
14.2 正则自编码器 . . . . .	431
14.2.1 稀疏自编码器 . . . . .	431
14.2.2 去噪自编码器 . . . . .	433
14.2.3 惩罚导数作为正则 . . . . .	434
14.3 表示能力、层的大小和深度 . . . . .	434
14.4 随机编码器和解码器 . . . . .	435

14.5 去噪自编码器 . . . . .	436
14.5.1 得分估计 . . . . .	437
14.5.2 历史展望 . . . . .	440
14.6 使用自编码器学习流形 . . . . .	440
14.7 收缩自编码器 . . . . .	445
14.8 预测稀疏分解 . . . . .	447
14.9 自编码器的应用 . . . . .	448
<b>第十五章 表示学习</b>	<b>449</b>
15.1 贪心逐层无监督预训练 . . . . .	450
15.1.1 何时以及为何无监督预训练有效? . . . . .	452
15.2 迁移学习和领域自适应 . . . . .	457
15.3 半监督解释因果关系 . . . . .	461
15.4 分布式表示 . . . . .	466
15.5 得益于深度的指数增益 . . . . .	471
15.6 提供发现潜在原因的线索 . . . . .	472
<b>第十六章 深度学习中的结构化概率模型</b>	<b>475</b>
16.1 非结构化建模的挑战 . . . . .	476
16.2 使用图描述模型结构 . . . . .	479
16.2.1 有向模型 . . . . .	480
16.2.2 无向模型 . . . . .	482
16.2.3 配分函数 . . . . .	484
16.2.4 基于能量的模型 . . . . .	485
16.2.5 分离和 d-分离 . . . . .	487
16.2.6 在有向模型和无向模型中转换 . . . . .	490
16.2.7 因子图 . . . . .	493
16.3 从图模型中采样 . . . . .	494
16.4 结构化建模的优势 . . . . .	495
16.5 学习依赖关系 . . . . .	496
16.6 推断和近似推断 . . . . .	497
16.7 结构化概率模型的深度学习方法 . . . . .	498
16.7.1 实例：受限玻尔兹曼机 . . . . .	499

<b>第十七章 蒙特卡罗方法</b>	<b>502</b>
17.1 采样和蒙特卡罗方法 . . . . .	502
17.1.1 为什么需要采样? . . . . .	502
17.1.2 蒙特卡罗采样的基础 . . . . .	503
17.2 重要采样 . . . . .	504
17.3 马尔可夫链蒙特卡罗方法 . . . . .	506
17.4 Gibbs 采样 . . . . .	510
17.5 不同的峰值之间的混合挑战 . . . . .	511
17.5.1 不同峰值之间通过回火来混合 . . . . .	513
17.5.2 深度也许会有助于混合 . . . . .	514
<b>第十八章 直面配分函数</b>	<b>516</b>
18.1 对数似然梯度 . . . . .	516
18.2 随机最大似然和对比散度 . . . . .	518
18.3 伪似然 . . . . .	524
18.4 得分匹配和比率匹配 . . . . .	526
18.5 去噪得分匹配 . . . . .	528
18.6 噪声对比估计 . . . . .	529
18.7 估计配分函数 . . . . .	531
18.7.1 退火重要采样 . . . . .	533
18.7.2 桥式采样 . . . . .	536
<b>第十九章 近似推断</b>	<b>538</b>
19.1 把推断视作优化问题 . . . . .	539
19.2 期望最大化 . . . . .	541
19.3 最大后验推断和稀疏编码 . . . . .	542
19.4 变分推断和变分学习 . . . . .	544
19.4.1 离散型潜变量 . . . . .	545
19.4.2 变分法 . . . . .	551
19.4.3 连续型潜变量 . . . . .	554
19.4.4 学习和推断之间的相互作用 . . . . .	556
19.5 学成近似推断 . . . . .	556
19.5.1 醒眠算法 . . . . .	557

19.5.2 学成推断的其他形式 . . . . .	557
<b>第二十章 深度生成模型</b>	<b>559</b>
20.1 玻尔兹曼机 . . . . .	559
20.2 受限玻尔兹曼机 . . . . .	561
20.2.1 条件分布 . . . . .	562
20.2.2 训练受限玻尔兹曼机 . . . . .	563
20.3 深度信念网络 . . . . .	564
20.4 深度玻尔兹曼机 . . . . .	566
20.4.1 有趣的性质 . . . . .	568
20.4.2 DBM 均匀场推断 . . . . .	569
20.4.3 DBM 的参数学习 . . . . .	571
20.4.4 逐层预训练 . . . . .	572
20.4.5 联合训练深度玻尔兹曼机 . . . . .	574
20.5 实值数据上的玻尔兹曼机 . . . . .	578
20.5.1 Gaussian-Bernoulli RBM . . . . .	578
20.5.2 条件协方差的无向模型 . . . . .	579
20.6 卷积玻尔兹曼机 . . . . .	583
20.7 用于结构化或序列输出的玻尔兹曼机 . . . . .	585
20.8 其他玻尔兹曼机 . . . . .	586
20.9 通过随机操作的反向传播 . . . . .	587
20.9.1 通过离散随机操作的反向传播 . . . . .	588
20.10 有向生成网络 . . . . .	591
20.10.1 sigmoid 信念网络 . . . . .	591
20.10.2 可微生成器网络 . . . . .	592
20.10.3 变分自编码器 . . . . .	594
20.10.4 生成式对抗网络 . . . . .	597
20.10.5 生成矩匹配网络 . . . . .	600
20.10.6 卷积生成网络 . . . . .	601
20.10.7 自回归网络 . . . . .	602
20.10.8 线性自回归网络 . . . . .	602
20.10.9 神经自回归网络 . . . . .	603
20.10.10 NADE . . . . .	604

20.11 从自编码器采样 . . . . .	606
20.11.1 与任意去噪自编码器相关的马尔可夫链 . . . . .	607
20.11.2 夹合与条件采样 . . . . .	607
20.11.3 回退训练过程 . . . . .	608
20.12 生成随机网络 . . . . .	609
20.12.1 判别性 GSN . . . . .	610
20.13 其他生成方案 . . . . .	610
20.14 评估生成模型 . . . . .	611
20.15 结论 . . . . .	613
<b>参考文献</b>	<b>615</b>
<b>术语</b>	<b>679</b>

# 中文版致谢

首先，我们要感谢原作者在本书翻译时给予我们的大力帮助。特别是，原作者和我们分享了书中的原图和参考文献库，这极大节省了我们的时间和精力。

本书涉及的内容博大且思想深刻，如果没有众多同学和网友的帮助，我们不可能顺利完成翻译。

我们才疏学浅而受此重任，深知自身水平难以将本书翻译得很准确。因此我们完成草稿后，将书稿公开于 Github，及早接受网友的批评和建议。以下网友为本书的翻译草稿提供了很多及时的反馈和宝贵的修改意见：@tttwwy @tankeco @fairmiracle @GageGao @huangpingchun @MaHongP @acgtyrant @yanhuibin315 @Buttonwood @titicacafz @weijy026a @RuiZhang1993 @zymiboxpay @xingkongliang @oisc @tielei @yuduowu @Qingmu @HC-2016 @xiaomingabc @bengordai @Bojian @JoyFYan @minoriwww @khty2000 @gump88 @zdx3578 @PassStory @imwebson @wlbksy @roachsinai @Elvinczp @endymecy @9578577 @linzhp @cnscottzheng @germany-zhu @zhangyafeikimi @showgood163 @kangqf @NeutronT @badpoem @kkpoker @Seaball @wheaio @angrymidiao @ZhiweiYang @corenel @zhaoyu611 @SiriusXDJ @dfcv24 @EmisXXY @FlyingFire @vsooda @friskit-china @poerin @ninesunqian @JiaqiYao @Sofring @wenlei @wizyoung @imageslr @indam @XuLYC @zhouqingping @freedomRen @runPenguin @piantou

在此期间，我们四位译者再次进行了校对并且相互之间也校对了一遍。然而仅仅通过我们的校对，实在难以发现翻译中存在的问题。因此，我们邀请一些同学和网友帮助我们校对。经过他们的校对，本书的翻译质量得到了极大的提升。在此我们一一列出，以表示我们由衷的感谢！

- 第一章（引言）：刘畅、许丁杰、潘雨粟和 NeutronT 对本章进行了阅读，并对

很多语句提出了不少修改建议。林中鹏进行了校对，他提出了很多独到的修改建议。

- 第二章（线性代数）：许丁杰和骆徐圣阅读本章，并修改语句。李若愚进行了校对，提出了很多细心的建议。
- 第三章（概率与信息论）：许丁杰阅读本章，并修改语句。李培炎和何翊卓进行了校对，并修改了很多中文用词，使翻译更加准确。
- 第四章（数值计算）：张亚霏阅读本章，并对其他章节也有提出了一些修改建议。张源源进行了校对，并指出了原文可能存在的问题，非常仔细。
- 第五章（机器学习基础）：郭浩和黄平春阅读本章，并修改语句。李东和林中鹏进行了校对。本章篇幅较长，能够有现在的翻译质量离不开这四位的贡献。
- 第六章（深度前馈网络）：周卫林、林中鹏和张远航阅读本章，并提出修改意见。
- 第七章（深度学习中的正则化）：周柏村进行了非常细心的校对，指出了大量问题，令翻译更加准确。
- 第八章（深度模型中的优化）：房晓宇和吴翔阅读本章。黄平春进行了校对，他提出的很多建议让行文更加流畅易懂。
- 第九章（卷积网络）：赵雨和潘雨粟阅读本章，并润色语句。丁志铭进行了非常仔细的校对，并指出很多翻译问题。
- 第十章（序列建模：循环和递归网络）：刘畅阅读本章。赵雨提供了详细的校对建议，尹瑞清根据他的翻译版本，给我们的版本提出了很多建议。虽然仍存在一些分歧，但我们两个版本的整合，让翻译质量提升很多。
- 第十二章（应用）：潘雨粟进行了校对，在他的校对之前，本章阅读起来比较困难。他提供的修改建议，不仅提高了行文流畅度，还提升了译文的准确度。
- 第十三章（线性因子模型）：贺天行阅读本章，修改语句。杨志伟校对本章，润色大量语句。
- 第十四章（自编码器）：李雨慧和黄平春进行了校对。李雨慧提升了语言的流畅度，黄平春纠正了不少错误，提高了准确性。

- 第十五章（表示学习）：cnscottzheng 阅读本章，并修改语句。
- 第十七章（蒙特卡罗方法）：张远航提供了非常细致的校对，后续还校对了一遍，使译文质量大大提升。
- 第十八章（直面配分函数）：吴家楠进行了校对，提升了译文准确性和可读性。
- 第十九章（近似推断）：黄浩军、张远航和张源源进行了校对。这章虽篇幅不大，但内容有深度，译文在三位的帮助下提高了准确度。

所有校对的修改建议都保存在 Github 上，再次感谢以上同学和网友的付出。经过这五个多月的修改，草稿慢慢变成了初稿。尽管还有很多问题，但大部分内容是可读的，并且是准确的。当然目前的翻译仍存在一些没有及时发现的问题，因此翻译也将持续更新，不断修改。我们非常希望读者能到 Github 提建议，并且非常欢迎，无论多么小的修改建议，都是非常宝贵的。

此外，我们还要感谢魏太云学长，他帮助我们与出版社沟通交流，并给予了我们很多排版上的指导。

最后，感谢我们的导师张志华教授，没有老师的 support，我们难以完成翻译。

# 原书致谢

如果没有他人的贡献，这本书将不可能完成。我们感谢为本书提出建议和帮助组织内容结构的人：Guillaume Alain, Kyunghyun Cho, Çağlar Gülcöhre, David Krueger, Hugo Larochelle, Razvan Pascanu and Thomas Rohée。

我们感谢为本书内容提供反馈的人。其中一些人对许多章都给出了建议：Martín Abadi, Guillaume Alain, Ion Androutsopoulos, Fred Bertsch, Olexa Bilaniuk, Ufuk Can Biçici, Matko Bošnjak, John Boersma, Greg Brockman, Alexandre de Brébisson, Pierre Luc Carrier, Sarath Chandar, Paweł Chilinski, Mark Daoust, Oleg Dashevskii, Laurent Dinh, Stephan Dreseitl, Jim Fan, Miao Fan, Meire Fortunato, Frédéric Francis, Nando de Freitas, Çağlar Gülcöhre, Jurgen Van Gael, Javier Alonso García, Jonathan Hunt, Gopi Jeyaram, Chingiz Kabytayev, Lukasz Kaiser, Varun Kanade, Asifullah Khan, Akiel Khan, John King, Diederik P. Kingma, Yann LeCun, Rudolf Mathey, Matías Mattamala, Abhinav Maurya, Kevin Murphy, Oleg Mürk, Roman Novak, Augustus Q. Odena, Simon Pavlik, Karl Pichotta, Eddie Pierce, Kari Pulli, Roussel Rahman, Tapani Raiko, Anurag Ranjan, Johannes Roith, Mihaela Rosca, Halis Sak, César Salgado, Grigory Sapunov, Yoshinori Sasaki, Mike Schuster, Julian Serban, Nir Shabat, Ken Shirriff, Andre Simpelo, Scott Stanley, David Sussillo, Ilya Sutskever, Carles Gelada Sáez, Graham Taylor, Valentin Tolmer, Massimiliano Tomassoli, An Tran, Shubhendu Trivedi, Alexey Umnov, Vincent Vanhoucke, Marco Visentini-Scarzanella, Martin Vita, David Warde-Farley, Dustin Webb, Kelvin Xu, Wei Xue, Ke Yang, Li Yao, Zygmunt Zająć and Ozan Çağlayan.

我们也要感谢对单个章节提供有效反馈的人：

- 数学符号：Zhang Yuanhang.

- 第一章（引言）：Yusuf Akgul, Sébastien Bratieres, Samira Ebrahimi, Charlie Gorichanaz, Brendan Loudermilk, Eric Morris, Cosmin Pârvulescu and Alfredo Solano.
- 第二章（线性代数）：Amjad Almahairi, Nikola Banić, Kevin Bennett, Philippe Castonguay, Oscar Chang, Eric Fosler-Lussier, Andrey Khalyavin, Sergey Ore-shkov, István Petrás, Dennis Prangle, Thomas Rohée, Gitanjali Gulve Sehgal, Colby Toland, Alessandro Vitale and Bob Welland.
- 第三章（概率与信息论）：John Philip Anderson, Kai Arulkumaran, Vincent Dumoulin, Rui Fa, Stephan Gouws, Artem Oboturov, Antti Rasmus, Alexey Surkov and Volker Tresp.
- 第四章（数值计算）：Tran Lam An Ian Fischer and Hu Yuhuang.
- 第五章（机器学习基础）：Dzmitry Bahdanau, Justin Domingue, Nikhil Garg, Makoto Otsuka, Bob Pepin, Philip Popien, Emmanuel Rayner, Peter Shepard, Kee-Bong Song, Zheng Sun and Andy Wu.
- 第六章（深度前馈网络）：Uriel Berdugo, Fabrizio Bottarel, Elizabeth Burl, Ishan Durugkar, Jeff Hlywa, Jong Wook Kim, David Krueger and Aditya Kumar Praharaj.
- 第七章（深度学习中的正则化）：Morten Kolbæk, Kshitij Lauria, Inkyu Lee, Sunil Mohan, Hai Phong Phan and Joshua Salisbury.
- 第八章（深度模型中的优化）：Marcel Ackermann, Peter Armitage, Rowel Atienza, Andrew Brock, Tegan Maharaj, James Martens, Kashif Rasul, Klaus Strobl and Nicholas Turner.
- 第九章（卷积网络）：Martín Arjovsky, Eugene Brevdo, Konstantin Divilov, Eric Jensen, Mehdi Mirza, Alex Paino, Marjorie Sayer, Ryan Stout and Wentao Wu.
- 第十章（序列建模：循环和递归网络）：Gökçen Eraslan, Steven Hickson, Razvan Pascanu, Lorenzo von Ritter, Rui Rodrigues, Dmitriy Serdyuk, Dongyu Shi and Kaiyu Yang.
- 第十一章（实践方法论）：Daniel Beckstein.

- 第十二章 (应用): George Dahl, Vladimir Nekrasov and Ribana Roscher.
- 第十三章 (线性因子模型): Jayanth Koushik.
- 第十五章 (表示学习): Kunal Ghosh.
- 第十六章 (深度学习中的结构化概率模型): Minh Lê and Anton Varfolom.
- 第十八章 (直面配分函数): Sam Bowman.
- 第十九章 (近似推断): Yujia Bao.
- 第二十章 (深度生成模型): Nicolas Chapados, Daniel Galvez, Wenming Ma, Fady Medhat, Shakir Mohamed and Grégoire Montavon.
- 参考文献: Lukas Michelbacher and Leslie N. Smith.

我们还要感谢那些允许我们从他们的出版物中复制图片、数据的人。我们在图片标题的文字中注明了他们的贡献。

我们还要感谢 Lu Wang 为我们写了 pdf2htmlEX，我们用它来制作这本书的网页版本，Lu Wang 还帮助我们改进了生成的 HTML 的质量。

我们还要感谢 Ian 的妻子 Daniela Flori Goodfellow，在 Ian 的写作过程中的耐心支持和检查。

我们还要感谢 Google Brain 团队提供了学术环境，从而使得 Ian 能够花费大量时间写作此书并接受同行的反馈和指导。我们特别感谢 Ian 的前任经理 Greg Corrado 和他的现任经理 Samy Bengio 对这个项目的支持。最后我们还要感谢 Geoffrey Hinton 在写作困难时的鼓励。

# 网站

[www.deeplearningbook.org](http://www.deeplearningbook.org)

这本书伴随有上述网站。网站提供了各种补充材料，包括练习、讲义幻灯片、错误更正以及其他应该对读者和讲师有用的资源。

# 数学符号

本节简要介绍本书所使用的数学符号。我们在第二章至第四章中描述大多数数学概念，如果你不熟悉任何相应的数学概念，可以参考对应的章节。

## 数和数组

$a$	标量 (整数或实数)
$\mathbf{a}$	向量
$\mathbf{A}$	矩阵
$\mathbf{A}$	张量
$\mathbf{I}_n$	$n$ 行 $n$ 列的单位矩阵
$\mathbf{I}$	维度蕴含于上下文的单位矩阵
$\mathbf{e}^{(i)}$	标准基向量 $[0, \dots, 0, 1, 0, \dots, 0]$ , 其中索引 $i$ 处值为 1
$\text{diag}(\mathbf{a})$	对角方阵, 其中对角元素由 $\mathbf{a}$ 给定
$\mathbf{a}$	标量随机变量
$\mathbf{a}$	向量随机变量
$\mathbf{A}$	矩阵随机变量

## 集合和图

$\mathbb{A}$	集合
$\mathbb{R}$	实数集
$\{0, 1\}$	包含 0 和 1 的集合
$\{0, 1, \dots, n\}$	包含 0 和 $n$ 之间所有整数的集合
$[a, b]$	包含 $a$ 和 $b$ 的实数区间
$(a, b]$	不包含 $a$ 但包含 $b$ 的实数区间
$\mathbb{A} \setminus \mathbb{B}$	差集, 即其元素包含于 $\mathbb{A}$ 但不包含于 $\mathbb{B}$
$\mathcal{G}$	图
$Pa_{\mathcal{G}}(\mathbf{x}_i)$	图 $\mathcal{G}$ 中 $\mathbf{x}_i$ 的父节点

## 索引

$a_i$	向量 $\mathbf{a}$ 的第 $i$ 个元素, 其中索引从 1 开始
$a_{-i}$	除了第 $i$ 个元素, $\mathbf{a}$ 的所有元素
$A_{i,j}$	矩阵 $\mathbf{A}$ 的 $i, j$ 元素
$\mathbf{A}_{:, i}$	矩阵 $\mathbf{A}$ 的第 $i$ 行
$\mathbf{A}_{:, :, i}$	矩阵 $\mathbf{A}$ 的第 $i$ 列
$A_{i,j,k}$	3 维张量 $\mathbf{A}$ 的 $(i, j, k)$ 元素
$\mathbf{A}_{:, :, :, i}$	3 维张量的 2 维切片
$\mathbf{a}_i$	随机向量 $\mathbf{a}$ 的第 $i$ 个元素

## 线性代数中的操作

 $\mathbf{A}^\top$  矩阵  $\mathbf{A}$  的转置 $\mathbf{A}^+$   $\mathbf{A}$  的Moore-Penrose 伪逆 $\mathbf{A} \odot \mathbf{B}$   $\mathbf{A}$  和  $\mathbf{B}$  的逐元素乘积 (Hadamard 乘积) $\det(\mathbf{A})$   $\mathbf{A}$  的行列式

## 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 $\nabla_{\mathbf{x}}y$   $y$  关于  $\mathbf{x}$  的梯度 $\nabla_{\mathbf{X}}y$   $y$  关于  $\mathbf{X}$  的矩阵导数 $\nabla_{\mathbf{X}}y$   $y$  关于  $\mathbf{X}$  求导后的张量 $\frac{\partial f}{\partial \mathbf{x}}$   $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$  的 Jacobian 矩阵  $\mathbf{J} \in \mathbb{R}^{m \times n}$  $\nabla_x^2 f(\mathbf{x})$  or  $\mathbf{H}(f)(\mathbf{x})$   $f$  在点  $\mathbf{x}$  处的 Hessian 矩阵 $\int f(\mathbf{x}) d\mathbf{x}$   $\mathbf{x}$  整个域上的定积分 $\int_{\mathbb{S}} f(\mathbf{x}) d\mathbf{x}$  集合  $\mathbb{S}$  上关于  $\mathbf{x}$  的定积分

### 概率和信息论

$a \perp b$	$a$ 和 $b$ 相互独立的随机变量
$a \perp b   c$	给定 $c$ 后条件独立
$P(a)$	离散变量上的概率分布
$p(a)$	连续变量（或变量类型未指定时）上的概率分布
$a \sim P$	具有分布 $P$ 的随机变量 $a$
$\mathbb{E}_{x \sim P}[f(x)]$ or $\mathbb{E}f(x)$	$f(x)$ 关于 $P(x)$ 的期望
$\text{Var}(f(x))$	$f(x)$ 在分布 $P(x)$ 下的方差
$\text{Cov}(f(x), g(x))$	$f(x)$ 和 $g(x)$ 在分布 $P(x)$ 下的协方差
$H(x)$	随机变量 $x$ 的香农熵
$D_{\text{KL}}(P \  Q)$	$P$ 和 $Q$ 的KL 散度
$\mathcal{N}(x; \mu, \Sigma)$	均值为 $\mu$ 协方差为 $\Sigma$ , $x$ 上的高斯分布

## 函数

$f : \mathbb{A} \rightarrow \mathbb{B}$	定义域为 $\mathbb{A}$ 值域为 $\mathbb{B}$ 的函数 $f$
$f \circ g$	$f$ 和 $g$ 的组合
$f(\mathbf{x}; \boldsymbol{\theta})$	由 $\boldsymbol{\theta}$ 参数化, 关于 $\mathbf{x}$ 的函数 (有时为简化表示, 我们忽略 $\boldsymbol{\theta}$ 记为 $f(\mathbf{x})$ )
$\log x$	$x$ 的自然对数
$\sigma(x)$	Logistic sigmoid, $\frac{1}{1 + \exp(-x)}$
$\zeta(x)$	Softplus, $\log(1 + \exp(x))$
$\ \mathbf{x}\ _p$	$\mathbf{x}$ 的 $L^p$ 范数
$\ \mathbf{x}\ $	$\mathbf{x}$ 的 $L^2$ 范数
$x^+$	$x$ 的正数部分, 即 $\max(0, x)$
$\mathbf{1}_{\text{condition}}$	如果条件为真则为 1, 否则为 0

有时候我们使用函数  $f$ , 它的参数是一个标量, 但应用到一个向量、矩阵或张量:  $f(\mathbf{x})$ ,  $f(\mathbf{X})$ , or  $f(\mathbf{X})$ 。这表示逐元素地将  $f$  应用于数组。例如,  $\mathbf{C} = \sigma(\mathbf{X})$ , 则对于所有合法的  $i$ 、 $j$  和  $k$ ,  $C_{i,j,k} = \sigma(X_{i,j,k})$ 。

## 数据集和分布

$p_{\text{data}}$	数据生成分布
$\hat{p}_{\text{train}}$	由训练集定义的经验分布
$\mathbb{X}$	训练样本的集合
$\mathbf{x}^{(i)}$	数据集的第 $i$ 个样本 (输入)
$y^{(i)}$ or $\mathbf{y}^{(i)}$	监督学习中与 $\mathbf{x}^{(i)}$ 关联的目标
$\mathbf{X}$	$m \times n$ 的矩阵, 其中行 $\mathbf{X}_{i,:}$ 为输入样本 $\mathbf{x}^{(i)}$



# 第一章 引言

远在古希腊时期，发明家就梦想着创造能自主思考的机器。神话人物皮格马利翁 (Pygmalion)、代达罗斯 (Daedalus) 和赫淮斯托斯 (Hephaestus) 可以被看作传说中的发明家，而加拉蒂亚 (Galatea)、塔洛斯 (Talos) 和潘多拉 (Pandora) 则可以被视为人造生命 (Ovid and Martin, 2004; Sparkes, 1996; Tandy, 1997)。

当人类第一次构思可编程计算机时，就已经在思考计算机能否变得智能（尽管这距造出第一台计算机还有一百多年）(Lovelace, 1842)。如今，**人工智能** (artificial intelligence, AI) 已经成为一个具有众多实际应用和活跃研究课题的领域，并且正在蓬勃发展。我们期望通过智能软件自动地处理常规劳动、理解语音或图像、帮助医学诊断和支持基础科学的研究。

在人工智能的早期，那些对人类智力来说非常困难、但对计算机来说相对简单的问题得到迅速解决，比如，那些可以通过一系列形式化的数学规则来描述的问题。人工智能的真正挑战在于解决那些对人来说很容易执行、但很难形式化描述的任务，如识别人们所说的话或图像中的脸。对于这些问题，我们人类往往可以凭借直觉轻易地解决。

针对这些比较直观的问题，本书讨论一种解决方案。该方案可以让计算机从经验中学习，并根据层次化的概念体系来理解世界，而每个概念则通过与某些相对简单的概念之间的关系来定义。让计算机从经验获取知识，可以避免由人类来给计算机形式化地指定它需要的所有知识。层次化的概念让计算机构建较简单的概念来学习复杂概念。如果绘制出这些概念如何建立在彼此之上的图，我们将得到一张“深”（层次很多）的图。基于这个原因，我们称这种方法为 **AI 深度学习** (deep learning)。

AI 许多早期的成功发生在相对朴素且形式化的环境中，而且不要求计算机具备很多关于世界的知识。例如，IBM 的深蓝 (Deep Blue) 国际象棋系统在 1997 年

击败了世界冠军Garry Kasparov(Hsu, 2002)。显然国际象棋是一个非常简单的领域，因为它仅含有 64 个位置并只能以严格限制的方式移动 32 个棋子。设计一种成功的国际象棋策略是巨大的成就，但向计算机描述棋子及其允许的走法并不是挑战的困难所在。国际象棋完全可以由一个非常简短的、完全形式化的规则列表来描述，并可以容易地由程序员事先准备好。

讽刺的是，抽象和形式化的任务对人类而言是最困难的脑力任务之一，但对计算机而言却属于最容易的。计算机早就能够打败人类最好的象棋选手，但直到最近计算机才在识别对象或语音任务中达到人类平均水平。一个人的日常生活需要关于世界的巨量知识。很多这方面的知识是主观的、直观的，因此很难通过形式化的方式表达清楚。计算机需要获取同样的知识才能表现出智能。人工智能的一个关键挑战就是如何将这些非形式化的知识传达给计算机。

一些人工智能项目力求将关于世界的知识用形式化的语言进行硬编码 (hard-code)。计算机可以使用逻辑推理规则来自动地理解这些形式化语言中的声明。这就是众所周知的人工智能的 **知识库** (knowledge base) 方法。然而，这些项目最终都没有取得重大的成功。其中最著名的项目是 Cyc (Lenat and Guha, 1989)。Cyc 包括一个推断引擎和一个使用 CycL 语言描述的声明数据库。这些声明是由人类监督者输入的。这是一个笨拙的过程。人们设法设计出足够复杂的形式化规则来精确地描述世界。例如，Cyc 不能理解一个关于名为 Fred 的人在早上剃须的故事 (Linde, 1992)。它的推理引擎检测到故事中的不一致性：它知道人体的构成不包含电气零件，但由于 Fred 正拿着一个电动剃须刀，它认为实体“正在剃须的 Fred”(“FredWhileShaving”) 含有电气部件。因此它产生了这样的疑问——Fred 在刮胡子的时候是否仍然是一个人。

依靠硬编码的知识体系面对的困难表明，AI 系统需要具备自己获取知识的能力，即从原始数据中提取模式的能力。这种能力被称为 **机器学习** (machine learning)。引入机器学习使计算机能够解决涉及现实世界知识的问题，并能作出看似主观的决策。比如，一个被称为 **逻辑回归** (logistic regression) 的简单机器学习算法可以决定是否建议剖腹产 (Mor-Yosef *et al.*, 1990)。而同样是简单机器学习算法的 **朴素贝叶斯** (naive Bayes) 则可以区分垃圾电子邮件和合法电子邮件。

这些简单的机器学习算法的性能在很大程度上依赖于给定数据的 **表示** (representation)。例如，当逻辑回归被用于判断产妇是否适合剖腹产时，AI 系统不会直接检查患者。相反，医生需要告诉系统几条相关的信息，诸如是否存在子宫疤痕。表示患者的每条信息被称为一个特征。逻辑回归学习病人的这些特征如何与各种结果

相关联。然而，它丝毫不能影响该特征定义的方式。如果将病人的 MRI 扫描作为逻辑回归的输入，而不是医生正式的报告，它将无法作出有用的预测。MRI 扫描的单一像素与分娩过程中并发症之间的相关性微乎其微。

在整个计算机科学乃至日常生活中，对表示的依赖都是一个普遍现象。在计算机科学中，如果数据集合被精巧地结构化并被智能地索引，那么诸如搜索之类的操作的处理速度就可以成指数级地加快。人们可以很容易地在阿拉伯数字的表示下进行算术运算，但在罗马数字的表示下运算会比较耗时。因此，毫不奇怪，表示的选择会对机器学习算法的性能产生巨大的影响。图 1.1 展示了一个简单的可视化例子。

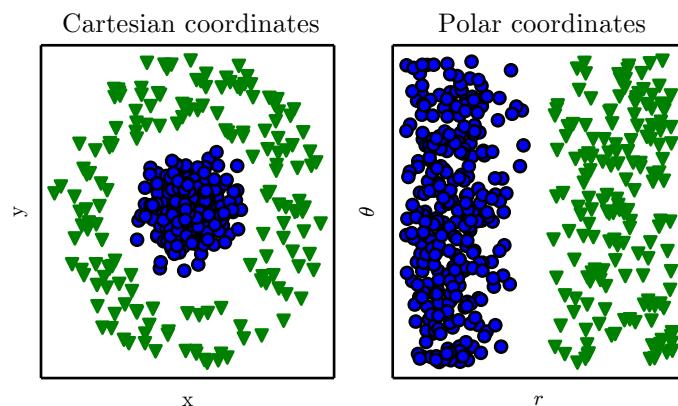


图 1.1：不同表示的例子：假设我们想在散点图中画一条线来分隔两类数据。在左图，我们使用笛卡尔坐标表示数据，这个任务是不可能的。右图中，我们用极坐标表示数据，可以用垂直线简单地解决这个任务。（与 David Warde-Farley 合作画出此图。）

许多人工智能任务都可以通过以下方式解决：先提取一个合适的特征集，然后将这些特征提供给简单的机器学习算法。例如，对于通过声音鉴别说话者的任务来说，一个有用的特征是对其声道大小的估计。这个特征为判断说话者是男性、女性还是儿童提供了有力线索。

然而，对于许多任务来说，我们很难知道应该提取哪些特征。例如，假设我们想编写一个程序来检测照片中的车。我们知道，汽车有轮子，所以我们可能会想用车轮的存在与否作为特征。不幸的是，我们难以准确地根据像素值来描述车轮看上去像什么。虽然车轮具有简单的几何形状，但它的图像可能会因场景而异，如落在车轮上的阴影、太阳照亮的车轮的金属零件、汽车的挡泥板或者遮挡的车轮一部分的前景物体等等。

解决这个问题的途径之一是使用机器学习来发掘表示本身，而不仅仅把表示映射到输出。这种方法我们称之为**表示学习**（representation learning）。学到的表示往往比手动设计的表示表现得更好。并且它们只需最少的人工干预，就能让AI系统迅速适应新的任务。表示学习算法只需几分钟就可以为简单的任务发现一个很好的特征集，对于复杂任务则需要几小时到几个月。手动为一个复杂的任务设计特征需要耗费大量的人工时间和精力；甚至需要花费整个社群研究人员几十年的时间。

表示学习算法的典型例子是**自编码器**（autoencoder）。自编码器由一个**编码器**（encoder）函数和一个**解码器**（decoder）函数组合而成。编码器函数将输入数据转换为一种不同的表示，而解码器函数则将这个新的表示转换到原来的形式。我们期望当输入数据经过编码器和解码器之后尽可能多地保留信息，同时希望新的表示有各种好的特性，这也是自编码器的训练目标。为了实现不同的特性，我们可以设计不同形式的自编码器。

当设计特征或设计用于学习特征的算法时，我们的目标通常是分离出能解释观察数据的**变差因素**（factors of variation）。在此背景下，“因素”这个词仅指代影响的不同来源；因素通常不是乘性组合。这些因素通常是不能被直接观察到的量。相反，它们可能是现实世界中观察不到的物体或者不可观测的力，但会影响可观测的量。为了对观察到的数据提供有用的简化解释或推断其原因，它们还可能以概念的形式存在于人类的思维中。它们可以被看作数据的概念或者抽象，帮助我们了解这些数据的丰富多样性。当分析语音记录时，变差因素包括说话者的年龄、性别、他们的口音和他们正在说的词语。当分析汽车的图像时，变差因素包括汽车的位置、它的颜色、太阳的角度和亮度。

在许多现实的人工智能应用中，困难主要源于多个变差因素同时影响着我们能够观察到的每一个数据。比如，在一张包含红色汽车的图片中，其单个像素在夜间可能会非常接近黑色。汽车轮廓的形状取决于视角。大多数应用需要我们理清变差因素并忽略我们不关心的因素。

显然，从原始数据中提取如此高层次、抽象的特征是非常困难的。许多诸如说话口音这样的变差因素，只能通过对数据进行复杂的、接近人类水平的理解来辨识。这几乎与获得原问题的表示一样困难，因此，乍一看，表示学习似乎并不能帮助我们。

**深度学习**（deep learning）通过其他较简单的表示来表达复杂表示，解决了表示学习中的核心问题。

深度学习让计算机通过较简单概念构建复杂的概念。图1.2展示了深度学习系统

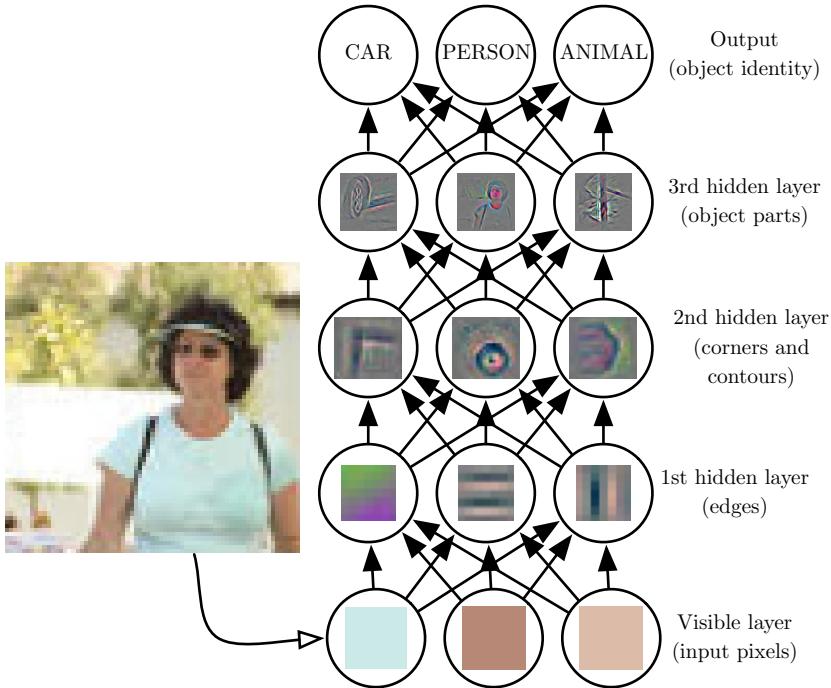


图 1.2: 深度学习模型的示意图。计算机难以理解原始感观输入数据的含义，如表示为像素值集合的图像。将一组像素映射到对象标识的函数非常复杂。如果直接处理，学习或评估此映射似乎是不可能的。深度学习将所需的复杂映射分解为一系列嵌套的简单映射（每个由模型的不同层描述）来解决这一难题。输入展示在可见层（visible layer），这样命名的原因是因为它包含我们能观察到的变量。然后是一系列从图像中提取越来越多抽象特征的隐藏层（hidden layer）。因为它们的值不在数据中给出，所以将这些层称为“隐藏”；模型必须确定哪些概念有利于解释观察数据中的关系。这里的图像是每个隐藏单元表示的特征的可视化。给定像素，第一层可以轻易地通过比较相邻像素的亮度来识别边缘。有了第一隐藏层描述的边缘，第二隐藏层可以容易地搜索可识别为角和扩展轮廓的边集合。给定第二隐藏层中关于角和轮廓的图像描述，第三隐藏层可以找到轮廓和角的特定集合来检测特定对象的整个部分。最后，根据图像描述中包含的对象部分，可以识别图像中存在的对象。经 Zeiler and Fergus (2014) 许可转载此图。

如何通过组合较简单的概念（例如转角和轮廓，它们转而由边线定义）来表示图像中人的概念。深度学习模型的典型例子是前馈深度网络或**多层感知机**（multilayer perceptron, MLP）。多层感知机仅仅是一个将一组输入值映射到输出值的数学函数。该函数由许多较简单的函数复合而成。我们可以认为不同数学函数的每一次应用都为输入提供了新的表示。

学习数据的正确表示的想法是解释深度学习的一个视角。另一个视角是深度促使计算机学习一个多步骤的计算机程序。每一层表示都可以被认为是并行执行另一组指令之后计算机的存储器状态。更深的网络可以按顺序执行更多的指令。顺序指令提供了极大的能力，因为后面的指令可以参考早期指令的结果。从这个角度上看，在某层激活函数里，并非所有信息都蕴涵着解释输入的变差因素。表示还存储着状态信息，用于帮助程序理解输入。这里的状态信息类似于传统计算机程序中的计数器或指针。它与具体的输入内容无关，但有助于模型组织其处理过程。

目前主要有两种度量模型深度的方式。第一种方式是基于评估架构所需执行的顺序指令的数目。假设我们将模型表示为给定输入后，计算对应输出的流程图，则可以将这张流程图中的最长路径视为模型的深度。正如两个使用不同语言编写的等价程序将具有不同的长度；相同的函数可以被绘制为具有不同深度的流程图，其深度取决于我们可以用来作为一个步骤的函数。图 1.3 说明了语言的选择如何给相同的架构两个不同的衡量。

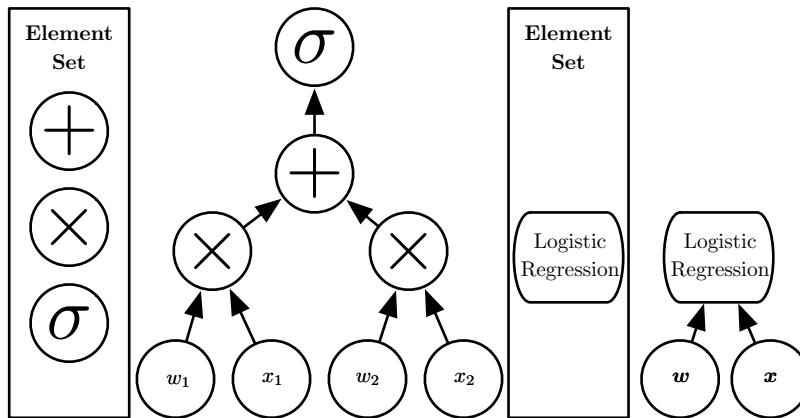


图 1.3: 将输入映射到输出的计算图表的示意图，其中每个节点执行一个操作。深度是从输入到输出的最长路径的长度，但这取决于可能的计算步骤的定义。这些图中所示的计算是逻辑回归模型的输出， $\sigma(\mathbf{w}^T \mathbf{x})$ ，其中  $\sigma$  是 logistic sigmoid 函数。如果我们使用加法、乘法和 logistic sigmoid 作为我们计算机语言的元素，那么这个模型深度为三。如果我们将逻辑回归视为元素本身，那么这个模型深度为一。

另一种是在深度概率模型中使用的方法，它不是将计算图的深度视为模型深度，而是将描述概念彼此如何关联的图的深度视为模型深度。在这种情况下，计算每个概念表示的计算流程图的深度可能比概念本身的图更深。这是因为系统对较简单概念的理解在给出更复杂概念的信息后可以进一步精细化。例如，一个 AI 系统观察其

中一只眼睛在阴影中的脸部图像时，它最初可能只看到一只眼睛。但当检测到脸部的存在后，系统可以推断第二只眼睛也可能是存在的。在这种情况下，概念的图仅包括两层（关于眼睛的层和关于脸的层），但如果我们将每个概念的估计将需要额外的  $n$  次计算，即计算的图将包含  $2n$  层。

由于并不总是清楚计算图的深度或概率模型图的深度哪一个是最重要的，并且由于不同的人选择不同的最小元素集来构建相应的图，因此就像计算机程序的长度不存在单一的正确值一样，架构的深度也不存在单一的正确值。另外，也不存在模型多么深才能被修饰为“深”的共识。但相比传统机器学习，深度学习研究的模型涉及更多学到功能或学到概念的组合，这点毋庸置疑。

总之，这本书的主题——深度学习是通向人工智能的途径之一。具体来说，它是机器学习的一种，一种能够使计算机系统从经验和数据中得到提高的技术。我们坚信机器学习可以构建出在复杂实际环境下运行的 AI 系统，并且是唯一切实可行的方法。深度学习是一种特定类型的机器学习，具有强大的能力和灵活性，它将大千世界表示为嵌套的层次概念体系（由较简单概念间的联系定义复杂概念、从一般抽象概括到高级抽象表示）。图 1.4 说明了这些不同的 AI 学科之间的关系。图 1.5 展示了每个学科如何工作的高层次原理。

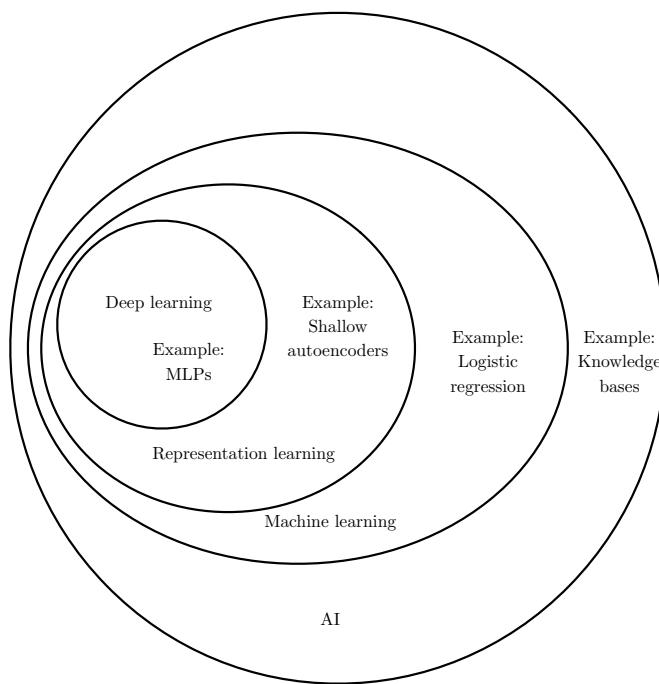


图 1.4: 维恩图展示了深度学习是一种表示学习, 也是一种机器学习, 可以用于许多 (但不是全部) AI 方法。维恩图的每个部分包括一个 AI 技术的示例。

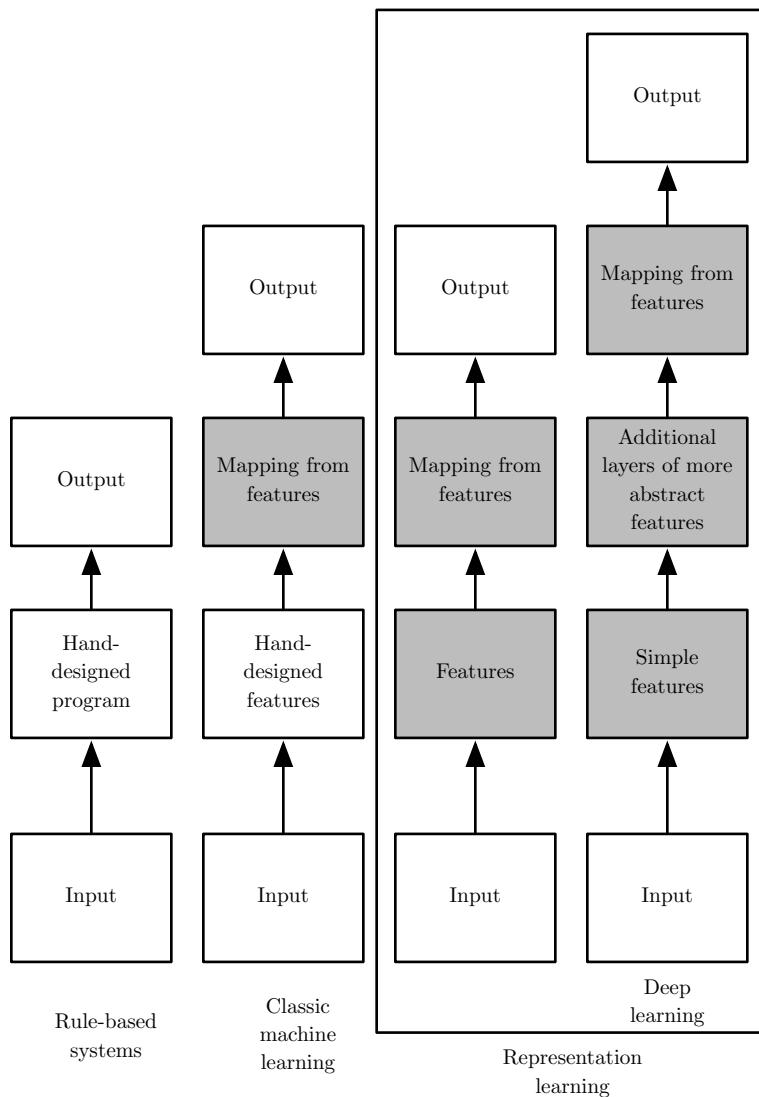


图 1.5: 流程图展示了 AI 系统的不同部分如何在不同的 AI 学科中彼此相关。阴影框表示能从数据中学习的组件。

## 1.1 本书面向的读者

这本书对各类读者都有一定用处，但我们主要是为两类受众对象而写的。其中一类受众对象是学习机器学习的大学生（本科或研究生），包括那些已经开始职业生涯的深度学习和人工智能研究者。另一类受众对象是没有机器学习或统计背景但希望能快速地掌握这方面知识并在他们的产品或平台中使用深度学习的软件工程师。深度学习在许多软件领域都已被证明是有用的，包括计算机视觉、语音和音频处理、自然语言处理、机器人技术、生物信息学和化学、电子游戏、搜索引擎、网络广告和金融。

为了最好地服务各类读者，我们将本书组织为三个部分。第一部分介绍基本的数学工具和机器学习的概念。第二部分介绍最成熟的深度学习算法，这些技术基本上已经得到解决。第三部分讨论某些具有展望性的想法，它们被广泛地认为是深度学习未来的研究重点。

读者可以随意跳过不感兴趣或与自己背景不相关的部分。熟悉线性代数、概率和基本机器学习概念的读者可以跳过第一部分，例如，当读者只是想实现一个能工作的系统则不需要阅读超出第二部分的内容。为了帮助读者选择章节，图 1.6 展示了这本书的高层组织结构的流程图。

我们假设所有读者都具备计算机科学背景。也假设读者熟悉编程，并且对计算的性能问题、复杂性理论、入门级微积分和一些图论术语有基本的了解。

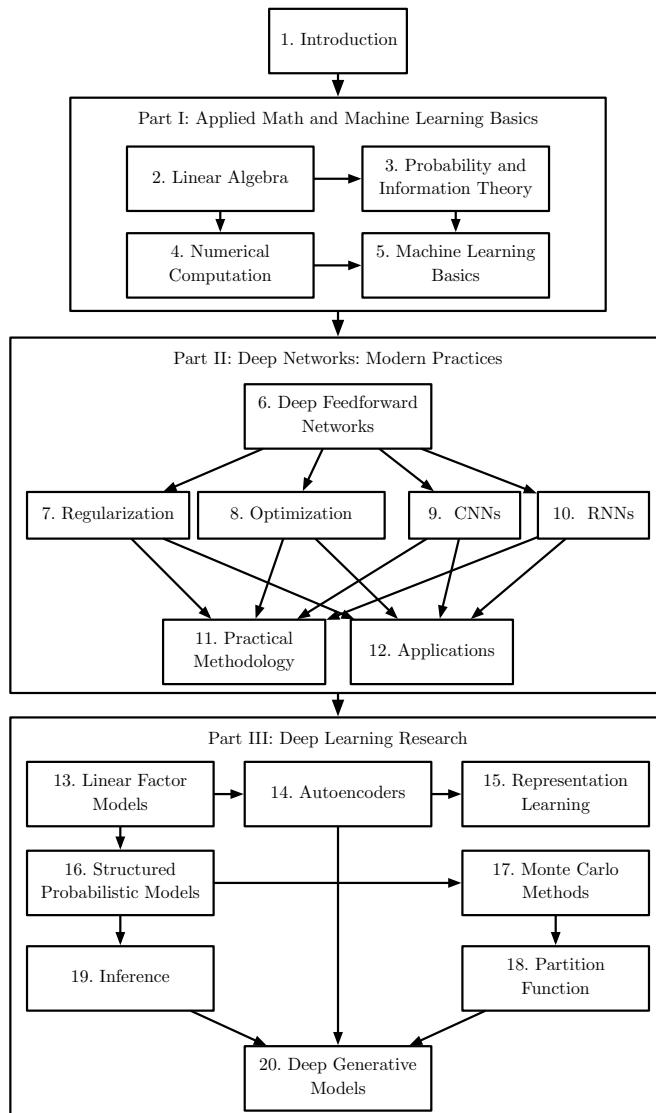


图 1.6: 本书的高层组织。从一章到另一章的箭头表示前一章是理解后一章的必备内容。

## 1.2 深度学习的历史趋势

通过历史背景了解深度学习是最简单的方式。这里我们仅指出深度学习的几个关键趋势，而不是提供其详细的历史：

- 深度学习有着悠久而丰富的历史，但随着许多不同哲学观点的渐渐消逝，与之对应的名称也渐渐尘封。
- 随着可用的训练数据量不断增加，深度学习变得更加有用。
- 随着时间的推移，针对深度学习的计算机软硬件基础设施都有所改善，深度学习模型的规模也随之增长。
- 随着时间的推移，深度学习已经解决日益复杂的应用，并且精度不断提高。

### 1.2.1 神经网络的众多名称和命运变迁

我们期待这本书的许多读者都听说过深度学习这一激动人心的新技术，并对一本书提及一个新兴领域的“历史”而感到惊讶。事实上，深度学习的历史可以追溯到 20 世纪 40 年代。深度学习看似是一个全新的领域，只不过因为在目前流行的前几年它是相对冷门的，同时也因为它被赋予了许多不同的名称（其中大部分已经不再使用），最近才成为众所周知的“深度学习”。这个领域已经更换了很多名称，它反映了不同的研究人员和不同观点的影响。

全面地讲述深度学习的历史超出了本书的范围。然而，一些基本的背景对理解深度学习是有用的。一般来说，目前为止深度学习已经经历了三次发展浪潮：20 世纪 40 年代到 60 年代深度学习的雏形出现在控制论（cybernetics）中，20 世纪 80 年代到 90 年代深度学习表现为联结主义（connectionism），直到 2006 年，才真正以深度学习之名复兴。图 1.7 给出了定量的展示。

我们今天知道的一些最早的学习算法，是旨在模拟生物学习的计算模型，即大脑怎样学习或为什么能学习的模型。其结果是深度学习以人工神经网络（artificial neural network, ANN）之名而淡去。彼时，深度学习模型被认为是受生物大脑（无论人类大脑或其他动物的大脑）所启发而设计出来的系统。尽管有些机器学习的神经网络有时被用来理解大脑功能（Hinton and Shallice, 1991），但它们一般都没有被设计成生物功能的真实模型。深度学习的神经观点受两个主要思想启发。一个想法是大脑作为例子证明智能行为是可能的，因此，概念上，建立智能的直接途径是逆向大脑背后的计算原理，并复制其功能。另一种看法是，理解大脑和人类智能背后的原理也非常有趣，因此机器学习模型除了解决工程应用的能力，如果能让人类对这些基本的科学问题有进一步的认识也将会有用。

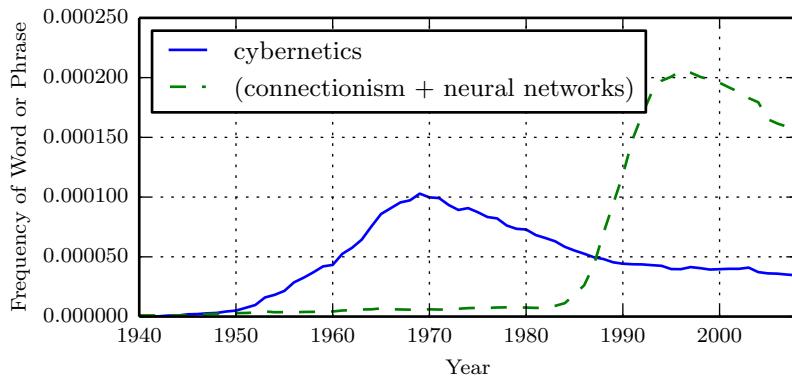


图 1.7: 根据 Google 图书中短语“控制论”、“联结主义”或“神经网络”频率衡量的人工神经网络研究的历史浪潮 (图中展示了三次浪潮的前两次, 第三次最近才出现)。第一次浪潮开始于 20 世纪 40 年代到 20 世纪 60 年代的控制论, 随着生物学习理论的发展 (McCulloch and Pitts, 1943; Hebb, 1949) 和第一个模型的实现 (如感知机 (Rosenblatt, 1958)) , 能实现单个神经元的训练。第二次浪潮开始于 1980-1995 年间的联结主义方法, 可以使用反向传播 (Rumelhart *et al.*, 1986a) 训练具有一个或两个隐藏层的神经网络。当前第三次浪潮, 也就是深度学习, 大约始于 2006 年 (Hinton *et al.*, 2006a; Bengio *et al.*, 2007a; Ranzato *et al.*, 2007a), 并且现在在 2016 年以书的形式出现。另外两次浪潮类似地出现在书中的时间比相应的科学活动晚得多。

现代术语“深度学习”超越了目前机器学习模型的神经科学观点。它诉诸于学习多层次组合这一更普遍的原理, 这一原理也可以应用于那些并非受神经科学启发的机器学习框架。

现代深度学习的最早前身是从神经科学的角度出发的简单线性模型。这些模型被设计为使用一组  $n$  个输入  $x_1, \dots, x_n$  并将它们与一个输出  $y$  相关联。这些模型希望学习一组权重  $w_1, \dots, w_n$ , 并计算它们的输出  $f(\mathbf{x}, \mathbf{w}) = x_1 w_1 + \dots + x_n w_n$ 。如图 1.7 所示, 这第一波神经网络研究浪潮被称为控制论。

McCulloch-Pitts 神经元 (McCulloch and Pitts, 1943) 是脑功能的早期模型。该线性模型通过检验函数  $f(\mathbf{x}, \mathbf{w})$  的正负来识别两种不同类别的输入。显然, 模型的权重需要正确设置后才能使模型的输出对应于期望的类别。这些权重可以由操作人员设定。在 20 世纪 50 年代, 感知机 (Rosenblatt, 1956, 1958) 成为第一个能根据每个类别的输入样本来学习权重的模型。约在同一时期, 自适应线性单元 (adaptive linear element, ADALINE) 简单地返回函数  $f(\mathbf{x})$  本身的值来预测一个实数 (Widrow and Hoff, 1960), 并且它还可以学习从数据预测这些数。

这些简单的学习算法大大影响了机器学习的现代景象。用于调节 ADALINE 权重的训练算法是被称为 **随机梯度下降** (stochastic gradient descent) 的一种特例。稍加改进后的随机梯度下降算法仍然是当今深度学习的主要训练算法。

基于感知机和 ADALINE 中使用的函数  $f(\mathbf{x}, \mathbf{w})$  的模型被称为 **线性模型** (linear model)。尽管在许多情况下，这些模型以不同于原始模型的方式进行训练，但仍是目前最广泛使用的机器学习模型。

线性模型有很多局限性。最著名的是，它们无法学习异或 (XOR) 函数，即  $f([0, 1], \mathbf{w}) = 1$  和  $f([1, 0], \mathbf{w}) = 1$ ，但  $f([1, 1], \mathbf{w}) = 0$  和  $f([0, 0], \mathbf{w}) = 0$ 。观察到线性模型这个缺陷的批评者对受生物学启发的学习普遍地产生了抵触 (Minsky and Papert, 1969)。这导致了神经网络热潮的第一次大衰退。

现在，神经科学被视为深度学习研究的一个重要灵感来源，但它已不再是该领域的主要指导。

如今神经科学在深度学习研究中的作用被削弱，主要原因是我们根本没有足够的关于大脑的信息来作为指导去使用它。要获得对被大脑实际使用算法的深刻理解，我们需要有能力同时监测（至少是）数千相连神经元的活动。我们不能够做到这一点，所以我们甚至连大脑最简单、最深入研究的部分都还远远没有理解 (Olshausen and Field, 2005)。

神经科学已经给了我们依靠单一深度学习算法解决许多不同任务的理由。神经学家们发现，如果将雪貂的大脑重新连接，使视觉信号传送到听觉区域，它们可以学会用大脑的听觉处理区域去“看” (Von Melchner *et al.*, 2000)。这暗示着大多数哺乳动物的大脑能够使用单一的算法就可以解决其大脑可以解决的大部分不同任务。在这个假设之前，机器学习研究是比较分散的，研究人员在不同的社群研究自然语言处理、计算机视觉、运动规划和语音识别。如今，这些应用社群仍然是独立的，但是对于深度学习研究团体来说，同时研究许多或甚至所有这些应用领域是很常见的。

我们能够从神经科学得到一些粗略的指南。仅通过计算单元之间的相互作用而变得智能的基本思想是受大脑启发的。新认知机 (Fukushima, 1980) 受哺乳动物视觉系统的结构启发，引入了一个处理图片的强大模型架构，它后来成为了现代卷积网络的基础 (LeCun *et al.*, 1998c)（我们将会在第 9.10 节看到）。目前大多数神经网络是基于一个称为 **整流线性单元** (rectified linear unit) 的神经单元模型。原始认知机 (Fukushima, 1975) 受我们关于大脑功能知识的启发，引入了一个更复杂的版本。简化的现代版通过吸收来自不同观点的思想而形成，Nair and Hinton (2010b)

和 Glorot *et al.* (2011a) 援引神经科学作为影响, Jarrett *et al.* (2009a) 援引更多面向工程的影响。虽然神经科学是灵感的重要来源, 但它不需要被视为刚性指导。我们知道, 真实的神经元计算着与现代整流线性单元非常不同的函数, 但更接近真实神经网络的系统并没有导致机器学习性能的提升。此外, 虽然神经科学已经成功地启发了一些神经网络架构, 但我们对用于神经科学的生物学习还没有足够多的了解, 因此也就不能为训练这些架构用的学习算法提供太多的借鉴。

媒体报道经常强调深度学习与大脑的相似性。的确, 深度学习研究者比其他机器学习领域 (如核方法或贝叶斯统计) 的研究者更可能地引用大脑作为影响, 但是大家不应该认为深度学习在尝试模拟大脑。现代深度学习从许多领域获取灵感, 特别是应用数学的基本内容如线性代数、概率论、信息论和数值优化。尽管一些深度学习的研究人员引用神经科学作为灵感的重要来源, 然而其他学者完全不关心神经科学。

值得注意的是, 了解大脑是如何在算法层面上工作的尝试确实存在且发展良好。这项尝试主要被称为“计算神经科学”, 并且是独立于深度学习的领域。研究人员在两个领域之间来回研究是很常见的。深度学习领域主要关注如何构建计算机系统, 从而成功解决需要智能才能解决的任务, 而计算神经科学领域主要关注构建大脑如何真实工作的比较精确的模型。

在 20 世纪 80 年代, 神经网络研究的第二次浪潮在很大程度上是伴随一个被称为 **联结主义** (connectionism) 或 **并行分布处理** (parallel distributed processing) 潮流而出现的 (Rumelhart *et al.*, 1986d; McClelland *et al.*, 1995)。联结主义是在认知科学的背景下出现的。认知科学是理解思维的跨学科途径, 即它融合多个不同的分析层次。在 20 世纪 80 年代初期, 大多数认知科学家研究符号推理模型。尽管这很流行, 但符号模型很难解释大脑如何真正使用神经元实现推理功能。联结主义者开始研究真正基于神经系统实现的认知模型 (Touretzky and Minton, 1985), 其中很多复苏的想法可以追溯到心理学家 Donald Hebb 在 20 世纪 40 年代的工作 (Hebb, 1949)。

联结主义的中心思想是, 当网络将大量简单的计算单元连接在一起时可以实现智能行为。这种见解同样适用于生物神经系统中的神经元, 因为它和计算模型中隐藏单元起着类似的作用。

在上世纪 80 年代的联结主义期间形成的几个关键概念在今天的深度学习中仍然是非常重要的。

其中一个概念是 **分布式表示** (distributed representation) (Hinton *et al.*, 1986)。其思想是：系统的每一个输入都应该由多个特征表示，并且每一个特征都应该参与到多个可能输入的表示。例如，假设我们有一个能够识别红色、绿色、或蓝色的汽车、卡车和鸟类的视觉系统，表示这些输入的其中一个方法是将九个可能的组合：红卡车，红汽车，红鸟，绿卡车等等使用单独的神经元或隐藏单元激活。这需要九个不同的神经元，并且每个神经必须独立地学习颜色和对象身份的概念。改善这种情况的方法之一是使用分布式表示，即用三个神经元描述颜色，三个神经元描述对象身份。这仅仅需要 6 个神经元而不是 9 个，并且描述红色的神经元能够从汽车、卡车和鸟类的图像中学习红色，而不仅仅是从一个特定类别的图像中学习。分布式表示的概念是本书的核心，我们将在第十五章中更加详细地描述。

联结主义潮流的另一个重要成就是反向传播在训练具有内部表示的深度神经网络中的成功使用以及反向传播算法的普及 (Rumelhart *et al.*, 1986c; LeCun, 1987)。这个算法虽然曾黯然失色不再流行，但截至写书之时，它仍是训练深度模型的主导方法。

在 20 世纪 90 年代，研究人员在使用神经网络进行序列建模的方面取得了重要进展。Hochreiter (1991b) 和 Bengio *et al.* (1994a) 指出了对长序列进行建模的一些根本性数学难题，这将在第 10.7 节中描述。Hochreiter and Schmidhuber (1997) 引入 **长短期记忆** (long short-term memory, LSTM) 网络来解决这些难题。如今，LSTM 在许多序列建模任务中广泛应用，包括 Google 的许多自然语言处理任务。

神经网络研究的第二次浪潮一直持续到上世纪 90 年代中期。基于神经网络和其他 AI 技术的创业公司开始寻求投资，其做法野心勃勃但不切实际。当 AI 研究不能实现这些不合理的期望时，投资者感到失望。同时，机器学习的其他领域取得了进步。比如，核方法 (Boser *et al.*, 1992; Cortes and Vapnik, 1995; Schölkopf *et al.*, 1999) 和图模型 (Jordan, 1998) 都在很多重要任务上实现了很好的效果。这两个因素导致了神经网络热潮的第二次衰退，并一直持续到 2007 年。

在此期间，神经网络继续在某些任务上获得令人印象深刻的表现 (LeCun *et al.*, 1998c; Bengio *et al.*, 2001a)。加拿大高级研究所 (CIFAR) 通过其神经计算和自适应感知 (NCAP) 研究计划帮助维持神经网络研究。该计划联合了分别由 Geoffrey Hinton、Yoshua Bengio 和 Yann LeCun 领导的多伦多大学、蒙特利尔大学和纽约大学的机器学习研究小组。这个多学科的 CIFAR NCAP 研究计划还囊括了神经科学家、人类和计算机视觉专家。

在那个时候，人们普遍认为深度网络是难以训练的。现在我们知道，20世纪80年代就存在的算法能工作得非常好，但是直到在2006年前后都没有体现出来。这可能仅仅由于其计算代价太高，而以当时可用的硬件难以进行足够的实验。

神经网络研究的第三次浪潮始于2006年的突破。Geoffrey Hinton表明名为深度信念网络的神经网络可以使用一种称为贪婪逐层预训练的策略来有效地训练(Hinton *et al.*, 2006a)，我们将在第15.1节中更详细地描述。其他CIFAR附属研究小组很快表明，同样的策略可以被用来训练许多其他类型的深度网络(Bengio and LeCun, 2007a; Ranzato *et al.*, 2007b)，并能系统地帮助提高在测试样例上的泛化能力。神经网络研究的这一次浪潮普及了“深度学习”这一术语的使用，强调研究者现在有能力训练以前不可能训练的比较深的神经网络，并着力于深度的理论重要性上(Bengio and LeCun, 2007b; Delalleau and Bengio, 2011; Pascanu *et al.*, 2014a; Montufar *et al.*, 2014)。此时，深度神经网络已经优于与之竞争的基于其他机器学习技术以及手工设计功能的AI系统。在写这本书的时候，神经网络的第三次发展浪潮仍在继续，尽管深度学习的研究重点在这一段时间内发生了巨大变化。第三次浪潮已开始着眼于新的无监督学习技术和深度模型在小数据集的泛化能力，但目前更多的兴趣点仍是比較传统的监督学习算法和深度模型充分利用大型标注数据集的能力。

### 1.2.2 与日俱增的数据量

人们可能想问，既然人工神经网络的第一个实验在20世纪50年代就完成了，但为什么深度学习直到最近才被认为是关键技术。自20世纪90年代以来，深度学习就已经成功用于商业应用，但通常被视为是一种只有专家才可以使用的艺术而不是一种技术，这种观点一直持续到最近。确实，要从一个深度学习算法获得良好的性能需要一些技巧。幸运的是，随着训练数据的增加，所需的技巧正在减少。目前在复杂的任务达到人类水平的学习算法，与20世纪80年代努力解决玩具问题(toy problem)的学习算法几乎是一样的，尽管我们使用这些算法训练的模型经历了变革，即简化了极深架构的训练。最重要的新进展是现在我们有了这些算法得以成功训练所需的资源。图1.8展示了基准数据集的大小如何随着时间的推移而显著增加。这种趋势是由社会日益数字化驱动的。由于我们的活动越来越多发生在计算机上，我们做什么也越来越多地被记录。由于我们的计算机越来越多地联网在一起，这些记录变得更容易集中管理，并更容易将它们整理成适于机器学习应用的数据集。因为

统计估计的主要负担（观察少量数据以在新数据上泛化）已经减轻，“大数据”时代使机器学习更加容易。截至 2016 年，一个粗略的经验法则是，监督深度学习算法在每类给定约 5000 个标注样本情况下一般将达到可以接受的性能，当至少有 1000 万个标注样本的数据集用于训练时，它将达到或超过人类表现。此外，在更小的数据集上获得成功是一个重要的研究领域，为此我们应特别侧重于如何通过无监督或半监督学习充分利用大量的未标注样本。

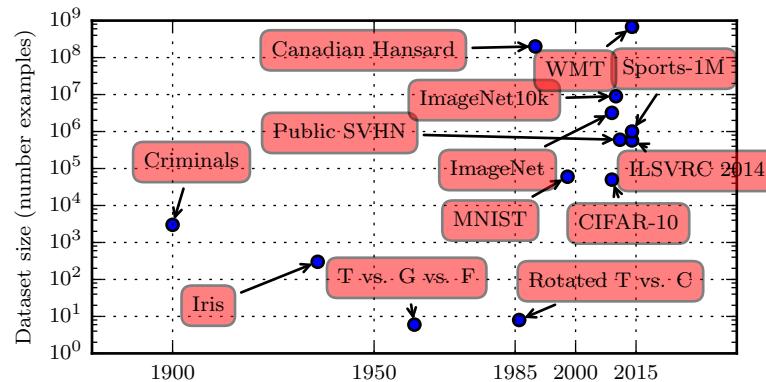


图 1.8: 与日俱增的数据量。20 世纪初，统计学家使用数百或数千的手工制作的度量来研究数据集 (Garson, 1900; Gosset, 1908; Anderson, 1935; Fisher, 1936)。20 世纪 50 年代到 80 年代，受生物启发的机器学习开拓者通常使用小的合成数据集，如低分辨率的字母位图，设计为在低计算成本下表明神经网络能够学习特定功能 (Widrow and Hoff, 1960; Rumelhart *et al.*, 1986b)。20 世纪 80 年代和 90 年代，机器学习变得更加统计，并开始利用包含成千上万个样本的更大数据集，如手写扫描数字的 MNIST 数据集 (如图 1.9 所示 (LeCun *et al.*, 1998c))。在 21 世纪初的第一个十年，相同大小更复杂的数据集持续出现，如 CIFAR-10 数据集 (Krizhevsky and Hinton, 2009)。在这十年结束和下五年，明显更大的数据集（包含数万到数千万的样例）完全改变了深度学习的可能实现的事。这些数据集包括公共 Street View House Numbers 数据集 (Netzer *et al.*, 2011)、各种版本的 ImageNet 数据集 (Deng *et al.*, 2009, 2010a; Russakovsky *et al.*, 2014a) 以及 Sports-1M 数据集 (Karpathy *et al.*, 2014)。在图顶部，我们看到翻译句子的数据集通常远大于其他数据集，如根据 Canadian Hansard 制作的 IBM 数据集 (Brown *et al.*, 1990) 和 WMT 2014 英法数据集 (Schwenk, 2014)。

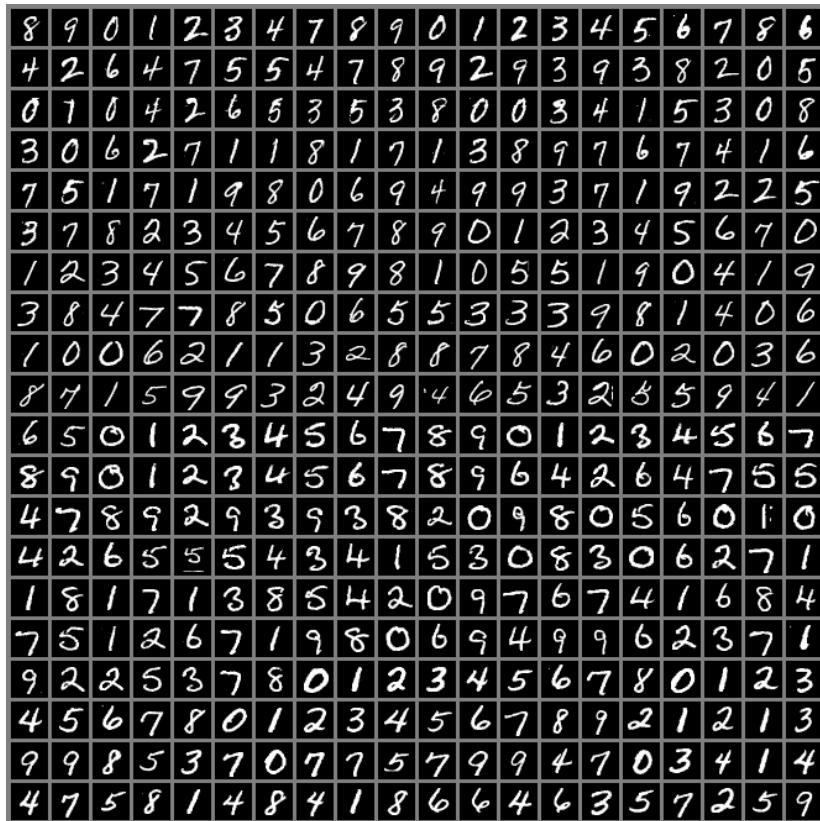


图 1.9: MNIST 数据集的输入样例。“NIST”代表国家标准和技术研究所 (National Institute of Standards and Technology)，是最初收集这些数据的机构。“M”代表“修改的 (Modified)”，为更容易地与机器学习算法一起使用，数据已经过预处理。MNIST 数据集包括手写数字的扫描和相关标签 (描述每个图像中包含 0-9 中哪个数字)。这个简单的分类问题是深度学习研究中最简单和最广泛使用的测试之一。尽管现代技术很容易解决这个问题，它仍然很受欢迎。Geoffrey Hinton 将其描述为“机器学习的果蝇”，这意味着机器学习研究人员可以在受控的实验室条件下研究他们的算法，就像生物学家经常研究果蝇一样。

### 1.2.3 与日俱增的模型规模

20 世纪 80 年代，神经网络只能取得相对较小的成功，而现在神经网络非常成功的另一个重要原因是我们现在拥有的计算资源可以运行更大的模型。联结主义的主要见解之一是，当动物的许多神经元一起工作时会变得聪明。单独神经元或小集合的神经元不是特别有用。

生物神经元不是特别稠密地连接在一起。如图 1.10 所示，几十年来，我们的机器学习模型中每个神经元的连接数量已经与哺乳动物的大脑在同一数量级上。

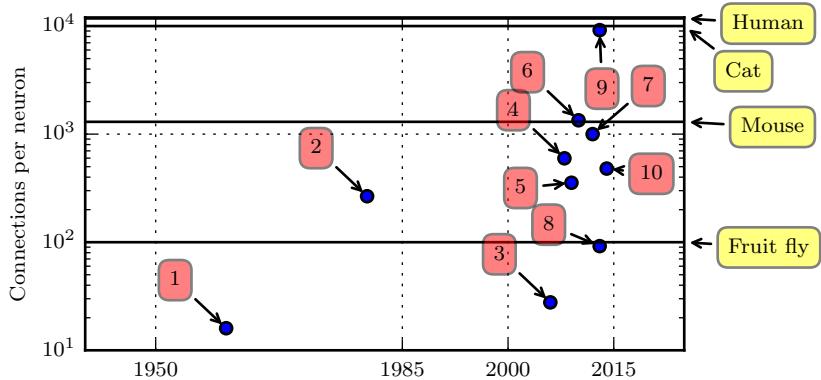


图 1.10：与日俱增的每神经元连接数。最初，人工神经网络中神经元之间的连接数受限于硬件能力。而现在，神经元之间的连接数大多是出于设计考虑。一些人工神经网络中每个神经元的连接数与猫一样多，并且对于其他神经网络来说，每个神经元的连接与较小哺乳动物（如小鼠）一样多是非常普遍的。甚至人类大脑每个神经元的连接也没有过高的数量。生物神经网络规模来自 Wikipedia (2015)。

1. 自适应线性单元 (Widrow and Hoff, 1960)
2. 神经认知机 (Fukushima, 1980)
3. GPU-加速 卷积网络 (Chellapilla *et al.*, 2006)
4. 深度玻尔兹曼机 (Salakhutdinov and Hinton, 2009a)
5. 无监督卷积网络 (Jarrett *et al.*, 2009b)
6. GPU-加速 多层感知机 (Ciresan *et al.*, 2010)
7. 分布式自编码器 (Le *et al.*, 2012)
8. Multi-GPU 卷积网络 (Krizhevsky *et al.*, 2012a)
9. COTS HPC 无监督卷积网络 (Coates *et al.*, 2013)
10. GoogLeNet (Szegedy *et al.*, 2014a)

如图 1.11 所示，就神经元的总数目而言，直到最近神经网络都是惊人的小。自从隐藏单元引入以来，人工神经网络的规模大约每 2.4 年扩大一倍。这种增长是由更大内存、更快的计算机和更大的可用数据集驱动的。更大的网络能够在更复杂的任务中实现更高的精度。这种趋势看起来将持续数十年。除非有能力迅速扩展的新技术，否则至少要到 21 世纪 50 年代，人工神经网络将才能具备与人脑相同数量级的神经元。生物神经元表示的功能可能比目前的人工神经元所表示的更复杂，因此

生物神经网络可能比图中描绘的甚至要更大。

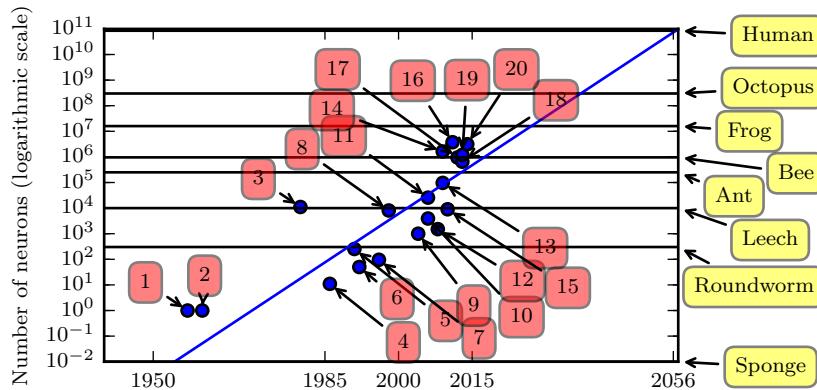


图 1.11: 与日俱增的神经网络规模。自从引入隐藏单元，人工神经网络的大小大约每 2.4 年翻一倍。生物神经网络规模来自 Wikipedia (2015)。

1. 感知机 (Rosenblatt, 1958, 1962)
2. 自适应线性单元 (Widrow and Hoff, 1960)
3. 神经认知机 (Fukushima, 1980)
4. 早期后向传播网络 (Rumelhart *et al.*, 1986b)
5. 用于语音识别的循环神经网络 (Robinson and Fallside, 1991)
6. 用于语音识别的多层感知机 (Bengio *et al.*, 1991)
7. 均匀场sigmoid信念网络 (Saul *et al.*, 1996)
8. LeNet-5 (LeCun *et al.*, 1998c)
9. 回声状态网络 (Jaeger and Haas, 2004)
10. 深度信念网络 (Hinton *et al.*, 2006a)
11. GPU-加速卷积网络 (Chellapilla *et al.*, 2006)
12. 深度玻尔兹曼机 (Salakhutdinov and Hinton, 2009a)
13. GPU-加速深度信念网络 (Raina *et al.*, 2009a)
14. 无监督卷积网络 (Jarrett *et al.*, 2009b)
15. GPU-加速多层次感知机 (Ciresan *et al.*, 2010)
16. OMP-1 网络 (Coates and Ng, 2011)
17. 分布式自编码器 (Le *et al.*, 2012)
18. Multi-GPU卷积网络 (Krizhevsky *et al.*, 2012a)
19. COTS HPC 无监督卷积网络 (Coates *et al.*, 2013)
20. GoogLeNet (Szegedy *et al.*, 2014a)

现在看来，其神经元比一个水蛭还少的神经网络不能解决复杂的人工智能问题是不足为奇的。即使现在的网络，从计算系统角度来看它可能相当大的，但实际上它比相对原始的脊椎动物如青蛙的神经系统还要小。

由于更快的 CPU、通用 GPU 的出现（在第 12.1.2 节中讨论）、更快的网络连接和更好的分布式计算的软件基础设施，模型规模随着时间的推移不断增加是深度学习历史中最重要的趋势之一。人们普遍预计这种趋势将很好地持续到未来。

### 1.2.4 与日俱增的精度、复杂度和对现实世界的冲击

20 世纪 80 年代以来，深度学习提供精确识别和预测的能力一直在提高。而且，深度学习持续成功地被应用于越来越广泛的实际问题中。

最早的深度模型被用来识别裁剪紧凑且非常小的图像中的单个对象 (Rumelhart *et al.*, 1986d)。此后，神经网络可以处理的图像尺寸逐渐增加。现代对象识别网络能处理丰富的高分辨率照片，并且不需要在被识别的对象附近进行裁剪 (Krizhevsky *et al.*, 2012b)。类似地，最早的网络只能识别两种对象（或在某些情况下，单类对象的存在与否），而这些现代网络通常能够识别至少 1000 个不同类别的对象。对象识别中最大的比赛是每年举行的 ImageNet 大型视觉识别挑战 (ILSVRC)。深度学习迅速崛起的激动人心的一幕是卷积网络第一次大幅赢得这一挑战，它将最高水准的前 5 错误率从 26.1% 降到 15.3% (Krizhevsky *et al.*, 2012b)，这意味着该卷积网络针对每个图像的可能类别生成一个顺序列表，除了 15.3% 的测试样本，其他测试样本的正确类标都出现在此列表中的前 5 项里。此后，深度卷积网络连续地赢得这些比赛，截至写本书时，深度学习的最新结果将这个比赛中的前 5 错误率降到了 3.6%，如图 1.12 所示。

深度学习也对语音识别产生了巨大影响。语音识别在 20 世纪 90 年代得到提高后，直到约 2000 年都停滞不前。深度学习的引入 (Dahl *et al.*, 2010; Deng *et al.*, 2010b; Seide *et al.*, 2011; Hinton *et al.*, 2012a) 使得语音识别错误率陡然下降，有些错误率甚至降低了一半。我们将在第 12.3 节更详细地探讨这个历史。

深度网络在行人检测和图像分割中也取得了引人注目的成功 (Sermanet *et al.*, 2013; Farabet *et al.*, 2013; Couprie *et al.*, 2013)，并且在交通标志分类上取得了超越人类的表现 (Ciresan *et al.*, 2012)。

在深度网络的规模和精度有所提高的同时，它们可以解决的任务也日益复杂。Goodfellow *et al.* (2014d) 表明，神经网络可以学习输出描述图像的整个字符序列，而不是仅仅识别单个对象。此前，人们普遍认为，这种学习需要对序列中的单个元素进行标注 (Gulcehre and Bengio, 2013)。循环神经网络，如之前提到的 LSTM 序列模型，现在用于对序列和其他序列之间的关系进行建模，而不是仅仅固定输入之

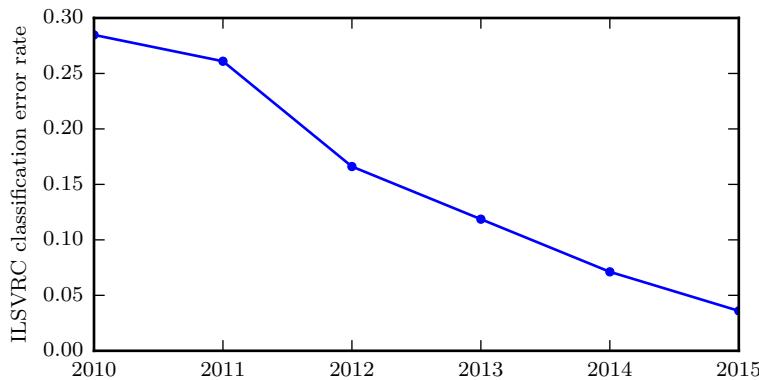


图 1.12: 日益降低的错误率。由于深度网络达到了在 ImageNet 大规模视觉识别挑战中竞争所必需的规模, 它们每年都能赢得胜利, 并且产生越来越低的错误率。数据来源于 Russakovsky *et al.* (2014b) 和 He *et al.* (2015)。

间的关系。这种序列到序列的学习似乎引领着另一个应用的颠覆性发展, 即机器翻译 (Sutskever *et al.*, 2014; Bahdanau *et al.*, 2015)。

这种复杂性日益增加的趋势已将其推向逻辑结论, 即神经图灵机 (Graves *et al.*, 2014) 的引入, 它能学习读取存储单元和向存储单元写入任意内容。这样的神经网络可以从期望行为的样本中学习简单的程序。例如, 从杂乱和排好序的样本中学习对一系列数进行排序。这种自我编程技术正处于起步阶段, 但原则上未来可以适用于几乎所有的任务。

深度学习的另一个最大的成就是其在 强化学习 (reinforcement learning) 领域的扩展。在强化学习中, 一个自主的智能体必须在没有人类操作者指导下, 通过试错来学习执行任务。DeepMind 表明, 基于深度学习的强化学习系统能够学会玩 Atari 视频游戏, 并在多种任务中可与人类匹敌 (Mnih *et al.*, 2015)。深度学习也显著改善了机器人强化学习的性能 (Finn *et al.*, 2015)。

许多深度学习应用都是高利润的。现在深度学习被许多顶级的技术公司使用, 包括 Google、Microsoft、Facebook、IBM、Baidu、Apple、Adobe、Netflix、NVIDIA 和 NEC 等。

深度学习的进步也严重依赖于软件基础架构的进展。软件库如 Theano (Bergstra *et al.*, 2010a; Bastien *et al.*, 2012a)、PyLearn2 (Goodfellow *et al.*, 2013e)、Torch (Collobert *et al.*, 2011b)、DistBelief (Dean *et al.*, 2012)、Caffe (Jia, 2013)、MXNet (Chen

*et al.*, 2015) 和 TensorFlow (Abadi *et al.*, 2015) 都能支持重要的研究项目或商业产品。

深度学习也为其他科学做出了贡献。用于对象识别的现代卷积网络为神经科学家们提供了可以研究的视觉处理模型 (DiCarlo, 2013)。深度学习也为处理海量数据以及在科学领域作出有效的预测提供了非常有用的工具。它已成功地用于预测分子如何相互作用从而帮助制药公司设计新的药物 (Dahl *et al.*, 2014)，搜索亚原子粒子 (Baldi *et al.*, 2014)，以及自动解析用于构建人脑三维图的显微镜图像 (Knowles-Barley *et al.*, 2014) 等。我们期待深度学习未来能够出现在越来越多的科学领域中。

总之，深度学习是机器学习的一种方法。在过去几十年的发展中，它大量借鉴了我们关于人脑、统计学和应用数学的知识。近年来，得益于更强大的计算机、更大的数据集和能够训练更深网络的技术，深度学习的普及性和实用性都有了极大的发展。未来几年充满了进一步提高深度学习并将它带到新领域的挑战和机遇。

# 第一部分

## 应用数学与机器学习基础

本书这一部分将介绍理解深度学习所需的基本数学概念。我们从应用数学的一般概念开始，这能使我们定义许多变量的函数，找到这些函数的最高和最低点，并量化信念度。

接着，我们描述机器学习的基本目标，并描述如何实现这些目标。我们需要指定代表某些信念的模型、设计衡量这些信念与现实对应程度的代价函数以及使用训练算法最小化这个代价函数。

这个基本框架是广泛多样的机器学习算法的基础，其中也包括非深度的机器学习方法。在本书的后续部分，我们将在这个框架下开发深度学习算法。

# 第二章 线性代数

线性代数作为数学的一个分支，广泛应用于科学和工程中。然而，因为线性代数主要是面向连续数学，而非离散数学，所以很多计算机科学家很少接触它。掌握好线性代数对于理解和从事机器学习算法相关工作是很有必要的，尤其对于深度学习算法而言。因此，在开始介绍深度学习之前，我们集中探讨一些必备的线性代数知识。

如果你已经很熟悉线性代数，那么可以轻松地跳过本章。如果你已经了解这些概念，但是需要一份索引表来回顾一些重要公式，那么我们推荐 *The Matrix Cookbook* (Petersen and Pedersen, 2006)。如果你没有接触过线性代数，那么本章将告诉你本书所需的线性代数知识，不过我们仍然非常建议你参考其他专门讲解线性代数的文献，例如 Shilov (1977)。最后，本章略去了很多重要但是对于理解深度学习非必需的线性代数知识。

## 2.1 标量、向量、矩阵和张量

学习线性代数，会涉及以下几类数学概念：

- **标量 ( scalar )**: 一个标量就是一个单独的数，它不同于线性代数中研究的其他大部分对象（通常是多个数的数组）。我们用斜体表示标量。标量通常被赋予小写的变量名称。当我们介绍标量时，会明确它们是哪种类型的数。比如，在定义实数标量时，我们可能会说“令  $s \in \mathbb{R}$  表示一条线的斜率”；在定义自然数标量时，我们可能会说“令  $n \in \mathbb{N}$  表示元素的数目”。
- **向量 ( vector )**: 一个向量是一列数。这些数是有序排列的。通过次序中的索引，我们可以确定每个单独的数。通常我们赋予向量粗体的小写变量名称，比如  $\mathbf{x}$ 。向量中的元素可以通过带脚标的斜体表示。向量  $\mathbf{x}$  的第一个元素是  $x_1$ ，

第二个元素是  $x_2$ , 等等。我们也会注明存储在向量中的元素是什么类型的。如果每个元素都属于  $\mathbb{R}$ , 并且该向量有  $n$  个元素, 那么该向量属于实数集  $\mathbb{R}$  的  $n$  次笛卡尔乘积构成的集合, 记为  $\mathbb{R}^n$ 。当需要明确表示向量中的元素时, 我们会将元素排列成一个方括号包围的纵列:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}. \quad (2.1)$$

我们可以把向量看作空间中的点, 每个元素是不同坐标轴上的坐标。

有时我们需要索引向量中的一些元素。在这种情况下, 我们定义一个包含这些元素索引的集合, 然后将该集合写在脚标处。比如, 指定  $x_1$ ,  $x_3$  和  $x_6$ , 我们定义集合  $S = \{1, 3, 6\}$ , 然后写作  $\mathbf{x}_S$ 。我们用符号 $-$ 表示集合的补集中的索引。比如  $\mathbf{x}_{-1}$  表示  $\mathbf{x}$  中除  $x_1$  外的所有元素,  $\mathbf{x}_{-S}$  表示  $\mathbf{x}$  中除  $x_1$ ,  $x_3$ ,  $x_6$  外所有元素构成的向量。

- **矩阵 (matrix):** 矩阵是一个二维数组, 其中的每一个元素被两个索引 (而非一个) 所确定。我们通常会赋予矩阵粗体的大写变量名称, 比如  $\mathbf{A}$ 。如果一个实数矩阵高度为  $m$ , 宽度为  $n$ , 那么我们说  $\mathbf{A} \in \mathbb{R}^{m \times n}$ 。我们在表示矩阵中的元素时, 通常以不加粗的斜体形式使用其名称, 索引用逗号间隔。比如,  $A_{1,1}$  表示  $\mathbf{A}$  左上的元素,  $A_{m,n}$  表示  $\mathbf{A}$  右下的元素。我们通过用 “ $:$ ” 表示水平坐标, 以表示垂直坐标  $i$  中的所有元素。比如,  $\mathbf{A}_{:,i}$  表示  $\mathbf{A}$  中垂直坐标  $i$  上的一横排元素。这也被称为  $\mathbf{A}$  的第  $i$  行 (row)。同样地,  $\mathbf{A}_{:,i}$  表示  $\mathbf{A}$  的第  $i$  列 (column)。当我们需要明确表示矩阵中的元素时, 我们将它们写在用方括号括起来的数组中:

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}. \quad (2.2)$$

有时我们需要矩阵值表达式的索引, 而不是单个元素。在这种情况下, 我们在表达式后面接下标, 但不必将矩阵的变量名称小写化。比如,  $f(\mathbf{A})_{i,j}$  表示函数  $f$  作用在  $\mathbf{A}$  上输出的矩阵的第  $i$  行第  $j$  列元素。

- **张量 (tensor):** 在某些情况下, 我们会讨论坐标超过二维的数组。一般地, 一个数组中的元素分布在若干维坐标的规则网格中, 我们称之为张量。我们使用字体  $\mathbf{A}$  来表示张量 “ $A$ ”。张量  $\mathbf{A}$  中坐标为  $(i, j, k)$  的元素记作  $A_{i,j,k}$ 。

转置 ( transpose ) 是矩阵的重要操作之一。矩阵的转置是以对角线为轴的镜像，这条从左上角到右下角的对角线被称为 **主对角线** ( main diagonal )。图 2.1 显示了这个操作。我们将矩阵  $\mathbf{A}$  的转置表示为  $\mathbf{A}^\top$ ，定义如下

$$(\mathbf{A}^\top)_{i,j} = A_{j,i}. \quad (2.3)$$

向量可以看作只有一列的矩阵。对应地，向量的转置可以看作是只有一行的矩阵。有时，我们通过将向量元素作为行矩阵写在文本行中，然后使用转置操作将其变为标准的列向量，来定义一个向量，比如  $\mathbf{x} = [x_1, x_2, x_3]^\top$ 。

标量可以看作是只有一个元素的矩阵。因此，标量的转置等于它本身， $a = a^\top$ 。

$$\mathbf{A} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \\ A_{3,1} & A_{3,2} \end{bmatrix} \Rightarrow \mathbf{A}^\top = \begin{bmatrix} A_{1,1} & A_{2,1} & A_{3,1} \\ A_{1,2} & A_{2,2} & A_{3,2} \end{bmatrix}$$

图 2.1: 矩阵的转置可以看成以主对角线为轴的一个镜像。

只要矩阵的形状一样，我们可以把两个矩阵相加。两个矩阵相加是指对应位置的元素相加，比如  $\mathbf{C} = \mathbf{A} + \mathbf{B}$ ，其中  $C_{i,j} = A_{i,j} + B_{i,j}$ 。

标量和矩阵相乘，或是和矩阵相加时，我们只需将其与矩阵的每个元素相乘或相加，比如  $\mathbf{D} = a \cdot \mathbf{B} + c$ ，其中  $D_{i,j} = a \cdot B_{i,j} + c$ 。

在深度学习中，我们也使用一些不那么常规的符号。我们允许矩阵和向量相加，产生另一个矩阵： $\mathbf{C} = \mathbf{A} + \mathbf{b}$ ，其中  $C_{i,j} = A_{i,j} + b_j$ 。换言之，向量  $\mathbf{b}$  和矩阵  $\mathbf{A}$  的每一行相加。这个简写方法使我们无需在加法操作前定义一个将向量  $\mathbf{b}$  复制到每一行而生成的矩阵。这种隐式地复制向量  $\mathbf{b}$  到很多位置的方式，被称为 **广播** ( broadcasting )。

## 2.2 矩阵和向量相乘

矩阵乘法是矩阵运算中最重要的操作之一。两个矩阵  $\mathbf{A}$  和  $\mathbf{B}$  的矩阵乘积 ( matrix product ) 是第三个矩阵  $\mathbf{C}$ 。为了使乘法定义良好，矩阵  $\mathbf{A}$  的列数必须和矩阵  $\mathbf{B}$  的行数相等。如果矩阵  $\mathbf{A}$  的形状是  $m \times n$ ，矩阵  $\mathbf{B}$  的形状是  $n \times p$ ，那么矩阵

$C$  的形状是  $m \times p$ 。我们可以通过将两个或多个矩阵并列放置以书写矩阵乘法，例如

$$C = AB. \quad (2.4)$$

具体地，该乘法操作定义为

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}. \quad (2.5)$$

需要注意的是，两个矩阵的标准乘积不是指两个矩阵中对应元素的乘积。不过，那样的矩阵操作确实是存在的，被称为 **元素对应乘积** (element-wise product) 或者 **Hadamard 乘积** (Hadamard product)，记为  $A \odot B$ 。

两个相同维数的向量  $x$  和  $y$  的 **点积** (dot product) 可看作是矩阵乘积  $x^\top y$ 。我们可以把矩阵乘积  $C = AB$  中计算  $C_{i,j}$  的步骤看作是  $A$  的第  $i$  行和  $B$  的第  $j$  列之间的点积。

矩阵乘积运算有许多有用的性质，从而使矩阵的数学分析更加方便。比如，矩阵乘积服从分配律：

$$A(B + C) = AB + AC. \quad (2.6)$$

矩阵乘积也服从结合律：

$$A(BC) = (AB)C. \quad (2.7)$$

不同于标量乘积，矩阵乘积并不满足交换律 ( $AB = BA$  的情况并非总是满足)。然而，两个向量的 **点积** (dot product) 满足交换律：

$$x^\top y = y^\top x. \quad (2.8)$$

矩阵乘积的转置有着简单的形式：

$$(AB)^\top = B^\top A^\top. \quad (2.9)$$

利用两个向量点积的结果是标量，标量转置是自身的事，我们可以证明式 (2.8)：

$$x^\top y = (x^\top y)^\top = y^\top x. \quad (2.10)$$

由于本书的重点不是线性代数，我们并不试图展示矩阵乘积的所有重要性质，但读者应该知道矩阵乘积还有很多有用的性质。

现在我们已经知道了足够多的线性代数符号，可以表达下列线性方程组：

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad (2.11)$$

其中  $\mathbf{A} \in \mathbb{R}^{m \times n}$  是一个已知矩阵， $\mathbf{b} \in \mathbb{R}^m$  是一个已知向量， $\mathbf{x} \in \mathbb{R}^n$  是一个我们要求解的未知向量。向量  $\mathbf{x}$  的每一个元素  $x_i$  都是未知的。矩阵  $\mathbf{A}$  的每一行和  $\mathbf{b}$  中对应的元素构成一个约束。我们可以把式 (2.11) 重写为

$$\mathbf{A}_{1,:}\mathbf{x} = b_1 \quad (2.12)$$

$$\mathbf{A}_{2,:}\mathbf{x} = b_2 \quad (2.13)$$

$$\dots \quad (2.14)$$

$$\mathbf{A}_{m,:}\mathbf{x} = b_m \quad (2.15)$$

或者，更明确地，写作

$$\mathbf{A}_{1,1}x_1 + \mathbf{A}_{1,2}x_2 + \dots + \mathbf{A}_{1,n}x_n = b_1 \quad (2.16)$$

$$\mathbf{A}_{2,1}x_1 + \mathbf{A}_{2,2}x_2 + \dots + \mathbf{A}_{2,n}x_n = b_2 \quad (2.17)$$

$$\dots \quad (2.18)$$

$$\mathbf{A}_{m,1}x_1 + \mathbf{A}_{m,2}x_2 + \dots + \mathbf{A}_{m,n}x_n = b_m. \quad (2.19)$$

矩阵向量乘积符号为这种形式的方程提供了更紧凑的表示。

## 2.3 单位矩阵和逆矩阵

线性代数提供了被称为 **矩阵逆** (matrix inversion) 的强大工具。对于大多数矩阵  $\mathbf{A}$ ，我们都能通过矩阵逆解析地求解式 (2.11)。

为了描述矩阵逆，我们首先需要定义 **单位矩阵** (identity matrix) 的概念。任意向量和单位矩阵相乘，都不会改变。我们将保持  $n$  维向量不变的单位矩阵记作  $\mathbf{I}_n$ 。形式上， $\mathbf{I}_n \in \mathbb{R}^{n \times n}$ ，

$$\forall \mathbf{x} \in \mathbb{R}^n, \mathbf{I}_n \mathbf{x} = \mathbf{x}. \quad (2.20)$$

单位矩阵的结构很简单：所有沿主对角线的元素都是 1，而所有其他位置的元素都是 0。如图 2.2 所示。

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

图 2.2: 单位矩阵的一个样例: 这是  $I_3$ 。

矩阵  $A$  的矩阵逆 (matrix inversion) 记作  $A^{-1}$ , 其定义的矩阵满足如下条件

$$A^{-1}A = I_n. \quad (2.21)$$

现在我们可以通过以下步骤求解式 (2.11) :

$$Ax = b \quad (2.22)$$

$$A^{-1}Ax = A^{-1}b \quad (2.23)$$

$$I_n x = A^{-1}b \quad (2.24)$$

$$x = A^{-1}b. \quad (2.25)$$

当然, 这取决于我们能否找到一个逆矩阵  $A^{-1}$ 。在接下来的章节中, 我们会讨论逆矩阵  $A^{-1}$  存在的条件。

当逆矩阵  $A^{-1}$  存在时, 有几种不同的算法都能找到它的闭解形式。理论上, 相同的逆矩阵可用于多次求解不同向量  $b$  的方程。然而, 逆矩阵  $A^{-1}$  主要是作为理论工具使用的, 并不会在大多数软件应用程序中实际使用。这是因为逆矩阵  $A^{-1}$  在数字计算机上只能表现出有限的精度, 有效使用向量  $b$  的算法通常可以得到更精确的  $x$ 。

## 2.4 线性相关和生成子空间

如果逆矩阵  $A^{-1}$  存在, 那么式 (2.11) 肯定对于每一个向量  $b$  恰好存在一个解。但是, 对于方程组而言, 对于向量  $b$  的某些值, 有可能不存在解, 或者存在无限多个解。存在多于一个解但是少于无限多个解的情况是不可能发生的; 因为如果  $x$  和  $y$  都是某方程组的解, 则

$$z = \alpha x + (1 - \alpha)y \quad (2.26)$$

(其中  $\alpha$  取任意实数) 也是该方程组的解。

为了分析方程有多少个解, 我们可以将  $A$  的列向量看作从原点 (origin) (元素都是零的向量) 出发的不同方向, 确定有多少种方法可以到达向量  $b$ 。在这个观点下, 向量  $x$  中的每个元素表示我们应该沿着这些方向走多远, 即  $x_i$  表示我们需要沿着第  $i$  个向量的方向走多远:

$$Ax = \sum_i x_i A_{:,i}. \quad (2.27)$$

一般而言, 这种操作被称为 **线性组合** (linear combination)。形式上, 一组向量的线性组合, 是指每个向量乘以对应标量系数之后的和, 即:

$$\sum_i c_i v^{(i)}. \quad (2.28)$$

一组向量的 **生成子空间** (span) 是原始向量线性组合后所能抵达的点的集合。

确定  $Ax = b$  是否有解相当于确定向量  $b$  是否在  $A$  列向量的生成子空间中。这个特殊的生成子空间被称为  $A$  的 **列空间** (column space) 或者  $A$  的 **值域** (range)。

为了使方程  $Ax = b$  对于任意向量  $b \in \mathbb{R}^m$  都存在解, 我们要求  $A$  的列空间构成整个  $\mathbb{R}^m$ 。如果  $\mathbb{R}^m$  中的某个点不在  $A$  的列空间中, 那么该点对应的  $b$  会使得该方程没有解。矩阵  $A$  的列空间是整个  $\mathbb{R}^m$  的要求, 意味着  $A$  至少有  $m$  列, 即  $n \geq m$ 。否则,  $A$  列空间的维数会小于  $m$ 。例如, 假设  $A$  是一个  $3 \times 2$  的矩阵。目标  $b$  是 3 维的, 但是  $x$  只有 2 维。所以无论如何修改  $x$  的值, 也只能描绘出  $\mathbb{R}^3$  空间中的二维平面。当且仅当向量  $b$  在该二维平面中时, 该方程有解。

不等式  $n \geq m$  仅是方程对每一点都有解的必要条件。这不是一个充分条件, 因为有些列向量可能是冗余的。假设有一个  $\mathbb{R}^{2 \times 2}$  中的矩阵, 它的两个列向量是相同的。那么它的列空间和它的一个列向量作为矩阵的列空间是一样的。换言之, 虽然该矩阵有 2 列, 但是它的列空间仍然只是一条线, 不能涵盖整个  $\mathbb{R}^2$  空间。

正式地说, 这种冗余被称为 **线性相关** (linear dependence)。如果一组向量中的任意一个向量都不能表示成其他向量的线性组合, 那么这组向量称为 **线性无关** (linearly independent)。如果某个向量是一组向量中某些向量的线性组合, 那么我们将这个向量加入这组向量后不会增加这组向量的生成子空间。这意味着, 如果一个矩阵的列空间涵盖整个  $\mathbb{R}^m$ , 那么该矩阵必须包含至少一组  $m$  个线性无关的向量。这是式 (2.11) 对于每一个向量  $b$  的取值都有解的充分必要条件。值得注意的是, 这个条件是说该向量集恰好有  $m$  个线性无关的列向量, 而不是至少  $m$  个。不存在一

个  $m$  维向量的集合具有多于  $m$  个彼此线性不相关的列向量，但是一个有多于  $m$  个列向量的矩阵有可能拥有不止一个大小为  $m$  的线性无关向量集。

要想使矩阵可逆，我们还需要保证式 (2.11) 对于每一个  $\mathbf{b}$  值至多有一个解。为此，我们需要确保该矩阵至多有  $m$  个列向量。否则，该方程会有不止一个解。

综上所述，这意味着该矩阵必须是一个 **方阵** (square)，即  $m = n$ ，并且所有列向量都是线性无关的。一个列向量线性相关的方阵被称为 **奇异的** (singular)。

如果矩阵  $\mathbf{A}$  不是一个方阵或者是一个奇异的方阵，该方程仍然可能有解。但是我们不能使用矩阵逆去求解。

目前为止，我们已经讨论了逆矩阵左乘。我们也可以定义逆矩阵右乘：

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}. \quad (2.29)$$

对于方阵而言，它的左逆和右逆是相等的。

## 2.5 范数

有时我们需要衡量一个向量的大小。在机器学习中，我们经常使用被称为 **范数** (norm) 的函数衡量向量大小。形式上， $L^p$  范数定义如下

$$\|\mathbf{x}\|_p = \left( \sum_i |x_i|^p \right)^{\frac{1}{p}} \quad (2.30)$$

其中  $p \in \mathbb{R}$ ,  $p \geq 1$ 。

范数（包括  $L^p$  范数）是将向量映射到非负值的函数。直观上来说，向量  $\mathbf{x}$  的范数衡量从原点到点  $\mathbf{x}$  的距离。更严格地说，范数是满足下列性质的任意函数：

- $f(\mathbf{x}) = 0 \Rightarrow \mathbf{x} = \mathbf{0}$
- $f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y})$  ( **三角不等式** (triangle inequality) )
- $\forall \alpha \in \mathbb{R}, f(\alpha \mathbf{x}) = |\alpha| f(\mathbf{x})$

当  $p = 2$  时， $L^2$  范数被称为 **欧几里得范数** (Euclidean norm)。它表示从原点出发到向量  $\mathbf{x}$  确定的点的欧几里得距离。 $L^2$  范数在机器学习中出现地十分频繁，经

常简化表示为  $\|x\|$ , 略去了下标 2。平方  $L^2$  范数也经常用来衡量向量的大小, 可以简单地通过点积  $x^\top x$  计算。

平方  $L^2$  范数在数学和计算上都比  $L^2$  范数本身更方便。例如, 平方  $L^2$  范数对  $x$  中每个元素的导数只取决于对应的元素, 而  $L^2$  范数对每个元素的导数却和整个向量相关。但是在很多情况下, 平方  $L^2$  范数也可能不受欢迎, 因为它在原点附近增长得十分缓慢。在某些机器学习应用中, 区分恰好是零的元素和非零但值很小的元素是很重要的。在这些情况下, 我们转而使用在各个位置斜率相同, 同时保持简单的数学形式的函数:  $L^1$  范数。 $L^1$  范数可以简化如下:

$$\|x\|_1 = \sum_i |x_i|. \quad (2.31)$$

当机器学习问题中零和非零元素之间的差异非常重要时, 通常会使用  $L^1$  范数。每当  $x$  中某个元素从 0 增加  $\epsilon$ , 对应的  $L^1$  范数也会增加  $\epsilon$ 。

有时候我们会统计向量中非零元素的个数来衡量向量的大小。有些作者将这种函数称为 “ $L^0$  范数”, 但是这个术语在数学意义上是不对的。向量的非零元素的数目不是范数, 因为对向量缩放  $\alpha$  倍不会改变该向量非零元素的数目。因此,  $L^1$  范数经常作为表示非零元素数目的替代函数。

另外一个经常在机器学习中出现的范数是  $L^\infty$  范数, 也被称为 最大范数 (max norm)。这个范数表示向量中具有最大幅值的元素的绝对值:

$$\|x\|_\infty = \max_i |x_i|. \quad (2.32)$$

有时候我们可能也希望衡量矩阵的大小。在深度学习中, 最常见的做法是使用 **Frobenius 范数** (Frobenius norm),

$$\|A\|_F = \sqrt{\sum_{i,j} A_{i,j}^2}, \quad (2.33)$$

其类似于向量的  $L^2$  范数。

两个向量的 点积 (dot product) 可以用范数来表示。具体地,

$$x^\top y = \|x\|_2 \|y\|_2 \cos \theta \quad (2.34)$$

其中  $\theta$  表示  $x$  和  $y$  之间的夹角。

## 2.6 特殊类型的矩阵和向量

有些特殊类型的矩阵和向量是特别有用的。

**对角矩阵** (diagonal matrix) 只在主对角线上含有非零元素，其他位置都是零。形式上，矩阵  $\mathbf{D}$  是对角矩阵，当且仅当对于所有的  $i \neq j$ ,  $D_{i,j} = 0$ 。我们已经看到过一个对角矩阵：单位矩阵，对角元素全部是 1。我们用  $\text{diag}(\mathbf{v})$  表示一个对角元素由向量  $\mathbf{v}$  中元素给定的对角方阵。对角矩阵受到关注的部分原因是对角矩阵的乘法计算很高效。计算乘法  $\text{diag}(\mathbf{v})\mathbf{x}$ ，我们只需要将  $\mathbf{x}$  中的每个元素  $x_i$  放大  $v_i$  倍。换言之， $\text{diag}(\mathbf{v})\mathbf{x} = \mathbf{v} \odot \mathbf{x}$ 。计算对角方阵的逆矩阵也很高效。对角方阵的逆矩阵存在，当且仅当对角元素都是非零值，在这种情况下， $\text{diag}(\mathbf{v})^{-1} = \text{diag}([1/v_1, \dots, 1/v_n]^\top)$ 。在很多情况下，我们可以根据任意矩阵导出一些通用的机器学习算法；但通过将一些矩阵限制为对角矩阵，我们可以得到计算代价较低的（并且简明扼要的）算法。

不是所有的对角矩阵都是方阵。长方形的矩阵也有可能是对角矩阵。非方阵的对角矩阵没有逆矩阵，但我们仍然可以高效地计算它们的乘法。对于一个长方形对角矩阵  $\mathbf{D}$  而言，乘法  $\mathbf{D}\mathbf{x}$  会涉及到  $\mathbf{x}$  中每个元素的缩放，如果  $\mathbf{D}$  是瘦长型矩阵，那么在缩放后的末尾添加一些零；如果  $\mathbf{D}$  是胖宽型矩阵，那么在缩放后去掉最后一些元素。

**对称** (symmetric) 矩阵是转置和自己相等的矩阵：

$$\mathbf{A} = \mathbf{A}^\top. \quad (2.35)$$

当某些不依赖参数顺序的双参数函数生成元素时，对称矩阵经常会出现。例如，如果  $\mathbf{A}$  是一个距离度量矩阵， $A_{i,j}$  表示点  $i$  到点  $j$  的距离，那么  $A_{i,j} = A_{j,i}$ ，因为距离函数是对称的。

**单位向量** (unit vector) 是具有 **单位范数** (unit norm) 的向量：

$$\|\mathbf{x}\|_2 = 1. \quad (2.36)$$

如果  $\mathbf{x}^\top \mathbf{y} = 0$ ，那么向量  $\mathbf{x}$  和向量  $\mathbf{y}$  互相 **正交** (orthogonal)。如果两个向量都有非零范数，那么这两个向量之间的夹角是 90 度。在  $\mathbb{R}^n$  中，至多有  $n$  个范数非零向量互相正交。如果这些向量不仅互相正交，并且范数都为 1，那么我们称它们是 **标准正交** (orthonormal)。

正交矩阵 (orthogonal matrix) 是指行向量和列向量是分别标准正交的方阵：

$$\mathbf{A}^\top \mathbf{A} = \mathbf{A} \mathbf{A}^\top = \mathbf{I}. \quad (2.37)$$

这意味着

$$\mathbf{A}^{-1} = \mathbf{A}^\top, \quad (2.38)$$

所以正交矩阵受到关注是因为求逆计算代价小。我们需要注意正交矩阵的定义。违反直觉的是，正交矩阵的行向量不仅是正交的，还是标准正交的。对于行向量或列向量互相正交但不是标准正交的矩阵，没有对应的专有术语。

## 2.7 特征分解

许多数学对象可以通过将它们分解成多个组成部分或者找到它们的一些属性而更好地理解，这些属性是通用的，而不是由我们选择表示它们的方式产生的。

例如，整数可以分解为质因数。我们可以用十进制或二进制等不同方式表示整数 12，但是  $12 = 2 \times 2 \times 3$  永远是对的。从这个表示中我们可以获得一些有用的信息，比如 12 不能被 5 整除，或者 12 的倍数可以被 3 整除。

正如我们可以通过分解质因数来发现整数的一些内在性质，我们也可以通过分解矩阵来发现矩阵表示成数组元素时不明显的函数性质。

**特征分解** (eigendecomposition) 是使用最广的矩阵分解之一，即我们将矩阵分解成一组特征向量和特征值。

方阵  $\mathbf{A}$  的 **特征向量** (eigenvector) 是指与  $\mathbf{A}$  相乘后相当于对该向量进行缩放的非零向量  $\mathbf{v}$ :

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}. \quad (2.39)$$

标量  $\lambda$  被称为这个特征向量对应的**特征值** (eigenvalue)。（类似地，我们也可以定义**左特征向量** (left eigenvector)  $\mathbf{v}^\top \mathbf{A} = \lambda\mathbf{v}^\top$ ，但是通常我们更关注**右特征向量** (right eigenvector)。）

如果  $\mathbf{v}$  是  $\mathbf{A}$  的特征向量，那么任何缩放后的向量  $s\mathbf{v}$  ( $s \in \mathbb{R}$ ,  $s \neq 0$ ) 也是  $\mathbf{A}$  的特征向量。此外， $s\mathbf{v}$  和  $\mathbf{v}$  有相同的特征值。基于这个原因，通常我们只考虑单位特征向量。

假设矩阵  $\mathbf{A}$  有  $n$  个线性无关的特征向量  $\{\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(n)}\}$ ，对应着特征值  $\{\lambda_1, \dots, \lambda_n\}$ 。我们将特征向量连接成一个矩阵，使得每一列是一个特征向量：

$\mathbf{V} = [\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(n)}]$ . 类似地, 我们也可以将特征值连接成一个向量  $\boldsymbol{\lambda} = [\lambda_1, \dots, \lambda_n]^\top$ 。因此  $\mathbf{A}$  的 特征分解 (eigendecomposition) 可以记作

$$\mathbf{A} = \mathbf{V}\text{diag}(\boldsymbol{\lambda})\mathbf{V}^{-1}. \quad (2.40)$$

我们已经看到了构建具有特定特征值和特征向量的矩阵, 能够使我们在目标方向上延伸空间。然而, 我们也常常希望将矩阵 分解 (decompose) 成特征值和特征向量。这样可以帮助我们分析矩阵的特定性质, 就像质因数分解有助于我们理解整数。

不是每一个矩阵都可以分解成特征值和特征向量。在某些情况下, 特征分解存在, 但是会涉及复数而非实数。幸运的是, 在本书中, 我们通常只需要分解一类有简单分解的矩阵。具体来讲, 每个实对称矩阵都可以分解成实特征向量和实特征值:

$$\mathbf{A} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^\top. \quad (2.41)$$

其中  $\mathbf{Q}$  是  $\mathbf{A}$  的特征向量组成的正交矩阵,  $\boldsymbol{\Lambda}$  是对角矩阵。特征值  $\Lambda_{i,i}$  对应的特征向量是矩阵  $\mathbf{Q}$  的第  $i$  列, 记作  $\mathbf{Q}_{:,i}$ 。因为  $\mathbf{Q}$  是正交矩阵, 我们可以将  $\mathbf{A}$  看作沿方向  $\mathbf{v}^{(i)}$  延展  $\lambda_i$  倍的空间。如图 2.3 所示的例子。

虽然任意一个实对称矩阵  $\mathbf{A}$  都有特征分解, 但是特征分解可能并不唯一。如果两个或多个特征向量拥有相同的特征值, 那么在由这些特征向量产生的生成子空间中, 任意一组正交向量都是该特征值对应的特征向量。因此, 我们可以等价地从这些特征向量中构成  $\mathbf{Q}$  作为替代。按照惯例, 我们通常按降序排列  $\boldsymbol{\Lambda}$  的元素。在该约定下, 特征分解唯一当且仅当所有的特征值都是唯一的。

矩阵的特征分解给了我们很多关于矩阵的有用信息。矩阵是奇异的当且仅当含有零特征值。实对称矩阵的特征分解也可以用于优化二次方程  $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{A} \mathbf{x}$ , 其中限制  $\|\mathbf{x}\|_2 = 1$ 。当  $\mathbf{x}$  等于  $\mathbf{A}$  的某个特征向量时,  $f$  将返回对应的特征值。在限制条件下, 函数  $f$  的最大值是最大特征值, 最小值是最小特征值。

所有特征值都是正数的矩阵被称为 正定 (positive definite); 所有特征值都是非负数的矩阵被称为 半正定 (positive semidefinite)。同样地, 所有特征值都是负数的矩阵被称为 负定 (negative definite); 所有特征值都是非正数的矩阵被称为 半负定 (negative semidefinite)。半正定矩阵受到关注是因为它们保证  $\forall \mathbf{x}, \mathbf{x}^\top \mathbf{A} \mathbf{x} \geq 0$ 。此外, 正定矩阵还保证  $\mathbf{x}^\top \mathbf{A} \mathbf{x} = 0 \Rightarrow \mathbf{x} = \mathbf{0}$ 。

Effect of eigenvectors and eigenvalues

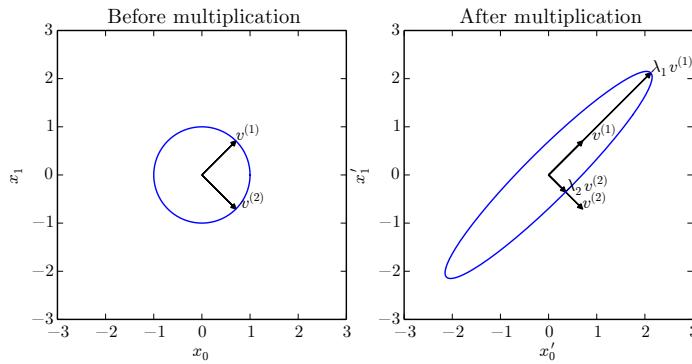


图 2.3: 特征向量和特征值的作用效果。特征向量和特征值的作用效果的一个实例。在这里，矩阵  $\mathbf{A}$  有两个标准正交的特征向量，对应特征值为  $\lambda_1$  的  $\mathbf{v}^{(1)}$  以及对应特征值为  $\lambda_2$  的  $\mathbf{v}^{(2)}$ 。(左) 我们画出了所有的单位向量  $\mathbf{u} \in \mathbb{R}^2$  的集合，构成一个单位圆。(右) 我们画出了所有的  $\mathbf{Au}$  点的集合。通过观察  $\mathbf{A}$  拉伸单位圆的方式，我们可以看到它将  $\mathbf{v}^{(i)}$  方向的空间拉伸了  $\lambda_i$  倍。

## 2.8 奇异值分解

在第 2.7 节，我们探讨了如何将矩阵分解成特征向量和特征值。还有另一种分解矩阵的方法，被称为 **奇异值分解** (singular value decomposition, SVD)，将矩阵分解为 **奇异向量** (singular vector) 和 **奇异值** (singular value)。通过奇异值分解，我们会得到一些与特征分解相同类型的信息。然而，奇异值分解有更广泛的应用。每个实数矩阵都有一个奇异值分解，但不一定都有特征分解。例如，非方阵的矩阵没有特征分解，这时我们只能使用奇异值分解。

回想一下，我们使用特征分解去分析矩阵  $\mathbf{A}$  时，得到特征向量构成的矩阵  $\mathbf{V}$  和特征值构成的向量  $\boldsymbol{\lambda}$ ，我们可以重新将  $\mathbf{A}$  写作

$$\mathbf{A} = \mathbf{V} \text{diag}(\boldsymbol{\lambda}) \mathbf{V}^{-1}. \quad (2.42)$$

奇异值分解是类似的，只不过这回我们将矩阵  $\mathbf{A}$  分解成三个矩阵的乘积：

$$\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^\top. \quad (2.43)$$

假设  $\mathbf{A}$  是一个  $m \times n$  的矩阵，那么  $\mathbf{U}$  是一个  $m \times m$  的矩阵， $\mathbf{D}$  是一个  $m \times n$  的矩阵， $\mathbf{V}$  是一个  $n \times n$  矩阵。

这些矩阵中的每一个经定义后都拥有特殊的结构。矩阵  $\mathbf{U}$  和  $\mathbf{V}$  都定义为正交矩阵，而矩阵  $\mathbf{D}$  定义为对角矩阵。注意，矩阵  $\mathbf{D}$  不一定是方阵。

对角矩阵  $\mathbf{D}$  对角线上的元素被称为矩阵  $\mathbf{A}$  的 **奇异值** (singular value)。矩阵  $\mathbf{U}$  的列向量被称为**左奇异向量** (left singular vector)，矩阵  $\mathbf{V}$  的列向量被称为**右奇异向量** (right singular vector)。

事实上，我们可以用与  $\mathbf{A}$  相关的特征分解去解释  $\mathbf{A}$  的奇异值分解。 $\mathbf{A}$  的**左奇异向量** (left singular vector) 是  $\mathbf{A}\mathbf{A}^\top$  的特征向量。 $\mathbf{A}$  的**右奇异向量** (right singular vector) 是  $\mathbf{A}^\top\mathbf{A}$  的特征向量。 $\mathbf{A}$  的非零奇异值是  $\mathbf{A}^\top\mathbf{A}$  特征值的平方根，同时也是  $\mathbf{A}\mathbf{A}^\top$  特征值的平方根。

SVD最有用的一个性质可能是拓展矩阵求逆到非方矩阵上。我们将在下一节中探讨。

## 2.9 Moore-Penrose 伪逆

对于非方矩阵而言，其逆矩阵没有定义。假设在下面的问题中，我们希望通过矩阵  $\mathbf{A}$  的左逆  $\mathbf{B}$  来求解线性方程，

$$\mathbf{A}\mathbf{x} = \mathbf{y} \quad (2.44)$$

等式两边左乘左逆  $\mathbf{B}$  后，我们得到

$$\mathbf{x} = \mathbf{B}\mathbf{y}. \quad (2.45)$$

取决于问题的形式，我们可能无法设计一个唯一的映射将  $\mathbf{A}$  映射到  $\mathbf{B}$ 。

如果矩阵  $\mathbf{A}$  的行数大于列数，那么上述方程可能没有解。如果矩阵  $\mathbf{A}$  的行数小于列数，那么上述矩阵可能有多个解。

**Moore-Penrose 伪逆** (Moore-Penrose pseudoinverse) 使我们在这类问题上取得了一定的进展。矩阵  $\mathbf{A}$  的伪逆定义为：

$$\mathbf{A}^+ = \lim_{\alpha \searrow 0} (\mathbf{A}^\top \mathbf{A} + \alpha \mathbf{I})^{-1} \mathbf{A}^\top. \quad (2.46)$$

计算伪逆的实际算法没有基于这个定义，而是使用下面的公式：

$$\mathbf{A}^+ = \mathbf{V} \mathbf{D}^+ \mathbf{U}^\top. \quad (2.47)$$

其中，矩阵  $\mathbf{U}$ ,  $\mathbf{D}$  和  $\mathbf{V}$  是矩阵  $\mathbf{A}$  奇异值分解后得到的矩阵。对角矩阵  $\mathbf{D}$  的伪逆  $\mathbf{D}^+$  是其非零元素取倒数之后再转置得到的。

当矩阵  $\mathbf{A}$  的列数多于行数时，使用伪逆求解线性方程是众多可能解法中的一种。特别地， $\mathbf{x} = \mathbf{A}^+ \mathbf{y}$  是方程所有可行解中欧几里得范数  $\|\mathbf{x}\|_2$  最小的一个。

当矩阵  $\mathbf{A}$  的行数多于列数时，可能没有解。在这种情况下，通过伪逆得到的  $\mathbf{x}$  使得  $\mathbf{Ax}$  和  $\mathbf{y}$  的欧几里得距离  $\|\mathbf{Ax} - \mathbf{y}\|_2$  最小。

## 2.10 迹运算

迹运算返回的是矩阵对角元素的和：

$$\text{Tr}(\mathbf{A}) = \sum_i \mathbf{A}_{i,i}. \quad (2.48)$$

迹运算因为很多原因而有用。若不使用求和符号，有些矩阵运算很难描述，而通过矩阵乘法和迹运算符号可以清楚地表示。例如，迹运算提供了另一种描述矩阵Frobenius范数的方式：

$$\|\mathbf{A}\|_F = \sqrt{\text{Tr}(\mathbf{A} \mathbf{A}^\top)}. \quad (2.49)$$

用迹运算表示表达式，我们可以使用很多有用的等式巧妙地处理表达式。例如，迹运算在转置运算下是不变的：

$$\text{Tr}(\mathbf{A}) = \text{Tr}(\mathbf{A}^\top). \quad (2.50)$$

多个矩阵相乘得到的方阵的迹，和将这些矩阵中的最后一个挪到最前面之后相乘的迹是相同的。当然，我们需要考虑挪动之后矩阵乘积依然定义良好：

$$\text{Tr}(\mathbf{ABC}) = \text{Tr}(\mathbf{CAB}) = \text{Tr}(\mathbf{BCA}). \quad (2.51)$$

或者更一般地，

$$\mathrm{Tr}\left(\prod_{i=1}^n \mathbf{F}^{(i)}\right) = \mathrm{Tr}(\mathbf{F}^{(n)} \prod_{i=1}^{n-1} \mathbf{F}^{(i)}). \quad (2.52)$$

即使循环置换后矩阵乘积得到的矩阵形状变了，迹运算的结果依然不变。例如，假设矩阵  $\mathbf{A} \in \mathbb{R}^{m \times n}$ , 矩阵  $\mathbf{B} \in \mathbb{R}^{n \times m}$ , 我们可以得到

$$\mathrm{Tr}(\mathbf{AB}) = \mathrm{Tr}(\mathbf{BA}) \quad (2.53)$$

尽管  $\mathbf{AB} \in \mathbb{R}^{m \times m}$  和  $\mathbf{BA} \in \mathbb{R}^{n \times n}$ 。

另一个有用的事是标量在迹运算后仍然是它自己： $a = \mathrm{Tr}(a)$ 。

## 2.11 行列式

行列式，记作  $\det(\mathbf{A})$ ，是一个将方阵  $\mathbf{A}$  映射到实数的函数。行列式等于矩阵特征值的乘积。行列式的绝对值可以用来衡量矩阵参与矩阵乘法后空间扩大或者缩小了多少。如果行列式是 0，那么空间至少沿着某一维完全收缩了，使其失去了所有的体积。如果行列式是 1，那么这个转换保持空间体积不变。

## 2.12 实例：主成分分析

主成分分析（principal components analysis, PCA）是一个简单的机器学习算法，可以通过基础的线性代数知识推导。

假设在  $\mathbb{R}^n$  空间中我们有  $m$  个点  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ ，我们希望对这些点进行有损压缩。有损压缩表示我们使用更少的内存，但损失一些精度去存储这些点。我们希望损失的精度尽可能少。

一种编码这些点的方式是用低维表示。对于每个点  $\mathbf{x}^{(i)} \in \mathbb{R}^n$ ，会有一个对应的编码向量  $\mathbf{c}^{(i)} \in \mathbb{R}^l$ 。如果  $l$  比  $n$  小，那么我们便使用了更少的内存来存储原来的数据。我们希望找到一个编码函数，根据输入返回编码， $f(\mathbf{x}) = \mathbf{c}$ ；我们也希望找到一个解码函数，给定编码重构输入， $\mathbf{x} \approx g(f(\mathbf{x}))$ 。

PCA 由我们选择的解码函数而定。具体地，为了简化解码器，我们使用矩阵乘法将编码映射回  $\mathbb{R}^n$ ，即  $g(\mathbf{c}) = \mathbf{D}\mathbf{c}$ ，其中  $\mathbf{D} \in \mathbb{R}^{n \times l}$  是定义解码的矩阵。

目前为止所描述的问题，可能会有多个解。因为如果我们按比例地缩小所有点对应的编码向量  $c_i$ ，那么我们只需按比例放大  $D_{:,i}$ ，即可保持结果不变。为了使问题有唯一解，我们限制  $D$  中所有列向量都有单位范数。

计算这个解码器的最优编码可能是一个困难的问题。为了使编码问题简单一些，PCA 限制  $D$  的列向量彼此正交（注意，除非  $l = n$ ，否则严格意义上  $D$  不是一个正交矩阵）。

为了将这个基本想法变为我们能够实现的算法，首先我们需要明确如何根据每一个输入  $x$  得到一个最优编码  $c^*$ 。一种方法是最小化原始输入向量  $x$  和重构向量  $g(c^*)$  之间的距离。我们使用范数来衡量它们之间的距离。在 PCA 算法中，我们使用  $L^2$  范数：

$$c^* = \arg \min_c \|x - g(c)\|_2. \quad (2.54)$$

我们可以用平方  $L^2$  范数替代  $L^2$  范数，因为两者在相同的值  $c$  上取得最小值。这是因为  $L^2$  范数是非负的，并且平方运算在非负值上是单调递增的。

$$c^* = \arg \min_c \|x - g(c)\|_2^2. \quad (2.55)$$

该最小化函数可以简化成

$$(x - g(c))^\top (x - g(c)) \quad (2.56)$$

(式(2.30)中  $L^2$  范数的定义)

$$= x^\top x - x^\top g(c) - g(c)^\top x + g(c)^\top g(c) \quad (2.57)$$

(分配律)

$$= x^\top x - 2x^\top g(c) + g(c)^\top g(c) \quad (2.58)$$

(因为标量  $g(c)^\top x$  的转置等于自己)

因为第一项  $x^\top x$  不依赖于  $c$ ，所以我们可以忽略它，得到如下的优化目标：

$$c^* = \arg \min_c - 2x^\top g(c) + g(c)^\top g(c). \quad (2.59)$$

更进一步，我们代入  $g(c)$  的定义：

$$c^* = \arg \min_c - 2x^\top Dc + c^\top D^\top Dc \quad (2.60)$$

$$= \arg \min_c - 2x^\top Dc + c^\top I_l c \quad (2.61)$$

(矩阵  $\mathbf{D}$  的正交性和单位范数约束)

$$= \arg \min_{\mathbf{c}} -2\mathbf{x}^\top \mathbf{D}\mathbf{c} + \mathbf{c}^\top \mathbf{c} \quad (2.62)$$

我们可以通过向量微积分来求解这个最优化问题（如果你不清楚怎么做，请参考第 4.3 节）

$$\nabla_{\mathbf{c}}(-2\mathbf{x}^\top \mathbf{D}\mathbf{c} + \mathbf{c}^\top \mathbf{c}) = 0 \quad (2.63)$$

$$-2\mathbf{D}^\top \mathbf{x} + 2\mathbf{c} = 0 \quad (2.64)$$

$$\mathbf{c} = \mathbf{D}^\top \mathbf{x}. \quad (2.65)$$

这使得算法很高效：最优编码  $\mathbf{x}$  只需要一个矩阵-向量乘法操作。为了编码向量，我们使用编码函数：

$$f(\mathbf{x}) = \mathbf{D}^\top \mathbf{x}. \quad (2.66)$$

进一步使用矩阵乘法，我们也可以定义 PCA 重构操作：

$$r(\mathbf{x}) = g(f(\mathbf{x})) = \mathbf{D}\mathbf{D}^\top \mathbf{x}. \quad (2.67)$$

接下来，我们需要挑选编码矩阵  $\mathbf{D}$ 。要做到这一点，我们回顾最小化输入和重构之间  $L^2$  距离的这个想法。因为用相同的矩阵  $\mathbf{D}$  对所有点进行解码，我们不能再孤立地看待每个点。反之，我们必须最小化所有维数和所有点上的误差矩阵的 Frobenius 范数：

$$\mathbf{D}^* = \arg \min_{\mathbf{D}} \sqrt{\sum_{i,j} \left( \mathbf{x}_j^{(i)} - r(\mathbf{x}^{(i)})_j \right)^2} \text{ subject to } \mathbf{D}^\top \mathbf{D} = \mathbf{I}_l. \quad (2.68)$$

为了推导用于寻求  $\mathbf{D}^*$  的算法，我们首先考虑  $l = 1$  的情况。在这种情况下， $\mathbf{D}$  是一个单一向量  $\mathbf{d}$ 。将式 (2.67) 代入式 (2.68)，简化  $\mathbf{D}$  为  $\mathbf{d}$ ，问题简化为

$$\mathbf{d}^* = \arg \min_{\mathbf{d}} \sum_i \left\| \mathbf{x}^{(i)} - \mathbf{d}\mathbf{d}^\top \mathbf{x}^{(i)} \right\|_2^2 \text{ subject to } \|\mathbf{d}\|_2 = 1. \quad (2.69)$$

上述公式是直接代入得到的，但不是文体表述最舒服的方式。在上述公式中，我们将标量  $\mathbf{d}^\top \mathbf{x}^{(i)}$  放在向量  $\mathbf{d}$  的右边。将该标量放在左边的写法更为传统。于是我们通常写作

$$\mathbf{d}^* = \arg \min_{\mathbf{d}} \sum_i \left\| \mathbf{x}^{(i)} - \mathbf{d}^\top \mathbf{x}^{(i)} \mathbf{d} \right\|_2^2 \text{ subject to } \|\mathbf{d}\|_2 = 1, \quad (2.70)$$

或者，考虑到标量的转置和自身相等，我们也可以写作

$$\mathbf{d}^* = \arg \min_{\mathbf{d}} \sum_i \left\| \mathbf{x}^{(i)} - \mathbf{x}^{(i)\top} \mathbf{d} \mathbf{d} \right\|_2^2 \text{ subject to } \|\mathbf{d}\|_2 = 1. \quad (2.71)$$

读者应该对这些重排写法慢慢熟悉起来。

此时，使用单一矩阵来重述问题，比将问题写成求和形式更有帮助。这有助于我们使用更紧凑的符号。将表示各点的向量堆叠成一个矩阵，记为  $\mathbf{X} \in \mathbb{R}^{m \times n}$ ，其中  $\mathbf{X}_{i,:} = \mathbf{x}^{(i)\top}$ 。原问题可以重新表述为：

$$\mathbf{d}^* = \arg \min_{\mathbf{d}} \left\| \mathbf{X} - \mathbf{X} \mathbf{d} \mathbf{d}^\top \right\|_F^2 \text{ subject to } \mathbf{d}^\top \mathbf{d} = 1. \quad (2.72)$$

暂时不考虑约束，我们可以将 Frobenius 范数简化成下面的形式：

$$\arg \min_{\mathbf{d}} \left\| \mathbf{X} - \mathbf{X} \mathbf{d} \mathbf{d}^\top \right\|_F^2 \quad (2.73)$$

$$= \arg \min_{\mathbf{d}} \text{Tr} \left( (\mathbf{X} - \mathbf{X} \mathbf{d} \mathbf{d}^\top)^\top (\mathbf{X} - \mathbf{X} \mathbf{d} \mathbf{d}^\top) \right) \quad (2.74)$$

(式(2.49))

$$= \arg \min_{\mathbf{d}} \text{Tr} \left( \mathbf{X}^\top \mathbf{X} - \mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top - \mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X} + \mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top \right) \quad (2.75)$$

$$= \arg \min_{\mathbf{d}} \text{Tr}(\mathbf{X}^\top \mathbf{X}) - \text{Tr}(\mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) - \text{Tr}(\mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X}) + \text{Tr}(\mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) \quad (2.76)$$

$$= \arg \min_{\mathbf{d}} -\text{Tr}(\mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) - \text{Tr}(\mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X}) + \text{Tr}(\mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) \quad (2.77)$$

(因为与  $\mathbf{d}$  无关的项不影响  $\arg \min$ )

$$= \arg \min_{\mathbf{d}} -2\text{Tr}(\mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) + \text{Tr}(\mathbf{d} \mathbf{d}^\top \mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) \quad (2.78)$$

(因为循环改变迹运算中相乘矩阵的顺序不影响结果，如式(2.52)所示)

$$= \arg \min_{\mathbf{d}} -2\text{Tr}(\mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) + \text{Tr}(\mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top \mathbf{d} \mathbf{d}^\top) \quad (2.79)$$

(再次使用上述性质)

此时，我们再来考虑约束条件：

$$\arg \min_{\mathbf{d}} -2\text{Tr}(\mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top) + \text{Tr}(\mathbf{X}^\top \mathbf{X} \mathbf{d} \mathbf{d}^\top \mathbf{d} \mathbf{d}^\top) \text{ subject to } \mathbf{d}^\top \mathbf{d} = 1 \quad (2.80)$$

$$= \arg \min_d -2\text{Tr}(\mathbf{X}^\top \mathbf{X} d d^\top) + \text{Tr}(\mathbf{X}^\top \mathbf{X} d d^\top) \text{ subject to } d^\top d = 1 \quad (2.81)$$

(因为约束条件)

$$= \arg \min_d -\text{Tr}(\mathbf{X}^\top \mathbf{X} d d^\top) \text{ subject to } d^\top d = 1 \quad (2.82)$$

$$= \arg \max_d \text{Tr}(\mathbf{X}^\top \mathbf{X} d d^\top) \text{ subject to } d^\top d = 1 \quad (2.83)$$

$$= \arg \max_d \text{Tr}(\mathbf{d}^\top \mathbf{X}^\top \mathbf{X} \mathbf{d}) \text{ subject to } d^\top d = 1. \quad (2.84)$$

这个优化问题可以通过特征分解来求解。具体来讲，最优的  $\mathbf{d}$  是  $\mathbf{X}^\top \mathbf{X}$  最大特征值对应的特征向量。

以上推导特定于  $l = 1$  的情况，仅得到了第一个主成分。更一般地，当我们希望得到主成分的基时，矩阵  $\mathbf{D}$  由前  $l$  个最大的特征值对应的特征向量组成。这个结论可以通过归纳法证明，我们建议将此证明作为练习。

线性代数是理解深度学习所必须掌握的基础数学学科之一。另一门在机器学习中无处不在的重要数学学科是概率论，我们将在下一章探讨。

# 第三章 概率与信息论

本章我们讨论概率论和信息论。

概率论是用于表示不确定性声明的数学框架。它不仅提供了量化不确定性的方法，也提供了用于导出新的不确定性声明（statement）的公理。在人工智能领域，概率论主要有两种用途。首先，概率法则告诉我们 AI 系统如何推理，据此我们设计一些算法来计算或者估算由概率论导出的表达式。其次，我们可以用概率和统计从理论上分析我们提出的 AI 系统的行为。

概率论是众多科学学科和工程学科的基本工具。我们提供这一章，是为了确保那些背景偏软件工程而较少接触概率论的读者也可以理解本书的内容。

概率论使我们能够提出不确定的声明以及在不确定性存在的情况下进行推理，而信息论使我们能够量化概率分布中的不确定性总量。

如果你已经对概率论和信息论很熟悉了，那么除了第 3.14 节以外的整章内容，你都可以跳过。而在第 3.14 节中，我们会介绍用来描述机器学习中结构化概率模型的图。即使你对这些主题没有任何的先验知识，本章对于完成深度学习的研究项目来说也已经足够，尽管如此我们还是建议你能够参考一些额外的资料，例如 Jaynes (2003)。

## 3.1 为什么要使用概率？

计算机科学的许多分支处理的实体大部分都是完全确定且必然的。程序员通常可以安全地假定 CPU 将完美地执行每条机器指令。虽然硬件错误确实会发生，但它们足够罕见，以致于大部分软件应用在设计时并不需要考虑这些因素的影响。鉴于许多计算机科学家和软件工程师在一个相对干净和确定的环境中工作，机器学习对

于概率论的大量使用是很令人吃惊的。

这是因为机器学习通常必须处理不确定量，有时也可能需要处理随机（非确定性的）量。不确定性和随机性可能来自多个方面。至少从 20 世纪 80 年代开始，研究人员就对使用概率论来量化不确定性提出了令人信服的论据。这里给出的许多论据都是根据 Pearl (1988) 的工作总结或启发得到的。

几乎所有的活动都需要一些在不确定性存在的情况下进行推理的能力。事实上，除了那些被定义为真的数学声明，我们很难认定某个命题是千真万确的或者确保某件事一定会发生。

不确定性有三种可能的来源：

1. 被建模系统内在的随机性。例如，大多数量子力学的解释，都将亚原子粒子的动力学描述为概率的。我们还可以创建一些我们假设具有随机动态的理论情境，例如一个假想的纸牌游戏，在这个游戏中我们假设纸牌被真正混洗成了随机顺序。
2. 不完全观测。即使是确定的系统，当我们不能观测到所有驱动系统行为的变量时，该系统也会呈现随机性。例如，在 Monty Hall 问题中，一个游戏节目的参与者被要求在三个门之间选择，并且会赢得放置在选中门后的奖品。其中两扇门通向山羊，第三扇门通向一辆汽车。选手的每个选择所导致的结果是确定的，但是站在选手的角度，结果是不确定的。
3. 不完全建模。当我们使用一些必须舍弃某些观测信息的模型时，舍弃的信息会导致模型的预测出现不确定性。例如，假设我们制作了一个机器人，它可以准确地观察周围每一个对象的位置。在对这些对象将来的位置进行预测时，如果机器人采用的是离散化的空间，那么离散化的方法将使得机器人无法确定对象们的精确位置：因为每个对象都可能处于它被观测到的离散单元的任何一个角落。

在很多情况下，使用一些简单而不确定的规则要比复杂而确定的规则更为实用，即使真正的规则是确定的并且我们建模的系统可以足够精确地容纳复杂的规则。例如，“多数鸟儿都会飞”这个简单的规则描述起来很简单很并且使用广泛，而正式的规则——“除了那些还没学会飞翔的幼鸟，因为生病或是受伤而失去了飞翔能力的鸟，包括食火鸟 (cassowary)、鸵鸟 (ostrich)、几维 (kiwi, 一种新西兰产的无翼鸟)

等不会飞的鸟类……以外，鸟儿会飞”，很难应用、维护和沟通，即使经过这么多的努力，这个规则还是很脆弱而且容易失效。

尽管我们的确需要一种用以对不确定性进行表示和推理的方法，但是概率论并不能明显地提供我们在人工智能领域需要的所有工具。概率论最初的发展是为了分析事件发生的频率。我们可以很容易地看出概率论，对于像在扑克牌游戏中抽出一手特定的牌这种事件的研究中，是如何使用的。这类事件往往是可以重复的。当我们说一个结果发生的概率为  $p$ ，这意味着如果我们反复实验（例如，抽取一手牌）无限次，有  $p$  的比例可能会导致这样的结果。这种推理似乎并不立即适用于那些不可重复的命题。如果一个医生诊断了病人，并说该病人患流感的几率为 40%，这意味着非常不同的事情——我们既不能让病人有无穷多的副本，也没有任何理由去相信病人的不同副本在具有不同的潜在条件下表现出相同的症状。在医生诊断病人的例子中，我们用概率来表示一种 **信任度**（degree of belief），其中 1 表示非常肯定病人患有流感，而 0 表示非常肯定病人没有流感。前面那种概率，直接与事件发生的频率相联系，被称为 **频率派概率**（frequentist probability）；而后者，涉及到确定性水平，被称为 **贝叶斯概率**（Bayesian probability）。

关于不确定性的常识推理，如果我们已经列出了若干条我们期望它具有的性质，那么满足这些性质的唯一一种方法就是将贝叶斯概率和频率派概率视为等同的。例如，如果我们要在扑克牌游戏中根据玩家手上的牌计算她能够获胜的概率，我们使用和医生情境完全相同的公式，就是我们依据病人的某些症状计算她是否患病的概率。为什么一小组常识性假设蕴含了必须是相同的公理控制两种概率？更多的细节参见 Ramsey (1926)。

概率可以被看作是用于处理不确定性的逻辑扩展。逻辑提供了一套形式化的规则，可以在给定某些命题是真或假的假设下，判断另外一些命题是真的还是假的。概率论提供了一套形式化的规则，可以在给定一些命题的似然后，计算其他命题为真的似然。

## 3.2 随机变量

**随机变量**（random variable）是可以随机地取不同值的变量。我们通常用无格式字体（plain typeface）中的小写字母来表示随机变量本身，而用手写体中的小写字母来表示随机变量能够取到的值。例如， $x_1$  和  $x_2$  都是随机变量  $x$  可能的取值。对

于向量值变量，我们会将随机变量写成  $x$ ，它的一个可能取值为  $x$ 。就其本身而言，一个随机变量只是对可能的状态的描述；它必须伴随着一个概率分布来指定每个状态的可能性。

随机变量可以是离散的或者连续的。离散随机变量拥有有限或者可数无限多的状态。注意这些状态不一定非要是整数；它们也可能只是一些被命名的状态而没有数值。连续随机变量伴随着实数值。

### 3.3 概率分布

概率分布 (probability distribution) 用来描述随机变量或一族随机变量在每一个可能取到的状态的可能性大小。我们描述概率分布的方式取决于随机变量是离散的还是连续的。

#### 3.3.1 离散型变量和概率质量函数

离散型变量的概率分布可以用概率质量函数 (probability mass function, PMF)<sup>1</sup> 来描述。我们通常用大写字母  $P$  来表示概率质量函数。通常每一个随机变量都会有一个不同的概率质量函数，并且读者必须根据随机变量来推断所使用的 PMF，而不是根据函数的名称来推断；例如， $P(x)$  通常和  $P(y)$  不一样。

概率质量函数将随机变量能够取得的每个状态映射到随机变量取得该状态的概率。 $x = x$  的概率用  $P(x)$  来表示，概率为 1 表示  $x = x$  是确定的，概率为 0 表示  $x = x$  是不可能发生的。有时为了使得 PMF 的使用不相互混淆，我们会明确写出随机变量的名称： $P(x = x)$ 。有时我们会先定义一个随机变量，然后用  $\sim$  符号来说明它遵循的分布： $x \sim P(x)$ 。

概率质量函数可以同时作用于多个随机变量。这种多个变量的概率分布被称为联合概率分布 (joint probability distribution)。 $P(x = x, y = y)$  表示  $x = x$  和  $y = y$  同时发生的概率。我们也可以简写为  $P(x, y)$ 。

如果一个函数  $P$  是随机变量  $x$  的 PMF，必须满足下面这几个条件：

- $P$  的定义域必须是  $x$  所有可能状态的集合。

---

<sup>1</sup>译者注：国内有些教材也将它翻译成概率分布律。

- $\forall x \in \mathbf{x}, 0 \leq P(x) \leq 1$ . 不可能发生的事件概率为 0，并且不存在比这概率更低的状态。类似的，能够确保一定发生的事件概率为 1，而且不存在比这概率更高的状态。
- $\sum_{x \in \mathbf{x}} P(x) = 1$ . 我们把这条性质称之为 **归一化**的 (normalized)。如果没有这条性质，当我们计算很多事件其中之一发生的概率时可能会得到大于 1 的概率。

例如，考虑一个离散型随机变量  $\mathbf{x}$  有  $k$  个不同的状态。我们可以假设  $\mathbf{x}$  是 **均匀分布** (uniform distribution) 的（也就是将它的每个状态视为等可能的），通过将它的PMF设为

$$P(\mathbf{x} = x_i) = \frac{1}{k} \quad (3.1)$$

对于所有的  $i$  都成立。我们可以看出这满足上述成为概率质量函数的条件。因为  $k$  是一个正整数，所以  $\frac{1}{k}$  是正的。我们也可以看出

$$\sum_i P(\mathbf{x} = x_i) = \sum_i \frac{1}{k} = \frac{k}{k} = 1, \quad (3.2)$$

因此分布也满足归一化条件。

### 3.3.2 连续型变量和概率密度函数

当我们研究的对象是连续型随机变量时，我们用 **概率密度函数** (probability density function, PDF) 而不是概率质量函数来描述它的概率分布。如果一个函数  $p$  是概率密度函数，必须满足下面这几个条件：

- $p$  的定义域必须是  $\mathbf{x}$  所有可能状态的集合。
- $\forall x \in \mathbf{x}, p(x) \geq 0$ . 注意，我们并不要求  $p(x) \leq 1$ 。
- $\int p(x)dx = 1$ .

概率密度函数  $p(x)$  并没有直接对特定的状态给出概率，相对的，它给出了落在面积为  $\delta x$  的无限小的区域内的概率为  $p(x)\delta x$ 。

我们可以对概率密度函数求积分来获得点集的真实概率质量。特别地， $x$  落在集合  $\mathbb{S}$  中的概率可以通过  $p(x)$  对这个集合求积分来得到。在单变量的例子中， $x$  落在区间  $[a, b]$  的概率是  $\int_{[a,b]} p(x)dx$ 。

为了给出一个连续型随机变量的 PDF 的例子，我们可以考虑实数区间上的均匀分布。我们可以使用函数  $u(x; a, b)$ ，其中  $a$  和  $b$  是区间的端点且满足  $b > a$ 。符号“;”表示“以什么为参数”；我们把  $x$  作为函数的自变量， $a$  和  $b$  作为定义函数的参数。为了确保区间外没有概率，我们对所有的  $x \notin [a, b]$ ，令  $u(x; a, b) = 0$ 。在  $[a, b]$  内，有  $u(x; a, b) = \frac{1}{b-a}$ 。我们可以看出任何一点都非负。另外，它的积分为 1。我们通常用  $x \sim U(a, b)$  表示  $x$  在  $[a, b]$  上是均匀分布的。

## 3.4 边缘概率

有时候，我们知道了一组变量的联合概率分布，但想要了解其中一个子集的概率分布。这种定义在子集上的概率分布被称为 **边缘概率分布** (marginal probability distribution)。

例如，假设有离散型随机变量  $x$  和  $y$ ，并且我们知道  $P(x, y)$ 。我们可以依据下面的 **求和法则** (sum rule) 来计算  $P(x)$ :

$$\forall x \in X, P(x = x) = \sum_y P(x = x, y = y). \quad (3.3)$$

“边缘概率”的名称来源于手算边缘概率的计算过程。当  $P(x, y)$  的每个值被写在由每行表示不同的  $x$  值，每列表示不同的  $y$  值形成的网格中时，对网格中的每行求和是很自然的事情，然后将求和的结果  $P(x)$  写在每行右边的纸的边缘处。

对于连续型变量，我们需要用积分替代求和：

$$p(x) = \int p(x, y) dy. \quad (3.4)$$

## 3.5 条件概率

在很多情况下，我们感兴趣的是某个事件，在给定其他事件发生时出现的概率。这种概率叫做条件概率。我们将给定  $x = x$ ,  $y = y$  发生的条件概率记为  $P(y = y | x = x)$ 。这个条件概率可以通过下面的公式计算：

$$P(y = y | x = x) = \frac{P(y = y, x = x)}{P(x = x)}. \quad (3.5)$$

条件概率只在  $P(x = x) > 0$  时有定义。我们不能计算给定在永远不会发生的事件上的条件概率。

这里需要注意的是，不要把条件概率和计算当采用某个动作后会发生什么相混淆。假定某个人说德语，那么他是德国人的条件概率是非常高的，但是如果随机选择的一个人会说德语，他的国籍不会因此而改变。计算一个行动的后果被称为 **干预查询** ( intervention query )。干预查询属于 **因果模型** ( causal modeling ) 的范畴，我们不会在本书中讨论。

## 3.6 条件概率的链式法则

任何多维随机变量的联合概率分布，都可以分解成只有一个变量的条件概率相乘的形式：

$$P(x^{(1)}, \dots, x^{(n)}) = P(x^{(1)}) \prod_{i=2}^n P(x^{(i)} | x^{(1)}, \dots, x^{(i-1)}). \quad (3.6)$$

这个规则被称为概率的 **链式法则** ( chain rule ) 或者 **乘法法则** ( product rule )。它可以直接从式 (3.5) 条件概率的定义中得到。例如，使用两次定义可以得到

$$\begin{aligned} P(a, b, c) &= P(a | b, c)P(b, c) \\ P(b, c) &= P(b | c)P(c) \\ P(a, b, c) &= P(a | b, c)P(b | c)P(c). \end{aligned}$$

## 3.7 独立性和条件独立性

两个随机变量  $x$  和  $y$ ，如果它们的概率分布可以表示成两个因子的乘积形式，并且一个因子只包含  $x$  另一个因子只包含  $y$ ，我们就称这两个随机变量是 **相互独立的** ( independent )：

$$\forall x \in X, y \in Y, p(x = x, y = y) = p(x = x)p(y = y). \quad (3.7)$$

如果关于  $x$  和  $y$  的条件概率分布对于  $z$  的每一个值都可以写成乘积的形式，那么这两个随机变量  $x$  和  $y$  在给定随机变量  $z$  时是 **条件独立的** ( conditionally

independent ):

$$\forall x \in \mathbf{x}, y \in \mathbf{y}, z \in \mathbf{z}, p(\mathbf{x} = x, \mathbf{y} = y | \mathbf{z} = z) = p(\mathbf{x} = x | \mathbf{z} = z)p(\mathbf{y} = y | \mathbf{z} = z). \quad (3.8)$$

我们可以采用一种简化形式来表示独立性和条件独立性:  $\mathbf{x} \perp \mathbf{y}$  表示  $\mathbf{x}$  和  $\mathbf{y}$  相互独立,  $\mathbf{x} \perp \mathbf{y} | \mathbf{z}$  表示  $\mathbf{x}$  和  $\mathbf{y}$  在给定  $\mathbf{z}$  时条件独立。

## 3.8 期望、方差和协方差

函数  $f(x)$  关于某分布  $P(\mathbf{x})$  的期望 (expectation) 或者期望值 (expected value) 是指, 当  $x$  由  $P$  产生,  $f$  作用于  $x$  时,  $f(x)$  的平均值。对于离散型随机变量, 这可以通过求和得到:

$$\mathbb{E}_{\mathbf{x} \sim P}[f(x)] = \sum_x P(x)f(x), \quad (3.9)$$

对于连续型随机变量可以通过求积分得到:

$$\mathbb{E}_{\mathbf{x} \sim p}[f(x)] = \int p(x)f(x)dx. \quad (3.10)$$

当概率分布在上下文中指明时, 我们可以只写出期望作用的随机变量的名称来进行简化, 例如  $\mathbb{E}_{\mathbf{x}}[f(x)]$ 。如果期望作用的随机变量也很明确, 我们可以完全不写脚标, 就像  $\mathbb{E}[f(x)]$ 。默认地, 我们假设  $\mathbb{E}[\cdot]$  表示对方括号内的所有随机变量的值求平均。类似的, 当没有歧义时, 我们还可以省略方括号。

期望是线性的, 例如,

$$\mathbb{E}_{\mathbf{x}}[\alpha f(x) + \beta g(x)] = \alpha \mathbb{E}_{\mathbf{x}}[f(x)] + \beta \mathbb{E}_{\mathbf{x}}[g(x)], \quad (3.11)$$

其中  $\alpha$  和  $\beta$  不依赖于  $x$ 。

方差 (variance) 衡量的是当我们对  $x$  依据它的概率分布进行采样时, 随机变量  $x$  的函数值会呈现多大的差异:

$$\text{Var}(f(x)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2]. \quad (3.12)$$

当方差很小时,  $f(x)$  的值形成的簇比较接近它们的期望值。方差的平方根被称为标准差 (standard deviation)。

协方差 ( covariance ) 在某种意义上给出了两个变量线性相关性的强度以及这些变量的尺度：

$$\text{Cov}(f(x), g(y)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])]. \quad (3.13)$$

协方差的绝对值如果很大则意味着变量值变化很大并且它们同时距离各自的均值很远。如果协方差是正的，那么两个变量都倾向于同时取得相对较大的值。如果协方差是负的，那么其中一个变量倾向于取得相对较大的值的同时，另一个变量倾向于取得相对较小的值，反之亦然。其他的衡量指标如相关系数 ( correlation ) 将每个变量的贡献归一化，为了只衡量变量的相关性而不受各个变量尺度大小的影响。

协方差和相关性是有联系的，但实际上是有不同的概念。它们是有联系的，因为两个变量如果相互独立那么它们的协方差为零，如果两个变量的协方差不为零那么它们一定是相关的。然而，独立性又是和协方差完全不同的性质。两个变量如果协方差为零，它们之间一定没有线性关系。独立性比零协方差的要求更强，因为独立性还排除了非线性的关系。两个变量相互依赖但具有零协方差是可能的。例如，假设我们首先从区间  $[-1, 1]$  上的均匀分布中采样出一个实数  $x$ 。然后我们对一个随机变量  $s$  进行采样。 $s$  以  $\frac{1}{2}$  的概率值为 1，否则为 -1。我们可以通过令  $y = sx$  来生成一个随机变量  $y$ 。显然， $x$  和  $y$  不是相互独立的，因为  $x$  完全决定了  $y$  的尺度。然而， $\text{Cov}(x, y) = 0$ 。

随机向量  $\mathbf{x} \in \mathbb{R}^n$  的 协方差矩阵 ( covariance matrix ) 是一个  $n \times n$  的矩阵，并且满足

$$\text{Cov}(\mathbf{x})_{i,j} = \text{Cov}(\mathbf{x}_i, \mathbf{x}_j). \quad (3.14)$$

协方差矩阵的对角元是方差：

$$\text{Cov}(\mathbf{x}_i, \mathbf{x}_i) = \text{Var}(\mathbf{x}_i). \quad (3.15)$$

## 3.9 常用概率分布

许多简单的概率分布在机器学习的众多领域中都是有用的。

### 3.9.1 Bernoulli 分布

**Bernoulli 分布** (Bernoulli distribution) 是单个二值随机变量的分布。它由单个参数  $\phi \in [0, 1]$  控制,  $\phi$  给出了随机变量等于 1 的概率。它具有如下的一些性质:

$$P(x = 1) = \phi \quad (3.16)$$

$$P(x = 0) = 1 - \phi \quad (3.17)$$

$$P(x = x) = \phi^x(1 - \phi)^{1-x} \quad (3.18)$$

$$\mathbb{E}_x[x] = \phi \quad (3.19)$$

$$\text{Var}_x(x) = \phi(1 - \phi) \quad (3.20)$$

### 3.9.2 Multinoulli 分布

**Multinoulli 分布** (multinoulli distribution) 或者 **范畴分布** (categorical distribution) 是指在具有  $k$  个不同状态的单个离散型随机变量上的分布, 其中  $k$  是一个有限值。<sup>2</sup> Multinoulli 分布由向量  $\mathbf{p} \in [0, 1]^{k-1}$  参数化, 其中每一个分量  $p_i$  表示第  $i$  个状态的概率。最后的第  $k$  个状态的概率可以通过  $1 - \mathbf{1}^\top \mathbf{p}$  给出。注意我们必须限制  $\mathbf{1}^\top \mathbf{p} \leq 1$ 。Multinoulli 分布经常用来表示对象分类的分布, 所以我们很少假设状态 1 具有数值 1 之类的。因此, 我们通常不需要去计算 Multinoulli 分布的随机变量的期望和方差。

Bernoulli 分布和 Multinoulli 分布足够用来描述在它们领域内的任意分布。它们能够描述这些分布, 不是因为它们特别强大, 而是因为它们的领域很简单; 它们可以对那些, 能够将所有的状态进行枚举的离散型随机变量进行建模。当处理的是连续型随机变量时, 会有不可数无限多的状态, 所以任何通过少量参数描述的概率分布都必须在分布上加以严格的限制。

---

<sup>2</sup>“multinoulli”这个术语是最近被 Gustavo Lacerdo 发明、被 Murphy (2012) 推广的。Multinoulli 分布是多项式分布 (multinomial distribution) 的一个特例。多项式分布是  $\{0, \dots, n\}^k$  中的向量的分布, 用于表示当对 Multinoulli 分布采样  $n$  次时  $k$  个类中的每一个被访问的次数。很多文章使用“多项式分布”而实际上说的是 Multinoulli 分布, 但是他们并没有说是对  $n = 1$  的情况, 这点需要注意。

### 3.9.3 高斯分布

实数上最常用的分布就是正态分布 (normal distribution)，也称为高斯分布 (Gaussian distribution)：

$$\mathcal{N}(x; \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right). \quad (3.21)$$

图 3.1 画出了正态分布的概率密度函数。

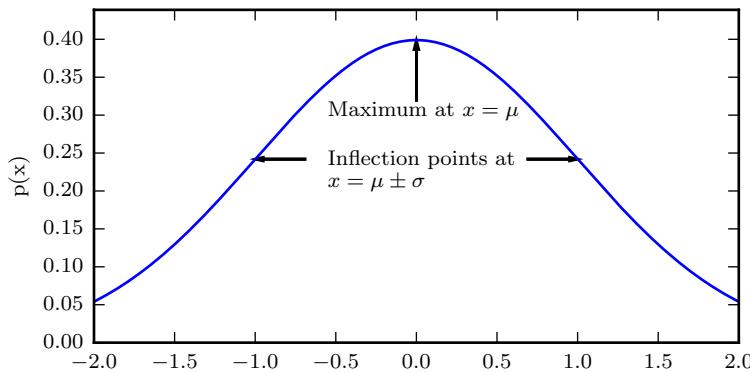


图 3.1：正态分布。正态分布  $\mathcal{N}(x; \mu, \sigma^2)$  呈现经典的“钟形曲线”的形状，其中中心峰的  $x$  坐标由  $\mu$  给出，峰的宽度受  $\sigma$  控制。在这个示例中，我们展示的是标准正态分布 (standard normal distribution)，其中  $\mu = 0, \sigma = 1$ 。

正态分布由两个参数控制， $\mu \in \mathbb{R}$  和  $\sigma \in (0, \infty)$ 。参数  $\mu$  给出了中心峰值的坐标，这也是分布的均值： $\mathbb{E}[x] = \mu$ 。分布的标准差用  $\sigma$  表示，方差用  $\sigma^2$  表示。

当我们对概率密度函数求值时，我们需要对  $\sigma$  平方并且取倒数。当我们需要经常对不同参数下的概率密度函数求值时，一种更高效的参数化分布的方式是使用参数  $\beta \in (0, \infty)$ ，来控制分布的精度 (precision) (或方差的倒数)：

$$\mathcal{N}(x; \mu, \beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} \exp\left(-\frac{1}{2}\beta(x - \mu)^2\right). \quad (3.22)$$

采用正态分布在很多应用中都是一个明智的选择。当我们由于缺乏关于某个实数上分布的先验知识而不知道该选择怎样的形式时，正态分布是默认的比较好的选择，其中有两个原因。

第一，我们想要建模的很多分布的真实情况是比较接近正态分布的。中心极限定理 (central limit theorem) 说明很多独立随机变量的和近似服从正态分布。这意味着在实际中，很多复杂系统都可以被成功地建模成正态分布的噪声，即使系统可以被分解成一些更结构化的部分。

第二，在具有相同方差的所有可能的概率分布中，正态分布在实数上具有最大的不确定性。因此，我们可以认为正态分布是对模型加入的先验知识量最少的分布。充分利用和证明这个想法需要更多的数学工具，我们推迟到第 19.4.2 节进行讲解。

正态分布可以推广到  $\mathbb{R}^n$  空间，这种情况下被称为 **多维正态分布** (multivariate normal distribution)。它的参数是一个正定对称矩阵  $\Sigma$ :

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Sigma) = \sqrt{\frac{1}{(2\pi)^n \det(\Sigma)}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})\right). \quad (3.23)$$

参数  $\boldsymbol{\mu}$  仍然表示分布的均值，只不过现在是向量值。参数  $\Sigma$  给出了分布的协方差矩阵。和单变量的情况类似，当我们希望对很多不同参数下的概率密度函数多次求值时，协方差矩阵并不是一个很高效的参数化分布的方式，因为对概率密度函数求值时需要对  $\Sigma$  求逆。我们可以使用一个 **精度矩阵** (precision matrix)  $\beta$  进行替代:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \beta^{-1}) = \sqrt{\frac{\det(\beta)}{(2\pi)^n}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \beta (\mathbf{x} - \boldsymbol{\mu})\right). \quad (3.24)$$

我们常常把协方差矩阵固定成一个对角阵。一个更简单的版本是**各向同性** (isotropic) 高斯分布，它的协方差矩阵是一个标量乘以单位阵。

### 3.9.4 指数分布和 Laplace 分布

在深度学习中，我们经常会需要一个在  $x = 0$  点处取得边界点 (sharp point) 的分布。为了实现这一目的，我们可以使用 **指数分布** (exponential distribution):

$$p(x; \lambda) = \lambda \mathbf{1}_{x \geq 0} \exp(-\lambda x). \quad (3.25)$$

指数分布使用指示函数(indicator function)  $\mathbf{1}_{x \geq 0}$  来使得当  $x$  取负值时的概率为零。

一个联系紧密的概率分布是 **Laplace 分布** (Laplace distribution)，它允许我们在任意一点  $\mu$  处设置概率质量的峰值

$$\text{Laplace}(x; \mu, \gamma) = \frac{1}{2\gamma} \exp\left(-\frac{|x - \mu|}{\gamma}\right). \quad (3.26)$$

### 3.9.5 Dirac 分布和经验分布

在一些情况下，我们希望概率分布中的所有质量都集中在一个点上。这可以通过 **Dirac delta 函数** (Dirac delta function)  $\delta(x)$  定义概率密度函数来实现：

$$p(x) = \delta(x - \mu). \quad (3.27)$$

Dirac delta 函数被定义成在除了 0 以外的所有点的值都为 0，但是积分为 1。Dirac delta 函数不像普通函数一样对  $x$  的每一个值都有一个实数值的输出，它是一种不同类型的数学对象，被称为 **广义函数** (generalized function)，广义函数是依据积分性质定义的数学对象。我们可以把 Dirac delta 函数想成一系列函数的极限点，这一系列函数把除 0 以外的所有点的概率密度越变越小。

通过把  $p(x)$  定义成  $\delta$  函数左移  $-\mu$  个单位，我们得到了一个在  $x = \mu$  处具有无限窄也无限高的峰值的概率质量。

Dirac 分布经常作为 **经验分布** (empirical distribution) 的一个组成部分出现：

$$\hat{p}(\mathbf{x}) = \frac{1}{m} \sum_{i=1}^m \delta(\mathbf{x} - \mathbf{x}^{(i)}) \quad (3.28)$$

经验分布将概率密度  $\frac{1}{m}$  赋给  $m$  个点  $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}$  中的每一个，这些点是给定的数据集或者采样的集合。只有在定义连续型随机变量的经验分布时，Dirac delta 函数才是必要的。对于离散型随机变量，情况更加简单：经验分布可以被定义成一个 Multinoulli 分布，对于每一个可能的输入，其概率可以简单地设为在训练集上那个输入值的 **经验频率** (empirical frequency)。

当我们在训练集上训练模型时，我们可以认为从这个训练集上得到的经验分布指明了我们采样来源的分布。关于经验分布另外一种重要的观点是，它是训练数据的似然最大的那个概率密度函数 (见第 5.5 节)。

### 3.9.6 分布的混合

通过组合一些简单的概率分布来定义新的概率分布也是很常见的。一种通用的组合方法是构造 **混合分布** (mixture distribution)。混合分布由一些组件 (component) 分布构成。每次实验，样本是由哪个组件分布产生的取决于从一个 Multinoulli 分布中采样的结果：

$$P(\mathbf{x}) = \sum_i P(\mathbf{c} = i) P(\mathbf{x} | \mathbf{c} = i), \quad (3.29)$$

这里  $P(c)$  是对各组件的一个 Multinoulli 分布。

我们已经看过一个混合分布的例子了：实值变量的经验分布对于每一个训练实例来说，就是以 Dirac 分布为组件的混合分布。

混合模型是组合简单概率分布来生成更丰富的分布的一种简单策略。在第十六章中，我们更加详细地探讨从简单概率分布构建复杂模型的技术。

混合模型使我们能够一瞥以后会用到的一个非常重要的概念——**潜变量** (latent variable)。潜变量是我们不能直接观测到的随机变量。混合模型的组件标识变量  $c$  就是其中一个例子。潜变量在联合分布中可能和  $x$  有关，在这种情况下， $P(x, c) = P(x | c)P(c)$ 。潜变量的分布  $P(c)$  以及关联潜变量和观测变量的条件分布  $P(x | c)$ ，共同决定了分布  $P(x)$  的形状，尽管描述  $P(x)$  时可能并不需要潜变量。潜变量将在第 16.5 节中深入讨论。

一个非常强大且常见的混合模型是 **高斯混合模型** (Gaussian Mixture Model)，它的组件  $p(\mathbf{x} | c = i)$  是高斯分布。每个组件都有各自的参数，均值  $\mu^{(i)}$  和协方差矩阵  $\Sigma^{(i)}$ 。有一些混合可以有更多的限制。例如，协方差矩阵可以通过  $\Sigma^{(i)} = \Sigma, \forall i$  的形式在组件之间共享参数。和单个高斯分布一样，高斯混合模型有时会限制每个组件的协方差矩阵为对角的或者各向同性的 (标量乘以单位矩阵)。

除了均值和协方差以外，高斯混合模型的参数指明了给每个组件  $i$  的 **先验概率** (prior probability)  $\alpha_i = P(c = i)$ 。“先验”一词表明了在观测到  $\mathbf{x}$  之前传递给模型关于  $c$  的信念。作为对比， $P(c | \mathbf{x})$  是 **后验概率** (posterior probability)，因为它是在观测到  $\mathbf{x}$  之后进行计算的。高斯混合模型是概率密度的 **万能近似器** (universal approximator)，在这种意义上，任何平滑的概率密度都可以用具有足够多组件的高斯混合模型以任意精度来逼近。

图 3.2 演示了某个高斯混合模型生成的样本。

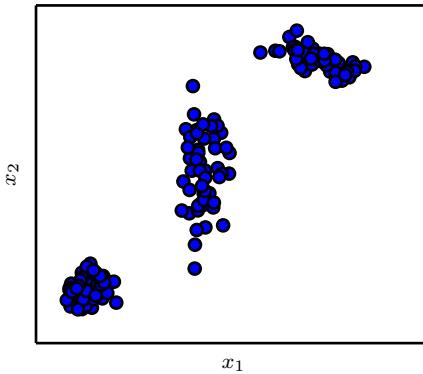


图 3.2: 来自高斯混合模型的样本。在这个示例中，有三个组件。从左到右，第一个组件具有各向同性的协方差矩阵，这意味着它在每个方向上具有相同的方差。第二个组件具有对角的协方差矩阵，这意味着它可以沿着每个轴的对齐方向单独控制方差。该示例中，沿着  $x_2$  轴的方差要比沿着  $x_1$  轴的方差大。第三个组件具有满秩的协方差矩阵，使它能够沿着任意基的方向单独地控制方差。

## 3.10 常用函数的有用性质

某些函数在处理概率分布时经常会出现，尤其是深度学习的模型中用到的概率分布。

其中一个函数是 **logistic sigmoid** 函数：

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \quad (3.30)$$

logistic sigmoid 函数通常用来产生 Bernoulli 分布中的参数  $\phi$ ，因为它的范围是  $(0, 1)$ ，处在  $\phi$  的有效取值范围内。图 3.3 给出了 sigmoid 函数的图示。sigmoid 函数在变量取绝对值非常大的正值或负值时会出现 **饱和 (saturate)** 现象，意味着函数会变得很平，并且对输入的微小改变会变得不敏感。

另外一个经常遇到的函数是 **softplus** 函数 ( softplus function ) (Dugas *et al.*, 2001)：

$$\zeta(x) = \log(1 + \exp(x)). \quad (3.31)$$

softplus 函数可以用来产生正态分布的  $\beta$  和  $\sigma$  参数，因为它的范围是  $(0, \infty)$ 。当处理包含 sigmoid 函数的表达式时它也经常出现。softplus 函数名来源于它是另外一个

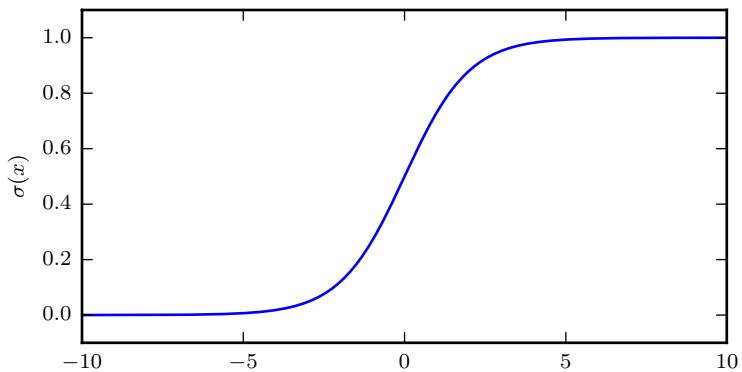


图 3.3: logistic sigmoid 函数。

函数的平滑（或“软化”）形式，这个函数是

$$x^+ = \max(0, x). \quad (3.32)$$

图 3.4 给出了 softplus 函数的图示。

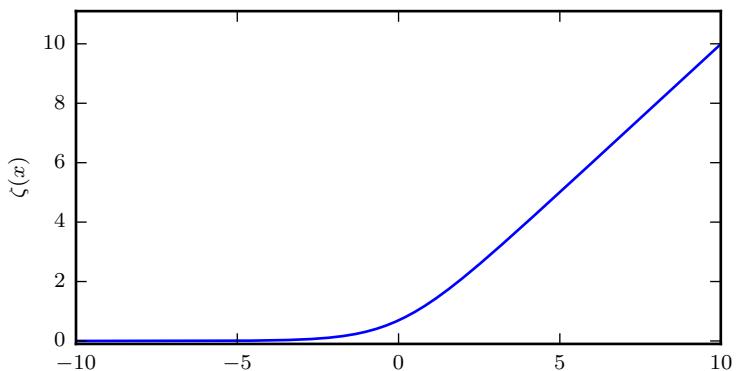


图 3.4: softplus 函数。

下面一些性质非常有用，你可能要记下来：

$$\sigma(x) = \frac{\exp(x)}{\exp(x) + \exp(0)} \quad (3.33)$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x)) \quad (3.34)$$

$$1 - \sigma(x) = \sigma(-x) \quad (3.35)$$

$$\log \sigma(x) = -\zeta(-x) \quad (3.36)$$

$$\frac{d}{dx}\zeta(x) = \sigma(x) \quad (3.37)$$

$$\forall x \in (0, 1), \sigma^{-1}(x) = \log\left(\frac{x}{1-x}\right) \quad (3.38)$$

$$\forall x > 0, \zeta^{-1}(x) = \log(\exp(x) - 1) \quad (3.39)$$

$$\zeta(x) = \int_{-\infty}^x \sigma(y) dy \quad (3.40)$$

$$\zeta(x) - \zeta(-x) = x \quad (3.41)$$

函数  $\sigma^{-1}(x)$  在统计学中被称为 **分对数** ( logit )，但这个函数在机器学习中很少用到。

式 (3.41) 为函数名“softplus”提供了其他的正当理由。softplus 函数被设计成**正部函数** ( positive part function ) 的平滑版本，这个正部函数是指  $x^+ = \max\{0, x\}$ 。与正部函数相对的是**负部函数** ( negative part function )  $x^- = \max\{0, -x\}$ 。为了获得类似负部函数的一个平滑函数，我们可以使用  $\zeta(-x)$ 。就像  $x$  可以用它的正部和负部通过等式  $x^+ - x^- = x$  恢复一样，我们也可以用同样的方式对  $\zeta(x)$  和  $\zeta(-x)$  进行操作，就像式 (3.41) 中那样。

## 3.11 贝叶斯规则

我们经常会需要在已知  $P(y | x)$  时计算  $P(x | y)$ 。幸运的是，如果还知道  $P(x)$ ，我们可以用**贝叶斯规则** ( Bayes' rule ) 来实现这一目的：

$$P(x | y) = \frac{P(x)P(y | x)}{P(y)}. \quad (3.42)$$

注意到  $P(y)$  出现在上面的公式中，它通常使用  $P(y) = \sum_x P(y | x)P(x)$  来计算，所以我们并不需要事先知道  $P(y)$  的信息。

贝叶斯规则可以从条件概率的定义直接推导得出，但我们最好记住这个公式的名字，因为很多文献通过名字来引用这个公式。这个公式是以牧师 Thomas Bayes 的名字来命名的，他是第一个发现这个公式特例的人。这里介绍的一般形式由 Pierre-Simon Laplace 独立发现。

## 3.12 连续型变量的技术细节

连续型随机变量和概率密度函数的深入理解需要用到数学分支 测度论 ( measure theory ) 的相关内容来扩展概率论。测度论超出了本书的范畴，但我们可以简要勾勒一些测度论用来解决的问题。

在第 3.3.2 节中，我们已经看到连续型向量值随机变量  $\mathbf{x}$  落在某个集合  $\mathbb{S}$  中的概率是通过  $p(\mathbf{x})$  对集合  $\mathbb{S}$  积分得到的。对于集合  $\mathbb{S}$  的一些选择可能会引起悖论。例如，构造两个集合  $\mathbb{S}_1$  和  $\mathbb{S}_2$  使得  $p(\mathbf{x} \in \mathbb{S}_1) + p(\mathbf{x} \in \mathbb{S}_2) > 1$  并且  $\mathbb{S}_1 \cap \mathbb{S}_2 = \emptyset$  是可能的。这些集合通常是大量使用了实数的无限精度来构造的，例如通过构造分形形状 (fractal-shaped) 的集合或者是通过有理数相关集合的变换定义的集合。<sup>3</sup> 测度论的一个重要贡献就是提供了一些集合的特征使得我们在计算概率时不会遇到悖论。在本书中，我们只对相对简单的集合进行积分，所以测度论的这个方面不会成为一个相关考虑。

对于我们的目的，测度论更多的是用来描述那些适用于  $\mathbb{R}^n$  上的大多数点，却不适用于一些边界情况的定理。测度论提供了一种严格的方式来描述那些非常微小的点集。这种集合被称为“零测度 ( measure zero )”的。我们不会在本书中给出这个概念的正式定义。然而，直观地理解这个概念是有用的，我们可以认为零测度集在我们的度量空间中不占有任何的体积。例如，在  $\mathbb{R}^2$  空间中，一条直线的测度为零，而填充的多边形具有正的测度。类似的，一个单独的点的测度为零。可数多个零测度集的并仍然是零测度的 ( 所以所有有理数构成的集合测度为零 )。

另外一个有用的测度论中的术语是“几乎处处 ( almost everywhere )”。某个性质如果是几乎处处都成立的，那么它在整个空间中除了一个测度为零的集合以外都是成立的。因为这些例外只在空间中占有极其微小的量，它们在多数应用中都可以被放心地忽略。概率论中的一些重要结果对于离散值成立但对于连续值只能是“几乎处处”成立。

<sup>3</sup>Banach-Tarski 定理给出了这类集合的一个有趣的例子。译者注：我们这里把“the set of rational numbers”翻译成“有理数相关集合”，理解为“一些有理数组成的集合”，如果直接用后面的翻译读起来会比较拗口。

连续型随机变量的另一技术细节，涉及到处理那种相互之间有确定性函数关系的连续型变量。假设我们有两个随机变量  $\mathbf{x}$  和  $\mathbf{y}$  满足  $\mathbf{y} = g(\mathbf{x})$ ，其中  $g$  是可逆的、连续可微的函数。可能有人会想  $p_y(\mathbf{y}) = p_x(g^{-1}(\mathbf{y}))$ 。但实际上这并不对。

举一个简单的例子，假设我们有两个标量值随机变量  $x$  和  $y$ ，并且满足  $y = \frac{x}{2}$  以及  $x \sim U(0, 1)$ 。如果我们使用  $p_y(y) = p_x(2y)$ ，那么  $p_y$  除了区间  $[0, \frac{1}{2}]$  以外都为 0，并且在这个区间上的值为 1。这意味着

$$\int p_y(y) dy = \frac{1}{2}, \quad (3.43)$$

而这违背了概率密度的定义（积分为 1）。这个常见错误之所以错是因为它没有考虑到引入函数  $g$  后造成空间变形。回忆一下， $\mathbf{x}$  落在无穷小的体积为  $\delta\mathbf{x}$  的区域内的概率为  $p(\mathbf{x})\delta\mathbf{x}$ 。因为  $g$  可能会扩展或者压缩空间，在  $\mathbf{x}$  空间内的包围着  $\mathbf{x}$  的无穷小体积在  $\mathbf{y}$  空间中可能有不同的体积。

为了看出如何改正这个问题，我们回到标量值的情况。我们需要保持下面这个性质：

$$|p_y(g(x))dy| = |p_x(x)dx|. \quad (3.44)$$

求解上式，我们得到

$$p_y(y) = p_x(g^{-1}(y)) \left| \frac{\partial x}{\partial y} \right| \quad (3.45)$$

或者等价地，

$$p_x(x) = p_y(g(x)) \left| \frac{\partial g(x)}{\partial x} \right|. \quad (3.46)$$

在高维空间中，微分运算扩展为 **Jacobian 矩阵** (Jacobian matrix) 的行列式——矩阵的每个元素为  $J_{i,j} = \frac{\partial x_i}{\partial y_j}$ 。因此，对于实值向量  $\mathbf{x}$  和  $\mathbf{y}$ ，

$$p_x(\mathbf{x}) = p_y(g(\mathbf{x})) \left| \det \left( \frac{\partial g(\mathbf{x})}{\partial \mathbf{x}} \right) \right|. \quad (3.47)$$

## 3.13 信息论

信息论是应用数学的一个分支，主要研究的是对一个信号包含信息的多少进行量化。它最初被发明是用来研究在一个含有噪声的信道上用离散的字母表来发送消息，例如通过无线电传输来通信。在这种情况下，信息论告诉我们如何对消息设计最优编码以及计算消息的期望长度，这些消息是使用多种不同编码机制、从特定

的概率分布上采样得到的。在机器学习中，我们也可以把信息论应用于连续型变量，此时某些消息长度的解释不再适用。信息论是电子工程和计算机科学中许多领域的基础。在本书中，我们主要使用信息论的一些关键思想来描述概率分布或者量化概率分布之间的相似性。有关信息论的更多细节，参见 Cover and Thomas (2006) 或者 MacKay (2003)。

信息论的基本想法是一个不太可能的事件居然发生了，要比一个非常可能的事件发生，能提供更多的信息。消息说：“今天早上太阳升起”信息量是如此之少以至于没有必要发送，但一条消息说：“今天早上有日食”信息量就很丰富。

我们想要通过这种基本想法来量化信息。特别地，

- 非常可能发生的事件信息量要比较少，并且极端情况下，确保能够发生的事件应该没有信息量。
- 较不可能发生的事件具有更高的信息量。
- 独立事件应具有增量的信息。例如，投掷的硬币两次正面朝上传递的信息量，应该是投掷一次硬币正面朝上的信息量的两倍。

为了满足上述三个性质，我们定义一个事件  $x = x$  的 **自信息** (self-information) 为

$$I(x) = -\log P(x). \quad (3.48)$$

在本书中，我们总是用  $\log$  来表示自然对数，其底数为  $e$ 。因此我们定义的  $I(x)$  单位是 **奈特** (nats)。一奈特是以  $\frac{1}{e}$  的概率观测到一个事件时获得的信息量。其他的材料中使用底数为 2 的对数，单位是 **比特** (bit) 或者 **香农** (shannons)；通过比特度量的信息只是通过奈特度量信息的常数倍。

当  $x$  是连续的，我们使用类似的关于信息的定义，但有些来源于离散形式的性质就丢失了。例如，一个具有单位密度的事件信息量仍然为 0，但是不能保证它一定发生。

自信息只处理单个的输出。我们可以用 **香农熵** (Shannon entropy) 来对整个概率分布中的不确定性总量进行量化：

$$H(x) = \mathbb{E}_{x \sim P}[I(x)] = -\mathbb{E}_{x \sim P}[\log P(x)], \quad (3.49)$$

也记作  $H(P)$ 。换言之，一个分布的香农熵是指遵循这个分布的事件所产生的期望信息总量。它给出了对依据概率分布  $P$  生成的符号进行编码所需的比特数在平均意义

上的下界 (当对数底数不是 2 时, 单位将有所不同)。那些接近确定性的分布 (输出几乎可以确定) 具有较低的熵; 那些接近均匀分布的概率分布具有较高的熵。图 3.5 给出了一个说明。当  $x$  是连续的, 香农熵被称为 **微分熵** (differential entropy)。

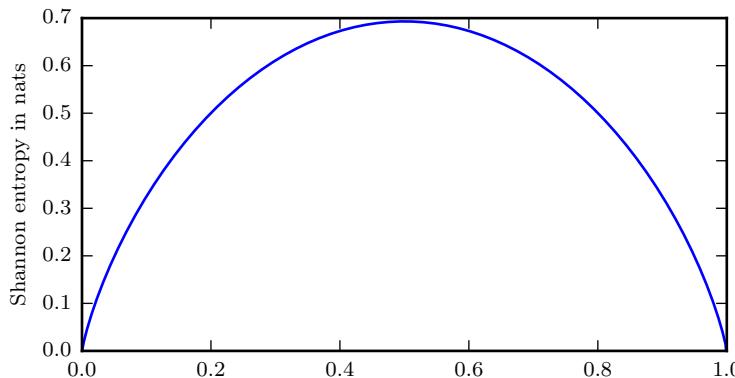


图 3.5: 二值随机变量的香农熵。该图说明了更接近确定性的分布是如何具有较低的香农熵, 而更接近均匀分布的分布是如何具有较高的香农熵。水平轴是  $p$ , 表示二值随机变量等于 1 的概率。熵由  $(p - 1) \log(1 - p) - p \log p$  给出。当  $p$  接近 0 时, 分布几乎是确定的, 因为随机变量几乎总是 0。当  $p$  接近 1 时, 分布也几乎是确定的, 因为随机变量几乎总是 1。当  $p = 0.5$  时, 熵是最大的, 因为分布在两个结果 (0 和 1) 上是均匀的。

如果我们对于同一个随机变量  $x$  有两个单独的概率分布  $P(x)$  和  $Q(x)$ , 我们可以使用 **KL 散度** (Kullback-Leibler (KL) divergence) 来衡量这两个分布的差异:

$$D_{\text{KL}}(P||Q) = \mathbb{E}_{x \sim P} \left[ \log \frac{P(x)}{Q(x)} \right] = \mathbb{E}_{x \sim P} [\log P(x) - \log Q(x)]. \quad (3.50)$$

在离散型变量的情况下, KL 散度衡量的是, 当我们使用一种被设计成能够使得概率分布  $Q$  产生的消息的长度最小的编码, 发送包含由概率分布  $P$  产生的符号的消息时, 所需要的额外信息量 (如果我们使用底数为 2 的对数时, 信息量用比特衡量, 但在机器学习中, 我们通常用奈特和自然对数。)

KL 散度有很多有用的性质, 最重要的是它是非负的。KL 散度为 0 当且仅当  $P$  和  $Q$  在离散型变量的情况下是相同的分布, 或者在连续型变量的情况下是“几乎处处”相同的。因为 KL 散度是非负的并且衡量的是两个分布之间的差异, 它经常被用作分布之间的某种距离。然而, 它并不是真的距离因为它不是对称的: 对于某些  $P$  和  $Q$ ,  $D_{\text{KL}}(P||Q) \neq D_{\text{KL}}(Q||P)$ 。这种非对称性意味着选择  $D_{\text{KL}}(P||Q)$  还是

$D_{\text{KL}}(Q||P)$  影响很大。更多细节可以看图 3.6。

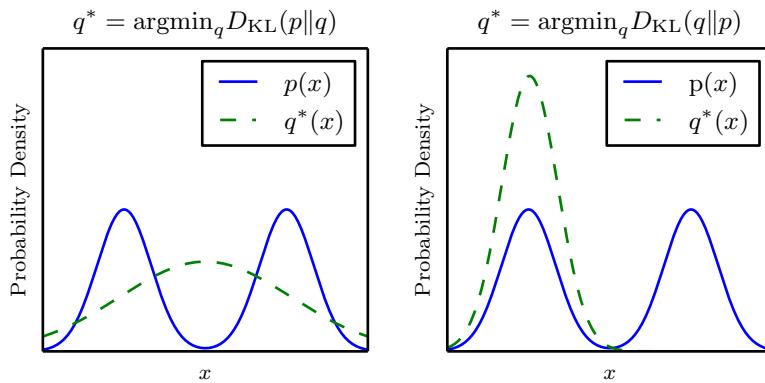


图 3.6: KL 散度是不对称的。假设我们有一个分布  $p(x)$ , 并且希望用另一个分布  $q(x)$  来近似它。我们可以选择最小化  $D_{\text{KL}}(p||q)$  或最小化  $D_{\text{KL}}(q||p)$ 。为了说明每种选择的效果, 我们令  $p$  是两个高斯分布的混合, 令  $q$  为单个高斯分布。选择使用 KL 散度的哪个方向是取决于问题的。一些应用需要这个近似分布  $q$  在真实分布  $p$  放置高概率的所有地方都放置高概率, 而其他应用需要这个近似分布  $q$  在真实分布  $p$  放置低概率的所有地方都很少放置高概率。KL 散度方向的选择反映了对于每种应用, 优先考虑哪一种选择。(左) 最小化  $D_{\text{KL}}(p||q)$  的效果。在这种情况下, 我们选择一个  $q$  使得它在  $p$  具有高概率的地方具有高概率。当  $p$  具有两个峰时,  $q$  选择将这些峰模糊到一起, 以便将高概率质量放到所有峰上。(右) 最小化  $D_{\text{KL}}(q||p)$  的效果。在这种情况下, 我们选择一个  $q$  使得它在  $p$  具有低概率的地方具有低概率。当  $p$  具有两个峰并且这些峰间隔很宽时, 如该图所示, 最小化 KL 散度会选择单个峰, 以避免将概率质量放置在  $p$  的多个峰之间的低概率区域中。这里, 我们说明当  $q$  被选择成强调左边峰时的结果。我们也可以通过选择右边峰来得到 KL 散度相同的值。如果这些峰没有被足够强的低概率区域分离, 那么 KL 散度的这个方向仍然可能选择模糊这些峰。

一个和 KL 散度密切联系的量是 交叉熵 (cross-entropy)  $H(P, Q) = H(P) + D_{\text{KL}}(P||Q)$ , 它和 KL 散度很像但是缺少左边一项:

$$H(P, Q) = -\mathbb{E}_{x \sim P} \log Q(x). \quad (3.51)$$

针对  $Q$  最小化交叉熵等价于最小化 KL 散度, 因为  $Q$  并不参与被省略的那一项。

当我们计算这些量时, 经常会遇到  $0 \log 0$  这个表达式。按照惯例, 在信息论中, 我们将这个表达式处理为  $\lim_{x \rightarrow 0} x \log x = 0$ 。

## 3.14 结构化概率模型

机器学习的算法经常会涉及到在非常多的随机变量上的概率分布。通常，这些概率分布涉及到的直接相互作用都是介于非常少的变量之间的。使用单个函数来描述整个联合概率分布是非常低效的（无论是计算上还是统计上）。

我们可以把概率分布分解成许多因子的乘积形式，而不是使用单一的函数来表示概率分布。例如，假设我们有三个随机变量  $a, b$  和  $c$ ，并且  $a$  影响  $b$  的取值， $b$  影响  $c$  的取值，但是  $a$  和  $c$  在给定  $b$  时是条件独立的。我们可以把全部三个变量的概率分布重新表示为两个变量的概率分布的连乘形式：

$$p(a, b, c) = p(a)p(b | a)p(c | b). \quad (3.52)$$

这种分解可以极大地减少用来描述一个分布的参数数量。每个因子使用的参数数目是它的变量数目的指数倍。这意味着，如果我们能够找到一种使每个因子分布具有更少变量的分解方法，我们就能极大地降低表示联合分布的成本。

我们可以用图来描述这种分解。这里我们使用的是图论中的“图”的概念：由一些可以通过边互相连接的顶点的集合构成。当我们用图来表示这种概率分布的分解，我们把它称为 **结构化概率模型** (structured probabilistic model) 或者 **图模型** (graphical model)。

有两种主要的结构化概率模型：有向的和无向的。两种图模型都使用图  $\mathcal{G}$ ，其中图的每个节点对应着一个随机变量，连接两个随机变量的边意味着概率分布可以表示成这两个随机变量之间的直接作用。

**有向** (directed) 模型使用带有有向边的图，它们用条件概率分布来表示分解，就像上面的例子。特别地，有向模型对于分布中的每一个随机变量  $x_i$  都包含着一个影响因子，这个组成  $x_i$  条件概率的影响因子被称为  $x_i$  的父节点，记为  $Pa_{\mathcal{G}}(x_i)$ ：

$$p(\mathbf{x}) = \prod_i p(x_i | Pa_{\mathcal{G}}(x_i)). \quad (3.53)$$

图 3.7 给出了一个有向图的例子以及它表示的概率分布的分解。

**无向** (undirected) 模型使用带有无向边的图，它们将分解表示成一组函数；不像有向模型那样，这些函数通常不是任何类型的概率分布。 $\mathcal{G}$  中任何满足两两之间有边连接的顶点的集合被称为团。无向模型中的每个团  $C^{(i)}$  都伴随着一个因子  $\phi^{(i)}(C^{(i)})$ 。这些因子仅仅是函数，并不是概率分布。每个因子的输出都必须是非负

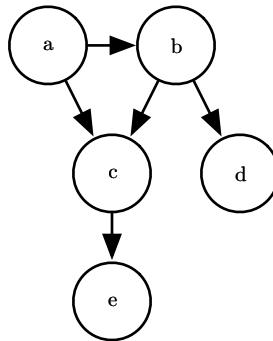


图 3.7: 关于随机变量  $a, b, c, d$  和  $e$  的有向图模型。这幅图对应的概率分布可以分解为

$$p(a, b, c, d, e) = p(a)p(b | a)p(c | a, b)p(d | b)p(e | c). \quad (3.54)$$

该图模型使我们能够快速看出此分布的一些性质。例如， $a$  和  $c$  直接相互影响，但  $a$  和  $e$  只有通过  $c$  间接相互影响。

的，但是并没有像概率分布中那样要求因子的和或者积分为 1。

随机变量的联合概率与所有这些因子的乘积 成比例 (proportional) ——意味着因子的值越大则可能性越大。当然，不能保证这种乘积的求和为 1。所以我们需要除以一个归一化常数  $Z$  来得到归一化的概率分布，归一化常数  $Z$  被定义为  $\phi$  函数乘积的所有状态的求和或积分。概率分布为：

$$p(\mathbf{x}) = \frac{1}{Z} \prod_i \phi^{(i)} (\mathcal{C}^{(i)}). \quad (3.55)$$

图 3.8 给出了一个无向图的例子以及它表示的概率分布的分解。

请记住，这些图模型表示的分解仅仅是描述概率分布的一种语言。它们不是互相排斥的概率分布族。有向或者无向不是概率分布的特性；它是概率分布的一种特殊 描述 (description) 所具有的特性，而任何概率分布都可以用这两种方式进行描述。

在本书第一部分和第二部分中，我们仅仅将结构化概率模型视作一门语言，来描述不同的机器学习算法选择表示的直接的概率关系。在讨论研究课题之前，读者不需要更深入地理解结构化概率模型。在第三部分的研究课题中，我们将更为详尽地探讨结构化概率模型。

本章复习了概率论中与深度学习最为相关的一些基本概念。我们还剩下一些基

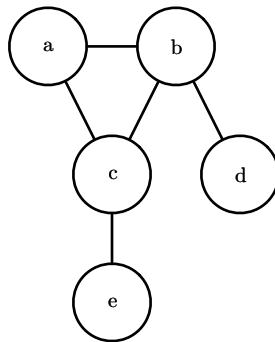


图 3.8: 关于随机变量  $a, b, c, d$  和  $e$  的无向图模型。这幅图对应的概率分布可以分解为

$$p(a, b, c, d, e) = \frac{1}{Z} \phi^{(1)}(a, b, c) \phi^{(2)}(b, d) \phi^{(3)}(c, e). \quad (3.56)$$

该图模型使我们能够快速看出此分布的一些性质。例如， $a$  和  $c$  直接相互影响，但  $a$  和  $e$  只有通过  $c$  间接相互影响。

本的数学工具需要讨论：数值方法。

# 第四章 数值计算

机器学习算法通常需要大量的数值计算。这通常是指通过迭代过程更新解的估计值来解决数学问题的算法，而不是通过解析过程推导出公式来提供正确解的方法。常见的操作包括优化（找到最小化或最大化函数值的参数）和线性方程组的求解。对数字计算机来说实数无法在有限内存下精确表示，因此仅仅是计算涉及实数的函数也是困难的。

## 4.1 上溢和下溢

连续数学在数字计算机上的根本困难是，我们需要通过有限数量的位模式来表示无限多的实数。这意味着我们在计算机中表示实数时，几乎总会引入一些近似误差。在许多情况下，这仅仅是舍入误差。舍入误差会导致一些问题，特别是当许多操作复合时，即使是理论上可行的算法，如果在设计时没有考虑最小化舍入误差的累积，在实践时也可能会导致算法失效。

一种极具毁灭性的舍入误差是 **下溢** (underflow)。当接近零的数被四舍五入为零时发生下溢。许多函数在其参数为零而不是一个很小的正数时才会表现出质的不同。例如，我们通常要避免被零除（一些软件环境将在这种情况下抛出异常，有些会返回一个非数字 (not-a-number, NaN) 的占位符）或避免取零的对数（这通常被视为  $-\infty$ ，进一步的算术运算会使其变成非数字）。

另一个极具破坏力的数值错误形式是 **上溢** (overflow)。当大量级的数被近似为  $\infty$  或  $-\infty$  时发生上溢。进一步的运算通常会导致这些无限值变为非数字。

必须对上溢和下溢进行数值稳定的一个例子是 **softmax 函数** (softmax func-

tion)。softmax 函数经常用于预测与 Multinoulli 分布相关联的概率，定义为

$$\text{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)}. \quad (4.1)$$

考虑一下当所有  $x_i$  都等于某个常数  $c$  时会发生什么。从理论分析上说，我们可以发现所有的输出都应该为  $\frac{1}{n}$ 。从数值计算上说，当  $c$  量级很大时，这可能不会发生。如果  $c$  是很小的负数， $\exp(c)$  就会下溢。这意味着 softmax 函数的分母会变成 0，所以最后的结果是未定义的。当  $c$  是非常大的正数时， $\exp(c)$  的上溢再次导致整个表达式未定义。这两个困难能通过计算  $\text{softmax}(\mathbf{z})$  同时解决，其中  $\mathbf{z} = \mathbf{x} - \max_i x_i$ 。简单的代数计算表明，softmax 解析上的函数值不会因为从输入向量减去或加上标量而改变。减去  $\max_i x_i$  导致  $\exp$  的最大参数为 0，这排除了上溢的可能性。同样地，分母中至少有一个值为 1 的项，这就排除了因分母下溢而导致被零除的可能性。

还有一个小问题。分子中的下溢仍可以导致整体表达式被计算为零。这意味着，如果我们在计算  $\log \text{softmax}(\mathbf{x})$  时，先计算 softmax 再把结果传给 log 函数，会错误地得到  $-\infty$ 。相反，我们必须实现一个单独的函数，并以数值稳定的方式计算  $\log \text{softmax}$ 。我们可以使用相同的技巧来稳定  $\log \text{softmax}$  函数。

在大多数情况下，我们没有明确地对本书描述的各种算法所涉及的数值考虑进行详细说明。底层库的开发者在实现深度学习算法时应该牢记数值问题。本书的大多数读者可以简单地依赖保证数值稳定的底层库。在某些情况下，我们有可能在实现一个新的算法时自动保持数值稳定。Theano (Bergstra *et al.*, 2010a; Bastien *et al.*, 2012a) 就是这样软件包的一个例子，它能自动检测并稳定深度学习中许多常见的数值不稳定的表达式。

## 4.2 病态条件

条件数表征函数相对于输入的微小变化而变化的快慢程度。输入被轻微扰动而迅速改变的函数对于科学计算来说可能是有问题的，因为输入中的舍入误差可能导致输出的巨大变化。

考虑函数  $f(\mathbf{x}) = \mathbf{A}^{-1}\mathbf{x}$ 。当  $\mathbf{A} \in \mathbb{R}^{n \times n}$  具有特征值分解时，其条件数为

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|. \quad (4.2)$$

这是最大和最小特征值的模之比<sup>1</sup>。当该数很大时，矩阵求逆对输入的误差特别敏感。

这种敏感性是矩阵本身的固有特性，而不是矩阵求逆期间舍入误差的结果。即使我们乘以完全正确的矩阵逆，病态条件的矩阵也会放大预先存在的误差。在实践中，该错误将与求逆过程本身的数值误差进一步复合。

### 4.3 基于梯度的优化方法

大多数深度学习算法都涉及某种形式的优化。优化指的是改变  $\mathbf{x}$  以最小化或最大化某个函数  $f(\mathbf{x})$  的任务。我们通常以最小化  $f(\mathbf{x})$  指代大多数最优化问题。最大化可经由最小化算法最小化  $-f(\mathbf{x})$  来实现。

我们把要最小化或最大化的函数称为 **目标函数** (objective function) 或 **准则** (criterion)。当我们对其进行最小化时，我们也把它称为 **代价函数** (cost function)、**损失函数** (loss function) 或 **误差函数** (error function)。虽然有些机器学习著作赋予这些名称特殊的意义，但在这本书中我们交替使用这些术语。

我们通常使用一个上标 \* 表示最小化或最大化函数的  $\mathbf{x}$  值。如我们记  $\mathbf{x}^* = \arg \min f(\mathbf{x})$ 。

我们假设读者已经熟悉微积分，这里简要回顾微积分概念如何与优化联系。

假设我们有一个函数  $y = f(x)$ ，其中  $x$  和  $y$  是实数。这个函数的 **导数** (derivative) 记为  $f'(x)$  或  $\frac{dy}{dx}$ 。导数  $f'(x)$  代表  $f(x)$  在点  $x$  处的斜率。换句话说，它表明如何缩放输入的小变化才能在输出获得相应的变化： $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$ 。

因此导数对于最小化一个函数很有用，因为它告诉我们如何更改  $x$  来略微地改善  $y$ 。例如，我们知道对于足够小的  $\epsilon$  来说， $f(x - \epsilon \text{sign}(f'(x)))$  是比  $f(x)$  小的。因此我们可以将  $x$  往导数的反方向移动一小步来减小  $f(x)$ 。这种技术被称为 **梯度下降** (gradient descent) (Cauchy, 1847)。图 4.1 展示了一个例子。

当  $f'(x) = 0$ ，导数无法提供往哪个方向移动的信息。 $f'(x) = 0$  的点称为 **临界点** (critical point) 或 **驻点** (stationary point)。一个 **局部极小点** (local minimum) 意味着这个点的  $f(x)$  小于所有邻近点，因此不可能通过移动无穷小的步长来减小  $f(x)$ 。一个 **局部极大点** (local maximum) 意味着这个点的  $f(x)$  大于所有邻近点，因此不可能通过移动无穷小的步长来增大  $f(x)$ 。有些临界点既不是最小点也不是最大

---

<sup>1</sup>译者注：与通常的条件数定义有所不同。

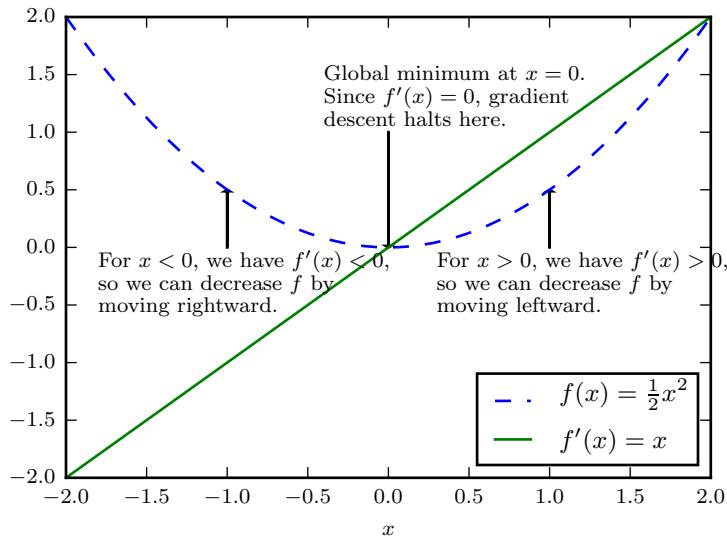


图 4.1: 梯度下降。梯度下降算法如何使用函数导数的示意图, 即沿着函数的下坡方向 ( 导数反方向 ) 直到最小。

点。这些点被称为 **鞍点** ( saddle point )。见图 4.2 给出的各种临界点的例子。

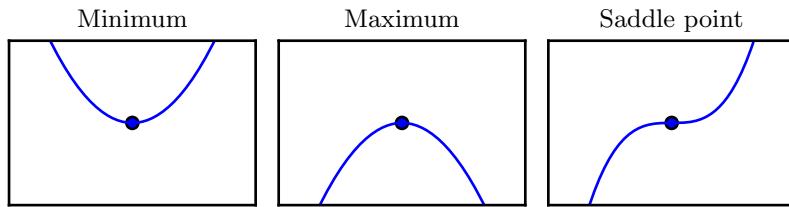


图 4.2: 临界点的类型。一维情况下, 三种临界点的示例。临界点是斜率为零的点。这样的点可以是 **局部极小点** ( local minimum ), 其值低于相邻点; **局部极大点** ( local maximum ), 其值高于相邻点; 或 **鞍点**, 同时存在更高和更低的相邻点。

使  $f(x)$  取得绝对的最小值 ( 相对所有其他值 ) 的点是 **全局最小点** ( global

minimum)。函数可能只有一个全局最小点或存在多个全局最小点，还可能存在不是全局最优的局部极小点。在深度学习的背景下，我们要优化的函数可能含有许多不是最优的局部极小点，或者还有很多处于非常平坦的区域内的鞍点。尤其是当输入是多维的时候，所有这些都将使优化变得困难。因此，我们通常寻找使  $f$  非常小的点，但这任何形式意义下并不一定是最小。见图 4.3 的例子。

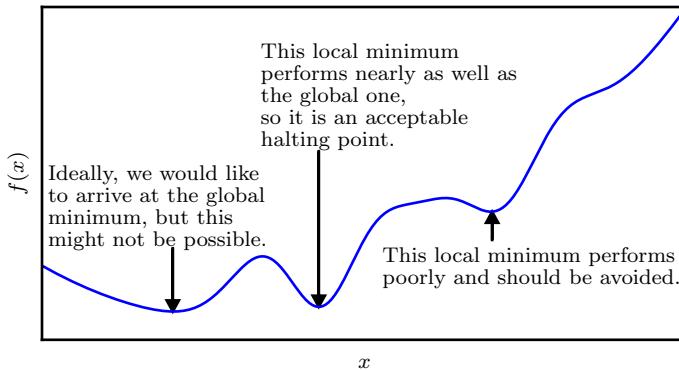


图 4.3: 近似最小化。当存在多个局部极小点或平坦区域时，优化算法可能无法找到全局最小点。在深度学习的背景下，即使找到的解不是真正最小的，但只要它们对应于代价函数显著低的值，我们通常就能接受这样的解。

我们经常最小化具有多维输入的函数： $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 。为了使“最小化”的概念有意义，输出必须是一维的（标量）。

针对具有多维输入的函数，我们需要用到 **偏导数** (partial derivative) 的概念。偏导数  $\frac{\partial}{\partial x_i} f(\mathbf{x})$  衡量点  $\mathbf{x}$  处只有  $x_i$  增加时  $f(\mathbf{x})$  如何变化。**梯度** (gradient) 是相对一个向量求导的导数： $f$  的导数是包含所有偏导数的向量，记为  $\nabla_{\mathbf{x}} f(\mathbf{x})$ 。梯度的第  $i$  个元素是  $f$  关于  $x_i$  的偏导数。在多维情况下，临界点是梯度中所有元素都为零的点。

在  $\mathbf{u}$  (单位向量) 方向的 **方向导数** (directional derivative) 是函数  $f$  在  $\mathbf{u}$  方向的斜率。换句话说，方向导数是函数  $f(\mathbf{x} + \alpha \mathbf{u})$  关于  $\alpha$  的导数 (在  $\alpha = 0$  时取得)。使用链式法则，我们可以看到当  $\alpha = 0$  时， $\frac{\partial}{\partial \alpha} f(\mathbf{x} + \alpha \mathbf{u}) = \mathbf{u}^\top \nabla_{\mathbf{x}} f(\mathbf{x})$ 。

为了最小化  $f$ , 我们希望找到使  $f$  下降得最快的方向。计算方向导数:

$$\min_{\mathbf{u}, \mathbf{u}^\top \mathbf{u}=1} \mathbf{u}^\top \nabla_{\mathbf{x}} f(\mathbf{x}) \quad (4.3)$$

$$= \min_{\mathbf{u}, \mathbf{u}^\top \mathbf{u}=1} \|\mathbf{u}\|_2 \|\nabla_{\mathbf{x}} f(\mathbf{x})\|_2 \cos \theta \quad (4.4)$$

其中  $\theta$  是  $\mathbf{u}$  与梯度的夹角。将  $\|\mathbf{u}\|_2 = 1$  代入, 并忽略与  $\mathbf{u}$  无关的项, 就能简化得到  $\min \cos \theta$ 。这在  $\mathbf{u}$  与梯度方向相同时取得最小。换句话说, 梯度向量指向上坡, 负梯度向量指向下坡。我们在负梯度方向上移动可以减小  $f$ 。这被称为最速下降法 (method of steepest descent) 或 梯度下降 (gradient descent)。

最速下降建议新的点为

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}) \quad (4.5)$$

其中  $\epsilon$  为 学习率 (learning rate), 是一个确定步长大小的正标量。我们可以通过几种不同的方式选择  $\epsilon$ 。普遍的方式是选择一个小常数。有时我们通过计算, 选择使方向导数消失的步长。还有一种方法是根据几个  $\epsilon$  计算  $f(\mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}))$ , 并选择其中能产生最小目标函数值的  $\epsilon$ 。这种策略被称为线搜索。

最速下降在梯度的每一个元素为零时收敛 (或在实践中, 很接近零时)。在某些情况下, 我们也许能够避免运行该迭代算法, 并通过解方程  $\nabla_{\mathbf{x}} f(\mathbf{x}) = 0$  直接跳到临界点。

虽然梯度下降被限制在连续空间中的优化问题, 但不断向更好的情况移动一小步 (即近似最佳的小移动) 的一般概念可以推广到离散空间。递增带有离散参数的目标函数被称为 爬山 (hill climbing) 算法 (Russel and Norvig, 2003)。

### 4.3.1 梯度之上: Jacobian 和 Hessian 矩阵

有时我们需要计算输入和输出都为向量的函数的所有偏导数。包含所有这样的偏导数的矩阵被称为 **Jacobian** 矩阵。具体来说, 如果我们有一个函数:  $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$ ,  $f$  的 Jacobian 矩阵  $\mathbf{J} \in \mathbb{R}^{n \times m}$  定义为  $J_{i,j} = \frac{\partial}{\partial x_j} f(\mathbf{x})_i$ 。

有时, 我们也对导数的导数感兴趣, 即 二阶导数 (second derivative)。例如, 有一个函数  $f: \mathbb{R}^m \rightarrow \mathbb{R}$ ,  $f$  的一阶导数(关于  $x_j$ ) 关于  $x_i$  的导数记为  $\frac{\partial^2}{\partial x_i \partial x_j} f$ 。在一维情况下, 我们可以将  $\frac{\partial^2}{\partial x^2} f$  为  $f''(x)$ 。二阶导数告诉我们, 一阶导数将如何随着输入的变化而改变。它表示只基于梯度信息的梯度下降步骤是否会产生如我们预期的那

样大的改善，因此它是重要的。我们可以认为，二阶导数是对曲率的衡量。假设我们有一个二次函数（虽然很多实践中的函数都不是二次的，但至少在局部可以很好地用二次近似）。如果这样的函数具有零二阶导数，那就没有曲率。也就是一条完全平坦的线，仅用梯度就可以预测它的值。我们使用沿负梯度方向大小为  $\epsilon$  的下降步，当该梯度是 1 时，代价函数将下降  $\epsilon$ 。如果二阶导数是负的，函数曲线向下凹陷（向上凸出），因此代价函数将下降的比  $\epsilon$  多。如果二阶导数是正的，函数曲线是向上凹陷（向下凸出），因此代价函数将下降的比  $\epsilon$  少。从图 4.4 可以看出不同形式的曲率如何影响基于梯度的预测值与真实的代价函数值的关系。

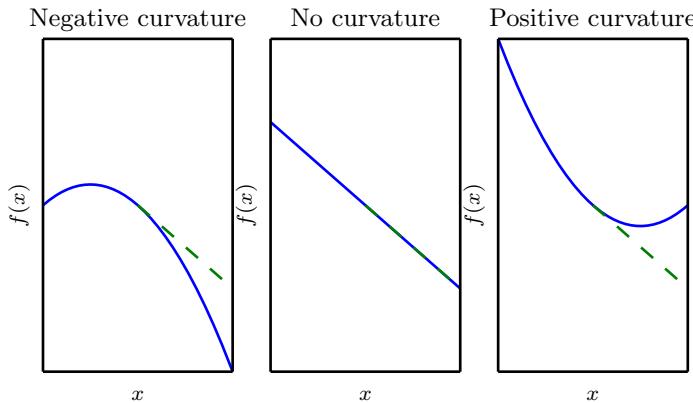


图 4.4: 二阶导数确定函数的曲率。这里我们展示具有各种曲率的二次函数。虚线表示我们仅根据梯度信息进行梯度下降后预期的代价函数值。对于负曲率，代价函数实际上比梯度预测下降得更快。没有曲率时，梯度正确预测下降值。对于正曲率，函数比预期下降得更慢，并且最终会开始增加，因此太大的步骤实际上可能会无意地增加函数值。

当我们的函数具有多维输入时，二阶导数也有很多。我们可以将这些导数合并成一个矩阵，称为 **Hessian** 矩阵。Hessian 矩阵  $\mathbf{H}(f)(\mathbf{x})$  定义为

$$\mathbf{H}(f)(\mathbf{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x}). \quad (4.6)$$

Hessian 等价于梯度的 Jacobian 矩阵。

微分算子在任何二阶偏导连续的点处可交换，也就是它们的顺序可以互换：

$$\frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x}) = \frac{\partial^2}{\partial x_j \partial x_i} f(\mathbf{x}). \quad (4.7)$$

这意味着  $H_{i,j} = H_{j,i}$ , 因此 Hessian 矩阵在这些点上是对称的。在深度学习背景下, 我们遇到的大多数函数的 Hessian 几乎处处都是对称的。因为 Hessian 矩阵是实对称的, 我们可以将其分解成一组实特征值和一组特征向量的正交基。在特定方向  $\mathbf{d}$  上的二阶导数可以写成  $\mathbf{d}^\top \mathbf{H} \mathbf{d}$ 。当  $\mathbf{d}$  是  $\mathbf{H}$  的一个特征向量时, 这个方向的二阶导数就是对应的特征值。对于其他的方向  $\mathbf{d}$ , 方向二阶导数是所有特征值的加权平均, 权重在 0 和 1 之间, 且与  $\mathbf{d}$  夹角越小的特征向量的权重越大。最大特征值确定最大二阶导数, 最小特征值确定最小二阶导数。

我们可以通过(方向)二阶导数预期一个梯度下降步骤能表现得多好。我们在当前点  $\mathbf{x}^{(0)}$  处作函数  $f(\mathbf{x})$  的近似二阶泰勒级数:

$$f(\mathbf{x}) \approx f(\mathbf{x}^{(0)}) + (\mathbf{x} - \mathbf{x}^{(0)})^\top \mathbf{g} + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{(0)})^\top \mathbf{H}(\mathbf{x} - \mathbf{x}^{(0)}), \quad (4.8)$$

其中  $\mathbf{g}$  是梯度,  $\mathbf{H}$  是  $\mathbf{x}^{(0)}$  点的 Hessian。如果我们使用学习率  $\epsilon$ , 那么新的点  $\mathbf{x}$  将会是  $\mathbf{x}^{(0)} - \epsilon \mathbf{g}$ 。代入上述的近似, 可得

$$f(\mathbf{x}^{(0)} - \epsilon \mathbf{g}) \approx f(\mathbf{x}^{(0)}) - \epsilon \mathbf{g}^\top \mathbf{g} + \frac{1}{2} \epsilon^2 \mathbf{g}^\top \mathbf{H} \mathbf{g}. \quad (4.9)$$

其中有 3 项: 函数的原始值、函数斜率导致的预期改善、函数曲率导致的校正。当最后一项太大时, 梯度下降实际上可能向上移动。当  $\mathbf{g}^\top \mathbf{H} \mathbf{g}$  为零或负时, 近似的泰勒级数表明增加  $\epsilon$  将永远使  $f$  下降。在实践中, 泰勒级数不会在  $\epsilon$  大的时候也保持准确, 因此在这种情况下我们必须采取更启发式的选择。当  $\mathbf{g}^\top \mathbf{H} \mathbf{g}$  为正时, 通过计算可得, 使近似泰勒级数下降最多的最优步长为

$$\epsilon^* = \frac{\mathbf{g}^\top \mathbf{g}}{\mathbf{g}^\top \mathbf{H} \mathbf{g}}. \quad (4.10)$$

最坏的情况下,  $\mathbf{g}$  与  $\mathbf{H}$  最大特征值  $\lambda_{\max}$  对应的特征向量对齐, 则最优步长是  $\frac{1}{\lambda_{\max}}$ 。我们要最小化的函数能用二次函数很好地近似的情况下, Hessian 的特征值决定了学习率的量级。

二阶导数还可以被用于确定一个临界点是否是局部极大点、局部极小点或鞍点。回想一下, 在临界点处  $f'(x) = 0$ 。而  $f''(x) > 0$  意味着  $f'(x)$  会随着我们移向右边而增加, 移向左边而减小, 也就是  $f'(x - \epsilon) < 0$  和  $f'(x + \epsilon) > 0$  对足够小的  $\epsilon$  成立。换句话说, 当我们移向右边, 斜率开始指向右边的上坡, 当我们移向左边, 斜率开始指向左边的上坡。因此我们得出结论, 当  $f'(x) = 0$  且  $f''(x) > 0$  时,  $\mathbf{x}$  是一个局部极小点。同样, 当  $f'(x) = 0$  且  $f''(x) < 0$  时,  $\mathbf{x}$  是一个局部极大点。这就是所谓

的二阶导数测试 (second derivative test)。不幸的是，当  $f''(x) = 0$  时测试是不确定的。在这种情况下， $\mathbf{x}$  可以是一个鞍点或平坦区域的一部分。

在多维情况下，我们需要检测函数的所有二阶导数。利用 Hessian 的特征值分解，我们可以将二阶导数测试扩展到多维情况。在临界点处 ( $\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{0}$ )，我们通过检测 Hessian 的特征值来判断该临界点是一个局部极大点、局部极小点还是鞍点。当 Hessian 是正定的（所有特征值都是正的），则该临界点是局部极小点。因为方向二阶导数在任意方向都是正的，参考单变量的二阶导数测试就能得出此结论。同样的，当 Hessian 是负定的（所有特征值都是负的），这个点就是局部极大点。在多维情况下，实际上我们可以找到确定该点是否为鞍点的积极迹象（某些情况下）。如果 Hessian 的特征值中至少一个是正的且至少一个是负的，那么  $\mathbf{x}$  是  $f$  某个横截面的局部极大点，却是另一个横截面的局部极小点。见图 4.5 中的例子。最后，多维二阶导数测试可能像单变量版本那样是不确定的。当所有非零特征值是同号的且至少有一个特征值是 0 时，这个检测就是不确定的。这是因为单变量的二阶导数测试在零特征值对应的横截面上是不确定的。

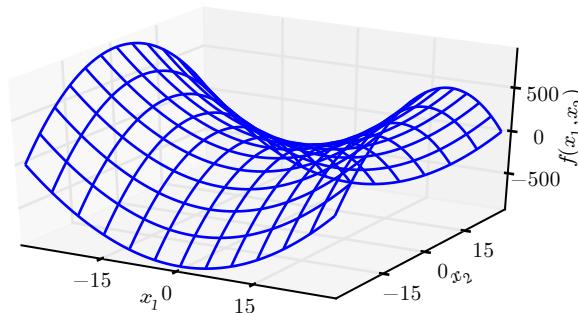


图 4.5: 既有正曲率又有负曲率的鞍点。示例中的函数是  $f(\mathbf{x}) = x_1^2 - x_2^2$ 。函数沿  $x_1$  轴向上弯曲。 $x_1$  轴是 Hessian 的一个特征向量，并且具有正特征值。函数沿  $x_2$  轴向下弯曲。该方向对应于 Hessian 负特征值的特征向量。名称“鞍点”源自该处函数的鞍状形状。这是具有鞍点函数的典型示例。维度多于一个时，鞍点不一定要具有 0 特征值：仅需要同时具有正特征值和负特征值。我们可以想象这样一个鞍点（具有正负特征值）在一个横截面内是局部极大点，而在另一个横截面内是局部极小点。

多维情况下，单个点处每个方向上的二阶导数是不同。Hessian 的条件数衡量这些二阶导数的变化范围。当 Hessian 的条件数很差时，梯度下降法也会表现得很

差。这是因为一个方向上的导数增加得很快，而在另一个方向上增加得很慢。梯度下降不知道导数的这种变化，所以它不知道应该优先探索导数长期为负的方向。病态条件也导致很难选择合适的步长。步长必须足够小，以免冲过最小而向具有较强正曲率的方向上升。这通常意味着步长太小，以致于在其他较小曲率的方向上进展不明显。见图 4.6 的例子。

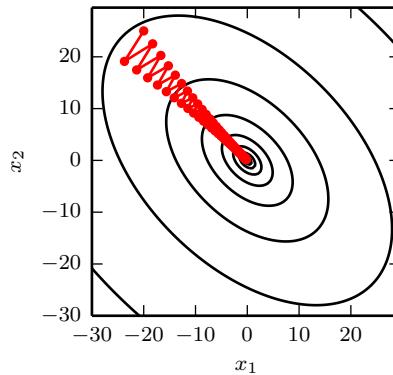


图 4.6: 梯度下降无法利用包含在 Hessian 矩阵中的曲率信息。这里我们使用梯度下降来最小化 Hessian 矩阵条件数为 5 的二次函数  $f(\mathbf{x})$ 。这意味着最大曲率方向具有比最小曲率方向多五倍的曲率。在这种情况下，最大曲率在  $[1, 1]^\top$  方向上，最小曲率在  $[1, -1]^\top$  方向上。红线表示梯度下降的路径。这个非常细长的二次函数类似一个长峡谷。梯度下降把时间浪费于在峡谷壁反复下降，因为它们是最陡峭的特征。由于步长有点大，有超过函数底部的趋势，因此需要在下一次迭代时在对面的峡谷壁下降。与指向该方向的特征向量对应的 Hessian 的大的正特征值表示该方向上的导数快速增加，因此基于 Hessian 的优化算法可以预测，在此情况下最陡峭方向实际上不是有前途的搜索方向。

我们可以使用 Hessian 矩阵的信息来指导搜索，以解决这个问题。其中最简单的方法是牛顿法 (Newton's method)。牛顿法基于一个二阶泰勒展开来近似  $\mathbf{x}^{(0)}$  附近的  $f(\mathbf{x})$ :

$$f(\mathbf{x}) \approx f(\mathbf{x}^{(0)}) + (\mathbf{x} - \mathbf{x}^{(0)})^\top \nabla_{\mathbf{x}} f(\mathbf{x}^{(0)}) + \frac{1}{2} (\mathbf{x} - \mathbf{x}^{(0)})^\top \mathbf{H}(f)(\mathbf{x}^{(0)}) (\mathbf{x} - \mathbf{x}^{(0)}). \quad (4.11)$$

接着通过计算，我们可以得到这个函数的临界点：

$$\mathbf{x}^* = \mathbf{x}^{(0)} - \mathbf{H}(f)(\mathbf{x}^{(0)})^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}^{(0)}). \quad (4.12)$$

当  $f$  是一个正定二次函数时，牛顿法只要应用一次式 (4.12) 就能直接跳到函数的最小点。如果  $f$  不是一个真正二次但能在局部近似为正定二次，牛顿法则需要多次迭

代应用式 (4.12)。迭代地更新近似函数和跳到近似函数的最小点可以比梯度下降更快地到达临界点。这在接近局部极小点时是一个特别有用的性质，但是在鞍点附近是有害的。如式 (8.2.3) 所讨论的，当附近的临界点是最小点 (Hessian 的所有特征值都是正的) 时牛顿法才适用，而梯度下降不会被吸引到鞍点(除非梯度指向鞍点)。

仅使用梯度信息的优化算法被称为 **一阶优化算法** (first-order optimization algorithms)，如梯度下降。使用 Hessian 矩阵的优化算法被称为 **二阶最优化算法** (second-order optimization algorithms)(Nocedal and Wright, 2006)，如牛顿法。

在本书大多数上下文中使用的优化算法适用于各种各样的函数，但几乎没有保证。因为在深度学习中使用的函数族是相当复杂的，所以深度学习算法往往缺乏保证。在许多其他领域，优化的主要方法是为有限的函数族设计优化算法。

在深度学习的背景下，限制函数满足 **Lipschitz 连续** (Lipschitz continuous) 或其导数Lipschitz连续可以获得一些保证。Lipschitz 连续函数的变化速度以 **Lipschitz 常数** (Lipschitz constant)  $\mathcal{L}$  为界：

$$\forall \mathbf{x}, \forall \mathbf{y}, |f(\mathbf{x}) - f(\mathbf{y})| \leq \mathcal{L} \|\mathbf{x} - \mathbf{y}\|_2. \quad (4.13)$$

这个属性允许我们量化我们的假设——梯度下降等算法导致的输入的微小变化将使输出只产生微小变化，因此是很有用的。Lipschitz 连续性也是相当弱的约束，并且深度学习中很多优化问题经过相对较小的修改后就能变得 Lipschitz 连续。

最成功的特定优化领域或许是 **凸优化** (Convex optimization)。凸优化通过更强的限制提供更多的保证。凸优化算法只对凸函数适用，即 Hessian 处处半正定的函数。因为这些函数没有鞍点而且其所有局部极小点必然是全局最小点，所以表现很好。然而，深度学习中的大多数问题都难以表示成凸优化的形式。凸优化仅用作一些深度学习算法的子程序。凸优化中的分析思路对证明深度学习算法的收敛性非常有用，然而一般来说，深度学习背景下凸优化的重要性大大减少。有关凸优化的详细信息，详见 Boyd and Vandenberghe (2004) 或 Rockafellar (1997)。

## 4.4 约束优化

有时候，在  $\mathbf{x}$  的所有可能值下最大化或最小化一个函数  $f(\mathbf{x})$  不是我们所希望的。相反，我们可能希望在  $\mathbf{x}$  的某些集合  $\mathbb{S}$  中找  $f(\mathbf{x})$  的最大值或最小值。这被称为 **约束优化** (constrained optimization)。在约束优化术语中，集合  $\mathbb{S}$  内的点  $\mathbf{x}$  被称

为可行 (feasible) 点。

我们常常希望找到在某种意义上小的解。针对这种情况下的常见方法是强加一个范数约束, 如  $\|\mathbf{x}\| \leq 1$ 。

约束优化的一个简单方法是将约束考虑在内后简单地对梯度下降进行修改。如果我们使用一个小的恒定步长  $\epsilon$ , 我们可以先取梯度下降的单步结果, 然后将结果投影回  $\mathbb{S}$ 。如果我们使用线搜索, 我们只能在步长为  $\epsilon$  范围内搜索可行的新  $\mathbf{x}$  点, 或者我们可以将线上的每个点投影到约束区域。如果可能的话, 在梯度下降或线搜索前将梯度投影到可行域的切空间会更高效 (Rosen, 1960)。

一个更复杂的方法是设计一个不同的、无约束的优化问题, 其解可以转化成原始约束优化问题的解。例如, 我们要在  $\mathbf{x} \in \mathbb{R}^2$  中最小化  $f(\mathbf{x})$ , 其中  $\mathbf{x}$  约束为具有单位  $L^2$  范数。我们可以关于  $\theta$  最小化  $g(\theta) = f([\cos \theta, \sin \theta]^\top)$ , 最后返回  $[\cos \theta, \sin \theta]$  作为原问题的解。这种方法需要创造性; 优化问题之间的转换必须专门根据我们遇到的每一种情况进行设计。

**Karush–Kuhn–Tucker (KKT)** 方法<sup>2</sup>是针对约束优化非常通用的解决方案。为介绍KKT方法, 我们引入一个称为**广义 Lagrangian** (generalized Lagrangian) 或**广义 Lagrange 函数** (generalized Lagrange function) 的新函数。

为了定义Lagrangian, 我们先要通过等式和不等式的形式描述  $\mathbb{S}$ 。我们希望通过  $m$  个函数  $g^{(i)}$  和  $n$  个函数  $h^{(j)}$  描述  $\mathbb{S}$ , 那么  $\mathbb{S}$  可以表示为  $\mathbb{S} = \{\mathbf{x} \mid \forall i, g^{(i)}(\mathbf{x}) = 0 \text{ and } \forall j, h^{(j)}(\mathbf{x}) \leq 0\}$ 。其中涉及  $g^{(i)}$  的等式称为**等式约束** (equality constraint), 涉及  $h^{(j)}$  的不等式称为**不等式约束** (inequality constraint)。

我们为每个约束引入新的变量  $\lambda_i$  和  $\alpha_j$ , 这些新变量被称为 KKT 乘子。广义 Lagrangian 可以如下定义:

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\mathbf{x}) + \sum_i \lambda_i g^{(i)}(\mathbf{x}) + \sum_j \alpha_j h^{(j)}(\mathbf{x}). \quad (4.14)$$

现在, 我们可以通过优化无约束的广义 Lagrangian 解决约束最小化问题。只要存在至少一个可行点且  $f(\mathbf{x})$  不允许取  $\infty$ , 那么

$$\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) \quad (4.15)$$

---

<sup>2</sup>KKT 方法是 Lagrange 乘子法 (只允许等式约束) 的推广。

与如下函数有相同的最优目标函数值和最优点集  $\mathbf{x}$

$$\min_{\mathbf{x} \in \mathbb{S}} f(\mathbf{x}). \quad (4.16)$$

这是因为当约束满足时，

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\mathbf{x}), \quad (4.17)$$

而违反任意约束时，

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = \infty. \quad (4.18)$$

这些性质保证不可行点不会是最佳的，并且可行点范围内的最优点不变。

要解决约束最大化问题，我们可以构造  $-f(\mathbf{x})$  的广义 Lagrange 函数，从而导致以下优化问题：

$$\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} -f(\mathbf{x}) + \sum_i \lambda_i g^{(i)}(\mathbf{x}) + \sum_j \alpha_j h^{(j)}(\mathbf{x}). \quad (4.19)$$

我们也可将其转换为在外层最大化问题：

$$\max_{\mathbf{x}} \min_{\boldsymbol{\lambda}} \min_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} f(\mathbf{x}) + \sum_i \lambda_i g^{(i)}(\mathbf{x}) - \sum_j \alpha_j h^{(j)}(\mathbf{x}). \quad (4.20)$$

等式约束对应项的符号并不重要；因为优化可以自由选择每个  $\lambda_i$  的符号，我们可以随意将其定义为加法或减法。

不等式约束特别有趣。如果  $h^{(i)}(\mathbf{x}^*) = 0$ ，我们就说这个约束  $h^{(i)}(\mathbf{x})$  是活跃 (active) 的。如果约束不是活跃的，则有该约束的问题的解与去掉该约束的问题的解至少存在一个相同的局部解。一个不活跃约束有可能排除其他解。例如，整个区域（代价相等的宽平区域）都是全局最优点的凸问题可能因约束消去其中的某个子区域，或在非凸问题的情况下，收敛时不活跃的约束可能排除了较好的局部驻点。然而，无论不活跃的约束是否被包括在内，收敛时找到的点仍然是一个驻点。因为一个不活跃的约束  $h^{(i)}$  必有负值，那么  $\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$  中的  $\alpha_i = 0$ 。因此，我们可以观察到在该解中  $\boldsymbol{\alpha} \odot \mathbf{h}(\mathbf{x}) = 0$ 。换句话说，对于所有的  $i$ ,  $\alpha_i \geq 0$  或  $h^{(j)}(\mathbf{x}) \leq 0$  在收敛时必有一个是活跃的。为了获得关于这个想法的一些直观解释，我们可以说这个解是由不等式强加的边界，我们必须通过对应的 KKT 乘子影响  $\mathbf{x}$  的解，或者不等式对解没有影响，我们则归零 KKT 乘子。

我们可以使用一组简单的性质来描述约束优化问题的最优点。这些性质称为 **Karush–Kuhn–Tucker** (KKT) 条件 (Karush, 1939; Kuhn and Tucker, 1951)。这些是确定一个点是最优点的必要条件，但不一定是充分条件。这些条件是：

- 广义 Lagrangian 的梯度为零。
- 所有关于  $\mathbf{x}$  和 KKT 乘子的约束都满足。
- 不等式约束显示的“互补松弛性”： $\alpha \odot h(\mathbf{x}) = 0$ 。

有关 KKT 方法的详细信息，请参阅 Nocedal and Wright (2006)。

## 4.5 实例：线性最小二乘

假设我们希望找到最小化下式的  $\mathbf{x}$  值

$$f(\mathbf{x}) = \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2. \quad (4.21)$$

存在专门的线性代数算法能够高效地解决这个问题；但是，我们也可以探索如何使用基于梯度的优化来解决这个问题，这可以作为这些技术是如何工作的一个简单例子。

首先，我们计算梯度：

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{A}^\top (\mathbf{Ax} - \mathbf{b}) = \mathbf{A}^\top \mathbf{Ax} - \mathbf{A}^\top \mathbf{b}. \quad (4.22)$$

然后，我们可以采用小的步长，并按照这个梯度下降。见算法 4.1 中的详细信息。

---

**算法 4.1** 从任意点  $\mathbf{x}$  开始，使用梯度下降关于  $\mathbf{x}$  最小化  $f(\mathbf{x}) = \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2$  的算法。

---

将步长 ( $\epsilon$ ) 和容差 ( $\delta$ ) 设为小的正数。

```

while  $\|\mathbf{A}^\top \mathbf{Ax} - \mathbf{A}^\top \mathbf{b}\|_2 > \delta$  do
     $\mathbf{x} \leftarrow \mathbf{x} - \epsilon (\mathbf{A}^\top \mathbf{Ax} - \mathbf{A}^\top \mathbf{b})$ 
end while
```

---

我们也可以使用牛顿法解决这个问题。因为在这个情况下，真实函数是二次的，牛顿法所用的二次近似是精确的，该算法会在一步后收敛到全局最小点。

现在假设我们希望最小化同样的函数，但受  $\mathbf{x}^\top \mathbf{x} \leq 1$  的约束。要做到这一点，我们引入 Lagrangian

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda(\mathbf{x}^\top \mathbf{x} - 1). \quad (4.23)$$

现在，我们解决以下问题

$$\min_{\mathbf{x}} \max_{\lambda, \lambda \geq 0} L(\mathbf{x}, \lambda). \quad (4.24)$$

我们可以用 Moore-Penrose 伪逆： $\mathbf{x} = \mathbf{A}^+ \mathbf{b}$  找到无约束最小二乘问题的最小范数解。如果这一点是可行，那么这也是约束问题的解。否则，我们必须找到约束是活跃的解。关于  $\mathbf{x}$  对 Lagrangian 微分，我们得到方程

$$\mathbf{A}^\top \mathbf{A} \mathbf{x} - \mathbf{A}^\top \mathbf{b} + 2\lambda \mathbf{x} = 0. \quad (4.25)$$

这就告诉我们，该解的形式将会是

$$\mathbf{x} = (\mathbf{A}^\top \mathbf{A} + 2\lambda \mathbf{I})^{-1} \mathbf{A}^\top \mathbf{b}. \quad (4.26)$$

$\lambda$  的选择必须使结果服从约束。我们可以关于  $\lambda$  进行梯度上升找到这个值。为了做到这一点，观察

$$\frac{\partial}{\partial \lambda} L(\mathbf{x}, \lambda) = \mathbf{x}^\top \mathbf{x} - 1. \quad (4.27)$$

当  $\mathbf{x}$  的范数超过 1 时，该导数是正的，所以为了跟随导数上坡并相对  $\lambda$  增加 Lagrangian，我们需要增加  $\lambda$ 。因为  $\mathbf{x}^\top \mathbf{x}$  的惩罚系数增加了，求解关于  $\mathbf{x}$  的线性方程现在将得到具有较小范数的解。求解线性方程和调整  $\lambda$  的过程将一直持续到  $\mathbf{x}$  具有正确的范数并且关于  $\lambda$  的导数是 0。

本章总结了开发机器学习算法所需的数学基础。现在，我们已经准备好建立和分析一些成熟的学习系统。

# 第五章 机器学习基础

深度学习是机器学习的一个特定分支。我们要想充分理解深度学习，必须对机器学习的基本原理有深刻的理解。本章将探讨贯穿本书其余部分的一些机器学习重要原理。我们建议新手读者或是希望更全面了解的读者参考一些更全面覆盖基础知识的机器学习参考书，例如 Murphy (2012) 或者 Bishop (2006)。如果你已经熟知机器学习，可以跳过前面的部分，前往第 5.11 节。第 5.11 节涵盖了一些传统机器学习技术观点，这些技术对深度学习的发展有着深远影响。

首先，我们将介绍学习算法的定义，并介绍一个简单的示例：线性回归算法。接下来，我们会探讨拟合训练数据与寻找能够泛化到新数据的模式存在哪些不同的挑战。大部分机器学习算法都有超参数（必须在学习算法外设定）；我们将探讨如何使用额外的数据设置超参数。机器学习本质上属于应用统计学，更多地关注于如何用计算机统计地估计复杂函数，不太关注为这些函数提供置信区间；因此我们会探讨两种统计学的主要方法：频率派估计和贝叶斯推断。大部分机器学习算法可以分成监督学习和无监督学习两类；我们将探讨不同的分类，并为每类提供一些简单的机器学习算法作为示例。大部分深度学习算法都是基于被称为随机梯度下降的算法求解的。我们将介绍如何组合不同的算法部分，例如优化算法、代价函数、模型和数据集，来建立一个机器学习算法。最后在第 5.11 节，我们会介绍一些限制传统机器学习泛化能力的因素。这些挑战促进了解决这些问题的深度学习算法的发展。

## 5.1 学习算法

机器学习算法是一种能够从数据中学习的算法。然而，我们所谓的“学习”是什么意思呢？Mitchell (1997) 提供了一个简洁的定义：“对于某类任务  $T$  和性能度量  $P$ ，一个计算机程序被认为可以从经验  $E$  中学习是指，通过经验  $E$  改进后，它在任

务  $T$  上由性能度量  $P$  衡量的性能有所提升。”经验  $E$ , 任务  $T$  和性能度量  $P$  的定义范围非常宽广, 在本书中我们并不会试图去解释这些定义的具体意义。相反, 我们会在接下来的章节中提供直观的解释和示例来介绍不同的任务、性能度量和经验, 这些将被用来构建机器学习算法。

### 5.1.1 任务 $T$

机器学习可以让我们解决一些人为设计和使用确定性程序很难解决的问题。从科学和哲学的角度来看, 机器学习受到关注是因为提高我们对机器学习的认识需要提高我们对智能背后原理的理解。

从“任务”的相对正式的定义上说, 学习过程本身不能算是任务。学习是我们所谓的获取完成任务的能力。例如, 我们的目标是使机器人能够行走, 那么行走便是任务。我们可以编程让机器人学会如何行走, 或者可以人工编写特定的指令来指导机器人如何行走。

通常机器学习任务定义为机器学习系统应该如何处理 **样本** (example)。样本是指我们从某些希望机器学习系统处理的对象或事件中收集到的已经量化的**特征** (feature) 的集合。我们通常会将样本表示成一个向量  $\mathbf{x} \in \mathbb{R}^n$ , 其中向量的每一个元素  $x_i$  是一个特征。例如, 一张图片的特征通常是指这张图片的像素值。

机器学习可以解决很多类型的任务。一些非常常见的机器学习任务列举如下:

- **分类:** 在这类任务中, 计算机程序需要指定某些输入属于  $k$  类中的哪一类。为了完成这个任务, 学习算法通常会返回一个函数  $f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$ 。当  $y = f(\mathbf{x})$  时, 模型将向量  $\mathbf{x}$  所代表的输入分类到数字码  $y$  所代表的类别。还有一些其他的分类问题, 例如,  $f$  输出的是不同类别的概率分布。分类任务中有一个任务是对象识别, 其中输入是图片 (通常由一组像素亮度值表示), 输出是表示图片物体的数字码。例如, Willow Garage PR2 机器人能像服务员一样识别不同饮料, 并送给点餐的顾客 (Goodfellow *et al.*, 2010)。目前, 最好的对象识别工作正是基于深度学习的 (Krizhevsky *et al.*, 2012a; Ioffe and Szegedy, 2015)。对象识别同时也是计算机识别人脸的基本技术, 可用于标记相片合辑中的人脸 (Taigman *et al.*, 2014), 有助于计算机更自然地与用户交互。
- **输入缺失分类:** 当输入向量的每个度量不被保证的时候, 分类问题将会变得更有挑战性。为了解决分类任务, 学习算法只需要定义一个从输入向量映射到输

出类别的函数。当一些输入可能丢失时，学习算法必须学习一组函数，而不是单个分类函数。每个函数对应着分类具有不同缺失输入子集的  $x$ 。这种情况在医疗诊断中经常出现，因为很多类型的医学测试是昂贵的，对身体有害的。有效地定义这样一个大集合函数的方法是学习所有相关变量的概率分布，然后通过边缘化缺失变量来解决分类任务。使用  $n$  个输入变量，我们现在可以获得每个可能的缺失输入集合所需的所有  $2^n$  个不同的分类函数，但是计算机程序仅需要学习一个描述联合概率分布的函数。参见 Goodfellow *et al.* (2013d) 了解以这种方式将深度概率模型应用于这类任务的示例。本节中描述的许多其他任务也可以推广到缺失输入的情况；缺失输入分类只是机器学习能够解决的问题的一个示例。

- **回归：**在这类任务中，计算机程序需要对给定输入预测数值。为了解决这个任务，学习算法需要输出函数  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 。除了返回结果的形式不一样外，这类问题和分类问题是很像的。这类任务的一个示例是预测投保人的索赔金额（用于设置保险费），或者预测证券未来的价格。这类预测也用在算法交易中。
- **转录：**这类任务中，机器学习系统观测一些相对非结构化表示的数据，并转录信息为离散的文本形式。例如，光学字符识别要求计算机程序根据文本图片返回文字序列（ASCII 码或者 Unicode 码）。谷歌街景以这种方式使用深度学习处理街道编号 (Goodfellow *et al.*, 2014d)。另一个例子是语音识别，计算机程序输入一段音频波形，输出一序列音频记录中所说的字符或单词 ID 的编码。深度学习是现代语音识别系统的重要组成部分，被各大公司广泛使用，包括微软，IBM 和谷歌 (Hinton *et al.*, 2012b)。
- **机器翻译：**在机器翻译任务中，输入是一种语言的符号序列，计算机程序必须将其转化成另一种语言的符号序列。这通常适用于自然语言，如将英语译成法语。最近，深度学习已经开始在这个任务上产生重要影响 (Sutskever *et al.*, 2014; Bahdanau *et al.*, 2015)。
- **结构化输出：**结构化输出任务的输出是向量或者其他包含多个值的数据结构，并且构成输出的这些不同元素间具有重要关系。这是一个很大的范畴，包括上述转录任务和翻译任务在内的很多其他任务。例如语法分析——映射自然语言句子到语法结构树，并标记树的节点为动词、名词、副词等等。参考 Collobert (2011) 将深度学习应用到语法分析的示例。另一个例子是图像的像素级分割，将每一个像素分配到特定类别。例如，深度学习可用于标注航拍照片中的道路

位置 (Mnih and Hinton, 2010)。在这些标注型的任务中，输出的结构形式不需要和输入尽可能相似。例如，在为图片添加描述的任务中，计算机程序观察到一幅图，输出描述这幅图的自然语言句子 (Kiros *et al.*, 2014a,b; Mao *et al.*, 2014; Vinyals *et al.*, 2015b; Donahue *et al.*, 2014; Karpathy and Li, 2015; Fang *et al.*, 2015; Xu *et al.*, 2015)。这类任务被称为结构化输出任务是因为输出值之间内部紧密相关。例如，为图片添加标题的程序输出的单词必须组合成一个通顺的句子。

- **异常检测：**在这类任务中，计算机程序在一组事件或对象中筛选，并标记不正常或非典型的个体。异常检测任务的一个示例是信用卡欺诈检测。通过对你的购买习惯建模，信用卡公司可以检测到你的卡是否被滥用。如果窃贼窃取你的信用卡或信用卡信息，窃贼采购物品的分布通常和你的不同。当该卡发生了不正常的购买行为时，信用卡公司可以尽快冻结该卡以防欺诈。参考 Chandola *et al.* (2009) 了解欺诈检测方法。
- **合成和采样：**在这类任务中，机器学习程序生成一些和训练数据相似的新样本。通过机器学习，合成和采样可能在媒体应用中非常有用，可以避免艺术家大量昂贵或者乏味费时的手动工作。例如，视频游戏可以自动生成大型物体或风景的纹理，而不是让艺术家手动标记每个像素 (Luo *et al.*, 2013)。在某些情况下，我们希望采样或合成过程可以根据给定的输入生成一些特定类型的输出。例如，在语音合成任务中，我们提供书写的句子，要求程序输出这个句子语音的音频波形。这是一类结构化输出任务，但是多了每个输入并非只有一个正确输出的条件，并且我们明确希望输出有很多变化，这可以使结果看上去更加自然和真实。
- **缺失值填补：**在这类任务中，机器学习算法给定一个新样本  $\mathbf{x} \in \mathbb{R}^n$ ， $\mathbf{x}$  中某些元素  $x_i$  缺失。算法必须填补这些缺失值。
- **去噪：**在这类任务中，机器学习算法的输入是，干净样本  $\mathbf{x} \in \mathbb{R}^n$  经过未知损坏过程后得到的损坏样本  $\tilde{\mathbf{x}} \in \mathbb{R}^n$ 。算法根据损坏后的样本  $\tilde{\mathbf{x}}$  预测干净的样本  $\mathbf{x}$ ，或者更一般地预测条件概率分布  $p(\mathbf{x} | \tilde{\mathbf{x}})$ 。
- **密度估计或概率质量函数估计：**在密度估计问题中，机器学习算法学习函数  $p_{\text{model}} : \mathbb{R}^n \rightarrow \mathbb{R}$ ，其中  $p_{\text{model}}(\mathbf{x})$  可以解释成样本采样空间的概率密度函数（如果  $\mathbf{x}$  是连续的）或者概率质量函数（如果  $\mathbf{x}$  是离散的）。要做好这样的任务

(当我们讨论性能度量  $P$  时，我们会明确定义任务是什么)，算法需要学习观测到的数据的结构。算法必须知道什么情况下样本聚集出现，什么情况下不太可能出现。以上描述的大多数任务都要求学习算法至少能隐式地捕获概率分布的结构。密度估计可以让我们显式地捕获该分布。原则上，我们可以在该分布上计算以便解决其他任务。例如，如果我们通过密度估计得到了概率分布  $p(\mathbf{x})$ ，我们可以用该分布解决缺失值填补任务。如果  $x_i$  的值是缺失的，但是其他的变量值  $\mathbf{x}_{-i}$  已知，那么我们可以得到条件概率分布  $p(x_i | \mathbf{x}_{-i})$ 。实际情况中，密度估计并不能够解决所有这类问题，因为在很多情况下  $p(\mathbf{x})$  是难以计算的。

当然，还有很多其他同类型或其他类型的任务。这里我们列举的任务类型只是用来介绍机器学习可以做哪些任务，并非严格地定义机器学习任务分类。

### 5.1.2 性能度量 $P$

为了评估机器学习算法的能力，我们必须设计其性能的定量度量。通常性能度量  $P$  是特定于系统执行的任务  $T$  而言的。

对于诸如分类、缺失输入分类和转录任务，我们通常度量模型的准确率 (accuracy)。准确率是指该模型输出正确结果的样本比率。我们也可以通过错误率 (error rate) 得到相同的信息。错误率是指该模型输出错误结果的样本比率。我们通常把错误率称为 0 – 1 损失的期望。在一个特定的样本上，如果结果是对的，那么 0 – 1 损失是 0；否则是 1。但是对于密度估计这类任务而言，度量准确率，错误率或者其他类型的 0 – 1 损失是没有意义的。反之，我们必须使用不同的性能度量，使模型对每个样本都输出一个连续数值的得分。最常用的方法是输出模型在一些样本上概率对数的平均值。

通常，我们会更加关注机器学习算法在未观测数据上的性能如何，因为这将决定其在实际应用中的性能。因此，我们使用测试集 (test set) 数据来评估系统性能，将其与训练机器学习系统的训练集数据分开。

性能度量的选择或许看上去简单且客观，但是选择一个与系统理想表现对应的性能度量通常是很难的。

在某些情况下，这是因为很难确定应该度量什么。例如，在执行转录任务时，我们是应该度量系统转录整个序列的准确率，还是应该用一个更细粒度的指标，对序列中正确的部分元素以正面评价？在执行回归任务时，我们应该更多地惩罚频繁犯

一些中等错误的系统，还是较少犯错但是犯很大错误的系统？这些设计的选择取决于应用。

还有一些情况，我们知道应该度量哪些数值，但是度量它们不太现实。这种情况经常出现在密度估计中。很多最好的概率模型只能隐式地表示概率分布。在许多这类模型中，计算空间中特定点的概率是不可行的。在这些情况下，我们必须设计一个仍然对应于设计对象的替代标准，或者设计一个理想标准的良好近似。

### 5.1.3 经验 $E$

根据学习过程中的不同经验，机器学习算法可以大致分类为 **无监督**（unsupervised）算法和 **监督**（supervised）算法。

本书中的大部分学习算法可以被理解为在整个 **数据集**（dataset）上获取经验。数据集是指很多样本组成的集合，如第 5.1.1 节所定义的。有时我们也将样本称为 **数据点**（data point）。

Iris（鸢尾花卉）数据集（Fisher, 1936）是统计学家和机器学习研究者使用了很久的数据集。它是 150 个鸢尾花卉植物不同部分测量结果的集合。每个单独的植物对应一个样本。每个样本的特征是该植物不同部分的测量结果：萼片长度、萼片宽度、花瓣长度和花瓣宽度。这个数据集也记录了每个植物属于什么品种，其中共有三个不同的品种。

**无监督学习算法**（unsupervised learning algorithm）训练含有很多特征的数据集，然后学习出这个数据集上有用的结构性质。在深度学习中，我们通常要学习生成数据集的整个概率分布，显式地，比如密度估计，或是隐式地，比如合成或去噪。还有一些其他类型的无监督学习任务，例如聚类，将数据集分成相似样本的集合。

**监督学习算法**（supervised learning algorithm）训练含有很多特征的数据集，不过数据集中的样本都有一个 **标签**（label）或 **目标**（target）。例如，Iris 数据集注明了每个鸢尾花卉样本属于什么品种。监督学习算法通过研究 Iris 数据集，学习如何根据测量结果将样本划分为三个不同品种。

大致说来，无监督学习涉及到观察随机向量  $\mathbf{x}$  的好几个样本，试图显式或隐式地学习出概率分布  $p(\mathbf{x})$ ，或者是该分布一些有意思的性质；而监督学习包含观察随机向量  $\mathbf{x}$  及其相关联的值或向量  $\mathbf{y}$ ，然后从  $\mathbf{x}$  预测  $\mathbf{y}$ ，通常是估计  $p(\mathbf{y} | \mathbf{x})$ 。术语 **监督学习**（supervised learning）源自这样一个视角，教员或者老师提供目标  $\mathbf{y}$  给机器

学习系统，指导其应该做什么。在无监督学习中，没有教员或者老师，算法必须学会在没有指导的情况下理解数据。

无监督学习和监督学习不是严格定义的术语。它们之间界线通常是模糊的。很多机器学习技术可以用于这两个任务。例如，概率的链式法则表明对于向量  $\mathbf{x} \in \mathbb{R}^n$ ，联合分布可以分解成

$$p(\mathbf{x}) = \prod_{i=1}^n p(x_i | x_1, \dots, x_{i-1}). \quad (5.1)$$

该分解意味着我们可以将其拆分成  $n$  个监督学习问题，来解决表面上的无监督学习  $p(\mathbf{x})$ 。另外，我们求解监督学习问题  $p(y | \mathbf{x})$  时，也可以使用传统的无监督学习策略学习联合分布  $p(\mathbf{x}, y)$ ，然后推断

$$p(y | \mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')}. \quad (5.2)$$

尽管无监督学习和监督学习并非完全没有交集的正式概念，它们确实有助于粗略分类我们研究机器学习算法时遇到的问题。传统地，人们将回归、分类或者结构化输出问题称为监督学习。支持其他任务的密度估计通常被称为无监督学习。

学习范式的其他变种也是有可能的。例如，半监督学习中，一些样本有监督目标，但其他样本没有。在多实例学习中，样本的整个集合被标记为含有或者不含有该类的样本，但是集合中单独的样本是没有标记的。参考 Kotzias *et al.* (2015) 了解最近深度模型进行多实例学习的示例。

有些机器学习算法并不是训练于一个固定的数据集上。例如，**强化学习** (reinforcement learning) 算法会和环境进行交互，所以学习系统和它的训练过程会有反馈回路。这类算法超出了本书的范畴。请参考 Sutton and Barto (1998) 或 Bertsekas and Tsitsiklis (1996) 了解强化学习相关知识，Mnih *et al.* (2013) 介绍了强化学习方向的深度学习方法。

大部分机器学习算法简单地训练于一个数据集上。数据集可以用很多不同方式来表示。在所有的情况下，数据集都是样本的集合，而样本是特征的集合。

表示数据集的常用方法是**设计矩阵** (design matrix)。设计矩阵的每一行包含一个不同的样本。每一列对应不同的特征。例如，Iris 数据集包含 150 个样本，每个样本有 4 个特征。这意味着我们可以将该数据集表示为设计矩阵  $\mathbf{X} \in \mathbb{R}^{150 \times 4}$ ，其中  $X_{i,1}$  表示第  $i$  个植物的萼片长度， $X_{i,2}$  表示第  $i$  个植物的萼片宽度等等。我们在本书中描述的大部分学习算法都是讲述它们是如何运行在设计矩阵数据集上的。

当然，每一个样本都能表示成向量，并且这些向量的维度相同，才能将一个数据集表示成设计矩阵。这一点并非永远可能。例如，你有不同宽度和高度的照片的集合，那么不同的照片将会包含不同数量的像素。因此不是所有的照片都可以表示成相同长度的向量。第 9.7 节和第十章将会介绍如何处理这些不同类型的异构数据。在上述这类情况下，我们不会将数据集表示成  $m$  行的矩阵，而是表示成  $m$  个元素的结合： $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(m)}\}$ 。这种表示方式意味着样本向量  $\mathbf{x}^{(i)}$  和  $\mathbf{x}^{(j)}$  可以有不同的大小。

在监督学习中，样本包含一个标签或目标和一组特征。例如，我们希望使用学习算法从照片中识别对象。我们需要明确哪些对象会出现在每张照片中。我们或许会用数字编码表示，如 0 表示人、1 表示车、2 表示猫等等。通常在处理包含观测特征的设计矩阵  $\mathbf{X}$  的数据集时，我们也会提供一个标签向量  $\mathbf{y}$ ，其中  $y_i$  表示样本  $i$  的标签。

当然，有时标签可能不止一个数。例如，如果我们想要训练语音模型转录整个句子，那么每个句子样本的标签是一个单词序列。

正如监督学习和无监督学习没有正式的定义，数据集或者经验也没有严格的区分。这里介绍的结构涵盖了大多数情况，但始终有可能为新的应用设计出新的结构。

#### 5.1.4 示例：线性回归

我们将机器学习算法定义为，通过经验以提高计算机程序在某些任务上性能的算法。这个定义有点抽象。为了使这个定义更具体点，我们展示一个简单的机器学习示例：**线性回归** (linear regression)。当我们介绍更多有助于理解机器学习特性的概念时，我们会反复回顾这个示例。

顾名思义，线性回归解决回归问题。换言之，我们的目标是建立一个系统，将向量  $\mathbf{x} \in \mathbb{R}^n$  作为输入，预测标量  $y \in \mathbb{R}$  作为输出。线性回归的输出是其输入的线性函数。令  $\hat{y}$  表示模型预测  $y$  应该取的值。我们定义输出为

$$\hat{y} = \mathbf{w}^\top \mathbf{x}, \quad (5.3)$$

其中  $\mathbf{w} \in \mathbb{R}^n$  是 **参数** (parameter) 向量。

参数是控制系统行为的值。在这种情况下， $w_i$  是系数，会和特征  $x_i$  相乘之后全部相加起来。我们可以将  $\mathbf{w}$  看作是一组决定每个特征如何影响预测的权重 (weight)。如果特征  $x_i$  对应的权重  $w_i$  是正的，那么特征的值增加，我们的预测值

$\hat{y}$  也会增加。如果特征  $x_i$  对应的权重  $w_i$  是负的，那么特征的值增加，我们的预测值  $\hat{y}$  会减少。如果特征权重的大小很大，那么它对预测有很大的影响；如果特征权重的大小是零，那么它对预测没有影响。

因此，我们可以定义任务  $T$ ：通过输出  $\hat{y} = \mathbf{w}^\top \mathbf{x}$  从  $\mathbf{x}$  预测  $y$ 。接下来我们需要定义性能度量—— $P$ 。

假设我们有  $m$  个输入样本组成的设计矩阵，我们不用它来训练模型，而是评估模型性能如何。我们也有每个样本对应的正确值  $y$  组成的回归目标向量。因为这个数据集只是用来评估性能，我们称之为 **测试集** (test set)。我们将输入的设计矩阵记作  $\mathbf{X}^{(\text{test})}$ ，回归目标向量记作  $\mathbf{y}^{(\text{test})}$ 。

度量模型性能的一种方法是计算模型在测试集上的 **均方误差** (mean squared error)。如果  $\hat{\mathbf{y}}^{(\text{test})}$  表示模型在测试集上的预测值，那么均方误差表示为：

$$\text{MSE}_{\text{test}} = \frac{1}{m} \sum_i (\hat{\mathbf{y}}^{(\text{test})} - \mathbf{y}^{(\text{test})})_i^2. \quad (5.4)$$

直观上，当  $\hat{\mathbf{y}}^{(\text{test})} = \mathbf{y}^{(\text{test})}$  时，我们会发现误差降为 0。我们也可以看到

$$\text{MSE}_{\text{test}} = \frac{1}{m} \left\| \hat{\mathbf{y}}^{(\text{test})} - \mathbf{y}^{(\text{test})} \right\|_2^2, \quad (5.5)$$

所以当预测值和目标值之间的欧几里得距离增加时，误差也会增加。

为了构建一个机器学习算法，我们需要设计一个算法，通过观察训练集  $(\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$  获得经验，减少  $\text{MSE}_{\text{test}}$  以改进权重  $\mathbf{w}$ 。一种直观方式（我们将在后续的第 5.5.1 节说明其合法性）是最小化训练集上的均方误差，即  $\text{MSE}_{\text{train}}$ 。

最小化  $\text{MSE}_{\text{train}}$ ，我们可以简单地求解其导数为  $\mathbf{0}$  的情况：

$$\nabla_{\mathbf{w}} \text{MSE}_{\text{train}} = 0 \quad (5.6)$$

$$\Rightarrow \nabla_{\mathbf{w}} \frac{1}{m} \left\| \hat{\mathbf{y}}^{(\text{train})} - \mathbf{y}^{(\text{train})} \right\|_2^2 = 0 \quad (5.7)$$

$$\Rightarrow \frac{1}{m} \nabla_{\mathbf{w}} \left\| \mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})} \right\|_2^2 = 0 \quad (5.8)$$

$$\Rightarrow \nabla_{\mathbf{w}} \left( \mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})} \right)^\top \left( \mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})} \right) = 0 \quad (5.9)$$

$$\Rightarrow \nabla_{\mathbf{w}} \left( \mathbf{w}^\top \mathbf{X}^{(\text{train})\top} \mathbf{X}^{(\text{train})} \mathbf{w} - 2 \mathbf{w}^\top \mathbf{X}^{(\text{train})\top} \mathbf{y}^{(\text{train})} + \mathbf{y}^{(\text{train})\top} \mathbf{y}^{(\text{train})} \right) = 0 \quad (5.10)$$

$$\Rightarrow 2 \mathbf{X}^{(\text{train})\top} \mathbf{X}^{(\text{train})} \mathbf{w} - 2 \mathbf{X}^{(\text{train})\top} \mathbf{y}^{(\text{train})} = 0 \quad (5.11)$$

$$\Rightarrow \mathbf{w} = \left( \mathbf{X}^{(\text{train})\top} \mathbf{X}^{(\text{train})} \right)^{-1} \mathbf{X}^{(\text{train})\top} \mathbf{y}^{(\text{train})} \quad (5.12)$$

通过式(5.12)给出解的系统方程被称为**正规方程**(normal equation)。计算式(5.12)构成了一个简单的机器学习算法。图5.1展示了线性回归算法的使用示例。

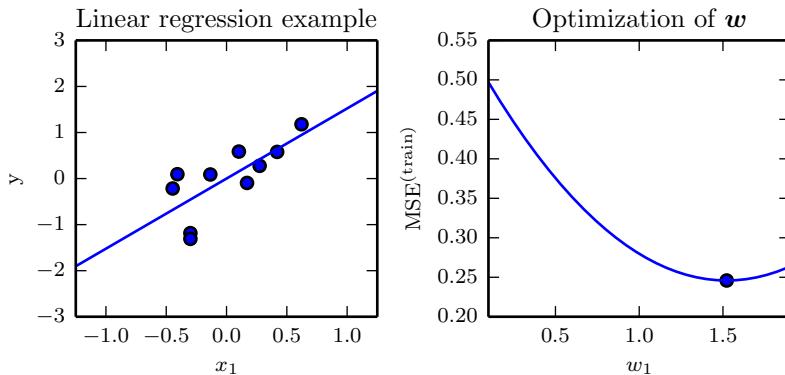


图5.1:一个线性回归问题，其中训练集包括十个数据点，每个数据点包含一个特征。因为只有一个特征，权重向量  $\mathbf{w}$  也只有一个要学习的参数  $w_1$ 。(左)我们可以观察到线性回归学习  $w_1$ ，从而使得直线  $y = w_1 x$  能够尽量接近穿过所有的训练点。(右)标注的点表示由正规方程学习到的  $w_1$  的值，我们发现它可以最小化训练集上的均方误差。

值得注意的是，术语**线性回归**(linear regression)通常用来指稍微复杂一些，附加额外参数(截距项  $b$ )的模型。在这个模型中，

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b, \quad (5.13)$$

因此从参数到预测的映射仍是一个线性函数，而从特征到预测的映射是一个仿射函数。如此扩展到仿射函数意味着模型预测的曲线仍然看起来像是一条直线，只是这条直线没必要经过原点。除了通过添加偏置参数  $b$ ，我们还可以使用仅含权重的模型，但是  $\mathbf{x}$  需要增加一项永远为1的元素。对应于额外1的权重起到了偏置参数的作用。当我们在本书中提到仿射函数时，我们会经常使用术语“线性”。

截距项  $b$  通常被称为仿射变换的**偏置**(bias)参数。这个术语的命名源自该变换的输出在没有任何输入时会偏移  $b$ 。它和统计偏差中指代统计估计算法的某个量的期望估计偏离真实值的意思是不一样的。

线性回归当然是一个极其简单且有局限的学习算法，但是它提供了一个说明学习算法如何工作的例子。在接下来的小节中，我们将会介绍一些设计学习算法的基

本原则，并说明如何使用这些原则来构建更复杂的学习算法。

## 5.2 容量、过拟合和欠拟合

机器学习的主要挑战是我们的算法必须能够在先前未观测的新输入上表现良好，而不只是在训练集上表现良好。在先前未观测到的输入上表现良好的能力被称为 **泛化** (generalization)。

通常情况下，当我们训练机器学习模型时，我们可以使用某个训练集，在训练集上计算一些被称为 **训练误差** (training error) 的度量误差，目标是降低训练误差。目前为止，我们讨论的是一个简单的优化问题。机器学习和优化不同的地方在于，我们也希望 **泛化误差** (generalization error) (也被称为 **测试误差** (test error)) 很低。泛化误差被定义为新输入的误差期望。这里，期望的计算基于不同的可能输入，这些输入采自于系统在现实中遇到的分布。

通常，我们度量模型在训练集中分出来的 **测试集** (test set) 样本上的性能，来评估机器学习模型的泛化误差。

在我们的线性回归示例中，我们通过最小化训练误差来训练模型，

$$\frac{1}{m^{(\text{train})}} \left\| \mathbf{X}^{(\text{train})} \mathbf{w} - \mathbf{y}^{(\text{train})} \right\|_2^2, \quad (5.14)$$

但是我们真正关注的是测试误差  $\frac{1}{m^{(\text{test})}} \left\| \mathbf{X}^{(\text{test})} \mathbf{w} - \mathbf{y}^{(\text{test})} \right\|_2^2$ 。

当我们只能观测到训练集时，我们如何才能影响测试集的性能呢？**统计学习理论** (statistical learning theory) 提供了一些答案。如果训练集和测试集的数据是任意收集的，那么我们能够做的确实很有限。如果我们可以对训练集和测试集数据的收集方式有些假设，那么我们能够对算法做些改进。

训练集和测试集数据通过数据集上被称为 **数据生成过程** (data generating process) 的概率分布生成。通常，我们会做一系列被统称为 **独立同分布假设** (i.i.d. assumption) 的假设。该假设是说，每个数据集中的样本都是彼此 **相互独立的** (independent)，并且训练集和测试集是 **同分布的** (identically distributed)，采样自相同的分布。这个假设使我们能够在单个样本的概率分布描述数据生成过程。然后相同的分布可以用来生成每一个训练样本和每一个测试样本。我们将这个共享的潜在分布称为 **数据生成分布** (data generating distribution)，记作  $p_{\text{data}}$ 。这个概率框架和独立同分布假设允许我们从数学上研究训练误差和测试误差之间的关系。

我们能观察到训练误差和测试误差之间的直接联系是，随机模型训练误差的期望和该模型测试误差的期望是一样的。假设我们有概率分布  $p(\mathbf{x}, y)$ ，从中重复采样生成训练集和测试集。对于某个固定的  $\mathbf{w}$ ，训练集误差的期望恰好和测试集误差的期望一样，这是因为这两个期望的计算都使用了相同的数据集生成过程。这两种情况的唯一区别是数据集的名字不同。

当然，当我们使用机器学习算法时，我们不会提前固定参数，然后采样得到两个数据集。我们采样得到训练集，然后挑选参数去降低训练集误差，然后采样得到测试集。在这个过程中，测试误差期望会大于或等于训练误差期望。以下是决定机器学习算法效果是否好的因素：

1. 降低训练误差。
2. 缩小训练误差和测试误差的差距。

这两个因素对应机器学习的两个主要挑战：欠拟合（underfitting）和过拟合（overfitting）。欠拟合是指模型不能在训练集上获得足够低的误差。而过拟合是指训练误差和测试误差之间的差距太大。

通过调整模型的容量（capacity），我们可以控制模型是否偏向于过拟合或者欠拟合。通俗地，模型的容量是指其拟合各种函数的能力。容量低的模型可能很难拟合训练集。容量高的模型可能会过拟合，因为记住了不适用于测试集的训练集性质。

一种控制训练算法容量的方法是选择假设空间（hypothesis space），即学习算法可以选择为解决方案的函数集。例如，线性回归算法将关于其输入的所有线性函数作为假设空间。广义线性回归的假设空间包括多项式函数，而非仅有线性函数。这样做就增加了模型的容量。

一次多项式提供了我们已经熟悉的线性回归模型，其预测如下：

$$\hat{y} = b + wx. \quad (5.15)$$

通过引入  $x^2$  作为线性回归模型的另一个特征，我们能够学习关于  $x$  的二次函数模型：

$$\hat{y} = b + w_1x + w_2x^2. \quad (5.16)$$

尽管该模型是输入的二次函数，但输出仍是参数的线性函数。因此我们仍然可以用正规方程得到模型的闭解。我们可以继续添加  $x$  的更高幂作为额外特征，例如下面的

9 次多项式：

$$\hat{y} = b + \sum_{i=1}^9 w_i x^i. \quad (5.17)$$

当机器学习算法的容量适合于所执行任务的复杂度和所提供的训练数据的数量时，算法效果通常会最佳。容量不足的模型不能解决复杂任务。容量高的模型能够解决复杂的任务，但是当其容量高于任务所需时，有可能会过拟合。

图 5.2 展示了这个原理的使用情况。我们比较了线性，二次和 9 次预测器拟合真实二次函数的效果。线性函数无法刻画真实函数的曲率，所以欠拟合。9 次函数能够表示正确的函数，但是因为训练参数比训练样本还多，所以它也能够表示无限多个刚好穿越训练样本点的很多其他函数。我们不太可能从这很多不同的解中选出一个泛化良好的。在这个问题中，二次模型非常符合任务的真实结构，因此它可以很好地泛化到新数据上。

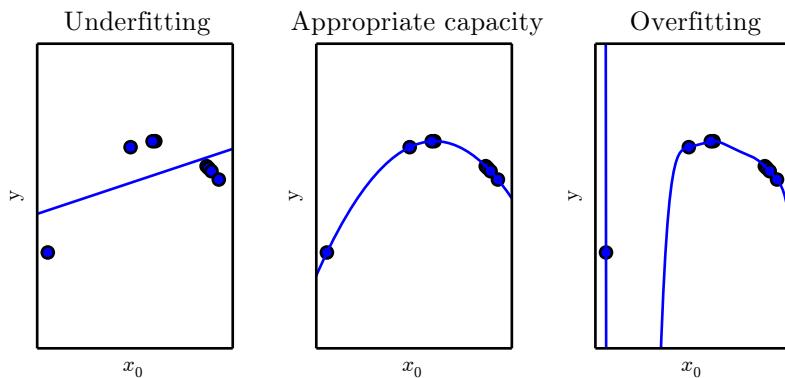


图 5.2：我们用三个模型拟合了这个训练集的样本。训练数据是通过随机抽取  $x$  然后用二次函数确定性地生成  $y$  来合成的。(左) 用一个线性函数拟合数据会导致欠拟合——它无法捕捉数据中的曲率信息。(中) 用二次函数拟合数据在未观察到的点上泛化得很好。这并不会导致明显的欠拟合或者过拟合。(右) 一个 9 阶的多项式拟合数据会导致过拟合。在这里我们使用 Moore-Penrose 伪逆来解这个欠定的正规方程。得出的解能够精确地穿过所有的训练点，但可惜我们无法提取有效的结构信息。在两个数据点之间它有一个真实的函数所不包含的深谷。在数据的左侧，它也会急剧增长，而在这一区域真实的函数却是下降的。

目前为止，我们探讨了通过改变输入特征的数目和加入这些特征对应的参数，改变模型的容量。事实上，还有很多方法可以改变模型的容量。容量不仅取决于模型的选择。模型规定了调整参数降低训练目标时，学习算法可以从哪些函数族中选择

函数。这被称为模型的 表示容量 (representational capacity)。在很多情况下，从这些函数中挑选出最优函数是非常困难的优化问题。实际中，学习算法不会真的找到最优函数，而仅是找到一个可以大大降低训练误差的函数。额外的限制因素，比如优化算法的不完美，意味着学习算法的 有效容量 (effective capacity) 可能小于模型族的表示容量。

提高机器学习模型泛化的现代思想可以追溯到早在托勒密时期的哲学家的思想。许多早期的学者提出一个简约原则，现在广泛被称为 奥卡姆剃刀 (Occam's razor) (c. 1287-1387)。该原则指出，在同样能够解释已知观测现象的假设中，我们应该挑选“最简单”的那一个。这个想法是在 20 世纪，由统计学习理论创始人形式化并精确化的 (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer *et al.*, 1989; Vapnik, 1995)。

统计学习理论提供了量化模型容量的不同方法。在这些中，最有名的是 **Vapnik-Chervonenkis 维度** (Vapnik-Chervonenkis dimension, VC)。VC 维度量二元分类器的容量。VC 维定义为该分类器能够分类的训练样本的最大数目。假设存在  $m$  个不同  $x$  点的训练集，分类器可以任意地标记该  $m$  个不同的  $x$  点，VC 维被定义为  $m$  的最大可能值。

量化模型的容量使得统计学习理论可以进行量化预测。统计学习理论中最重要的结论阐述了训练误差和泛化误差之间差异的上界随着模型容量增长而增长，但随着训练样本增多而下降 (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer *et al.*, 1989; Vapnik, 1995)。这些边界为机器学习算法可以有效解决问题提供了理论验证，但是它们很少应用于实际中的深度学习算法。一部分原因是边界太松，另一部分原因是很难确定深度学习算法的容量。由于有效容量受限于优化算法的能力，确定深度学习模型容量的问题特别困难。而且对于深度学习中的一般非凸优化问题，我们只有很少的理论分析。

我们必须记住虽然更简单的函数更可能泛化（训练误差和测试误差的差距小），但我们仍然需要选择一个充分复杂的假设以达到低的训练误差。通常，当模型容量上升时，训练误差会下降，直到其渐近最小可能误差（假设误差度量有最小值）。通常，泛化误差是一个关于模型容量的 U 形曲线函数。如图 5.3 所示。

为考虑容量任意高的极端情况，我们介绍 **非参数** (non-parametric) 模型的概念。至此，我们只探讨过参数模型，例如线性回归。参数模型学习的函数在观测到新数据前，参数向量的分量个数是有限且固定的。非参数模型没有这些限制。

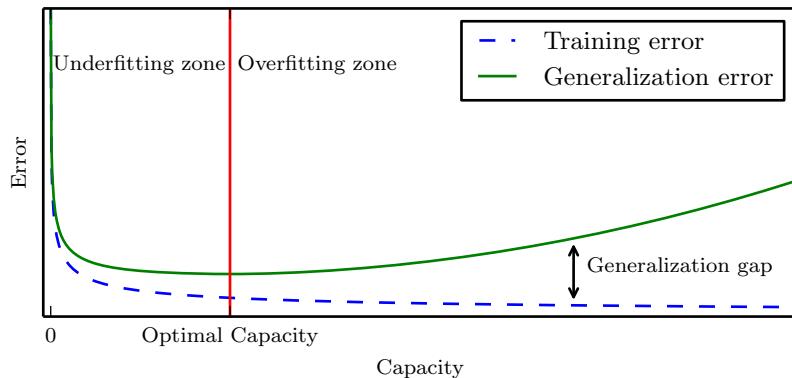


图 5.3: 容量和误差之间的典型关系。训练误差和测试误差表现得非常不同。在图的左端，训练误差和泛化误差都非常高。这是欠拟合机制 (underfitting regime)。当我们增加容量时，训练误差减小，但是训练误差和泛化误差之间的间距却不断扩大。最终，这个间距的大小超过了训练误差的下降，我们进入到了过拟合机制 (overfitting regime)，其中容量过大，超过了最佳容量 (optimal capacity)。

有时，非参数模型仅是一些不能实际实现的理论抽象（比如搜索所有可能概率分布的算法）。然而，我们也可以设计一些实用的非参数模型，使它们的复杂度和训练集大小有关。这种算法的一个示例是最近邻回归 (nearest neighbor regression)。不像线性回归有固定长度的向量作为权重，最近邻回归模型存储了训练集中所有的  $\mathbf{X}$  和  $\mathbf{y}$ 。当需要为测试点  $\mathbf{x}$  分类时，模型会查询训练集中离该点最近的点，并返回相关的回归目标。换言之， $\hat{y} = y_i$  其中  $i = \arg \min \|\mathbf{X}_{i,:} - \mathbf{x}\|_2^2$ 。该算法也可以扩展成  $L^2$  范数以外的距离度量，例如学成距离度量 (Goldberger *et al.*, 2005)。在最近向量不唯一的情况下，如果允许算法对所有离  $\mathbf{x}$  最近的  $\mathbf{X}_{i,:}$  关联的  $y_i$  求平均，那么该算法会在任意回归数据集上达到最小可能的训练误差（如果存在两个相同的输入对应不同的输出，那么训练误差可能会大于零）。

最后，我们也可以将参数学习算法嵌入另一个增加参数数目的算法来创建非参数学习算法。例如，我们可以想象这样一个算法，外层循环调整多项式的次数，内层循环通过线性回归学习模型。

理想模型假设我们能够预先知道生成数据的真实概率分布。然而这样的模型仍然会在很多问题上发生一些错误，因为分布中仍然会有一些噪声。在监督学习中，从  $\mathbf{x}$  到  $\mathbf{y}$  的映射可能内在是随机的，或者  $\mathbf{y}$  可能是其他变量（包括  $\mathbf{x}$  在内）的确定性

函数。从预先知道的真实分布  $p(\mathbf{x}, y)$  预测而出现的误差被称为 **贝叶斯误差** (Bayes error)。

训练误差和泛化误差会随训练集的大小发生变化。泛化误差的期望从不会因训练样本数目的增加而增加。对于非参数模型而言，更多的数据会得到更好的泛化能力，直到达到最佳可能的泛化误差。任何模型容量小于最优容量的固定参数模型会渐近到大于贝叶斯误差的误差值。如图 5.4 所示。值得注意的是，具有最优容量的模型仍然有可能在训练误差和泛化误差之间存在很大的差距。在这种情况下，我们可以通过收集更多的训练样本来缩小差距。

### 5.2.1 没有免费午餐定理

学习理论表明机器学习算法能够在有限个训练集样本中很好地泛化。这似乎违背一些基本的逻辑原则。归纳推理，或是从一组有限的样本中推断一般的规则，在逻辑上不是很有效。为了逻辑地推断一个规则去描述集合中的元素，我们必须具有集合中每个元素的信息。

在一定程度上，机器学习仅通过概率法则就可以避免这个问题，而无需使用纯逻辑推理整个确定性法则。机器学习保证找到一个在所关注的大多数样本上可能正确的规则。

可惜，即使这样也不能解决整个问题。机器学习的 **没有免费午餐定理** (no free lunch theorem) 表明 (Wolpert, 1996)，在所有可能的数据生成分布上平均之后，每一个分类算法在未事先观测的点上都有相同的错误率。换言之，在某种意义上，没有一个机器学习算法总是比其他的要好。我们能够设想的最先进的算法和简单地将所有点归为同一类的简单算法有着相同的平均性能 (在所有可能的任务上)。

幸运的是，这些结论仅在我们考虑所有可能的数据生成分布时才成立。在真实世界应用中，如果我们对遇到的概率分布进行假设的话，那么我们可以设计在这些分布上效果良好的学习算法。

这意味着机器学习研究的目标不是找一个通用学习算法或是绝对最好的学习算法。反之，我们的目标是理解什么样的分布与人工智能获取经验的“真实世界”相关，什么样的学习算法在我们关注的数据生成分布上效果最好。

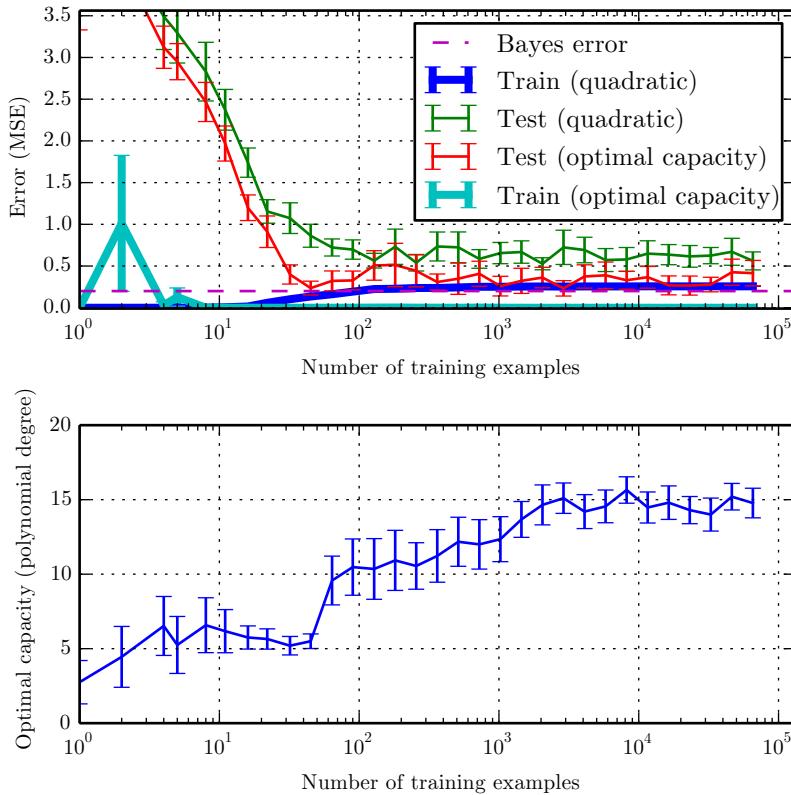


图 5.4: 训练集大小对训练误差, 测试误差以及最佳容量的影响。通过给一个 5 阶多项式添加适当大小的噪声, 我们构造了一个合成的回归问题, 生成单个测试集, 然后生成一些不同尺寸的训练集。为了描述 95% 置信区间的误差条, 对于每一个尺寸, 我们生成了 40 个不同的训练集。(上) 两个不同的模型上训练集和测试集的 MSE, 一个二次模型, 另一个模型的阶数通过最小化测试误差来选择。两个模型都是用闭式解来拟合。对于二次模型来说, 当训练集增加时训练误差也随之增大。这是由于越大的数据集越难以拟合。同时, 测试误差随之减小, 因为关于训练数据的不正确的假设越来越少。二次模型的容量并不足以解决这个问题, 所以它的测试误差趋近于一个较高的值。最佳容量点处的测试误差趋近于贝叶斯误差。训练误差可以低于贝叶斯误差, 因为训练算法有能力记住训练集中特定的样本。当训练集趋向于无穷大时, 任何固定容量的模型(在这里指的是二次模型)的训练误差都至少增至贝叶斯误差。(下) 当训练集大小增大时, 最佳容量(在这里是用最优多项式回归器的阶数衡量的)也会随之增大。最佳容量在达到足够捕捉模型复杂度之后就不再增长了。

### 5.2.2 正则化

没有免费午餐定理暗示我们必须在特定任务上设计性能良好的机器学习算法。我们建立一组学习算法的偏好来达到这个要求。当这些偏好和我们希望算法解决的学习问题相吻合时，性能会更好。

至此，我们具体讨论修改学习算法的方法只有，通过增加或减少学习算法可选假设空间的函数来增加或减少模型的表示容量。我们列举的一个具体示例是线性回归增加或减少多项式的次数。目前为止讨论的观点都是过度简化的。

算法的效果不仅很大程度上受影响于假设空间的函数数量，也取决于这些函数的具体形式。我们已经讨论的学习算法（线性回归）具有包含其输入的线性函数集的假设空间。对于输入和输出确实接近线性相关的问题，这些线性函数是很有用的。对于完全非线性的问题它们不太有效。例如，我们用线性回归，从  $x$  预测  $\sin(x)$ ，效果不会好。因此我们可以通过两种方式控制算法的性能，一是允许使用的函数种类，二是这些函数的数量。

在假设空间中，相比于某一个学习算法，我们可能更偏好另一个学习算法。这意味着两个函数都是符合条件的，但是我们更偏好其中一个。只有非偏好函数比偏好函数在训练数据集上效果明显好很多时，我们才会考虑非偏好函数。

例如，我们可以加入 权重衰减 ( weight decay ) 来修改线性回归的训练标准。带权重衰减的线性回归最小化训练集上的均方误差和正则项的和  $J(\mathbf{w})$ ，其偏好于平方  $L^2$  范数较小的权重。具体如下：

$$J(\mathbf{w}) = \text{MSE}_{\text{train}} + \lambda \mathbf{w}^\top \mathbf{w}, \quad (5.18)$$

其中  $\lambda$  是提前挑选的值，控制我们偏好小范数权重的程度。当  $\lambda = 0$ ，我们没有任何偏好。越大的  $\lambda$  偏好范数越小的权重。最小化  $J(\mathbf{w})$  可以看作是拟合训练数据和偏好小权重范数之间的权衡。这会使得解决方案的斜率较小，或是将权重放在较少的特征上。我们可以训练具有不同  $\lambda$  值的高次多项式回归模型，来举例说明如何通过权重衰减控制模型欠拟合或过拟合的趋势。如图 5.5 所示。

更一般地，正则化一个学习函数  $f(\mathbf{x}; \boldsymbol{\theta})$  的模型，我们可以给代价函数添加被称为 正则化项 ( regularizer ) 的惩罚。在权重衰减的例子中，正则化项是  $\Omega(\mathbf{w}) = \mathbf{w}^\top \mathbf{w}$ 。在第七章，我们将看到很多其他可能的正则化项。

表示对函数的偏好是比增减假设空间的成员函数更一般的控制模型容量的方法。我们可以将去掉假设空间中的某个函数看作是对不赞成这个函数的无限偏好。

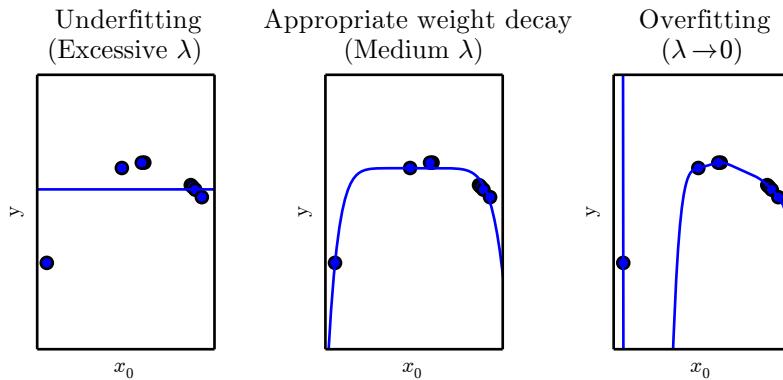


图 5.5: 我们使用高阶多项式回归模型来拟合图 5.2 中训练样本。真实函数是二次的，但是在这里我们只使用 9 阶多项式。我们通过改变权重衰减的量来避免高阶模型的过拟合问题。(左) 当  $\lambda$  非常大时，我们可以强迫模型学习到了一个没有斜率的函数。由于它只能表示一个常数函数，所以会导致欠拟合。(中) 取一个适当的  $\lambda$  时，学习算法能够用一个正常的形状来恢复曲率。即使模型能够用更复杂的形状来表示函数，权重衰减鼓励用一个带有更小参数的更简单的模型来描述它。(右) 当权重衰减趋近于 0 (即使用 Moore-Penrose 伪逆来解这个带有最小正则化的欠定问题) 时，这个 9 阶多项式会导致严重的过拟合，这和我们在图 5.2 中看到的一样。

在我们权重衰减的示例中，通过在最小化目标中额外增加一项，我们明确地表示了偏好权重较小的线性函数。有很多其他方法隐式或显式地表示对不同解的偏好。总而言之，这些不同的方法都被称为 **正则化** (regularization)。正则化是指我们修改学习算法，使其降低泛化误差而非训练误差。正则化是机器学习领域的中心问题之一，只有优化能够与其重要性相媲美。

没有免费午餐定理已经清楚地阐述了没有最优的学习算法，特别地，没有最优的正则化形式。反之，我们必须挑选一个非常适合于我们所要解决的任务的正则形式。深度学习中普遍的（特别是本书中的）理念是大量任务（例如所有人类能做的智能任务）也许都可以使用非常通用的正则化形式来有效解决。

## 5.3 超参数和验证集

大多数机器学习算法都有超参数，可以设置来控制算法行为。超参数的值不是通过学习算法本身学习出来的（尽管我们可以设计一个嵌套的学习过程，一个学习

算法为另一个学习算法学出最优超参数)。

在图 5.2 所示的多项式回归示例中，有一个超参数：多项式的次数，作为容量超参数。控制权重衰减程度的  $\lambda$  是另一个超参数。

有时一个选项被设为学习算法不用学习的超参数，是因为它太难优化了。更多的情况是，该选项必须是超参数，因为它不适合在训练集上学习。这适用于控制模型容量的所有超参数。如果在训练集上学习超参数，这些超参数总是趋向于最大可能的模型容量，导致过拟合(参考图 5.3)。例如，相比低次多项式和正的权重衰减设定，更高次的多项式和权重衰减参数设定  $\lambda = 0$  总能在训练集上更好地拟合。

为了解决这个问题，我们需要一个训练算法观测不到的验证集(validation set)样本。

早先我们讨论过和训练数据相同分布的样本组成的测试集，它可以用来估计学习过程完成之后的学习器的泛化误差。其重点在于测试样本不能以任何形式参与到模型的选择中，包括设定超参数。基于这个原因，测试集中的样本不能用于验证集。因此，我们总是从训练数据中构建验证集。特别地，我们将训练数据分成两个不相交的子集。其中一个用于学习参数。另一个作为验证集，用于估计训练中或训练后的泛化误差，更新超参数。用于学习参数的数据子集通常仍被称为训练集，尽管这会和整个训练过程用到的更大的数据集相混。用于挑选超参数的数据子集被称为验证集(validation set)。通常，80% 的训练数据用于训练，20% 用于验证。由于验证集是用来“训练”超参数的，尽管验证集的误差通常会比训练集误差小，验证集会低估泛化误差。所有超参数优化完成之后，泛化误差可能会通过测试集来估计。

在实际中，当相同的测试集已在很多年中重复地用于评估不同算法的性能，并且考虑学术界在该测试集上的各种尝试，我们最后可能也会对测试集有着乐观的估计。基准会因之变得陈旧，而不能反映系统的真实性能。值得庆幸的是，学术界往往会移到新的(通常会更巨大、更具挑战性)基准数据集上。

### 5.3.1 交叉验证

将数据集分成固定的训练集和固定的测试集后，若测试集的误差很小，这将是有问题的。一个小规模的测试集意味着平均测试误差估计的统计不确定性，使得很难判断算法 A 是否比算法 B 在给定的任务上做得更好。

当数据集有十万计或者更多的样本时，这不会是一个严重的问题。当数据集太

小时，也有替代方法允许我们使用所有的样本估计平均测试误差，代价是增加了计算量。这些过程是基于在原始数据上随机采样或分离出的不同数据集上重复训练和测试的想法。最常见的是  $k$ -折交叉验证过程，如算法 5.1 所示，将数据集分成  $k$  个不重合的子集。测试误差可以估计为  $k$  次计算后的平均测试误差。在第  $i$  次测试时，数据的第  $i$  个子集用于测试集，其他的数据用于训练集。带来的一个问题是不存在平均误差方差的无偏估计 (Bengio and Grandvalet, 2004)，但是我们通常会使用近似来解决。

---

**算法 5.1**  $k$ -折交叉验证算法。当给定数据集  $\mathbb{D}$  对于简单的训练/测试或训练/验证分割而言太小难以产生泛化误差的准确估计时（因为在小的测试集上， $L$  可能具有过高的方差）， $k$ -折交叉验证算法可以用于估计学习算法  $A$  的泛化误差。数据集  $\mathbb{D}$  包含的元素是抽象的样本  $\mathbf{z}^{(i)}$ （对于第  $i$  个样本），在监督学习的情况下代表（输入，目标）对  $\mathbf{z}^{(i)} = (\mathbf{x}^{(i)}, y^{(i)})$ ，或者无监督学习的情况下仅用于输入  $\mathbf{z}^{(i)} = \mathbf{x}^{(i)}$ 。该算法返回  $\mathbb{D}$  中每个示例的误差向量  $\mathbf{e}$ ，其均值是估计的泛化误差。单个样本上的误差可用于计算平均值周围的置信区间（式 (5.47)）。虽然这些置信区间在使用交叉验证之后不能很好地证明，但是通常的做法是只有当算法  $A$  误差的置信区间低于并且不与算法  $B$  的置信区间相交时，我们才声明算法  $A$  比算法  $B$  更好。

---

**Define** KFoldXV( $\mathbb{D}, A, L, k$ ):

**Require:**  $\mathbb{D}$  为给定数据集，其中元素为  $\mathbf{z}^{(i)}$

**Require:**  $A$  为学习算法，可视为一个函数（使用数据集作为输入，输出一个学好的函数）

**Require:**  $L$  为损失函数，可视为来自学好的函数  $f$ ，将样本  $\mathbf{z}^{(i)} \in \mathbb{D}$  映射到  $\mathbb{R}$  中标量的函数

**Require:**  $k$  为折数

将  $\mathbb{D}$  分为  $k$  个互斥子集  $\mathbb{D}_i$ ，它们的并集为  $\mathbb{D}$

**for**  $i$  from 1 to  $k$  **do**

$f_i = A(\mathbb{D} \setminus \mathbb{D}_i)$

**for**  $\mathbf{z}^{(j)}$  in  $\mathbb{D}_i$  **do**

$e_j = L(f_i, \mathbf{z}^{(j)})$

**end for**

**end for**

**Return**  $\mathbf{e}$

---

## 5.4 估计、偏差和方差

统计领域为我们提供了很多工具来实现机器学习目标，不仅可以解决训练集上的任务，还可以泛化。基本的概念，例如参数估计、偏差和方差，对于正式地刻画泛化、欠拟合和过拟合都非常有帮助。

### 5.4.1 点估计

点估计试图为一些感兴趣的量提供单个“最优”预测。一般地，感兴趣的量可以是单个参数，或是某些参数模型中的一个向量参数，例如第 5.1.4 节线性回归中的权重，但是也有可能是整个函数。

为了区分参数估计和真实值，我们习惯将参数  $\theta$  的点估计表示为  $\hat{\theta}$ 。

令  $\{x^{(1)}, \dots, x^{(m)}\}$  是  $m$  个独立同分布 (i.i.d.) 的数据点。点估计 (point estimator) 或 统计量 (statistics) 是这些数据的任意函数：

$$\hat{\theta}_m = g(x^{(1)}, \dots, x^{(m)}). \quad (5.19)$$

这个定义不要求  $g$  返回一个接近真实  $\theta$  的值，或者  $g$  的值域恰好是  $\theta$  的允许取值范围。点估计的定义非常宽泛，给了估计量的设计者极大的灵活性。虽然几乎所有的函数都可以称为估计量，但是一个好的估计量的输出会接近生成训练数据的真实参数  $\theta$ 。

现在，我们采取频率派在统计上的观点。换言之，我们假设真实参数  $\theta$  是固定但未知的，而点估计  $\hat{\theta}$  是数据的函数。由于数据是随机过程采样出来的，数据的任何函数都是随机的。因此  $\hat{\theta}$  是一个随机变量。

点估计也可以指输入和目标变量之间关系的估计。我们将这种类型的点估计称为函数估计。

**函数估计** 有时我们会关注函数估计（或函数近似）。这时我们试图从输入向量  $x$  预测变量  $y$ 。我们假设有一个函数  $f(x)$  表示  $y$  和  $x$  之间的近似关系。例如，我们可能假设  $y = f(x) + \epsilon$ ，其中  $\epsilon$  是  $y$  中未能从  $x$  预测的一部分。在函数估计中，我们感兴趣的是用模型估计去近似  $f$ ，或者估计  $\hat{f}$ 。函数估计和估计参数  $\theta$  是一样的；函数估计  $\hat{f}$  是函数空间中的一个点估计。线性回归示例（第 5.1.4 节中讨论的）和多项

式回归示例（第 5.2 节中讨论的）都既可以被解释为估计参数  $\mathbf{w}$ ，又可以被解释为估计从  $x$  到  $y$  的函数映射  $\hat{f}$ 。

现在我们回顾点估计最常研究的性质，并探讨这些性质说明了估计的哪些特点。

### 5.4.2 偏差

估计的偏差被定义为：

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta}, \quad (5.20)$$

其中期望作用在所有数据（看作是从随机变量采样得到的）上， $\boldsymbol{\theta}$  是用于定义数据生成分布的  $\boldsymbol{\theta}$  的真实值。如果  $\text{bias}(\hat{\boldsymbol{\theta}}_m) = 0$ ，那么估计量  $\hat{\boldsymbol{\theta}}_m$  被称为是无偏（unbiased），这意味着  $\mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$ 。如果  $\lim_{m \rightarrow \infty} \text{bias}(\hat{\boldsymbol{\theta}}_m) = 0$ ，那么估计量  $\hat{\boldsymbol{\theta}}_m$  被称为是渐近无偏（asymptotically unbiased），这意味着  $\lim_{m \rightarrow \infty} \mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$ 。

**示例：伯努利分布** 考虑一组服从均值为  $\theta$  的伯努利分布的独立同分布的样本  $\{x^{(1)}, \dots, x^{(m)}\}$ ：

$$P(x^{(i)}; \theta) = \theta^{x^{(i)}} (1 - \theta)^{(1-x^{(i)})}. \quad (5.21)$$

这个分布中参数  $\theta$  的常用估计量是训练样本的均值：

$$\hat{\theta}_m = \frac{1}{m} \sum_{i=1}^m x^{(i)}. \quad (5.22)$$

判断这个估计量是否有偏，我们将式 (5.22) 代入式 (5.20)：

$$\text{bias}(\hat{\theta}_m) = \mathbb{E}[\hat{\theta}_m] - \theta \quad (5.23)$$

$$= \mathbb{E}\left[\frac{1}{m} \sum_{i=1}^m x^{(i)}\right] - \theta \quad (5.24)$$

$$= \frac{1}{m} \sum_{i=1}^m \mathbb{E}[x^{(i)}] - \theta \quad (5.25)$$

$$= \frac{1}{m} \sum_{i=1}^m \sum_{x^{(i)}=0}^1 \left( x^{(i)} \theta^{x^{(i)}} (1 - \theta)^{(1-x^{(i)})} \right) - \theta \quad (5.26)$$

$$= \frac{1}{m} \sum_{i=1}^m (\theta) - \theta \quad (5.27)$$

$$= \theta - \theta = 0 \quad (5.28)$$

因为  $\text{bias}(\hat{\theta}) = 0$ , 我们称估计  $\hat{\theta}$  是无偏的。

**示例：均值的高斯分布估计** 现在, 考虑一组独立同分布的样本  $\{x^{(1)}, \dots, x^{(m)}\}$  服从高斯分布  $p(x^{(i)}) = \mathcal{N}(x^{(i)}; \mu, \sigma^2)$ , 其中  $i \in \{1, \dots, m\}$ 。回顾高斯概率密度函数如下:

$$p(x^{(i)}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2} \frac{(x^{(i)} - \mu)^2}{\sigma^2}\right). \quad (5.29)$$

高斯均值参数的常用估计量被称为 **样本均值** (sample mean):

$$\hat{\mu}_m = \frac{1}{m} \sum_{i=1}^m x^{(i)} \quad (5.30)$$

判断样本均值是否有偏, 我们再次计算它的期望:

$$\text{bias}(\hat{\mu}_m) = \mathbb{E}[\hat{\mu}_m] - \mu \quad (5.31)$$

$$= \mathbb{E}\left[\frac{1}{m} \sum_{i=1}^m x^{(i)}\right] - \mu \quad (5.32)$$

$$= \left(\frac{1}{m} \sum_{i=1}^m \mathbb{E}[x^{(i)}]\right) - \mu \quad (5.33)$$

$$= \left(\frac{1}{m} \sum_{i=1}^m \mu\right) - \mu \quad (5.34)$$

$$= \mu - \mu = 0 \quad (5.35)$$

因此我们发现样本均值是高斯均值参数的无偏估计量。

**示例：高斯分布方差估计** 本例中, 我们比较高斯分布方差参数  $\sigma^2$  的两个不同估计。我们探讨是否有一个是有偏的。

我们考虑的第一个方差估计被称为 **样本方差** (sample variance):

$$\hat{\sigma}_m^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \hat{\mu}_m)^2, \quad (5.36)$$

其中  $\hat{\mu}_m$  是样本均值。更形式地, 我们对计算感兴趣

$$\text{bias}(\hat{\sigma}_m^2) = \mathbb{E}[\hat{\sigma}_m^2] - \sigma^2. \quad (5.37)$$

我们首先估计项  $\mathbb{E}[\hat{\sigma}_m^2]$ :

$$\mathbb{E}[\hat{\sigma}_m^2] = \mathbb{E}\left[\frac{1}{m} \sum_{i=1}^m (x^{(i)} - \hat{\mu}_m)^2\right] \quad (5.38)$$

$$= \frac{m-1}{m} \sigma^2 \quad (5.39)$$

回到式 (5.37)，我们可以得出  $\hat{\sigma}_m^2$  的偏差是  $-\sigma^2/m$ 。因此样本方差是有偏估计。

**无偏样本方差** (unbiased sample variance) 估计

$$\tilde{\sigma}_m^2 = \frac{1}{m-1} \sum_{i=1}^m (x^{(i)} - \hat{\mu}_m)^2 \quad (5.40)$$

提供了另一种可选方法。正如名字所言，这个估计是无偏的。换言之，我们会发现  $\mathbb{E}[\tilde{\sigma}_m^2] = \sigma^2$ ：

$$\mathbb{E}[\tilde{\sigma}_m^2] = \mathbb{E}\left[\frac{1}{m-1} \sum_{i=1}^m (x^{(i)} - \hat{\mu}_m)^2\right] \quad (5.41)$$

$$= \frac{m}{m-1} \mathbb{E}[\hat{\sigma}_m^2] \quad (5.42)$$

$$= \frac{m}{m-1} \left(\frac{m-1}{m} \sigma^2\right) \quad (5.43)$$

$$= \sigma^2. \quad (5.44)$$

我们有两个估计量：一个是有偏的，另一个是无偏的。尽管无偏估计显然是令人满意的，但它并不总是“最好”的估计。我们将看到，经常会使用其他具有重要性质的有偏估计。

### 5.4.3 方差和标准差

我们有时会考虑估计量的另一个性质是它作为数据样本的函数，期望的变化程度是多少。正如我们可以计算估计量的期望来决定它的偏差，我们也可以计算它的方差。估计量的**方差** (variance) 就是一个方差

$$\text{Var}(\hat{\theta}) \quad (5.45)$$

其中随机变量是训练集。另外，方差的平方根被称为**标准差** (standard error)，记作  $\text{SE}(\hat{\theta})$ 。

估计量的方差或标准差告诉我们，当独立地从潜在的数据生成过程中重采样数据集时，如何期望估计的变化。正如我们希望估计的偏差较小，我们也希望其方差较小。

当我们使用有限的样本计算任何统计量时，真实参数的估计都是不确定的，在这个意义上，从相同的分布得到其他样本时，它们的统计量也会不一样。任何方差估计量的期望程度是我们想量化的误差的来源。

均值的标准差被记作

$$\text{SE}(\hat{\mu}_m) = \sqrt{\text{Var}\left[\frac{1}{m} \sum_{i=1}^m x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}, \quad (5.46)$$

其中  $\sigma^2$  是样本  $x^{(i)}$  的真实方差。标准差通常被记作  $\sigma$ 。可惜，样本方差的平方根和方差无偏估计的平方根都不是标准差的无偏估计。这两种计算方法都倾向于低估真实的标准差，但仍用于实际中。相较而言，方差无偏估计的平方根较少被低估。对于较大的  $m$ ，这种近似非常合理。

均值的标准差在机器学习实验中非常有用。我们通常用测试集样本的误差均值来估计泛化误差。测试集中样本的数量决定了这个估计的精确度。中心极限定理告诉我们均值会接近一个高斯分布，我们可以用标准差计算出真实期望落在选定区间的概率。例如，以均值  $\hat{\mu}_m$  为中心的 95% 置信区间是

$$(\hat{\mu}_m - 1.96\text{SE}(\hat{\mu}_m), \hat{\mu}_m + 1.96\text{SE}(\hat{\mu}_m)), \quad (5.47)$$

以上区间是基于均值  $\hat{\mu}_m$  和方差  $\text{SE}(\hat{\mu}_m)^2$  的高斯分布。在机器学习实验中，我们通常说算法 A 比算法 B 好，是指算法 A 的误差的 95% 置信区间的上界小于算法 B 的误差的 95% 置信区间的下界。

**示例：伯努利分布** 我们再次考虑从伯努利分布（回顾  $P(x^{(i)}; \theta) = \theta^{x^{(i)}}(1-\theta)^{1-x^{(i)}}$ ）中独立同分布采样出来的一组样本  $\{x^{(1)}, \dots, x^{(m)}\}$ 。这次我们关注估计  $\hat{\theta}_m =$

$\frac{1}{m} \sum_{i=1}^m x^{(i)}$  的方差：

$$\text{Var}(\hat{\theta}_m) = \text{Var}\left(\frac{1}{m} \sum_{i=1}^m x^{(i)}\right) \quad (5.48)$$

$$= \frac{1}{m^2} \sum_{i=1}^m \text{Var}(x^{(i)}) \quad (5.49)$$

$$= \frac{1}{m^2} \sum_{i=1}^m \theta(1 - \theta) \quad (5.50)$$

$$= \frac{1}{m^2} m\theta(1 - \theta) \quad (5.51)$$

$$= \frac{1}{m} \theta(1 - \theta) \quad (5.52)$$

估计量方差的下降速率是关于数据集样本数目  $m$  的函数。这是常见估计量的普遍性质，在探讨一致性（参考第 5.4.5 节）时，我们会继续讨论。

#### 5.4.4 权衡偏差和方差以最小化均方误差

偏差和方差度量着估计量的两个不同误差来源。偏差度量着偏离真实函数或参数的误差期望。而方差度量着数据上任意特定采样可能导致的估计期望的偏差。

当我们可以在一个偏差更大的估计和一个方差更大的估计中进行选择时，会发生什么呢？我们该如何选择？例如，想象我们希望近似图 5.2 中的函数，我们只能选择一个偏差较大的估计或一个方差较大的估计，我们该如何选择呢？

判断这种权衡最常用的方法是交叉验证。经验上，交叉验证在真实世界的许多任务中都非常成功。另外，我们也可以比较这些估计的 **均方误差**（mean squared error, MSE）：

$$\text{MSE} = \mathbb{E}[(\hat{\theta}_m - \theta)^2] \quad (5.53)$$

$$= \text{Bias}(\hat{\theta}_m)^2 + \text{Var}(\hat{\theta}_m) \quad (5.54)$$

MSE 度量着估计和真实参数  $\theta$  之间平方误差的总体期望偏差。如式 (5.54) 所示，MSE 估计包含了偏差和方差。理想的估计具有较小的 MSE 或是在检查中会稍微约束它们的偏差和方差。

偏差和方差的关系和机器学习容量、欠拟合和过拟合的概念紧密相联。用 MSE 度量泛化误差（偏差和方差对于泛化误差都是有意义的）时，增加容量会增加方差，降

低偏差。如图 5.6 所示，我们再次在关于容量的函数中，看到泛化误差的 U 形曲线。

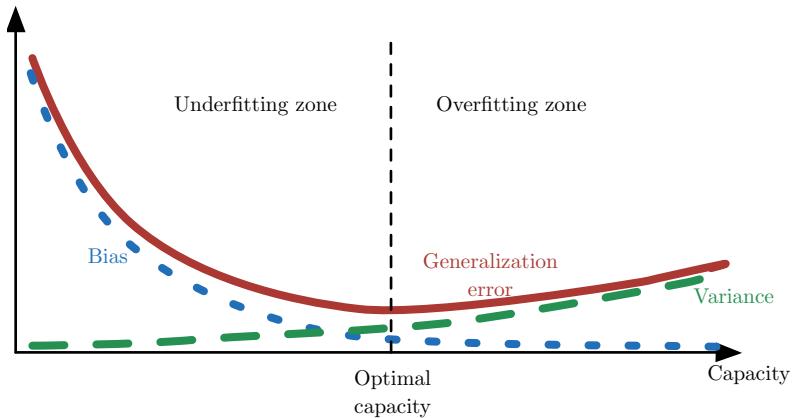


图 5.6：当容量增大（ $x$  轴）时，偏差（用点表示）随之减小，而方差（虚线）随之增大，使得泛化误差（加粗曲线）产生了另一种 U 形。如果我们沿着轴改变容量，会发现最佳容量，当容量小于最佳容量会呈现欠拟合，大于时导致过拟合。这种关系与第 5.2 节以及图 5.3 中讨论的容量、欠拟合和过拟合之间的关系类似。

### 5.4.5 一致性

目前我们已经探讨了固定大小训练集下不同估计量的性质。通常，我们也会关注训练数据增多后估计量的效果。特别地，我们希望当数据集中数据点的数量  $m$  增加时，点估计会收敛到对应参数的真实值。更形式地，我们想要

$$\text{plim}_{m \rightarrow \infty} \hat{\theta}_m = \theta. \quad (5.55)$$

符号  $\text{plim}$  表示依概率收敛，即对于任意的  $\epsilon > 0$ ，当  $m \rightarrow \infty$  时，有  $P(|\hat{\theta}_m - \theta| > \epsilon) \rightarrow 0$ 。式 (5.55) 表示的条件被称为 **一致性** (consistency)。有时它是指弱一致性，强一致性是指几乎必然 (almost sure) 从  $\hat{\theta}$  收敛到  $\theta$ 。几乎必然收敛 (almost sure convergence) 是指当  $p(\lim_{m \rightarrow \infty} \mathbf{x}^{(m)} = \mathbf{x}) = 1$  时，随机变量序列  $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots$  收敛到  $\mathbf{x}_0$ 。

一致性保证了估计量的偏差会随数据样本数目的增多而减少。然而，反过来是不正确的——渐近无偏并不意味着一致性。例如，考虑用包含  $m$  个样本的数据集  $\{x^{(1)}, \dots, x^{(m)}\}$  估计正态分布  $\mathcal{N}(x; \mu, \sigma^2)$  的均值参数  $\mu$ 。我们可以使用数据集的第

一个样本  $x^{(1)}$  作为无偏估计量:  $\hat{\theta} = x^{(1)}$ 。在该情况下,  $\mathbb{E}(\hat{\theta}_m) = \theta$ , 所以不管观测到多少数据点, 该估计量都是无偏的。然而, 这不是一个一致估计, 因为它不满足当  $m \rightarrow \infty$  时,  $\hat{\theta}_m \rightarrow \theta$ 。

## 5.5 最大似然估计

之前, 我们已经看过常用估计的定义, 并分析了它们的性质。但是这些估计是从哪里来的呢? 我们希望有些准则可以让我们从不同模型中得到特定函数作为好的估计, 而不是猜测某些函数可能是好的估计, 然后分析其偏差和方差。

最常用的准则是最大似然估计。

考虑一组含有  $m$  个样本的数据集  $\mathbb{X} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ , 独立地由未知的真实数据生成分布  $p_{\text{data}}(\mathbf{x})$  生成。

令  $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$  是一族由  $\boldsymbol{\theta}$  确定在相同空间上的概率分布。换言之,  $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$  将任意输入  $\mathbf{x}$  映射到实数来估计真实概率  $p_{\text{data}}(\mathbf{x})$ 。

对  $\boldsymbol{\theta}$  的最大似然估计被定义为:

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}), \quad (5.56)$$

$$= \arg \max_{\boldsymbol{\theta}} \prod_{i=1}^m p_{\text{model}}(\mathbf{x}^{(i)}; \boldsymbol{\theta}). \quad (5.57)$$

多个概率的乘积会因很多原因不便于计算。例如, 计算中很可能会出现数值下溢。为了得到一个便于计算的等价优化问题, 我们观察到似然对数不会改变其  $\arg \max$  但是将乘积转化成了便于计算的求和形式:

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^m \log p_{\text{model}}(\mathbf{x}^{(i)}; \boldsymbol{\theta}). \quad (5.58)$$

因为当我们重新缩放代价函数时  $\arg \max$  不会改变, 我们可以除以  $m$  得到和训练数据经验分布  $\hat{p}_{\text{data}}$  相关的期望作为准则:

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta}). \quad (5.59)$$

一种解释最大似然估计的观点是将它看作最小化训练集上的经验分布  $\hat{p}_{\text{data}}$  和模

型分布之间的差异，两者之间的差异程度可以通过 KL 散度度量。KL 散度被定义为

$$D_{\text{KL}}(\hat{p}_{\text{data}} \| p_{\text{model}}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} [\log \hat{p}_{\text{data}}(\mathbf{x}) - \log p_{\text{model}}(\mathbf{x})]. \quad (5.60)$$

左边一项仅涉及到数据生成过程，和模型无关。这意味着当我们训练模型最小化 KL 散度时，我们只需要最小化

$$-\mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} [\log p_{\text{model}}(\mathbf{x})], \quad (5.61)$$

当然，这和式 (5.59) 中最大化是相同的。

最小化 KL 散度其实就是在最小化分布之间的交叉熵。许多作者使用术语“交叉熵”特定表示伯努利或 softmax 分布的负对数似然，但那是用词不当的。任何一个由负对数似然组成的损失都是定义在训练集上的经验分布和定义在模型上的概率分布之间的交叉熵。例如，均方误差是经验分布和高斯模型之间的交叉熵。

我们可以将最大似然看作是使模型分布尽可能地和经验分布  $\hat{p}_{\text{data}}$  相匹配的尝试。理想情况下，我们希望匹配真实的数据生成分布  $p_{\text{data}}$ ，但我们没法直接知道这个分布。

虽然最优  $\theta$  在最大化似然或是最小化 KL 散度时是相同的，但目标函数值是不一样的。在软件中，我们通常将两者都称为最小化代价函数。因此最大化似然变成了最小化负对数似然 (NLL)，或者等价的是最小化交叉熵。将最大化似然看作最小化 KL 散度的视角在这个情况下是有帮助的，因为已知 KL 散度最小值是零。当  $x$  取实数时，负对数似然是负值。

### 5.5.1 条件对数似然和均方误差

最大似然估计很容易扩展到估计条件概率  $P(\mathbf{y} | \mathbf{x}; \theta)$ ，从而给定  $\mathbf{x}$  预测  $\mathbf{y}$ 。实际上这是最常见的情况，因为这构成了大多数监督学习的基础。如果  $\mathbf{X}$  表示所有的输入， $\mathbf{Y}$  表示我们观测到的目标，那么条件最大似然估计是

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\theta} P(\mathbf{Y} | \mathbf{X}; \theta). \quad (5.62)$$

如果假设样本是独立同分布的，那么这可以分解成

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\theta} \sum_{i=1}^m \log P(\mathbf{y}^{(i)} | \mathbf{x}^{(i)}; \theta). \quad (5.63)$$

**示例：线性回归作为最大似然** 第 5.1.4 节介绍的线性回归，可以被看作是最大似然过程。之前，我们将线性回归作为学习从输入  $\mathbf{x}$  映射到输出  $\hat{y}$  的算法。从  $\mathbf{x}$  到  $\hat{y}$  的映射选自最小化均方误差（我们或多或少介绍的一个标准）。现在，我们以最大似然估计的角度重新审视线性回归。我们现在希望模型能够得到条件概率  $p(y | \mathbf{x})$ ，而不仅仅是得到一个单独的预测  $\hat{y}$ 。想象有一个无限大的训练集，我们可能会观测到几个训练样本有相同的输入  $\mathbf{x}$  但是不同的  $y$ 。现在学习算法的目标是拟合分布  $p(y | \mathbf{x})$  到和  $\mathbf{x}$  相匹配的不同的  $y$ 。为了得到我们之前推导出的相同的线性回归算法，我们定义  $p(y | \mathbf{x}) = \mathcal{N}(y; \hat{y}(\mathbf{x}; \mathbf{w}), \sigma^2)$ 。函数  $\hat{y}(\mathbf{x}; \mathbf{w})$  预测高斯的均值。在这个例子中，我们假设方差是用户固定的某个常量  $\sigma^2$ 。这种函数形式  $p(y | \mathbf{x})$  会使得最大似然估计得出和之前相同的学习算法。由于假设样本是独立同分布的，条件对数似然（式 (5.63)）如下

$$\sum_{i=1}^m \log p(y^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta}) \quad (5.64)$$

$$= -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^m \frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2}, \quad (5.65)$$

其中  $\hat{y}^{(i)}$  是线性回归在第  $i$  个输入  $\mathbf{x}^{(i)}$  上的输出， $m$  是训练样本的数目。对比均方误差和对数似然，

$$\text{MSE}_{\text{train}} = \frac{1}{m} \sum_{i=1}^m \|\hat{y}^{(i)} - y^{(i)}\|^2, \quad (5.66)$$

我们立刻可以看出最大化关于  $\mathbf{w}$  的对数似然和最小化均方误差会得到相同的参数估计  $\mathbf{w}$ 。但是对于相同的最优  $\mathbf{w}$ ，这两个准则有着不同的值。这验证了 MSE 可以用于最大似然估计。正如我们将看到的，最大似然估计有几个理想的性质。

### 5.5.2 最大似然的性质

最大似然估计最吸引人的地方在于，它被证明当样本数目  $m \rightarrow \infty$  时，就收敛率而言是最好的渐近估计。

在合适的条件下，最大似然估计具有一致性（参考第 5.4.5 节），意味着训练样本数目趋向于无穷大时，参数的最大似然估计会收敛到参数的真实值。这些条件是：

- 真实分布  $p_{\text{data}}$  必须在模型族  $p_{\text{model}}(\cdot; \boldsymbol{\theta})$  中。否则，没有估计可以还原  $p_{\text{data}}$ 。

- 真实分布  $p_{\text{data}}$  必须刚好对应一个  $\theta$  值。否则，最大似然估计恢复出真实分布  $p_{\text{data}}$  后，也不能决定数据生成过程使用哪个  $\theta$ 。

除了最大似然估计，还有其他的归纳准则，其中许多共享一致估计的性质。然而，一致估计的 **统计效率** (statistic efficiency) 可能区别很大。某些一致估计可能会在固定数目的样本上获得一个较低的泛化误差，或者等价地，可能只需要较少的样本就能达到一个固定程度的泛化误差。

统计效率通常用于**有参情况** (parametric case) 的研究中 (例如线性回归)。有参情况下我们的目标是估计参数值 (假设有可能确定真实参数)，而不是函数值。一种度量我们和真实参数相差多少的方法是计算均方误差的期望，即计算  $m$  个从数据生成分布中出来的训练样本上的估计参数和真实参数之间差值的平方。有参均方误差估计随着  $m$  的增加而减少，当  $m$  较大时，Cramér-Rao 下界 (Rao, 1945; Cramér, 1946) 表明不存在均方误差低于最大似然估计的一致估计。

因为这些原因 (一致性和统计效率)，最大似然通常是机器学习中的首选估计。当样本数目小到会发生过拟合时，正则化策略如权重衰减可用于获得训练数据有限时方差较小的最大似然有偏版本。

## 5.6 贝叶斯统计

至此我们已经讨论了**频率派统计** (frequentist statistics) 方法和基于估计单一值  $\theta$  的方法，然后基于该估计作所有的预测。另一种方法是在做预测时会考虑所有可能的  $\theta$ 。后者属于**贝叶斯统计** (Bayesian statistics) 的范畴。

正如第 5.4.1 节中讨论的，频率派的视角是真实参数  $\theta$  是未知的定值，而点估计  $\hat{\theta}$  是考虑数据集上函数 (可以看作是随机的) 的随机变量。

贝叶斯统计的视角完全不同。贝叶斯用概率反映知识状态的确定性程度。数据集能够被直接观测到，因此不是随机的。另一方面，真实参数  $\theta$  是未知或不确定的，因此可以表示成随机变量。

在观察到数据前，我们将  $\theta$  的已知知识表示成**先验概率分布** (prior probability distribution)， $p(\theta)$  (有时简单地称为“先验”)。一般而言，机器学习实践者会选择一个相当宽泛的 (即，高熵的) 先验分布，反映在观测到任何数据前参数  $\theta$  的高度不确定性。例如，我们可能会假设先验  $\theta$  在有限区间中均匀分布。许多先验偏好于

“更简单”的解（如小幅度的系数，或是接近常数的函数）。

现在假设我们有一组数据样本  $\{x^{(1)}, \dots, x^{(m)}\}$ 。通过贝叶斯规则结合数据似然  $p(x^{(1)}, \dots, x^{(m)} | \boldsymbol{\theta})$  和先验，我们可以恢复数据对我们关于  $\boldsymbol{\theta}$  信念的影响：

$$p(\boldsymbol{\theta} | x^{(1)}, \dots, x^{(m)}) = \frac{p(x^{(1)}, \dots, x^{(m)} | \boldsymbol{\theta})p(\boldsymbol{\theta})}{p(x^{(1)}, \dots, x^{(m)})} \quad (5.67)$$

在贝叶斯估计常用的情景下，先验开始是相对均匀的分布或高熵的高斯分布，观测数据通常会使后验的熵下降，并集中在参数的几个可能性很高的值。

相对于最大似然估计，贝叶斯估计有两个重要区别。第一，不像最大似然方法预测时使用  $\boldsymbol{\theta}$  的点估计，贝叶斯方法使用  $\boldsymbol{\theta}$  的全分布。例如，在观测到  $m$  个样本后，下一个数据样本  $x^{(m+1)}$  的预测分布如下：

$$p(x^{(m+1)} | x^{(1)}, \dots, x^{(m)}) = \int p(x^{(m+1)} | \boldsymbol{\theta})p(\boldsymbol{\theta} | x^{(1)}, \dots, x^{(m)}) d\boldsymbol{\theta}. \quad (5.68)$$

这里，每个具有正概率密度的  $\boldsymbol{\theta}$  的值有助于下一个样本的预测，其中贡献由后验密度本身加权。在观测到数据集  $\{x^{(1)}, \dots, x^{(m)}\}$  之后，如果我们仍然非常不确定  $\boldsymbol{\theta}$  的值，那么这个不确定性会直接包含在我们所做的任何预测中。

在第 5.4 节中，我们已经探讨频率派方法解决给定点估计  $\boldsymbol{\theta}$  的不确定性的方法是评估方差，估计的方差评估了观测数据重新从观测数据中采样后，估计可能如何变化。对于如何处理估计不确定性的这个问题，贝叶斯派的答案是积分，这往往会防止过拟合。当然，积分仅仅是概率法则的应用，使贝叶斯方法容易验证，而频率派机器学习基于相当特别的决定构建了一个估计，将数据集里的所有信息归纳到一个单独的点估计。

贝叶斯方法和最大似然方法的第二个最大区别是由贝叶斯先验分布造成的。先验能够影响概率质量密度朝参数空间中偏好先验的区域偏移。实践中，先验通常表现为偏好更简单或更光滑的模型。对贝叶斯方法的批判认为先验是人为主观判断影响预测的来源。

当训练数据很有限时，贝叶斯方法通常泛化得更好，但是当训练样本数目很大时，通常会有很大的计算代价。

**示例：贝叶斯线性回归** 我们使用贝叶斯估计方法学习线性回归的参数。在线性回归中，我们学习从输入向量  $\mathbf{x} \in \mathbb{R}^n$  预测标量  $y \in \mathbb{R}$  的线性映射。该预测由向量

$\mathbf{w} \in \mathbb{R}^n$  参数化：

$$\hat{y} = \mathbf{w}^\top \mathbf{x}. \quad (5.69)$$

给定一组  $m$  个训练样本  $(\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$ , 我们可以表示整个训练集对  $y$  的预测：

$$\hat{\mathbf{y}}^{(\text{train})} = \mathbf{X}^{(\text{train})} \mathbf{w}. \quad (5.70)$$

表示为  $\mathbf{y}^{(\text{train})}$  上的高斯条件分布, 我们得到

$$p(\mathbf{y}^{(\text{train})} | \mathbf{X}^{(\text{train})}, \mathbf{w}) = \mathcal{N}(\mathbf{y}^{(\text{train})}; \mathbf{X}^{(\text{train})} \mathbf{w}, \mathbf{I}) \quad (5.71)$$

$$\propto \exp\left(-\frac{1}{2}(\mathbf{y}^{(\text{train})} - \mathbf{X}^{(\text{train})} \mathbf{w})^\top (\mathbf{y}^{(\text{train})} - \mathbf{X}^{(\text{train})} \mathbf{w})\right), \quad (5.72)$$

其中, 我们根据标准的 MSE 公式假设  $y$  上的高斯方差为 1。在下文中, 为减少符号负担, 我们将  $(\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$  简单表示为  $(\mathbf{X}, \mathbf{y})$ 。

为确定模型参数向量  $\mathbf{w}$  的后验分布, 我们首先需要指定一个先验分布。先验应该反映我们对这些参数取值的信念。虽然有时将我们的先验信念表示为模型的参数很难或很不自然, 但在实践中我们通常假设一个相当广泛的分布来表示  $\theta$  的高度不确定性。实数值参数通常使用高斯作为先验分布:

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}; \boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0) \propto \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Lambda}_0^{-1}(\mathbf{w} - \boldsymbol{\mu}_0)\right), \quad (5.73)$$

其中,  $\boldsymbol{\mu}_0$  和  $\boldsymbol{\Lambda}_0$  分别是先验分布的均值向量和协方差矩阵。<sup>1</sup>

确定好先验后, 我们现在可以继续确定模型参数的后验分布。

$$p(\mathbf{w} | \mathbf{X}, \mathbf{y}) \propto p(\mathbf{y} | \mathbf{X}, \mathbf{w}) p(\mathbf{w}) \quad (5.74)$$

$$\propto \exp\left(-\frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w})\right) \exp\left(-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Lambda}_0^{-1}(\mathbf{w} - \boldsymbol{\mu}_0)\right) \quad (5.75)$$

$$\propto \exp\left(-\frac{1}{2}(-2\mathbf{y}^\top \mathbf{X}\mathbf{w} + \mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} + \mathbf{w}^\top \boldsymbol{\Lambda}_0^{-1}\mathbf{w} - 2\boldsymbol{\mu}_0^\top \boldsymbol{\Lambda}_0^{-1}\mathbf{w})\right). \quad (5.76)$$

---

<sup>1</sup>除非有理由使用协方差矩阵的特定结构, 我们通常假设其为对角协方差矩阵  $\boldsymbol{\Lambda}_0 = \text{diag}(\boldsymbol{\lambda}_0)$ 。

现在我们定义  $\Lambda_m = (\mathbf{X}^\top \mathbf{X} + \Lambda_0^{-1})^{-1}$  和  $\mu_m = \Lambda_m(\mathbf{X}^\top \mathbf{y} + \Lambda_0^{-1}\mu_0)$ 。使用这些新的变量，我们发现后验可改写为高斯分布：

$$p(\mathbf{w} | \mathbf{X}, \mathbf{y}) \propto \exp\left(-\frac{1}{2}(\mathbf{w} - \mu_m)^\top \Lambda_m^{-1}(\mathbf{w} - \mu_m) + \frac{1}{2}\mu_m^\top \Lambda_m^{-1}\mu_m\right) \quad (5.77)$$

$$\propto \exp\left(-\frac{1}{2}(\mathbf{w} - \mu_m)^\top \Lambda_m^{-1}(\mathbf{w} - \mu_m)\right). \quad (5.78)$$

分布的积分必须归一这个事实意味着要删去所有不包括参数向量  $\mathbf{w}$  的项。式(3.23)显示了如何标准化多元高斯分布。

检查此后验分布可以让我们获得贝叶斯推断效果的一些直觉。大多数情况下，我们设置  $\mu_0 = 0$ 。如果我们设置  $\Lambda_0 = \frac{1}{\alpha} \mathbf{I}$ ，那么  $\mu_m$  对  $\mathbf{w}$  的估计就和频率派带权重衰减惩罚  $\alpha \mathbf{w}^\top \mathbf{w}$  的线性回归的估计是一样的。一个区别是若  $\alpha$  设为 0 则贝叶斯估计是未定义的——我们不能将贝叶斯学习过程初始化为一个无限宽的  $\mathbf{w}$  先验。更重要的区别是贝叶斯估计会给出一个协方差矩阵，表示  $\mathbf{w}$  所有不同值的可能范围，而不仅是估计  $\mu_m$ 。

### 5.6.1 最大后验 (MAP) 估计

原则上，我们应该使用参数  $\theta$  的完整贝叶斯后验分布进行预测，但单点估计常常也是需要的。希望使用点估计的一个常见原因是，对于大多数有意义的模型而言，大多数涉及到贝叶斯后验的计算是非常棘手的，点估计提供了一个可行的近似解。我们仍然可以让先验影响点估计的选择来利用贝叶斯方法的优点，而不是简单地回到最大似然估计。一种能够做到这一点的合理方式是选择最大后验 (Maximum A Posteriori, MAP) 点估计。MAP 估计选择后验概率最大的点（或在  $\theta$  是连续值的更常见情况下，概率密度最大的点）：

$$\theta_{\text{MAP}} = \arg \max_{\theta} p(\theta | \mathbf{x}) = \arg \max_{\theta} \log p(\mathbf{x} | \theta) + \log p(\theta). \quad (5.79)$$

我们可以认出上式右边的  $\log p(\mathbf{x} | \theta)$  对应着标准的对数似然项， $\log p(\theta)$  对应着先验分布。

例如，考虑具有高斯先验权重  $\mathbf{w}$  的线性回归模型。如果先验是  $\mathcal{N}(\mathbf{w}; \mathbf{0}, \frac{1}{\lambda} I^2)$ ，那么式(5.79)的对数先验项正比于熟悉的权重衰减惩罚  $\lambda \mathbf{w}^\top \mathbf{w}$ ，加上一个不依赖于  $\mathbf{w}$  也不会影响学习过程的项。因此，具有高斯先验权重的MAP 贝叶斯推断对应着权重衰减。

正如全贝叶斯推断，MAP 贝叶斯推断的优势是能够利用来自先验的信息，这些信息无法从训练数据中获得。该附加信息有助于减少最大后验点估计的方差（相比于 ML 估计）。然而，这个优点的代价是增加了偏差。

许多正规化估计方法，例如权重衰减正则化的最大似然学习，可以被解释为贝叶斯推断的 MAP 近似。这个适应于正则化时加到目标函数的附加项对应着  $\log p(\boldsymbol{\theta})$ 。并非所有的正则化惩罚都对应着 MAP 贝叶斯推断。例如，有些正则化项可能不是一个概率分布的对数。还有些正则化项依赖于数据，当然也不会是一个先验概率分布。

MAP 贝叶斯推断提供了一个直观的方法来设计复杂但可解释的正则化项。例如，更复杂的惩罚项可以通过混合高斯分布作为先验得到，而不是一个单独的高斯分布 (Nowlan and Hinton, 1992)。

## 5.7 监督学习算法

回顾第 5.1.3 节，粗略地说，监督学习算法是给定一组输入  $\mathbf{x}$  和输出  $\mathbf{y}$  的训练集，学习如何关联输入和输出。在许多情况下，输出  $\mathbf{y}$  很难自动收集，必须由人来提供“监督”，不过该术语仍然适用于训练集目标可以被自动收集的情况。

### 5.7.1 概率监督学习

本书的大部分监督学习算法都是基于估计概率分布  $p(y | \mathbf{x})$  的。我们可以使用最大似然估计找到对于有参分布族  $p(y | \mathbf{x}; \boldsymbol{\theta})$  最好的参数向量  $\boldsymbol{\theta}$ 。

我们已经看到，线性回归对应于分布族

$$p(y | \mathbf{x}; \boldsymbol{\theta}) = \mathcal{N}(y; \boldsymbol{\theta}^\top \mathbf{x}, \mathbf{I}). \quad (5.80)$$

通过定义一族不同的概率分布，我们可以将线性回归扩展到分类情况中。如果我们有两个类，类 0 和类 1，那么我们只需要指定这两类之一的概率。类 1 的概率决定了类 0 的概率，因为这两个值加起来必须等于 1。

我们用于线性回归的实数正态分布是用均值参数化的。我们提供这个均值的任何值都是有效的。二元变量上的分布稍微复杂些，因为它的均值必须始终在 0 和 1 之间。解决这个问题的一种方法是使用 logistic sigmoid 函数将线性函数的输出压缩

进区间  $(0, 1)$ 。该值可以解释为概率：

$$p(y = 1 \mid \mathbf{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^\top \mathbf{x}). \quad (5.81)$$

这个方法被称为 **逻辑回归** (logistic regression)，这个名字有点奇怪，因为该模型用于分类而非回归。

线性回归中，我们能够通过求解正规方程以找到最佳权重。相比而言，逻辑回归会更困难些。其最佳权重没有闭解。反之，我们必须最大化对数似然来搜索最优解。我们可以通过梯度下降算法最小化负对数似然来搜索。

通过确定正确的输入和输出变量上的有参条件概率分布族，相同的策略基本上可以用于任何监督学习问题。

## 5.7.2 支持向量机

**支持向量机** (support vector machine, SVM) 是监督学习中最有影响力的方法之一 (Boser *et al.*, 1992; Cortes and Vapnik, 1995)。类似于逻辑回归，这个模型也是基于线性函数  $\mathbf{w}^\top \mathbf{x} + b$  的。不同于逻辑回归的是，支持向量机不输出概率，只输出类别。当  $\mathbf{w}^\top \mathbf{x} + b$  为正时，支持向量机预测属于正类。类似地，当  $\mathbf{w}^\top \mathbf{x} + b$  为负时，支持向量机预测属于负类。

支持向量机的一个重要创新是 **核技巧** (kernel trick)。核技巧观察到许多机器学习算法都可以写成样本间点积的形式。例如，支持向量机中的线性函数可以重写为

$$\mathbf{w}^\top \mathbf{x} + b = b + \sum_{i=1}^m \alpha_i \mathbf{x}^\top \mathbf{x}^{(i)}, \quad (5.82)$$

其中， $\mathbf{x}^{(i)}$  是训练样本， $\boldsymbol{\alpha}$  是系数向量。学习算法重写为这种形式允许我们将  $\mathbf{x}$  替换为特征函数  $\phi(\mathbf{x})$  的输出，点积替换为被称为 **核函数** (kernel function) 的函数  $k(\mathbf{x}, \mathbf{x}^{(i)}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}^{(i)})$ 。运算符  $\cdot$  表示类似于  $\phi(\mathbf{x})^\top \phi(\mathbf{x}^{(i)})$  的点积。对于某些特征空间，我们可能不会书面地使用向量内积。在某些无限维空间中，我们需要使用其他类型的内积，如基于积分而非加和的内积。这种类型内积的完整介绍超出了本书的范围。

使用核估计替换点积之后，我们可以使用如下函数进行预测

$$f(\mathbf{x}) = b + \sum_i \alpha_i k(\mathbf{x}, \mathbf{x}^{(i)}). \quad (5.83)$$

这个函数关于  $\mathbf{x}$  是非线性的，关于  $\phi(\mathbf{x})$  是线性的。 $\alpha$  和  $f(\mathbf{x})$  之间的关系也是线性的。核函数完全等价于用  $\phi(\mathbf{x})$  预处理所有的输入，然后在新的转换空间学习线性模型。

核技巧十分强大有两个原因。首先，它使我们能够使用保证有效收敛的凸优化技术来学习非线性模型（关于  $\mathbf{x}$  的函数）。这是可能的，因为我们可以认为  $\phi$  是固定的，仅优化  $\alpha$ ，即优化算法可以将决策函数视为不同空间中的线性函数。其二，核函数  $k$  的实现方法通常有比直接构建  $\phi(\mathbf{x})$  再算点积高效很多。

在某些情况下， $\phi(\mathbf{x})$  甚至可以是无限维的，对于普通的显式方法而言，这将是无限的计算代价。在很多情况下，即使  $\phi(\mathbf{x})$  是难算的， $k(\mathbf{x}, \mathbf{x}')$  却会是一个关于  $\mathbf{x}$  非线性的、易算的函数。举个无限维空间易算的核的例子，我们构建一个作用于非负整数  $x$  上的特征映射  $\phi(x)$ 。假设这个映射返回一个由开头  $x$  个 1，随后是无限个 0 的向量。我们可以写一个核函数  $k(x, x^{(i)}) = \min(x, x^{(i)})$ ，完全等价于对应的无限维点积。

最常用的核函数是 **高斯核** (Gaussian kernel)，

$$k(\mathbf{u}, \mathbf{v}) = \mathcal{N}(\mathbf{u} - \mathbf{v}; \mathbf{0}, \sigma^2 I), \quad (5.84)$$

其中  $\mathcal{N}(x; \boldsymbol{\mu}, \boldsymbol{\Sigma})$  是标准正态密度。这个核也被称为 **径向基函数** (radial basis function, RBF) 核，因为其值沿  $\mathbf{v}$  中从  $\mathbf{u}$  向外辐射的方向减小。高斯核对应于无限维空间中的点积，但是该空间的推导没有整数上最小核的示例那么直观。

我们可以认为高斯核在执行一种**模板匹配** (template matching)。训练标签  $y$  相关的训练样本  $\mathbf{x}$  变成了类别  $y$  的模版。当测试点  $\mathbf{x}'$  到  $\mathbf{x}$  的欧几里得距离很小，对应的高斯核响应很大时，表明  $\mathbf{x}'$  和模版  $\mathbf{x}$  非常相似。该模型进而会赋予相对应的训练标签  $y$  较大的权重。总的来说，预测将会组合很多这种通过训练样本相似度加权的训练标签。

支持向量机不是唯一可以使用核技巧来增强的算法。许多其他的线性模型也可以通过这种方式来增强。使用核技巧的算法类别被称为 **核机器** (kernel machine) 或 **核方法** (kernel method) (Williams and Rasmussen, 1996; Schölkopf *et al.*, 1999)。

核机器的一个主要缺点是计算决策函数的成本关于训练样本的数目是线性的。因为第  $i$  个样本贡献  $\alpha_i k(\mathbf{x}, \mathbf{x}^{(i)})$  到决策函数。支持向量机能够通过学习主要包含零的向量  $\alpha$ ，以缓和这个缺点。那么判断新样本的类别仅需要计算非零  $\alpha_i$  对应的训练样本的核函数。这些训练样本被称为 **支持向量** (support vector)。

当数据集很大时，核机器的计算量也会很大。我们将会在第 5.9 节回顾这个想法。带通用核的核机器致力于泛化得更好。我们将在第 5.11 节解释原因。现代深度学习的设计旨在克服核机器的这些限制。当前深度学习的复兴始于 Hinton *et al.* (2006b) 表明神经网络能够在 MNIST 基准数据上胜过 RBF 核的支持向量机。

### 5.7.3 其他简单的监督学习算法

我们已经简要介绍过另一个非概率监督学习算法，最近邻回归。更一般地， $k$ -最近邻是一类可用于分类或回归的技术。作为一个非参数学习算法， $k$ -最近邻并不局限于固定数目的参数。我们通常认为  $k$ -最近邻算法没有任何参数，而是使用训练数据的简单函数。事实上，它甚至也没有一个真正的训练阶段或学习过程。反之，在测试阶段我们希望在新的测试输入  $\mathbf{x}$  上产生  $y$ ，我们需要在训练数据  $\mathbf{X}$  上找到  $\mathbf{x}$  的  $k$ -最近邻。然后我们返回训练集上对应的  $y$  值的平均值。这几乎适用于任何类型可以确定  $y$  值平均值的监督学习。在分类情况中，我们可以关于 one-hot 编码向量  $\mathbf{c}$  求平均，其中  $c_y = 1$ ，其他的  $i$  值取  $c_i = 0$ 。然后，我们可以解释这些 one-hot 编码的均值为类别的概率分布。作为一个非参数学习算法， $k$ -近邻能达到非常高的容量。例如，假设我们有一个用 0-1 误差度量性能的多分类任务。在此设定中，当训练样本数目趋向于无穷大时，1-最近邻收敛到两倍贝叶斯误差。超出贝叶斯误差的原因是它会随机从等距离的临近点中随机挑一个。而存在无限的训练数据时，所有测试点  $\mathbf{x}$  周围距离为零的邻近点有无限多个。如果我们使用所有这些临近点投票的决策方式，而不是随机挑选一个，那么该过程将会收敛到贝叶斯错误率。 $k$ -最近邻的高容量使其在训练样本数目大时能够获取较高的精度。然而，它的计算成本很高，另外在训练集较小时泛化能力很差。 $k$ -最近邻的一个弱点是它不能学习出哪一个特征比其他更具识别力。例如，假设我们要处理一个的回归任务，其中  $\mathbf{x} \in \mathbb{R}^{100}$  是从各向同性的高斯分布中抽取的，但是只有一个变量  $x_1$  和结果相关。进一步假设该特征直接决定了输出，即在所有情况下  $y = x_1$ 。最近邻回归不能检测到这个简单模式。大多数点  $\mathbf{x}$  的最近邻将取决于  $x_2$  到  $x_{100}$  的大多数特征，而不是单独取决于特征  $x_1$ 。因此，小训练集上的输出将会非常随机。

决策树 (decision tree) 及其变种是另一类将输入空间分成不同的区域，每个区域有独立参数的算法 (Breiman *et al.*, 1984)。如图 5.7 所示，决策树的每个节点都与输入空间的一个区域相关联，并且内部节点继续将区域分成子节点下的子区域（通常使用坐标轴拆分区域）。空间由此细分成不重叠的区域，叶节点和输入区域之间形

成一一对应的关系。每个叶结点将其输入区域的每个点映射到相同的输出。决策树通常有特定的训练算法，超出了本书的范围。如果允许学习任意大小的决策树，那么它可以被视作非参数算法。然而实践中通常有大小限制，作为正则化项将其转变成有参模型。由于决策树通常使用坐标轴相关的拆分，并且每个子节点关联到常数输出，因此有时解决一些对于逻辑回归很简单的问题很费力。例如，假设有一个二分类问题，当  $x_2 > x_1$  时分为正类，则决策树的分界不是坐标轴对齐的。因此，决策树将需要许多节点近似决策边界，坐标轴对齐使其算法步骤不断地来回穿梭于真正的决策函数。

正如我们已经看到的，最近邻预测和决策树都有很多的局限性。尽管如此，在计算资源受限制时，它们都是很有用的学习算法。通过思考复杂算法和  $k$ -最近邻或决策树之间的相似性和差异，我们可以建立对更复杂学习算法的直觉。

读者可以参考 Murphy (2012); Bishop (2006); Hastie *et al.* (2001) 或其他机器学习教科书了解更多的传统监督学习算法。

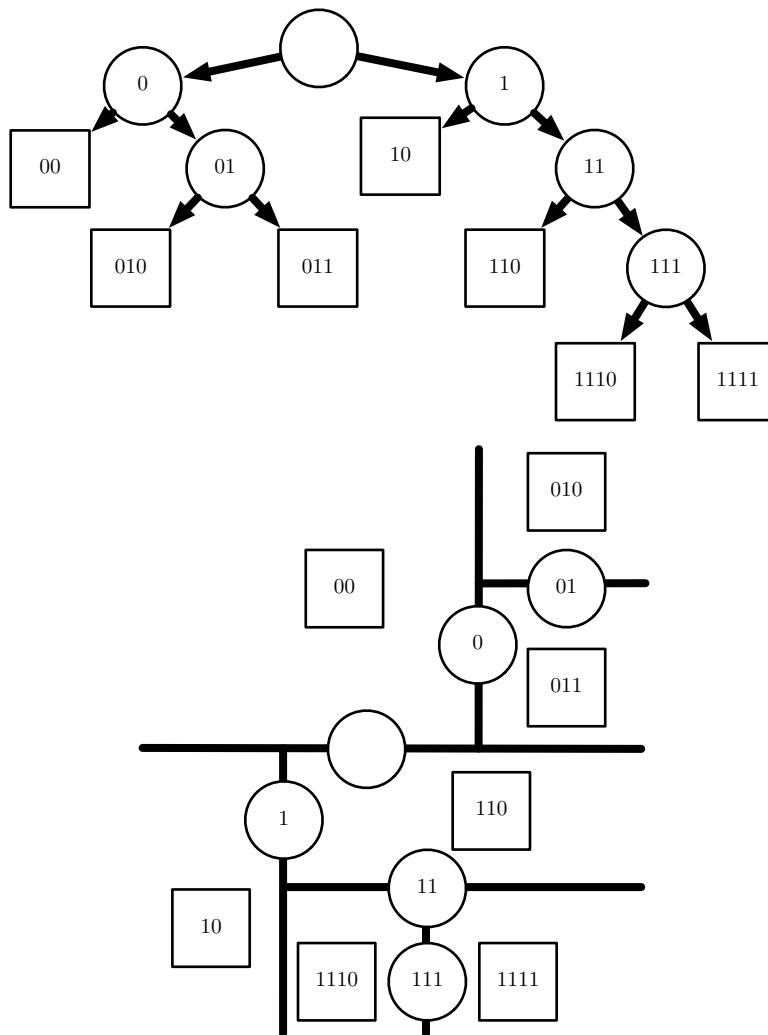


图 5.7: 描述一个决策树如何工作的示意图。(上) 树中每个节点都选择将输入样本送到左子节点 (0) 或者右子节点 (1)。内部的节点用圆圈表示, 叶节点用方块表示。每一个节点可以用一个二值的字符串识别并对应树中的位置, 这个字符串是通过给起父节点的字符串添加一个位元来实现的 (0 表示选择左或者上, 1 表示选择右或者下)。(下) 这个树将空间分为区域。这个二维平面说明决策树可以分割  $\mathbb{R}^2$ 。这个平面中画出了树的节点, 每个内部点穿过分割线并用来给样本分类, 叶节点画在样本所属区域的中心。结果是一个分块常数函数, 每一个叶节点一个区域。每个叶需要至少一个训练样本来定义, 所以决策树不可能用来学习一个局部极大值比训练样本数量还多的函数。

## 5.8 无监督学习算法

回顾第 5.1.3 节，无监督算法只处理“特征”，不操作监督信号。监督和无监督算法之间的区别没有规范严格的定义，因为没有客观的判断来区分监督者提供的值是特征还是目标。通俗地说，无监督学习的大多数尝试是指从不需要人为注释的样本的分布中抽取信息。该术语通常与密度估计相关，学习从分布中采样、学习从分布中去噪、寻找数据分布的流形或是将数据中相关的样本聚类。

一个经典的无监督学习任务是找到数据的“最佳”表示。“最佳”可以是不同的表示，但是一般来说，是指该表示在比本身表示的信息更简单或更易访问而受到一些惩罚或限制的情况下，尽可能地保存关于  $x$  更多的信息。

有很多方式定义较简单的表示。最常见的三种包括低维表示、稀疏表示和独立表示。低维表示尝试将  $x$  中的信息尽可能压缩在一个较小的表示中。稀疏表示将数据集嵌入到输入项大多数为零的表示中 (Barlow, 1989; Olshausen and Field, 1996; Hinton and Ghahramani, 1997)。稀疏表示通常用于需要增加表示维数的情况，使得大部分为零的表示不会丢失很多信息。这会使得表示的整体结构倾向于将数据分布在表示空间的坐标轴上。独立表示试图分开数据分布中变化的来源，使得表示的维度是统计独立的。

当然这三个标准并非相互排斥的。低维表示通常会产生比原始的高维数据具有较少或较弱依赖关系的元素。这是因为减少表示大小的一种方式是找到并消除冗余。识别并去除更多的冗余使得降维算法在丢失更少信息的同时显现更大的压缩。

表示的概念是深度学习核心主题之一，因此也是本书的核心主题之一。本节会介绍表示学习算法中的一些简单示例。总的来说，这些示例算法会说明如何实施上面的三个标准。剩余的大部分章节会介绍额外的表示学习算法，它们以不同方式处理这三个标准或是引入其他标准。

### 5.8.1 主成分分析

在第 2.12 节中，我们看到 PCA 算法提供了一种压缩数据的方式。我们也可以将 PCA 视为学习数据表示的无监督学习算法。这种表示基于上述简单表示的两个标准。PCA 学习一种比原始输入维数更低的表示。它也学习了一种元素之间彼此没有线性相关的表示。这是学习表示中元素统计独立标准的第一步。要实现完全独立性，表示学习算法也必须去掉变量间的非线性关系。

如图 5.8 所示，PCA 将输入  $\mathbf{x}$  投影表示成  $\mathbf{z}$ ，学习数据的正交线性变换。在第 2.12 节中，我们看到了如何学习重建原始数据的最佳一维表示（就均方误差而言），这种表示其实对应着数据的第一个主要成分。因此，我们可以用 PCA 作为保留数据尽可能多信息的降维方法（再次就最小重构误差平方而言）。在下文中，我们将研究 PCA 表示如何使原始数据表示  $\mathbf{X}$  去相关的。

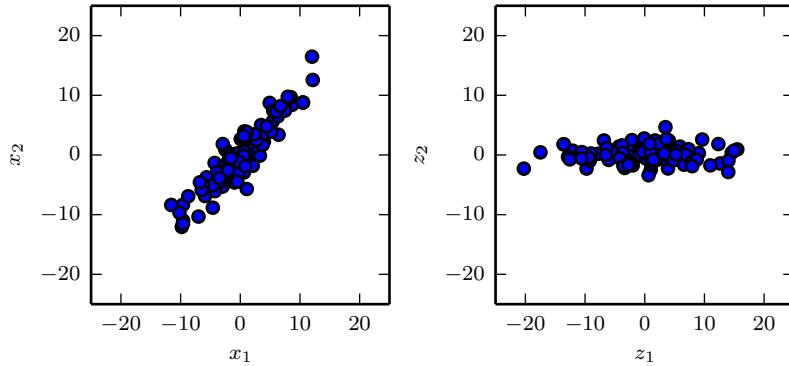


图 5.8：PCA 学习一种线性投影，使最大方差的方向和新空间的轴对齐。(左) 原始数据包含了  $\mathbf{x}$  的样本。在这个空间中，方差的方向与轴的方向并不是对齐的。(右) 变换过的数据  $\mathbf{z} = \mathbf{x}^\top \mathbf{W}$  在轴  $z_1$  的方向上有最大的变化。第二大变化方差的方向沿着轴  $z_2$ 。

假设有一个  $m \times n$  的设计矩阵  $\mathbf{X}$ ，数据的均值为零， $\mathbb{E}[\mathbf{x}] = 0$ 。若非如此，通过预处理步骤使所有样本减去均值，数据可以很容易地中心化。

$\mathbf{X}$  对应的无偏样本协方差矩阵给定如下

$$\text{Var}[\mathbf{x}] = \frac{1}{m-1} \mathbf{X}^\top \mathbf{X}. \quad (5.85)$$

PCA 通过线性变换找到一个  $\text{Var}[\mathbf{z}]$  是对角矩阵的表示  $\mathbf{z} = \mathbf{W}^\top \mathbf{x}$ 。

在第 2.12 节，我们已知设计矩阵  $\mathbf{X}$  的主成分由  $\mathbf{X}^\top \mathbf{X}$  的特征向量给定。从这个角度，我们有

$$\mathbf{X}^\top \mathbf{X} = \mathbf{W} \Lambda \mathbf{W}^\top. \quad (5.86)$$

本节中，我们会探索主成分的另一种推导。主成分也可以通过奇异值分解 (SVD) 得到。具体来说，它们是  $\mathbf{X}$  的右奇异向量。为了说明这点，假设  $\mathbf{W}$  是奇异值分解  $\mathbf{X} = \mathbf{U} \Sigma \mathbf{W}^\top$  的右奇异向量。以  $\mathbf{W}$  作为特征向量基，我们可以得到原来的特征向量

方程：

$$\mathbf{X}^\top \mathbf{X} = (\mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^\top)^\top \mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^\top = \mathbf{W}\boldsymbol{\Sigma}^2\mathbf{W}^\top. \quad (5.87)$$

SVD有助于说明PCA后的 $\text{Var}[\mathbf{z}]$ 是对角的。使用 $\mathbf{X}$ 的SVD分解， $\mathbf{X}$ 的方差可以表示为

$$\text{Var}[\mathbf{x}] = \frac{1}{m-1} \mathbf{X}^\top \mathbf{X} \quad (5.88)$$

$$= \frac{1}{m-1} (\mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^\top)^\top \mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^\top \quad (5.89)$$

$$= \frac{1}{m-1} \mathbf{W}\boldsymbol{\Sigma}^\top \mathbf{U}^\top \mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^\top \quad (5.90)$$

$$= \frac{1}{m-1} \mathbf{W}\boldsymbol{\Sigma}^2\mathbf{W}^\top, \quad (5.91)$$

其中，我们使用 $\mathbf{U}^\top \mathbf{U} = \mathbf{I}$ ，因为根据奇异值的定义矩阵 $\mathbf{U}$ 是正交的。这表明 $\mathbf{z}$ 的协方差满足对角的要求：

$$\text{Var}[\mathbf{z}] = \frac{1}{m-1} \mathbf{Z}^\top \mathbf{Z} \quad (5.92)$$

$$= \frac{1}{m-1} \mathbf{W}^\top \mathbf{X}^\top \mathbf{X} \mathbf{W} \quad (5.93)$$

$$= \frac{1}{m-1} \mathbf{W}^\top \mathbf{W}\boldsymbol{\Sigma}^2\mathbf{W}^\top \mathbf{W} \quad (5.94)$$

$$= \frac{1}{m-1} \boldsymbol{\Sigma}^2, \quad (5.95)$$

其中，再次使用SVD的定义有 $\mathbf{W}^\top \mathbf{W} = \mathbf{I}$ 。

以上分析指明当我们通过线性变换 $\mathbf{W}$ 将数据 $\mathbf{x}$ 投影到 $\mathbf{z}$ 时，得到的数据表示的协方差矩阵是对角的（即 $\boldsymbol{\Sigma}^2$ ），立刻可得 $\mathbf{z}$ 中的元素是彼此无关的。

PCA这种将数据变换为元素之间彼此不相关表示的能力是PCA的一个重要性质。它是消除数据中未知变化因素的简单表示示例。在PCA中，这个消除是通过寻找输入空间的一个旋转（由 $\mathbf{W}$ 确定），使得方差的主坐标和 $\mathbf{z}$ 相关的表示空间的基对齐。

虽然相关性是数据元素间依赖关系的一个重要范畴，但我们对于能够消除更复杂形式的特征依赖的表示学习也很感兴趣。对此，我们需要比简单线性变换更强的工具。

### 5.8.2 $k$ -均值聚类

另外一个简单的表示学习算法是  $k$ -均值聚类。 $k$ -均值聚类算法将训练集分成  $k$  个靠近彼此的不同样本聚类。因此我们可以认为该算法提供了  $k$ -维的 one-hot 编码向量  $\mathbf{h}$  以表示输入  $\mathbf{x}$ 。当  $\mathbf{x}$  属于聚类  $i$  时，有  $h_i = 1$ ， $\mathbf{h}$  的其他项为零。

$k$ -均值聚类提供的 one-hot 编码也是一种稀疏表示，因为每个输入的表示中大部分元素为零。之后，我们会介绍能够学习更灵活的稀疏表示的一些其他算法（表示中每个输入  $\mathbf{x}$  不只一个非零项）。one-hot 编码是稀疏表示的一个极端示例，丢失了很多分布式表示的优点。one-hot 编码仍然有一些统计优点（自然地传达了相同聚类中的样本彼此相似的观点），也具有计算上的优势，因为整个表示可以用一个单独的整数表示。

$k$ -均值聚类初始化  $k$  个不同的中心点  $\{\boldsymbol{\mu}^{(1)}, \dots, \boldsymbol{\mu}^{(k)}\}$ ，然后迭代交换两个不同的步骤直到收敛。步骤一，每个训练样本分配到最近的中心点  $\boldsymbol{\mu}^{(i)}$  所代表的聚类  $i$ 。步骤二，每一个中心点  $\boldsymbol{\mu}^{(i)}$  更新为聚类  $i$  中所有训练样本  $\mathbf{x}^{(j)}$  的均值。

关于聚类的一个问题是聚类问题本身是病态的。这是说没有单一的标准去度量聚类的数据在真实世界中效果如何。我们可以度量聚类的性质，例如类中元素到类中心点的欧几里得距离的均值。这使我们可以判断从聚类分配中重建训练数据的效果如何。然而我们不知道聚类的性质是否很好地对应到真实世界的性质。此外，可能有许多不同的聚类都能很好地对应到现实世界的某些属性。我们可能希望找到和一个特征相关的聚类，但是得到了一个和任务无关的，同样是合理的不同聚类。例如，假设我们在包含红色卡车图片、红色汽车图片、灰色卡车图片和灰色汽车图片的数据集上运行两个聚类算法。如果每个聚类算法聚两类，那么可能一个算法将汽车和卡车各聚一类，另一个根据红色和灰色各聚一类。假设我们还运行了第三个聚类算法，用来决定类别的数目。这有可能聚成了四类，红色卡车、红色汽车、灰色卡车和灰色汽车。现在这个新的聚类至少抓住了属性的信息，但是丢失了相似性信息。红色汽车和灰色汽车在不同的类中，正如红色汽车和灰色卡车也在不同的类中。该聚类算法没有告诉我们灰色汽车和红色汽车的相似度比灰色卡车和红色汽车的相似度更高。我们只知道它们是不同的。

这些问题说明了一些我们可能更偏好于分布式表示（相对于 one-hot 表示而言）的原因。分布式表示可以对每个车辆赋予两个属性——一个表示它颜色，一个表示它是汽车还是卡车。目前仍然不清楚什么是最优的分布式表示（学习算法如何知道我们关心的两个属性是颜色和是否汽车或卡车，而不是制造商和车龄？），但是多个

属性减少了算法去猜我们关心哪一个属性的负担，允许我们通过比较很多属性而非测试一个单一属性来细粒度地度量相似性。

## 5.9 随机梯度下降

几乎所有的深度学习算法都用到了一个非常重要的算法：随机梯度下降（stochastic gradient descent, SGD）。随机梯度下降是第4.3节介绍的梯度下降算法的一个扩展。

机器学习中反复出现的一个问题是好的泛化需要大的训练集，但大的训练集的计算代价也更大。

机器学习算法中的代价函数通常可以分解成每个样本的代价函数的总和。例如，训练数据的负条件对数似然可以写成

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim p_{\text{data}}} L(\mathbf{x}, y, \boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}), \quad (5.96)$$

其中  $L$  是每个样本的损失  $L(\mathbf{x}, y, \boldsymbol{\theta}) = -\log p(y | \mathbf{x}; \boldsymbol{\theta})$ 。

对于这些相加的代价函数，梯度下降需要计算

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}). \quad (5.97)$$

这个运算的计算代价是  $O(m)$ 。随着训练集规模增长为数十亿的样本，计算一步梯度也会消耗相当长的时间。

随机梯度下降的核心是，梯度是期望。期望可使用小规模的样本近似估计。具体而言，在算法的每一步，我们从训练集中均匀抽出一小批量（minibatch）样本  $\mathbb{B} = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m')}\}$ 。小批量的数目  $m'$  通常是一个相对较小的数，从一到几百。重要的是，当训练集大小  $m$  增长时， $m'$  通常是固定的。我们可能在拟合几十亿的样本时，每次更新计算只用到几百个样本。

梯度的估计可以表示成

$$\mathbf{g} = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}). \quad (5.98)$$

使用来自小批量  $\mathbb{B}$  的样本。然后，随机梯度下降算法使用如下的梯度下降估计：

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \mathbf{g}, \quad (5.99)$$

其中， $\epsilon$  是学习率。

梯度下降往往被认为很慢或不可靠。以前，将梯度下降应用到非凸优化问题被认为很鲁莽或没有原则。现在，我们知道梯度下降用于本书第二部分中的训练时效果不错。优化算法不一定能保证在合理的时间内达到一个局部最小值，但它通常能及时地找到代价函数一个很小的值，并且是有用的。

随机梯度下降在深度学习之外有很多重要的应用。它是在大规模数据上训练大型线性模型的主要方法。对于固定大小的模型，每一步随机梯度下降更新的计算量不取决于训练集的大小  $m$ 。在实践中，当训练集大小增长时，我们通常会使用一个更大的模型，但这并非是必须的。达到收敛所需的更新次数通常会随训练集规模增大而增加。然而，当  $m$  趋向于无穷大时，该模型最终会在随机梯度下降抽样完训练集上的所有样本之前收敛到可能的最优测试误差。继续增加  $m$  不会延长达到模型可能的最优测试误差的时间。从这点来看，我们可以认为用 SGD 训练模型的渐近代价是关于  $m$  的函数的  $O(1)$  级别。

在深度学习兴起之前，学习非线性模型的主要方法是结合核技巧的线性模型。很多核学习算法需要构建一个  $m \times m$  的矩阵  $G_{i,j} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$ 。构建这个矩阵的计算量是  $O(m^2)$ 。当数据集是几十亿个样本时，这个计算量是不能接受的。在学术界，深度学习从 2006 年开始受到关注的原因是，在数以万计样本的中等规模数据集上，深度学习在新样本上比当时很多热门算法泛化得更好。不久后，深度学习在工业界受到了更多的关注，因为其提供了一种训练大数据集上的非线性模型的可扩展方式。

我们将会在第八章继续探讨随机梯度下降及其很多改进方法。

## 5.10 构建机器学习算法

几乎所有的深度学习算法都可以被描述为一个相当简单的配方：特定的数据集、代价函数、优化过程和模型。

例如，线性回归算法由以下部分组成： $\mathbf{X}$  和  $\mathbf{y}$  构成的数据集，代价函数

$$J(\mathbf{w}, b) = -\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x}), \quad (5.100)$$

模型是  $p_{\text{model}}(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y}; \mathbf{x}^\top \mathbf{w} + b, 1)$ ，在大多数情况下，优化算法可以定义为求解代价函数梯度为零的正规方程。

意识到我们可以替换独立于其他组件的大多数组件，因此我们能得到很多不同

的算法。

通常代价函数至少含有一项使学习过程进行统计估计的成分。最常见的代价函数是负对数似然，最小化代价函数导致的最大似然估计。

代价函数也可能含有附加项，如正则化项。例如，我们可以将权重衰减加到线性回归的代价函数中

$$J(\mathbf{w}, b) = \lambda \|\mathbf{w}\|_2^2 - \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(y | \mathbf{x}). \quad (5.101)$$

该优化仍然有闭解。

如果我们将该模型变成非线性的，那么大多数代价函数不再能通过闭解优化。这就要求我们选择一个迭代数值优化过程，如梯度下降等。

组合模型、代价和优化算法来构建学习算法的配方同时适用于监督学习和无监督学习。线性回归示例说明了如何适用于监督学习的。无监督学习时，我们需要定义一个只包含  $\mathbf{X}$  的数据集、一个合适的无监督代价和一个模型。例如，通过指定如下损失函数可以得到 PCA 的第一个主向量

$$J(\mathbf{w}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \|\mathbf{x} - r(\mathbf{x}; \mathbf{w})\|_2^2 \quad (5.102)$$

模型定义为重构函数  $r(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} \mathbf{w}$ ，并且  $\mathbf{w}$  有范数为 1 的限制。

在某些情况下，由于计算原因，我们不能实际计算代价函数。在这种情况下，只要我们有近似其梯度的方法，那么我们仍然可以使用迭代数值优化近似最小化目标。

尽管有时候不显然，但大多数学习算法都用到了上述配方。如果一个机器学习算法看上去特别独特或是手动设计的，那么通常需要使用特殊的优化方法进行求解。有些模型，如决策树或  $k$ -均值，需要特殊的优化，因为它们的代价函数有平坦的区域，使其不适合通过基于梯度的优化去最小化。在我们认识到大部分机器学习算法可以使用上述配方描述之后，我们可以将不同算法视为出于相同原因解决相关问题的一类方法，而不是一长串各个不同的算法。

## 5.11 促使深度学习发展的挑战

本章描述的简单机器学习算法在很多不同的重要问题上效果都良好。但是它们不能成功解决人工智能中的核心问题，如语音识别或者对象识别。

深度学习发展动机的一部分原因是传统学习算法在这类人工智能问题上泛化能力不足。

本节介绍为何处理高维数据时在新样本上泛化特别困难，以及为何在传统机器学习中实现泛化的机制不适合学习高维空间中复杂的函数。这些空间经常涉及巨大的计算代价。深度学习旨在克服这些以及其他一些难题。

### 5.11.1 维数灾难

当数据的维数很高时，很多机器学习问题变得相当困难。这种现象被称为 **维数灾难** (curse of dimensionality)。特别值得注意的是，一组变量不同的可能配置数量会随着变量数目的增加而指数级增长。

维数灾难发生在计算机科学的许多地方，在机器学习中尤其如此。

由维数灾难带来的一个挑战是统计挑战。如图 5.9 所示，统计挑战产生于  $x$  的可能配置数目远大于训练样本的数目。为了充分理解这个问题，我们假设输入空间如图所示被分成单元格。空间是低维时，我们可以用由大部分数据占据的少量单元格去描述这个空间。泛化到新数据点时，通过检测和新输入点在相同单元格中的训练样本，我们可以判断如何处理新数据点。例如，如果要估计某点  $x$  处的概率密度，我们可以返回  $x$  处单位体积单元格内训练样本的数目除以训练样本的总数。如果我们希望对一个样本进行分类，我们可以返回相同单元格中训练样本最多的类别。如果我们是做回归分析，我们可以平均该单元格中样本对应的目标值。但是，如果该单元格中没有样本，该怎么办呢？因为在高维空间中参数配置数目远大于样本数目，大部分单元格中没有样本。我们如何能在这些新配置中找到一些有意义的东西呢？许多传统机器学习算法只是简单地假设在一个新点的输出应大致和最接近的训练点的输出相同。

### 5.11.2 局部不变性和平滑正则化

为了更好地泛化，机器学习算法需要由先验信念引导应该学习什么类型的函数。此前，我们已经看到过由模型参数的概率分布形成的先验。通俗地讲，我们也可以说明先验信念直接影响函数本身，而仅仅通过它们对函数的影响来间接改变参数。此外，我们还能通俗地说，先验信念还间接地体现在选择一些偏好某类函数的算法，尽管这些偏好并没有通过我们对不同函数置信程度的概率分布表现出来（也许根本没法

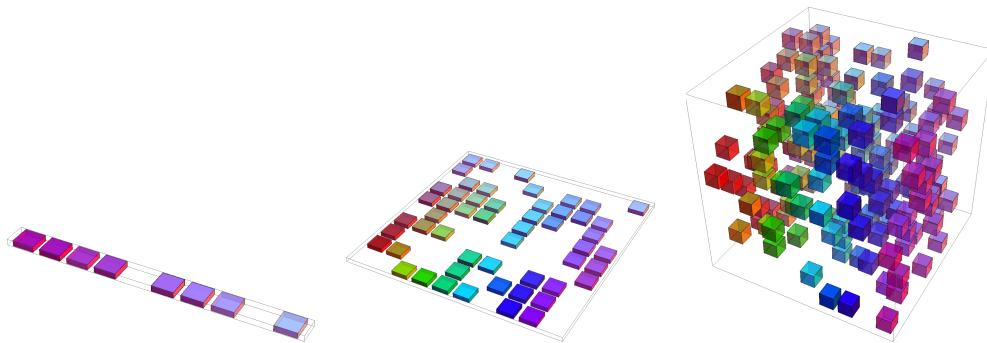


图 5.9: 当数据的相关维度增大时 (从左向右), 我们感兴趣的配置数目会随之指数级增长。(左) 在这个一维的例子中, 我们用一个变量来区分所感兴趣的 10 个区域。当每个区域都有足够的样本数时 (每个区域对应图中的一个单元格), 学习算法能够轻易地泛化得很好。泛化的一个直接方法是估计目标函数在每个区域的值 (可能是在相邻区域之间插值)。(中) 在二维情况下, 对每个变量区分 10 个不同的值更加困难。我们需要追踪  $10 \times 10 = 100$  个区域, 至少需要很多样本来覆盖所有的区域。(右) 三维情况下, 区域数量增加到了  $10^3 = 1000$ , 至少需要那么多的样本。对于需要区分的  $d$  维以及  $v$  个值来说, 我们需要  $O(v^d)$  个区域和样本。这就是维数灾难的一个示例。感谢由 Nicolas Chapados 提供的图片。

表现)。

其中最广泛使用的隐式“先验”是 **平滑先验** (smoothness prior), 或 **局部不变性先验** (local constancy prior)。这个先验表明我们学习的函数不应在小区域内发生很大的变化。

许多简单算法完全依赖于此先验达到良好的泛化, 其结果是不能推广去解决人工智能级别任务中的统计挑战。本书中, 我们将介绍深度学习如何引入额外的 (显式或隐式的) 先验去降低复杂任务中的泛化误差。这里, 我们解释为什么仅依靠平滑先验不足以应对这类任务。

有许多不同的方法来显式或隐式地表示学习函数应该具有光滑或局部不变的先验。所有这些不同的方法都旨在鼓励学习过程能够学习出函数  $f^*$ , 对于大多数设置  $\mathbf{x}$  和小变动  $\epsilon$ , 都满足条件

$$f^*(\mathbf{x}) \approx f^*(\mathbf{x} + \epsilon). \quad (5.103)$$

换言之, 如果我们知道对应输入  $\mathbf{x}$  的答案 (例如,  $\mathbf{x}$  是个有标签的训练样本), 那么该答案对于  $\mathbf{x}$  的邻域应该也适用。如果在有些邻域中我们有几个好答案, 那么我们可以组合它们 (通过某种形式的平均或插值法) 以产生一个尽可能和大多数输入一

致的答案。

局部不变方法的一个极端例子是  $k$ -最近邻系列的学习算法。当一个区域里的所有点  $x$  在训练集中的  $k$  个最近邻是一样的，那么对这些点的预测也是一样的。当  $k = 1$  时，不同区域的数目不会比训练样本还多。

虽然  $k$ -最近邻算法复制了附近训练样本的输出，大部分核机器也是在和附近训练样本相关的训练集输出上插值。一类重要的核函数是 **局部核** (local kernel)，其核函数  $k(u, v)$  在  $u = v$  时很大，当  $u$  和  $v$  距离拉大时而减小。局部核可以看作是执行模版匹配的相似函数，用于度量测试样本  $x$  和每个训练样本  $x^{(i)}$  有多么相似。近年来深度学习的很多推动力源自研究局部模版匹配的局限性，以及深度学习如何克服这些局限性 (Bengio *et al.*, 2006a)。

决策树也有平滑学习的局限性，因为它将输入空间分成和叶节点一样多的区间，并在每个区间使用单独的参数（或者有些决策树的拓展有多个参数）。如果目标函数需要至少拥有  $n$  个叶节点的树才能精确表示，那么至少需要  $n$  个训练样本去拟合。需要几倍于  $n$  的样本去达到预测输出上的某种统计置信度。

总的来说，区分输入空间中  $O(k)$  个区间，所有的这些方法需要  $O(k)$  个样本。通常会有  $O(k)$  个参数， $O(1)$  参数对应于  $O(k)$  区间之一。最近邻算法中，每个训练样本至多用于定义一个区间，如图 5.10 所示。

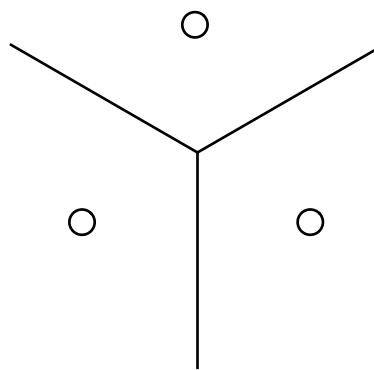


图 5.10：最近邻算法如何划分输入空间的示例。每个区域内的一个样本（这里用圆圈表示）定义了区域边界（这里用线表示）。每个样本相关的  $y$  值定义了对应区域内所有数据点的输出。由最近邻定义并且匹配几何模式的区域被称为 Voronoi 图。这些连续区域的数量不会比训练样本的数量增加得更快。尽管此图具体说明了最近邻算法的效果，其他的单纯依赖局部光滑先验的机器学习算法也表现出了类似的泛化能力：每个训练样本仅仅能告诉学习者如何在其周围的相邻区域泛化。

有没有什么方法能表示区间数目比训练样本数目还多的复杂函数？显然，只是假设函数的平滑性不能做到这点。例如，想象目标函数作用在西洋跳棋盘上。棋盘包含许多变化，但只有一个简单的结构。想象一下，如果训练样本数目远小于棋盘上的黑白方块数目，那么会发生什么。基于局部泛化和平滑性或局部不变性先验，如果新点和某个训练样本位于相同的棋盘方块中，那么我们能够保证正确地预测新点的颜色。但如果新点所在的方块没有训练样本，学习器不一定能举一反三。如果仅依靠这个先验，一个样本只能告诉我们它所在的方块的颜色。获得整个棋盘颜色的唯一方法是其上的每个方块至少要有一个样本。

只要在要学习的真实函数的峰值和谷值处有足够的样本，那么平滑性假设和相关的无参数学习算法的效果都非常好。当要学习的函数足够平滑，并且只在少数几维变化，这样做一般没问题。在高维空间中，即使是非常平滑的函数，也会在不同维度上有不同的变化方式。如果函数在不同的区间中表现不一样，那么就非常难以用一组训练样本去刻画函数。如果函数是复杂的（我们想区分多于训练样本数目的大量区间），有希望很好地泛化么？

这些问题，即是否可以有效地表示复杂的函数以及所估计的函数是否可以很好地泛化到新的输入，答案是有。关键观点是，只要我们通过额外假设生成数据的分布来建立区域间的依赖关系，那么  $O(k)$  个样本足以描述多如  $O(2^k)$  的大量区间。通过这种方式，我们确实能做到非局部的泛化 (Bengio and Monperrus, 2005; Bengio *et al.*, 2006b)。为了利用这些优势，许多不同的深度学习算法都提出了一些适用于多种 AI 任务的隐式或显式的假设。

一些其他的机器学习方法往往会提出更强的，针对特定问题的假设。例如，假设目标函数是周期性的，我们很容易解决棋盘问题。通常，神经网络不会包含这些很强的（针对特定任务的）假设，因此神经网络可以泛化到更广泛的各种结构中。人工智能任务的结构非常复杂，很难限制到简单的、人工手动指定的性质，如周期性，因此我们希望学习算法具有更通用的假设。深度学习的核心思想是假设数据由因素或特征组合产生，这些因素或特征可能来自一个层次结构的多个层级。许多其他类似的通用假设进一步提高了深度学习算法。这些很温和的假设允许了样本数目和可区分区间数目之间的指数增益。这类指数增益将在第 6.4.1 节、第 15.4 节和第 15.5 节中更详尽地介绍。深度的分布式表示带来的指数增益有效地解决了维数灾难带来的挑战。

### 5.11.3 流形学习

流形是一个机器学习中很多想法内心的重要概念。

流形 (manifold) 指连接在一起的区域。数学上，它是指一组点，且每个点都有其邻域。给定一个任意的点，其流形局部看起来像是欧几里得空间。日常生活中，我们将地球视为二维平面，但实际上它是三维空间中的球状流形。

每个点周围邻域的定义暗示着存在变换能够从一个位置移动到其邻域位置。例如在地球表面这个流形中，我们可以朝东南西北走。

尽管术语“流形”有正式的数学定义，但是机器学习倾向于更松散地定义一组点，只需要考虑少数嵌入在高维空间中的自由度或维数就能很好地近似。每一维都对应着局部的变化方向。如图 5.11 所示，训练数据位于二维空间中的一维流形中。在机器学习中，我们允许流形的维数从一个点到另一个点有所变化。这经常发生于流形和自身相交的情况下。例如，数字“8”形状的流形在大多数位置只有一维，但在中心的相交处有两维。

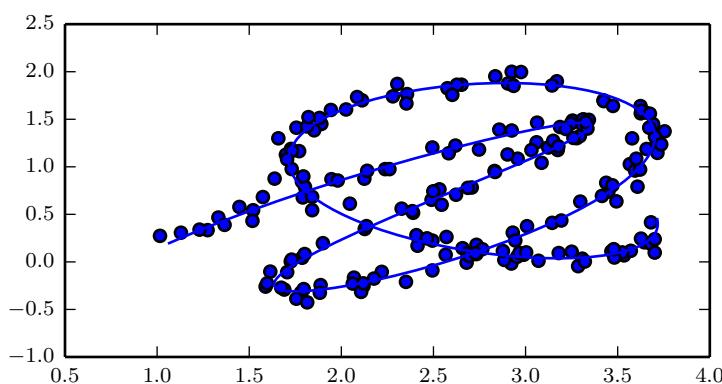


图 5.11：从一个二维空间的分布中抽取的数据样本，这些样本实际上聚集在一维流形附近，像一个缠绕的带子。实线代表学习器应该推断的隐式流形。

如果我们希望机器学习算法学习整个  $\mathbb{R}^n$  上有趣变化的函数，那么很多机器学习问题看上去都是无望的。流形学习 (manifold learning) 算法通过一个假设来克服这个障碍，该假设认为  $\mathbb{R}^n$  中大部分区域都是无效的输入，有意义的输入只分布在包含少量数据点的子集构成的一组流形中，而学习函数的输出中，有意义的变化都沿着流形的方向或仅发生在我们切换到另一流形时。流形学习最初用于连续数值和无

监督学习的环境，尽管这个概率集中的想法也能够泛化到离散数据和监督学习的设定下：关键假设仍然是概率质量高度集中。

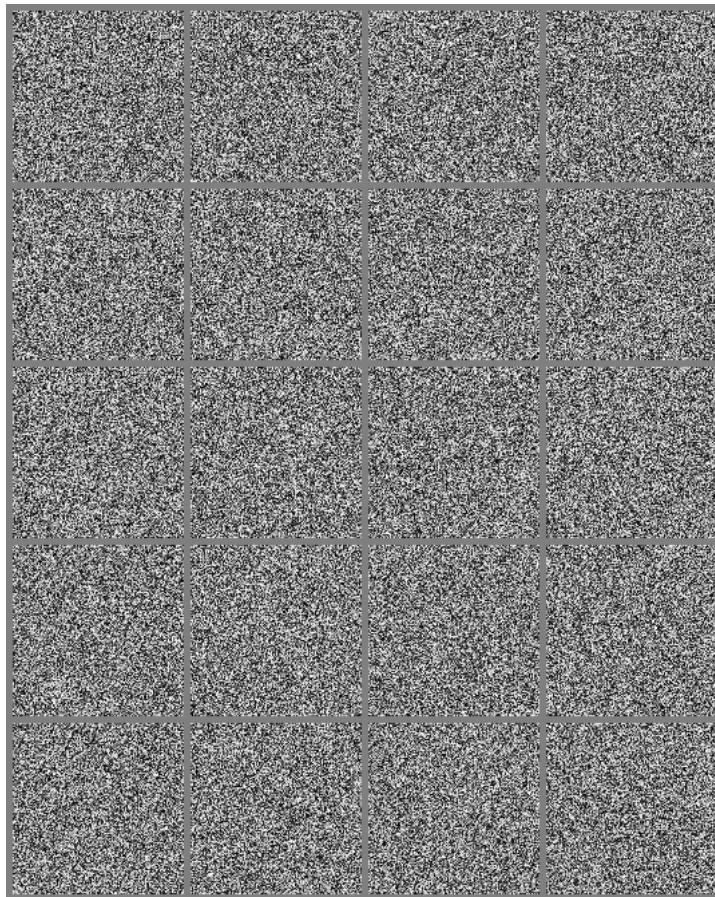


图 5.12: 随机地均匀抽取图像（根据均匀分布随机地选择每一个像素）会得到噪声图像。尽管在人工智能应用中以这种方式生成一个脸或者其他物体的图像是非零概率的，但是实际上我们从来没有观察到这种现象。这也意味着人工智能应用中遇到的图像在所有图像空间中的占比可以是忽略不计的。

数据位于低维流形的假设并不总是对的或者有用的。我们认为在人工智能的一些场景中，如涉及到处理图像、声音或者文本时，流形假设至少是近似对的。这个假设的支持证据包含两类观察结果。

第一个支持流形假设（manifold hypothesis）的观察是现实生活中的图像、文本、声音的概率分布都是高度集中的。均匀的噪声从来不会与这类领域的结构化输

入类似。图 5.12 显示均匀采样的点看上去像是没有信号时模拟电视上的静态模式。同样，如果我们均匀地随机抽取字母来生成文件，能有多大的概率得到一个有意义的英语文档？几乎是零。因为大部分字母长序列不对应着自然语言序列：自然语言序列的分布只占了字母序列的总空间里非常小的一部分。

当然，集中的概率分布不足以说明数据位于一个相当小的流形中。我们还必须确保，我们遇到的样本和其他样本相互连接，每个样本被其他高度相似的样本包围，而这些高度相似的样本可以通过变换来遍历该流形得到。支持流形假设的第二个论点是，我们至少能够非正式地想象这些邻域和变换。在图像中，我们当然会认为有很多可能的变换仍然允许我们描绘出图片空间的流形：我们可以逐渐变暗或变亮光泽、逐步移动或旋转图中对象、逐渐改变对象表面的颜色等等。在大多数应用中很有可能会涉及到多个流形。例如，人脸图像的流形不太可能连接到猫脸图像的流形。

这些支持流形假设的思维实验传递了一些支持它的直观理由。更严格的实验 (Cayton, 2005; Narayanan and Mitter, 2010; Schölkopf *et al.*, 1998a; Roweis and Saul, 2000; Tenenbaum *et al.*, 2000; Brand, 2003a; Belkin and Niyogi, 2003b; Donoho and Grimes, 2003; Weinberger and Saul, 2004a) 在人工智能中备受关注的一大类数据集上支持了这个假设。

当数据位于低维流形中时，使用流形中的坐标而非  $\mathbb{R}^n$  中的坐标表示机器学习数据更为自然。日常生活中，我们可以认为道路是嵌入在三维空间的一维流形。我们用一维道路中的地址号码确定地址，而非三维空间中的坐标。提取这些流形中的坐标是非常具有挑战性的，但是很有希望改进许多机器学习算法。这个一般性原则能够用在很多情况下。图 5.13 展示了包含人脸的数据集的流形结构。在本书的最后，我们会介绍一些学习这样的流形结构的必备方法。在图 20.6 中，我们将看到机器学习算法如何成功完成这个目标。

第一部分介绍了数学和机器学习中的基本概念，这将用于本书其他章节中。至此，我们已经做好了研究深度学习的准备。



图 5.13: QMUL Multiview Face 数据集中的训练样本 (Gong *et al.*, 2000), 其中的物体是移动的从而覆盖对应两个旋转角度的二维流形。我们希望学习算法能够发现并且理出这些流形坐标。图 20.6 提供了这样一个示例。

## 第二部分

### 深度网络：现代实践

本书这一部分总结现代深度学习用于解决实际应用的现状。

深度学习有着悠久的历史和许多愿景。数种提出的方法尚未完全结出果实。数个雄心勃勃的目标尚未实现。这些较不发达的深度学习分支将出现在本书的最后部分。

这一部分仅关注那些基本上已在工业中大量使用的技术方法。

现代深度学习为监督学习提供了一个强大的框架。通过添加更多层以及向层内添加更多单元，深度网络可以表示复杂性不断增加的函数。给定足够大的模型和足够大的标注训练数据集，我们可以通过深度学习将输入向量映射到输出向量，完成大多数对人来说能迅速处理的任务。其他任务，比如不能被描述为将一个向量与另一个相关联的任务，或者对于一个人来说足够困难并需要时间思考和反复琢磨才能完成的任务，现在仍然超出了深度学习的能力范围。

本书这一部分描述参数化函数近似技术的核心，几乎所有现代实际应用的深度学习背后都用到了这一技术。首先，我们描述用于表示这些函数的前馈深度网络模型。接着，我们提出正则化和优化这种模型的高级技术。将这些模型扩展到大输入（如高分辨率图像或长时间序列）需要专门化。我们将会介绍扩展到大图像的卷积网络和用于处理时间序列的循环神经网络。最后，我们提出实用方法的一般准则，有助于设计、构建和配置一些涉及深度学习的应用，并回顾其中一些应用。

这些章节对于从业者来说是最重要的，也就是现在想开始实现和使用深度学习算法解决现实问题的人需要阅读这些章节。

# 第六章 深度前馈网络

深度前馈网络 (deep feedforward network), 也叫作 前馈神经网络 (feedforward neural network) 或者 多层感知机 (multilayer perceptron, MLP), 是典型的深度学习模型。前馈网络的目标是近似某个函数  $f^*$ 。例如, 对于分类器,  $y = f^*(\mathbf{x})$  将输入  $\mathbf{x}$  映射到一个类别  $y$ 。前馈网络定义了一个映射  $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$ , 并且学习参数  $\boldsymbol{\theta}$  的值, 使它能够得到最佳的函数近似。

这种模型被称为 前向 (feedforward) 的, 是因为信息流过  $\mathbf{x}$  的函数, 流经用于定义  $f$  的中间计算过程, 最终到达输出  $\mathbf{y}$ 。在模型的输出和模型本身之间没有 反馈 (feedback) 连接。当前馈神经网络被扩展成包含反馈连接时, 它们被称为 循环神经网络 (recurrent neural network), 在第十章介绍。

前馈网络对于机器学习的从业者是极其重要的。它们是许多重要商业应用的基础。例如, 用于对照片中的对象进行识别的卷积神经网络就是一种专门的前馈网络。前馈网络是通往循环网络之路的概念基石, 后者在自然语言的许多应用中发挥着巨大作用。

前馈神经网络被称作 网络 (network) 是因为它们通常用许多不同函数复合在一起表示。该模型与一个有向无环图相关联, 而图描述了函数是如何复合在一起的。例如, 我们有三个函数  $f^{(1)}$ ,  $f^{(2)}$  和  $f^{(3)}$  连接在一个链上以形成  $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$ 。这些链式结构是神经网络中最常用的结构。在这种情况下,  $f^{(1)}$  被称为网络的第一层 (first layer),  $f^{(2)}$  被称为 第二层 (second layer), 以此类推。链的全长称为模型的 深度 (depth)。正是因为这个术语才出现了“深度学习”这个名字。前馈网络的最后一层被称为 输出层 (output layer)。在神经网络训练的过程中, 我们让  $f(\mathbf{x})$  去匹配  $f^*(\mathbf{x})$  的值。训练数据为我们提供了在不同训练点上取值的、含有噪声的  $f^*(\mathbf{x})$  的近似实例。每个样本  $\mathbf{x}$  都伴随着一个标签  $y \approx f^*(\mathbf{x})$ 。训练样本直接指明了输出层在每一点  $\mathbf{x}$  上必须做什么; 它必须产生一个接近  $y$  的值。

但是训练数据并没有直接指明其他层应该怎么做。学习算法必须决定如何使用这些层来产生想要的输出，但是训练数据并没有说每个单独的层应该做什么。相反，学习算法必须决定如何使用这些层来最好地实现  $f^*$  的近似。因为训练数据并没有给出这些层中的每一层所需的输出，所以这些层被称为 **隐藏层** (hidden layer)。

最后，这些网络被称为神经网络是因为它们或多或少地受到神经科学的启发。网络中的每个隐藏层通常都是向量值的。这些隐藏层的维数决定了模型的 **宽度** (width)。向量的每个元素都可以被视为起到类似一个神经元的作用。除了将层想象成向量到向量的单个函数，我们也可以把层想象成由许多并行操作的 **单元** (unit) 组成，每个单元表示一个向量到标量的函数。每个单元在某种意义上类似一个神经元，它接收的输入来源于许多其他的单元，并计算它自己的激活值。使用多层向量值表示的想法来源于神经科学。用于计算这些表示的函数  $f^{(i)}(\mathbf{x})$  的选择，也或多或少地受到神经科学观测的指引，这些观测是关于生物神经元计算功能的。然而，现代的神经网络研究受到更多的是来自许多数学和工程学科的指引，并且神经网络的目标并不是完美地给大脑建模。我们最好将前馈神经网络想成是为了实现统计泛化而设计出的函数近似机，它偶尔从我们了解的大脑中提取灵感，但并不是大脑功能的模型。

一种理解前馈网络的方式是从线性模型开始，并考虑如何克服它的局限性。线性模型，例如逻辑回归和线性回归，是非常吸引人的，因为无论是通过闭解形式还是使用凸优化，它们都能高效且可靠地拟合。线性模型也有明显的缺陷，那就是该模型的能力被局限在线性函数里，所以它无法理解任何两个输入变量间的相互作用。

为了扩展线性模型来表示  $\mathbf{x}$  的非线性函数，我们可以不把线性模型用于  $\mathbf{x}$  本身，而是用在一个变换后的输入  $\phi(\mathbf{x})$  上，这里  $\phi$  是一个非线性变换。同样，我们可以使用第 5.7.2 节中描述的核技巧，来得到一个基于隐含地使用  $\phi$  映射的非线性学习算法。我们可以认为  $\phi$  提供了一组描述  $\mathbf{x}$  的特征，或者认为它提供了  $\mathbf{x}$  的一个新的表示。

剩下的问题就是如何选择映射  $\phi$ 。

1. 其中一种选择是使用一个通用的  $\phi$ ，例如无限维的  $\phi$ ，它隐含地用在基于 RBF 核的核机器上。如果  $\phi(\mathbf{x})$  具有足够的维数，我们总是有足够的能力来拟合训练集，但是对于测试集的泛化往往不佳。非常通用的特征映射通常只基于局部光滑的原则，并且没有将足够的先验信息进行编码来解决高级问题。
2. 另一种选择是手动地设计  $\phi$ 。在深度学习出现以前，这一直是主流的方法。这

种方法对于每个单独的任务都需要人们数十年的努力，从业者各自擅长特定的领域（如语音识别或计算机视觉），并且不同领域之间很难迁移（transfer）。

3. 深度学习的策略是去学习  $\phi$ 。在这种方法中，我们有一个模型  $y = f(\mathbf{x}; \theta, \mathbf{w}) = \phi(\mathbf{x}; \theta)^\top \mathbf{w}$ 。我们现在有两种参数：用于从一大类函数中学习  $\phi$  的参数  $\theta$ ，以及用于将  $\phi(\mathbf{x})$  映射到所需的输出的参数  $\mathbf{w}$ 。这是深度前馈网络的一个例子，其中  $\phi$  定义了一个隐藏层。这是三种方法中唯一一种放弃了训练问题的凸性的，但是利大于弊。在这种方法中，我们将表示参数化为  $\phi(\mathbf{x}; \theta)$ ，并且使用优化算法来寻找  $\theta$ ，使它能够得到一个好的表示。如果我们想要的话，这种方法也可以通过使它变得高度通用以获得第一种方法的优点——我们只需使用一个非常广泛的函数族  $\phi(\mathbf{x}; \theta)$ 。这种方法也可以获得第二种方法的优点。人类专家可以将他们的知识编码进网络来帮助泛化，他们只需要设计那些他们期望能够表现优异的函数族  $\phi(\mathbf{x}; \theta)$  即可。这种方法的优点是人类设计者只需要寻找正确的函数族即可，而不需要去寻找精确的函数。

这种通过学习特征来改善模型的一般化原则不仅仅适用于本章描述的前馈神经网络。它是深度学习中反复出现的主题，适用于全书描述的所有种类的模型。前馈神经网络是这个原则的应用，它学习从  $\mathbf{x}$  到  $\mathbf{y}$  的确定性映射并且没有反馈连接。后面出现的其他模型会把这些原则应用到学习随机映射、学习带有反馈的函数以及学习单个向量的概率分布。

本章我们先从前馈网络的一个简单例子说起。接着，我们讨论部署一个前馈网络所需的每个设计决策。首先，训练一个前馈网络至少需要做和线性模型同样多的设计决策：选择一个优化模型、代价函数以及输出单元的形式。我们先回顾这些基于梯度学习的基本知识，然后去面对那些只出现在前馈网络中的设计决策。前馈网络已经引入了隐藏层的概念，这需要我们去选择用于计算隐藏层值的 **激活函数**（activation function）。我们还必须设计网络的结构，包括网络应该包含多少层、这些层应该如何连接，以及每一层包含多少单元。在深度神经网络的学习中需要计算复杂函数的梯度。我们给出 **反向传播**（back propagation）算法和它的现代推广，它们可以用来高效地计算这些梯度。最后，我们以某些历史观点来结束这一章。

## 6.1 实例：学习 XOR

为了使前馈网络的想法更加具体，我们首先从一个可以完整工作的前馈网络说起。这个例子解决一个非常简单的任务：学习 XOR 函数。

XOR 函数（“异或”逻辑）是两个二进制值  $x_1$  和  $x_2$  的运算。当这些二进制值中恰好有一个为 1 时，XOR 函数返回值为 1。其余情况下返回值为 0。XOR 函数提供了我们想要学习的目标函数  $y = f^*(\mathbf{x})$ 。我们的模型给出了一个函数  $y = f(\mathbf{x}; \boldsymbol{\theta})$  并且我们的学习算法会不断调整参数  $\boldsymbol{\theta}$  来使得  $f$  尽可能接近  $f^*$ 。

在这个简单的例子中，我们不会关心统计泛化。我们希望网络在这四个点  $\mathbb{X} = \{[0, 0]^\top, [0, 1]^\top, [1, 0]^\top, [1, 1]^\top\}$  上表现正确。我们会用全部这四个点来训练我们的网络，唯一的挑战是拟合训练集。

我们可以把这个问题当作是回归问题，并使用均方误差损失函数。我们选择这个损失函数是为了尽可能简化本例中用到的数学。在应用领域，对于二进制数据建模时，MSE 通常并不是一个合适的损失函数。更加合适的方法将在第 6.2.2.2 节中讨论。

评估整个训练集上表现的 MSE 损失函数为

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\mathbf{x} \in \mathbb{X}} (f^*(\mathbf{x}) - f(\mathbf{x}; \boldsymbol{\theta}))^2. \quad (6.1)$$

我们现在必须要选择我们模型  $f(\mathbf{x}; \boldsymbol{\theta})$  的形式。假设我们选择一个线性模型， $\boldsymbol{\theta}$  包含  $\mathbf{w}$  和  $b$ ，那么我们的模型被定义成

$$f(\mathbf{x}; \mathbf{w}, b) = \mathbf{x}^\top \mathbf{w} + b. \quad (6.2)$$

我们可以使用正规方程关于  $\mathbf{w}$  和  $b$  最小化  $J(\boldsymbol{\theta})$ ，来得到一个闭式解。

解正规方程以后，我们得到  $\mathbf{w} = 0$  以及  $b = \frac{1}{2}$ 。线性模型仅仅是在任意一点都输出 0.5。为什么会发生这种事？图 6.1 演示了线性模型为什么不能用来表示 XOR 函数。解决这个问题的其中一种方法是使用一个模型来学习一个不同的特征空间，在这个空间上线性模型能够表示这个解。

具体来说，我们这里引入一个非常简单的前馈神经网络，它有一层隐藏层并且隐藏层中包含两个单元。见图 6.2 中对该模型的解释。这个前馈网络有一个通过函数  $f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$  计算得到的隐藏单元的向量  $\mathbf{h}$ 。这些隐藏单元的值随后被用作第二层的输入。第二层就是这个网络的输出层。输出层仍然只是一个线性回归模型，只不过

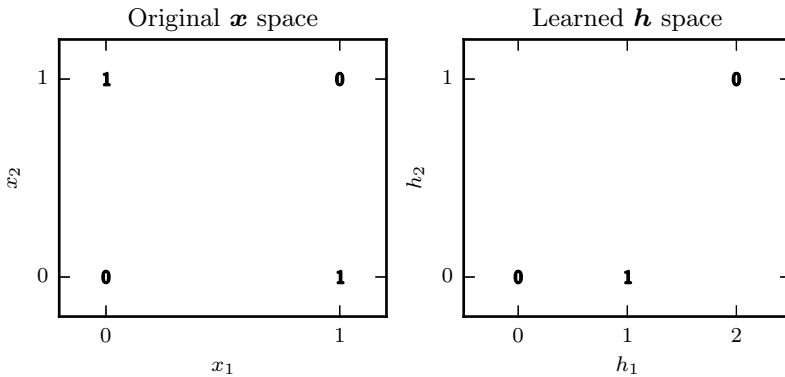


图 6.1: 通过学习一个表示来解决 XOR 问题。图上的粗体数字标明了学得的函数必须在每个点输出的值。(左) 直接应用于原始输入的线性模型不能实现 XOR 函数。当  $x_1 = 0$  时，模型的输出必须随着  $x_2$  的增大而增大。当  $x_1 = 1$  时，模型的输出必须随着  $x_2$  的增大而减小。线性模型必须对  $x_2$  使用固定的系数  $w_2$ 。因此，线性模型不能使用  $x_1$  的值来改变  $x_2$  的系数，从而不能解决这个问题。(右) 在由神经网络提取的特征表示的变换空间中，线性模型现在可以解决这个问题了。在我们的示例解决方案中，输出必须为 1 的两个点折叠到了特征空间中的单个点。换句话说，非线性特征将  $\mathbf{x} = [1, 0]^\top$  和  $\mathbf{x} = [0, 1]^\top$  都映射到了特征空间中的单个点  $\mathbf{h} = [1, 0]^\top$ 。线性模型现在可以将函数描述为  $h_1$  增大和  $h_2$  减小。在该示例中，学习特征空间的动机仅仅是使得模型的能力更大，使得它可以拟合训练集。在更现实的应用中，学习的表示也可以帮助模型泛化。

现在它作用于  $\mathbf{h}$  而不是  $\mathbf{x}$ 。网络现在包含链接在一起的两个函数： $\mathbf{h} = f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$  和  $y = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$ ，完整的模型是  $f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = f^{(2)}(f^{(1)}(\mathbf{x}))$ 。

$f^{(1)}$  应该是哪种函数？线性模型到目前为止都表现不错，让  $f^{(1)}$  也是线性的似乎很有诱惑力。可惜的是，如果  $f^{(1)}$  是线性的，那么前馈网络作为一个整体对于输入仍然是线性的。暂时忽略截距项，假设  $f^{(1)}(\mathbf{x}) = \mathbf{W}^\top \mathbf{x}$  并且  $f^{(2)}(\mathbf{h}) = \mathbf{h}^\top \mathbf{w}$ ，那么  $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{W}^\top \mathbf{x}$ 。我们可以将这个函数重新表示成  $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}'$  其中  $\mathbf{w}' = \mathbf{W}\mathbf{w}$ 。

显然，我们必须用非线性函数来描述这些特征。大多数神经网络通过仿射变换之后紧跟着一个被称为激活函数的固定非线性函数来实现这个目标，其中仿射变换由学得的参数控制。我们这里使用这种策略，定义  $\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{c})$ ，其中  $\mathbf{W}$  是线性变换的权重矩阵， $\mathbf{c}$  是偏置。此前，为了描述线性回归模型，我们使用权重向量和一个标量的偏置参数来描述从输入向量到输出标量的仿射变换。现在，因为我们描述的是向量  $\mathbf{x}$  到向量  $\mathbf{h}$  的仿射变换，所以我们需要一整个向量的偏置参数。激活函数  $g$  通常选择对每个元素分别起作用的函数，有  $h_i = g(\mathbf{x}^\top \mathbf{W}_{:,i} + c_i)$ 。在现代神经网络

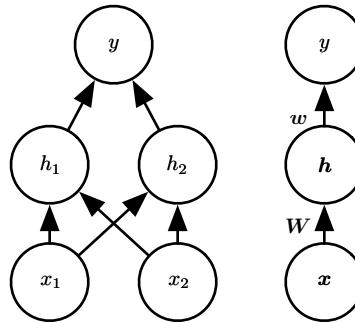


图 6.2: 使用两种不同样式绘制的前馈网络的示例。具体来说，这是我们用来解决 XOR 问题的前馈网络。它有单个隐藏层，包含两个单元。(左) 在这种样式中，我们将每个单元绘制为图中的一个节点。这种风格是清楚而明确的，但对于比这个例子更大的网络，它可能会消耗太多的空间。(右) 在这种样式中，我们将表示每一层激活的整个向量绘制为图中的一个节点。这种样式更加紧凑。有时，我们对图中的边使用参数名进行注释，这些参数是用来描述两层之间的关系的。这里，我们用矩阵  $W$  描述从  $x$  到  $h$  的映射，用向量  $w$  描述从  $h$  到  $y$  的映射。当标记这种图时，我们通常省略与每个层相关联的截距参数。

中，默认的推荐是使用由激活函数  $g(z) = \max\{0, z\}$  定义的整流线性单元 (rectified linear unit) 或者称为 ReLU (Jarrett *et al.*, 2009b; Nair and Hinton, 2010a; Glorot *et al.*, 2011a)，如图 6.3 所示。

我们现在可以指明我们的整个网络是

$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^\top \max\{0, \mathbf{W}^\top \mathbf{x} + \mathbf{c}\} + b. \quad (6.3)$$

我们现在可以给出 XOR 问题的一个解。令

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad (6.4)$$

$$\mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad (6.5)$$

$$\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad (6.6)$$

以及  $b = 0$ 。

我们现在可以了解这个模型如何处理一批输入。令  $\mathbf{X}$  表示设计矩阵，它包含二

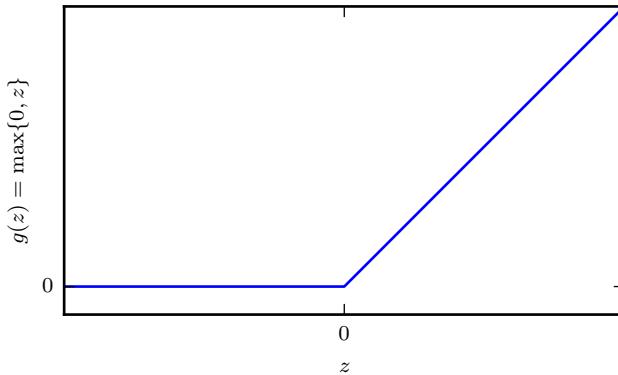


图 6.3: 整流线性激活函数。该激活函数是被推荐用于大多数前馈神经网络的默认激活函数。将此函数用于线性变换的输出将产生非线性变换。然而，函数仍然非常接近线性，在这种意义上它是具有两个线性部分的分段线性函数。由于整流线性单元几乎是线性的，因此它们保留了许多使得线性模型易于使用基于梯度的方法进行优化的属性。它们还保留了许多使得线性模型能够泛化良好的属性。计算机科学的一个通用原则是，我们可以从最小的组件构建复杂的系统。就像图灵机的内存只需要能够存储 0 或 1 的状态，我们可以从整流线性函数构建一个万能函数近似器。

进制输入空间中全部的四个点，每个样本占一行，那么矩阵表示为：

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \quad (6.7)$$

神经网络的第一步是将输入矩阵乘以第一层的权重矩阵：

$$\mathbf{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \quad (6.8)$$

然后，我们加上偏置向量  $\mathbf{c}$ ，得到

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \quad (6.9)$$

在这个空间中，所有的样本都处在一条斜率为 1 的直线上。当我们沿着这条直线移动时，输出需要从 0 升到 1，然后再降回 0。线性模型不能实现这样一种函数。为了用  $h$  对每个样本求值，我们使用整流线性变换：

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \quad (6.10)$$

这个变换改变了样本间的关系。它们不再处于同一条直线上了。如图 6.1 所示，它们现在处在一个可以用线性模型解决的空间上。

我们最后乘以一个权重向量  $w$ :

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \quad (6.11)$$

神经网络对这一批次中的每个样本都给出了正确的结果。

在这个例子中，我们简单地指定了解决方案，然后说明它得到的误差为零。在实际情况中，可能会有数十亿的模型参数以及数十亿的训练样本，所以不能像我们这里做的那样进行简单地猜解。与之相对的，基于梯度的优化算法可以找到一些参数使得产生的误差非常小。我们这里给出的 XOR 问题的解处在损失函数的全局最小点，所以梯度下降算法可以收敛到这一点。梯度下降算法还可以找到 XOR 问题一些其他的等价解。梯度下降算法的收敛点取决于参数的初始值。在实践中，梯度下降通常不会找到像我们这里给出的那种干净的、容易理解的、整数值的解。

## 6.2 基于梯度的学习

设计和训练神经网络与使用梯度下降训练其他任何机器学习模型并没有太大不同。在第 5.10 节中，我们描述了如何通过指定一个优化过程、代价函数和一个模型族来构建一个机器学习算法。

我们到目前为止看到的线性模型和神经网络的最大区别，在于神经网络的非线性导致大多数我们感兴趣的代价函数都变得非凸。这意味着神经网络的训练通常使

用迭代的、基于梯度的优化，仅仅使得代价函数达到一个非常小的值；而不是像用于训练线性回归模型的线性方程求解器，或者用于训练逻辑回归或 SVM 的凸优化算法那样保证全局收敛。凸优化从任何一种初始参数出发都会收敛（理论上如此——在实践中也很鲁棒但可能会遇到数值问题）。用于非凸损失函数的随机梯度下降没有这种收敛性保证，并且对参数的初始值很敏感。对于前馈神经网络，将所有的权重值初始化为小随机数是很重要的。偏置可以初始化为零或者小的正值。这种用于训练前馈神经网络以及几乎所有深度模型的迭代的基于梯度的优化算法会在第八章详细介绍，参数初始化会在第 8.4 节中具体说明。就目前而言，只需要懂得，训练算法几乎总是基于使用梯度来使得代价函数下降的各种方法即可。一些特别的算法是对梯度下降思想的改进和提纯（在第 4.3 节中介绍）还有一些更特别的，大多数是对随机梯度下降算法的改进（在第 5.9 节中介绍）。

我们当然也可以用梯度下降来训练诸如线性回归和支持向量机之类的模型，并且事实上当训练集相当大时这是很常用的。从这点来看，训练神经网络和训练其他任何模型并没有太大区别。计算梯度对于神经网络会略微复杂一些，但仍然可以很高效而精确地实现。第 6.5 节将会介绍如何用反向传播算法以及它的现代扩展算法来求得梯度。

和其他的机器学习模型一样，为了使用基于梯度的学习方法我们必须选择一个代价函数，并且我们必须选择如何表示模型的输出。现在，我们重温这些设计上的考虑，并且特别强调神经网络的情景。

### 6.2.1 代价函数

深度神经网络设计中的一个重要方面是代价函数的选择。幸运的是，神经网络的代价函数或多或少是和其他的参数模型例如线性模型的代价函数相同的。

在大多数情况下，我们的参数模型定义了一个分布  $p(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta})$  并且我们简单地使用最大似然原理。这意味着我们使用训练数据和模型预测间的交叉熵作为代价函数。

有时，我们使用一个更简单的方法，不是预测  $\mathbf{y}$  的完整概率分布，而是仅仅预测在给定  $\mathbf{x}$  的条件下  $\mathbf{y}$  的某种统计量。某些专门的损失函数允许我们来训练这些估计量的预测器。

用于训练神经网络的完整的代价函数，通常在我们这里描述的基本代价函数的

基础上结合一个正则项。我们已经在第 5.2.2 节中看到正则化应用到线性模型中的一些简单的例子。用于线性模型的权重衰减方法也直接适用于深度神经网络，而且是最流行的正则化策略之一。用于神经网络的更高级的正则化策略将在第七章中讨论。

### 6.2.1.1 使用最大似然学习条件分布

大多数现代的神经网络使用最大似然来训练。这意味着代价函数就是负的对数似然，它与训练数据和模型分布间的交叉熵等价。这个代价函数表示为

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y} \mid \mathbf{x}). \quad (6.12)$$

代价函数的具体形式随着模型而改变，取决于  $\log p_{\text{model}}$  的具体形式。上述方程的展开形式通常会有一些项不依赖于模型的参数，我们可以舍去。例如，正如我们在第 5.1.1 节中看到的，如果  $p_{\text{model}}(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{I})$ ，那么我们就重新得到了均方误差代价，

$$J(\boldsymbol{\theta}) = \frac{1}{2} \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \|\mathbf{y} - f(\mathbf{x}; \boldsymbol{\theta})\|^2 + \text{const}, \quad (6.13)$$

至少系数  $\frac{1}{2}$  和常数项不依赖于  $\boldsymbol{\theta}$ 。舍弃的常数是基于高斯分布的方差，在这种情况下我们选择不把它参数化。之前，我们看到了对输出分布的最大似然估计和对线性模型均方误差的最小化之间的等价性，但事实上，这种等价性并不要求  $f(\mathbf{x}; \boldsymbol{\theta})$  用于预测高斯分布的均值。

使用最大似然来导出代价函数的方法的一个优势是，它减轻了为每个模型设计代价函数的负担。明确一个模型  $p(\mathbf{y} \mid \mathbf{x})$  则自动地确定了一个代价函数  $\log p(\mathbf{y} \mid \mathbf{x})$ 。

贯穿神经网络设计的一个反复出现的主题是代价函数的梯度必须足够的大和具有足够的预测性，来为学习算法提供一个好的指引。饱和（变得非常平）的函数破坏了这一目标，因为它们把梯度变得非常小。这在很多情况下都会发生，因为用于产生隐藏单元或者输出单元的输出的激活函数会饱和。负的对数似然帮助我们在很多模型中避免这个问题。很多输出单元都会包含一个指数函数，这在它的变量取绝对值非常大的负值时会造成饱和。负对数似然代价函数中的对数函数消除了某些输出单元中的指数效果。我们将会在第 6.2.2 节中讨论代价函数和输出单元的选择间的相互作用。

用于实现最大似然估计的交叉熵代价函数有一个不同寻常的特性，那就是当它被应用于实践中经常遇到的模型时，它通常没有最小值。对于离散型输出变量，大

多数模型以一种特殊的形式来参数化，即它们不能表示概率零和一，但是可以无限接近。逻辑回归是其中一个例子。对于实值的输出变量，如果模型可以控制输出分布的密度（例如，通过学习高斯输出分布的方差参数），那么它可能对正确的训练集输出赋予极其高的密度，这将导致交叉熵趋向负无穷。第七章中描述的正则化技术提供了一些不同的方法来修正学习问题，使得模型不会通过这种方式来获得无限制的收益。

### 6.2.1.2 学习条件统计量

有时我们并不是想学习一个完整的概率分布  $p(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta})$ ，而仅仅是想学习在给定  $\mathbf{x}$  时  $\mathbf{y}$  的某个条件统计量。

例如，我们可能有一个预测器  $f(\mathbf{x}; \boldsymbol{\theta})$ ，我们想用它来预测  $\mathbf{y}$  的均值。如果我们使用一个足够强大的神经网络，我们可以认为这个神经网络能够表示一大类函数中的任何一个函数  $f$ ，这个类仅仅被一些特征所限制，例如连续性和有界，而不是具有特殊的参数形式。从这个角度来看，我们可以把代价函数看作是一个泛函（functional）而不仅仅是一个函数。泛函是函数到实数的映射。我们因此可以将学习看作是选择一个函数而不仅仅是选择一组参数。我们可以设计代价泛函在我们想要的某些特殊函数处取得最小值。例如，我们可以设计一个代价泛函，使它的最小值处于一个特殊的函数上，这个函数将  $\mathbf{x}$  映射到给定  $\mathbf{x}$  时  $\mathbf{y}$  的期望值。对函数求解优化问题需要用到变分法（calculus of variations）这个数学工具，我们将在第 19.4.2 节中讨论。理解变分法对于理解本章的内容不是必要的。目前，只需要知道变分法可以被用来导出下面的两个结果。

我们使用变分法导出的第一个结果是解优化问题

$$f^* = \arg \min_f \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}} \|\mathbf{y} - f(\mathbf{x})\|^2 \quad (6.14)$$

得到

$$f^*(\mathbf{x}) = \mathbb{E}_{\mathbf{y} \sim p_{\text{data}}(\mathbf{y} | \mathbf{x})} [\mathbf{y}], \quad (6.15)$$

要求这个函数处在我们要优化的类里。换句话说，如果我们能够用无穷多的、来源于真实的数据生成分布的样本进行训练，最小化均方误差代价函数将得到一个函数，它可以用来对每个  $\mathbf{x}$  的值预测出  $\mathbf{y}$  的均值。

不同的代价函数给出不同的统计量。第二个使用变分法得到的结果是

$$f^* = \arg \min_f \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim p_{\text{data}}} \|\mathbf{y} - f(\mathbf{x})\|_1 \quad (6.16)$$

将得到一个函数可以对每个  $\mathbf{x}$  预测  $\mathbf{y}$  取值的中位数，只要这个函数在我们要优化的函数族里。这个代价函数通常被称为 **平均绝对误差** ( mean absolute error )。

可惜的是，均方误差和平均绝对误差在使用基于梯度的优化方法时往往成效不佳。一些饱和的输出单元当结合这些代价函数时会产生非常小的梯度。这就是为什么交叉熵代价函数比均方误差或者平均绝对误差更受欢迎的原因之一了，即使是在没必要估计整个  $p(\mathbf{y} | \mathbf{x})$  分布时。

## 6.2.2 输出单元

代价函数的选择与输出单元的选择紧密相关。大多数时候，我们简单地使用数据分布和模型分布间的交叉熵。选择如何表示输出决定了交叉熵函数的形式。

任何可用作输出的神经网络单元，也可以被用作隐藏单元。这里，我们着重讨论将这些单元用作模型输出时的情况，不过原则上它们也可以在内部使用。我们将在第 6.3 节中重温这些单元，并且给出当它们被用作隐藏单元时一些额外的细节。

在本节中，我们假设前馈网络提供了一组定义为  $\mathbf{h} = f(\mathbf{x}; \boldsymbol{\theta})$  的隐藏特征。输出层的作用是随后对这些特征进行一些额外的变换来完成整个网络必须完成的任务。

### 6.2.2.1 用于高斯输出分布的线性单元

一种简单的输出单元是基于仿射变换的输出单元，仿射变换不具有非线性。这些单元往往被直接称为线性单元。

给定特征  $\mathbf{h}$ ，线性输出单元层产生一个向量  $\hat{\mathbf{y}} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}$ 。

线性输出层经常被用来产生条件高斯分布的均值：

$$p(\mathbf{y} | \mathbf{x}) = \mathcal{N}(\mathbf{y}; \hat{\mathbf{y}}, \mathbf{I}). \quad (6.17)$$

最大化其对数似然此时等价于最小化均方误差。

最大似然框架也使得学习高斯分布的协方差矩阵更加容易，或更容易地使高斯分布的协方差矩阵作为输入的函数。然而，对于所有输入，协方差矩阵都必须被限

定成一个正定矩阵。线性输出层很难满足这种限定，所以通常使用其他的输出单元来对协方差参数化。对协方差建模的方法将在第 6.2.2.4 节中简要介绍。

因为线性模型不会饱和，所以它们易于采用基于梯度的优化算法，甚至可以使用其他多种优化算法。

### 6.2.2.2 用于 Bernoulli 输出分布的 sigmoid 单元

许多任务需要预测二值型变量  $y$  的值。具有两个类的分类问题可以归结为这种形式。

此时最大似然的方法是定义  $y$  在  $\mathbf{x}$  条件下的 Bernoulli 分布。

Bernoulli 分布仅需单个参数来定义。神经网络只需要预测  $P(y = 1 | \mathbf{x})$  即可。为了使这个数是有效的概率，它必须处在区间  $[0, 1]$  中。

为满足该约束条件需要一些细致的设计工作。假设我们打算使用线性单元，并且通过阈值来限制它成为一个有效的概率：

$$P(y = 1 | \mathbf{x}) = \max \{0, \min\{1, \mathbf{w}^\top \mathbf{h} + b\}\}. \quad (6.18)$$

这的確定义了一个有效的条件概率分布，但我们无法使用梯度下降来高效地训练它。当  $\mathbf{w}^\top \mathbf{h} + b$  处于单位区间外时，模型的输出对其参数的梯度都将为  $\mathbf{0}$ 。梯度为  $\mathbf{0}$  通常是有问题的，因为学习算法对于如何改善相应的参数不再具有指导意义。

相反，最好是使用一种新的方法来保证无论何时模型给出了错误的答案时，总能有一个较大的梯度。这种方法是基于使用 sigmoid 输出单元结合最大似然来实现的。

sigmoid 输出单元定义为

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{h} + b), \quad (6.19)$$

这里  $\sigma$  是第 3.10 节中介绍的 logistic sigmoid 函数。

我们可以认为 sigmoid 输出单元具有两个部分。首先，它使用一个线性层来计算  $z = \mathbf{w}^\top \mathbf{h} + b$ 。接着，它使用 sigmoid 激活函数将  $z$  转化成概率。

我们暂时忽略对于  $\mathbf{x}$  的依赖性，只讨论如何用  $z$  的值来定义  $y$  的概率分布。sigmoid 可以通过构造一个非归一化（和不为 1）的概率分布  $\tilde{P}(y)$  来得到。我们可以随后除以一个合适的常数来得到有效的概率分布。如果我们假定非归一化的对数

概率对  $y$  和  $z$  是线性的，可以对它取指数来得到非归一化的概率。我们然后对它归一化，可以发现这服从 Bernoulli 分布，该分布受  $z$  的 sigmoid 变换控制：

$$\log \tilde{P}(y) = yz, \quad (6.20)$$

$$\tilde{P}(y) = \exp(yz), \quad (6.21)$$

$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^1 \exp(y'z)}, \quad (6.22)$$

$$P(y) = \sigma((2y - 1)z). \quad (6.23)$$

基于指数和归一化的概率分布在统计建模的文献中很常见。用于定义这种二值型变量分布的变量  $z$  被称为 **分对数 (logit)**。

这种在对数空间里预测概率的方法可以很自然地使用最大似然学习。因为用于最大似然的代价函数是  $-\log P(y | \mathbf{x})$ ，代价函数中的  $\log$  抵消了 sigmoid 中的  $\exp$ 。如果没有这个效果，sigmoid 的饱和性会阻止基于梯度的学习做出好的改进。我们使用最大似然来学习一个由 sigmoid 参数化的 Bernoulli 分布，它的损失函数为

$$J(\boldsymbol{\theta}) = -\log P(y | \mathbf{x}) \quad (6.24)$$

$$= -\log \sigma((2y - 1)z) \quad (6.25)$$

$$= \zeta((1 - 2y)z). \quad (6.26)$$

这个推导使用了第 3.10 节中的一些性质。通过将损失函数写成 softplus 函数的形式，我们可以看到它仅仅在  $(1 - 2y)z$  取绝对值非常大的负值时才会饱和。因此饱和只会出现在模型已经得到正确答案时——当  $y = 1$  且  $z$  取非常大的正值时，或者  $y = 0$  且  $z$  取非常小的负值时。当  $z$  的符号错误时，softplus 函数的变量  $(1 - 2y)z$  可以简化为  $|z|$ 。当  $|z|$  变得很大并且  $z$  的符号错误时，softplus 函数渐近地趋向于它的变量  $|z|$ 。对  $z$  求导则渐近地趋向于  $\text{sign}(z)$ ，所以，对于极限情况下极度不正确的  $z$ ，softplus 函数完全不会收缩梯度。这个性质很有用，因为它意味着基于梯度的学习可以很快地改正错误的  $z$ 。

当我们使用其他的损失函数，例如均方误差之类的，损失函数会在  $\sigma(z)$  饱和时饱和。sigmoid 激活函数在  $z$  取非常小的负值时会饱和到 0，当  $z$  取非常大的正值时会饱和到 1。这种情况一旦发生，梯度会变得非常小以至于不能用来学习，无论此时模型给出的是正确还是错误的答案。因此，最大似然几乎总是训练 sigmoid 输出单元的优选方法。

理论上，sigmoid 的对数总是确定和有限的，因为 sigmoid 的返回值总是被限制在开区间  $(0, 1)$  上，而不是使用整个闭区间  $[0, 1]$  的有效概率。在软件实现时，为了避免数值问题，最好将负的对数似然写作  $z$  的函数，而不是  $\hat{y} = \sigma(z)$  的函数。如果 sigmoid 函数下溢到零，那么之后对  $\hat{y}$  取对数会得到负无穷。

### 6.2.2.3 用于 Multinoulli 输出分布的 softmax 单元

任何时候当我们想要表示一个具有  $n$  个可能取值的离散型随机变量的分布时，我们都可以使用 softmax 函数。它可以看作是 sigmoid 函数的扩展，其中 sigmoid 函数用来表示二值型变量的分布。

softmax 函数最常用作分类器的输出，来表示  $n$  个不同类上的概率分布。比较少见的是，softmax 函数可以在模型内部使用，例如如果我们想要在某个内部变量的  $n$  个不同选项中进行选择。

在二值型变量的情况下，我们希望计算一个单独的数

$$\hat{y} = P(y = 1 \mid \mathbf{x}). \quad (6.27)$$

因为这个数需要处在 0 和 1 之间，并且我们想要让这个数的对数可以很好地用于对数似然的基于梯度的优化，我们选择去预测另外一个数  $z = \log \hat{P}(y = 1 \mid \mathbf{x})$ 。对其指数化和归一化，我们就得到了一个由 sigmoid 函数控制的 Bernoulli 分布。

为了推广到具有  $n$  个值的离散型变量的情况，我们现在需要创造一个向量  $\hat{\mathbf{y}}$ ，它的每个元素是  $\hat{y}_i = P(y = i \mid \mathbf{x})$ 。我们不仅要求每个  $\hat{y}_i$  元素介于 0 和 1 之间，还要使得整个向量的和为 1，使得它表示一个有效的概率分布。用于 Bernoulli 分布的方法同样可以推广到 Multinoulli 分布。首先，线性层预测了未归一化的对数概率：

$$\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}, \quad (6.28)$$

其中  $z_i = \log \hat{P}(y = i \mid \mathbf{x})$ 。softmax 函数然后可以对  $z$  指数化和归一化来获得需要的  $\hat{\mathbf{y}}$ 。最终，softmax 函数的形式为

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \quad (6.29)$$

和 logistic sigmoid一样，当使用最大化对数似然训练 softmax 来输出目标值  $y$  时，使用指数函数工作地非常好。这种情况下，我们想要最大化  $\log P(y = i; \mathbf{z}) =$

$\log \text{softmax}(\mathbf{z})_i$ 。将 softmax 定义成指数的形式是很自然的因为对数似然中的  $\log$  可以抵消 softmax 中的  $\exp$ :

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j). \quad (6.30)$$

式 (6.30) 中的第一项表示输入  $z_i$  总是对代价函数有直接的贡献。因为这一项不会饱和，所以即使  $z_i$  对式 (6.30) 的第二项的贡献很小，学习依然可以进行。当最大化对数似然时，第一项鼓励  $z_i$  被推高，而第二项则鼓励所有的  $z$  被压低。为了对第二项  $\log \sum_j \exp(z_j)$  有一个直观的理解，注意到这一项可以大致近似为  $\max_j z_j$ 。这种近似是基于对任何明显小于  $\max_j z_j$  的  $z_k$ ， $\exp(z_k)$  都是不重要的。我们能从这种近似中得到的直觉是，负对数似然代价函数总是强烈地惩罚最活跃的不正确预测。如果正确答案已经具有了 softmax 的最大输入，那么  $-z_i$  项和  $\log \sum_j \exp(z_j) \approx \max_j z_j = z_i$  项将大致抵消。这个样本对于整体训练代价贡献很小，这个代价主要由其他未被正确分类的样本产生。

到目前为止我们只讨论了一个例子。总体来说，未正则化的最大似然会驱动模型去学习一些参数，而这些参数会驱动 softmax 函数来预测在训练集中观察到的每个结果的比率：

$$\text{softmax}(\mathbf{z}(\mathbf{x}; \boldsymbol{\theta}))_i \approx \frac{\sum_{j=1}^m \mathbf{1}_{y^{(j)}=i, \mathbf{x}^{(j)}=\mathbf{x}}}{\sum_{j=1}^m \mathbf{1}_{\mathbf{x}^{(j)}=\mathbf{x}}}. \quad (6.31)$$

因为最大似然是一致的估计量，所以只要模型族能够表示训练的分布，这就能保证发生。在实践中，有限的模型能力和不完美的优化将意味着模型只能近似这些比率。

除了对数似然之外的许多目标函数对 softmax 函数不起作用。具体来说，那些不使用对数来抵消 softmax 中的指数的目标函数，当指数函数的变量取非常小的负值时会造成梯度消失，从而无法学习。特别是，平方误差对于 softmax 单元来说是一个很差的损失函数，即使模型做出高度可信的不正确预测，也不能训练模型改变其输出 (Bridle, 1990)。要理解为什么这些损失函数可能失败，我们需要检查 softmax 函数本身。

像 sigmoid 一样，softmax 激活函数可能会饱和。sigmoid 函数具有单个输出，当它的输入极端负或者极端正时会饱和。对于 softmax 的情况，它有多个输出值。当输入值之间的差异变得极端时，这些输出值可能饱和。当 softmax 饱和时，基于 softmax 的许多代价函数也饱和，除非它们能够转化饱和的激活函数。

为了说明 softmax 函数对于输入之间差异的响应，观察到当对所有的输入都加

上一个相同常数时 softmax 的输出不变：

$$\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} + c). \quad (6.32)$$

使用这个性质，我们可以导出一个数值方法稳定的 softmax 函数的变体：

$$\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} - \max_i z_i). \quad (6.33)$$

变换后的形式允许我们在对 softmax 函数求值时只有很小的数值误差，即使是当  $\mathbf{z}$  包含极正或者极负的数时。观察 softmax 数值稳定的变体，可以看到 softmax 函数由它的变量偏离  $\max_i z_i$  的量来驱动。

当其中一个输入是最大 ( $z_i = \max_i z_i$ ) 并且  $z_i$  远大于其他的输入时，相应的输出  $\text{softmax}(\mathbf{z})_i$  会饱和到 1。当  $z_i$  不是最大值并且最大值非常大时，相应的输出  $\text{softmax}(\mathbf{z})_i$  也会饱和到 0。这是 sigmoid 单元饱和方式的一般化，并且如果损失函数不被设计成对其进行补偿，那么也会造成类似的学习困难。

softmax 函数的变量  $\mathbf{z}$  可以通过两种方式产生。最常见的是简单地使神经网络较早的层输出  $\mathbf{z}$  的每个元素，就像先前描述的使用线性层  $\mathbf{z} = \mathbf{W}^\top \mathbf{h} + \mathbf{b}$ 。虽然很直观，但这种方法是对分布的过度参数化。 $n$  个输出总和必须为 1 的约束意味着只有  $n - 1$  个参数是必要的；第  $n$  个概率值可以通过 1 减去前面  $n - 1$  个概率来获得。因此，我们可以强制要求  $\mathbf{z}$  的一个元素是固定的。例如，我们可以要求  $z_n = 0$ 。事实上，这正是 sigmoid 单元所做的。定义  $P(y = 1 | \mathbf{x}) = \sigma(z)$  等价于用二维的  $\mathbf{z}$  以及  $z_1 = 0$  来定义  $P(y = 1 | \mathbf{x}) = \text{softmax}(\mathbf{z})_1$ 。无论是  $n - 1$  个变量还是  $n$  个变量的方法，都描述了相同的概率分布，但会产生不同的学习机制。在实践中，无论是过度参数化的版本还是限制的版本都很少有差别，并且实现过度参数化的版本更为简单。

从神经科学的角度看，有趣的是认为 softmax 是一种在参与其中的单元之间形成竞争的方式：softmax 输出总是和为 1，所以一个单元的值增加必然对应着其他单元值的减少。这与被认为存在于皮质中相邻神经元间的侧抑制类似。在极端情况下（当最大的  $a_i$  和其他的在幅度上差异很大时），它变成了赢家通吃（winner-take-all）的形式（其中一个输出接近 1，其他的接近 0）。

“softmax”的名称可能会让人产生困惑。这个函数更接近于 argmax 函数而不是 max 函数。“soft”这个术语来源于 softmax 函数是连续可微的。“argmax”函数的结果表示为一个 one-hot 向量（只有一个元素为 1，其余元素都为 0 的向量），不是连续和可微的。softmax 函数因此提供了 argmax 的“软化”版本。max 函数相应的软化版本是  $\text{softmax}(\mathbf{z})^\top \mathbf{z}$ 。可能最好是把 softmax 函数称为“softargmax”，但当前名称

已经是一个根深蒂固的习惯了。

### 6.2.2.4 其他的输出类型

之前描述的线性、sigmoid 和 softmax 输出单元是最常见的。神经网络可以推广到我们希望的几乎任何种类的输出层。最大似然原则给如何为几乎任何种类的输出层设计一个好的代价函数提供了指导。

一般的，如果我们定义了一个条件分布  $p(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta})$ ，最大似然原则建议我们使用  $-\log p(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta})$  作为代价函数。

一般来说，我们可以认为神经网络表示函数  $f(\mathbf{x}; \boldsymbol{\theta})$ 。这个函数的输出不是对  $\mathbf{y}$  值的直接预测。相反， $f(\mathbf{x}; \boldsymbol{\theta}) = \boldsymbol{\omega}$  提供了  $y$  分布的参数。我们的损失函数就可以表示成  $-\log p(\mathbf{y}; \boldsymbol{\omega}(\mathbf{x}))$ 。

例如，我们想要学习在给定  $\mathbf{x}$  时， $\mathbf{y}$  的条件高斯分布的方差。简单情况下，方差  $\sigma^2$  是一个常数，此时有一个解析表达式，这是因为方差的最大似然估计量仅仅是观测值  $\mathbf{y}$  与它们的期望值的差值的平方平均。一种计算上代价更加高但是不需要写特殊情况代码的方法是简单地将方差作为分布  $p(\mathbf{y} | \mathbf{x})$  的其中一个属性，这个分布由  $\boldsymbol{\omega} = f(\mathbf{x}; \boldsymbol{\theta})$  控制。负对数似然  $-\log p(\mathbf{y}; \boldsymbol{\omega}(\mathbf{x}))$  将为代价函数提供一个必要的合适项来使我们的优化过程可以逐渐地学到方差。在标准差不依赖于输入的简单情况下，我们可以在网络中创建一个直接复制到  $\boldsymbol{\omega}$  中的新参数。这个新参数可以是  $\sigma$  本身，或者可以是表示  $\sigma^2$  的参数  $v$ ，或者可以是表示  $\frac{1}{\sigma^2}$  的参数  $\beta$ ，取决于我们怎样对分布参数化。我们可能希望模型对不同的  $\mathbf{x}$  值预测出  $\mathbf{y}$  不同的方差。这被称为**异方差** (heteroscedastic) 模型。在异方差情况下，我们简单地把方差指定为  $f(\mathbf{x}; \boldsymbol{\theta})$  其中一个输出值。实现它的典型方法是使用精度而不是方差来表示高斯分布，就像式(3.22)所描述的。在多维变量的情况下，最常见的是使用一个对角精度矩阵

$$\text{diag}(\boldsymbol{\beta}). \quad (6.34)$$

这个公式适用于梯度下降，因为由  $\boldsymbol{\beta}$  参数化的高斯分布的对数似然的公式仅涉及  $\beta_i$  的乘法和  $\log \beta_i$  的加法。乘法、加法和对数运算的梯度表现良好。相比之下，如果我们用方差来参数化输出，我们需要用到除法。除法函数在零附近会变得任意陡峭。虽然大梯度可以帮助学习，但任意大的梯度通常导致不稳定。如果我们用标准差来参数化输出，对数似然仍然会涉及除法，并且还将涉及平方。通过平方运算的梯度可能在零附近消失，这使得学习被平方的参数变得困难。无论我们使用的是标准差，

方差还是精度，我们必须确保高斯分布的协方差矩阵是正定的。因为精度矩阵的特征值是协方差矩阵特征值的倒数，所以这等价于确保精度矩阵是正定的。如果我们使用对角矩阵，或者是一个常数乘以单位矩阵<sup>1</sup>，那么我们需要对模型输出强加的唯一条件是它的元素都为正。如果我们假设  $\mathbf{a}$  是用于确定对角精度的模型的原始激活，那么可以用 softplus 函数来获得正的精度向量： $\boldsymbol{\beta} = \zeta(\mathbf{a})$ 。这种相同的策略对于方差或标准差同样适用，也适用于常数乘以单位阵的情况。

学习一个比对角矩阵具有更丰富结构的协方差或者精度矩阵是很少见的。如果协方差矩阵是满的和有条件的，那么参数化的选择就必须要保证预测的协方差矩阵是正定的。这可以通过写成  $\Sigma(\mathbf{x}) = \mathbf{B}(\mathbf{x})\mathbf{B}^\top(\mathbf{x})$  来实现，这里  $\mathbf{B}$  是一个无约束的方阵。如果矩阵是满秩的，那么一个实际问题是计算似然的代价是很高的，计算一个  $d \times d$  的矩阵的行列式或者  $\Sigma(\mathbf{x})$  的逆（或者等价地并且更常用地，对它特征值分解或者  $\mathbf{B}(\mathbf{x})$  的特征值分解）需要  $O(d^3)$  的计算量。

我们经常想要执行多峰回归 (multimodal regression)，即预测条件分布  $p(\mathbf{y} | \mathbf{x})$  的实值，该条件分布对于相同的  $\mathbf{x}$  值在  $\mathbf{y}$  空间中有多个不同的峰值。在这种情况下，高斯混合是输出的自然表示 (Jacobs *et al.*, 1991; Bishop, 1994)。将高斯混合作为其输出的神经网络通常被称为 混合密度网络 (mixture density network)。具有  $n$  个分量的高斯混合输出由下面的条件分布定义：

$$p(\mathbf{y} | \mathbf{x}) = \sum_{i=1}^n p(c=i | \mathbf{x}) \mathcal{N}(\mathbf{y}; \boldsymbol{\mu}^{(i)}(\mathbf{x}), \boldsymbol{\Sigma}^{(i)}(\mathbf{x})). \quad (6.35)$$

神经网络必须有三个输出：定义  $p(c=i | \mathbf{x})$  的向量，对所有的  $i$  给出  $\boldsymbol{\mu}^{(i)}(\mathbf{x})$  的矩阵，以及对所有的  $i$  给出  $\boldsymbol{\Sigma}^{(i)}(\mathbf{x})$  的张量。这些输出必须满足不同的约束：

1. 混合组件  $p(c=i | \mathbf{x})$ ：它们由潜变量<sup>2</sup>  $c$  关联着，在  $n$  个不同组件上形成 Multinoulli 分布。这个分布通常可以由  $n$  维向量的 softmax 来获得，以确保这些输出是正的并且和为 1。
2. 均值  $\boldsymbol{\mu}^{(i)}(\mathbf{x})$ ：它们指明了与第  $i$  个高斯组件相关联的中心或者均值，并且是无约束的（通常对于这些输出单元完全没有非线性）。如果  $\mathbf{y}$  是个  $d$  维向量，那

<sup>1</sup>译者注：这里原文是 “If we use a diagonal matrix, or a scalar times the diagonal matrix...” 即 “如果我们使用对角矩阵，或者是一个标量乘以对角矩阵...”，但一个标量乘以对角矩阵和对角矩阵没区别，结合上下文可以看出，这里原作者误把“identity”写成了“diagonal matrix”，因此这里采用“常数乘以单位矩阵”的译法。

<sup>2</sup>我们之所以认为  $c$  是潜在的，是因为我们不能直接在数据中观测到它：给定输入  $\mathbf{x}$  和目标  $\mathbf{y}$ ，不可能确切地知道是哪个高斯组件产生  $\mathbf{y}$ ，但我们可以想象  $\mathbf{y}$  是通过选择其中一个来产生的，并且将那个未被观测到的选择作为随机变量。

么网络必须输出一个由  $n$  个这种  $d$  维向量组成的  $n \times d$  的矩阵。用最大似然来学习这些均值要比学习只有一个输出模式的分布的均值稍稍复杂一些。我们只想更新那个真正产生观测数据的组件的均值。在实践中，我们并不知道是哪个组件产生了观测数据。负对数似然表达式将每个样本对每个组件的贡献进行赋权，权重的大小由相应的组件产生这个样本的概率来决定。

3. 协方差  $\Sigma^{(i)}(\mathbf{x})$ : 它们指明了每个组件  $i$  的协方差矩阵。和学习单个高斯组件时一样，我们通常使用对角矩阵来避免计算行列式。和学习混合均值时一样，最大似然是很复杂的，它需要将每个点的部分责任分配给每个混合组件。如果给定了混合模型的正确的负对数似然，梯度下降将自动地遵循正确的过程。

有报告说基于梯度的优化方法对于混合条件高斯（作为神经网络的输出）可能是不可靠的，部分是因为涉及到除法（除以方差）可能是数值不稳定的（当某个方差对于特定的实例变得非常小时，会导致非常大的梯度）。一种解决方法是梯度截断（clip gradient）（见第 10.11.1 节），另外一种是启发式缩放梯度（Murray and Larochelle, 2014）。

高斯混合输出在语音生成模型 (Schuster, 1999) 和物理运动 (Graves, 2013) 中特别有效。混合密度策略为网络提供了一种方法来表示多种输出模式，并且控制输出的方差，这对于在这些实数域中获得高质量的结果是至关重要的。混合密度网络的一个实例如图 6.4 所示。

一般的，我们可能希望继续对包含更多变量的、更大的向量  $\mathbf{y}$  来建模，并在这些输出变量上施加更多更丰富的结构。例如，我们可能希望神经网络输出字符序列形成一个句子。在这些情况下，我们可以继续使用最大似然原理应用到我们的模型  $p(\mathbf{y}; \omega(\mathbf{x}))$  上，但我们用来描述  $\mathbf{y}$  的模型会变得非常复杂，超出了本章的范畴。第十章描述了如何使用循环神经网络来定义这种序列上的模型，第三部分描述了对任意概率分布进行建模的高级技术。

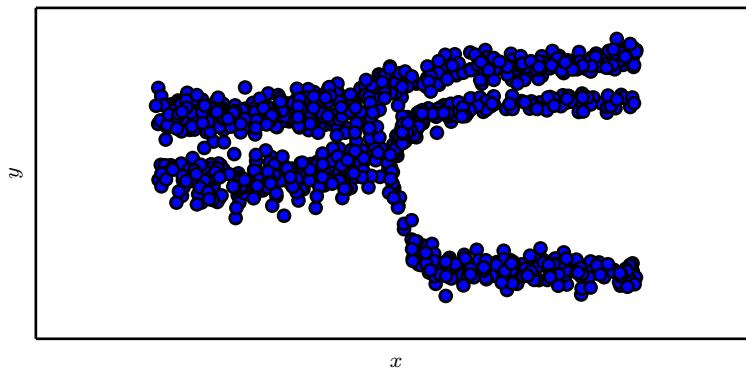


图 6.4: 从具有混合密度输出层的神经网络中抽取的样本。输入  $x$  从均匀分布中采样，输出  $y$  从  $p_{\text{model}}(y | x)$  中采样。神经网络能够学习从输入到输出分布的参数的非线性映射。这些参数包括控制三个组件中的哪一个将产生输出的概率，以及每个组件各自的参数。每个混合组件都是高斯分布，具有预测的均值和方差。输出分布的这些方面都能够相对输入  $x$  变化，并且以非线性的方式改变。

### 6.3 隐藏单元

到目前为止，我们集中讨论了神经网络的设计选择，这对于使用基于梯度的优化方法来训练的大多数参数化机器学习模型都是通用的。现在我们转向一个前馈神经网络独有的问题：该如何选择隐藏单元的类型，这些隐藏单元用在模型的隐藏层中。

隐藏单元的设计是一个非常活跃的研究领域，并且还没有许多明确的指导性理论原则。

整流线性单元是隐藏单元极好的默认选择。许多其他类型的隐藏单元也是可用的。决定何时使用哪种类型的隐藏单元是困难的事（尽管整流线性单元通常是一个可接受的选择）。我们这里描述对于每种隐藏单元的一些基本直觉。这些直觉可以用来建议我们何时来尝试一些单元。通常不可能预先预测出哪种隐藏单元工作得最好。设计过程充满了试验和错误，先直觉认为某种隐藏单元可能表现良好，然后用它组成神经网络进行训练，最后用验证集来评估它的性能。

这里列出的一些隐藏单元可能并不是在所有的输入点上都是可微的。例如，整流线性单元  $g(z) = \max\{0, z\}$  在  $z = 0$  处不可微。这似乎使得  $g$  对于基于梯度的学习算法无效。在实践中，梯度下降对这些机器学习模型仍然表现得足够好。部分原因

是神经网络训练算法通常不会达到代价函数的局部最小值，而是仅仅显著地减小它的值，如图 4.3 所示。这些想法会在第八章中进一步描述。因为我们不再期望训练能够实际到达梯度为  $\mathbf{0}$  的点，所以代价函数的最小值对应于梯度未定义的点是可以接受的。不可微的隐藏单元通常只在少数点上不可微。一般来说，函数  $g(z)$  具有左导数和右导数，左导数定义为紧邻在  $z$  左边的函数的斜率，右导数定义为紧邻在  $z$  右边的函数的斜率。只有当函数在  $z$  处的左导数和右导数都有定义并且相等时，函数在  $z$  点处才是可微的。神经网络中用到的函数通常对左导数和右导数都有定义。在  $g(z) = \max\{0, z\}$  的情况下，在  $z = 0$  处的左导数是 0，右导数是 1。神经网络训练的软件实现通常返回左导数或右导数的其中一个，而不是报告导数未定义或产生一个错误。这可以通过观察到在数字计算机上基于梯度的优化总是会受到数值误差的影响来启发式地给出理由。当一个函数被要求计算  $g(0)$  时，底层值真正为 0 是不太可能的。相对的，它可能是被舍入为 0 的一个小量  $\epsilon$ 。在某些情况下，理论上有更好的理由，但这些通常对神经网络训练并不适用。重要的是，在实践中，我们可以放心地忽略下面描述的隐藏单元激活函数的不可微性。

除非另有说明，大多数的隐藏单元都可以描述为接受输入向量  $\mathbf{x}$ ，计算仿射变换  $\mathbf{z} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$ ，然后使用一个逐元素的非线性函数  $g(\mathbf{z})$ 。大多数隐藏单元的区别仅仅在于激活函数  $g(\mathbf{z})$  的形式。

### 6.3.1 整流线性单元及其扩展

整流线性单元使用激活函数  $g(z) = \max\{0, z\}$ 。

整流线性单元易于优化，因为它们和线性单元非常类似。线性单元和整流线性单元的唯一区别在于整流线性单元在其一半的定义域上输出为零。这使得只要整流线性单元处于激活状态，它的导数都能保持较大。它的梯度不仅大而且一致。整流操作的二阶导数几乎处处为 0，并且在整流线性单元处于激活状态时，它的一阶导数处处为 1。这意味着相比于引入二阶效应的激活函数来说，它的梯度方向对于学习来说更加有用。

整流线性单元通常作用于仿射变换之上：

$$\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{b}). \quad (6.36)$$

当初始化仿射变换的参数时，可以将  $\mathbf{b}$  的所有元素设置成一个很小的正值，例如 0.1。这使得整流线性单元很可能初始时就对训练集中的大多数输入呈现激活状态，并且

允许导数通过。

有很多整流线性单元的扩展存在。大多数这些扩展的表现比得上整流线性单元，并且偶尔表现得更好。

整流线性单元的一个缺陷是它们不能通过基于梯度的方法学习那些使它们激活为零的样本。整流线性单元的各种扩展保证了它们能在各个位置都接收到梯度。

整流线性单元的三个扩展基于当  $z_i < 0$  时使用一个非零的斜率  $\alpha_i$ :  $h_i = g(\mathbf{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$ 。绝对值整流 (absolute value rectification) 固定  $\alpha_i = -1$  来得到  $g(z) = |z|$ 。它用于图像中的对象识别 (Jarrett *et al.*, 2009a)，其中寻找在输入照明极性反转下不变的特征是有意义的。整流线性单元的其他扩展比这应用地更广泛。渗漏整流线性单元 (Leaky ReLU) (Maas *et al.*, 2013) 将  $\alpha_i$  固定成一个类似 0.01 的小值，参数化整流线性单元 (parametric ReLU) 或者 PReLU 将  $\alpha_i$  作为学习的参数 (He *et al.*, 2015)。

**maxout 单元** (maxout unit) (Goodfellow *et al.*, 2013a) 进一步扩展了整流线性单元。maxout 单元将  $\mathbf{z}$  划分为每组具有  $k$  个值的组，而不是使用作用于每个元素的函数  $g(z)$ 。每个 maxout 单元则输出每组中的最大元素：

$$g(\mathbf{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j \quad (6.37)$$

这里  $\mathbb{G}^{(i)}$  是组  $i$  的输入索引集  $\{(i-1)k+1, \dots, ik\}$ 。这提供了一种方法来学习对输入  $\mathbf{x}$  空间中多个方向响应的分段线性函数。

maxout 单元可以学习具有多达  $k$  段的分段线性的凸函数。maxout 单元因此可以视为学习激活函数本身而不仅仅是单元之间的关系。使用足够大的  $k$ ，maxout 单元可以以任意的精确度来近似任何凸函数。特别地，具有两块的 maxout 层可以学习实现和传统层相同的输入  $\mathbf{x}$  的函数，这些传统层可以使用整流线性激活函数、绝对值整流、渗漏整流线性单元 或参数化整流线性单元，或者可以学习实现与这些都不同的函数。maxout 层的参数化当然也将与这些层不同，所以即使是 maxout 学习去实现和其他种类的层相同的  $\mathbf{x}$  的函数这种情况下，学习的机理也是不一样的。

每个 maxout 单元现在由  $k$  个权重向量来参数化，而不仅仅是一个，所以 maxout 单元通常比整流线性单元需要更多的正则化。如果训练集很大并且每个单元的块数保持很低的话，它们可以在没有正则化的情况下工作得不错 (Cai *et al.*, 2013)。

maxout 单元还有一些其他的优点。在某些情况下，要求更少的参数可以获得一些统计和计算上的优点。具体来说，如果由  $n$  个不同的线性过滤器描述的特征可以

在不损失信息的情况下，用每一组  $k$  个特征的最大值来概括的话，那么下一层可以获得  $k$  倍更少的权重数。

因为每个单元由多个过滤器驱动，maxout 单元具有一些冗余来帮助它们抵抗一种被称为 **灾难遗忘** (catastrophic forgetting) 的现象，这个现象是说神经网络忘记了如何执行它们过去训练的任务 (Goodfellow *et al.*, 2014a)。

整流线性单元和它们的这些扩展都是基于一个原则，那就是如果它们的行为更接近线性，那么模型更容易优化。使用线性行为更容易优化的一般性原则同样也适用于除深度线性网络以外的情景。循环网络可以从序列中学习并产生状态和输出的序列。当训练它们时，需要通过一些时间步来传播信息，当其中包含一些线性计算（具有大小接近 1 的某些方向导数）时，这会更容易。作为性能最好的循环网络结构之一，LSTM 通过求和在时间上传播信息，这是一种特别直观的线性激活。它将在第 10.10 节中进一步讨论。

### 6.3.2 logistic sigmoid 与 双曲正切函数

在引入整流线性单元之前，大多数神经网络使用 logistic sigmoid 激活函数

$$g(z) = \sigma(z) \quad (6.38)$$

或者是双曲正切激活函数

$$g(z) = \tanh(z). \quad (6.39)$$

这些激活函数紧密相关，因为  $\tanh(z) = 2\sigma(2z) - 1$ 。

我们已经看过 sigmoid 单元作为输出单元用来预测二值型变量取值为 1 的概率。与分段线性单元不同，sigmoid 单元在其大部分定义域内都饱和——当  $z$  取绝对值很大的正值时，它们饱和到一个高值，当  $z$  取绝对值很大的负值时，它们饱和到一个低值，并且仅仅当  $z$  接近 0 时它们才对输入强烈敏感。sigmoid 单元的广泛饱和性会使得基于梯度的学习变得非常困难。因为这个原因，现在不鼓励将它们用作前馈网络中的隐藏单元。当使用一个合适的代价函数来抵消 sigmoid 的饱和性时，它们作为输出单元可以与基于梯度的学习相兼容。

当必须要使用 sigmoid 激活函数时，双曲正切激活函数通常要比 logistic sigmoid 函数表现更好。在  $\tanh(0) = 0$  而  $\sigma(0) = \frac{1}{2}$  的意义上，它更像是单位函数。因为  $\tanh$  在 0 附近与单位函数类似，训练深层神经网络  $\hat{y} = \mathbf{w}^\top \tanh(\mathbf{U}^\top \tanh(\mathbf{V}^\top \mathbf{x}))$

类似于训练一个线性模型  $\hat{y} = \mathbf{w}^\top \mathbf{U}^\top \mathbf{V}^\top \mathbf{x}$ , 只要网络的激活能够被保持地很小。这使得训练 tanh 网络更加容易。

sigmoid 激活函数在除了前馈网络以外的情景中更为常见。循环网络、许多概率模型以及一些自编码器有一些额外的要求使得它们不能使用分段线性激活函数，并且使得 sigmoid 单元更具有吸引力，尽管它存在饱和性的问题。

### 6.3.3 其他隐藏单元

也存在许多其他种类的隐藏单元，但它们并不常用。

一般来说，很多种类的可微函数都表现得很好。许多未发布的激活函数与流行的激活函数表现得一样好。为了提供一个具体的例子，作者在 MNIST 数据集上使用  $\mathbf{h} = \cos(\mathbf{W}\mathbf{x} + \mathbf{b})$  测试了一个前馈网络，并获得了小于 1% 的误差率，这可以与更为传统的激活函数获得的结果相媲美。在新技术的研究和开发期间，通常会测试许多不同的激活函数，并且会发现许多标准方法的变体表现非常好。这意味着，通常新的隐藏单元类型只有在被明确证明能够提供显著改进时才会被发布。新的隐藏单元类型如果与已有的隐藏单元表现大致相当的话，那么它们是非常常见的，不会引起别人的兴趣。

列出文献中出现的所有隐藏单元类型是不切实际的。我们只对一些特别有用和独特的类型进行强调。

其中一种是完全没有激活函数  $g(z)$ 。也可以认为这是使用单位函数作为激活函数的情况。我们已经看过线性单元可以用作神经网络的输出。它也可以用作隐藏单元。如果神经网络的每一层都仅由线性变换组成，那么网络作为一个整体也将是线性的。然而，神经网络的一些层是纯线性也是可以接受的。考虑具有  $n$  个输入和  $p$  个输出的神经网络层  $\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + \mathbf{b})$ 。我们可以用两层来代替它，一层使用权重矩阵  $\mathbf{U}$ ，另一层使用权重矩阵  $\mathbf{V}$ 。如果第一层没有激活函数，那么我们对基于  $\mathbf{W}$  的原始层的权重矩阵进行因式分解。分解方法是计算  $\mathbf{h} = g(\mathbf{V}^\top \mathbf{U}^\top \mathbf{x} + \mathbf{b})$ 。如果  $\mathbf{U}$  产生了  $q$  个输出，那么  $\mathbf{U}$  和  $\mathbf{V}$  一起仅包含  $(n + p)q$  个参数，而  $\mathbf{W}$  包含  $np$  个参数。如果  $q$  很小，这可以在很大程度上节省参数。这是以将线性变换约束为低秩的代价来实现的，但这些低秩关系往往是足够的。线性隐藏单元因此提供了一种减少网络中参数数量的有效方法。

softmax 单元是另外一种经常用作输出的单元（如第 6.2.3 节中所描述的），但

有时也可以用作隐藏单元。softmax 单元很自然地表示具有  $k$  个可能值的离散型随机变量的概率分布，所以它们可以用作一种开关。这些类型的隐藏单元通常仅用于明确地学习操作内存的高级结构中，将在第 10.12 节中描述。

其他一些常见的隐藏单元类型包括：

- **径向基函数** (radial basis function, RBF):  $h_i = \exp\left(-\frac{1}{\sigma_i^2} \|\mathbf{W}_{:,i} - \mathbf{x}\|^2\right)$ 。这个函数在  $\mathbf{x}$  接近模板  $\mathbf{W}_{:,i}$  时更加活跃。因为它对大部分  $\mathbf{x}$  都饱和到 0，因此很难优化。
- **softplus** 函数:  $g(a) = \zeta(a) = \log(1 + e^a)$ 。这是整流线性单元的平滑版本，由 Dugas *et al.* (2001) 引入用于函数近似，由 Nair and Hinton (2010a) 引入用于无向概率模型的条件分布。Glorot *et al.* (2011a) 比较了 softplus 和整流线性单元，发现后者的结果更好。通常不鼓励使用 softplus 函数。softplus 表明隐藏单元类型的性能可能是非常反直觉的——因为它处处可导或者因为它不完全饱和，人们可能希望它具有优于整流线性单元的点，但根据经验来看，它并没有。
- **硬双曲正切函数** (hard tanh): 它的形状和 tanh 以及整流线性单元类似，但是不同于后者，它是有界的， $g(a) = \max(-1, \min(1, a))$ 。它由 Collobert (2004) 引入。

隐藏单元的设计仍然是一个活跃的研究领域，许多有用的隐藏单元类型仍有待发现。

## 6.4 架构设计

神经网络设计的另一个关键点是确定它的架构。架构 (architecture) 一词是指网络的整体结构：它应该具有多少单元，以及这些单元应该如何连接。

大多数神经网络被组织成称为层的单元组。大多数神经网络架构将这些层布置成链式结构，其中每一层都是前一层的函数。在这种结构中，第一层由下式给出：

$$\mathbf{h}^{(1)} = g^{(1)} \left( \mathbf{W}^{(1)\top} \mathbf{x} + \mathbf{b}^{(1)} \right); \quad (6.40)$$

第二层由

$$\mathbf{h}^{(2)} = g^{(2)} \left( \mathbf{W}^{(2)\top} \mathbf{h}^{(1)} + \mathbf{b}^{(2)} \right); \quad (6.41)$$

给出，以此类推。

在这些链式架构中，主要的架构考虑是选择网络的深度和每一层的宽度。我们将会看到，即使只有一个隐藏层的网络也足够适应训练集。更深层的网络通常能够对每一层使用更少的单元数和更少的参数，并且经常容易泛化到测试集，但是通常也更难以优化。对于一个具体的任务，理想的网络架构必须通过实验，观测在验证集上的误差来找到。

### 6.4.1 万能近似性质和深度

线性模型，通过矩阵乘法将特征映射到输出，顾名思义，仅能表示线性函数。它具有易于训练的优点，因为当使用线性模型时，许多损失函数会导出凸优化问题。可惜的是，我们经常希望我们的系统学习非线性函数。

乍一看，我们可能认为学习非线性函数需要为我们想要学习的那种非线性专门设计一类模型族。幸运的是，具有隐藏层的前馈网络提供了一种万能近似框架。具体来说，**万能近似定理** (universal approximation theorem) (Hornik *et al.*, 1989; Cybenko, 1989) 表明，一个前馈神经网络如果具有线性输出层和至少一层具有任何一种“挤压”性质的激活函数（例如logistic sigmoid激活函数）的隐藏层，只要给予网络足够数量的隐藏单元，它可以以任意的精度来近似任何从一个有限维空间到另一个有限维空间的 Borel 可测函数。前馈网络的导数也可以任意好地来近似函数的导数 (Hornik *et al.*, 1990)。Borel 可测的概念超出了本书的范畴；对于我们想要实现的目标，只需要知道定义在  $\mathbb{R}^n$  的有界闭集上的任意连续函数是 Borel 可测的，因此可以用神经网络来近似。神经网络也可以近似从任何有限维离散空间映射到另一个的任意函数。虽然原始定理最初以具有特殊激活函数的单元的形式来描述，这个激活函数当变量取绝对值非常大的正值和负值时都会饱和，万能近似定理也已经被证明对于更广泛类别的激活函数也是适用的，其中就包括现在常用的整流线性单元 (Leshno *et al.*, 1993)。

万能近似定理意味着无论我们试图学习什么函数，我们知道一个大的 MLP 一定能够表示这个函数。然而，我们不能保证训练算法能够学得这个函数。即使 MLP 能够表示该函数，学习也可能因两个不同的原因而失败。首先，用于训练的优化算法可能找不到用于期望函数的参数值。其次，训练算法可能由于过拟合而选择了错误的函数。回忆第 5.2.1 节中的“没有免费的午餐”定理，说明了没有普遍优越的机器学习算法。前馈网络提供了表示函数的万能系统，在这种意义上，给定一个函数，

存在一个前馈网络能够近似该函数。不存在万能的过程既能够验证训练集上的特殊样本，又能够选择一个函数来扩展到训练集上没有的点。

万能近似定理说明了，存在一个足够大的网络能够达到我们所希望的任意精度，但是定理并没有说这个网络有多大。Barron (1993) 提供了单层网络近似一大类函数所需大小的一些界。不幸的是，在最坏情况下，可能需要指数数量的隐藏单元（可能一个隐藏单元对应着一个需要区分的输入配置）。这在二进制值的情况下很容易看到：向量  $v \in \{0, 1\}^n$  上的可能的二值型函数的数量是  $2^{2^n}$ ，并且选择一个这样的函数需要  $2^n$  位，这通常需要  $O(2^n)$  的自由度。

总之，具有单层的前馈网络足以表示任何函数，但是网络层可能大得不可实现，并且可能无法正确地学习和泛化。在很多情况下，使用更深的模型能够减少表示期望函数所需的单元的数量，并且可以减少泛化误差。

存在一些函数族能够在网络的深度大于某个值  $d$  时被高效地近似，而当深度被限制到小于或等于  $d$  时需要一个远远大于之前的模型。在很多情况下，浅层模型所需的隐藏单元的数量是  $n$  的指数级。这个结果最初被证明是在那些不与连续可微的神经网络类似的机器学习模型中出现，但现在已经扩展到了这些模型。第一个结果是关于逻辑门电路的 (Håstad, 1986)。后来的工作将这些结果扩展到了具有非负权重的线性阈值单元 (Håstad and Goldmann, 1991; Hajnal *et al.*, 1993)，然后扩展到了具有连续值激活的网络 (Maass, 1992; Maass *et al.*, 1994)。许多现代神经网络使用整流线性单元。Leshno *et al.* (1993) 证明带有一大类非多项式激活函数族的浅层网络，包括整流线性单元，具有万能的近似性质，但是这些结果并没有强调深度或效率的问题——它们仅指出足够宽的整流网络能够表示任意函数。Montufar *et al.* (2014) 指出一些用深度整流网络表示的函数可能需要浅层网络（一个隐藏层）指数级的隐藏单元才能表示。更确切的说，他们说明分段线性网络（可以通过整流非线性或 maxout 单元获得）可以表示区域的数量是网络深度的指数级的函数。图 6.5 解释了带有绝对值整流的网络是如何创建函数的镜像图像的，这些函数在某些隐藏单元的顶部计算，作用于隐藏单元的输入。每个隐藏单元指定在哪里折叠输入空间，来创造镜像响应（在绝对值非线性的两侧）。通过组合这些折叠操作，我们获得指数级的分段线性区域，他们可以概括所有种类的规则模式（例如，重复）。

Montufar *et al.* (2014) 的主要定理指出，具有  $d$  个输入、深度为  $l$ 、每个隐藏

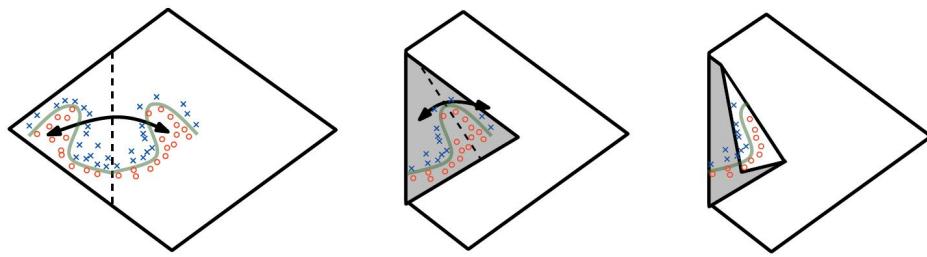


图 6.5: 关于更深的整流网络具有指数优势的一个直观的几何解释, 来自 Montufar *et al.* (2014)。(左) 绝对值整流单元对其输入中的每对镜像点有相同的输出。镜像的对称轴由单元的权重和偏置定义的超平面给出。在该单元顶部计算的函数 (绿色决策面) 将是横跨该对称轴的更简单模式的一个镜像。(中) 该函数可以通过折叠对称轴周围的空间来得到。(右) 另一个重复模式可以在第一个的顶部折叠 (由另一个下游单元) 以获得另外的对称性 (现在重复四次, 使用了两个隐藏层)。经 Montufar *et al.* (2014) 许可改编此图。

层具有  $n$  个单元的深度整流网络可以描述的线性区域的数量是

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right), \quad (6.42)$$

意味着, 这是深度  $l$  的指数级。在每个单元具有  $k$  个过滤器的 maxout 网络中, 线性区域的数量是

$$O(k^{(l-1)+d}). \quad (6.43)$$

当然, 我们不能保证在机器学习 (特别是 AI) 的应用中我们想要学得的函数类型享有这样的属性。

我们还可能出于统计原因来选择深度模型。任何时候, 当我们选择一个特定的机器学习算法时, 我们隐含地陈述了一些先验, 这些先验是关于算法应该学得什么样的函数的。选择深度模型默许了一个非常普遍的信念, 那就是我们想要学得的函数应该涉及几个更加简单的函数的组合。这可以从表示学习的观点来解释, 我们相信学习的问题包含发现一组潜在的变差因素, 它们可以根据其他更简单的潜在的变差因素来描述。或者, 我们可以将深度结构的使用解释为另一种信念, 那就是我们想要学得的函数是包含多个步骤的计算机程序, 其中每个步骤使用前一步骤的输出。这些中间输出不一定是变差因素, 而是可以类似于网络用来组织其内部处理的计数器或指针。根据经验, 更深的模型似乎确实在广泛的任务中泛化得更好 (Bengio *et al.*, 2007b;

Erhan *et al.*, 2009; Bengio, 2009; Mesnil *et al.*, 2011; Ciresan *et al.*, 2012; Krizhevsky *et al.*, 2012a; Sermanet *et al.*, 2013; Farabet *et al.*, 2013; Couprie *et al.*, 2013; Kahou *et al.*, 2013; Goodfellow *et al.*, 2014d; Szegedy *et al.*, 2014a)。图 6.6 和图 6.7 展示了一些实验结果的例子。这表明使用深层架构确实在模型学习的函数空间上表示了一个有用的先验。

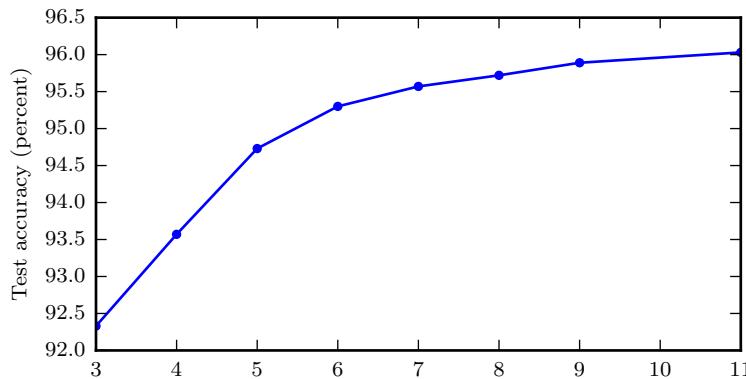


图 6.6: 深度的影响。实验结果表明, 当从地址照片转录多位数字时, 更深层的网络能够更好地泛化。数据来自 Goodfellow *et al.* (2014d)。测试集上的准确率随着深度的增加而不断增加。图 6.7 给出了一个对照实验, 它说明了对模型尺寸其他方面的增加并不能产生相同的效果。

#### 6.4.2 其他架构上的考虑

目前为止, 我们都将神经网络描述成层的简单链式结构, 主要考虑的因素是网络的深度和每层的宽度。在实践中, 神经网络显示出相当的多样性。

许多神经网络架构已经被开发用于特定的任务。用于计算机视觉的卷积神经网络的特殊架构将在第九章中介绍。前馈网络也可以推广到用于序列处理的循环神经网络, 但有它们自己的架构考虑, 将在第十章中介绍。

一般的, 层不需要连接在链中, 尽管这是最常见的做法。许多架构构建了一个主链, 但随后又添加了额外的架构特性, 例如从层  $i$  到层  $i+2$  或者更高层的跳跃连接。这些跳跃连接使得梯度更容易从输出层流向更接近输入的层。

架构设计考虑的另外一个关键点是如何将层与层之间连接起来。默认的神经网络层采用矩阵  $\mathbf{W}$  描述的线性变换, 每个输入单元连接到每个输出单元。在之后章节

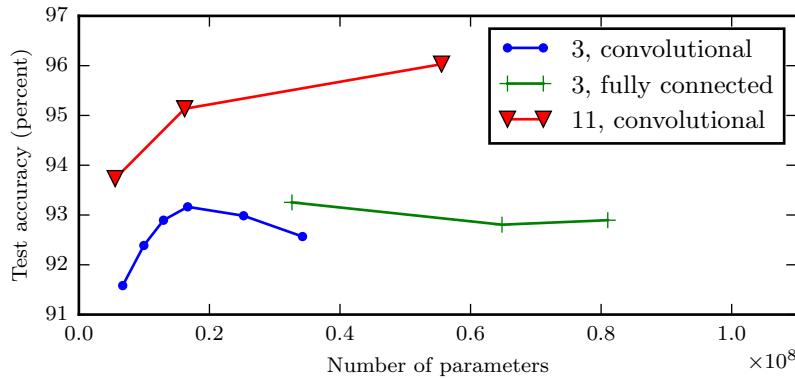


图 6.7: 参数数量的影响。更深的模型往往表现更好。这不仅仅是因为模型更大。Goodfellow *et al.* (2014d) 的这项实验表明, 增加卷积网络层中参数的数量, 但是不增加它们的深度, 在提升测试集性能方面几乎没有效果, 如此图所示。图例标明了用于画出每条曲线的网络深度, 以及曲线表示的是卷积层还是全连接层的大小变化。我们可以观察到, 在这种情况下, 浅层模型在参数数量达到 2000 万时就过拟合, 而深层模型在参数数量超过 6000 万时仍然表现良好。这表明, 使用深层模型表达出了对模型可以学习的函数空间的有用偏好。具体来说, 它表达了一种信念, 即该函数应该由许多更简单的函数复合在一起而得到。这可能导致学习由更简单的表示所组成的表示 (例如, 由边所定义的角) 或者学习具有顺序依赖步骤的程序 (例如, 首先定位一组对象, 然后分割它们, 之后识别它们)。

中的许多专用网络具有较少的连接, 使得输入层中的每个单元仅连接到输出层单元的一个小子集。这些用于减少连接数量的策略减少了参数的数量以及用于评估网络的计算量, 但通常高度依赖于问题。例如, 第九章描述的卷积神经网络使用对于计算机视觉问题非常有效的稀疏连接的专用模式。在这一章中, 很难对通用神经网络的架构给出更多具体的建议。我们在随后的章节中介绍一些特殊的架构策略, 可以在不同的领域工作良好。

## 6.5 反向传播和其他的微分算法

当我们使用前馈神经网络接收输入  $x$  并产生输出  $\hat{y}$  时, 信息通过网络向前流动。输入  $x$  提供初始信息, 然后传播到每一层的隐藏单元, 最终产生输出  $\hat{y}$ 。这称之为 **前向传播** (forward propagation)。在训练过程中, 前向传播可以持续向前直到它产生一个标量代价函数  $J(\theta)$ 。**反向传播** (back propagation) 算法 (Rumelhart

*et al.*, 1986c), 经常简称为**backprop**, 允许来自代价函数的信息通过网络向后流动, 以便计算梯度。

计算梯度的解析表达式是很直观的, 但是数值化地求解这样的表达式在计算上的代价可能很大。反向传播算法使用简单和廉价的程序来实现这个目标。

反向传播这个术语经常被误解为用于多层神经网络的整个学习算法。实际上, 反向传播仅指用于计算梯度的方法, 而另一种算法, 例如随机梯度下降, 使用该梯度来进行学习。此外, 反向传播经常被误解为仅适用于多层神经网络, 但是原则上它可以计算任何函数的导数 (对于一些函数, 正确的响应是报告函数的导数是未定义的)。特别地, 我们会描述如何计算一个任意函数  $f$  的梯度  $\nabla_x f(\mathbf{x}, \mathbf{y})$ , 其中  $\mathbf{x}$  是一组变量, 我们需要它们的导数, 而  $\mathbf{y}$  是函数的另外一组输入变量, 但我们并不需要它们的导数。在学习算法中, 我们最常需要的梯度是代价函数关于参数的梯度, 即  $\nabla_{\theta} J(\theta)$ 。许多机器学习任务需要计算其他导数, 来作为学习过程的一部分, 或者用来分析学得的模型。反向传播算法也适用于这些任务, 不局限于计算代价函数关于参数的梯度。通过在网络中传播信息来计算导数的想法非常普遍, 它还可以用于计算诸如多输出函数  $f$  的 Jacobian 的值。我们这里描述的是最常用的情况, 其中  $f$  只有单个输出。

### 6.5.1 计算图

目前为止, 我们已经用相对非正式的图形语言讨论了神经网络。为了更精确地描述反向传播算法, 使用更精确的**计算图** (computational graph) 语言是很有帮助的。

将计算形式化为图形的方法有很多。

这里, 我们使用图中的每一个节点来表示一个变量。变量可以是标量、向量、矩阵、张量、或者甚至是另一类型的变量。

为了形式化我们的图形, 我们还需引入**操作** (operation) 这一概念。操作是指一个或多个变量的简单函数。我们的图形语言伴随着一组被允许的操作。我们可以通过将多个操作复合在一起描述更为复杂的函数。

不失一般性, 我们定义一个操作仅返回单个输出变量。这并没有失去一般性, 是因为输出变量可以有多个条目, 例如向量。反向传播的软件实现通常支持具有多个输出的操作, 但是我们在描述中避免这种情况, 因为它引入了对概念理解不重要的

许多额外细节。

如果变量  $y$  是变量  $x$  通过一个操作计算得到的，那么我们画一条从  $x$  到  $y$  的有向边。我们有时用操作的名称来注释输出的节点，当上下文很明确时，有时也会省略这个标注。

计算图的实例可以参考图 6.8。

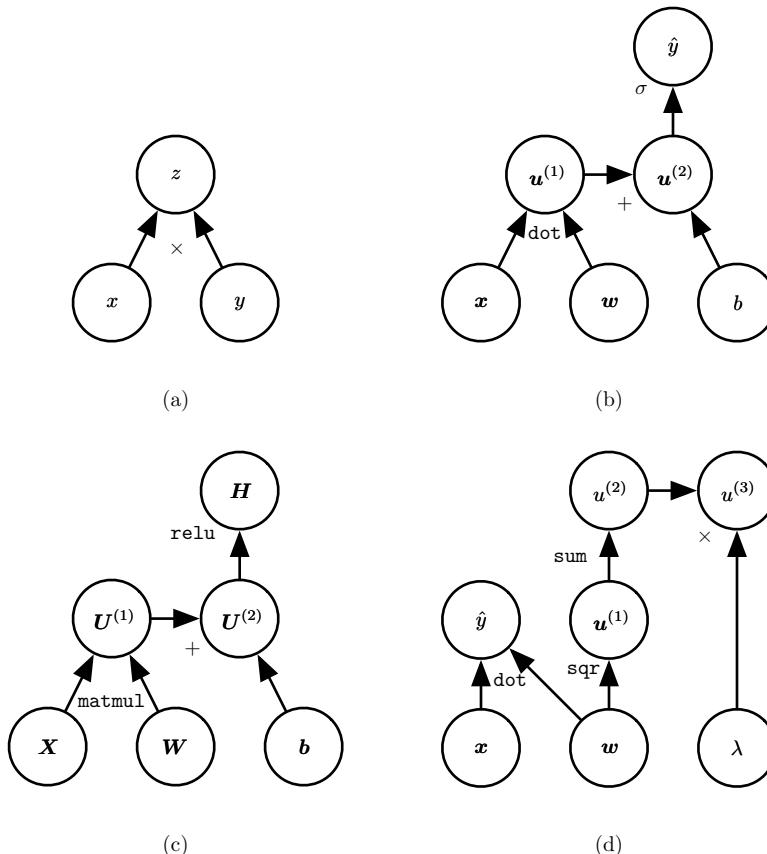


图 6.8: 一些计算图的示例。(a) 使用  $\times$  操作计算  $z = xy$  的图。(b) 用于逻辑回归预测  $\hat{y} = \sigma(\mathbf{x}^\top \mathbf{w} + b)$  的图。一些中间表达式在代数表达式中没有名称，但在图形中却需要。我们简单地将第  $i$  个这样的变量命名为  $u^{(i)}$ 。(c) 表达式  $\mathbf{H} = \max\{0, \mathbf{X}\mathbf{W} + \mathbf{b}\}$  的计算图，在给定包含小批量输入数据的设计矩阵  $\mathbf{X}$  时，它计算整流线性单元激活的设计矩阵  $\mathbf{H}$ 。(d) 示例 a-c 对每个变量最多只实施一个操作，但是对变量实施多个操作也是可能的。这里我们展示一个计算图，它对线性回归模型的权重  $\mathbf{w}$  实施多个操作。这个权重不仅用于预测  $\hat{y}$ ，也用于权重衰减罚项  $\lambda \sum_i w_i^2$ 。

### 6.5.2 微积分中的链式法则

微积分中的链式法则（为了不与概率中的链式法则相混淆）用于计算复合函数的导数。反向传播是一种计算链式法则的算法，使用高效的特定运算顺序。

设  $x$  是实数， $f$  和  $g$  是从实数映射到实数的函数。假设  $y = g(x)$  并且  $z = f(g(x)) = f(y)$ 。那么链式法则是说

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}. \quad (6.44)$$

我们可以将这种标量情况进行扩展。假设  $\mathbf{x} \in \mathbb{R}^m$ ,  $\mathbf{y} \in \mathbb{R}^n$ ,  $g$  是从  $\mathbb{R}^m$  到  $\mathbb{R}^n$  的映射， $f$  是从  $\mathbb{R}^n$  到  $\mathbb{R}$  的映射。如果  $\mathbf{y} = g(\mathbf{x})$  并且  $z = f(\mathbf{y})$ ，那么

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \quad (6.45)$$

使用向量记法，可以等价地写成

$$\nabla_{\mathbf{x}} z = \left( \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^\top \nabla_{\mathbf{y}} z, \quad (6.46)$$

这里  $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$  是  $g$  的  $n \times m$  的 Jacobian 矩阵。

从这里我们看到，变量  $\mathbf{x}$  的梯度可以通过 Jacobian 矩阵  $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$  和梯度  $\nabla_{\mathbf{y}} z$  相乘来得到。反向传播算法由图中每一个这样的 Jacobian 梯度的乘积操作所组成。

通常我们将反向传播算法应用于任意维度的张量，而不仅仅用于向量。从概念上讲，这与使用向量的反向传播完全相同。唯一的区别是如何将数字排列成网格以形成张量。我们可以想象，在我们运行反向传播之前，将每个张量变平为一个向量，计算一个向量值梯度，然后将该梯度重新构造成一个张量。从这种重新排列的观点上看，反向传播仍然只是将 Jacobian 乘以梯度。

为了表示值  $z$  关于张量  $\mathbf{X}$  的梯度，我们记为  $\nabla_{\mathbf{X}} z$ ，就像  $\mathbf{X}$  是向量一样。 $\mathbf{X}$  的索引现在有多个坐标——例如，一个 3 维的张量由三个坐标索引。我们可以通过使用单个变量  $i$  来表示完整的索引元组，从而完全抽象出来。对所有可能的元组  $i$ ,  $(\nabla_{\mathbf{X}} z)_i$  给出  $\frac{\partial z}{\partial X_i}$ 。这与向量中索引的方式完全一致， $(\nabla_{\mathbf{x}} z)_i$  给出  $\frac{\partial z}{\partial x_i}$ 。使用这种记法，我们可以写出适用于张量的链式法则。如果  $\mathbf{Y} = g(\mathbf{X})$  并且  $z = f(\mathbf{Y})$ ，那么

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}. \quad (6.47)$$

### 6.5.3 递归地使用链式法则来实现反向传播

使用链式规则，我们可以直接写出某个标量关于计算图中任何产生该标量的节点的梯度的代数表达式。然而，实际在计算机中计算该表达式时会引入一些额外的考虑。

具体来说，许多子表达式可能在梯度的整个表达式中重复若干次。任何计算梯度的程序都需要选择是存储这些子表达式还是重新计算它们几次。图 6.9 给出了一个例子来说明这些重复的子表达式是如何出现的。在某些情况下，计算两次相同的子表达式纯粹是浪费。在复杂图中，可能存在指数多的这种计算上的浪费，使得简单的链式法则不可实现。在其他情况下，计算两次相同的子表达式可能是以较高的运行时间为代价来减少内存开销的有效手段。

我们首先给出一个版本的反向传播算法，它指明了梯度的直接计算方式（算法 6.2 以及相关的正向计算的算法 6.1），按照它实际完成的顺序并且递归地使用链式法则。我们可以直接执行这些计算或者将算法的描述视为用于计算反向传播的计算图的符号表示。然而，这些公式并没有明确地操作和构造用于计算梯度的符号图。这些公式将在后面的第 6.5.6 节和算法 6.5 中给出，其中我们还推广到了包含任意张量的节点。

首先考虑描述如何计算单个标量  $u^{(n)}$ （例如训练样本上的损失函数）的计算图。我们想要计算这个标量对  $n_i$  个输入节点  $u^{(1)}$  到  $u^{(n_i)}$  的梯度。换句话说，我们希望对所有的  $i \in \{1, 2, \dots, n_i\}$  计算  $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ 。在使用反向传播计算梯度来实现参数的梯度下降时， $u^{(n)}$  将对应单个或者小批量实例的代价函数，而  $u^{(1)}$  到  $u^{(n_i)}$  则对应于模型的参数。

我们假设图的节点已经以一种特殊的方式被排序，使得我们可以一个接一个地计算他们的输出，从  $u^{(n_i+1)}$  开始，一直上升到  $u^{(n)}$ 。如算法 6.1 中所定义的，每个节点  $u^{(i)}$  与操作  $f^{(i)}$  相关联，并且通过对以下函数求值来得到

$$u^{(i)} = f(\mathbb{A}^{(i)}), \quad (6.48)$$

其中  $\mathbb{A}^{(i)}$  是  $u^{(i)}$  所有父节点的集合。

该算法详细说明了前向传播的计算，我们可以将其放入图  $\mathcal{G}$  中。为了执行反向传播，我们可以构造一个依赖于  $\mathcal{G}$  并添加额外一组节点的计算图。这形成了一个子图  $\mathcal{B}$ ，它的每个节点都是  $\mathcal{G}$  的节点。 $\mathcal{B}$  中的计算和  $\mathcal{G}$  中的计算顺序完全相反，而且  $\mathcal{B}$  中的每个节点计算导数  $\frac{\partial u^{(n)}}{\partial u^{(i)}}$  与前向图中的节点  $u^{(i)}$  相关联。这通过对标量输出

---

**算法 6.1** 计算将  $n_i$  个输入  $u^{(1)}$  到  $u^{(n_i)}$  映射到一个输出  $u^{(n)}$  的程序。这定义了一个计算图，其中每个节点通过将函数  $f^{(i)}$  应用到变量集合  $\mathbb{A}^{(i)}$  上来计算  $u^{(i)}$  的值， $\mathbb{A}^{(i)}$  包含先前节点  $u^{(j)}$  的值满足  $j < i$  且  $j \in Pa(u^{(i)})$ 。计算图的输入是向量  $\mathbf{x}$ ，并且被分配给前  $n_i$  个节点  $u^{(1)}$  到  $u^{(n_i)}$ 。计算图的输出可以从最后一个（输出）节点  $u^{(n)}$  读出。

---

```

for  $i = 1, \dots, n_i$  do
     $u^{(i)} \leftarrow x_i$ 
end for
for  $i = n_i + 1, \dots, n$  do
     $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$ 
     $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$ 
end for
return  $u^{(n)}$ 

```

---

$u^{(n)}$  使用链式法则来完成：

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}} \quad (6.49)$$

这在算法 6.2 中详细说明。子图  $\mathcal{B}$  恰好包含每一条对应着  $\mathcal{G}$  中从节点  $u^{(j)}$  到节点  $u^{(i)}$  的边。从  $u^{(j)}$  到  $u^{(i)}$  的边对应着计算  $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ 。另外，对于每个节点都要执行一个内积，内积的一个因子是对于  $u^j$  子节点  $u^{(i)}$  的已经计算的梯度，另一个因子是对于相同子节点  $u^{(i)}$  的偏导数  $\frac{\partial u^{(i)}}{\partial u^{(j)}}$  组成的向量。总而言之，执行反向传播所需的计算量与  $\mathcal{G}$  中的边的数量成比例，其中每条边的计算包括计算偏导数（节点关于它的一个父节点的偏导数）以及执行一次乘法和一次加法。下面，我们将此分析推广到张量值节点，这只是在同一节点中对多个标量值进行分组并能够更高效地实现。

反向传播算法被设计为减少公共子表达式的数量而不考虑存储的开销。具体来说，它大约对图中的每个节点执行一个 Jacobian 乘积。这可以从算法 6.2 中看出，反向传播算法访问了图中的节点  $u^{(j)}$  到节点  $u^{(i)}$  的每条边一次，以获得相关的偏导数  $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ 。反向传播因此避免了重复子表达式的指数爆炸。然而，其他算法可能通过对计算图进行简化来避免更多的子表达式，或者也可能通过重新计算而不是存储这些子表达式来节省内存。我们将在描述完反向传播算法本身后再重新审视这些想法。

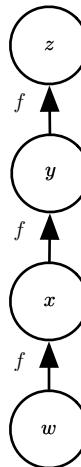


图 6.9: 计算梯度时导致重复子表达式的计算图。令  $w \in \mathbb{R}$  为图的输入。我们对链中的每一步使用相同的操作函数  $f : \mathbb{R} \rightarrow \mathbb{R}$ , 这样  $x = f(w), y = f(x), z = f(y)$ 。为了计算  $\frac{\partial z}{\partial w}$ , 我们应用式(6.44)得到:

$$\frac{\partial z}{\partial w} \quad (6.50)$$

$$= \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial w} \quad (6.51)$$

$$= f'(y) f'(x) f'(w) \quad (6.52)$$

$$= f'(f(f(w))) f'(f(w)) f'(w). \quad (6.53)$$

式(6.52)建议我们采用的实现方式是, 仅计算  $f(w)$  的值一次并将它存储在变量  $x$  中。这是反向传播算法所采用的方法。式(6.53)提出了一种替代方法, 其中子表达式  $f(w)$  出现了不止一次。在替代方法中, 每次只在需要时重新计算  $f(w)$ 。当存储这些表达式的值所需的存储较少时, 式(6.52)的反向传播方法显然是较优的, 因为它减少了运行时间。然而, 式(6.53)也是链式法则的有效实现, 并且当存储受限时它是有用的。

#### 6.5.4 全连接 MLP 中的反向传播计算

为了阐明反向传播的上述定义, 让我们考虑一个与全连接的多层 MLP 相关联的特定图。

算法 6.3 首先给出了前向传播, 它将参数映射到与单个训练样本 (输入, 目标)  $(\mathbf{x}, \mathbf{y})$  相关联的监督损失函数  $L(\hat{\mathbf{y}}, \mathbf{y})$ , 其中  $\hat{\mathbf{y}}$  是当  $\mathbf{x}$  提供输入时神经网络的输出。

算法 6.4 随后说明了将反向传播应用于该图所需的相关计算。

---

**算法 6.2** 反向传播算法的简化版本，用于计算  $u^{(n)}$  关于图中变量的导数。这个示例旨在通过演示所有变量都是标量的简化情况来进一步理解反向传播算法，这里我们希望计算关于  $u^{(1)}, \dots, u^{(n_i)}$  的导数。这个简化版本计算了关于图中所有节点的导数。假定与每条边相关联的偏导数计算需要恒定的时间的话，该算法的计算成本与图中边的数量成比例。这与前向传播的计算次数具有相同的阶。每个  $\frac{\partial u^{(i)}}{\partial u^{(j)}}$  是  $u^{(i)}$  的父节点  $u^{(j)}$  的函数，从而将前向图的节点链接到反向传播图中添加的节点。

---

运行前向传播（对此例是算法 6.1）获得网络的激活。

初始化 `grad_table`，用于存储计算好的导数的数据结构。`grad_table`[ $u^{(i)}$ ] 将存储  $\frac{\partial u^{(n)}}{\partial u^{(i)}}$  计算好的值。

```
grad_table[ $u^{(n)}$ ] ← 1
```

```
for  $j = n - 1$  down to 1 do
```

下一行使用存储的值计算  $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ ：

```
grad_table[ $u^{(j)}$ ] ←  $\sum_{i:j \in Pa(u^{(i)})}$  grad_table[ $u^{(i)}$ ]  $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ 
```

```
end for
```

```
return {grad_table[ $u^{(i)}$ ] |  $i = 1, \dots, n_i$ }
```

---

算法 6.3 和算法 6.4 是简单而直观的演示。然而，它们专门针对特定的问题。

现在的软件实现基于之后第 6.5.6 节中描述的一般形式的反向传播，它可以通过显式地操作表示符号计算的数据结构，来适应任何计算图。

### 6.5.5 符号到符号的导数

代数表达式和计算图都对 **符号**（symbol）或不具有特定值的变量进行操作。这些代数或者基于图的表达式被称为 **符号表示**（symbolic representation）。当我们实际使用或者训练神经网络时，我们必须给这些符号赋特定的值。我们用一个特定的**数值**（numeric value）来替代网络的符号输入  $x$ ，例如  $[1.2, 3, 765, -1.8]^\top$ 。

一些反向传播的方法采用计算图和一组用于图的输入的数值，然后返回在这些输入值处梯度的一组数值。我们将这种方法称为 **符号到数值的微分**。这种方法用在诸如 Torch (Collobert *et al.*, 2011b) 和 Caffe (Jia, 2013) 之类的库中。

另一种方法是采用计算图以及添加一些额外的节点到计算图中，这些额外的节点提供了我们所需导数的符号描述。这是 Theano (Bergstra *et al.*, 2010b; Bastien

---

**算法 6.3** 典型深度神经网络中的前向传播和代价函数的计算。损失函数  $L(\hat{\mathbf{y}}, \mathbf{y})$  取决于输出  $\hat{\mathbf{y}}$  和目标  $\mathbf{y}$  (参考第 6.2.1.1 节中损失函数的示例)。为了获得总代价  $J$ , 损失函数可以加上正则项  $\Omega(\theta)$ , 其中  $\theta$  包含所有参数 (权重和偏置)。算法 6.4 说明了如何计算  $J$  关于参数  $\mathbf{W}$  和  $\mathbf{b}$  的梯度。为简单起见, 该演示仅使用单个输入样本  $\mathbf{x}$ 。实际应用应该使用小批量。请参考第 6.5.7 节以获得更加真实的演示。

---

**Require:** 网络深度,  $l$

**Require:**  $\mathbf{W}^{(i)}, i \in \{1, \dots, l\}$ , 模型的权重矩阵

**Require:**  $\mathbf{b}^{(i)}, i \in \{1, \dots, l\}$ , 模型的偏置参数

**Require:**  $\mathbf{x}$ , 程序的输入

**Require:**  $\mathbf{y}$ , 目标输出

$$\mathbf{h}^{(0)} = \mathbf{x}$$

**for**  $k = 1, \dots, l$  **do**

$$\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}$$

$$\mathbf{h}^{(k)} = f(\mathbf{a}^{(k)})$$

**end for**

$$\hat{\mathbf{y}} = \mathbf{h}^{(l)}$$

$$J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \Omega(\theta)$$


---

*et al.*, 2012b) 和 TensorFlow (Abadi *et al.*, 2015) 所采用的方法。图 6.10 给出了该方法如何工作的一个例子。这种方法的主要优点是导数可以使用与原始表达式相同的语言来描述。因为导数只是另外一张计算图, 我们可以再次运行反向传播, 对导数再进行求导就能得到更高阶的导数。高阶导数的计算在第 6.5.10 节中描述。

我们将使用后一种方法, 并且使用构造导数的计算图的方法来描述反向传播算法。图的任意子集之后都可以使用特定的数值来求值。这允许我们避免精确地指明每个操作应该在何时计算。相反, 通用的图计算引擎只要当一个节点的父节点的值都可用时就可以进行求值。

基于符号到符号的方法的描述包含了符号到数值的方法。符号到数值的方法可以理解为执行了与符号到符号的方法中构建图的过程中完全相同的计算。关键的区别是符号到数值的方法不会显示出计算图。

---

**算法 6.4** 深度神经网络中算法 6.3 的反向计算，它不止使用了输入  $\mathbf{x}$  和目标  $\mathbf{y}$ 。该计算对于每一层  $k$  都产生了对激活  $\mathbf{a}^{(k)}$  的梯度，从输出层开始向后计算一直到第一个隐藏层。这些梯度可以看作是对每层的输出应如何调整以减小误差的指导，根据这些梯度可以获得对每层参数的梯度。权重和偏置上的梯度可以立即用作随机梯度更新的一部分（梯度算出后即可执行更新），或者与其他基于梯度的优化方法一起使用。

---

在前向计算完成后，计算顶层的梯度：

$$\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$$

**for**  $k = l, l - 1, \dots, 1$  **do**

将关于层输出的梯度转换为非线性激活输入前的梯度（如果  $f$  是逐元素的，则逐元素地相乘）：

$$\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(k)}} J = \mathbf{g} \odot f'(\mathbf{a}^{(k)})$$

计算关于权重和偏置的梯度（如果需要的话，还要包括正则项）：

$$\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g} + \lambda \nabla_{\mathbf{b}^{(k)}} \Omega(\theta)$$

$$\nabla_{\mathbf{W}^{(k)}} J = \mathbf{g} \mathbf{h}^{(k-1)\top} + \lambda \nabla_{\mathbf{W}^{(k)}} \Omega(\theta)$$

关于下一更低层的隐藏层传播梯度：

$$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = \mathbf{W}^{(k)\top} \mathbf{g}$$

**end for**

---

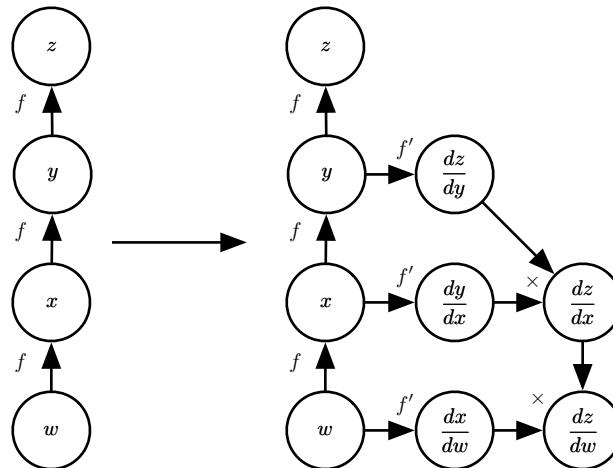


图 6.10: 使用符号到符号的方法计算导数的示例。在这种方法中，反向传播算法不需要访问任何实际的特定数值。相反，它将节点添加到计算图中来描述如何计算这些导数。通用图形求值引擎可以在随后计算任何特定数值的导数。(左) 在这个例子中，我们从表示  $z = f(f(f(w)))$  的图开始。(右) 我们运行反向传播算法，指导它构造表达式  $\frac{dz}{dw}$  对应的图。在这个例子中，我们不解释反向传播算法如何工作。我们的目的只是说明想要的结果是什么：符号描述的导数的计算图。

### 6.5.6 一般化的反向传播

反向传播算法非常简单。为了计算某个标量  $z$  关于图中它的一个祖先  $\mathbf{x}$  的梯度，我们首先观察到它关于  $z$  的梯度由  $\frac{dz}{dz} = 1$  给出。然后，我们可以计算对图中  $z$  的每个父节点的梯度，通过现有的梯度乘以产生  $z$  的操作的 Jacobian。我们继续乘以 Jacobian，以这种方式向后穿过图，直到我们到达  $\mathbf{x}$ 。对于从  $z$  出发可以经过两个或更多路径向后行进而到达的任意节点，我们简单地对该节点来自不同路径上的梯度进行求和。

更正式地，图  $\mathcal{G}$  中的每个节点对应着一个变量。为了实现最大的一般化，我们将这个变量描述为一个张量  $\mathbf{V}$ 。张量通常可以具有任意维度，并且包含标量、向量和矩阵。

我们假设每个变量  $\mathbf{V}$  与下列子程序相关联：

- `get_operation(V)`: 它返回用于计算  $\mathbf{V}$  的操作，代表了在计算图中流入  $\mathbf{V}$  的边。例如，可能有一个 Python 或者 C++ 的类表示矩阵乘法操作，以及 `get_operation` 函数。假设我们的一个变量是由矩阵乘法产生的， $\mathbf{C} = \mathbf{A}\mathbf{B}$ 。

那么，`get_operation(V)` 返回一个指向相应 C++ 类的实例的指针。

- `get_consumers(V, G)`: 它返回一组变量，是计算图  $G$  中  $V$  的子节点。
- `get_inputs(V, G)`: 它返回一组变量，是计算图  $G$  中  $V$  的父节点。

每个操作  $op$  也与 `bprop` 操作相关联。该 `bprop` 操作可以计算如式 (6.47) 所描述的 Jacobian 向量积。这是反向传播算法能够实现很大通用性的原因。每个操作负责了解如何通过它参与的图中的边来反向传播。例如，我们可以使用矩阵乘法操作来产生变量  $C = AB$ 。假设标量  $z$  关于  $C$  的梯度是  $G$ 。矩阵乘法操作负责定义两个反向传播规则，每个规则对应于一个输入变量。如果我们调用 `bprop` 方法来请求关于  $A$  的梯度，那么在给定输出的梯度为  $G$  的情况下，矩阵乘法操作的 `bprop` 方法必须说明关于  $A$  的梯度是  $GB^\top$ 。类似的，如果我们调用 `bprop` 方法来请求关于  $B$  的梯度，那么矩阵操作负责实现 `bprop` 方法并指定希望的梯度是  $A^\top G$ 。反向传播算法本身并不需要知道任何微分法则。它只需要使用正确的参数调用每个操作的 `bprop` 方法即可。正式地，`op.bprop(inputs, X, G)` 必须返回

$$\sum_i (\nabla_{\mathbf{X}} op.f(inputs)_i) G_i, \quad (6.54)$$

这只是如式 (6.47) 所表达的链式法则的实现。这里，`inputs` 是提供给操作的一组输入，`op.f` 是操作实现的数学函数， $\mathbf{X}$  是输入，我们想要计算关于它的梯度， $\mathbf{G}$  是操作对于输出的梯度。

`op.bprop` 方法应该总是假装它的所有输入彼此不同，即使它们不是。例如，如果 `mul` 操作传递两个  $x$  来计算  $x^2$ ，`op.bprop` 方法应该仍然返回  $x$  作为对于两个输入的导数。反向传播算法后面会将这些变量加起来获得  $2x$ ，这是  $x$  上总的正确的导数。

反向传播算法的软件实现通常提供操作和其 `bprop` 方法，所以深度学习软件库的用户能够对使用诸如矩阵乘法、指数运算、对数运算等等常用操作构建的图进行反向传播。构建反向传播新实现的软件工程师或者需要向现有库添加自己的操作的高级用户通常必须手动为新操作推导 `op.bprop` 方法。

反向传播算法的正式描述参考算法 6.5。

在第 6.5.2 节中，我们使用反向传播作为一种策略来避免多次计算链式法则中的相同子表达式。由于这些重复子表达式的存在，简单的算法可能具有指数运行时间。现在我们已经详细说明了反向传播算法，我们可以去理解它的计算成本。如果我们

---

**算法 6.5** 反向传播算法最外围的骨架。这部分做简单的设置和清理工作。大多数重要的工作发生在算法 6.6 的子程序 `build_grad` 中。

---

**Require:**  $\mathbb{T}$ , 需要计算梯度的目标变量集

**Require:**  $\mathcal{G}$ , 计算图

**Require:**  $z$ , 要微分的变量

令  $\mathcal{G}'$  为  $\mathcal{G}$  剪枝后的计算图, 其中仅包括  $z$  的祖先以及  $\mathbb{T}$  中节点的后代。

初始化 `grad_table`, 它是关联张量和对应导数的数据结构。

`grad_table`[ $z$ ]  $\leftarrow 1$

**for**  $\mathbf{V}$  in  $\mathbb{T}$  **do**

`build_grad`( $\mathbf{V}, \mathcal{G}, \mathcal{G}', \text{grad\_table}$ )

**end for**

Return `grad_table` restricted to  $\mathbb{T}$

---

假设每个操作的执行都有大致相同的开销, 那么我们可以依据执行操作的数量来分析计算成本。注意这里我们将一个操作记为计算图的基本单位, 它实际可能包含许多算术运算 (例如, 我们可能将矩阵乘法视为单个操作)。在具有  $n$  个节点的图中计算梯度, 将永远不会执行超过  $O(n^2)$  个操作, 或者存储超过  $O(n^2)$  个操作的输出。这里我们是对计算图中的操作进行计数, 而不是由底层硬件执行的单独操作, 所以重要的是要记住每个操作的运行时间可能是高度可变的。例如, 两个矩阵相乘可能对应着图中的一个单独的操作, 但这两个矩阵可能每个都包含数百万个元素。我们可以看到, 计算梯度至多需要  $O(n^2)$  的操作, 因为在最坏的情况下, 前向传播的步骤将在原始图的全部  $n$  个节点上运行 (取决于我们想要计算的值, 我们可能不需要执行整个图)。反向传播算法在原始图的每条边添加一个 Jacobian 向量积, 可以用  $O(1)$  个节点来表达。因为计算图是有向无环图, 它至多有  $O(n^2)$  条边。对于实践中常用图的类型, 情况会更好。大多数神经网络的代价函数大致是链式结构的, 使得反向传播只有  $O(n)$  的成本。这远远胜过简单的方法, 简单方法可能需要在指数级的节点上运算。这种潜在的指数级代价可以通过非递归地扩展和重写递归链式法则 (式 (6.49)) 来看出:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{path}(u^{(\pi_1)}, u^{(\pi_2)}, \dots, u^{(\pi_t)}), \\ \text{from } \pi_1=j \text{ to } \pi_t=n}} \prod_{k=2}^t \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}. \quad (6.55)$$

由于节点  $j$  到节点  $n$  的路径数目可以关于这些路径的长度上指数地增长, 所以上述

---

**算法 6.6** 反向传播算法的内循环子程序 `build_grad(V, G, G', grad_table)`, 由算法 6.5 中定义的反向传播算法调用。

---

**Require:**  $V$ , 应该被加到  $G$  和 `grad_table` 的变量。

**Require:**  $G$ , 要修改的图。

**Require:**  $G'$ , 根据参与梯度的节点  $G$  的受限图。

**Require:** `grad_table`, 将节点映射到对应梯度的数据结构。

```

if  $V$  is in grad_table then
    Return grad_table[V]
end if
 $i \leftarrow 1$ 
for  $C$  in get_consumers(V, G') do
    op  $\leftarrow$  get_operation(C)
     $D \leftarrow \text{build\_grad}(C, G, G', \text{grad\_table})$ 
     $G^{(i)} \leftarrow op.bprop(\text{get\_inputs}(C, G'), V, D)$ 
     $i \leftarrow i + 1$ 
end for
 $\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$ 
grad_table[V] = G
插入  $\mathbf{G}$  和将其生成到  $G$  中的操作
Return  $\mathbf{G}$ 
```

---

求和符号中的项数（这些路径的数目），可能以前向传播图的深度的指数级增长。会产生如此大的成本是因为对于  $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ ，相同的计算会重复进行很多次。为了避免这种重新计算，我们可以将反向传播看作一种表填充算法，利用存储的中间结果  $\frac{\partial u^{(n)}}{\partial u^{(i)}}$  来对表进行填充。图中的每个节点对应着表中的一个位置，这个位置存储对该节点的梯度。通过顺序填充这些表的条目，反向传播算法避免了重复计算许多公共子表达式。这种表填充策略有时被称为 动态规划 (dynamic programming)。

### 6.5.7 实例：用于 MLP 训练的反向传播

作为一个例子，我们利用反向传播算法来训练多层感知机。

这里，我们考虑一个具有单个隐藏层的非常简单的多层感知机。为了训练这个

模型，我们将使用小批量随机梯度下降算法。反向传播算法用于计算单个小批量上的代价的梯度。具体来说，我们使用训练集上的一小批量实例，将其规范化为一个设计矩阵  $\mathbf{X}$  以及相关联的类标签向量  $\mathbf{y}$ 。网络计算隐藏特征层  $\mathbf{H} = \max\{0, \mathbf{XW}^{(1)}\}$ 。为了简化表示，我们在这个模型中不使用偏置。假设我们的图语言包含 `relu` 操作，该操作可以对  $\max\{0, \mathbf{Z}\}$  表达式的每个元素分别进行计算。类的非归一化对数概率的预测将随后由  $\mathbf{HW}^{(2)}$  给出。假设我们的图语言包含 `cross_entropy` 操作，用以计算目标  $\mathbf{y}$  和由这些未归一化对数概率定义的概率分布间的交叉熵。所得到的交叉熵定义了代价函数  $J_{\text{MLE}}$ 。最小化这个交叉熵将执行对分类器的最大似然估计。然而，为了使得这个例子更加真实，我们也包含一个正则项。总的代价函数为

$$J = J_{\text{MLE}} + \lambda \left( \sum_{i,j} (W_{i,j}^{(1)})^2 + \sum_{i,j} (W_{i,j}^{(2)})^2 \right) \quad (6.56)$$

包含了交叉熵和系数为  $\lambda$  的权重衰减项。它的计算图在图 6.11 中给出。

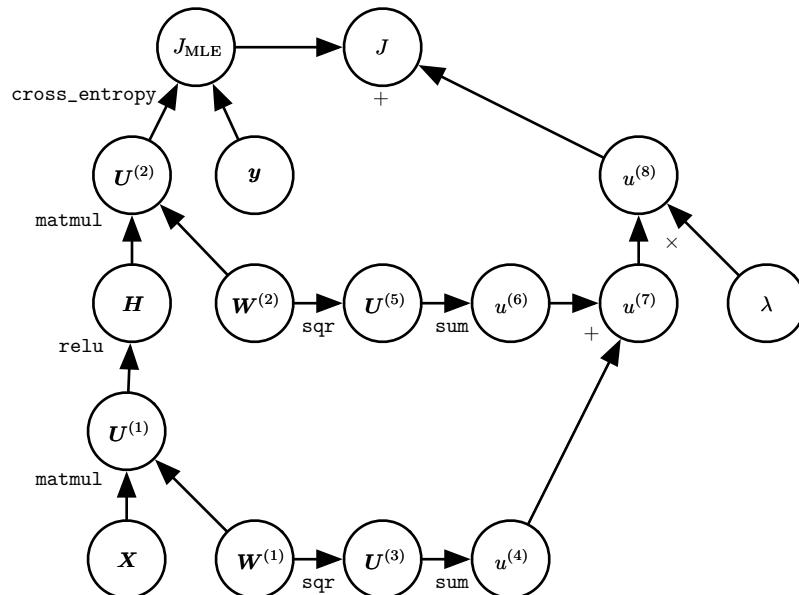


图 6.11: 用于计算代价函数的计算图，这个代价函数是使用交叉熵损失以及权重衰减训练我们的单层 MLP 示例所产生的。

这个示例的梯度计算图实在太大，以致绘制或者阅读都将乏味的。这显示出了反向传播算法的优点之一，即它可以自动生成梯度，而这种计算对于软件工程师

来说需要进行直观但冗长的手动推导。

我们可以通过观察图 6.11 中的正向传播图来粗略地描述反向传播算法的行为。为了训练，我们希望计算  $\nabla_{\mathbf{W}^{(1)}} J$  和  $\nabla_{\mathbf{W}^{(2)}} J$ 。有两种不同的路径从  $J$  后退到权重：一条通过交叉熵代价，另一条通过权重衰减代价。权重衰减代价相对简单，它总是对  $\mathbf{W}^{(i)}$  上的梯度贡献  $2\lambda \mathbf{W}^{(i)}$ 。

另一条通过交叉熵代价的路径稍微复杂一些。令  $\mathbf{G}$  是由 `cross_entropy` 操作提供的对未归一化对数概率  $\mathbf{U}^{(2)}$  的梯度。反向传播算法现在需要探索两个不同的分支。在较短的分支上，它使用对矩阵乘法的第二个变量的反向传播规则，将  $\mathbf{H}^\top \mathbf{G}$  加到  $\mathbf{W}^{(2)}$  的梯度上。另一条更长些的路径沿着网络逐步下降。首先，反向传播算法使用对矩阵乘法的第一个变量的反向传播规则，计算  $\nabla_{\mathbf{H}} J = \mathbf{G} \mathbf{W}^{(2)\top}$ 。接下来，`relu` 操作使用其反向传播规则对先前梯度的部分位置清零，这些位置对应着  $\mathbf{U}^{(1)}$  中所有小于 0 的元素。记上述结果为  $\mathbf{G}'$ 。反向传播算法的最后一步是使用对 `matmul` 操作的第二个变量的反向传播规则，将  $\mathbf{X}^\top \mathbf{G}'$  加到  $\mathbf{W}^{(1)}$  的梯度上。

在计算了这些梯度以后，梯度下降算法或者其他优化算法所要做的就是使用这些梯度来更新参数。

对于 MLP，计算成本主要来源于矩阵乘法。在前向传播阶段，我们乘以每个权重矩阵，得到了  $O(w)$  数量的乘-加，其中  $w$  是权重的数量。在反向传播阶段，我们乘以每个权重矩阵的转置，这具有相同的计算成本。算法主要的存储成本是我们需要将输入存储到隐藏层的非线性中去。这些值从被计算时开始存储，直到反向过程回到了同一点。因此存储成本是  $O(mn_h)$ ，其中  $m$  是小批量中样本的数目， $n_h$  是隐藏单元的数量。

### 6.5.8 复杂化

我们这里描述的反向传播算法要比实践中实际使用的实现要简单。

正如前面提到的，我们将操作的定义限制为返回单个张量的函数。大多数软件实现需要支持可以返回多个张量的操作。例如，如果我们希望计算张量中的最大值和该值的索引，则最好在单次运算中计算两者，因此将该过程实现为具有两个输出的操作效率更高。

我们还没有描述如何控制反向传播的内存消耗。反向传播经常涉及将许多张量加在一起。在朴素方法中，将分别计算这些张量中的每一个，然后在第二步中对所

有些张量求和。朴素方法具有过高的存储瓶颈，可以通过保持一个缓冲器，并且在计算时将每个值加到该缓冲器中来避免该瓶颈。

反向传播的现实实现还需要处理各种数据类型，例如 32 位浮点数、64 位浮点数和整型。处理这些类型的策略需要特别的设计考虑。

一些操作具有未定义的梯度，并且重要的是跟踪这些情况并且确定用户请求的梯度是否是未定义的。

各种其他技术的特性使现实世界的微分更加复杂。这些技术性并不是不可逾越的，本章已经描述了计算微分所需的关键知识工具，但重要的是要知道还有许多的精妙之处存在。

### 6.5.9 深度学习界以外的微分

深度学习界在某种程度上已经与更广泛的计算机科学界隔离开来，并且在很大程度上发展了自己关于如何进行微分的文化态度。更一般地，**自动微分** (automatic differentiation) 领域关心如何以算法方式计算导数。这里描述的反向传播算法只是自动微分的一种方法。它是一种称为**反向模式累加** (reverse mode accumulation) 的更广泛类型的技术的特殊情况。其他方法以不同的顺序来计算链式法则的子表达式。一般来说，确定一种计算的顺序使得计算开销最小，是困难的问题。找到计算梯度的最优操作序列是 NP 完全问题 (Naumann, 2008)，在这种意义上，它可能需要将代数表达式简化为它们最廉价的形式。

例如，假设我们有变量  $p_1, p_2 \dots, p_n$  表示概率，以及变量  $z_1, z_2, \dots, z_n$  表示未归一化的对数概率。假设我们定义

$$q_i = \frac{\exp(z_i)}{\sum_i \exp(z_i)}, \quad (6.57)$$

其中我们通过指数化、求和与除法运算构建 softmax 函数，并构造交叉熵损失函数  $J = -\sum_i p_i \log q_i$ 。人类数学家可以观察到  $J$  对  $z_i$  的导数有一个非常简单的形式： $q_i - p_i$ 。反向传播算法不能够以这种方式来简化梯度，而是会通过原始图中的所有对数和指数操作显式地传播梯度。一些软件库如 Theano (Bergstra *et al.*, 2010b; Bastien *et al.*, 2012b) 能够执行某些种类的代数替换来改进由纯反向传播算法提出的图。

当前向图  $\mathcal{G}$  具有单个输出节点，并且每个偏导数  $\frac{\partial u^{(i)}}{\partial u^{(j)}}$  都可以用恒定的计算量来计算时，反向传播保证梯度计算的计算数目和前向计算的计算数目是同一个量级：

这可以在算法 6.2 中看出，因为每个局部偏导数  $\frac{\partial u^{(i)}}{\partial u^{(j)}}$  以及递归链式公式（式 (6.49)）中相关的乘和加都只需计算一次。因此，总的计算量是  $O(\# \text{edges})$ 。然而，可能通过对反向传播算法构建的计算图进行简化来减少这些计算量，并且这是 NP 完全问题。诸如 Theano 和 TensorFlow 的实现使用基于匹配已知简化模式的试探法，以便重复地尝试去简化图。我们定义反向传播仅用于计算标量输出的梯度，但是反向传播可以扩展到计算 Jacobian 矩阵（该 Jacobian 矩阵或者来源于图中的  $k$  个不同标量节点，或者来源于包含  $k$  个值的张量值节点）。朴素的实现可能需要  $k$  倍的计算：对于原始前向图中的每个内部标量节点，朴素的实现计算  $k$  个梯度而不是单个梯度。当图的输出数目大于输入的数目时，有时更偏向于使用另外一种形式的自动微分，称为 **前向模式累加** (forward mode accumulation)。前向模式计算已经被提出用于循环神经网络梯度的实时计算，例如 (Williams and Zipser, 1989)。这也避免了存储整个图的值和梯度的需要，是计算效率和内存使用的折中。前向模式和后向模式的关系类似于左乘和右乘一系列矩阵之间的关系，例如

$$\mathbf{A}\mathbf{B}\mathbf{C}\mathbf{D}, \quad (6.58)$$

其中的矩阵可以认为是 Jacobian 矩阵。例如，如果  $\mathbf{D}$  是列向量，而  $\mathbf{A}$  有很多行，那么这对应于一幅具有单个输出和多个输入的图，并且从最后开始乘，反向进行，只需要矩阵-向量的乘积。这对应着反向模式。相反，从左边开始乘将涉及一系列的矩阵-矩阵乘积，这使得总的计算变得更加昂贵。然而，如果  $\mathbf{A}$  的行数小于  $\mathbf{D}$  的列数，则从左到右乘更为便宜，这对应着前向模式。

在机器学习以外的许多社区中，更常见的是使用传统的编程语言来直接实现微分软件，例如用 Python 或者 C 来编程，并且自动生成使用这些语言编写的不同函数的程序。在深度学习界中，计算图通常使用由专用库创建的明确的数据结构表示。专用方法的缺点是需要库开发人员为每个操作定义 `bprop` 方法，并且限制了库的用户仅使用定义好的那些操作。然而，专用方法也允许定制每个操作的反向传播规则，允许开发者以非显而易见的方式提高速度或稳定性，对于这种方式自动的过程可能不能复制。

因此，反向传播不是计算梯度的唯一方式或最佳方式，但它是一个非常实用的方法，继续为深度学习社区服务。在未来，深度网络的微分技术可能会提高，因为深度学习的从业者更加懂得了更广泛的自动微分领域的进步。

### 6.5.10 高阶微分

一些软件框架支持使用高阶导数。在深度学习软件框架中，这至少包括 Theano 和 TensorFlow。这些库使用一种数据结构来描述要被微分的原始函数，它们使用相同类型的数据结构来描述这个函数的导数表达式。这意味着符号微分机制可以应用于导数（从而产生高阶导数）。

在深度学习的相关领域，很少会计算标量函数的单个二阶导数。相反，我们通常对 Hessian 矩阵的性质比较感兴趣。如果我们有函数  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ ，那么 Hessian 矩阵的大小是  $n \times n$ 。在典型的深度学习应用中， $n$  将是模型的参数数量，可能很容易达到数十亿。因此，完整的 Hessian 矩阵甚至不能表示。

典型的深度学习方法是使用 **Krylov 方法** (Krylov method)，而不是显式地计算 Hessian 矩阵。Krylov 方法是用于执行各种操作的一组迭代技术，这些操作包括像近似求解矩阵的逆、或者近似矩阵的特征值或特征向量等，而不使用矩阵-向量乘法以外的任何操作。

为了在 Hessian 矩阵上使用 Krylov 方法，我们只需要能够计算 Hessian 矩阵  $\mathbf{H}$  和一个任意向量  $\mathbf{v}$  间的乘积即可。实现这一目标的一种直观方法 (Christianson, 1992) 是

$$\mathbf{H}\mathbf{v} = \nabla_{\mathbf{x}} [(\nabla_{\mathbf{x}} f(\mathbf{x}))^\top \mathbf{v}] . \quad (6.59)$$

该表达式中两个梯度的计算都可以由适当的软件库自动完成。注意，外部梯度表达式是内部梯度表达式的函数的梯度。

如果  $\mathbf{v}$  本身是由计算图产生的一个向量，那么重要的是指定自动微分软件不要对产生  $\mathbf{v}$  的图进行微分。

虽然计算 Hessian 通常是不可取的，但是可以使用 Hessian 向量积。可以对所有的  $i = 1, \dots, n$  简单地计算  $\mathbf{H}\mathbf{e}^{(i)}$ ，其中  $\mathbf{e}^{(i)}$  是  $e_i^{(i)} = 1$  并且其他元素都为 0 的one-hot向量。

## 6.6 历史小记

前馈网络可以被视为一种高效的非线性函数近似器，它以使用梯度下降来最小化函数近似误差为基础。从这个角度来看，现代前馈网络是一般函数近似任务的几个世纪进步的结晶。

处于反向传播算法底层的链式法则是 17 世纪发明的 (Leibniz, 1676; L'Hôpital, 1696)。微积分和代数长期以来被用于求解优化问题的封闭形式，但梯度下降直到 19 世纪才作为优化问题的一种迭代近似的求解方法被引入 (Cauchy, 1847)。

从 20 世纪 40 年代开始，这些函数近似技术被用于导出诸如感知机的机器学习模型。然而，最早的模型都是基于线性模型。来自包括 Marvin Minsky 的批评指出了线性模型族的几个缺陷，例如它无法学习 XOR 函数，这导致了对整个神经网络方法的抵制。

学习非线性函数需要多层感知机的发展和计算该模型梯度的方法。基于动态规划的链式法则的高效应用开始出现在 20 世纪 60 年代和 70 年代，主要用于控制领域 (Kelley, 1960; Bryson and Denham, 1961; Dreyfus, 1962; Bryson and Ho, 1969; Dreyfus, 1973)，也用于灵敏度分析 (Linnainmaa, 1976)。Werbos (1981) 提出应用这些技术来训练人工神经网络。这个想法以不同的方式被独立地重新发现后 (LeCun, 1985; Parker, 1985; Rumelhart *et al.*, 1986a)，最终在实践中得以发展。**并行分布式处理** (Parallel Distributed Processing) 一书在其中一章提供了第一次成功使用反向传播的一些实验的结果 (Rumelhart *et al.*, 1986b)，这对反向传播的普及做出了巨大的贡献，并且开启了一个研究多层神经网络非常活跃的时期。然而，该书作者提出的想法，特别是 Rumelhart 和 Hinton 提出的想法远远超过了反向传播。它们包括一些关键思想，关于可能通过计算实现认知和学习的几个核心方面，后来被冠以“联结主义”的名称，因为它强调了神经元之间的连接作为学习和记忆的轨迹的重要性。特别地，这些想法包括分布式表示的概念 (Hinton *et al.*, 1986)。

在反向传播的成功之后，神经网络研究获得了普及，并在 20 世纪 90 年代初达到高峰。随后，其他机器学习技术变得更受欢迎，直到 2006 年开始的现代深度学习复兴。

现代前馈网络的核心思想自 20 世纪 80 年代以来没有发生重大变化。仍然使用相同的反向传播算法和相同的梯度下降方法。1986 年至 2015 年神经网络性能的大部分改进可归因于两个因素。首先，较大的数据集减少了统计泛化对神经网络的挑战的程度。第二，神经网络由于更强大的计算机和更好的软件基础设施已经变得更大。然而，少量算法上的变化也显著改善了神经网络的性能。

其中一个算法上的变化是用交叉熵族损失函数替代均方误差损失函数。均方误差在 20 世纪 80 年代和 90 年代流行，但逐渐被交叉熵损失替代，并且最大似然原理的想法在统计学界和机器学习界之间广泛传播。使用交叉熵损失大大提高了具

有 sigmoid 和 softmax 输出的模型的性能，而当使用均方误差损失时会存在饱和和学习缓慢的问题。

另一个显著改善前馈网络性能的算法上的主要变化是使用分段线性隐藏单元来替代 sigmoid 隐藏单元，例如用整流线性单元。使用  $\max\{0, z\}$  函数的整流在早期神经网络中已经被引入，并且至少可以追溯到认知机（Cognitron）和神经认知机（Neocognitron）(Fukushima, 1975, 1980)。这些早期的模型没有使用整流线性单元，而是将整流用于非线性函数。尽管整流在早期很普及，在 20 世纪 80 年代，整流很大程度上被 sigmoid 所取代，也许是因为当神经网络非常小时，sigmoid 表现更好。到 21 世纪初，由于有些迷信的观念，认为必须避免具有不可导点的激活函数，所以避免了整流线性单元。这在 2009 年开始发生改变。Jarrett *et al.* (2009b) 观察到，在神经网络结构设计的几个不同因素中“使用整流非线性是提高识别系统性能的最重要的唯一因素”。

对于小的数据集，Jarrett *et al.* (2009b) 观察到，使用整流非线性甚至比学习隐藏层的权重值更加重要。随机的权重足以通过整流网络传播有用的信息，允许在顶部的分类器层学习如何将不同的特征向量映射到类标识。

当有更多数据可用时，学习开始提取足够的有用知识来超越随机选择参数的性能。Glorot *et al.* (2011a) 说明，在深度整流网络中的学习比在激活函数具有曲率或两侧饱和的深度网络中的学习更容易。

整流线性单元还具有历史意义，因为它们表明神经科学继续对深度学习算法的发展产生影响。Glorot *et al.* (2011a) 从生物学考虑整流线性单元的导出。半整流非线性旨在描述生物神经元的这些性质：(1) 对于某些输入，生物神经元是完全不活跃的。(2) 对于某些输入，生物神经元的输出和它的输入成比例。(3) 大多数时间，生物神经元是在它们不活跃的状态下进行操作（即它们应该具有稀疏激活（sparse activation））。

当 2006 年深度学习开始现代复兴时，前馈网络仍然有不良的声誉。从 2006 年至 2012 年，人们普遍认为，前馈网络不会表现良好，除非它们得到其他模型的辅助，例如概率模型。现在已经知道，只要具备适当的资源和工程实践，前馈网络表现得非常好。今天，前馈网络中基于梯度的学习被用作发展概率模型的工具，例如第二十章中描述的变分自编码器和生成式对抗网络。前馈网络中基于梯度的学习自 2012 年以来一直被视为一种强大的技术，并应用于许多其他机器学习任务，而不是被视为必须由其他技术支持的不可靠技术。在 2006 年，业内使用无监督学习来支持监督

学习，现在更讽刺的是，更常见的是使用监督学习来支持无监督学习。

前馈网络还有许多未实现的潜力。未来，我们期望它们用于更多的任务，优化算法和模型设计的进步将进一步提高它们的性能。本章主要描述了神经网络模型族。在接下来的章节中，我们将讨论如何使用这些模型——如何对它们进行正则化和训练。

# 第七章 深度学习中的正则化

机器学习中的一个核心问题是设计不仅在训练数据上表现好，并且能在新输入上泛化好的算法。在机器学习中，许多策略显式地被设计来减少测试误差（可能会以增大训练误差为代价）。这些策略被统称为正则化。我们将在后文看到，深度学习工作者可以使用许多不同形式的正则化策略。事实上，开发更有效的正则化策略已成为本领域的主要研究工作之一。

第五章介绍了泛化、欠拟合、过拟合、偏差、方差和正则化的基本概念。如果你不熟悉这些概念，请参考该章节再继续阅读本章。

在本章中，我们会更详细地介绍正则化，重点介绍深度模型（或组成深度模型的模块）的正则化策略。

本章中的某些章节涉及机器学习中的标准概念。如果你已经熟悉了这些概念，可以随意跳过相关章节。然而，本章的大多数内容是关于这些基本概念在特定神经网络中的扩展概念。

在第 5.2.2 节中，我们将正则化定义为“对学习算法的修改——旨在减少泛化误差而不是训练误差”。目前有许多正则化策略。有些策略向机器学习模型添加限制参数值的额外约束。有些策略向目标函数增加额外项来对参数值进行软约束。如果我们细心选择，这些额外的约束和惩罚可以改善模型在测试集上的表现。有时候，这些约束和惩罚被设计为编码特定类型的先验知识；其他时候，这些约束和惩罚被设计为偏好简单模型，以便提高泛化能力。有时，惩罚和约束对于确定欠定的问题是必要的。其他形式的正则化，如被称为集成的方法，则结合多个假说来解释训练数据。

在深度学习的背景下，大多数正则化策略都会对估计进行正则化。估计的正则化以偏差的增加换取方差的减少。一个有效的正则化是有利的“交易”，也就是能显著减少方差而不过度增加偏差。我们在第五章中讨论泛化和过拟合时，主要侧重模

型族训练的 3 个情形：(1) 不包括真实的数据生成过程——对应欠拟合和含有偏差的情况，(2) 匹配真实数据生成过程，(3) 除了包括真实的数据生成过程，还包括许多其他可能的生成过程——方差（而不是偏差）主导的过拟合。正则化的目标是使模型从第三种情况转化为第二种情况。

在实践中，过于复杂的模型族不一定包括目标函数或真实数据生成过程，甚至也不包括近似过程。我们几乎从未知晓真实数据的生成过程，所以我们永远不知道被估计的模型族是否包括生成过程。然而，深度学习算法的大多数应用都是针对这样的情况，其中真实数据的生成过程几乎肯定在模型族之外。深度学习算法通常应用于极为复杂的领域，如图像、音频序列和文本，本质上这些领域的生成过程涉及模拟整个宇宙。从某种程度上说，我们总是持方枘（数据生成过程）而欲内圆凿（我们的模型族）。

这意味着控制模型的复杂度不是找到合适规模的模型（带有正确的参数个数）这样一个简单的事情。相反，我们可能会发现，或者说在实际的深度学习场景中我们几乎总是会发现，最好的拟合模型（从最小化泛化误差的意义上）是一个适当正则化的大型模型。

现在我们回顾几种策略，以创建这些正则化的大型深度模型。

## 7.1 参数范数惩罚

正则化在深度学习的出现前就已经被使用了数十年。线性模型，如线性回归和逻辑回归可以使用简单、直接、有效的正则化策略。

许多正则化方法通过对目标函数  $J$  添加一个参数范数惩罚项  $\Omega(\theta)$ ，限制模型（如神经网络、线性回归或逻辑回归）的学习能力。我们将正则化后的目标函数记为  $\tilde{J}$ ：

$$\tilde{J}(\theta; \mathbf{X}, \mathbf{y}) = J(\theta; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\theta), \quad (7.1)$$

其中  $\alpha \in [0, \infty)$  是权衡范数惩罚项  $\Omega$  和标准目标函数  $J(\mathbf{X}; \theta)$  相对贡献的超参数。将  $\alpha$  设为 0 表示没有正则化。 $\alpha$  越大，对应正则化惩罚越大。

当我们的训练算法最小化正则化后的目标函数  $\tilde{J}$  时，它会降低原始目标  $J$  关于训练数据的误差并同时减小在某些衡量标准下参数  $\theta$ （或参数子集）的规模。选择不同的参数范数  $\Omega$  会偏好不同的解。在本节中，我们会讨论各种范数惩罚对模型的

影响。

在探究不同范数的正则化表现之前，我们需要说明一下，在神经网络中，参数包括每一层仿射变换的权重和偏置，我们通常只对权重做惩罚而不对偏置做正则惩罚。精确拟合偏置所需的数据通常比拟合权重少得多。每个权重会指定两个变量如何相互作用。我们需要在各种条件下观察这两个变量才能良好地拟合权重。而每个偏置仅控制一个单变量。这意味着，我们不对其进行正则化也不会导致太大的方差。另外，正则化偏置参数可能会导致明显的欠拟合。因此，我们使用向量  $\mathbf{w}$  表示所有应受范数惩罚影响的权重，而向量  $\boldsymbol{\theta}$  表示所有参数（包括  $\mathbf{w}$  和无需正则化的参数）。

在神经网络的情况下，有时希望对网络的每一层使用单独的惩罚，并分配不同的  $\alpha$  系数。寻找合适的多个超参数的代价很大，因此为了减少搜索空间，我们会在所有层使用相同的权重衰减。

### 7.1.1 $L^2$ 参数正则化

在第 5.2 节中我们已经看到过最简单而又最常见的参数范数惩罚，即通常被称为权重衰减（weight decay）的  $L^2$  参数范数惩罚。这个正则化策略通过向目标函数添加一个正则项  $\Omega(\boldsymbol{\theta}) = \frac{1}{2} \|\mathbf{w}\|_2^2$ ，使权重更加接近原点<sup>1</sup>。在其他学术圈， $L^2$  也被称为岭回归或 Tikhonov 正则。

我们可以通过研究正则化后目标函数的梯度，洞察一些权重衰减的正则化表现。为了简单起见，我们假定其中没有偏置参数，因此  $\boldsymbol{\theta}$  就是  $\mathbf{w}$ 。这样一个模型具有以下总的目标函数：

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2} \mathbf{w}^\top \mathbf{w} + J(\mathbf{w}; \mathbf{X}, \mathbf{y}), \quad (7.2)$$

与之对应的梯度为

$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \mathbf{w} + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}). \quad (7.3)$$

使用单步梯度下降更新权重，即执行以下更新：

$$\mathbf{w} \leftarrow \mathbf{w} - \epsilon (\alpha \mathbf{w} + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y})). \quad (7.4)$$

---

<sup>1</sup>更一般地，我们可以将参数正则化为接近空间中的任意特定点，令人惊讶的是这样也仍有正则化效果，但是特定点越接近真实值结果越好。当我们不知道正确的值应该是正还是负时，零是有意义的默认值。由于模型参数正则化为零的情况更为常见，我们将只探讨这种特殊情况。

换种写法就是：

$$\mathbf{w} \leftarrow (1 - \epsilon\alpha)\mathbf{w} - \epsilon\nabla_{\mathbf{w}}J(\mathbf{w}; \mathbf{X}, \mathbf{y}). \quad (7.5)$$

我们可以看到，加入权重衰减后会引起学习规则的修改，即在每步执行通常的梯度更新之前先收缩权重向量（将权重向量乘以一个常数因子）。这是单个步骤发生的变化。但是，在训练的整个过程会发生什么呢？

我们进一步简化分析，令  $\mathbf{w}^*$  为未正则化的目标函数取得最小训练误差时的权重向量，即  $\mathbf{w}^* = \arg \min_{\mathbf{w}} J(\mathbf{w})$ ，并在  $\mathbf{w}^*$  的邻域对目标函数做二次近似。如果目标函数确实是二次的（如以均方误差拟合线性回归模型的情况），则该近似是完美的。近似的  $\hat{J}(\boldsymbol{\theta})$  如下

$$\hat{J}(\boldsymbol{\theta}) = J(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^*), \quad (7.6)$$

其中  $\mathbf{H}$  是  $J$  在  $\mathbf{w}^*$  处计算的 Hessian 矩阵（关于  $\mathbf{w}$ ）。因为  $\mathbf{w}^*$  被定义为最优，即梯度消失为 0，所以该二次近似中没有一阶项。同样地，因为  $\mathbf{w}^*$  是  $J$  的一个最优点，我们可以得出  $\mathbf{H}$  是半正定的结论。

当  $\hat{J}$  取得最小时，其梯度

$$\nabla_{\mathbf{w}}\hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*) \quad (7.7)$$

为 0。

为了研究权重衰减带来的影响，我们在式 (7.7) 中添加权重衰减的梯度。现在我们探讨最小化正则化后的  $\hat{J}$ 。我们使用变量  $\tilde{\mathbf{w}}$  表示此时的最优点：

$$\alpha\tilde{\mathbf{w}} + \mathbf{H}(\tilde{\mathbf{w}} - \mathbf{w}^*) = 0 \quad (7.8)$$

$$(\mathbf{H} + \alpha\mathbf{I})\tilde{\mathbf{w}} = \mathbf{H}\mathbf{w}^* \quad (7.9)$$

$$\tilde{\mathbf{w}} = (\mathbf{H} + \alpha\mathbf{I})^{-1}\mathbf{H}\mathbf{w}^* \quad (7.10)$$

当  $\alpha$  趋向于 0 时，正则化的解  $\tilde{\mathbf{w}}$  会趋向  $\mathbf{w}^*$ 。那么当  $\alpha$  增加时会发生什么呢？因为  $\mathbf{H}$  是实对称的，所以我们可以将其分解为一个对角矩阵  $\Lambda$  和一组特征向量的标准正交基  $\mathbf{Q}$ ，并且有  $\mathbf{H} = \mathbf{Q}\Lambda\mathbf{Q}^\top$ 。将其应用于式 (7.10)，可得：

$$\tilde{\mathbf{w}} = (\mathbf{Q}\Lambda\mathbf{Q}^\top + \alpha\mathbf{I})^{-1}\mathbf{Q}\Lambda\mathbf{Q}^\top\mathbf{w}^* \quad (7.11)$$

$$= [\mathbf{Q}(\Lambda + \alpha\mathbf{I})\mathbf{Q}^\top]^{-1}\mathbf{Q}\Lambda\mathbf{Q}^\top\mathbf{w}^* \quad (7.12)$$

$$= \mathbf{Q}(\Lambda + \alpha\mathbf{I})^{-1}\Lambda\mathbf{Q}^\top\mathbf{w}^*. \quad (7.13)$$

我们可以看到权重衰减的效果是沿着由  $\mathbf{H}$  的特征向量所定义的轴缩放  $\mathbf{w}^*$ 。具体来说，我们会根据  $\frac{\lambda_i}{\lambda_i + \alpha}$  因子缩放与  $\mathbf{H}$  第  $i$  个特征向量对齐的  $\mathbf{w}^*$  的分量。（不妨查看图 2.3 回顾这种缩放的原理）。

沿着  $\mathbf{H}$  特征值较大的方向（如  $\lambda_i \gg \alpha$ ）正则化的影响较小。而  $\lambda_i \ll \alpha$  的分量将会收缩到几乎为零。这种效应如图 7.1 所示。

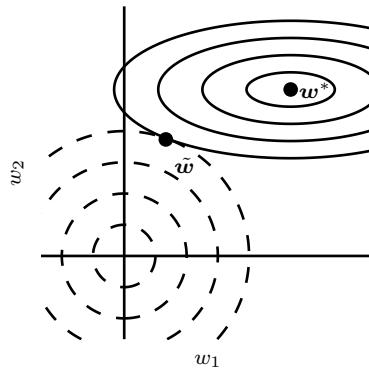


图 7.1:  $L^2$ （或权重衰减）正则化对最佳  $\mathbf{w}$  值的影响。实线椭圆表示没有正则化目标的等值线。虚线圆圈表示  $L^2$  正则化项的等值线。在  $\tilde{\mathbf{w}}$  点，这两个竞争目标达到平衡。目标函数  $J$  的 Hessian 的第一维特征值很小。当从  $\mathbf{w}^*$  水平移动时，目标函数不会增加得太多。因为目标函数对这个方向没有强烈的偏好，所以正则化项对该轴具有强烈的影响。正则化项将  $w_1$  拉向零。而目标函数对沿着第二维远离  $\mathbf{w}^*$  的移动非常敏感。对应的特征值较大，表示高曲率。因此，权重衰减对  $w_2$  的位置影响相对较小。

只有在显著减小目标函数方向上的参数会保留得相对完好。在无助于目标函数减小的方向（对应 Hessian 矩阵较小的特征值）上改变参数不会显著增加梯度。这种不重要方向对应的分量会在训练过程中因正则化而衰减掉。

目前为止，我们讨论了权重衰减对优化一个抽象通用的二次代价函数的影响。这些影响具体是怎么和机器学习关联的呢？我们可以研究线性回归，它的真实代价函数是二次的，因此我们可以使用相同的方法分析。再次应用分析，我们会在这种情况下得到相同的结果，但这次我们使用训练数据的术语表述。线性回归的代价函数是平方误差之和：

$$(\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}). \quad (7.14)$$

我们添加  $L^2$  正则项后，目标函数变为

$$(\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{1}{2}\alpha\mathbf{w}^\top \mathbf{w}. \quad (7.15)$$

这将普通方程的解从

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \quad (7.16)$$

变为

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}. \quad (7.17)$$

式(7.16)中的矩阵  $\mathbf{X}^\top \mathbf{X}$  与协方差矩阵  $\frac{1}{m} \mathbf{X}^\top \mathbf{X}$  成正比。 $L^2$  正则项将这个矩阵替换为式(7.17)中的  $(\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1}$  这个新矩阵与原来的是一样的，不同的仅仅是在对角加了  $\alpha$ 。这个矩阵的对角项对应每个输入特征的方差。我们可以看到， $L^2$  正则化能让学习算法“感知”到具有较高方差的输入  $\mathbf{x}$ ，因此与输出目标的协方差较小（相对增加方差）的特征的权重将会收缩。

### 7.1.2 $L^1$ 参数正则化

$L^2$  权重衰减是权重衰减最常见的形式，我们还可以使用其他的方法限制模型参数的规模。一个选择是使用  $L^1$  正则化。

形式地，对模型参数  $\mathbf{w}$  的  $L^1$  正则化被定义为：

$$\Omega(\boldsymbol{\theta}) = \|\mathbf{w}\|_1 = \sum_i |w_i|, \quad (7.18)$$

即各个参数的绝对值之和<sup>2</sup>。接着我们将讨论  $L^1$  正则化对简单线性回归模型的影响，与分析  $L^2$  正则化时一样不考虑偏置参数。我们尤其感兴趣的是找出  $L^1$  和  $L^2$  正则化之间的差异。与  $L^2$  权重衰减类似，我们也可以通过缩放惩罚项  $\Omega$  的正超参数  $\alpha$  来控制  $L^1$  权重衰减的强度。因此，正则化的目标函数  $\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y})$  如下所示

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \|\mathbf{w}\|_1 + J(\mathbf{w}; \mathbf{X}, \mathbf{y}), \quad (7.19)$$

对应的梯度（实际上是次梯度）：

$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha \text{sign}(\mathbf{w}) + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}), \quad (7.20)$$

---

<sup>2</sup>如同  $L^2$  正则化，我们能将参数正则化到其他非零值  $\mathbf{w}^{(o)}$ 。在这种情况下， $L^1$  正则化将会引入不同的项  $\Omega(\boldsymbol{\theta}) = \|\mathbf{w} - \mathbf{w}^{(o)}\|_1 = \sum_i |w_i - w_i^{(o)}|$ 。

其中  $\text{sign}(\mathbf{w})$  只是简单地取  $\mathbf{w}$  各个元素的正负号。

观察式(7.20)，我们立刻发现  $L^1$  的正则化效果与  $L^2$  大不一样。具体来说，我们可以看到正则化对梯度的影响不再是线性地缩放每个  $w_i$ ；而是添加了一项与  $\text{sign}(w_i)$  同号的常数。使用这种形式的梯度之后，我们不一定能得到  $J(\mathbf{X}, \mathbf{y}; \mathbf{w})$  二次近似的直接算术解 ( $L^2$  正则化时可以)。

简单线性模型具有二次代价函数，我们可以通过泰勒级数表示。或者我们可以设想，这是逼近更复杂模型的代价函数的截断泰勒级数。在这个设定下，梯度由下式给出

$$\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*), \quad (7.21)$$

同样， $\mathbf{H}$  是  $J$  在  $\mathbf{w}^*$  处的Hessian矩阵 (关于  $\mathbf{w}$ )。

由于  $L^1$  惩罚项在完全一般化的 Hessian 的情况下，无法得到直接清晰的代数表达式，因此我们将进一步简化假设 Hessian 是对角的，即  $\mathbf{H} = \text{diag}([H_{1,1}, \dots, H_{n,n}])$ ，其中每个  $H_{i,i} > 0$ 。如果线性回归问题中的数据已被预处理 (如可以使用 PCA)，去除了输入特征之间的相关性，那么这一假设成立。

我们可以将  $L^1$  正则化目标函数的二次近似分解成关于参数的求和：

$$\hat{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = J(\mathbf{w}^*; \mathbf{X}, \mathbf{y}) + \sum_i \left[ \frac{1}{2} H_{i,i} (w_i - w_i^*)^2 + \alpha |w_i| \right]. \quad (7.22)$$

如下列形式的解析解 (对每一维  $i$ ) 可以最小化这个近似代价函数：

$$w_i = \text{sign}(w_i^*) \max \left\{ |w_i^*| - \frac{\alpha}{H_{i,i}}, 0 \right\}. \quad (7.23)$$

对每个  $i$ ，考虑  $w_i^* > 0$  的情形，会有两种可能结果：

1.  $w_i^* \leq \frac{\alpha}{H_{i,i}}$  的情况。正则化后目标中的  $w_i$  最优值是  $w_i = 0$ 。这是因为在方向  $i$  上  $J(\mathbf{w}; \mathbf{X}, \mathbf{y})$  对  $\hat{J}(\mathbf{w}; \mathbf{X}, \mathbf{y})$  的贡献被抵消， $L^1$  正则化项将  $w_i$  推至 0。
2.  $w_i^* > \frac{\alpha}{H_{i,i}}$  的情况。在这种情况下，正则化不会将  $w_i$  的最优值推至 0，而仅仅在那个方向上移动  $\frac{\alpha}{H_{i,i}}$  的距离。

$w_i^* < 0$  的情况与之类似，但是  $L^1$  惩罚项使  $w_i$  更接近 0 (增加  $\frac{\alpha}{H_{i,i}}$ ) 或者为 0。

相比  $L^2$  正则化， $L^1$  正则化会产生更稀疏 (sparse) 的解。此处稀疏性指的是最优值中的一些参数为 0。和  $L^2$  正则化相比， $L^1$  正则化的稀疏性具有本质的不同。

式(7.13)给出了 $L^2$ 正则化的解 $\tilde{\mathbf{w}}$ 。如果我们使用Hessian矩阵 $\mathbf{H}$ 为对角正定矩阵的假设(与 $L^1$ 正则化分析时一样),重新考虑这个等式,我们发现 $\tilde{w}_i = \frac{H_{i,i}}{H_{i,i} + \alpha} w_i^*$ 。如果 $w_i^*$ 不是零,那么 $\tilde{w}_i$ 也会保持非零。这表明 $L^2$ 正则化不会使参数变得稀疏,而 $L^1$ 正则化有可能通过足够大的 $\alpha$ 实现稀疏。

由 $L^1$ 正则化导出的稀疏性质已经被广泛地用于特征选择(feature selection)机制。特征选择从可用的特征子集选择出有意义的特征,化简机器学习问题。著名的LASSO(Tibshirani, 1995)(Least Absolute Shrinkage and Selection Operator)模型将 $L^1$ 惩罚和线性模型结合,并使用最小二乘代价函数。 $L^1$ 惩罚使部分子集的权重为零,表明相应的特征可以被安全地忽略。

在第5.6.1节,我们看到许多正则化策略可以被解释为MAP贝叶斯推断,特别是 $L^2$ 正则化相当于权重是高斯先验的MAP贝叶斯推断。对于 $L^1$ 正则化,用于正则化代价函数的惩罚项 $\alpha\Omega(\mathbf{w}) = \alpha\sum_i |w_i|$ 与通过MAP贝叶斯推断最大化的对数先验项是等价的( $\mathbf{w} \in \mathbb{R}^n$ 并且权重先验是各向同性的拉普拉斯分布(式(3.26))):

$$\log p(\mathbf{w}) = \sum_i \log \text{Laplace}(w_i; 0, \frac{1}{\alpha}) = -\alpha \|\mathbf{w}\|_1 + n \log \alpha - n \log 2. \quad (7.24)$$

因为是关于 $\mathbf{w}$ 最大化进行学习,我们可以忽略 $\log \alpha - \log 2$ 项,因为它们与 $\mathbf{w}$ 无关。

## 7.2 作为约束的范数惩罚

考虑经过参数范数正则化的代价函数:

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta}). \quad (7.25)$$

回顾第4.4节我们可以构造一个广义Lagrange函数来最小化带约束的函数,即在原始目标函数上添加一系列惩罚项。每个惩罚是一个被称为**Karush-Kuhn-Tucker**(Karush-Kuhn-Tucker)乘子的系数以及一个表示约束是否满足的函数之间的乘积。如果我们想约束 $\Omega(\boldsymbol{\theta})$ 小于某个常数 $k$ ,我们可以构建广义Lagrange函数

$$\mathcal{L}(\boldsymbol{\theta}, \alpha; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha(\Omega(\boldsymbol{\theta}) - k). \quad (7.26)$$

这个约束问题的解由下式给出

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \max_{\alpha, \alpha \geq 0} \mathcal{L}(\boldsymbol{\theta}, \alpha). \quad (7.27)$$

如第 4.4 节中描述的，解决这个问题我们需要对  $\theta$  和  $\alpha$  都做出调整。第 4.5 节给出了一个带  $L^2$  约束的线性回归实例。还有许多不同的优化方法，有些可能会使用梯度下降而其他可能会使用梯度为 0 的解析解，但在所有过程中  $\alpha$  在  $\Omega(\theta) > k$  时必须增加，在  $\Omega(\theta) < k$  时必须减小。所有正值的  $\alpha$  都鼓励  $\Omega(\theta)$  收缩。最优值  $\alpha^*$  也将鼓励  $\Omega(\theta)$  收缩，但不会强到使得  $\Omega(\theta)$  小于  $k$ 。

为了洞察约束的影响，我们可以固定  $\alpha^*$ ，把这个问题看成只跟  $\theta$  有关的函数：

$$\theta^* = \arg \min_{\theta} \mathcal{L}(\theta, \alpha^*) = \arg \min_{\theta} J(\theta; X, y) + \alpha^* \Omega(\theta). \quad (7.28)$$

这和最小化  $\tilde{J}$  的正则化训练问题是完全一样的。因此，我们可以把参数范数惩罚看作对权重强加的约束。如果  $\Omega$  是  $L^2$  范数，那么权重就是被约束在一个  $L^2$  球中。如果  $\Omega$  是  $L^1$  范数，那么权重就是被约束在一个  $L^1$  范数限制的区域中。通常我们不知道权重衰减系数  $\alpha^*$  约束的区域大小，因为  $\alpha^*$  的值不直接告诉我们  $k$  的值。原则上我们可以解得  $k$ ，但  $k$  和  $\alpha^*$  之间的关系取决于  $J$  的形式。虽然我们不知道约束区域的确切大小，但我们可以通过增加或者减小  $\alpha$  来大致扩大或收缩约束区域。较大的  $\alpha$ ，将得到一个较小的约束区域。较小的  $\alpha$ ，将得到一个较大的约束区域。

有时候，我们希望使用显式的限制，而不是惩罚。如第 4.4 节所述，我们可以修改下降算法（如随机梯度下降算法），使其先计算  $J(\theta)$  的下降步，然后将  $\theta$  投影到满足  $\Omega(\theta) < k$  的最近点。如果我们知道什么样的  $k$  是合适的，而不想花时间寻找对应于此  $k$  处的  $\alpha$  值，这会非常有用。

另一个使用显式约束和重投影而不是使用惩罚强加约束的原因是惩罚可能会导致目标函数非凸而使算法陷入局部极小（对应于小的  $\theta$ ）。当训练神经网络时，这通常表现为训练带有几个“死亡单元”的神经网络。这些单元不会对网络学到的函数有太大影响，因为进入或离开它们的权重都非常小。当使用权重范数的惩罚训练时，即使可以通过增加权重以显著减少  $J$ ，这些配置也可能是局部最优的。因为重投影实现的显式约束不鼓励权重接近原点，所以在这些情况下效果更好。通过重投影实现的显式约束只在权重变大并试图离开限制区域时产生作用。

最后，因为重投影的显式约束还对优化过程增加了一定的稳定性，所以这是另一个好处。当使用较高的学习率时，很可能进入正反馈，即大的权重诱导大梯度，然后使得权重获得较大更新。如果这些更新持续增加权重的大小， $\theta$  就会迅速增大，直到离原点很远而发生溢出。重投影的显式约束可以防止这种反馈环引起权重无限制地持续增加。Hinton *et al.* (2012c) 建议结合使用约束和高学习速率，这样能更快地探索参数空间，并保持一定的稳定性。

Hinton *et al.* (2012c) 尤其推荐由Srebro and Shraibman (2005) 引入的策略：约束神经网络层的权重矩阵每列的范数，而不是限制整个权重矩阵的 Frobenius 范数。分别限制每一列的范数可以防止某一隐藏单元有非常大的权重。如果我们将此约束转换成 Lagrange 函数中的一个惩罚，这将与  $L^2$  权重衰减类似但每个隐藏单元的权重都具有单独的 KKT 乘子。每个 KKT 乘子分别会被动态更新，以便每个隐藏单元服从约束。在实践中，列范数的限制总是通过重投影的显式约束来实现。

### 7.3 正则化和欠约束问题

在某些情况下，为了正确定义机器学习问题，正则化是必要的。机器学习中许多线性模型，包括线性回归和 PCA，都依赖于对矩阵  $\mathbf{X}^\top \mathbf{X}$  求逆。只要  $\mathbf{X}^\top \mathbf{X}$  是奇异的，这些方法就会失效。当数据生成分布在一些方向上确实没有差异时，或因为例子较少（即相对输入特征的维数来说）而在一些方向上没有观察到方差时，这个矩阵就是奇异的。在这种情况下，正则化的许多形式对应求逆  $\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I}$ 。这个正则化矩阵可以保证是可逆的。

相关矩阵可逆时，这些线性问题有闭式解。没有闭式解的问题也可能是欠定的。一个例子是应用于线性可分问题的逻辑回归。如果权重向量  $\mathbf{w}$  能够实现完美分类，那么  $2\mathbf{w}$  也会以更高似然实现完美分类。类似随机梯度下降的迭代优化算法将持续增加  $\mathbf{w}$  的大小，理论上永远不会停止。在实践中，数值实现的梯度下降最终会达到导致数值溢出的超大权重，此时的行为将取决于程序员如何处理这些不是真正数字的值。

大多数形式的正则化能够保证应用于欠定问题的迭代方法收敛。例如，当似然的斜率等于权重衰减的系数时，权重衰减将阻止梯度下降继续增加权重的大小。

使用正则化解决欠定问题的想法不局限于机器学习。同样的想法在几个基本线性代数问题中也非常有用。

正如我们在第 2.9 节看到的，我们可以使用 Moore-Penrose 求解欠定线性方程。回想  $\mathbf{X}$  伪逆  $\mathbf{X}^+$  的一个定义：

$$\mathbf{X}^+ = \lim_{\alpha \searrow 0} (\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{I})^{-1} \mathbf{X}^\top. \quad (7.29)$$

现在我们可以将第 7.29 节看作进行具有权重衰减的线性回归。具体来说，当正则化系数趋向 0 时，式(7.29)是式(7.17)的极限。因此，我们可以将伪逆解释为使用正则

化来稳定欠定问题。

## 7.4 数据集增强

让机器学习模型泛化得更好的最好办法是使用更多的数据进行训练。当然，在实践中，我们拥有的数据量是很有限的。解决这个问题的一种方法是创建假数据并添加到训练集中。对于一些机器学习任务，创建新的假数据相当简单。

对分类来说这种方法是最简单的。分类器需要一个复杂的高维输入  $\mathbf{x}$ ，并用单个类别标识  $y$  概括  $\mathbf{x}$ 。这意味着分类面临的一个主要任务是要对各种各样的变换保持不变。我们可以轻易通过转换训练集中的  $\mathbf{x}$  来生成新的  $(\mathbf{x}, y)$  对。

这种方法对于其他许多任务来说并不那么容易。例如，除非我们已经解决了密度估计问题，否则在密度估计任务中生成新的假数据是很困难的。

数据集增强对一个具体的分类问题来说是特别有效的方法：对象识别。图像是高维的并包括各种巨大的变化因素，其中有许多可以轻易地模拟。即使模型已使用卷积和池化技术（第九章）对部分平移保持不变，沿训练图像每个方向平移几个像素的操作通常可以大大改善泛化。许多其他操作如旋转图像或缩放图像也已被证明非常有效。

我们必须小心，不能使用会改变类别的转换。例如，光学字符识别任务需要认识到“b”和“d”以及“6”和“9”的区别，所以对这些任务来说，水平翻转和旋转 180° 并不是合适的数据集增强方式。

能保持我们希望的分类不变，但不容易执行的转换也是存在的。例如，平面外绕轴转动难以通过简单的几何运算在输入像素上实现。

数据集增强对语音识别任务也是有效的 (Jaity and Hinton, 2013)。

在神经网络的输入层注入噪声 (Sietsma and Dow, 1991) 也可以被看作是数据增强的一种方式。对于许多分类甚至一些回归任务而言，即使小的随机噪声被加到输入，任务仍应该是能够被解决的。然而，神经网络被证明对噪声不是非常健壮 (Tang and Eliasmith, 2010)。改善神经网络健壮性的方法之一是简单地将随机噪声添加到输入再进行训练。输入噪声注入是一些无监督学习算法的一部分，如去噪自编码器(Vincent *et al.*, 2008a)。向隐藏单元施加噪声也是可行的，这可以被看作在多个抽象层上进行的数据集增强。Poole *et al.* (2014) 最近表明，噪声的幅度被细心调整后，

该方法是非常高效的。我们将在第 7.12 节介绍一个强大的正则化策略 Dropout，该策略可以被看作是通过与噪声相乘构建新输入的过程。

在比较机器学习基准测试的结果时，考虑其采取的数据集增强是很重要的。通常情况下，人工设计的数据集增强方案可以大大减少机器学习技术的泛化误差。将一个机器学习算法的性能与另一个进行对比时，对照实验是必要的。在比较机器学习算法 A 和机器学习算法 B 时，应该确保这两个算法使用同一人工设计的数据集增强方案。假设算法 A 在没有数据集增强时表现不佳，而 B 结合大量人工转换的数据后表现良好。在这样的情况下，很可能是合成转化引起了性能改进，而不是机器学习算法 B 比算法 A 更好。有时候，确定实验是否已经适当控制需要主观判断。例如，向输入注入噪声的机器学习算法是执行数据集增强的一种形式。通常，普适操作（例如，向输入添加高斯噪声）被认为是机器学习算法的一部分，而特定于一个应用领域（如随机地裁剪图像）的操作被认为是独立的预处理步骤。

## 7.5 噪声鲁棒性

第 7.4 节已经提出将噪声作用于输入，作为数据集增强策略。对于某些模型而言，向输入添加方差极小的噪声等价于对权重施加范数惩罚 (Bishop, 1995a,b)。在一般情况下，注入噪声远比简单地收缩参数强大，特别是噪声被添加到隐藏单元时会更加强大。向隐藏单元添加噪声是值得单独讨论重要的话题；在第 7.12 节所述 Dropout 算法是这种做法的主要发展方向。

另一种正则化模型的噪声使用方式是将其加到权重。这项技术主要用于循环神经网络 (Jim *et al.*, 1996; Graves, 2011)。这可以被解释为关于权重的贝叶斯推断的随机实现。贝叶斯学习过程将权重视为不确定的，并且可以通过概率分布表示这种不确定性。向权重添加噪声是反映这种不确定性的实用的随机方法。

在某些假设下，施加于权重的噪声可以被解释为与更传统的正则化形式等同，鼓励要学习的函数保持稳定。我们研究回归的情形，也就是训练将一组特征  $\mathbf{x}$  映射成一个标量的函数  $\hat{y}(\mathbf{x})$ ，并使用最小二乘代价函数衡量模型预测值  $\hat{y}(\mathbf{x})$  与真实值  $y$  的误差：

$$J = \mathbb{E}_{p(x,y)}[(\hat{y}(\mathbf{x}) - y)^2]. \quad (7.30)$$

训练集包含  $m$  对标注样例  $\{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(m)}, y^{(m)})\}$ 。

现在我们假设对每个输入表示，网络权重添加随机扰动  $\epsilon_w \sim \mathcal{N}(\epsilon; 0, \eta \mathbf{I})$ 。想象我们有一个标准的  $l$  层 MLP。我们将扰动模型记为  $\hat{y}_{\epsilon_w}(\mathbf{x})$ 。尽管有噪声注入，我们仍然希望减少网络输出误差的平方。因此目标函数变为：

$$\tilde{J}_w = \mathbb{E}_{p(\mathbf{x}, y, \epsilon_w)}[(\hat{y}_{\epsilon_w}(\mathbf{x}) - y)^2] \quad (7.31)$$

$$= \mathbb{E}_{p(\mathbf{x}, y, \epsilon_w)}[\hat{y}_{\epsilon_w}^2(\mathbf{x}) - 2y\hat{y}_{\epsilon_w}(\mathbf{x}) + y^2]. \quad (7.32)$$

对于小的  $\eta$ ，最小化带权重噪声（方差为  $\eta \mathbf{I}$ ）的  $J$  等同于最小化附加正则化项： $\eta \mathbb{E}_{p(\mathbf{x}, y)}[\|\nabla_{\mathbf{w}} \hat{y}(\mathbf{x})\|^2]$  的  $J$ 。这种形式的正则化鼓励参数进入权重小扰动对输出相对影响较小的参数空间区域。换句话说，它推动模型进入对权重小的变化相对不敏感的区域，找到的点不只是极小点，还是由平坦区域所包围的极小点 (Hochreiter and Schmidhuber, 1995)。在简化的线性回归中（例如， $\hat{y}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$ ），正则项退化为  $\eta \mathbb{E}_{p(\mathbf{x})}[\|\mathbf{x}\|^2]$ ，这与函数的参数无关，因此不会对  $\tilde{J}_w$  关于模型参数的梯度有影响。

### 7.5.1 向输出目标注入噪声

大多数数据集的  $y$  标签都有一定错误。错误的  $y$  不利于最大化  $\log p(y | \mathbf{x})$ 。避免这种情况的一种方法是显式地对标签上的噪声进行建模。例如，我们可以假设，对于一些小常数  $\epsilon$ ，训练集标记  $y$  是正确的概率是  $1 - \epsilon$ ，（以  $\epsilon$  的概率）任何其他可能的标签也可能是正确的。这个假设很容易就能解析地与代价函数结合，而不用显式地抽取噪声样本。例如，**标签平滑** (label smoothing) 通过把确切分类目标从 0 和 1 替换成  $\frac{\epsilon}{k-1}$  和  $1 - \epsilon$ ，正则化具有  $k$  个输出的 softmax 函数的模型。标准交叉熵损失可以用在这些非确切目标的输出上。使用 softmax 函数和明确目标的最大似然学习可能永远不会收敛——softmax 函数永远无法真正预测 0 概率或 1 概率，因此它会继续学习越来越大的权重，使预测更极端。使用如权重衰减等其他正则化策略能够防止这种情况。标签平滑的优势是能够防止模型追求确切概率而不影响模型学习正确分类。这种策略自 20 世纪 80 年代就已经被使用，并在现代神经网络继续保持显著特色 (Szegedy et al., 2015)。

## 7.6 半监督学习

在半监督学习的框架下， $P(\mathbf{x})$  产生的未标记样本和  $P(\mathbf{x}, \mathbf{y})$  中的标记样本都用于估计  $P(\mathbf{y} | \mathbf{x})$  或者根据  $\mathbf{x}$  预测  $\mathbf{y}$ 。

在深度学习的背景下，半监督学习通常指的是学习一个表示  $h = f(\mathbf{x})$ 。学习表示的目的是使相同类中的样本有类似的表示。无监督学习可以为如何在表示空间聚集样本提供有用线索。在输入空间紧密聚集的样本应该被映射到类似的表示。在许多情况下，新空间上的线性分类器可以达到较好的泛化 (Belkin and Niyogi, 2002; Chapelle *et al.*, 2003)。这种方法的一个经典变种是使用主成分分析作为分类前（在投影后的数据上分类）的预处理步骤。

我们可以构建这样一个模型，其中生成模型  $P(\mathbf{x})$  或  $P(\mathbf{x}, \mathbf{y})$  与判别模型  $P(\mathbf{y} | \mathbf{x})$  共享参数，而不用分离无监督和监督部分。我们权衡监督模型准则  $-\log P(\mathbf{y} | \mathbf{x})$  和无监督或生成模型准则（如  $-\log P(\mathbf{x})$  或  $-\log P(\mathbf{x}, \mathbf{y})$ ）。生成模型准则表达了对监督学习问题解的特殊形式的先验知识 (Lasserre *et al.*, 2006)，即  $P(\mathbf{x})$  的结构通过某种共享参数的方式连接到  $P(\mathbf{y} | \mathbf{x})$ 。通过控制在总准则中的生成准则，我们可以获得比纯生成或纯判别训练准则更好的权衡 (Lasserre *et al.*, 2006; Larochelle and Bengio, 2008a)。

Salakhutdinov and Hinton (2008) 描述了一种学习回归核机器中核函数的方法，其中建模  $P(\mathbf{x})$  时使用的未标记样本大大提高了  $P(\mathbf{y} | \mathbf{x})$  的效果。

更多半监督学习的信息，请参阅 Chapelle *et al.* (2006)。

## 7.7 多任务学习

多任务学习 (Caruana, 1993) 是通过合并几个任务中的样例（可以视为对参数施加的软约束）来提高泛化的一种方式。正如额外的训练样本能够将模型参数推向具有更好泛化能力的值一样，当模型的一部分被多个额外的任务共享时，这部分将被约束为良好的值（如果共享合理），通常会带来更好的泛化能力。

图 7.2 展示了多任务学习中非常普遍的一种形式，其中不同的监督任务（给定  $\mathbf{x}$  预测  $\mathbf{y}^{(i)}$ ）共享相同的输入  $\mathbf{x}$  以及一些中间层表示  $h^{(\text{share})}$ ，能学习共同的因素池。该模型通常可以分为两类相关的参数：

1. 具体任务的参数（只能从各自任务的样本中实现良好的泛化）。如图 7.2 中的上层。
2. 所有任务共享的通用参数（从所有任务的汇集数据中获益）。如图 7.2 中的下层。

因为共享参数，其统计强度可大大提高（共享参数的样本数量相对于单任务模

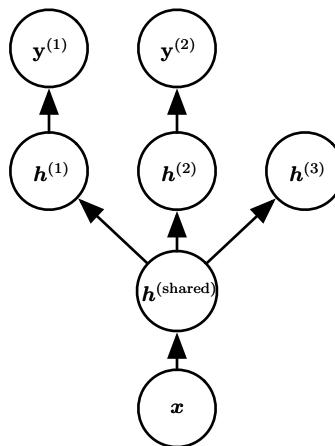


图 7.2: 多任务学习在深度学习框架中可以以多种方式进行, 该图说明了任务共享相同输入但涉及不同目标随机变量的常见情况。深度网络的较低层 (无论是监督前馈的, 还是包括向下箭头的生成组件) 可以跨这样的任务共享, 而任务特定的参数 (分别与从  $h^{(1)}$  和  $h^{(2)}$  进入和发出的权重) 可以在共享表示  $h^{(shared)}$  之上学习。这里的基本假设是存在解释输入  $x$  变化的共同因素池, 而每个任务与这些因素的子集相关联。在该示例中, 额外假设顶层隐藏单元  $h^{(1)}$  和  $h^{(2)}$  专用于每个任务 (分别预测  $y^{(1)}$  和  $y^{(2)}$ ), 而一些中间层表示  $h^{(shared)}$  在所有任务之间共享。在无监督学习情况下, 一些顶层因素不与输出任务 ( $h^{(3)}$ ) 的任意一个关联是有意义的: 这些因素可以解释一些输入变化但与预测  $y^{(1)}$  或  $y^{(2)}$  不相关。

式增加的比例), 并能改善泛化和泛化误差的范围 (Baxter, 1995)。当然, 仅当不同的任务之间存在某些统计关系的假设是合理 (意味着某些参数能通过不同任务共享) 时才会发生这种情况。

从深度学习的观点看, 底层的先验知识如下: 能解释数据变化 (在与之相关联的不同任务中观察到) 的因素中, 某些因素是跨两个或更多任务共享的。

## 7.8 提前终止

当训练有足够的表示能力甚至会过拟合的大模型时, 我们经常观察到, 训练误差会随着时间的推移逐渐降低但验证集的误差会再次上升。图 7.3 是这些现象的一个例子, 这种现象几乎一定会出现。

这意味着我们只要返回使验证集误差最低的参数设置, 就可以获得验证集误差更低的模型 (并且因此有希望获得更好的测试误差)。在每次验证集误差有所改善

后，我们存储模型参数的副本。当训练算法终止时，我们返回这些参数而不是最新的参数。当验证集上的误差在事先指定的循环次数内没有进一步改善时，算法就会终止。此过程在算法 7.1 中有更正式的说明。

这种策略被称为 **提前终止** (early stopping)。这可能是深度学习中最常用的正则化形式。它的流行主要是因为有效性和简单性。

---

**算法 7.1** 用于确定最佳训练时间量的提前终止元算法。这种元算法是一种通用策略，可以很好地在各种训练算法和各种量化验证集误差的方法上工作。

---

令  $n$  为评估间隔的步数。

令  $p$  为“耐心 (patience)”，即观察到较坏的验证集表现  $p$  次后终止。

令  $\theta_o$  为初始参数。

$\theta \leftarrow \theta_o$

$i \leftarrow 0$

$j \leftarrow 0$

$v \leftarrow \infty$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

**while**  $j < p$  **do**

    运行训练算法  $n$  步，更新  $\theta$ 。

$i \leftarrow i + n$

$v' \leftarrow \text{ValidationSetError}(\theta)$

**if**  $v' < v$  **then**

$j \leftarrow 0$

$\theta^* \leftarrow \theta$

$i^* \leftarrow i$

$v \leftarrow v'$

**else**

$j \leftarrow j + 1$

**end if**

**end while**

---

最佳参数为  $\theta^*$ ，最佳训练步数为  $i^*$

---

我们可以认为提前终止是非常高效的超参数选择算法。按照这种观点，训练步

数仅是另一个超参数。我们从图 7.3 可以看到，这个超参数在验证集上具有 U 型性能曲线。很多控制模型容量的超参数在验证集上都是这样的 U 型性能曲线，如图 5.3。在提前终止的情况下，我们通过控制拟合训练集的步数来控制模型的有效容量。大多数超参数的选择必须使用高代价的猜测和检查过程，我们需要在训练开始时猜测一个超参数，然后运行几个步骤检查它的训练效果。“训练时间”是唯一只要跑一次训练就能尝试很多值的超参数。通过提前终止自动选择超参数的唯一显著的代价是训练期间要定期评估验证集。在理想情况下，这可以并行在与主训练过程分离的机器上，或独立的 CPU，或独立的 GPU 上完成。如果没有这些额外的资源，可以使用比训练集小的验证集或较不频繁地评估验证集来减小评估代价，较粗略地估算取得最佳的训练时间。

另一个提前终止的额外代价是需要保持最佳的参数副本。这种代价一般是可忽略的，因为可以将它储存在较慢较大的存储器上（例如，在 GPU 内存中训练，但将最佳参数存储在主存储器或磁盘驱动器上）。由于最佳参数的写入很少发生而且从不在训练过程中读取，这些偶发的慢写入对总训练时间的影响不大。

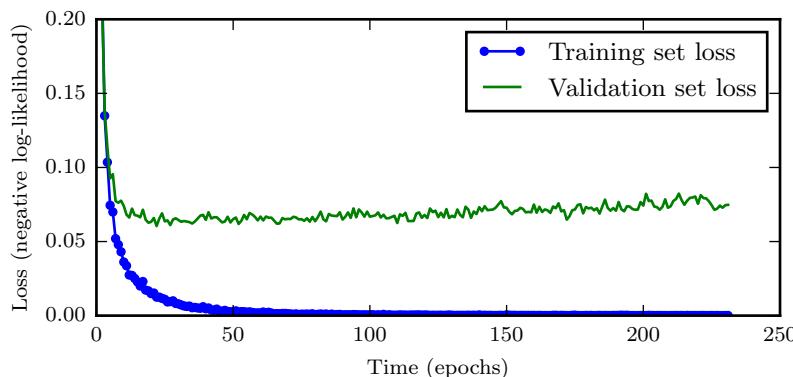


图 7.3: 学习曲线显示负对数似然损失如何随时间变化（表示为遍历数据集的训练迭代数，或 轮数 (epochs)）。在这个例子中，我们在 MNIST 上训练了一个 maxout 网络。我们可以观察到训练目标随时间持续减小，但验证集上的平均损失最终会再次增加，形成不对称的 U 形曲线。

提前终止是一种非常不显眼的正则化形式，它几乎不需要改变基本训练过程、目标函数或一组允许的参数值。这意味着，无需破坏学习动态就能很容易地使用提前终止。相对于权重衰减，必须小心不能使用太多的权重衰减，以防网络陷入不良局部极小点(对应于病态的小权重)。

提前终止可单独使用或与其他的正则化策略结合使用。即使为鼓励更好泛化，使用正则化策略改进目标函数，在训练目标的局部极小点达到最好泛化也是非常罕见的。

提前终止需要验证集，这意味着某些训练数据不能被馈送到模型。为了更好地利用这一额外的数据，我们可以在完成提前终止的首次训练之后，进行额外的训练。在第二轮，即额外的训练步骤中，所有的训练数据都被包括在内。有两个基本的策略都可以用于第二轮训练过程。

一个策略（算法 7.2）是再次初始化模型，然后使用所有数据再次训练。在这个第二轮训练过程中，我们使用第一轮提前终止训练确定的最佳步数。此过程有一些细微之处。例如，我们没有办法知道重新训练时，对参数进行相同次数的更新和对数据集进行相同次数的遍历哪一个更好。由于训练集变大了，在第二轮训练时，每一次遍历数据集将会更多次地更新参数。

另一个策略是保持从第一轮训练获得的参数，然后使用全部的数据继续训练。在这个阶段，已经没有验证集指导我们需要在训练多少步后终止。取而代之，我们可以监控验证集的平均损失函数，并继续训练，直到它低于提前终止过程终止时的目标值。此策略避免了重新训练模型的高成本，但表现并没有那么好。例如，验证集的目标不一定能达到之前的目标值，所以这种策略甚至不能保证终止。我们会在算法 7.3 中更正式地介绍这个过程。

提前终止对减少训练过程的计算成本也是有用的。除了由于限制训练的迭代次数而明显减少的计算成本，还带来了正则化的益处（不需要添加惩罚项的代价函数或计算这种附加项的梯度）。

### 算法 7.2 使用提前终止确定训练步数，然后在所有数据上训练的元算法。

令  $\mathbf{X}^{(\text{train})}$  和  $\mathbf{y}^{(\text{train})}$  为训练集。

将  $\mathbf{X}^{(\text{train})}$  和  $\mathbf{y}^{(\text{train})}$  分别分割为  $(\mathbf{X}^{(\text{subtrain})}, \mathbf{X}^{(\text{valid})})$  和  $(\mathbf{y}^{(\text{subtrain})}, \mathbf{y}^{(\text{valid})})$ 。

从随机  $\boldsymbol{\theta}$  开始，使用  $\mathbf{X}^{(\text{subtrain})}$  和  $\mathbf{y}^{(\text{subtrain})}$  作为训练集， $\mathbf{X}^{(\text{valid})}$  和  $\mathbf{y}^{(\text{valid})}$  作为验证集，运行（算法 7.1）。这将返回最佳训练步数  $i^*$ 。

将  $\boldsymbol{\theta}$  再次设为随机值。

在  $\mathbf{X}^{(\text{train})}$  和  $\mathbf{y}^{(\text{train})}$  上训练  $i^*$  步。

---

**算法 7.3** 使用提前终止确定将会过拟合的目标值，然后在所有数据上训练直到再次达到该值的元算法。

---

令  $\mathbf{X}^{(\text{train})}$  和  $\mathbf{y}^{(\text{train})}$  为训练集。

将  $\mathbf{X}^{(\text{train})}$  和  $\mathbf{y}^{(\text{train})}$  分别分割为  $(\mathbf{X}^{(\text{subtrain})}, \mathbf{X}^{(\text{valid})})$  和  $(\mathbf{y}^{(\text{subtrain})}, \mathbf{y}^{(\text{valid})})$ 。

从随机  $\boldsymbol{\theta}$  开始，使用  $\mathbf{X}^{(\text{subtrain})}$  和  $\mathbf{y}^{(\text{subtrain})}$  作为训练集， $\mathbf{X}^{(\text{valid})}$  和  $\mathbf{y}^{(\text{valid})}$  作为验证集，运行 (算法 7.1)。这会更新  $\boldsymbol{\theta}$ 。

$\epsilon \leftarrow J(\boldsymbol{\theta}, \mathbf{X}^{(\text{subtrain})}, \mathbf{y}^{(\text{subtrain})})$

**while**  $J(\boldsymbol{\theta}, \mathbf{X}^{(\text{valid})}, \mathbf{y}^{(\text{valid})}) > \epsilon$  **do**

    在  $\mathbf{X}^{(\text{train})}$  和  $\mathbf{y}^{(\text{train})}$  上训练  $n$  步。

**end while**

---

**提前终止为何具有正则化效果：** 目前为止，我们已经声明提前终止是一种正则化策略，但我们只通过展示验证集误差的学习曲线是一个 U 型曲线来支持这种说法。提前终止正则化模型的真正机制是什么呢？Bishop (1995a) 和 Sjöberg and Ljung (1995) 认为提前终止可以将优化过程的参数空间限制在初始参数值  $\boldsymbol{\theta}_0$  的小邻域内。更具体地，想象用学习率  $\epsilon$  进行  $\tau$  个优化步骤（对应于  $\tau$  个训练迭代）。我们可以将  $\epsilon\tau$  作为有效容量的度量。假设梯度有界，限制迭代的次数和学习速率能够限制从  $\boldsymbol{\theta}_0$  到达的参数空间的大小，如图 7.4 所示。在这个意义上， $\epsilon\tau$  的效果就好像是权重衰减系数的倒数。

事实上，在二次误差的简单线性模型和简单的梯度下降情况下，我们可以展示提前终止相当于  $L^2$  正则化。

为了与经典  $L^2$  正则化比较，我们只考察唯一的参数是线性权重 ( $\boldsymbol{\theta} = \mathbf{w}$ ) 的简单情形。我们在权重  $\mathbf{w}$  的经验最佳值  $\mathbf{w}^*$  附近以二次近似建模代价函数  $J$ ：

$$\hat{J}(\boldsymbol{\theta}) = J(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^*), \quad (7.33)$$

其中  $\mathbf{H}$  是  $J$  关于  $\mathbf{w}$  在  $\mathbf{w}^*$  点的 Hessian。鉴于假设  $\mathbf{w}^*$  是  $J(\mathbf{w})$  的最小点，我们知道  $\mathbf{H}$  为半正定。在局部泰勒级数逼近下，梯度由下式给出：

$$\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*). \quad (7.34)$$

接下来我们研究训练时参数向量的轨迹。为简化起见，我们将参数向量初始化为原点<sup>3</sup>，也就是  $\mathbf{w}^{(0)} = \mathbf{0}$ 。我们通过分析  $\hat{J}$  上的梯度下降来研究  $J$  上近似的梯度

<sup>3</sup>对于神经网络，我们需要打破隐藏单元间的对称平衡因此不能将所有参数都初始化为  $\mathbf{0}$  (如第 6.2 节所讨论的)。

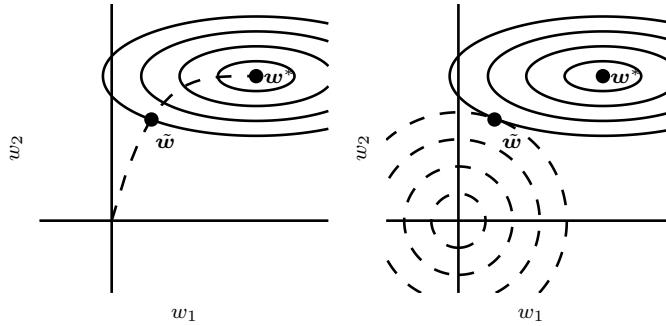


图 7.4: 提前终止效果的示意图。(左) 实线轮廓线表示负对数似然的轮廓。虚线表示从原点开始的 SGD 所经过的轨迹。提前终止的轨迹在较早的点  $\tilde{w}$  处停止, 而不是停止在最小化代价的点  $w^*$  处。(右) 为了对比, 使用  $L^2$  正则化效果的示意图。虚线圆圈表示  $L^2$  惩罚的轮廓,  $L^2$  惩罚使得总代价的最小值比非正则化代价的最小值更靠近原点。

下降的效果:

$$\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \epsilon \nabla_{\mathbf{w}} \hat{J}(\mathbf{w}^{(\tau-1)}) \quad (7.35)$$

$$= \mathbf{w}^{(\tau-1)} - \epsilon \mathbf{H}(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*), \quad (7.36)$$

$$\mathbf{w}^{(\tau)} - \mathbf{w}^* = (\mathbf{I} - \epsilon \mathbf{H})(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*). \quad (7.37)$$

现在让我们在  $\mathbf{H}$  特征向量的空间中改写表达式, 利用  $\mathbf{H}$  的特征分解:  $\mathbf{H} = \mathbf{Q}\Lambda\mathbf{Q}^\top$ , 其中  $\Lambda$  是对角矩阵,  $\mathbf{Q}$  是特征向量的一组标准正交基。

$$\mathbf{w}^{(\tau)} - \mathbf{w}^* = (\mathbf{I} - \epsilon \mathbf{Q}\Lambda\mathbf{Q}^\top)(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*) \quad (7.38)$$

$$\mathbf{Q}^\top(\mathbf{w}^{(\tau)} - \mathbf{w}^*) = (\mathbf{I} - \epsilon \Lambda)\mathbf{Q}^\top(\mathbf{w}^{(\tau-1)} - \mathbf{w}^*) \quad (7.39)$$

假定  $\mathbf{w}^{(0)} = 0$  并且  $\epsilon$  选择得足够小以保证  $|1 - \epsilon \lambda_i| < 1$ , 经过  $\tau$  次参数更新后轨迹如下:

$$\mathbf{Q}^\top \mathbf{w}^{(\tau)} = [\mathbf{I} - (\mathbf{I} - \epsilon \Lambda)^\tau] \mathbf{Q}^\top \mathbf{w}^*. \quad (7.40)$$

现在, 式 (7.13) 中  $\mathbf{Q}^\top \tilde{\mathbf{w}}$  的表达式能被重写为:

$$\mathbf{Q}^\top \tilde{\mathbf{w}} = (\Lambda + \alpha \mathbf{I})^{-1} \Lambda \mathbf{Q}^\top \mathbf{w}^*, \quad (7.41)$$

$$\mathbf{Q}^\top \tilde{\mathbf{w}} = [\mathbf{I} - (\Lambda + \alpha \mathbf{I})^{-1} \alpha] \mathbf{Q}^\top \mathbf{w}^*. \quad (7.42)$$

---

然而, 对于其他任何初始值  $\mathbf{w}_{(0)}$  该论证都成立

比较式(7.40)和式(7.42)，我们能够发现，如果超参数 $\epsilon, \alpha$ 和 $\tau$ 满足如下：

$$(\mathbf{I} - \epsilon \boldsymbol{\Lambda})^\tau = (\boldsymbol{\Lambda} + \alpha \mathbf{I})^{-1} \alpha, \quad (7.43)$$

那么 $L^2$ 正则化和提前终止可以被看作是等价的（至少在目标函数的二次近似下）。进一步取对数，使用 $\log(1+x)$ 的级数展开，我们可以得出结论：如果所有 $\lambda_i$ 是小的（即 $\epsilon\lambda_i \ll 1$ 且 $\lambda_i/\alpha \ll 1$ ），那么

$$\tau \approx \frac{1}{\epsilon\alpha}, \quad (7.44)$$

$$\alpha \approx \frac{1}{\tau\epsilon}. \quad (7.45)$$

也就是说，在这些假设下，训练迭代次数 $\tau$ 起着与 $L^2$ 参数成反比的作用， $\tau\epsilon$ 的倒数与权重衰减系数的作用类似。

在大曲率（目标函数）方向上的参数值受正则化影响小于小曲率方向。当然，在提前终止的情况下，这实际上意味着在大曲率方向的参数比较小曲率方向的参数更早地学习到。

本节中的推导表明长度为 $\tau$ 的轨迹结束于 $L^2$ 正则化目标的极小点。当然，提前终止比简单的轨迹长度限制更丰富；取而代之，提前终止通常涉及监控验证集误差，以便在空间特别好的点处终止轨迹。因此提前终止比权重衰减更具有优势，提前终止能自动确定正则化的正确量，而权重衰减需要进行多个不同超参数值的训练实验。

## 7.9 参数绑定和参数共享

目前为止，本章讨论对参数添加约束或惩罚时，一直是相对于固定的区域或点。例如， $L^2$ 正则化（或权重衰减）对参数偏离零的固定值进行惩罚。然而，有时我们可能需要其他的方式来表达我们对模型参数适当值的先验知识。有时候，我们可能无法准确地知道应该使用什么样的参数，但我们根据相关领域和模型结构方面的知识得知模型参数之间应该存在一些相关性。

我们经常想要表达的一种常见依赖是某些参数应当彼此接近。考虑以下情形：我们有两个模型执行相同的分类任务（具有相同类别），但输入分布稍有不同。形式地，我们有参数为 $\mathbf{w}^{(A)}$ 的模型 $A$ 和参数为 $\mathbf{w}^{(B)}$ 的模型 $B$ 。这两种模型将输入映射到两个不同但相关的输出： $\hat{y}^{(A)} = f(\mathbf{w}^{(A)}, \mathbf{x})$ 和 $\hat{y}^{(B)} = f(\mathbf{w}^{(B)}, \mathbf{x})$ 。

我们可以想象，这些任务会足够相似（或许具有相似的输入和输出分布），因此我们认为模型参数应彼此靠近： $\forall i, w_i^{(A)}$  应该与  $w_i^{(B)}$  接近。我们可以通过正则化利用此信息。具体来说，我们可以使用以下形式的参数范数惩罚： $\Omega(\mathbf{w}^{(A)}, \mathbf{w}^{(B)}) = \|\mathbf{w}^{(A)} - \mathbf{w}^{(B)}\|_2^2$ 。在这里我们使用  $L^2$  惩罚，但也可以使用其他选择。

这种方法由Lasserre *et al.* (2006) 提出，正则化一个模型（监督模式下训练的分类器）的参数，使其接近另一个无监督模式下训练的模型（捕捉观察到的输入数据的分布）的参数。构造的这种架构使得分类模型中的许多参数能与无监督模型中对应的参数匹配。

参数范数惩罚是正则化参数使其彼此接近的一种方式，而更流行的方法是使用约束：强迫某些参数相等。由于我们将各种模型或模型组件解释为共享唯一的一组参数，这种正则化方法通常被称为 **参数共享** (parameter sharing)。和正则化参数使其接近（通过范数惩罚）相比，参数共享的一个显著优点是，只有参数（唯一一个集合）的子集需要被存储在内存中。对于某些特定模型，如卷积神经网络，这可能可以显著减少模型所占用的内存。

### 7.9.1 卷积神经网络

目前为止，最流行和广泛使用的参数共享出现在应用于计算机视觉的 **卷积神经网络** (CNN) 中。

自然图像有许多统计属性是对转换不变的。例如，猫的照片即使向右边移了一个像素，仍保持猫的照片。CNN通过在图像多个位置共享参数来考虑这个特性。相同的特征（具有相同权重的隐藏单元）在输入的不同位置上计算获得。这意味着无论猫出现在图像中的第  $i$  列或  $i + 1$  列，我们都可以使用相同的猫探测器找到猫。

参数共享显著降低了CNN模型的参数数量，并显著提高了网络的大小而不需要相应地增加训练数据。它仍然是将领域知识有效地整合到网络架构的最佳范例之一。

我们将会在第九章中更详细地讨论卷积神经网络。

## 7.10 稀疏表示

前文所述的权重衰减直接惩罚模型参数。另一种策略是惩罚神经网络中的激活单元，稀疏化激活单元。这种策略间接地对模型参数施加了复杂惩罚。

我们已经讨论过（在第 7.1.2 节中） $L^1$  惩罚如何诱导稀疏的参数，即许多参数为零（或接近于零）。另一方面，表示的稀疏描述了许多元素是零（或接近零）的表示。我们可以线性回归的情况下简单说明这种区别：

$$\begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix} \quad (7.46)$$

$\mathbf{y} \in \mathbb{R}^m \qquad \mathbf{A} \in \mathbb{R}^{m \times n} \qquad \mathbf{x} \in \mathbb{R}^n$

$$\begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix} \quad (7.47)$$

$\mathbf{y} \in \mathbb{R}^m \qquad \mathbf{B} \in \mathbb{R}^{m \times n} \qquad \mathbf{h} \in \mathbb{R}^n$

第一个表达式是参数稀疏的线性回归模型的例子。第二个表达式是数据  $\mathbf{x}$  具有稀疏表示  $\mathbf{h}$  的线性回归。也就是说， $\mathbf{h}$  是  $\mathbf{x}$  的一个函数，在某种意义上表示存在于  $\mathbf{x}$  中的信息，但只是用一个稀疏向量表示。

表示的正则化可以使用参数正则化中同种类型的机制实现。

表示的范数惩罚正则化是通过向损失函数  $J$  添加对表示的范数惩罚来实现的。我们将这个惩罚记作  $\Omega(\mathbf{h})$ 。和以前一样，我们将正则化后的损失函数记作  $\tilde{J}$ ：

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha \Omega(\mathbf{h}), \quad (7.48)$$

其中  $\alpha \in [0, \infty]$  权衡范数惩罚项的相对贡献，越大的  $\alpha$  对应越多的正则化。

正如对参数的  $L^1$  惩罚诱导参数稀疏性，对表示元素的  $L^1$  惩罚诱导稀疏的表示： $\Omega(\mathbf{h}) = \|\mathbf{h}\|_1 = \sum_i |h_i|$ 。当然  $L^1$  惩罚是使表示稀疏的方法之一。其他方法还包括从表示上的Student- $t$  先验导出的惩罚 (Olshausen and Field, 1996; Bergstra, 2011) 和KL 散度惩罚 (Larochelle and Bengio, 2008b)，这些方法对于将表示中的元素约束于单位区间上特别有用。Lee *et al.* (2008) 和Goodfellow *et al.* (2009) 都提供了正则化几个样本平均激活的例子，即令  $\frac{1}{m} \sum_i \mathbf{h}^{(i)}$  接近某些目标值（如每项都是 .01 的向

量)。

还有一些其他方法通过激活值的硬性约束来获得表示稀疏。例如，**正交匹配追踪**(orthogonal matching pursuit)(Pati *et al.*, 1993) 通过解决以下约束优化问题将输入值  $\mathbf{x}$  编码成表示  $\mathbf{h}$

$$\arg \min_{\mathbf{h}, \|\mathbf{h}\|_0 < k} \|\mathbf{x} - \mathbf{W}\mathbf{h}\|^2, \quad (7.49)$$

其中  $\|\mathbf{h}\|_0$  是  $\mathbf{h}$  中非零项的个数。当  $\mathbf{W}$  被约束为正交时，我们可以高效地解决这个问题。这种方法通常被称为OMP- $k$ ，通过  $k$  指定允许的非零特征数量。Coates and Ng (2011) 证明OMP-1 可以成为深度架构中非常有效的特征提取器。

含有隐藏单元的模型在本质上都能变得稀疏。在本书中，我们将看到在各种情况下使用稀疏正则化的例子。

## 7.11 Bagging 和其他集成方法

**Bagging** (bootstrap aggregating) 是通过结合几个模型降低泛化误差的技术(Breiman, 1994)。主要想法是分别训练几个不同的模型，然后让所有模型表决测试样例的输出。这是机器学习中常规策略的一个例子，被称为**模型平均**(model averaging)。采用这种策略的技术被称为集成方法。

**模型平均**(model averaging)奏效的原因是不同的模型通常不会在测试集上产生完全相同的误差。

假设我们有  $k$  个回归模型。假设每个模型在每个例子上的误差是  $\epsilon_i$ ，这个误差服从零均值方差为  $\mathbb{E}[\epsilon_i^2] = v$  且协方差为  $\mathbb{E}[\epsilon_i \epsilon_j] = c$  的多维正态分布。通过所有集成模型的平均预测所得误差是  $\frac{1}{k} \sum_i \epsilon_i$ 。集成预测器平方误差的期望是

$$\mathbb{E}\left[\left(\frac{1}{k} \sum_i \epsilon_i\right)^2\right] = \frac{1}{k^2} \mathbb{E}\left[\sum_i \left(\epsilon_i^2 + \sum_{j \neq i} \epsilon_i \epsilon_j\right)\right], \quad (7.50)$$

$$= \frac{1}{k} v + \frac{k-1}{k} c. \quad (7.51)$$

在误差完全相关即  $c = v$  的情况下，均方误差减少到  $v$ ，所以模型平均没有任何帮助。在错误完全不相关即  $c = 0$  的情况下，该集成平方误差的期望仅为  $\frac{1}{k}v$ 。这意味着集成平方误差的期望会随着集成规模增大而线性减小。换言之，平均上，集成至

少与它的任何成员表现得一样好，并且如果成员的误差是独立的，集成将显著地比其成员表现得更好。

不同的集成方法以不同的方式构建集成模型。例如，集成的每个成员可以使用不同的算法和目标函数训练成完全不同的模型。Bagging是一种允许重复多次使用同一种模型、训练算法和目标函数的方法。

具体来说，Bagging涉及构造  $k$  个不同的数据集。每个数据集从原始数据集中重采样构成，和原始数据集具有相同数量的样例。这意味着，每个数据集以高概率缺少一些来自原始数据集的例子，还包含若干重复的例子（如果所得训练集与原始数据集大小相同，那所得数据集中大概有原始数据集  $2/3$  的实例）。模型  $i$  在数据集  $i$  上训练。每个数据集所含样本的差异导致了训练模型之间的差异。图 7.5 是一个例子。

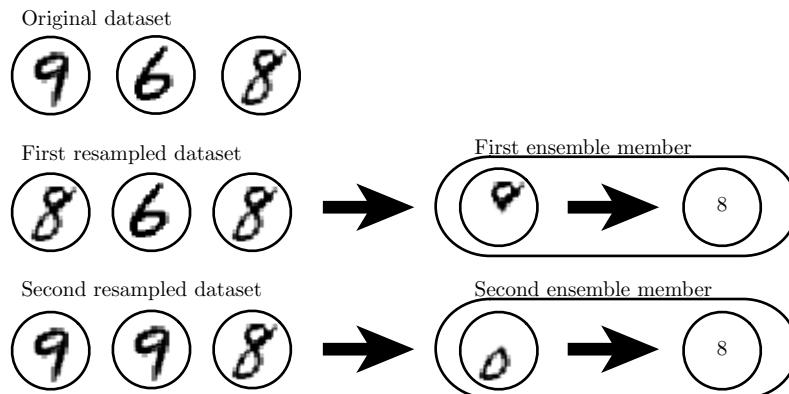


图 7.5: 描述 Bagging 如何工作的草图。假设我们在上述数据集（包含一个 8、一个 6 和一个 9）上训练数字 8 的检测器。假设我们制作了两个不同的重采样数据集。Bagging 训练程序通过有放回采样构建这些数据集。第一个数据集忽略 9 并重复 8。在这个数据集上，检测器得知数字顶部有一个环就对应于一个 8。第二个数据集中，我们忽略 6 并重复 9。在这种情况下，检测器得知数字底部有一个环就对应于一个 8。这些单独的分类规则中的每一个都是不可靠的，但如果我们平均它们的输出，就能得到鲁棒的检测器，只有当 8 的两个环都存在时才能实现最大置信度。

神经网络能找到足够多的不同的解，意味着他们可以从模型平均中受益（即使所有模型都在同一数据集上训练）。神经网络中随机初始化的差异、小批量的随机选择、超参数的差异或不同输出的非确定性实现往往足以使得集成中的不同成员具有部分独立的误差。

模型平均是一个减少泛化误差的非常强大可靠的方法。在作为科学论文算法的

基准时，它通常是不鼓励使用的，因为任何机器学习算法都可以从模型平均中大幅获益（以增加计算和存储为代价）。

机器学习比赛中的取胜算法通常是使用超过几十种模型平均的方法。最近一个突出的例子是Netflix Grand Prize(Koren, 2009)。

不是所有构建集成的技术都是为了让集成模型比单一模型更加正则化。例如，一种被称为 **Boosting** 的技术 (Freund and Schapire, 1996b,a) 构建比单个模型容量更高的集成模型。通过向集成逐步添加神经网络，Boosting已经被应用于构建神经网络的集成(Schwenk and Bengio, 1998)。通过逐渐增加神经网络的隐藏单元，Boosting也可以将单个神经网络解释为一个集成。

## 7.12 Dropout

**Dropout** (Srivastava *et al.*, 2014) 提供了正则化一大类模型的方法，计算方便但功能强大。在第一种近似下，Dropout可以被认为是集成大量深层神经网络的实用Bagging方法。Bagging涉及训练多个模型，并在每个测试样本上评估多个模型。当每个模型都是一个很大的神经网络时，这似乎是不切实际的，因为训练和评估这样的网络需要花费很多运行时间和内存。通常我们只能集成五至十个神经网络，如Szegedy *et al.* (2014a)集成了六个神经网络赢得 ILSVRC，超过这个数量就会迅速变得难以处理。Dropout提供了一种廉价的Bagging集成近似，能够训练和评估指数级数量的神经网络。

具体而言，Dropout训练的集成包括所有从基础网络除去非输出单元后形成的子网络，如图 7.6 所示。最先进的神经网络基于一系列仿射变换和非线性变换，我们只需将一些单元的输出乘零就能有效地删除一个单元。这个过程需要对模型（如径向基函数网络，单元的状态和参考值之间存在一定区别）进行一些修改。为了简单起见，我们在这里提出乘零的简单Dropout算法，但是它被简单修改后，可以与从网络中移除单元的其他操作结合使用。

回想一下Bagging学习，我们定义  $k$  个不同的模型，从训练集有放回采样构造  $k$  个不同的数据集，然后在训练集  $i$  上训练模型  $i$ 。Dropout的目标是在指数级数量的神经网络上近似这个过程。具体来说，在训练中使用Dropout时，我们会使用基于小批量产生较小步长的学习算法，如随机梯度下降等。我们每次在小批量中加载一个样本，然后随机抽样应用于网络中所有输入和隐藏单元的不同二值掩码。对于

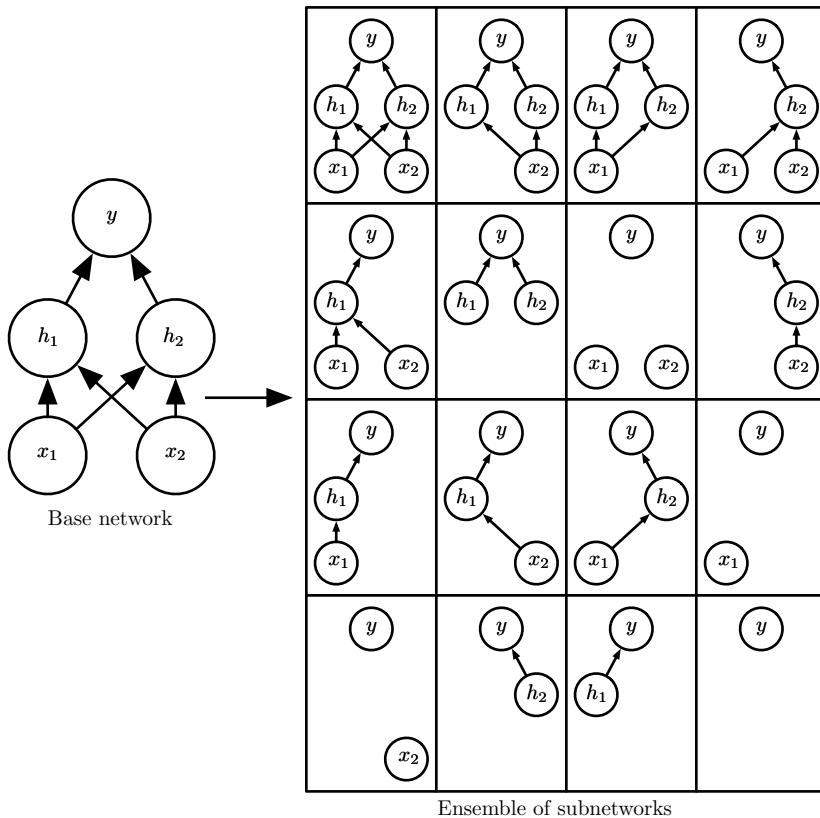


图 7.6: Dropout训练由所有子网络组成的集成，其中子网络通过从基本网络中删除非输出单元构建。我们从具有两个可见单元和两个隐藏单元的基本网络开始。这四个单元有十六个可能的子集。右图展示了从原始网络中丢弃不同的单元子集而形成的所有十六个子网络。在这个小例子中，所得到的大部分网络没有输入单元或没有从输入连接到输出的路径。当层较宽时，丢弃所有从输入到输出的可能路径的概率变小，所以这个问题不太可能在出现层较宽的网络中。

每个单元，掩码是独立采样的。掩码值为 1 的采样概率（导致包含一个单元）是训练开始前一个固定的超参数。它不是模型当前参数值或输入样本的函数。通常在每一个小批量训练的神经网络中，一个输入单元被包括的概率为 0.8，一个隐藏单元被包括的概率为 0.5。然后，我们运行和之前一样的前向传播、反向传播以及学习更新。图 7.7 说明了在Dropout下的前向传播。

更正式地说，假设一个掩码向量  $\mu$  指定被包括的单元， $J(\theta, \mu)$  是由参数  $\theta$  和掩码  $\mu$  定义的模型代价。那么Dropout训练的目标是最小化  $\mathbb{E}_\mu J(\theta, \mu)$ 。这个期望包含

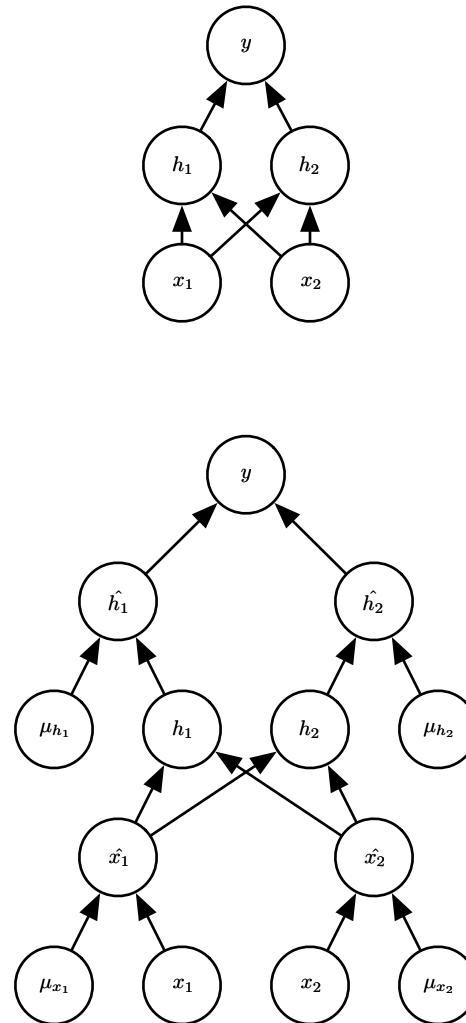


图 7.7: 在使用Dropout的前馈网络中前向传播的示例。(顶部) 在此示例中, 我们使用具有两个输入单元, 具有两个隐藏单元的隐藏层以及一个输出单元的前馈网络。(底部) 为了执行具有Dropout的前向传播, 我们随机地对向量  $\mu$  进行采样, 其中网络中的每个输入或隐藏单元对应一项。 $\mu$  中的每项都是二值的且独立于其他项采样。超参数的采样概率为 1, 隐藏层的采样概率通常为 0.5, 输入的采样概率通常为 0.8。网络中的每个单元乘以相应的掩码, 然后正常地继续沿着网络的其余部分前向传播。这相当于从图 7.6 中随机选择一个子网络并沿着前向传播。

多达指数级的项, 但我们可以抽样  $\mu$  获得梯度的无偏估计。

Dropout训练与Bagging训练不太一样。在Bagging的情况下, 所有模型都是独立

的。在Dropout的情况下，所有模型共享参数，其中每个模型继承父神经网络参数的不同子集。参数共享使得在有限可用的内存下表示指数级数量的模型变得可能。在Bagging的情况下，每一个模型在其相应训练集上训练到收敛。在Dropout的情况下，通常大部分模型都没有显式地被训练，因为通常父神经网络会很大，以致于到宇宙毁灭都不可能采样完所有的子网络。取而代之的是，在单个步骤中我们训练一小部分的子网络，参数共享会使得剩余的子网络也能有好的参数设定。这些是仅有的区别。除了这些，Dropout与Bagging算法一样。例如，每个子网络中遇到的训练集确实是有放回采样的原始训练集的一个子集。

Bagging集成必须根据所有成员的累积投票做一个预测。在这种背景下，我们将这个过程称为 **推断** (inference)。目前为止，我们在介绍Bagging和Dropout时没有要求模型具有明确的概率。现在，我们假定该模型的作用是输出一个概率分布。在Bagging的情况下，每个模型  $i$  产生一个概率分布  $p^{(i)}(y | \mathbf{x})$ 。集成的预测由这些分布的算术平均值给出，

$$\frac{1}{k} \sum_{i=1}^k p^{(i)}(y | \mathbf{x}). \quad (7.52)$$

在Dropout的情况下，通过掩码  $\boldsymbol{\mu}$  定义每个子模型的概率分布  $p(y | \mathbf{x}, \boldsymbol{\mu})$ 。所有掩码的算术平均值由下式给出

$$\sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu}) p(y | \mathbf{x}, \boldsymbol{\mu}), \quad (7.53)$$

其中  $p(\boldsymbol{\mu})$  是训练时采样  $\boldsymbol{\mu}$  的概率分布。

因为这个求和包含多达指数级的项，除非该模型的结构允许某种形式的简化，否则是不可能计算的。目前为止，无法得知深度神经网络是否允许某种可行的简化。相反，我们可以通过采样近似推断，即平均许多掩码的输出。即使是 10 – 20 个掩码就足以获得不错的表现。

然而，一个更好的方法能不错地近似整个集成的预测，且只需一个前向传播的代价。要做到这一点，我们改用集成成员预测分布的几何平均而不是算术平均。Warde-Farley *et al.* (2014) 提出的论点和经验证据表明，在这个情况下几何平均与算术平均表现得差不多。

多个概率分布的几何平均不能保证是一个概率分布。为了保证结果是一个概率分布，我们要求没有子模型给某一事件分配概率 0，并重新标准化所得分布。通过几

何平均直接定义的非标准化概率分布由下式给出

$$\tilde{p}_{\text{ensemble}}(y \mid \mathbf{x}) = \sqrt[2^d]{\prod_{\mu} p(y \mid \mathbf{x}, \mu)}, \quad (7.54)$$

其中  $d$  是可被丢弃的单元数。这里为简化介绍，我们使用均匀分布的  $\mu$ ，但非均匀分布也是可以的。为了作出预测，我们必须重新标准化集成：

$$p_{\text{ensemble}}(y \mid \mathbf{x}) = \frac{\tilde{p}_{\text{ensemble}}(y \mid \mathbf{x})}{\sum_{y'} \tilde{p}_{\text{ensemble}}(y' \mid \mathbf{x})}. \quad (7.55)$$

涉及Dropout的一个重要观点 (Hinton *et al.*, 2012c) 是，我们可以通过评估模型中  $p(y \mid \mathbf{x})$  来近似  $p_{\text{ensemble}}$ ：该模型具有所有单元，但我们将单元  $i$  的输出的权重乘以单元  $i$  的被包含概率。这个修改的动机是得到从该单元输出的正确期望值。我们把这种方法称为 **权重比例推断规则** (weight scaling inference rule)。目前还没有在深度非线性网络上对这种近似推断规则的准确性作任何理论分析，但经验上表现得很好。

因为我们通常使用  $\frac{1}{2}$  的包含概率，权重比例规则一般相当于在训练结束后将权重除 2，然后像平常一样使用模型。实现相同结果的另一种方法是在训练期间将单元的状态乘 2。无论哪种方式，我们的目标是确保在测试时一个单元的期望总输入与在训练时该单元的期望总输入是大致相同的（即使近半单位在训练时丢失）。

对许多不具有非线性隐藏单元的模型族而言，权重比例推断规则是精确的。举个简单的例子，考虑softmax 函数回归分类，其中由向量  $\mathbf{v}$  表示  $n$  个输入变量：

$$P(y = y \mid \mathbf{v}) = \text{softmax}(\mathbf{W}^\top \mathbf{v} + \mathbf{b})_y. \quad (7.56)$$

我们可以根据二值向量  $\mathbf{d}$  逐元素的乘法将一类子模型进行索引：

$$P(y = y \mid \mathbf{v}; \mathbf{d}) = \text{softmax}(\mathbf{W}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b})_y. \quad (7.57)$$

集成预测器被定义为重新标准化所有集成成员预测的几何平均：

$$P_{\text{ensemble}}(y = y \mid \mathbf{v}) = \frac{\tilde{P}_{\text{ensemble}}(y = y \mid \mathbf{v})}{\sum_{y'} \tilde{P}_{\text{ensemble}}(y = y' \mid \mathbf{v})}, \quad (7.58)$$

其中

$$\tilde{P}_{\text{ensemble}}(y = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{d \in \{0,1\}^n} P(y = y \mid \mathbf{v}; \mathbf{d})}. \quad (7.59)$$

为了证明权重比例推断规则是精确的，我们简化  $\tilde{P}_{\text{ensemble}}$ ：

$$\tilde{P}_{\text{ensemble}}(y = y \mid \mathbf{v}) = \sqrt[2^n]{\prod_{d \in \{0,1\}^n} P(y = y \mid \mathbf{v}; d)} \quad (7.60)$$

$$= \sqrt[2^n]{\prod_{d \in \{0,1\}^n} \text{softmax}(\mathbf{W}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b})_y} \quad (7.61)$$

$$= \sqrt[2^n]{\prod_{d \in \{0,1\}^n} \frac{\exp(\mathbf{W}_{y,:}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b}_y)}{\sum_{y'} \exp(\mathbf{W}_{y',:}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b}_{y'})}} \quad (7.62)$$

$$= \frac{\sqrt[2^n]{\prod_{d \in \{0,1\}^n} \exp(\mathbf{W}_{y,:}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b}_y)}}{\sqrt[2^n]{\prod_{d \in \{0,1\}^n} \sum_{y'} \exp(\mathbf{W}_{y',:}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b}_{y'})}} \quad (7.63)$$

由于  $\tilde{P}$  将被标准化，我们可以放心地忽略那些相对  $y$  不变的乘法：

$$\tilde{P}_{\text{ensemble}}(y = y \mid \mathbf{v}) \propto \sqrt[2^n]{\prod_{d \in \{0,1\}^n} \exp(\mathbf{W}_{y,:}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b}_y)} \quad (7.64)$$

$$= \exp\left(\frac{1}{2^n} \sum_{d \in \{0,1\}^n} \mathbf{W}_{y,:}^\top (\mathbf{d} \odot \mathbf{v}) + \mathbf{b}_y\right) \quad (7.65)$$

$$= \exp\left(\frac{1}{2} \mathbf{W}_{y,:}^\top \mathbf{v} + \mathbf{b}_y\right). \quad (7.66)$$

将其代入式(7.58)，我们得到了一个权重为  $\frac{1}{2} \mathbf{W}$  的softmax 函数分类器。

权重比例推断规则在其他设定下也是精确的，包括条件正态输出的回归网络以及那些隐藏层不包含非线性的深度网络。然而，权重比例推断规则对具有非线性的深度模型仅仅是一个近似。虽然这个近似尚未有理论上的分析，但在实践中往往效果很好。Goodfellow et al. (2013b) 实验发现，在对集成预测的近似方面，权重比例推断规则比蒙特卡罗近似更好（就分类精度而言）。即使允许蒙特卡罗近似采样多达 1000 子网络时也比不过权重比例推断规则。Gal and Ghahramani (2015) 发现一些模型可以通过二十个样本和蒙特卡罗近似获得更好的分类精度。似乎推断近似的最佳选择是与问题相关的。

Srivastava et al. (2014) 显示，Dropout 比其他标准的计算开销小的正则化方法（如权重衰减、过滤器范数约束和稀疏激活的正则化）更有效。Dropout 也可以与其他形式的正则化合并，得到进一步的提升。

计算方便是Dropout的一个优点。训练过程中使用Dropout产生  $n$  个随机二进制数与状态相乘，每个样本每次更新只需  $\mathcal{O}(n)$  的计算复杂度。根据实现，也可能需要

$\mathcal{O}(n)$  的存储空间来持续保存这些二进制数（直到反向传播阶段）。使用训练好的模型推断时，计算每个样本的代价与不使用Dropout是一样的，尽管我们必须在开始运行推断前将权重除以 2。

Dropout的另一个显著优点是不怎么限制适用的模型或训练过程。几乎在所有使用分布式表示且可以用随机梯度下降训练的模型上都表现很好。包括前馈神经网络、概率模型，如受限玻尔兹曼机(Srivastava *et al.*, 2014)，以及循环神经网络(Bayer and Osendorfer, 2014; Pascanu *et al.*, 2014a)。许多效果差不多的其他正则化策略对模型结构的限制更严格。

虽然Dropout在特定模型上每一步的代价是微不足道的，但在一个完整的系统上使用Dropout的代价可能非常显著。因为Dropout是一个正则化技术，它减少了模型的有效容量。为了抵消这种影响，我们必须增大模型规模。不出意外的话，使用Dropout时最佳验证集的误差会低很多，但这是以更大的模型和更多训练算法的迭代次数为代价换来的。对于非常大的数据集，正则化带来的泛化误差减少得很小。在这些情况下，使用Dropout和更大模型的计算代价可能超过正则化带来的好处。

只有极少的训练样本可用时，Dropout不会很有效。在只有不到 5000 的样本的Alternative Splicing数据集上 (Xiong *et al.*, 2011)，贝叶斯神经网络 (Neal, 1996) 比Dropout表现得更好 (Srivastava *et al.*, 2014)。当有其他未分类的数据可用时，无监督特征学习也比Dropout更有优势。

Wager *et al.* (2013) 表明，当Dropout作用于线性回归时，相当于每个输入特征具有不同权重衰减系数的  $L^2$  权重衰减。每个特征的权重衰减系数的大小是由其方差来确定的。其他线性模型也有类似的结果。而对于深度模型而言，Dropout与权重衰减是不等同的。

使用Dropout训练时的随机性不是这个方法成功的必要条件。它仅仅是近似所有子模型总和的一个方法。Wang and Manning (2013) 导出了近似这种边缘分布的解析解。他们的近似被称为 **快速 Dropout** (fast dropout)，减小梯度计算中的随机性而获得更快的收敛速度。这种方法也可以在测试时应用，能够比权重比例推断规则更合理地（但计算也更昂贵）近似所有子网络的平均。快速 Dropout在小神经网络上的性能几乎与标准的Dropout相当，但在大问题上尚未产生显著改善或尚未应用。

随机性对实现Dropout的正则化效果不是必要的，同时也不是充分的。为了证明这一点，Warde-Farley *et al.* (2014) 使用一种被称为 **Dropout Boosting** 的方法设计了一个对照实验，具有与传统Dropout方法完全相同的噪声掩码，但缺乏正则化效

果。Dropout Boosting训练整个集成以最大化训练集上的似然。从传统Dropout类似于Bagging的角度来看，这种方式类似于Boosting。如预期一样，和单一模型训练整个网络相比，Dropout Boosting几乎没有正则化效果。这表明，使用Bagging解释Dropout比使用稳健性噪声解释Dropout更好。只有当随机抽样的集成成员相互独立地训练好后，才能达到Bagging集成的正则化效果。

Dropout启发其他以随机方法训练指数量级的共享权重的集成。DropConnect是Dropout的一个特殊情况，其中一个标量权重和单个隐藏单元状态之间的每个乘积被认为是可以丢弃的一个单元 (Wan *et al.*, 2013)。随机池化是构造卷积神经网络集成的一种随机化池化的形式 (见第 9.3 节)，其中每个卷积网络参与每个特征图的不同空间位置。目前为止，Dropout仍然是最广泛使用的隐式集成方法。

一个关于Dropout的重要见解是，通过随机行为训练网络并平均多个随机决定进行预测，实现了一种参数共享的Bagging形式。早些时候，我们将Dropout描述为通过包括或排除单元形成模型集成的Bagging。然而，这种参数共享策略不一定要基于包括和排除。原则上，任何一种随机的修改都是可接受的。在实践中，我们必须选择让神经网络能够学习对抗的修改类型。在理想情况下，我们也应该使用可以快速近似推断的模型族。我们可以认为由向量  $\mu$  参数化的任何形式的修改，是对  $\mu$  所有可能的值训练  $p(y | \mathbf{x}, \mu)$  的集成。注意，这里不要求  $\mu$  具有有限数量的值。例如， $\mu$  可以是实值。Srivastava *et al.* (2014) 表明，权重乘以  $\mu \sim \mathcal{N}(\mathbf{1}, \mathbf{I})$  比基于二值掩码Dropout表现得更好。由于  $\mathbb{E}[\mu] = 1$ ，标准网络自动实现集成的近似推断，而不需要权重比例推断规则。

目前为止，我们将Dropout介绍为一种纯粹高效近似Bagging的方法。然而，还有比这更进一步的Dropout观点。Dropout不仅仅是训练一个Bagging的集成模型，并且是共享隐藏单元的集成模型。这意味着无论其他隐藏单元是否在模型中，每个隐藏单元必须都能够表现良好。隐藏单元必须准备好进行模型之间的交换和互换。Hinton *et al.* (2012d) 由生物学的想法受到启发：有性繁殖涉及到两个不同生物体之间交换基因，进化产生的压力使得基因不仅是良好的而且要准备好不同有机体之间的交换。这样的基因和这些特点对环境的变化是非常稳健的，因为它们一定会正确适应任何一个有机体或模型不寻常的特性。因此Dropout正则化每个隐藏单元不仅是一个很好的特征，更要在许多情况下是良好的特征。Warde-Farley *et al.* (2014) 将Dropout与大集成的训练相比并得出结论：相比独立模型集成获得泛化误差改进，Dropout会带来额外的改进。

Dropout强大的大部分原因来自施加到隐藏单元的掩码噪声，了解这一事实是重

要的。这可以看作是对输入内容的信息高度智能化、自适应破坏的一种形式，而不是对输入原始值的破坏。例如，如果模型学得通过鼻检测脸的隐藏单元  $h_i$ ，那么丢失  $h_i$  对应于擦除图像中有鼻子的信息。模型必须学习另一种  $h_i$ ，要么是鼻子存在的冗余编码，要么是像嘴这样的脸部的另一特征。传统的噪声注入技术，在输入端加非结构化的噪声不能够随机地从脸部图像中抹去关于鼻子的信息，除非噪声的幅度大到几乎能抹去图像中所有的信息。破坏提取的特征而不是原始值，让破坏过程充分利用该模型迄今获得的关于输入分布的所有知识。

Dropout的另一个重要方面是噪声是乘性的。如果是固定规模的加性噪声，那么加了噪声  $\epsilon$  的整流线性隐藏单元可以简单地学会使  $h_i$  变得很大（使增加的噪声  $\epsilon$  变得不显著）。乘性噪声不允许这样病态地解决噪声鲁棒性问题。

另一种深度学习算法——批标准化，在训练时向隐藏单元引入加性和乘性噪声重新参数化模型。批标准化的主要目的是改善优化，但噪声具有正则化的效果，有时没必要再使用Dropout。批标准化将会在第 8.7.1 节中被更详细地讨论。

## 7.13 对抗训练

在许多情况下，神经网络在独立同分布的测试集上进行评估已经达到了人类表现。因此，我们自然要怀疑这些模型在这些任务上是否获得了真正的人类层次的理解。为了探索网络对底层任务的理解层次，我们可以探索这个模型错误分类的例子。Szegedy *et al.* (2014b) 发现，在精度达到人类水平的神经网络上通过优化过程故意构造数据点，其上的误差率接近100%，模型在这个输入点  $x'$  的输出与附近的数据点  $x$  非常不同。在许多情况下， $x'$  与  $x$  非常近似，人类观察者不会察觉原始样本和**对抗样本** (adversarial example) 之间的差异，但是网络会作出非常不同的预测。见图 7.8 中的例子。

对抗样本在很多领域有很多影响，例如计算机安全，这超出了本章的范围。然而，它们在正则化的背景下很有意思，因为我们可以通过对**对抗训练** (adversarial training) 减少原有独立同分布的测试集的错误率——在对抗扰动的训练集样本上训练网络 (Szegedy *et al.*, 2014b; Goodfellow *et al.*, 2014b)。

Goodfellow *et al.* (2014b) 表明，这些对抗样本的主要原因之一是过度线性。神经网络主要是基于线性块构建的。因此在一些实验中，它们实现的整体函数被证明是高度线性的。这些线性函数很容易优化。不幸的是，如果一个线性函数具有许多

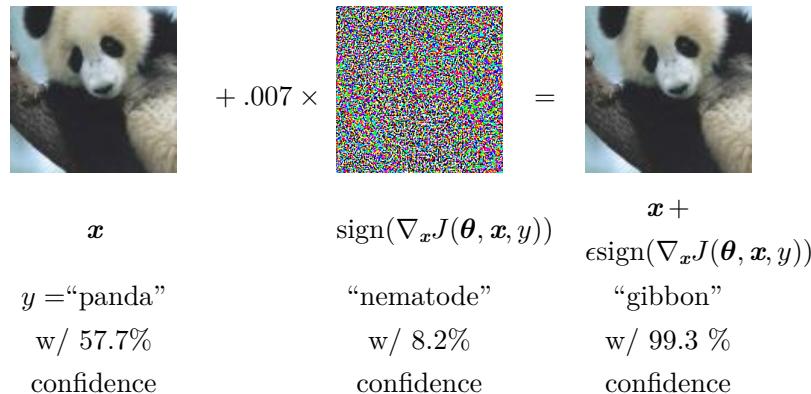


图 7.8: 在 ImageNet 上应用 GoogLeNet (Szegedy *et al.*, 2014a) 的对抗样本生成的演示。通过添加一个不可察觉的小向量 (其中元素等于代价函数相对于输入的梯度元素的符号), 我们可以改变 GoogLeNet 对此图像的分类结果。经 Goodfellow *et al.* (2014b) 许可转载。

输入, 那么它的值可以非常迅速地改变。如果我们用  $\epsilon$  改变每个输入, 那么权重为  $\mathbf{w}$  的线性函数可以改变  $\epsilon \|\mathbf{w}\|_1$  之多, 如果  $\mathbf{w}$  是高维的这会是一个非常大的数。对抗训练通过鼓励网络在训练数据附近的局部区域恒定来限制这一高度敏感的局部线性行为。这可以被看作是一种明确地向监督神经网络引入局部恒定先验的方法。

对抗训练有助于体现积极正则化与大型函数族结合的力量。纯粹的线性模型, 如逻辑回归, 由于它们被限制为线性而无法抵抗对抗样本。神经网络能够将函数从接近线性转化为局部近似恒定, 从而可以灵活地捕获到训练数据中的线性趋势同时学习抵抗局部扰动。

对抗样本也提供了一种实现半监督学习的方法。在与数据集中的标签不相关联的点  $\mathbf{x}$  处, 模型本身为其分配一些标签  $\hat{y}$ 。模型的标记  $\hat{y}$  未必是真正的标签, 但如果模型是高品质的, 那么  $\hat{y}$  提供正确标签的可能性很大。我们可以搜索一个对抗样本  $\mathbf{x}'$ , 导致分类器输出一个标签  $y'$  且  $y' \neq \hat{y}$ 。不使用真正的标签, 而是由训练好的模型提供标签产生的对抗样本被称为 **虚拟对抗样本** (virtual adversarial example) (Miyato *et al.*, 2015)。我们可以训练分类器为  $\mathbf{x}$  和  $\mathbf{x}'$  分配相同的标签。这鼓励分类器学习一个沿着未标签数据所在流形上任意微小变化都很鲁棒的函数。驱动这种方法的假设是, 不同的类通常位于分离的流形上, 并且小扰动不会使数据点从一个类的流形跳到另一个类的流形上。

## 7.14 切面距离、正切传播和流形正切分类器

如第 5.11.3 节所述，许多机器学习通过假设数据位于低维流形附近来克服维数灾难。

一个利用流形假设的早期尝试是 **切面距离** (tangent distance) 算法 (Simard *et al.*, 1993, 1998)。它是一种非参数的最近邻算法，其中使用的度量不是通用的欧几里德距离，而是根据邻近流形关于聚集概率的知识导出的。这个算法假设我们尝试分类的样本和同一流形上的样本具有相同的类别。由于分类器应该对局部因素（对应于流形上的移动）的变化保持不变，一种合理的度量是将点  $\mathbf{x}_1$  和  $\mathbf{x}_2$  各自所在流形  $M_1$  和  $M_2$  的距离作为点  $\mathbf{x}_1$  和  $\mathbf{x}_2$  之间的最近邻距离。然而这可能在计算上是困难的（它需要解决一个寻找  $M_1$  和  $M_2$  最近点对的优化问题），一种局部合理的廉价替代是使用  $\mathbf{x}_i$  点处切平面近似  $M_i$ ，并测量两条切平面或一个切平面和点之间的距离。这可以通过求解一个低维线性系统（就流形的维数而言）来实现。当然，这种算法需要指定那些切向量。

受相关启发，**正切传播** (tangent prop) 算法 (Simard *et al.*, 1992) (图 7.9) 训练带有额外惩罚的神经网络分类器，使神经网络的每个输出  $f(\mathbf{x})$  对已知的变化因素是局部不变的。这些变化因素对应于沿着的相同样本聚集的流形的移动。这里实现局部不变性的方法是要求  $\nabla_{\mathbf{x}}f(\mathbf{x})$  与已知流形的切向  $\mathbf{v}^{(i)}$  正交，或者等价地通过正则化惩罚  $\Omega$  使  $f$  在  $\mathbf{x}$  的  $\mathbf{v}^{(i)}$  方向的导数较小：

$$\Omega(f) = \sum_i \left( (\nabla_{\mathbf{x}}f(\mathbf{x})^\top \mathbf{v}^{(i)}) \right)^2. \quad (7.67)$$

这个正则化项当然可以通过适当的超参数缩放，并且对于大多数神经网络，我们需要对许多输出求和（此处为描述简单， $f(\mathbf{x})$  为唯一输出）。与切面距离算法一样，我们根据切向量推导先验，通常从变换（如平移、旋转和缩放图像）的效果获得形式知识。正切传播不仅用于监督学习(Simard *et al.*, 1992)，还在强化学习(Thrun, 1995) 中有所应用。

正切传播与数据集增强密切相关。在这两种情况下，该算法的用户通过指定一组应当不会改变网络输出的转换，将其先验知识编码至算法中。不同的是在数据集增强的情况下，网络显式地训练正确分类这些施加大量变换后产生的不同输入。正切传播不需要显式访问一个新的输入点。取而代之，它解析地对模型正则化从而在指定转换的方向抵抗扰动。虽然这种解析方法是聪明优雅的，但是它有两个主要的缺点。首先，模型的正则化只能抵抗无穷小的扰动。显式的数据集增强能抵抗较大的扰

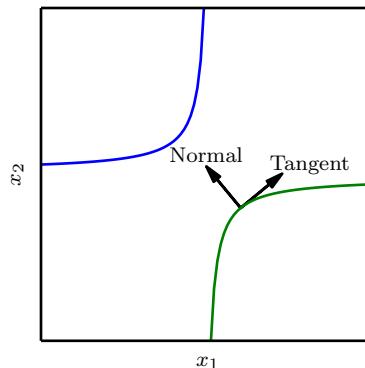


图 7.9: 正切传播算法 (Simard *et al.*, 1992) 和流形正切分类器主要思想的示意图 (Rifai *et al.*, 2011c)，它们都正则化分类器的输出函数  $f(\mathbf{x})$ 。每条曲线表示不同类别的流形，这里表示嵌入二维空间中的一维流形。在一条曲线上，我们选择单个点并绘制一个与类别流形（平行并接触流形）相切的向量以及与类别流形（与流形正交）垂直的向量。在多维情况下，可以存在许多切线方向和法线方向。我们希望分类函数在垂直于流形方向上快速改变，并且在类别流形的方向上保持不变。正切传播和流形正切分类器都会正则化  $f(\mathbf{x})$ ，使其不随  $\mathbf{x}$  沿流形的移动而剧烈变化。正切传播需要用户手动指定正切方向的计算函数（例如指定小平移后的图像保留在相同类别的流形中），而流形正切分类器通过训练自编码器拟合训练数据来估计流形的正切方向。我们将在第十四章中讨论使用自编码器来估计流形。

动。其次，我们很难在基于整流线性单元的模型上使用无限小的方法。这些模型只能通过关闭单元或缩小它们的权重才能缩小它们的导数。它们不能像sigmoid或tanh单元一样通过较大权重在高值处饱和以收缩导数。数据集增强在整流线性单元上工作得很好，因为不同的整流单元会在每一个原始输入的不同转换版本上被激活。

正切传播也和双反向传播(Drucker and LeCun, 1992) 以及对抗训练(Szegedy *et al.*, 2014b; Goodfellow *et al.*, 2014b) 有关联。双反向传播正则化使Jacobian矩阵偏小，而对抗训练找到原输入附近的点，训练模型在这些点上产生与原来输入相同的输出。正切传播和手动指定转换的数据集增强都要求模型在输入变化的某些特定的方向上保持不变。双反向传播和对抗训练都要求模型对输入所有方向中的变化（只要该变化较小）都应当保持不变。正如数据集增强是正切传播非无限小的版本，对抗训练是双反向传播非无限小的版本。

流形正切分类器 (Rifai *et al.*, 2011d) 无需知道切线向量的先验。我们将在第十四章看到，自编码器可以估算流形的切向量。流形正切分类器使用这种技术来避免

用户指定切向量。如图 14.10 所示，这些估计的切向量不仅对图像经典几何变换（如转化、旋转和缩放）保持不变，还必须掌握对特定对象（如正在移动的身体某些部分）保持不变的因素。因此根据流形正切分类器提出的算法相当简单：（1）使用自编码器通过无监督学习来学习流形的结构，以及（2）如正切传播（式(7.67)）一样使用这些切面正则化神经网络分类器。

在本章中，我们已经描述了大多数用于正则化神经网络的通用策略。正则化是机器学习的中心主题，因此我们将不时在其余各章中重新回顾。机器学习的另一个中心主题是优化，我们将在下一章描述。

# 第八章 深度模型中的优化

深度学习算法在许多情况下都涉及到优化。例如，模型中的进行推断（如PCA）涉及到求解优化问题。我们经常使用解析优化去证明或设计算法。在深度学习涉及到的诸多优化问题中，最难的是神经网络训练。甚至是用几百台机器投入几天到几个月来解决单个神经网络训练问题，也是很常见的。因为这其中的优化问题很重要，代价也很高，因此研究者们开发了一组专门为此设计的优化技术。本章会介绍神经网络训练中的这些优化技术。

如果你不熟悉基于梯度优化的基本原则，我们建议回顾第四章。该章简要概述了一般的数值优化。

本章主要关注这一类特定的优化问题：寻找神经网络上的一组参数  $\theta$ ，它能显著地降低代价函数  $J(\theta)$ ，该代价函数通常包括整个训练集上的性能评估和额外的正则化项。

首先，我们会介绍在机器学习任务中作为训练算法使用的优化与纯优化有哪些不同。接下来，我们会介绍导致神经网络优化困难的几个具体挑战。然后，我们会介绍几个实用算法，包括优化算法本身和初始化参数的策略。更高级的算法能够在训练中自适应调整学习率，或者使用代价函数二阶导数包含的信息。最后，我们会介绍几个将简单优化算法结合成高级过程的优化策略，以此作为总结。

## 8.1 学习和纯优化有什么不同

用于深度模型训练的优化算法与传统的优化算法在几个方面有所不同。机器学习通常是间接作用的。在大多数机器学习问题中，我们关注某些性能度量  $P$ ，其定义于测试集上并且可能是不可解的。因此，我们只是间接地优化  $P$ 。我们希望通过

降低代价函数  $J(\boldsymbol{\theta})$  来提高  $P$ 。这一点与纯优化不同，纯优化最小化目标  $J$  本身。训练深度模型的优化算法通常也会包括一些针对机器学习目标函数的特定结构进行的特化。

通常，代价函数可写为训练集上的平均，如

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x}, y) \sim \hat{p}_{\text{data}}} L(f(\mathbf{x}; \boldsymbol{\theta}), y), \quad (8.1)$$

其中  $L$  是每个样本的损失函数， $f(\mathbf{x}; \boldsymbol{\theta})$  是输入  $\mathbf{x}$  时所预测的输出， $\hat{p}_{\text{data}}$  是经验分布。监督学习中， $y$  是目标输出。在本章中，我们会介绍不带正则化的监督学习， $L$  的变量是  $f(\mathbf{x}; \boldsymbol{\theta})$  和  $y$ 。不难将这种监督学习扩展成其他形式，如包括  $\boldsymbol{\theta}$  或者  $\mathbf{x}$  作为参数，或是去掉参数  $y$ ，以发展不同形式的正则化或是无监督学习。

式 (8.1) 定义了训练集上的目标函数。通常，我们更希望最小化取自数据生成分布  $p_{\text{data}}$  的期望，而不仅仅是有限训练集上的对应目标函数：

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\mathbf{x}, y) \sim p_{\text{data}}} L(f(\mathbf{x}; \boldsymbol{\theta}), y). \quad (8.2)$$

### 8.1.1 经验风险最小化

机器学习算法的目标是降低式 (8.2) 所示的期望泛化误差。这个数据量被称为 **风险** (risk)。在这里，我们强调该期望取自真实的潜在分布  $p_{\text{data}}$ 。如果我们知道了真实分布  $p_{\text{data}}(\mathbf{x}, y)$ ，那么最小化风险变成了一个可以被优化算法解决的优化问题。然而，我们遇到的机器学习问题，通常是不知道  $p_{\text{data}}(\mathbf{x}, y)$ ，只知道训练集中的样本。

将机器学习问题转化回一个优化问题的最简单方法是最小化训练集上的期望损失。这意味着用训练集上的经验分布  $\hat{p}(\mathbf{x}, y)$  替代真实分布  $p(\mathbf{x}, y)$ 。现在，我们将最小化 **经验风险** (empirical risk)：

$$\mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} [L(f(\mathbf{x}; \boldsymbol{\theta}), y)] = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}), \quad (8.3)$$

其中  $m$  表示训练样本的数目。

基于最小化这种平均训练误差的训练过程被称为 **经验风险最小化** (empirical risk minimization)。在这种情况下，机器学习仍然和传统的直接优化很相似。我们并不直接最优化风险，而是最优化经验风险，希望也能够很大地降低风险。一系列不同的理论构造了一些条件，使得在这些条件下真实风险的期望可以下降不同的量。

然而，经验风险最小化很容易导致过拟合。高容量的模型会简单地记住训练集。在很多情况下，经验风险最小化并非真的可行。最有效的现代优化算法是基于梯度下降的，但是很多有用的损失函数，如  $0 - 1$  损失，没有有效的导数（导数要么为零，要么处处未定义）。这两个问题说明，在深度学习中我们很少使用经验风险最小化。反之，我们会使用一个稍有不同的方法，我们真正优化的目标会更加不同于我们希望优化的目标。

### 8.1.2 代理损失函数和提前终止

有时，我们真正关心的损失函数（比如分类误差）并不能被高效地优化。例如，即使对于线性分类器而言，精确地最小化  $0 - 1$  损失通常是不可解的（复杂度是输入维数的指数组级别）(Marcotte and Savard, 1992)。在这种情况下，我们通常会优化代理损失函数 (surrogate loss function)。代理损失函数作为原目标的代理，还具备一些优点。例如，正确类别的负对数似然通常用作  $0 - 1$  损失的替代。负对数似然允许模型估计给定样本的类别的条件概率，如果该模型效果好，那么它能够输出期望最小分类误差所对应的类别。

在某些情况下，代理损失函数比原函数学到的更多。例如，使用对数似然替代函数时，在训练集上的  $0 - 1$  损失达到 0 之后，测试集上的  $0 - 1$  损失还能持续下降很长一段时间。这是因为即使  $0 - 1$  损失期望是零时，我们还能拉开不同类别的距离以改进分类器的鲁棒性，获得一个更强壮的、更值得信赖的分类器，从而，相对于简单地最小化训练集上的平均  $0 - 1$  损失，它能够从训练数据中抽取更多信息。

一般的优化和我们用于训练算法的优化有一个重要不同：训练算法通常不会停止在局部极小点。反之，机器学习通常优化代理损失函数，但是在基于提前终止（第 7.8 节）的收敛条件满足时停止。通常，提前终止使用真实潜在损失函数，如验证集上的  $0 - 1$  损失，并设计为在过拟合发生之前终止。与纯优化不同的是，提前终止时代理损失函数仍然有较大的导数，而纯优化终止时导数较小。

### 8.1.3 批量算法和小批量算法

机器学习算法和一般优化算法不同的一点是，机器学习算法的目标函数通常可以分解为训练样本上的求和。机器学习中的优化算法在计算参数的每一次更新时通常仅使用整个代价函数中一部分项来估计代价函数的期望值。

例如，最大似然估计问题可以在对数空间中分解成各个样本的总和：

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^m \log p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}). \quad (8.4)$$

最大化这个总和等价于最大化训练集在经验分布上的期望：

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{x}, y; \boldsymbol{\theta}). \quad (8.5)$$

优化算法用到的目标函数  $J$  中的大多数属性也是训练集上的期望。例如，最常用的属性是梯度：

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\text{model}}(\mathbf{x}, y; \boldsymbol{\theta}). \quad (8.6)$$

准确计算这个期望的计算代价非常大，因为我们需要在整个数据集上的每个样本上评估模型。在实践中，我们可以从数据集中随机采样少量的样本，然后计算这些样本上的平均值。

回想一下， $n$  个样本均值的标准差（式 (5.46)）是  $\sigma/\sqrt{n}$ ，其中  $\sigma$  是样本值真实的标准差。分母  $\sqrt{n}$  表明使用更多样本来估计梯度的方法的回报是低于线性的。比较两个假想的梯度计算，一个基于 100 个样本，另一个基于 10,000 个样本。后者需要的计算量是前者的 100 倍，但却只降低了 10 倍的均值标准差。如果能够快速地计算出梯度估计值，而不是缓慢地计算准确值，那么大多数优化算法会收敛地更快（就总的计算量而言，而不是指更新次数）。

另一个促使我们从小数目样本中获得梯度的统计估计的动机是训练集的冗余。在最坏的情况下，训练集中所有的  $m$  个样本都是彼此相同的拷贝。基于采样的梯度估计可以使用单个样本计算出正确的梯度，而比原来的做法少花了  $m$  倍时间。实践中，我们不太可能真的遇到这种最坏情况，但我们可能会发现大量样本都对梯度做出了非常相似的贡献。

使用整个训练集的优化算法被称为 **批量** (batch) 或 **确定性** (deterministic) 梯度算法，因为它们会在一个大批量中同时处理所有样本。这个术语可能有点令人困惑，因为这个词“批量”也经常被用来描述小批量随机梯度下降算法中用到的小批量样本。通常，术语“批量梯度下降”指使用全部训练集，而术语“批量”单独出现时指一组样本。例如，我们普遍使用术语“批量大小”表示小批量的大小。

每次只使用单个样本的优化算法有时被称为 **随机** (stochastic) 或者 **在线** (online) 算法。术语“在线”通常是指从连续产生样本的数据流中抽取样本的情况，而不是从一个固定大小的训练集中遍历多次采样的情况。

大多数用于深度学习的算法介于以上两者之间，使用一个以上，而又不是全部的训练样本。传统上，这些会被称为**小批量**（minibatch）或**小批量随机**（minibatch stochastic）方法，现在通常将它们简单地称为**随机**（stochastic）方法。

随机方法的典型示例是随机梯度下降，这将在第 8.3.1 节中详细描述。

小批量的大小通常由以下几个因素决定：

- 更大的批量会计算更精确的梯度估计，但是回报却是小于线性的。
- 极小批量通常难以充分利用多核架构。这促使我们使用一些绝对最小批量，低于这个值的小批量处理不会减少计算时间。
- 如果批量处理中的所有样本可以并行地处理（通常确是如此），那么内存消耗和批量大小会正比。对于很多硬件设施，这是批量大小的限制因素。
- 在某些硬件上使用特定大小的数组时，运行时间会更少。尤其是在使用 GPU 时，通常使用 2 的幂数作为批量大小可以获得更少的运行时间。一般，2 的幂数的取值范围是 32 到 256，16 有时在尝试大模型时使用。
- 可能是由于小批量在学习过程中加入了噪声，它们会有一些正则化效果 (Wilson and Martinez, 2003)。泛化误差通常在批量大小为 1 时最好。因为梯度估计的高方差，小批量训练需要较小的学习率以保持稳定性。因为降低的学习率和消耗更多步骤来遍历整个训练集都会产生更多的步骤，所以会导致总的运行时间非常大。

不同的算法使用不同的方法从小批量中获取不同的信息。有些算法对采样误差比其他算法更敏感，这通常有两个可能原因。一个是它们使用了很难在少量样本上精确估计的信息，另一个是它们以放大采样误差的方式使用了信息。仅基于梯度  $\mathbf{g}$  的更新方法通常相对鲁棒，并能使用较小的批量获得成功，如 100。使用Hessian矩阵  $\mathbf{H}$ ，计算如  $\mathbf{H}^{-1}\mathbf{g}$  更新的二阶方法通常需要更大的批量，如 10,000。这些大批量需要最小化估计  $\mathbf{H}^{-1}\mathbf{g}$  的波动。假设  $\mathbf{H}$  被精确估计，但是有病态条件数。乘以  $\mathbf{H}$  或是其逆会放大之前存在的误差（这个示例中是指  $\mathbf{g}$  的估计误差）。即使  $\mathbf{H}$  被精确估计， $\mathbf{g}$  中非常小的变化也会导致更新值  $\mathbf{H}^{-1}\mathbf{g}$  中非常大的变化。当然，我们通常只会近似地估计  $\mathbf{H}$ ，因此相对于我们使用具有较差条件的操作去估计  $\mathbf{g}$ ，更新  $\mathbf{H}^{-1}\mathbf{g}$  会含有更多的误差。

小批量是随机抽取的这点也很重要。从一组样本中计算出梯度期望的无偏估计要求这些样本是独立的。我们也希望两个连续的梯度估计是互相独立的，因此两个连续的小批量样本也应该是彼此独立的。很多现实的数据集自然排列，从而使得连续的样本之间具有高度相关性。例如，假设我们有一个很长的血液样本测试结果清单。清单上的数据有可能是这样获取的，头五个血液样本于不同时间段取自第一个病人，接下来三个血液样本取自第二个病人，再随后的血液样本取自第三个病人，等等。如果我们从这个清单上顺序抽取样本，那么我们的每个小批量数据的偏差都很大，因为这个小批量很可能只代表着数据集上众多患者中的某一个患者。在这种数据集中的顺序有很大影响的情况下，很有必要在抽取小批量样本前打乱样本顺序。对于非常大的数据集，如数据中心含有几十亿样本的数据集，我们每次构建小批量样本时都将样本完全均匀地抽取出来是不太现实的。幸运的是，实践中通常将样本顺序打乱一次，然后按照这个顺序存储起来就足够了。之后训练模型时会用到的一组组小批量连续样本是固定的，每个独立的模型每次遍历训练数据时都会重复使用这个顺序。然而，这种偏离真实随机采样的方法并没有很严重的有害影响。不以某种方式打乱样本顺序才会极大地降低算法的性能。

很多机器学习上的优化问题都可以分解成并行地计算不同样本上单独的更新。换言之，我们在计算小批量样本  $\mathbf{X}$  上最小化  $J(\mathbf{X})$  的更新时，同时可以计算其他小批量样本上的更新。这类异步并行分布式方法将在第 12.1.3 节中进一步讨论。

小批量随机梯度下降的一个有趣动机是，只要没有重复使用样本，它将遵循着真实泛化误差（式 (8.2)）的梯度。很多小批量随机梯度下降方法的实现都会打乱数据顺序一次，然后多次遍历数据来更新参数。第一次遍历时，每个小批量样本都用来计算真实泛化误差的无偏估计。第二次遍历时，估计将会是有偏的，因为它重新抽取了已经用过的样本，而不是从和原先样本相同的数据生成分布中获取新的无偏的样本。

我们不难从在线学习的情况中看出随机梯度下降最小化泛化误差的原因。这时样本或者小批量都是从数据流（stream）中抽取出来的。换言之，学习器好像是一个每次看到新样本的人，每个样本  $(\mathbf{x}, y)$  都来自数据生成分布  $p_{\text{data}}(\mathbf{x}, y)$ ，而不是使用大小固定的训练集。这种情况下，样本永远不会重复；每次更新的样本是从分布  $p_{\text{data}}$  中采样获得的无偏样本。

在  $\mathbf{x}$  和  $y$  是离散时，以上的等价性很容易得到。在这种情况下，泛化误差

(式(8.2))可以表示为

$$J^*(\boldsymbol{\theta}) = \sum_{\mathbf{x}} \sum_y p_{\text{data}}(\mathbf{x}, y) L(f(\mathbf{x}; \boldsymbol{\theta}), y), \quad (8.7)$$

上式的准确梯度为

$$\mathbf{g} = \nabla_{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}) = \sum_{\mathbf{x}} \sum_y p_{\text{data}}(\mathbf{x}, y) \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}; \boldsymbol{\theta}), y). \quad (8.8)$$

在式(8.5)和式(8.6)中，我们已经在对数似然中看到了相同的结果；现在我们发现这一点在包括似然的其他函数  $L$  上也是成立的。在一些关于  $p_{\text{data}}$  和  $L$  的温和假设下，在  $\mathbf{x}$  和  $y$  是连续时也能得到类似的结果。

因此，我们可以从数据生成分布  $p_{\text{data}}$  抽取小批量样本  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  以及对应的目标  $y^{(i)}$ ，然后计算该小批量上损失函数关于对应参数的梯度

$$\hat{\mathbf{g}} = \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}). \quad (8.9)$$

以此获得泛化误差准确梯度的无偏估计。最后，在泛化误差上使用SGD方法在方向  $\hat{\mathbf{g}}$  上更新  $\boldsymbol{\theta}$ 。

当然，这个解释只能用于样本没有重复使用的情况。然而，除非训练集特别大，通常最好是多次遍历训练集。当多次遍历数据集更新时，只有第一遍满足泛化误差梯度的无偏估计。但是，额外的遍历更新当然会由于减小训练误差而得到足够的好处，以抵消其带来的训练误差和测试误差间差距的增加。

随着数据集的规模迅速增长，超越了计算能力的增速，机器学习应用每个样本只使用一次的情况变得越来越常见，甚至是不完整地使用训练集。在使用一个非常大的训练集时，过拟合不再是问题，而欠拟合和计算效率变成了主要的顾虑。读者也可以参考 Bottou and Bousquet (2008a) 中关于训练样本数目增长时，泛化误差上计算瓶颈影响的讨论。

## 8.2 神经网络优化中的挑战

优化通常是一个极其困难的任务。传统的机器学习会小心设计目标函数和约束，以确保优化问题是凸的，从而避免一般优化问题的复杂度。在训练神经网络时，我们肯定会遇到一般的非凸情况。即使是凸优化，也并非没有任何问题。在这一节中，我们会总结几个训练深度模型时会涉及到的主要挑战。

### 8.2.1 病态

在优化凸函数时，会遇到一些挑战。这其中最突出的是 Hessian 矩阵  $\mathbf{H}$  的病态。这是数值优化、凸优化或其他形式的优化中普遍存在的问题，更多细节请回顾第 4.3.1 节。

病态问题一般被认为存在于神经网络训练过程中。病态体现在随机梯度下降会“卡”在某些情况，此时即使很小的更新步长也会增加代价函数。

回顾式 (4.9)，代价函数的二阶泰勒级数展开预测梯度下降中的  $-\epsilon \mathbf{g}$  会增加

$$\frac{1}{2} \epsilon^2 \mathbf{g}^\top \mathbf{H} \mathbf{g} - \epsilon \mathbf{g}^\top \mathbf{g} \quad (8.10)$$

到代价中。当  $\frac{1}{2} \epsilon^2 \mathbf{g}^\top \mathbf{H} \mathbf{g}$  超过  $\epsilon \mathbf{g}^\top \mathbf{g}$  时，梯度的病态会成为问题。判断病态是否不利于神经网络训练任务，我们可以监测平方梯度范数  $\mathbf{g}^\top \mathbf{g}$  和  $\mathbf{g}^\top \mathbf{H} \mathbf{g}$ 。在很多情况下，梯度范数不会在训练过程中显著缩小，但是  $\mathbf{g}^\top \mathbf{H} \mathbf{g}$  的增长会超过一个数量级。其结果是尽管梯度很强，学习会变得非常缓慢，因为学习率必须收缩以弥补更强的曲率。如图 8.1 所示，成功训练的神经网络中，梯度显著增加。

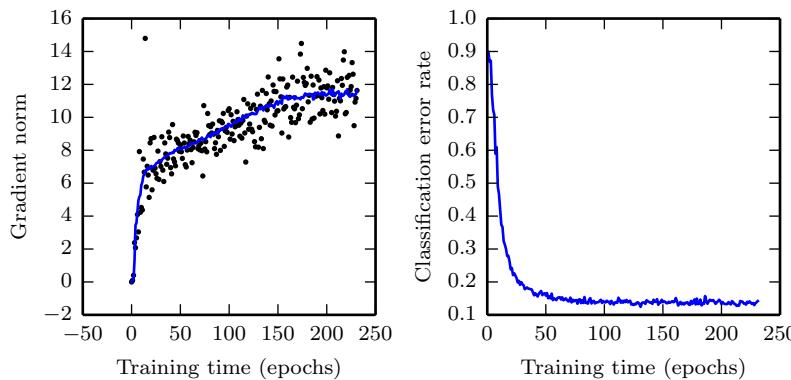


图 8.1：梯度下降通常不会到达任何类型的临界点。此示例中，在用于对象检测的卷积网络的整个训练期间，梯度范数持续增加。(左) 各个梯度计算的范数如何随时间分布的散点图。为了方便作图，每轮仅绘制一个梯度范数。我们将所有梯度范数的移动平均绘制为实曲线。梯度范数明显随时间增加，而不是如我们所期望的那样随训练过程收敛到临界点而减小。(右) 尽管梯度递增，训练过程却相当成功。验证集上的分类误差可以降低到较低水平。

尽管病态还存在于除了神经网络训练的其他情况中，有些适用于其他情况的解决病态的技术并不适用于神经网络。例如，牛顿法在解决带有病态条件的 Hessian 矩

阵的凸优化问题时，是一个非常优秀的工具，但是我们将会在以下小节中说明牛顿法运用到神经网络时需要很大的改动。

### 8.2.2 局部极小值

凸优化问题的一个突出特点是其可以简化为寻找一个局部极小点的问题。任何一个局部极小点都是全局最小点。有些凸函数的底部是一个平坦的区域，而不是单一的全局最小点，但该平坦区域中的任意点都是一个可以接受的解。优化一个凸问题时，若发现了任何形式的临界点，我们都会知道已经找到了一个不错的可行解。

对于非凸函数时，如神经网络，有可能会存在多个局部极小值。事实上，几乎所有的深度模型基本上都会有非常多的局部极小值。然而，我们会发现这并不是主要问题。

由于 **模型可辨识性** (model identifiability) 问题，神经网络和任意具有多个等效参数化潜变量的模型都会具有多个局部极小值。如果一个足够大的训练集可以唯一确定一组模型参数，那么该模型被称为可辨认的。带有潜变量的模型通常是不可辨认的，因为通过相互交换潜变量我们能得到等价的模型。例如，考虑神经网络的第一层，我们可以交换单元  $i$  和单元  $j$  的传入权重向量、传出权重向量而得到等价的模型。如果神经网络有  $m$  层，每层有  $n$  个单元，那么会有  $n!^m$  种排列隐藏单元的方式。这种不可辨认性被称为 **权重空间对称性** (weight space symmetry)。

除了权重空间对称性，很多神经网络还有其他导致不可辨认的原因。例如，在任意整流线性网络或者 maxout 网络中，我们可以将传入权重和偏置扩大  $\alpha$  倍，然后将传出权重扩大  $\frac{1}{\alpha}$  倍，而保持模型等价。这意味着，如果代价函数不包括如权重衰减这种直接依赖于权重而非模型输出的项，那么整流线性网络或者 maxout 网络的每一个局部极小点都在等价的局部极小值的  $(m \times n)$  维双曲线上。

这些模型可辨识性问题意味着神经网络代价函数具有非常多、甚至不可数无限多的局部极小值。然而，所有这些由于不可辨识性问题而产生的局部极小值都有相同的代价函数值。因此，这些局部极小值并非是非凸所带来的问题。

如果局部极小值相比全局最小点拥有很大的代价，局部极小值会带来很大的隐患。我们可以构建没有隐藏单元的小规模神经网络，其局部极小值的代价比全局最小点的代价大很多 (Sontag and Sussman, 1989; Brady *et al.*, 1989; Gori and Tesi, 1992)。如果具有很大代价的局部极小值是常见的，那么这将给基于梯度的优化算法

带来极大的问题。

对于实际中感兴趣的网络，是否存在大量代价很高的局部极小值，优化算法是否会碰到这些局部极小值，都是尚未解决的公开问题。多年来，大多数从业者认为局部极小值是困扰神经网络优化的常见问题。如今，情况有所变化。这个问题仍然是学术界的热点问题，但是学者们现在猜想，对于足够大的神经网络而言，大部分局部极小值都具有很小的代价函数，我们能不能找到真正的全局最小点并不重要，而是需要在参数空间中找到一个代价很小（但不是最小）的点 (Saxe *et al.*, 2013; Dauphin *et al.*, 2014; Goodfellow *et al.*, 2015; Choromanska *et al.*, 2014)。

很多从业者将神经网络优化中的所有困难都归结于局部极小值。我们鼓励从业者要仔细分析特定的问题。一种能够排除局部极小值是主要问题的检测方法是画出梯度范数随时间的变化。如果梯度范数没有缩小到一个微小的值，那么该问题既不是局部极小值，也不是其他形式的临界点。在高维空间中，很难明确证明局部极小值是导致问题的原因。许多并非局部极小值的结构也具有很小的梯度。

### 8.2.3 高原、鞍点和其他平坦区域

对于很多高维非凸函数而言，局部极小值（以及极大值）事实上都远少于另一类梯度为零的点：鞍点。鞍点附近的某些点比鞍点有更大的代价，而其他点则有更小的代价。在鞍点处，Hessian 矩阵同时具有正负特征值。位于正特征值对应的特征向量方向的点比鞍点有更大的代价，反之，位于负特征值对应的特征向量方向的点有更小的代价。我们可以将鞍点视为代价函数某个横截面上的局部极小点，同时也可视为代价函数某个横截面上的局部极大点。图 4.5 给了一个示例。

多类随机函数表现出以下性质：低维空间中，局部极小值很普遍。在更高维空间中，局部极小值很罕见，而鞍点则很常见。对于这类函数  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  而言，鞍点和局部极小值的数目比率的期望随  $n$  指数级增长。我们可以从直觉上理解这种现象——Hessian 矩阵在局部极小点处只有正特征值。而在鞍点处，Hessian 矩阵则同时具有正负特征值。试想一下，每个特征值的正负号由抛硬币决定。在一维情况下，很容易抛硬币得到正面朝上一次而获取局部极小点。在  $n$ -维空间中，要抛掷  $n$  次硬币都正面朝上的难度是指数级的。具体可以参考 Dauphin *et al.* (2014)，它回顾了相关的理论工作。

很多随机函数一个惊人性质是，当我们到达代价较低的区间时，Hessian 矩阵的特征值为正的可能性更大。和抛硬币类比，这意味着如果我们处于低代价的临界

点时，抛掷硬币正面朝上  $n$  次的概率更大。这也意味着，局部极小值具有低代价的可能性比高代价要大得多。具有高代价的临界点更有可能是鞍点。具有极高代价的临界点就很可能是局部极大值了。

以上现象出现在许多种类的随机函数中。那么是否在神经网络中也有发生呢？Baldi and Hornik (1989) 从理论上证明，不具非线性的浅层自编码器（第十四章中将介绍的一种将输出训练为输入拷贝的前馈网络）只有全局极小值和鞍点，没有代价比全局极小值更大的局部极小值。他们还发现这些结果能够扩展到不具非线性的更深的网络上，不过没有证明。这类网络的输出是其输入的线性函数，但它们仍然有助于分析非线性神经网络模型，因为它们的损失函数是关于参数的非凸函数。这类网络本质上是多个矩阵组合在一起。Saxe *et al.* (2013) 精确解析了这类网络中完整的学习动态，表明这些模型的学习能够捕捉到许多在训练具有非线性激活函数的深度模型时观察到的定性特征。Dauphin *et al.* (2014) 通过实验表明，真实的神经网络也存在包含很多高代价鞍点的损失函数。Choromanska *et al.* (2014) 提供了额外的理论论点，表明另一类和神经网络相关的高维随机函数也满足这种情况。

鞍点激增对于训练算法来说有哪些影响呢？对于只使用梯度信息的一阶优化算法而言，目前情况还不清楚。鞍点附近的梯度通常会非常小。另一方面，实验中梯度下降似乎可以在许多情况下逃离鞍点。Goodfellow *et al.* (2015) 可视化了最新神经网络的几个学习轨迹，图 8.2 给了一个例子。这些可视化显示，在突出的鞍点附近，代价函数都是平坦的，权重都为零。但是他们也展示了梯度下降轨迹能够迅速逸出该区间。Goodfellow *et al.* (2015) 也主张，应该可以通过分析来表明连续时间的梯度下降会逃离而不是吸引到鞍点，但对梯度下降更现实的使用场景来说，情况或许会有所不同。

对于牛顿法而言，鞍点显然是一个问题。梯度下降旨在朝“下坡”移动，而非明确寻求临界点。而牛顿法的目标是寻求梯度为零的点。如果没有适当的修改，牛顿法就会跳进一个鞍点。高维空间中鞍点的激增或许解释了在神经网络训练中为什么二阶方法无法成功取代梯度下降。Dauphin *et al.* (2014) 介绍了二阶优化的无鞍牛顿法 (saddle-free Newton method)，并表明和传统算法相比有显著改进。二阶方法仍然难以扩展到大型神经网络，但是如果这类无鞍算法能够扩展的话，还是很有希望的。

除了极小值和鞍点，还存在其他梯度为零的点。例如从优化的角度看与鞍点很相似的极大值，很多算法不会被吸引到极大值，除了未经修改的牛顿法。和极小值一样，许多种类的随机函数的极大值在高维空间中也是指数级稀少。

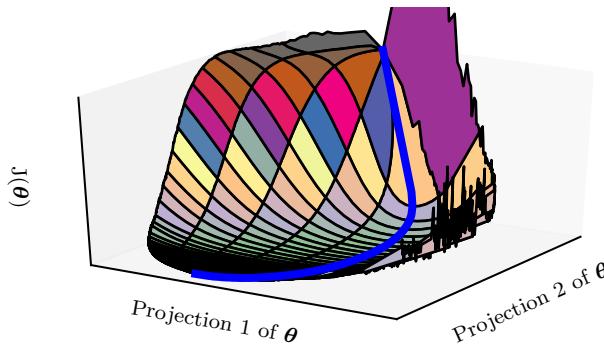


图 8.2: 神经网络代价函数的可视化。这些可视化对应用于真实对象识别和自然语言处理任务的前馈神经网络、卷积网络和循环网络而言是类似的。令人惊讶的是，这些可视化通常不会显示出很多明显的障碍。大约 2012 年，在随机梯度下降开始成功训练非常大的模型之前，相比这些投影所显示的神经网络代价函数的表面通常被认为有更多的非凸结构。该投影所显示的主要障碍是初始参数附近的高代价鞍点，但如由蓝色路径所示，SGD 训练轨迹能轻易地逃脱该鞍点。大多数训练时间花费在横穿代价函数中相对平坦的峡谷，可能由于梯度中的高噪声、或该区域中 Hessian 矩阵的病态条件，或者需要经过间接的弧路径绕过图中可见的高“山”。图经 Goodfellow *et al.* (2015) 许可改编。

也可能存在恒值的、宽且平坦的区域。在这些区域，梯度和 Hessian 矩阵都是零。这种退化的情形是所有数值优化算法的主要问题。在凸问题中，一个宽而平坦的区间肯定包含全局极小值，但是对于一般的优化问题而言，这样的区域可能会对应着目标函数中一个较高的值。

#### 8.2.4 悬崖和梯度爆炸

多层神经网络通常存在像悬崖一样的斜率较大区域，如图 8.3 所示。这是由于几个较大的权重相乘导致的。遇到斜率极大的悬崖结构时，梯度更新会很大程度地改变参数值，通常会完全跳过这类悬崖结构。

不管我们是从上还是从下接近悬崖，情况都很糟糕，但幸运的是我们可以使用第 10.11.1 节介绍的启发式 **梯度截断** (gradient clipping) 来避免其严重的后果。其基本想法源自梯度并没有指明最佳步长，只说明了在无限小区域内的最佳方向。当传统的梯度下降算法提议更新很大一步时，启发式梯度截断会干涉来减小步长，从

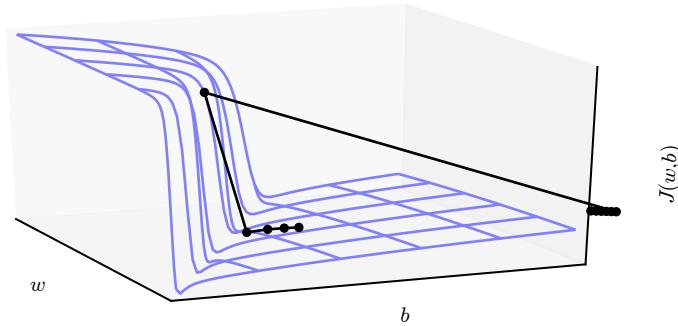


图 8.3: 高度非线性的深度神经网络或循环神经网络的目标函数通常包含由几个参数连乘而导致的参数空间中尖锐非线性。这些非线性在某些区域会产生非常大的导数。当参数接近这样的悬崖区域时，梯度下降更新可以使参数弹射得非常远，可能会使大量已完成的优化工作成为无用功。图经 Pascanu *et al.* (2013a) 许可改编。

而使其不太可能走出梯度近似为最陡下降方向的悬崖区域。悬崖结构在循环神经网络的代价函数中很常见，因为这类模型会涉及到多个因子的相乘，其中每个因子对应一个时间步。因此，长期时间序列会产生大量相乘。

### 8.2.5 长期依赖

当计算图变得极深时，神经网络优化算法会面临的另外一个难题就是长期依赖问题——由于变深的结构使模型丧失了学习到先前信息的能力，让优化变得极其困难。深层的计算图不仅存在于前馈网络，还存在于之后介绍的循环网络中（在第十章中描述）。因为循环网络要在很长时间序列的各个时刻重复应用相同操作来构建非常深的计算图，并且模型参数共享，这使问题更加凸显。

例如，假设某个计算图中包含一条反复与矩阵  $\mathbf{W}$  相乘的路径。那么  $t$  步后，相当于乘以  $\mathbf{W}^t$ 。假设  $\mathbf{W}$  有特征值分解  $\mathbf{W} = \mathbf{V}\text{diag}(\boldsymbol{\lambda})\mathbf{V}^{-1}$ 。在这种简单的情况下，很容易看出

$$\mathbf{W}^t = (\mathbf{V}\text{diag}(\boldsymbol{\lambda})\mathbf{V}^{-1})^t = \mathbf{V}\text{diag}(\boldsymbol{\lambda})^t\mathbf{V}^{-1}. \quad (8.11)$$

当特征值  $\lambda_i$  不在 1 附近时，若在量级上大于 1 则会爆炸；若小于 1 时则会消失。梯度消失与爆炸问题（vanishing and exploding gradient problem）是指该计算图上的

梯度也会因为  $\text{diag}(\boldsymbol{\lambda})^t$  大幅度变化。梯度消失使得我们难以知道参数朝哪个方向移动能够改进代价函数，而梯度爆炸会使得学习不稳定。之前描述的促使我们使用梯度截断的悬崖结构便是梯度爆炸现象的一个例子。

此处描述的在各时间步重复与  $\mathbf{W}$  相乘非常类似于寻求矩阵  $\mathbf{W}$  的最大特征值及对应特征向量的幂方法 (power method)。从这个观点来看,  $\mathbf{x}^\top \mathbf{W}^t$  最终会丢弃  $\mathbf{x}$  中所有与  $\mathbf{W}$  的主特征向量正交的成分。

循环网络在各时间步上使用相同的矩阵  $\mathbf{W}$ , 而前馈网络并没有。所以即使使用非常深层的前馈网络, 也能很大程度上有效地避免梯度消失与爆炸问题 (Sussillo, 2014)。

在更详细地描述循环网络之后, 我们将会在第 10.7 节进一步讨论循环网络训练中的挑战。

### 8.2.6 非精确梯度

大多数优化算法的先决条件都是我们知道精确的梯度或是 Hessian 矩阵。在实践中, 通常这些量会有噪声, 甚至是偏的估计。几乎每一个深度学习算法都需要基于采样的估计, 至少使用训练样本的小批量来计算梯度。

在其他情况, 我们希望最小化的目标函数实际上是以处理的。当目标函数不可解时, 通常其梯度也是难以处理的。在这种情况下, 我们只能近似梯度。这些问题主要出现在第三部分中更高级的模型中。例如, 对比散度是用来近似玻尔兹曼机中难以处理的对数似然梯度的一种技术。

各种神经网络优化算法的设计都考虑到了梯度估计的缺陷。我们可以选择比真实损失函数更容易估计的代理损失函数来避免这个问题。

### 8.2.7 局部和全局结构间的弱对应

迄今为止, 我们讨论的许多问题都是关于损失函数在单个点的性质——若  $J(\boldsymbol{\theta})$  是当前点  $\boldsymbol{\theta}$  的病态条件, 或者  $\boldsymbol{\theta}$  在悬崖中, 或者  $\boldsymbol{\theta}$  是一个下降方向不明显的鞍点, 那么会很难更新当前步。

如果该方向在局部改进很大, 但并没有指向代价低得多的遥远区域, 那么我们有可能在单点处克服以上所有困难, 但仍然表现不佳。

Goodfellow *et al.* (2015) 认为大部分训练的运行时间取决于到达解决方案的轨迹长度。如图 8.2 所示，学习轨迹将花费大量的时间探寻一个围绕山形结构的宽弧。

大多数优化研究的难点集中于训练是否找到了全局最小点、局部极小点或是鞍点，但在实践中神经网络不会到达任何一种临界点。图 8.1 表明神经网络通常不会到达梯度很小的区域。甚至，这些临界点不一定存在。例如，损失函数  $-\log p(y | \mathbf{x}; \boldsymbol{\theta})$  可以没有全局最小点，而是当随着训练模型逐渐稳定后，渐近地收敛于某个值。对于具有离散的  $y$  和 softmax 分布  $p(y | \mathbf{x})$  的分类器而言，若模型能够正确分类训练集上的每个样本，则负对数似然可以无限趋近但不会等于零。同样地，实值模型  $p(y | \mathbf{x}) = \mathcal{N}(y; f(\boldsymbol{\theta}), \beta^{-1})$  的负对数似然会趋向于负无穷——如果  $f(\boldsymbol{\theta})$  能够正确预测所有训练集中的目标  $y$ ，学习算法会无限制地增加  $\beta$ 。图 8.4 给出了一个失败的例子，即使没有局部极小值和鞍点，该例还是不能从局部优化中找到一个良好的代价函数值。

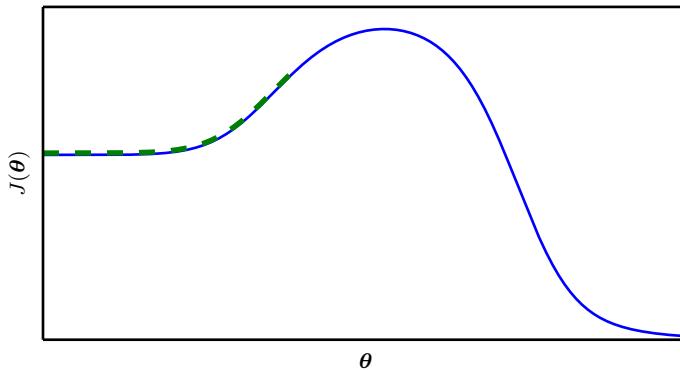


图 8.4：如果局部表面没有指向全局解，基于局部下坡移动的优化可能就会失败。这里我们提供一个例子，说明即使在没有鞍点或局部极小值的情况下，优化过程会如何失败。此例中的代价函数仅包含朝向低值而不是极小值的渐近线。在这种情况下，造成这种困难的主要原因是初始化在“山”的错误一侧，并且无法遍历。在高维空间中，学习算法通常可以绕过这样的高山，但是相关的轨迹可能会很长，并且导致过长的训练时间，如图 8.2 所示。

未来的研究需要进一步探索影响学习轨迹长度和更好地表征训练过程的结果。

许多现有研究方法在求解具有困难全局结构的问题时，旨在寻求良好的初始点，而不是开发非局部范围更新的算法。

梯度下降和基本上所有的可以有效训练神经网络的学习算法，都是基于局部较

小更新。之前的小节主要集中于为何这些局部范围更新的正确方向难以计算。我们也许能计算目标函数的一些性质，如近似的有偏梯度或正确方向估计的方差。在这些情况下，难以确定局部下降能否定义通向有效解的足够短的路径，但我们并不能真的遵循局部下降的路径。目标函数可能有诸如病态条件或不连续梯度的问题，使得梯度为目标函数提供较好近似的区间非常小。在这些情况下，步长为  $\epsilon$  的局部下降可能定义了到达解的合理的短路经，但是我们只能计算步长为  $\delta \ll \epsilon$  的局部下降方向。在这些情况下，局部下降或许能定义通向解的路径，但是该路径包含很多次更新，因此遵循该路径会带来很高的计算代价。有时，比如说当目标函数有一个宽而平的区域，或是我们试图寻求精确的临界点（通常来说后一种情况只发生于显式求解临界点的方法，如牛顿法）时，局部信息不能为我们提供任何指导。在这些情况下，局部下降完全无法定义通向解的路径。在其他情况下，局部移动可能太过贪心，朝着下坡方向移动，却和所有可行解南辕北辙，如图 8.4 所示，或者是用舍近求远的方法来求解问题，如图 8.2 所示。目前，我们还不了解这些问题中的哪一个与神经网络优化中的难点最相关，这是研究领域的热点方向。

不管哪个问题最重要，如果存在一个区域，我们遵循局部下降便能合理地直接到达某个解，并且我们能够在该良好区域上初始化学习，那么这些问题都可以避免。最终的观点还是建议在传统优化算法上研究怎样选择更佳的初始化点，以此来实现目标更切实可行。

### 8.2.8 优化的理论限制

一些理论结果表明，我们为神经网络设计的任何优化算法都有性能限制 (Blum and Rivest, 1992; Judd, 1989; Wolpert and MacReady, 1997)。通常这些结果不影响神经网络在实践中的应用。

一些理论结果仅适用于神经网络的单元输出离散值的情况。然而，大多数神经网络单元输出光滑的连续值，使得局部搜索求解优化可行。一些理论结果表明，存在某类问题是不可解的，但很难判断一个特定问题是否属于该类。其他结果表明，寻找给定规模的网络的一个可行解是很困难的，但在实际情况中，我们通过设置更多参数，使用更大的网络，能轻松找到可接受的解。此外，在神经网络训练中，我们通常不关注某个函数的精确极小点，而只关注将其值下降到足够小以获得一个良好的泛化误差。对优化算法是否能完成此目标进行理论分析是非常困难的。因此，研究优化算法更现实的性能上界仍然是学术界的一个重要目标。

## 8.3 基本算法

之前我们已经介绍了梯度下降（第 4.3 节），即沿着整个训练集的梯度方向下降。这可以使用随机梯度下降很大程度地加速，沿着随机挑选的小批量数据的梯度下降方向，就像第 5.9 节和第 8.1.3 节中讨论的一样。

### 8.3.1 随机梯度下降

随机梯度下降 (SGD) 及其变种很可能是一般机器学习中应用最多的优化算法，特别是在深度学习中。如第 8.1.3 节中所讨论的，按照数据生成分布抽取  $m$  个小批量（独立同分布的）样本，通过计算它们梯度均值，我们可以得到梯度的无偏估计。

算法 8.1 展示了如何沿着这个梯度的估计下降。

---

#### 算法 8.1 随机梯度下降 (SGD) 在第 $k$ 个训练迭代的更新

---

**Require:** 学习率  $\epsilon_k$

**Require:** 初始参数  $\theta$

**while** 停止准则未满足 **do**

从训练集中采包含  $m$  个样本  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  的小批量，其中  $\mathbf{x}^{(i)}$  对应目标为  $\mathbf{y}^{(i)}$ 。

计算梯度估计： $\hat{\mathbf{g}} \leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

应用更新： $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

**end while**

---

SGD 算法中的一个关键参数是学习率。之前，我们介绍的 SGD 使用固定的学习率。在实践中，有必要随着时间的推移逐渐降低学习率，因此我们将第  $k$  步迭代的学习率记作  $\epsilon_k$ 。

这是因为 SGD 中梯度估计引入的噪声源 ( $m$  个训练样本的随机采样) 并不会在极小点处消失。相比之下，当我们使用批量梯度下降到达极小点时，整个代价函数的真实梯度会变得很小，之后为  $\mathbf{0}$ ，因此批量梯度下降可以使用固定的学习率。保证 SGD 收敛的一个充分条件是

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \quad (8.12)$$

且

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty. \quad (8.13)$$

实践中，一般会线性衰减学习率直到第  $\tau$  次迭代：

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_{\tau} \quad (8.14)$$

其中  $\alpha = \frac{k}{\tau}$ 。在  $\tau$  步迭代之后，一般使  $\epsilon$  保持常数。

学习率可通过试验和误差来选取，通常最好的选择方法是监测目标函数值随时间变化的学习曲线。与其说是科学，这更像是一门艺术，我们应该谨慎地参考关于这个问题的大部分指导。使用线性策略时，需要选择的参数为  $\epsilon_0$ ,  $\epsilon_{\tau}$ ,  $\tau$ 。通常  $\tau$  被设为需要反复遍历训练集几百次的迭代次数。通常  $\epsilon_{\tau}$  应设为大约  $\epsilon_0$  的 1%。主要问题是如何设置  $\epsilon_0$ 。若  $\epsilon_0$  太大，学习曲线将会剧烈振荡，代价函数值通常会明显增加。温和的振荡是良好的，容易在训练随机代价函数（例如使用 Dropout 的代价函数）时出现。如果学习率太小，那么学习过程会很缓慢。如果初始学习率太低，那么学习可能会卡在一个相当高的代价值。通常，就总训练时间和最终代价值而言，最优初始学习率会高于大约迭代 100 次左右后达到最佳效果的学习率。因此，通常最好是检测最早的几轮迭代，选择一个比在效果上表现最佳的学习率更大的学习率，但又不能太大导致严重的震荡。

SGD 及相关的小批量亦或更广义的基于梯度优化的在线学习算法，一个重要的性质是每一步更新的计算时间不依赖训练样本数目的多寡。即使训练样本数目非常大时，它们也能收敛。对于足够大的数据集，SGD 可能会在处理整个训练集之前就收敛到最终测试集误差的某个固定容差范围内。

研究优化算法的收敛率，一般会衡量额外误差 (excess error)  $J(\boldsymbol{\theta}) - \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ ，即当前代价函数超出最低可能代价的量。SGD 应用于凸问题时， $k$  步迭代后的额外误差量级是  $O(\frac{1}{\sqrt{k}})$ ，在强凸情况下是  $O(\frac{1}{k})$ 。除非假定额外的条件，否则这些界限不能进一步改进。批量梯度下降在理论上比随机梯度下降有更好的收敛率。然而，Cramér-Rao 界限 (Cramér, 1946; Rao, 1945) 指出，泛化误差的下降速度不会快于  $O(\frac{1}{k})$ 。Bottou and Bousquet (2008b) 因此认为对于机器学习任务，不值得探寻收敛快于  $O(\frac{1}{k})$  的优化算法——更快的收敛可能对应着过拟合。此外，渐近分析掩盖了随机梯度下降在少量更新步之后的很多优点。对于大数据集，SGD 只需非常少量样本计算梯度从而实现初始快速更新，远远超过了其缓慢的渐近收敛。本章剩余部分介绍的大多数算法在实践中都受益于这种性质，但是损失了常数倍  $O(\frac{1}{k})$  的渐近分析。

我们也可以在学习过程中逐渐增大小批量的大小，以此权衡批量梯度下降和随机梯度下降两者的特点。

了解 SGD 更多的信息，请查看 Bottou (1998)。

### 8.3.2 动量

虽然随机梯度下降仍然是非常受欢迎的优化方法，但其学习过程有时会很慢。动量方法 (Polyak, 1964) 旨在加速学习，特别是处理高曲率、小但一致的梯度，或是带噪声的梯度。动量算法积累了之前梯度指数级衰减的移动平均，并且继续沿该方向移动。动量的效果如图 8.5 所示。

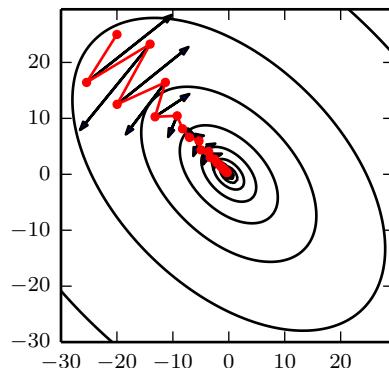


图 8.5：动量的主要目的是解决两个问题：Hessian 矩阵的病态条件和随机梯度的方差。我们通过此图说明动量如何克服这两个问题的第一个。等高线描绘了一个二次损失函数（具有病态条件的 Hessian 矩阵）。横跨轮廓的红色路径表示动量学习规则所遵循的路径，它使该函数最小化。我们在该路径的每个步骤画一个箭头，表示梯度下降将在该点采取的步骤。我们可以看到，一个病态条件的二次目标函数看起来像一个长而窄的山谷或具有陡峭边的峡谷。动量正确地纵向穿过峡谷，而普通的梯度步骤则会浪费时间在峡谷的窄轴上来回移动。比较图 4.6，它也显示了没有动量的梯度下降的行为。

从形式上看，动量算法引入了变量  $v$  充当速度角色——它代表参数在参数空间移动的方向和速率。速度被设为负梯度的指数衰减平均。名称 **动量** (momentum) 来自物理类比，根据牛顿运动定律，负梯度是移动参数空间中粒子的力。动量在物理学上定义为质量乘以速度。在动量学习算法中，我们假设是单位质量，因此速度向量  $v$  也可以看作是粒子的动量。超参数  $\alpha \in [0, 1)$  决定了之前梯度的贡献衰减得有

多快。更新规则如下：

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\theta} \left( \frac{1}{m} \sum_{i=1}^m L(\mathbf{f}(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}) \right), \quad (8.15)$$

$$\theta \leftarrow \theta + \mathbf{v}. \quad (8.16)$$

速度  $\mathbf{v}$  累积了梯度元素  $\nabla_{\theta} (\frac{1}{m} \sum_{i=1}^m L(\mathbf{f}(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}))$ 。相对于  $\epsilon$ ,  $\alpha$  越大, 之前梯度对现在方向的影响也越大。带动量的 SGD 算法如算法 8.2 所示。

---

### 算法 8.2 使用动量的随机梯度下降 (SGD)

---

**Require:** 学习率  $\epsilon$ , 动量参数  $\alpha$

**Require:** 初始参数  $\theta$ , 初始速度  $\mathbf{v}$

**while** 没有达到停止准则 **do**

从训练集中采包含  $m$  个样本  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  的小批量, 对应目标为  $\mathbf{y}^{(i)}$ 。

计算梯度估计:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(\mathbf{f}(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

计算速度更新:  $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \mathbf{g}$

应用更新:  $\theta \leftarrow \theta + \mathbf{v}$

**end while**

---

之前, 步长只是梯度范数乘以学习率。现在, 步长取决于梯度序列的大小和排列。当许多连续的梯度指向相同的方向时, 步长最大。如果动量算法总是观测到梯度  $\mathbf{g}$ , 那么它会在方向  $-\mathbf{g}$  上不停加速, 直到达到最终速度, 其中步长大小为

$$\frac{\epsilon \|\mathbf{g}\|}{1 - \alpha}. \quad (8.17)$$

因此将动量的超参数视为  $\frac{1}{1-\alpha}$  有助于理解。例如,  $\alpha = 0.9$  对应着最大速度 10 倍于梯度下降算法。

在实践中,  $\alpha$  的一般取值为 0.5, 0.9 和 0.99。和学习率一样,  $\alpha$  也会随着时间不断调整。一般初始值是一个较小的值, 随后会慢慢变大。随着时间推移调整  $\alpha$  没有收缩  $\epsilon$  重要。

我们可以将动量算法视为模拟连续时间下牛顿力学下的粒子。这种物理类比有助于直觉上理解动量和梯度下降算法是如何表现的。

粒子在任意时间点的位置由  $\theta(t)$  给定。粒子会受到净力  $\mathbf{f}(t)$ 。该力会导致粒子加速:

$$\mathbf{f}(t) = \frac{\partial^2}{\partial t^2} \theta(t). \quad (8.18)$$

与其将其视为位置的二阶微分方程，我们不如引入表示粒子在时间  $t$  处速度的变量  $\mathbf{v}(t)$ ，将牛顿动力学重写为一阶微分方程：

$$\mathbf{v}(t) = \frac{\partial}{\partial t} \boldsymbol{\theta}(t), \quad (8.19)$$

$$\mathbf{f}(t) = \frac{\partial}{\partial t} \mathbf{v}(t). \quad (8.20)$$

由此，动量算法包括通过数值模拟求解微分方程。求解微分方程的一个简单数值方法是欧拉方法，通过在每个梯度方向上小且有限的步来简单模拟该等式定义的动力学。

这解释了动量更新的基本形式，但具体什么是力呢？力正比于代价函数的负梯度  $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ 。该力推动粒子沿着代价函数表面下坡的方向移动。梯度下降算法基于每个梯度简单地更新一步，而使用动量算法的牛顿方案则使用该力改变粒子的速度。我们可以将粒子视作在冰面上滑行的冰球。每当它沿着表面最陡的部分下降时，它会累积继续在该方向上滑行的速度，直到其开始向上滑动为止。

另一个力也是必要的。如果代价函数的梯度是唯一的力，那么粒子可能永远不会停下来。想象一下，假设理想情况下冰面没有摩擦，一个冰球从山谷的一端下滑，上升到另一端，永远来回振荡。要解决这个问题，我们添加另一个正比于  $-\mathbf{v}(t)$  的力。在物理术语中，此力对应于粘性阻力，就像粒子必须通过一个抵抗介质，如糖浆。这会导致粒子随着时间推移逐渐失去能量，最终收敛到局部极小点。

为什么要特别使用  $-\mathbf{v}(t)$  和粘性阻力呢？部分原因是因为  $-\mathbf{v}(t)$  在数学上的便利——速度的整数幂很容易处理。然而，其他物理系统具有基于速度的其他整数幂的其他类型的阻力。例如，颗粒通过空气时会受到正比于速度平方的湍流阻力，而颗粒沿着地面移动时会受到恒定大小的摩擦力。这些选择都不合适。湍流阻力，正比于速度的平方，在速度很小时会很弱。不够强到使粒子停下来。非零值初始速度的粒子仅受到湍流阻力，会从初始位置永远地移动下去，和初始位置的距离大概正比于  $O(\log t)$ 。因此我们必须使用速度较低幂次的力。如果幂次为零，相当于干摩擦，那么力太强了。当代价函数的梯度表示的力很小但非零时，由于摩擦导致的恒力会使得粒子在达到局部极小点之前就停下来。粘性阻力避免了这两个问题——它足够弱，可以使梯度引起的运动直到达到最小，但又足够强，使得坡度不够时可以阻止运动。

### 8.3.3 Nesterov 动量

受 Nesterov 加速梯度算法 (Nesterov, 1983, 2004) 启发, Sutskever *et al.* (2013) 提出了动量算法的一个变种。这种情况的更新规则如下：

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\theta} \left[ \frac{1}{m} \sum_{i=1}^m L(\mathbf{f}(\mathbf{x}^{(i)}; \theta + \alpha \mathbf{v}), \mathbf{y}^{(i)}) \right], \quad (8.21)$$

$$\theta \leftarrow \theta + \mathbf{v}, \quad (8.22)$$

其中参数  $\alpha$  和  $\epsilon$  发挥了和标准动量方法中类似的作用。Nesterov 动量和标准动量之间的区别体现在梯度计算上。Nesterov 动量中, 梯度计算在施加当前速度之后。因此, Nesterov 动量可以解释为往标准动量方法中添加了一个校正因子。完整的 Nesterov 动量算法如算法 8.3 所示。

---

#### 算法 8.3 使用 Nesterov 动量的随机梯度下降 (SGD)

---

**Require:** 学习率  $\epsilon$ , 动量参数  $\alpha$

**Require:** 初始参数  $\theta$ , 初始速度  $\mathbf{v}$

**while** 没有达到停止准则 **do**

从训练集中采包含  $m$  个样本  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  的小批量, 对应目标为  $\mathbf{y}^{(i)}$ 。

应用临时更新:  $\tilde{\theta} \leftarrow \theta + \alpha \mathbf{v}$

计算梯度 (在临时点):  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$

计算速度更新:  $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \mathbf{g}$

应用更新:  $\theta \leftarrow \theta + \mathbf{v}$

**end while**

---

在凸批量梯度的情况下, Nesterov 动量将额外误差收敛率从  $O(1/k)$  ( $k$  步后) 改进到  $O(1/k^2)$ , 如 Nesterov (1983) 所示。可惜, 在随机梯度的情况下, Nesterov 动量没有改进收敛率。

## 8.4 参数初始化策略

有些优化算法本质上是非迭代的, 只是求解一个解点。有些其它优化算法本质上是迭代的, 但是应用于这一类的优化问题时, 能在可接受的时间内收敛到可接受的解, 并且与初始值无关。深度学习训练算法通常没有这两种奢侈的性质。深度学

习模型的训练算法通常是迭代的，因此要求使用者指定一些开始迭代的初始点。此外，训练深度模型是一个足够困难的问题，以致于大多数算法都很大程度地受到初始化选择的影响。初始点能够决定算法是否收敛，有些初始点十分不稳定，使得该算法会遭遇数值困难，并完全失败。当学习收敛时，初始点可以决定学习收敛得多快，以及是否收敛到一个代价高或低的点。此外，差不多代价的点可以具有区别极大的泛化误差，初始点也可以影响泛化。

现代的初始化策略是简单的、启发式的。设定改进的初始化策略是一项困难的任务，因为神经网络优化至今还未被很好地理解。大多数初始化策略基于在神经网络初始化时实现一些很好的性质。然而，我们并没有很好地理解这些性质中的哪些会在学习开始进行后的哪些情况下得以保持。进一步的难点是，有些初始点从优化的观点看或许是有利的，但是从泛化的观点看是不利的。我们对于初始点如何影响泛化的理解是相当原始的，几乎没有提供如何选择初始点的任何指导。

也许完全可知的唯一特性是初始参数需要在不同单元间“破坏对称性”。如果具有相同激活函数的两个隐藏单元连接到相同的输入，那么这些单元必须具有不同的初始参数。如果它们具有相同的初始参数，然后应用到确定性损失和模型的确定性学习算法将一直以相同的方式更新这两个单元。即使模型或训练算法能够使用随机性为不同的单元计算不同的更新（例如使用Dropout的训练），通常来说，最好还是初始化每个单元使其和其他单元计算不同的函数。这或许有助于确保没有输入模式丢失在前向传播的零空间中，没有梯度模式丢失在反向传播的零空间中。每个单元计算不同函数的目标促使了参数的随机初始化。我们可以明确地搜索一大组彼此互不相同的基函数，但这经常会导致明显的计算代价。例如，如果我们有和输出一样多的输入，我们可以使用Gram-Schmidt正交化于初始的权重矩阵，保证每个单元计算彼此非常不同的函数。在高维空间上使用高熵分布来随机初始化，计算代价小并且不太可能分配单元计算彼此相同的函数。

通常情况下，我们可以为每个单元的偏置设置启发式挑选的常数，仅随机初始化权重。额外的参数（例如用于编码预测条件方差的参数）通常和偏置一样设置为启发式选择的常数。

我们几乎总是初始化模型的权重为高斯或均匀分布中随机抽取的值。高斯或均匀分布的选择似乎不会有很大的差别，但也没有被详尽地研究。然而，初始分布的大小确实对优化过程的结果和网络泛化能力都有很大的影响。

更大的初始权重具有更强的破坏对称性的作用，有助于避免冗余的单元。它们

也有助于避免在每层线性成分的前向或反向传播中丢失信号——矩阵中更大的值在矩阵乘法中有更大的输出。如果初始权重太大，那么会在前向传播或反向传播中产生爆炸的值。在循环网络中，很大的权重也可能导致混沌（chaos）（对于输入中很小的扰动非常敏感，导致确定性前向传播过程表现随机）。在一定程度上，梯度爆炸问题可以通过梯度截断来缓解（执行梯度下降步骤之前设置梯度的阈值）。较大的权重也会产生使得激活函数饱和的值，导致饱和单元的梯度完全丢失。这些竞争因素决定了权重的理想初始大小。

关于如何初始化网络，正则化和优化有着非常不同的观点。优化观点建议权重应该足够大以成功传播信息，但是正则化希望其小一点。诸如随机梯度下降这类对权重较小的增量更新，趋于停止在更靠近初始参数的区域（不管是由于卡在低梯度的区域，还是由于触发了基于过拟合的提前终止准则）的优化算法倾向于最终参数应接近于初始参数。回顾第7.8节，在某些模型上，提前终止的梯度下降等价于权重衰减。在一般情况下，提前终止的梯度下降和权重衰减不同，但是提供了一个宽松的类比去考虑初始化的影响。我们可以将初始化参数  $\theta$  为  $\theta_0$  类比于强置均值为  $\theta_0$  的高斯先验  $p(\theta)$ 。从这个角度来看，选择  $\theta_0$  接近 0 是有道理的。这个先验表明，单元间彼此互不交互比交互更有可能。只有在目标函数的似然项表达出对交互很强的偏好时，单元才会交互。另一方面，如果我们初始化  $\theta_0$  为很大的值，那么我们的先验指定了哪些单元应互相交互，以及它们应如何交互。

有些启发式方法可用于选择权重的初始大小。一种初始化  $m$  个输入和  $n$  输出的全连接层的权重的启发式方法是从分布  $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$  中采样权重，而 Glorot and Bengio (2010) 建议使用 **标准初始化**（normalized initialization）

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right). \quad (8.23)$$

后一种启发式方法初始化所有的层，折衷于使其具有相同激活方差和使其具有相同梯度方差之间。这假设网络是不含非线性的链式矩阵乘法，据此推导得出。现实的神经网络显然会违反这个假设，但很多设计于线性模型的策略在其非线性对应中的效果也不错。

Saxe *et al.* (2013) 推荐初始化为随机正交矩阵，仔细挑选负责每一层非线性缩放或增益（gain）因子  $g$ 。他们得到了用于不同类型的非线性激活函数的特定缩放因子。这种初始化方案也是启发于不含非线性的矩阵相乘序列的深度网络。在该模型下，这个初始化方案保证了达到收敛所需的训练迭代总数独立于深度。

增加缩放因子  $g$  将网络推向网络前向传播时激活范数增加，反向传播时梯度范数增加的区域。Sussillo (2014) 表明，正确设置缩放因子足以训练深达 1000 层的网络，而不需要使用正交初始化。这种方法的一个重要观点是，在前馈网络中，激活和梯度会在每一步前向传播或反向传播中增加或缩小，遵循随机游走行为。这是因为前馈网络在每一层使用了不同的权重矩阵。如果该随机游走调整到保持范数，那么前馈网络能够很大程度地避免相同权重矩阵用于每层的梯度消失与爆炸问题，如第 8.2.5 节所述。

可惜，这些初始权重的最佳准则往往不会带来最佳效果。这可能有三种不同的原因。首先，我们可能使用了错误的标准——它实际上并不利于保持整个网络信号的范数。其次，初始化时强加的性质可能在学习开始进行后不能保持。最后，该标准可能成功提高了优化速度，但意外地增大了泛化误差。在实践中，我们通常需要将权重范围视为超参数，其最优值大致接近，但并不完全等于理论预测。

数值范围准则的一个缺点是，设置所有的初始权重具有相同的标准差，例如  $\frac{1}{\sqrt{m}}$ ，会使得层很大时每个单一权重会变得极其小。Martens (2010) 提出了一种被称为 **稀疏初始化** (sparse initialization) 的替代方案，每个单元初始化为恰好有  $k$  个非零权重。这个想法保持该单元输入的总数量独立于输入数目  $m$ ，而不使单一权重元素的大小随  $m$  缩小。稀疏初始化有助于实现单元之间在初始化时更具多样性。但是，获得较大取值的权重也同时被加了很强的先验。因为梯度下降需要很长时间缩小“不正确”的大值，这个初始化方案可能会导致某些单元出问题，例如 maxout 单元有几个过滤器，互相之间必须仔细调整。

计算资源允许的话，将每层权重的初始数值范围设为超参数通常是个好主意，使用第 11.4.2 节介绍的超参数搜索算法，如随机搜索，挑选这些数值范围。是否选择使用密集或稀疏初始化也可以设为一个超参数。作为替代，我们可以手动搜索最优初始范围。一个好的挑选初始数值范围的经验法则是观测单个小批量数据上的激活或梯度的幅度或标准差。如果权重太小，那么当激活值在小批量上前向传播于网络时，激活值的幅度会缩小。通过重复识别具有小得不可接受的激活值的第一层，并提高其权重，最终有可能得到一个初始激活全部合理的网络。如果学习在这点上仍然很慢，观测梯度的幅度或标准差可能也会有所帮助。这个过程原则上是自动的，且通常计算量低于基于验证集误差的超参数优化，因为它是基于初始模型在单批数据上的行为反馈，而不是在验证集上训练模型的反馈。由于这个协议很长时间都被启发式使用，最近 Mishkin and Matas (2015) 更正式地研究了该协议。

目前为止，我们关注在权重的初始化上。幸运的是，其他参数的初始化通常更

容易。

设置偏置的方法必须和设置权重的方法协调。设置偏置为零通常在大多数权重初始化方案中是可行的。存在一些我们可能设置偏置为非零值的情况：

- 如果偏置是作为输出单元，那么初始化偏置以获取正确的输出边缘统计通常是有利的。要做到这一点，我们假设初始权重足够小，该单元的输出仅由偏置决定。这说明设置偏置为应用于训练集上输出边缘统计的激活函数的逆。例如，如果输出是类上的分布，且该分布是高度偏态分布，第  $i$  类的边缘概率由某个向量  $\mathbf{c}$  的第  $i$  个元素给定，那么我们可以通过求解方程  $\text{softmax}(\mathbf{b}) = \mathbf{c}$  来设置偏置向量  $\mathbf{b}$ 。这不仅适用于分类器，也适用于我们将在第三部分遇到的模型，例如自编码器和玻尔兹曼机。这些模型拥有输出类似于输入数据  $\mathbf{x}$  的网络层，非常有助于初始化这些层的偏置以匹配  $\mathbf{x}$  上的边缘分布。
- 有时，我们可能想要选择偏置以避免初始化引起太大饱和。例如，我们可能会将 ReLU 的隐藏单元设为 0.1 而非 0，以避免 ReLU 在初始化时饱和。尽管这种方法违背不希望偏置具有很强输入的权重初始化准则。例如，不建议使用随机游走初始化 (Sussillo, 2014)。
- 有时，一个单元会控制其他单元能否参与到等式中。在这种情况下，我们有一个单元输出  $u$ ，另一个单元  $h \in [0, 1]$ ，那么我们可以将  $h$  视作门，以决定  $uh \approx 1$  还是  $uh \approx 0$ 。在这种情形下，我们希望设置偏置  $h$ ，使得在初始化的大多数情况下  $h \approx 1$ 。否则， $u$  没有机会学习。例如，Jozefowicz *et al.* (2015) 提议设置 LSTM 模型遗忘门的偏置为 1，如第 10.10 节所述。

另一种常见类型的参数是方差或精确度参数。例如，我们用以下模型进行带条件方差估计的线性回归

$$p(y | \mathbf{x}) = \mathcal{N}(y | \mathbf{w}^\top \mathbf{x} + b, 1/\beta), \quad (8.24)$$

其中  $\beta$  是精确度参数。通常我们能安全地初始化方差或精确度参数为 1。另一种方法假设初始权重足够接近零，设置偏置可以忽略权重的影响，然后设定偏置以产生输出的正确边缘均值，并将方差参数设置为训练集输出的边缘方差。

除了这些初始化模型参数的简单常数或随机方法，还有可能使用机器学习初始化模型参数。在本书第三部分讨论的一个常用策略是使用相同的输入数据集，用无监督模型训练出来的参数来初始化监督模型。我们也可以在相关问题上使用监督训

练。即使是在一个不相关的任务上运行监督训练，有时也能得到一个比随机初始化具有更快收敛率的初始值。这些初始化策略有些能够得到更快的收敛率和更好的泛化误差，因为它们编码了模型初始参数的分布信息。其他策略显然效果不错的原因主要在于它们设置参数为正确的数值范围，或是设置不同单元计算互相不同的函数。

## 8.5 自适应学习率算法

神经网络研究员早就意识到学习率肯定是可以设置的超参数之一，因为它对模型的性能有显著的影响。正如我们在第 4.3 节和第 8.2 节中所探讨的，损失通常高度敏感于参数空间中的某些方向，而不敏感于其他。动量算法可以在一定程度缓解这些问题，但这样做的代价是引入了另一个超参数。在这种情况下，自然会问有没有其他方法。如果我们相信方向敏感度在某种程度上是轴对齐的，那么每个参数设置不同的学习率，在整个学习过程中自动适应这些学习率是有道理的。

**Delta-bar-delta** 算法 (Jacobs, 1988) 是一个早期的在训练时适应模型参数各自学习率的启发式方法。该方法基于一个很简单的想法，如果损失对于某个给定模型参数的偏导保持相同的符号，那么学习率应该增加。如果对于该参数的偏导变化了符号，那么学习率应减小。当然，这种方法只能应用于全批量优化中。

最近，提出了一些增量（或者基于小批量）的算法来自适应模型参数的学习率。这节将简要回顾其中一些算法。

### 8.5.1 AdaGrad

**AdaGrad** 算法，如算法 8.4 所示，独立地适应所有模型参数的学习率，缩放每个参数反比于其所有梯度历史平方值总和的平方根 (Duchi *et al.*, 2011)。具有损失最大偏导的参数相应地有一个快速下降的学习率，而具有小偏导的参数在学习率上有相对较小的下降。净效果是在参数空间中更为平缓的倾斜方向会取得更大的进步。

在凸优化背景中，AdaGrad 算法具有一些令人满意的理论性质。然而，经验上已经发现，对于训练深度神经网络模型而言，从训练开始时积累梯度平方会导致有效学习率过早和过量的减小。AdaGrad 在某些深度学习模型上效果不错，但不是全部。

---

**算法 8.4 AdaGrad 算法**

---

**Require:** 全局学习率  $\epsilon$ **Require:** 初始参数  $\theta$ **Require:** 小常数  $\delta$ , 为了数值稳定大约设为  $10^{-7}$ 初始化梯度累积变量  $r = 0$ **while** 没有达到停止准则 **do**从训练集中采包含  $m$  个样本  $\{x^{(1)}, \dots, x^{(m)}\}$  的小批量, 对应目标为  $y^{(i)}$ 。计算梯度:  $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$ 累积平方梯度:  $r \leftarrow r + g \odot g$ 计算更新:  $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$  (逐元素地应用除和求平方根)应用更新:  $\theta \leftarrow \theta + \Delta\theta$ **end while**

---

### 8.5.2 RMSProp

**RMSProp** 算法 (Hinton, 2012) 修改 AdaGrad 以在非凸设定下效果更好, 改变梯度积累为指数加权的移动平均。AdaGrad 旨在应用于凸问题时快速收敛。当应用于非凸函数训练神经网络时, 学习轨迹可能穿过了很多不同的结构, 最终到达一个局部是凸碗的区域。AdaGrad 根据平方梯度的整个历史收缩学习率, 可能使得学习率在达到这样的凸结构前就变得太小了。RMSProp 使用指数衰减平均以丢弃遥远过去的历史, 使其能够在找到凸碗状结构后快速收敛, 它就像一个初始化于该碗状结构的 AdaGrad 算法实例。

RMSProp 的标准形式如算法 8.5 所示, 结合 Nesterov 动量的形式如算法 8.6 所示。相比于 AdaGrad, 使用移动平均引入了一个新的超参数  $\rho$ , 用来控制移动平均的长度范围。

经验上, RMSProp 已被证明是一种有效且实用的深度神经网络优化算法。目前它是深度学习从业者经常采用的优化方法之一。

### 8.5.3 Adam

**Adam** (Kingma and Ba, 2014) 是另一种学习率自适应的优化算法, 如算法 8.7 所示。“Adam”这个名字派生自短语“adaptive moments”。早期算法背景下, 它也许

---

算法 8.5 RMSProp 算法

---

**Require:** 全局学习率  $\epsilon$ , 衰减速率  $\rho$

**Require:** 初始参数  $\theta$

**Require:** 小常数  $\delta$ , 通常设为  $10^{-6}$  (用于被小数除时的数值稳定)

初始化累积变量  $r = 0$

**while** 没有达到停止准则 **do**

    从训练集中采包含  $m$  个样本  $\{x^{(1)}, \dots, x^{(m)}\}$  的小批量, 对应目标为  $y^{(i)}$ 。

    计算梯度:  $g \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

    累积平方梯度:  $r \leftarrow \rho r + (1 - \rho)g \odot g$

    计算参数更新:  $\Delta\theta = -\frac{\epsilon}{\sqrt{\delta+r}} \odot g$  ( $\frac{1}{\sqrt{\delta+r}}$  逐元素应用)

    应用更新:  $\theta \leftarrow \theta + \Delta\theta$

**end while**

---

最好被看作结合 RMSProp 和具有一些重要区别的动量的变种。首先, 在 Adam 中, 动量直接并入了梯度一阶矩 (指数加权) 的估计。将动量加入 RMSProp 最直观的方法是将动量应用于缩放后的梯度。结合缩放的动量使用没有明确的理论动机。其次, Adam 包括偏置修正, 修正从原点初始化的一阶矩 (动量项) 和 (非中心的) 二阶矩的估计 (算法 8.7)。RMSProp 也采用了 (非中心的) 二阶矩估计, 然而缺失了修正因子。因此, 不像 Adam, RMSProp 二阶矩估计可能在训练初期有很高的偏置。Adam 通常被认为对超参数的选择相当鲁棒, 尽管学习率有时需要从建议的默认修改。

#### 8.5.4 选择正确的优化算法

在本节中, 我们讨论了一系列算法, 通过自适应每个模型参数的学习率以解决优化深度模型中的难题。此时, 一个自然的问题是: 该选择哪种算法呢?

遗憾的是, 目前在这一点上没有达成共识。Schaul *et al.* (2014) 展示了许多优化算法在大量学习任务上极具价值的比较。虽然结果表明, 具有自适应学习率 (以 RMSProp 和 AdaDelta 为代表) 的算法族表现得相当鲁棒, 不分伯仲, 但没有哪个算法能脱颖而出。

目前, 最流行并且使用很高的优化算法包括 SGD、具动量的 SGD、RMSProp、具动量的 RMSProp、AdaDelta 和 Adam。此时, 选择哪一个算法似乎主要取决于

---

**算法 8.6 使用 Nesterov 动量的 RMSProp 算法**

---

**Require:** 全局学习率  $\epsilon$ , 衰减速率  $\rho$ , 动量系数  $\alpha$

**Require:** 初始参数  $\theta$ , 初始参数  $v$

初始化累积变量  $r = 0$

**while** 没有达到停止准则 **do**

从训练集中采包含  $m$  个样本  $\{x^{(1)}, \dots, x^{(m)}\}$  的小批量, 对应目标为  $y^{(i)}$ 。

计算临时更新:  $\tilde{\theta} \leftarrow \theta + \alpha v$

计算梯度:  $g \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L(f(x^{(i)}; \tilde{\theta}), y^{(i)})$

累积梯度:  $r \leftarrow \rho r + (1 - \rho)g \odot g$

计算速度更新:  $v \leftarrow \alpha v - \frac{\epsilon}{\sqrt{r}} \odot g$  ( $\frac{1}{\sqrt{r}}$  逐元素应用)

应用更新:  $\theta \leftarrow \theta + v$

**end while**

---

使用者对算法的熟悉程度 (以便调节超参数)。

---

**算法 8.7 Adam 算法**

---

**Require:** 步长  $\epsilon$  (建议默认为: 0.001)

**Require:** 矩估计的指数衰减速率,  $\rho_1$  和  $\rho_2$  在区间  $[0, 1)$  内。 (建议默认为: 分别为 0.9 和 0.999)

**Require:** 用于数值稳定的小常数  $\delta$  (建议默认为:  $10^{-8}$ )

**Require:** 初始参数  $\theta$

初始化一阶和二阶矩变量  $s = 0, r = 0$

初始化时间步  $t = 0$

**while** 没有达到停止准则 **do**

从训练集中采包含  $m$  个样本  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  的小批量, 对应目标为  $\mathbf{y}^{(i)}$ 。

计算梯度:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

$t \leftarrow t + 1$

更新有偏一阶矩估计:  $s \leftarrow \rho_1 s + (1 - \rho_1) \mathbf{g}$

更新有偏二阶矩估计:  $r \leftarrow \rho_2 r + (1 - \rho_2) \mathbf{g} \odot \mathbf{g}$

修正一阶矩的偏差:  $\hat{s} \leftarrow \frac{s}{1 - \rho_1^t}$

修正二阶矩的偏差:  $\hat{r} \leftarrow \frac{r}{1 - \rho_2^t}$

计算更新:  $\Delta \theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$  (逐元素应用操作)

应用更新:  $\theta \leftarrow \theta + \Delta \theta$

**end while**

---

## 8.6 二阶近似方法

在本节中, 我们会讨论训练深度神经网络的二阶方法。参考LeCun *et al.* (1998a) 了解该问题的早期处理方法。为表述简单起见, 我们只考察目标函数为经验风险:

$$J(\theta) = \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}(\mathbf{x}, \mathbf{y})}[L(f(\mathbf{x}; \theta), \mathbf{y})] = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}). \quad (8.25)$$

然而, 我们在这里讨论的方法很容易扩展到更一般的目标函数, 例如, 第七章讨论的包括参数正则项的函数。

### 8.6.1 牛顿法

在第 4.3 节，我们介绍了二阶梯度方法。与一阶方法相比，二阶方法使用二阶导数改进了优化。最广泛使用的二阶方法是牛顿法。我们现在更详细地描述牛顿法，重点在其应用于神经网络的训练。

牛顿法是基于二阶泰勒级数展开在某点  $\theta_0$  附近来近似  $J(\theta)$  的优化方法，其忽略了高阶导数：

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^\top \nabla_{\theta} J(\theta_0) + \frac{1}{2} (\theta - \theta_0)^\top H(\theta - \theta_0), \quad (8.26)$$

其中  $H$  是  $J$  相对于  $\theta$  的 Hessian 矩阵在  $\theta_0$  处的估计。如果我们再求解这个函数的临界点，我们将得到牛顿参数更新规则：

$$\theta^* = \theta_0 - H^{-1} \nabla_{\theta} J(\theta_0). \quad (8.27)$$

因此，对于局部的二次函数（具有正定的  $H$ ），用  $H^{-1}$  重新调整梯度，牛顿法会直接跳到极小值。如果目标函数是凸的但非二次的（有高阶项），该更新将是迭代的，得到和牛顿法相关的算法，如算法 8.8 所示。

对于非二次的表面，只要 Hessian 矩阵保持正定，牛顿法能够迭代地应用。这意味着一个两步迭代过程。首先，更新或计算 Hessian 逆（通过更新二阶近似）。其次，根据式 (8.27) 更新参数。

在第 8.2.3 节，我们讨论了牛顿法只适用于 Hessian 矩阵是正定的情况。在深度学习中，目标函数的表面通常非凸（有很多特征），如鞍点。因此使用牛顿法是有问题的。如果 Hessian 矩阵的特征值并不都是正的，例如，靠近鞍点处，牛顿法实际上会导致更新朝错误的方向移动。这种情况可以通过正则化 Hessian 矩阵来避免。常用的正则化策略包括在 Hessian 矩阵对角线上增加常数  $\alpha$ 。正则化更新变为

$$\theta^* = \theta_0 - [H(f(\theta_0)) + \alpha I]^{-1} \nabla_{\theta} f(\theta_0). \quad (8.28)$$

这个正则化策略用于牛顿法的近似，例如 Levenberg-Marquardt 算法 (Levenberg, 1944; Marquardt, 1963)，只要 Hessian 矩阵的负特征值仍然相对接近零，效果就会很好。在曲率方向更极端的情况下， $\alpha$  的值必须足够大，以抵消负特征值。然而，如果  $\alpha$  持续增加，Hessian 矩阵会变得由对角矩阵  $\alpha I$  主导，通过牛顿法所选择的方向会收敛到普通梯度除以  $\alpha$ 。当很强的负曲率存在时， $\alpha$  可能需要特别大，以致于牛顿法比选择合适学习率的梯度下降的步长更小。

---

**算法 8.8** 目标为  $J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$  的牛顿法

---

**Require:** 初始参数  $\boldsymbol{\theta}_0$

**Require:** 包含  $m$  个样本的训练集

**while** 没有达到停止准则 **do**

计算梯度:  $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$

计算 Hessian 矩阵:  $\mathbf{H} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}}^2 \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$

计算 Hessian 逆:  $\mathbf{H}^{-1}$

计算更新:  $\Delta \boldsymbol{\theta} = -\mathbf{H}^{-1} \mathbf{g}$

应用更新:  $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$

**end while**

---

除了目标函数的某些特征带来的挑战，如鞍点，牛顿法用于训练大型神经网络还受限于其显著的计算负担。Hessian 矩阵中元素数目是参数数量的平方，因此，如果参数数目为  $k$ （甚至是在非常小的神经网络中  $k$  也可能是百万级别），牛顿法需要计算  $k \times k$  矩阵的逆，计算复杂度为  $O(k^3)$ 。另外，由于参数将每次更新都会改变，每次训练迭代都需要计算 Hessian 矩阵的逆。其结果是，只有参数很少的网络才能在实际中用牛顿法训练。在本节的剩余部分，我们将讨论一些试图保持牛顿法优点，同时避免计算障碍的替代算法。

### 8.6.2 共轭梯度

共轭梯度是一种通过迭代下降的共轭方向（conjugate directions）以有效避免 Hessian 矩阵求逆计算的方法。这种方法的灵感来自于对最速下降方法弱点的仔细研究（详细信息请查看第 4.3 节），其中线搜索迭代地用于与梯度相关的方向上。图 8.6 说明了该方法在二次碗型目标中如何表现的，是一个相当低效的来回往复，锯齿形模式。这是因为每一个由梯度给定的线搜索方向，都保证正交于上一个线搜索方向。

假设上一个搜索方向是  $\mathbf{d}_{t-1}$ 。在极小值处，线搜索终止，方向  $\mathbf{d}_{t-1}$  处的方向导数为零:  $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \cdot \mathbf{d}_{t-1} = 0$ 。因为该点的梯度定义了当前的搜索方向， $\mathbf{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$  将不会贡献于方向  $\mathbf{d}_{t-1}$ 。因此方向  $\mathbf{d}_t$  正交于  $\mathbf{d}_{t-1}$ 。最速下降多次迭代中，方向  $\mathbf{d}_{t-1}$  和  $\mathbf{d}_t$  之间的关系如图 8.6 所示。如图展示的，下降正交方向的选择不会保持前一搜索方向上的最小值。这产生了锯齿形的过程。在当前梯度方向下降到极小值，我们

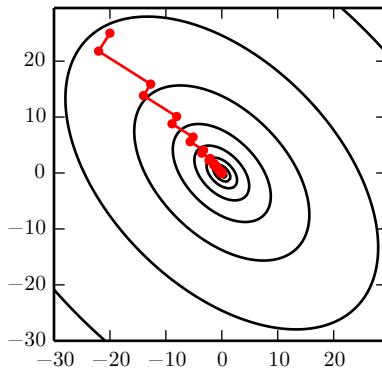


图 8.6: 将最速下降法应用于二次代价表面。在每个步骤，最速下降法沿着由初始点处的梯度定义的线跳到最低代价的点。这解决了图 4.6 中使用固定学习率所遇到的一些问题，但即使使用最佳步长，算法仍然朝最优方向曲折前进。根据定义，在沿着给定方向的目标最小值处，最终点处的梯度与该方向正交。

必须重新最小化之前梯度方向上的目标。因此，通过遵循每次线搜索结束时的梯度，我们在某种程度上撤销了在之前线搜索的方向上取得的进展。共轭梯度试图解决这个问题。

在共轭梯度法中，我们寻求一个和先前线搜索方向 共轭 (conjugate) 的搜索方向，即它不会撤销该方向上的进展。在训练迭代  $t$  时，下一步的搜索方向  $d_t$  的形式如下：

$$d_t = \nabla_{\theta} J(\theta) + \beta_t d_{t-1}, \quad (8.29)$$

其中，系数  $\beta_t$  的大小控制我们应沿方向  $d_{t-1}$  加回多少到当前搜索方向上。

如果  $d_t^\top H d_{t-1} = 0$ ，其中  $H$  是 Hessian 矩阵，则两个方向  $d_t$  和  $d_{t-1}$  被称为共轭的。

适应共轭的直接方法会涉及到  $H$  特征向量的计算以选择  $\beta_t$ 。这将无法满足我们的开发目标：寻找在大问题比牛顿法计算更加可行的方法。我们能否不进行这些计算而得到共轭方向？幸运的是这个问题的答案是肯定的。

两种用于计算  $\beta_t$  的流行方法是：

1. Fletcher-Reeves:

$$\beta_t = \frac{\nabla_{\theta} J(\theta_t)^\top \nabla_{\theta} J(\theta_t)}{\nabla_{\theta} J(\theta_{t-1})^\top \nabla_{\theta} J(\theta_{t-1})} \quad (8.30)$$

## 2. Polak-Ribière:

$$\beta_t = \frac{(\nabla_{\theta} J(\theta_t) - \nabla_{\theta} J(\theta_{t-1}))^\top \nabla_{\theta} J(\theta_t)}{\nabla_{\theta} J(\theta_{t-1})^\top \nabla_{\theta} J(\theta_{t-1})} \quad (8.31)$$

对于二次曲面而言，共轭方向确保梯度沿着前一方向大小不变。因此，我们在前一方向上仍然是极小值。其结果是，在  $k$ -维参数空间中，共轭梯度只需要至多  $k$  次线搜索就能达到极小值。共轭梯度算法如算法 8.9 所示。

**算法 8.9 共轭梯度方法**

**Require:** 初始参数  $\theta_0$

**Require:** 包含  $m$  个样本的训练集

初始化  $\rho_0 = 0$

初始化  $g_0 = 0$

初始化  $t = 1$

**while** 没有达到停止准则 **do**

    初始化梯度  $\mathbf{g}_t = 0$

    计算梯度： $\mathbf{g}_t \leftarrow \frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

    计算  $\beta_t = \frac{(\mathbf{g}_t - \mathbf{g}_{t-1})^\top \mathbf{g}_t}{\mathbf{g}_{t-1}^\top \mathbf{g}_{t-1}}$  (Polak-Ribière)

    (非线性共轭梯度：视情况可重置  $\beta_t$  为零，例如  $t$  是常数  $k$  的倍数时，如  $k = 5$ )

    计算搜索方向： $\rho_t = -\mathbf{g}_t + \beta_t \rho_{t-1}$

    执行线搜索寻找： $\epsilon^* = \operatorname{argmin}_{\epsilon} \frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \theta_t + \epsilon \rho_t), \mathbf{y}^{(i)})$

    (对于真正二次的代价函数，存在  $\epsilon^*$  的解析解，而无需显式地搜索)

    应用更新： $\theta_{t+1} = \theta_t + \epsilon^* \rho_t$

$t \leftarrow t + 1$

**end while**

**非线性共轭梯度：** 目前，我们已经讨论了用于二次目标函数的共轭梯度法。当然，本章我们主要关注于探索训练神经网络和其他相关深度学习模型的优化方法，其对应的目标函数比二次函数复杂得多。或许令人惊讶，共轭梯度法在这种情况下仍然是适用的，尽管需要作一些修改。没有目标是二次的保证，共轭方向也不再保证在以前方向上的目标仍是极小值。其结果是，**非线性共轭梯度**算法会包括一些偶尔的重设，共轭梯度法沿未修改的梯度重启线搜索。

实践者报告在实践中使用非线性共轭梯度算法训练神经网络是合理的，尽管在开始非线性共轭梯度前使用随机梯度下降迭代若干步来初始化效果更好。另外，尽管（非线性）共轭梯度算法传统上作为批方法，小批量版本已经成功用于训练神经网络 (Le *et al.*, 2011)。针对神经网路的共轭梯度应用早已被提出，例如缩放的共轭梯度算法 (Moller, 1993)。

### 8.6.3 BFGS

**Broyden-Fletcher-Goldfarb-Shanno (BFGS)** 算法具有牛顿法的一些优点，但没有牛顿法的计算负担。在这方面，BFGS 和 CG 很像。然而，BFGS 使用了一个更直接的方法近似牛顿更新。回顾牛顿更新由下式给出

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0), \quad (8.32)$$

其中， $\mathbf{H}$  是  $J$  相对于  $\boldsymbol{\theta}$  的 Hessian 矩阵在  $\boldsymbol{\theta}_0$  处的估计。运用牛顿法的主要计算难点在于计算 Hessian 逆  $\mathbf{H}^{-1}$ 。拟牛顿法所采用的方法 (BFGS 是其中最突出的) 是使用矩阵  $\mathbf{M}_t$  近似逆，迭代地低秩更新精度以更好地近似  $\mathbf{H}^{-1}$ 。

BFGS 近似的说明和推导出现在很多关于优化的教科书中，包括 Luenberger (1984)。

当 Hessian 逆近似  $\mathbf{M}_t$  更新时，下降方向  $\boldsymbol{\rho}_t$  为  $\boldsymbol{\rho}_t = \mathbf{M}_t \mathbf{g}_t$ 。该方向上的线搜索用于决定该方向上的步长  $\epsilon^*$ 。参数的最后更新为：

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t. \quad (8.33)$$

和共轭梯度法相似，BFGS 算法迭代一系列线搜索，其方向含二阶信息。然而和共轭梯度不同的是，该方法的成功并不严重依赖于线搜索寻找该方向上和真正极小值很近的一点。因此，相比于共轭梯度，BFGS 的优点是其花费较少的时间改进每个线搜索。在另一方面，BFGS 算法必须存储 Hessian 逆矩阵  $\mathbf{M}$ ，需要  $O(n^2)$  的存储空间，使 BFGS 不适用于大多数具有百万级参数的现代深度学习模型。

**存储受限的 BFGS (或 L-BFGS)** 通过避免存储完整的 Hessian 逆近似  $\mathbf{M}$ ，BFGS 算法的存储代价可以显著降低。L-BFGS 算法使用和 BFGS 算法相同的方法计算  $\mathbf{M}$  的近似，但起始假设是  $\mathbf{M}^{(t-1)}$  是单位矩阵，而不是一步一步都要存储近似。如果使用精确的线搜索，L-BFGS 定义的方向会是相互共轭的。然而，不同于共轭梯

度法，即使只是近似线搜索的极小值，该过程的效果仍然不错。这里描述的无存储的 L-BFGS 方法可以拓展为包含 Hessian 矩阵更多的信息，每步存储一些用于更新  $\mathbf{M}$  的向量，且每步的存储代价是  $O(n)$ 。

## 8.7 优化策略和元算法

许多优化技术并非真正的算法，而是一般化的模板，可以特定地产生算法，或是并入到很多不同的算法中。

### 8.7.1 批标准化

批标准化 (Ioffe and Szegedy, 2015) 是优化深度神经网络中最激动人心的最新创新之一。实际上它并不是一个优化算法，而是一个自适应的重参数化的方法，试图解决训练非常深的模型的困难。

非常深的模型会涉及多个函数或层组合。在其他层不改变的假设下，梯度用于如何更新每一个参数。在实践中，我们同时更新所有层。当我们进行更新时，可能会发生一些意想不到的结果，这是因为许多组合在一起的函数同时改变时，计算更新的假设是其他函数保持不变。举一个简单的例子，假设我们有一个深度神经网络，每一层只有一个单元，并且在每个隐藏层不使用激活函数： $\hat{y} = xw_1w_2w_3 \dots w_l$ 。此处， $w_i$  表示用于层  $i$  的权重。层  $i$  的输出是  $h_i = h_{i-1}w_i$ 。输出  $\hat{y}$  是输入  $x$  的线性函数，但是权重  $w_i$  的非线性函数。假设我们的代价函数  $\hat{y}$  上的梯度为 1，所以我们希望稍稍降低  $\hat{y}$ 。然后反向传播算法可以计算梯度  $\mathbf{g} = \nabla_{\mathbf{w}}\hat{y}$ 。想想我们在更新  $\mathbf{w} \leftarrow \mathbf{w} - \epsilon \mathbf{g}$  时会发生什么。近似  $\hat{y}$  的一阶泰勒级数会预测  $\hat{y}$  的值下降  $\epsilon \mathbf{g}^\top \mathbf{g}$ 。如果我们希望  $\hat{y}$  下降 0.1，那么梯度中的一阶信息表明我们应设置学习率  $\epsilon$  为  $\frac{0.1}{\mathbf{g}^\top \mathbf{g}}$ 。然而，实际的更新将包括二阶，三阶，直到  $l$  阶的影响。 $\hat{y}$  的更新值为

$$x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \dots (w_l - \epsilon g_l), \quad (8.34)$$

这个更新中所产生的一个二阶项示例是  $\epsilon^2 g_1 g_2 \prod_{i=3}^l w_i$ 。如果  $\prod_{i=3}^l w_i$  很小，那么该项可以忽略不计。而如果层 3 到层  $l$  的权重都比 1 大时，该项可能会指数级大。这使得我们很难选择一个合适的学习率，因为某一层中参数更新的效果很大程度上取决于其他所有层。二阶优化算法通过考虑二阶相互影响来解决这个问题，但我们可以看到，在非常深的网络中，更高阶的相互影响会很显著。即使是二阶优化算法，计

算代价也很高，并且通常需要大量近似，以免真正计算所有的重要二阶相互作用。因此对于  $n > 2$  的情况，建立  $n$  阶优化算法似乎是无望的。那么我们可以做些什么呢？

批标准化提出了一种几乎可以重参数化所有深度网络的优雅方法。重参数化显著减少了多层之间协调更新的问题。批标准化可应用于网络的任何输入层或隐藏层。设  $\mathbf{H}$  是需要标准化的某层的小批量激活函数，排布为设计矩阵，每个样本的激活出现在矩阵的每一行中。为了标准化  $\mathbf{H}$ ，我们将其替换为

$$\mathbf{H}' = \frac{\mathbf{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \quad (8.35)$$

其中  $\boldsymbol{\mu}$  是包含每个单元均值的向量， $\boldsymbol{\sigma}$  是包含每个单元标准差的向量。此处的算术是基于广播向量  $\boldsymbol{\mu}$  和向量  $\boldsymbol{\sigma}$  应用于矩阵  $\mathbf{H}$  的每一行。在每一行内，运算是逐元素的，因此  $H_{i,j}$  标准化为减去  $\mu_j$  再除以  $\sigma_j$ 。网络的其余部分操作  $\mathbf{H}'$  的方式和原网络操作  $\mathbf{H}$  的方式一样。

在训练阶段，

$$\boldsymbol{\mu} = \frac{1}{m} \sum_i \mathbf{H}_{i,:}, \quad (8.36)$$

和

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m} \sum_i (\mathbf{H} - \boldsymbol{\mu})_i^2}, \quad (8.37)$$

其中  $\delta$  是个很小的正值，比如  $10^{-8}$ ，以强制避免遇到  $\sqrt{z}$  的梯度在  $z = 0$  处未定义的问题。至关重要的是，我们反向传播这些操作，来计算均值和标准差，并应用它们于标准化  $\mathbf{H}$ 。这意味着，梯度不会再简单地增加  $h_i$  的标准差或均值；标准化操作会除掉这一操作的影响，归零其在梯度中的元素。这是批标准化方法的一个重大创新。以前的方法添加代价函数的惩罚，以鼓励单元标准化激活统计量，或是在每个梯度下降步骤之后重新标准化单元统计量。前者通常会导致不完全的标准化，而后者通常会显著地消耗时间，因为学习算法会反复改变均值和方差而标准化步骤会反复抵消这种变化。批标准化重参数化模型，以使一些单元总是被定义标准化，巧妙地回避了这两个问题。

在测试阶段， $\boldsymbol{\mu}$  和  $\boldsymbol{\sigma}$  可以被替换为训练阶段收集的运行均值。这使得模型可以对单一样本评估，而无需使用定义于整个小批量的  $\boldsymbol{\mu}$  和  $\boldsymbol{\sigma}$ 。

回顾例子  $\hat{y} = xw_1w_2 \dots w_l$ ，我们看到，我们可以通过标准化  $h_{l-1}$  很大程度地解决了学习这个模型的问题。假设  $x$  采样自一个单位高斯。那么  $h_{l-1}$  也是来自高斯，因为从  $x$  到  $h_l$  的变换是线性的。然而， $h_{l-1}$  不再有零均值和单位方差。使用批

标准化后，我们得到的归一化  $\hat{h}_{l-1}$  恢复了零均值和单位方差的特性。对于底层的几乎任意更新而言， $\hat{h}_{l-1}$  仍然保持着单位高斯。然后输出  $\hat{y}$  可以学习为一个简单的线性函数  $\hat{y} = w_l \hat{h}_{l-1}$ 。现在学习这个模型非常简单，因为低层的参数在大多数情况下没有什么影响；它们的输出总是重新标准化为单位高斯。只在少数个例中，低层会有影响。改变某个低层权重为 0，可能使输出退化；改变低层权重的符号可能反转  $\hat{h}_{l-1}$  和  $y$  之间的关系。这些情况都是非常罕见的。没有标准化，几乎每一个更新都会对  $h_{l-1}$  的统计量有着极端的影响。因此，批标准化显著地使得模型更易学习。在这个示例中，容易学习的代价是使得底层网络没有用。在我们的线性示例中，较低层不再有任何有害的影响，但它们也不再有任何有益的影响。这是因为我们已经标准化了一阶和二阶统计量，这是线性网络可以影响的所有因素。在具有非线性激活函数的深度神经网络中，较低层可以进行数据的非线性变换，所以它们仍然是有用的。批标准化仅标准化每个单元的均值和方差，以稳定化学习，但允许单元和单个单元的非线性统计量之间的关系发生变化。

由于网络的最后一层能够学习线性变换，实际上我们可能希望移除一层内单元之间的所有线性关系。事实上，这是 Guillaume Desjardins (2015) 中采用的方法，为批标准化提供了灵感。令人遗憾的是，消除所有的线性关联比标准化各个独立单元的均值和标准差代价更高，因此批标准化仍是迄今最实用的方法。

标准化一个单元的均值和标准差会降低包含该单元的神经网络的表达能力。为了保持网络的表现力，通常会将批量隐藏单元激活  $\mathbf{H}$  替换为  $\gamma \mathbf{H}' + \beta$ ，而不是简单地使用标准化的  $\mathbf{H}'$ 。变量  $\gamma$  和  $\beta$  是允许新变量有任意均值和标准差的学习参数。乍一看，这似乎是无用的——为什么我们将均值设为 0，然后又引入参数允许它被重设为任意值  $\beta$ ？答案是新的参数可以表示旧参数作为输入的同一族函数，但是新参数有不同的学习动态。在旧参数中， $\mathbf{H}$  的均值取决于  $\mathbf{H}$  下层中参数的复杂关联。在新参数中， $\gamma \mathbf{H}' + \beta$  的均值仅由  $\beta$  确定。新参数很容易通过梯度下降来学习。

大多数神经网络层会采取  $\phi(\mathbf{XW} + \mathbf{b})$  的形式，其中  $\phi$  是某个固定的非线性激活函数，如整流线性变换。自然想到我们应该将批标准化应用于输入  $\mathbf{X}$  还是变换后的值  $\mathbf{XW} + \mathbf{b}$ 。Ioffe and Szegedy (2015) 推荐后者。更具体地， $\mathbf{XW} + \mathbf{b}$  应替换为  $\mathbf{XW}$  的标准化形式。偏置项应被忽略，因为参数  $\beta$  会加入批标准化重参数化，它是冗余的。一层的输入通常是前一层的非线性激活函数（如整流线性函数）的输出。因此，输入的统计量更符合非高斯，而更不服从线性操作的标准化。

第九章所述的卷积网络，在特征映射中每个空间位置同样地标准化  $\mu$  和  $\sigma$  是很重要的，能使特征映射的统计量在不同的空间位置，仍然保持相同。

### 8.7.2 坐标下降

在某些情况下，将一个优化问题分解成几个部分，可以更快地解决原问题。如果我们相对于某个单一变量  $x_i$  最小化  $f(\mathbf{x})$ ，然后相对于另一个变量  $x_j$  等等，反复循环所有的变量，我们会保证到达（局部）极小值。这种做法被称为 **坐标下降** (coordinate descent)，因为我们一次优化一个坐标。更一般地，**块坐标下降** (block coordinate descent) 是指对于某个子集的变量同时最小化。术语“坐标下降”通常既指块坐标下降，也指严格的单个坐标下降。

当优化问题中的不同变量能够清楚地分成相对独立的组，或是当优化一组变量明显比优化所有变量效率更高时，坐标下降最有意义。例如，考虑代价函数

$$J(\mathbf{H}, \mathbf{W}) = \sum_{i,j} |H_{i,j}| + \sum_{i,j} (\mathbf{X} - \mathbf{W}^\top \mathbf{H})_{i,j}^2. \quad (8.38)$$

该函数描述了一种被称为稀疏编码的学习问题，其目标是寻求一个权重矩阵  $\mathbf{W}$ ，可以线性解码激活值矩阵  $\mathbf{H}$  以重构训练集  $\mathbf{X}$ 。稀疏编码的大多数应用还涉及到权重衰减或  $\mathbf{W}$  列范数的约束，以避免极小  $\mathbf{H}$  和极大  $\mathbf{W}$  的病态解。

函数  $J$  不是凸的。然而，我们可以将训练算法的输入分成两个集合：字典参数  $\mathbf{W}$  和编码表示  $\mathbf{H}$ 。最小化关于这两者之一的任意一组变量的目标函数都是凸问题。因此，块坐标下降允许我们使用高效的凸优化算法，交替固定  $\mathbf{H}$  优化  $\mathbf{W}$  和固定  $\mathbf{W}$  优化  $\mathbf{H}$ 。

当一个变量的值很大程度地影响另一个变量的最优值时，坐标下降不是一个很好的方法，如函数  $f(\mathbf{x}) = (x_1 - x_2)^2 + \alpha(x_1^2 + x_2^2)$ ，其中  $\alpha$  是正值常数。第一项鼓励两个变量具有相似的值，而第二项鼓励它们接近零。解是两者都为零。牛顿法可以一步解决这个问题，因为它是一个正定二次问题。但是，对于小值  $\alpha$  而言，坐标下降会使进展非常缓慢，因为第一项不允许单个变量变为和其他变量当前值显著不同的值。

### 8.7.3 Polyak 平均

Polyak 平均 (Polyak and Juditsky, 1992) 会平均优化算法在参数空间访问轨迹中的几个点。如果  $t$  次迭代梯度下降访问了点  $\boldsymbol{\theta}^{(1)}, \dots, \boldsymbol{\theta}^{(t)}$ ，那么 Polyak 平均算法的输出是  $\hat{\boldsymbol{\theta}}^{(t)} = \frac{1}{t} \sum_i \boldsymbol{\theta}^{(i)}$ 。在某些问题中，如梯度下降应用于凸问题时，这种方法具有较强的收敛保证。当应用于神经网络时，其验证更多是启发式的，但在实践中表

现良好。基本想法是，优化算法可能会来回穿过山谷好几次而没经过山谷底部附近的点。尽管两边所有位置的均值应比较接近谷底。

在非凸问题中，优化轨迹的路径可以非常复杂，并且经过了许多不同的区域。包括参数空间中遥远过去的点，可能与当前点在代价函数上相隔很大的障碍，看上去不像一个有用的行为。其结果是，当应用 Polyak 平均于非凸问题时，通常会使用指数衰减计算平均值：

$$\hat{\theta}^{(t)} = \alpha \hat{\theta}^{(t-1)} + (1 - \alpha) \theta^{(t)}. \quad (8.39)$$

这个计算平均值的方法被用于大量数值应用中。最近的例子请查看 Szegedy *et al.* (2015)。

#### 8.7.4 监督预训练

有时，如果模型太复杂难以优化，或是如果任务非常困难，直接训练模型来解决特定任务的挑战可能太大。有时训练一个较简单的模型来求解问题，然后使模型更复杂会更有效。训练模型来求解一个简化的问题，然后转移到最后的问题，有时也会更有效些。这些在直接训练目标模型求解目标问题之前，训练简单模型求解简化问题的方法统称为 **预训练** (pretraining)。

**贪心算法** (greedy algorithm) 将问题分解成许多部分，然后独立地在每个部分求解最优值。令人遗憾的是，结合各个最佳的部分不能保证得到一个最佳的完整解。然而，贪心算法计算上比求解最优联合解的算法高效得多，并且贪心算法的解在不是最优的情况下，往往也是可以接受的。贪心算法也可以紧接一个 **精调** (fine-tuning) 阶段，联合优化算法搜索全问题的最优解。使用贪心解初始化联合优化算法，可以极大地加速算法，并提高寻找到的解的质量。

预训练算法，特别是贪心预训练，在深度学习中是普遍存在的。在本节中，我们会具体描述这些将监督学习问题分解成其他简化的监督学习问题的预训练算法。这种方法被称为 **贪心监督预训练** (greedy supervised pretraining)。

在贪心监督预训练的原始版本 (Bengio *et al.*, 2007c) 中，每个阶段包括一个仅涉及最终神经网络的子集层的监督学习训练任务。贪心监督预训练的一个例子如图 8.7 所示，其中每个附加的隐藏层作为浅层监督多层感知机的一部分预训练，以先前训练的隐藏层输出作为输入。Simonyan and Zisserman (2015) 预训练深度卷积网络 (11 层权重)，然后使用该网络前四层和最后三层初始化更深的网络 (多达 19 层

权重), 并非一次预训练一层。非常深的新网络的中间层是随机初始化的。然后联合训练新网络。还有一种选择, 由Yu *et al.* (2010) 提出, 将先前训练多层感知机的输出, 以及原始输入, 作为每个附加阶段的输入。

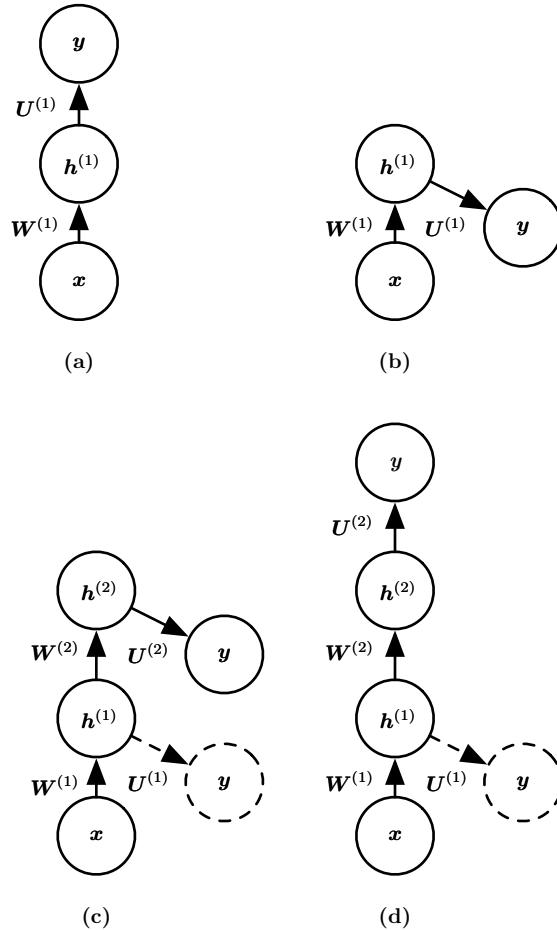


图 8.7: 一种形式的贪心监督预训练的示意图 (Bengio *et al.*, 2007a)。(a) 我们从训练一个足够浅的架构开始。(b) 同一个架构的另一描绘。(c) 我们只保留原始网络的输入到隐藏层, 并丢弃隐藏到输出层。我们将第一层隐藏层的输出作为输入发送到另一监督单隐层 MLP (使用与第一个网络相同的目标训练), 从而可以添加第二层隐藏层。这可以根据需要重复多层。(d) 所得架构的另一种描绘, 可视为前馈网络。为了进一步改进优化, 我们可以联合地精调所有层 (仅在该过程的结束或者该过程的每个阶段)。

为什么贪心监督预训练会有帮助呢? 最初由 Bengio *et al.* (2007d) 提出的假说

是，其有助于更好地指导深层结构的中间层的学习。一般情况下，预训练对于优化和泛化都是有帮助的。

另一个与监督预训练有关的方法扩展了迁移学习的想法：Yosinski *et al.* (2014) 在一组任务上预训练了 8 层权重的深度卷积网络（1000 个 ImageNet 对象类的子集），然而用该网络的前  $k$  层初始化同样规模的网络。然后第二个网络的所有层（上层随机初始化）联合训练以执行不同的任务（1000 个 ImageNet 对象类的另一个子集），但训练样本少于第一个任务。神经网络中另一个和迁移学习相关的方法将在第 15.2 节讨论。

另一条相关的工作线是 **FitNets** (Romero *et al.*, 2015) 方法。这种方法始于训练深度足够低和宽度足够大（每层单元数），容易训练的网络。然后，这个网络成为第二个网络（被指定为学生）的老师。学生网络更深更窄（11 至 19 层），且在正常情况下很难用 SGD 训练。训练学生网络不仅需要预测原任务的输出，还需要预测教师网络中间层的值，这样使得训练学生网络变得更容易。这个额外的任务说明了隐藏层应如何使用，并且能够简化优化问题。附加参数被引入来从更深的学生网络中间层去回归 5 层教师网络的中间层。然而，该目标是预测教师网络的中间隐藏层，并非预测最终分类目标。学生网络的低层因而具有两个目标：帮助学生网络的输出完成其目标和预测教师网络的中间层。尽管一个窄而深的网络似乎比宽而浅的网络更难训练，但窄而深网络的泛化能力可能更好，并且如果其足够窄，参数足够少，那么其计算代价更小。没有隐藏层的提示，学生网络在训练集和测试集上的实验表现都很差。因而中间层的提示是有助于训练很难训练的网络的方法之一，但是其他优化技术或是架构上的变化也可能解决这个问题。

### 8.7.5 设计有助于优化的模型

改进优化的最好方法并不总是改进优化算法。相反，深度模型中优化的许多改进来自于设计易于优化的模型。

原则上，我们可以使用呈锯齿非单调模式上上下下的激活函数，但是，这将使优化极为困难。在实践中，选择一族容易优化的模型比使用一个强大的优化算法更重要。神经网络学习在过去 30 年的大多数进步主要来自于改变模型族，而非改变优化过程。1980 年代用于训练神经网络的带动量的随机梯度下降，仍然是现代神经网络应用中的前沿算法。

具体来说，现代神经网络的设计选择体现在层之间的线性变换，几乎处处可导

的激活函数，和大部分定义域都有明显的梯度。特别地，创新的模型，如 LSTM，整流线性单元和 maxout 单元都比先前的模型（如基于 sigmoid 单元的深度网络）使用更多的线性函数。这些模型都具有简化优化的性质。如果线性变换的 Jacobian 具有相对合理的奇异值，那么梯度能够流经很多层。此外，线性函数在一个方向上一致增加，所以即使模型的输出远离正确值，也可以简单清晰地计算梯度，使其输出方向朝降低损失函数的方向移动。换言之，现代神经网络的设计方案旨在使其局部梯度信息合理地对应着移向一个遥远的解。

其他的模型设计策略有助于使优化更简单。例如，层之间的线性路径或是跳跃连接减少了从较低层参数到输出最短路径的长度，因而缓解了梯度消失的问题 (Srivastava *et al.*, 2015)。一个和跳跃连接相关的想法是添加和网络中间隐藏层相连的输出的额外副本，如 GoogLeNet (Szegedy *et al.*, 2014a) 和深度监督网络 (Lee *et al.*, 2014)。这些“辅助头”被训练来执行和网络顶层主要输出相同的任务，以确保底层网络能够接受较大的梯度。当训练完成时，辅助头可能被丢弃。这是之前小节介绍到的预训练策略的替代方法。以这种方式，我们可以在一个阶段联合训练所有层，而不改变架构，使得中间层（特别是低层）能够通过更短的路径得到一些如何更新的有用信息。这些信息为底层提供了误差信号。

### 8.7.6 延拓法和课程学习

正如第 8.2.7 节探讨的，许多优化挑战都来自于代价函数的全局结构，不能仅通过局部更新方向上更好的估计来解决。解决这个问题的主要方法是尝试初始化参数到某种区域内，该区域可以通过局部下降很快连接到参数空间中的解。

**延拓法** (continuation method) 是一族通过挑选初始点使优化更容易的方法，以确保局部优化花费大部分时间在表现良好的空间。延拓法的背后想法是构造一系列具有相同参数的目标函数。为了最小化代价函数  $J(\theta)$ ，我们构建新的代价函数  $\{J^{(0)}, \dots, J^{(n)}\}$ 。这些代价函数的难度逐步提高，其中  $J^{(0)}$  是最容易最小化的， $J^{(n)}$  是最难的，真正的代价函数驱动整个过程。当我们说  $J^{(i)}$  比  $J^{(i+1)}$  更容易时，是指其在更多的  $\theta$  空间上表现良好。随机初始化更有可能落入局部下降可以成功最小化代价函数的区域，因为其良好区域更大。这系列代价函数设计为前一个解是下一个的良好初始点。因此，我们首先解决一个简单的问题，然后改进解以解决逐步变难的问题，直到我们求解真正问题的解。

传统的延拓法（用于神经网络训练之前的延拓法）通常基于平滑目标函数。读

者可以查看 Wu (1997) 了解这类方法的示例，以及一些相关方法的综述。延拓法也和参数中加入噪声的模拟退火紧密相关 (Kirkpatrick *et al.*, 1983)。延拓法在最近几年非常成功。参考 Mobahi and Fisher (2015) 了解近期文献的概述，特别是在 AI 方面的应用。

传统上，延拓法主要用来克服局部极小值的问题。具体地，它被设计来在有很多局部极小值的情况下，求解一个全局最小点。这些连续方法会通过“模糊”原来的代价函数来构建更容易的代价函数。这些模糊操作可以是用采样来近似

$$J^{(i)}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\theta}' \sim \mathcal{N}(\boldsymbol{\theta}'; \boldsymbol{\theta}, \sigma^{(i)2})} J(\boldsymbol{\theta}') \quad (8.40)$$

这个方法的直觉是有些非凸函数在模糊后会近似凸的。在许多情况下，这种模糊保留了关于全局极小值的足够信息，我们可以通过逐步求解模糊更少的问题来求解全局极小值。这种方法有三种可能失败的方式。首先，它可能成功地定义了一连串代价函数，并从开始的一个凸函数起（逐一地）沿着函数链最佳轨迹逼近全局最小值，但可能需要非常多的逐步代价函数，整个过程的成本仍然很高。另外，即使延拓法可以适用，NP-hard 的优化问题仍然是 NP-hard。其他两种延拓法失败的原因是不实用。其一，不管如何模糊，函数都没法变成凸的，比如函数  $J(\boldsymbol{\theta}) = -\boldsymbol{\theta}^\top \boldsymbol{\theta}$ 。其二，函数可能在模糊后是凸的，但模糊函数的最小值可能会追踪到一个局部最小值，而非原始代价函数的全局最小值。

尽管延拓法最初用来解决局部最小值的问题，而局部最小值已不再认为是神经网络优化中的主要问题了。幸运的是，延拓法仍然有所帮助。延拓法引入的简化目标函数能够消除平坦区域，减少梯度估计的方差，提高 Hessian 矩阵的条件数，使局部更新更容易计算，或是改进局部更新方向与朝向全局解方向之间的对应关系。

Bengio *et al.* (2009) 指出被称为课程学习 (curriculum learning) 或者塑造 (shaping) 的方法可以被解释为延拓法。课程学习基于规划学习过程的想法，首先学习简单的概念，然后逐步学习依赖于这些简化概念的复杂概念。之前这一基本策略被用来加速动物训练过程 (Skinner, 1958; Peterson, 2004; Krueger and Dayan, 2009) 和机器学习过程 (Solomonoff, 1989; Elman, 1993; Sanger, 1994)。Bengio *et al.* (2009) 验证这一策略为延拓法，通过增加简单样本的影响（通过分配它们较大的系数到代价函数，或者更频繁地采样），先前的  $J^{(i)}$  会变得更容易。实验证明，在大规模的神经语言模型任务上使用课程学习，可以获得更好的结果。课程学习已经成功应用于大量的自然语言 (Spitkovsky *et al.*, 2010; Collobert *et al.*, 2011a; Mikolov *et al.*, 2011b; Tu and Honavar, 2011) 和计算机视觉 (Kumar *et al.*, 2010; Lee and

Grauman, 2011; Supancic and Ramanan, 2013) 任务上。课程学习被证实为与人类教学方式一致 (Khan *et al.*, 2011): 教师刚开始会展示更容易、更典型的示例，然后帮助学习者在不太显然的情况下提炼决策面。在人类教学上，基于课程学习的策略比基于样本均匀采样的策略更有效，也能提高其他学习策略的效率 (Basu and Christensen, 2013)。

课程学习研究的另一个重要贡献体现在训练循环神经网络捕获长期依赖: Zaremba and Sutskever (2014) 发现使用随机课程获得了更好的结果，其中容易和困难的示例混合在一起，随机提供给学习者，更难示例（这些具有长期依赖）的平均比例在逐渐上升。而使用确定性课程，并没有发现超过基线（完整训练集的普通训练）的改进。

现在我们已经介绍了一些基本的神经网络模型，以及如何进行正则化和优化。在接下来的章节中，我们转向特化的神经网络家族，允许其扩展到能够处理很大规模的数据和具有特殊结构的数据。在本章中讨论的优化算法在较少改动后或者无需改动，通常就可以直接用于这些特化的架构。

# 第九章 卷积网络

卷积网络 (convolutional network) (LeCun, 1989), 也叫做卷积神经网络 (convolutional neural network, CNN), 是一种专门用来处理具有类似网格结构的数据的神经网络。例如时间序列数据 (可以认为是在时间轴上有规律地采样形成的一维网格) 和图像数据 (可以看作是二维的像素网格)。卷积网络在诸多应用领域都表现优异。“卷积神经网络”一词表明该网络使用了卷积 (convolution) 这种数学运算。卷积是一种特殊的线性运算。卷积网络是指那些至少在网络的一层中使用卷积运算来替代一般的矩阵乘法运算的神经网络。

本章, 我们首先说明什么是卷积运算。接着, 我们会解释在神经网络中使用卷积运算的动机。然后我们会介绍池化 (pooling), 这是一种几乎所有的卷积网络都会用到的操作。通常来说, 卷积神经网络中用到的卷积运算和其他领域 (例如工程领域以及纯数学领域) 中的定义并不完全一致。我们会对神经网络实践中广泛应用的几种卷积函数的变体进行说明。我们也会说明如何在多种不同维数的数据上使用卷积运算。之后我们讨论使得卷积运算更加高效的一些方法。卷积网络是神经科学原理影响深度学习的典型代表。我们之后也会讨论这些神经科学的原理, 并对卷积网络在深度学习发展史中的作用作出评价。本章没有涉及如何为你的卷积网络选择合适的结构, 因为本章的目标是说明卷积网络提供的各种工具。第十一章将会对如何在具体环境中选择使用相应的工具给出通用的准则。对于卷积网络结构的研究进展得如此迅速, 以至于针对特定基准 (benchmark), 数月甚至几周就会公开一个新的最优的网络结构, 甚至在写这本书时也不好描述究竟哪种结构是最好的。然而, 最好的结构也是由本章所描述的基本部件逐步搭建起来的。

## 9.1 卷积运算

在通常形式中，卷积是对两个实变函数的一种数学运算<sup>1</sup>。为了给出卷积的定义，我们从两个可能会用到的函数的例子出发。

假设我们正在用激光传感器追踪一艘宇宙飞船的位置。我们的激光传感器给出一个单独的输出  $x(t)$ ，表示宇宙飞船在时刻  $t$  的位置。 $x$  和  $t$  都是实值的，这意味着我们可以在任意时刻从传感器中读出飞船的位置。

现在假设我们的传感器受到一定程度的噪声干扰。为了得到飞船位置的低噪声估计，我们对得到的测量结果进行平均。显然，时间上越近的测量结果越相关，所以我们采用一种加权平均的方法，对于最近的测量结果赋予更高的权重。我们可以采用一个加权函数  $w(a)$  来实现，其中  $a$  表示测量结果距当前时刻的时间间隔。如果我们对任意时刻都采用这种加权平均的操作，就得到了一个新的对于飞船位置的平滑估计函数  $s$ ：

$$s(t) = \int x(a)w(t-a)da. \quad (9.1)$$

这种运算就叫做 **卷积** ( convolution )。卷积运算通常用星号表示：

$$s(t) = (x * w)(t). \quad (9.2)$$

在我们的例子中， $w$  必须是一个有效的概率密度函数，否则输出就不再是一个加权平均。另外，在参数为负值时， $w$  的取值必须为 0，否则它会预测到未来，这不是我们能够推得的。但这些限制仅仅是对于我们这个例子来说。通常，卷积被定义在满足上述积分式的任意函数上，并且也可能被用于加权平均以外的目的。

在卷积网络的术语中，卷积的第一个参数（在这个例子中，函数  $x$ ）通常叫做 **输入** ( input )，第二个参数（函数  $w$ ）叫做 **核函数** ( kernel function )。输出有时被称作 **特征映射** ( feature map )。

在本例中，激光传感器在每个瞬间反馈测量结果的想法是不切实际的。一般地，当我们用计算机处理数据时，时间会被离散化，传感器会定期地反馈数据。所以在我们的例子中，假设传感器每秒反馈一次测量结果是比较现实的。这样，时刻  $t$  只能取整数值。如果我们假设  $x$  和  $w$  都定义在整数时刻  $t$  上，就可以定义离散形式的卷积：

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \quad (9.3)$$

---

<sup>1</sup>译者注：本书中 operation 视语境有时翻译成“运算”，有时翻译成“操作”。

在机器学习的应用中，输入通常是多维数组的数据，而核通常是由学习算法优化得到的多维数组的参数。我们把这些多维数组叫做张量。因为在输入与核中的每一个元素都必须明确地分开存储，我们通常假设在存储了数值的有限点集以外，这些函数的值都为零。这意味着在实际操作中，我们可以通过对有限个数组元素的求和来实现无限求和。

最后，我们经常一次在多个维度上进行卷积运算。例如，如果把一张二维的图像  $I$  作为输入，我们也许也想要使用一个二维的核  $K$ ：

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n). \quad (9.4)$$

卷积是可交换的 (commutative)，我们可以等价地写作：

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n). \quad (9.5)$$

通常，下面的公式在机器学习库中实现更为简单，因为  $m$  和  $n$  的有效取值范围相对较小。

卷积运算可交换性的出现是因为我们将核相对输入进行了翻转 (flip)，从  $m$  增大的角度来看，输入的索引在增大，但是核的索引在减小。我们将核翻转的唯一目的是实现可交换性。尽管可交换性在证明时很有用，但在神经网络的应用中却不是一个重要的性质。与之不同的是，许多神经网络库会实现一个相关的函数，称为互相关函数 (cross-correlation)，和卷积运算几乎一样但是并没有对核进行翻转：

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n). \quad (9.6)$$

许多机器学习的库实现的是互相关函数但是称之为卷积。在这本书中我们遵循把两种运算都叫做卷积的这个传统，在与核翻转有关的上下文中，我们会特别指明是否对核进行了翻转。在机器学习中，学习算法会在核合适的位置学得恰当的值，所以一个基于核翻转的卷积运算的学习算法所学得的核，是对未进行翻转的算法学得的核的翻转。单独使用卷积运算在机器学习中是很少见的，卷积经常与其他的函数一起使用，无论卷积运算是否对它的核进行了翻转，这些函数的组合通常是不可交换的。

图 9.1 演示了一个在 2 维张量上的卷积运算（没有对核进行翻转）的例子。

离散卷积可以看作矩阵的乘法，然而，这个矩阵的一些元素被限制为必须和另外一些元素相等。例如对于单变量的离散卷积，矩阵每一行中的元素都与上一行对应

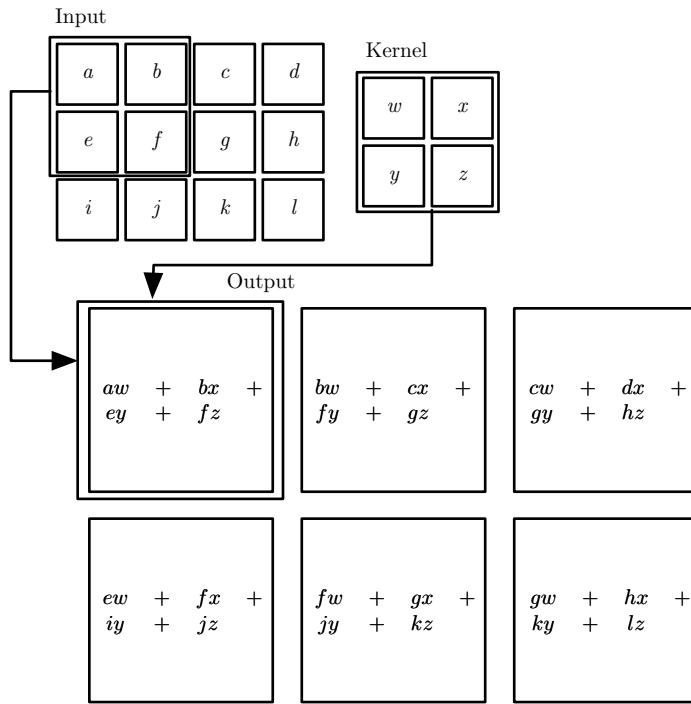


图 9.1: 一个 2 维卷积的例子 (没有对核进行翻转)。我们限制只对核完全处在图像中的位置进行输出，在一些上下文中称为“有效”卷积。我们用画有箭头的盒子来说明输出张量的左上角元素是如何通过对输入张量相应的左上角区域应用核进行卷积得到的。

位置平移一个单位的元素相同。这种矩阵叫做 **Toeplitz 矩阵** (Toeplitz matrix)。对于二维情况，卷积对应着一个 **双重分块循环矩阵** (doubly block circulant matrix)。除了这些元素相等的限制以外，卷积通常对应着一个非常稀疏的矩阵 (一个几乎所有元素都为零的矩阵)。这是因为核的大小通常要远小于输入图像的大小。任何一个使用矩阵乘法但是并不依赖矩阵结构的特殊性质的神经网络算法，都适用于卷积运算，并且不需要对神经网络做出大的修改。典型的卷积神经网络为了更有效地处理大规模输入，确实使用了一些专门化的技巧，但这些在理论分析方面并不是严格必要的。

## 9.2 动机

卷积运算通过三个重要的思想来帮助改进机器学习系统：稀疏交互（sparse interactions）、参数共享（parameter sharing）、等变表示（equivariant representations）。另外，卷积提供了一种处理大小可变的输入的方法。我们下面依次介绍这些思想。

传统的神经网络使用矩阵乘法来建立输入与输出的连接关系。其中，参数矩阵中每一个单独的参数都描述了一个输入单元与一个输出单元间的交互。这意味着每一个输出单元与每一个输入单元都产生交互。然而，卷积网络具有稀疏交互（sparse interactions）（也叫做稀疏连接（sparse connectivity）或者稀疏权重（sparse weights））的特征。这是使核的大小远小于输入的大小来达到的。举个例子，当处理一张图像时，输入的图像可能包含成千上万个像素点，但是我们可以通过只占用几十到上百个像素点的核来检测一些小的有意义的特征，例如图像的边缘。这意味着我们需要存储的参数更少，不仅减少了模型的存储需求，而且提高了它的统计效率。这也意味着为了得到输出我们只需要更少的计算量。这些效率上的提高往往是很显著的。如果有  $m$  个输入和  $n$  个输出，那么矩阵乘法需要  $m \times n$  个参数并且相应算法的时间复杂度为  $O(m \times n)$ （对于每一个例子）。如果我们限制每一个输出拥有的连接数为  $k$ ，那么稀疏的连接方法只需要  $k \times n$  个参数以及  $O(k \times n)$  的运行时间。在很多实际应用中，只需保持  $k$  比  $m$  小几个数量级，就能在机器学习的任务中取得好的表现。稀疏连接的图形化解释如图 9.2 和图 9.3 所示。在深度卷积网络中，处在网络深层的单元可能与绝大部分输入是间接交互的，如图 9.4 所示。这允许网络可以通过只描述稀疏交互的基石来高效地描述多个变量的复杂交互。

参数共享（parameter sharing）是指在一个模型的多个函数中使用相同的参数。在传统的神经网络中，当计算一层的输出时，权重矩阵的每一个元素只使用一次，当它乘以输入的一个元素后就再也不会用到了。作为参数共享的同义词，我们可以说一个网络含有绑定的权重（tied weights），因为用于一个输入的权重也会被绑定在其他的权重上。在卷积神经网络中，核的每一个元素都作用在输入的每一位置上（是否考虑边界像素取决于对边界决策的设计）。卷积运算中的参数共享保证了我们只需要学习一个参数集合，而不是对于每一位置都需要学习一个单独的参数集合。这虽然没有改变前向传播的运行时间（仍然是  $O(k \times n)$ ），但它显著地把模型的存储需求降低至  $k$  个参数，并且  $k$  通常要比  $m$  小很多个数量级。因为  $m$  和  $n$  通常有着大致相同的大小， $k$  在实际中相对于  $m \times n$  是很小的。因此，卷积在存储需求和统计效

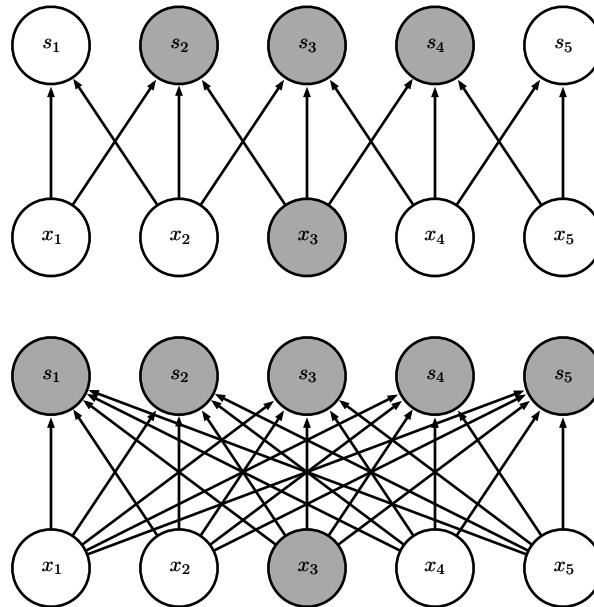


图 9.2: 稀疏连接, 对每幅图从下往上看。我们强调了一个输入单元  $x_3$  以及在  $s$  中受该单元影响的输出单元。(上) 当  $s$  是由核宽度为 3 的卷积产生时, 只有三个输出受到  $x$  的影响<sup>2</sup>。(下) 当  $s$  是由矩阵乘法产生时, 连接不再是稀疏的, 所以所有的输出都会受到  $x_3$  的影响。

率方面极大地优于稠密矩阵的乘法运算。图 9.5 演示了参数共享是如何实现的。

作为前两条原则的一个实际例子, 图 9.6 说明了稀疏连接和参数共享是如何显著提高线性函数在一张图像上进行边缘检测的效率的。

对于卷积, 参数共享的特殊形式使得神经网络层具有对平移 等变 (equivariance) 的性质。如果一个函数满足输入改变, 输出也以同样的方式改变这一性质, 我们就说它是等变 (equivariant) 的。特别地, 如果函数  $f(x)$  与  $g(x)$  满足  $f(g(x)) = g(f(x))$ , 我们就说  $f(x)$  对于变换  $g$  具有等变性。对于卷积来说, 如果令  $g$  是输入的任意平移函数, 那么卷积函数对于  $g$  具有等变性。举个例子, 令  $I$  表示图像在整数坐标上的亮度函数,  $g$  表示图像函数的变换函数 (把一个图像函数映射到另一个图像函数的函数) 使得  $I' = g(I)$ , 其中图像函数  $I'$  满足  $I'(x, y) = I(x - 1, y)$ 。这个函数把  $I$  中的每个像素向右移动一个单位。如果我们先对  $I$  进行这种变换然后进行卷积操作所得到的结果, 与先对  $I$  进行卷积然后再对输出使用平移函数  $g$  得到的结果是一样的<sup>4</sup>。当处理时间序列数据时, 这意味着通过卷积可以得到一个由输入中出现不同特

<sup>4</sup>译者注: 原文将此处误写成了  $I'$ 。

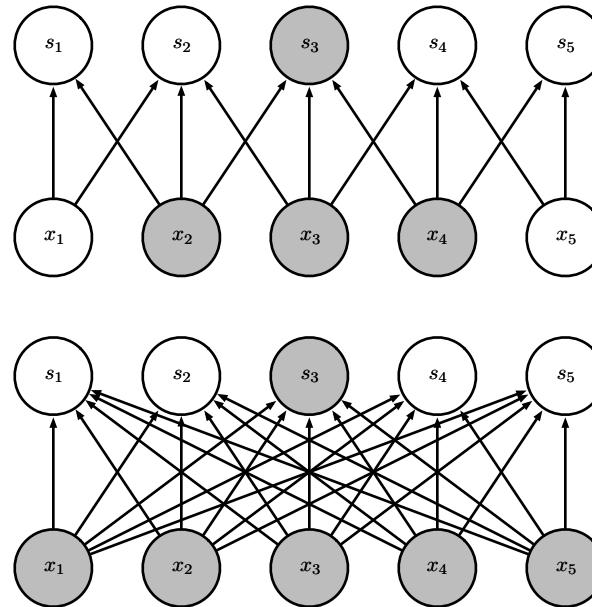


图 9.3: 稀疏连接, 对每幅图从上往下看。我们强调了一个输出单元  $s_3$  以及  $x$  中影响该单元的输入单元。这些单元被称为  $s_3$  的 接受域 (receptive field)<sup>3</sup>。(上) 当  $s$  是由核宽度为 3 的卷积产生时, 只有三个输入影响  $s_3$ 。(下) 当  $s$  是由矩阵乘法产生时, 连接不再是稀疏的, 所以所有的输入都会影响  $s_3$ 。

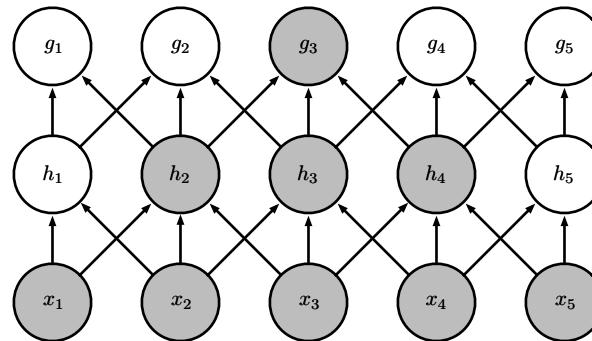


图 9.4: 处于卷积网络更深的层中的单元, 它们的接受域要比处在浅层的单元的接受域更大。如果网络还包含类似步幅卷积 (图 9.12) 或者池化 (第 9.3 节) 之类的结构特征, 这种效应会加强。这意味着在卷积网络中尽管直接连接都是很稀疏的, 但处在更深的层中的单元可以间接地连接到全部或者大部分输入图像。

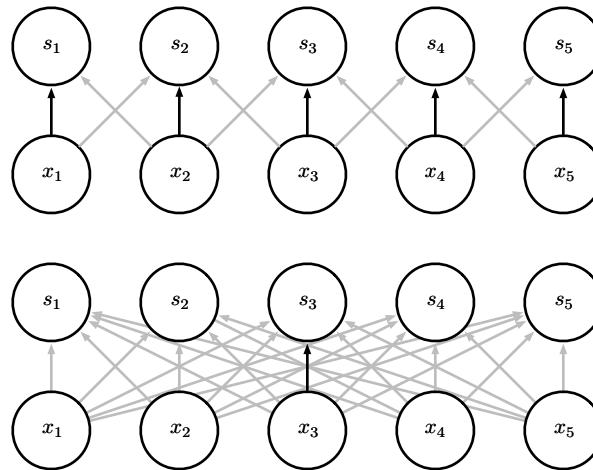


图 9.5: 参数共享。黑色箭头表示在两个不同的模型中使用了特殊参数的连接。(上) 黑色箭头表示在卷积模型中对 3 元素核的中间元素的使用。因为参数共享, 这个单独的参数被用于所有的输入位置。(下) 这个单独的黑色箭头表示在全连接模型中对权重矩阵的中间元素的使用。这个模型没有使用参数共享, 所以参数只使用了一次。

征的时刻所组成的时间轴。如果我们把输入中的一个事件向后延时, 在输出中仍然会有完全相同的表示, 只是时间延后了。图像与之类似, 卷积产生了一个 2 维映射来表明某些特征在输入中出现的位置。如果我们移动输入中的对象, 它的表示也会在输出中移动同样的量。当处理多个输入位置时, 一些作用在邻居像素的函数是很有用的。例如在处理图像时, 在卷积网络的第一层进行图像的边缘检测是很有用的。相同的边缘或多或少地散落在图像的各处, 所以应当对整个图像进行参数共享。但在某些情况下, 我们并不希望对整幅图进行参数共享。例如, 在处理已经通过剪裁而使其居中的人脸图像时, 我们可能想要提取不同位置上的不同特征(处理人脸上部的部分网络需要去搜寻眉毛, 处理人脸下部的部分网络就需要去搜寻下巴了)。

卷积对其他的一些变换并不是天然等变的, 例如对于图像的放缩或者旋转变换, 需要其他的一些机制来处理这些变换。

最后, 一些不能被传统的由(固定大小的)矩阵乘法定义的神经网络处理的特殊数据, 可能通过卷积神经网络来处理, 我们将在第 9.7 节中进行讨论。

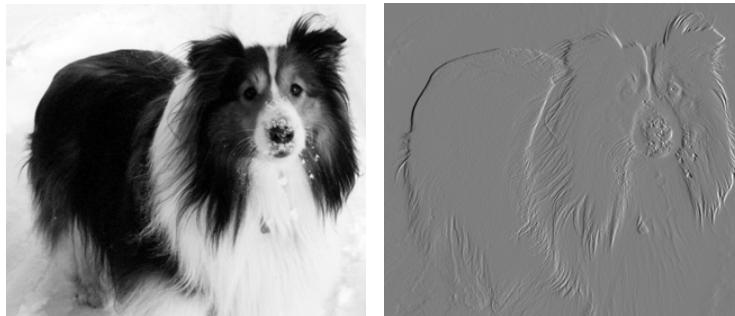


图 9.6: 边缘检测的效率。右边的图像是通过先获得原始图像中的每个像素，然后减去左边相邻像素的值而形成的。这个操作给出了输入图像中所有垂直方向上的边缘的强度，对目标检测来说是有用的。两个图像的高度均为 280 个像素。输入图像的宽度为 320 个像素，而输出图像的宽度为 319 个像素。这个变换可以通过包含两个元素的卷积核来描述，使用卷积需要  $319 \times 280 \times 3 = 267,960$  次浮点运算（每个输出像素需要两次乘法和一次加法）。为了用矩阵乘法描述相同的变换，需要一个包含  $320 \times 280 \times 319 \times 280$  个或者说超过 80 亿个元素的矩阵，这使得卷积对于表示这种变换更有效 40 亿倍。直接运行矩阵乘法的算法将执行超过 160 亿次浮点运算，这使得卷积在计算上大约有 60,000 倍的效率。当然，矩阵的大多数元素将为零。如果我们只存储矩阵的非零元，则矩阵乘法和卷积都需要相同数量的浮点运算来计算。矩阵仍然需要包含  $2 \times 319 \times 280 = 178,640$  个元素。将小的局部区域上的相同线性变换应用到整个输入上，卷积是描述这种变换的极其有效的方法。照片来源：Paula Goodfellow。

### 9.3 池化

卷积网络中一个典型层包含三级（如图 9.7 所示）。在第一级中，这一层并行地计算多个卷积产生一组线性激活响应。在第二级中，每一个线性激活响应将会通过一个非线性的激活函数，例如整流线性激活函数。这一级有时也被称为 **探测级** (detector stage)。在第三级中，我们使用 **池化函数** (pooling function) 来进一步调整这一层的输出。

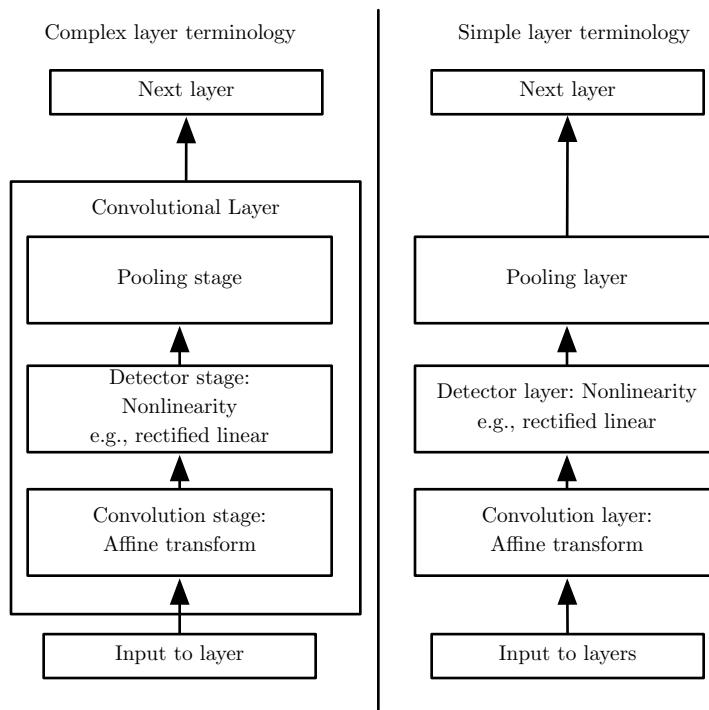


图 9.7: 一个典型卷积神经网络层的组件。有两组常用的术语用于描述这些层。(左) 在这组术语中，卷积网络被视为少量相对复杂的层，每层具有许多“级”。在这组术语中，核张量与网络层之间存在一一对应关系。在本书中，我们通常使用这组术语。(右) 在这组术语中，卷积网络被视为更多数量的简单层；每一个处理步骤都被认为是一个独立的层。这意味着不是每一“层”都有参数。

池化函数使用某一位置的相邻输出的总体统计特征来代替网络在该位置的输出。例如，**最大池化** (max pooling) 函数 (Zhou and Chellappa, 1988) 给出相邻矩形区域内的最大值。其他常用的池化函数包括相邻矩形区域内的平均值、 $L^2$  范数以及基于据中心像素距离的加权平均函数。

不管采用什么样的池化函数，当输入作出少量平移时，池化能够帮助输入的表示近似不变（invariant）。对于平移的不变性是指当我们对输入进行少量平移时，经过池化函数后的大多数输出并不会发生改变。图 9.8 用了一个例子来说明这是如何实现的。局部平移不变性是一个很有用的性质，尤其是当我们关心某个特征是否出现而不关心它出现的具体位置时。例如，当判定一张图像中是否包含人脸时，我们并不需要知道眼睛的精确像素位置，我们只需要知道有一只眼睛在脸的左边，有一只在右边就行了。但在一些其他领域，保存特征的具体位置却很重要。例如当我们想要寻找一个由两条边相交而成的拐角时，我们就需要很好地保存边的位置来判定它们是否相交。

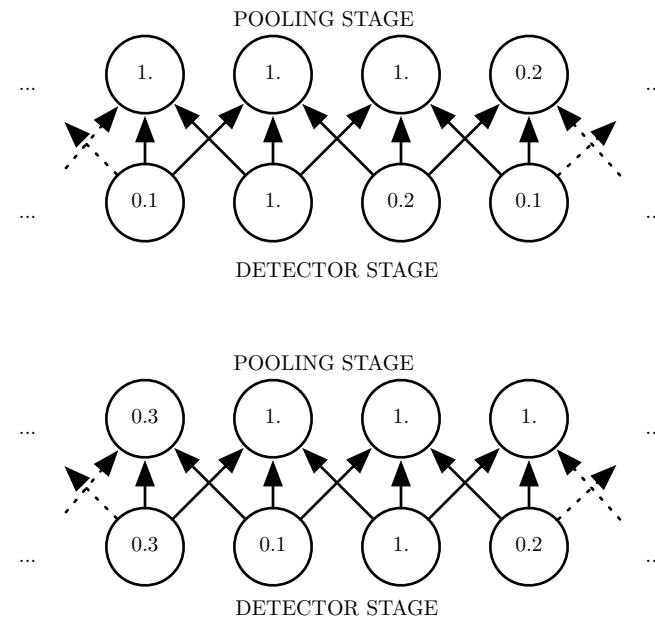


图 9.8：最大池化引入了不变性。(上) 卷积层中间输出的视图。下面一行显示非线性的输出。上面一行显示最大池化的输出，每个池的宽度为三个像素并且池化区域的步幅为一个像素。(下) 相同网络的视图，不过对输入右移了一个像素。下面一行的所有值都发生了改变，但上面一行只有一半的值发生了改变，这是因为最大池化单元只对周围的最大值比较敏感，而不是对精确的位置。

使用池化可以看作是增加了一个无限强的先验：这一层学得的函数必须具有对少量平移的不变性。当这个假设成立时，池化可以极大地提高网络的统计效率。

对空间区域进行池化产生了平移不变性，但当我们对分离参数的卷积的输出进行池化时，特征能够学得应该对于哪种变换具有不变性（如图 9.9 所示）。

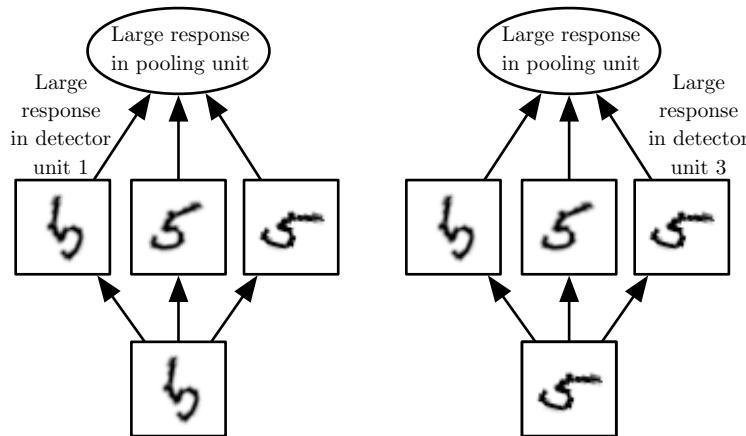


图 9.9: 学习不变性的示例。使用分离的参数学得多个特征，再使用池化单元进行池化，可以学得对输入的某些变换的不变性。这里我们展示了用三个学得的过滤器和一个最大池化单元可以学得对旋转变换的不变性。这三个过滤器都旨在检测手写的数字 5。每个过滤器尝试匹配稍微不同方向的 5。当输入中出现 5 时，相应的过滤器会匹配它并且在探测单元中引起大的激活。然后，无论哪个探测单元被激活，最大池化单元都具有大的激活。我们在这里演示了网络如何处理两个不同的输入，这导致两个不同的探测单元被激活，然而对池化单元的影响大致相同。这个原则在 maxout 网络 (Goodfellow *et al.*, 2013b) 和其他卷积网络中更有影响。空间位置上的最大池化对于平移是天然不变的；这种多通道方法只在学习其他变换时是必要的。

因为池化综合了全部邻居的反馈，这使得池化单元少于探测单元成为可能，我们可以通过综合池化区域的  $k$  个像素的统计特征而不是单个像素来实现。图 9.10 给出了一个例子。这种方法提高了网络的计算效率，因为下一层少了约  $k$  倍的输入。当下一层的参数数目是关于那一层输入大小的函数时（例如当下一层是全连接的基于矩阵乘法的网络层时），这种对于输入规模的减小也可以提高统计效率并且减少对于参数的存储需求。

在很多任务中，池化对于处理不同大小的输入具有重要作用。例如我们想对不同大小的图像进行分类时，分类层的输入必须是固定的大小，而这通常通过调整池化区域的偏置大小来实现，这样分类层总是能接收到相同数量的统计特征而不管最初的输入大小了。例如，最终的池化层可能会输出四组综合统计特征，每组对应着图像的一个象限，而与图像的大小无关。

一些理论工作对于在不同情况下应当使用哪种池化函数给出了一些指导 (Boureau *et al.*, 2010)。将特征一起动态地池化也是可行的，例如，对于感兴趣

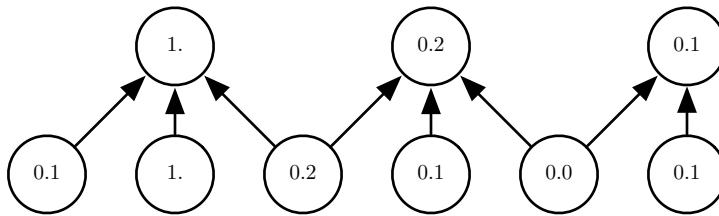


图 9.10: 带有降采样的池化。这里我们使用最大池化，池的宽度为三并且池之间的步幅为二。这使得表示的大小减少了一半，减轻了下一层的计算和统计负担。注意到最右边的池化区域尺寸较小，但如果我们不想忽略一些探测单元的话就必须包含这个区域。

特征的位置运行聚类算法 (Boureau *et al.*, 2011)。这种方法对于每幅图像产生一个不同的池化区域集合。另一种方法是先学习一个单独的池化结构，再应用到全部的图像中 (Jia *et al.*, 2012)。

池化可能会使得一些利用自顶向下信息的神经网络结构变得复杂，例如玻尔兹曼机和自编码器。这些问题将在第三章中当我们遇到这些类型的网络时进一步讨论。卷积玻尔兹曼机中的池化出现在第 20.6 节。一些可微网络中需要的在池化单元上进行的类逆运算将在第 20.10.6 节中讨论。

图 9.11 给出了一些使用卷积和池化操作的用于分类的完整卷积网络结构的例子。

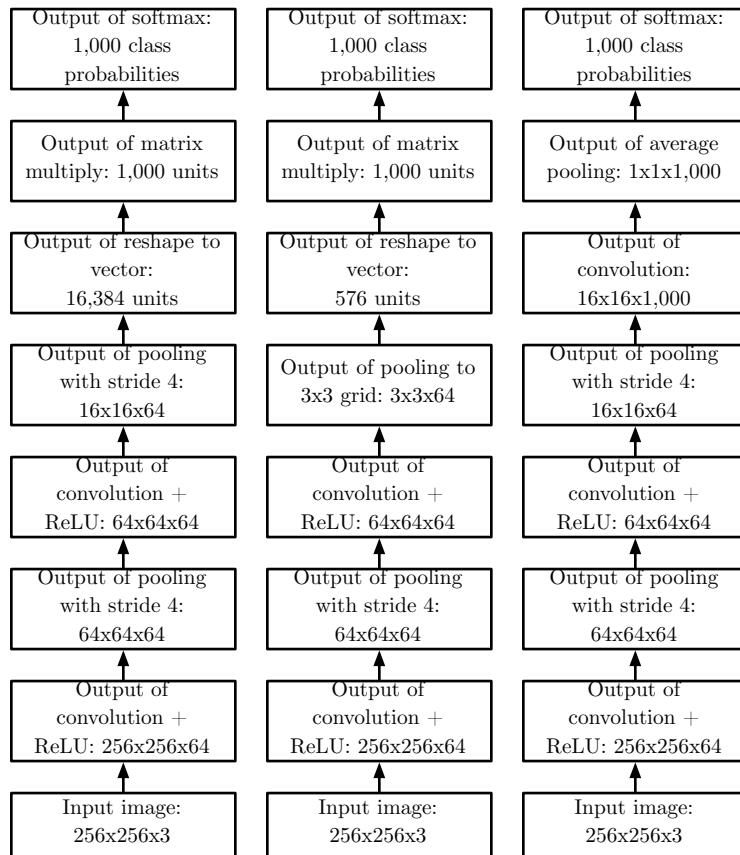


图 9.11: 卷积网络用于分类的结构示例。本图中使用的具体步幅和深度并不建议实际使用；它们被设计得非常浅以适合页面。实际的卷积网络还常常涉及大量的分支，不同于这里为简单起见所使用的链式结构。(左) 处理固定大小的图像的卷积网络。在卷积层和池化层几层交替之后，卷积特征映射的张量被重新变形以展平空间维度。网络的其余部分是一个普通的前馈网络分类器，如第六章所述。(中) 处理大小可变的图像的卷积网络，但仍保持全连接的部分。该网络使用具有可变大小但是数量固定的池的池化操作，以便向网络的全连接部分提供固定 576 个单位大小的向量。(右) 没有任何全连接权重层的卷积网络。相对的，最后的卷积层为每个类输出一个特征映射。该模型可能会用来学习每个类出现在每个空间位置的可能性的映射。将特征映射进行平均得到的单个值，提供了顶部 softmax 分类器的变量。

## 9.4 卷积与池化作为一种无限强的先验

回忆一下第 5.2 节中 **先验概率分布** (prior probability distribution) 的概念。这是一个模型参数的概率分布，它刻画了在我们看到数据之前我们认为什么样的模型是合理的信念。

先验被认为是强或者弱取决于先验中概率密度的集中程度。弱先验具有较高的熵值，例如方差很大的高斯分布。这样的先验允许数据对于参数的改变具有或多或少的自由性。强先验具有较低的熵值，例如方差很小的高斯分布。这样的先验在决定参数最终取值时起着更加积极的作用。

一个无限强的先验需要对一些参数的概率置零并且完全禁止对这些参数赋值，无论数据对于这些参数的值给出了多大的支持。

我们可以把卷积网络类比成全连接网络，但对于这个全连接网络的权重有一个无限强的先验。这个无限强的先验是说一个隐藏单元的权重必须和它邻居的权重相同，但可以在空间上移动。这个先验也要求除了那些处在隐藏单元的小的空间连续的接受域内的权重以外，其余的权重都为零。总之，我们可以把卷积的使用当作是对网络中一层的参数引入了一个无限强的先验概率分布。这个先验说明了该层应该学得的函数只包含局部连接关系并且对平移具有等变性。类似的，使用池化也是一个无限强的先验：每一个单元都具有对少量平移的不变性。

当然，把卷积神经网络当作一个具有无限强先验的全连接网络来实现会导致极大的计算浪费。但把卷积神经网络想成具有无限强先验的全连接网络可以帮助我们更好地洞察卷积神经网络是如何工作的。

其中一个关键的洞察是卷积和池化可能导致欠拟合。与任何其他先验类似，卷积和池化只有当先验的假设合理且正确时才有用。如果一项任务依赖于保存精确的空间信息，那么在所有的特征上使用池化将会增大训练误差。一些卷积网络结构 (Szegedy *et al.*, 2014a) 为了既获得具有较高不变性的特征又获得当平移不变性不合理时不会导致欠拟合的特征，被设计成在一些通道上使用池化而在另一些通道上不使用。当一项任务涉及到要对输入中相隔较远的信息进行合并时，那么卷积所利用的先验可能就不正确了。

另一个关键洞察是当我们比较卷积模型的统计学习表现时，只能以基准中的其他卷积模型作为比较的对象。其他不使用卷积的模型即使我们把图像中的所有像素点都置换后依然有可能进行学习。对于许多图像数据集，还有一些分别的基准，有

些是针对那些具有 置换不变性 (permutation invariant) 并且必须通过学习发现拓扑结构的模型，还有一些是针对模型设计者将空间关系的知识植入了它们的模型。

## 9.5 基本卷积函数的变体

当在神经网络的上下文中讨论卷积时，我们通常不是特指数学文献中使用的那种标准的离散卷积运算。实际应用中的函数略有不同。这里我们详细讨论一下这些差异，并且对神经网络中用到的函数的一些重要性质进行重点说明。

首先，当我们提到神经网络中的卷积时，我们通常是指由多个并行卷积组成的运算。这是因为具有单个核的卷积只能提取一种类型的特征，尽管它作用在多个空间位置上。我们通常希望网络的每一层能够在多个位置提取多种类型的特征。

另外，输入通常也不仅仅是实值的网格，而是由一系列观测数据的向量构成的网格。例如，一幅彩色图像在每一个像素点都会有红绿蓝三种颜色的亮度。在多层的卷积网络中，第二层的输入是第一层的输出，通常在每个位置包含多个不同卷积的输出。当处理图像时，我们通常把卷积的输入输出都看作是 3 维的张量，其中一个索引用于标明不同的通道（例如红绿蓝），另外两个索引标明在每个通道上的空间坐标。软件实现通常使用批处理模式，所以实际上会使用 4 维的张量，第四维索引用于标明批处理中不同的实例，但我们为简明起见这里忽略批处理索引。

因为卷积网络通常使用多通道的卷积，所以即使使用了核翻转，也不一定保证网络的线性运算是可交换的。只有当其中的每个运算的输出和输入具有相同的通道数时，这些多通道的运算才是可交换的。

假定我们有一个 4 维的核张量  $\mathbf{K}$ ，它的每一个元素是  $K_{i,j,k,l}$ ，表示输出中处于通道  $i$  的一个单元和输入中处于通道  $j$  中的一个单元的连接强度，并且在输出单元和输入单元之间有  $k$  行  $l$  列的偏置。假定我们的输入由观测数据  $\mathbf{V}$  组成，它的每一个元素是  $V_{i,j,k}$ ，表示处在通道  $i$  中第  $j$  行第  $k$  列的值。假定我们的输出  $\mathbf{Z}$  和输入  $\mathbf{V}$  具有相同的形式。如果输出  $\mathbf{Z}$  是通过对  $\mathbf{K}$  和  $\mathbf{V}$  进行卷积而不涉及翻转  $\mathbf{K}$  得到的，那么

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}, \quad (9.7)$$

这里对所有的  $l$ ,  $m$  和  $n$  进行求和是对所有（在求和式中）有效的张量索引的值进行求和。在线性代数中，向量的索引通常从 1 开始，这就是上述公式中  $-1$  的由来。

但是像 C 或 Python 这类编程语言索引通常从 0 开始，这使得上述公式可以更加简洁。

我们有时会希望跳过核中的一些位置来降低计算的开销（相应的代价是提取特征没有先前那么好了）。我们可以把这一过程看作是对全卷积函数输出的下采样 (downsampling)。如果我们只想在输出的每个方向上每间隔  $s$  个像素进行采样，那么我们可以定义一个下采样卷积函数  $c$  使得

$$Z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} [V_{l,(j-1) \times s + m, (k-1) \times s + n}, K_{i,l,m,n}]. \quad (9.8)$$

我们把  $s$  称为下采样卷积的 **步幅** (stride)。当然也可以对每个移动方向定义不同的步幅。图 9.12 演示了一个实例。

在任何卷积网络的实现中都有一个重要性质，那就是能够隐含地对输入  $\mathbf{V}$  用零进行填充 (pad) 使得它加宽。如果没有这个性质，表示的宽度在每一层就会缩减，缩减的幅度是比核少一个像素这么多。对输入进行零填充允许我们对核的宽度和输出的大小进行独立的控制。如果没有零填充，我们就被迫面临二选一的局面，要么选择网络空间宽度的快速缩减，要么选择一个小型的核——这两种情境都会极大得限制网络的表示能力。图 9.13 给出了一个例子。

有三种零填充设定的情况值得注意。第一种是无论怎样都不使用零填充的极端情况，并且卷积核只允许访问那些图像中能够完全包含整个核的位置。在 MATLAB 的术语中，这称为 **有效** (valid) 卷积。在这种情况下，输出的所有像素都是输入中相同数量像素的函数，这使得输出像素的表示更加规范。然而，输出的大小在每一层都会缩减。如果输入的图像宽度是  $m$ ，核的宽度是  $k$ ，那么输出的宽度就会变成  $m - k + 1$ 。如果卷积核非常大的话缩减率会非常显著。因为缩减数大于 0，这限制了网络中能够包含的卷积层的层数。当层数增加时，网络的空间维度最终会缩减到  $1 \times 1$ ，这种情况下增加的层就不可能进行有意义的卷积了。第二种特殊的情况是只进行足够的零填充来保持输出和输入具有相同的大小。在 MATLAB 的术语中，这称为 **相同** (same) 卷积。在这种情况下，只要硬件支持，网络就能包含任意多的卷积层，这是因为卷积运算不改变下一层的结构。。然而，输入像素中靠近边界的部分相比于中间部分对于输出像素的影响更小。这可能会导致边界像素存在一定程度的欠表示。这使得第三种极端情况产生了，在 MATLAB 中称为 **全** (full) 卷积。它进行了足够多的零填充使得每个像素在每个方向上恰好被访问了  $k$  次，最终输出图像的宽度为  $m + k - 1$ 。在这种情况下，输出像素中靠近边界的部分相比于中间部分是更少像素的函数。这将导致学得一个在卷积特征映射的所有位置都表现不错的单核

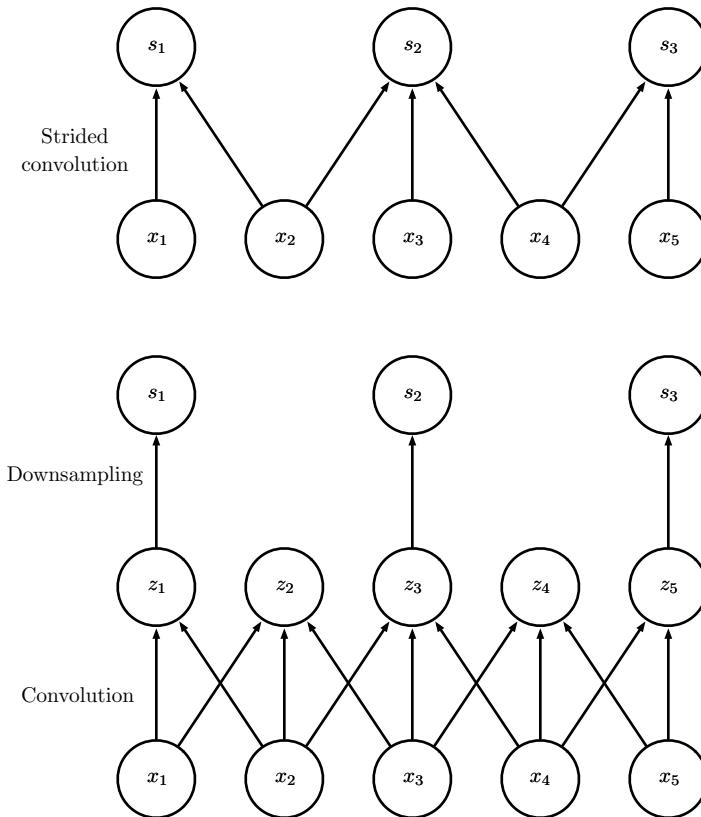


图 9.12: 带有步幅的卷积。在这个例子中，我们的步幅为二。(上)在单个操作中实现的步幅为二的卷积。(下)步幅大于一个像素的卷积在数学上等价于单位步幅的卷积随后降采样。显然，涉及降采样的两步法在计算上是浪费的，因为它计算了许多将被丢弃的值。

更为困难。通常零填充的最优数量（对于测试集的分类正确率）处于“有效卷积”和“相同卷积”之间的某个位置。

在一些情况下，我们并不是真的想使用卷积，而是想用一些局部连接的网络层 (LeCun, 1986, 1989)。在这种情况下，我们的多层感知机对应的邻接矩阵是相同的，但每一个连接都有它自己的权重，用一个 6 维的张量  $\mathbf{W}$  来表示。 $\mathbf{W}$  的索引分别是：输出的通道  $i$ ，输出的行  $j$  和列  $k$ ，输入的通道  $l$ ，输入的行偏置  $m$  和列偏置  $n$ 。局部连接层的线性部分可以表示为

$$Z_{i,j,k} = \sum_{l,m,n} [V_{l,j+m-1,k+n-1} w_{i,j,k,l,m,n}]. \quad (9.9)$$

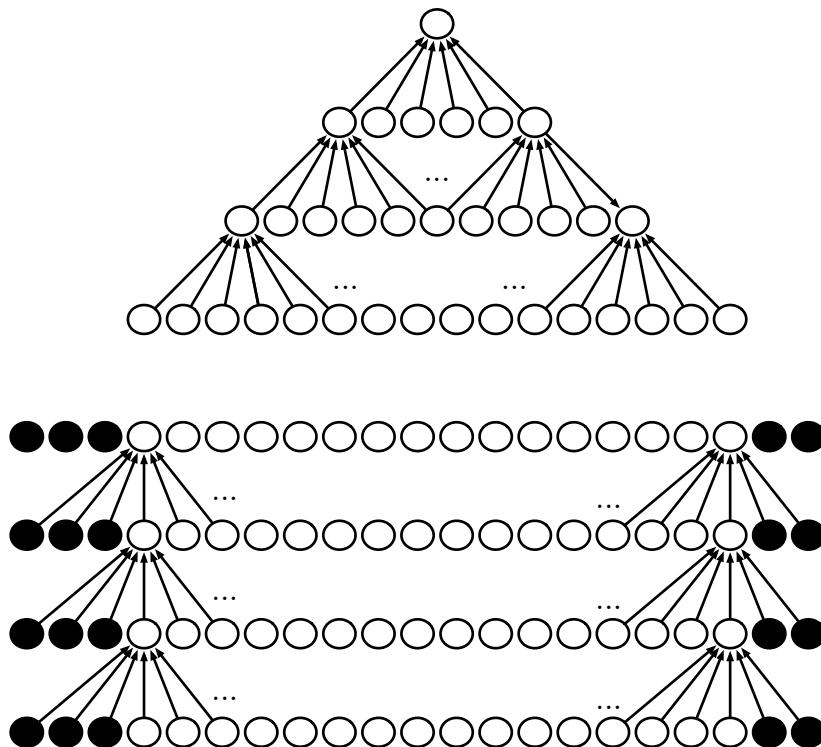


图 9.13: 零填充对网络大小的影响。考虑一个卷积网络，每层有一个宽度为六的核。在这个例子中，我们不使用任何池化，所以只有卷积操作本身缩小网络的大小。(上) 在这个卷积网络中，我们不使用任何隐含的零填充。这使得表示在每层缩小五个像素。从十六个像素的输入开始，我们只能有三个卷积层，并且最后一层不能移动核，所以可以说只有两层是真正的卷积层。可以通过使用较小的核来减缓收缩速率，但是较小的核表示能力不足，并且在这种结构中一些收缩是不可避免的。(下) 通过向每层添加五个隐含的零，我们防止了表示随深度收缩。这允许我们设计一个任意深的卷积网络。

这有时也被称为 **非共享卷积** (unshared convolution)，因为它和具有一个小核的离散卷积运算很像，但并不横跨位置来共享参数。图9.14比较了局部连接、卷积和全连接的区别。

当我们知道每一个特征都是一小块空间的函数并且相同的特征不会出现在所有的空间上时，局部连接层是很有用的。例如，如果我们想要辨别一张图片是否是人脸图像时，我们只需要去寻找嘴是否在图像下半部分即可。

使用那些连接被更进一步限制的卷积或者局部连接层也是有用的，例如，限制

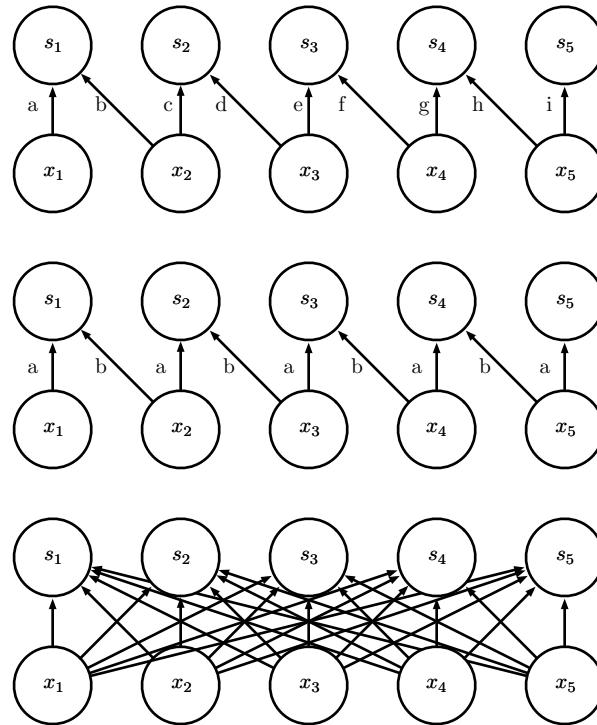


图 9.14: 局部连接, 卷积和全连接的比较。(上)每一小片(接受域)有两个像素的局部连接层。每条边用唯一的字母标记, 来显示每条边都有自身的权重参数。(中)核宽度为两个像素的卷积层。该模型与局部连接层具有完全相同的连接。区别不在于哪些单元相互交互, 而在于如何共享参数。局部连接层没有参数共享。正如用于标记每条边的字母重复出现所指示的, 卷积层在整个输入上重复使用相同的两个权重。(下)全连接层类似于局部连接层, 它的每条边都有其自身的参数(在该图中用字母明确标记的话就太多了)。然而, 它不具有局部连接层的连接受限的特征。

每一个输出的通道  $i$  仅仅是输入通道  $l$  的一部分的函数时。实现这种情况的一种通用方法是使输出的前  $m$  个通道仅仅连接到输入的前  $n$  个通道, 输出的接下来的  $m$  个通道仅仅连接到输入的接下来的  $n$  个通道, 以此类推。图 9.15 给出了一个例子。对少量通道间的连接进行建模允许网络使用更少的参数, 这降低了存储的消耗以及提高了统计效率, 并且减少了前向和反向传播所需要的计算量。这些目标的实现并没有减少隐藏单元的数目。

**平铺卷积** ( tiled convolution ) (Gregor and LeCun, 2010a; Le *et al.*, 2010) 对卷积层和局部连接层进行了折衷。这里并不是对每一个空间位置的权重集合进行学习, 我们学习一组核使得当我们在空间移动时它们可以循环利用。这意味着在近邻的位

置上拥有不同的过滤器，就像局部连接层一样，但是对于这些参数的存储需求仅仅会增长常数倍，这个常数就是核的集合的大小，而不是整个输出的特征映射的大小。图 9.16 对局部连接层、平铺卷积和标准卷积进行了比较。

为了用代数的方法定义平铺卷积，令  $\mathbf{K}$  是一个 6 维的张量<sup>5</sup>，其中的两维对应着输出映射中的不同位置。 $\mathbf{K}$  在这里并没有对输出映射中的每一个位置使用单独的索引，输出的位置在每个方向上在  $t$  个不同的核组成的集合中进行循环。如果  $t$  等于输出的宽度，这就是局部连接层了。

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n,j \% t+1, k \% t+1}, \quad (9.10)$$

这里百分号是取模运算，它的性质包括  $t \% t = 0, (t + 1) \% t = 1$  等等。在每一维上使用不同的  $t$  可以很容易对这个方程进行扩展。

局部连接层与平铺卷积层都和最大池化有一些有趣的关联：这些层的探测单元都是由不同的过滤器驱动的。如果这些过滤器能够学会探测相同隐含特征的不同变换形式，那么最大池化的单元对于学得的变换就具有不变性（如图 9.9 所示）。卷积层对于平移具有内置的不变性。

实现卷积网络时，通常也需要除卷积以外的其他运算。为了实现学习，必须在给定输出的梯度时能够计算核的梯度。在一些简单情况下，这种运算可以通过卷积来实现，但在很多我们感兴趣的情况下，包括步幅大于 1 的情况，并不具有这样的性质。

回忆一下卷积是一种线性运算，所以可以表示成矩阵乘法的形式（如果我们首先把输入张量变形为一个扁平的向量）。其中包含的矩阵是关于卷积核的函数。这个矩阵是稀疏的并且核的每个元素都复制给矩阵的多个元素。这种观点能够帮助我们导出实现一个卷积网络所需的很多其他运算。

通过卷积定义的矩阵转置的乘法就是这样一种运算。这种运算用于在卷积层反向传播误差的导数，所以它在训练多于一个隐藏层的卷积网络时是必要的。如果我们想要从隐藏层单元重构可视化单元时，同样的运算也是需要的 (Simard *et al.*, 1992)。重构可视化单元是本书第三部分的模型广泛用到的一种运算，这些模型包括自编码器、RBM 和稀疏编码等等。构建这些模型的卷积化的版本都要用到转置化卷积。类似核梯度运算，这种输入梯度运算在某些情况下可以用卷积来实现，但在一般情况下需要用到第三种运算来实现。必须非常小心地来使这种转置运算和前向传播过程

---

<sup>5</sup>译者注：原文将  $\mathbf{K}$  误写成了  $k$ 。

相协调。转置运算返回的输出的大小取决于三个方面：零填充的策略、前向传播运算的步幅以及前向传播的输出映射的大小。在一些情况下，不同大小的输入通过前向传播过程能够得到相同大小的输出映射，所以必须明确地告知转置运算原始输入的大小。

这三种运算——卷积、从输出到权重的反向传播和从输出到输入的反向传播——对于训练任意深度的前馈卷积网络，以及训练带有（基于卷积的转置的）重构函数的卷积网络，这三种运算都足以计算它们所需的所有梯度。对于完全一般的多维、多样例情况下的公式，完整的推导可以参考 Goodfellow (2010)。为了直观说明这些公式是如何起作用的，我们这里给出一个二维单个样例的版本。

假设我们想要训练这样一个卷积网络，它包含步幅为  $s$  的步幅卷积，该卷积的核为  $\mathbf{K}$ ，作用于多通道的图像  $\mathbf{V}$ ，定义为  $c(\mathbf{K}, \mathbf{V}, s)$ ，就像式 (9.8) 中一样。假设我们想要最小化某个损失函数  $J(\mathbf{V}, \mathbf{K})$ 。在前向传播过程中，我们需要用  $c$  本身来输出  $\mathbf{Z}$ ，然后  $\mathbf{Z}$  传递到网络的其余部分并且被用来计算损失函数  $J$ 。在反向传播过程中，我们会得到一个张量  $\mathbf{G}$  满足  $G_{i,j,k} = \frac{\partial}{\partial Z_{i,j,k}} J(\mathbf{V}, \mathbf{K})$ 。

为了训练网络，我们需要对核中的权重求导。为了实现这个目的，我们可以使用一个函数

$$g(\mathbf{G}, \mathbf{V}, s)_{i,j,k,l} = \frac{\partial}{\partial K_{i,j,k,l}} J(\mathbf{V}, \mathbf{K}) = \sum_{m,n} G_{i,m,n} V_{j,(m-1) \times s+k, (n-1) \times s+l}. \quad (9.11)$$

如果这一层不是网络的底层，我们需要对  $\mathbf{V}$  求梯度来使得误差进一步反向传播。我们可以使用如下的函数

$$h(\mathbf{K}, \mathbf{G}, s)_{i,j,k} = \frac{\partial}{\partial V_{i,j,k}} J(\mathbf{V}, \mathbf{K}) \quad (9.12)$$

$$= \sum_{\substack{l,m \\ \text{s.t.} \\ (l-1) \times s+m=j}} \sum_{\substack{n,p \\ \text{s.t.} \\ (n-1) \times s+p=k}} \sum_q K_{q,i,m,p} G_{q,l,n}. \quad (9.13)$$

第十四章描述的自编码器网络，是一些被训练成把输入拷贝到输出的前馈网络。一个简单的例子是PCA算法，将输入  $\mathbf{x}$  拷贝到一个近似的重构值  $\mathbf{r}$ ，通过函数  $\mathbf{W}^\top \mathbf{Wx}$  来实现。使用权重矩阵转置的乘法，就像PCA算法这种，在一般的自编码器中是很常见的。为了使这些模型卷积化，我们可以用函数  $h$  来实现卷积运算的转置。假定我们有和  $\mathbf{Z}$  相同形式的隐藏单元  $\mathbf{H}$ ，并且我们定义一种重构运算

$$\mathbf{R} = h(\mathbf{K}, \mathbf{H}, s). \quad (9.14)$$

为了训练自编码器，我们会得到关于  $\mathbf{R}$  的梯度，表示为一个张量  $\mathbf{E}$ 。为了训练解码器，我们需要获得对于  $\mathbf{K}$  的梯度，这通过  $g(\mathbf{H}, \mathbf{E}, s)$  来得到。为了训练编码器，我们需要获得对于  $\mathbf{H}$  的梯度，这通过  $c(\mathbf{K}, \mathbf{E}, s)$  来得到。通过用  $c$  和  $h$  对  $g$  求微分也是可行的，但这些运算对于任何标准神经网络上的反向传播算法来说都是不需要的。

一般来说，在卷积层从输入到输出的变换中我们不仅仅只用线性运算。我们一般也会在进行非线性运算前，对每个输出加入一些偏置项。这样就产生了如何在偏置项中共享参数的问题。对于局部连接层，很自然地对每个单元都给定它特有的偏置，对于平铺卷积，也很自然地用与核一样的平铺模式来共享参数。对于卷积层来说，通常的做法是在输出的每一个通道上都设置一个偏置，这个偏置在每个卷积映射的所有位置上共享。然而，如果输入是已知的固定大小，也可以在输出映射的每个位置学习一个单独的偏置。分离这些偏置可能会稍稍降低模型的统计效率，但同时也允许模型来校正图像中不同位置的统计差异。例如，当使用隐含的零填充时，图像边缘的探测单元接收到较少的输入，因此需要较大的偏置。

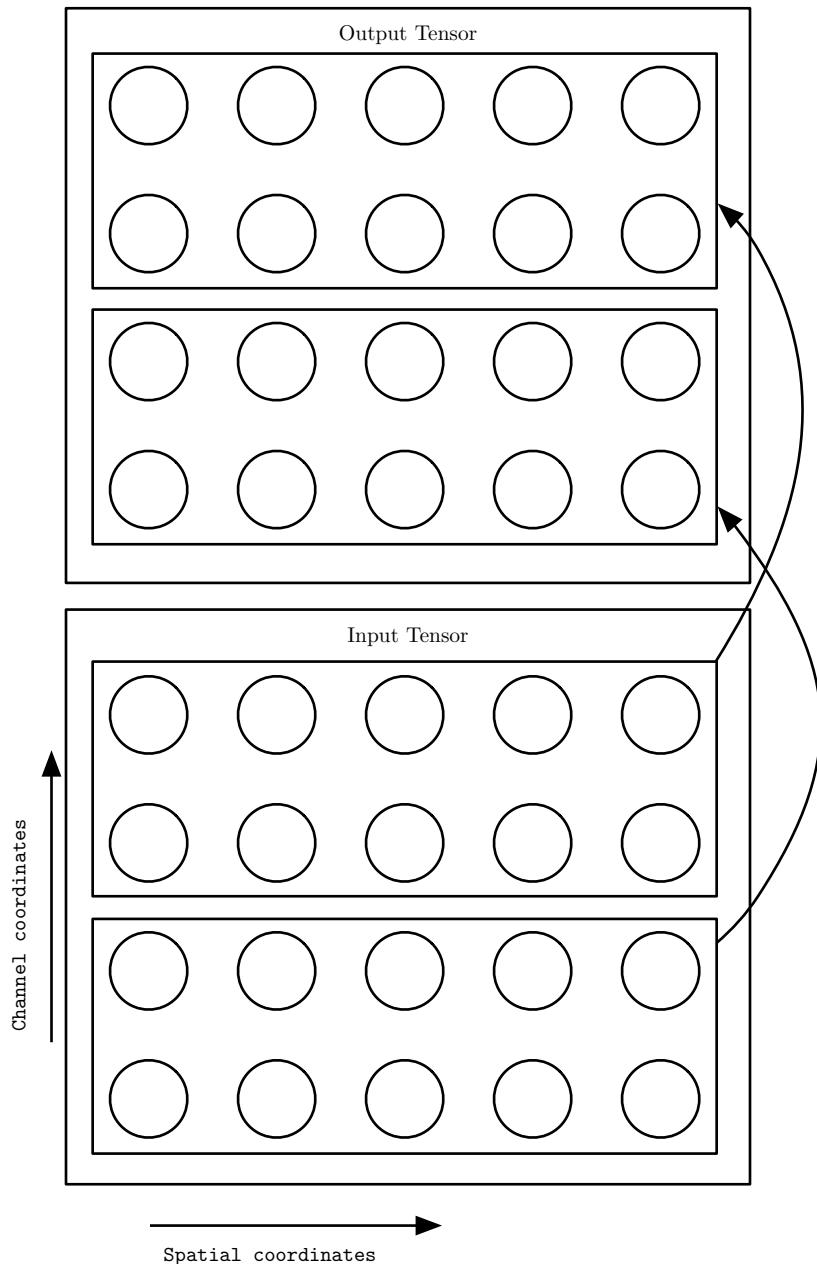


图 9.15: 卷积网络的前两个输出通道只和前两个输入通道相连，随后的两个输出通道只和随后的两个输入通道相连。

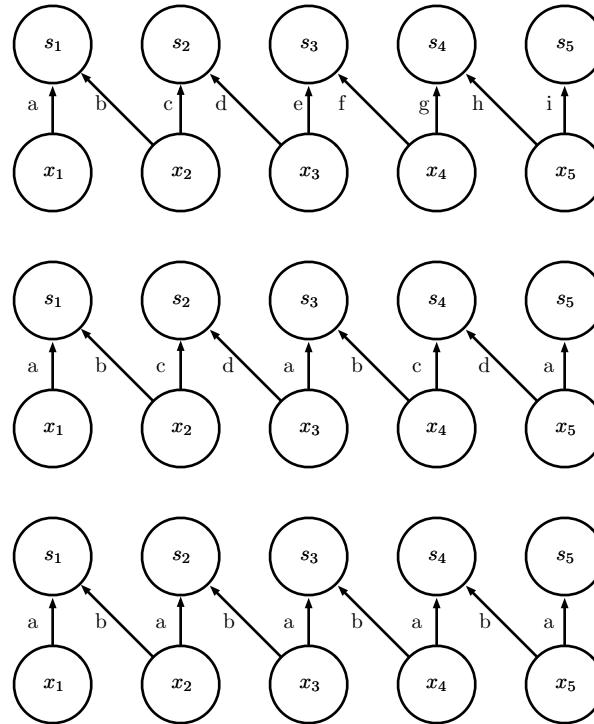


图 9.16: 局部连接层、平铺卷积和标准卷积的比较。当使用相同大小的核时，这三种方法在单元之间具有相同的连接。此图是对使用两个像素宽的核的说明。这三种方法之间的区别在于它们如何共享参数。(上) 局部连接层根本没有共享参数。我们对每个连接使用唯一的字母标记，来表明每个连接都有它自身的权重。(中) 平铺卷积有  $t$  个不同的核。这里我们说明  $t = 2$  的情况。其中一个核具有标记为“a”和“b”的边，而另一个具有标记为“c”和“d”的边。每当我们右移一个像素后，我们使用一个不同的核。这意味着，与局部连接层类似，输出中的相邻单元具有不同的参数。与局部连接层不同的是，在我们遍历所有可用的  $t$  个核之后，我们循环回到了第一个核。如果两个输出单元间隔  $t$  个步长的倍数，则它们共享参数。(下) 传统卷积等效于  $t = 1$  的平铺卷积。它只有一个核，并且被应用到各个地方，我们在图中表示为在各处使用具有标记为“a”和“b”的边的核。

## 9.6 结构化输出

卷积神经网络可以用于输出高维的结构化对象，而不仅仅是预测分类任务的类标签或回归任务的实数值。通常这个对象只是一个张量，由标准卷积层产生。例如，模型可以产生张量  $\mathbf{S}$ ，其中  $S_{i,j,k}$  是网络的输入像素  $(j, k)$  属于类  $i$  的概率。这允许模型标记图像中的每个像素，并绘制沿着单个对象轮廓的精确掩模。

经常出现的一个问题是输出平面可能比输入平面要小，如图 9.13 所示。用于对图像中单个对象分类的常用结构中，网络空间维数的最大减少来源于使用大步幅的池化层。为了产生与输入大小相似的输出映射，我们可以避免把池化放在一起 (Jain *et al.*, 2007)。另一种策略是单纯地产生一张低分辨率的标签网格 (Pinheiro and Collobert, 2014, 2015)。最后，原则上可以使用具有单位步幅的池化操作。

对图像逐个像素标记的一种策略是先产生图像标签的原始猜测，然后使用相邻像素之间的交互来修正该原始猜测。重复这个修正步骤数次对应于在每一步使用相同的卷积，该卷积在深层网络的最后几层之间共享权重 (Jain *et al.*, 2007)。这使得在层之间共享参数的连续的卷积层所执行的一系列运算，形成了一种特殊的循环神经网络 (Pinheiro and Collobert, 2014, 2015)。图 9.17 给出了这样一个循环卷积网络的结构。

一旦对每个像素都进行了预测，我们就可以使用各种方法来进一步处理这些预测，以便获得图像在区域上的分割 (Briggman *et al.*, 2009; Turaga *et al.*, 2010; Farabet *et al.*, 2013)。一般的想法是假设大片相连的像素倾向于对应着相同的标签。图模型可以描述相邻像素间的概率关系。或者，卷积网络可以被训练来最大化地近似图模型的训练目标 (Ning *et al.*, 2005; Thompson *et al.*, 2014)。

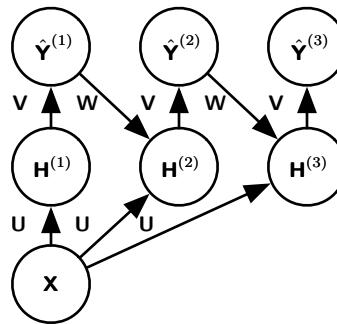


图 9.17: 用于像素标记的循环卷积网络的示例。输入是图像张量  $X$ ，它的轴对应图像的行、列和通道（红，绿，蓝）。目标是输出标签张量  $\hat{Y}$ ，它遵循每个像素的标签的概率分布。该张量的轴对应图像的行、列和不同类别。循环网络通过使用  $\hat{Y}$  的先前估计作为创建新估计的输入，来迭代地改善其估计，而不是单次输出  $\hat{Y}$ 。每个更新的估计使用相同的参数，并且估计可以如我们所愿地被改善任意多次。每一步使用的卷积核张量  $U$ ，是用来计算给定输入图像的隐藏表示的。核张量  $V$  用于产生给定隐藏值时标签的估计。除了第一步之外，核  $W$  都对  $\hat{Y}$  进行卷积来提供隐藏层的输入。在第一步中，此项由零代替。因为每一步使用相同的参数，所以这是一个循环网络的例子，如第十章所述。

## 9.7 数据类型

卷积网络使用的数据通常包含多个通道，每个通道是时间上或空间中某一点的不同观测量。参考表 9.1 来了解具有不同维数和通道数的数据类型的例子。

卷积网络用于视频的例子，可以参考 Chen *et al.* (2010)。

到目前为止，我们仅讨论了训练和测试数据中的每个样例都有相同的空间维度的情况。卷积网络的一个优点是它们还可以处理具有可变的空间尺度的输入。这些类型的输入不能用传统的基于矩阵乘法的神经网络来表示。这为卷积网络的使用提供了令人信服的理由，即使当计算开销和过拟合都不是主要问题时。

例如，考虑一组图像的集合，其中每个图像具有不同的高度和宽度。目前还不清楚如何用固定大小的权重矩阵对这样的输入进行建模。卷积就可以很直接地应用；核依据输入的大小简单地被使用不同次，并且卷积运算的输出也相应地放缩。卷积可以被视为矩阵乘法；相同的卷积核为每种大小的输入引入了一个不同大小的双重分块循环矩阵。有时，网络的输出允许和输入一样具有可变的大小，例如如果我们想要为输入的每个像素分配一个类标签。在这种情况下，不需要进一步的设计工作。在其他情况下，网络必须产生一些固定大小的输出，例如，如果我们想要为整个图

	单通道	多通道
1 维	音频波形：卷积的轴对应于时间。我们将时间离散化并且在每个时间点测量一次波形的振幅。	骨架动画 (skeleton animation) 数据：计算机渲染的 3D 角色动画是通过随时间调整“骨架”的姿势而生成的。在每个时间点，角色的姿势通过骨架中的每个关节的角度来描述。我们输入到卷积模型的数据的每个通道，表示一个关节关于一个轴的角度。
2 维	已经使用傅立叶变换预处理过的音频数据：我们可以将音频波形转换成 2 维张量，不同的行对应不同的频率，不同的列对应不同的时间点。在时间轴上使用卷积使模型等效于在时间上移动。在频率轴上使用卷积使得模型等效于在频率上移动，这使得在不同八度音阶中播放的相同旋律产生相同的表示，但处于网络输出中的不同高度。	彩色图像数据：其中一个通道包含红色像素，另一个包含绿色像素，最后一个包含蓝色像素。在图像的水平轴和竖直轴上移动卷积核，赋予了两个方向上平移等变性。
3 维	体积数据：这种数据一般来源于医学成像技术，例如 CT 扫描等。	彩色视频数据：其中一个轴对应着时间，另一个轴对应着视频帧的高度，最后一个对应着视频帧的宽度。

表 9.1: 用于卷积网络的不同数据格式的示例。

像指定单个类标签。在这种情况下，我们必须进行一些额外的设计步骤，例如插入一个池化层，池化区域的大小要与输入的大小成比例，以便保持固定数量的池化输出。这种策略的一些例子可以参考图 9.11。

注意，使用卷积处理可变尺寸的输入，仅对输入是因为包含对同种事物的不同量的观察（时间上不同长度的记录，空间上不同宽度的观察等）而导致的尺寸变化这种情况才有意义。如果输入是因为它可以选择性地包括不同种类的观察而具有可变尺寸，使用卷积是不合理的。例如，如果我们正在处理大学申请，并且我们的特征包括成绩等级和标准化测试分数，但不是每个申请人都进行了标准化测试，则使用相同的权重来对成绩特征和测试分数特征进行卷积是没有意义的。

## 9.8 高效的卷积算法

现代卷积网络的应用通常需要包含超过百万个单元的网络。利用并行计算资源的强大实现是很关键的，如第 12.1 节中所描述的。然而，在很多情况下，也可以通过选择适当的卷积算法来加速卷积。

卷积等效于使用傅立叶变换将输入与核都转换到频域、执行两个信号的逐点相乘，再使用傅立叶逆变换转换回时域。对于某些问题的规模，这种算法可能比离散卷积的朴素实现更快。

当一个  $d$  维的核可以表示成  $d$  个向量（每一维一个向量）的外积时，该核被称为可分离的（separable）。当核可分离时，朴素的卷积是低效的。它等价于组合  $d$  个一维卷积，每个卷积使用这些向量中的一个。组合方法显著快于使用它们的外积来执行一个  $d$  维的卷积。并且核也只要更少的参数来表示成向量。如果核在每一维都是  $w$  个元素宽，那么朴素的多维卷积需要  $O(w^d)$  的运行时间和参数存储空间，而可分离卷积只需要  $O(w \times d)$  的运行时间和参数存储空间。当然，并不是每个卷积都可以表示成这种形式。

设计更快的执行卷积或近似卷积，而不损害模型准确性的方法，是一个活跃的研究领域。甚至仅提高前向传播效率的技术也是有用的，因为在商业环境中，通常部署网络比训练网络还要耗资源。

## 9.9 随机或无监督的特征

通常，卷积网络训练中最昂贵的部分是学习特征。输出层的计算代价通常相对不高，因为在通过若干层池化之后作为该层输入的特征的数量较少。当使用梯度下降执行监督训练时，每步梯度计算需要完整地运行整个网络的前向传播和反向传播。减少卷积网络训练成本的一种方式是使用那些不是由监督方式训练得到的特征。

有三种基本策略可以不通过监督训练而得到卷积核。其中一种是简单地随机初始化它们。另一种是手动设计它们，例如设置每个核在一个特定的方向或尺度来检测边缘。最后，可以使用无监督的标准来学习核。例如，Coates *et al.* (2011) 将  $k$  均值聚类算法应用于小图像块，然后使用每个学得的中心作为卷积核。第三部分描述了更多的无监督学习方法。使用无监督的标准来学习特征，允许这些特征的确定与位于网络结构顶层的分类层相分离。然后只需提取一次全部训练集的特征，构造用于最后一层的新训练集。假设最后一层类似逻辑回归或者 SVM，那么学习最后一层通常是凸优化问题。

随机过滤器经常在卷积网络中表现得出乎意料得好 Jarrett *et al.* (2009b); Saxe *et al.* (2011); Pinto *et al.* (2011); Cox and Pinto (2011)。Saxe *et al.* (2011) 说明，由卷积和随后的池化组成的层，当赋予随机权重时，自然地变得具有频率选择性和平移不变性。他们认为这提供了一种廉价的方法来选择卷积网络的结构：首先通过仅训练最后一层来评估几个卷积网络结构的性能，然后选择最好的结构并使用更昂贵的方法来训练整个网络。

一个中间方法是学习特征，但是使用那种不需要在每个梯度计算步骤中都进行完整的前向和反向传播的方法。与多层感知机一样，我们使用贪心逐层预训练，单独训练第一层，然后一次性地从第一层提取所有特征，之后用那些特征单独训练第二层，以此类推。第八章描述了如何实现监督的贪心逐层预训练，第三部分将此扩展到了无监督的范畴。卷积模型的贪心逐层预训练的经典模型是卷积深度信念网络(Lee *et al.*, 2009)。卷积网络为我们提供了相对于多层感知机更进一步采用预训练策略的机会。并非一次训练整个卷积层，我们可以训练一小块模型，就像 Coates *et al.* (2011) 使用  $k$  均值做的那样。然后，我们可以用来自这个小块模型的参数来定义卷积层的核。这意味着使用无监督学习来训练卷积网络并且在训练的过程中完全不使用卷积是可能的。使用这种方法，我们可以训练非常大的模型，并且只在推断期间产生高计算成本 (Ranzato *et al.*, 2007c; Jarrett *et al.*, 2009b; Kavukcuoglu *et al.*, 2010; Coates *et al.*, 2013)。这种方法大约在 2007 到 2013 年间流行，当时标记的数

据集很小，并且计算能力有限。如今，大多数卷积网络以纯粹监督的方式训练，在每次训练迭代中使用通过整个网络的完整的前向和反向传播。

与其他无监督预训练的方法一样，使用这种方法的一些好处仍然难以说清。无监督预训练可以提供一些相对于监督训练的正则化，或者它可以简单地允许我们训练更大的结构，因为它的学习规则降低了计算成本。

## 9.10 卷积网络的神经科学基础

卷积网络也许是生物学启发人工智能的最为成功的案例。虽然卷积网络也经过许多其他领域的指导，但是神经网络的一些关键设计原则来自于神经科学。

卷积网络的历史始于神经科学实验，远早于相关计算模型的发展。为了确定关于哺乳动物视觉系统如何工作的许多最基本的事实，神经生理学家 David Hubel 和 Torsten Wiesel 合作多年 (Hubel and Wiesel, 1959, 1962, 1968)。他们的成就最终获得了诺贝尔奖。他们的发现对当代深度学习模型有最大影响的是基于记录猫的单个神经元的活动。他们观察了猫的脑内神经元如何响应投影在猫前面屏幕上精确位置的图像。他们的伟大发现是，处于视觉系统较为前面的神经元对非常特定的光模式（例如精确定向的条纹）反应最强烈，但对其他模式几乎没有反应。

他们的工作有助于表征大脑功能的许多方面，这些方面超出了本书的范围。从深度学习的角度来看，我们可以专注于简化的、草图形式的大脑功能视图。

在这个简化的视图中，我们关注被称为 V1 的大脑的一部分，也称为 **初级视觉皮层** (primary visual cortex)。V1 是大脑对视觉输入开始执行显著高级处理的第一个区域。在该草图视图中，图像是由光到达眼睛并刺激视网膜（眼睛后部的光敏组织）形成的。视网膜中的神经元对图像执行一些简单的预处理，但是基本不改变它被表示的方式。然后图像通过视神经和称为外侧膝状核的脑部区域。这些解剖区域的主要作用是仅仅将信号从眼睛传递到位于头后部的 V1。

卷积网络层被设计为描述 V1 的三个性质：

1. V1 可以进行空间映射。它实际上具有二维结构来反映视网膜中的图像结构。例如，到达视网膜下半部的光仅影响 V1 相应的一半。卷积网络通过用二维映射定义特征的方式来描述该特性。
2. V1 包含许多 **简单细胞** (simple cell)。简单细胞的活动在某种程度上可以概括

为在一个小的空间位置感受野内的图像的线性函数。卷积网络的检测器单元被设计为模拟简单细胞的这些性质。

3. V1 还包括许多 **复杂细胞** (complex cell)。这些细胞响应类似于由简单细胞检测的那些特征，但是复杂细胞对于特征的位置微小偏移具有不变性。这启发了卷积网络的池化单元。复杂细胞对于照明中的一些变化也是不变的，不能简单地通过在空间位置上池化来刻画。这些不变性激发了卷积网络中的一些跨通道池化策略，例如 maxout 单元 (Goodfellow *et al.*, 2013b)。

虽然我们最了解 V1，但是一般认为相同的基本原理也适用于视觉系统的其他区域。在我们视觉系统的草图视图中，当我们逐渐深入大脑时，遵循池化的基本探测策略被反复执行。当我们穿过大脑的多个解剖层时，我们最终找到了响应一些特定概念的细胞，并且这些细胞对输入的很多种变换都具有不变性。这些细胞被昵称为“祖母细胞”——这个想法是一个人可能有一个神经元，当看到他祖母的照片时该神经元被激活，无论祖母是出现在照片的左边或右边，无论照片是她的脸部的特写镜头还是她的全身照，也无论她处在光亮还是黑暗中，等等。

这些祖母细胞已经被证明确实存在于人脑中，在一个被称为内侧颞叶的区域 (Quiroga *et al.*, 2005)。研究人员测试了单个神经元是否会响应名人的照片。他们发现了后来被称为“Halle Berry 神经元”的神经元：由 Halle Berry 的概念激活的单个神经元。当一个人看到 Halle Berry 的照片，Halle Berry 的图画，甚至包含单词“Halle Berry”的文本时，这个神经元会触发。当然，这与 Halle Berry 本人无关；其他神经元会对 Bill Clinton, Jennifer Aniston 等的出现做出响应。

这些内侧颞叶神经元比现代卷积网络更通用一些，这些网络在读取名称时不会自动联想到识别人或对象。与卷积网络的最后一层在特征上最接近的类比是称为颞下皮质 (IT) 的脑区。当查看一个对象时，信息从视网膜经 LGN 流到 V1，然后到 V2, V4，之后是 IT。这发生在瞥见对象的前 100ms 内。如果允许一个人继续观察对象更多的时间，那么信息将开始回流，因为大脑使用自上而下的反馈来更新较低级脑区中的激活。然而，如果我们打断人的注视，并且只观察前 100ms 内的大多数前向激活导致的放电率，那么 IT 被证明与卷积网络非常相似。卷积网络可以预测 IT 放电率，并且在执行对象识别任务时与人类（时间有限的情况）非常类似 (DiCarlo, 2013)。

话虽如此，卷积网络和哺乳动物的视觉系统之间还是有许多区别。这些区别有一些是计算神经科学家所熟知的，但超出了本书的范围。还有一些区别尚未知晓，因

为关于哺乳动物视觉系统如何工作的许多基本问题仍未得到回答。简要列表如下：

- 人眼大部分是非常低的分辨率，除了一个被称为中央凹（fovea）的小块。中央凹仅观察在手臂长度距离内一块拇指大小的区域。虽然我们觉得我们可以看到高分辨率的整个场景，但这是由我们的大脑的潜意识部分创建的错觉，因为它缝合了我们瞥见的若干个小区域。大多数卷积网络实际上接收大的全分辨率的照片作为输入。人类大脑控制几次眼动，称为扫视（saccade），以瞥见场景中最显眼的或任务相关的部分。将类似的注意力机制融入深度学习模型是一个活跃的研究方向。在深度学习的背景下，注意力机制对于自然语言处理是最成功的，参考第 12.4.5.1 节。研究者已经研发了几种具有视觉机制的视觉模型，但到目前为止还没有成为主导方法 (Larochelle and Hinton, 2010; Denil *et al.*, 2012)。
- 人类视觉系统集成了许多其他感觉，例如听觉，以及像我们的心情和想法一样的因素。卷积网络迄今为止纯粹是视觉的。
- 人类视觉系统不仅仅用于识别对象。它能够理解整个场景，包括许多对象和对象之间的关系，以及处理我们的身体与世界交互所需的丰富的三维几何信息。卷积网络已经应用于这些问题中的一些，但是这些应用还处于起步阶段。
- 即使像 V1 这样简单的大脑区域也受到来自较高级别的反馈的严重影响。反馈已经在神经网络模型中被广泛地探索，但还没有被证明提供了引人注目的改进。
- 虽然前馈 IT 放电频率刻画了与卷积网络特征很多相同的信息，但是仍不清楚中间计算的相似程度。大脑可能使用非常不同的激活和池化函数。单个神经元的激活可能不能用单个线性过滤器的响应来很好地表征。最近的 V1 模型涉及对每个神经元的多个二次过滤器 (Rust *et al.*, 2005)。事实上，我们的“简单细胞”和“复杂细胞”的草图图片可能并没有区别；简单细胞和复杂细胞可能是相同种类的细胞，但是它们的“参数”使得它们能够实现从我们所说的“简单”到“复杂”的连续的行为。

还值得一提的是，神经科学很少告诉我们该如何训练卷积网络。具有跨多个空间位置的参数共享的模型结构，可以追溯到早期关于视觉的联结主义模型 (Marr and Poggio, 1976)，但是这些模型没有使用现代的反向传播算法和梯度下降。例如，(Fukushima, 1980) 结合了现代卷积网络的大多数模型结构设计元素，但依赖于层次化的无监督聚类算法。

Lang and Hinton (1988) 引入反向传播来训练时延神经网络 (time delay neural network, TDNN)。使用当代术语来说, TDNN 是用于时间序列的一维卷积网络。用于这些模型的反向传播不受任何神经科学观察的启发, 并且被一些人认为是生物不可信的。在基于使用反向传播训练的 TDNN 成功之后, LeCun *et al.* (1989) 通过将相同的训练算法应用于图像的 2 维卷积来发展现代卷积网络。

到目前为止, 我们已经描述了简单细胞对于某些特征是如何呈现粗略的线性和选择性, 复杂细胞是如何更加的非线性, 并且对于这些简单细胞特征的某些变换具有不变性, 以及在选择性和不变性之间交替放置的层可以产生对非常特定现象的祖母细胞。我们还没有精确描述这些单个细胞检测到了什么。在深度非线性网络中, 可能难以理解单个细胞的功能。第一层中的简单细胞相对更容易分析, 因为它们的响应由线性函数驱动。在人工神经网络中, 我们可以直接显示卷积核的图像, 来查看卷积层的相应通道是如何响应的。在生物神经网络中, 我们不能访问权重本身。相反, 我们在神经元自身中放置一个电极, 在动物视网膜前显示几个白噪声图像样本, 并记录这些样本中的每一个是如何导致神经元激活的。然后, 我们可以对这些响应拟合线性模型, 以获得近似的神经元权重。这种方法被称为 **反向相关** (reverse correlation) (Ringach and Shapley, 2004)。

反向相关向我们表明, 大多数的 V1 细胞具有由 **Gabor 函数** (Gabor function) 所描述的权重。Gabor 函数描述在图像中的 2 维点处的权重。我们可以认为图像是 2 维坐标  $I(x, y)$  的函数。类似地, 我们可以认为简单细胞是在图像中的一组位置采样, 这组位置由一组  $x$  坐标  $\mathbb{X}$  和一组  $y$  坐标  $\mathbb{Y}$  来定义, 并且使用的权重  $w(x, y)$  也是位置的函数。从这个观点来看, 简单细胞对于图像的响应由下式给出

$$s(I) = \sum_{x \in \mathbb{X}} \sum_{y \in \mathbb{Y}} w(x, y) I(x, y). \quad (9.15)$$

特别地,  $w(x, y)$  采用 Gabor 函数的形式:

$$w(x, y; \alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau) = \alpha \exp(-\beta_x x'^2 - \beta_y y'^2) \cos(f x' + \phi), \quad (9.16)$$

其中

$$x' = (x - x_0) \cos(\tau) + (y - y_0) \sin(\tau) \quad (9.17)$$

以及

$$y' = -(x - x_0) \sin(\tau) + (y - y_0) \cos(\tau). \quad (9.18)$$

这里  $\alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau$  都是控制 Gabor 函数性质的参数。图 9.18 给出了 Gabor 函数在不同参数集上的一些例子。

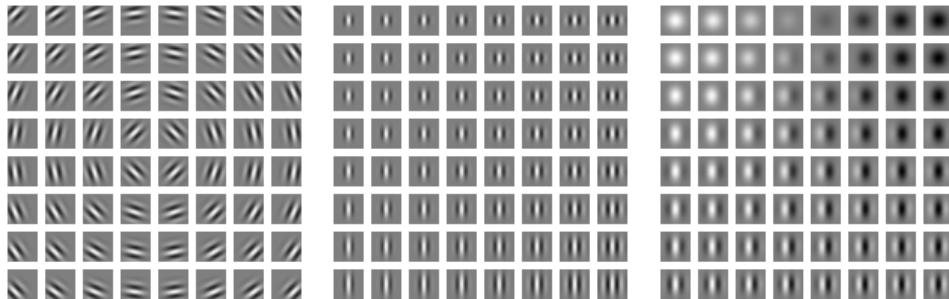


图 9.18: 具有各种参数设置的Gabor 函数。白色表示绝对值大的正权重，黑色表示绝对值大的负权重，背景灰色对应于零权重。(左) 控制坐标系的参数具有不同值的Gabor 函数，这些参数包括： $x_0$ 、 $y_0$  和  $\gamma$ 。在该网格中的每个Gabor 函数被赋予和它在网格中的位置成比例的  $x_0$  和  $y_0$  的值，并且  $\tau$  被选择为使得每个 Gabor 过滤器对从网格中心辐射出的方向非常敏感。对于其他两幅图， $x_0$ 、 $y_0$  和  $\gamma$  固定为零。(中) 具有不同高斯比例参数  $\beta_x$  和  $\beta_y$  的Gabor 函数。当我们从左到右通过网格时，Gabor 函数被设置为增加宽度(减少  $\beta_x$ )；当我们从上到下通过网格时，Gabor 函数被设置为增加高度(减少  $\beta_y$ )。对于其他两幅图， $\beta$  值固定为图像宽度的 1.5 倍。(右) 具有不同的正弦参数  $f$  和  $\phi$  的Gabor 函数。当我们从上到下移动时， $f$  增加；当我们从左到右移动时， $\phi$  增加。对于其他两幅图， $\phi$  固定为 0， $f$  固定为图像宽度的 5 倍。

参数  $x_0$ ,  $y_0$  和  $\tau$  定义坐标系。我们平移和旋转  $x$  和  $y$  来得到  $x'$  和  $y'$ 。具体地，简单细胞会响应以点  $(x_0, y_0)$  为中心的图像特征，并且当我们沿着从水平方向旋转  $\tau$  弧度的线移动时，简单细胞将响应亮度的变化。

作为  $x'$  和  $y'$  的函数，函数  $w$  会响应当我们沿着  $x'$  移动时的亮度变化。它有两个重要的因子：一个是高斯函数，另一个是余弦函数。

高斯因子  $\alpha \exp(-\beta_x x'^2 - \beta_y y'^2)$  可以被视为阈值项，用于保证简单细胞仅对接近  $x'$  和  $y'$  都为零点处的值响应，换句话说，接近细胞接受域的中心。尺度因子  $\alpha$  调整简单细胞响应的总的量级，而  $\beta_x$  和  $\beta_y$  控制接受域消退的速度。

余弦因子  $\cos(f x' + \phi)$  控制简单细胞如何响应延  $x'$  轴的亮度改变。参数  $f$  控制余弦的频率， $\phi$  控制它的相位偏移。

合在一起，简单细胞的这个草图视图意味着，简单细胞对在特定位置处、特定方向上、特定空间频率的亮度进行响应。当图像中的光波与细胞的权重具有相同的相位时，简单细胞是最兴奋的。这种情况发生在当图像亮时，它的权重为正，而图像暗时，它的权重为负。当光波与权重完全异相时，简单细胞被抑制——当图像较

暗时，它的权重为正；较亮时，它的权重为负。

复杂细胞的草图视图是它计算包含两个简单细胞响应的 2 维向量的  $L^2$  范数： $c(I) = \sqrt{s_0(I)^2 + s_1(I)^2}$ 。一个重要的特殊情况是当  $s_1$  和  $s_0$  具有除  $\phi$  以外都相同的参数，并且  $\phi$  被设置为使得  $s_1$  与  $s_0$  相位相差四分之一周期时。在这种情况下， $s_0$  和  $s_1$  形成 **象限对** (quadrature pair)。当高斯重新加权的图像  $I(x, y) \exp(-\beta_x x'^2 - \beta_y y^2)$  包含具有频率  $f$ 、在方向  $\tau$  上、接近  $(x_0, y_0)$  的高振幅正弦波时，用先前方法定义的复杂细胞会响应，并且不管该波的相位偏移。换句话说，复杂细胞对于图像在方向  $\tau$  上的微小变换或者翻转图像（用白色代替黑色，反之亦然）具有不变性。

神经科学和机器学习之间最显著的对应关系，是从视觉上比较机器学习模型学得的特征与使用 V1 得到的特征。Olshausen and Field (1996) 说明，一个简单的无监督学习算法，稀疏编码，学习的特征具有与简单细胞类似的感受野。从那时起，我们发现，当应用于自然图像时，极其多样的统计学习算法学习类 Gabor 函数的特征。这包括大多数深度学习算法，它们在其第一层中学习这些特征。图 9.19 给出了一些例子。因为如此众多不同的学习算法学习边缘检测器，所以很难仅基于学习算法学得的特征，来断定哪一个特定的学习算法是“正确”的大脑模型（虽然，当应用于自然图像时，如果一个算法不能学得某种检测器时，它能够作为一种否定标志）。这些特征是自然图像的统计结构的重要部分，并且可以通过许多不同的统计建模方法来重新获得。读者可以参考 (Hyvärinen et al., 2009) 来获得自然图像统计领域的综述。

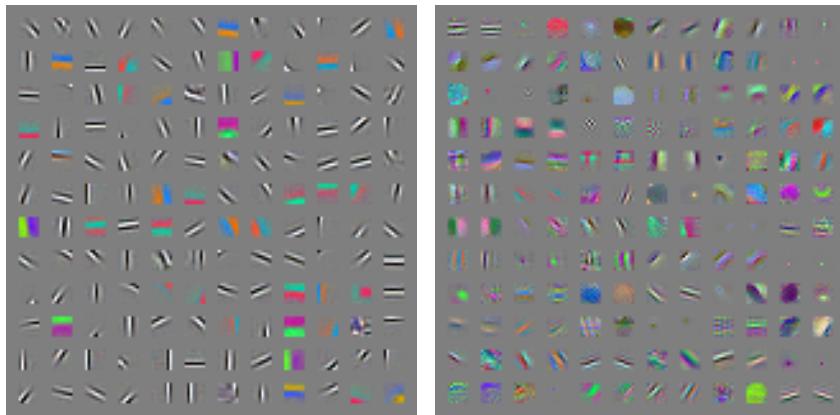


图 9.19: 许多机器学习算法在应用于自然图像时, 会学习那些用来检测边缘或边缘的特定颜色的特征。这些特征检测器使人联想到已知存在于初级视觉皮层中的 Gabor 函数。(左) 通过应用于小图像块的无监督学习算法 (尖峰和平板稀疏编码) 学得的权重。(右) 由完全监督的卷积 maxout 网络的第一层学得的卷积核。相邻的一对过滤器驱动相同的 maxout 单元。

## 9.11 卷积网络与深度学习的历史

卷积网络在深度学习的历史中发挥了重要作用。它们是将研究大脑获得的深刻理解成功用于机器学习应用的关键例子。它们也是首批表现良好的深度模型之一, 远远早于任意深度模型被认为是可行之前。卷积网络也是第一个解决重要商业应用的神经网络, 并且仍然处于当今深度学习商业应用的前沿。例如, 在 20 世纪 90 年代, AT&T 的神经网络研究小组开发了一个用于读取支票的卷积网络(LeCun *et al.*, 1998c)。到 90 年代末, NEC 部署的这个系统已经被用于读取美国 10% 以上的支票。后来, 微软部署了若干个基于卷积网络的 OCR 和手写识别系统 (Simard *et al.*, 2003)。关于卷积网络的这种应用和更现代应用的更多细节, 参考第十二章。读者可以参考 (LeCun *et al.*, 2010) 了解 2010 年之前的更为深入的卷积网络历史。

卷积网络也被用作在许多比赛中的取胜手段。当前对深度学习的商业兴趣的高度始于Krizhevsky *et al.* (2012a) 赢得了 ImageNet 对象识别挑战, 但是在那之前, 卷积网络也已经被用于赢得前些年影响较小的其他机器学习和计算机视觉竞赛了。

卷积网络是第一批能使用反向传播有效训练的深度网络之一。现在仍不完全清楚为什么卷积网络在一般的反向传播网络被认为已经失败时反而成功了。这可能可以简单地归结为卷积网络比全连接网络计算效率更高, 因此使用它们运行多个实验

并调整它们的实现和超参数更容易。更大的网络也似乎更容易训练。利用现代硬件，大型全连接的网络在许多任务上也表现得很合理，即使使用过去那些全连接网络被认为不能工作得很好的数据集和当时流行的激活函数时，现在也能执行得很好。心理可能是神经网络成功的主要阻碍（实践者没有期望神经网络有效，所以他们没有认真努力地使用神经网络）。无论如何，幸运的是卷积网络在几十年前就表现良好。在许多方面，它们为余下的深度学习传递火炬，并为一般的神经网络被接受铺平了道路。

卷积网络提供了一种方法来特化神经网络，使其能够处理具有清楚的网格结构拓扑的数据，以及将这样的模型扩展到非常大的规模。这种方法在二维图像拓扑上是最成功的。为了处理一维序列数据，我们接下来转向神经网络框架的另一种强大的特化：循环神经网络。

# 第十章 序列建模：循环和递归网络

循环神经网络 (recurrent neural network) 或 RNN (Rumelhart *et al.*, 1986c) 是一类用于处理序列数据的神经网络。就像卷积网络是专门用于处理网格化数据  $\mathbf{X}$  (如一个图像) 的神经网络, 循环神经网络是专门用于处理序列  $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}$  的神经网络。正如卷积网络可以很容易地扩展到具有很大宽度和高度的图像, 以及处理大小可变的图像, 循环网络可以扩展到更长的序列 (比不基于序列的特化网络长得多)。大多数循环网络也能处理可变长度的序列。

从多层网络出发到循环网络, 我们需要利用上世纪 80 年代机器学习和统计模型早期思想的优点: 在模型的不同部分共享参数。参数共享使得模型能够扩展到不同形式的样本 (这里指不同长度的样本) 并进行泛化。如果我们在每个时间点都有一个单独的参数, 我们不但不能泛化到训练时没有见过序列长度, 也不能在时间上共享不同序列长度和不同位置的统计强度。当信息的特定部分会在序列内多个位置出现时, 这样的共享尤为重要。例如, 考虑这两句话: “I went to Nepal in 2009” 和 “In 2009, I went to Nepal.” 如果我们让一个机器学习模型读取这两个句子, 并提取叙述者去Nepal的年份, 无论 “2009 年” 是作为句子的第六个单词还是第二个单词出现, 我们都希望模型能认出 “2009 年” 作为相关资料片段。假设我们要训练一个处理固定长度句子的前馈网络。传统的全连接前馈网络会给每个输入特征分配一个单独的参数, 所以需要分别学习句子每个位置的所有语言规则。相比之下, 循环神经网络在几个时间步内共享相同的权重, 不需要分别学习句子每个位置的所有语言规则。

一个相关的想法是在 1 维时间序列上使用卷积。这种卷积方法是时延神经网络的基础 (Lang and Hinton, 1988; Waibel *et al.*, 1989; Lang *et al.*, 1990)。卷积操作允许网络跨时间共享参数, 但是浅层的。卷积的输出是一个序列, 其中输出中的每一项是相邻几项输入的函数。参数共享的概念体现在每个时间步中使用的相同卷积核。循环神经网络以不同的方式共享参数。输出的每一项是前一项的函数。输出的

每一项对先前的输出应用相同的更新规则而产生。这种循环方式导致参数通过很深的计算图共享。

为简单起见，我们说的 RNN 是指在序列上的操作，并且该序列在时刻  $t$ （从 1 到  $\tau$ ）包含向量  $\mathbf{x}^{(t)}$ 。在实际情况中，循环网络通常在序列的小批量上操作，并且小批量的每项具有不同序列长度  $\tau$ 。我们省略了小批量索引来简化记号。此外，时间步索引不必是字面上现实世界中流逝的时间。有时，它仅表示序列中的位置。RNN 也可以应用于跨越两个维度的空间数据（如图像）。当应用于涉及时间的数据，并且将整个序列提供给网络之前就能观察到整个序列时，该网络可具有关于时间向后的连接。

本章将计算图的思想扩展到包括循环。这些周期代表变量自身的值在未来某一时间步对自身值的影响。这样的计算图允许我们定义循环神经网络。然后，我们描述许多构建、训练和使用循环神经网络的不同方式。

本章将简要介绍循环神经网络，为获取更多详细信息，我们建议读者参考 Graves (2012) 的著作。

## 10.1 展开计算图

计算图是形式化一组计算结构的方式，如那些涉及将输入和参数映射到输出和损失的计算。综合的介绍请参考第 6.5.1 节。本节，我们对展开（unfolding）递归或循环计算得到的重复结构进行解释，这些重复结构通常对应于一个事件链。展开（unfolding）这个计算图将导致深度网络结构中的参数共享。

例如，考虑动态系统的经典形式：

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}; \boldsymbol{\theta}), \quad (10.1)$$

其中  $\mathbf{s}^{(t)}$  称为系统的状态。

$\mathbf{s}$  在时刻  $t$  的定义需要参考时刻  $t - 1$  时同样的定义，因此式 (10.1) 是循环的。

对有限时间步  $\tau$ ,  $\tau - 1$  次应用这个定义可以展开这个图。例如  $\tau = 3$ ，我们对式 (10.1) 展开，可以得到：

$$\mathbf{s}^{(3)} = f(\mathbf{s}^{(2)}; \boldsymbol{\theta}) \quad (10.2)$$

$$= f(f(\mathbf{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta}). \quad (10.3)$$

以这种方式重复应用定义，展开等式，就能得到不涉及循环的表达。现在我们可以使用传统的有向无环计算图呈现这样的表达。

式(10.1)和式(10.3)的展开计算图如图10.1所示。

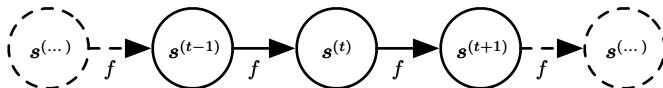


图10.1：将式(10.1)描述的经典动态系统表示为展开的计算图。每个节点表示在某个时刻 $t$ 的状态，并且函数 $f$ 将 $t$ 处的状态映射到 $t+1$ 处的状态。所有时间步都使用相同的参数（用于参数化 $f$ 的相同 $\theta$ 值）。

作为另一个例子，让我们考虑由外部信号 $\mathbf{x}^{(t)}$ 驱动的动态系统，

$$\mathbf{s}^{(t)} = f(\mathbf{s}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta}), \quad (10.4)$$

我们可以看到，当前状态包含了整个过去序列的信息。

循环神经网络可以通过许多不同的方式建立。就像几乎所有函数都可以被认为是前馈网络，本质上任何涉及循环的函数都可以被认为是一个循环神经网络。

很多循环神经网络使用式(10.5)或类似的公式定义隐藏单元的值。为了表明状态是网络的隐藏单元，我们使用变量 $\mathbf{h}$ 代表状态重写式(10.4)：

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \boldsymbol{\theta}), \quad (10.5)$$

如图10.2所示，典型RNN会增加额外的架构特性，如读取状态信息 $\mathbf{h}$ 进行预测的输出层。

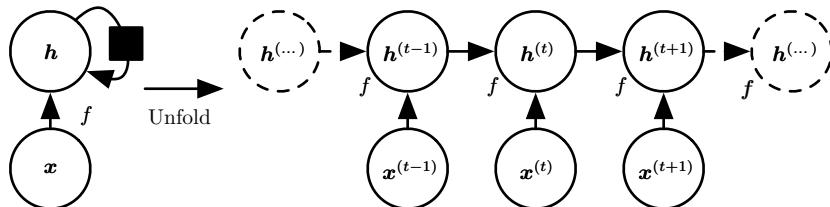


图10.2：没有输出的循环网络。此循环网络只处理来自输入 $\mathbf{x}$ 的信息，将其合并到经过时间向前传播的状态 $\mathbf{h}$ 。(左)回路原理图。黑色方块表示单个时间步的延迟。(右)同一网络被视为展开的计算图，其中每个节点现在与一个特定的时间实例相关联。

当训练循环网络根据过去预测未来时，网络通常要学会使用  $\mathbf{h}^{(t)}$  作为过去序列（直到  $t$ ）与任务相关方面的有损摘要。此摘要一般而言一定是有损的，因为其映射任意长度的序列  $(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \dots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)})$  到一固定长度的向量  $\mathbf{h}^{(t)}$ 。根据不同的训练准则，摘要可能选择性地精确保留过去序列的某些方面。例如，如果在统计语言建模中使用的 RNN，通常给定前一个词预测下一个词，可能没有必要存储时刻  $t$  前输入序列中的所有信息；而仅仅存储足够预测句子其余部分的信息。最苛刻的情况是我们要求  $\mathbf{h}^{(t)}$  足够丰富，并能大致恢复输入序列，如自编码器框架（第十四章）。

式(10.5)可以用两种不同的方式绘制。一种方法是为可能在模型的物理实现中存在的部分赋予一个节点，如生物神经网络。在这个观点下，网络定义了实时操作的回路，如图 10.2 的左侧，其当前状态可以影响其未来的状态。在本章中，我们使用回路图的黑色方块表明在时刻  $t$  的状态到时刻  $t+1$  的状态单个时刻延迟中的相互作用。另一个绘制 RNN 的方法是展开的计算图，其中每一个组件由许多不同的变量表示，每个时间步一个变量，表示在该时间点组件的状态。每个时间步的每个变量绘制为计算图的一个独立节点，如图 10.2 的右侧。我们所说的展开是将左图中的回路映射为右图中包含重复组件的计算图的操作。目前，展开图的大小取决于序列长度。

我们可以用一个函数  $g^{(t)}$  代表经  $t$  步展开后的循环：

$$\mathbf{h}^{(t)} = g^{(t)}(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \dots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)}) \quad (10.6)$$

$$= f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \theta). \quad (10.7)$$

函数  $g^{(t)}$  将全部的过去序列  $(\mathbf{x}^{(t)}, \mathbf{x}^{(t-1)}, \mathbf{x}^{(t-2)}, \dots, \mathbf{x}^{(2)}, \mathbf{x}^{(1)})$  作为输入来生成当前状态，但是展开的循环架构允许我们将  $g^{(t)}$  分解为函数  $f$  的重复应用。因此，展开过程引入两个主要优点：

1. 无论序列的长度，学成的模型始终具有相同的输入大小，因为它指定的是从一种状态到另一种状态的转移，而不是在可变长度的历史状态上操作。
2. 我们可以在每个时间步使用相同参数的相同转移函数  $f$ 。

这两个因素使得学习在所有时间步和所有序列长度上操作单一的模型  $f$  是可能的，而不需要在所有可能时间步学习独立的模型  $g^{(t)}$ 。学习单一的共享模型允许泛化到没有见过的序列长度（没有出现在训练集中），并且估计模型所需的训练样本远远少于不带参数共享的模型。

无论是循环图和展开图都有其用途。循环图简洁。展开图能够明确描述其中的计算流程。展开图还通过显式的信息流动路径帮助说明信息在时间上向前（计算输出和损失）和向后（计算梯度）的思想。

## 10.2 循环神经网络

基于第 10.1 节中的图展开和参数共享的思想，我们可以设计各种循环神经网络。

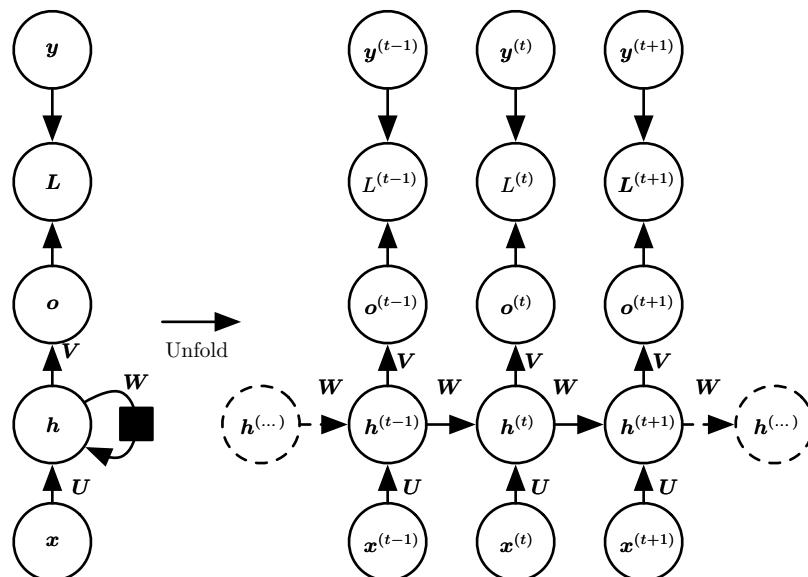


图 10.3: 计算循环网络(将  $x$  值的输入序列映射到输出值  $o$  的对应序列)训练损失的计算图。损失  $L$  衡量每个  $o$  与相应的训练目标  $y$  的距离。当使用 softmax 输出时，我们假设  $o$  是未归一化的对数概率。损失  $L$  内部计算  $\hat{y} = \text{softmax}(o)$ ，并将其与目标  $y$  比较。RNN 输入到隐藏的连接由权重矩阵  $U$  参数化，隐藏到隐藏的循环连接由权重矩阵  $W$  参数化以及隐藏到输出的连接由权重矩阵  $V$  参数化。式(10.8)定义了该模型中的前向传播。(左) 使用循环连接绘制的 RNN 和它的损失。(右) 同一网络被视为展开的计算图，其中每个节点现在与一个特定的时间实例相关联。

循环神经网络中一些重要的设计模式包括以下几种：

1. 每个时间步都有输出，并且隐藏单元之间有循环连接的循环网络，如图 10.3 所示。

2. 每个时间步都产生一个输出，只有当前时刻的输出到下个时刻的隐藏单元之间有循环连接的循环网络，如图 10.4 所示。
3. 隐藏单元之间存在循环连接，但读取整个序列后产生单个输出的循环网络，如图 10.5 所示。

图 10.3 是非常具有代表性的例子，我们将会在本章大部分涉及这个例子。

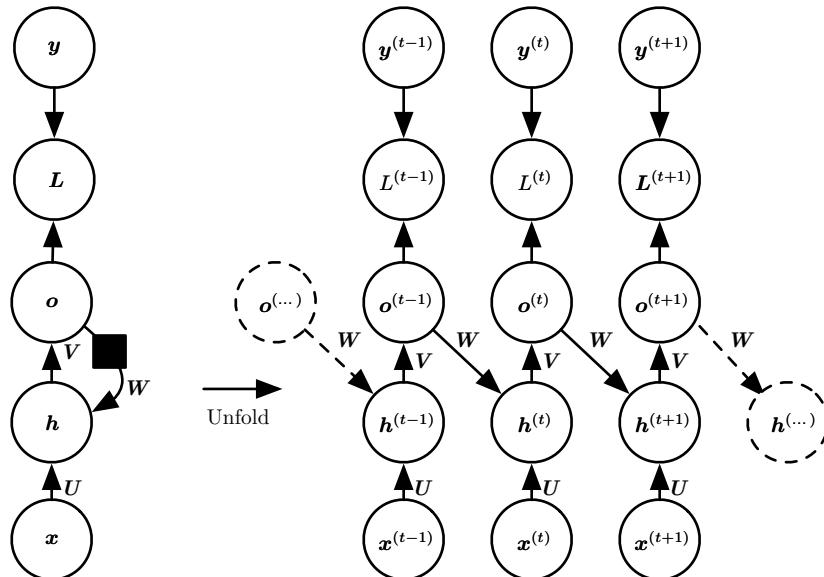


图 10.4: 此类 RNN 的唯一循环是从输出到隐藏层的反馈连接。在每个时间步  $t$ ，输入为  $\mathbf{x}_t$ ，隐藏层激活为  $\mathbf{h}^{(t)}$ ，输出为  $\mathbf{o}^{(t)}$ ，目标为  $\mathbf{y}^{(t)}$ ，损失为  $L^{(t)}$ 。(左)回路原理图。(右)展开的计算图。这样的 RNN 没有图 10.3 表示的 RNN 那样强大（只能表示更小的函数集合）。图 10.3 中的 RNN 可以选择将其想要的关于过去的任何信息放入隐藏表示  $\mathbf{h}$  中并且将  $\mathbf{h}$  传播到未来。该图中的 RNN 被训练为将特定输出值放入  $\mathbf{o}$  中，并且  $\mathbf{o}$  是允许传播到未来的唯一信息。此处没有从  $\mathbf{h}$  前向传播的直接连接。之前的  $\mathbf{h}$  仅通过产生的预测间接地连接到当前。 $\mathbf{o}$  通常缺乏过去的重要信息，除非它非常高维且内容丰富。这使得该图中的 RNN 不那么强大，但是它更容易训练，因为每个时间步可以与其他时间步分离训练，允许训练期间更多的并行化，如第 10.2.1 节所述。

任何图灵可计算的函数都可以通过这样一个有限维的循环网络计算，在这个意义上图 10.3 和式 (10.8) 的循环神经网络是万能的。RNN 经过若干时间步后读取输出，这与由图灵机所用的时间步是渐近线性的，与输入长度也是渐近线性的 (Siegelmann and Sontag, 1991; Siegelmann, 1995; Siegelmann and Sontag, 1995;

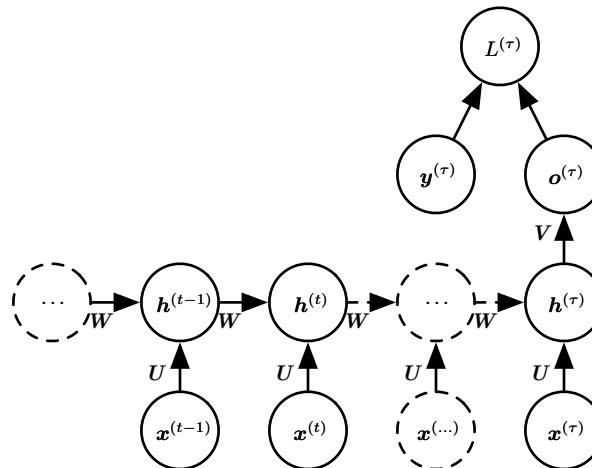


图 10.5: 关于时间展开的循环神经网络，在序列结束时具有单个输出。这样的网络可以用于概括序列并产生用于进一步处理的固定大小的表示。在结束处可能存在目标（如此处所示），或者通过更下游模块的反向传播来获得输出  $\mathbf{o}^{(t)}$  上的梯度。

Hyötyniemi, 1996)。由图灵机计算的函数是离散的，所以这些结果都是函数的具体实现，而不是近似。RNN 作为图灵机使用时，需要一个二进制序列作为输入，其输出必须离散化以提供二进制输出。利用单个有限大小的特定 RNN 计算在此设置下的所有函数是可能的（Siegelmann and Sontag (1995) 用了 886 个单元）。图灵机的“输入”是要计算函数的详细说明 (specification)，所以模拟此图灵机的相同网络足以应付所有问题。用于证明的理论 RNN 可以通过激活和权重（由无限精度的有理数表示）来模拟无限堆栈。

现在我们研究图 10.3 中 RNN 的前向传播公式。这个图没有指定隐藏单元的激活函数。我们假设使用双曲正切激活函数。此外，图中没有明确指定何种形式的输出和损失函数。我们假定输出是离散的，如用于预测词或字符的 RNN。表示离散变量的常规方式是把输出  $\mathbf{o}$  作为每个离散变量可能值的非标准化对数概率。然后，我们可以应用 softmax 函数后续处理后，获得标准化后概率的输出向量  $\hat{\mathbf{y}}$ 。RNN 从特定的初始状态  $\mathbf{h}^{(0)}$  开始前向传播。从  $t = 1$  到  $t = \tau$  的每个时间步，我们应用以下

更新方程：

$$\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}\mathbf{x}^{(t)}, \quad (10.8)$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{a}^{(t)}), \quad (10.9)$$

$$\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}, \quad (10.10)$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{o}^{(t)}), \quad (10.11)$$

其中的参数的偏置向量  $\mathbf{b}$  和  $\mathbf{c}$  连同权重矩阵  $\mathbf{U}$ 、 $\mathbf{V}$  和  $\mathbf{W}$ ，分别对应于输入到隐藏、隐藏到输出和隐藏到隐藏的连接。这个循环网络将一个输入序列映射到相同长度的输出序列。与  $\mathbf{x}$  序列配对的  $\mathbf{y}$  的总损失就是所有时间步的损失之和。例如， $L^{(t)}$  为给定的  $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}$  后  $\mathbf{y}^{(t)}$  的负对数似然，则

$$L(\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}\}, \{\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau)}\}) \quad (10.12)$$

$$= \sum_t L^{(t)} \quad (10.13)$$

$$= - \sum_t \log p_{\text{model}}(\mathbf{y}^{(t)} \mid \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\}), \quad (10.14)$$

其中  $p_{\text{model}}(\mathbf{y}^{(t)} \mid \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}\})$  需要读取模型输出向量  $\hat{\mathbf{y}}^{(t)}$  中对应于  $y^{(t)}$  的项。关于各个参数计算这个损失函数的梯度是计算成本很高的操作。梯度计算涉及执行一次前向传播（如在图 10.3 展开图中从左到右的传播），接着是由右到左的反向传播。运行时间是  $\mathcal{O}(\tau)$ ，并且不能通过并行化来降低，因为前向传播图是固有循序的；每个时间步只能一前一后地计算。前向传播中的各个状态必须保存，直到它们反向传播中被再次使用，因此内存代价也是  $\mathcal{O}(\tau)$ 。应用于展开图且代价为  $\mathcal{O}(\tau)$  的反向传播算法称为 **通过时间反向传播** (back-propagation through time, BPTT)，将在第 10.2.2 节进一步讨论。因此隐藏单元之间存在循环的网络非常强大但训练代价也很大。我们是否有其他选择呢？

### 10.2.1 导师驱动过程和输出循环网络

仅在一个时间步的输出和下一个时间步的隐藏单元间存在循环连接的网络（示于图 10.4）确实没有那么强大（因为缺乏隐藏到隐藏的循环连接）。例如，它不能模拟通用图灵机。因为这个网络缺少隐藏到隐藏的循环，它要求输出单元捕捉用于预测未来的关于过去的所有信息。因为输出单元明确地训练成匹配训练集的目标，它们不太能捕获关于过去输入历史的必要信息，除非用户知道如何描述系统的全部状

态，并将它作为训练目标的一部分。消除隐藏到隐藏循环的优点在于，任何基于比较时刻  $t$  的预测和时刻  $t$  的训练目标的损失函数中的所有时间步都解耦了。因此训练可以并行化，即在各时刻  $t$  分别计算梯度。因为训练集提供输出的理想值，所以没有必要先计算前一时刻的输出。

由输出反馈到模型而产生循环连接的模型可用 **导师驱动过程** (teacher forcing) 进行训练。训练模型时，导师驱动过程不再使用最大似然准则，而在时刻  $t + 1$  接收真实值  $y^{(t)}$  作为输入。我们可以通过检查两个时间步的序列得知这一点。条件最大似然准则是

$$\log p(\mathbf{y}^{(1)}, \mathbf{y}^{(2)} | \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) \quad (10.15)$$

$$= \log p(\mathbf{y}^{(2)} | \mathbf{y}^{(1)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}) + \log p(\mathbf{y}^{(1)} | \mathbf{x}^{(1)}, \mathbf{x}^{(2)}). \quad (10.16)$$

在这个例子中，同时给定迄今为止的  $\mathbf{x}$  序列和来自训练集的前一  $\mathbf{y}$  值，我们可以看到在时刻  $t = 2$  时，模型被训练为最大化  $\mathbf{y}^{(2)}$  的条件概率。因此最大似然在训练时指定正确反馈，而不是将自己的输出反馈到模型。如图 10.6 所示。

我们使用导师驱动过程的最初动机是为了在缺乏隐藏到隐藏连接的模型中避免通过时间反向传播。只要模型一个时间步的输出与下一时间步计算的值存在连接，导师驱动过程仍然可以应用到这些存在隐藏到隐藏连接的模型。然而，只要隐藏单元成为较早时间步的函数，BPTT 算法是必要的。因此训练某些模型时要同时使用导师驱动过程和 BPTT。

如果之后网络在开环 (open-loop) 模式下使用，即网络输出（或输出分布的样本）反馈作为输入，那么完全使用导师驱动过程进行训练的缺点就会出现。在这种情况下，训练期间该网络看到的输入与测试时看到的会有很大的不同。减轻此问题的一种方法是同时使用导师驱动过程和自由运行的输入进行训练，例如在展开循环的输出到输入路径上预测几个步骤的正确目标值。通过这种方式，网络可以学会考虑在训练时没有接触到的输入条件（如自由运行模式下，自身生成自身），以及将状态映射回使网络几步之后生成正确输出的状态。另外一种方式 (Bengio *et al.*, 2015b) 是通过随意选择生成值或真实的数据值作为输入以减小训练时和测试时看到的输入之间的差别。这种方法利用了课程学习策略，逐步使用更多生成值作为输入。

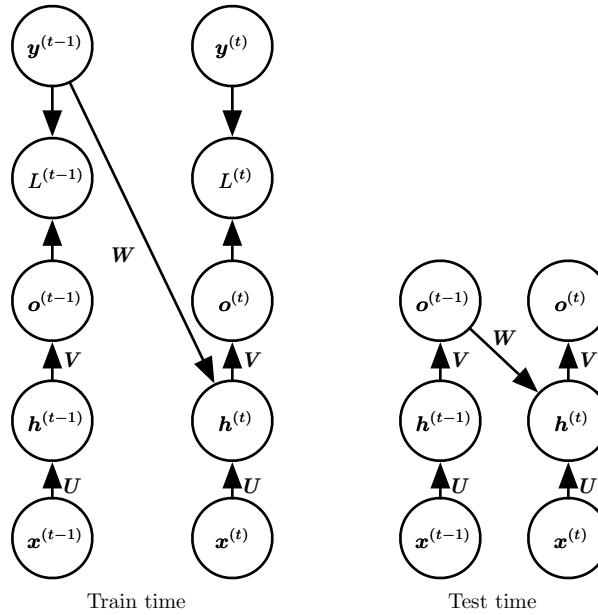


图 10.6: 导师驱动过程的示意图。导师驱动过程是一种训练技术，适用于输出与下一时间步的隐藏状态存在连接的 RNN。(左) 训练时，我们将训练集中正确的输出  $y^{(t)}$  反馈到  $h^{(t+1)}$ 。(右) 当模型部署后，真正的输出通常是未知的。在这种情况下，我们用模型的输出  $o^{(t)}$  近似正确的输出  $y^{(t)}$ ，并反馈回模型。

## 10.2.2 计算循环神经网络的梯度

计算循环神经网络的梯度是容易的。我们可以简单地将第 6.5.6 节中的推广反向传播算法应用于展开的计算图，而不需要特殊化的算法。由反向传播计算得到的梯度，并结合任何通用的基于梯度的技术就可以训练 RNN。

为了获得 BPTT 算法行为的一些直观理解，我们举例说明如何通过 BPTT 计算上述 RNN 公式（式 (10.8) 和式 (10.12)）的梯度。计算图的节点包括参数  $U, V, W, b$  和  $c$ ，以及以  $t$  为索引的节点序列  $x^{(t)}, h^{(t)}, o^{(t)}$  和  $L^{(t)}$ 。对于每一个节点  $\mathbf{N}$ ，我们需要基于  $\mathbf{N}$  后面的节点的梯度，递归地计算梯度  $\nabla_{\mathbf{N}} L$ 。我们从紧接着最终损失的节点开始递归：

$$\frac{\partial L}{\partial L^{(t)}} = 1. \quad (10.17)$$

在这个导数中，我们假设输出  $o^{(t)}$  作为 softmax 函数的参数，我们可以从 softmax

函数可以获得关于输出概率的向量  $\hat{\mathbf{y}}$ 。我们也假设损失是迄今为止给定了输入后的真实目标  $y^{(t)}$  的负对数似然。对于所有  $i, t$ , 关于时间步  $t$  输出的梯度  $\nabla_{o^{(t)}} L$  如下：

$$(\nabla_{o^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i, y^{(t)}}. \quad (10.18)$$

我们从序列的末尾开始, 反向进行计算。在最后的时间步  $\tau$ ,  $\mathbf{h}^{(\tau)}$  只有  $\mathbf{o}^{(\tau)}$  作为后续节点, 因此这个梯度很简单:

$$\nabla_{\mathbf{h}^{(\tau)}} L = \mathbf{V}^\top \nabla_{\mathbf{o}^{(\tau)}} L. \quad (10.19)$$

然后, 我们可以从时刻  $t = \tau - 1$  到  $t = 1$  反向迭代, 通过时间反向传播梯度, 注意  $\mathbf{h}^{(t)} (t < \tau)$  同时具有  $\mathbf{o}^{(t)}$  和  $\mathbf{h}^{(t+1)}$  两个后续节点。因此, 它的梯度由下式计算

$$\nabla_{\mathbf{h}^{(t)}} L = \left( \frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \right)^\top (\nabla_{\mathbf{h}^{(t+1)}} L) + \left( \frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{h}^{(t)}} \right)^\top (\nabla_{\mathbf{o}^{(t)}} L) \quad (10.20)$$

$$= \mathbf{W}^\top (\nabla_{\mathbf{h}^{(t+1)}} L) \text{diag}\left(1 - (\mathbf{h}^{(t+1)})^2\right) + \mathbf{V}^\top (\nabla_{\mathbf{o}^{(t)}} L), \quad (10.21)$$

其中  $\text{diag}\left(1 - (\mathbf{h}^{(t+1)})^2\right)$  表示包含元素  $1 - (h_i^{(t+1)})^2$  的对角矩阵。这是关于时刻  $t+1$  与隐藏单元  $i$  关联的双曲正切的Jacobian。

一旦获得了计算图内部节点的梯度, 我们就可以得到关于参数节点的梯度。因为参数在许多时间步共享, 我们必须在表示这些变量的微积分操作时谨慎对待。我们希望实现的等式使用第 6.5.6 节中的 `bprop` 方法计算计算图中单一边对梯度的贡献。然而微积分中的  $\nabla_{\mathbf{w}f}$  算子, 计算  $\mathbf{W}$  对于  $f$  的贡献时将计算图中的所有边都考虑进去了。为了消除这种歧义, 我们定义只在  $t$  时刻使用的虚拟变量  $\mathbf{W}^{(t)}$  作为  $\mathbf{W}$  的副本。然后, 我们可以使用  $\nabla_{\mathbf{W}^{(t)}}$  表示权重在时间步  $t$  对梯度的贡献。

使用这个表示，关于剩下参数的梯度可以由下式给出：

$$\nabla_c L = \sum_t \left( \frac{\partial \mathbf{o}^{(t)}}{\partial \mathbf{c}} \right)^\top \nabla_{\mathbf{o}^{(t)}} L = \sum_t \nabla_{\mathbf{o}^{(t)}} L, \quad (10.22)$$

$$\nabla_b L = \sum_t \left( \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{b}^{(t)}} \right)^\top \nabla_{\mathbf{h}^{(t)}} L = \sum_t \text{diag}\left(1 - (\mathbf{h}^{(t)})^2\right) \nabla_{\mathbf{h}^{(t)}} L, \quad (10.23)$$

$$\nabla_v L = \sum_t \sum_i \left( \frac{\partial L}{\partial o_i^{(t)}} \right) \nabla_{v o_i^{(t)}} = \sum_t (\nabla_{\mathbf{o}^{(t)}} L) \mathbf{h}^{(t)\top}, \quad (10.24)$$

$$\nabla_w L = \sum_t \sum_i \left( \frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{w^{(t)}} h_i^{(t)} \quad (10.25)$$

$$= \sum_t \text{diag}\left(1 - (\mathbf{h}^{(t)})^2\right) (\nabla_{\mathbf{h}^{(t)}} L) \mathbf{h}^{(t-1)\top}, \quad (10.26)$$

$$\nabla_u L = \sum_t \sum_i \left( \frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{u^{(t)}} h_i^{(t)} \quad (10.27)$$

$$= \sum_t \text{diag}\left(1 - (\mathbf{h}^{(t)})^2\right) (\nabla_{\mathbf{h}^{(t)}} L) \mathbf{x}^{(t)\top}, \quad (10.28)$$

因为计算图中定义的损失的任何参数都不是训练数据  $\mathbf{x}^{(t)}$  的父节点，所以我们不需要计算关于它的梯度。

### 10.2.3 作为有向图模型的循环网络

目前为止，我们接触的循环网络例子中损失  $L^{(t)}$  是训练目标  $\mathbf{y}^{(t)}$  和输出  $\mathbf{o}^{(t)}$  之间的交叉熵。与前馈网络类似，原则上循环网络几乎可以使用任何损失。但必须根据任务来选择损失。如前馈网络，我们通常希望将 RNN 的输出解释为一个概率分布，并且我们通常使用与分布相关联的交叉熵来定义损失。均方误差是与单位高斯分布的输出相关联的交叉熵损失，例如前馈网络中所使用的。

当我们使用一个预测性对数似然的训练目标，如式 (10.12)，我们将 RNN 训练为能够根据之前的输入估计下一个序列元素  $\mathbf{y}^{(t)}$  的条件分布。这可能意味着，我们最大化对数似然

$$\log p(\mathbf{y}^{(t)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}), \quad (10.29)$$

或者，如果模型包括来自一个时间步的输出到下一个时间步的连接，

$$\log p(\mathbf{y}^{(t)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}, \mathbf{y}^{(1)}, \dots, \mathbf{y}^{(t-1)}). \quad (10.30)$$

将整个序列  $\mathbf{y}$  的联合分布分解为一系列单步的概率预测是捕获关于整个序列完整联合分布的一种方法。当我们不把过去的  $\mathbf{y}$  值反馈给下一步作为预测的条件时，那么有向图模型不包含任何从过去  $\mathbf{y}^{(i)}$  到当前  $\mathbf{y}^{(t)}$  的边。在这种情况下，输出  $\mathbf{y}$  与给定的  $\mathbf{x}$  序列是条件独立的。当我们反馈真实的  $\mathbf{y}$  值（不是它们的预测值，而是真正观测到或生成的值）给网络时，那么有向图模型包含所有从过去  $\mathbf{y}^{(i)}$  到当前  $\mathbf{y}^{(t)}$  的边。

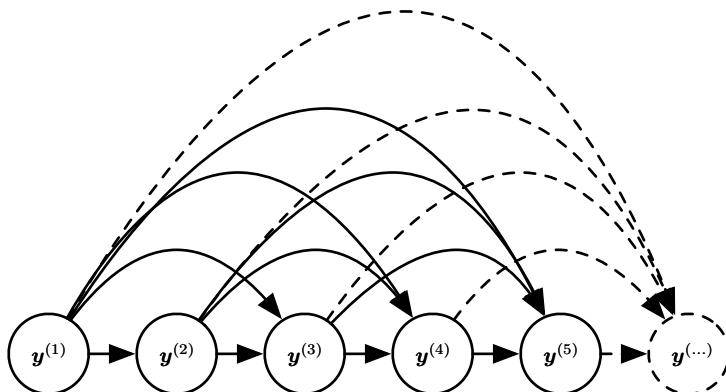


图 10.7: 序列  $y^{(1)}, y^{(2)}, \dots, y^{(t)}, \dots$  的全连接图模型。给定先前的值，每个过去的观察值  $y^{(i)}$  可以影响一些  $y^{(t)} (t > i)$  的条件分布。当序列中每个元素的输入和参数的数目越来越多，根据此图直接参数化图模型（如式 (10.6) 中）可能非常低效的。RNN 可以通过高效的参数化获得相同的全连接，如图 10.8 所示。

举一个简单的例子，让我们考虑对标量随机变量序列  $\mathbb{Y} = \{y^{(1)}, \dots, y^{(\tau)}\}$  建模的 RNN，也没有额外的输入  $\mathbf{x}$ 。在时间步  $t$  的输入仅仅是时间步  $t - 1$  的输出。该 RNN 定义了关于  $\mathbf{y}$  变量的有向图模型。我们使用链式法则（用于条件概率的式 (3.6)）参数化这些观察值的联合分布：

$$P(\mathbb{Y}) = P(y^{(1)}, \dots, y^{(\tau)}) = \prod_{t=1}^{\tau} P(y^{(t)} | y^{(t-1)}, y^{(t-2)}, \dots, y^{(1)}), \quad (10.31)$$

其中当  $t = 1$  时竖杠右侧显然为空。因此，根据这样一个模型，一组值  $\{y^{(1)}, \dots, y^{(\tau)}\}$  的负对数似然为

$$L = \sum_t L^{(t)}, \quad (10.32)$$

其中

$$L^{(t)} = -\log P(y^{(t)} = y^{(t)} \mid y^{(t-1)}, y^{(t-2)}, \dots, y^{(1)}). \quad (10.33)$$

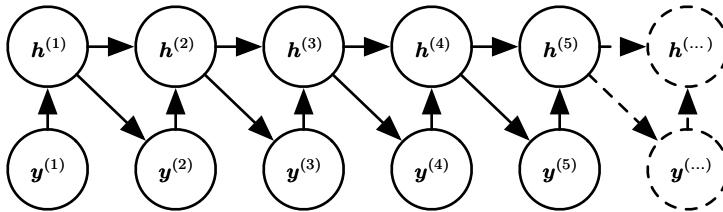


图 10.8: 在 RNN 图模型中引入状态变量, 尽管它是输入的确定性函数, 但它有助于我们根据式 (10.5) 获得非常高效的参数化。序列中的每个阶段 (对于  $h^{(t)}$  和  $y^{(t)}$ ) 使用相同的结构 (每个节点具有相同数量的输入), 并且可以与其他阶段共享相同的参数。

图模型中的边表示哪些变量直接依赖于其他变量。许多图模型的目标是省略不存在强相互作用的边以实现统计和计算的效率。例如, 我们通常可以作Markov假设, 即图模型应该只包含从  $\{y^{(t-k)}, \dots, y^{(t-1)}\}$  到  $y^{(t)}$  的边, 而不是包含整个过去历史的边。然而, 在一些情况下, 我们认为整个过去的输入会对序列的下一个元素有一定影响。当我们认为  $y^{(t)}$  的分布可能取决于遥远过去 (在某种程度) 的  $y^{(i)}$  的值, 且无法通过  $y^{(t-1)}$  捕获  $y^{(i)}$  的影响时, RNN 将会很有用。

解释 RNN 作为图模型的一种方法是将RNN视为定义一个结构为完全图的图模型, 且能够表示任何一对  $y$  值之间的直接联系。图 10.7 是关于  $y$  值且具有完全图结构的图模型。该 RNN 完全图的解释基于排除并忽略模型中的隐藏单元  $h^{(t)}$ 。

更有趣的是, 将隐藏单元  $h^{(t)}$  视为随机变量, 从而产生 RNN 的图模型结构<sup>1</sup>。在图模型中包括隐藏单元预示 RNN 能对观测的联合分布提供非常有效的参数化。假设我们用表格表示法来表示离散值上任意的联合分布, 即对每个值可能的赋值分配一个单独条目的数组, 该条目表示发生该赋值的概率。如果  $y$  可以取  $k$  个不同的值, 表格表示法将有  $\mathcal{O}(k^r)$  个参数。对比 RNN, 由于参数共享, RNN 的参数数目为  $\mathcal{O}(1)$  且是序列长度的函数。我们可以调节 RNN 的参数数量来控制模型容量, 但不用被迫与序列长度成比例。式 (10.5) 展示了所述 RNN 通过循环应用相同的函数  $f$  以及在每个时间步的相同参数  $\theta$ , 有效地参数化的变量之间的长期联系。图 10.8 说

<sup>1</sup>给定这些变量的父变量, 其条件分布是确定性的。尽管设计具有这样确定性的隐藏单元的图模型是很少见的, 但这是完全合理的。

明了这个图模型的解释。在图模型中结合  $h^{(t)}$  节点可以用作过去和未来之间的中间量，从而将它们解耦。遥远过去的变量  $y^{(i)}$  可以通过其对  $h$  的影响来影响变量  $y^{(t)}$ 。该图的结构表明可以在时间步使用相同的条件概率分布有效地参数化模型，并且当观察到全部变量时，可以高效地评估联合分配给所有变量的概率。

即便使用高效参数化的图模型，某些操作在计算上仍然具有挑战性。例如，难以预测序列中缺少的值。

循环网络为减少的参数数目付出的代价是优化参数可能变得困难。

在循环网络中使用的参数共享的前提是相同参数可用于不同时间步的假设。也就是说，假设给定时刻  $t$  的变量后，时刻  $t + 1$  变量的条件概率分布是 **平稳的** (stationary)，这意味着之前的时间步与下个时间步之间的关系并不依赖于  $t$ 。原则上，可以使用  $t$  作为每个时间步的额外输入，并让学习器在发现任何时间依赖性的同时，在不同时间步之间尽可能多地共享。相比在每个  $t$  使用不同的条件概率分布已经好很多了，但网络将必须在面对新  $t$  时进行推断。

为了完整描述将 RNN 作为图模型的观点，我们必须描述如何从模型采样。我们需要执行的主要操作是简单地从每一时间步的条件分布采样。然而，这会导致额外的复杂性。RNN 必须有某种机制来确定序列的长度。这可以通过多种方式实现。

在当输出是从词汇表获取的符号的情况下，我们可以添加一个对应于序列末端的特殊符号 (Schmidhuber, 2012)。当产生该符号时，采样过程停止。在训练集中，我们将该符号作为序列的一个额外成员，即紧跟每个训练样本  $x^{(\tau)}$  之后。

另一种选择是在模型中引入一个额外的 Bernoulli 输出，表示在每个时间步决定继续生成或停止生成。相比向词汇表增加一个额外符号，这种方法更普遍，因为它适用于任何 RNN，而不仅仅是输出符号序列的 RNN。例如，它可以应用于一个产生实数序列的 RNN。新的输出单元通常使用 sigmoid 单元，并通过交叉熵训练。在这种方法中，sigmoid 被训练为最大化正确预测的对数似然，即在每个时间步序列决定结束或继续。

确定序列长度  $\tau$  的另一种方法是将一个额外的输出添加到模型并预测整数  $\tau$  本身。模型可以采出  $\tau$  的值，然后采  $\tau$  步有价值的数据。这种方法需要在每个时间步的循环更新中增加一个额外输入，使得循环更新知道它是否是靠近所产生序列的末尾。这种额外的输入可以是  $\tau$  的值，也可以是  $\tau - t$  即剩下时间步的数量。如果没有这个额外的输入，RNN 可能会产生突然结束序列，如一个句子在最终完整前结

束。此方法基于分解

$$P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)}) = P(\tau)P(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)} | \tau). \quad (10.34)$$

直接预测  $\tau$  的例子见 Goodfellow et al. (2014d)。

#### 10.2.4 基于上下文的 RNN 序列建模

上一节描述了没有输入  $\mathbf{x}$  时，关于随机变量序列  $y^{(t)}$  的 RNN 如何对应于有向图模型。当然，如式 (10.8) 所示的 RNN 包含一个输入序列  $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(\tau)}$ 。一般情况下，RNN 允许将图模型的观点扩展到不仅代表  $y$  变量的联合分布也能表示给定  $\mathbf{x}$  后  $y$  条件分布。如在第 6.2.1.1 节的前馈网络情形中所讨论的，任何代表变量  $P(\mathbf{y}; \boldsymbol{\theta})$  的模型都能被解释为代表条件分布  $P(\mathbf{y} | \boldsymbol{\omega})$  的模型，其中  $\boldsymbol{\omega} = \boldsymbol{\theta}$ 。我们能像之前一样使用  $P(\mathbf{y} | \boldsymbol{\omega})$  代表分布  $P(\mathbf{y} | \mathbf{x})$  来扩展这样的模型，但要令  $\boldsymbol{\omega}$  是关于  $\mathbf{x}$  的函数。在 RNN 的情况，这可以通过不同的方式来实现。此处，我们回顾最常见和最明显的选择。

之前，我们已经讨论了将  $t = 1, \dots, \tau$  的向量  $\mathbf{x}^{(t)}$  序列作为输入的 RNN。另一种选择是只使用单个向量  $\mathbf{x}$  作为输入。当  $\mathbf{x}$  是一个固定大小的向量时，我们可以简单地将其看作产生  $\mathbf{y}$  序列 RNN 的额外输入。将额外输入提供到 RNN 的一些常见方法是：

1. 在每个时刻作为一个额外输入，或
2. 作为初始状态  $\mathbf{h}^{(0)}$ ，或
3. 结合两种方式。

第一个也是最常用的方法如图 10.9 所示。输入  $\mathbf{x}$  和每个隐藏单元向量  $\mathbf{h}^{(t)}$  之间的相互作用是通过新引入的权重矩阵  $\mathbf{R}$  参数化的，这是只包含  $y$  序列的模型所没有的。同样的乘积  $\mathbf{x}^\top \mathbf{R}$  在每个时间步作为隐藏单元的一个额外输入。我们可以认为  $\mathbf{x}$  的选择（确定  $\mathbf{x}^\top \mathbf{R}$  值），是有效地用于每个隐藏单元的一个新偏置参数。权重与输入保持独立。我们可以认为这种模型采用了非条件模型的  $\boldsymbol{\theta}$ ，并将  $\boldsymbol{\omega}$  代入  $\boldsymbol{\theta}$ ，其中  $\boldsymbol{\omega}$  内的偏置参数现在是输入的函数。

RNN 可以接收向量序列  $\mathbf{x}^{(t)}$  作为输入，而不是仅接收单个向量  $\mathbf{x}$  作为输入。式 (10.8) 描述的 RNN 对应条件分布  $P(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau)})$ ，并在条件独立

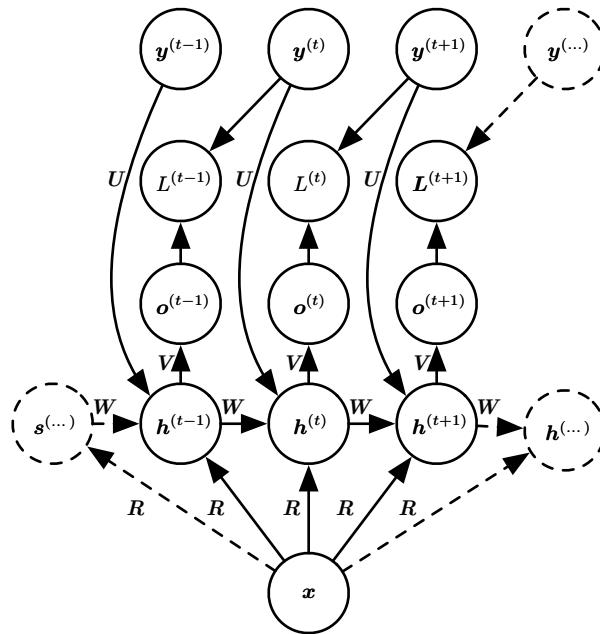


图 10.9: 将固定长度的向量  $x$  映射到序列  $\mathbf{Y}$  上分布的 RNN。这类 RNN 适用于很多任务如图注，其中单个图像作为模型的输入，然后产生描述图像的词序列。观察到的输出序列的每个元素  $y^{(t)}$  同时用作输入（对于当前时间步）和训练期间的目标（对于前一时间步）。

的假设下这个分布分解为

$$\prod_t P(\mathbf{y}^{(t)} \mid \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(t)}). \quad (10.35)$$

为去掉条件独立的假设，我们可以在时刻  $t$  的输出到时刻  $t + 1$  的隐藏单元添加连接，如图 10.10 所示。该模型就可以代表关于  $\mathbf{y}$  序列的任意概率分布。这种给定一个序列表示另一个序列分布的模型的还是有一个限制，就是这两个序列的长度必须是相同的。我们将在第 10.4 节描述如何消除这种限制。

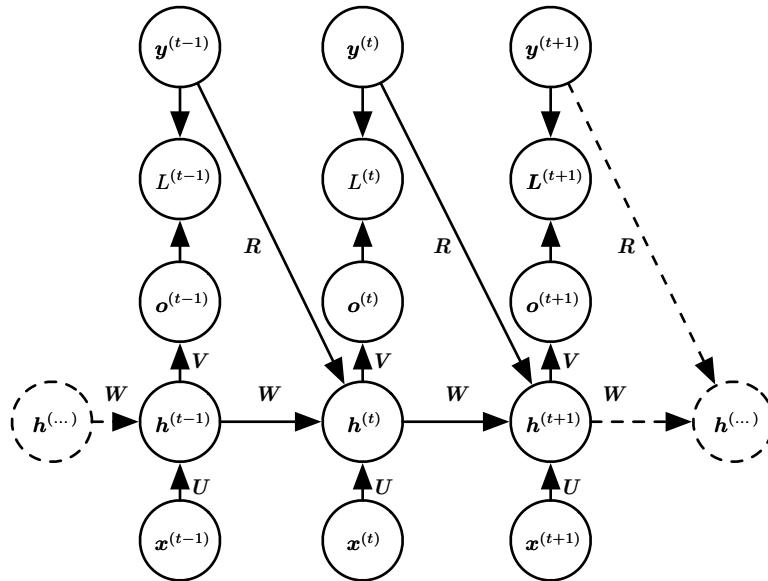


图 10.10: 将可变长度的  $x$  值序列映射到相同长度的  $y$  值序列上分布的条件循环神经网络。对比图 10.3，此 RNN 包含从前一个输出到当前状态的连接。这些连接允许此 RNN 对给定  $x$  的序列后相同长度的  $y$  序列上的任意分布建模。图 10.3 的 RNN 仅能表示在给定  $x$  值的情况下， $y$  值彼此条件独立的分布。

### 10.3 双向 RNN

目前为止我们考虑的所有循环神经网络有一个“因果”结构，意味着在时刻  $t$  的状态只能从过去的序列  $x^{(1)}, \dots, x^{(t-1)}$  以及当前的输入  $x^{(t)}$  捕获信息。我们还讨论了某些在  $y$  可用时，允许过去的  $y$  值信息影响当前状态的模型。

然而，在许多应用中，我们要输出的  $y^{(t)}$  的预测可能依赖于整个输入序列。例如，在语音识别中，由于协同发音，当前声音作为音素的正确解释可能取决于未来几个音素，甚至潜在的可能取决于未来的几个词，因为词与附近的词之间的存在语义依赖：如果当前的词有两种声学上合理的解释，我们可能要在更远的未来（和过去）寻找信息区分它们。这在手写识别和许多其他序列到序列学习的任务中也是如此，将会在下一节中描述。

双向循环神经网络（或双向 RNN）为满足这种需要而被发明 (Schuster and Paliwal, 1997)。他们在需要双向信息的应用中非常成功 (Graves, 2012)，如手写

识别 (Graves *et al.*, 2008; Graves and Schmidhuber, 2009), 语音识别 (Graves and Schmidhuber, 2005; Graves *et al.*, 2013) 以及生物信息学 (Baldi *et al.*, 1999)。

顾名思义, 双向 RNN 结合时间上从序列起点开始移动的 RNN 和另一个时间上从序列末尾开始移动的 RNN。图 10.11 展示了典型的双向 RNN, 其中  $h^{(t)}$  代表通过时间向前移动的子 RNN 的状态,  $g^{(t)}$  代表通过时间向后移动的子 RNN 的状态。这允许输出单元  $o^{(t)}$  能够计算同时依赖于过去和未来且对时刻  $t$  的输入值最敏感的表示, 而不必指定  $t$  周围固定大小的窗口 (这是前馈网络、卷积网络或具有固定大小的先行缓存器的常规 RNN 所必须要做的)。

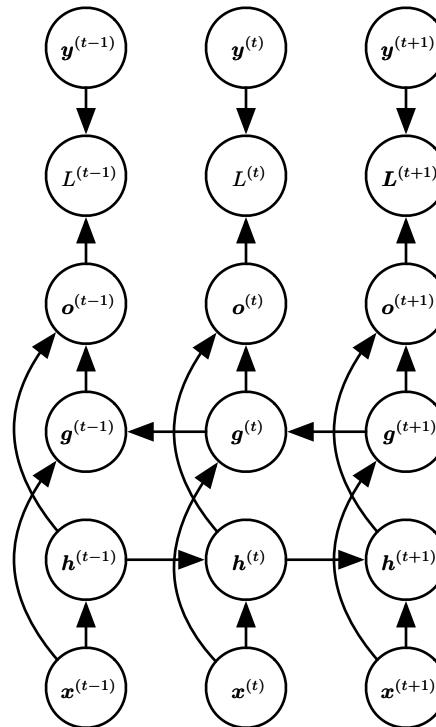


图 10.11: 典型的双向循环神经网络中的计算, 意图学习将输入序列  $\mathbf{x}$  映射到目标序列  $\mathbf{y}$  (在每个步骤  $t$  具有损失  $L^{(t)}$ )。循环性  $h$  在时间上向前传播信息 (向右), 而循环性  $g$  在时间上向后传播信息 (向左)。因此在每个点  $t$ , 输出单元  $o^{(t)}$  可以受益于输入  $h^{(t)}$  中关于过去的相关概要以及输入  $g^{(t)}$  中关于未来的相关概要。

这个想法可以自然地扩展到 2 维输入, 如图像, 由四个 RNN 组成, 每一个沿着四个方向中的一个计算: 上、下、左、右。如果 RNN 能够学习到承载长期信息,

那在 2 维网格每个点  $(i, j)$  的输出  $O_{i,j}$  就能计算一个能捕捉到大多局部信息但仍依赖于长期输入的表示。相比卷积网络，应用于图像的 RNN 计算成本通常更高，但允许同一特征图的特征之间存在长期横向的相互作用 (Visin *et al.*, 2015; Kalchbrenner *et al.*, 2015)。实际上，对于这样的 RNN，前向传播公式可以写成表示使用卷积的形式，计算自底向上到每一层的输入（在整合横向相互作用的特征图的循环传播之前）。

## 10.4 基于编码-解码的序列到序列架构

我们已经在图 10.5 看到 RNN 如何将输入序列映射成固定大小的向量，在图 10.9 中看到 RNN 如何将固定大小的向量映射成一个序列，在图 10.3、图 10.4、图 10.10 和图 10.11 中看到 RNN 如何将一个输入序列映射到等长的输出序列。

本节我们讨论如何训练 RNN，使其将输入序列映射到不一定等长的输出序列。这在许多场景中都有应用，如语音识别、机器翻译或问答，其中训练集的输入和输出序列的长度通常不相同（虽然它们的长度可能相关）。

我们经常将 RNN 的输入称为“上下文”。我们希望产生此上下文的表示， $C$ 。这个上下文  $C$  可能是一个概括输入序列  $\mathbf{X} = (\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n_x)})$  的向量或者向量序列。

用于映射可变长度序列到另一可变长度序列最简单的 RNN 架构最初由 Cho *et al.* (2014a) 提出，之后不久由 Sutskever *et al.* (2014) 独立开发，并且第一个使用这种方法获得翻译的最好结果。前一系统是对另一个机器翻译系统产生的建议进行评分，而后者使用独立的循环网络生成翻译。这些作者分别将该架构称为编码-解码或序列到序列架构，如图 10.12 所示。这个想法非常简单：(1) 编码器 (encoder) 或读取器 (reader) 或输入 (input) RNN 处理输入序列。编码器输出上下文  $C$  (通常是最终隐藏状态的简单函数)。(2) 解码器 (decoder) 或写入器 (writer) 或输出 (output) RNN 则以固定长度的向量 (如图 10.9) 为条件产生输出序列  $\mathbf{Y} = (\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n_y)})$ 。这种架构对比本章前几节提出的架构的创新之处在于长度  $n_x$  和  $n_y$  可以彼此不同，而之前的架构约束  $n_x = n_y = \tau$ 。在序列到序列的架构中，两个 RNN 共同训练以最大化  $\log P(\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n_y)} | \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n_x)})$  (关于训练集中所有  $\mathbf{x}$  和  $\mathbf{y}$  对的平均)。编码器 RNN 的最后一个状态  $\mathbf{h}_{n_x}$  通常被当作输入的表示  $C$  并作为解码器 RNN 的输入。

如果上下文  $C$  是一个向量，则解码器 RNN 只是在第 10.2.4 节描述的向量到序

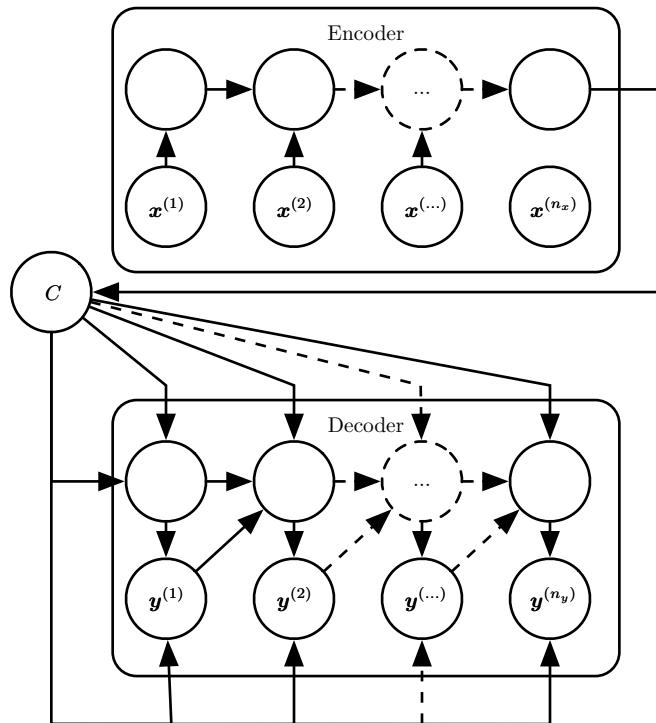


图 10.12: 在给定输入序列  $(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n_x)})$  的情况下学习生成输出序列  $(\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \dots, \mathbf{y}^{(n_y)})$  的编码器-解码器或序列到序列的 RNN 架构的示例。它由读取输入序列的编码器 RNN 以及生成输出序列（或计算给定输出序列的概率）的解码器 RNN 组成。编码器 RNN 的最终隐藏状态用于计算一般为固定大小的上下文变量  $C$ ， $C$  表示输入序列的语义概要并且作为解码器 RNN 的输入。

列 RNN。正如我们所见，向量到序列 RNN 至少有两种接受输入的方法。输入可以被提供为 RNN 的初始状态，或连接到每个时间步中的隐藏单元。这两种方式也可以结合。

这里并不强制要求编码器与解码器的隐藏层具有相同的大小。

此架构的一个明显不足是，编码器 RNN 输出的上下文  $C$  的维度太小而难以适当地概括一个长序列。这种现象由 Bahdanau *et al.* (2015) 在机器翻译中观察到。他们提出让  $C$  成为可变长度的序列，而不是一个固定大小的向量。此外，他们还引入了将序列  $C$  的元素和输出序列的元素相关联的 **注意力机制** (attention mechanism)。读者可在第 12.4.5.1 节了解更多细节。

## 10.5 深度循环网络

大多数 RNN 中的计算可以分解成三块参数及其相关的变换：

1. 从输入到隐藏状态，
2. 从前一隐藏状态到下一隐藏状态，以及
3. 从隐藏状态到输出。

根据图 10.3 中的 RNN 架构，这三个块都与单个权重矩阵相关联。换句话说，当网络被展开时，每个块对应一个浅的变换。能通过深度 MLP 内单个层来表示的变换称为浅变换。通常，这是由学成的仿射变换和一个固定非线性表示组成的变换。

在这些操作中引入深度会有利的吗？实验证据 (Graves *et al.*, 2013; Pascanu *et al.*, 2014a) 强烈暗示理应如此。实验证据与我们需要足够的深度以执行所需映射的想法一致。读者可以参考 Schmidhuber (1992); El Hihi and Bengio (1996) 或 Jaeger (2007a) 了解更早的关于深度 RNN 的研究。

Graves *et al.* (2013) 第一个展示了将 RNN 的状态分为多层的显著好处，如图 10.13 (左)。我们可以认为，在图 10.13 (a) 所示层次结构中较低的层起到了将原始输入转化为对更高层的隐藏状态更合适表示的作用。Pascanu *et al.* (2014a) 更进一步提出在上述三个块中各使用一个单独的 MLP (可能是深度的)，如图 10.13 (b) 所示。考虑表示容量，我们建议在这三个步中都分配足够的容量，但增加深度可能会因为优化困难而损害学习效果。在一般情况下，更容易优化较浅的架构，加入图 10.13 (b) 的额外深度导致从时间步  $t$  的变量到时间步  $t+1$  的最短路径变得更长。例如，如果有单个隐藏层的 MLP 被用于状态到状态的转换，那么与图 10.3 相比，我们就会加倍任何两个不同时间步变量之间最短路径的长度。然而 Pascanu *et al.* (2014a) 认为，在隐藏到隐藏的路径中引入跳跃连接可以缓和这个问题，如图 10.13 (c) 所示。

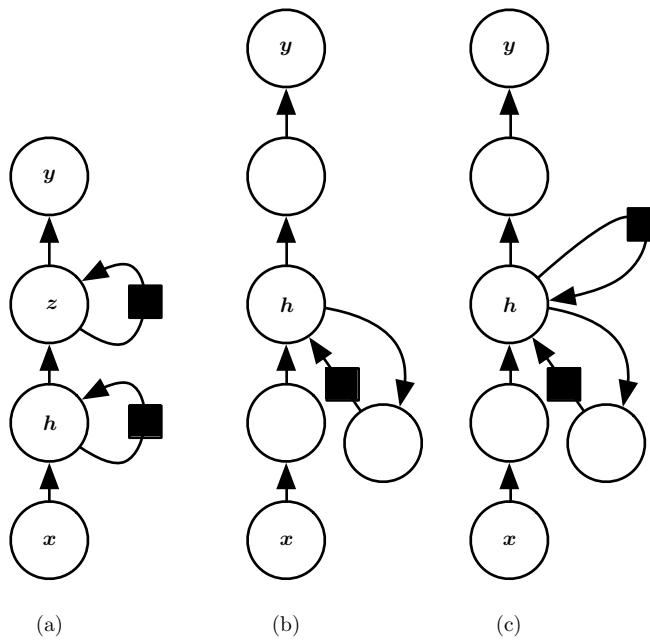


图 10.13: 循环神经网络可以通过许多方式变得更深 (Pascanu *et al.*, 2014a)。(a) 隐藏循环状态可以被分解为具有层次的组。(b) 可以向输入到隐藏, 隐藏到隐藏以及隐藏到输出的部分引入更深的计算 (如 MLP)。这可以延长链接不同时间步的最短路径。(c) 可以引入跳跃连接来缓解路径延长的效果。

## 10.6 递归神经网络

递归神经网络<sup>2</sup>代表循环网络的另一个扩展, 它被构造为深的树状结构而不是 RNN 的链状结构, 因此是不同类型的计算图。递归网络的典型计算图如图 10.14 所示。递归神经网络由 Pollack (1990) 引入, 而 Bottou (2011) 描述了这类网络的潜在用途——学习推论。递归网络已成功地应用于输入是数据结构的神经网络 (Frasconi *et al.*, 1997, 1998), 如自然语言处理 (Socher *et al.*, 2011a,c, 2013a) 和计算机视觉 (Socher *et al.*, 2011b)。

递归网络的一个明显优势是, 对于具有相同长度  $\tau$  的序列, 深度 (通过非线性操作的组合数量来衡量) 可以急剧地从  $\tau$  减小为  $\mathcal{O}(\log \tau)$ , 这可能有助于解决长期

<sup>2</sup>我们建议不要将“递归神经网络”缩写为“RNN”, 以免与“循环神经网络”混淆。

依赖。一个悬而未决的问题是如何以最佳的方式构造树。一种选择是使用不依赖于数据的树结构，如平衡二叉树。在某些应用领域，外部方法可以为选择适当的树结构提供借鉴。例如，处理自然语言的句子时，用于递归网络的树结构可以被固定为句子语法分析树的结构（可以由自然语言语法分析程序提供）(Socher *et al.*, 2011a,c)。理想的情况下，人们希望学习器自行发现和推断适合于任意给定输入的树结构，如(Bottou, 2011) 所建议。

