



动手学深度学习

**MXNet Community**

2018年06月06日



---

# 目录

---

<b>1 前言</b>	<b>3</b>
1.1 为什么要做这个项目 . . . . .	3
1.2 前言 . . . . .	6
1.3 主要符号一览 . . . . .	8
<b>2 预备知识</b>	<b>11</b>
2.1 机器学习简介 . . . . .	11
2.2 安装和运行 . . . . .	32
2.3 数据操作 . . . . .	38
2.4 自动求梯度 . . . . .	46
<b>3 深度学习基础</b>	<b>51</b>
3.1 单层神经网络 . . . . .	51
3.2 线性回归——从零开始 . . . . .	57
3.3 线性回归——使用 Gluon . . . . .	62
3.4 分类模型 . . . . .	66
3.5 Softmax 回归——从零开始 . . . . .	70
3.6 Softmax 回归——使用 Gluon . . . . .	77
3.7 多层神经网络 . . . . .	79
3.8 多层感知机——从零开始 . . . . .	87

3.9	多层感知机——使用 Gluon . . . . .	90
3.10	欠拟合、过拟合和模型选择 . . . . .	91
3.11	正则化——从零开始 . . . . .	99
3.12	正则化——使用 Gluon . . . . .	105
3.13	丢弃法——从零开始 . . . . .	108
3.14	丢弃法——使用 Gluon . . . . .	113
3.15	正向传播和反向传播 . . . . .	115
3.16	实战 Kaggle 比赛：预测房价 . . . . .	119
<b>4</b>	<b>深度学习计算</b>	<b>133</b>
4.1	模型构造 . . . . .	133
4.2	模型参数的访问、初始化和共享 . . . . .	138
4.3	模型参数的延后初始化 . . . . .	143
4.4	自定义层 . . . . .	146
4.5	读取和存储 . . . . .	149
4.6	GPU 计算 . . . . .	152
<b>5</b>	<b>卷积神经网络</b>	<b>157</b>
5.1	二维卷积层 . . . . .	157
5.2	填充和步幅 . . . . .	162
5.3	多输入和输出通道 . . . . .	165
5.4	池化层 . . . . .	169
5.5	卷积神经网络 . . . . .	173
5.6	深度卷积神经网络：AlexNet . . . . .	177
5.7	使用重复元素的网络：VGG . . . . .	182
5.8	网络中的网络：NiN . . . . .	185
5.9	含并行连结的网络：GoogLeNet . . . . .	189
5.10	批量归一化——从零开始 . . . . .	194
5.11	批量归一化——使用 Gluon . . . . .	199
5.12	残差网络：ResNet . . . . .	200
5.13	稠密连接网络：DenseNet . . . . .	205
<b>6</b>	<b>循环神经网络</b>	<b>211</b>
6.1	语言模型 . . . . .	212
6.2	隐藏状态 . . . . .	214

6.3	循环神经网络——从零开始 . . . . .	216
6.4	通过时间反向传播 . . . . .	231
6.5	门控循环单元 (GRU)——从零开始 . . . . .	235
6.6	长短期记忆 (LSTM)——从零开始 . . . . .	241
6.7	深度循环神经网络 . . . . .	247
6.8	双向循环神经网络 . . . . .	249
6.9	循环神经网络——使用 Gluon . . . . .	251
<b>7</b>	<b>优化算法</b>	<b>259</b>
7.1	优化算法概述 . . . . .	260
7.2	梯度下降和随机梯度下降——从零开始 . . . . .	265
7.3	梯度下降和随机梯度下降——使用 Gluon . . . . .	275
7.4	动量法——从零开始 . . . . .	280
7.5	动量法——使用 Gluon . . . . .	287
7.6	Adagrad——从零开始 . . . . .	291
7.7	Adagrad——使用 Gluon . . . . .	295
7.8	RMSProp——从零开始 . . . . .	297
7.9	RMSProp——使用 Gluon . . . . .	302
7.10	Adadelta——从零开始 . . . . .	304
7.11	Adadelta——使用 Gluon . . . . .	308
7.12	Adam——从零开始 . . . . .	310
7.13	Adam——使用 Gluon . . . . .	315
<b>8</b>	<b>计算性能</b>	<b>319</b>
8.1	命令式和符号式混合编程 . . . . .	319
8.2	异步计算 . . . . .	327
8.3	自动并行计算 . . . . .	334
8.4	多 GPU 计算——从零开始 . . . . .	337
8.5	多 GPU 计算——使用 Gluon . . . . .	343
<b>9</b>	<b>计算机视觉</b>	<b>349</b>
9.1	图片增广 . . . . .	349
9.2	微调 . . . . .	357
9.3	物体检测：边界框和预测 . . . . .	363
9.4	目标检测 . . . . .	370

9.5	目标检测模型: SSD	385
9.6	目标检测模型: YOLO	407
9.7	语义分割	421
9.8	样式迁移	434
9.9	实战 Kaggle 比赛: 对原始图像文件分类 (CIFAR-10)	448
9.10	实战 Kaggle 比赛: 识别 120 种狗 (ImageNet Dogs)	460
<b>10</b>	<b>自然语言处理</b>	<b>473</b>
10.1	词向量: word2vec	473
10.2	词向量: GloVe 和 fastText	480
10.3	求近似词和类比词	485
10.4	文本分类: 情感分析	491
10.5	编码器—解码器 (seq2seq)	499
10.6	注意力机制	502
10.7	机器翻译	504
<b>11</b>	<b>附录</b>	<b>515</b>
11.1	数学基础	515
11.2	在 AWS 上运行教程	518
11.3	GPU 购买指南	529
11.4	gluonbook 函数索引	533

这是一个深度学习的教学项目。我们将使用 [Apache MXNet \(incubating\)](#) 的最新 gluon 接口来演示如何从 0 开始实现深度学习的各个算法。我们的将利用 [Jupyter notebook](#) 能将文档，代码，公式和图形统一在一起的优势，提供一个交互式的学习体验。这个项目可以作为一本书，上课用的材料，现场演示的案例，和一个可以尽情拷贝的代码库。据我们所知，目前并没有哪个项目能既覆盖全面深度学习，又提供交互式的可执行代码。我们将尝试弥补这个空白。

- 第一季十九课视频汇总
- 可打印的 PDF 版本[在这里](#)
- 课程源代码在[Github](#)（亲，给个好评加颗星）
- 请使用 <http://discuss.gluon.ai/> 来进行讨论



---

## 前言

---

### 1.1 为什么要做这个项目

两年前我们开始了 MXNet 这个项目，有一件事情一直困扰我们：每当 MXNet 发布新特性的时候，总会收到“做啥新东西，赶紧去更新文档”的留言。我们曾一度都很费解，文档明明很多啊，比我们以前所有做的项目都好。而且你看隔壁家轮子，都没文档，大家照样也不是用的很嗨。

后来有一天，Zack 问了这样一个问题：假设回到你刚开始学机器学习的时候，那么你需要什么样的文档？

我是大二开始接触机器学习。那时候并没有太多很好资料，抱着晦涩的翻译版《The Elements of Statistical Learning》读了大半年仍是懵懵懂懂。后来 08 年的时候又啃了好几个月《Pattern Recognition And Machine Learning》，被贝叶斯那一套绕得云里雾里。10 年去港科大的时候 James 问我，你最熟悉的模型是哪个？使劲想了想，竟然答不出来。

虽然在我认识的人里，好些人能够读一篇论文或者听一个报告后就能问出很好的问题，然后就基本弄懂了。但我在这个上笨很多。读过的论文就像喝过的水，第二天就不记得了。一定是需要静下心来，从头到尾实现一篇，跑上几个数据，调些参数，才能心安地觉得懂了。例如在港科大的

两年读了很多论文，但现在反过来看，仍然记得可能就是那两个老老实实动手实现过写过论文的模型了。即使后来在机器学习这个方向又走了五年，学习任何新东西仍然是要靠动手。



## 纸上得来终觉浅，绝知此事要躬行

几年前我开始学习深度学习，在 MXNet 这个项目里也帮助和目睹了很多小伙伴上手深度学习。我发现也有很多小伙伴跟我一样，动手去实现、去调参、去跑实验才会真正成为专家（或者合格的炼丹师）。

虽然深度学习崛起前的年代，不写代码不跑实验可以做出很好的理论工作。但在深度学习领域，动手能力才是核心竞争力。例如就算我熟知卷积的三种写法，Relu 的十个变种，理解 BatchNorm 为什么能加速收敛，对 Imagenet 历届冠军的错误率随手拈来，能滔滔不绝说上几小时神经网络几度沉浮的恩怨史。但调不出参数，一切都是枉然。发论文被问你为啥跟 state-of-the-art 差老远，做产品被喷你这精度还不如我的便宜 100 倍的线性模型。



## 道理我都懂，但仍调不出这个参

在过去一年我在 AWS 工作中，很大一部分是在帮助 Amazon 内部团队和云上的用户来了解深度学习，并将其应用到他们的产品中。在今年夏威夷的 CVPR 上，遇到很多老朋友，例如地平线的凯哥，今日头条的李磊，第四范式的文渊和雨强，也认识了很多新朋友，例如 Momenta 旭东和商汤俊杰。我说 MXNet 有了新 Gluon 前端，可以一次性解决产品和研究的需求。大家纷纷表示，好啊好啊，来我们这里讲讲吧。而且特别强调说，我们这里新人很多，最好能讲讲入门知识。

所以很自然的会想，我们能不能帮助更多人。于是我们想开设一些系列课程，从深度学习入门到最新最前沿的算法，从 0 开始通过交互式的代码来讲解每个算法和概念。希望通过这个让大家既能了解算法的细节，又能调得出参数。既赢得了竞赛，又做的出产品。

为此我们做了（正在做）这五件事情：

1. Eric 和 Sheng 开发了 MXNet 的新前端 Gluon，详细可以参见 Eric 的这篇介绍。这个前端带

来跟 Python 更一致的便利的编程环境，不管是 debug 还是在交互上，都比 TensorFlow 之类通过计算图编程的框架更适合学习深度学习。

2. Zack, Alex, Aston 和很多小伙伴一起写了一系列的 notebook 来讲解各个模型。Zack 从一个外行（他是专业音乐人）和老师（CMU 计算机教授）的角度，从 0 开始讲解和实现各个算法。
3. 我们同时将 notebook 翻译成中文。虽然翻译进度落后了英文版，但对每个翻译了教程都做了大量的改进（之后会 merge 回英文版）
4. 建立了中文社区 [discuss.gluon.ai](#) 方便大家来讨论和学习。
5. 我们联合 [将门](#) 在斗鱼上直播一系列课程，深入讲解各个教程。

在我们准备这个的时候，Andrew Ng 也开设了深度学习课程。从课程单上看非常好，讲得特别细。而且 Andrew 讲东西一向特别清楚，所以这个课程必然是精品。但我们做的跟 Andrew 的主要有几个区别：

1. 我们不仅介绍深度学习模型，而且提供简单易懂的代码实现。我们不是通过幻灯片来讲解，而是通过解读代码，实际动手调参数和跑实验来学习。
2. 我们使用中文。不管是教材，直播，还是论坛。（虽然在美国呆了 5, 6 年了，事实上我仍然对一边听懂各式口音的英文一边理解内容很费力。）
3. Andrew 课目前免费版只能看视频，而我们不仅仅直播教学，而且提供练习题，提供大家交流的论坛，并鼓励大家在 [github](#) 上参与到课程的改进中来。希望能与大家有更近距离的交互。

从大出发点上我们跟 Andrew 一致，希望能够帮助小伙伴们快速掌握深度学习。这一次技术上的创新可能会持续辐射技术圈数年，希望小伙伴们能更快更好的参与到这一次热潮来。

@mli

## 1.2 前言

这个项目是我们尝试构建的一个有关深度学习的新型教育资源。我们的目标，是利用 Jupyter notebooks 的优势，将文字、图片、公式、以及（非常重要的）代码呈现在一起。如果这个尝试能够成功，其成果将会是一个极好的资源，它既是一本书、同时也是课程材料、现场教学的补充，甚至有剽窃价值的代码库（此处附上我们的“祝福”）。据我们所了解的，目前仅有很少的资源旨

在教授（1）全方位有关现代机器学习的概念，或（2）一本引人入胜的教科书并搭配可运行的代码。相信这次尝试最终能够告诉我们，这种空白是否情有可原。

这些年机器学习社区和生态圈进入一个令人费解的状态。二十一世纪早期的时候虽然只有少数一些问题被攻克了，但当时我们自认为理解了这些模型运行的方式及原因（以及不少的坑）。对比现在，机器学习系统已经非常强大，但却留下一个巨大问题：为什么它们如此有效？

这个新世界提供了巨大的机会，同时也带来了浮躁的投机。现在研究预印本被标题党和肤浅的内容充斥，人工智能创业公司只需要几个演示就能获得巨大的估值，朋友圈也被不懂技术的营销人员写的小白文刷屏。这的确是个看似混乱、充斥着快钱和宽松标准的时代。

于是，我们精心打磨了这套深度学习教程项目。

### 1.2.1 教程的组织方式

目前我们使用下面这个方式来组织每个具体教程（除背景知识介绍教程）：

1. 引入一个（或者少数几个）新概念
2. 提供一个使用真实数据的完整样例

在这套教程中，我们会穿插介绍相应的背景知识。为了保证教程的流畅性，有些时候我们会将某个深度学习的模块视作一个黑箱。这种情况下，我们仅简要介绍该模块的基本作用，而将它的详细介绍放在稍后的篇章。举例来说，虽然深度学习需要使用某个特定的优化算法，但我们在一开始介绍某些深度学习方法时并不会对其中所使用的优化算法做具体展开，而是会在稍后的篇章里详细描述和讨论这些优化算法。这样一来，读者可以在不关心具体模块细节的情况下，用最短的时间掌握深度学习的主要框架和基本脉络。从业者也可快速了解自己需要使用的模型并简单粗暴地将教程里的代码直接应用在解决自己的实际问题中。

### 1.2.2 独特的学习体验

这套深度学习教程将为大家呈现以下独特的学习体验。

#### 易用高效的 MXNet

我们将使用 MXNet 作为这套教程所使用的深度学习库，并重点介绍全新的高层抽象包 Gluon。我们选用 MXNet 是因为它兼具易用和高效的优点。无论对研究者还是对工程师而言，无论是在科研机构还是在工业界，工具的易用与高效将从各个方面显著提升生产效率。

## 双轨学习法

在介绍大多数机器学习模型时，我们既会教授大家如何从零开始实现模型，也会教授大家如何使用高层抽象包 Gluon 实现模型。从零开始实现模型有助大家深入理解深度学习底层设计。使用高层抽象包 Gluon 将把大家从繁琐的模型模块设计与实现中解放出来。

## 通过动手来学习

许多教科书在介绍深度学习时，都极尽所能地呈现所有细节。例如 Chris Bishop 的经典教材，《模式识别和机器学习》，将每个课题都讲解得极为详细，以致于阅读至线性回归的一章就需要巨大的工作量。当我（Zack）首次接触机器学习时，便发现这种讲解方法并不适合作为一本入门读物。而多年后重读它时，我热爱它精确而缜密的讲解，但仍不认为这是一本初次学习时应该使用的教材。

我们坚信，学习深度学习的最好方式就是动手实现深度学习模型。

游戏之所以好玩，是因为游戏给玩家提供了及时反馈：提高属性立即就可以虐怪、打个怪立即就可以升经验值、捡个包裹立即就多了装备。学习之所以枯燥，是因为很多时候我们并没有在学习过程中获得及时反馈。

这套教程通过描述深度学习模型是如何一步步实现的，为大家提供了宝贵的动手实践的机会。因为教程里实现的代码都是可执行的，读者可以根据自己所学和思考课后问题运行或修改代码而得到及时的学习反馈。每个人可以通过及时反馈不断实现自我迭代，从而加深对深度学习的理解。

最后，英文中有句话叫做

“Get hands dirty.”

直译过来就是

“撸起袖子加油干。”

## 1.3 主要符号一览

以下列举了本教程中使用的主要符号。

### 1.3.1 数

$x$ : 标量 (整数或实数)

$\boldsymbol{x}$ : 向量

$\boldsymbol{X}$ : 矩阵

$\mathbf{x}$ : 张量

### 1.3.2 集合

$\mathcal{X}$ : 集合

$\mathbb{R}$ : 实数集合

$\mathbb{R}^n$ :  $n$  维的实数向量集合

$\mathbb{R}^{x \times y}$ :  $x \times y$  维的实数矩阵集合

### 1.3.3 操作符

$(\cdot)^\top$ : 向量或矩阵的转置

$\odot$ : 按元素相乘

$|\mathcal{X}|$ : 集合  $\mathcal{X}$  中元素个数

$\|\cdot\|_p$ :  $L_p$  范数

$\|\cdot\|$ :  $L_2$  范数

$\sum$ : 连加

$\prod$ : 连乘

### 1.3.4 函数

$f(\cdot)$ : 函数

$\log(\cdot)$ : 自然对数函数

$\exp(\cdot)$ : 指数函数

### 1.3.5 导数和梯度

$\frac{dy}{dx}$ :  $y$  关于  $x$  的导数

$\frac{\partial y}{\partial x}$ :  $y$  关于  $x$  的偏导数

$\nabla_y$ :  $y$  关于  $\cdot$  的梯度

### 1.3.6 概率和统计

$\mathbb{P}(\cdot)$ : 概率分布

$\cdot \sim \mathbb{P}$ : 随机变量  $\cdot$  的概率分布是  $\mathbb{P}$

$\mathbb{P}(\cdot | \cdot)$ : 条件概率分布

$\mathbb{E}_\cdot(f(\cdot))$ : 函数  $f(\cdot)$  对  $\cdot$  的数学期望

### 1.3.7 复杂度

$\mathcal{O}$ : 大 O 符号 (渐进符号)

---

## 预备知识

---

### 2.1 机器学习简介

本书作者跟广大程序员一样，在开始写作前需要来一杯咖啡。我们跳进车准备出发，Alex 掏出他的安卓喊一声“OK Google”唤醒语言助手，Mu 操着他的中式英语命令到“去蓝瓶咖啡店”。手机快速识别并显示出命令，同时判断我们需要导航，并调出地图应用，给出数条路线方案，每条方案均有预估的到达时间并自动选择最快的线路。好吧，这是一个虚构的例子，因为我们一般在办公室喝自己的手磨咖啡。但这个例子展示了在短短几秒钟里，我们跟数个机器学习模型进行了交互。

如果你从来没有使用过机器学习，你会想，“这不就是编程吗？”或者，“机器学习(**machine learning**)到底是什么？”首先，我们确实是使用编程语言来实现机器学习模型，我们跟计算机其他领域一样，使用同样的编程语言和硬件。但不是每个程序都涉及机器学习。对于第二个问题，精确定义机器学习就像定义什么是数学一样难，但我们试图在这章提供一些直观的解释。

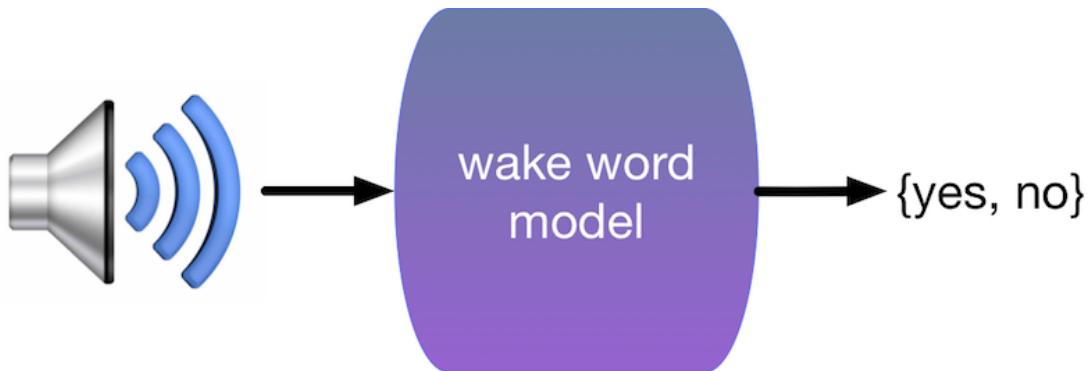
### 2.1.1 一个例子

我们日常交互的大部分计算机程序，都可以使用最基本的命令来实现。当你把一个商品加进购物车时，你触发了电商的电子商务程序，把一个商品 ID 和你的用户 ID 插入一个叫做购物车的数据表格中。你可以在没有见到任何真正客户前，用最基本的程序指令来实现这个功能。如果你发现可以这么做，那么这时就不需要使用机器学习。

对于机器学习科学家来说，幸运的是大部分应用没有那么简单。回到前面那个例子，想象下如何写一个程序来回应唤醒词，例如“Okay, Google”，“Siri”，和“Alexa”。如果在一个只有你自己和代码编辑器的房间里，仅使用最基本的指令编写这个程序，你该怎么做？不妨思考一下……这个问题非常困难。你可能会想像下面的程序：

```
if input_command == 'Okay, Google':  
    run_voice_assistant()
```

但实际上，你能拿到的只有麦克风里采集到的原始语音信号，可能是每秒 44,000 个样本点。怎样的规则才能把这些样本点转成一个字符串呢？或者简单点，判断这些信号中是否包含唤醒词。



如果你被这个问题难住了，不用担心。这就是我们为什么需要机器学习。

虽然我们不知道怎么告诉机器去把语音信号转成对应的字符串，但我们自己可以。换句话说，就算你不清楚怎么编写程序，好让机器识别出唤醒词“Alexa”，你自己完全能够识别出“Alexa”这个词。由此，我们可以收集一个巨大的数据集（**data set**），里面包含了大量语音信号，以及每个语音型号是否对应我们需要的唤醒词。使用机器学习的解决方式，我们并非直接设计一个系统去准确地辨别唤醒词，而是写一个灵活的程序，并带有大量的参数（**parameters**）。通过调整这些参数，我们能够改变程序的行为。我们将这样的程序称为模型（models）。总体上看，我们的模型仅仅是一个机器，通过某种方式，将输入转换为输出。在上面的例子中，这个模型的输入

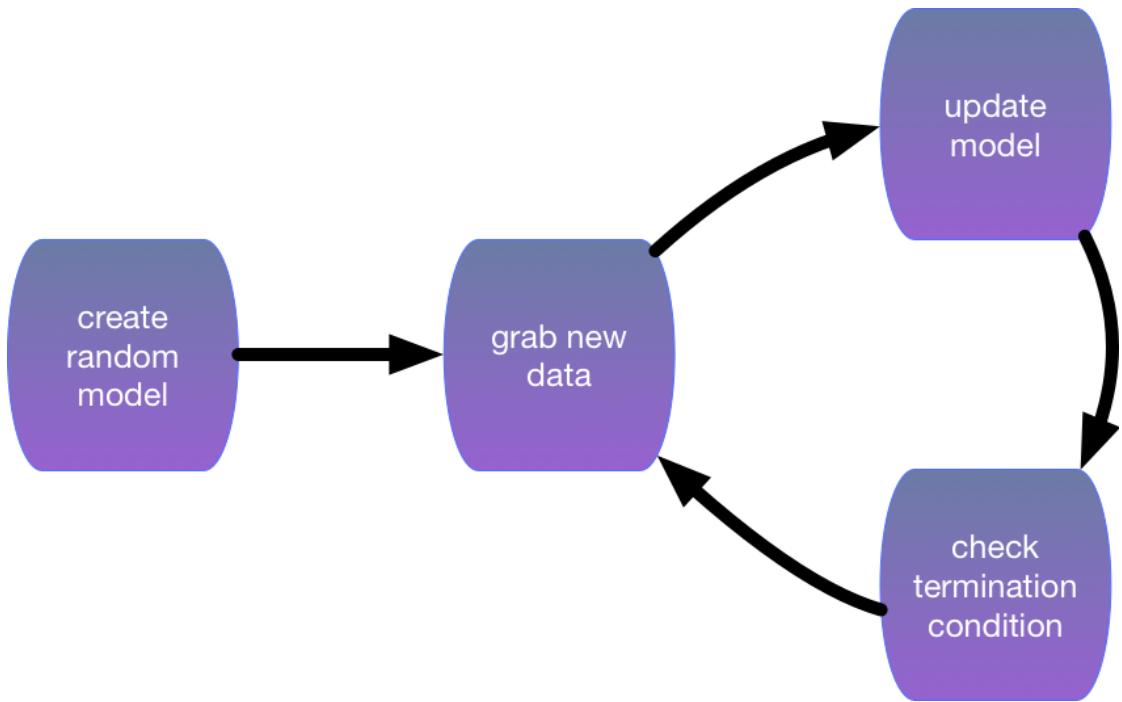
(**input**) 是一段语音信号，它的输出则是一个回答 {yes, no}，告诉我们这段语音信号是否包含了唤醒词。

如果我们选择了正确的模型，必然有一组参数设定，每当它听见“Alexa”时，都能触发 yes 的回答；也会有另一组参数，针对“Apricot”触发 yes。我们希望这个模型既可以辨别“Alexa”，也可以辨别“Apricot”，因为它们是类似的任务。不过，如果是本质上完全不同的输入和输出，比如输入图片，输出文本；或者输入英文，输出中文，这时我们则需要另一个的模型来完成这些转换。

这时候你大概能猜到了，如果我们随机地设定这些参数，模型可能无法辨别“Alexa”，“Apricot”，甚至任何英文单词。在而大多数的深度学习中，学习 (**learning**) 就是指在训练过程 (**training period**) 中更新模型的行为（通过调整参数）。

换言之，我们需要用数据训练机器学习模型，其过程通常如下：

1. 初始化一个几乎什么也不能做的模型；
2. 抓一些有标注的数据集（例如音频段落及其是否为唤醒词的标注）；
3. 修改模型使得它在抓取的数据集上能够更准确执行任务（例如使得它在判断这些抓取的音频段落是否为唤醒词上判断更准确）；
4. 重复以上步骤 2 和 3，直到模型看起来不错。



总而言之，我们并非直接去写一个唤醒词辨别器，而是一个程序，当提供一个巨大的有标注的数据集时，它能学习如何辨别唤醒词。你可以认为这种方式是利用数据编程。

如果给我们的机器学习系统提供足够多猫和狗的图片，我们可以“编写”一个喵星人辨别器：



如果是一只猫，辨别器会给出一个非常大的正数；如果是一只狗，会给出一个非常大的负数；如果不能确定的话，数字则接近于零。这仅仅是一个机器学习应用的粗浅例子。

## 2.1.2 眼花缭乱的机器学习应用

机器学习背后的核心思想是，设计程序使得它可以在执行的时候提升它在某任务上的能力，而非直接编写程序的固定行为。机器学习包括多种问题的定义，提供很多不同的算法，能解决不同领域的各种问题。我们之前讲到的是一个讲监督学习（**supervised learning**）应用到语言识别的例子。

正因为机器学习提供多种工具，可以利用数据来解决简单规则无法或者难以解决的问题，它被广泛应用于了搜索引擎、无人驾驶、机器翻译、医疗诊断、垃圾邮件过滤、玩游戏（国际象棋、围棋）、人脸识别、数据匹配、信用评级和给图片加滤镜等任务中。

虽然这些问题各式各样，但他们有着共同的模式，都可以使用机器学习模型解决。我们无法直接编程解决这些问题，但我们能够使用配合数据编程来解决。最常见的描述这些问题的方法是通过数学，但不像其他机器学习和神经网络的书那样，我们会主要关注真实数据和代码。下面我们来看点数据和代码。

## 2.1.3 用代码编程和用数据编程

这个例子灵感来自 Joel Grus 的一次应聘面试。面试官让他写个程序来玩 Fizz Buzz。这是一个小孩子游戏。玩家从 1 数到 100，如果数字被 3 整除，那么喊‘fizz’，如果被 5 整除就喊‘buzz’，如果两个都满足就喊‘fizzbuzz’，不然就直接说数字。这个游戏玩起来就像是：

1 2 fizz 4 buzz fizz 7 8 fizz buzz 11 fizz 13 14 fizzbuzz 16 …

传统的实现是这样的：

```
In [1]: res = []
for i in range(1, 101):
    if i % 15 == 0:
        res.append('fizzbuzz')
    elif i % 3 == 0:
        res.append('fizz')
    elif i % 5 == 0:
        res.append('buzz')
    else:
        res.append(str(i))
print(' '.join(res))
```

```
1 2 fizz 4 buzz fizz 7 8 fizz buzz 11 fizz 13 14 fizzbuzz 16 17 fizz 19 buzz fizz 22
↪ 23 fizz buzz 26 fizz 28 29 fizzbuzz 31 32 fizz 34 buzz fizz 37 38 fizz buzz 41
↪ fizz 43 44 fizzbuzz 46 47 fizz 49 buzz fizz 52 53 fizz buzz 56 fizz 58 59
↪ fizzbuzz 61 62 fizz 64 buzz fizz 67 68 fizz buzz 71 fizz 73 74 fizzbuzz 76 77
↪ fizz 79 buzz fizz 82 83 fizz buzz 86 fizz 88 89 fizzbuzz 91 92 fizz 94 buzz fizz
↪ 97 98 fizz buzz
```

对于经验丰富的程序员来说这个太不够一颗赛艇了。所以 Joel 尝试用机器学习来实现这个。为了让程序能学，他需要准备下面这个数据集：

- 数据  $X$   $[1, 2, 3, 4, \dots]$  和标注  $Y$   $['fizz', 'buzz', 'fizzbuzz', identity]$
- 训练数据，也就是系统输入输出的实例。例如  $[(2, 2), (6, \text{fizz}), (15, \text{fizzbuzz}), (23, 23), (40, \text{buzz})]$
- 从输入数据中抽取的特征，例如  $x \rightarrow [(x \% 3), (x \% 5), (x \% 15)]$ .

有了这些，Joel 利用 TensorFlow 写了一个分类器。对于不按常理出牌的 Joel，面试官一脸黑线。而且这个分类器不是总是对的。

显而易见，这么解决问题蠢爆了。为什么要用复杂和容易出错的东西替代几行 Python 呢？但是，很多情况下，这么一个简单的 Python 脚本并不存在，而一个三岁小孩就能完美地解决问题。这时候，机器学习就该上场了。

## 2.1.4 机器学习最简要素

针对识别唤醒语的任务，我们将语音片段和标注（label）放在一起组成数据集。接着我们训练一个机器学习模型，给定一段语音，预测它的标注。这种给定样例预测标注的方式，仅仅是机器学习的一种，称为监督学习。深度学习包含很多不同的方法，我们会在后面的章节讨论。成功的机器学习有四个要素：数据、转换数据的模型、衡量模型好坏的损失函数和一个调整模型权重来最小化损失函数的算法。

### 数据（Data）

越多越好。事实上，数据是深度学习复兴的核心，因为复杂的非线性模型比其他机器学习需要更多的数据。数据的例子包括：

- 图片：你的手机图片，里面可能包含猫、狗、恐龙、高中同学聚会或者昨天的晚饭；卫星照片；医疗图片例如超声、CT 扫描以及 MRI 等等。

- 文本：邮件、高中作文、微博、新闻、医嘱、书籍、翻译语料库和微信聊天记录。
- 声音：发送给智能设备（比如 Amazon Echo, iPhone 或安卓智能机）的声控指令、有声书籍、电话记录、音乐录音，等等。
- 影像：电视节目和电影，Youtube 视频，手机短视频，家用监控，多摄像头监控等等。
- 结构化数据：Jupyter notebook（里面有文本，图片和代码）、网页、电子病历、租车记录和电费账单。

## 模型（Models）

通常，我们拿到的数据和最终想要的结果相差甚远。例如，想知道照片中的人是不是开心，我们希望有一个模型，能将成千上万的低级特征（像素值），转化为高度抽象的输出（开心程度）。选择正确模型并不简单，不同的模型适合不同的数据集。在这本书中，我们会主要聚焦于深度神经网络模型。这些模型包含了自上而下联结的数据多层连续变换，因此称之为深度学习（**deep learning**）。在讨论深度神经网络之前，我们也会讨论一些简单、浅显的模型。

## 损失函数（Loss Functions）

我们需要对比模型的输出和真实值之间的误差。损失函数可以衡量输出结果对比真实数据的好坏。例如，我们训练了一个基于图片预测病人心率的模型。如果模型预测某个病人的心率是 100bpm，而实际上仅有 60bpm，这时候，我们就需要某个方法来提点一下这个的模型了。

类似的，一个模型通过给电子邮件打分来预测是不是垃圾邮件，我们同样需要某个方法判断模型的结果是否准确。典型的机器学习过程包括将损失函数最小化。通常，模型包含很多参数。我们通过最小化损失函数来“学习”这些参数。可惜，将损失降到最小，并不能保证我们的模型在遇到（未见过的）测试数据时表现良好。由此，我们需要跟踪两项数据：

- 训练误差（**training error**）：这是模型在用于训练的数据集上的误差。类似于考试前我们在模拟试卷上拿到的分数。有一定的指向性，但不一定保证真实考试分数。
- 测试误差（**test error**）：这是模型在没见过的新数据上的误差，可能会跟训练误差很不一样（统计上称之为过拟合）。类似于考前模考次次拿高分，但实际考起来却失误了。（笔者之一曾经做 GRE 真题时次次拿高分，高兴之下背了一遍红宝书就真上阵考试了，结果最终拿了一个刚刚够用的低分。后来意识到这是因为红宝书里包含了大量的真题。）

## 优化算法 (Optimization Algorithms)

最后，我们需要算法来通盘考虑模型本身和损失函数，对参数进行搜索，从而逐渐最小化损失。最常见的神经网络优化使用梯度下降法作为优化算法。简单地说，轻微地改动参数，观察训练集的损失将如何移动。然后将参数向减小损失的方向调整。

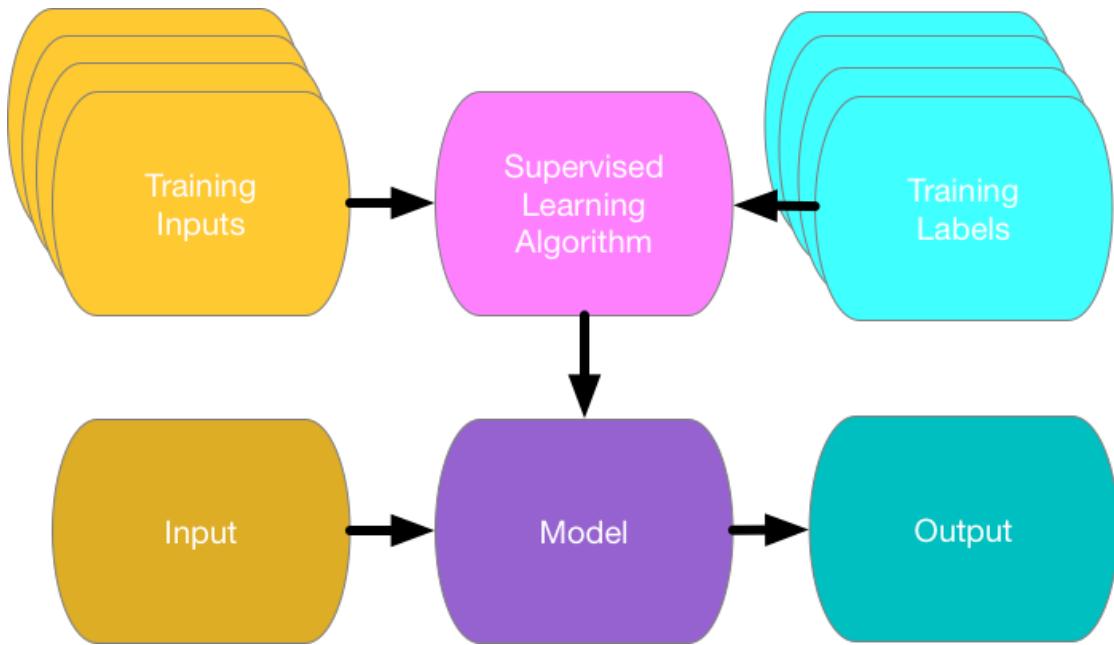
下面我们详细讨论一些不同的机器学习应用，以便理解我们具体会做什么。每个例子，我们都会介绍问题的解决方式，如训练方式、数据类型等等。这些内容仅仅是开胃小菜，并为大家提供一些共同的谈资。随着学习的推进，我们会介绍更多的类似问题。

### 2.1.5 监督学习 (Supervised Learning)

监督学习描述的任务是，当给定输入  $x$ ，如何通过在有标注输入和输出的数据上训练模型而能够预测输出  $y$ 。从统计角度来说，监督学习主要关注如何估计条件概率  $P(y|x)$ 。监督学习仅仅是机器学习的方法之一，在实际情景中却最为常用。部分原因，许多重要任务都可以描述为给定数据，预测结果。例如，给定一位患者的 CT 图像，预测该患者是否得癌症；给定英文句子，预测它的正确中文翻译；给定本月公司财报数据，预测下个月该公司股票价格。

“根据输入预测结果”，看上去简单，监督学习的形式却十分多样，其模型的选择也至关重要，数据类型、大小、输入和输出的体量都会产生影响。例如，针对序列型数据（文本字符串或时间序列数据）和固定长度的矢量表达，这两种不同的输入，会采用不同的模型。我们会在本书的前 9 章逐一介绍这些问题。

简单概括，学习过程看起来是这样的：在一大组数据中随机地选择样本输入，并获得其真实 (ground-truth) 的标注 (**label**)；这些输入和标注（即期望的结果）构成了训练集 (**training set**)。我们把训练集放入一个监督学习算法 (**supervised learning algorithm**)。算法的输入是训练集，输出则是学得模型 (**learned model**)。基于这个学得模型，我们输入之前未见过的测试数据，并预测相应的标注。



## 回归分析（Regression）

回归分析也许是监督学习里最简单的一类任务。在该项任务里，输入是任意离散或连续的、单一或多个的变量，而输出是连续的数值。例如我们可以把本月公司财报数据抽取出若干特征，如营收总额、支出总额以及是否有负面报道，利用回归分析预测下个月该公司股票价格。

一个更详细的例子，一个房屋的销售的数据集。构建一张表：每一行对应一幢房子；每一列对应一个属性，譬如面积、卧室数量、卫生间数量、与市中心的距离；我们将这个数据集里这样的一行，称作一个特征向量（**feature vector**），它所代表的对象（比如一幢房子），称作样例（**example**）。

如果住在纽约或三藩这种大城市，你也不是某个知名科技公司 CEO，那么这个特征向量（面积、卧室数量、卫生间数量、与市中心的距离）可能是这样的  $[100, 0, .5, 60]$ 。如果住在匹兹堡，则可能是这样的  $[3000, 4, 3, 10]$ 。这些特征向量，是所有经典机器学习问题的关键。我们一般将一个特征向量标为  $\mathbf{x}_i$ ，将所有特征向量的集合标为  $X$ 。

一个问题是否应采用回归分析，取决于它的输出。比如你想预测一幢房子的合理的市场价格，并提供了类似的特征向量。它的目标市场价是一个实数。我们将单个目标（对应每一个特征向量代表的样例  $\mathbf{x}_i$  标为  $y_i$ ，所有目标的集合为  $\mathbf{y}$ （对应所有样例的集合  $X$ ）。当我们的目标是某个范围内的任意实数值时，这就是一个回归分析问题。模型的目标就是输出预测（在这个例子中，即价

格的预测)，且尽可能近似实际的目标值。我们将这些预测标为  $\hat{y}_i$ ；如果这堆符号看起来费解，不用担心，在接下来的章节中我们会详细讲解。

许多实际问题都是充分描述的回归问题。比如预测观众给某部电影的打分；如果 2009 年时你设计出一个这样一个算法，Netflix 的一百万大奖就是你的了；预测病人会在医院停留的时间也是一个回归问题。一条经验就是，问题中如果包含“多少？”，这类问题一般是回归问题。“这次手术需要几个小时？”……回归分析。“这张照片里有几只狗？”……回归分析。不过，如果问题能够转化为“这是一个 \_\_\_\_\_ 吗？”，那这很有可能是一个分类，或者属于其余我们将会谈及的问题。

如果我们把模型预测的输出值和真实的输出值之间的差别定义为残差，常见的回归分析的损失函数包括训练数据的残差的平方和或者绝对值的和。机器学习的任务是找到一组模型参数使得损失函数最小化。我们会在之后的章节里详细介绍回归分析。

## 分类 (Classification)

回归分析能够很好地解答“多少？”的问题，但还有许多问题并不能套用这个模板。比如，银行手机 App 上的支票存入功能，用户用手机拍摄支票照片，然后，机器学习模型需要能够自动辨别照片中的文字。包括某些手写支票上龙飞凤舞的字体。这类的系统通常被称作光学字符识别 (OCR)，解决方法则是分类。较之回归分析，它的算法需要对离散集进行处理。

回归分析所关注的预测可以解答输出为连续数值的问题。当预测的输出是离散的类别时，这个监督学习任务就叫做分类。分类在我们日常生活中很常见。例如我们可以把本月公司财报数据抽取出若干特征，如营收总额、支出总额以及是否有负面报道，利用分类预测下个月该公司的 CEO 是否会离职。在计算机视觉领域，把一张图片识别成众多物品类别中的某一类，例如猫、狗等。

在分类问题中，我们的特征向量，例如，图片中的一个像素值，然后，预测它在一组可能值中所属的分类（一般称作类别 (**class**)）。对于手写数字来说，我们有数字 1~10 这 10 个类别，最简单的分类问题莫过于二分类 (binary classification)，即仅有两个类别的分类问题。例如，我们的数据集  $X$  是含有动物的图片，标注  $Y$  是类别 {cat, dog}。在回归分析里，输出是一个实数  $\hat{y}$  值。而在分类中，我们构建一个分类器 (**classifier**)，其最终的输出  $\hat{y}$  是预测的类别。

随着本书在技术上的逐渐深入，我们会发现，训练一个能够输出绝对分类的模型，譬如“不是猫就是狗”，是很困难的。简单一点的做法是，我们使用概率描述它。例如，针对一个样例  $x$ ，模型会输出每一类标注  $k$  的一个概率  $\hat{y}_k$ 。因为是概率，其值必然为正，且和为 1。换句话说，总共有  $K$  个类别的分类问题，我们只需要知道其中  $K - 1$  个类别的概率，自然就能知道剩下那个类别的概率。这个办法针对二分类问题尤其管用，如果抛一枚不均匀的硬币有 0.6(60%) 的概率正面朝上，那么，反面朝上的概率就是 0.4(40%)。回到我们的喵汪分类，一个分类器看到了一张图片，输出

图上的动物是猫的概率  $\Pr(y = \text{cat} | x) = 0.9$ 。对于这个结果，我们可以表述成，分类器 90% 确信图片描绘了一只猫。预测所得的类别的概率大小，仅仅是一个确信度的概念。我们会在后面的章节中进一步讨论有关确信度的问题。

多于两种可能类别的问题称为多分类。常见的例子有手写字符识别 [0, 1, 2, 3 ... 9, a, b, c, ...]。分类问题的损失函数称为交叉熵（**cross-entropy**）损失函数。可以在 MXNet Gluon 中找到。

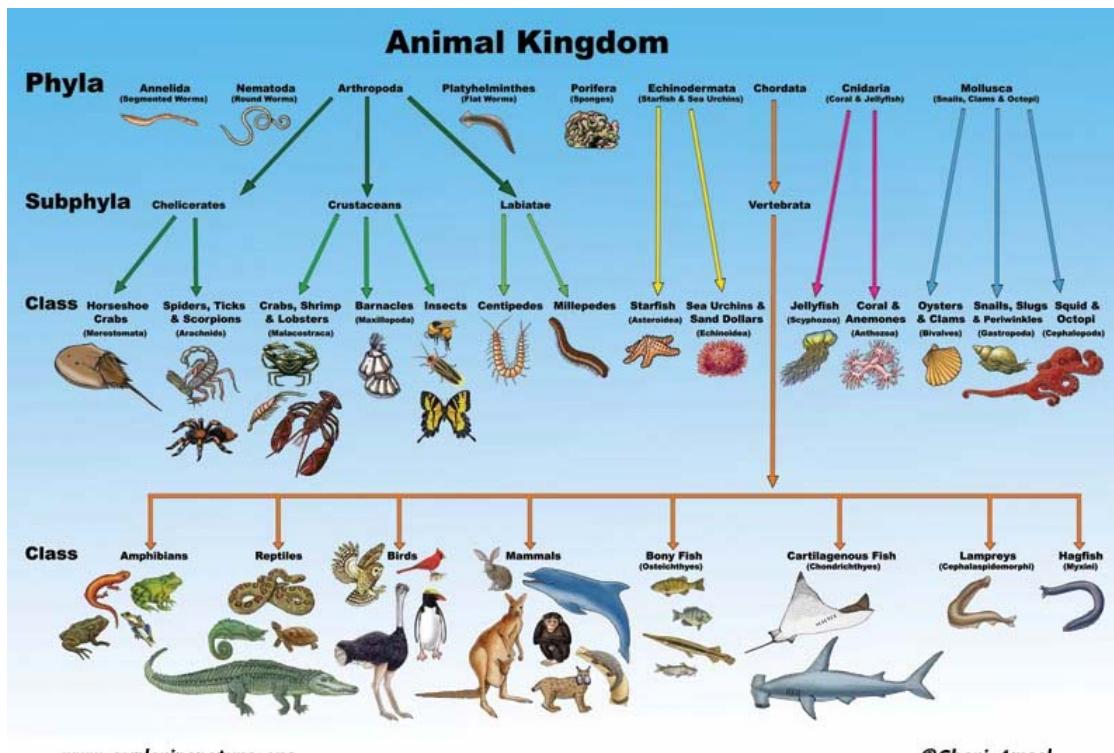
请注意，预测结果并不能左右最终的决策。比如，你在院子里发现了这个萌萌的蘑菇：



假定我们训练了一个分类器，输入图片，判断图片上的蘑菇是否有毒。这个“是否有毒分类器”输出了  $\Pr(y = \text{deathcap} | \text{image}) = 0.2$ 。换句话说，这个分类器 80% 确信这个蘑菇不是入口即挂的毒鹅膏。但我们不会蠢到去吃它。这点口腹之欲不值得去冒 20% 挂掉的风险。不确定的风险远远超过了收益。数学上，最最基本的，我们需要算一下预期风险，即，把结果概率与相应的收益（或损失）相乘：

$$L(\text{action} | x) = \mathbf{E}_{y \sim p(y|x)}[\text{loss}(\text{action}, y)]$$

很走运：正如任何真菌学家都会告诉我们的，上图真的就是一个毒鹅膏。实际上，分类问题可能比二分类、多分类、甚至多标签分类要复杂得多。例如，分类问题的一个变体，层次分类（**hierarchical classification**）。层次假设各个类别之间或多或少地存在某种关系。并非所有的错误都同等严重——错误地归入相近的类别，比距离更远的类别要好得多。一个早期的例子来自 [卡尔·林奈](#)，他发明了沿用至今的物种分类法。



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

©Sheri Amsel

图 2.1: 动物的分类

在动物分类的中，将贵宾犬误认为雪纳瑞并不是很严重的错误，但把贵宾犬和恐龙混淆就是另一回事了。但是，什么结构是合适的，取决于模型的使用。例如，响尾蛇和束带蛇在演化树上可能很近，但是把一条有毒的响尾蛇认成无毒的束带蛇可是致命的。

### 标注 (Tagging)

事实上，有一些看似分类的问题在实际中却难以归于分类。例如，把下面这张图无论分类成猫还是狗看上去都有些问题。



正如你所见，上图里既有猫又有狗。其实还没完呢，里面还有草啊、轮胎啊、石头啊等等。与其将上图仅仅分类为其中一类，倒不如把这张图里面我们所关心的类别都标注出来。比如，给定一张图片，我们希望知道里面是否有猫、是否有狗、是否有草等。给定一个输入，输出不定量的类别，这个就叫做标注任务。

这类任务预测非互斥分类的任务，叫做多标签分类。想象一下，人们可能会把多个标签同时标注在自己的某篇技术类博客文章上，例如“机器学习”、“科技”、“编程语言”、“云计算”、“安全与隐私”和“AWS”。这里面的标签其实有时候相互关联，比如“云计算”和“安全与隐私”。当一篇文章可能被标注的数量很大时，人力标注就显得很吃力。这就需要使用机器学习了。

生物医学领域也有这个问题，正确地标注研究文献能让研究人员对文献进行彻底的审阅。在美国国家医学图书馆，一些专业的文章标注者会浏览 PubMed 中索引的每篇文章，并与 MeSH，一个大约 28k 个标签的集合中的术语相关联。这是一个漫长的过程，通常在文章归档一年后才会轮到标注。在手动审阅和标注之前，机器学习可以用来提供临时标签。事实上，近几年，BioASQ 组织已经举办过相关比赛。

## 搜索与排序（Search and Ranking）

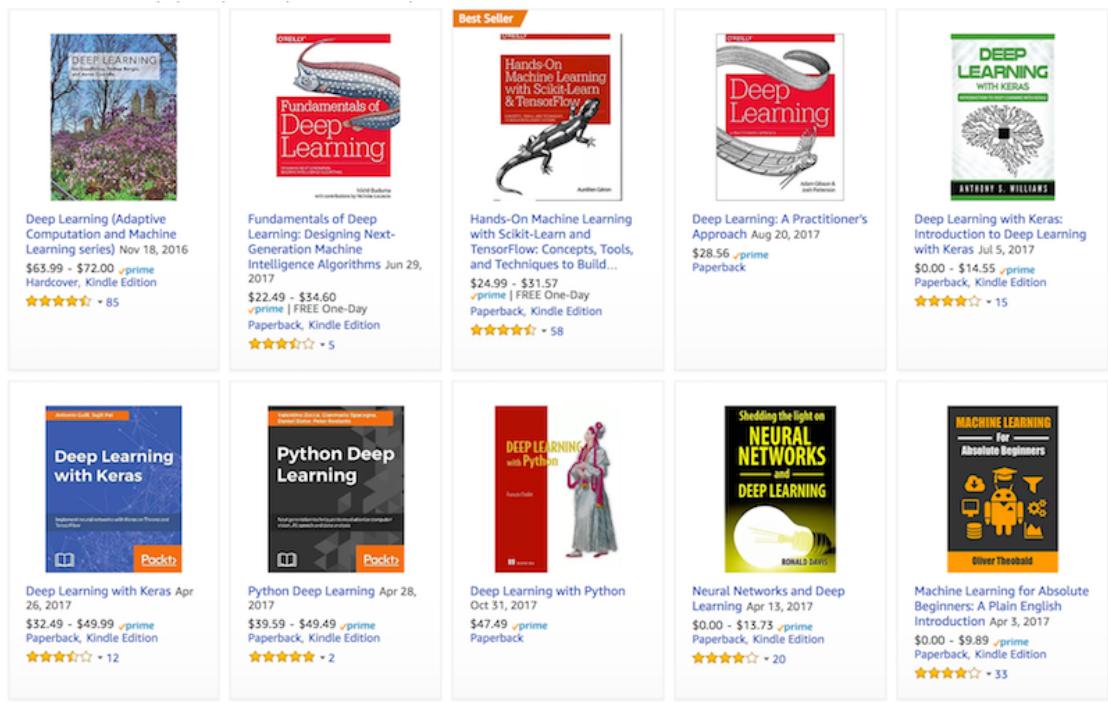
在信息检索领域，我们常常需要进行排序。以网络搜索为例，相较于判断页面是否与检索条目有关，我们更倾向于判断在浩如烟海的搜索结果中，应向用户显示哪个结果。这就要求我们的算法，能从较大的集合中生成一个有序子集。换句话说，如果要求生成字母表中的前 5 个字母，返回 A B C D E 和 C A B E D 是完全不同的。哪怕集合中的元素还是这些，但元素的顺序意义重大。

一个可行的解决方案，是用相关性分数对集合中的每个元素进行评分，并检索得分最高的元素。互联网时代早期有一个著名的网页排序算法叫做[PageRank](#)，就使用了这种方法。该算法的排序结果并不取决于用户检索条目，而是对包含检索条目的结果进行排序。现在的搜索引擎使用机器学习和行为模型来获得检索的相关性分数。有不少专门讨论这个问题的会议。

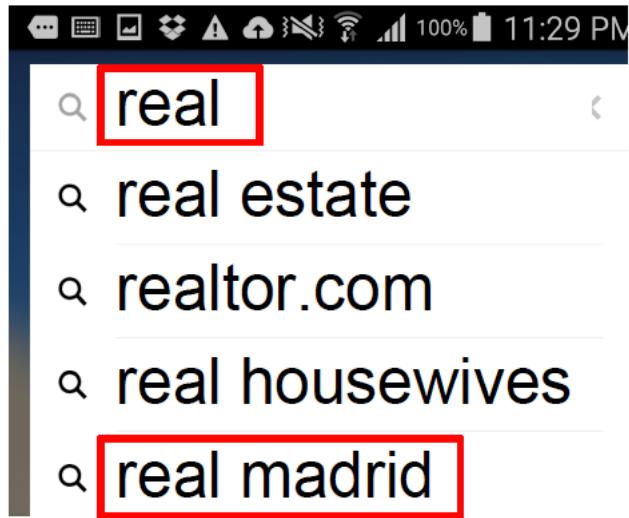
## 推荐系统（Recommender Systems）

推荐系统与搜索排序关系紧密，其目的都是向用户展示一组相关条目。主要区别在于，推荐系统特别强调每个用户的个性化需求。例如电影推荐，科幻电影粉和伍迪·艾伦喜剧爱好者的页面可能天差地别。

推荐系统被广泛应用于购物网站、搜索引擎、新闻门户网站等等。有时，客户会详细地表达对产品的喜爱（例如亚马逊的产品评论）。但有时，如果对结果不满意，客户只会提交简单的反馈（跳过播放列表中的标题）。通常，系统为用户  $u_i$  针对产品  $o_j$  构建函数，并估测出一个分数  $y_{ij}$ 。得分  $y_{ij}$  最高的产品  $o_j$  会被推荐。实际的系统，会更加先进地把详尽的用户活动和产品特点一并考虑。以下图片是亚马逊基于个性化算法和并结合作者偏好，推荐的深度学习书籍。



搜索引擎的搜索条目自动补全系统也是个好例子。它可根据用户输入的前几个字符把用户可能搜索的条目实时推荐自动补全。在笔者之一的某项工作里，如果系统发现用户刚刚开启了体育类的手机应用，当用户在搜索框拼出”real”时，搜索条目自动补全系统会把”real madrid”（皇家马德里，足球球队）推荐在比通常更频繁被检索的”real estate”（房地产）更靠前的位置，而不是总像下图中这样。



## 序列学习（Sequence Learning）

目前为止，我们提及的问题，其输入输出都有固定的长度，且连续输入之间不存在相互影响。如果我们的输入是一个视频片段呢？每个片段可能由不同数量的帧组成。且对每一帧的内容，如果我们一并考虑之前或之后的帧，猜测会更加准确。语言也是如此。热门的深度学习问题就包含机器翻译：在源语言中摄取句子，并预测另一种语言的翻译。

医疗领域也有类似的例子。我们希望构建一个模型，在密集治疗中监控患者的病情，并在未来 24 小时内的死亡风险超过某个阈值时发出警报。我们绝不希望这个模型每小时就把患者医疗记录丢弃一次，而仅根据最近的测量结果进行预测。

这些问题同样是机器学习的应用，它们是序列学习（**sequence learning**）的实例。这类模型通常可以处理任意长度的输入序列，或者输出任意长度的序列（或者两者兼顾！）。当输入和输出都是不定长的序列时，我们也把这类模型叫做 seq2seq，例如语言翻译模型和语音转录文本模型。以下列举了一些常见的序列学习案例。

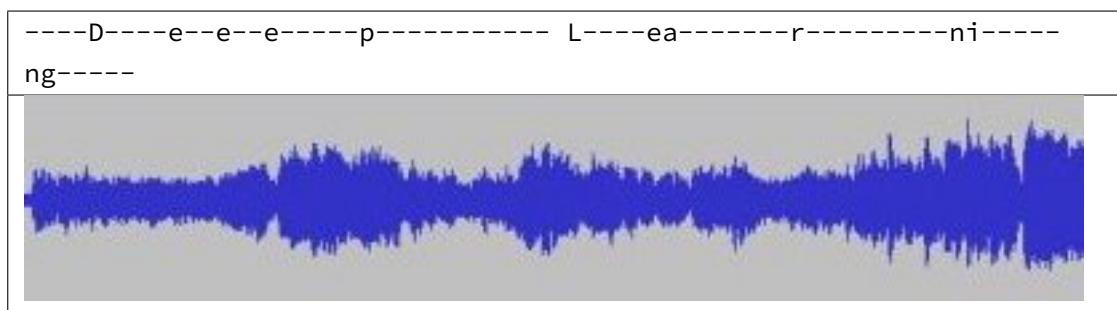
## 词类标注和句法分析（Tagging and Parsing）

使用属性来注释文本序列。给定一个文本序列，动词和主语分别在哪里，哪些词是命名实体。其目的是根据结构、语法假设来分解和注释文本。实际上并没有这么复杂。以下是一个这样的例子。其中 Tom、Washington 和 Sally 都是命名实体。

Tom wants to have dinner in Washington with Sally.
E - - - - E - E

## 语音识别（Automatic Speech Recognition）

在语音识别的问题里，输入序列  $x$  通常都是麦克风的声音，而输出  $y$  是对麦克风所说的话的文本转录。这类问题通常有一个难点，声音的采样率(常见的有 8kHz 或 16kHz)导致声音的帧数比文本长得多，声音和文本之间不存在一一对应，成千的帧样例可能仅对应一个单词。语音识别是一类 seq2seq 问题，这里的输出往往比输入短很多。



## 文本转语音（Text to Speech）

文本转语音（TTS）是语音识别问题的逆问题。这里的输入  $x$  是一个文本序列，而输出  $y$  是声音序列。因此，这类问题的输出比输入长。

## 机器翻译（Machine Translation）

机器翻译的目标是把一段话从一种语言翻译成另一种语言。目前，机器翻译时常会翻译出令人啼笑皆非的结果。主要来说，机器翻译的复杂程度非常高。同一个词在两种不同语言下的对应有时候是多对多。另外，符合语法或者语言习惯的语序调整也令问题更加复杂。

具体展开讲，在之前的例子中，输出中的数据点的顺序与输入保持一致，而在机器翻译中，改变顺序是需要考虑的至关重要的因素。虽然我们仍是将一个序列转换为另一个序列，但是，不论是输入和输出的数量，还是对应的数据点的顺序，都可能发生变化。比如下面这个例子，这句德语（Alex 写了这段话）翻译成英文时，需要将动词的位置调整到前面。

## 2.1.6 无监督学习（Unsupervised Learning）

迄今为止的例子都与监督学习有关，即我们为模型提供了一系列样例和一系列相应的目标值。你可以把监督学习看成一个非常专业的工作，有一个非常龟毛的老板。老板站在你的身后，告诉你每一种情况下要做什么，直到学会所有情形下应采取的行动。为这样的老板工作听起来很无趣；另一方面，这样的老板也很容易取悦，你只要尽快识别出相应的模式并模仿其行为。

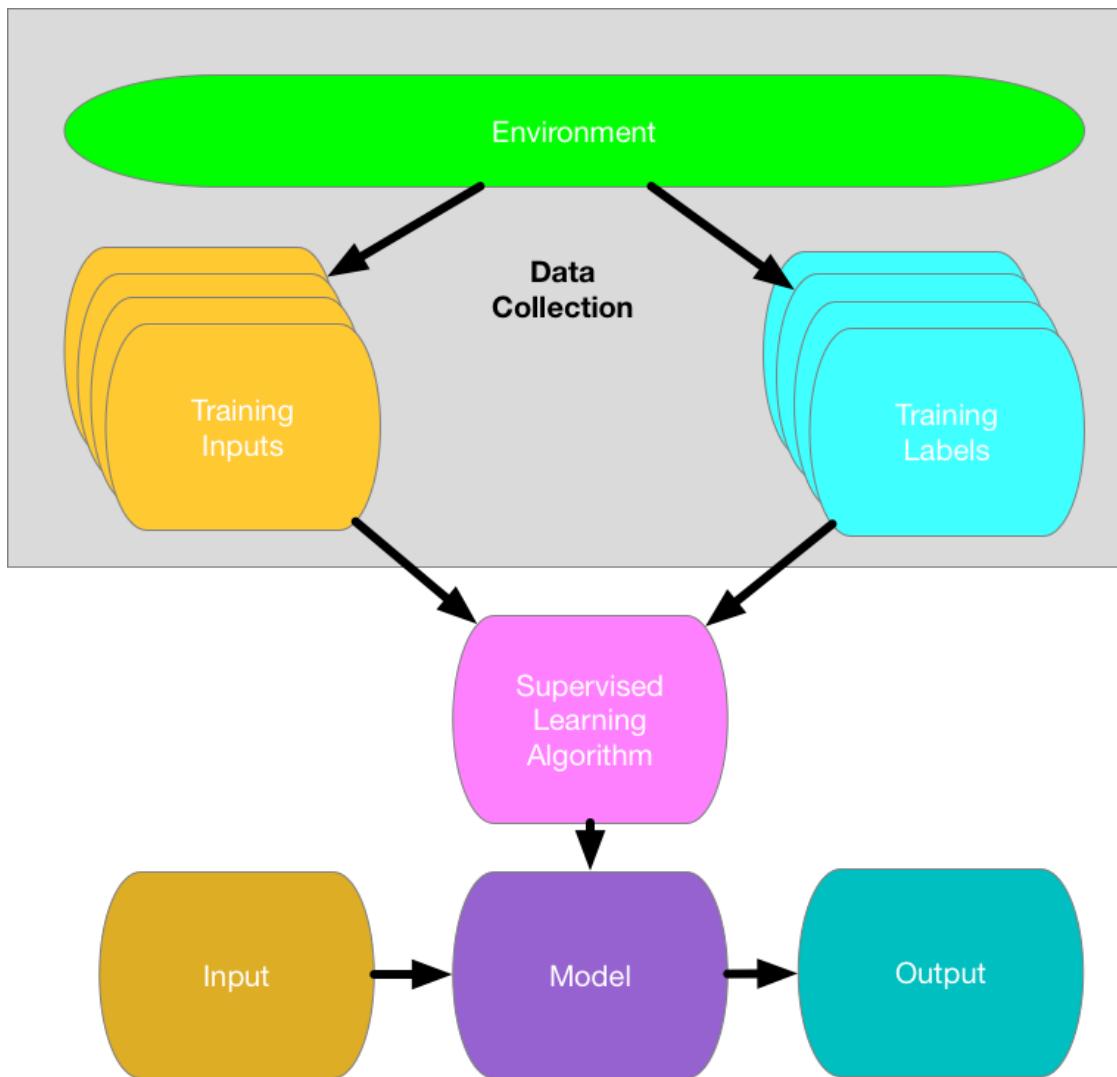
而相反的情形，给那种他们自己都不知道想让你做什么的老板打工也很糟心。不过，如果你打算成为一名数据科学家，你最好习惯这点。老板很可能就扔给你一堆数据，说，用上一点数据科学的方法吧。这种要求很模棱两可。我们称这类问题为无监督学习（**unsupervised learning**）。我们能提出的问题的类型和数量仅仅受创造力的限制。我们将在后面的章节中讨论一些无监督学习的技术。现在，介绍一些常见的非监督学习任务作为开胃小菜。

- 我们能用少量的原型，精准地概括数据吗？给我们一堆照片，能把它们分成风景、狗、婴儿、猫、山峰的照片吗？类似的，给定一堆用户浏览记录，我们能把他们分成几类典型的用户吗？这类问题通常被称为聚类（**clustering**）。
- 我们可以用少量的参数，准确地捕获数据的相关属性吗？球的轨迹可以很好地用速度，直径和质量准确描述。裁缝们也有一组参数，可以准确地描述客户的身材，以及适合的版型。这类问题被称为子空间估计（**subspace estimation**）问题。如果决定因素是线性关系的，则称为主成分分析（**principal component analysis**）。
- 在欧几里德空间（例如， $\mathbb{R}^n$  中的向量空间）中是否存在一种符号属性，可以表示出（任意构建的）原始对象？这被称为表征学习（**representation learning**）。例如我们希望找到城市的向量表示，从而可以进行这样的向量运算：罗马 - 意大利 + 法国 = 巴黎。
- 针对我们观察到的大量数据，是否存在一个根本性的描述？例如，如果我们有关于房价、污染、犯罪、地理位置、教育、工资等等的统计数据的，我们能否基于已有的经验数据，找出这些因素互相间的关联？贝叶斯图模型可用于类似的问题。
- 一个最近很火的领域，生成对抗网络（**generative adversarial networks**）。简单地说，是一套生成数据的流程，并检查真实数据与生成的数据是否在统计上相似。

## 2.1.7 与环境因素交互

目前为止，我们还没讨论过，我们的数据实际上来自哪里，以及当机器学习模型生成结果时，究竟发生了什么。这是因为，无论是监督学习还是无监督学习，都不会着重于这点。我们在初期抓

取大量数据，然后在不再与环境发生交互的情况下进行模式识别。所有的学习过程，都发生在算法和环境断开以后，这称作离线学习（**offline learning**）。对于监督学习，其过程如下：



离线学习的简洁别有魅力。优势在于，我们仅仅需要担心模式识别本身，而不需要考虑其它因素；劣势则在于，能处理问题的形式相当有限。如果你的胃口更大，从小就阅读阿西莫夫的机器人系列，那么你大概会想象一种人工智能机器人，不仅仅可以做出预测，而会采取实际行动。我们想要的是智能体（**agent**），而不仅仅是预测模型。意味着我们还要考虑选择恰当的动作（**action**），而动作会影响到环境，以及今后的观察到的数据。

一旦考虑到要与周围环境交互，一系列的问题接踵而来。我们需要考虑这个环境：

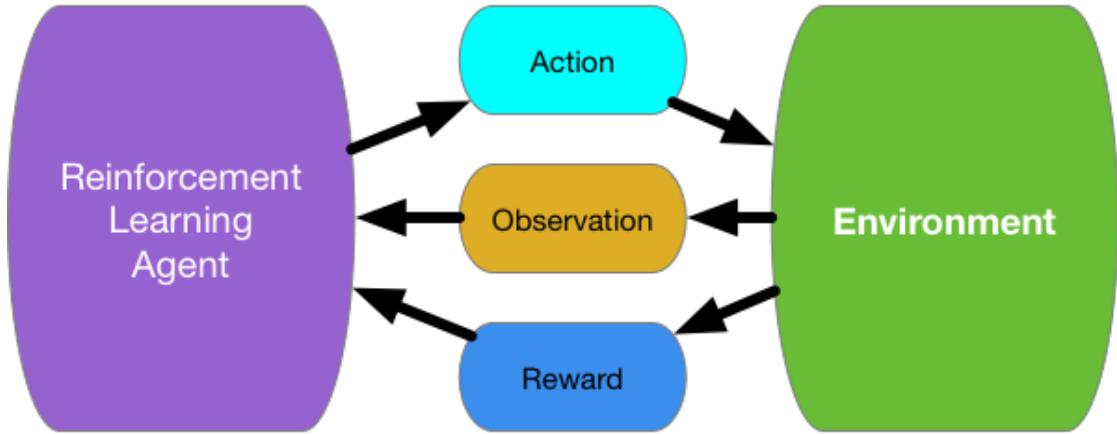
- 记得我们之前的行为吗？
- 愿意帮助我们吗？比如，一个能识别用户口述内容的语音识别器。
- 想要对抗我们？比如，一个对抗装置，类似垃圾邮件过滤（针对垃圾邮件发送者）或游戏玩家（针对对手）？
- 啥都不管（就像大多数情况）？
- 会动态地改变立场（随着时间表现稳定 vs 变化）？

最后的这个问题，引出了协变量转移（**covariate shift**）的问题（当训练和测试数据不同时）。这个坑想必不少人都经历过，平时的作业都是助教出题，而到了考试，题目却换成由课程老师编写。我们会简要介绍强化学习（**reinforcement learning**）和对抗学习（**adversarial learning**），这是两个会明确考虑与环境交互的问题。

### 强化学习（Reinforcement Learning）

如果你真的有兴趣用机器学习开发一个能与周围环境交互并产生影响的智能体，你大概需要专注于强化学习（以下简称 RL）。包括机器人程序、对话系统、甚至是电子游戏 AI 的开发。深度强化学习（**Deep reinforcement learning, DRL**）将深度神经网络应用到强化学习问题上是新的风潮。这一领域两个突出的例子，一个是突破性的 Deep Q Network，仅仅使用视觉输入就在街机游戏上击败了人类；另一个是著名的 AlphaGo 击败了围棋世界冠军。

强化学习给出了一个非常笼统的问题陈述，一个智能体在一系列的时间步长（**time steps**）中与周围的环境交互。在每个时间步长  $t$ ，智能体从环境中接收到一些观察数据  $o_t$ ，并选择一个动作  $a_t$ ，将其传回环境。最后，智能体会从环境中获得一个奖励（reward） $r_t$ 。由此，智能体接收一系列的观察数据，并选择一系列的后续动作，并一直持续。RL 智能体的行为受到策略（**policy**）约束。简而言之，所谓的策略，就是一组（对环境的）观察和动作的映射。强化学习的目标就是制定一个好的策略。



RL 框架的普适性并没有被夸大。例如，我们可以将任何监督学习问题转化为 RL 问题。譬如分类问题。我们可以创建一个 RL 智能体，每个分类都有一个对应的动作；然后，创建一个可以给予奖励的环境，完全等同于原先在监督学习中使用的损失函数。

除此以外，RL 还可以解决很多监督学习无法解决的问题。例如，在监督学习中，我们总是期望训练使用的输入是与正确的标注相关联。但在 RL 中，我们并不给予每一次观察这样的期望，环境自然会告诉我们最优的动作，我们只是得到一些奖励。此外，环境甚至不会告诉我们是哪些动作导致了奖励。

举一个国际象棋的例子。唯一真正的奖励信号来自游戏结束时的胜利与否；如果赢了，分配奖励 1；输了，分配 -1；而究竟是那些动作导致了输赢却并不明确。因此，强化学习者必须处理信用分配问题（**credit assignment problem**）。另一个例子，一个雇员某天被升职；这说明在过去一年中他选择了不少好的动作。想要继续升职，就必须知道是那些动作导致了这一升职奖励。

强化学习者可能也要处理部分可观察性（partial observability）的问题。即当前的观察结果可能无法反映目前的所有状态（state）。一个扫地机器人发现自己被困在一个衣柜里，而房间中所有的衣柜都一模一样，要推测它的精确位置（即状态），机器人需要将进入衣柜前的观察一并放入考虑因素。

最后，在任何一个特定的时刻，强化学习者可能知道一个好的策略，但也许还有不少更优的策略，智能体从未尝试过。强化学习者必须不断决策，是否利用目前已知的最佳战略作为策略，还是去探索其它策略而放弃一些短期的奖励，以获得更多的信息。

## 马尔可夫决策过程，赌博机问题

一般的强化学习问题的初期设置都很笼统。动作会影响后续的观察数据。奖励只能根据所选择的动作观察到。整个环境可能只有部分被观察到。将这些复杂的因素通盘考虑，对研究人员来说要求有点高。此外，并非每一个实际问题都表现出这些复杂性。因此，研究人员研究了一些强化学习问题的特殊情况。

当环境得到充分观察时，我们将这类 RL 问题称为马尔可夫决策过程（**Markov Decision Process**，简称 **MDP**）。当状态不依赖于以前的动作时，我们称这个问题为情境式赌博机问题（**contextual bandit problem**）。当不存在状态时，仅仅是一组可选择的动作，并在问题最初搭配未知的奖励，这是经典的多臂赌博机问题（**multi-armed bandit problem**）。

### 2.1.8 小结

机器学习是一个庞大的领域。我们在此无法也无需介绍有关它的全部。有了这些背景知识铺垫，你是否对接下来的学习更有兴趣了呢？

吐槽和讨论欢迎点[这里](#)

## 2.2 安装和运行

为了便于动手学深度学习，让我们获取本书代码、安装并运行所需要的工具，例如 MXNet。在这一节中，我们将描述安装和运行所需要的命令。执行命令需要进入命令行模式：Linux/macOS 用户可以打开 Terminal 应用，Windows 用户可以在文件资源管理器的地址栏输入 cmd。

### 2.2.1 获取代码并安装运行环境

我们可以通过 Conda 或者 Docker 来获取本书代码并安装运行环境。下面将分别介绍这两种选项。

#### 选项一：通过 Conda 安装（推荐）

第一步，根据操作系统下载并安装 Miniconda（网址：<https://conda.io/miniconda.html>）。

第二步，下载包含本书全部代码的包，解压后进入文件夹。Linux/macOS 用户可以使用如下命令。

```
mkdir gluon-tutorials && cd gluon-tutorials  
curl https://zh.gluon.ai/gluon_tutorials_zh.tar.gz -o tutorials.tar.gz  
tar -xzvf tutorials.tar.gz && rm tutorials.tar.gz
```

Windows 用户可以用浏览器下载压缩文件(下载地址:[https://zh.gluon.ai/gluon\\_tutorials\\_zh.zip](https://zh.gluon.ai/gluon_tutorials_zh.zip)) 并解压。在解压目录文件资源管理器的地址栏输入 cmd 进入命令行模式。

在本步骤中，我们也可以配置下载源来使用国内镜像加速下载：

```
# 优先使用清华 conda 镜像。  
conda config --prepend channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/  
--free/  
  
# 或者选用科大 conda 镜像。  
conda config --prepend channels http://mirrors.ustc.edu.cn/anaconda/pkgs/free/
```

第三步，安装运行所需的依赖包并激活该运行环境。Linux/macOS 用户可以使用如下命令。

```
conda env create -f environment.yml  
source activate gluon
```

由于教程会使用 `matplotlib.pyplot` 函数作图，macOS 用户需要创建或访问 `~/.matplotlib/matplotlibrc` 文件并添加一行代码：`backend: TkAgg`。

Windows 用户可以使用如下命令。

```
conda env create -f environment.yml  
activate gluon
```

第四步，打开 Jupyter notebook。运行下面命令。

```
jupyter notebook
```

这时在浏览器打开 <http://localhost:8888> (通常会自动打开) 就可以查看和运行本书中每一节的代码了。

第五步 (可选项)，如果你是国内用户，建议使用国内 Gluon 镜像加速数据集和预训练模型的下载。Linux/macOS 用户可以运行下面命令。

```
MXNET_GLUON_REPO=https://apache-mxnet.s3.cn-north-1.amazonaws.com.cn/jupyter_notebook
```

Windows 用户可以运行下面命令。

```
set MXNET_GLUON_REPO=https://apache-mxnet.s3.cn-north-1.amazonaws.com.cn/jupyter_notebook
```

## 选项二：通过 Docker 安装

第一步，下载并安装Docker。

如果你是 Linux 用户，可以运行下面命令。之后登出一次。

```
wget -qO- https://get.docker.com/ | sh  
sudo usermod -aG docker
```

第二步，运行下面命令。

```
docker run -p 8888:8888 muli/gluon-tutorials-zh
```

第三步，在浏览器打开 <http://localhost:8888>，这时通常需要填 Docker 运行时产生的 token。

### 2.2.2 更新代码和运行环境

目前我们仍然一直在快速更新教程，通常每周都会加入新的章节。同时 MXNet 的 Gluon 前端也在快速发展，因此我们推荐大家也做及时的更新。更新包括下载最新的教程，和更新对应的依赖（通常是升级 MXNet）。

由于 MXNet 在快速发展中，我们会根据改进的 MXNet 版本定期更新书中的代码。同时，我们也会不断补充新的教学内容，以适应深度学习的快速发展。因此，我们推荐大家定期更新代码和运行环境。以下列举了几种更新选项。

#### 选项一：通过 Conda 更新（推荐）

第一步，重新下载最新的包含本书全部代码的包，解压后进入文件夹。下载地址可以从以下二者之间选择。

- [https://zh.gluon.ai/gluon\\_tutorials\\_zh.zip](https://zh.gluon.ai/gluon_tutorials_zh.zip)
- [https://zh.gluon.ai/gluon\\_tutorials\\_zh.tar.gz](https://zh.gluon.ai/gluon_tutorials_zh.tar.gz)

第二步，使用下面命令更新运行环境。

```
conda env update -f environment.yml
```

### 选项二：通过 Docker 更新

我们可以直接下载新的 Docker image，例如执行下面的命令。

```
docker pull mli/gluon-tutorials-zh
```

### 选项三：通过 Git 更新

第一步，如果你熟悉 Git 操作，可以直接 pull 并且合并可能造成的冲突：

```
git pull https://github.com/mli/gluon-tutorials-zh
```

如果不想造成冲突，在保存完有价值的本地修改以后，你可以在 pull 前先用 reset 还原到上次更新的版本：

```
git reset --hard
```

第二步，使用下面命令更新运行环境。

```
conda env update -f environment.yml
```

## 2.2.3 高级选项

以下针对不同的使用场景列举了一些安装和使用上的可选项。如果它们和你无关，请放心忽略。

### 使用 GPU

通过上述方式安装的 MXNet 只支持 CPU。本书中有部分章节需要或推荐使用 GPU 来运行。假设电脑有 Nvidia 显卡并且安装了 CUDA7.5、8.0 或 9.0，那么先卸载 CPU 版本：

```
pip uninstall mxnet
```

然后，根据电脑上安装的 CUDA 版本，使用以下三者之一安装相应的 GPU 版 MXNet。

```
pip install --pre mxnet-cu75 # CUDA 7.5  
pip install --pre mxnet-cu80 # CUDA 8.0  
pip install --pre mxnet-cu90 # CUDA 9.0
```

我们建议国内用户使用豆瓣 pypi 镜像加速下载。以 mxnet-cu80 为例，我们可以使用如下命令。

```
pip install --pre mxnet-cu80 -i https://pypi.douban.com/simple # CUDA 8.0
```

需要注意的是，如果你安装 GPU 版的 MXNet，使用 conda update 命令不会自动升级 GPU 版的 MXNet。这时候可以运行了 source activate gluon 后手动更新 MXNet。以 mxnet-cu80 为例，我们可以使用以下命令手动更新 MXNet。

```
pip install --pre mxnet-cu80 # CUDA 8.0
```

## 用 Jupyter Notebook 读写 GitHub 源文件

如果你希望为本书内容做贡献，需要修改在 GitHub 上 Markdown 格式的源文件（.md 文件非.ipynb 文件）。通过 notedown 插件，我们就可以使用 Jupyter Notebook 修改并运行 Markdown 格式的源代码。Linux/macOS 用户可以执行以下命令获得 GitHub 源文件并激活运行环境。

```
git clone https://github.com/mli/gluon-tutorials-zh  
cd gluon-tutorials-zh  
conda env create -f environment.yml  
source activate gluon # Windows 用户运行 activate gluon
```

下面安装 notedown 插件，运行 Jupyter Notebook 并加载插件：

```
pip install https://github.com/mli/notedown/tarball/master  
jupyter notebook --NotebookApp.contents_manager_class='notedown.NotedownContentsManager'  
→ !
```

如果你希望每次运行 Jupyter Notebook 时默认开启 notedown 插件，可以参考下面步骤。

首先，执行下面命令生成 Jupyter Notebook 配置文件（如果已经生成可以跳过）。

```
jupyter notebook --generate-config
```

然后，将下面这一行加入到 Jupyter Notebook 配置文件的末尾（Linux/macOS 上一般在 `~/.jupyter/jupyter_notebook_config.py`）

```
c.NotebookApp.contents_manager_class = 'notedown.NotedownContentsManager'
```

之后，我们只需要运行 `jupyter notebook` 即可默认开启 notedown 插件。

### 在远端服务器上运行 Jupyter Notebook

有时候，我们希望在远端服务器上运行 Jupyter Notebook，并通过本地电脑上的浏览器访问。如果本地机器上安装了 Linux 或者 macOS（Windows 通过第三方软件例如 putty 应该也能支持），那么可以使用端口映射：

```
ssh myserver -L 8888:localhost:8888
```

以上 `myserver` 是远端服务器地址。然后我们可以使用 `http://localhost:8888` 打开远端服务器 `myserver` 上运行 Jupyter Notebook。

### 运行计时

我们可以通过 ExecutionTime 插件来对 Jupyter Notebook 的每个代码单元的运行计时。以下是安装该插件的命令。

```
pip install jupyter_contrib_nbextensions  
jupyter contrib nbextension install --user  
jupyter nbextension enable execute_time/ExecuteTime
```

## 2.2.4 小结

- 为了能够动手学深度学习，我们需要获取本书代码并安装运行环境。
- 我们建议大家定期更新代码和运行环境。

## 2.2.5 练习

- 获取本书代码并安装运行环境。如果你在安装时碰到任何问题，请查阅讨论区中的疑难问题汇总，或者向社区小伙伴们提问。

## 2.2.6 扫码直达讨论区



## 2.3 数据操作

在深度学习中，我们通常会频繁地对数据进行操作。作为动手学深度学习的基础，本节将介绍如何对内存中的数据进行操作。

在 MXNet 中，NDArray 是存储和变换数据的主要工具。如果你之前用过 NumPy，你会发现 NDArray 和 NumPy 的多维数组非常类似。然而，NDArray 提供更多的功能，例如 CPU 和 GPU 的异步计算，以及自动求导。这些都使得 NDArray 更加适合深度学习。

### 2.3.1 创建 NDArray

我们先介绍 NDArray 的最基本功能。如果你对我们用到的数学操作不是很熟悉，可以参阅“[数学基础](#)”一节。

首先从 MXNet 导入 NDArray。

```
In [1]: from mxnet import nd
```

然后我们用 NDArray 创建一个行向量。

```
In [2]: x = nd.arange(12)
x
```

```
Out[2]:
```

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.]  
<NDArray 12 @cpu(0)>
```

以上创建的 NDArray 一共包含 12 个元素 (element)，分别为 `arange(12)` 所指定的从 0 开始的 12 个连续整数。可以看到，打印的 `x` 中还标注了属性 `<NDArray 12 @cpu(0)>`。其中，12 指的是 NDArray 的形状，即向量的长度；而 `@cpu(0)` 说明默认情况下 NDArray 被创建在 CPU 上。

下面使用 `reshape` 函数把向量 `x` 的形状改为  $(3, 4)$ ，也就是一个 3 行 4 列的矩阵。除了形状改变之外，`x` 中的元素保持不变。

```
In [3]: x = x.reshape((3, 4))  
x
```

```
Out[3]:
```

```
[[ 0.  1.  2.  3.]  
 [ 4.  5.  6.  7.]  
 [ 8.  9.  10. 11.]]  
<NDArray 3x4 @cpu(0)>
```

上面 `x.reshape((3, 4))` 也可写成 `x.reshape((-1, 4))` 或 `x.reshape((3, -1))`。由于 `x` 的元素个数是已知的，这里的-1 是能够通过元素个数和其他维的大小推断出来的。

接下来，我们创建一个各元素为 0，形状为  $(2, 3, 4)$  的张量。实际上，之前创建的向量和矩阵都是特殊的张量。

```
In [4]: nd.zeros((2, 3, 4))
```

```
Out[4]:
```

```
[[[ 0.  0.  0.  0.]  
 [ 0.  0.  0.  0.]  
 [ 0.  0.  0.  0.]]  
  
[[ 0.  0.  0.  0.]  
 [ 0.  0.  0.  0.]  
 [ 0.  0.  0.  0.]]]  
<NDArray 2x3x4 @cpu(0)>
```

类似地，我们可以创建各元素为 1 的张量。

```
In [5]: nd.ones((3, 4))
```

```
Out[5]:
```

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

```
[ 1.  1.  1.  1.]]  
<NDArray 3x4 @cpu(0)>
```

我们也可以通过 Python 的列表 (list) 指定需要创建的 NDArray 中每个元素的值。

```
In [6]: y = nd.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
y
```

```
Out[6]:
```

```
[[ 2.  1.  4.  3.]  
 [ 1.  2.  3.  4.]  
 [ 4.  3.  2.  1.]]  
<NDArray 3x4 @cpu(0)>
```

有些情况下，我们需要随机生成 NDArray 中每个元素的值。下面我们创建一个形状为 (3, 4) 的 NDArray。它的每个元素都随机采样于均值为 0 标准差为 1 的正态分布。

```
In [7]: nd.random.normal(0, 1, shape=(3, 4))
```

```
Out[7]:
```

```
[[ 2.21220636  0.7740038   1.04344046  1.18392551]  
 [ 1.89171135 -1.23474145 -1.771029    -0.45138445]  
 [ 0.57938355 -1.85608196 -1.9768796   -0.20801921]]  
<NDArray 3x4 @cpu(0)>
```

每个 NDArray 的形状可以通过 `shape` 属性来获取。

```
In [8]: y.shape
```

```
Out[8]: (3, 4)
```

一个 NDArray 的大小 (size) 即其元素的总数。

```
In [9]: y.size
```

```
Out[9]: 12
```

### 2.3.2 运算

NDArray 支持大量的运算符 (operator)。例如，我们可以对之前创建的两个形状为 (3, 4) 的 NDArray 做按元素加法。所得结果形状不变。

```
In [10]: x + y
```

```
Out[10]:
```

```
[[ 2.  2.  6.  6.]  
 [ 5.  7.  9.  11.]]
```

```
[ 12. 12. 12. 12.]]  
<NDArray 3x4 @cpu(0)>
```

以下是按元素乘法。

```
In [11]: x * y
```

```
Out[11]:
```

```
[[ 0. 1. 8. 9.]  
 [ 4. 10. 18. 28.]  
 [ 32. 27. 20. 11.]]  
<NDArray 3x4 @cpu(0)>
```

以下是按元素除法。

```
In [12]: x / y
```

```
Out[12]:
```

```
[[ 0. 1. 0.5 1. ]  
 [ 4. 2.5 2. 1.75]  
 [ 2. 3. 5. 11. ]]  
<NDArray 3x4 @cpu(0)>
```

以下分别基于最外层和最里层连结两个矩阵。

```
In [13]: nd.concat(x, y, dim=0), nd.concat(x, y, dim=1)
```

```
Out[13]: (
```

```
[[ 0. 1. 2. 3.]  
 [ 4. 5. 6. 7.]  
 [ 8. 9. 10. 11.]  
 [ 2. 1. 4. 3.]  
 [ 1. 2. 3. 4.]  
 [ 4. 3. 2. 1.]]  
<NDArray 6x4 @cpu(0)>,  
[[ 0. 1. 2. 3. 2. 1. 4. 3.]  
 [ 4. 5. 6. 7. 1. 2. 3. 4.]  
 [ 8. 9. 10. 11. 4. 3. 2. 1.]]  
<NDArray 3x8 @cpu(0)>)
```

以下是按元素做指数运算。

```
In [14]: y.exp()
```

```
Out[14]:
```

```
[[ 7.38905621 2.71828175 54.59814835 20.08553696]  
 [ 2.71828175 7.38905621 20.08553696 54.59814835]
```

```
[ 54.59814835  20.08553696   7.38905621   2.71828175]  
<NDArray 3x4 @cpu(0)>
```

我们也可以使用条件判断式来得到元素为 0 或 1 的新的 NDArray。以  $x == y$  为例，如果  $x$  和  $y$  在相同位置的条件判断为真（值相等），那么新 NDArray 在相同位置的值为 1；反之为 0。

In [15]:  $x == y$

Out[15]:

```
[[ 0.  1.  0.  1.]  
 [ 0.  0.  0.  0.]  
 [ 0.  0.  0.  0.]]  
<NDArray 3x4 @cpu(0)>
```

接下来，我们对矩阵  $y$  做转置，并做矩阵乘法操作。由于  $x$  是 3 行 4 列的矩阵， $y$  转置为 4 行 3 列的矩阵，两个矩阵相乘得到 3 行 3 列的矩阵。

In [16]:  $nd.dot(x, y.T)$

Out[16]:

```
[[ 18.  20.  10.]  
 [ 58.  60.  50.]  
 [ 98.  100.  90.]]  
<NDArray 3x3 @cpu(0)>
```

下面，我们对 NDArray 中的元素求和。结果虽然是个标量，却依然保留了 NDArray 格式。

In [17]:  $x = nd.array([3, 4])$   
 $x.sum()$

Out[17]:

```
[ 7.]  
<NDArray 1 @cpu(0)>
```

其实，我们可以把为标量的 NDArray 通过 `asscalar` 函数直接变换为 Python 中的数。下面例子中  $x$  的  $L_2$  范数不再是一个 NDArray。

In [18]:  $x.norm().asscalar()$

Out[18]: 5.0

以上  $y.exp()$ 、 $x.sum()$  和  $x.norm()$  也可分别写作  $nd.exp(y)$ 、 $nd.sum(x)$  和  $nd.norm(x)$ 。

### 2.3.3 广播机制

正如我们所见，我们可以对两个形状相同的 NDArray 做按元素操作。然而，当我们对两个形状不同的 NDArray 做按元素操作时，可能会触发广播（broadcasting）机制：先令这两个 NDArray 形状相同再按元素操作。

让我们先看个例子。

```
In [19]: a = nd.arange(3).reshape((3, 1))
b = nd.arange(2).reshape((1, 2))
a, b, a+b
```

```
Out[19]: (
    [[ 0.]
     [ 1.]
     [ 2.]]
    <NDArray 3x1 @cpu(0)>,
    [[ 0.  1.]]
    <NDArray 1x2 @cpu(0)>,
    [[ 0.  1.]
     [ 1.  2.]
     [ 2.  3.]]
    <NDArray 3x2 @cpu(0)>)
```

由于 `a` 和 `b` 分别是 3 行 1 列和 1 行 2 列的矩阵，为了使它们可以按元素相加，计算时 `a` 中第一列的三个元素被广播（复制）到了第二列，而 `b` 中第一行的两个元素被广播（复制）到了第二行和第三行。如此，我们就可以对两个 3 行 2 列的矩阵按元素相加，得到上面的结果。

### 2.3.4 运算的内存开销

在前面的例子中，我们为每个操作新开内存来存储它的结果。举个例子，假设 `x` 和 `y` 都是 NDArray，在执行 `y = x + y` 操作后，`y` 所对应的内存地址将变成为存储 `x + y` 计算结果而新开内存的地址。为了展示这一点，我们可以使用 Python 自带的 `id` 函数：如果两个实例的 ID 一致，它们所对应的内存地址相同；反之则不同。

```
In [20]: x = nd.ones((3, 4))
y = nd.ones((3, 4))
before = id(y)
y = y + x
id(y) == before
```

```
Out[20]: False
```

在下面的例子中，我们先通过 `nd.zeros_like(y)` 创建和 `y` 形状相同且元素为 0 的 `NDArray`，记为 `z`。接下来，我们把 `x + y` 的结果通过 `[:] =` 写进 `z` 所对应的内存中。

```
In [21]: z = nd.zeros_like(y)
```

```
before = id(z)
```

```
z[:] = x + y
```

```
id(z) == before
```

```
Out[21]: True
```

然而，这里我们还是为 `x + y` 创建了临时内存来存储计算结果，再复制到 `z` 所对应的内存。为了避免这个内存开销，我们可以使用运算符的全名函数中的 `out` 参数。

```
In [22]: nd.elemwise_add(x, y, out=z)
```

```
id(z) == before
```

```
Out[22]: True
```

如果现有的 `NDArray` 的值在之后的程序中不会复用，我们也可以用 `x[:] = x + y` 或者 `x += y` 来减少运算的内存开销。

```
In [23]: before = id(x)
```

```
x += y
```

```
id(x) == before
```

```
Out[23]: True
```

## 2.3.5 索引

在 `NDArray` 中，索引（index）代表了元素的位置。`NDArray` 的索引从 0 开始逐一递增。例如一个 3 行 2 列的矩阵的行索引分别为 0、1 和 2，列索引分别为 0 和 1。

在下面的例子中，我们指定了 `NDArray` 的行索引截取范围 `[1:3]`。依据左闭右开指定范围的惯例，它截取了矩阵 `x` 中行索引为 1 和 2 的两行。

```
In [24]: x = nd.arange(9).reshape((3, 3))
```

```
x, x[1:3]
```

```
Out[24]: (
```

```
[[ 0.  1.  2.]
```

```
[ 3.  4.  5.]
```

```
[ 6.  7.  8.]]
```

```
<NDArray 3x3 @cpu(0)>,
```

```
[[ 3.  4.  5.]
```

```
[ 6.  7.  8.]]
```

```
<NDArray 2x3 @cpu(0)>)
```

我们可以指定 NDArray 中需要访问的单个元素的位置，例如矩阵中行和列的索引，并重设该元素的值。

```
In [25]: x[1, 2] = 9
x
Out[25]:
[[ 0.  1.  2.]
 [ 3.  4.  9.]
 [ 6.  7.  8.]]
<NDArray 3x3 @cpu(0)>
```

当然，我们也可以截取一部分元素，并重设它们的值。

```
In [26]: x[1:2, 1:3] = 10
x
Out[26]:
[[ 0.  1.  2.]
 [ 3.  10. 10.]
 [ 6.  7.  8.]]
<NDArray 3x3 @cpu(0)>
```

### 2.3.6 NDArray 和 NumPy 相互变换

我们可以通过 `array` 和 `asnumpy` 函数令数据在 NDArray 和 Numpy 格式之间相互转换。以下是一个例子。

```
In [27]: import numpy as np
x = np.ones((2, 3))
y = nd.array(x) # NumPy 变换成 NDArray。
z = y.asnumpy() # NDArray 变换成 NumPy。
z, y
Out[27]: (array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]], dtype=float32),
 [[ 1.  1.  1.]
 [ 1.  1.  1.]])
<NDArray 2x3 @cpu(0)>)
```

### 2.3.7 小结

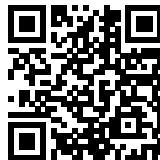
- NDArray 是 MXNet 中存储和变换数据的主要工具。

- 我们可以轻松地对 NDArray 进行创建、运算、指定索引和与 NumPy 之间的相互变换。

### 2.3.8 练习

- 运行本节代码。将本节中条件判断式  $x == y$  改为  $x < y$  或  $x > y$ , 看看能够得到什么样的 NDArray。
- 将广播机制中按元素操作的两个 NDArray 替换成其他形状, 结果是否和预期一样?
- 查阅 MXNet 官方网站上的[文档](#), 了解 NDArray 支持的其他操作。

### 2.3.9 扫码直达讨论区



## 2.4 自动求梯度

在深度学习中, 我们经常需要对函数求梯度 (gradient)。如果你对本节中的数学概念 (例如梯度) 不是很熟悉, 可以参阅 “[数学基础](#)” 一节。

MXNet 提供 `autograd` 包来自动化求梯度的过程。虽然大部分的深度学习框架要求编译计算图来自动求梯度, MXNet 却无需如此。

首先导入本节实验需要的包。

```
In [1]: from mxnet import autograd, nd
```

### 2.4.1 简单例子

我们先看一个简单例子：对函数  $y = 2x^\top x$  求关于列向量  $x$  的梯度。

我们先创建变量  $x$ , 并赋初值。

```
In [2]: x = nd.arange(4).reshape((4, 1))
x
```

Out[2]:

```
[[ 0.]
 [ 1.]
 [ 2.]
 [ 3.]]
<NDArray 4x1 @cpu(0)>
```

为了求有关变量  $x$  的梯度，我们需要先调用 `attach_grad` 函数来申请存储梯度所需要的内存。

```
In [3]: x.attach_grad()
```

下面定义有关变量  $x$  的函数。默认条件下，为了减少计算和内存开销，MXNet 不会记录用于求梯度的计算图。我们需要调用 `record` 函数来要求 MXNet 记录与求梯度有关的计算。

```
In [4]: with autograd.record():
    y = 2 * nd.dot(x.T, x)
```

由于  $x$  的形状为  $(4, 1)$ ， $y$  是一个标量。接下来我们可以通过调用 `backward` 函数自动求梯度。需要注意的是，如果  $y$  不是一个标量，MXNet 将先对  $y$  中元素求和得到新的变量，再求该变量有关  $x$  的梯度。

```
In [5]: y.backward()
```

函数  $y = 2x^T x$  关于  $x$  的梯度应为  $4x$ 。现在我们来验证一下求出来的梯度是正确的。

```
In [6]: x.grad, x.grad == 4*x # 1 为真, 0 为假。
```

Out[6]:

```
([[ 0.]
 [ 4.]
 [ 8.]
 [12.]]
<NDArray 4x1 @cpu(0)>,
[[ 1.]
 [ 1.]
 [ 1.]
 [ 1.]]
<NDArray 4x1 @cpu(0)>)
```

## 2.4.2 对 Python 控制流求梯度

使用 MXNet 的一个便利之处是，即使函数的计算图包含了 Python 的控制流（例如条件和循环控制），我们也有可能对变量求梯度。

考虑下面程序，其中包含 Python 的条件和循环控制。需要强调的是，这里循环（while 循环）迭代的次数和条件判断（if 语句）的执行都取决于输入  $b$  的值。由于不同的输入会导致计算图不同，我们有时把这类计算图称作动态图。

```
In [7]: def f(a):
    b = a * 2
    while b.norm().asscalar() < 1000:
        b = b * 2
    if b.sum().asscalar() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

我们依然跟之前一样使用 `record` 函数记录计算图，并调用 `backward` 函数求梯度。

```
In [8]: a = nd.random.normal(shape=1)
a.attach_grad()
with autograd.record():
    c = f(a)
c.backward()
```

让我们仔细观察上面定义的  $f$  函数。事实上，给定任意输入  $a$ ，其输出必然是  $f(a) = xa$  的形式，且标量系数  $x$  的值取决于输入  $a$ 。由于  $c$  有关  $a$  的梯度为  $x = c/a$ ，我们可以像下面这样验证对本例中控制流求梯度的结果是正确的。

```
In [9]: a.grad == c / a
Out[9]:
[ 1.]
<NDArray 1 @cpu(0)>
```

## 2.4.3 小结

- MXNet 提供 `autograd` 包来自动化求导过程。
- MXNet 的 `autograd` 包可以对正常的命令式程序进行求导。

#### 2.4.4 练习

- 在本节对控制流求梯度的例子中，把变量  $a$  改成一个随机向量或矩阵。此时计算结果  $c$  不再是标量，运行结果将有何变化？该如何分析此结果？
- 自己重新设计一个对控制流求梯度的例子。运行并分析结果。

#### 2.4.5 扫码直达讨论区





---

## 深度学习基础

---

从本章开始，我们将一起探索深度学习的奥秘。我们先以线性回归和 Softmax 回归为例，介绍机器学习的基础知识。接着，我们由多层感知机引入深度学习模型。在观察并了解模型过拟合现象后，我们将描述深度学习中应对过拟合的常用方法：正则化和丢弃法。为了进一步理解深度学习模型训练的本质，我们将详细解释正向传播和反向传播。最后，我们将通过一个深度学习应用案例来实践本章学习的内容。

### 3.1 单层神经网络

在本章的前几节，让我们重温一些经典的浅层模型，例如线性回归和 Softmax 回归。前者适用于回归问题：模型最终输出是一个连续值，例如房价；后者适用于分类问题：模型最终输出是一个离散值，例如图片的类别。这两种浅层模型本质上都是单层神经网络。它们涉及到的概念和技术对大多数深度学习模型来说同样适用。

本节中，我们以线性回归为例，介绍大多数深度学习模型的基本要素和表示方法。

### 3.1.1 线性回归的基本要素

为了简单起见，我们先从一个具体案例来解释线性回归的基本要素。

#### 模型

给定一个有关房屋的数据集，其中每栋房屋的相关数据包括面积（平方米）、房龄（年）和价格（元）。假设我们想使用任意一栋房屋的面积（设  $x_1$ ）和房龄（设  $x_2$ ）来估算它的真实价格（设  $y$ ）。那么  $x_1$  和  $x_2$  即每栋房屋的特征（feature）， $y$  为标签（label）或真实值（ground truth）。在线性回归模型中，房屋估计价格（设  $\hat{y}$ ）的表达式为

$$\hat{y} = x_1 w_1 + x_2 w_2 + b,$$

其中  $w_1, w_2$  是权重（weight），通常用向量  $w = [w_1, w_2]^\top$  来表示； $b$  是偏差（bias）。这里的权重和偏差是线性回归模型的参数（parameter）。接下来，让我们了解一下如何通过训练模型来学习模型参数。

#### 训练数据

假设我们使用上文所提到的房屋数据集训练模型，该数据集即训练数据集（training data set）。

在训练数据集中，一栋房屋的特征和标签即为一个数据样本。设训练数据集样本数为  $n$ ，索引为  $i$  的样本的特征为  $x_1^{(i)}, x_2^{(i)}$ ，标签为  $y^{(i)}$ 。对于索引为  $i$  的房屋，线性回归模型的价格估算表达式为

$$\hat{y}^{(i)} = x_1^{(i)} w_1 + x_2^{(i)} w_2 + b.$$

#### 损失函数

在模型训练中，我们希望模型的估计值和真实值在训练数据集上尽可能接近。用平方损失（square loss）来定义数据样本  $i$  上的损失（loss）为

$$\ell^{(i)}(w_1, w_2, b) = \frac{(\hat{y}^{(i)} - y^{(i)})^2}{2},$$

当该损失越小时，模型在数据样本  $i$  上的估计值和真实值越接近。已知训练数据集样本数为  $n$ 。线性回归的目标是找到一组模型参数  $w_1, w_2, b$  来最小化损失函数

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

在上式中，损失函数  $\ell(w_1, w_2, b)$  可看作是训练数据集中各个样本上损失的平均。

## 优化算法

虽然线性回归中我们可通过微分最小化损失函数，对大多数深度学习模型来说，我们需要使用优化算法并通过有限次迭代模型参数来最小化损失函数。一种常用的优化算法叫做小批量随机梯度下降（mini-batch stochastic gradient descent）。每一次迭代前，我们可以随机均匀采样一个由训练数据样本索引所组成的小批量（mini-batch） $\mathcal{B}$ ；然后求小批量中数据样本平均损失有关模型参数的导数（梯度）；用此结果与人为设定的正数的乘积作为模型参数在本次迭代的减小量。在本节讨论的线性回归模型中，模型的每个参数将迭代如下：

$$\begin{aligned}w_1 &\leftarrow w_1 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial w_1} = w_1 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} x_1^{(i)}(x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)}), \\w_2 &\leftarrow w_2 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial w_2} = w_2 - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} x_2^{(i)}(x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)}), \\b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial b} = b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} (x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)}).\end{aligned}$$

在上式中， $|\mathcal{B}|$  代表每个小批量中的样本个数（批量大小，batch size）， $\eta$  称作学习率（learning rate）并取正数。需要强调的是，这里的批量大小和学习率的值是人为设定的，并不需要通过模型训练学出，也叫做超参数（hyperparameter）。我们通常所说的“调参”指的正是调节超参数。

我们将在后面“优化算法”一章中详细解释小批量随机梯度下降和其他优化算法。在模型训练中，我们会使用优化算法迭代模型参数若干次。之后便学出了模型参数值  $w_1, w_2, b$ 。这时，我们就可以使用学出的线性回归模型  $x_1w_1 + x_2w_2 + b$  来估算训练数据集以外任意一栋面积（平方米）为  $x_1$  且房龄（年）为  $x_2$  的房屋的价格了。这也叫做模型测试、模型预测或模型推断。

### 3.1.2 线性回归的表示方法

我们继续以上文中的房屋数据集和线性回归模型为例，介绍线性回归的表示方法。

#### 神经网络图

在深度学习中，我们可以使用神经网络图直观地表现模型结构。为了更清晰地展示线性回归作为神经网络的结构，图 3.1 使用神经网络图表示本节中介绍的线性回归模型。

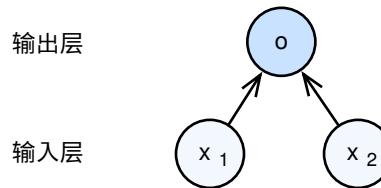


图 3.1: 线性回归是一个单层神经网络

在图 3.1 所表示的神经网络中，输入分别为  $x_1$  和  $x_2$ ，因此输入层的输入个数为 2。输入个数也叫特征数或特征向量维度。由于图 3.1 中网络的输出为  $o$ ，输出层的输出个数为 1。需要注意的是，我们直接将图 3.1 中神经网络的输出  $o$  作为线性回归的输出，即  $\hat{y} = o$ 。由于输入层并不涉及计算，按照惯例，图 3.1 所示的神经网络的层数为 1。所以，线性回归是一个单层神经网络。输出层中负责计算  $o$  的单元又叫神经元。在线性回归中， $o$  的计算依赖于  $x_1$  和  $x_2$ 。也就是说，输出层中的神经元和输入层中各个输入完全连接。因此，这里的输出层又叫全连接层或稠密层（fully-connected layer 或 dense layer）。

## 矢量计算表达式

当训练或推断模型时，我们常常会同时处理多个数据样本并用到矢量计算。在介绍线性回归的矢量计算表达式之前，让我们先考虑对两个向量相加的两种方法。

下面先定义两个 1000 维的向量。

```
In [1]: from mxnet import nd
        from time import time

a = nd.ones(shape=1000)
b = nd.ones(shape=1000)
```

向量相加的一种方法是，将这两个向量按元素逐一做标量加法：

```
In [2]: start = time()
        c = nd.zeros(shape=1000)
        for i in range(1000):
            c[i] = a[i] + b[i]
        time() - start

Out[2]: 0.0959923267364502
```

向量相加的另一种方法是，将这两个向量直接做矢量加法：

```
In [3]: start = time()
d = a + b
time() - start

Out[3]: 0.00024819374084472656
```

结果很明显，后者比前者更省时。因此，在深度学习中我们应该尽可能采用矢量计算，以提升计算效率。

让我们再次回到本节的房价估算问题。如果我们将训练数据集中 3 个房屋样本（索引分别为 1、2 和 3）逐一估算价格，将得到

$$\begin{aligned}\hat{y}^{(1)} &= x_1^{(1)}w_1 + x_2^{(1)}w_2 + b, \\ \hat{y}^{(2)} &= x_1^{(2)}w_1 + x_2^{(2)}w_2 + b, \\ \hat{y}^{(3)} &= x_1^{(3)}w_1 + x_2^{(3)}w_2 + b.\end{aligned}$$

现在，我们将上面三个等式转化成矢量计算。设

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \hat{y}^{(3)} \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ x_1^{(3)} & x_2^{(3)} \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix},$$

对 3 个房屋样本估算价格的矢量计算表达式为

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

其中的加法运算使用了广播机制（参见“数据操作”一节）。例如

```
In [4]: a = np.ones(shape=3)
b = 10
a + b
```

```
Out[4]:
[ 11.  11.  11.]
<NDArray 3 @cpu(0)>
```

广义上，当数据样本数为  $n$ ，特征数为  $d$ ，线性回归的矢量计算表达式为

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

其中模型输出  $\hat{\mathbf{y}} \in \mathbb{R}^{n \times 1}$ ，批量数据样本特征  $\mathbf{X} \in \mathbb{R}^{n \times d}$ ，权重  $\mathbf{w} \in \mathbb{R}^{d \times 1}$ ，偏差  $b \in \mathbb{R}$ 。相应地，批量数据样本标签  $\mathbf{y} \in \mathbb{R}^{n \times 1}$ 。在矢量计算中，我们将两个向量  $\hat{\mathbf{y}}$  和  $\mathbf{y}$  作为损失函数的输入。我们将在下一节介绍线性回归矢量计算的实现。

同理，我们也可以在模型训练中对优化算法做矢量计算。设模型参数  $\theta = [w_1, w_2, b]^\top$ ，本节中小批量随机梯度下降的迭代步骤将相应地改写为

$$\theta \leftarrow \theta - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta \ell^{(i)}(\theta),$$

其中梯度是损失有关三个标量模型参数的偏导数组成的向量：

$$\nabla_\theta \ell^{(i)}(\theta) = \begin{bmatrix} \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial w_1} \\ \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial w_2} \\ \frac{\partial \ell^{(i)}(w_1, w_2, b)}{\partial b} \end{bmatrix} = \begin{bmatrix} x_1^{(i)}(x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)}) \\ x_2^{(i)}(x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)}) \\ x_1^{(i)}w_1 + x_2^{(i)}w_2 + b - y^{(i)} \end{bmatrix}.$$

### 3.1.3 小结

- 和大多数深度学习模型一样，对于线性回归这样一个单层神经网络，它的基本要素包括模型、训练数据、损失函数和优化算法。
- 我们既可以用神经网络图表示线性回归，又可以用矢量计算表示该模型。
- 在深度学习中我们应该尽可能采用矢量计算，以提升计算效率。

### 3.1.4 练习

- 使用其他包（例如 NumPy）或其他编程语言（例如 MATLAB），比较相加两个向量的两种方法的运行时间。

### 3.1.5 扫码直达讨论区



## 3.2 线性回归——从零开始

在了解了线性回归的背景知识之后，现在我们可以动手实现它了。尽管强大的深度学习框架可以减少大量重复性工作，但若过于依赖它提供的便利，我们就会很难深入理解深度学习是如何工作的。因此，本节将介绍如何只利用 NDArray 和 autograd 来实现一个线性回归的训练。

### 3.2.1 线性回归

让我们先回忆一下上节中的内容。设数据样本数为  $n$ ，输入个数为  $d$ 。给定批量数据样本的特征  $\mathbf{X} \in \mathbb{R}^{n \times d}$  和标签  $\mathbf{y} \in \mathbb{R}^{n \times 1}$ ，线性回归的批量输出  $\hat{\mathbf{y}} \in \mathbb{R}^{n \times 1}$  的计算表达式为

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

其中  $\mathbf{w} \in \mathbb{R}^{d \times 1}$  和  $b \in \mathbb{R}$  分别为线性回归的模型参数：权重和偏差。为了学习权重和偏差，我们用预测值  $\hat{\mathbf{y}}$  和真实值  $\mathbf{y}$  之间的平方损失作为模型的损失函数。在模型训练过程中，我们使用小批量随机梯度下降不断迭代模型参数的值，以最小化损失函数。最终，在有限次迭代后，我们便学出了模型参数的值。

下面我们开始动手实现线性回归的训练。首先，导入本节中实验所需的包或模块。

```
In [1]: from matplotlib import pyplot as plt
        from mxnet import autograd, nd
        import random
```

### 3.2.2 生成数据集

我们在这里描述用来生成人工训练数据集的真实模型。

设训练数据集样本数为 1000，输入个数为 2。给定随机生成的批量样本特征  $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$ ，我们使用线性回归模型真实权重  $\mathbf{w} = [2, -3.4]^\top$  和偏差  $b = 4.2$ ，以及一个随机噪音项  $\epsilon$  来生成标签

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon,$$

其中噪音项  $\epsilon$  服从均值为 0 和标准差为 0.01 的正态分布。下面，让我们生成数据集。

```
In [2]: num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
```

```
features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)
```

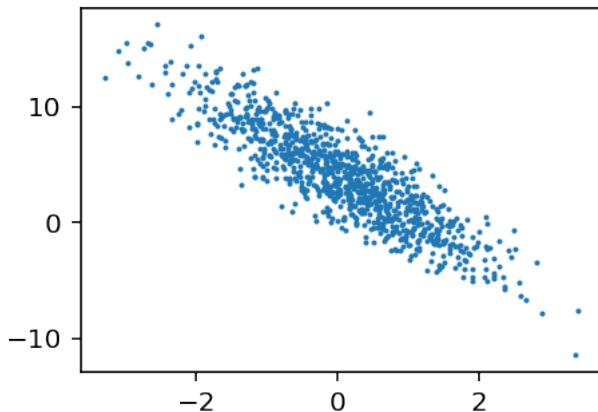
注意到 `features` 的每一行是一个长度为 2 的向量，而 `labels` 的每一行是一个长度为 1 的向量（标量）。

```
In [3]: features[0], labels[0]
```

```
Out[3]: (
    [ 2.21220636  0.7740038 ]
    <NDArray 2 @cpu(0)>,
    [ 6.00058699]
    <NDArray 1 @cpu(0)>)
```

通过生成第二个特征 `features[:, 1]` 和标签 `labels` 的散点图，我们可以更直观地观察两者间的线性关系。

```
In [4]: %config InlineBackend.figure_format = 'retina'
plt.rcParams['figure.figsize'] = (3.5, 2.5)
plt.scatter(features[:, 1].asnumpy(), labels.asnumpy(), 1)
plt.show()
```



我们将上面的 `plt` 作图函数定义在 `gluonbook` 包里。以后在作图时，我们将直接调用 `gluonbook.plt`，而无需执行 `from matplotlib import pyplot as plt`。

### 3.2.3 读取数据

在训练模型的时候，我们需要遍历数据集并不断读取小批量数据样本。这里我们定义一个函数：它每次返回 `batch_size` 个随机样本的特征和标签。设批量大小 (`batch_size`) 为 10。

```
In [5]: batch_size = 10
def data_iter(batch_size, num_examples, features, labels):
    indices = list(range(num_examples))
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        j = nd.array(indices[i: min(i + batch_size, num_examples)])
        yield features.take(j), labels.take(j)
```

让我们读取第一个小批量数据样本并打印。每个批量的特征形状为 `(10, 2)`，分别对应批量大小 `batch_size` 和输入个数 `num_inputs`；标签形状为 `10`，也就是批量大小。

```
In [6]: for X, y in data_iter(batch_size, num_examples, features, labels):
    print(X, y)
    break
```

```
[[ -1.27275753  0.77353251]
 [ 1.04665864  1.30097735]
 [-0.52480155  0.3005414 ]
 [ 0.03365511  0.61478573]
 [ 0.14459428 -0.28338912]
 [ 0.86821365  1.3929764 ]
 [-0.11068931 -1.35708642]
 [ 1.41386247  0.80119449]
 [-0.01991243 -0.48885924]
 [ 1.50098407 -1.20090294]]
<NDArray 10x2 @cpu(0)>
[ -0.98361695   1.87119663   2.13838172   2.17679954   5.44580126
  1.18035686   8.59291267   4.29428959   5.82623625  11.27250099]
<NDArray 10 @cpu(0)>
```

我们将 `data_iter` 函数定义在 `gluonbook` 包中供后面章节调用。

### 3.2.4 初始化模型参数

下面我们随机初始化模型参数。

```
In [7]: w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
b = nd.zeros(shape=(1,))
```

```
params = [w, b]
```

之后训练时我们需要对这些参数求梯度来迭代它们的值，以使损失函数不断减小。因此我们需要创建它们的梯度。

```
In [8]: for param in params:  
    param.attach_grad()
```

### 3.2.5 定义模型

下面是线性回归的矢量计算表达式的实现。我们使用 `nd.dot` 函数做矩阵乘法。

```
In [9]: def linreg(X, w, b):  
    return nd.dot(X, w) + b
```

### 3.2.6 定义损失函数

我们使用上一节描述的平方损失来定义线性回归的损失函数。在实现中，我们需要把真实值 `y` 变形成预测值 `y_hat` 的形状。以下函数返回的结果也将和 `y_hat` 的形状相同。

```
In [10]: def squared_loss(y_hat, y):  
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

我们将 `linreg` 和 `squared_loss` 函数定义在 `gluonbook` 包中供后面章节调用。

### 3.2.7 定义优化算法

以下的 `sgd` 函数实现了上一节中介绍的小批量随机梯度下降算法。这是我们最小化损失函数所需要的优化算法。

```
In [11]: def sgd(params, lr, batch_size):  
    for param in params:  
        param[:] = param - lr * param.grad / batch_size
```

我们将该函数定义在 `gluonbook` 包中供后面章节调用。

### 3.2.8 训练模型

现在我们可以开始训练模型了。在训练中，我们将有限次地迭代模型参数。在每次迭代中，我们根据当前读取的小批量数据样本（特征 `features` 和标签 `label`），通过调用反向函数 `backward`

计算小批量随机梯度，并调用优化算法 `sgd` 迭代模型参数。在一个迭代周期（epoch）中，我们将完整遍历一遍 `data_iter` 函数，并对训练数据集中所有样本都使用一次。这里的迭代周期数 `num_epochs` 和学习率 `lr` 都是超参数，分别设 3 和 0.03。在实践中，大多超参数都是需要通过反复试错来不断调节。当迭代周期数设的越大时，虽然模型可能更有效，但是训练时间可能过长。而有关学习率对模型的影响，我们会在后面“优化算法”一章中详细介绍。

```
In [12]: lr = 0.03
        num_epochs = 3
        net = linreg
        loss = squared_loss

        for epoch in range(1, num_epochs + 1):
            for X, y in data_iter(batch_size, num_examples, features, labels):
                with autograd.record():
                    l = loss(net(X, w, b), y)
                    l.backward()
                    sgd([w, b], lr, batch_size)
            print("epoch %d, loss %f"
                  % (epoch, loss(net(features, w, b), labels).mean().asnumpy()))

epoch 1, loss 0.040579
epoch 2, loss 0.000156
epoch 3, loss 0.000051
```

训练完成后，我们可以比较学到的参数和真实参数。它们应该很接近。

```
In [13]: true_w, w
Out[13]: ([2, -3.4],
           [[ 1.9997896 ]
            [-3.40016294]]
           <NDArray 2x1 @cpu(0)>

In [14]: true_b, b
Out[14]: (4.2,
           [ 4.19906378]
           <NDArray 1 @cpu(0)>)
```

### 3.2.9 小结

- 我们现在看到，仅使用 `NDArray` 和 `autograd` 就可以很容易地实现一个模型。在接下来的章节中，我们会在此基础上描述更多深度学习模型，并介绍怎样使用更简洁的代码（例如下一节）实现它们。

### 3.2.10 练习

- 尝试用不同的学习率查看损失函数值的下降速度。
- 回顾“自动求梯度”一节。本节代码中变量 `l` 并不是一个标量，运行 `l.backward()` 将如何对模型参数求梯度？

### 3.2.11 扫码直达讨论区



## 3.3 线性回归——使用 Gluon

随着深度学习框架的发展，开发深度学习应用变得越来越便利。实践中，我们通常可以用比上一节中更简洁的代码来实现相同模型。本节中，我们将介绍如何使用 MXNet 提供的 Gluon 接口更方便地实现线性回归的训练。

首先，导入本节中实验所需的一部分包或模块。我们在之前的章节里使用过它们。

```
In [1]: from mxnet import autograd, nd
```

### 3.3.1 生成数据集

我们生成与上一节中相同的数据集。其中 `X` 是训练数据特征，`y` 是标签。

```
In [2]: num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)
```

### 3.3.2 读取数据

这里，我们使用 Gluon 提供的 `data` 模块来读取数据。在每一次迭代中，我们将随机读取包含 10 个数据样本的小批量。

```
In [3]: from mxnet.gluon import data as gdata

batch_size = 10
dataset = gdata.ArrayDataset(features, labels)
data_iter = gdata.DataLoader(dataset, batch_size, shuffle=True)
```

和上一节一样，让我们读取并打印第一个小批量数据样本。

```
In [4]: for X, y in data_iter:
    print(X, y)
    break
```

```
[[ -0.22558607 -0.20714636]
 [ 0.23976259  1.46481967]
 [ 1.05327892  0.24552767]
 [ 0.88062727  1.98851633]
 [ 0.76604581  0.24043775]
 [-0.00972697  0.28969124]
 [-1.24220824  0.16625001]
 [ 2.17994285 -0.38305327]
 [-1.27117789  0.96106035]
 [ 0.44383761  2.13921428]]
<NDArray 10x2 @cpu(0)>
[ 4.44664669 -0.31075928  5.46872282 -0.79267985  4.92062283  3.1917932
 1.13409281  9.86030197 -1.61434817 -2.18518162]
<NDArray 10 @cpu(0)>
```

### 3.3.3 定义模型

在上一节从零开始的实现中，我们需要定义模型参数，并使用它们一步步描述模型是怎样计算的。当模型结构变得更复杂时，这些步骤将变得更加繁琐。其实，Gluon 提供了大量预定义的层，这使我们只需关注使用哪些层来构造模型。下面将介绍如何使用 Gluon 更简洁地定义线性回归。

首先，导入 `nn` 模块。我们先定义一个模型变量 `net`，它是一个 `Sequential` 实例。在 Gluon 中，`Sequential` 实例可以看做是一个串联各个层的容器。在构造模型时，我们在该容器中依次添加层。当给定输入数据时，容器中的每一层将依次计算并将输出作为下一层的输入。

```
In [5]: from mxnet.gluon import nn  
  
net = nn.Sequential()
```

回顾图 3.1 中线性回归在神经网络图中的表示。作为一个单层神经网络，线性回归输出层中的神经元和输入层中各个输入完全连接。因此，线性回归的输出层又叫全连接层。在 Gluon 中，全连接层是一个 `Dense` 实例。我们定义该层输出个数为 1。

```
In [6]: net.add(nn.Dense(1))
```

值得一提的是，在 Gluon 中我们无需指定每一层输入的形状，例如线性回归的输入个数。当模型看见数据时，例如后面执行 `net(X)` 时，模型将自动推断出每一层的输入个数。我们将在之后“深度学习计算基础”一章详细介绍这个机制。Gluon 的这一设计为模型开发带来便利。

### 3.3.4 初始化模型参数

在使用 `net` 前，我们需要初始化模型参数，例如线性回归模型中的权重和偏差。这里我们从 MXNet 中导入 `init` 模块，并通过 `init.Normal(sigma=0.01)` 指定权重参数每个元素将在初始化时随机采样于均值为 0 标准差为 0.01 的正态分布。偏差参数全部元素初始化为零。

```
In [7]: from mxnet import init  
  
net.initialize(init.Normal(sigma=0.01))
```

### 3.3.5 定义损失函数

我们从 Gluon 中导入 `loss` 模块，并直接使用它所提供的平方损失作为模型的损失函数。

```
In [8]: from mxnet.gluon import loss as gloss  
  
loss = gloss.L2Loss()
```

### 3.3.6 定义优化算法

同样，我们也无需实现小批量随机梯度下降。在导入 Gluon 后，我们可以创建一个 `Trainer` 实例，并且将模型参数传递给它。下面定义了学习率为 0.03 的小批量随机梯度下降。

```
In [9]: from mxnet import gluon  
  
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.03})
```

### 3.3.7 训练模型

和上一节不同，我们通过调用 `step` 函数来迭代模型参数。由于变量 `l` 是 `batch_size` 维的 `NDArray`，执行 `l.backward()` 等价于 `l.sum().backward()`。按照小批量随机梯度下降的定义，我们在 `step` 函数中提供 `batch_size`，以确保小批量随机梯度是该批量中每个样本梯度的平均。

```
In [10]: num_epochs = 3
for epoch in range(1, num_epochs + 1):
    for X, y in data_iter:
        with autograd.record():
            l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
    print("epoch %d, loss: %f"
          % (epoch, loss(net(features), labels).mean().asnumpy()))

epoch 1, loss: 0.040974
epoch 2, loss: 0.000150
epoch 3, loss: 0.000051
```

下面我们分别比较学到的和真实的模型参数。我们从 `net` 获得需要的层，并访问其权重和位移。学到的和真实的参数很接近。

```
In [11]: dense = net[0]
true_w, dense.weight.data()

Out[11]: ([2, -3.4],
           [[ 1.99986839 -3.39976525]]
           <NDArray 1x2 @cpu(0)>

In [12]: true_b, dense.bias.data()

Out[12]: (4.2,
           [ 4.19911051]
           <NDArray 1 @cpu(0)>)
```

### 3.3.8 小结

- 使用 Gluon 可以更简洁地实现模型。

### 3.3.9 练习

- 如果将 `l = loss(output, y)` 替换成 `l = loss(output, y).mean()`, 我们需要将 `trainer.step(batch_size)` 相应地改成 `trainer.step(1)`。这是为什么呢？

### 3.3.10 扫码直达讨论区



## 3.4 分类模型

前几节介绍的线性回归模型适用于输出为连续值的情景，例如输出为房价。在其他情景中，模型输出还可以是一个离散值，例如图片类别。对于这样的分类问题，我们可以使用分类模型，例如 softmax 回归。和线性回归不同，softmax 回归的输出单元从一个变成了多个。本节以 softmax 回归模型为例，介绍神经网络中的分类模型。Softmax 回归是一个单层神经网络。

让我们考虑一个简单的分类问题。为了便于讨论，让我们假设输入图片的尺寸为  $2 \times 2$ ，并设图片的四个特征值，即像素值分别为  $x_1, x_2, x_3, x_4$ 。假设训练数据集中图片的真实标签为狗、猫或鸡，这些标签分别对应离散值  $y_1, y_2, y_3$ 。举个例子，如果  $y_1 = 0, y_2 = 1, y_3 = 2$ ，任意一张狗图片的标签记作 0。

### 3.4.1 Softmax 运算

我们将一步步地描述 softmax 回归是怎样对单个  $2 \times 2$  图片样本分类的。它将会用到 softmax 运算。

设带下标的  $w$  和  $b$  分别为 softmax 回归的权重和偏差参数。给定单个图片的输入特征

$x_1, x_2, x_3, x_4$ , 我们有

$$o_1 = x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1,$$

$$o_2 = x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2,$$

$$o_3 = x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3.$$

图 3.2 用神经网络图描绘了上面的计算。和线性回归一样，softmax 回归也是一个单层神经网络。和线性回归有所不同的是，softmax 回归输出层中的输出个数等于类别个数，因此从一个变成了多个。在 softmax 回归中， $o_1, o_2, o_3$  的计算都要依赖于  $x_1, x_2, x_3, x_4$ 。所以，softmax 回归的输出层是一个全连接层。

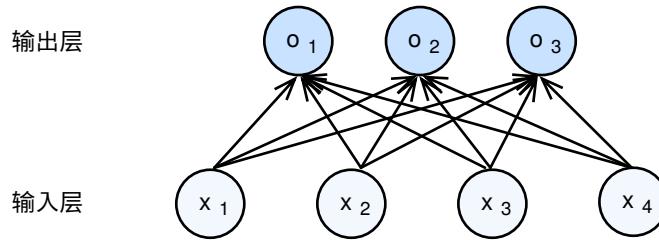


图 3.2: Softmax 回归是一个单层神经网络

在得到输出层的三个输出后，我们需要预测输出分别为狗、猫或鸡的概率。不妨设它们分别为  $\hat{y}_1, \hat{y}_2, \hat{y}_3$ 。下面，我们通过对  $o_1, o_2, o_3$  做 softmax 运算，得到模型最终输出

$$\begin{aligned}\hat{y}_1 &= \frac{\exp(o_1)}{\sum_{i=1}^3 \exp(o_i)}, \\ \hat{y}_2 &= \frac{\exp(o_2)}{\sum_{i=1}^3 \exp(o_i)}, \\ \hat{y}_3 &= \frac{\exp(o_3)}{\sum_{i=1}^3 \exp(o_i)}.\end{aligned}$$

由于  $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$  且  $\hat{y}_1 \geq 0, \hat{y}_2 \geq 0, \hat{y}_3 \geq 0$ ， $\hat{y}_1, \hat{y}_2, \hat{y}_3$  是一个合法的概率分布。我们可将上面 softmax 运算中的三式记作

$$\hat{y}_1, \hat{y}_2, \hat{y}_3 = \text{softmax}(o_1, o_2, o_3).$$

我们有时把 softmax 运算叫做 softmax 层。

### 3.4.2 单样本分类的矢量计算表达式

为了提高计算效率，我们可以将单样本分类通过矢量计算来表达。在上面的图片分类问题中，假设 softmax 回归的权重和偏差参数分别为

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix},$$

设  $2 \times 2$  图片样本  $i$  的特征为

$$\mathbf{x}^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix},$$

输出层输出为

$$\mathbf{o}^{(i)} = \begin{bmatrix} o_1^{(i)} & o_2^{(i)} & o_3^{(i)} \end{bmatrix},$$

预测为狗、猫或鸡的概率分布为

$$\hat{\mathbf{y}}^{(i)} = \begin{bmatrix} \hat{y}_1^{(i)} & \hat{y}_2^{(i)} & \hat{y}_3^{(i)} \end{bmatrix}.$$

我们对样本  $i$  分类的矢量计算表达式为

$$\begin{aligned} \mathbf{o}^{(i)} &= \mathbf{x}^{(i)}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{y}}^{(i)} &= \text{softmax}(\mathbf{o}^{(i)}). \end{aligned}$$

### 3.4.3 小批量样本分类的矢量计算表达式

为了进一步提升计算效率，我们通常对小批量数据做矢量计算。广义上，给定一个小批量样本，其批量大小为  $n$ ，输入个数（特征数）为  $x$ ，输出个数（类别数）为  $y$ 。设批量特征为  $\mathbf{X} \in \mathbb{R}^{n \times x}$ ，批量标签  $\mathbf{y} \in \mathbb{R}^{n \times 1}$ 。假设 softmax 回归的权重和偏差参数分别为  $\mathbf{W} \in \mathbb{R}^{x \times y}$ ,  $\mathbf{b} \in \mathbb{R}^{1 \times y}$ 。Softmax 回归的矢量计算表达式为

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

其中的加法运算使用了广播机制， $\mathbf{O}, \hat{\mathbf{Y}} \in \mathbb{R}^{n \times y}$  且这两个矩阵的第  $i$  行分别为  $\mathbf{o}^{(i)}$  和  $\hat{\mathbf{y}}^{(i)}$ 。

### 3.4.4 交叉熵损失函数

Softmax 回归使用了交叉熵损失函数 (cross-entropy loss)。以本节中的图片分类为例，真实标签狗、猫或鸡分别对应离散值  $y_1, y_2, y_3$ ，它们的预测概率分别为  $\hat{y}_1, \hat{y}_2, \hat{y}_3$ 。为了便于描述，设样本  $i$  的标签的被预测概率为  $p_{\text{label}_i}$ 。例如，如果样本  $i$  的标签为  $y_3$ ，那么  $p_{\text{label}_i} = \hat{y}_3$ 。直观上，训练数据集上每个样本的真实标签的被预测概率越大（最大为 1），分类越准确。假设训练数据集的样本数为  $n$ 。由于对数函数是单调递增的，且最大化函数与最小化该函数的相反数等价，我们希望最小化

$$\ell(\Theta) = -\frac{1}{n} \sum_{i=1}^n \log p_{\text{label}_i},$$

其中  $\Theta$  为模型参数。该函数即交叉熵损失函数。在训练 softmax 回归时，我们将使用优化算法来迭代模型参数并不断降低损失函数的值。

### 3.4.5 模型预测及评价

在训练好 softmax 回归模型后，给定任一样本特征，我们可以预测每个输出类别的概率。通常，我们把预测概率最大的类别作为输出类别。如果它与真实类别（标签）一致，说明这次预测是正确的。在下一节的实验中，我们将使用准确率（accuracy）来评价模型的表现。它等于正确预测数量与总预测数量的比。

### 3.4.6 小结

- Softmax 回归适用于分类问题。它使用 softmax 运算输出类别的概率分布。
- Softmax 回归是一个单层神经网络，输出个数等于分类问题中的类别个数。

### 3.4.7 练习

- 如果按本节 softmax 运算的定义来实现它，可能会有什么问题？

### 3.4.8 扫码直达讨论区



## 3.5 Softmax 回归——从零开始

下面我们来动手实现 Softmax 回归。首先，导入实验所需的包或模块。

```
In [1]: import sys  
        sys.path.append('..')  
        import gluonbook as gb  
        from mxnet import autograd, nd  
        from mxnet.gluon import data as gdata
```

### 3.5.1 获取 Fashion-MNIST 数据集

本节中，我们考虑图片分类问题。我们使用一个类别为服饰的数据集 Fashion-MNIST [1]。该数据集中，图片尺寸为 $28 \times 28$ ，一共包括了 10 个类别，分别为：t-shirt (T 恤)、trouser (裤子)、pullover (套衫)、dress (连衣裙)、coat (外套)、sandal (凉鞋)、shirt (衬衫)、sneaker (运动鞋)、bag (包) 和 ankle boot (短靴)。

下面，我们通过 Gluon 的 `data` 包来下载这个数据集。由于图片中每个像素的值在 0 到 255 之间，我们可以通过定义 `transform` 函数将每个值转换为 0 到 1 之间。

```
In [2]: def transform(feature, label):  
        return feature.astype('float32') / 255, label.astype('float32')  
  
mnist_train = gdata.vision.FashionMNIST(train=True, transform=transform)  
mnist_test = gdata.vision.FashionMNIST(train=False, transform=transform)
```

打印一个样本的形状和它的标签看看。

```
In [3]: feature, label = mnist_train[0]  
        'feature shape: ', feature.shape, 'label: ', label
```

```
Out[3]: ('feature shape: ', (28, 28, 1), 'label: ', 2.0)
```

注意到上面的标签是个数字。以下函数可以将数字标签转成相应的文本标签。

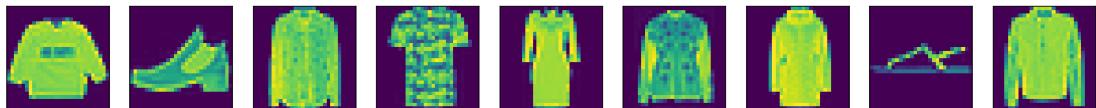
```
In [4]: def get_text_labels(labels):
    text_labels = [
        't-shirt', 'trouser', 'pullover', 'dress', 'coat',
        'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot'
    ]
    return [text_labels[int(i)] for i in labels]
```

我们再定义一个函数来描绘图片内容。

```
In [5]: def show_fashion_imgs(images):
    n = images.shape[0]
    _, figs = plt.subplots(1, n, figsize=(15, 15))
    for i in range(n):
        figs[i].imshow(images[i].reshape((28, 28)).asnumpy())
        figs[i].axes.get_xaxis().set_visible(False)
        figs[i].axes.get_yaxis().set_visible(False)
    plt.show()
```

现在，我们看一下训练数据集中前 9 个样本的图片内容和文本标签。

```
In [6]: X, y = mnist_train[0:9]
show_fashion_imgs(X)
print(get_text_labels(y))
```



```
['pullover', 'ankle boot', 'shirt', 't-shirt', 'dress', 'coat', 'coat', 'sandal',
→  'coat']
```

### 3.5.2 读取数据

Fashion-MNIST 包括训练数据集和测试数据集（testing data set）。我们将在训练数据集上训练模型，并将训练好的模型在测试数据集上评价模型的表现。我们可以像“线性回归——从零开始”一节中那样通过 `yield` 来定义读取小批量数据样本的函数。为了简洁，这里我们直接创建 `DataLoader` 实例，从而每次读取一个样本数为 `batch_size` 的小批量。这里的批量大小 `batch_size` 是一个超参数。需要注意的是，我们每次从训练数据集里读取的小批量是由随机样本组成的。

```
In [7]: batch_size = 256
    train_iter = gdata.DataLoader(mnist_train, batch_size, shuffle=True)
    test_iter = gdata.DataLoader(mnist_test, batch_size, shuffle=False)
```

我们将获取并读取 Fashion-MNIST 数据集的逻辑封装在 gluonbook.load\_data\_fashion\_mnist 函数中供后面章节调用。

### 3.5.3 初始化模型参数

跟线性回归中的例子一样，我们将使用向量表示每个样本。已知每个样本是大小为  $28 \times 28$  的图片。模型的输入向量的长度是  $28 \times 28 = 784$ ：该向量的每个元素对应图片中每个像素。由于图片有 10 个类别，单层神经网络输出层的输出个数为 10。由上一节可知，Softmax 回归的权重和偏差参数分别为  $784 \times 10$  和  $1 \times 10$  的矩阵。

```
In [8]: num_inputs = 784
        num_outputs = 10

        W = nd.random.normal(scale=0.01, shape=(num_inputs, num_outputs))
        b = nd.zeros(num_outputs)
        params = [W, b]
```

同之前一样，我们要对模型参数附上梯度。

```
In [9]: for param in params:
            param.attach_grad()
```

### 3.5.4 定义 Softmax 运算

在介绍如何定义 Softmax 回归之前，我们先描述一下对如何对多维 NDArray 按维度操作。

在下面例子中，给定一个 NDArray 矩阵 X。我们可以只对其中每一列 (`axis=0`) 或每一行 (`axis=1`) 求和，并在结果中保留行和列这两个维度 (`keepdims=True`)。

```
In [10]: X = nd.array([[1,2,3], [4,5,6]])
        X.sum(axis=0, keepdims=True), X.sum(axis=1, keepdims=True)

Out[10]: (
    [[ 5.  7.  9.]]
    <NDArray 1x3 @cpu(0)>,
    [[ 6.]
     [ 15.]]
    <NDArray 2x1 @cpu(0)>)
```

下面我们可以定义上一节中介绍的 Softmax 运算了。在下面的函数中，矩阵  $X$  的行数是样本数，列数是输出个数。为了表达样本预测各个输出的概率，Softmax 运算会先通过  $\exp = \text{nd}.\exp(X)$  对每个元素做指数运算，再对  $\exp$  矩阵的每行求和，最后令矩阵每行各元素与该行元素之和相除。这样一来，最终得到的矩阵每行元素和为 1 且非负（应用了指数运算）。因此，该矩阵每行都是合法的概率分布。Softmax 运算的输出矩阵中的任意一行元素是一个样本在各个类别上的概率。

```
In [11]: def softmax(X):
    exp = X.exp()
    partition = exp.sum(axis=1, keepdims=True)
    return exp / partition # 这里应用了广播机制。
```

可以看到，对于随机输入，我们将每个元素变成了非负数，而且每一行加起来为 1。

```
In [12]: X = nd.random.normal(shape=(2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(axis=1)

Out[12]: (
[[ 0.21324193  0.33961776  0.1239742   0.27106097  0.05210521]
 [ 0.11462264  0.3461234   0.19401033  0.29583326  0.04941036]]
<NDArray 2x5 @cpu(0)>,
[ 1.00000012  1.          ]
<NDArray 2 @cpu(0)>)
```

### 3.5.5 定义模型

有了 Softmax 运算，我们可以定义上节描述的 Softmax 回归模型了。这里通过 `reshape` 函数将每张原始图片改成长度为 `num_inputs` 的向量。

```
In [13]: def net(X):
    return softmax(nd.dot(X.reshape((-1, num_inputs)), W) + b)
```

### 3.5.6 定义损失函数

上一节中，我们介绍了 Softmax 回归使用的交叉熵损失函数。为了得到标签的被预测概率，我们可以使用 `pick` 函数。在下面例子中，`y_hat` 是 2 个样本在 3 个类别的预测概率，`y` 是两个样本的标签类别。通过使用 `pick` 函数，我们得到了 2 个样本的标签的被预测概率。

```
In [14]: y_hat = nd.array([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y = nd.array([0, 2])
```

```
nd.pick(y_hat, y)
```

Out[14]:

```
[ 0.1  0.5]  
<NDArray 2 @cpu(0)>
```

下面，我们直接将上一节中的交叉熵损失函数翻译成代码。

```
In [15]: def cross_entropy(y_hat, y):  
    return -nd.pick(y_hat.log(), y)
```

### 3.5.7 计算分类准确率

给定一个类别的预测概率分布  $y_{\text{hat}}$ ，我们把预测概率最大的类别作为输出类别。如果它与真实类别  $y$  一致，说明这次预测是正确的。分类准确率即正确预测数量与总预测数量的比。

下面定义 `accuracy` 函数。其中  $y_{\text{hat}}.\text{argmax}(\text{axis}=1)$  返回矩阵  $y_{\text{hat}}$  每行中最大元素的索引，且返回结果与  $y$  形状相同。我们在“数据操作”一节介绍过，条件判断式  $(y_{\text{hat}}.\text{argmax}(\text{axis}=1) == y)$  是一个值为 0 或 1 的 NDArray。

```
In [16]: def accuracy(y_hat, y):  
    return (y_hat.argmax(axis=1) == y).mean().asscalar()
```

让我们继续使用在演示 `pick` 函数时定义的  $y_{\text{hat}}$  和  $y$ ，分别作为预测概率分布和标签。可以看到，第一个样本预测类别为 2（该行最大元素 0.6 在本行的索引），与真实标签不一致；第二个样本预测类别为 2（该行最大元素 0.5 在本行的索引），与真实标签一致。因此，这两个样本上的分类准确率为 0.5。

```
In [17]: accuracy(y_hat, y)
```

Out[17]: 0.5

类似地，我们可以评价模型 `net` 在数据集 `data_iter` 上的准确率。

```
In [18]: def evaluate_accuracy(data_iter, net):  
    acc = 0  
    for X, y in data_iter:  
        acc += accuracy(net(X), y)  
    return acc / len(data_iter)
```

因为我们随机初始化了模型 `net`，所以这个模型的准确率应该接近于  $1 / \text{num\_outputs} = 0.1$ 。

```
In [19]: evaluate_accuracy(test_iter, net)
```

Out[19]: 0.0947265625

我们将 `accuracy` 和 `evaluate_accuracy` 函数定义在 `gluonbook` 包中供后面章节调用。

### 3.5.8 训练模型

训练 Softmax 回归的实现跟前面线性回归中的实现非常相似。我们同样使用小批量随机梯度下降来优化模型的损失函数。在训练模型时，迭代周期数 `num_epochs` 和学习率 `lr` 都是可以调的超参数。改变它们的值可能会得到分类更准确的模型。

```
In [20]: num_epochs = 5
        lr = 0.1
        loss = cross_entropy

def train_cpu(net, train_iter, test_iter, loss, num_epochs, batch_size,
             params=None, lr=None, trainer=None):
    for epoch in range(1, num_epochs + 1):
        train_l_sum = 0
        train_acc_sum = 0
        for X, y in train_iter:
            with autograd.record():
                y_hat = net(X)
                l = loss(y_hat, y)
            l.backward()
            if trainer is None:
                gb.sgd(params, lr, batch_size)
            else:
                trainer.step(batch_size)
            train_l_sum += l.mean().asscalar()
            train_acc_sum += accuracy(y_hat, y)
        test_acc = evaluate_accuracy(test_iter, net)
        print("epoch %d, loss %.4f, train acc %.3f, test acc %.3f"
              % (epoch, train_l_sum / len(train_iter),
                 train_acc_sum / len(train_iter), test_acc))

train_cpu(net, train_iter, test_iter, loss, num_epochs, batch_size, params,
          lr)

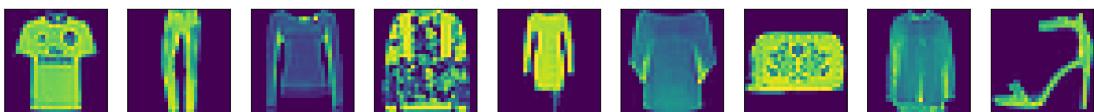
epoch 1, loss 0.7890, train acc 0.747, test acc 0.803
epoch 2, loss 0.5737, train acc 0.811, test acc 0.821
epoch 3, loss 0.5288, train acc 0.825, test acc 0.832
epoch 4, loss 0.5053, train acc 0.831, test acc 0.838
epoch 5, loss 0.4893, train acc 0.834, test acc 0.844
```

我们将 `train_cpu` 函数定义在 `gluonbook` 包中供后面章节调用。

### 3.5.9 预测

训练完成后，现在我们可以演示如何对图片进行分类。给定一系列图片，我们比较一下它们的真实标签和模型预测结果。

```
In [21]: data, label = mnist_test[0:9]
show_fashion_imgs(data)
print('labels:', get_text_labels(label))
predicted_labels = net(data).argmax(axis=1)
print('predictions:', get_text_labels(predicted_labels.astype('numpy')))
```



```
labels: ['t-shirt', 'trouser', 'pullover', 'pullover', 'dress', 'pullover', 'bag',
         'shirt', 'sandal']
predictions: ['t-shirt', 'trouser', 'pullover', 't-shirt', 'coat', 'shirt', 'bag',
              'shirt', 'sandal']
```

### 3.5.10 小结

- 与训练线性回归相比，你会发现训练 Softmax 回归的步骤跟其非常相似：获取并读取数据、定义模型和损失函数并使用优化算法训练模型。事实上，绝大多数深度学习模型的训练都有着类似的步骤。
- 我们可以使用 Softmax 回归做多类别分类。

### 3.5.11 练习

- 本节中，我们直接按照 Softmax 运算的数学定义来实现 softmax 函数。这可能会造成什么问题？（试一试计算  $e^{50}$  的大小。）
- 本节中的 cross\_entropy 函数同样是按照交叉熵损失函数的数学定义实现的。这样的实现方式可能有什么问题？（思考一下对数函数的定义域。）
- 你能想到哪些办法来解决上面这两个问题？

### 3.5.12 扫码直达讨论区



### 3.5.13 参考文献

[1] Xiao, Han, Kashif Rasul, and Roland Vollgraf. “Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.” arXiv preprint arXiv:1708.07747 (2017).

## 3.6 Softmax 回归——使用 Gluon

我们在“[线性回归——使用 Gluon](#)”一节中已经了解了使用 Gluon 实现模型的便利。下面，让我们使用 Gluon 来实现一个 Softmax 回归模型。

首先导入本节实现所需的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import loss as gloss, nn
```

### 3.6.1 获取和读取数据

我们仍然使用 Fashion-MNIST 数据集。我们使用和上一节中相同的批量大小。

```
In [2]: batch_size = 256
        train_iter, test_iter = gb.load_data_fashion_mnist(batch_size)
```

### 3.6.2 定义和初始化模型

在使用 Gluon 定义模型时，我们先通过添加 Flatten 实例将每张原始图片用向量表示。它的输出是一个行数为 `batch_size` 的矩阵，其中每一行代表了一个样本向量。在“[分类模型](#)”一节中，我们提到 Softmax 回归的输出层是一个全连接层。因此，我们继续添加一个输出个数为 10 的全连接层。我们使用均值为 0 标准差为 0.01 的正态分布随机初始化模型的权重参数。

```
In [3]: net = nn.Sequential()
    net.add(nn.Flatten())
    net.add(nn.Dense(10))
    net.initialize(init.Normal(sigma=0.01))
```

### 3.6.3 Softmax 和交叉熵损失函数

如果你做了上一节的练习，那么你可能意识到了分开定义 Softmax 运算和交叉熵损失函数可能会造成数值不稳定。因此，Gluon 提供了一个包括 Softmax 运算和交叉熵损失计算的函数。它的数值稳定性更好。

```
In [4]: loss = gloss.SoftmaxCrossEntropyLoss()
```

### 3.6.4 定义优化算法

我们使用学习率为 0.1 的小批量随机梯度下降作为优化算法。

```
In [5]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

### 3.6.5 训练模型

接下来，我们使用上一节中定义的训练函数来训练模型。

```
In [6]: num_epochs = 5
        gb.train_cpu(net, train_iter, test_iter, loss, num_epochs, batch_size, None,
                     None, trainer)

epoch 1, loss 0.7912, train acc 0.747, test acc 0.804
epoch 2, loss 0.5741, train acc 0.811, test acc 0.816
epoch 3, loss 0.5299, train acc 0.823, test acc 0.825
epoch 4, loss 0.5048, train acc 0.831, test acc 0.833
epoch 5, loss 0.4899, train acc 0.835, test acc 0.840
```

### 3.6.6 小结

- Gluon 提供的函数往往具有更好的数值稳定性。
- 我们可以使用 Gluon 更简洁地实现 Softmax 回归。

### 3.6.7 练习

- 尝试调一调超参数，例如批量大小、迭代周期和学习率，看看结果会怎样。

### 3.6.8 扫码直达讨论区



## 3.7 多层神经网络

我们已经介绍了包括线性回归和 Softmax 回归在内的单层神经网络。本节中，我们将以多层感知机（multilayer perceptron，简称 MLP）为例，介绍多层神经网络的概念。

多层感知机是最基础的深度学习模型。

### 3.7.1 隐藏层

多层感知机在单层神经网络的基础上引入了一到多个隐藏层（hidden layer）。隐藏层位于输入层和输出层之间。图 3.3 展示了一个多层感知机的神经网络图。

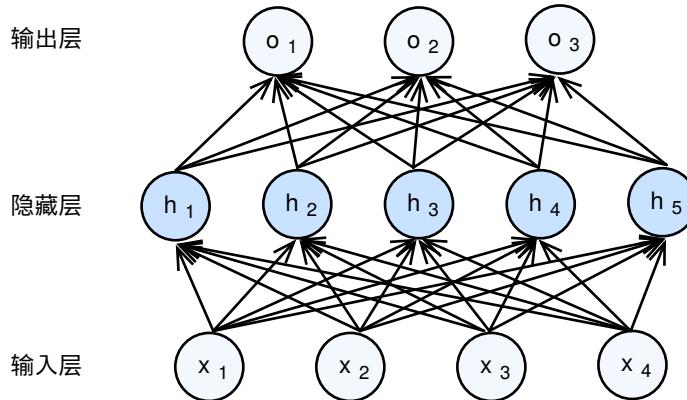


图 3.3: 带有隐藏层的多层感知机。它含有一个隐藏层，该层中有 5 个隐藏单元

在图 3.3 的多层感知机中，输入和输出个数分别为 4 和 3，中间的隐藏层中包含了 5 个隐藏单元 (hidden unit)。由于输入层不涉及计算，图 3.3 中的多层感知机的层数为 2。由图 3.3 可见，隐藏层中的神经元和输入层中各个输入完全连接，输出层中的神经元和隐藏层中的各个神经元也完全连接。因此，多层感知机中的隐藏层和输出层都是全连接层。

### 3.7.2 仿射变换

在描述隐藏层的计算之前，让我们看看多层感知机输出层是怎样计算的。它的计算和之前介绍的单层神经网络的输出层的计算类似：只是输出层的输入变成了隐藏层的输出。我们通常将隐藏层的输出称为隐藏层变量或隐藏变量。

给定一个小批量样本，其批量大小为  $n$ ，输入个数为  $x$ ，输出个数为  $y$ 。假设多层感知机只有一个隐藏层，其中隐藏单元个数为  $h$ ，隐藏变量  $\mathbf{H} \in \mathbb{R}^{n \times h}$ 。假设输出层的权重和偏差参数分别为  $\mathbf{W}_o \in \mathbb{R}^{h \times y}, \mathbf{b}_o \in \mathbb{R}^{1 \times y}$ ，多层感知机输出

$$\mathbf{O} = \mathbf{H}\mathbf{W}_o + \mathbf{b}_o,$$

其中的加法运算使用了广播机制， $\mathbf{O} \in \mathbb{R}^{n \times y}$ 。实际上，多层感知机的输出  $\mathbf{O}$  是对上一层的输出  $\mathbf{H}$  的仿射变换 (affine transformation)。它包括一次通过乘以权重参数的线性变换和一次通过加上偏差参数的平移。

那么，如果隐藏层也只对输入做仿射变换会怎么样呢？设单个样本的特征为  $\mathbf{x} \in \mathbb{R}^{1 \times x}$ ，隐藏层的权重参数和偏差参数分别为  $\mathbf{W}_h \in \mathbb{R}^{x \times h}, \mathbf{b}_h \in \mathbb{R}^{1 \times h}$ 。假设  $\mathbf{h} = \mathbf{x}\mathbf{W}_h + \mathbf{b}_h$  且  $\mathbf{o} = \mathbf{h}\mathbf{W}_o + \mathbf{b}_o$ ，

联立两式可得  $\mathbf{o} = \mathbf{x}\mathbf{W}_h\mathbf{W}_o + \mathbf{b}_h\mathbf{W}_o + \mathbf{b}_o$ : 它等价于单层神经网络的输出  $\mathbf{o} = \mathbf{x}\mathbf{W}' + \mathbf{b}'$ , 其中  $\mathbf{W}' = \mathbf{W}_h\mathbf{W}_o$ ,  $\mathbf{b}' = \mathbf{b}_h\mathbf{W}_o + \mathbf{b}_o$ 。因此, 仅使用仿射变换的隐藏层使多层感知机与前面介绍的单层神经网络没什么区别。

### 3.7.3 激活函数

由上面的例子可以看出, 我们必须在隐藏层中使用其他变换, 例如添加非线性变换, 这样才能使多层感知机变得有意义。我们将这些非线性变换称为激活函数 (activation function)。激活函数能对任意形状的输入按元素操作且不改变输入的形状。以下列举了三种常用的激活函数。

#### ReLU 函数

ReLU (rectified linear unit) 函数提供了一个很简单的非线性变换。给定元素  $x$ , 该函数的输出是

$$\text{relu}(x) = \max(x, 0).$$

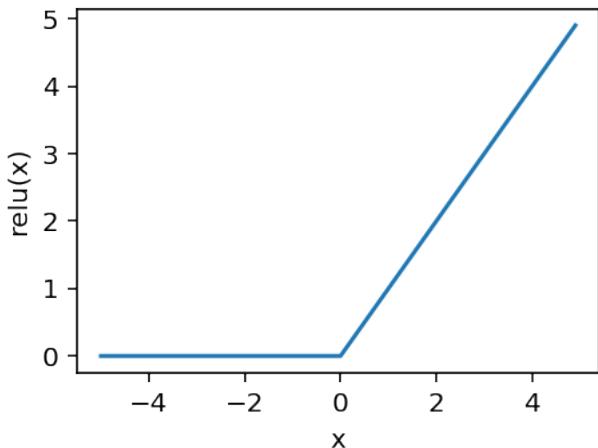
ReLU 函数只保留正数元素, 并将负数元素清零。为了直观地观察这一非线性变换, 让我们先定义一个绘图函数 `xyplot`。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import nd

def xyplot(x_vals, y_vals, x_label, y_label):
    %config InlineBackend.figure_format = 'retina'
    gb.plt.rcParams['figure.figsize'] = (3.5, 2.5)
    gb.plt.plot(x_vals,y_vals)
    gb.plt.xlabel(x_label)
    gb.plt.ylabel(y_label)
    gb.plt.show()
```

让我们绘制 ReLU 函数。当元素值非负时, ReLU 函数实际上在做线性变换。

```
In [2]: x = nd.arange(-5.0, 5.0, 0.1)
        xyplot(x.asnumpy(), x.relu().asnumpy(), 'x', 'relu(x)')
```



## Sigmoid 函数

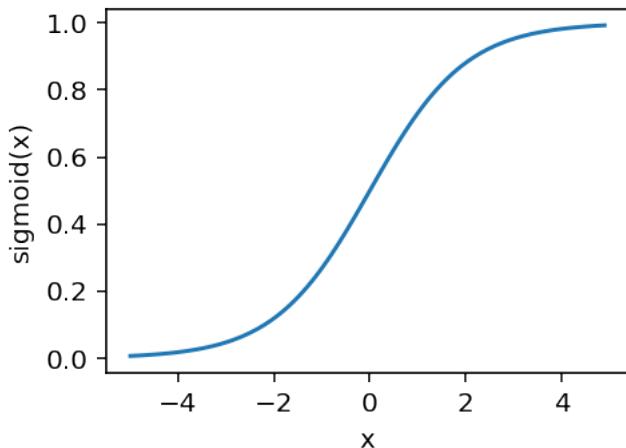
Sigmoid 函数可以将元素的值变换到 0 和 1 之间：

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

我们会在后面“循环神经网络”一章中介绍如何利用 sigmoid 函数值域在 0 到 1 之间这一特性来控制信息在神经网络中的流动。

下面绘制了 sigmoid 函数。当元素值接近 0 时，sigmoid 函数接近线性变换。

```
In [3]: xyplot(x.astype(np.float32), x.sigmoid().astype(np.float32), 'x', 'sigmoid(x)')
```



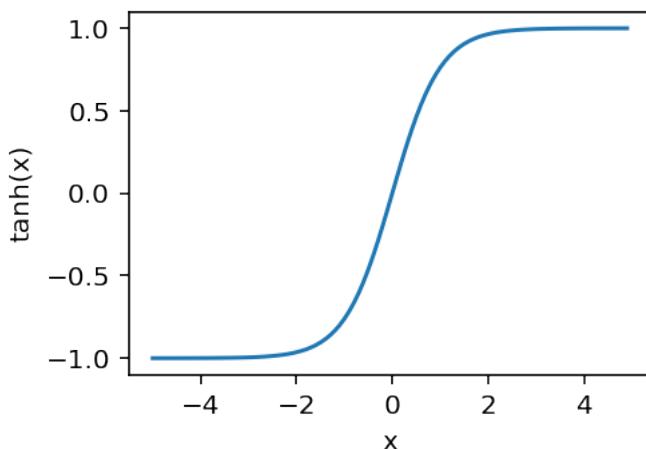
## Tanh 函数

Tanh (双曲正切) 函数可以将元素的值变换到-1 和 1 之间:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

下面绘制了  $\tanh$  函数。当元素值接近 0 时， $\tanh$  函数接近线性变换。值得一提的是，它的形状和 sigmoid 函数很像，且当元素在实数域上均匀分布时， $\tanh$  函数值的均值为 0。

```
In [4]: xyplot(x.astype(np.float32), x.tanh().astype(np.float32), 'x', 'tanh(x)')
```



下面，我们使用三种激活函数来变换输入。按元素操作后，输入和输出形状相同。

```
In [5]: X = np.array([[[0,1], [-2,3], [4,-5]], [[6,-7], [8,-9], [10,-11]]])
X.relu(), X.sigmoid(), X.tanh()

Out[5]: (
    [[[ 0.   1.]
      [ 0.   3.]
      [ 4.   0.]]

     [[ 6.   0.]
      [ 8.   0.]
      [ 10.  0.]]]

    <NDArray 2x3x2 @cpu(0)>,
    [[[ 5.0000000e-01   7.31058598e-01
       [ 1.19202919e-01   9.52574134e-01
       [ 9.82013762e-01   6.69285096e-03]

     [[ 9.97527421e-01   9.11051175e-04]
      [ 9.99664664e-01   1.23394580e-04]
      [ 9.99954581e-01   1.67014223e-05]]]

    <NDArray 2x3x2 @cpu(0)>,
    [[[ 0.          0.76159418]
      [-0.96402758  0.99505478]
      [ 0.99932933 -0.99990922]]

     [[ 0.99998772 -0.99999833]
      [ 0.99999976 -0.99999994]
      [ 1.          -1.        ]]]]

    <NDArray 2x3x2 @cpu(0)>)
```

### 3.7.4 多层感知机

现在，我们可以给出多层感知机的矢量计算表达式了。

给定一个小批量样本  $X \in \mathbb{R}^{n \times x}$ ，其批量大小为  $n$ ，输入个数为  $x$ ，输出个数为  $y$ 。假设多层感知机只有一个隐藏层，其中隐藏单元个数为  $h$ ，激活函数为  $\phi$ 。假设隐藏层的权重和偏差参数分别为  $W_h \in \mathbb{R}^{x \times h}$ ,  $b_h \in \mathbb{R}^{1 \times h}$ ，输出层的权重和偏差参数分别为  $W_o \in \mathbb{R}^{h \times y}$ ,  $b_o \in \mathbb{R}^{1 \times y}$ 。多层感知机的矢量计算表达式为

$$H = \phi(XW_h + b_h),$$

$$O = HW_o + b_o,$$

其中的加法运算使用了广播机制， $\mathbf{H} \in \mathbb{R}^{n \times h}$ ,  $\mathbf{O} \in \mathbb{R}^{n \times y}$ 。在分类问题中，我们可以对输出  $\mathbf{O}$  做 Softmax 运算，并使用 Softmax 回归中的交叉熵损失函数。在回归问题中，我们将输出层的输出个数设为 1，并将输出  $\mathbf{O}$  直接提供给线性回归中使用的平方损失函数。

我们可以添加更多的隐藏层来构造更深的模型。需要指出的是，多层感知机的层数和各隐藏层中隐藏单元个数都是超参数。

### 3.7.5 计算梯度

有了多层感知机的损失函数后，优化算法将迭代模型参数从而不断降低损失函数的值。我们在前面章节里用了小批量随机梯度下降，和大多数其他优化算法一样，它需要计算损失函数有关模型参数的梯度。我们可以把数学上对复合函数求梯度的链式法则应用于多层感知机的梯度计算。我们将在后面章节详细介绍深度学习模型中的梯度计算，例如多层感知机的正向传播和反向传播。

### 3.7.6 衰减和爆炸

需要注意的是，当深度学习模型的层数较多时，模型的数值稳定性容易变得较差。假设一个层数为  $L$  的多层感知机的第  $l$  层  $\mathbf{H}^{(l)}$  的权重参数为  $\mathbf{W}^{(l)}$ ，输出层  $\mathbf{H}^{(L)}$  的权重参数为  $\mathbf{W}^{(L)}$ 。为了便于讨论，假设不考虑偏差参数，且所有隐藏层的激活函数为  $\phi(x) = x$ 。给定输入  $\mathbf{X}$ ，多层感知机的第  $l$  层的输出  $\mathbf{H}^{(l)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} \dots \mathbf{W}^{(l)}$ 。这时候，如果层数  $l$  较大， $\mathbf{H}^{(l)}$  的计算可能会出现衰减 (vanishing) 或爆炸 (explosion)。举个例子，假如输入和所有层的权重参数都是标量的话，比如 0.5 和 2，多层感知机的第 50 层输出在输入前的系数分别为  $0.5^{50}$  (衰减) 和  $2^{50}$  (爆炸)。类似地，当层数较多时，梯度的计算也更容易出现衰减或爆炸。我们会在后面章节还会介绍梯度的衰减或爆炸，例如循环神经网络的通过时间反向传播。

### 3.7.7 随机初始化模型参数

在神经网络中，我们需要随机初始化模型参数。下面我们来解释这一点。

以图 3.3 为例，假设输出层只保留一个输出单元  $o_1$  (删去  $o_2, o_3$  和指向它们的箭头)，且隐藏层使用相同的激活函数。如果初始化后每个隐藏单元的参数都相同，那么在模型训练时每个隐藏单元将根据相同输入计算出相同的值。接下来输出层也将从各个隐藏单元拿到完全一样的值。在迭代每个隐藏单元的参数时，这些参数在每轮迭代的值都相同。那么，由于每个隐藏单元拥有相同激活函数和相同参数，所有隐藏单元将继续根据下一次迭代时的相同输入计算出相同的值。如此周

而复始。这种情况下，无论隐藏单元个数有多大，隐藏层本质上只有 1 个隐藏单元在发挥作用。因此，我们通常会随机初始化神经网络的模型参数，例如每个神经元的权重参数。

### MXNet 的默认随机初始化

随机初始化模型参数的方法有很多。在“[线性回归——使用 Gluon](#)”一节中，我们使用 `net.initialize(init.Normal(sigma=0.01))` 令模型 `net` 的权重参数采用正态分布的随机初始化方式。如果不指定初始化方法，例如 `net.initialize()`，我们将使用 MXNet 的默认随机初始化方法。在默认条件下的初始化时，权重参数每个元素随机采样于 -0.07 到 0.07 之间的均匀分布，偏差参数全部元素清零。

### Xavier 随机初始化

还有一种比较常用的随机初始化方法叫做 Xavier 随机初始化 [1]。假设某全连接层的输入个数为  $a$ ，输出个数为  $b$ ，Xavier 随机初始化将该层权重参数的每个元素随机采样于均匀分布

$$U\left(-\sqrt{\frac{6}{a+b}}, \sqrt{\frac{6}{a+b}}\right).$$

它的设计主要考虑到，模型参数初始化后，每层输出的方差不该被该层输入个数所影响，且每层梯度的方差不该被该层输出个数所影响。这两点与我们之后将要介绍的正向传播和反向传播有关。

### 3.7.8 小结

- 多层感知机对输入做了一系列的线性和非线性的变换。
- 常用的激活函数包括 ReLU 函数、sigmoid 函数和 tanh 函数。
- 我们需要随机初始化神经网络的模型参数。

### 3.7.9 练习

- 有人说随机初始化模型参数时为了“打破对称性”。这里的“对称”应如何理解？

### 3.7.10 扫码直达讨论区



### 3.7.11 参考文献

[1] Glorot, Xavier, and Yoshua Bengio. “Understanding the difficulty of training deep feed-forward neural networks.” Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010.

## 3.8 多层感知机——从零开始

我们已经从上一章里了解了多层感知机的原理。下面我们一起来动手实现一个多层感知机。首先导入实现所需的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, gluon, nd
        from mxnet.gluon import loss as gloss
```

### 3.8.1 获取和读取数据

我们继续使用 Fashion-MNIST 数据集。我们将使用多层感知机对图片进行分类。

```
In [2]: batch_size = 256
        train_iter, test_iter = gb.load_data_fashion_mnist(batch_size)
```

### 3.8.2 定义模型参数

我们在“Softmax 回归——从零开始”一节里已经介绍了，Fashion-MNIST 数据集中图片尺寸为  $28 \times 28$ ，类别数为 10。本节中我们依然使用长度为  $28 \times 28 = 784$  的向量表示每一张图片。因此，输入个数为 784，输出个数为 10。实验中，我们设超参数隐藏单元个数为 256。

```
In [3]: num_inputs = 784
        num_outputs = 10
        num_hiddens = 256

        W1 = nd.random.normal(scale=0.01, shape=(num_inputs, num_hiddens))
        b1 = nd.zeros(num_hiddens)
        W2 = nd.random.normal(scale=0.01, shape=(num_hiddens, num_outputs))
        b2 = nd.zeros(num_outputs)
        params = [W1, b1, W2, b2]

        for param in params:
            param.attach_grad()
```

### 3.8.3 定义激活函数

这里我们使用 ReLU 作为隐藏层的激活函数。

```
In [4]: def relu(X):
        return nd.maximum(X, 0)
```

### 3.8.4 定义模型

同 Softmax 回归一样，我们通过 `reshape` 函数将每张原始图片改成长度为 `num_inputs` 的向量。然后我们将上一节多层感知机的矢量计算表达式翻译成代码。

```
In [5]: def net(X):
        X = X.reshape((-1, num_inputs))
        H = relu(nd.dot(X, W1) + b1)
        return nd.dot(H, W2) + b2
```

### 3.8.5 定义损失函数

为了得到更好的数值稳定性，我们直接使用 Gluon 提供的包括 Softmax 运算和交叉熵损失计算的函数。

```
In [6]: loss = gloss.SoftmaxCrossEntropyLoss()
```

### 3.8.6 训练模型

训练多层感知机的步骤和之前训练 Softmax 回归的步骤没什么区别。我们直接调用 gluonbook 包中的 `train_cpu` 函数，它的实现已经在“Softmax 回归——从零开始”一节里介绍了。

这里设超参数迭代周期为 5，学习率为 0.5。

```
In [7]: num_epochs = 5
        lr = 0.5
        gb.train_cpu(net, train_iter, test_iter, loss, num_epochs, batch_size,
                    params, lr)

epoch 1, loss 0.8227, train acc 0.694, test acc 0.789
epoch 2, loss 0.4919, train acc 0.818, test acc 0.847
epoch 3, loss 0.4348, train acc 0.839, test acc 0.846
epoch 4, loss 0.3935, train acc 0.854, test acc 0.856
epoch 5, loss 0.3685, train acc 0.863, test acc 0.872
```

### 3.8.7 小结

- 我们可以通过手动定义模型及其参数来实现简单的多层感知机。
- 当多层感知机的层数较多时，本节的实现方法会显得较繁琐，特别在定义模型参数上。

### 3.8.8 练习

- 改变 `num_hidden` 超参数的值，看看对结果有什么影响。
- 试着加入一个新的隐藏层，看看对结果有什么影响。

### 3.8.9 扫码直达讨论区



## 3.9 多层感知机——使用 Gluon

下面我们使用 Gluon 来实现上一节中的多层感知机。导入所需的包或模块。

```
In [1]: import sys  
        sys.path.append('..')  
        import gluonbook as gb  
        from mxnet import autograd, gluon, init, nd  
        from mxnet.gluon import loss as gloss, nn
```

### 3.9.1 定义模型

和 Softmax 回归唯一的不同在于，我们多加了一个全连接层作为隐藏层。我们指定了该层的隐藏单元个数为 256，并使用 ReLU 作为激活函数。

```
In [2]: net = nn.Sequential()  
        net.add(nn.Dense(256, activation='relu'))  
        net.add(nn.Dense(10))  
        net.add(nn.Dense(10))  
        net.initialize(init.Normal(sigma=0.01))
```

### 3.9.2 读取数据并训练模型

我们使用和训练 Softmax 回归几乎相同的步骤来读取数据并训练模型。

```
In [3]: batch_size = 256  
        train_iter, test_iter = gb.load_data_fashion_mnist(batch_size)  
  
        loss = gloss.SoftmaxCrossEntropyLoss()  
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.5})
```

```
num_epochs = 5
gb.train_cpu(net, train_iter, test_iter, loss, num_epochs, batch_size,
             None, None, trainer)

epoch 1, loss 1.6850, train acc 0.368, test acc 0.643
epoch 2, loss 0.7244, train acc 0.715, test acc 0.664
epoch 3, loss 0.6038, train acc 0.778, test acc 0.810
epoch 4, loss 0.4966, train acc 0.817, test acc 0.800
epoch 5, loss 0.4464, train acc 0.835, test acc 0.841
```

### 3.9.3 小结

- 通过 Gluon 我们可以更方便地构造多层感知机。

### 3.9.4 练习

- 尝试多加入几个隐藏层，对比上节中从零开始的实现。
- 使用其他的激活函数，看看对结果的影响。

### 3.9.5 扫码直达讨论区



## 3.10 欠拟合、过拟合和模型选择

在前几节基于 Fashion-MNIST 数据集的实验中，我们评价了机器学习模型在训练数据集和测试数据集上的表现。如果你动手改变过实验中的模型结构或者超参数的话，你也许发现了：当模型在训练数据集上更准确时，在测试数据集上的准确率既可能上升又可能下降。这是为什么呢？

### 3.10.1 训练误差和泛化误差

在解释上面提到的现象之前，我们需要区分训练误差 (training error) 和泛化误差 (generalization error)：前者指模型在训练数据集上表现出的误差，后者指模型在任意一个测试数据样本上表现出的误差的期望。训练误差和泛化误差的计算可以利用我们之前介绍过的损失函数，例如线性回归用到的平方损失函数和 Softmax 回归用到的交叉熵损失函数。

假设训练数据集和测试数据集里的每一个样本都是从同一个概率分布中相互独立地生成的。基于该独立同分布假设，给定任意一个机器学习模型及其参数和超参数，它的训练误差的期望和泛化误差都是一样的。然而从之前的章节中我们了解到，模型的参数并不是事先给定的，而是通过训练数据集训练模型而学习出的。所以，训练误差的期望小于或等于泛化误差。也就是说，通常情况下，由训练数据集学到的模型参数会使模型在训练数据集上的表现优于或等于在测试数据集上的表现。由于无法从训练误差估计泛化误差，降低训练误差并不意味着泛化误差一定会降低。我们希望通过适当降低模型的训练误差，从而能够间接降低模型的泛化误差。

### 3.10.2 欠拟合和过拟合

给定测试数据集，我们通常用机器学习模型在该测试数据集上的误差来反映泛化误差。当模型无法得到较低的训练误差时，我们将这一现象称作欠拟合 (underfitting)。当模型的训练误差远小于它在测试数据集上的误差时，我们称该现象为过拟合 (overfitting)。在实践中，我们要尽可能同时避免欠拟合和过拟合的出现。虽然有很多因素可能导致这两种拟合问题，在这里我们重点讨论两个因素：模型复杂度和训练数据集大小。

#### 模型复杂度

为了解释模型复杂度，让我们以多项式函数拟合为例。给定一个由标量数据特征  $x$  和对应的标量标签  $y$  组成的训练数据集，多项式函数拟合的目标是找一个  $K$  阶多项式函数

$$\hat{y} = b + \sum_{k=1}^K x^k w_k$$

来近似  $y$ 。上式中，带下标的  $w$  是模型的权重参数， $b$  是偏差参数。和线性回归相同，多项式函数拟合也使用平方损失函数。特别地，一阶多项式函数拟合又叫线性函数拟合。

高阶多项式函数比低阶多项式函数的复杂度更高，例如高阶多项式函数模型参数更多，模型函数的选择范围更广。正因为如此，高阶多项式函数比低阶多项式函数更容易在相同的训练数据集上得到更低的训练误差。给定训练数据集，模型复杂度的和误差之间的关系通常如图 3.4 所示。给

定训练数据集，如果模型的复杂度过低，很容易出现欠拟合；如果模型复杂度过高，很容易出现过拟合。

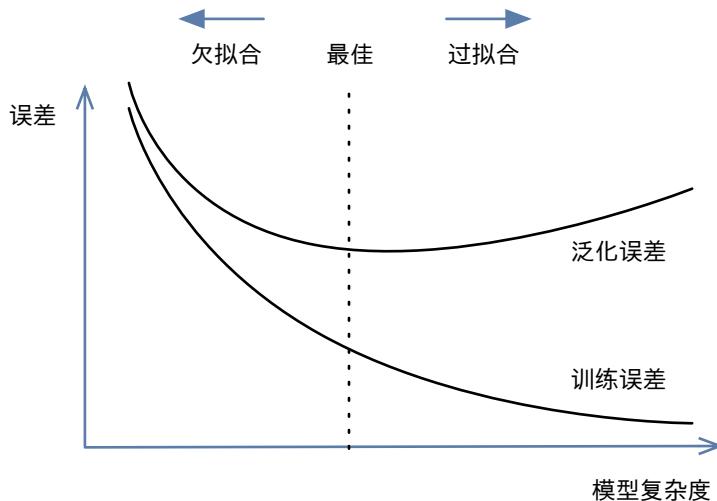


图 3.4: 模型复杂度对欠拟合和过拟合的影响

### 训练数据集大小

影响欠拟合和过拟合的另一个重要因素是训练数据集大小。一般来说，如果训练数据集过小，特别是比模型参数数量更小时，过拟合更容易发生。

此外，泛化误差不会随训练数据集里样本数量增加而增大。因此，在计算资源允许范围之内，我们通常希望训练数据集大一些，特别当模型复杂度较高时，例如训练层数较多的深度学习模型时。

#### 3.10.3 多项式函数拟合实验

为了理解模型复杂度和训练数据集大小对欠拟合和过拟合的影响，下面让我们以多项式函数拟合为例来实验。首先导入实现需要的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, gluon, nd
        from mxnet.gluon import data as gdata, loss as gloss, nn
```

## 生成数据集

我们将生成一个人工数据集。在训练数据集和测试数据集中，给定样本特征  $x$ ，我们使用如下的三阶多项式函数来生成该样本的标签：

$$y = 1.2x - 3.4x^2 + 5.6x^3 + 5 + \epsilon,$$

其中噪音项  $\epsilon$  服从均值为 0 和标准差为 0.1 的正态分布。训练数据集和测试数据集的样本数都设为 100。

```
In [2]: n_train = 100
        n_test = 100
        true_w = [1.2, -3.4, 5.6]
        true_b = 5

        features = nd.random.normal(shape=(n_train + n_test, 1))
        poly_features = nd.concat(features, nd.power(features, 2),
                                   nd.power(features, 3))
        labels = (true_w[0] * poly_features[:, 0] + true_w[1] * poly_features[:, 1]
                  + true_w[2] * poly_features[:, 2] + true_b)
        labels += nd.random.normal(scale=0.1, shape=labels.shape)
```

看一看生成数据集的前 5 个样本。

```
In [3]: features[:5], poly_features[:5], labels[:5]
```

```
Out[3]: (
    [[ 2.1220636
      [ 0.7740038
      [ 1.04344046
      [ 1.18392551
      [ 1.89171135]]
    <NDArray 5x1 @cpu(0)>,
    [[ 2.1220636   4.893857   10.82622147
      [ 0.7740038   0.59908187   0.46369165]
      [ 1.04344046   1.08876801   1.13606453]
      [ 1.18392551   1.40167964   1.65948427]
      [ 1.89171135   3.5785718   6.76962519]]
    <NDArray 5x3 @cpu(0)>,
    [ 51.6748848   6.3585763   8.94907284   11.09345436   33.03696442]
    <NDArray 5 @cpu(0)>)
```

## 定义、训练和测试模型

我们先定义作图函数，其中 y 轴是基于对数尺度。该作图函数 `semilogy` 也被定义在 `gluon-book` 包中供后面章节调用。

```
In [4]: from IPython.display import set_matplotlib_formats
def semilogy(x_vals, y_vals, x_label, y_label, x2_vals=None, y2_vals=None,
             legend=None, figsize=(3.5, 2.5)):
    gb=plt.rcParams['figure.figsize'] = figsize
    set_matplotlib_formats('retina')
    gb=plt.xlabel(x_label)
    gb=plt.ylabel(y_label)
    gb=plt.semilogy(x_vals, y_vals)
    if x2_vals and y2_vals:
        gb=plt.semilogy(x2_vals, y2_vals)
        gb=plt.legend(legend)
    gb=plt.show()
```

和线性回归一样，多项式函数拟合也使用平方损失函数。由于我们将尝试使用不同复杂度的模型来拟合生成的数据集，我们把模型定义部分放在 `fit_and_plot` 函数中。多项式函数拟合的训练和测试步骤与之前介绍的 Softmax 回归中的这些步骤类似。

```
In [5]: num_epochs = 100
loss = gloss.L2Loss()

def fit_and_plot(train_features, test_features, train_labels, test_labels):
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize()
    batch_size = min(10, train_labels.shape[0])
    train_iter = gdata.DataLoader(gdata.ArrayDataset(
        train_features, train_labels), batch_size, shuffle=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                           {'learning_rate': 0.01})
    train_ls, test_ls = [], []
    for _ in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        train_ls.append(loss(net(train_features),
                             train_labels).mean().asscalar())
    test_ls.append(loss(net(test_features),
                        test_labels).mean().asscalar())
```

```

        test_ls.append(loss(net(test_features),
                             test_labels).mean().asscalar())
    print('final epoch: train loss', train_ls[-1], 'test loss', test_ls[-1])
    semilogy(range(1, num_epochs+1), train_ls, 'epochs', 'loss',
              range(1, num_epochs+1), test_ls, ['train', 'test'])
    return ('weight:', net[0].weight.data(), 'bias:', net[0].bias.data())

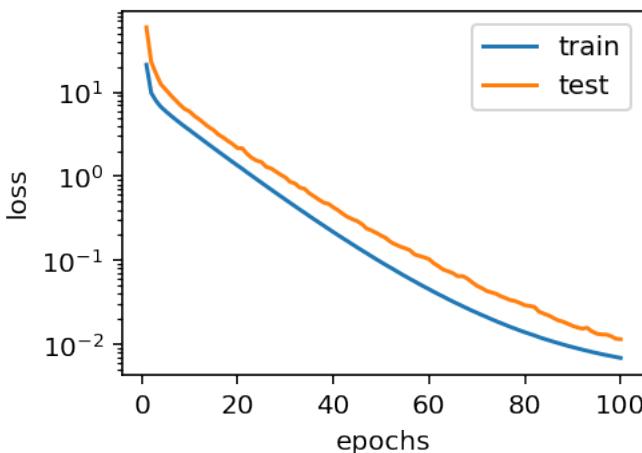
```

### 三阶多项式函数拟合（正常）

我们先使用与数据生成函数同阶的三阶多项式函数拟合。实验表明，这个模型的训练误差和在测试数据集的误差都较低。训练出的模型参数也接近真实值。

```
In [6]: fit_and_plot(poly_features[:n_train, :], poly_features[n_train:, :],
                     labels[:n_train], labels[n_train:])

final epoch: train loss 0.00693874 test loss 0.0115709
```



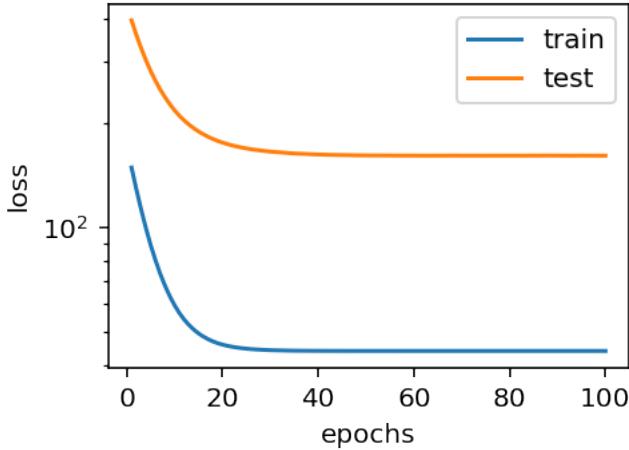
```
Out[6]: ('weight:',
 [[ 1.32278931 -3.3634181   5.56269598]]
<NDArray 1x3 @cpu(0)>, 'bias:',
 [ 4.95207453]
<NDArray 1 @cpu(0)>)
```

### 线性函数拟合（欠拟合）

我们再试试线性函数拟合。很明显，该模型的训练误差在迭代早期下降后便很难继续降低。在完成最后一次迭代周期后，训练误差依旧很高。线性模型在非线性模型（例如三阶多项式函数）生

成的数据集上容易欠拟合。

```
In [7]: fit_and_plot(features[:n_train, :], features[n_train:, :], labels[:n_train],  
                     labels[n_train:])  
  
final epoch: train loss 43.9977 test loss 160.85
```



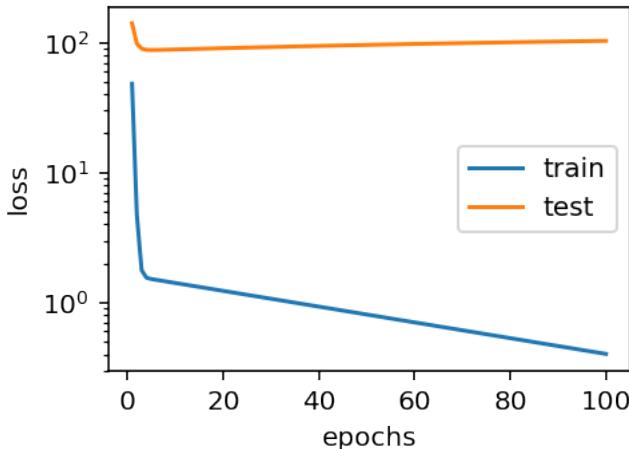
```
Out[7]: ('weight:',  
         [[ 15.54629898]]  
         <NDArray 1x1 @cpu(0)>, 'bias:',  
         [ 2.27738953]  
         <NDArray 1 @cpu(0)>)
```

### 训练量不足（过拟合）

事实上，即便是使用与数据生成模型同阶的三阶多项式函数模型，如果训练量不足，该模型依然容易过拟合。

让我们仅仅使用两个样本来训练模型。显然，训练样本过少了，甚至少于模型参数的数量。这使模型显得过于复杂，以至于容易被训练数据中的噪音影响。在迭代过程中，即便训练误差较低，但是测试数据集上的误差却很高。这是典型的过拟合现象。

```
In [8]: fit_and_plot(poly_features[0:2, :], poly_features[n_train:, :], labels[0:2],  
                     labels[n_train:])  
  
final epoch: train loss 0.402737 test loss 103.314
```



```
Out[8]: ('weight:',
 [[ 1.38723636  1.93765891  3.50859237]]
<NDArray 1x3 @cpu(0)>, 'bias:',
 [ 1.23128557]
<NDArray 1 @cpu(0)>)
```

我们将在后面的章节继续讨论过拟合问题以及应对过拟合的方法，例如正则化和丢弃法。

### 3.10.4 模型选择

我们已经知道，训练误差无法被用来估计泛化误差。那我们是否可以根据测试数据集上的误差来调节超参数和选择模型呢？答案是否定的。原因很简单：为降低测试数据集误差而修改模型将使本节开始的“独立同分布假设”不再成立。此时测试数据集的误差无法正确反映泛化误差。

在选择模型时，我们可以切分原始训练数据集：其中大部分样本组成新的训练数据集，剩下的组成为验证数据集（validation data set）。我们在新的训练数据集上训练模型，并根据模型在验证数据集上的表现调参和选择模型。最后，我们在测试数据集上评价模型的表现。

#### *K* 折交叉验证

验证模型还有很多其他的方法。其中一种常用方法叫做 *K* 折交叉验证 (*k*-fold cross-validation)。

在 *K* 折交叉验证中，我们把原始训练数据集分割成 *K* 个不重合的子数据集。然后我们做 *K* 次模型训练和验证。每一次，我们使用一个子数据集验证模型，并使用其他  $K - 1$  个子数据集来训练

模型。在这  $K$  次训练和验证中，每次用来验证模型的子数据集都不同。最后，我们只需对这  $K$  次训练误差和验证误差分别求平均作为最终的训练误差和验证误差。

我们将在本章最后一节实验  $K$  折交叉验证。

### 3.10.5 小结

- 我们希望通过适当降低模型的训练误差，从而能够间接降低模型的泛化误差。
- 欠拟合指模型无法得到较低的训练误差；过拟合指模型的训练误差远小于它在测试数据集上的误差。
- 我们应选择复杂度合适的模型并避免使用过少的训练样本。
- 我们要避免根据测试数据集上的误差来选择模型和调节超参数。

### 3.10.6 练习

- 如果用一个三阶多项式模型来拟合一个线性模型生成的数据，可能会有什么问题？为什么？
- 在我们本节提到的三阶多项式拟合问题里，有没有可能把 100 个样本的训练误差的期望降到 0，为什么？

### 3.10.7 扫码直达讨论区



## 3.11 正则化——从零开始

上一节中我们观察了过拟合现象，即模型的训练误差远小于它在测试数据集上的误差。本节将介绍应对过拟合问题的常用方法：正则化。

### 3.11.1 $L_2$ 范数正则化

在深度学习中，我们常使用  $L_2$  范数正则化，也就是在模型原先损失函数基础上添加  $L_2$  范数惩罚项，从而得到训练所需要最小化的函数。 $L_2$  范数惩罚项指的是模型权重参数每个元素的平方和与一个超参数的乘积。以“单层神经网络”一节中线性回归的损失函数  $\ell(w_1, w_2, b)$  为例（ $w_1, w_2$  是权重参数， $b$  是偏差参数），带有  $L_2$  范数惩罚项的新损失函数为

$$\ell(w_1, w_2, b) + \frac{\lambda}{2}(w_1^2 + w_2^2),$$

其中超参数  $\lambda > 0$ 。当权重参数均为 0 时，惩罚项最小。当  $\lambda$  较大时，惩罚项在损失函数中的比重较大，这通常会使学到的权重参数的元素较接近 0。当  $\lambda$  设为 0 时，惩罚项完全不起作用。

有了  $L_2$  范数惩罚项后，在小批量随机梯度下降中，我们将“单层神经网络”一节中权重  $w_1$  和  $w_2$  的迭代方式更改为

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

可见， $L_2$  范数正则化令权重  $w_1$  和  $w_2$  的每一步迭代分别添加了  $-\lambda w_1$  和  $-\lambda w_2$ 。因此，我们有时也把  $L_2$  范数正则化称为权重衰减（weight decay）。

在实际中，我们有时也在惩罚项中添加偏差元素的平方和。假设神经网络中某一个神经元的输入是  $x_1, x_2$ ，使用激活函数  $\phi$  并输出  $\phi(x_1 w_1 + x_2 w_2 + b)$ 。假设激活函数  $\phi$  是 ReLU、tanh 或 sigmoid，如果  $w_1, w_2, b$  都非常接近 0，那么输出也接近 0。也就是说，这个神经元的作用比较小，甚至就像是令神经网络少了一个神经元一样。上一节我们提到，给定训练数据集，过高复杂度的模型容易过拟合。因此， $L_2$  范数正则化可能对过拟合有效。

### 3.11.2 高维线性回归实验

下面，我们通过高维线性回归为例来引入一个过拟合问题，并使用  $L_2$  范数正则化来试着应对过拟合。

## 生成数据集

设数据样本特征的维度为  $p$ 。对于训练数据集和测试数据集中特征为  $x_1, x_2, \dots, x_p$  的任一样本，我们使用如下的线性函数来生成该样本的标签：

$$y = 0.05 + \sum_{i=1}^p 0.01x_i + \epsilon,$$

其中噪音项  $\epsilon$  服从均值为 0 和标准差为 0.1 的正态分布。为了较容易地观察过拟合，我们考虑高维线性回归问题，例如设维度  $p = 200$ ；同时，我们特意把训练数据集的样本数设低，例如 20。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, gluon, nd

n_train = 20
n_test = 100

num_inputs = 200
true_w = nd.ones((num_inputs, 1)) * 0.01
true_b = 0.05

features = nd.random.normal(shape=(n_train+n_test, num_inputs))
labels = nd.dot(features, true_w) + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)
train_features, test_features = features[:n_train, :], features[n_train:, :]
train_labels, test_labels = labels[:n_train], labels[n_train:]
```

## 初始化模型参数

下面定义函数来随机初始化模型参数，并为它们附上梯度。

```
In [2]: def init_params():
        w = nd.random.normal(scale=1, shape=(num_inputs, 1))
        b = nd.zeros(shape=(1,))
        params = [w, b]
        for param in params:
            param.attach_grad()
        return params
```

## 定义 $L_2$ 范数惩罚项

下面定义 $L_2$ 范数惩罚项。这里只惩罚模型的权重参数。

```
In [3]: def l2_penalty(w):
    return (w**2).sum() / 2
```

## 定义训练和测试

下面定义如何在训练数据集和测试数据集上分别训练和测试模型。和前面几节中不同的是，这里在计算最终的损失函数时添加了 $L_2$ 范数惩罚项。

```
In [4]: batch_size = 1
        num_epochs = 10
        lr = 0.003

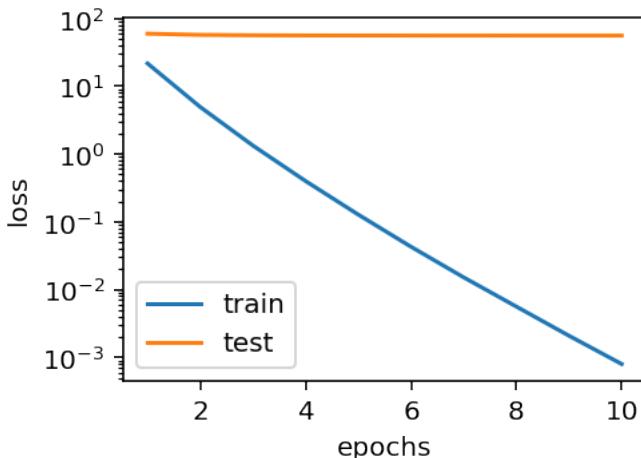
        net = gb.linreg
        loss = gb.squared_loss
        %config InlineBackend.figure_format = 'retina'
        gb.pyplot.rcParams['figure.figsize'] = (3.5, 2.5)

        def fit_and_plot(lambd):
            w, b = params = init_params()
            train_ls = []
            test_ls = []
            for _ in range(num_epochs):
                for X, y in gb.data_iter(batch_size, n_train, features, labels):
                    with autograd.record():
                        # 添加了 $L_2$ 范数惩罚项。
                        l = loss(net(X, w, b), y) + lambd * l2_penalty(w)
                        l.backward()
                        gb.sgd(params, lr, batch_size)
                train_ls.append(loss(net(train_features, w, b),
                                     train_labels).mean().asscalar())
                test_ls.append(loss(net(test_features, w, b),
                                    test_labels).mean().asscalar())
            gb.semilogy(range(1, num_epochs+1), train_ls, 'epochs', 'loss',
                        range(1, num_epochs+1), test_ls, ['train', 'test'])
            return 'w[:10]:', w[:10].T, 'b:', b
```

## 观察过拟合

接下来，让我们训练并测试高维线性回归模型。当 `lambda` 设为 0 时，我们没有使用正则化。结果训练误差远小于测试数据集上的误差。这是典型的过拟合现象。

```
In [5]: fit_and_plot(lambda=0)
```



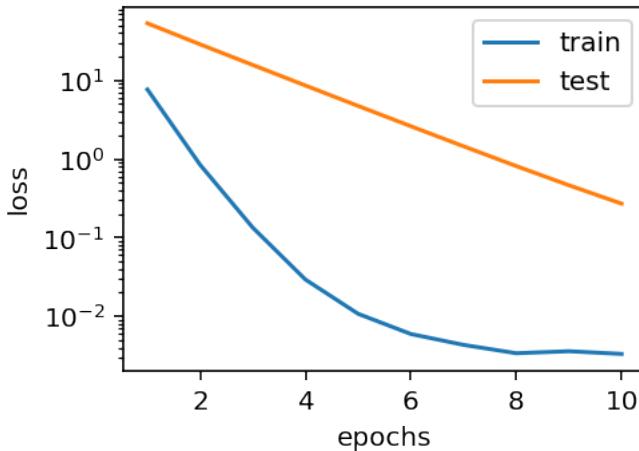
```
Out[5]: ('w[:10]:',
 [[-0.81736666 -0.22291766 -0.74899858 -2.25943184 -0.10184114 -0.69823253
 -0.09742846  0.60911506 -1.10479963 -1.44806385]]
<NDArray 1x10 @cpu(0)>, 'b:',
 [-0.38648978]
<NDArray 1 @cpu(0)>)
```

## 使用正则化

下面我们使用  $L_2$  范数正则化。我们发现训练误差虽然有所提高，但测试数据集上的误差有所下降。过拟合现象得到一定程度上的缓解。另外，学到的权重参数的绝对值比不使用正则化时相比更接近 0。

然而，即便是使用了正则化的模型依然没有学出较准确的模型参数。这主要是因为训练数据集的样本数相对维度来说太小。

```
In [6]: fit_and_plot(lambda=5)
```



```
Out[6]: ('w[:10]:',
 [[ 0.01604074 -0.00429949 -0.02734855  0.00608798  0.00674389  0.0158298
   0.00823366  0.01440063  0.01973089 -0.0029494 ]],
 <NDArray 1x10 @cpu(0)>, 'b:',
 [-0.15303984]
 <NDArray 1 @cpu(0)>)
```

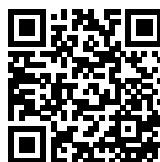
### 3.11.3 小结

- 我们可以使用正则化来应对过拟合问题。
- $L_2$  范数正则化通常会使学到的权重参数的元素较接近 0。
- $L_2$  范数正则化也叫权重衰减。

### 3.11.4 练习

- 除了正则化、增大训练量、以及使用复杂度合适的模型，你还能想到哪些办法可以应对过拟合现象？
- 如果你了解贝叶斯统计，你觉得  $L_2$  范数正则化对应贝叶斯统计里的哪个重要概念？

### 3.11.5 扫码直达讨论区



## 3.12 正则化——使用 Gluon

本节将介绍如何使用 Gluon 实现上一节介绍的正则化。导入实验所需的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import data as gdata, loss as gloss, nn
```

### 3.12.1 生成数据集

我们使用和上一节完全一样的方法生成数据集。

```
In [2]: n_train = 20
        n_test = 100
        num_inputs = 200
        true_w = nd.ones((num_inputs, 1)) * 0.01
        true_b = 0.05

        features = nd.random.normal(shape=(n_train+n_test, num_inputs))
        labels = nd.dot(features, true_w) + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)
        train_features, test_features = features[:n_train, :], features[n_train:, :]
        train_labels, test_labels = labels[:n_train], labels[n_train:]

        num_epochs = 10
        learning_rate = 0.003
        batch_size = 1
        train_iter = gdata.DataLoader(gdata.ArrayDataset(
```

```
    train_features, train_labels), batch_size, shuffle=True)
loss = gloss.L2Loss()
```

### 3.12.2 定义训练和测试

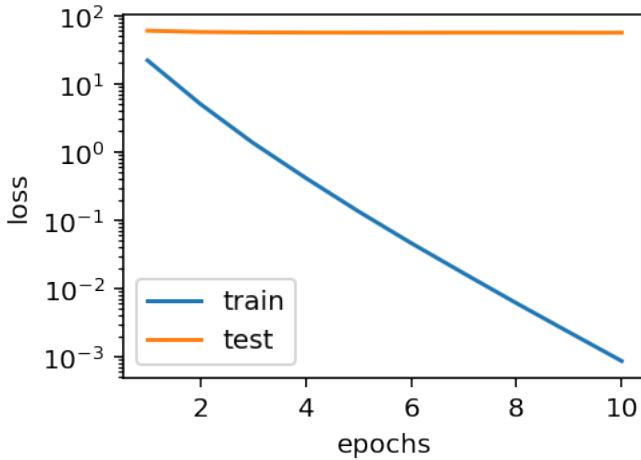
在训练和测试的定义中，我们分别定义了两个 Trainer 实例。其中一个对权重参数做  $L_2$  范数正则化，另一个并没有对偏差参数做正则化。我们在上一节也提到了，实际中有时也对偏差参数做正则化。这样只需要定义一个 Trainer 实例就可以了。

```
In [3]: def fit_and_plot(weight_decay):
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize(init.Normal(sigma=1))
    # 对权重参数做  $L_2$  范数正则化，即权重衰减。
    trainer_w = gluon.Trainer(net.collect_params('.*weight'), 'sgd', {
        'learning_rate': learning_rate, 'wd': weight_decay})
    # 不对偏差参数做  $L_2$  范数正则化。
    trainer_b = gluon.Trainer(net.collect_params('.*bias'), 'sgd', {
        'learning_rate': learning_rate})
    train_ls = []
    test_ls = []
    for _ in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
                l.backward()
            # 对两个 Trainer 实例分别调用 step 函数。
            trainer_w.step(batch_size)
            trainer_b.step(batch_size)
        train_ls.append(loss(net(train_features),
                             train_labels).mean().asscalar())
        test_ls.append(loss(net(test_features),
                            test_labels).mean().asscalar())
    gb.semilogy(range(1, num_epochs+1), train_ls, 'epochs', 'loss',
                range(1, num_epochs+1), test_ls, ['train', 'test'])
    return 'w[:10]:', net[0].weight.data()[:, :10], 'b:', net[0].bias.data()
```

### 3.12.3 观察实验结果

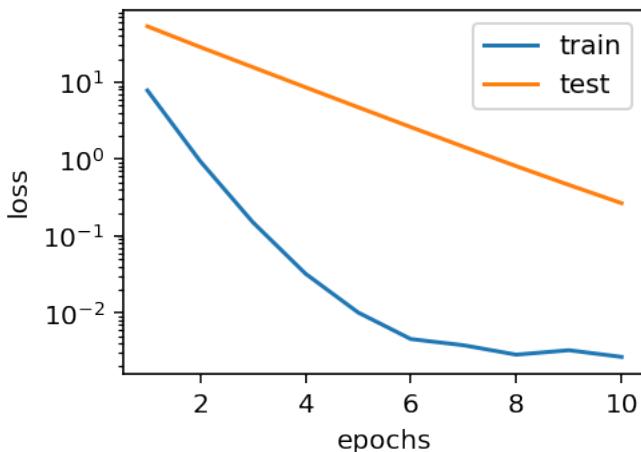
以下实验结果和上一节中的类似。同样地，使用正则化可以从一定程度上缓解过拟合问题。

```
In [4]: fit_and_plot(0)
```



```
Out[4]: ('w[:10]:',
 [[-0.81725901 -0.22302632 -0.7489292 -2.25925446 -0.10191265 -0.69825613
 -0.09722897 0.60890985 -1.10465002 -1.44829524]]
<NDArray 1x10 @cpu(0)>, 'b:',
 [-0.38644427]
<NDArray 1 @cpu(0)>)
```

```
In [5]: fit_and_plot(5)
```



```
Out[5]: ('w[:10]:',
 [[ 0.01571397 -0.0043435 -0.02749268 0.00553531 0.00681707 0.01545126
```

```
0.00819598 0.01347636 0.01890009 -0.00176157]]  
<NDArray 1x10 @cpu(0)>, 'b:',  
[-0.13182944]  
<NDArray 1 @cpu(0)>)
```

### 3.12.4 小结

- 使用 Gluon 的 `wd` 超参数可以使用正则化来应对过拟合问题。
- 我们可以定义多个 `Trainer` 实例对不同的模型参数使用不同的迭代方法。

### 3.12.5 练习

- 调一调本节实验中的 `wd` 超参数。观察并分析实验结果。

### 3.12.6 扫码直达讨论区



## 3.13 丢弃法——从零开始

除了前两节介绍的权重衰减以外，深度学习模型常常使用丢弃法（dropout）来应对过拟合问题。丢弃法有一些不同的变体。本节中提到的丢弃法特指倒置丢弃法（inverted dropout）。它被广泛使用于深度学习。

### 3.13.1 方法和原理

为了确保测试模型的确定性，丢弃法的使用只发生在训练模型时，并非测试模型时。当神经网络中的某一层使用丢弃法时，该层的神经元将有一定概率被丢弃掉。设丢弃概率为  $p$ 。具体来说，该

层任一神经元在应用激活函数后，有  $p$  的概率自乘 0，有  $1 - p$  的概率自除以  $1 - p$  做拉伸。丢弃概率是丢弃法的超参数。

我们在“多层神经网络”一节的图 3.3 中描述了一个未使用丢弃法的多层感知机。假设其中隐藏单元  $h_i$  ( $i = 1, \dots, 5$ ) 的计算表达式为

$$h_i = \phi(x_1 w_1^{(i)} + x_2 w_2^{(i)} + x_3 w_3^{(i)} + x_4 w_4^{(i)} + b^{(i)}),$$

其中  $\phi$  是激活函数， $x_1, \dots, x_4$  是输入， $w_1^{(i)}, \dots, w_4^{(i)}$  是权重参数， $b^{(i)}$  是偏差参数。设丢弃概率为  $p$ ，并设随机变量  $\xi_i$  有  $p$  概率为 0，有  $1 - p$  概率为 1。那么，使用丢弃法的隐藏单元  $h_i$  的计算表达式变为

$$h_i = \frac{\xi_i}{1-p} \phi(x_1 w_1^{(i)} + x_2 w_2^{(i)} + x_3 w_3^{(i)} + x_4 w_4^{(i)} + b^{(i)}).$$

注意到测试模型时不使用丢弃法。由于  $\mathbb{E}(\xi_i) = 1 - p$ ，同一神经元在模型训练和测试时的输出值的期望不变。

让我们对图 3.3 中的隐藏层使用丢弃法，一种可能的结果如图 3.5 所示。

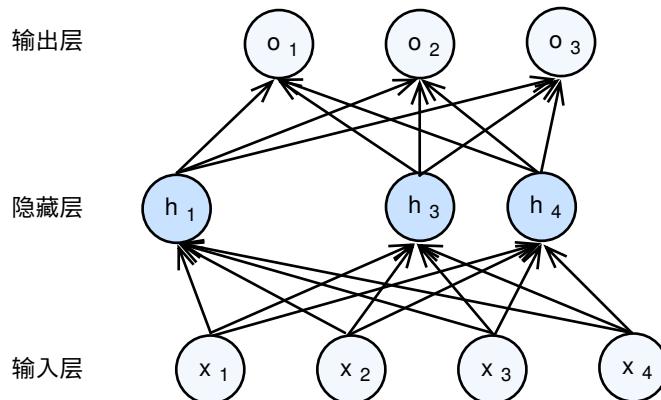


图 3.5：隐藏层使用了丢弃法的多层感知机

以图 3.5 为例，每次训练迭代时，隐藏层中每个神经元都有可能被丢弃，即  $h_i$  ( $i = 1, \dots, 5$ ) 都有可能为 0。因此，输出层每个单元的计算，例如  $o_1 = \phi(h_1 w'_1 + h_2 w'_2 + h_3 w'_3 + h_4 w'_4 + h_5 w'_5 + b')$ ，都无法过分依赖  $h_1, \dots, h_5$  中的任一个。这样通常会造成  $o_1$  表达式中的权重参数  $w'_1, \dots, w'_5$  都接近 0。因此，丢弃法可以起到正则化的作用，并可以用来应对过拟合。

### 3.13.2 实现丢弃法

根据丢弃法的定义，我们可以很容易地实现丢弃法。下面的 `dropout` 函数将以 `drop_prob` 的概率丢弃 NDArray 输入 `X` 中的元素。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, gluon, nd
        from mxnet.gluon import loss as gloss

        def dropout(X, drop_prob):
            assert 0 <= drop_prob <= 1
            keep_prob = 1 - drop_prob
            # 这种情况下把全部元素都丢弃。
            if keep_prob == 0:
                return X.zeros_like()
            mask = nd.random.uniform(0, 1, X.shape) < keep_prob
            return mask * X / keep_prob
```

我们运行几个例子来验证一下 `dropout` 函数。

```
In [2]: X = nd.arange(20).reshape((5, 4))
        dropout(X, 0)
```

Out[2]:

```
[[ 0.   1.   2.   3.]
 [ 4.   5.   6.   7.]
 [ 8.   9.  10.  11.]
 [ 12.  13.  14.  15.]
 [ 16.  17.  18.  19.]]
<NDArray 5x4 @cpu(0)>
```

```
In [3]: dropout(X, 0.5)
```

Out[3]:

```
[[ 0.   0.   0.   6.]
 [ 0.  10.   0.   0.]
 [ 16.  18.  20.   0.]
 [ 24.  26.   0.   0.]
 [ 0.  34.   0.   0.]]
<NDArray 5x4 @cpu(0)>
```

```
In [4]: dropout(X, 1)
```

Out[4]:

```
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
<NDArray 5x4 @cpu(0)>
```

### 3.13.3 定义模型参数

实验中，我们依然使用“Softmax 回归——从零开始”一节中介绍的 Fashion-MNIST 数据集。我们将定义一个包含两个隐藏层的多层感知机。其中两个隐藏层的输出个数都是 256。

```
In [5]: num_inputs = 784
       num_outputs = 10
       num_hiddens1 = 256
       num_hiddens2 = 256

       W1 = nd.random.normal(scale=0.01, shape=(num_inputs, num_hiddens1))
       b1 = nd.zeros(num_hiddens1)
       W2 = nd.random.normal(scale=0.01, shape=(num_hiddens1, num_hiddens2))
       b2 = nd.zeros(num_hiddens2)
       W3 = nd.random.normal(scale=0.01, shape=(num_hiddens2, num_outputs))
       b3 = nd.zeros(num_outputs)

       params = [W1, b1, W2, b2, W3, b3]
       for param in params:
           param.attach_grad()
```

### 3.13.4 定义模型

我们的模型就是将全连接层和激活函数 ReLU 串起来，并对激活函数的输出使用丢弃法。我们可以分别设置各个层的丢弃概率。通常，建议把靠近输入层的丢弃概率设的小一点。在这个实验中，我们把第一个隐藏层的丢弃概率设为 0.2，把第二个隐藏层的丢弃概率设为 0.5。我们只需在训练模型时使用丢弃法。

```
In [6]: drop_prob1 = 0.2
       drop_prob2 = 0.5

       def net(X):
           X = X.reshape((-1, num_inputs))
           H1 = (nd.dot(X, W1) + b1).relu()
           H2 = (nd.dot(H1, W2) + b2).relu()
           H3 = (nd.dot(H2, W3) + b3).relu()
           return H3
```

```
# 只在训练模型时使用丢弃法。
if autograd.is_training():
    # 在第一层全连接后添加丢弃层。
    H1 = dropout(H1, drop_prob1)
H2 = (nd.dot(H1, W2) + b2).relu()
if autograd.is_training():
    # 在第二层全连接后添加丢弃层。
    H2 = dropout(H2, drop_prob2)
return nd.dot(H2, W3) + b3
```

### 3.13.5 训练和测试模型

这部分和之前多层感知机的训练与测试类似。

```
In [7]: num_epochs = 5
lr = 0.5
batch_size = 256
loss = gloss.SoftmaxCrossEntropyLoss()
train_iter, test_iter = gb.load_data_fashion_mnist(batch_size)
gb.train_cpu(net, train_iter, test_iter, loss, num_epochs, batch_size, params,
            lr)

epoch 1, loss 1.1666, train acc 0.553, test acc 0.752
epoch 2, loss 0.5889, train acc 0.782, test acc 0.832
epoch 3, loss 0.4958, train acc 0.818, test acc 0.831
epoch 4, loss 0.4507, train acc 0.837, test acc 0.852
epoch 5, loss 0.4217, train acc 0.846, test acc 0.855
```

### 3.13.6 小结

- 我们可以通过使用丢弃法应对过拟合。
- 只需在训练模型时使用丢弃法。

### 3.13.7 练习

- 尝试不使用丢弃法，看看这个包含两个隐藏层的多层感知机可以得到什么结果。
- 如果把本节中的两个丢弃概率超参数对调，会有什么结果？

### 3.13.8 扫码直达讨论区



## 3.14 丢弃法——使用 Gluon

本节中，我们将上一节的实验代码用 Gluon 实现一遍。你会发现代码将精简很多。

### 3.14.1 定义模型并添加丢弃层

在多层感知机中 Gluon 实现的基础上，我们只需要在全连接层后添加 Dropout 实例并指定丢弃概率。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import loss as gloss, nn

        drop_prob1 = 0.2
        drop_prob2 = 0.5

        net = nn.Sequential()
        net.add(nn.Flatten())
        net.add(nn.Dense(256, activation="relu"))
        # 在第一个全连接层后添加丢弃层。
        net.add(nn.Dropout(drop_prob1))
        net.add(nn.Dense(256, activation="relu"))
        # 在第二个全连接层后添加丢弃层。
        net.add(nn.Dropout(drop_prob2))
        net.add(nn.Dense(10))
        net.initialize(init.Normal(sigma=0.01))
```

### 3.14.2 训练和测试模型

这部分依然和多层感知机中的训练和测试没有多少区别。

```
In [2]: num_epochs = 5
batch_size = 256
loss = gloss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.5})
train_iter, test_iter = gb.load_data_fashion_mnist(batch_size)
gb.train_cpu(net, train_iter, test_iter, loss, num_epochs, batch_size,
             None, None, trainer)

epoch 1, loss 1.2764, train acc 0.510, test acc 0.704
epoch 2, loss 0.6164, train acc 0.770, test acc 0.822
epoch 3, loss 0.5150, train acc 0.814, test acc 0.838
epoch 4, loss 0.4663, train acc 0.829, test acc 0.853
epoch 5, loss 0.4380, train acc 0.841, test acc 0.862
```

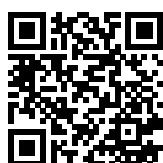
### 3.14.3 小结

- 使用 Gluon，我们可以更方便地构造多层神经网络并使用丢弃法。

### 3.14.4 练习

- 尝试不同丢弃概率超参数组合，观察并分析结果。

### 3.14.5 扫码直达讨论区



## 3.15 正向传播和反向传播

我们一直使用优化算法来训练深度学习模型，例如小批量随机梯度下降。实际上，优化算法通常都会依赖模型参数梯度的计算。然而，在深度学习模型中，由于网络结构的复杂性，模型参数梯度的计算往往并不直观。虽然我们可以通过 MXNet 轻松获取模型参数的梯度，但了解它们的计算将有助于我们进一步理解深度学习模型训练的本质。

### 3.15.1 概念

反向传播（back-propagation）是计算神经网络参数梯度的方法。总的来说，反向传播中会依据微积分中的链式法则，沿着从输出层到输入层的顺序，依次计算并存储损失函数有关神经网络各层的中间变量以及参数的梯度。

反向传播对于各层变量和参数的梯度计算可能会依赖各层变量和参数的当前值。对神经网络沿着从输入层到输出层的顺序，依次计算并存储模型中间变量的过程叫做正向传播（forward propagation）。

### 3.15.2 案例分析——正则化的多层感知机

为了解释正向传播和反向传播，我们以一个简单的  $L_2$  范数正则化的多层感知机为例。

#### 定义模型

考虑类别数为  $y$  的分类问题。给定一个特征为  $\mathbf{x} \in \mathbb{R}^x$  和标签为离散值  $y'$  的训练数据样本。不考虑偏差项，我们可以得到中间变量

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x},$$

其中  $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times x}$  是模型参数。中间变量  $\mathbf{z} \in \mathbb{R}^h$  应用按元素操作的激活函数  $\phi$  后将得到向量长度为  $h$  的隐藏层变量

$$\mathbf{h} = \phi(\mathbf{z}).$$

隐藏层  $\mathbf{h} \in \mathbb{R}^h$  也是一个中间变量。通过模型参数  $\mathbf{W}^{(2)} \in \mathbb{R}^{y \times h}$  可以得到向量长度为  $y$  的输出层变量

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$$

假设损失函数为  $\ell$ , 我们可以计算出单个数据样本的损失项

$$L = \ell(\mathbf{o}, y').$$

根据  $L_2$  范数正则化的定义, 给定超参数  $\lambda$ , 正则化项即

$$s = \frac{\lambda}{2} (\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2),$$

其中每个矩阵 Frobenius 范数的平方项即该矩阵元素的平方和。最终, 模型在给定的数据样本上带正则化的损失为

$$J = L + s.$$

我们将  $J$  叫做有关给定数据样本的目标函数, 并在以下的讨论中简称目标函数。

### 模型计算图

为了可视化模型变量和参数之间在计算中的依赖关系, 我们可以绘制模型计算图, 如图 3.6 所示。例如, 正则化项  $s$  的计算依赖模型参数  $\mathbf{W}^{(1)}$  和  $\mathbf{W}^{(2)}$ 。

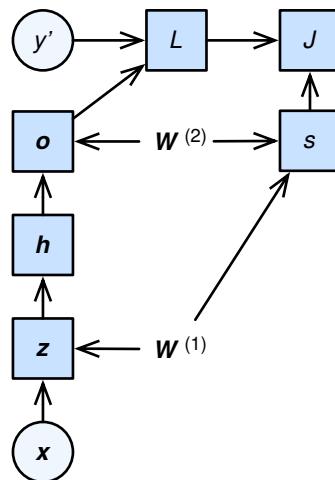


图 3.6: 正则化的多层感知机模型计算中的依赖关系。方框中字母代表变量, 圆圈中字母代表数据样本特征和标签, 无边框的字母代表模型参数。

## 正向传播

在反向传播计算梯度之前，我们先做一次正向传播。也就是说，按照图 3.6 中箭头顺序，并根据模型参数的当前值，依次计算并存储模型中各个中间变量的值。例如，在计算损失项  $L$  之前，我们需要依次计算并存储  $z, h, o$  的值。

## 反向传播

刚刚提到，图 3.6 中模型的参数是  $\mathbf{W}^{(1)}$  和  $\mathbf{W}^{(2)}$ 。根据“单层神经网络”一节中定义的小批量随机梯度下降，我们需要对小批量中每个样本目标函数  $J$  关于  $\mathbf{W}^{(1)}$  和  $\mathbf{W}^{(2)}$  的梯度求平均来迭代  $\mathbf{W}^{(1)}$  和  $\mathbf{W}^{(2)}$ 。也就是说，每一次迭代都需要计算模型参数梯度  $\partial J / \partial \mathbf{W}^{(1)}$  和  $\partial J / \partial \mathbf{W}^{(2)}$ 。根据图 3.6 中的依赖关系，我们可以按照其中箭头所指的反方向依次计算并存储梯度。

为了表述方便，对输入输出  $X, Y, Z$  为任意形状张量的函数  $Y = f(X)$  和  $Z = g(Y)$ ，我们使用

$$\frac{\partial Z}{\partial X} = \text{prod}\left(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X}\right)$$

来表达链式法则。

首先，我们计算目标函数有关损失项和有关正则项的梯度

$$\frac{\partial J}{\partial L} = 1,$$

$$\frac{\partial J}{\partial s} = 1.$$

其次，我们依据链式法则计算目标函数有关输出层变量的梯度  $\partial J / \partial o \in \mathbb{R}^y$ :

$$\frac{\partial J}{\partial o} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial o}\right) = \frac{\partial L}{\partial o}.$$

正则项有关两个参数的梯度可以很直观地计算：

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)},$$

$$\frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}.$$

现在我们可以计算最靠近输出层的模型参数的梯度  $\partial J / \partial \mathbf{W}^{(2)} \in \mathbb{R}^{y \times h}$ 。在图 3.6 中， $J$  分别通过  $o$  和  $s$  依赖  $\mathbf{W}^{(2)}$ 。依据链式法则，我们有

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial o}, \frac{\partial o}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial o} h^\top + \lambda \mathbf{W}^{(2)}.$$

沿着输出层向隐藏层继续反向传播，隐藏层变量的梯度  $\partial J / \partial \mathbf{h} \in \mathbb{R}^h$  可以这样计算：

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = \mathbf{W}^{(2)\top} \frac{\partial J}{\partial \mathbf{o}}.$$

注意到激活函数  $\phi$  是按元素操作的，中间变量  $\mathbf{z}$  的梯度  $\partial J / \partial \mathbf{z} \in \mathbb{R}^h$  的计算需要使用按元素乘法符  $\odot$ ：

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z}).$$

最终，我们可以得到最靠近输入层的模型参数的梯度  $\partial J / \partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times x}$ 。在图 3.6 中， $J$  分别通过  $\mathbf{z}$  和  $s$  依赖  $\mathbf{W}^{(1)}$ 。依据链式法则，我们有

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}.$$

需要再次提醒的是，每次迭代中，上述各个依次计算出的梯度会被依次存储或更新。这是为了避免重复计算。例如，由于输出层变量梯度  $\partial J / \partial \mathbf{o}$  被计算存储，反向传播稍后的参数梯度  $\partial J / \partial \mathbf{W}^{(2)}$  和隐藏层变量梯度  $\partial J / \partial \mathbf{h}$  的计算可以直接读取输出层变量梯度的值，而无需重复计算。

### 正向传播和反向传播相互依赖

事实上，正向传播和反向传播相互依赖。

一方面，正向传播的计算可能依赖模型参数的当前值。而这些模型参数是在反向传播的梯度计算后通过优化算法迭代的。例如，图 3.6 中，计算正则化项  $s$  依赖模型参数  $\mathbf{W}^{(1)}$  和  $\mathbf{W}^{(2)}$  的当前值。而这些当前值是优化算法最近一次根据反向传播算出梯度后迭代得到的。

另一方面，反向传播的梯度计算可能依赖各变量的当前值。而这些变量的当前值是通过正向传播计算的。举例来说，参数梯度  $\partial J / \partial \mathbf{W}^{(2)}$  的计算需要依赖隐藏层变量的当前值  $\mathbf{h}$ 。这个当前值是通过从输入层到输出层的正向传播计算并存储得到的。

因此，在模型参数初始化完成后，我们可以交替进行正向传播和反向传播，并根据反向传播计算的梯度迭代模型参数。

### 3.15.3 小结

- 反向传播沿着从输出层到输入层的顺序，依次计算并存储神经网络中间变量和参数的梯度。
- 正向传播沿着从输入层到输出层的顺序，依次计算并存储神经网络的中间变量。
- 正向传播和反向传播相互依赖。

### 3.15.4 练习

- 学习了本节内容后，你是否能解释“[多层神经网络](#)”一节中提到的层数较多时梯度可能会衰减或爆炸的原因？

### 3.15.5 扫码直达讨论区



## 3.16 实战 Kaggle 比赛：预测房价

作为深度学习基础篇章的总结，我们将对本章内容学以致用。下面，让我们动手实战一个 Kaggle 比赛：预测房价。

在这个房价预测比赛中，我们还将以 `pandas` 为工具介绍如何对真实世界中的数据进行重要的预处理，例如：

- 处理离散数据；
- 处理丢失的数据特征；
- 对数据进行标准化。

需要注意的是，本节中对于数据的预处理、模型的设计和超参数的选择等，我们特意只提供最基础的版本。我们希望大家通过动手实战、仔细观察实验现象、认真分析实验结果并不断调整方法，从而得到令自己满意的结果。

### 3.16.1 Kaggle 比赛

Kaggle 是一个著名的供机器学习爱好者交流的平台 [1]。图 3.7 展示了 Kaggle 网站首页。为了便于提交结果，请大家注册 Kaggle 账号。

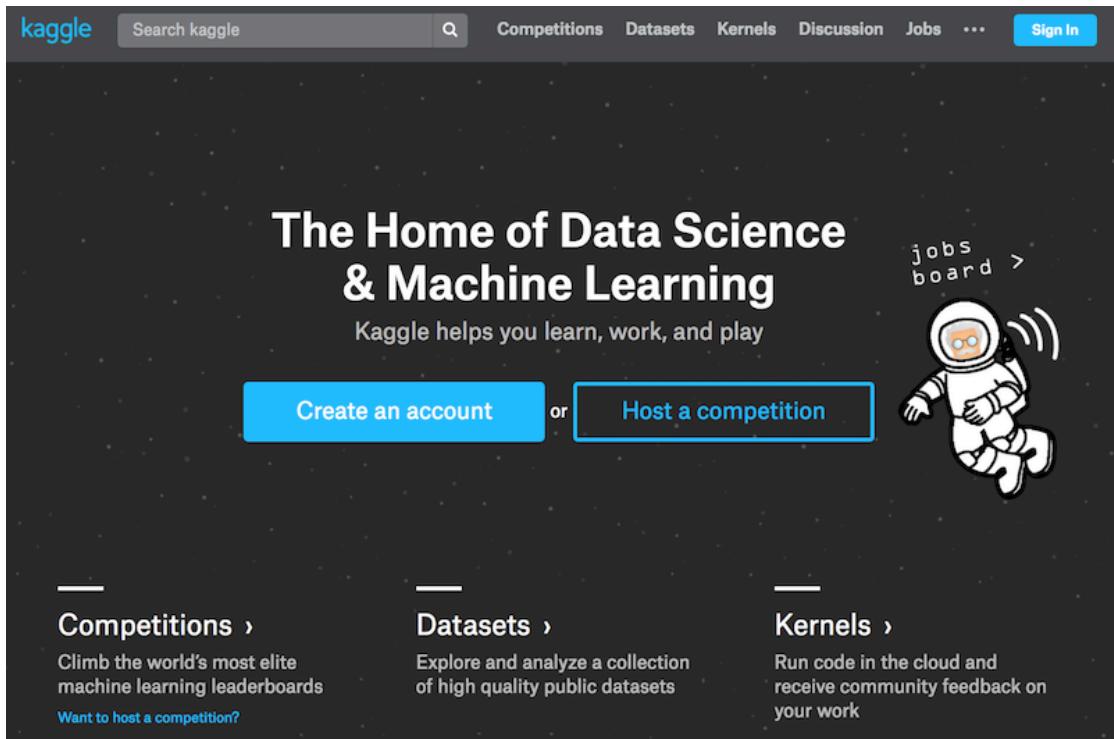


图 3.7: Kaggle 网站首页

我们可以在预测房价比赛的网页上了解比赛信息和参赛者成绩、下载数据集并提交自己的预测结果 [2]。图 3.8 展示了预测房价比赛的网页信息。



## House Prices: Advanced Regression Techniques

Predict sales prices and practice feature engineering, RFs, and gradient boosting  
1,698 teams · 2 years to go

[Overview](#) [Data](#) [Kernels](#) [Discussion](#) [Leaderboard](#) [Rules](#)

**Overview**

Description	Start here if...
<a href="#">Evaluation</a>	You have some experience with R or Python and machine learning basics. This is a perfect competition for data science students who have completed an online course in machine learning and are looking to expand their skill set before trying a featured competition.
<a href="#">Frequently Asked Questions</a>	
<a href="#">Tutorials</a>	
	<b>Competition Description</b>
	 <p>Ask a home buyer to describe their dream house, and they probably won't begin with the height of the basement ceiling or the proximity to an east-west railroad. But this playground competition's dataset proves that much more influences price negotiations than the number of bedrooms or a white-picket fence.</p> <p>With 79 explanatory variables describing (almost) every aspect of residential homes in Ames, Iowa, this competition challenges you to predict the final price of each home.</p>

图 3.8: 预测房价比赛的网页信息

### 3.16.2 获取和读取数据集

比赛数据分为训练数据集和测试数据集。两个数据集都包括每栋房子的特征，例如街道类型、建造年份、房顶类型、地下室状况等特征值。这些特征值有连续的数字、离散的标签甚至是缺失值“na”。只有训练数据集包括了每栋房子的价格。我们可以访问比赛网页，点击“Data”标签，并下载这些数据集 [2]。

下面，我们通过使用 `pandas` 读入数据。请确保已安装 `pandas` (命令行执行 “`pip install pandas`”)。

```
In [1]: import sys
        sys.path.append('..')
```

```
import gluonbook as gb
from mxnet import autograd, init, gluon, nd
from mxnet.gluon import data as gdata, loss as gloss, nn
import numpy as np
import pandas as pd

train_data = pd.read_csv("../data/kaggle_house_pred_train.csv")
test_data = pd.read_csv("../data/kaggle_house_pred_test.csv")
all_features = pd.concat((train_data.loc[:, 'MSSubClass':'SaleCondition'],
                           test_data.loc[:, 'MSSubClass':'SaleCondition']))
```

训练数据集包括 1460 个样本、80 个特征和 1 个标签。

```
In [2]: train_data.shape
```

```
Out[2]: (1460, 81)
```

测试数据集包括 1459 个样本和 80 个特征。我们需要预测测试数据集上每个样本的标签。

```
In [3]: test_data.shape
```

```
Out[3]: (1459, 80)
```

### 3.16.3 预处理数据

我们对连续数值的特征做标准化处理。如果一个特征的值是连续的，设该特征在训练数据集和测试数据集上的均值为  $\mu$ ，标准差为  $\sigma$ 。那么，该特征的每个值将先减去  $\mu$  再除以  $\sigma$ 。

```
In [4]: numeric_features = all_features.dtypes[all_features.dtypes != "object"].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
```

现在，对离散数值的特征进一步处理，并把缺失数据值用本特征的平均值估计。

```
In [5]: all_features = pd.get_dummies(all_features, dummy_na=True)
all_features = all_features.fillna(all_features.mean())
```

下面把数据转换一下格式。

```
In [6]: n_train = train_data.shape[0]
train_features = all_features[:n_train].as_matrix()
test_features = all_features[n_train:].as_matrix()
train_labels = train_data.SalePrice.as_matrix()

/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/ipykernel_l
↳ auncher.py:2: FutureWarning: Method .as_matrix will be removed in a future
↳ version. Use .values instead.
```

```
/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/ipykernel_lj
↳  auncher.py:3: FutureWarning: Method .as_matrix will be removed in a future
↳  version. Use .values instead.
This is separate from the ipykernel package so we can avoid doing imports until
/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/ipykernel_lj
↳  auncher.py:4: FutureWarning: Method .as_matrix will be removed in a future
↳  version. Use .values instead.
after removing the cwd from sys.path.
```

### 3.16.4 导入 NDArray 格式数据

为了便于和 Gluon 交互，我们需要导入 NDArray 格式数据。

```
In [7]: train_features = nd.array(train_features)
train_labels = nd.array(train_labels)
train_labels.reshape((n_train, 1))
test_features = nd.array(test_features)
```

我们使用平方损失函数训练模型，并定义比赛用来评价模型的函数。

```
In [8]: loss = gloss.L2Loss()
def get_rmse_log(net, train_features, train_labels):
    clipped_preds = nd.clip(net(train_features), 1, float('inf'))
    return nd.sqrt(2 * loss(clipped_preds.log(),
                           train_labels.log().mean()).asnumpy())
```

### 3.16.5 定义模型

我们将模型的定义放在一个函数里供多次调用。这是一个基本的线性回归模型，并使用了 Xavier 随机初始化。

```
In [9]: def get_net():
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize(init=init.Xavier())
    return net
```

### 3.16.6 定义训练函数

下面定义模型的训练函数。和本章中前几节不同，这里使用了 Adam 优化算法。我们将在之后的“优化算法”一章里详细介绍它。

```
In [10]: def train(net, train_features, train_labels, test_features, test_labels,
             num_epochs, verbose_epoch, learning_rate, weight_decay, batch_size):
    train_ls = []
    if test_features is not None:
        test_ls = []
    train_iter = gdata.DataLoader(gdata.ArrayDataset(
        train_features, train_labels), batch_size, shuffle=True)
    # 这里使用了 Adam 优化算法。
    trainer = gluon.Trainer(net.collect_params(), 'adam', {
        'learning_rate': learning_rate, 'wd': weight_decay})
    net.initialize(init=init.Xavier(), force_reinit=True)
    for epoch in range(1, num_epochs + 1):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
                l.backward()
                trainer.step(batch_size)
                cur_train_l = get_rmse_log(net, train_features, train_labels)
        if epoch >= verbose_epoch:
            print("epoch %d, train loss: %f" % (epoch, cur_train_l))
        train_ls.append(cur_train_l)
        if test_features is not None:
            cur_test_l = get_rmse_log(net, test_features, test_labels)
            test_ls.append(cur_test_l)
        if test_features is not None:
            gb.semilogy(range(1, num_epochs+1), train_ls, 'epochs', 'loss',
                        range(1, num_epochs+1), test_ls, ['train', 'test'])
        else:
            gb.semilogy(range(1, num_epochs+1), train_ls, 'epochs', 'loss')
        if test_features is not None:
            return cur_train_l, cur_test_l
        else:
            return cur_train_l
```

### 3.16.7 定义 $K$ 折交叉验证

“欠拟合、过拟合、选择模型和调节超参数”一节中介绍了 $K$ 折交叉验证。下面定义了 $K$ 折交

叉验证函数。我们将根据  $K$  折交叉验证的结果选择模型设计并调参。

```
In [11]: def k_fold_cross_valid(k, epochs, verbose_epoch, X_train, y_train,
                                learning_rate, weight_decay, batch_size):
    assert k > 1
    fold_size = X_train.shape[0] // k
    train_l_sum = 0.0
    test_l_sum = 0.0
    for test_i in range(k):
        X_val_test = X_train[test_i * fold_size: (test_i + 1) * fold_size, :]
        y_val_test = y_train[test_i * fold_size: (test_i + 1) * fold_size]
        val_train_defined = False
        for i in range(k):
            if i != test_i:
                X_cur_fold = X_train[i * fold_size: (i + 1) * fold_size, :]
                y_cur_fold = y_train[i * fold_size: (i + 1) * fold_size]
                if not val_train_defined:
                    X_val_train = X_cur_fold
                    y_val_train = y_cur_fold
                    val_train_defined = True
                else:
                    X_val_train = nd.concat(X_val_train, X_cur_fold, dim=0)
                    y_val_train = nd.concat(y_val_train, y_cur_fold, dim=0)
        net = get_net()
        train_l, test_l = train(
            net, X_val_train, y_val_train, X_val_test, y_val_test,
            epochs, verbose_epoch, learning_rate, weight_decay, batch_size)
        train_l_sum += train_l
        print("test loss: %f" % test_l)
        test_l_sum += test_l
    return train_l_sum / k, test_l_sum / k
```

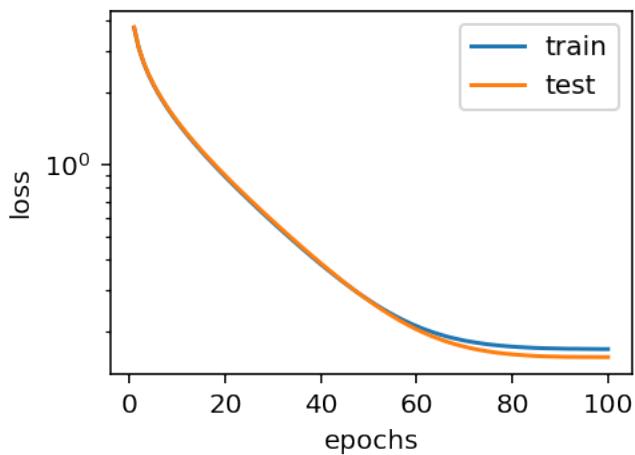
### 3.16.8 交叉验证模型

现在，我们可以交叉验证模型了。以下的超参数都是可以调节的。

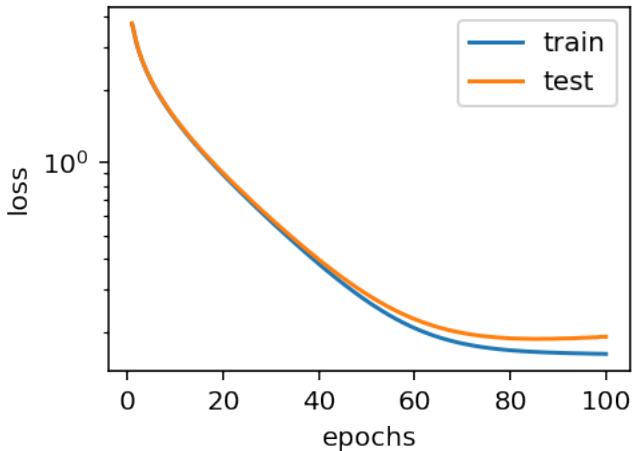
```
In [12]: k = 5
num_epochs = 100
verbose_epoch = num_epochs - 2
lr = 5
weight_decay = 0
batch_size = 64
```

```
train_l, test_l = k_fold_cross_valid(k, num_epochs, verbose_epoch,
                                      train_features, train_labels, lr,
                                      weight_decay, batch_size)
print("%d-fold validation: avg train loss: %f, avg test loss: %f"
      % (k, train_l, test_l))

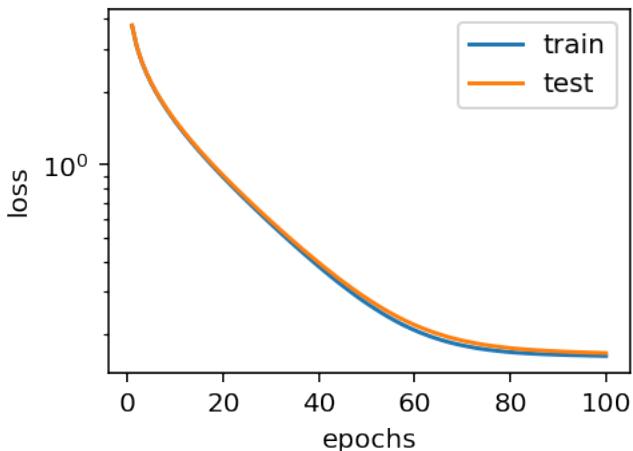
epoch 98, train loss: 0.169431
epoch 99, train loss: 0.169301
epoch 100, train loss: 0.169354
```



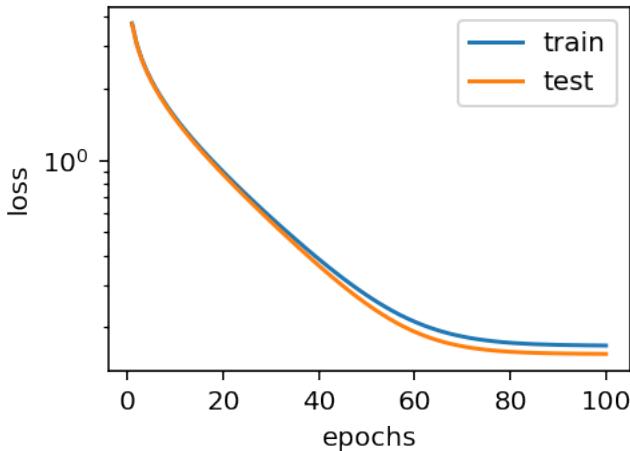
```
test loss: 0.156811
epoch 98, train loss: 0.162526
epoch 99, train loss: 0.162502
epoch 100, train loss: 0.162293
```



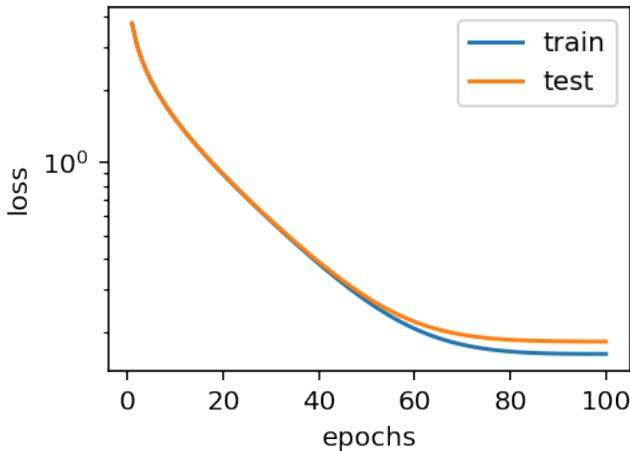
```
test loss: 0.190973
epoch 98, train loss: 0.163851
epoch 99, train loss: 0.163660
epoch 100, train loss: 0.163541
```



```
test loss: 0.168012
epoch 98, train loss: 0.167868
epoch 99, train loss: 0.167789
epoch 100, train loss: 0.167666
```



```
test loss: 0.154682
epoch 98, train loss: 0.162814
epoch 99, train loss: 0.162691
epoch 100, train loss: 0.162735
```



```
test loss: 0.182821
5-fold validation: avg train loss: 0.165118, avg test loss: 0.170660
```

在设定了一组参数后，即便训练误差可以达到很低，但是  $K$  折交叉验证上的误差可能更高。这很可能是由于过拟合造成的。因此，当训练误差特别低时，要观察  $K$  折交叉验证上的误差是否同时降低。

### 3.16.9 预测并在 Kaggle 提交结果

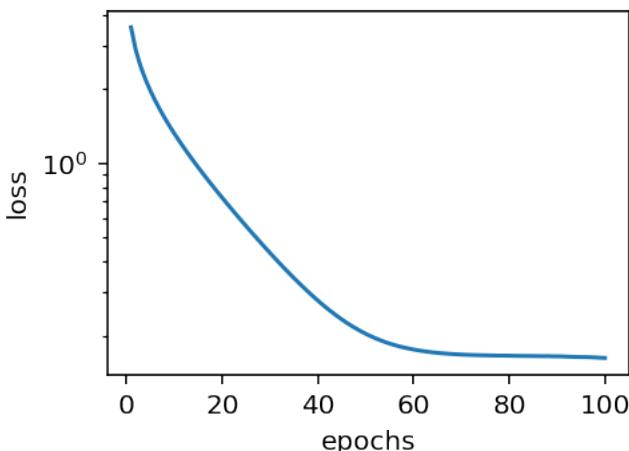
我们首先定义预测函数。在预测之前，我们会使用完整的训练数据集来重新训练模型。

```
In [13]: def train_and_pred(num_epochs, verbose_epoch, train_features, test_feature,
                           train_labels, test_data, lr, weight_decay, batch_size):
    net = get_net()
    train(net, train_features, train_labels, None, None, num_epochs,
          verbose_epoch, lr, weight_decay, batch_size)
    preds = net(test_features).asnumpy()
    test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
    submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
    submission.to_csv('submission.csv', index=False)
```

设计好模型并调好超参数以后，让我们对测试数据集上的房屋样本做价格预测，并在 Kaggle 上提交结果。

```
In [14]: train_and_pred(num_epochs, verbose_epoch, train_features, test_features,
                           train_labels, test_data, lr, weight_decay, batch_size)

epoch 98, train loss: 0.163319
epoch 99, train loss: 0.162830
epoch 100, train loss: 0.162577
```



执行完上述代码后，会生成一个“submission.csv”文件。这个文件符合 Kaggle 比赛要求的提交格式。这时我们可以在 Kaggle 上把我们预测得出的结果提交并查看与测试数据集上真实房价（标签）的误差。你需要登录 Kaggle 网站，访问预测房价比赛网页，并点击右侧“Submit Predictions”或“Late Submission”按钮[2]。然后，点击页面下方“Upload Submission File”选择需要提交

的预测结果文件。最后，点击页面最下方的“Make Submission”按钮就可以查看结果了。如图 3.9 所示。

The screenshot shows the Kaggle submission interface. At the top left, it says "Step 1 Upload submission file". Below this is a dashed-line box containing an "Upload Submission File" button with an upward arrow icon. To its left is the text "File Format: Your submission should be in CSV format. You can upload this in a zip/gz/rar/7z archive, if you prefer." To its right is "Number of Predictions: We expect the solution file to have 1459 prediction rows. This file should have a header row. Please see sample submission file on the data page." Below this section is a horizontal toolbar with icons for bold, italic, code, etc., followed by the text "Styling with Markdown supported". The next section, "Step 2 Describe submission", contains a text area with the placeholder "Briefly describe your submission." At the bottom is a blue "Make Submission" button.

图 3.9: Kaggle 预测房价比赛的预测结果提交页面

### 3.16.10 小结

- 我们通常需要对真实数据做预处理。
- 我们可以使用  $K$  折交叉验证来选择模型并调参。

### 3.16.11 练习

- 在 Kaggle 提交本教程的预测结果。观察一下，这个结果能在 Kaggle 上拿到什么样的分数？
- 对照  $K$  折交叉验证结果，不断修改模型（例如添加隐藏层）和调参，你能提高 Kaggle 上的分数吗？

- 如果不使用本节中对连续数值特征的标准化处理，结果会有什么变化？
- 扫码直达讨论区，在社区交流方法和结果。相信你一定会有收获。

### 3.16.12 扫码直达讨论区



### 3.16.13 参考文献

- [1] Kaggle 网站。<https://www.kaggle.com>
- [2] Kaggle 房价预测比赛网址。<https://www.kaggle.com/c/house-prices-advanced-regression-techniques>



---

## 深度学习计算

---

上一章介绍了包括多层感知机在内的简单深度学习模型的原理和实现。这一章我们将介绍深度学习计算的各个重要组成部分，例如模型构造、参数访问、自定义层和使用 GPU。通过本章的学习，你将能够深入了解模型实现和计算的各个方面，为在之后章节实现更复杂模型打下基础。

### 4.1 模型构造

回忆在“多层感知机——使用 Gluon”一节中我们是如何实现一个单隐藏层感知机。我们首先构造 `Sequential` 实例，然后依次添加两个全连接层。其中第一层的输出大小为 256，即隐藏层单元个数是 256；第二层的输出大小为 10，即输出层单元个数是 10。这个简单例子已经包含了深度学习模型计算的方方面面，接下来的小节我们将围绕这个例子展开。

我们之前都是用了 `Sequential` 类来构造模型。这里我们另外一种基于 `Block` 类的模型构造方法，它让构造模型更加灵活，也将让你能更好的理解 `Sequential` 的运行机制。

### 4.1.1 继承 Block 类来构造模型

Block 类是 `gluon.nn` 里提供的一个模型构造类，我们可以继承它来定义我们想要的模型。例如，我们在这里构造一个同前提到的相同的多层感知机。这里定义的 MLP 类重载了 Block 类的两个函数：`__init__` 和 `forward`.

```
In [1]: from mxnet import nd
        from mxnet.gluon import nn

class MLP(nn.Block):
    # 声明带有模型参数的层，这里我们声明了两个全链接层。
    def __init__(self, **kwargs):
        # 调用 MLP 父类 Block 的构造函数来进行必要的初始化。这样在构造实例时还可以指定
        # 其他函数参数，例如下一节将介绍的模型参数 params。
        super(MLP, self).__init__(**kwargs)
        # 隐藏层。
        self.hidden = nn.Dense(256, activation='relu')
        # 输出层。
        self.output = nn.Dense(10)
        # 定义模型的前向计算，即如何根据输出计算输出。
    def forward(self, x):
        return self.output(self.hidden(x))
```

我们可以实例化 MLP 类得到 `net`, 其使用跟“[多层感知机——使用 Gluon](#)”一节中通过 `Sequential` 类构造的 `net` 一致。下面代码初始化 `net` 并传入输入数据 `x` 做一次前向计算。

```
In [2]: x = nd.random.uniform(shape=(2,20))
        net = MLP()
        net.initialize()
        net(x)

Out[2]:
[[ 0.09543004  0.04614332 -0.00286654 -0.07790349 -0.05130243  0.02942037
   0.08696642 -0.0190793  -0.04122177  0.05088576]
 [ 0.0769287   0.03099705  0.00856576 -0.04467199 -0.06926839  0.09132434
   0.06786595 -0.06187842 -0.03436673  0.04234694]]
<NDArray 2x10 @cpu(0)>
```

其中，`net(x)` 会调用了 `MLP` 继承至 `Block` 的 `__call__` 函数，这个函数将调用 `MLP` 定义的 `forward` 函数来完成前向计算。

我们无需在这里定义反向传播函数，系统将通过自动求导，参考“[自动求梯度](#)”一节，来自动生成 `backward` 函数。

注意到我们不是将 Block 叫做层或者模型之类的名字，这是因为它是一个可以自由组建的部件。它的子类既可以一个层，例如 Gluon 提供的 Dense 类，也可以是一个模型，我们定义的 MLP 类，或者是模型的一个部分，例如我们会在之后介绍的 ResNet 的残差块。我们下面通过两个例子说明它。

### 4.1.2 Sequential 类继承自 Block 类

当模型的前向计算就是简单串行计算模型里面各个层的时候，我们可以将模型定义变得更加简单，这个就是 Sequential 类的目的，它通过 add 函数来添加 Block 子类实例，前向计算时就是将添加的实例逐一运行。下面我们实现一个跟 Sequential 类有相同功能的类，这样你可以看的更加清楚它的运行机制。

```
In [3]: class MySequential(nn.Block):
    def __init__(self, **kwargs):
        super(MySequential, self).__init__(**kwargs)

    def add(self, block):
        # block 是一个 Block 子类实例，假设它有一个独一无二的名字。我们将它保存在
        # Block 类的成员变量 _children 里，其类型是 OrderedDict。当调用
        # initialize 函数时，系统会自动对 _children 里面所有成员初始化。
        self._children[block.name] = block

    def forward(self, x):
        # OrderedDict 保证会按照插入时的顺序便利元素。
        for block in self._children.values():
            x = block(x)
        return x
```

我们用 MySequential 类来实现前面的 MLP 类：

```
In [4]: net = MySequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(x)

Out[4]:
[[ 0.00362228  0.00633332  0.03201144 -0.01369375  0.10336449 -0.03508018
 -0.00032164 -0.01676023  0.06978628  0.01303309]
 [ 0.03871715  0.02608213  0.03544959 -0.02521311  0.11005433 -0.0143066
 -0.03052466 -0.03852827  0.06321152  0.0038594 ]]
<NDArray 2x10 @cpu(0)>
```

你会发现这里 MySequential 类的使用跟“[多层感知机——使用 Gluon](#)”一节中 Sequential 类使用一致。

### 4.1.3 构造复杂的模型

虽然 Sequential 类可以使得模型构造更加简单，不需要定义 `forward` 函数，但直接继承 Block 类可以极大的拓展灵活性。下面我们构造一个稍微复杂点的网络：

1. 在前向计算中使用了 NDArray 函数和 Python 的控制流
2. 多次调用同一层

```
In [5]: class FancyMLP(nn.Block):  
    def __init__(self, **kwargs):  
        super(FancyMLP, self).__init__(**kwargs)  
        # 不会被更新的随机权重。  
        self.rand_weight = nd.random.uniform(shape=(20, 20))  
        self.dense = nn.Dense(20, activation='relu')  
  
    def forward(self, x):  
        x = self.dense(x)  
        # 使用了 nd 包下 relu 和 dot 函数。  
        x = nd.relu(nd.dot(x, self.rand_weight) + 1)  
        # 重用了 dense，等价于两层网络但共享了参数。  
        x = self.dense(x)  
        # 控制流，这里我们需要调用 asscalar 来返回标量进行比较。  
        while x.norm().asscalar() > 1:  
            x /= 2  
        if x.norm().asscalar() < 0.8:  
            x *= 10  
        return x.sum()
```

在这个 FancyMLP 模型中，我们使用了常数权重 `rand_weight`（注意它不是模型参数）、做了矩阵乘法操作（`nd.dot`）并重复使用了相同的 `Dense` 层。测试一下：

```
In [6]: net = FancyMLP()  
net.initialize()  
net(x)  
  
Out[6]:  
[ 18.57195282]  
<NDArray 1 @cpu(0)>
```

由于 FancyMLP 和 Sequential 都是 Block 的子类，我们可以嵌套调用他们。

```
In [7]: class NestMLP(nn.Block):
    def __init__(self, **kwargs):
        super(NestMLP, self).__init__(**kwargs)
        self.net = nn.Sequential()
        self.net.add(nn.Dense(64, activation='relu'),
                    nn.Dense(32, activation='relu'))
        self.dense = nn.Dense(16, activation='relu')

    def forward(self, x):
        return self.dense(self.net(x))

net = nn.Sequential()
net.add(NestMLP(), nn.Dense(20), FancyMLP())

net.initialize()
net(x)

Out[7]:
[ 24.86621094]
<NDArray 1 @cpu(0)>
```

#### 4.1.4 小结

- 我们可以通过继承 Block 类来构造复杂的模型。
- Sequential 是 Block 的子类。

#### 4.1.5 练习

- 在 FancyMLP 类里我们重用了 dense，这样对输入形状有了一定要求，尝试改变下输入数据形状试试
- 如果我们去掉 FancyMLP 里面的 asscalar 会有什么问题？
- 在 NestMLP 里假设我们改成 self.net=[nn.Dense(64, activation='relu'), nn.Dense(32, activation='relu')], 而不是用 Sequential 类来构造，会有什么问题？

#### 4.1.6 扫码直达讨论区



## 4.2 模型参数的访问、初始化和共享

在之前的小节里我们一直在使用默认的初始函数, `net.initialize()`, 来初始化模型参数。我们也同时介绍过如何访问模型参数的简单方法。这一节我们将深入讲解模型参数的访问和初始化, 以及如何在多个层之间共享同一份参数。

我们首先定义同前的多层感知机、初始化权重和计算前向结果。同前比一点不同的是, 在这里我们从 MXNet 中导入了 `init` 这个包, 它包含了多种模型初始化方法。

```
In [1]: from mxnet import init, nd
        from mxnet.gluon import nn

        net = nn.Sequential()
        net.add(nn.Dense(256, activation='relu'))
        net.add(nn.Dense(10))
        net.initialize()

        x = nd.random.uniform(shape=(2, 20))
        y = net(x)
```

### 4.2.1 访问模型参数

我们知道可以通过 `[]` 来访问 `Sequential` 类构造出来的网络的特定层。对于带有模型参数的层, 我们可以通过 `Block` 类的 `params` 属性来得到它包含的所有参数。例如我们查看隐藏层的参数:

```
In [2]: net[0].params
Out[2]: dense0_ (
    Parameter dense0_weight (shape=(256, 20), dtype=float32)
    Parameter dense0_bias (shape=(256,), dtype=float32)
)
```

可以看到我们得到了一个由参数名称映射到参数的字典。第一个参数的名称为 `dense0_weight`, 它由 `net[0]` 的名称 (`dense0_`) 和自己的变量名 (`weight`) 组成。而且可以看到它参数的形状为 `(256, 20)`, 且数据类型为 32 位浮点数。

为了访问特定参数, 我们既可以通过名字来访问字典里的元素, 也可以直接使用它的变量名。下面两种方法是等价的, 但通常后者的代码可读性更好。

```
In [3]: (net[0].params['dense0_weight'], net[0].weight)

Out[3]: (Parameter dense0_weight (shape=(256, 20), dtype=float32),
          Parameter dense0_weight (shape=(256, 20), dtype=float32))
```

Gluon 里参数类型为 `Parameter` 类, 其包含参数权重和它对应的梯度, 它们可以分别通过 `data` 和 `grad` 函数来访问。因为我们随机初始化了权重, 所以它是一个由随机数组成的形状为 `(256, 20)` 的 `NDArray`.

```
In [4]: net[0].weight.data()

Out[4]:
[[ 0.06700657 -0.00369488  0.0418822 ... , -0.05517294 -0.01194733
  -0.00369594]
 [-0.03296221 -0.04391347  0.03839272 ... ,  0.05636378  0.02545484
  -0.007007 ]
 [-0.0196689   0.01582889 -0.00881553 ... ,  0.01509629 -0.01908049
  -0.02449339]
 ...
 [[ 0.00010955  0.0439323 -0.04911506 ... ,  0.06975312  0.0449558
  -0.03283203]
 [ 0.04106557  0.05671307 -0.00066976 ... ,  0.06387014 -0.01292654
  0.00974177]
 [ 0.00297424 -0.0281784 -0.06881659 ... , -0.04047417  0.00457048
  0.05696651]]
<NDArray 256x20 @cpu(0)>
```

梯度的形状跟权重一样。但我们还没有进行反向传播计算, 所以它的值全为 0.

```
In [5]: net[0].weight.grad()

Out[5]:
[[ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
 ...
 [[ 0.  0.  0. ...,  0.  0.  0.]
 [ 0.  0.  0. ...,  0.  0.  0.]
```

```
[ 0.  0.  0. ...,  0.  0.  0.]]  
<NDArray 256x20 @cpu(0)>
```

类似我们可以访问其他的层的参数。例如输出层的偏差权重：

```
In [6]: net[1].bias.data()  
  
Out[6]:  
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.]  
<NDArray 10 @cpu(0)>
```

最后，我们可以使用 Block 类提供的 `collect_params` 函数来获取这个实例包含的所有参数，它的返回同样是一个参数名称到参数的字典。

```
In [7]: net.collect_params()  
  
Out[7]: sequential0_ (  
    Parameter dense0_weight (shape=(256, 20), dtype=float32)  
    Parameter dense0_bias (shape=(256,), dtype=float32)  
    Parameter dense1_weight (shape=(10, 256), dtype=float32)  
    Parameter dense1_bias (shape=(10,), dtype=float32)  
)
```

## 4.2.2 初始化模型参数

当使用默认的模型初始化，Gluon 会将权重参数元素初始化为  $[-0.07, 0.07]$  之间均匀分布的随机数，偏差参数则全为 0. 但经常我们需要使用其他的方法来初始化权重，MXNet 的 `init` 模块 <<https://mxnet.incubator.apache.org/api/python/optimization/optimization.html#module-mxnet.initializer>> 里提供了多种预设的初始化方法。例如下面例子我们将权重参数初始化成均值为 0，标准差为 0.01 的正态分布随机数。

```
In [8]: # 非首次对模型初始化需要指定 force_reinit。  
net.initialize(init=init.Normal(sigma=0.01), force_reinit=True)  
net[0].weight.data()[0]  
  
Out[8]:  
[ 0.01074176  0.00066428  0.00848699 -0.0080038 -0.00168822  0.00936328  
 0.00357444  0.00779328 -0.01010307 -0.00391573  0.01316619 -0.00432926  
 0.0071536   0.00925416 -0.00904951 -0.00074684  0.0082254  -0.01878511  
 0.00885884  0.01911872]  
<NDArray 20 @cpu(0)>
```

如果想只对某个特定参数进行初始化，我们可以调用 Paramter 类的 `initialize` 函数，它的使用跟 Block 类提供的一致。下例中我们对第一个隐藏层的权重使用 Xavier 方法来进行初始化。

```
In [9]: net[0].weight.initialize(init=init.Xavier(), force_reinit=True)
      net[0].weight.data()[0]

Out[9]:
[ 0.00512482 -0.06579044 -0.10849719 -0.09586414  0.06394844  0.06029618
 -0.03065033 -0.01086642  0.01929168  0.1003869 -0.09339568 -0.08703034
 -0.10472868 -0.09879824 -0.00352201 -0.11063069 -0.04257748  0.06548801
  0.12987629 -0.13846186]
<NDArray 20 @cpu(0)>
```

### 4.2.3 自定义初始化方法

有时候我们需要的初始化方法并没有在 `init` 模块中提供，这时我们有两种方法来自定义参数初始化。一种是实现一个 `Initializer` 类的子类使得我们可以跟前面使用 `init.Normal` 那样使用它。在这个方法里，我们只需要实现 `_init_weight` 这个函数，将其传入的 `NDArray` 修改成需要的内容。下面例子里我们把权重初始化成  $[-10, -5]$  和  $[5, 10]$  两个区间里均匀分布的随机数。

```
In [10]: class MyInit(init.Initializer):
    def _init_weight(self, name, data):
        print('Init', name, data.shape)
        data[:] = nd.random.uniform(low=-10, high=10, shape=data.shape)
        data *= data.abs() >= 5

    net.initialize(MyInit(), force_reinit=True)
    net[0].weight.data()[0]

Init dense0_weight (256, 20)
Init dense1_weight (10, 256)

Out[10]:
[-5.36596727  7.57739449  8.98637581 -0.           8.8275547   0.
 5.98405075 -0.           0.           0.           7.48575974 -0.
 6.89100075  6.97887039 -6.11315536  0.           5.46652031 -9.73526287
 9.48517227]
<NDArray 20 @cpu(0)>
```

第二种方法是我们通过 `Parameter` 类的 `set_data` 函数来直接改写模型参数。例如下例中我们将隐藏层参数在现有的基础上加 1。

```
In [11]: net[0].weight.set_data(net[0].weight.data()+1)
net[0].weight.data()[0]
```

```
Out[11]:  
[ -4.36596727  8.57739449  9.98637581  1.          9.8275547   1.  
  6.98405075  1.          1.          1.          8.48575974  1.  
  ↪  1.  
  7.89100075  7.97887039  -5.11315536  1.          6.46652031  
  -8.73526287  10.48517227]  
<NDArray 20 @cpu(0)>
```

#### 4.2.4 共享模型参数

在有些情况下，我们希望在多个层之间共享模型参数。我们在“模型构造”这一节看到了如何在 Block 类里 `forward` 函数里多次调用同一个类来完成。这里将介绍另外一个方法，它在构造层的时候指定使用特定的参数。如果不同层使用同一份参数，那么它们不管是在前向计算还是反向传播时都会共享共同的参数。

在下面例子里，我们让模型的第二隐藏层和第三隐藏层共享模型参数。

```
In [12]: from mxnet import nd  
      from mxnet.gluon import nn  
  
      net = nn.Sequential()  
      shared = nn.Dense(8, activation='relu')  
      net.add(nn.Dense(8, activation='relu'),  
              shared,  
              nn.Dense(8, activation='relu', params=shared.params),  
              nn.Dense(10))  
      net.initialize()  
  
      x = nd.random.uniform(shape=(2,20))  
      net(x)  
  
      net[1].weight.data()[0] == net[2].weight.data()[0]
```

Out[12]:  
[ 1. 1. 1. 1. 1. 1. 1.]  
<NDArray 8 @cpu(0)>

我们在构造第三隐藏层时通过 `params` 来指定它使用第二隐藏层的参数。由于模型参数里包含了梯度，所以在反向传播计算时，第二隐藏层和第三隐藏层的梯度都会被累加在 `shared.params.grad()` 里。

## 4.2.5 小结

- 我们有多种方法来访问、初始化和共享模型参数。

## 4.2.6 练习

- 查阅MXNet 文档，了解不同的参数初始化方式。
- 尝试在 `net.initialize()` 后和 `net(x)` 前访问模型参数，看看会发生什么。
- 构造一个含共享参数层的多层感知机并训练。观察每一层的模型参数和梯度计算。

## 4.2.7 扫码直达讨论区



## 4.3 模型参数的延后初始化

如果你注意到了上节练习，你会发现在 `net.initialize()` 后和 `net(x)` 前模型参数的形状都是空。直觉上 `initialize` 会完成了所有参数初始化过程，然而 Gluon 中这是不一定。我们这里详细讨论这个话题。

### 4.3.1 延后的初始化

注意到前面使用 Gluon 的章节里，我们在创建全连接层时都没有指定输入大小。例如在一直使用的多层感知机例子里，我们创建了输出大小为 256 的隐藏层。但是当在调用 `initialize` 函数的时候，我们并不知道这个层的参数到底有多大，因为它的输入大小仍然是未知。只有在当我们把形状是  $(2, 20)$  的 `x` 输入进网络时，我们这时候才知道这一层的参数大小应该是  $(256, 20)$ 。所以这个时候我们才能真正开始初始化参数。

让我们使用上节定义的 MyInit 类来清楚的演示这一个过程。下面我们创建多层感知机，然后使用 MyInit 实例来进行初始化。

```
In [1]: from mxnet import init, nd
        from mxnet.gluon import nn

        class MyInit(init.Initializer):
            def _init_weight(self, name, data):
                print('Init', name, data.shape)
                # 实际的初始化逻辑在此省略了。

            net = nn.Sequential()
            net.add(nn.Dense(256, activation='relu'))
            net.add(nn.Dense(10))

            net.initialize(init=MyInit())
```

注意到 MyInit 在调用时会打印信息，但当前我们并没有看到相应的日志。下面我们执行前向计算。

```
In [2]: x = nd.random.uniform(shape=(2,20))
        y = net(x)

Init dense0_weight (256, 20)
Init dense1_weight (10, 256)
```

这时候系统根据输入  $x$  的形状自动推测数所有层参数形状，例如隐藏层大小是  $(256, 20)$ ，并创建参数。之后调用 MyInit 实例来进行初始方法，然后再进行前向计算。

当然，这个初始化只会在第一次执行被调用。之后我们再运行  $\text{net}(x)$  时则不会重新初始化，即我们不会再次看到 MyInit 实例的输出。

```
In [3]: y = net(x)
```

我们将这个系统将真正的参数初始化延后到获得了足够信息到时候称之为延后初始化。它可以让模型创建更加简单，因为我们只需要定义每个层的输出大小，而不用去推测它们的输入大小。这个对于之后将介绍的多达数十甚至数百层的网络尤其有用。

当然正如本节开头提到那样，延后初始化也可能会造成一定的困解。在调用第一次前向计算之前我们无法直接操作模型参数。例如无法使用 `data` 和 `set_data` 函数来获取和改写参数。所以经常我们会额外调用一次  $\text{net}(x)$  来是的参数被真正的初始化。

### 4.3.2 避免延后初始化

当系统在调用 `initialize` 函数时能够知道所有参数形状，那么延后初始化就不会发生。我们这里给两个这样的情况。

第一个是模型已经被初始化过，而且我们要对模型进行重新初始化时。因为我们知道参数大小不会变，所以能够立即进行重新初始化。

```
In [4]: net.initialize(init=MyInit(), force_reinit=True)
```

```
Init dense0_weight (256, 20)
Init dense1_weight (10, 256)
```

第二种情况是我们在创建层到时候指定了每个层的输入大小，使得系统不需要额外的信息来推测参数形状。下例中我们通过 `in_units` 来指定每个全连接层的输入大小，使得初始化能够立即进行。

```
In [5]: net = nn.Sequential()
    net.add(nn.Dense(256, in_units=20, activation='relu'))
    net.add(nn.Dense(10, in_units=256))

    net.initialize(init=MyInit())

Init dense2_weight (256, 20)
Init dense3_weight (10, 256)
```

### 4.3.3 小结

- 在调用 `initialize` 函数时，系统可能将真正的初始化延后到后面，例如前向计算时，来执行。这样到主要好处是让模型定义可以更加简单。

### 4.3.4 练习

- 如果在下一次 `net(x)` 前改变 `x` 形状，包括批量大小和特征大小，会发生什么？

### 4.3.5 扫码直达讨论区



## 4.4 自定义层

深度学习的一个魅力之处在于神经网络中各式各样的层，例如全连接层和后面章节中将要介绍的卷积层、池化层与循环层。虽然 Gluon 提供了大量常用的层，但有时候我们依然希望自定义层。本节将介绍如何使用 NDArray 来自定义一个 Gluon 的层，从而以后可以被重复调用。

### 4.4.1 不含模型参数的自定义层

我们先介绍如何定义一个不含模型参数的自定义层。事实上，这和“模型构造”一节中介绍的使用 Block 构造模型类似。

首先，导入本节中实验需要的包或模块。

```
In [1]: from mxnet import nd, gluon  
      from mxnet.gluon import nn
```

下面通过继承 Block 自定义了一个将输入减掉均值的层：CenteredLayer 类，并将层的计算放在 forward 函数里。这个层里不含模型参数。

```
In [2]: class CenteredLayer(nn.Block):  
        def __init__(self, **kwargs):  
            super(CenteredLayer, self).__init__(**kwargs)  
  
        def forward(self, x):  
            return x - x.mean()
```

我们可以实例化这个层用起来。

```
In [3]: layer = CenteredLayer()  
layer(nd.array([1, 2, 3, 4, 5]))
```

```
Out[3]:  
[-2. -1. 0. 1. 2.]  
<NDArray 5 @cpu(0)>
```

我们也可以用它来构造更复杂的模型。

```
In [4]: net = nn.Sequential()  
net.add(nn.Dense(128))  
net.add(nn.Dense(10))  
net.add(CenteredLayer())
```

打印自定义层输出的均值。由于均值是浮点数，它的值是个很接近 0 的数。

```
In [5]: net.initialize()  
y = net(nd.random.uniform(shape=(4, 8)))  
y.mean()
```

```
Out[5]:  
[-6.63567312e-10]  
<NDArray 1 @cpu(0)>
```

## 4.4.2 含模型参数的自定义层

我们还可以自定义含模型参数的自定义层。这样，自定义层里的模型参数就可以通过训练学出来了。我们在“[模型参数](#)”一节里介绍了 Parameter 类。其实，在自定义层的时候我们还可以使用 Block 自带的 ParameterDict 类型的成员变量 params。顾名思义，这是一个由字符串类型的参数名字映射到 Parameter 类型的模型参数的字典。我们可以通过 get 函数从 ParameterDict 创建 Parameter。

```
In [6]: params = gluon.ParameterDict()  
params.get("param2", shape=(2, 3))  
params  
  
Out[6]: (  
    Parameter param2 (shape=(2, 3), dtype=<class 'numpy.float32'>)  
)
```

现在我们看下如何实现一个含权重参数和偏差参数的全连接层。它使用 ReLU 作为激活函数。其中 in\_units 和 units 分别是输入单元个数和输出单元个数。

```
In [7]: class MyDense(nn.Block):  
    def __init__(self, units, in_units, **kwargs):  
        super(MyDense, self).__init__(**kwargs)  
        self.weight = self.params.get('weight', shape=(in_units, units))  
        self.bias = self.params.get('bias', shape=(units,))
```

```
def forward(self, x):
    linear = nd.dot(x, self.weight.data()) + self.bias.data()
    return nd.relu(linear)
```

下面，我们实例化 MyDense 类来看下它的模型参数。

```
In [8]: dense = MyDense(5, in_units=10)
dense.params

Out[8]: mydense0_ (
    Parameter mydense0_weight (shape=(10, 5), dtype=<class 'numpy.float32'>)
    Parameter mydense0_bias (shape=(5,), dtype=<class 'numpy.float32'>)
)
```

我们可以直接使用自定义层做计算。

```
In [9]: dense.initialize()
dense(nd.random.uniform(shape=(2, 10)))

Out[9]:
[[ 0.          0.09092736  0.          0.17156085  0.          ]
 [ 0.          0.06395531  0.          0.09730551  0.          ]]
<NDArray 2x5 @cpu(0)>
```

我们也可以使用自定义层构造模型。它用起来和 Gluon 的其他层很类似。

```
In [10]: net = nn.Sequential()
net.add(MyDense(32, in_units=64))
net.add(MyDense(2, in_units=32))
net.initialize()
net(nd.random.uniform(shape=(2, 64)))

Out[10]:
[[ 0.  0.]
 [ 0.  0.]]
<NDArray 2x2 @cpu(0)>
```

### 4.4.3 小结

- 使用 Block，我们可以方便地自定义层。

### 4.4.4 练习

- 如何修改自定义层里模型参数的默认初始化函数？

#### 4.4.5 扫码直达讨论区



## 4.5 读取和存储

到目前为止，我们介绍了如何处理数据以及构建、训练和测试深度学习模型。然而在实际中，我们有时需要把训练好的模型部署到很多不同的设备。这种情况下，我们可以把内存中训练好的模型参数存储在硬盘上供后续读取使用。

### 4.5.1 读写 NDArrays

我们首先看如何读写 NDArray。我们可以直接使用 `save` 和 `load` 函数分别存储和读取 NDArray。下面是例子我们创建 `x`，并将其存在文件名同为 `x` 的文件里。

```
In [1]: from mxnet import nd
        from mxnet.gluon import nn

        x = nd.ones(3)
        nd.save('x', x)
```

然后我们再将数据从文件读回内存。

```
In [2]: x2 = nd.load('x')
        x2

Out[2]: [
    [ 1.  1.  1.]
    <NDArray 3 @cpu(0)>]
```

同样我们可以存储一列 NArray 并读回内存。

```
In [3]: y = nd.zeros(4)
        nd.save('xy', [x, y])
        x2, y2 = nd.load('xy')
        (x2, y2)
```

```
Out[3]: (
    [ 1.  1.  1.]
    <NDArray 3 @cpu(0)>,
    [ 0.  0.  0.  0.]
    <NDArray 4 @cpu(0)>)
```

或者是一个从字符串到 NDArray 的字典。

```
In [4]: mydict = {'x': x, 'y': y}
nd.save('mydict', mydict)
mydict2 = nd.load('mydict')
mydict2

Out[4]: {'x':
    [ 1.  1.  1.]
    <NDArray 3 @cpu(0)>, 'y':
    [ 0.  0.  0.  0.]
    <NDArray 4 @cpu(0)>}
```

## 4.5.2 读写 Gluon 模型的参数

Block 类提供了 `save_params` 和 `load_params` 函数来读写模型参数。它实际做的事情就是将所有参数保存成一个名称到 NDArray 的字典到文件。读取的时候会根据参数名称找到对应的 NDArray 并赋值。下面的例子我们首先创建一个多层感知机，初始化后将模型参数保存到文件里。

下面，我们创建一个多层感知机。

```
In [5]: class MLP(nn.Block):
    def __init__(self, **kwargs):
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Dense(256, activation='relu')
        self.output = nn.Dense(10)
    def forward(self, x):
        return self.output(self.hidden(x))

net = MLP()
net.initialize()

# 由于延后初始化，我们需要先运行一次前向计算才能实际初始化模型参数。
x = nd.random.uniform(shape=(2, 20))
y = net(x)

net.save_params('mlp.params')
```

下面我们将该模型的参数存起来。

```
In [6]: filename = "../data/mlp.params"
net.save_params(filename)
```

然后，我们再实例化一次我们定义的多层感知机。但跟前面不一样是我们不是随机初始化模型参数，而是直接读取保存在文件里的参数。

```
In [7]: net2 = MLP()
net2.load_params('mlp.params')
```

因为这两个实例都有同样的参数，那么对同一个  $x$  的计算结果将会是一样。

```
In [8]: y2 = net2(x)
y2 == y
```

```
Out[8]:
[[ 1.  1.  1.  1.  1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.]]
<NDArray 2x10 @cpu(0)>
```

### 4.5.3 小结

- 通过 `save` 和 `load` 可以很方便地读写 `NDArray`。
- 通过 `load_params` 和 `save_params` 可以很方便地读写 Gluon 的模型参数。

### 4.5.4 练习

- 即使无需把训练好的模型部署到不同的设备，存储模型参数在实际中还有哪些好处？

### 4.5.5 扫码直达讨论区



## 4.6 GPU 计算

目前为止我们一直在使用 CPU 计算。的确，绝大部分的计算设备都有 CPU。然而 CPU 的设计目的是处理通用的计算。对于复杂的神经网络和大规模的数据来说，使用 CPU 来计算可能不够高效。

本节中，我们将介绍如何使用单块 Nvidia GPU 来计算。首先，需要确保至少有一块 Nvidia 显卡已经安装好了。然后，下载[CUDA](#)并按照提示设置好相应的路径。这些准备工作都完成后，下面就可以通过 `nvidia-smi` 来查看显卡信息了。

In [1]: !nvidia-smi

Mon Jun 4 19:10:04 2018

NVIDIA-SMI 375.26				Driver Version: 375.26			
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr. ECC	
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	
	0	Tesla M60	On	0000:00:1D.0	Off	0	
	N/A	37C	P0	38W / 150W	298MiB / 7612MiB	0% Default	
+-----+-----+-----+-----+-----+-----+-----+		1	Tesla M60	On	0000:00:1E.0	Off	0
+-----+-----+-----+-----+-----+-----+-----+		N/A	48C	P0	38W / 150W	289MiB / 7612MiB	0% Default
+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+	Processes:			GPU Memory		
+-----+-----+-----+-----+-----+-----+-----+		GPU	PID	Type	Process name	Usage	
+-----+-----+-----+-----+-----+-----+-----+	+-----+-----+-----+-----+-----+-----+-----+						

可以看到我们使用的机器上面有两块 Tesla M60，每块卡有 7.6GB 内存。

接下来，我们需要确认安装了 MXNet 的 GPU 版本。如果装了 MXNet 的 CPU 版本，我们需要先卸载它。例如我们可以使用 `pip uninstall mxnet`。然后根据 CUDA 的版本安装对应的 MXNet 版本。假设你安装了 CUDA 9.1，那么我们可以通过 `pip install --pre mxnet-cu91` 来安装支持 CUDA 9.1 的 MXNet 版本。

## 4.6.1 计算设备

MXNet 使用 `context` 来指定用来存储和计算的设备，例如可以是 CPU 或者 GPU。默认情况下，MXNet 会将数据创建在主内存，然后利用 CPU 来计算。在 MXNet 中，CPU 和 GPU 可分别由 `cpu()` 和 `gpu()` 来表示。需要注意的是，`mx.cpu()` 表示所有的物理 CPU 和内存。这意味着计算上会尽量使用所有的 CPU 核。但 `mx.gpu()` 只代表一块显卡和相应的显卡内存。如果有多个 GPU，我们用 `mx.gpu(i)` 来表示第  $i$  块 GPU ( $i$  从 0 开始)。

```
In [2]: import mxnet as mx
        from mxnet import nd
        from mxnet.gluon import nn

[mx.cpu(), mx.gpu(), mx.gpu(1)]

Out[2]: [cpu(0), gpu(0), gpu(1)]
```

## 4.6.2 NDArray 的 GPU 计算

默认情况下，NDArray 存在 CPU 上。因此，之前我们每次打印 NDArray 的时候都会看到 `@cpu(0)` 这个标识。

```
In [3]: x = nd.array([1,2,3])
        x

Out[3]:
[ 1.  2.  3.]
<NDArray 3 @cpu(0)>
```

我们可以通过 NDArray 的 `context` 属性来查看其所在的设备。

```
In [4]: x.context
Out[4]: cpu(0)
```

## GPU 上的存储

我们有多种方法将 NDArray 放置在 GPU 上。例如我们可以在创建 NDArray 的时候通过 `ctx` 指定存储设备。下面我们将 `a` 创建在 GPU 0 上。注意到在打印 `a` 时，设备信息变成了 `@gpu(0)`。创建在 GPU 上时我们会只用 GPU 内存，你可以通过 `nvidia-smi` 查看 GPU 内存使用情况。通常你需要确保不要创建超过 GPU 内存上限的数据。

```
In [5]: a = nd.array([1, 2, 3], ctx=mx.gpu())
a
```

```
Out[5]:
[ 1.  2.  3.]
<NDArray 3 @gpu(0)>
```

假设你至少有两块 GPU，下面代码将会在 GPU 1 上创建随机数组

```
In [6]: b = nd.random.uniform(shape=(2, 3), ctx=mx.gpu(1))
b
```

```
Out[6]:
[[ 0.59118998  0.313164    0.76352036]
 [ 0.97317863  0.35454726  0.11677533]]
<NDArray 2x3 @gpu(1)>
```

除了在创建时指定，我们也可以通过 `copyto` 和 `as_in_context` 函数在设备之间传输数据。下面我们将 CPU 上的 `x` 复制到 GPU 0 上。

```
In [7]: y = x.copyto(mx.gpu())
y
```

```
Out[7]:
[ 1.  2.  3.]
<NDArray 3 @gpu(0)>
```

```
In [8]: z = x.as_in_context(mx.gpu())
z
```

```
Out[8]:
[ 1.  2.  3.]
<NDArray 3 @gpu(0)>
```

需要区分的是，如果源变量和目标变量的 `context` 一致，`as_in_context` 使目标变量和源变量共享源变量的内存

```
In [9]: y.as_in_context(mx.gpu()) is y
```

```
Out[9]: True
```

而 `copyto` 总是为目标变量新创建内存。

```
In [10]: y.copyto(mx.gpu()) is y
```

```
Out[10]: False
```

## GPU 上的计算

MXNet 的计算会在数据的 `context` 上执行。为了使用 GPU 计算，我们只需要事先将数据放在 GPU 上面。而计算结果会自动保存在相同的 GPU 上。

```
In [11]: (z + 2).exp() * y  
Out[11]:  
[ 20.08553696 109.19629669 445.23950195]  
<NDArray 3 @gpu(0)>
```

注意，MXNet 要求计算的所有输入数据都在同一个 CPU/GPU 上。这个设计的原因是不同 CPU/GPU 之间的数据交互通常比较耗时。因此，MXNet 希望用户确切地指明计算的输入数据都在同一个 CPU/GPU 上。例如，如果将 CPU 上的 `x` 和 GPU 上的 `y` 做运算，会出现错误信息。

当我们打印 `NDArray` 或将 `NDArray` 转换成 NumPy 格式时，如果数据不在主内存里，MXNet 会自动将其先复制到主内存，从而带来隐形的传输开销。

### 4.6.3 Gluon 的 GPU 计算

同 `NDArray` 类似，Gluon 的模型可以在初始化时通过 `ctx` 指定设备。下面代码将模型参数初始化在 GPU 上。

```
In [12]: net = nn.Sequential()  
net.add(nn.Dense(1))  
net.initialize(ctx=mx.gpu())
```

当输入是 GPU 上的 `NDArray` 时，Gluon 会在相同的 GPU 上计算结果。

```
In [13]: net(y)  
Out[13]:  
[[ 0.0068339 ]  
[ 0.01366779]  
[ 0.02050169]]  
<NDArray 3x1 @gpu(0)>
```

确认一下模型参数存储在相同的 GPU 上。

```
In [14]: net[0].weight.data()  
Out[14]:  
[[ 0.0068339]]  
<NDArray 1x1 @gpu(0)>
```

#### 4.6.4 小结

- 通过 `context`, 我们可以在不同的 CPU/GPU 上存储数据和计算。

#### 4.6.5 练习

- 试试大一点的计算任务, 例如大矩阵的乘法, 看看 CPU 和 GPU 的速度区别。如果是计算量很小的任务呢?
- GPU 上应如何读写模型参数?

#### 4.6.6 扫码直达讨论区



---

## 卷积神经网络

---

本章我们介绍卷积神经网络 (convolutional neural network)。它是近年来深度学习能在计算机视觉中取得巨大成果的基石，它也逐渐在被其他诸如自然语言处理、推荐系统和语音识别等领域广泛使用。这一章我们将讲解卷积神经网络的原理，并介绍过去几年中数个在 ImageNet 竞赛（一个著名的计算机视觉竞赛）取得优异成绩的深度卷积神经网络。

### 5.1 二维卷积层

卷积神经网络是指主要由卷积层 (convolutional layer) 组成的网络。因为它最常用来处理图片数据，其有高和宽两个空间维度（彩色图片的颜色通道维度将在之后小节讨论），所以最常用到的是二维卷积层。本小节本节我们将介绍简单形式的二维卷积层的是怎么工作的。

#### 5.1.1 二维相关运算符

虽然卷积层得名于卷积运算符 (convolution)，但我们常用更加直观的相关运算符 (correlation) 来实现卷积层。一个二维相关运算符将一个二维核 (kernel) 数组作用在一个二维输入数据上来

计算一个二维数组输出。下图演示了如何对一个高宽为 3 的输入  $X$  作用高宽为 2 的核  $K$  来计算输出  $Y$ 。

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 19 & 25 \\ \hline 37 & 43 \\ \hline \end{array}$$

图 5.1: 二维相关运算符, 高亮了计算第一个输出元素所使用的输入和核数组元素。

可以看到输出  $Y$  的形状是  $(2, 2)$ , 且第一个元素是由  $X$  的左上的高宽为 2 的子数组与核数组按元素相乘后再相加得来。即  $Y[0, 0] = (X[0:2, 0:2] * K).sum()$ , 这里  $X, K$  和  $Y$  的类型都是 NDArray。接下来我们将这个高宽为 2 的窗口在  $X$  上向左滑动一列来计算  $Y$  的第二列第一个元素。以此类推计算得到所有结果。

下面我们将上述过程实现在 `corr2d` 函数里, 它接受  $X$  和  $K$ , 输出  $Y$ 。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, nd
        from mxnet.gluon import nn

def corr2d(X, K):
    h, w = K.shape
    Y = nd.zeros((X.shape[0]-h+1, X.shape[1]-w+1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i+h, j:j+w]*K).sum()
    return Y
```

构造上图中的数据来测试实现的正确性。

```
In [2]: X = nd.array([[0,1,2], [3,4,5], [6,7,8]])
        K = nd.array([[0,1], [2,3]])
        corr2d(X, K)
```

```
Out[2]:
[[ 19.  25.]
 [ 37.  43.]]
<NDArray 2x2 @cpu(0)>
```

## 5.1.2 二维卷积层

二维卷积层就是将输入和其维护的核数组，也称作卷积核，做相关运算，然后加上一个标量偏差来得到输出。它的模型参数包括了卷积核和标量偏差。在训练的时候，我们通常首先对卷积核进行随机初始化，然后不断迭代更新卷积核和偏差来拟合数据。

下面的我们基于 `corr2d` 函数来实现一个自定义的二维卷积层。在初始化函数里我们声明 `weight` 和 `bias` 这两个模型参数，前向计算函数则是直接调用 `corr2d` 再加上偏差。

```
In [3]: class Conv2D(nn.Block):
    def __init__(self, kernel_size, **kwargs):
        super(Conv2D, self).__init__(**kwargs)
        self.weight = self.params.get('weight', shape=kernel_size)
        self.bias = self.params.get('bias', shape=(1,))

    def forward(self, x):
        return corr2d(x, self.weight.data()) + self.bias.data()
```

你也许会好奇既然称之为卷积层，为什么不使用卷积运算符呢？其实卷积运算的计算与二维相关运算类似，唯一的区别是反向的将核数组跟输入做乘法，即  $Y[0, 0] = (X[0:2, 0:2] * K[::-1, ::-1]).sum()$ 。但是因为在卷积层里  $K$  是学习而来而，所以不论是正向还是反向访问都可以。

## 5.1.3 图片物体边缘检测

下面我们来看一个应用卷积层的简单应用：检测图片中物体的边缘，即找到像素变化的位置。首先我们构造一张  $6 \times 8$  的图，它中间 4 列为黑（0），其余为白（1）。

```
In [4]: X = nd.ones((6, 8))
X[:, 2:6] = 0
X

Out[4]:
[[ 1.  1.  0.  0.  0.  0.  1.  1.]
 [ 1.  1.  0.  0.  0.  0.  1.  1.]
 [ 1.  1.  0.  0.  0.  0.  1.  1.]
 [ 1.  1.  0.  0.  0.  0.  1.  1.]
 [ 1.  1.  0.  0.  0.  0.  1.  1.]
 [ 1.  1.  0.  0.  0.  0.  1.  1.]]
```

然后我们构造一个形状为  $(1, 2)$  的卷积核，使得其作用在相同的横向相邻元素上输出为 0，否则输出非 0。

```
In [5]: K = nd.array([[1, -1]])
```

对  $X$  作用我们设计的核  $K$  后可以发现，从白到黑的边缘我们检测成了 1，从黑到白则是 -1，其余全是 0。

```
In [6]: Y = corr2d(X, K)
Y
```

Out[6]:

```
[[ 0.  1.  0.  0.  0. -1.  0.]
 [ 0.  1.  0.  0.  0. -1.  0.]
 [ 0.  1.  0.  0.  0. -1.  0.]
 [ 0.  1.  0.  0.  0. -1.  0.]
 [ 0.  1.  0.  0.  0. -1.  0.]
 [ 0.  1.  0.  0.  0. -1.  0.]]
<NDArray 6x7 @cpu(0)>
```

这里我们可以看到卷积层通过重复的使用  $K$  来有效的发掘局部空间特征。

### 5.1.4 通过数据学习核数组

最后我们来看一个例子，它使用前面的  $X$  和  $Y$  来学习我们构造的  $K$ 。我们首先构造一个卷积层，将其卷积核初始化成随机数组。然后在每一个迭代里，我们使用平方误差来比较  $Y$  和卷积层的输出，然后计算梯度来更新权重。

虽然我们之前构造了 Conv2D 类，但由于 `corr2d` 使用了对单个元素赋值 ( $[i, j] =$ ) 的操作会导致无法自动求导，下面我们使用 Gluon 提供的 Conv2D 类来实现这个例子。

```
In [7]: # 构造一个输出通道是 1 (将在后面小节介绍通道)，核数组形状是 (1, 2) 的二维卷积层。
conv2d = nn.Conv2D(1, kernel_size=(1, 2))
conv2d.initialize()

# 二维卷积层使用 4 维输入输出，格式为 (批量大小, 通道数, 高, 宽)，这里批量和通道均为 1。
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))

for i in range(10):
    with autograd.record():
        Y_hat = conv2d(X)
        loss = (Y_hat - Y) ** 2
        if i % 2 == 1:
```

```
        print('batch %d, loss %.3f' % (i, loss.sum().asscalar()))
    loss.backward()
    # 为了简单起见这里忽略了偏差。
    conv2d.weight.data()[:] -= 3e-2 * conv2d.weight.grad()

batch 1, loss 4.949
batch 3, loss 0.831
batch 5, loss 0.140
batch 7, loss 0.024
batch 9, loss 0.004
```

可以看到 10 次迭代后误差已经降到了一个比较小的值，现在来看一下学习到的核。

In [8]: conv2d.weight.data().reshape((1,2))

Out[8]:

```
[[ 0.98949999 -0.98737049]]
<NDArray 1x2 @cpu(0)>
```

我们看到学到的核与我们之前定义的  $K$  非常接近。

### 5.1.5 小结

- 二维卷积层的核心计算是二维相关运算。在最简单的形式下，它对二维输入数据和卷积核做相关运算然后加上偏差。
- 我们可以设计卷积核来检测图片中的边缘，同时也可以通过数据来学习它。

### 5.1.6 练习

- 构造一个  $X$  它有水平方向的边缘，如何设计  $K$  来检测它？如果是对角方向的边缘呢？
- 试着对我们构造的 Conv2D 进行自动求导，会有什么样的错误信息？
- 在 Conv2D 的 forward 函数里，将 corr2d 替换成 nd.Convolution 使得其可以求导。
- 试着将 conv2d 的核构造成  $(2, 2)$ ，会学出什么样的结果？
- 如果通过变化输入和核的矩阵来将相关运算表示成一个矩阵乘法。
- 如何构造一个全连接层来进行物体边缘检测？

### 5.1.7 扫码直达讨论区



## 5.2 填充和步幅

在上一节的例子里，我们使用高宽为 3 的输入和高宽为 2 的卷积核得到高宽为 2 的输出。一般来说，假设输入形状是  $n_h \times n_w$ ，卷积核形状是  $k_h \times k_w$ ，那么输出形状将会是

$$(n_h - k_h + 1) \times (n_w - k_w + 1).$$

所以卷积层的输出形状由输入形状和卷积核形状决定。这一节我们将介绍卷积层的两个超参数，填充和步幅，它们可以在给定形状的输入和卷积核下来改变输出形状。

### 5.2.1 填充

填充是指在输入高和宽的两端填充元素。下图里我们在宽和高的两侧分别添加了值为 0 的元素，使得输入高宽从 2 变成了 5，从而导致输出高宽由 2 增加到 4。

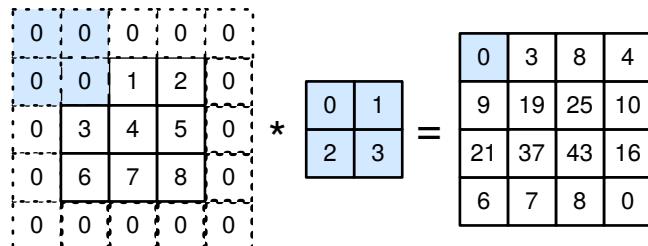


图 5.2: 在输入的高和宽两侧分别填充了 0 的二维相关计算。

一般来说，如果在高两侧一共填充  $p_h$  行，在宽两侧一共填充  $p_w$  列，那么输出形状将会是

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1),$$

也就是说输出的高宽分别会增加  $p_h$  和  $p_w$ 。

通常我们会设置  $p_h = k_h - 1$  和  $p_w = k_w - 1$  使得输入和输出有相同的高宽，这样方便在构造网络时容易推测每个层的输出形状。假设这里  $k_h$  是奇数，我们会在高的两侧分别填充  $p_h/2$  行。如果其是偶数，一种可能是上面填充  $\lceil p_h/2 \rceil$  行，而下面填充  $\lfloor p_h/2 \rfloor$  行。在宽上行为类似。

卷积神经网络经常使用奇数高宽的卷积核，例如 1、3、5、和 7，所以填充在两端上是对称的。这样还有一点方便的是我们知道输出  $Y[i, j]$  是由输入以  $X[i, j]$  为中心的窗口同卷积核进行相关计算得来。

下面例子里我们创建一个核高宽为 3 的二维卷积层，然后在输入高和宽的两侧分别填充 1。输入一个高宽为 8 的输入，我们会发现输出的高宽也是 8。

```
In [1]: from mxnet import nd
        from mxnet.gluon import nn

        # 定义一个便利函数来计算卷积层。它初始化卷积层权重，并对输入和输出做相应的升维和降维。
        def comp_conv2d(conv2d, X):
            conv2d.initialize()
            X = X.reshape((1, 1,) + X.shape)
            Y = conv2d(X)
            return Y.reshape(Y.shape[2:])

        X = nd.random.uniform(shape=(8, 8))

        # 注意这里是两侧分别填充 1，所以  $p_w = p_h = 2$ 。
        conv2d = nn.Conv2D(1, kernel_size=3, padding=1)
        comp_conv2d(conv2d, X).shape
```

Out[1]: (8, 8)

当然我们可以使用非方形卷积核，使用对应的填充同样可得相同高宽的输出。

```
In [2]: # 使用高为 5，宽为 3 的卷积核。
        conv2d = nn.Conv2D(1, kernel_size=(5, 3), padding=(2, 1))
        comp_conv2d(conv2d, X).shape
```

Out[2]: (8, 8)

## 5.2.2 步幅

在上一节里我们介绍了输出元素是如何计算而来。我们使用一个和卷积核有相同高宽的窗口，从输入的左上角开始，每次往左滑动一列或者往下滑动一行逐一计算输出。我们将每次滑动的行数

和列数称之为步幅。

目前我们看到的例子里，在高和宽两个方向上步幅均为 1。自然我们可以使用更大步幅。下图展示了在高上使用步幅 3，在宽上使用步幅 2 的情况。可以看到，计算第一列第二个元素时，窗口向下滑动了三行。而在计算输出的第一行第二个元素时窗口向右滑动了两列，再向右滑动两列时只剩一列输入元素从而填不满窗口，我们将其结果舍弃。

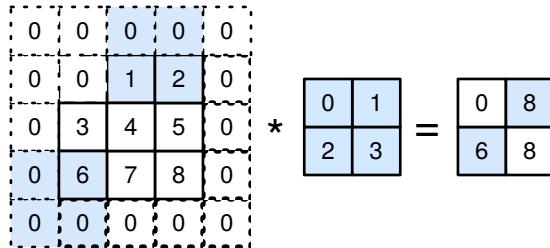


图 5.3: 高上使用步幅 3，宽上使用步幅 2。

一般来说，如果在高上使用步幅  $s_h$ ，在宽上使用步幅  $s_w$ ，那么输出大小将是

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor.$$

如果我们设置  $p_h = k_h - 1$  和  $p_w = k_w - 1$ ，那么输出大小简化为  $\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$ 。更进一步，如果输出高宽能分别被高宽上的步幅整除，那么输出将是  $n_h/s_h \times n_w/s_w$ 。也就是说我们成倍的减小了输入的高宽。

下面我们看一个符合简化形状公式例子，其将高宽减半。

```
In [3]: conv2d = nn.Conv2D(1, kernel_size=3, padding=1, strides=2)
comp_conv2d(conv2d, X).shape
```

```
Out[3]: (4, 4)
```

接下来是一个稍微复杂点的例子。

```
In [4]: conv2d = nn.Conv2D(1, kernel_size=(3,5), padding=(0,1), strides=(3,4))
comp_conv2d(conv2d, X).shape
```

```
Out[4]: (2, 2)
```

### 5.2.3 小结

通过填充可以增加输出的高宽，常用来使得输出与输入同高宽。通过步幅可以成倍的减少输出的高宽。

### 5.2.4 练习

- 对最后一个例子通过形状计算公式来计算其输出形状。
- 试一试其他的填充和步幅组合。

### 5.2.5 扫码直达讨论区



## 5.3 多输入和输出通道

前面小节里我们用到的输入和输出都是二维数组，但实际数据的维度经常更高。例如彩色图片在高宽两个维度外还有 RGB 这三个通道。假设它的高和宽分别是  $h$  和  $w$ ，那么内存中它可以被表示成一个  $3 \times h \times w$  的多维数组。我们将大小为 3 的这一维称之为通道 (channel)。这一节我们将介绍卷积层是如何处理多输入通道，和以及计算多通道输出。

### 5.3.1 多输入通道

假设输入通道数是  $c_i$ ，且卷积核窗口为  $k_h \times k_w$ 。当  $c_i = 1$  时，我们知道卷积核就是一个  $k_h \times k_w$  数组。当其大于 1 时，我们将会为每个输入通道分配一个单独的  $k_h \times k_w$  核数组。我们将这些数组合并起来，将得到一个  $c_i \times k_h \times k_w$  形状的卷积核。然后在每个通道里对相应的输入矩阵和核矩阵做相关计算，然后再将通道之间的结果相加得到最终结果。下图展示了输入通道是 2 的一个例子。

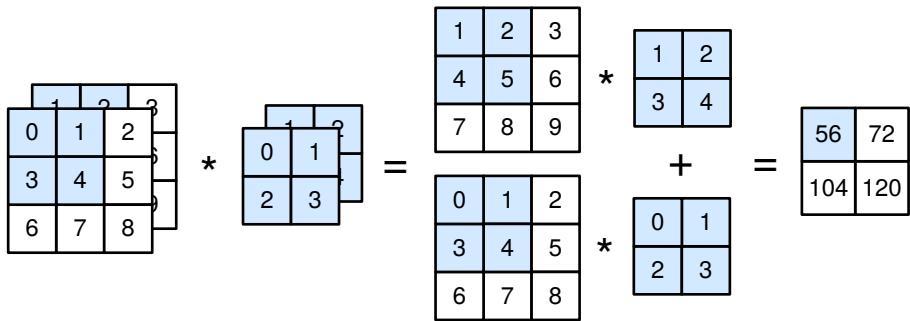


图 5.4: 输入通道为 2 的二维相关计算。

接下来我们实现处理多输入通道的相关运算符。首先我们将前面小节实现的 `corr2d` 复制过来。

```
In [1]: from mxnet import nd, autograd
from mxnet.gluon import nn

def corr2d(X, K):
    n, m = K.shape
    Y = nd.zeros((X.shape[0]-n+1, X.shape[1]-m+1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i+n, j:j+m]*K).sum()
    return Y
```

为了实现多输入通道的版本，我们只需要对每个通道做相关计算，然后通过 `nd.add_n` 来进行累加。

```
In [2]: def corr2d_multi_in(X, K):
    # 我们首先沿着 X 和 K 的第 0 维（通道维）遍历。然后使用 * 将结果列表（list）变成
    # add_n 的位置参数（positional argument）来进行相加。
    return nd.add_n(*[corr2d(x, k) for x, k in zip(X, K)])
```

我们构造上图中的输入数据来验证实现的正确性。

```
In [3]: X = nd.array([[[0,1,2], [3,4,5], [6,7,8]],
                   [[1,2,3], [4,5,6], [7,8,9]]])
K = nd.array([[0,1], [2,3], [[1,2], [3,4]]])

corr2d_multi_in(X, K)
```

```
Out[3]:
[[ 56.   72.]
```

```
[ 104. 120.]  
<NDArray 2x2 @cpu(0)>
```

### 5.3.2 多输出通道

在多输入通道下，由于我们对各个通道结果做了累加，因此不论输入通道数是多少，输入通道总是为1。如果想得到  $c_o > 1$  通道的输出，我们为每个输出通道创建单独的  $c_i \times k_h \times k_w$  形状的核数组。将它们合并起来，那么卷积核的形状是  $c_o \times c_i \times k_h \times k_w$ 。在计算的时候，每个输出通道的数据由整个输入数据和对应的核矩阵计算得来。其实现见下面代码。

```
In [4]: def corr2d_multi_in_out(X, K):  
    # 对 K 的第 0 维遍历，每次同输入 X 做相关计算。所有结果使用 nd.stack 合并在一起。  
    return nd.stack(*[corr2d_multi_in(X, k) for k in K])
```

我们将三维核矩阵 K 同 K+1 和 K+2 拼在一起构造一个输出通道为3的四维卷积核。

```
In [5]: K = nd.stack(K, K+1, K+2)  
K.shape  
  
Out[5]: (3, 2, 2, 2)
```

然后计算它的输出。可以发现计算结果有三个通道，其中第一个通道跟上例中输出一致。

```
In [6]: corr2d_multi_in_out(X, K)
```

```
Out[6]:  
[[[ 56. 72.]  
 [ 104. 120.]  
  
 [[ 76. 100.]  
 [ 148. 172.]  
  
 [[ 96. 128.]  
 [ 192. 224.]]]]  
<NDArray 3x2x2 @cpu(0)>
```

### 5.3.3 $1 \times 1$ 卷积层

最后我们讨论卷积窗口为  $1 \times 1$  ( $k_h = k_w = 1$ ) 的多通道卷积层。因为使用了最小窗口，它失去了卷积层可以识别高宽维上相邻元素构成的模式的功能，它的主要计算则是在通道维上。下图展示了输入通道为3和输出通道为2的情况。输出中的每个元素来自输入中对应位置的元素在不同通道之间的按权重累加。

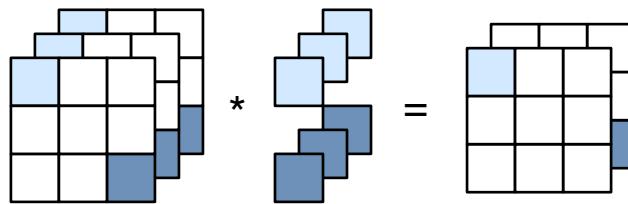


图 5.5: 多输入通道的  $1 \times 1$  卷积层

假设我们将通道维当做是特征维，而高宽中的元素则当成数据点。那么  $1 \times 1$  卷积层则等价于一个全连接层。下面代码里我们将输入和卷积核变形为二维数组，然后使用矩阵乘法来计算输出，之后再变形回我们需要的样子。

```
In [7]: def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = X.reshape((c_i, h*w))
    K = K.reshape((c_o, c_i))
    Y = nd.dot(K, X)
    return Y.reshape((c_o, h, w))
```

生成一组随机数来验证实现的正确性。

```
In [8]: X = nd.random.uniform(shape=(3,3,3))
K = nd.random.uniform(shape=(2,3,1,1))

Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)

(Y1-Y2).norm().asscalar() < 1e-6
```

Out[8]: True

在之后的模型里我们将会看到  $1 \times 1$  卷积层如何是当做保持高宽维形状的全连接层使用，其通过调整网络层之间的通道数来控制模型复杂度。

### 5.3.4 小结

- 使用多通道可以极大拓展卷积层的模型参数。
- $1 \times 1$  卷积层通常用来调节网络层之间的通道数。

### 5.3.5 练习

- 假设输入大小为  $c_i \times h \times w$ , 且使用  $c_o \times c_i \times k_h \times k_w$  卷积核, 填充为  $(p_h, p_w)$  以及步幅为  $(s_h, s_w)$ , 那么这个卷积层的前向计算需要多少次乘法, 多少次加法?
- 翻倍输入通道  $c_i$  和输出通道  $c_o$  会增加多少倍计算? 翻倍填充呢?
- 如果使用  $k_h = k_w = 1$ , 能减低多少倍计算?
- 例子中  $Y_1$  和  $Y_2$  结果完全一致吗? 原因是什么?
- 对于非  $1 \times 1$  卷积层, 如何也将其表示成一个矩阵乘法。

### 5.3.6 扫码直达讨论区



## 5.4 池化层

回忆在“卷积层”小节里介绍的图片物体边缘检测应用中, 我们构造了卷积核来精确的找到像素变化的位置。例如如果输出  $Y[i, j]=1$ , 那么  $X[i, j]$  和  $X[i, j+1]$  数值不一样, 这可能意味着物体边缘通过这两个元素之间。但实际上图片里我们感兴趣的物体不会总出现在固定位置, 例如即使我们连续拍摄同一个物体也极有可能出现像素上的偏移。这样导致同一个边缘对应的输出可能出现在  $Y$  中不同位置, 进而对后面的模式识别造成不便。

这一节我们介绍池化层 (pooling layer), 它的提出是为了缓解卷积层对位置的过渡敏感性。

### 5.4.1 二维最大、平均池化层

池化层同卷积层一样每次对输入数据的一个固定形状窗口元素计算输出。不同于卷积层里计算输入和核相关性, 池化层直接计算窗口内元素的最大值或者平均值。下图展示了  $2 \times 2$  最大池化层,

其输出的第一个元素是输入的左上  $2 \times 2$  窗口里的四个元素的最大值。然后同卷积层一样依次向左或向下移动窗口来计算其余的输出。

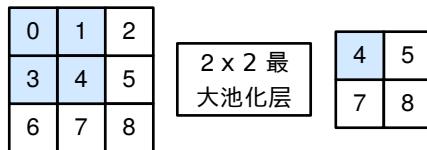


图 5.6:  $2 \times 2$  最大池化层

如果这个池化层的输入是来自于前面我们构造了卷积核的卷积层的输出。假设此卷积层输入是  $X$ , 那么不管是  $X[i, j]$  和  $X[i, j+1]$  值不同, 还是  $X[i, j+1]$  和  $X[i, j+2]$  不同, 池化层输出均有  $Y[i, j]=1$ 。换句话说, 使用  $2 \times 2$  最大池化层, 只要卷积层识别的模式在高和宽上移动不超过一个元素, 我们均可以将其检测出来。

池化层的前向计算实现在 `pool2d` 函数里。它跟“卷积层”里 `corr2d` 函数非常类似, 唯一的区别是在计算  $Y[h, w]$  上。

```
In [1]: from mxnet import nd
        from mxnet.gluon import nn

def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = nd.zeros((X.shape[0]-p_h+1, X.shape[1]-p_w+1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i:i+p_h, j:j+p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i:i+p_h, j:j+p_w].mean()
    return Y
```

构造上图中的数据来验证实现的正确性。

```
In [2]: X = nd.array([[0,1,2], [3,4,5], [6,7,8]])
        pool2d(X, (2,2))

Out[2]:
[[ 4.  5.]
 [ 7.  8.]]
<NDArray 2x2 @cpu(0)>
```

同时我们试一试平均池化层。

```
In [3]: pool2d(X, (2,2), 'avg')
```

```
Out[3]:
```

```
[[ 2.  3.]
 [ 5.  6.]]
<NDArray 2x2 @cpu(0)>
```

## 5.4.2 填充和步幅

同卷积层一样，池化层也可以填充输入高宽两侧的数据和调整窗口的移动步幅来改变输入大小。我们将通过 `nn` 模块里的二维最大池化层 `MaxPool2D` 来演示它的工作机制。我们先构造一个  $(1, 1, 4, 4)$  形状的输入数据，前两个维度分别是批量和通道。

```
In [4]: X = nd.arange(16).reshape((1, 1, 4, 4))
X
```

```
Out[4]:
```

```
[[[[ 0.   1.   2.   3.]
   [ 4.   5.   6.   7.]
   [ 8.   9.  10.  11.]
   [ 12.  13.  14.  15.]]]]
<NDArray 1x1x4x4 @cpu(0)>
```

`MaxPool2D` 类里默认步幅设置成跟池化窗大小一样。下面使用  $(3, 3)$  窗口，默认获得  $(3, 3)$  步幅。

```
In [5]: pool2d = nn.MaxPool2D(3)
# 因为池化层没有模型参数，所以不需要调用参数初始化函数。
pool2d(X)
```

```
Out[5]:
```

```
[[[[ 10.]]]]
<NDArray 1x1x1x1 @cpu(0)>
```

我们可以手动指定步幅和填充。

```
In [6]: pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)
```

```
Out[6]:
```

```
[[[[ 5.   7.]
   [ 13.  15.]]]]
<NDArray 1x1x2x2 @cpu(0)>
```

当然，我们也可以是非方形的窗口，并且指定各个方向上的填充和步幅。

```
In [7]: pool2d = nn.MaxPool2D((2,3), padding=(1,2), strides=(2,3))
pool2d(X)

Out[7]:
[[[[ 0.  3.]
   [ 8. 11.]
   [12. 15.]]]]
<NDArray 1x1x3x2 @cpu(0)>
```

### 5.4.3 多通道

在处理多通道输入数据时，池化层对每个输入通道分别池化，而不是像卷积层那么混合输入通道。这个意味着池化层的输出通道跟输入通道数相同。下面我们将  $X$  和  $X+1$  在通道维度上合并来构造通道数为 2 输入。

```
In [8]: X = nd.concat(X, X+1, dim=1)
X

Out[8]:
[[[[ 0.  1.  2.  3.]
   [ 4.  5.  6.  7.]
   [ 8.  9. 10. 11.]
   [12. 13. 14. 15.]]

  [[ 1.  2.  3.  4.]
   [ 5.  6.  7.  8.]
   [ 9. 10. 11. 12.]
   [13. 14. 15. 16.]]]]
<NDArray 1x2x4x4 @cpu(0)>
```

做池化后我们发现输出通道仍然是 2，而且通道 0 的结果跟之前一致。

```
In [9]: pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)

Out[9]:
[[[[ 5.  7.]
   [13. 15.]]

  [[ 6.  8.]
   [14. 16.]]]]
<NDArray 1x2x2x2 @cpu(0)>
```

#### 5.4.4 小结

- 池化层的通过滑动窗口计算结果，其通常直接取输入窗口内元素的最大值或者平均值作为输出。
- 池化层的一个主要作用是缓解卷积层对位置的敏感性。

#### 5.4.5 练习

- 分析池化层的计算复杂度。假设输入大小为  $c \times h \times w$ ，我们使用  $p_h \times p_w$  的池化窗，而且使用  $(p_h, p_w)$  填充和  $(s_h, s_w)$  步幅，那么这个池化层的前向计算需要操作？
- 想一想最大池化层和平均池化层的区别主要在哪里？
- 你觉得最小池化层这个想法怎么样？

#### 5.4.6 扫码直达讨论区



### 5.5 卷积神经网络

在“[多层感知机——从零开始](#)”一节里我们构造了一个两层感知机模型来对 FashionMNIST 里图片进行分类。每张图片高宽均是 28，我们将其展开成长为 784 的向量输入到模型里。这样的做法虽然简单，但也有局限性：

1. 垂直方向接近的像素在这个向量的图片表示里可能相距很远，它们组成的模式难被模型识别。
2. 对于大尺寸的输入图片，我们会得到过大的模型。假设输入是高宽为 1000 的彩色照片，即使隐藏层输出仍是 256，这一层的模型形状是  $3,000,000 \times 256$ ，其占用将近 3GB 的内存，这带来过复杂的模型和过高的存储开销。

卷积层尝试解决这两个问题：它保留输入形状，使得可以有效的发掘水平和垂直两个方向上的数据关联。它通过滑动窗口将卷积核重复作用在输入上，而得到更紧凑的模型参数表示。

卷积神经网络就是主要由卷积层组成的网络，本小节里我们将介绍一个早期用来识别手写数字图片的卷积神经网络：LeNet [1]，其名字来源于论文一作 Yann LeCun。LeNet 证明了通过梯度下降训练卷积神经网络可以达到手写数字识别的最先进的结果。这个奠基性的工作第一次将卷积神经网络推上舞台，为世人所知。

### 5.5.1 LeNet 模型

LeNet 分为卷积层块和全连接层块两个部分。卷积层块里的基本单位是卷积层后接最大池化层：卷积层用来识别图片里的空间模式，例如线条和物体局部，之后的最大池化层则用来降低卷积层对位置的敏感性。卷积层块由两个这样的基础块构成，即两个卷积层和两个最大池化层。每个卷积层都使用  $5 \times 5$  的窗口，且在输出上作用 sigmoid 激活函数  $f(x) = \frac{1}{1+e^{-x}}$  来将输出非线性变换到  $(0, 1)$  区间。第一个卷积层输出通道为 6，第二个则增加到 16，这是因为其输入高宽比之前卷积层要小，所以增加输出通道来保持相似的模型复杂度。两个最大池化层的窗口均为  $2 \times 2$ ，且步幅为 2。这意味着每个作用的池化窗口都是不重叠的。

卷积层块对每个样本输出被拉升成向量输入到全连接层块中。全连接层块由两个输出大小分别为 120 和 84 的全连接层，然后接上输出大小为 10（因为标号类数为 10）的输出层构成。下面我们用过 Sequential 类来实现 LeNet。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
        from mxnet import nd, gluon, init
        from mxnet.gluon import nn

        net = nn.Sequential()
        net.add(
            nn.Conv2D(channels=6, kernel_size=5, activation='sigmoid'),
            nn.MaxPool2D(pool_size=2, strides=2),
            nn.Conv2D(channels=16, kernel_size=5, activation='sigmoid'),
            nn.MaxPool2D(pool_size=2, strides=2),
            # Dense 会默认将 (批量大小, 通道, 高, 宽) 形状的输入转换成
            # (批量大小, 通道 x 高 x 宽) 形状的输入。
            nn.Dense(120, activation='sigmoid'),
            nn.Dense(84, activation='sigmoid'),
```

```
        nn.Dense(10)
    )

接下来我们构造一个高宽均为 28 的单通道数据点，并逐层进行前向计算来查看每个层的输出大小。
```

```
In [2]: X = nd.random.uniform(shape=(1,1,28,28))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:', X.shape)

conv0 output shape: (1, 6, 24, 24)
pool0 output shape: (1, 6, 12, 12)
conv1 output shape: (1, 16, 8, 8)
pool1 output shape: (1, 16, 4, 4)
dense0 output shape: (1, 120)
dense1 output shape: (1, 84)
dense2 output shape: (1, 10)
```

可以看到在卷积层块中图片的高宽在逐层减小，卷积层由于没有使用填充从而将高宽减 4，池化层则减半，但通道数则从 1 增加到 16。全连接层则进一步减小输出大小直到变成 10。

## 5.5.2 获取数据和训练

我们仍然使用 FashionMNIST 作为训练数据。

```
In [3]: train_data, test_data = gb.load_data_fashion_mnist(batch_size=256)
```

因为卷积神经网络计算比多层感知机要复杂，因此我们使用 GPU 来加速计算。我们尝试在 GPU 0 上创建 NDArray，如果成功则使用 GPU 0，否则则使用 CPU。（下面代码将保存在 GluonBook 的 `try_gpu` 函数里来方便重复使用）。

```
In [4]: try:
    ctx = mx.gpu()
    _ = nd.zeros((1,), ctx=ctx)
except:
    ctx = mx.cpu()
ctx
```

```
Out[4]: gpu(0)
```

我们重新将模型参数初始化到 `ctx`，且使用 Xavier [2]（使用论文一作姓氏命名）来进行随机初始化。Xavier 根据每个层的输入输出大小来选择随机数的上下区间，来使得每一层输出有相似的

方差，从而使得训练时数值更加稳定。损失函数和训练算法则使用跟之前一样的交叉熵损失函数和小批量随机梯度下降。

```
In [5]: lr = 1
    net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)

training on gpu(0)
epoch 1, loss 2.3192, train acc 0.101, test acc 0.100, time 1.8 sec
epoch 2, loss 1.9459, train acc 0.242, test acc 0.607, time 1.5 sec
epoch 3, loss 0.9598, train acc 0.618, test acc 0.673, time 1.5 sec
epoch 4, loss 0.7440, train acc 0.708, test acc 0.705, time 1.5 sec
epoch 5, loss 0.6661, train acc 0.737, test acc 0.745, time 1.5 sec
```

### 5.5.3 小节

LeNet 交替使用卷积层和最大池化层后接全连接层来进行图片分类。

### 5.5.4 练习

- LeNet 的设计是针对 MNIST，但在我们这里使用的 FashionMNIST 复杂度更高。尝试基于 LeNet 构造更复杂的网络来改善精度。例如可以考虑调整卷积窗口大小、输出层大小、激活函数和全连接层输出大小。在优化方面，可以尝试使用不同学习率、初始化方法和多使用一些迭代周期。
- 找出 Xavier 的具体初始化方法。

### 5.5.5 扫码直达讨论区



## 5.5.6 参考文献

- [1] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.
- [2] Glorot, X., & Bengio, Y. (2010, March). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249-256).

## 5.6 深度卷积神经网络：AlexNet

在 LeNet 提出后的将近二十年里，神经网络一度被其他方法（例如支持向量机）超越。虽然 LeNet 可以在 MNIST 上取得到好的成绩，在更大的真实数据集上神经网络表现并不佳。一方面神经网络计算复杂，虽然 90 年代也有过一些针对神经网络的加速硬件，但并没有大量普及。因此训练一个多通道、多层和有大量参数的卷积神经网络在当年很难完成。另一方面，当年研究者还没有大量深入研究参数初始化和非凸优化算法等诸多领域，导致复杂的神经网络收敛通常很困难。

即使神经网络可以从原始像素直接预测标签，这种称为端到端（end-to-end）的途径节省了很多中间步奏。但很长一段时间里流行的是研究者们通过勤劳智慧和黑魔法生成的手工特征。通常的模式是

1. 找个数据集；
2. 用一堆已有的特征提取函数生成特征；
3. 把这些特征表示放进一个简单的线性模型。

当时认为的机器学习部分仅限最后这一步。如果那时候你跟机器学习研究者们交谈，他们会认为机器学习既重要又优美。优雅的定理证明了许多分类器的性质。机器学习领域生机勃勃、严谨、而且极其有用。然而如果你跟一个计算机视觉研究者交谈，则是另外一幅景象。他们会告诉你图像识别里“不可告人”的现实是，计算机视觉流程中真正重要的是数据和特征。稍微干净点的数据集，或者略微好些的手调特征对最终准确度意味着天壤之别。反而分类器的选择对表现的区别影响不大。说到底，把特征扔进逻辑回归、支持向量机、或者其他任何分类器，表现都差不多。

### 5.6.1 学习特征表示

虽然相当长的时间里特征表示都是基于硬拼出来的直觉和机械化手工地生成的。事实上，做出一组特征、改进结果、并把方法写出来是计算机视觉论文里的一个重要流派。

另一些研究者则持异议。他们认为特征本身也应该由学习得来。他们还相信，为了表征足够复杂的输入，特征本身应该阶级式地组合起来。持这一想法的研究者们相信通过把许多神经网络层组合起来训练，他们可能可以让网络学得阶级式的数据表征。

例如在图片中，靠近数据的神经层可以表示边、色彩和纹理这一些底层的图片特征。中间的神经层可能可以基于这些表示来表征更大的结构，如眼睛、鼻子、草叶和其他特征。更靠近输出的神经层可能可以表征整个物体，如人、飞机、狗和飞盘。最终，在分类器层前的隐含层可能会表征经过汇总的内容，其中不同的类别将会是线性可分的。尽管这群执着的研究者不断钻研，试图学习深度的视觉数据表征，很长的一段时间里这些野心都未能实现，这其中有很多因素。

#### 缺失要素一：数据

包含许多特征的深度模型需要大量的有标签的数据才能表现得比其他经典方法更好。虽然通过大量累加数据很快就达到了线性模型的上限在工业界应用，例如广告点击预测，已经被广泛认同，但学术界直到很后来才普遍认识到这个问题。限于当时计算机有限的存储和相对囊中羞涩的 90 年代研究预算，大部分研究基于小的公开数据集。比如，大部分可信的研究论文是基于 UCI 提供的若干个数据集，其中许多只有几百至几千张图片。

这一状况在 2010 前后兴起的大数据浪潮里得到改善。尤其是 2009 年李飞飞团队贡献了 ImageNet 数据集。它包含了 1000 类，每类可能多达数千张不同的图片，这一规模是当时其他公开数据集不可相提并论的。这个数据集同时推动了计算机视觉和机器学习研究进入新的阶段，使得之前的最佳方法不再有优势。

#### 缺失要素二：硬件

深度学习对计算资源要求很高。这也是为什么上世纪 90 年代左右基于凸优化的算法更被青睐的原因。毕竟凸优化方法里能很快收敛，并可以找到全局最小值和高效的算法。

GPU 的到来改变了格局。很久以来，GPU 都是为了图像处理和计算机游戏而设计，尤其是为了大吞吐量的矩阵和向量乘法来用于基本的图形转换。值得庆幸的是，这其中的数学与深度网络中的卷积层非常类似。通用计算 GPU (GPGPU) 这个概念在 2001 年开始兴起，涌现诸如 OpenCL 和 CUDA 的编程框架。而且 GPU 也在 2010 年前后开始被机器学习社区使用。

## 5.6.2 AlexNet

2012 年 AlexNet [1]，名字来源于论文一作名字 Alex Krizhevsky，横空出世，它使用 8 层卷积神经网络以很大的优势赢得了 ImageNet 2012 图像识别挑战。它首次证明了学习到的特征可以超越手工设计的特征，从而一举打破计算机视觉研究的前状。

AlexNet 与 LeNet 的设计理念非常相似。但也有非常显著的特征。

1. 与相对较小的 LeNet 相比，AlexNet 包含 8 层变换，其中有五层卷积和两层全连接隐含层，以及一个输出层。

第一层中的卷积窗口是  $11 \times 11$ 。因为 ImageNet 图片高宽均比 MNIST 大十倍以上，对应图片的物体占用更多的像素，所以使用更大的窗口来捕获物体。第二层减少到  $5 \times 5$ ，之后全采用  $3 \times 3$ 。此外，第一，第二和第五个卷积层之后都跟了窗口为  $3 \times 3$  步幅为 2 的最大池化层。另外，AlexNet 使用的通道数也数十倍大于 LeNet。

紧接着卷积层是两个输出大小为 4096 的全连接层们。这两个巨大的全连接层带来将近 1GB 的模型大小。由于早期 GPU 显存的限制，最早的 AlexNet 使用双数据流的设计使得一个 GPU 处理一半模型。幸运的是 GPU 内存在过去几年得到了长足的发展，除了一些特殊的结构外，我们也就不再需要这样的特别设计了。

2. 将 sigmoid 激活函数改成了更加简单的 relu 函数  $f(x) = \max(x, 0)$ 。它计算上更简单，同时在不同的参数初始化方法下收敛更加稳定。
3. 通过丢弃法（参见“丢弃法”这一小节）来控制全连接层的模型复杂度。
4. 引入了大量的图片增广，例如翻转、裁剪和颜色变化，进一步扩大数据集来减小过拟合。我们将在后面的“图片增广”的小节来详细讨论。

下面我们实现（稍微简化过的）Alexnet：

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import nd, init, gluon
        from mxnet.gluon import nn

        net = nn.Sequential()
        net.add(
            # 使用较大的 11 x 11 窗口来捕获物体。同时使用步幅 4 来较大减小输出高宽。
            # 这里使用的输入通道数比 LeNet 也要大很多。
            nn.Conv2D(96, kernel_size=11, strides=4, activation='relu'),
```

```

        nn.MaxPool2D(pool_size=3, strides=2),
        # 减小卷积窗口，使用填充为 2 来使得输入输出高宽一致。且增大输出通道数。
        nn.Conv2D(256, kernel_size=5, padding=2, activation='relu'),
        nn.MaxPool2D(pool_size=3, strides=2),
        # 连续三个卷积层，且使用更小的卷积窗口。除了最后的卷积层外，
        # 进一步增大了输出通道数。前两个卷积层后不使用池化层来减小输入的高宽。
        nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
        nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
        nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'),
        nn.MaxPool2D(pool_size=3, strides=2),
        # 使用比 LeNet 输出大数倍了全连接层。其使用丢弃层来控制复杂度。
        nn.Dense(4096, activation="relu"), nn.Dropout(.5),
        nn.Dense(4096, activation="relu"), nn.Dropout(.5),
        # 输出层。我们这里使用 FashionMNIST，所以用 10，而不是论文中的 1000。
        nn.Dense(10)
    )

```

我们构造一个高宽均为 224 的单通道数据点来观察每一层的输出大小。

```

In [2]: X = nd.random.uniform(shape=(1,1,224,224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:', X.shape)

conv0 output shape: (1, 96, 54, 54)
pool0 output shape: (1, 96, 26, 26)
conv1 output shape: (1, 256, 26, 26)
pool1 output shape: (1, 256, 12, 12)
conv2 output shape: (1, 384, 12, 12)
conv3 output shape: (1, 384, 12, 12)
conv4 output shape: (1, 256, 12, 12)
pool2 output shape: (1, 256, 5, 5)
dense0 output shape: (1, 4096)
dropout0 output shape: (1, 4096)
dense1 output shape: (1, 4096)
dropout1 output shape: (1, 4096)
dense2 output shape: (1, 10)

```

### 5.6.3 读取数据

虽然论文中 Alexnet 使用 Imagenet 数据，它因为 Imagenet 数据训练时间较长，我们仍用前面的 FashionMNIST 来演示。读取数据的时候我们额外做了一步将图片高宽扩大到原版 Alexnet 使用

的 224。

```
In [3]: train_data, test_data = gb.load_data_fashion_mnist(batch_size=128, resize=224)
```

## 5.6.4 训练

这时候我们可以开始训练。相对于上节的 LeNet，这里的主要改动是使用了更小的学习率。

```
In [4]: lr = 0.01
ctx = gb.try_gpu()
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)

training on gpu(0)
epoch 1, loss 1.2983, train acc 0.512, test acc 0.752, time 99.2 sec
epoch 2, loss 0.6498, train acc 0.758, test acc 0.804, time 97.5 sec
epoch 3, loss 0.5298, train acc 0.803, test acc 0.839, time 97.7 sec
epoch 4, loss 0.4647, train acc 0.830, test acc 0.858, time 97.7 sec
epoch 5, loss 0.4238, train acc 0.846, test acc 0.858, time 97.6 sec
```

## 5.6.5 小结

AlexNet 跟 LeNet 类似，但使用了更多的卷积层和更大的参数空间来拟合大规模数据集 ImageNet。它是浅层神经网络和深度神经网络的分界线。虽然看上去 AlexNet 的实现比 LeNet 也就多了几行而已。但这个观念上的转变和真正跑出好实验结果，学术界整整花了 20 年。

## 5.6.6 练习

- 多迭代几轮看看？跟 LeNet 比有什么区别？为什么？
- AlexNet 对于 FashionMNIST 过于复杂，试着简化模型来使得训练更快，同时精度不明显下降。
- 修改批量大小，观察性能和 GPU 内存的变化。

### 5.6.7 扫码直达讨论区



### 5.6.8 参考文献

[1] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

## 5.7 使用重复元素的网络：VGG

AlexNet 在 LeNet 的基础上增加了三个卷积层。但作者对它们的卷积窗口、通道数和构造顺序均做了大量的调整。虽然 AlexNet 指明了深度卷积神经网络可以取得很高的结果，但并没有提供简单的规则来告诉后来的研究者如何设计新的网络。我们将在接下来数个小节里介绍几种不同的网络设计思路。

本节我们介绍 VGG [1]，它名字来源于论文作者所在实验室 Visual Geometry Group。VGG 提出了可以通过重复使用简单的基础块来构建深层模型。

### 5.7.1 VGG 块

VGG 模型的基础组成单位是连续数个相同的使用填充 1 的  $3 \times 3$  卷积层后接上一个步幅为 2 的  $2 \times 2$  最大池化层。卷积层保持输入高宽，而池化层则对其减半。我们使用 `vgg_block` 函数来实现这个基础块，它可以指定使用多少卷积层和其输出通道数。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import nd, init, gluon
        from mxnet.gluon import nn
```

```
def vgg_block(num_convs, num_channels):
    blk = nn.Sequential()
    for _ in range(num_convs):
        blk.add(nn.Conv2D(
            num_channels, kernel_size=3, padding=1, activation='relu'))
    blk.add(nn.MaxPool2D(pool_size=2, strides=2))
    return blk
```

## 5.7.2 VGG 模型

VGG 网络同 AlexNet 和 LeNet 一样由卷积层模块后接全连接层模块构成。卷积层模块串联数个 `vgg_block`, 其超参数由 `conv_arch` 定义, 其指定每个块里卷积层个数和输出通道。全连接模块则跟 AlexNet 一样。

现在我们构造一个 VGG 网络。它有 5 个卷积块, 前三块使用单卷积层, 而后两块使用双卷基层。第一块的输出通道是 64, 之后每次对输出通道数翻倍。因为这个网络使用了 8 个卷基层和 3 个全连接层, 所以经常被称之为 VGG 11。

```
In [2]: conv_arch = ((1,64), (1,128), (2,256), (2,512), (2,512))
```

下面我们根据架构实现 VGG 11。

```
In [3]: def vgg(conv_arch):
    net = nn.Sequential()
    # 卷积层部分
    for (num_convs, num_channels) in conv_arch:
        net.add(vgg_block(num_convs, num_channels))
    # 全连接层部分
    net.add(nn.Dense(4096, activation="relu"), nn.Dropout(.5),
            nn.Dense(4096, activation="relu"), nn.Dropout(.5),
            nn.Dense(10))
    return net

net = vgg(conv_arch)
```

然后我们打印每个卷积块的输出变化。

```
In [4]: net.initialize()
X = nd.random.uniform(shape=(1,1,224,224))
for blk in net:
    X = blk(X)
    print(blk.name, 'output shape:', X.shape)
```

```
sequential1 output shape:      (1, 64, 112, 112)
sequential2 output shape:      (1, 128, 56, 56)
sequential3 output shape:      (1, 256, 28, 28)
sequential4 output shape:      (1, 512, 14, 14)
sequential5 output shape:      (1, 512, 7, 7)
dense0 output shape:          (1, 4096)
dropout0 output shape:        (1, 4096)
dense1 output shape:          (1, 4096)
dropout1 output shape:        (1, 4096)
dense2 output shape:          (1, 10)
```

可以看到每次我们将长宽减半，最后高宽变成 7 后进入全连接层。同时输出通道数每次都翻倍。因为每个卷积层的窗口大小一样，所以每层的模型参数大小和计算复杂度跟高  $\times$  宽  $\times$  输入通道数  $\times$  输出通道数成正比。VGG 这种高宽减半和通道翻倍的设计使得每个卷基层都有相同的模型参数大小和计算复杂度。

### 5.7.3 模型训练

因为 VGG 11 计算上比 AlexNet 更加复杂，我们构造一个通道数更小，或者说更窄的，的网络来训练 FashionMNIST。

```
In [5]: ratio = 4
small_conv_arch = [(pair[0], pair[1]//ratio) for pair in conv_arch]
net = vgg(small_conv_arch)
```

模型训练跟上一节的 AlexNet 类似，除了使用使用了稍大些的学习率。

```
In [6]: lr = 0.05
ctx = gb.try_gpu()
net.initialize(ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
train_data, test_data = gb.load_data_fashion_mnist(batch_size=128, resize=224)
loss = gluon.loss.SoftmaxCrossEntropyLoss()
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=3)

training on gpu(0)
epoch 1, loss 0.9763, train acc 0.655, test acc 0.852, time 166.5 sec
epoch 2, loss 0.4056, train acc 0.851, test acc 0.862, time 164.3 sec
epoch 3, loss 0.3337, train acc 0.879, test acc 0.897, time 163.9 sec
```

#### 5.7.4 小结

VGG 通过 5 个可以重复使用的卷积块来构造网络。根据每块里卷积层个数和输出通道不同可以定义出不同的 VGG 模型。

#### 5.7.5 练习

- VGG 的计算比 AlexNet 慢很多，也需要很多的 GPU 内存。分析下原因。
- 尝试将 FashionMNIST 的高宽由 224 改成 96，实验其带来的影响。
- 参考 [1] 里的表 1 来构造 VGG 其他常用模型，例如 VGG16 和 VGG19。

#### 5.7.6 扫码直达讨论区



#### 5.7.7 参考文献

[1] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

### 5.8 网络中的网络：NiN

前面小节里我们看到 LeNet、AlexNet 和 VGG 均由两个部分组成：主要有卷积层构成的模块充分抽取空间特征，然后主要有全连接层构成的模块来输出最终分类结果。AlexNet 和 VGG 对 LeNet 的改进主要在于如何加深加宽这两模块。这一节我们介绍网络中的网络（NiN）[1]。它提出了另外一个思路，它串联多个由卷积层和“全连接”层构成的小网络来构建一个深层网络。

### 5.8.1 NiN 块

我们知道卷积层的输入和输出都是四维数组，而全连接层则是二维数组。如果想在全连接层后接上卷积层，则需要将其输出转回到四维。回忆在“[多输入和输出通道](#)”这一小节里介绍的  $1 \times 1$  卷积，它可以看成将空间维（高和宽）上每个元素当做样本，并作用在通道维上的全连接层。NiN 使用  $1 \times 1$  卷积层来替代全连接层使得空间信息能够自然传递到后面的层去。下图对比了 NiN 同 AlexNet 和 VGG 等网络的主要区别。

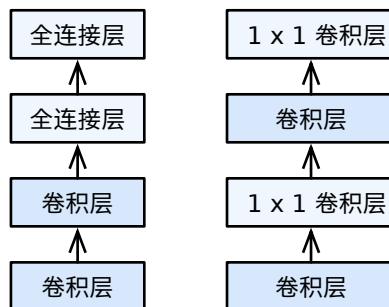


图 5.7: 对比 NiN (右) 和其他 (左)

NiN 中的一个基础块由一个卷积层外加两个充当全连接层的  $1 \times 1$  卷积层构成。第一个卷积层我们可以设置它的超参数，而第二和第三卷积层则使用固定超参数。

```
In [1]: import sys
        sys.path.insert(0, '..')
        import gluonbook as gb
        from mxnet import nd, gluon, init
        from mxnet.gluon import nn

def nin_block(num_channels, kernel_size, strides, padding):
    blk = nn.Sequential()
    blk.add(nn.Conv2D(num_channels, kernel_size,
                    strides, padding, activation='relu'),
           nn.Conv2D(num_channels, kernel_size=1, activation='relu'),
           nn.Conv2D(num_channels, kernel_size=1, activation='relu'))
    return blk
```

## 5.8.2 NiN 模型

NiN 紧跟 AlexNet 后提出，所以它的卷积层设定跟 Alexnet 类似。它使用窗口分别为  $11 \times 11$ 、 $5 \times 5$  和  $3 \times 3$  的卷积层，输出通道数也与之相同。卷积层后跟步幅为 2 的  $3 \times 3$  最大池化层。

除了使用 NiN 块外，NiN 还有一个重要的跟 AlexNet 不同的地方：NiN 去掉了最后的三个全连接层，取而代之的是使用输出通道数等于标签类数的卷积层，然后使用一个窗口为输入高宽的平均池化层来将每个通道里的数值平均成一个标量直接用于分类。这个设计好处是可以显著的减小模型参数大小，从而能很好的避免过拟合。但它也可能会造成训练时收敛变慢。

```
In [2]: net = nn.Sequential()
    net.add(
        nin_block(96, kernel_size=11, strides=4, padding=0),
        nn.MaxPool2D(pool_size=3, strides=2),
        nin_block(256, kernel_size=5, strides=1, padding=2),
        nn.MaxPool2D(pool_size=3, strides=2),
        nin_block(384, kernel_size=3, strides=1, padding=1),
        nn.MaxPool2D(pool_size=3, strides=2), nn.Dropout(.5),
        # 标签类数是 10。
        nin_block(10, kernel_size=3, strides=1, padding=1),
        # 全局平均池化层将窗口形状自动设置成输出的高和宽。
        nn.GlobalAvgPool2D(),
        # 将四维的输出转成二维的输出，其形状为（批量大小， 10）。
        nn.Flatten())
```

我们构建一个数据来查看每一层的输出大小。

```
In [3]: X = nd.random.uniform(shape=(1,1,224,224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:', X.shape)

sequential1 output shape:      (1, 96, 54, 54)
pool0 output shape:      (1, 96, 26, 26)
sequential2 output shape:      (1, 256, 26, 26)
pool1 output shape:      (1, 256, 12, 12)
sequential3 output shape:      (1, 384, 12, 12)
pool2 output shape:      (1, 384, 5, 5)
dropout0 output shape:  (1, 384, 5, 5)
sequential4 output shape:      (1, 10, 5, 5)
pool3 output shape:      (1, 10, 1, 1)
flatten0 output shape:  (1, 10)
```

### 5.8.3 获取数据并训练

跟 Alexnet 和 VGG 类似，但使用了更大的学习率。

```
In [4]: lr = .1
ctx = gb.try_gpu()
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
train_data, test_data = gb.load_data_fashion_mnist(batch_size=128, resize=224)
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=3)

training on gpu(0)
epoch 1, loss 2.1633, train acc 0.201, test acc 0.264, time 132.1 sec
epoch 2, loss 1.5340, train acc 0.444, test acc 0.616, time 130.5 sec
epoch 3, loss 0.9198, train acc 0.637, test acc 0.723, time 130.7 sec
```

### 5.8.4 小结

NiN 提供了两个重要的设计思路：

- 重复使用由卷积层和代替全连接层的  $1 \times 1$  卷积层构成的基础块来构建深层网络；
- 去除了容易造成过拟合的全连接层，而是替代成由输出通道数为标签类数的卷积层和全局平均池化层作为输出。

虽然因为精度和收敛速度等问题 NiN 并没有像本章中介绍的其他网络那么被广泛使用，但 NiN 的设计思想影响了后面的一系列网络的设计。

### 5.8.5 练习

- 多用几个迭代周期来观察网络收敛速度。
- 为什么 NiN 块里要有两个  $1 \times 1$  卷积层，去除一个看看？

### 5.8.6 扫码直达讨论区



### 5.8.7 参考文献

- [1] Lin, M., Chen, Q., & Yan, S. (2013). Network in network. arXiv preprint arXiv:1312.4400.

## 5.9 合并行连结的网络：GoogLeNet

在 2014 年的 Imagenet 竞赛中，一个名叫 GoogLeNet [1] 的网络结构大放光彩。它虽然在名字上是向 LeNet 致敬，但在网络结构上已经很难看到 LeNet 的影子。GoogLeNet 吸收了 NiN 的网络嵌套网络的想法，并在此基础上做了很大的改进。在随后的几年里研究人员对它进行了数次改进，本小节将介绍这个模型系列的第一个版本。

### 5.9.1 Inception 块

GoogLeNet 中的基础卷积块叫做 Inception，得名于同名电影《Inception》，寓意梦中嵌套梦。比较上一节介绍的 NiN，这个基础块在结构上更加复杂。

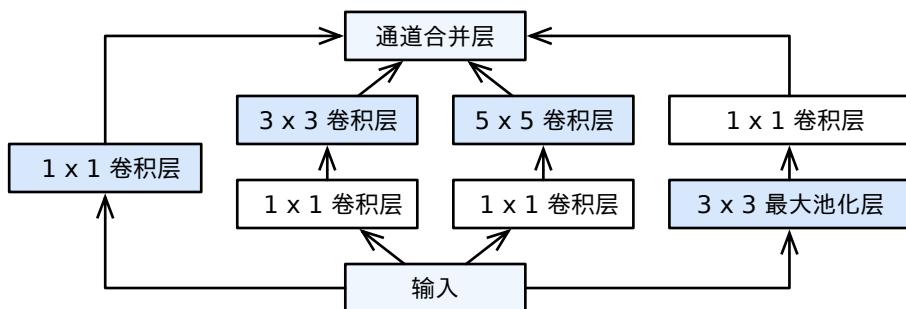


图 5.8: Inception 块。

由上图可以看出，Inception里有四个并行的线路。前三个线路里使用窗口大小分别是 $1 \times 1$ 、 $3 \times 3$ 和 $5 \times 5$ 的卷积层来抽取不同空间尺寸下的信息。其中中间两个线路会对输入先作用 $1 \times 1$ 卷积来将减小输入通道数，以此减低模型复杂度。第四条线路则是使用 $3 \times 3$ 最大池化层，后接 $1 \times 1$ 卷基层来变换通道。四条线路都使用了合适的填充来使得输入输出高宽一致。最后我们将每条线路的输出在通道维上合并在一起，输入到接下来的层中去。

Inception 块中可以自定义的超参数是每个层的输出通道数，以此我们来控制模型复杂度。

```
In [1]: import sys
        sys.path.insert(0, '...')
        import gluonbook as gb
        from mxnet import nd, init, gluon
        from mxnet.gluon import nn

        class Inception(nn.Block):
            # c1 ~ c4 为每条线路里的层的输出通道数。
            def __init__(self, c1, c2, c3, c4, **kwargs):
                super(Inception, self).__init__(**kwargs)
                # 线路 1, 单  $1 \times 1$  卷积层。
                self.p1_1 = nn.Conv2D(c1, kernel_size=1, activation='relu')
                # 线路 2,  $1 \times 1$  卷积层后接  $3 \times 3$  卷积层。
                self.p2_1 = nn.Conv2D(c2[0], kernel_size=1, activation='relu')
                self.p2_2 = nn.Conv2D(c2[1], kernel_size=3, padding=1,
                                     activation='relu')
                # 线路 3,  $1 \times 1$  卷积层后接  $5 \times 5$  卷积层。
                self.p3_1 = nn.Conv2D(c3[0], kernel_size=1, activation='relu')
                self.p3_2 = nn.Conv2D(c3[1], kernel_size=5, padding=2,
                                     activation='relu')
                # 线路 4,  $3 \times 3$  最大池化层后接  $1 \times 1$  卷积层。
                self.p4_1 = nn.MaxPool2D(pool_size=3, strides=1, padding=1)
                self.p4_2 = nn.Conv2D(c4, kernel_size=1, activation='relu')

            def forward(self, x):
                p1 = self.p1_1(x)
                p2 = self.p2_2(self.p2_1(x))
                p3 = self.p3_2(self.p3_1(x))
                p4 = self.p4_2(self.p4_1(x))
                # 在通道维上合并输出
                return nd.concat(p1, p2, p3, p4, dim=1)
```

## 5.9.2 GoogLeNet 模型

GoogLeNet 跟 VGG 一样，在主体卷积部分中使用五个模块，每个模块之间使用步幅为 2 的  $3 \times 3$  最大池化层来减小输出高宽。第一模块使用一个 64 通道的  $7 \times 7$  卷积层。

```
In [2]: b1 = nn.Sequential()
b1.add(
    nn.Conv2D(64, kernel_size=7, strides=2, padding=3, activation='relu'),
    nn.MaxPool2D(pool_size=3, strides=2, padding=1)
)
```

第二模块使用两个卷积层，首先是 64 通道的  $1 \times 1$  卷基层，然后是将通道增大 3 倍的  $3 \times 3$  卷基层。它对应 Inception 块中的第二线路。

```
In [3]: b2 = nn.Sequential()
b2.add(
    nn.Conv2D(64, kernel_size=1),
    nn.Conv2D(192, kernel_size=3, padding=1),
    nn.MaxPool2D(pool_size=3, strides=2, padding=1)
)
```

第三模块串联两个完整的 Inception 块。第一个 Inception 块的输出通道数为 256，其中四个线路的输出通道比例为 2: 4: 1: 1。且第二、三线路先分别将输入通道减小 2 倍和 12 倍后再进入第二层卷积层。第二个 Inception 块输出通道数增至 480，每个线路比例为 4: 6: 3: 2。且第二、三线路先分别减少 2 倍和 8 倍通道数。

```
In [4]: b3 = nn.Sequential()
b3.add(
    Inception(64, (96, 128), (16, 32), 32),
    Inception(128, (128, 192), (32, 96), 64),
    nn.MaxPool2D(pool_size=3, strides=2, padding=1)
)
```

第四模块更加复杂，它串联了五个 Inception 块，其输出通道分别是 512、512、512、528 和 832。其线路的通道分配类似之前， $3 \times 3$  卷积层线路输出最多通道，其次是  $1 \times 1$  卷积层线路，之后是  $5 \times 5$  卷基层和  $3 \times 3$  最大池化层线路。其中两个线路都会先按比减小通道数。这些比例在各个 Inception 块中都略有不同。

```
In [5]: b4 = nn.Sequential()
b4.add(
    Inception(192, (96, 208), (16, 48), 64),
    Inception(160, (112, 224), (24, 64), 64),
    Inception(128, (128, 256), (24, 64), 64),
    Inception(112, (144, 288), (32, 64), 64),
```

```
Inception(256, (160, 320), (32, 128), 128),
nn.MaxPool2D(pool_size=3, strides=2, padding=1)
)
```

第五模块有输出通道数为 832 和 1024 的两个 Inception 块，每个线路的通道分配使用同前的原则，但具体数字又是不同。因为这个模块后面紧跟输出层，所以它同 NiN 一样使用全局平均池化层来将每个通道高宽变成 1。最后我们将输出变成二维数组后加上一个输出大小为标签类数的全连接层作为输出。

```
In [6]: b5 = nn.Sequential()
b5.add(
    Inception(256, (160, 320), (32, 128), 128),
    Inception(384, (192, 384), (48, 128), 128),
    nn.GlobalAvgPool2D()
)

net = nn.Sequential()
net.add(b1, b2, b3, b4, b5, nn.Dense(10))
```

因为这个模型相计算复杂，而且修改通道数不如 VGG 那样简单。本节里我们将输入高宽从 224 降到 96 来加速计算。下面演示各个模块之间的输出形状变化。

```
In [7]: X = nd.random.uniform(shape=(1,1,96,96))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)

sequential0 output shape:      (1, 64, 24, 24)
sequential1 output shape:      (1, 192, 12, 12)
sequential2 output shape:      (1, 480, 6, 6)
sequential3 output shape:      (1, 832, 3, 3)
sequential4 output shape:      (1, 1024, 1, 1)
dense0 output shape:      (1, 10)
```

### 5.9.3 获取数据并训练

我们使用高宽为 96 的数据来训练。

```
In [8]: lr = 0.1
ctx = gb.try_gpu()
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
train_data, test_data = gb.load_data_fashion_mnist(batch_size=128, resize=96)
```

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)

training on gpu(0)
epoch 1, loss 1.8455, train acc 0.334, test acc 0.726, time 83.8 sec
epoch 2, loss 0.9221, train acc 0.661, test acc 0.783, time 81.4 sec
epoch 3, loss 0.5989, train acc 0.786, test acc 0.818, time 81.4 sec
epoch 4, loss 0.4288, train acc 0.840, test acc 0.864, time 81.4 sec
epoch 5, loss 0.3610, train acc 0.864, test acc 0.877, time 81.7 sec
```

## 5.9.4 小结

Inception 定义了一个有四条线路的子网络。它通过不同窗口大小的卷积层和最大池化层来并行抽取信息，并使用  $1 \times 1$  卷基层减低通道数来减少模型复杂度。GoogLeNet 将多个精细设计的 Inception 块和其他层串联起来。其通道分配比例是在 ImageNet 数据集上通过大量的实验得来。GoogLeNet 和它的后继者一度是 ImageNet 上最高效的模型之一，即在给定同样的测试精度下计算复杂度更低。

## 5.9.5 练习

1. GoogLeNet 有数个后续版本，尝试实现他们并运行看看有什么不一样。本小节介绍的是最先的版本 [1]。[2] 加入批量归一化层（后一小节将介绍），[3] 对 Inception 块做了调整。[4] 则加入了残差连接（后面小节将介绍）。
2. 对比 AlexNet、VGG 和 NiN、GoogLeNet 的模型参数大小。分析为什么后两个网络可以显著减小模型大小。

## 5.9.6 扫码直达讨论区



## 5.9.7 参考文献

- [1] Szegedy, Christian, et al. “Going deeper with convolutions.” CVPR, 2015.
- [2] Ioffe, Sergey, and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift.” arXiv:1502.03167 (2015).
- [3] Szegedy, Christian, et al. “Rethinking the inception architecture for computer vision.” CVPR. 2016.
- [4] Szegedy, Christian, et al. “Inception-v4, inception-resnet and the impact of residual connections on learning.” AAAI. 2017.

## 5.10 批量归一化——从零开始

这一节我们介绍批量归一化（batch normalization）层 [1]，它的主要作用是使得深层卷积网络训练更加容易。回忆在‘“实战 Kaggle 比赛：预测房价” <..//chapter\_supervised-learning/kaggle-gluon-kfold.md>’一节里，我们对输入数据做了归一化处理。就是我们将每个特征在所有样本上的值转归一化成均值 0 方差 1。这样我们保证训练数据里数值都同样量级上，从而使得训练的时候数值更加稳定。

对于浅层模型来说，通常数据归一化预处理足够有效。输出数值在只经过几个神经层后通常不会出现剧烈变化。但对于深层神经网络来说，情况一般比较复杂。因为每一层里都对输入乘以权重后得到输出。当很多层这样的相乘累计在一起时，一个输出数据较大的改变都可以导致输出产生巨大变化，从而带来不稳定性。

批量归一化层的提出是针对这个情况。它将一个批量里的输入数据进行归一化然后输出。如果我们将批量归一化层放置在网络的各个层之间，那么就可以不断的对中间输出进行调整，从而保证整个网络的中间输出的数值稳定性。

### 5.10.1 批量归一化层

我们首先看将批量归一化层放置在全连接层后时的情况，它的机制类似于数据归一处理。输入一个批量数据时，假设这个全连接层输出  $n$  个向量数据点  $X = \{x_1, \dots, x_n\}$ ，其中  $x_i \in \mathbb{R}^p$ 。我们

可以计算数据点在这个批量里面的均值和方差，其均为长度  $p$  的向量：

$$\mu \leftarrow \frac{1}{n} \sum_{i=1}^n x_i,$$

$$\sigma^2 \leftarrow \frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2.$$

对于数据点  $x_i$ ，我们可以对它的每一个特征维进行归一化：

$$\hat{x}_i \leftarrow \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}},$$

这里  $\epsilon$  是一个很小的常数保证不除以 0。在上面归一化的基础之上，批量归一化层引入了两个可以学习的模型参数，拉升参数  $\gamma$  和偏移参数  $\beta$ 。它们是长为  $p$  的向量，作用在  $\hat{x}_i$  上：

$$y_i \leftarrow \gamma \hat{x}_i + \beta.$$

这里  $Y = \{y_1, \dots, y_n\}$  是批量归一化层的输出。

如果批量归一化层是放置在卷积层后面，那么我们将通道维当做是特征维，空间维（高和宽）里的元素则当成是样本（参考“[多输入和输出通道](#)”里我们对  $1 \times 1$  卷积层的讨论）。

通常训练的时候我们使用较大的批量大小来获取更好的计算性能，这时批量内样本均值和方差的计算都较为准确。但在预测的时候，我们可能使用很小的批量大小，甚至每次我们只对一个样本做预测，这时我们无法得到较为准确的均值和方差。对此，批量归一化层的解决方法是维护一个移动平滑的样本均值和方差来在预测时使用。

下面我们通过 NDArray 来实现这个计算。

```
In [1]: import sys
        sys.path.insert(0, '...')
        import gluonbook as gb
        from mxnet import nd, gluon, init, autograd
        from mxnet.gluon import nn

def batch_norm(X, gamma, beta, moving_mean, moving_var,
               eps, momentum):
    # 通过 autograd 来获取是不是在训练环境下。
    if not autograd.is_training():
        # 如果是在预测模式下，直接使用传入的移动平滑均值和方差。
        X_hat = (X - moving_mean) / nd.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
```

```

# 接在全连接层后情况，计算特征维上的均值和方差。
if len(X.shape) == 2:
    mean = X.mean(axis=0)
    var = ((X - mean)**2).mean(axis=0)
# 接在二维卷积层后的情况，计算通道维上 (axis=1) 的均值和方差。这里我们需要保持 X
# 的形状以便后面可以正常的做广播运算。
else:
    mean = X.mean(axis=(0,2,3), keepdims=True)
    var = ((X - mean)**2).mean(axis=(0,2,3), keepdims=True)
# 训练模式下用当前的均值和方差做归一化。
X_hat = (X - mean) / nd.sqrt(var + eps)
# 更新移动平滑均值和方差。
moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
moving_var = momentum * moving_var + (1.0 - momentum) * var
# 拉升和偏移
Y = gamma * X_hat + beta
return (Y, moving_mean, moving_var)

```

接下来我们自定义一个 BatchNorm 层。它保存参与求导和更新的模型参数 `beta` 和 `gamma`。同时也维护移动平滑的均值和方差使得在预测时可以使用。

```

In [2]: class BatchNorm(nn.Block):
    def __init__(self, num_features, num_dims, **kwargs):
        super(BatchNorm, self).__init__(**kwargs)
        shape = (1,num_features) if num_dims == 2 else (1,num_features,1,1)
        # 参与求导和更新的模型参数，分别初始化成 0 和 1。
        self.beta = self.params.get('beta', shape=shape, init=init.Zero())
        self.gamma = self.params.get('gamma', shape=shape, init=init.One())
        # 不参与求导的模型参数。全在 CPU 上初始化成 0。
        self.moving_mean = nd.zeros(shape)
        self.moving_variance = nd.zeros(shape)
    def forward(self, X):
        # 如果 X 不在 CPU 上，将 moving_mean 和 moving_varience 复制到对应设备上。
        if self.moving_mean.context != X.context:
            self.moving_mean = self.moving_mean.copyto(X.context)
            self.moving_variance = self.moving_variance.copyto(X.context)
        # 保存更新过的 moving_mean 和 moving_var。
        Y, self.moving_mean, self.moving_variance = batch_norm(
            X, self.gamma.data(), self.beta.data(), self.moving_mean,
            self.moving_variance, eps=1e-5, momentum=0.9)
        return Y

```

## 5.10.2 使用批量归一化层的 LeNet

下面我们修改“卷积神经网络”这一节介绍的 LeNet 来使用批量归一化层。我们在所有的卷积层和全连接层与激活层之间加入批量归一化层，来使得每层的输出都被归一化。

```
In [3]: net = nn.Sequential()
net.add(
    nn.Conv2D(6, kernel_size=5),
    BatchNorm(6, num_dims=4),
    nn.Activation('sigmoid'),
    nn.MaxPool2D(pool_size=2, strides=2),
    nn.Conv2D(16, kernel_size=5),
    BatchNorm(16, num_dims=4),
    nn.Activation('sigmoid'),
    nn.MaxPool2D(pool_size=2, strides=2),
    nn.Dense(120),
    BatchNorm(120, num_dims=2),
    nn.Activation('sigmoid'),
    nn.Dense(84),
    BatchNorm(84, num_dims=2),
    nn.Activation('sigmoid'),
    nn.Dense(10)
)
```

使用同前一样的超参数，可以发现前面五个迭代周期的收敛有明显加速。

```
In [4]: lr = 1.0
ctx = gb.try_gpu()
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
train_data, test_data = gb.load_data_fashion_mnist(batch_size=256)
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)

training on gpu(0)
epoch 1, loss 0.6715, train acc 0.760, test acc 0.781, time 3.4 sec
epoch 2, loss 0.3998, train acc 0.853, test acc 0.839, time 3.3 sec
epoch 3, loss 0.3522, train acc 0.873, test acc 0.876, time 3.3 sec
epoch 4, loss 0.3231, train acc 0.883, test acc 0.851, time 3.3 sec
epoch 5, loss 0.3072, train acc 0.888, test acc 0.834, time 3.3 sec
```

最后我们查看下第一个批量归一化层学习到了 `beta` 和 `gamma`。

```
In [5]: (net[1].beta.data().reshape((-1,)),
        net[1].gamma.data().reshape((-1,)))
```

```
Out[5]: (
    [ 1.3120991 -0.13143063  0.08233833  0.59572172 -0.3261897 -1.3080616 ]
    <NDArray 6 @gpu(0)>,
    [ 2.05584097  1.44259536  1.57963598  1.42150044  1.90112901  1.40278089]
    <NDArray 6 @gpu(0)>)
```

### 5.10.3 小结

批量归一化层对网络中间层的输出做归一化，来使得深层网络学习时数值更加稳定。

### 5.10.4 练习

- 尝试调大学习率，看看跟前面的 LeNet 比，是不是可以使用更大的学习率。
- 尝试将批量归一化层插入到 LeNet 的其他地方，看看效果如何，想一想为什么。
- 尝试不不学习 `beta` 和 `gamma`（构造的时候加入这个参数 `grad_req='null'` 来避免计算梯度），看看效果会怎么样。

### 5.10.5 扫码直达讨论区



### 5.10.6 参考文献

[1] Ioffe, Sergey, and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift.” arXiv:1502.03167 (2015).

## 5.11 批量归一化——使用 Gluon

相比于前小节定义的 BatchNorm 类，nn 模块定义的 BatchNorm 使用更加简单。它不需要指定输出数据的维度和特征维的大小，这些都将通过延后初始化来获取。我们实现同前小节一样的批量归一化的 LeNet。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import nd, gluon, init
        from mxnet.gluon import nn

        net = nn.Sequential()
        net.add(
            nn.Conv2D(6, kernel_size=5),
            nn.BatchNorm(),
            nn.Activation('sigmoid'),
            nn.MaxPool2D(pool_size=2, strides=2),
            nn.Conv2D(16, kernel_size=5),
            nn.BatchNorm(),
            nn.Activation('sigmoid'),
            nn.MaxPool2D(pool_size=2, strides=2),
            nn.Dense(120),
            nn.BatchNorm(),
            nn.Activation('sigmoid'),
            nn.Dense(84),
            nn.BatchNorm(),
            nn.Activation('sigmoid'),
            nn.Dense(10)
        )
```

和使用同样的超参数进行训练。

```
In [2]: lr = 1.0
        ctx = gb.try_gpu()
        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        loss = gluon.loss.SoftmaxCrossEntropyLoss()
        train_data, test_data = gb.load_data_fashion_mnist(batch_size=256)
        gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)

        training on gpu(0)
epoch 1, loss 0.6881, train acc 0.755, test acc 0.792, time 2.3 sec
epoch 2, loss 0.4150, train acc 0.848, test acc 0.864, time 1.9 sec
```

```
epoch 3, loss 0.3610, train acc 0.869, test acc 0.849, time 2.0 sec
epoch 4, loss 0.3348, train acc 0.878, test acc 0.839, time 1.9 sec
epoch 5, loss 0.3131, train acc 0.886, test acc 0.867, time 1.9 sec
```

### 5.11.1 小结

Gluon 提供的 BatchNorm 使用上更加简单。

### 5.11.2 练习

查看 BatchNorm 文档来了解更多使用方法，例如如何在训练时使用全局平均的均值和方差。

### 5.11.3 扫码直达讨论区



## 5.12 残差网络：ResNet

上一小节介绍的批量归一化层对网络中间层的输出做归一化，使得训练时数值更加稳定和收敛更容易。但对于深层网络来说，还有一个问题困扰训练。在进行梯度反传计算时，我们从误差函数（顶部）开始，朝着输入数据方向（底部）逐层计算梯度。当我们把层串联在一起的时候，根据链式法则我们将每层的梯度乘在一起，这样经常导致梯度大小指数衰减。从而在靠近底部的层只得到很小的梯度，随之权重的更新量也变小，使得他们的收敛缓慢。

ResNet [1] 成功增加跨层的数据线路来允许梯度可以快速的到达底部层来有效避免这一情况。这一节我们将介绍 ResNet 的工作原理。

### 5.12.1 残差块

ResNet 的基础块叫做残差块。如下图所示，它将层 A 的输出在输入给层 B 的同时跨过 B，并和 B 的输出相加作为下面层的输入。它可以看成是两个网络相加，一个网络只有层 A，一个则有层 A 和 B。这里层 A 在两个网络之间共享参数。在求梯度的时候，来自层 B 上层的梯度既可以通过层 B 也可以直接到达层 A，从而使得层 A 可以更容易获取足够大的梯度来进行模型更新。

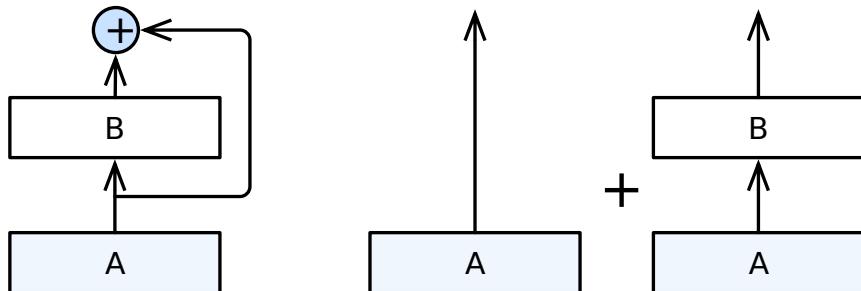


图 5.9: 残差块（左）和它的分解（右）

ResNet 沿用了 VGG 全  $3 \times 3$  卷积层设计。残差块里首先是两次有同样输出通道的  $3 \times 3$  卷积层，每个卷积层后跟一个批量归一化层和 ReLU 激活层。然后我们将输入跳过这两个卷积层后直接加在最后的 ReLU 激活层前。这样的设计要求这两个卷积层的输出都保持跟输入形状一样来保证相加可以进行。如果想改变输出的通道数，我们则引入一个额外的  $1 \times 1$  卷积层来将输入转换成需要的形状后再相加。

残差块的实现见下。它可以设定输出通道数，是否是用额外的卷积层来修改输入通道数，以及步幅大小。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import nd, gluon, init
        from mxnet.gluon import nn

        class Residual(nn.Block):
            def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
                super(Residual, self).__init__(**kwargs)
                self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                                    strides=strides)
```

```

    self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
    if use_1x1conv:
        self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                             strides=strides)
    else:
        self.conv3 = None
    self.bn1 = nn.BatchNorm()
    self.bn2 = nn.BatchNorm()

    def forward(self, X):
        Y = nd.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return nd.relu(Y + X)

```

查看输入输出形状一致的情况:

```

In [2]: blk = Residual(3)
blk.initialize()
X = nd.random.uniform(shape=(4, 3, 6, 6))
blk(X).shape

```

```
Out[2]: (4, 3, 6, 6)
```

改变输出形状的同时减半输出高宽:

```

In [3]: blk = Residual(6, use_1x1conv=True, strides=2)
blk.initialize()
blk(X).shape

```

```
Out[3]: (4, 6, 3, 3)
```

## 5.12.2 ResNet 模型

ResNet 前面两层跟前面介绍的 GoogLeNet 一样，在输出通道为 64 步幅为 2 的  $7 \times 7$  卷积层后接步幅为 2 的  $3 \times 3$  的最大池化层。不同在于一点在于 ResNet 的每个卷积层后面增加的批量归一化层。

```

In [4]: net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
       nn.BatchNorm(), nn.Activation('relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))

```

GoogLeNet 在后面接四个由 Inception 块组成的模块。而 ResNet 则是使用四个由残差块组成的模块。每个模块使用数个同样输出通道的残差块。第一个模块的通道数同输入一致。之后每个模块对之前的通道数翻倍。同时因为之前已经使用了步幅为 2 的最大池化层，所以第一个模块不继续减小高宽。后面的模块则在第一个残差块里减半高宽。

下面我们实现这个模块，注意我们根据它是不是第一个模块而使用了不同的策略。

```
In [5]: def resnet_block(num_channels, num_residuals, first_block=False):
    blk = nn.Sequential()
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.add(Residual(num_channels))
    return blk
```

下面我们为 ResNet 加入所有残差块。这里每个模块使用两个残差块。

```
In [6]: net.add(resnet_block(64, 2, first_block=True),
               resnet_block(128, 2),
               resnet_block(256, 2),
               resnet_block(512, 2))
```

最后与 GoogLeNet 一样我们加入全局平均池化层后接上全连接层输出。

```
In [7]: net.add(nn.GlobalAvgPool2D(), nn.Dense(10))
```

这里每个模块里有 4 个卷积层（不计算  $1 \times 1$  卷积层），加上最开始的卷积层和最后的全连接层，一共有 18 层。这个模型也通常被称之为 ResNet 18。通过配置不同的通道数和模块里的残差块数我们可以得到不同的 ResNet 模型。

注意到每个残差块里我们都将输入直接或者通过简单的  $1 \times 1$  卷积层加在输出上，所以即使层数很多，损失函数的梯度也能很快的传递到靠近输入的层那里。这使得即使是很深的 ResNet（例如 ResNet 152），在收敛速度上也同浅的 ResNet（例如这里实现的 ResNet 18）类似。同时虽然它的主体架构上跟 GoogLeNet 类似，但 ResNet 结构更加简单，修改也更加方便。这些因素都导致了 ResNet 迅速的被广泛使用。

最后我们考察输入在 ResNet 不同模块之间的变化。

```
In [8]: X = nd.random.uniform(shape=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:', X.shape)
```

```
conv5 output shape:      (1, 64, 112, 112)
batchnorm4 output shape:      (1, 64, 112, 112)
relu0 output shape:      (1, 64, 112, 112)
pool0 output shape:      (1, 64, 56, 56)
sequential1 output shape:      (1, 64, 56, 56)
sequential2 output shape:      (1, 128, 28, 28)
sequential3 output shape:      (1, 256, 14, 14)
sequential4 output shape:      (1, 512, 7, 7)
pool1 output shape:      (1, 512, 1, 1)
dense0 output shape:      (1, 10)
```

### 5.12.3 获取数据并训练

使用跟 GoogLeNet 一样的超参数，但减半了学习率。

```
In [9]: ctx = gb.try_gpu()
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.05})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
train_data, test_data = gb.load_data_fashion_mnist(batch_size=256, resize=96)
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)

training on gpu(0)
epoch 1, loss 0.4953, train acc 0.828, test acc 0.869, time 68.8 sec
epoch 2, loss 0.2547, train acc 0.906, test acc 0.903, time 67.4 sec
epoch 3, loss 0.1884, train acc 0.932, test acc 0.907, time 67.4 sec
epoch 4, loss 0.1445, train acc 0.948, test acc 0.915, time 67.5 sec
epoch 5, loss 0.1036, train acc 0.963, test acc 0.919, time 67.5 sec
```

### 5.12.4 小结

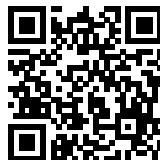
残差块通过将输入加在卷积层作用过的输出上来引入跨层通道。这使得即使非常深的网络也能很容易训练。

### 5.12.5 练习

- 参考 [1] 的表 1 来实现不同的 ResNet 版本。
- 在对于比较深的网络，[1] 介绍了一个“bottleneck”架构来降低模型复杂度。尝试实现它。

- 在 ResNet 的后续版本里 [2]，作者将残差块里的“卷积、批量归一化和激活”结构改成了“批量归一化、激活和卷积”（参考 [2] 中的图 1），实现这个改进。

### 5.12.6 扫码直达讨论区



### 5.12.7 参考文献

- [1] He, Kaiming, et al. “Deep residual learning for image recognition.” CVPR. 2016.
- [2] He, Kaiming, et al. “Identity mappings in deep residual networks.” ECCV, 2016.

## 5.13 稠密连接网络：DenseNet

ResNet 中的跨层连接设计引申出了数个后续工作。这一节我们介绍其中的一个：稠密连接网络（DenseNet）[1]。它与 ResNet 的主要区别如下图演示。

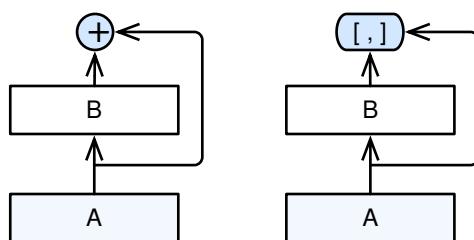


图 5.10: ResNet (左) 对比 DenseNet (右)。

可以 DenseNet 里层 B 的输出不是像 ResNet 那样通过加法和其输入，即层 A 的输出，合并，而是通过在通道维上的合并，这样层 A 的输出可以不受影响的进入上面的神经层。这样层 A 直接跟上面的所有层连接在了一起，这是为什么称之为“稠密连接”的原因。

DenseNet 的主要构建模块是稠密块和过渡块，前者定义了输入和输出是如何合并的，后者则用来控制通道数不要过大。

### 5.13.1 稠密块

DenseNet 使用了 ResNet 改良版的“批量归一化、激活和卷积”结构（参见上一节习题），我们首先在 `conv_block` 函数里实现这个结构。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import nd, gluon, init
        from mxnet.gluon import nn

def conv_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(), nn.Activation('relu'),
           nn.Conv2D(num_channels, kernel_size=3, padding=1))
    return blk
```

稠密块由多个 `conv_block` 组成，每块使用相同的输出通道数。但在前向计算时，我们将每块的输出在通道维上同其输出合并进入下一个块。

```
In [2]: class DenseBlock(nn.Block):
        def __init__(self, num_convs, num_channels, **kwargs):
            super(DenseBlock, self).__init__(**kwargs)
            self.net = nn.Sequential()
            for _ in range(num_convs):
                self.net.add(conv_block(num_channels))

        def forward(self, X):
            for blk in self.net:
                Y = blk(X)
                # 在通道维上将输入和输出合并。
                X = nd.concat(X, Y, dim=1)
            return Y
```

下面例子中我们定义一个有两个输出通道数为 10 的卷积块，使用通道数为 3 的输入时，我们会得到通道数为  $3 + 2 \times 10 = 23$  的输出。卷积块的通道数控制了输出通道数相对于输入通道数的增长，因此也被成为之增长率（growth rate）。

```
In [3]: blk = DenseBlock(2, 10)
blk.initialize()
```

```
X = nd.random.uniform(shape=(4,3,8,8))
Y = blk(X)
Y.shape

Out[3]: (4, 10, 8, 8)
```

## 5.13.2 过渡块

由于每个稠密块都会带来通道数的增加。使用过多则会导致过于复杂的模型复杂度。过渡块 (transition block) 则用来控制模型复杂度。它通过  $1 \times 1$  卷积层来减小通道数。同时它使用步幅为 2 的平均池化层来将高宽减半来进一步降低复杂度。

```
In [4]: def transition_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(), nn.Activation('relu'),
           nn.Conv2D(num_channels, kernel_size=1),
           nn.AvgPool2D(pool_size=2, strides=2))
    return blk
```

我们对前面的稠密块的输出使用通道数为 10 的过渡块。

```
In [5]: blk = transition_block(10)
blk.initialize()
blk(Y).shape

Out[5]: (4, 10, 4, 4)
```

## 5.13.3 DenseNet 模型

DenseNet 首先使用跟 ResNet 一样的单卷积层和最大池化层：

```
In [6]: net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
       nn.BatchNorm(), nn.Activation('relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

不同于 ResNet 接下来使用四个基于残差块的模块，DenseNet 使用的是四个稠密块。同 ResNet 一样我们可以设置每个稠密块使用多少个卷积层，这里我们设成 4，跟上一节的 ResNet 18 保持一致。稠密块里的卷积层通道数（既增长率）设成 32，所以每个稠密块将增加 128 通道。

ResNet 里通过步幅为 2 的残差块来在每个模块之间减小高宽，这里我们则是使用过渡块来减半高宽，并且减半输入通道数。

```
In [7]: num_channels = 64 # 当前的数据通道数。  
growth_rate = 32  
num_convs_in_dense_blocks = [4, 4, 4, 4]  
  
for i, num_convs in enumerate(num_convs_in_dense_blocks):  
    net.add(DenseBlock(num_convs, growth_rate))  
    num_channels += num_convs * growth_rate # 上一个稠密的输出通道数。  
    # 在稠密块之间加入通道数减半的过渡块。  
    if i != len(num_convs_in_dense_blocks)-1:  
        net.add(transition_block(num_channels//2))
```

最后同 ResNet 一样我们接上全局池化层和全连接层来输出。

```
In [8]: net.add(nn.BatchNorm(), nn.Activation('relu'),  
            nn.GlobalAvgPool2D(), nn.Dense(10))
```

## 5.13.4 获取数据并训练

因为这里我们使用了比较深的网络，所以我们进一步把输入减少到  $32 \times 32$  来训练。

```
In [9]: ctx = gb.try_gpu()  
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())  
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})  
loss = gluon.loss.SoftmaxCrossEntropyLoss()  
train_data, test_data = gb.load_data_fashion_mnist(batch_size=256, resize=96)  
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)  
  
training on gpu(0)  
epoch 1, loss 0.5727, train acc 0.807, test acc 0.847, time 58.4 sec  
epoch 2, loss 0.3076, train acc 0.890, test acc 0.894, time 56.9 sec  
epoch 3, loss 0.2516, train acc 0.909, test acc 0.890, time 56.7 sec  
epoch 4, loss 0.2213, train acc 0.921, test acc 0.904, time 56.7 sec  
epoch 5, loss 0.1978, train acc 0.929, test acc 0.902, time 56.6 sec
```

## 5.13.5 小结

不同于 ResNet 中将输入加在输出上完成跨层连接，DenseNet 在通道维上合并输入和输出使得底部神经层能跟其上面所有层连接起来。

### 5.13.6 练习

- DesNet 论文中提交的一个优点是其模型参数比 ResNet 更小，这是为什么？
- DesNet 被人诟病的一个问题是内存消耗过多。真的会这样吗？可以把输入换成  $224 \times 224$ ，来看看实际（GPU）内存消耗。
- 实现 [1] 中的表 1 提出的各个 DenseNet 版本。

### 5.13.7 扫码直达讨论区



### 5.13.8 参考文献

[1] Huang, Gao, et al. “Densely connected convolutional networks.” CVPR. 2017.



---

## 循环神经网络

---

与之前介绍的多层感知机和卷积神经网络不同，循环神经网络（recurrent neural networks）引入了状态变量。在一个序列中，循环神经网络当前时刻的状态不仅保存了过去时刻的信息，还与当前时刻的输入共同决定当前时刻的输出。

循环神经网络常用于处理序列数据，例如一段文字或声音、购物或观影的顺序、甚至是图片中的一行或一列像素。因此，循环神经网络在实际中有着极为广泛的应用，例如语言模型、文本分类、机器翻译、语音识别、图像分析、手写识别和推荐系统。

由于本章中的应用基于语言模型，我们将先介绍语言模型的基本概念，并以此问题激发循环神经网络的设计灵感。接着，我们将描述循环神经网络中梯度计算方法，来探究循环神经网络训练可能存在的问题。对于其中的部分问题，我们可以使用本章稍后介绍的含门控的循环神经网络来解决。最后，我们将拓展循环神经网络的架构并介绍更精简的 Gluon 实现。

## 6.1 语言模型

语言模型 (language model) 是自然语言处理的重要技术。自然语言处理中最常见的数据是文本数据。实际上，我们可以把一段自然语言文本看作是一段离散的时间序列。假设一段长度为  $T$  的文本中的词依次为  $w_1, w_2, \dots, w_T$ ，那么在离散的时间序列中， $w_t$  ( $1 \leq t \leq T$ ) 可看作是在时间步 (time step)  $t$  的输出或标签。给定一个长度为  $T$  的词的序列： $w_1, w_2, \dots, w_T$ ，语言模型将计算该序列的概率：

$$\mathbb{P}(w_1, w_2, \dots, w_T).$$

语言模型可用于提升语音识别和机器翻译的性能。例如在语音识别中，给定一段“厨房里食油用完了”的语音，有可能会输出“厨房里食油用完了”和“厨房里石油用完了”这两个读音完全一样的文本序列。如果语言模型判断出前者的概率大于后者的概率，我们可以根据相同读音的语音输出“厨房里食油用完了”的文本序列。在机器翻译中，如果对英文“you go first”逐词翻译成中文的话，可能得到“你走先”、“你先走”等排列方式的文本序列。如果语言模型判断出“你先走”的概率大于排列方式的文本序列的概率，我们可以把“you go first”翻译成“你先走”。

### 6.1.1 语言模型的计算

既然语言模型很有用，那该如何计算它呢？根据全概率公式，我们有

$$\mathbb{P}(w_1, w_2, \dots, w_T) = \prod_{t=1}^T \mathbb{P}(w_t | w_1, \dots, w_{t-1}).$$

例如一段含有四个词的文本序列的概率

$$\mathbb{P}(w_1, w_2, w_3, w_4) = \mathbb{P}(w_1)\mathbb{P}(w_2 | w_1)\mathbb{P}(w_3 | w_1, w_2)\mathbb{P}(w_4 | w_1, w_2, w_3).$$

为了计算语言模型，我们需要计算词的概率和给定前几个词的条件概率，即语言模型参数。设训练数据集为一个大型文本语料库。词的概率可以通过该词在训练数据集中的相对词频计算。例如， $\mathbb{P}(w_1)$  可以计算为  $w_1$  在训练数据集中的词频与训练数据集的总词数的比值。因此，根据条件概率定义，一个词在给定前几个词的条件概率也可以通过训练数据集中的相对词频计算。例如， $\mathbb{P}(w_2 | w_1) = \mathbb{P}(w_1, w_2)/\mathbb{P}(w_1)$  可以计算为  $w_1, w_2$  两词相邻的频率与  $w_1$  词频的比值。而  $\mathbb{P}(w_3 | w_1, w_2) = \mathbb{P}(w_1, w_2, w_3)/\mathbb{P}(w_1, w_2)$  可以计算为  $w_1, w_2, w_3$  三词相邻的频率与  $w_1, w_2$  两词相邻的频率的比值。以此类推。

### 6.1.2 $N$ 元语法

实际上，我们可以通过马尔可夫假设（虽然并不一定成立）来简化语言模型的计算。基于  $n - 1$  阶马尔可夫假设，我们将语言模型改写为

$$\mathbb{P}(w_1, w_2, \dots, w_T) \approx \prod_{t=1}^T \mathbb{P}(w_t | w_{t-(n-1)}, \dots, w_{t-1}).$$

以上也叫  $n$  元语法 ( $n$ -grams)。它是基于  $n - 1$  阶马尔可夫链的概率语言模型。当  $n$  分别为 1、2 和 3 时，我们将其分别称作一元语法 (unigram)、二元语法 (bigram) 和三元语法 (trigram)。例如， $w_1, w_2, w_3, w_4$  在一元、二元和三元语法中的概率分别为

$$\begin{aligned}\mathbb{P}(w_1, w_2, w_3, w_4) &= \mathbb{P}(w_1)\mathbb{P}(w_2)\mathbb{P}(w_3)\mathbb{P}(w_4), \\ \mathbb{P}(w_1, w_2, w_3, w_4) &= \mathbb{P}(w_1)\mathbb{P}(w_2 | w_1)\mathbb{P}(w_3 | w_2)\mathbb{P}(w_4 | w_3), \\ \mathbb{P}(w_1, w_2, w_3, w_4) &= \mathbb{P}(w_1)\mathbb{P}(w_2 | w_1)\mathbb{P}(w_3 | w_1, w_2)\mathbb{P}(w_4 | w_2, w_3).\end{aligned}$$

当  $n$  较小时， $n$  元语法往往并不准确。例如，在一元语法中，由三个词组成的句子“你走先”和“你先走”的概率是一样的。然而，当  $n$  较大时， $n$  元语法需要计算并存储大量的词频和多词相邻频率。

那么，有没有方法在语言模型中更好地平衡以上这两点呢？我们将在本章探究这样的方法。

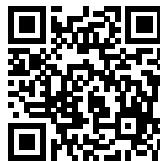
### 6.1.3 小结

- 语言模型是自然语言处理的重要技术。
- $N$  元语法是基于  $n - 1$  阶马尔可夫链的概率语言模型。但它有一定的局限性。

### 6.1.4 练习

- 假设训练数据集中有十万个词，四元语法需要存储多少词频和多词相邻频率？
- 你还能想到哪些语言模型的应用？

### 6.1.5 扫码直达讨论区



## 6.2 隐藏状态

上一节介绍的  $n$  元语法中，基于文本序列最近  $n - 1$  个词生成时间步  $t$  的词  $w_t$  的条件概率为

$$\mathbb{P}(w_t \mid w_{t-(n-1)}, \dots, w_{t-1}).$$

需要注意的是，以上概率并没有考虑到比  $t - (n - 1)$  更早时间步的词对  $w_t$  可能的影响。然而，考虑这些影响需要增大  $n$  的值，那么  $n$  元语法的模型参数的数量将随之呈指数级增长（可参考上一节的练习）。为了解决  $n$  元语法的局限性，我们可以在神经网络中引入隐藏状态。我们既要捕捉时间序列的历史信息，又希望模型参数的数量不随历史增长而增长。

### 6.2.1 不含隐藏状态的神经网络

让我们先回顾一下不含隐藏状态的神经网络，例如只有一个隐藏层的多层感知机。

给定样本数为  $n$ 、输入个数（特征数或特征向量维度）为  $x$  的小批量数据样本  $\mathbf{X} \in \mathbb{R}^{n \times x}$ 。设隐藏层的激活函数为  $\phi$ ，那么隐藏层的输出  $\mathbf{H} \in \mathbb{R}^{n \times h}$  计算为

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h),$$

其中权重参数  $\mathbf{W}_{xh} \in \mathbb{R}^{x \times h}$ ，偏差参数  $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ， $h$  为隐藏单元个数。上式相加的两项形状不同，因此将按照广播机制相加（参见“[数据操作](#)”一节）。把隐藏变量  $\mathbf{H}$  作为输出层的输入，且设输出个数为  $y$ （例如分类问题中的类别数），输出层的输出

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hy} + \mathbf{b}_y,$$

其中输出变量  $\mathbf{O} \in \mathbb{R}^{n \times y}$ ，输出层权重参数  $\mathbf{W}_{hy} \in \mathbb{R}^{h \times y}$ ，输出层偏差参数  $\mathbf{b}_y \in \mathbb{R}^{1 \times y}$ 。如果是分类问题，我们可以使用  $\text{softmax}(\mathbf{O})$  来计算输出类别的概率分布。

## 6.2.2 含隐藏状态的循环神经网络

现在我们考虑时间序列数据，并基于上面描述的多层感知机引入隐藏状态，从而构造循环神经网络。

假设  $\mathbf{X}_t \in \mathbb{R}^{n \times x}$  是序列中时间步  $t$  的小批量输入（样本数为  $n$ , 输入个数为  $x$ ），该时间步隐藏层变量是  $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ （隐藏单元个数为  $h$ , 是超参数），输出层变量是  $\mathbf{O}_t \in \mathbb{R}^{n \times y}$ （输出个数为  $y$ ）。

为了使隐藏层变量能够捕捉时间序列的历史信息，我们引入一个新的权重参数  $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ ，并且使当前时间步隐藏层变量同时取决于当前时间步输入  $\mathbf{X}_t$  和上一时间步隐藏层变量  $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ ：

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h).$$

这里的隐藏层变量又叫隐藏状态。通常，我们会将隐藏状态全部元素初始化为 0。隐藏状态捕捉了截至当前时间步的序列历史信息，就像是神经网络当前时间步的状态或记忆一样。神经网络下一时间步的隐藏状态既取决于下一时间步的输入，又取决于当前时间步的隐藏状态。如此循环往复。我们将此类神经网络称作循环神经网络。在时间步  $t$ ，循环神经网络的输出层输出和多层感知机中的计算类似：

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hy} + \mathbf{b}_y.$$

可见，循环神经网络在时间步  $t$  的输出基于相同时间步的隐藏状态。循环神经网络的参数包括隐藏层的权重  $\mathbf{W}_{xh} \in \mathbb{R}^{x \times h}$ ,  $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$  和偏差  $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ，以及输出层的权重  $\mathbf{W}_{hy} \in \mathbb{R}^{h \times y}$  和偏差  $\mathbf{b}_y \in \mathbb{R}^{1 \times y}$ 。值得一提的是，即便在不同时间步，循环神经网络始终使用这些模型参数。因此，循环神经网络模型参数的数量不随历史增长而增长。

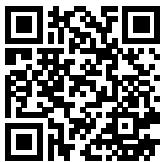
## 6.2.3 小结

- 循环神经网络通过引入隐藏状态来捕捉时间序列的历史信息。
- 循环神经网络模型参数的数量不随历史增长而增长。

## 6.2.4 练习

- 如果我们使用循环神经网络来预测一段文本序列的下一个词，输出个数应该是多少？
- 为什么循环神经网络可以表达某时间步的词基于文本序列中所有过去的词的条件概率？

## 6.2.5 扫码直达讨论区



## 6.3 循环神经网络——从零开始

前两节介绍了语言模型和循环神经网络的设计。在本节中，我们将从零开始实现一个基于循环神经网络的语言模型，并应用它创作歌词。循环神经网络还有更广泛的应用。我们将在“自然语言处理”篇章中使用循环神经网络对不定长的文本序列分类，或把它翻译成不定长的另一语言的文本序列。

### 6.3.1 基于循环神经网络的语言模型

首先让我们简单回顾一下上一节描述的循环神经网络表达式。给定时间步  $t$  的小批量输入  $\mathbf{X}_t \in \mathbb{R}^{n \times x}$  (样本数为  $n$ , 输入个数为  $x$ )，设该时间步隐藏状态为  $\mathbf{H}_t \in \mathbb{R}^{n \times h}$  (隐藏单元个数为  $h$ )，输出层变量为  $\mathbf{O}_t \in \mathbb{R}^{n \times y}$  (输出个数为  $y$ )，隐藏层的激活函数为  $\phi$ 。循环神经网络的矢量计算表达式为

$$\begin{aligned}\mathbf{H}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h), \\ \mathbf{O}_t &= \mathbf{H}_t \mathbf{W}_{hy} + \mathbf{b}_y,\end{aligned}$$

其中隐藏层的权重  $\mathbf{W}_{xh} \in \mathbb{R}^{x \times h}$ ,  $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$  和偏差  $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ，以及输出层的权重  $\mathbf{W}_{hy} \in \mathbb{R}^{h \times y}$  和偏差  $\mathbf{b}_y \in \mathbb{R}^{1 \times y}$  为循环神经网络的模型参数。

在语言模型中，输入个数  $x$  为任意词的特征向量长度（本节稍后将讨论）；输出个数  $y$  为语料库中所有可能的词的个数。对循环神经网络的输出做 softmax 运算，我们可以得到时间步  $t$  输出所有可能的词的概率分布  $\hat{\mathbf{Y}}_t \in \mathbb{R}^{n \times y}$ ：

$$\hat{\mathbf{Y}}_t = \text{softmax}(\mathbf{O}_t).$$

由于隐藏状态  $H_t$  捕捉了时间步 1 到时间步  $t$  的小批量输入  $\mathbf{X}_1, \dots, \mathbf{X}_t$  的信息， $\hat{\mathbf{Y}}_t$  可以批量表达语言模型中给定文本序列中过去词生成下一个词的条件概率。有了这些条件概率，语言模型可以计算任意文本序列的概率。

### 6.3.2 字符级循环神经网络

本节实验中的循环神经网络将每个字符视作词。我们有时将该模型称为字符级循环神经网络 (character-level recurrent neural network)。设小批量中样本数  $n = 1$ ，文本序列为“你”、“好”、“世”、“界”。为了表达给定文本序列中过去词生成下一个词的条件概率，我们需要把输入序列和标签序列分别设为“你”、“好”、“世”和“好”、“世”、“界”，如图 6.1 所示。

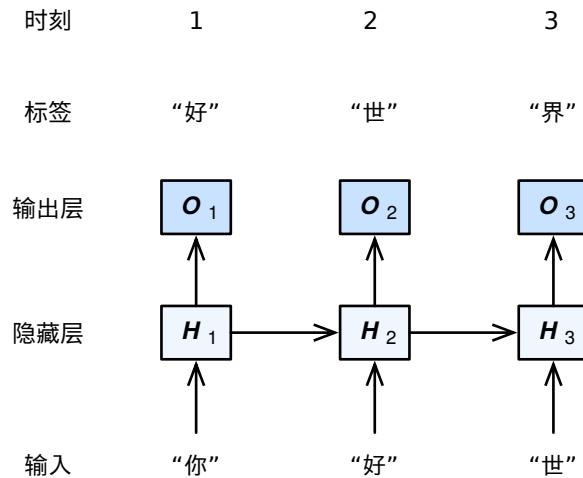


图 6.1：基于循环神经网络的语言模型。输入序列和标签序列分别为“你”、“好”、“世”和“好”、“世”、“界”。

当训练模型时，我们可以使用分类模型中常用的交叉熵损失函数计算各个时间步的损失。在图 6.1 中，由于隐藏层中隐藏状态的循环迭代，时间步 3 的输出  $O_3$  取决于文本序列“你”、“好”、“世”。由于训练数据中该序列的下一个词为“界”，时间步 3 的损失将取决于该时间步基于序列“你好世”生成下一个词的概率分布与该时间步标签“界”。

### 6.3.3 创作歌词

在创作歌词的实验中，我们将应用基于字符级循环神经网络的语言模型。与图 6.1 中的例子类似，我们将根据训练数据集的文本序列得到输入序列和标签序列。当模型训练好后，我们将以一种简单的方式创作歌词：根据给定的前缀，输出预测概率最大的下一个词；然后将该词附在前缀后继续输出预测概率最大的下一个词；如此循环。

创作歌词也可采用其他方式，例如将输入拆分成以词语而不是字符为单位的序列，或使用“自然语言处理”篇章中介绍的束搜索。

### 6.3.4 歌词数据集

我们使用周杰伦歌词数据集来训练模型作词。该数据集里包含了著名创作型歌手周杰伦从第一张专辑《Jay》到第十张专辑《跨时代》中歌曲的歌词。

首先导入实现所需的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, nd
        from mxnet.gluon import loss as gloss
        import random
        import zipfile
```

下面我们读取这个数据集，看看前 50 个字符是什么样的。

```
In [2]: with zipfile.ZipFile('../data/jaychou_lyrics.txt.zip', 'r') as zin:
            zin.extractall('../data/')

            with open('../data/jaychou_lyrics.txt') as f:
                corpus_chars = f.read()

corpus_chars[0:50]
```

Out[2]: '想要有直升机\n想要和你飞到宇宙去\n想要和你融化在一起\n融化在宇宙里\n我每天每天每天在想想想  
→ 想著你\n这'

看一下数据集中文本序列的长度。

```
In [3]: len(corpus_chars)
```

Out[3]: 63282

接着我们稍微处理下数据集。为了打印方便，我们把换行符替换成空格。我们使用序列的前两万个字符训练模型。

```
In [4]: corpus_chars = corpus_chars.replace('\n', ' ').replace('\r', ' ')
corpus_chars = corpus_chars[0:20000]
```

### 6.3.5 建立字符索引

我们将数据集里面所有不同的字符取出来做成词典。打印 `vocab_size`，即词典中不同字符的个数。

```
In [5]: idx_to_char = list(set(corpus_chars))
char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
vocab_size = len(char_to_idx)
vocab_size
```

```
Out[5]: 1447
```

然后，把每个字符转成从 0 开始的索引从而方便之后的使用。

```
In [6]: corpus_indices = [char_to_idx[char] for char in corpus_chars]
sample = corpus_indices[:40]
print('chars: \n', ''.join([idx_to_char[idx] for idx in sample]))
print('\nindices: \n', sample)

chars:
想要有直升机 想要和你飞到宇宙去 想要和你融化在一起 融化在宇宙里 我每天每天每

indices:
[1384, 1007, 470, 368, 623, 567, 636, 1384, 1007, 798, 391, 82, 595, 1245, 51, 355,
→ 636, 1384, 1007, 798, 391, 1014, 1215, 1015, 785, 558, 636, 1014, 1215, 1015,
→ 1245, 51, 181, 636, 826, 800, 1093, 800, 1093, 800]
```

### 6.3.6 时序数据的采样

同之前的实验一样，我们需要每次随机读取小批量样本和标签。不同的是，时序数据的一个样本通常包含连续的字符。假设时间步数为 5，样本序列为 5 个字符：“想”、“要”、“有”、“直”、“升”。那么该样本的标签序列为这些字符分别在训练集中的下一个字符：“要”、“有”、“直”、“升”、“机”。

我们有两种方式对时序数据采样，分别是随机采样和相邻采样。

## 随机采样

下面代码每次从数据里随机采样一个小批量。其中批量大小 `batch_size` 指每个小批量的样本数，`num_steps` 为每个样本所包含的时间步数。在随机采样中，每个样本是原始序列上任意截取的一段序列。相邻的两个随机小批量在原始序列上的位置不一定相毗邻。因此，我们无法用一个小批量最终时间步的隐藏状态来初始化下一个小批量的隐藏状态。在训练模型时，每次随机采样前都需要重新初始化隐藏状态。

```
In [7]: def data_iter_random(corpus_indices, batch_size, num_steps, ctx=None):
    # 减一是因为输出的索引是相应输入的索引加一。
    num_examples = (len(corpus_indices) - 1) // num_steps
    epoch_size = num_examples // batch_size
    example_indices = list(range(num_examples))
    random.shuffle(example_indices)

    def _data(pos):
        return corpus_indices[pos: pos+num_steps]

    for i in range(epoch_size):
        # 每次读取 batch_size 个随机样本。
        i = i * batch_size
        batch_indices = example_indices[i: i+batch_size]
        X = nd.array(
            [_data(j * num_steps) for j in batch_indices], ctx=ctx)
        Y = nd.array(
            [_data(j * num_steps + 1) for j in batch_indices], ctx=ctx)
        yield X, Y
```

让我们输入一个从 0 到 29 的人工序列，设批量大小和时间步数分别为 2 和 3，打印随机采样每次读取的小批量样本的输入 `X` 和标签 `Y`。可见，相邻的两个随机小批量在原始序列上的位置不一定相毗邻。

```
In [8]: my_seq = list(range(30))
for X, Y in data_iter_random(my_seq, batch_size=2, num_steps=3):
    print('X: ', X, '\nY: ', Y, '\n')

X:
[[ 18.  19.  20.]
 [ 21.  22.  23.]]
<NDArray 2x3 @cpu(0)>
Y:
[[ 19.  20.  21.]
 [ 22.  23.  24.]]
<NDArray 2x3 @cpu(0)>
```

```

X:
[[ 6.  7.  8.]
 [ 12. 13. 14.]]
<NDArray 2x3 @cpu(0)>
Y:
[[ 7.  8.  9.]
 [ 13. 14. 15.]]
<NDArray 2x3 @cpu(0)>

X:
[[ 3.  4.  5.]
 [ 24. 25. 26.]]
<NDArray 2x3 @cpu(0)>
Y:
[[ 4.  5.  6.]
 [ 25. 26. 27.]]
<NDArray 2x3 @cpu(0)>

X:
[[ 0.  1.  2.]
 [ 15. 16. 17.]]
<NDArray 2x3 @cpu(0)>
Y:
[[ 1.  2.  3.]
 [ 16. 17. 18.]]
<NDArray 2x3 @cpu(0)>

```

## 相邻采样

除了对原始序列做随机采样之外，我们还可以使相邻的两个随机小批量在原始序列上的位置相毗邻。这时候，我们就可以用一个小批量最终时间步的隐藏状态来初始化下一个小微量的隐藏状态，从而使下一个小微量的输出也取决于当前小微量输入，并如此循环下去。这对实现循环神经网络造成了两方面影响。一方面，在训练模型时，我们只需在每一个迭代周期开始时初始化隐藏状态。另一方面，当多个相邻小微量通过传递隐藏状态串联起来时，模型参数的梯度计算将依赖所有串联起来的小微量序列。同一迭代周期中，随着迭代次数的增加，梯度的计算开销会越来越大。为了使模型参数的梯度计算只依赖一次迭代读取的小微量序列，我们可以在每次读取小微量前将隐藏状态从计算图分离出来。

```
In [9]: def data_iter_consecutive(corpus_indices, batch_size, num_steps, ctx=None):
    corpus_indices = nd.array(corpus_indices, ctx=ctx)
```

```

data_len = len(corpus_indices)
batch_len = data_len // batch_size
indices = corpus_indices[0: batch_size*batch_len].reshape((
    batch_size, batch_len))
# 减一是因为输出的索引是相应输入的索引加一。
epoch_size = (batch_len - 1) // num_steps
for i in range(epoch_size):
    i = i * num_steps
    X = indices[:, i: i+num_steps]
    Y = indices[:, i+1: i+num_steps+1]
    yield X, Y

```

让我们输入一个从 0 到 29 的人工序列，设批量大小和时间步数分别为 2 和 3，打印相邻采样每次读取的小批量样本的输入 X 和标签 Y。相邻的两个随机小批量在原始序列上的位置相毗邻。

```

In [10]: my_seq = list(range(30))
        for X, Y in data_iter_consecutive(my_seq, batch_size=2, num_steps=3):
            print('X: ', X, '\nY: ', Y, '\n')

X:
[[ 0.   1.   2.]
 [ 15.  16.  17.]]
<NDArray 2x3 @cpu(0)>
Y:
[[ 1.   2.   3.]
 [ 16.  17.  18.]]
<NDArray 2x3 @cpu(0)>

X:
[[ 3.   4.   5.]
 [ 18.  19.  20.]]
<NDArray 2x3 @cpu(0)>
Y:
[[ 4.   5.   6.]
 [ 19.  20.  21.]]
<NDArray 2x3 @cpu(0)>

X:
[[ 6.   7.   8.]
 [ 21.  22.  23.]]
<NDArray 2x3 @cpu(0)>
Y:
[[ 7.   8.   9.]
 [ 22.  23.  24.]]

```

```
<NDArray 2x3 @cpu(0)>

X:
[[ 9. 10. 11.]
 [ 24. 25. 26.]]
<NDArray 2x3 @cpu(0)>
Y:
[[ 10. 11. 12.]
 [ 25. 26. 27.]]
<NDArray 2x3 @cpu(0)>
```

### 6.3.7 One-hot 向量

为了用向量表示词，一个简单的办法是使用 one-hot 向量。假设词典中不同字符的数量为  $N$ ，每个字符可以和从 0 到  $N - 1$  的连续整数一一对应。这些与字符对应的整数也叫字符的索引。如果一个字符的索引是整数  $i$ ，那么我们创建一个全 0 的长为 `vocab_size` 的向量，并将其位置为  $i$  的元素设成 1。该向量就是对原字符的 one-hot 向量。因此，本节实验中循环神经网络的输入个数  $x$  是任意词的特征向量长度 `vocab_size`。

下面分别展示了索引为 0 和 2 的 one-hot 向量。

```
In [11]: nd.one_hot(nd.array([0, 2]), vocab_size)

Out[11]:
[[ 1. 0. 0. ..., 0. 0. 0.]
 [ 0. 0. 1. ..., 0. 0. 0.]]
<NDArray 2x1447 @cpu(0)>
```

我们每次采样的小批量的形状是  $(batch\_size, num\_steps)$ 。下面这个函数将其转换成  $num\_steps$  个可以输入进网络的形状为  $(batch\_size, num\_steps)$  的矩阵。对于一个时间步数为  $num\_steps$  的序列，每个批量输入  $X \in \mathbb{R}^{n \times x}$ ，其中  $n = batch\_size$ ,  $x = vocab\_size$  (one-hot 向量长度)。

```
In [12]: def to_onehot(X, size):
    return [nd.one_hot(x, size) for x in X.T]

get_inputs = to_onehot
inputs = get_inputs(X, vocab_size)
len(inputs), inputs[0].shape

Out[12]: (3, (2, 1447))
```

### 6.3.8 初始化模型参数

接下来，我们初始化模型参数。隐藏单元个数 `num_hiddens` 是一个超参数。

```
In [13]: ctx = gb.try_gpu()
print('will use', ctx)

num_inputs = vocab_size
num_hiddens = 256
num_outputs = vocab_size

def get_params():
    # 隐藏层参数。
    W_xh = nd.random.normal(scale=0.01, shape=(num_inputs, num_hiddens),
                           ctx=ctx)
    W_hh = nd.random.normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                           ctx=ctx)
    b_h = nd.zeros(num_hiddens, ctx=ctx)
    # 输出层参数。
    W_hy = nd.random.normal(scale=0.01, shape=(num_hiddens, num_outputs),
                           ctx=ctx)
    b_y = nd.zeros(num_outputs, ctx=ctx)

    params = [W_xh, W_hh, b_h, W_hy, b_y]
    for param in params:
        param.attach_grad()
    return params

will use gpu(0)
```

### 6.3.9 定义模型

我们根据循环神经网络的表达式实现该模型。这里的激活函数使用了 `tanh` 函数。“多层神经网络”一节中介绍过，当元素在实数域上均匀分布时，`tanh` 函数值的均值为 0。

假设小批量中样本数为 `batch_size`，时间步数为 `num_steps`。以下 `rnn` 函数的 `inputs` 和 `outputs` 皆为 `num_steps` 个形状为  $(batch\_size, vocab\_size)$  的矩阵，隐藏状态 `H` 是一个形状为  $(batch\_size, num\_hiddens)$  的矩阵。

```
In [14]: def rnn(inputs, state, *params):
    H = state
    W_xh, W_hh, b_h, W_hy, b_y = params
    outputs = []
```

```

for X in inputs:
    H = nd.tanh(nd.dot(X, W_xh) + nd.dot(H, W_hh) + b_h)
    Y = nd.dot(H, W_ty) + b_y
    outputs.append(Y)
return outputs, H

```

做个简单的测试：

```

In [15]: state = nd.zeros(shape=(X.shape[0], num_hiddens), ctx=ctx)
params = get_params()
outputs, state_new = rnn(get_inputs(X.as_in_context(ctx), vocab_size), state,
                         *params)
len(outputs), outputs[0].shape, state_new.shape

Out[15]: (3, (2, 1447), (2, 256))

```

### 6.3.10 定义预测函数

以下函数预测基于前缀 `prefix` 接下来的 `num_chars` 个字符。我们将用它根据训练得到的循环神经网络 `rnn` 来创作歌词。

```

In [16]: def predict_rnn(rnn, prefix, num_chars, params, num_hiddens, vocab_size, ctx,
                      idx_to_char, char_to_idx, get_inputs, is_lstm=False):
    prefix = prefix.lower()
    state_h = nd.zeros(shape=(1, num_hiddens), ctx=ctx)
    if is_lstm:
        # 当 RNN 使用 LSTM 时才会用到 (后面章节会介绍), 本节可以忽略。
        state_c = nd.zeros(shape=(1, num_hiddens), ctx=ctx)
        output = [char_to_idx[prefix[0]]]
        for i in range(num_chars + len(prefix)):
            X = nd.array([output[-1]], ctx=ctx)
            # 在序列中循环迭代隐藏状态。
            if is_lstm:
                # 当 RNN 使用 LSTM 时才会用到 (后面章节会介绍), 本节可以忽略。
                Y, state_h, state_c = rnn(get_inputs(X, vocab_size), state_h,
                                           state_c, *params)
            else:
                Y, state_h = rnn(get_inputs(X, vocab_size), state_h, *params)
            if i < len(prefix) - 1:
                next_input = char_to_idx[prefix[i + 1]]
            else:
                next_input = int(Y[0].argmax(axis=1).asscalar())
            output.append(next_input)
        return ''.join([idx_to_char[i] for i in output])
    else:
        state_h = nd.zeros(shape=(1, num_hiddens), ctx=ctx)
        for i in range(num_chars + len(prefix)):
            X = nd.array([char_to_idx[prefix[i]]], ctx=ctx)
            Y, state_h = rnn(get_inputs(X, vocab_size), state_h, *params)
            next_input = int(Y[0].argmax(axis=1).asscalar())
            output.append(next_input)
        return ''.join([idx_to_char[i] for i in output])

```

### 6.3.11 裁剪梯度

循环神经网络中较容易出现梯度衰减或爆炸。我们会在[下一节](#)中解释原因。为了应对梯度爆炸，我们可以裁剪梯度 (clipping gradient)。假设我们把所有模型参数梯度的元素拼接成一个向量  $g$ ，并设裁剪的阈值是  $\theta$ 。裁剪后梯度

$$\min\left(\frac{\theta}{\|g\|}, 1\right)g$$

的  $L_2$  范数不超过  $\theta$ 。

```
In [17]: def grad_clipping(params, state_h, Y, theta, ctx):
    if theta is not None:
        norm = nd.array([0.0], ctx)
        for param in params:
            norm += (param.grad ** 2).sum()
        norm = norm.sqrt().asscalar()
        if norm > theta:
            for param in params:
                param.grad[:] *= theta / norm
```

### 6.3.12 定义模型训练函数

跟之前章节的训练模型函数相比，这里有以下几个不同。

1. 使用困惑度 (perplexity) 评价模型。
2. 在迭代模型参数前裁剪梯度。
3. 对时序数据采用不同采样方法将导致隐藏状态初始化的不同。

```
In [18]: def train_and_predict_rnn(rnn, is_random_iter, num_epochs, num_steps,
                                 num_hiddens, lr, clipping_theta, batch_size,
                                 vocab_size, pred_period, pred_len, prefixes,
                                 get_params, get_inputs, ctx, corpus_indices,
                                 idx_to_char, char_to_idx, is_lstm=False):
    if is_random_iter:
        data_iter = data_iter_random
    else:
        data_iter = data_iter_consecutive
    params = get_params()
    loss = gloss.SoftmaxCrossEntropyLoss()

    for epoch in range(1, num_epochs + 1):
```

```

# 如使用相邻采样，隐藏变量只需在该 epoch 开始时初始化。
if not is_random_iter:
    state_h = nd.zeros(shape=(batch_size, num_hiddens), ctx=ctx)
    if is_lstm:
        state_c = nd.zeros(shape=(batch_size, num_hiddens), ctx=ctx)
train_l_sum = nd.array([0], ctx=ctx)
train_l_cnt = 0
for X, Y in data_iter(corpus_indices, batch_size, num_steps, ctx):
    # 如使用随机采样，读取每个随机小批量前都需要初始化隐藏变量。
    if is_random_iter:
        state_h = nd.zeros(shape=(batch_size, num_hiddens), ctx=ctx)
        if is_lstm:
            state_c = nd.zeros(shape=(batch_size, num_hiddens),
                               ctx=ctx)
    # 如使用相邻采样，需要使用 detach 函数从计算图分离隐藏状态变量。
    else:
        state_h = state_h.detach()
        if is_lstm:
            state_c = state_c.detach()
    with autograd.record():
        # outputs 形状: (batch_size, vocab_size)。
        if is_lstm:
            outputs, state_h, state_c = rnn(
                get_inputs(X, vocab_size), state_h, state_c, *params)
        else:
            outputs, state_h = rnn(
                get_inputs(X, vocab_size), state_h, *params)
    # 设 t_ib_j 为时间步 i 批量中的元素 j:
    # y 形状: (batch_size * num_steps,)
    # y = [t_0b_0, t_0b_1, ..., t_1b_0, t_1b_1, ..., ]。
    y = Y.T.reshape((-1,))
    # 拼接 outputs，形状: (batch_size * num_steps, vocab_size)。
    outputs = nd.concat(*outputs, dim=0)
    l = loss(outputs, y)
    l.backward()
    # 裁剪梯度。
    grad_clipping(params, state_h, Y, clipping_theta, ctx)
    gb.sgd(params, lr, 1)
    train_l_sum = train_l_sum + l.sum()
    train_l_cnt += l.size
    if epoch % pred_period == 0:
        print("\nepoch %d, perplexity %f"
              % (epoch, (train_l_sum / train_l_cnt).exp().asscalar()))

```

```
for prefix in prefixes:  
    print(' - ', predict_rnn(  
        rnn, prefix, pred_len, params, num_hiddens, vocab_size,  
        ctx, idx_to_char, char_to_idx, get_inputs, is_lstm))
```

## 困惑度

回忆一下“[分类模型](#)”一节中交叉熵损失函数的定义。困惑度是对交叉熵损失函数做指数运算后得到的值。特别地，

- 最佳情况下，模型总是把标签类别的概率预测为 1。此时困惑度为 1。
- 最坏情况下，模型总是把标签类别的概率预测为 0。此时困惑度为正无穷。
- 基线情况下，模型总是预测所有类别的概率都相同。此时困惑度为类别数。

显然，任何一个有效模型的困惑度必须小于类别数。在本例中，困惑度必须小于词典中不同的字符数 `vocab_size`。

### 6.3.13 训练模型并创作歌词

以上介绍的 `to_onehot`、`data_iter_random`、`data_iter_consecutive`、`grad_clipping`、`predict_rnn` 和 `train_and_predict_rnn` 函数均定义在 `gluonbook` 包中供后面章节调用。有了这些函数以后，我们就可以训练模型了。

首先，设置模型超参数。我们将根据前缀“分开”和“不分开”分别创作长度为 100 个字符的一段歌词。我们每过 40 个迭代周期便根据当前训练的模型创作一段歌词。

```
In [19]: num_epochs = 200  
       num_steps = 35  
       batch_size = 32  
       lr = 0.2  
       clipping_theta = 5  
       prefixes = ['分开', '不分开']  
       pred_period = 40  
       pred_len = 100
```

下面采用随机采样训练模型并创作歌词。

```
In [20]: train_and_predict_rnn(rnn, True, num_epochs, num_steps, num_hiddens, lr,  
                           clipping_theta, batch_size, vocab_size, pred_period,
```

```
pred_len, prefixes, get_params, get_inputs, ctx,
corpus_indices, idx_to_char, char_to_idx)
```

epoch 40, perplexity 83.299210

- 分开 我想要你想说你 爱你的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人
- 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人
- 坏坏的让我疯狂的可爱女人 坏
- 不分开 我想不要 我不要这样 我不要你想 我不要你想 我不要你想 我不要你想 我不要你想 我不要你想
- 我不要你想 我不要你想 我不要你想 我不要你想 我不要你想 我不要你想 我不要你想 我不要你想
- 我不要你想

epoch 80, perplexity 13.577624

- 分开 我只想要你了天 什么我们你堡多样 别想这样 你就会感到更加沮丧 难道这不是我要的天堂景象
- 沉沦假象 你只会感到更加沮丧 难道这不是我要的天堂景象 沉沦假象 你只会感到更加沮丧 难道这不是
- 不分开吗 我根你看想你的非 这样的天我疯疼的可爱女人 坏坏的让我疯狂的可爱女人
- 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人
- 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人

epoch 120, perplexity 4.923253

- 分开 快静的让我面红的可爱女人 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人
- 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人
- 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可
- 不分开简 我叫你看想 这不是逃你 有在人真的来不 我 在小依 别沉银 娘一堂 旧属一种都会记 我看到也远
- 开人的距丽 是你的侧板 我慢想很想一点我 趁落猜强发迹 我的等我面开笑不能 想要你 别只对手跟一打 但

epoch 160, perplexity 2.930238

- 分开 有静不 有挡她人三 有盒我 想子你 别要对手 为漠我 别怪变 给 这道你 我知是这样的 你
- 古我胸恨已经 一直都了天场 我说想再你一遍 从身为龙梦故兄 我会天地远封开人帮水掏空 人在古老河背
- 配就
- 不分开简简一直走 我也眼自会 没人了战滴 这里就 爱皮箱的烟味 什么声 让人开 什么却有 沙漠页 的日段
- 有一些人霜 老唱了 旧皮箱 装属于那了片的铁盒 一切一点木著 他在我遇见你是碎没人你 没有多你的风争来

epoch 200, perplexity 2.300173

- 分开 我沉好醒生活 爱你想想好多离开分 看着轻你在你在等向来 这样烟废想过就在回着
- 话狼完飞过一句默鸥向的眼等胡 用睛丁文念咒满一种好 印在一种实停要你的回忆你没听过
- 谁在用琵琶弹奏一曲东风破 枫叶将故上染
- 不分开扫 我不能再想 我不 我不能再想你 不知不觉 你已经离开你 不知不觉 再跟好离 我该透好生记
- 我右拳打开了天 化身为龙 把山河重新移动 填平裂缝 将东方 的日有调剩下一种 等待英雄 我就是那条龙

接下来采用相邻采样训练模型并创作歌词。

```
In [21]: train_and_predict_rnn(rnn, False, num_epochs, num_steps, num_hiddens, lr,
                           clipping_theta, batch_size, vocab_size, pred_period,
                           pred_len, prefixes, get_params, get_inputs, ctx,
```

```
corpus_indices, idx_to_char, char_to_idx)
```

epoch 40, perplexity 63.798656

- 分开 我不能再想 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想  
→ 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想 我不要再想
- 不分开 一天到 的灵魂 翻滚 我想 我不能 爱情的天 有一种 在我的可 你说了一直 篮你的可爱 我的世界  
→ 我不要 别你 篮你的天 我们了 是你 我想着我想要 你知着我 说你的天 我说了 我不见 可爱人

epoch 80, perplexity 8.934068

- 分开 这不是 爱想走 看是我的别球 有伤是一步秋步 一壶好力 再世地心脏汹涌 不安跳动 全世界  
→ 的表情只剩下一种 等待英雄 我就是那条龙 我右拳打开了天 化身为龙 把世界心脏汹涌 不安跳动 全世界  
→ 的表情只
- 不分开 我怎么这不要 你没有这不到 你在你也多难过 我想能慢慢走开 为什么有一起我 是你心了汉  
→ 我的世界将被摧毁 也许我看多难 不想这不是我的手道 好伤到一种人能 你想好远 开世的人尘刻 一起正着  
→ 我们了的

epoch 120, perplexity 3.534548

- 分开 我不要这想 我不要再想 我不 我不 我不能 爱情走的太快就像龙卷风 不能承受我已无处可躲  
→ 我不要再想 我不要再想 我不 我不 我不能 爱情走的太快就像龙卷风 不能承受我已无处可躲 我不要再想  
→ 我不要再
- 不分开 你我想感你对的快 想录回斯的世号 被人风依靠 有一样人留 不容就有 在家村风屋 白色蜡烛  
→ 温暖了空屋 白色蜡烛 温暖了空屋 白色蜡烛 温暖了空屋 白色蜡烛 温暖了空屋 白色蜡烛 温暖了空屋  
→ 白色蜡

epoch 160, perplexity 2.493779

- 分开 我不要 不想他 太神伦是我的左手 换取被宽 清晨我早已功猜透看透 我拉远 想想开你一手四  
→ 这么好好的书 就本么觉半 是天没有运球 篮什么传 快面了双起极 塞生水怯的我 相思寄红豆 相思寄红豆无  
→ 为什么
- 不分开 让我们 你兽人 爱手就的手沙 想伤是 瞎过了 什么我有直球开 但使用留 神山的人尘刻围了我  
→ 好暗好暗 铁盒的钥匙我找不到 我在糖空为你再多 誓言翼着重比谁河容b 印地安斑鸠 会学人开口  
→ 仙人掌怕羞 蜗

epoch 200, perplexity 2.122915

- 分开 我不要这想 黑不是篮滴 我留三拖来 再不在篮落 有战在真驳 到底就么驳 三底就 它杰  
→ 是否在海角对面 难因九岁才知道浪费时着 小茶妈世纪的模样 说著我 选想着你怎么久 我想要你来微没人  
→ 你不不受 你打
- 不分开 让我们 半兽人 的灵魂 单纯 而远贪婪着永恒 只对暴力忠诚 让我们的半乐 想在等我 说你说  
→ 犹数怎么停留 一直在停留 谁让它停留的 为什么我女朋睡场外加油 你在之那是底大日空念瘦  
→ 那向一名到底要多强

### 6.3.14 小结

- 我们可以应用基于字符级循环神经网络的语言模型来创作歌词。
- 时序数据采样方式包括随机采样和相邻采样。使用这两种方式的循环神经网络训练略有不同。
- 当训练循环神经网络时，为了应对梯度爆炸，我们可以裁剪梯度。
- 困惑度是对交叉熵损失函数做指数运算后得到的值。

### 6.3.15 练习

- 调调超参数，观察并分析对运行时间、困惑度以及创作歌词的结果造成的影响。
- 不裁剪梯度，运行本节代码。结果会怎样？
- 将 `pred_period` 改为 1，观察未充分训练的模型（困惑度高）是如何创作歌词的。你获得了什么启发？
- 将相邻采样改为不从计算图分离隐藏状态，运行时间有没有变化？
- 将本节中使用的激活函数替换成 ReLU，重复本节的实验。

### 6.3.16 扫码直达讨论区



## 6.4 通过时间反向传播

如果你做了上一节的练习，你会发现，如果不裁剪梯度，模型将无法正常训练。为了深刻理解这一现象，本节将介绍循环神经网络中梯度的计算和存储方法，即通过时间反向传播 (back-propagation through time)。

我们在“正向传播和反向传播”一节中介绍了神经网络中梯度计算与存储的一般思路，并强调正向传播和反向传播相互依赖。正向传播在循环神经网络比较直观。通过时间反向传播其实是反向传播在循环神经网络的具体应用。我们需要将循环神经网络按时间步展开，从而得到模型变量和参数之间的依赖关系，并依据链式法则应用反向传播计算并存储梯度。

### 6.4.1 定义模型

为了简洁，我们考虑一个无偏差项的循环神经网络，且激活函数的输入输出相同。

设时间步  $t$  的输入为  $\mathbf{x}_t \in \mathbb{R}^x$ ，标签为  $y_t$ ，隐藏状态  $\mathbf{h}_t \in \mathbb{R}^h$  的计算表达式为

$$\mathbf{h}_t = \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1},$$

其中  $\mathbf{W}_{hx} \in \mathbb{R}^{h \times x}$  和  $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$  是隐藏层权重参数。设输出层权重参数  $\mathbf{W}_{yh} \in \mathbb{R}^{y \times h}$ ，时间步  $t$  的输出层变量  $\mathbf{o}_t \in \mathbb{R}^y$  计算为

$$\mathbf{o}_t = \mathbf{W}_{yh}\mathbf{h}_t.$$

设时间步  $t$  的损失为  $\ell(\mathbf{o}_t, y_t)$ 。时间步数为  $T$  的损失函数  $L$  定义为

$$L = \frac{1}{T} \sum_{t=1}^T \ell(\mathbf{o}_t, y_t).$$

我们将  $L$  叫做有关给定时间步数的数据样本的目标函数，并在以下的讨论中简称目标函数。

### 6.4.2 模型计算图

为了可视化模型变量和参数之间在计算中的依赖关系，我们可以绘制模型计算图，如图 6.2 所示。例如，时间步 3 的隐藏状态  $\mathbf{h}_3$  的计算依赖模型参数  $\mathbf{W}_{hx}, \mathbf{W}_{hh}$ 、上一时间步隐藏状态  $\mathbf{h}_2$  以及当前时间步输入  $\mathbf{x}_3$ 。

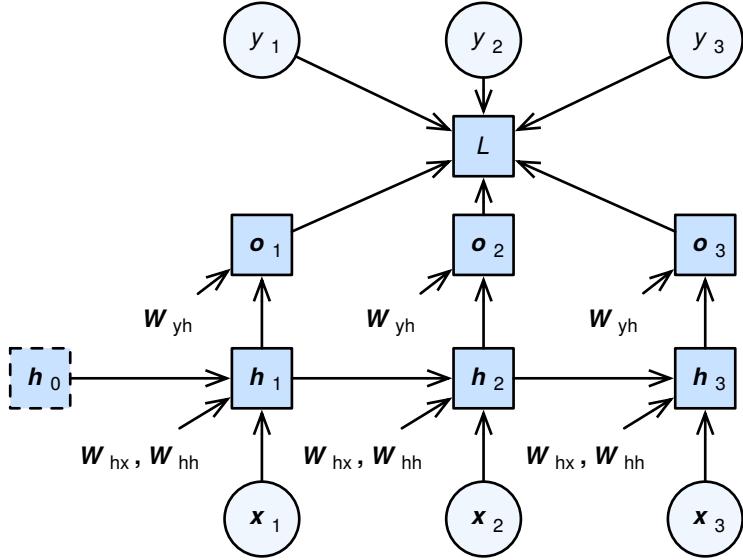


图 6.2: 时间步数为 3 的循环神经网络模型计算中的依赖关系。方框中字母代表变量，圆圈中字母代表数据样本特征和标签，无边框的字母代表模型参数。

### 6.4.3 通过时间反向传播

刚刚提到，图 6.2 中模型的参数是  $\mathbf{W}_{hx}$ 、 $\mathbf{W}_{hh}$  和  $\mathbf{W}_{yh}$ 。与“正向传播和反向传播”一节中类似，训练模型通常需要模型参数的梯度  $\partial L / \partial \mathbf{W}_{hx}$ 、 $\partial L / \partial \mathbf{W}_{hh}$  和  $\partial L / \partial \mathbf{W}_{yh}$ 。根据图 6.2 中的依赖关系，我们可以按照其中箭头所指的反方向依次计算并存储梯度。

为了表述方便，我们依然使用“正向传播和反向传播”一节中表达链式法则的操作符 `prod`。

首先，目标函数有关各时间步输出层变量的梯度  $\partial L / \partial \mathbf{o}_t \in \mathbb{R}^y$  可以很容易地计算：

$$\frac{\partial L}{\partial \mathbf{o}_t} = \frac{\partial \ell(\mathbf{o}_t, y_t)}{T \cdot \partial \mathbf{o}_t}.$$

下面，我们可以计算目标函数有关模型参数  $\mathbf{W}_{yh}$  的梯度  $\partial L / \partial \mathbf{W}_{yh} \in \mathbb{R}^{y \times h}$ 。根据图 6.2， $L$  通过  $\mathbf{o}_1, \dots, \mathbf{o}_T$  依赖  $\mathbf{W}_{yh}$ 。依据链式法则，

$$\frac{\partial L}{\partial \mathbf{W}_{yh}} = \sum_{t=1}^T \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{yh}}\right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{o}_t} \mathbf{h}_t^\top$$

其次，我们注意到隐藏状态之间也有依赖关系。在图 6.2 中， $L$  只通过  $\mathbf{o}_T$  依赖最终时间步  $T$  的隐藏状态  $\mathbf{h}_T$ 。因此，我们先计算目标函数有关最终时间步隐藏状态的梯度  $\partial L / \partial \mathbf{h}_T \in \mathbb{R}^h$ 。依据

链式法则，我们得到

$$\frac{\partial L}{\partial \mathbf{h}_T} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_T}, \frac{\partial \mathbf{o}_T}{\partial \mathbf{h}_T}\right) = \mathbf{W}_{yh}^\top \frac{\partial L}{\partial \mathbf{o}_T}.$$

接下来，对于时间步  $t < T$ ，在图 6.2 中， $L$  通过  $\mathbf{h}_{t+1}$  和  $\mathbf{o}_t$  依赖  $\mathbf{h}_t$ 。依据链式法则，目标函数有关时间步  $t < T$  的隐藏状态的梯度  $\partial L / \partial \mathbf{h}_t \in \mathbb{R}^h$  需要按照时间步从晚到早依次计算：

$$\frac{\partial L}{\partial \mathbf{h}_t} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_{t+1}}, \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}\right) + \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\right) = \mathbf{W}_{hh}^\top \frac{\partial L}{\partial \mathbf{h}_{t+1}} + \mathbf{W}_{yh}^\top \frac{\partial L}{\partial \mathbf{o}_t}.$$

将上面的递归公式展开，对任意时间步  $1 \leq t \leq T$ ，我们可以得到目标函数有关隐藏状态梯度的通项公式

$$\frac{\partial L}{\partial \mathbf{h}_t} = \sum_{i=t}^T (\mathbf{W}_{hh}^\top)^{T-i} \mathbf{W}_{yh}^\top \frac{\partial L}{\partial \mathbf{o}_{T+t-i}}.$$

由上式中的指数项可见，当时间步数  $T$  较大或者时间步  $t$  较小，目标函数有关隐藏状态的梯度较容易出现衰减和爆炸。这也会影响其他计算中包含  $\partial L / \partial \mathbf{h}_t$  的梯度，例如隐藏层中模型参数的梯度  $\partial L / \partial \mathbf{W}_{hx} \in \mathbb{R}^{h \times x}$  和  $\partial L / \partial \mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ 。在图 6.2 中， $L$  通过  $\mathbf{h}_1, \dots, \mathbf{h}_T$  依赖这些模型参数。依据链式法则，我们有

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{W}_{hx}} &= \sum_{t=1}^T \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hx}}\right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{x}_t^\top, \\ \frac{\partial L}{\partial \mathbf{W}_{hh}} &= \sum_{t=1}^T \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}\right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{h}_{t-1}^\top. \end{aligned}$$

“正向传播和反向传播”一节里解释过，每次迭代中，上述各个依次计算出的梯度会被依次存储或更新。这是为了避免重复计算。例如，由于隐藏状态梯度  $\partial L / \partial \mathbf{h}_t$  被计算存储，之后的模型参数梯度  $\partial L / \partial \mathbf{W}_{hx}$  和  $\partial L / \partial \mathbf{W}_{hh}$  的计算可以直接读取  $\partial L / \partial \mathbf{h}_t$  的值，而无需重复计算。此外，反向传播对于各层中变量和参数的梯度计算可能会依赖通过正向传播计算出的各层变量的当前值。举例来说，参数梯度  $\partial L / \partial \mathbf{W}_{hh}$  的计算需要依赖隐藏状态在时间步  $t = 0, \dots, T-1$  的当前值  $\mathbf{h}_t$  ( $\mathbf{h}_0$  是初始化得到的)。这些值是通过从输入层到输出层的正向传播计算并存储得到的。

#### 6.4.4 小结

- 通过时间反向传播是反向传播在循环神经网络的具体应用。
- 当时间步数较大时，循环神经网络的梯度较容易衰减或爆炸。

### 6.4.5 练习

- 除了梯度裁剪，你还能想到别的什么方法应对循环神经网络中的梯度爆炸？

### 6.4.6 扫码直达讨论区



## 6.5 门控循环单元（GRU）——从零开始

上一节介绍了循环神经网络中的梯度计算方法。我们发现，循环神经网络的梯度可能会衰减或爆炸。虽然裁剪梯度可以应对梯度爆炸，但无法解决梯度衰减的问题。给定一个时间序列，例如文本序列，循环神经网络在实际中较难捕捉时间步距离较大的词之间的依赖关系。

门控循环神经网络（gated recurrent neural network）的提出，是为了更好地捕捉时间序列中时间步距离较大的依赖关系。其中，门控循环单元（gated recurrent unit，简称 GRU）是一种常用的门控循环神经网络 [1, 2]。我们将在下一节介绍另一种门控循环神经网络：长短期记忆。

### 6.5.1 门控循环单元

下面将介绍门控循环单元的设计。它引入了门的概念，从而修改了循环神经网络中隐藏状态的计算方式。输出层的设计不变。

#### 重置门和更新门

假设隐藏单元个数为  $h$ ，给定时间步  $t$  的小批量输入  $\mathbf{X}_t \in \mathbb{R}^{n \times x}$ （样本数为  $n$ ，输入个数为  $x$ ）和上一时间步隐藏状态  $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ 。重置门（reset gate） $\mathbf{R}_t \in \mathbb{R}^{n \times h}$  和更新门（update gate）

$\mathbf{Z}_t \in \mathbb{R}^{n \times h}$  的计算如下：

$$\begin{aligned}\mathbf{R}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r), \\ \mathbf{Z}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z).\end{aligned}$$

其中  $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{x \times h}$  和  $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$  是权重参数， $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$  是偏移参数。激活函数  $\sigma$  是 sigmoid 函数。“多层神经网络”一节中介绍过，sigmoid 函数可以将元素的值变换到 0 和 1 之间。因此，重置门  $\mathbf{R}_t$  和更新门  $\mathbf{Z}_t$  中每个元素的值域都是  $[0, 1]$ 。

我们可以通过元素值域在  $[0, 1]$  的更新门和重置门来控制隐藏状态中信息的流动：这通常可以应用按元素乘法符  $\odot$ 。

### 候选隐藏状态

接下来，时间步  $t$  的候选隐藏状态  $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$  的计算使用了值域在  $[-1, 1]$  的  $\tanh$  函数做激活函数。它在之前描述的循环神经网络隐藏状态表达式的基础上，引入了重置门和按元素乘法：

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{R}_t \odot \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h),$$

其中  $\mathbf{W}_{xh} \in \mathbb{R}^{x \times h}$  和  $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$  是权重参数， $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$  是偏移参数。需要注意的是，候选隐藏状态使用了重置门，从而控制包含时间序列历史信息的上一个时间步的隐藏状态如何流入当前时间步的候选隐藏状态。如果重置门近似 0，上一个隐藏状态将被丢弃。因此，重置门可以丢弃与预测未来无关的历史信息。

### 隐藏状态

最后，隐藏状态  $\mathbf{H}_t \in \mathbb{R}^{n \times h}$  的计算使用更新门  $\mathbf{Z}_t$  来对上一时间步的隐藏状态  $\mathbf{H}_{t-1}$  和当前时间步的候选隐藏状态  $\tilde{\mathbf{H}}_t$  做组合：

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.$$

值得注意的是，更新门可以控制隐藏状态应该如何被包含当前时间步信息的候选隐藏状态所更新。假设更新门在时间步  $t'$  到  $t$  ( $t' < t$ ) 之间一直近似 1。那么，在时间步  $t'$  到  $t$  之间的输入信息几乎没有流入时间步  $t$  的隐藏状态  $\mathbf{H}_t$ 。实际上，这可以看作是较早时刻的隐藏状态  $\mathbf{H}_{t'-1}$  一直通过时间保存并传递至当前时间步  $t$ 。这个设计可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时间序列中时间步距离较大的依赖关系。

我们对门控循环单元的设计稍作总结：

- 重置门有助于捕捉时间序列里短期的依赖关系。
- 更新门有助于捕捉时间序列里长期的依赖关系。

## 6.5.2 实验

为了实现并展示门控循环单元，我们依然使用周杰伦歌词数据集来训练模型作词。这里除门控循环单元以外的实现已在“循环神经网络——从零开始”一节中介绍。

### 处理数据

我们先读取并简单处理数据集。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import nd
        import zipfile

        with zipfile.ZipFile('../data/jaychou_lyrics.txt.zip', 'r') as zin:
            zin.extractall('../data/')
        with open('../data/jaychou_lyrics.txt') as f:
            corpus_chars = f.read()

        corpus_chars = corpus_chars.replace('\n', ' ').replace('\r', ' ')
        corpus_chars = corpus_chars[0:20000]
        idx_to_char = list(set(corpus_chars))
        char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
        corpus_indices = [char_to_idx[char] for char in corpus_chars]
        vocab_size = len(char_to_idx)
```

### 初始化模型参数

以下部分对模型参数进行初始化。超参数 `num_hiddens` 定义了隐藏单元的个数。

```
In [2]: ctx = gb.try_gpu()
        num_inputs = vocab_size
        num_hiddens = 256
        num_outputs = vocab_size

        def get_params():
```

```

# 更新门参数。
W_xz = nd.random_normal(scale=0.01, shape=(num_inputs, num_hiddens),
                        ctx=ctx)
W_hz = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                        ctx=ctx)
b_z = nd.zeros(num_hiddens, ctx=ctx)
# 重置门参数。
W_xr = nd.random_normal(scale=0.01, shape=(num_inputs, num_hiddens),
                        ctx=ctx)
W_hr = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                        ctx=ctx)
b_r = nd.zeros(num_hiddens, ctx=ctx)
# 候选隐藏状态参数。
W_xh = nd.random_normal(scale=0.01, shape=(num_inputs, num_hiddens),
                        ctx=ctx)
W_hh = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                        ctx=ctx)
b_h = nd.zeros(num_hiddens, ctx=ctx)
# 输出层参数。
W_hy = nd.random_normal(scale=0.01, shape=(num_hiddens, num_outputs),
                        ctx=ctx)
b_y = nd.zeros(num_outputs, ctx=ctx)

params = [W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hy, b_y]
for param in params:
    param.attach_grad()
return params

```

### 6.5.3 定义模型

下面根据门控循环单元的计算表达式定义模型。

```
In [3]: def gru_rnn(inputs, H, *params):
    W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hy, b_y = params
    outputs = []
    for X in inputs:
        Z = nd.sigmoid(nd.dot(X, W_xz) + nd.dot(H, W_hz) + b_z)
        R = nd.sigmoid(nd.dot(X, W_xr) + nd.dot(H, W_hr) + b_r)
        H_tilda = nd.tanh(nd.dot(X, W_xh) + R * nd.dot(H, W_hh) + b_h)
        H = Z * H + (1 - Z) * H_tilda
        Y = nd.dot(H, W_hy) + b_y
        outputs.append(Y)
    return (outputs, H)
```

## 训练模型并创作歌词

设置好超参数后，我们将训练模型并根据前缀“分开”和“不分开”分别创作长度为 100 个字符的一段歌词。我们每过 30 个迭代周期便根据当前训练的模型创作一段歌词。训练模型时采用了相邻采样。

```
In [4]: get_inputs = gb.to_onehot
        num_epochs = 150
        num_steps = 35
        batch_size = 32
        lr = 0.25
        clipping_theta = 5
        prefixes = ['分开', '不分开']
        pred_period = 30
        pred_len = 100

        gb.train_and_predict_rnn(gru_rnn, False, num_epochs, num_steps, num_hiddens,
                                lr, clipping_theta, batch_size, vocab_size,
                                pred_period, pred_len, prefixes, get_params,
                                get_inputs, ctx, corpus_indices, idx_to_char,
                                char_to_idx)

epoch 30, perplexity 114.506500
- 分开 我不能再想你 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不要我想你
→ 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不
- 不分开 我不能再想你 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不要
→ 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不要我想你 我不

epoch 60, perplexity 10.057642
- 分开 我就是你心碎开人 你看着你看离开 这样的钥匙我的红道 没有一直一直到要多强
→ 连成线一点两步三步四步望著天 看星星 一颗两颗三颗四颗 连成线背著背 有一种热昏 的天出 一切莫重重
→ 老色哈秋 快使用双截棍
- 不分开 我想就这样天 你的经应还是我 想散你的国 我说的可爱的没人前 深埋在美索不达米亚平原
→ 几楔形文字后的誓头 一切又重演 我的世界将被摧毁 他许着没有 我马儿有些瘦 化身为龙 那天黄人脏汹
→ 塞北的客栈人多

epoch 90, perplexity 1.917150
- 分开 我也想就想就开 为什么我连分开都迁就着你 我真的没有天份 安静的没这么快 我会学着放弃你
→ 是因为我太爱你 是因为我太爱你 该不该搁下重重的壳 这里到底哪里背 有谁在安悟的黑 随时间备豆袭
→ 让我们 半兽
- 不分开 我想要的生笑 没有你在嘴离 没有黑间的勇 我会一直好好过 你已经远远离开 我也会慢慢走开
→ 为什么我连分开都迁就着你 我真的没有天份 安静的没这么快 我会学着放弃你 是因为我太爱你
→ 是因为我太爱你 该
```

epoch 120, perplexity 1.135159

- 分开 我也能慢想 微迷 为什么很了我 说散 你想很久了吧？败给你的黑色幽默 说散 你想很久了吧？
  - ↪ 我的认真败给黑色幽默 走过了很多地方 我来到伊斯坦堡 就像是童话故事 有教堂有城堡 每天忙碌地的寻找
  - ↪ 到
- 不分开 在角的觉育吹了一曲弓破 他看将我的世界 想你说着汉堡 这样也撑交 有伤堂人运你 说透了真的画面
  - ↪ 残绪是蒙蒙的雾 父亲还在我 想要你的想叫我回爱不听 去择来自己单就了空 为什么这种速度你做不到 不好

epoch 150, perplexity 1.061613

- 分开 我留着陪你 强忍着泪滴 有些事真的来不及回不去 你脸在抽搐 就快没力气 家乡事不准我再提
  - ↪ 我留着陪你 最后的距离 是你的侧脸倒在我的怀里 你慢慢睡去 我摇不醒你 泪水在战壕里决了堤
  - ↪ 泪水在战壕里决了堤
- 不分开 在角的消育吹了一记色破破 那长城那年染的誓言 一切又重演 爱在西元前 爱话让我的朋据
  - ↪ 没人莹说的画度 维持纯白的象征 然后还原为人 让我们 半兽人 的灵魂 翻滚 收起残忍 回忆兽化的过程
  - ↪ 让我们 半

## 6.5.4 小结

- 门控循环神经网络的可以更好地捕捉时间序列中时间步距离较大的依赖关系，它包括门控循环单元和长短期记忆。
- 门控循环单元引入了门的概念，从而修改了循环神经网络中隐藏状态的计算方式。它包括重置门、更新门、候选隐藏状态和隐藏状态。
- 重置门有助于捕捉时间序列里短期的依赖关系。
- 更新门有助于捕捉时间序列里长期的依赖关系。

## 6.5.5 练习

- 假设时间步  $t' < t$ 。如果我们只希望用时间步  $t'$  的输入来预测时间步  $t$  的输出，每个时间步的重置门和更新门的值最好是多少？
- 调调超参数，观察并分析对运行时间、困惑度以及创作歌词的结果造成的影响。
- 在相同条件下，比较门控循环单元和循环神经网络的运行时间。

### 6.5.6 扫码直达讨论区



### 6.5.7 参考文献

- [1] Cho, Kyunghyun, et al. “On the properties of neural machine translation: Encoder-decoder approaches.” arXiv:1409.1259 (2014).
- [2] Chung, Junyoung, et al. “Empirical evaluation of gated recurrent neural networks on sequence modeling.” arXiv:1412.3555 (2014).

## 6.6 长短期记忆 (LSTM)——从零开始

本节将介绍另一种常用的门控循环神经网络：长短期记忆 (long short-term memory, 简称 LSTM) [1]。它比门控循环单元的结构稍微更复杂一点。

### 6.6.1 长短期记忆

我们先介绍长短期记忆的设计。它修改了循环神经网络隐藏状态的计算方式，并引入了与隐藏状态形状相同的记忆细胞（某些文献把记忆细胞当成一种特殊的隐藏状态）。

#### 输入门、遗忘门和输出门

假设隐藏单元个数为  $h$ ，给定时间步  $t$  的小批量输入  $\mathbf{X}_t \in \mathbb{R}^{n \times x}$ （样本数为  $n$ ，输入个数为  $x$ ）和上一时间步隐藏状态  $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ 。时间步  $t$  的输入门 (input gate)  $\mathbf{I}_t \in \mathbb{R}^{n \times h}$ 、遗忘门 (forget

gate)  $\mathbf{F}_t \in \mathbb{R}^{n \times h}$  和输出门 (output gate)  $\mathbf{O}_t \in \mathbb{R}^{n \times h}$  分别计算如下：

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i), \\ \mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f), \\ \mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o).\end{aligned}$$

其中的  $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{x \times h}$  和  $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$  是权重参数， $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$  是偏移参数。激活函数  $\sigma$  是 sigmoid 函数。和门控循环单元中的重置门和更新门一样，这里的输入门、遗忘门和输出门中每个元素的值域都是  $[0, 1]$ 。

## 候选记忆细胞

和门控循环单元中的候选隐藏状态一样，时间步  $t$  的候选记忆细胞  $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$  也使用了值域在  $[-1, 1]$  的  $\tanh$  函数做激活函数。它的计算和不带门控的循环神经网络的隐藏状态的计算没什么区别：

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c).$$

其中的  $\mathbf{W}_{xc} \in \mathbb{R}^{x \times h}$  和  $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$  是权重参数， $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$  是偏移参数。

## 记忆细胞

我们可以通过元素值域在  $[0, 1]$  的输入门、遗忘门和输出门来控制隐藏状态中信息的流动：这通常可以应用按元素乘法符  $\odot$ 。当前时间步记忆细胞  $\mathbf{C}_t \in \mathbb{R}^{n \times h}$  的计算组合了上一时间步记忆细胞和当前时间步候选记忆细胞的信息，并通过遗忘门和输入门来控制信息的流动：

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t.$$

需要注意的是，如果遗忘门一直近似 1 且输入门一直近似 0，过去的记忆细胞将一直通过时间保存并传递至当前时间步。这个设计可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时序数据中间隔较大的依赖关系。

## 隐藏状态

有了记忆细胞以后，接下来我们还可以通过输出门来控制从记忆细胞到隐藏状态  $\mathbf{H}_t \in \mathbb{R}^{n \times h}$  的信息的流动：

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t).$$

这里的  $\tanh$  函数确保隐藏状态元素值在 -1 到 1 之间。需要注意的是，当输出门近似 1，记忆细胞信息将传递到隐藏状态供输出层使用；当输出门近似 0，记忆细胞信息只自己保留。

## 输出层

在时间步  $t$ ，长短期记忆的输出层计算和之前描述的循环神经网络输出层计算一样：我们只需将该时刻的隐藏状态  $H_t$  传递进输出层，从而计算时间步  $t$  的输出。

### 6.6.2 实验

和前几节中的实验一样，我们依然使用周杰伦歌词数据集来训练模型作词。

#### 处理数据

我们先读取并简单处理数据集。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import nd
        import zipfile

        with zipfile.ZipFile('../data/jaychou_lyrics.txt.zip', 'r') as zin:
            zin.extractall('../data/')
        with open('../data/jaychou_lyrics.txt') as f:
            corpus_chars = f.read()

        corpus_chars = corpus_chars.replace('\n', ' ').replace('\r', ' ')
        corpus_chars = corpus_chars[0:20000]
        idx_to_char = list(set(corpus_chars))
        char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
        corpus_indices = [char_to_idx[char] for char in corpus_chars]
        vocab_size = len(char_to_idx)
```

#### 初始化模型参数

以下部分对模型参数进行初始化。超参数 `num_hiddens` 定义了隐藏单元的个数。

```
In [2]: ctx = gb.try_gpu()
        input_dim = vocab_size
        num_hiddens = 256
        output_dim = vocab_size

def get_params():
    # 输入门参数。
    W_xi = nd.random_normal(scale=0.01, shape=(input_dim, num_hiddens),
                           ctx=ctx)
    W_hi = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                           ctx=ctx)
    b_i = nd.zeros(num_hiddens, ctx=ctx)
    # 遗忘门参数。
    W_xf = nd.random_normal(scale=0.01, shape=(input_dim, num_hiddens),
                           ctx=ctx)
    W_hf = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                           ctx=ctx)
    b_f = nd.zeros(num_hiddens, ctx=ctx)
    # 输出门参数。
    W_xo = nd.random_normal(scale=0.01, shape=(input_dim, num_hiddens),
                           ctx=ctx)
    W_ho = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                           ctx=ctx)
    b_o = nd.zeros(num_hiddens, ctx=ctx)
    # 候选细胞参数。
    W_xc = nd.random_normal(scale=0.01, shape=(input_dim, num_hiddens),
                           ctx=ctx)
    W_hc = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                           ctx=ctx)
    b_c = nd.zeros(num_hiddens, ctx=ctx)
    # 输出层参数。
    W_hy = nd.random_normal(scale=0.01, shape=(num_hiddens, output_dim),
                           ctx=ctx)
    b_y = nd.zeros(output_dim, ctx=ctx)

    params = [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc,
              b_c, W_hy, b_y]
    for param in params:
        param.attach_grad()
    return params
```

### 6.6.3 定义模型

下面根据长短期记忆的计算表达式定义模型。

```
In [3]: def lstm_rnn(inputs, state_h, state_c, *params):
    [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c,
     W_ty, b_y] = params
    H = state_h
    C = state_c
    outputs = []
    for X in inputs:
        I = nd.sigmoid(nd.dot(X, W_xi) + nd.dot(H, W_hi) + b_i)
        F = nd.sigmoid(nd.dot(X, W_xf) + nd.dot(H, W_hf) + b_f)
        O = nd.sigmoid(nd.dot(X, W_xo) + nd.dot(H, W_ho) + b_o)
        C_tilda = nd.tanh(nd.dot(X, W_xc) + nd.dot(H, W_hc) + b_c)
        C = F * C + I * C_tilda
        H = O * C.tanh()
        Y = nd.dot(H, W_ty) + b_y
        outputs.append(Y)
    return (outputs, H, C)
```

### 训练模型并创作歌词

设置好超参数后，我们将训练模型并根据前缀“分开”和“不分开”分别创作长度为 100 个字符的一段歌词。我们每过 30 个迭代周期便根据当前训练的模型创作一段歌词。训练模型时采用了相邻采样。

```
In [4]: get_inputs = gb.to_onehot
num_epochs = 150
num_steps = 35
batch_size = 32
lr = 0.25
clipping_theta = 5
prefixes = ['分开', '不分开']
pred_period = 30
pred_len = 100

gb.train_and_predict_rnn(lstm_rnn, False, num_epochs, num_steps, num_hiddens,
                        lr, clipping_theta, batch_size, vocab_size,
                        pred_period, pred_len, prefixes, get_params,
                        get_inputs, ctx, corpus_indices, idx_to_char,
                        char_to_idx, is_lstm=True)
```

epoch 30, perplexity 188.568253

- 分开 我不要我 我不要  
→ 我不要 我不要
- 不分开 我不要我 我不要  
→ 我不要 我不要

epoch 60, perplexity 33.237305

- 分开 我想能这样 我的世界 有天了空 木滚 一直忿 一直有 有炭 一直 有一起 有一止 有炭 一直用  
→ 木炭 一直忿 木炭 一直 停炭 一直 停炭 一直 停炭 一直 停炭 一直 停炭 一直 停炭 一直
- 不分开 我不能再想 我不能再不样 我不能再不知 你不是我 爱你有人 我想就这样牵着你的手不放开 爱爱不能  
→ 不不再再单单我 不想再这不想 让我是我太爱你 爱因为我太爱你 爱因为我太爱你 爱因为我太爱你 爱因为我

epoch 90, perplexity 6.666142

- 分开 你在那不是我开 难难你 你怎么我不想活 你去我会汉你 是因为我太爱你 爱因我的太笑 你说了起不到  
→ 你永远远不到 你永远远不了 我永远远不到 你永远赢不了 我永远远不到 你永远远不了 我永远远不到 你永  
- 不分开 我的世界 你已了这 没有你在你的手 我想揍你的微笑 想想就陪不到 说永着我太腔开  
→ 是因为我太多多 我说 难不我 你你久了 是你的手太重 像知的侧笑倒倒 我想想慢放你 想想我  
→ 你不了我的微笑 我想揍你

epoch 120, perplexity 2.599554

- 分开 你是我说爱你是一场悲剧 我想我这辈子注定一个人演戏 最后再一个人慢慢的回忆 没有了过去  
→ 我将往事抽离 如果我遇见你是一场悲剧 我可以让生命就这样毫无意义 或许在最后能到到你一句 轻轻的叹息  
→ 后悔着对
- 不分开 我该不陪你的裁判 不想再你心碎在一起 还要在宇宙里 我每天每天每天在想想想着你 这样的甜蜜  
→ 让我开始乡相信命运 感谢心心引力 让我碰到你 漂亮的让我面红的可爱女人 温柔的让我心疼的可爱女人  
→ 透坏的让

epoch 150, perplexity 1.595566

- 分开 我是儿着你走强我 不成你看了背下 你就我要 你不要要去打 我后不觉 看远心心 停慢 一直筐重留  
→ 它在几中的牛肉 我说店小二 三两银够不够 景色入秋 漫天黄沙凉过 塞北的客栈人多 牧草有没有 我马儿有些
- 不分开留一场 想想要到飞走试事 我要和到斯坦堡 想像像童担故事 有教堂有城堡 每天忙碌地的寻找  
→ 到底什么我想要 却发现迷了路怎么找也找不着 心血来潮起个大早 怎么我也睡不着 昨晚梦里你来找 我才  
→ 原来我

#### 6.6.4 小结

- 长短期记忆的隐藏层输出包括隐藏状态和记忆细胞。只有隐藏状态会传递进输出层。
- 长短期记忆的输入门、遗忘门和输出门可以控制信息的流动。
- 长短期记忆的可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时序数据中间隔较

大的依赖关系。

### 6.6.5 练习

- 调调超参数，观察并分析对运行时间、困惑度以及创作歌词的结果造成的影响。
- 在相同条件下，比较长短期记忆、门控循环单元和不带门控的循环神经网络的运行时间。
- 既然候选记忆细胞已通过使用  $\tanh$  函数确保值域在-1 到 1 之间，为什么隐藏状态还需再次使用  $\tanh$  函数来确保输出值域在-1 到 1 之间？

### 6.6.6 扫码直达讨论区



### 6.6.7 参考文献

[1] Hochreiter, Sepp, and Jürgen Schmidhuber. “Long short-term memory.” Neural computation 9.8 (1997): 1735-1780.

## 6.7 深度循环神经网络

本章到目前为止介绍的循环神经网络只有一个单向的隐藏层：隐藏状态里的信息沿着时间步从早到晚依次传递。在实际中，我们有时会用到其他架构的循环神经网络。本节和下一节将分别介绍多隐藏层和双向架构。它们分别称作深度循环神经网络和双向循环神经网络。

给定时间步  $t$  的小批量输入  $\mathbf{X}_t \in \mathbb{R}^{n \times x}$  (样本数为  $n$ , 输入个数为  $x$ )。在深度循环神经网络中，设该时间步第  $l$  隐藏层的隐藏状态为  $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$  (隐藏单元个数为  $h$ )，输出层变量为  $\mathbf{O}_t \in \mathbb{R}^{n \times y}$  (输出个数为  $y$ )，隐藏层的激活函数为  $\phi$ 。第一隐藏层的隐藏状态和之前的计算一样：

$$\mathbf{H}_t^{(1)} = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(1)} + \mathbf{H}_{t-1}^{(1)} \mathbf{W}_{hh}^{(1)} + \mathbf{b}_h^{(1)}),$$

其中权重  $\mathbf{W}_{xh}^{(1)} \in \mathbb{R}^{x \times h}$ ,  $\mathbf{W}_{hh}^{(1)} \in \mathbb{R}^{h \times h}$  和偏差  $\mathbf{b}_h^{(1)} \in \mathbb{R}^{1 \times h}$  分别为第一隐藏层的模型参数。

假设隐藏层个数为  $L$ , 当  $1 < l \leq L$  时, 第  $l$  隐藏层的隐藏状态的表达式为

$$\mathbf{H}_t^{(l)} = \phi(\mathbf{H}_t^{(l-1)} \mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(1)} \mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}),$$

其中权重  $\mathbf{W}_{xh}^{(l)} \in \mathbb{R}^{h \times h}$ ,  $\mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h \times h}$  和偏差  $\mathbf{b}_h^{(l)} \in \mathbb{R}^{1 \times h}$  分别为第  $l$  隐藏层的模型参数。

最终, 输出层的输出只需基于第  $L$  隐藏层的隐藏状态:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{hy} + \mathbf{b}_y,$$

其中权重  $\mathbf{W}_{hy} \in \mathbb{R}^{h \times y}$  和偏差  $\mathbf{b}_y \in \mathbb{R}^{1 \times y}$  为输出层的模型参数。

深度循环神经网络的架构如图 6.3 所示。隐藏状态的信息不断传递至当前层的下一时间步和当前时间步的下一层。

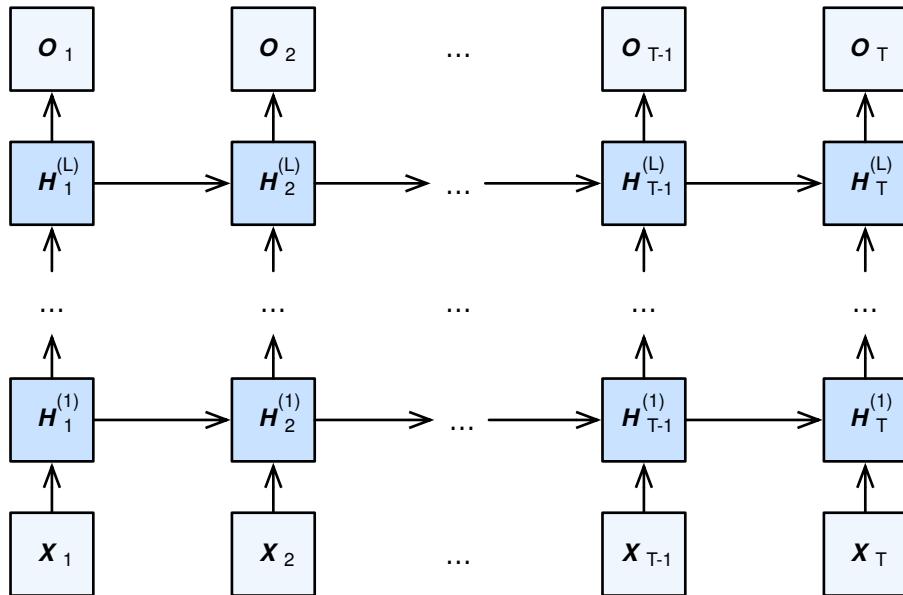


图 6.3: 深度循环神经网络的架构。

我们将在本章最后一节用 Gluon 实验深度循环神经网络。

### 6.7.1 小结

- 在深度循环神经网络中，隐藏状态的信息不断传递至当前层的下一时间步和当前时间步的下一层。

### 6.7.2 练习

- 将“循环神经网络——从零开始”一节中的模型改为含有 2 个隐藏层的循环神经网络。观察并分析实验现象。

### 6.7.3 扫码直达讨论区



## 6.8 双向循环神经网络

下面我们介绍双向循环神经网络的架构。

给定时间步  $t$  的小批量输入  $\mathbf{X}_t \in \mathbb{R}^{n \times x}$  (样本数为  $n$ , 输入个数为  $x$ ) 和隐藏层激活函数为  $\phi$ 。在双向架构中, 设该时间步正向隐藏状态为  $\vec{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$  (正向隐藏单元个数为  $h$ ), 反向隐藏状态为  $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$  (反向隐藏单元个数为  $h$ )。我们可以分别计算正向和反向隐藏状态:

$$\begin{aligned}\vec{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \vec{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}), \\ \overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}),\end{aligned}$$

其中权重  $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{x \times h}$ ,  $\mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}$ ,  $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{x \times h}$ ,  $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$  和偏差  $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$ ,  $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$  均为模型参数。

双向循环神经网络在时间步  $t$  的隐藏状态  $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$  即连结两个方向的隐藏状态  $\vec{\mathbf{H}}_t$  和  $\overleftarrow{\mathbf{H}}_t$  的结果。输出层只需基于连结后的隐藏状态计算输出  $\mathbf{O}_t \in \mathbb{R}^{n \times y}$  (输出个数为  $y$ ):

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hy} + \mathbf{b}_y,$$

其中权重  $\mathbf{W}_{hy} \in \mathbb{R}^{2h \times y}$  和偏差  $\mathbf{b}_y \in \mathbb{R}^{1 \times y}$  为输出层的模型参数。

双向循环神经网络架构如图 6.4 所示。和前面介绍的单向循环神经网络不同，给定一段时间序列，双向循环神经网络在每个时间步的隐藏状态同时取决于该时间步之前和之后的子序列（包括当前时间步的输入），并编码了整个序列的信息。

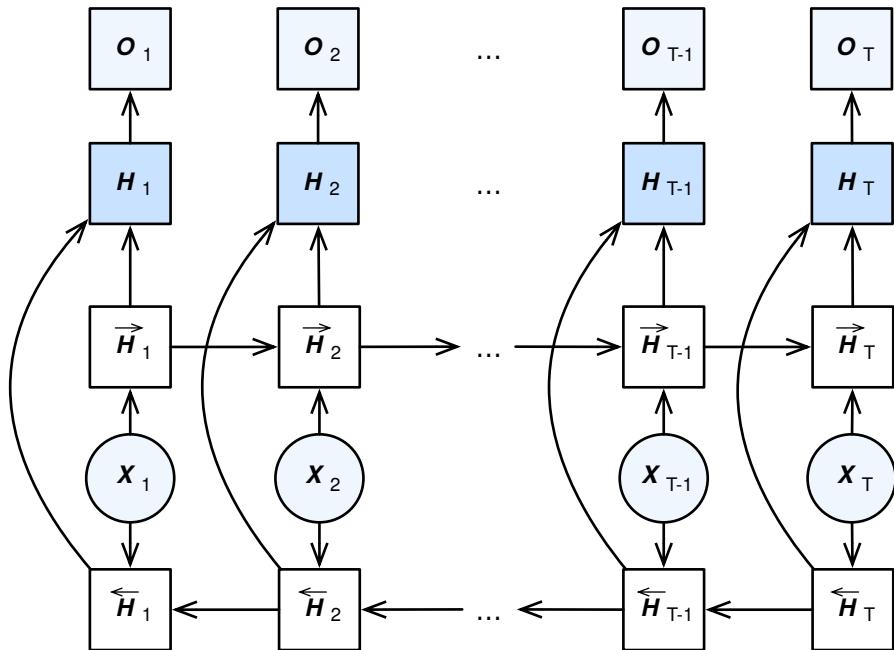


图 6.4: 双向循环神经网络架构。

我们将在“自然语言处理”篇章中应用并实验双向循环神经网络。

### 6.8.1 小结

- 双向循环神经网络在每个时间步的隐藏状态同时取决于该时间步之前和之后的子序列（包括当前时间步的输入）。

### 6.8.2 练习

- 参考图 6.3 和图 6.4，设计含多个隐藏层的双向循环神经网络。

### 6.8.3 扫码直达讨论区



## 6.9 循环神经网络——使用 Gluon

本节介绍如何使用 Gluon 训练循环神经网络。

### 6.9.1 Penn Tree Bank 数据集

我们以英文单词为单元来训练基于循环神经网络的语言模型。Penn Tree Bank (PTB) 是一个标准的文本序列数据集 [1]。它包括训练集、验证集和测试集。

首先导入实验所需的包或模块，并抽取数据集。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        import math
        import mxnet as mx
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import loss as gloss, nn, rnn, utils as gutils
        import numpy as np
        import time
        import zipfile

        with zipfile.ZipFile('../data/ptb.zip', 'r') as zin:
            zin.extractall('../data/')
```

### 6.9.2 建立词语索引

下面定义了 `Dictionary` 类来映射词语和整数索引。

```
In [2]: class Dictionary(object):
    def __init__(self):
        self.word_to_idx = {}
        self.idx_to_word = []

    def add_word(self, word):
        if word not in self.word_to_idx:
            self.idx_to_word.append(word)
            self.word_to_idx[word] = len(self.idx_to_word) - 1
        return self.word_to_idx[word]

    def __len__(self):
        return len(self.idx_to_word)
```

以下的 `Corpus` 类按照读取的文本数据集建立映射词语和索引的词典，并将文本转换成词语索引的序列。这样，每个文本数据集就变成了 `NDArray` 格式的整数序列。

```
In [3]: class Corpus(object):
    def __init__(self, path):
        self.dictionary = Dictionary()
        self.train = self.tokenize(path + 'train.txt')
        self.valid = self.tokenize(path + 'valid.txt')
        self.test = self.tokenize(path + 'test.txt')

    def tokenize(self, path):
        # 将词语添加至词典。
        with open(path, 'r') as f:
            num_words = 0
            for line in f:
                words = line.split() + ['<eos>']
                num_words += len(words)
                for word in words:
                    self.dictionary.add_word(word)
        # 将文本转换成词语索引的序列（NDArray 格式）。
        with open(path, 'r') as f:
            indices = np.zeros((num_words,), dtype='int32')
            idx = 0
            for line in f:
                words = line.split() + ['<eos>']
                for word in words:
                    indices[idx] = self.dictionary.word_to_idx[word]
                    idx += 1
        return nd.array(indices, dtype='int32')
```

看一下词典的大小。

```
In [4]: data = '../data/ptb/ptb.'
corpus = Corpus(data)
vocab_size = len(corpus.dictionary)
vocab_size
```

```
Out[4]: 10000
```

### 6.9.3 定义循环神经网络模型库

我们可以定义一个循环神经网络模型库。这样我们就可以使用以 ReLU 或 tanh 函数为激活函数的循环神经网络，以及长短期记忆和门控循环单元。和本章中其他实验不同，这里使用了 Embedding 实例将每个词索引转换成一个长度为 `embed_size` 的词向量。这些词向量实际上也是模型参数。在随机初始化后，它们会在模型训练结束时被学到。此外，我们使用了丢弃法来应对过拟合。

```
In [5]: class RNNModel(nn.Block):
    def __init__(self, mode, vocab_size, embed_size, num_hiddens,
                 num_layers, drop_prob=0.5, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        with self.name_scope():
            self.dropout = nn.Dropout(drop_prob)
            # 将词索引转换成词向量。这些词向量也是模型参数。
            self.embedding = nn.Embedding(
                vocab_size, embed_size, weight_initializer=init.Uniform(0.1))
            if mode == 'rnn_relu':
                self.rnn = rnn.RNN(num_hiddens, num_layers, activation='relu',
                                   dropout=drop_prob, input_size=embed_size)
            elif mode == 'rnn_tanh':
                self.rnn = rnn.RNN(num_hiddens, num_layers, activation='tanh',
                                   dropout=drop_prob, input_size=embed_size)
            elif mode == 'lstm':
                self.rnn = rnn.LSTM(num_hiddens, num_layers,
                                   dropout=drop_prob, input_size=embed_size)
            elif mode == 'gru':
                self.rnn = rnn.GRU(num_hiddens, num_layers, dropout=drop_prob,
                                   input_size=embed_size)
            else:
                raise ValueError("Invalid mode %s. Options are rnn_relu, "
                                "rnn_tanh, lstm, and gru" % mode)
            self.dense = nn.Dense(vocab_size, in_units=num_hiddens)
            self.num_hiddens = num_hiddens
```

```

def forward(self, inputs, state):
    embedding = self.dropout(self.embedding(inputs))
    output, state = self.rnn(embedding, state)
    output = self.dropout(output)
    output = self.dense(output.reshape((-1, self.num_hiddens)))
    return output, state

def begin_state(self, *args, **kwargs):
    return self.rnn.begin_state(*args, **kwargs)

```

## 6.9.4 设置超参数

我们接着设置超参数。这里选择使用以 ReLU 为激活函数的循环神经网络。它包含 2 个隐藏层。为了得到更好的实验结果，这些超参数还需要重新设置。

```

In [6]: model_name = 'rnn_relu'
        embed_size = 100
        num_hiddens = 100
        num_layers = 2
        lr = 0.5
        clipping_theta = 0.2
        num_epochs = 2
        batch_size = 32
        num_steps = 5
        drop_prob = 0.2
        eval_period = 1000

        ctx = gb.try_gpu()
        model = RNNModel(model_name, vocab_size, embed_size, num_hiddens, num_layers,
                          drop_prob)
        model.initialize(init.Xavier(), ctx=ctx)
        trainer = gluon.Trainer(model.collect_params(), 'sgd',
                               {'learning_rate': lr, 'momentum': 0, 'wd': 0})
        loss = gloss.SoftmaxCrossEntropyLoss()

```

## 6.9.5 相邻采样

我们将在实验中使用相邻采样。

```

In [7]: def batchify(data, batch_size):
            num_batches = data.shape[0] // batch_size

```

```

data = data[:num_batches*batch_size]
data = data.reshape((batch_size, num_batches)).T
return data

train_data = batchify(corpus.train, batch_size).as_in_context(ctx)
val_data = batchify(corpus.valid, batch_size).as_in_context(ctx)
test_data = batchify(corpus.test, batch_size).as_in_context(ctx)

def get_batch(source, i):
    seq_len = min(num_steps, source.shape[0]-1-i)
    X = source[i : i+seq_len]
    Y = source[i+1 : i+1+seq_len]
    return X, Y.reshape((-1,))

```

“循环神经网络——从零开始”一节里已经解释了，相邻采样应在每次读取小批量前将隐藏状态从计算图分离出来。

```
In [8]: def detach(state):
    if isinstance(state, (tuple, list)):
        state = [i.detach() for i in state]
    else:
        state = state.detach()
    return state
```

## 6.9.6 训练和评价模型

以下定义了模型评价函数。

```
In [9]: def eval_rnn(data_source):
    l_sum = nd.array([0], ctx=ctx)
    n = 0
    state = model.begin_state(func=nd.zeros, batch_size=batch_size, ctx=ctx)
    for i in range(0, data_source.shape[0] - 1, num_steps):
        X, y = get_batch(data_source, i)
        output, state = model(X, state)
        l = loss(output, y)
        l_sum += l.sum()
        n += l.size
    return l_sum / n
```

下面的 `train_rnn` 函数将训练模型并在每个迭代周期结束时评价模型在验证集上的表现。我们可以参考验证集上的结果调节超参数。

```
In [10]: def train_rnn():
    for epoch in range(1, num_epochs + 1):
        train_l_sum = nd.array([0], ctx=ctx)
        start_time = time.time()
        state = model.begin_state(func=nd.zeros, batch_size=batch_size,
                                   ctx=ctx)
        for batch_i, idx in enumerate(range(0, train_data.shape[0] - 1,
                                             num_steps)):
            X, y = get_batch(train_data, idx)
            # 从计算图分离隐藏状态变量（包括 LSTM 的记忆细胞）。
            state = detach(state)
            with autograd.record():
                output, state = model(X, state)
                # l 形状: (batch_size * num_steps,)。
                l = loss(output, y).sum() / (batch_size * num_steps)
            l.backward()
            grads = [p.grad(ctx) for p in model.collect_params().values()]
            # 梯度裁剪。需要注意的是，这里的梯度是整个批量的梯度。
            # 因此我们将 clipping_theta 乘以 num_steps 和 batch_size。
            gutils.clip_global_norm(
                grads, clipping_theta * num_steps * batch_size)
            trainer.step(1)
            train_l_sum += l
            if batch_i % eval_period == 0 and batch_i > 0:
                cur_l = train_l_sum / eval_period
                print('epoch %d, batch %d, train loss %.2f, perplexity %.2f'
                      % (epoch, batch_i, cur_l.asscalar(),
                         cur_l.exp().asscalar()))
                train_l_sum = nd.array([0], ctx=ctx)
            val_l = eval_rnn(val_data)
            print('epoch %d, time %.2fs, valid loss %.2f, perplexity %.2f'
                  % (epoch, time.time() - start_time, val_l.asscalar(),
                     val_l.exp().asscalar()))
```

训练完模型以后，我们就可以在测试集上评价模型了。

```
In [11]: train_rnn()
test_l = eval_rnn(test_data)
print('test loss %.2f, perplexity %.2f'
      % (test_l.asscalar(), test_l.exp().asscalar()))

epoch 1, batch 1000, train loss 7.21, perplexity 1356.29
epoch 1, batch 2000, train loss 6.43, perplexity 618.23
epoch 1, batch 3000, train loss 6.22, perplexity 502.70
epoch 1, batch 4000, train loss 6.11, perplexity 450.60
```

```
epoch 1, batch 5000, train loss 6.03, perplexity 413.85
epoch 1, time 42.17s, valid loss 5.85, perplexity 348.87
epoch 2, batch 1000, train loss 5.94, perplexity 379.93
epoch 2, batch 2000, train loss 5.88, perplexity 358.65
epoch 2, batch 3000, train loss 5.80, perplexity 329.77
epoch 2, batch 4000, train loss 5.76, perplexity 316.76
epoch 2, batch 5000, train loss 5.71, perplexity 301.73
epoch 2, time 41.84s, valid loss 5.61, perplexity 273.55
test loss 5.57, perplexity 262.83
```

### 6.9.7 小结

- 我们可以使用 Gluon 训练循环神经网络。它更简洁，例如无需我们手动实现含有多个隐藏层的复杂模型。
- 在训练语言模型时，我们可以将词索引变换成词向量，并将这些词向量视为模型参数。

### 6.9.8 练习

- 回忆“模型参数的访问、初始化和共享”一节中有关共享模型参数的描述。将本节中 RNNModel 类里的 `self.dense` 的定义改为 `nn.Dense(vocab_size, in_units = num_hiddens, params=self.embedding.params)` 并运行本节实验。这里为什么可以共享词向量参数？有哪些好处？
- 调调超参数，观察并分析对运行时间以及训练集、验证集和测试集上困惑度的影响。

### 6.9.9 扫码直达讨论区



### 6.9.10 参考文献

[1] Penn Tree Bank. <https://catalog.ldc.upenn.edu/ldc99t42>

---

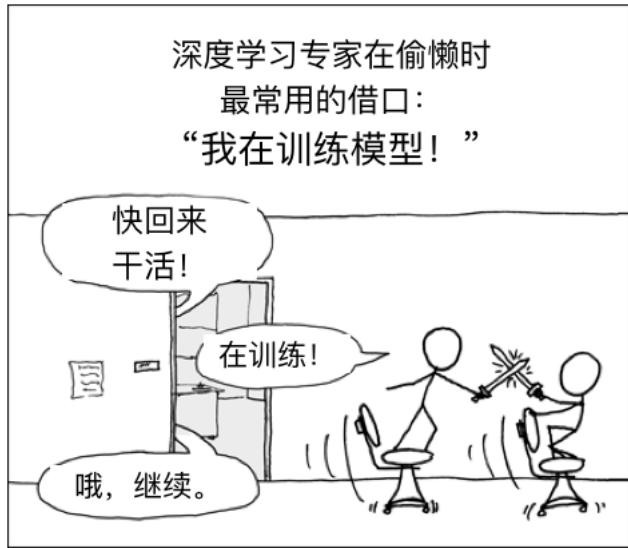
## 优化算法

---

如果你一直按照本书的顺序读到这里，很可能已经使用了优化算法来训练深度学习模型。具体来说，在训练模型时，我们会使用优化算法不断迭代模型参数以最小化模型的损失函数。当迭代终止时，模型的训练随之终止。此时的模型参数就是模型通过训练所学习到的参数。

优化算法对于深度学习十分重要。一方面，如图 7.1 所表现的那样，训练一个复杂的深度学习模型可能需要数小时、数日、甚至数周时间。而优化算法的表现直接影响模型训练效率。另一方面，理解各种优化算法的原理以及其中各参数的意义将有助于我们更有针对性地调参，从而使深度学习模型表现地更好。

本章将详细介绍深度学习中的常用优化算法。



## 7.1 优化算法概述

本节将讨论优化与深度学习的关系以及优化在深度学习中的挑战。

### 7.1.1 优化与深度学习

在一个深度学习问题中，通常我们会预先定义一个损失函数。有了损失函数以后，我们就可以使用优化算法试图使其最小化。在优化中，这样的损失函数通常被称作优化问题的目标函数 (objective function)。依据惯例，优化算法通常只考虑最小化目标函数。其实，任何最大化问题都可以很容易地转化为最小化问题：我们只需把目标函数前面的正号或负号取相反。

虽然优化为深度学习提供了最小化损失函数的方法，但本质上，这两者之间的目标是有区别的。在“欠拟合、过拟合和模型选择”一节中，我们区分了训练误差和泛化误差。由于优化算法的目标函数通常是一个基于训练数据集的损失函数，优化的目标在于降低训练误差。而深度学习的目标在于降低泛化误差。为了降低泛化误差，除了使用优化算法降低训练误差以外，我们还需要注意应对过拟合。

本章中，我们只关注优化算法在最小化目标函数上的表现，而不关注模型的泛化误差。

## 7.1.2 优化在深度学习中的挑战

绝大多数深度学习中的目标函数都很复杂。因此，很多优化问题并不存在解析解，而需要使用基于数值方法的优化算法找到近似解。这类优化算法一般通过不断迭代更新解的数值来找到近似解。我们讨论的优化算法都是这类基于数值方法的算法。

优化在深度学习中有很多挑战。以下描述了其中的两个挑战：局部最小值和鞍点。为了更好地描述问题，我们先导入本节中实验需要的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mpl_toolkits import mplot3d
        import numpy as np
```

### 局部最小值

对于目标函数  $f(x)$ ，如果  $f(x)$  在  $x$  上的值比在  $x$  邻近的其他点的值更小，那么  $f(x)$  可能是一个局部最小值 (local minimum)。如果  $f(x)$  在  $x$  上的值是目标函数在整个定义域上的最小值，那么  $f(x)$  是全局最小值 (global minimum)。

举个例子，给定函数

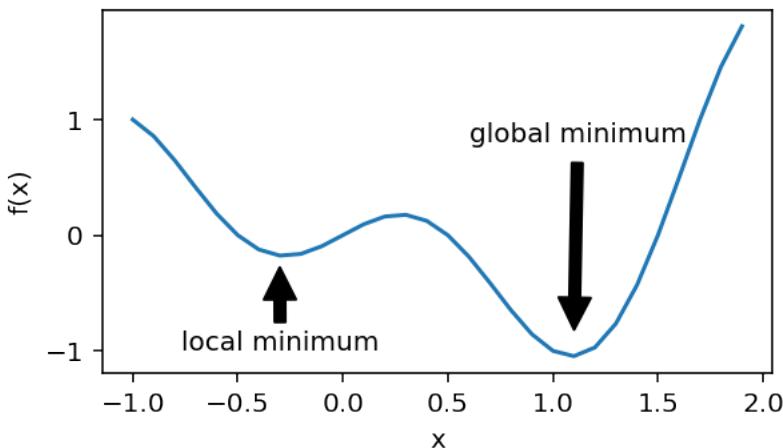
$$f(x) = x \cdot \cos(\pi x), \quad -1.0 \leq x \leq 2.0,$$

我们可以大致找出该函数的局部最小值和全局最小值的位置。需要注意的是，图中箭头所指示的只是大致位置。

```
In [2]: def f(x):
        return x * np.cos(np.pi * x)

gb.pyplot.rcParams['figure.figsize'] = (4.5, 2.5)
x = np.arange(-1.0, 2.0, 0.1)
fig = gb.pyplot.figure()
subplt = fig.add_subplot(111)
subplt.annotate('local minimum', xy=(-0.3, -0.25), xytext=(-0.77, -1.0),
                arrowprops=dict(facecolor='black', shrink=0.05))
subplt.annotate('global minimum', xy=(1.1, -0.9), xytext=(0.6, 0.8),
                arrowprops=dict(facecolor='black', shrink=0.05))
gb.pyplot.plot(x, f(x))
gb.pyplot.xlabel('x')
```

```
gb=plt.ylabel('f(x)')  
gb=plt.show()
```



深度学习模型的目标函数可能有若干局部最优值。当一个优化问题的数值解在局部最优解附近时，由于目标函数有关解的梯度接近或变成零，最终迭代求得的数值解可能只令目标函数局部最小化而非全局最小化。

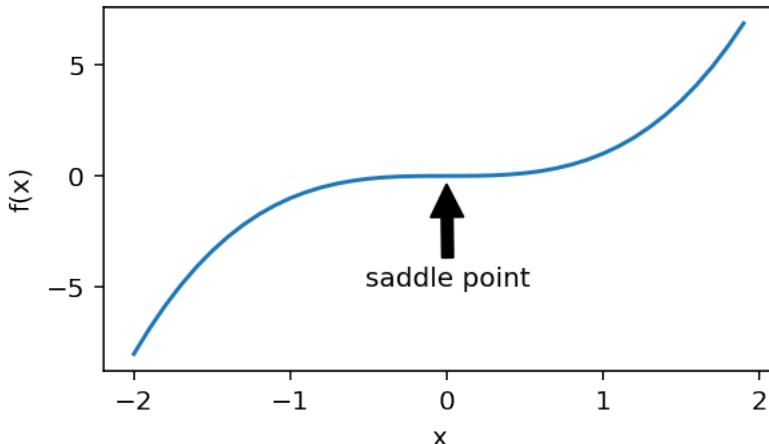
### 鞍点

刚刚我们提到，梯度接近或变成零可能是由于当前解在局部最优解附近所造成的。事实上，另一种可能性是当前解在鞍点（saddle point）附近。举个例子，给定函数

$$f(x) = x^3,$$

我们可以找出该函数的鞍点位置。

```
In [3]: x = np.arange(-2.0, 2.0, 0.1)  
fig = gb.pyplot.figure()  
subplt = fig.add_subplot(111)  
subplt.annotate('saddle point', xy=(0, -0.2), xytext=(-0.52, -5.0),  
                arrowprops=dict(facecolor='black', shrink=0.05))  
gb.pyplot.plot(x, x**3)  
gb.pyplot.xlabel('x')  
gb.pyplot.ylabel('f(x)')  
gb.pyplot.show()
```

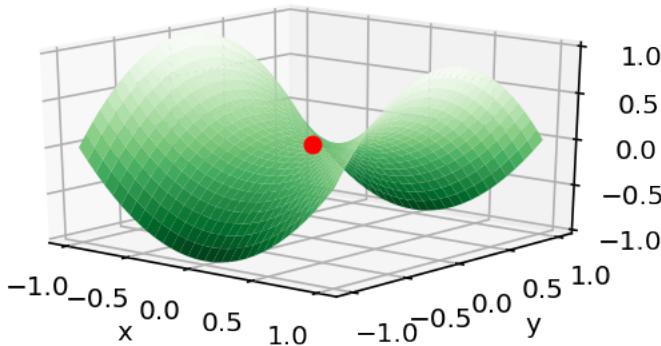


再举个定义在二维空间的函数的例子，例如

$$f(x, y) = x^2 - y^2.$$

我们可以找出该函数的鞍点位置。也许你已经发现了，该函数看起来像一个马鞍，而鞍点恰好是马鞍上可坐区域的中心。

```
In [4]: fig = gb.pyplot.figure()
ax = fig.add_subplot(111, projection='3d')
x, y = np.mgrid[-1:1:31j, -1:1:31j]
z = x**2 - y**2
ax.plot_surface(x, y, z, **{'rstride': 1, 'cstride': 1, 'cmap': "Greens_r"})
ax.plot([0], [0], [0], 'ro')
ax.view_init(azim=-50, elev=20)
gb.pyplot.xticks([-1, -0.5, 0, 0.5, 1])
gb.pyplot.yticks([-1, -0.5, 0, 0.5, 1])
ax.set_zticks([-1, -0.5, 0, 0.5, 1])
gb.pyplot.xlabel('x')
gb.pyplot.ylabel('y')
gb.pyplot.show()
```



在上图的鞍点位置，目标函数在  $x$  轴方向上是局部最小值，而在  $y$  轴方向上是局部最大值。

假设一个函数的输入为  $k$  维向量，输出为标量，那么它的黑塞矩阵（Hessian matrix）有  $k$  个特征值。需要注意的是，该函数在梯度为零的位置上可能是局部最小值、局部最大值或者鞍点：

- 当函数的黑塞矩阵在梯度为零的位置上的特征值全为正时，该函数得到局部最小值。
- 当函数的黑塞矩阵在梯度为零的位置上的特征值全为负时，该函数得到局部最大值。
- 当函数的黑塞矩阵在梯度为零的位置上的特征值有正有负时，该函数得到鞍点。

随机矩阵理论告诉我们，对于一个大的高斯随机矩阵来说，任一特征值是正或者是负的概率都是 0.5 [1]。那么，以上第一种情况的概率为:math:0.5^k。由于深度学习模型参数通常都是高维的 ( $k$  很大)，目标函数的鞍点通常比局部最小值更常见。

深度学习中，虽然找到目标函数的全局最优解很难，但这并非必要。我们将在接下来的章节中逐一介绍深度学习中常用的优化算法，它们在很多实际问题中都训练出了十分有效的深度学习模型。

### 7.1.3 小结

- 由于优化算法的目标函数通常是一个基于训练数据集的损失函数，优化的目标在于降低训练误差。
- 由于深度学习模型参数通常都是高维的，目标函数的鞍点通常比局部最小值更常见。

### 7.1.4 练习

- 你还能想到哪些深度学习中的优化问题的挑战？

### 7.1.5 扫码直达讨论区



### 7.1.6 参考文献

[1] Wigner, Eugene P. “On the distribution of the roots of certain symmetric matrices.” *Annals of Mathematics* (1958): 325-327.

## 7.2 梯度下降和随机梯度下降——从零开始

本节中，我们将介绍梯度下降（gradient descent）和随机梯度下降（stochastic gradient descent）算法。由于梯度下降是优化算法的核心部分，理解梯度的涵义十分重要。为了帮助大家深刻理解梯度，我们将从数学角度阐释梯度下降的意义。本节中，我们假设目标函数连续可导。

### 7.2.1 一维梯度下降

我们先以简单的一维梯度下降为例，解释梯度下降算法可以降低目标函数值的原因。一维梯度是一个标量，也称导数。

假设函数  $f : \mathbb{R} \rightarrow \mathbb{R}$  的输入和输出都是标量。给定足够小的数  $\epsilon$ ，根据泰勒展开公式（参见“数学基础”一节），我们得到以下的近似

$$f(x + \epsilon) \approx f(x) + f'(x)\epsilon.$$

假设  $\eta$  是一个常数，将  $\epsilon$  替换为  $-\eta f'(x)$  后，我们有

$$f(x - \eta f'(x)) \approx f(x) - \eta f'(x)^2.$$

如果  $\eta$  是一个很小的正数，那么

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

也就是说，如果目标函数  $f(x)$  当前的导数  $f'(x) \neq 0$ ，按照

$$x \leftarrow x - \eta f'(x).$$

迭代自变量  $x$  可能会降低  $f(x)$  的值。由于导数  $f'(x)$  是梯度  $\nabla_x f$  在一维空间的特殊情况，上述迭代自变量  $x$  的方法也即一维空间的梯度下降。一维空间的梯度下降图 7.2（左）所示，自变量  $x$  沿着梯度方向迭代。

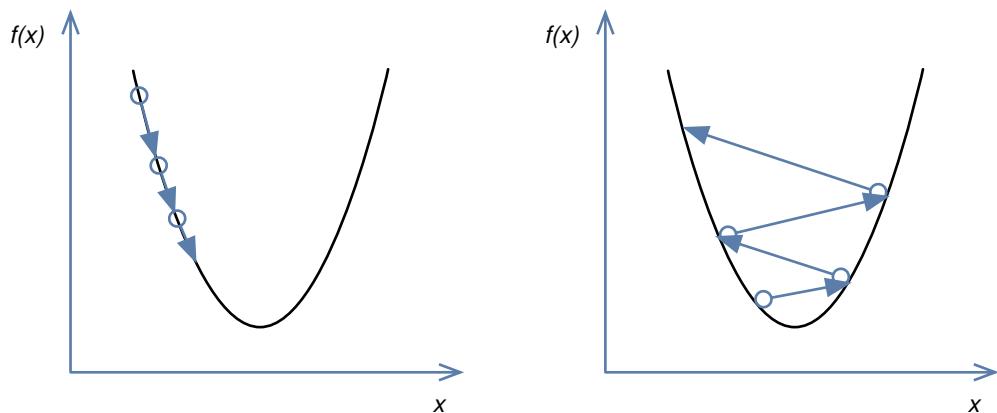


图 7.1: 梯度下降中，目标函数  $f(x)$  的自变量  $x$ （圆圈的横坐标）沿着梯度方向迭代

## 7.2.2 学习率

上述梯度下降算法中的  $\eta$  叫做学习率。这是一个超参数，需要人工设定。学习率  $\eta$  要取正数。

需要注意的是，学习率过大可能会造成自变量  $x$  越过（overshoot）目标函数  $f(x)$  的最优解，甚至发散。见图 7.2（右）。

然而，如果学习率过小，目标函数中自变量的收敛速度会过慢。实际中，一个合适的学习率通常需要通过多次实验找到的。

## 7.2.3 多维梯度下降

现在我们考虑一个更广义的情况：目标函数的输入为向量，输出为标量。

假设目标函数  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  的输入是一个  $d$  维向量  $\mathbf{x} = [x_1, x_2, \dots, x_d]^\top$ 。目标函数  $f(\mathbf{x})$  有关  $\mathbf{x}$  的梯度是一个由  $d$  个偏导数组成的向量：

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

为表示简洁，我们用  $\nabla f(\mathbf{x})$  代替  $\nabla_{\mathbf{x}} f(\mathbf{x})$ 。梯度中每个偏导数元素  $\partial f(\mathbf{x})/\partial x_i$  代表着  $f$  在  $\mathbf{x}$  有关输入  $x_i$  的变化率。为了测量  $f$  沿着单位向量  $\mathbf{u}$  方向上的变化率，在多元微积分中，我们定义  $f$  在  $\mathbf{x}$  上沿着  $\mathbf{u}$  方向的方向导数为

$$D_{\mathbf{u}} f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{u}) - f(\mathbf{x})}{h}.$$

依据方向导数性质 [1, 14.6 节定理三]，该方向导数可以改写为

$$D_{\mathbf{u}} f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{u}.$$

方向导数  $D_{\mathbf{u}} f(\mathbf{x})$  给出了  $f$  在  $\mathbf{x}$  上沿着所有可能方向的变化率。为了最小化  $f$ ，我们希望找到  $f$  能被降低最快的方向。因此，我们可以通过单位向量  $\mathbf{u}$  来最小化方向导数  $D_{\mathbf{u}} f(\mathbf{x})$ 。

由于  $D_{\mathbf{u}} f(\mathbf{x}) = \|\nabla f(\mathbf{x})\| \cdot \|\mathbf{u}\| \cdot \cos(\theta) = \|\nabla f(\mathbf{x})\| \cdot \cos(\theta)$ ，其中  $\theta$  为梯度  $\nabla f(\mathbf{x})$  和单位向量  $\mathbf{u}$  之间的夹角，当  $\theta = \pi$ ， $\cos(\theta)$  取得最小值-1。因此，当  $\mathbf{u}$  在梯度方向  $\nabla f(\mathbf{x})$  的相反方向时，方向导数  $D_{\mathbf{u}} f(\mathbf{x})$  被最小化。所以，我们可能通过下面的梯度下降算法来不断降低目标函数  $f$  的值：

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

相同地，其中  $\eta$ （取正数）称作学习率。

## 7.2.4 随机梯度下降

然而，当训练数据集很大时，梯度下降算法可能会难以使用。为了解释这个问题，考虑目标函数

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

其中  $f_i(\mathbf{x})$  是有关索引为  $i$  的训练数据样本的损失函数， $n$  是训练数据样本数。由于

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

梯度下降每次迭代的计算开销随着  $n$  线性增长。因此，当训练数据样本数很大时，梯度下降每次迭代的计算开销很高。这时我们可以使用随机梯度下降。给定学习率  $\eta$ （取正数），在每次迭代时，

随机梯度下降算法随机均匀采样  $i$  并计算  $\nabla f_i(\mathbf{x})$  来迭代  $\mathbf{x}$ :

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}).$$

事实上，随机梯度  $\nabla f_i(\mathbf{x})$  是对梯度  $\nabla f(\mathbf{x})$  的无偏估计:

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

### 7.2.5 小批量随机梯度下降

广义上，每一次迭代可以随机均匀采样一个由训练数据样本索引所组成的小批量（mini-batch） $\mathcal{B}$ 。一般来说，我们可以通过重复采样（sampling with replacement）或者不重复采样（sampling without replacement）得到同一个小批量中的各个样本。前者允许同一个小批量中出现重复的样本，后者则不允许如此。对于这两者间的任一种方式，我们可以使用

$$\nabla f_{\mathcal{B}}(\mathbf{x}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla f_i(\mathbf{x})$$

来迭代  $\mathbf{x}$ :

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_{\mathcal{B}}(\mathbf{x}).$$

在上式中， $|\mathcal{B}|$  代表样本批量大小， $\eta$ （取正数）称作学习率。同样，小批量随机梯度  $\nabla f_{\mathcal{B}}(\mathbf{x})$  也是对梯度  $\nabla f(\mathbf{x})$  的无偏估计:

$$\mathbb{E}_{\mathcal{B}} \nabla f_{\mathcal{B}}(\mathbf{x}) = \nabla f(\mathbf{x}).$$

这个算法叫做小批量随机梯度下降。该算法每次迭代的计算开销为  $\mathcal{O}(|\mathcal{B}|)$ 。当批量大小为 1 时，该算法即随机梯度下降；当批量大小等于训练数据样本数，该算法即梯度下降。和学习率一样，批量大小也是一个超参数。当批量较小时，虽然每次迭代的计算开销较小，但计算机并行处理批量中各个样本的能力往往只得到较少利用。因此，当训练数据集的样本较少时，我们可以使用梯度下降；当样本较多时，我们可以使用小批量梯度下降并依据计算资源选择合适的批量大小。

实际中，我们通常在每个迭代周期（epoch）开始前随机打乱数据集中样本的先后顺序，然后在同一个迭代周期中依次读取小批量的样本。理论上，这种方法能让符合某些特征的目标函数的优化收敛更快 [2]。我们在实验中也用这种方法随机采样小批量样本。

## 7.2.6 小批量随机梯度下降的实现

我们只需要实现小批量随机梯度下降。当批量大小等于训练集大小时，该算法即为梯度下降；批量大小为 1 即为随机梯度下降。

```
In [1]: def sgd(params, lr, batch_size):
    for param in params:
        param[:] = param - lr * param.grad / batch_size
```

## 7.2.7 实验

首先，导入本节中实验所需的包或模块。

```
In [2]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, nd
        import numpy as np
        import random
```

实验中，我们以之前介绍过的线性回归为例。我们直接调用 `gluonbook` 中的线性回归模型和平方损失函数。它们已在“[线性回归——从零开始](#)”一节中实现过了。

```
In [3]: net = gb.linreg
        loss = gb.squared_loss
```

设数据集的样本数为 1000，我们使用权重 `w` 为  $[2, -3.4]$ ，偏差 “`b`” 为 4.2 的线性回归模型来生成数据集。该模型的平方损失函数即所需优化的目标函数，模型参数即目标函数自变量。

```
In [4]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)

        # 初始化模型参数。
        def init_params():
            w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
            b = nd.zeros(shape=(1,))
            params = [w, b]
```

```
for param in params:  
    param.attach_grad()  
return params
```

下面我们描述一下优化函数 `optimize`。

由于随机梯度的方差在迭代过程中无法减小，（小批量）随机梯度下降的学习率通常会采用自我衰减的方式。如此一来，学习率和随机梯度乘积的方差会衰减。实验中，当迭代周期（epoch）大于 2 时，（小批量）随机梯度下降的学习率在每个迭代周期开始时自乘 0.1 作自我衰减。而梯度下降在迭代过程中一直使用目标函数的真实梯度，无需自我衰减学习率。

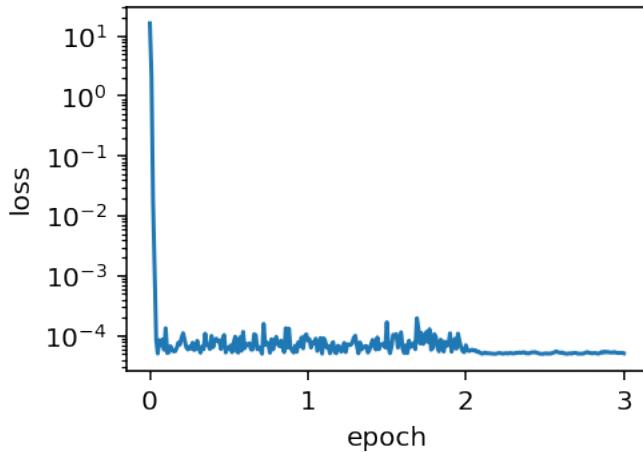
在迭代过程中，每当 `log_interval` 个样本被采样过后，模型当前的损失函数值（`loss`）被记录下并用于作图。例如，当 `batch_size` 和 `log_interval` 都为 10 时，每次迭代后的损失函数值都被用来作图。

```
In [5]: def optimize(batch_size, lr, num_epochs, log_interval, decay_epoch):  
    w, b = init_params()  
    ls = [loss(net(features, w, b), labels).mean().asnumpy()]  
    for epoch in range(1, num_epochs + 1):  
        # 学习率自我衰减。  
        if decay_epoch and epoch > decay_epoch:  
            lr *= 0.1  
        for batch_i, (X, y) in enumerate(  
            gb.data_iter(batch_size, num_examples, features, labels)):  
            with autograd.record():  
                l = loss(net(X, w, b), y)  
                l.backward()  
                sgd([w, b], lr, batch_size)  
            if batch_i * batch_size % log_interval == 0:  
                ls.append(loss(net(features, w, b), labels).mean().asnumpy())  
    print('w:', w, '\nb:', b, '\n')  
    es = np.linspace(0, num_epochs, len(ls), endpoint=True)  
    gb.semilogy(es, ls, 'epoch', 'loss')
```

当批量大小为 1 时，优化使用的是随机梯度下降。在当前学习率下，损失函数值在早期快速下降后略有波动。这是由于随机梯度的方差在迭代过程中无法减小。当迭代周期大于 2，学习率自我衰减后，损失函数值下降后较平稳。最终，优化所得的模型参数值 `w` 和 `b` 与它们的真实值 [2, -3.4] 和 4.2 较接近。

```
In [6]: optimize(batch_size=1, lr=0.2, num_epochs=3, decay_epoch=2, log_interval=10)  
  
w:  
[[ 1.99918675]  
 [-3.399575 ]]
```

```
<NDArray 2x1 @cpu(0)>
b:
[ 4.20092201]
<NDArray 1 @cpu(0)>
```

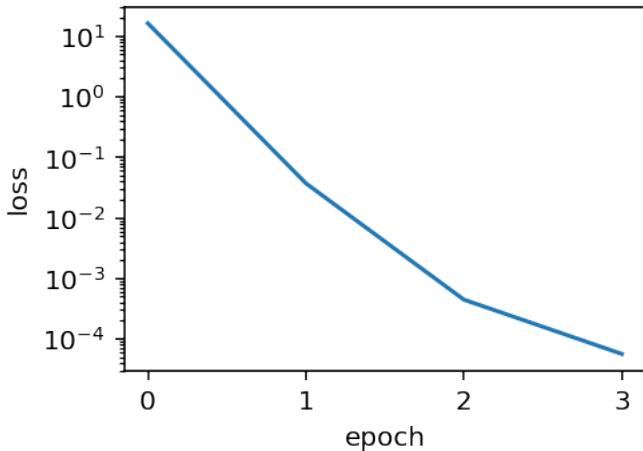


当批量大小为 1000 时，由于数据样本总数也是 1000，优化使用的是梯度下降。梯度下降无需自我衰减学习率 (decay\_epoch=None)。最终，优化所得的模型参数值与它们的真实值较接近。

需要注意的是，梯度下降的 1 个迭代周期对模型参数只迭代 1 次。而随机梯度下降的批量大小为 1，它在 1 个迭代周期对模型参数迭代了 1000 次。我们观察到，1 个迭代周期后，梯度下降所得的损失函数值比随机梯度下降所得的损失函数值略大。而在 3 个迭代周期后，这两个算法所得的损失函数值很接近。

```
In [7]: optimize(batch_size=1000, lr=0.999, num_epochs=3, decay_epoch=None,
                log_interval=1000)

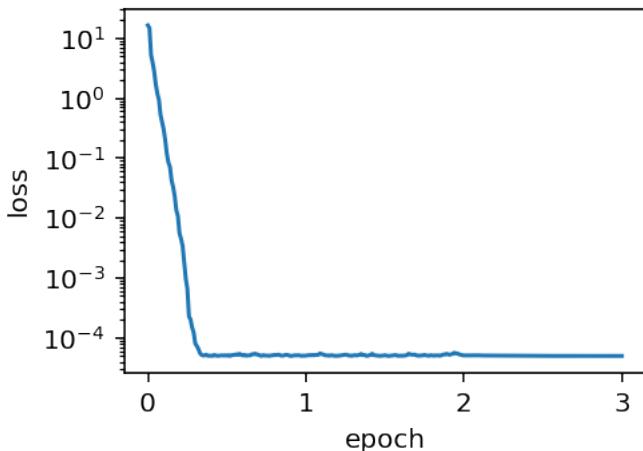
w:
[[ 2.00053716]
 [-3.40285206]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.19865417]
<NDArray 1 @cpu(0)>
```



当批量大小为 10 时，由于数据样本总数也是 1000，优化使用的是小批量随机梯度下降。最终，优化所得的模型参数值与它们的真实值较接近。

```
In [8]: optimize(batch_size=10, lr=0.2, num_epochs=3, decay_epoch=2, log_interval=10)

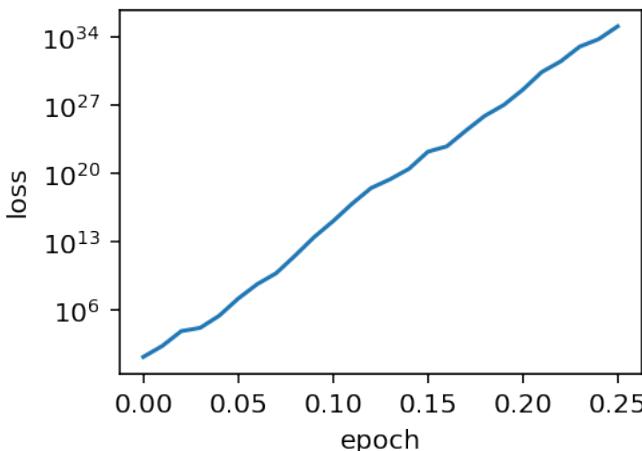
w:
[[ 1.9997673 ]
 [-3.40002227]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.20033741]
<NDArray 1 @cpu(0)>
```



同样是批量大小为 10，我们把学习率改大。这时损失函数值不断增大，直到出现“nan”（not a number，非数）。这是因为，过大的学习率造成了模型参数越过最优解并发散。最终学到的模型参数也是“nan”。

```
In [9]: optimize(batch_size=10, lr=5, num_epochs=3, decay_epoch=2, log_interval=10)

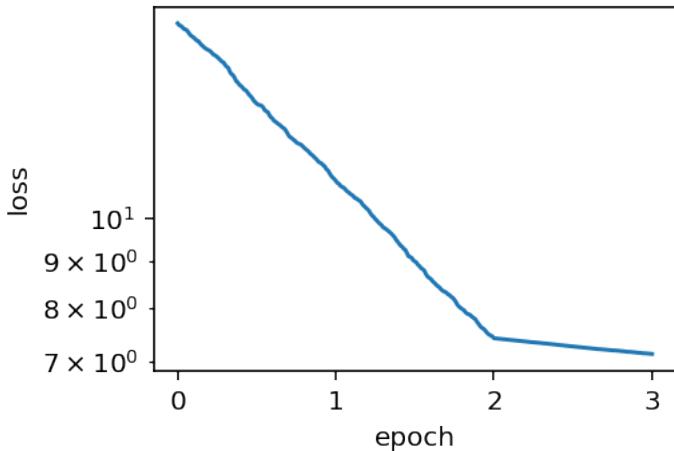
w:
[[ nan]
 [ nan]]
<NDArray 2x1 @cpu(0)>
b:
[ nan]
<NDArray 1 @cpu(0)>
```



同样是批量大小为 10，我们把学习率改小。这时我们观察到损失函数值下降较慢，直到 3 个迭代周期模型参数也没能接近它们的真实值。

```
In [10]: optimize(batch_size=10, lr=0.002, num_epochs=3, decay_epoch=2,
log_interval=10)

w:
[[ 0.69030839]
 [-1.20037329]]
<NDArray 2x1 @cpu(0)>
b:
[ 1.37690783]
<NDArray 1 @cpu(0)>
```



### 7.2.8 小结

- 当训练数据较大，梯度下降每次迭代计算开销较大，因而（小批量）随机梯度下降更受青睐。
- 学习率过大过小都有问题。一个合适的学习率通常是需要通过多次实验找到的。

### 7.2.9 练习

- 运行本节中实验代码。比较一下随机梯度下降和梯度下降的运行时间。
- 梯度下降和随机梯度下降虽然看上去有效，但可能会有哪些问题？

### 7.2.10 扫码直达讨论区



## 7.2.11 参考文献

- [1] Stewart, James. “Calculus: Early Transcendentals (7th Edition).” Brooks Cole (2010).
- [2] Gürbüzbalaban, Mert, Asu Ozdaglar, and Pablo Parrilo. “Why random reshuffling beats stochastic gradient descent.” arXiv preprint arXiv:1510.08560 (2018).

## 7.3 梯度下降和随机梯度下降——使用 Gluon

在 Gluon 里，使用小批量随机梯度下降很方便，我们无需重新实现该算法。特别地，当批量大小等于数据集样本数时，该算法即为梯度下降；批量大小为 1 即为随机梯度下降。

首先，导入本节中实验所需的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import nn, data as gdata, loss as gloss
        import numpy as np
```

下面生成实验数据集并定义线性回归模型。

```
In [2]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)

        # 线性回归模型。
        net = nn.Sequential()
        net.add(nn.Dense(1))
```

为了使学习率能够自我衰减，我们需要访问 `gluon.Trainer` 的 `learning_rate` 属性并使用 `set_learning_rate` 函数。

```
In [3]: # 优化目标函数。
        def optimize(batch_size, trainer, num_epochs, decay_epoch, log_interval,
                    features, labels, net):
```

```

dataset = gdata.ArrayDataset(features, labels)
data_iter = gdata.DataLoader(dataset, batch_size, shuffle=True)
loss = gloss.L2Loss()
ls = [loss(net(features), labels).mean().asnumpy()]
for epoch in range(1, num_epochs + 1):
    # 学习率自我衰减。
    if decay_epoch and epoch > decay_epoch:
        trainer.set_learning_rate(trainer.learning_rate * 0.1)
    for batch_i, (X, y) in enumerate(data_iter):
        with autograd.record():
            l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        if batch_i * batch_size % log_interval == 0:
            ls.append(loss(net(features), labels).mean().asnumpy())
    # 为了便于打印，改变输出形状并转化成 numpy 数组。
    print('w:', net[0].weight.data(), '\nb:', net[0].bias.data(), '\n')
es = np.linspace(0, num_epochs, len(ls), endpoint=True)
gb.semilogy(es, ls, 'epoch', 'loss')

```

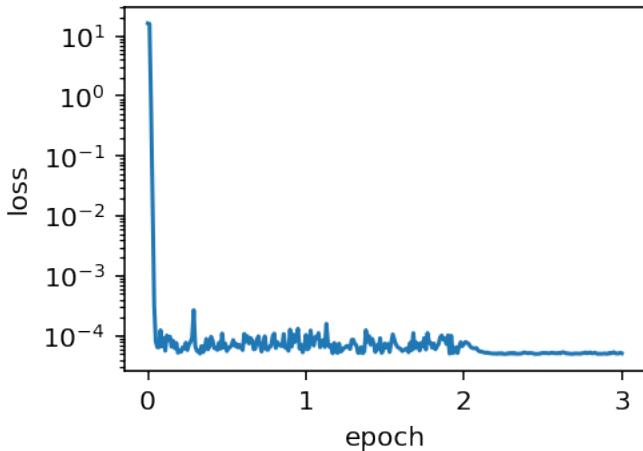
以下几组实验分别重现了“梯度下降和随机梯度下降——从零开始”一节中实验结果。

```

In [4]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.2})
        optimize(batch_size=1, trainer=trainer, num_epochs=3, decay_epoch=2,
                 log_interval=10, features=features, labels=labels, net=net)

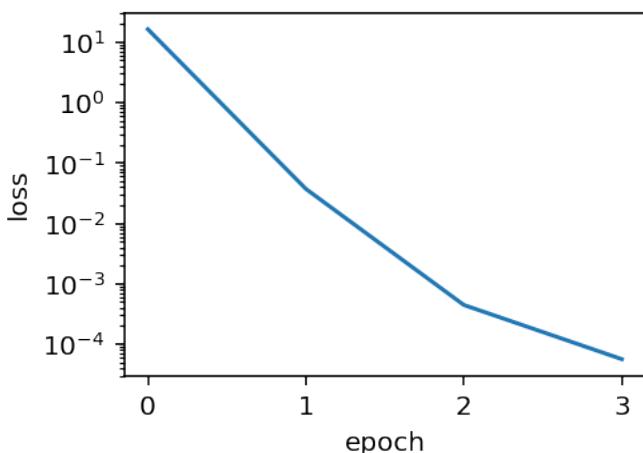
w:
[[ 2.00109863 -3.39862871]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.19896603]
<NDArray 1 @cpu(0)>

```



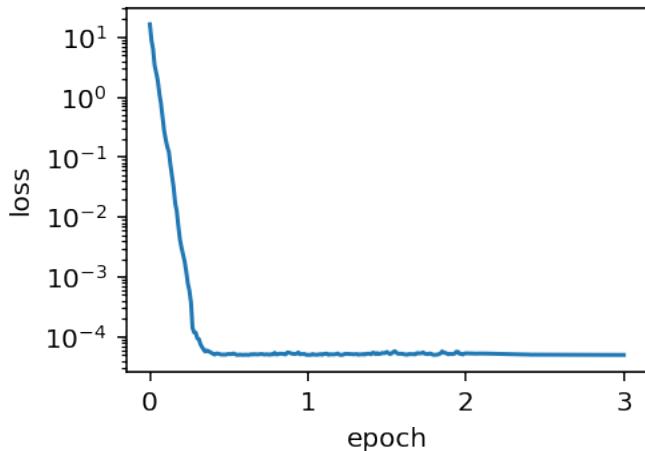
```
In [5]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.999})
        optimize(batch_size=1000, trainer=trainer, num_epochs=3, decay_epoch=None,
                 log_interval=1000, features=features, labels=labels, net=net)

w:
[[ 2.00053716 -3.40285206]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.19865417]
<NDArray 1 @cpu(0)>
```



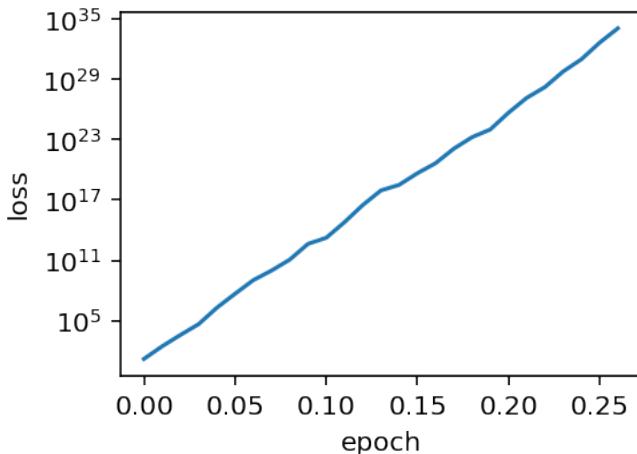
```
In [6]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.2})
    optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
            log_interval=10, features=features, labels=labels, net=net)

w:
[[ 1.99975026 -3.39993167]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20001078]
<NDArray 1 @cpu(0)>
```



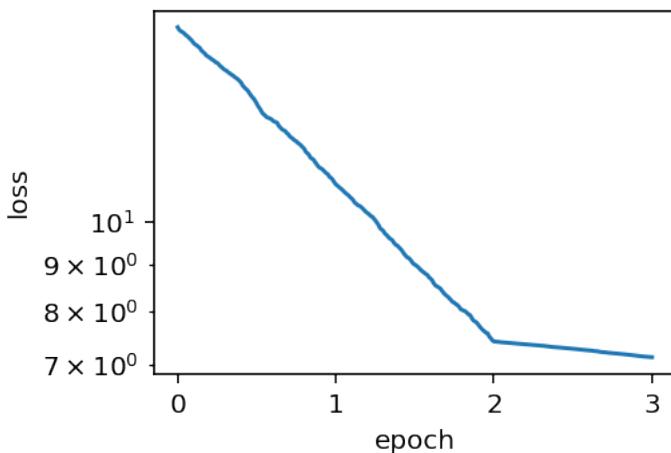
```
In [7]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 5})
    optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
            log_interval=10, features=features, labels=labels, net=net)

w:
[[ nan nan]]
<NDArray 1x2 @cpu(0)>
b:
[ nan]
<NDArray 1 @cpu(0)>
```



```
In [8]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.002})
        optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
                 log_interval=10, features=features, labels=labels, net=net)

w:
[[ 0.69031787 -1.20052135]]
<NDArray 1x2 @cpu(0)>
b:
[ 1.37669992]
<NDArray 1 @cpu(0)>
```



### 7.3.1 小结

- 使用 Gluon 的 `Trainer` 可以方便地使用小批量随机梯度下降。
- 访问 `gluon.Trainer` 的 `learning_rate` 属性并使用 `set_learning_rate` 函数可以在迭代过程中调整学习率。

### 7.3.2 练习

- 查阅网络或书本资料，了解学习率自我衰减的其他方法。

### 7.3.3 扫码直达讨论区



## 7.4 动量法——从零开始

我们已经介绍了梯度下降。每次迭代时，该算法根据自变量当前所在位置，沿着目标函数下降最快的方向更新自变量。因此，梯度下降有时也叫做最陡下降（steepest descent）。目标函数有关自变量的梯度代表了目标函数下降最快的方向。

### 7.4.1 梯度下降的问题

给定目标函数，在梯度下降中，自变量的迭代方向仅仅取决于自变量当前位置。这可能会带来一些问题。

考虑一个输入和输出分别为二维向量  $\mathbf{x} = [x_1, x_2]^\top$  和标量的目标函数  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ 。图 7.3 展示了该函数的等高线示意图。

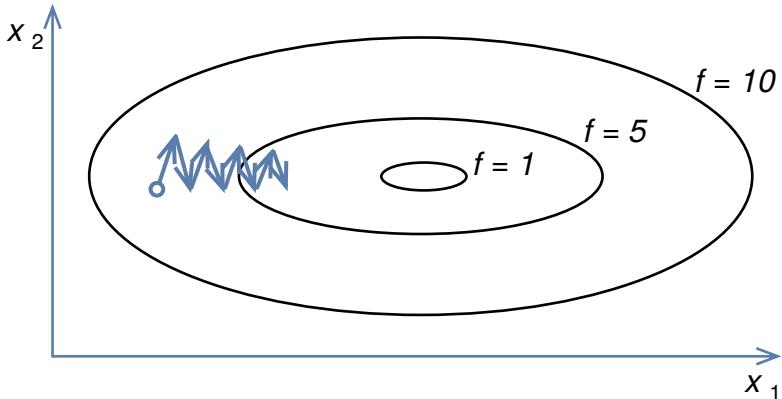


图 7.2: 目标函数  $f$  的等高线图和自变量  $[x_1, x_2]$  在梯度下降中的迭代。每条等高线（椭圆实线）代表所有函数值相同的自变量的坐标。实心圆代表自变量初始坐标。每个箭头头部代表自变量在每次迭代后的坐标。

由于目标函数在竖直方向 ( $x_2$  轴方向) 比在水平方向 ( $x_1$  轴方向) 更弯曲, 给定学习率, 梯度下降迭代自变量时会使自变量在竖直方向比在水平方向移动幅度更大。因此, 我们需要一个较小的学习率从而避免自变量在竖直方向上越过目标函数最优解。然而, 这造成了图 7.3 中自变量向最优解移动较慢。

## 7.4.2 动量法

动量法的提出是为了应对梯度下降的上述问题。广义上, 以小批量随机梯度下降为例 (当批量大小等于训练集样本数时, 该算法即为梯度下降; 批量大小为 1 时即为随机梯度下降), 我们对小批量随机梯度算法在每次迭代的步骤做如下修改:

$$\mathbf{v} \leftarrow \gamma \mathbf{v} + \eta \nabla f_{\mathcal{B}}(\mathbf{x}),$$

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{v}.$$

其中  $\mathbf{v}$  是速度变量, 动量超参数  $\gamma$  满足  $0 \leq \gamma \leq 1$ 。动量法中的学习率  $\eta$  和有关小批量  $B$  的随机梯度  $\nabla f_{\mathcal{B}}(\mathbf{x})$  已在“梯度下降和随机梯度下降”一节中描述。

## 指数加权移动平均

为了更清晰地理解动量法，让我们先解释指数加权移动平均（exponentially weighted moving average）。给定超参数  $\gamma$  且  $0 \leq \gamma < 1$ ，当前时刻  $t$  的变量  $y^{(t)}$  是上一时刻  $t - 1$  的变量  $y^{(t-1)}$  和当前时刻另一变量  $x^{(t)}$  的线性组合：

$$y^{(t)} = \gamma y^{(t-1)} + (1 - \gamma)x^{(t)}.$$

我们可以对  $y^{(t)}$  展开：

$$\begin{aligned} y^{(t)} &= (1 - \gamma)x^{(t)} + \gamma y^{(t-1)} \\ &= (1 - \gamma)x^{(t)} + (1 - \gamma) \cdot \gamma x^{(t-1)} + \gamma^2 y^{(t-2)} \\ &= (1 - \gamma)x^{(t)} + (1 - \gamma) \cdot \gamma x^{(t-1)} + (1 - \gamma) \cdot \gamma^2 x^{(t-2)} + \gamma^3 y^{(t-3)} \\ &\dots \end{aligned}$$

由于

$$\lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right)^n = \exp(-1) \approx 0.3679,$$

我们可以将  $\gamma^{1/(1-\gamma)}$  近似为  $\exp(-1)$ 。例如  $0.95^{20} \approx \exp(-1)$ 。如果把  $\exp(-1)$  当做一个比较小的数，我们可以在近似中忽略所有含  $\gamma^{1/(1-\gamma)}$  和比  $\gamma^{1/(1-\gamma)}$  更高阶的系数的项。例如，当  $\gamma = 0.95$  时，

$$y^{(t)} \approx 0.05 \sum_{i=0}^{19} 0.95^i x^{(t-i)}.$$

因此，在实际中，我们常常将  $y$  看作是对最近  $1/(1 - \gamma)$  个时刻的  $x$  值的加权平均。例如，当  $\gamma = 0.95$  时， $y$  可以被看作是对最近 20 个时刻的  $x$  值的加权平均；当  $\gamma = 0.9$  时， $y$  可以看作是对最近 10 个时刻的  $x$  值的加权平均：离当前时刻越近的  $x$  值获得的权重越大。

## 由指数加权移动平均理解动量法

现在，我们对动量法的速度变量做变形：

$$\mathbf{v} \leftarrow \gamma \mathbf{v} + (1 - \gamma) \frac{\eta \nabla f_{\mathcal{B}}(\mathbf{x})}{1 - \gamma}.$$

由指数加权移动平均的形式可得，速度变量  $\mathbf{v}$  实际上对  $(\eta \nabla f_{\mathcal{B}}(\mathbf{x})) / (1 - \gamma)$  做了指数加权移动平均。给定动量超参数  $\gamma$  和学习率  $\eta$ ，含动量法的小批量随机梯度下降可被看作使用了特殊梯度来迭代目标函数的自变量。这个特殊梯度是最近  $1/(1 - \gamma)$  个时刻的  $\nabla f_{\mathcal{B}}(\mathbf{x}) / (1 - \gamma)$  的加权平均。

给定目标函数，在动量法的每次迭代中，自变量在各个方向上的移动幅度不仅取决于当前梯度，还取决于过去各个梯度在各个方向上是否一致。图 7.4 展示了使用动量法的梯度下降迭代图 7.3 中目标函数自变量的情景。我们将每个梯度代表的箭头方向在水平方向和竖直方向做分解。由于所有梯度的水平方向为正（向右）、在竖直上时正（向上）时负（向下），自变量在水平方向移动幅度逐渐增大，而在竖直方向移动幅度逐渐减小。这样，我们就可以使用较大的学习率，从而使自变量向最优解更快移动。

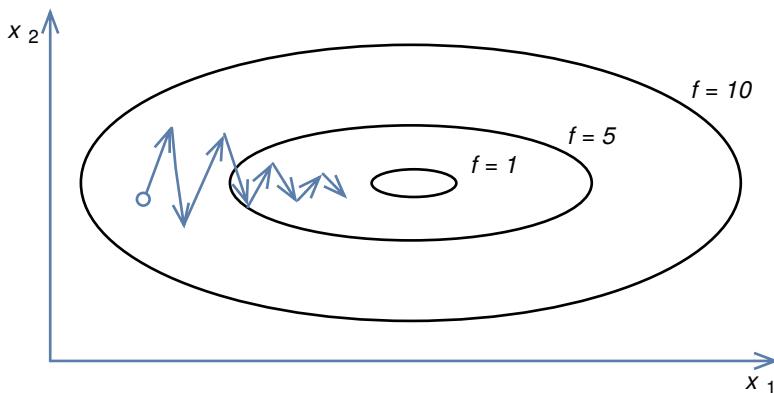


图 7.3: 目标函数  $f$  的等高线图和自变量  $[x_1, x_2]$  在使用动量法的梯度下降中的迭代。每条等高线（椭圆实线）代表所有函数值相同的自变量的坐标。实心圆代表自变量初始坐标。每个箭头头部代表自变量在每次迭代后的坐标。

### 7.4.3 动量法的实现

动量法的实现也很简单。我们在小批量随机梯度下降的基础上添加速度变量。

```
In [1]: def sgd_momentum(params, vs, lr, mom, batch_size):
    for param, v in zip(params, vs):
        v[:] = mom * v + lr * param.grad / batch_size
        param[:] -= v
```

### 7.4.4 实验

首先，导入本节中实验所需的包或模块。

```
In [2]: import sys
sys.path.append('..')
```

```
import gluonbook as gb
from mxnet import autograd, nd
import numpy as np
```

实验中，我们以之前介绍过的线性回归为例。设数据集的样本数为 1000，我们使用权重  $w$  为 [2, -3.4]，偏差 “ $b$ ” 为 4.2 的线性回归模型来生成数据集。该模型的平方损失函数即所需优化的目标函数，模型参数即目标函数自变量。

```
In [3]: # 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)

# 初始化模型参数。
def init_params():
    w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    vs = []
    for param in params:
        param.attach_grad()
        # 把速度项初始化为和参数形状相同的零张量。
        vs.append(param.zeros_like())
    return params, vs
```

优化函数 `optimize` 与 “梯度下降和随机梯度下降——从零开始” 一节中的类似。

```
In [4]: net = gb.linreg
loss = gb.squared_loss

def optimize(batch_size, lr, mom, num_epochs, log_interval):
    [w, b], vs = init_params()
    ls = [loss(net(features, w, b), labels).mean().asnumpy()]
    for epoch in range(1, num_epochs + 1):
        # 学习率自我衰减。
        if epoch > 2:
            lr *= 0.1
        for batch_i, (X, y) in enumerate(
            gb.data_iter(batch_size, num_examples, features, labels)):
            with autograd.record():
```

```

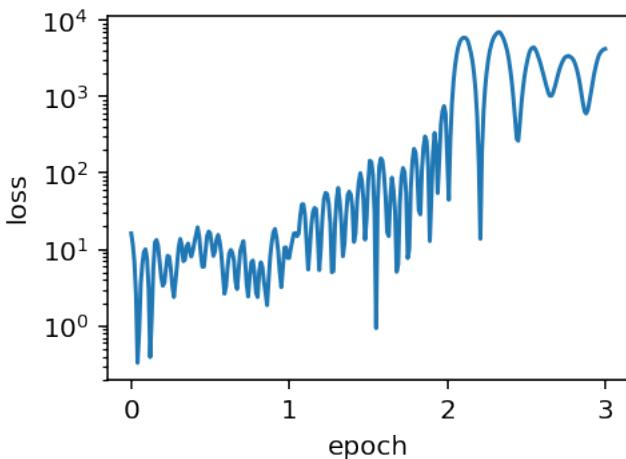
        l = loss(net(X, w, b), y)
        l.backward()
        sgd_momentum([w, b], vs, lr, mom, batch_size)
        if batch_i * batch_size % log_interval == 0:
            ls.append(loss(net(features, w, b), labels).mean().asnumpy())
    print('w:', w, '\nb:', b, '\n')
    es = np.linspace(0, num_epochs, len(ls), endpoint=True)
    gb.semilogy(es, ls, 'epoch', 'loss')

```

我们先将动量超参数  $\gamma$  (mom) 设 0.99。此时，小梯度随机梯度下降可被看作使用了特殊梯度：这个特殊梯度是最近 100 个时刻的  $100\nabla f_B(x)$  的加权平均。我们观察到，损失函数值在 3 个迭代周期后上升。这很可能是由于特殊梯度中较大的系数 100 造成的。

In [5]: `optimize(batch_size=10, lr=0.2, mom=0.99, num_epochs=3, log_interval=10)`

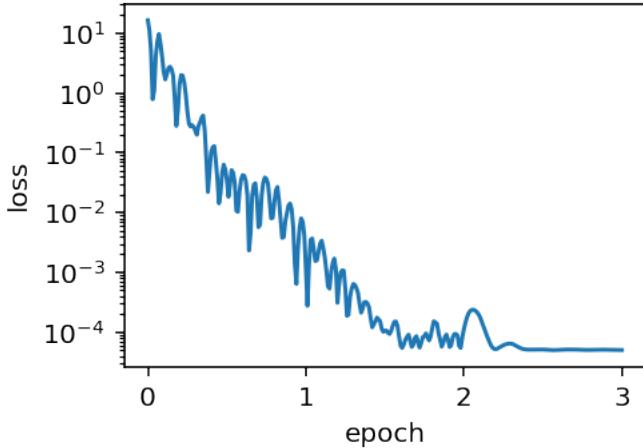
```
w:
[-84.86100006]
[-21.47795296]
<NDArray 2x1 @cpu(0)>
b:
[ 23.79038239]
<NDArray 1 @cpu(0)>
```



假设学习率不变，为了降低上述特殊梯度中的系数，我们将动量超参数  $\gamma$  (mom) 设 0.9。此时，上述特殊梯度变成最近 10 个时刻的  $10\nabla f_B(x)$  的加权平均。我们观察到，损失函数值在 3 个迭代周期后下降。

In [6]: `optimize(batch_size=10, lr=0.2, mom=0.9, num_epochs=3, log_interval=10)`

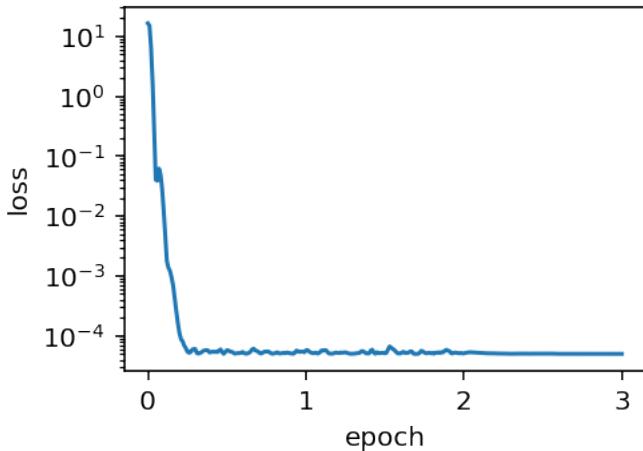
```
w:  
[[ 1.99947262]  
[-3.39972782]]  
<NDArray 2x1 @cpu(0)>  
b:  
[ 4.19970846]  
<NDArray 1 @cpu(0)>
```



继续保持学习率不变，我们将动量超参数  $\gamma$  (mom) 设 0.5。此时，小梯度随机梯度下降可被看作使用了新的特殊梯度：这个特殊梯度是最近 2 个时刻的  $2\nabla f_B(x)$  的加权平均。我们观察到，损失函数值在 3 个迭代周期后下降，且下降曲线较平滑。最终，优化所得的模型参数值与它们的真实值较接近。

```
In [7]: optimize(batch_size=10, lr=0.2, mom=0.5, num_epochs=3, log_interval=10)
```

```
w:  
[[ 1.99960148]  
[-3.40039492]]  
<NDArray 2x1 @cpu(0)>  
b:  
[ 4.20015764]  
<NDArray 1 @cpu(0)>
```



#### 7.4.5 小结

- 动量法使用了指数加权移动平均的思想。

#### 7.4.6 练习

- 使用其他动量超参数和学习率的组合，观察实验结果。

#### 7.4.7 扫码直达讨论区



### 7.5 动量法——使用 Gluon

在 Gluon 里，使用动量法很方便，我们无需重新实现该算法。

首先，导入本节中实验所需的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import gluon, init, nd
        from mxnet.gluon import nn
```

下面生成实验数据集并定义线性回归模型。

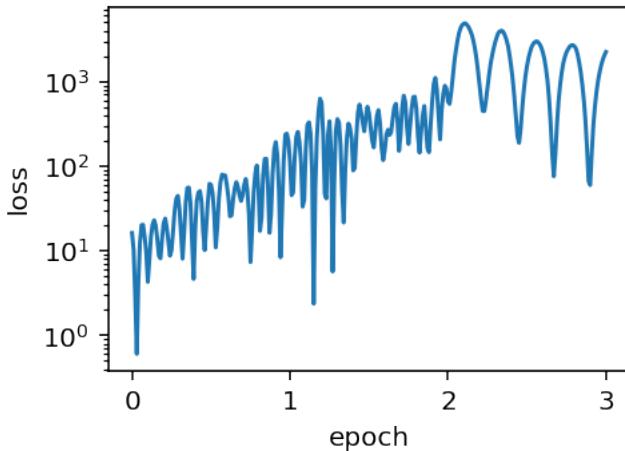
```
In [2]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)

        # 线性回归模型。
        net = nn.Sequential()
        net.add(nn.Dense(1))
```

例如，以使用动量法的小批量随机梯度下降为例，我们可以在 `Trainer` 中定义动量超参数 `momentum`。以下几组实验分别重现了“动量法——从零开始”一节中实验结果。

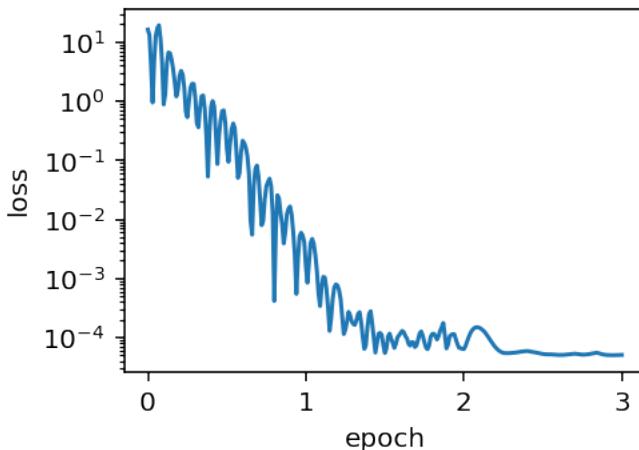
```
In [3]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd',
                               {'learning_rate': 0.2, 'momentum': 0.99})
        gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
                    log_interval=10, features=features, labels=labels, net=net)

w:
[[ 31.66460037  39.30710983]]
<NDArray 1x2 @cpu(0)>
b:
[-38.30800247]
<NDArray 1 @cpu(0)>
```



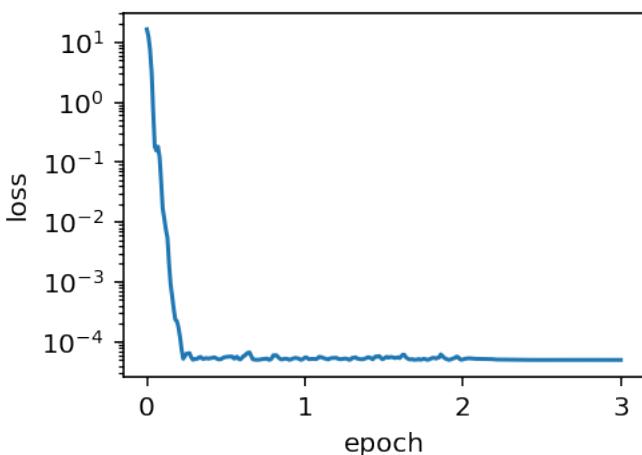
```
In [4]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd',
                               {'learning_rate': 0.2, 'momentum': 0.9})
        gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
                    log_interval=10, features=features, labels=labels, net=net)

w:
[[ 1.99976742 -3.39952111]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20124578]
<NDArray 1 @cpu(0)>
```



```
In [5]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                           {'learning_rate': 0.2, 'momentum': 0.5})
    gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
                log_interval=10, features=features, labels=labels, net=net)

w:
[[ 1.99998426 -3.40014744]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20027399]
<NDArray 1 @cpu(0)>
```



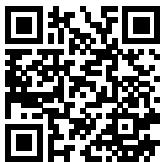
### 7.5.1 小结

- 使用 Gluon 的 Trainer 可以方便地使用动量法。

### 7.5.2 练习

- 如果想用以上代码重现小批量随机梯度下降，应该把动量参数改为多少？

### 7.5.3 扫码直达讨论区



## 7.6 Adagrad——从零开始

在我们之前介绍过的优化算法中，无论是梯度下降、随机梯度下降、小批量随机梯度下降还是使用动量法，目标函数自变量的每一个元素在相同时刻都使用同一个学习率来自我迭代。

举个例子，假设目标函数为  $f$ ，自变量为一个多维向量  $[x_1, x_2]^\top$ ，该向量中每一个元素在更新时都使用相同的学习率。例如在学习率为  $\eta$  的梯度下降中，元素  $x_1$  和  $x_2$  都使用相同的学习率  $\eta$  来自我迭代：

$$\begin{aligned}x_1 &\leftarrow x_1 - \eta \frac{\partial f}{\partial x_1}, \\x_2 &\leftarrow x_2 - \eta \frac{\partial f}{\partial x_2}.\end{aligned}$$

如果让  $x_1$  和  $x_2$  使用不同的学习率自我迭代呢？实际上，Adagrad 就是一个在迭代过程中不断自我调整学习率，并让模型参数中每个元素都使用不同学习率的优化算法 [1]。

下面，我们将介绍 Adagrad 算法。关于本节中涉及到的按元素运算，例如标量与向量计算以及按元素相乘  $\odot$ ，请参见“数学基础”一节。

### 7.6.1 Adagrad 算法

Adagrad 的算法会使用一个小批量随机梯度按元素平方的累加变量  $s$ ，并将其中每个元素初始化为 0。在每次迭代中，首先计算小批量随机梯度  $g$ ，然后将该梯度按元素平方后累加到变量  $s$ ：

$$s \leftarrow s + g \odot g.$$

然后，我们将目标函数自变量中每个元素的学习率通过按元素运算重新调整一下：

$$g' \leftarrow \frac{\eta}{\sqrt{s + \epsilon}} \odot g,$$

其中  $\eta$  是初始学习率且  $\eta > 0$ ,  $\epsilon$  是为了维持数值稳定性而添加的常数, 例如  $10^{-7}$ 。我们需要注意其中按元素开方、除法和乘法的运算。这些按元素运算使得目标函数自变量中每个元素都分别拥有自己的学习率。

最后, 自变量的迭代步骤与小批量随机梯度下降类似。只是这里梯度前的学习率已经被调整过了:

$$x \leftarrow x - g'.$$

## 7.6.2 Adagrad 的特点

需要强调的是, 小批量随机梯度按元素平方的累加变量  $s$  出现在含调整后学习率的梯度  $g'$  的分母项。因此, 如果目标函数有关自变量中某个元素的偏导数一直都较大, 那么就让该元素的学习率下降快一点; 反之, 如果目标函数有关自变量中某个元素的偏导数一直都较小, 那么就让该元素的学习率下降慢一点。然而, 由于  $s$  一直在累加按元素平方的梯度, 自变量中每个元素的学习率在迭代过程中一直在降低 (或不变)。所以, 当学习率在迭代早期降得较快且当前解依然不佳时, Adagrad 在迭代后期由于学习率过小, 可能较难找到一个有用的解。

## 7.6.3 Adagrad 的实现

Adagrad 的实现很简单。我们只需要把上面的数学公式翻译成代码。

```
In [1]: def adagrad(params, sqrs, lr, batch_size):
    eps_stable = 1e-7
    for param, sqr in zip(params, sqrs):
        g = param.grad / batch_size
        sqr[:] += g.square()
        param[:] -= lr * g / (sqr + eps_stable).sqrt()
```

## 7.6.4 实验

首先, 导入本节中实验所需的包或模块。

```
In [2]: import sys
sys.path.append('..')
import gluonbook as gb
from mxnet import autograd, nd
import numpy as np
```

实验中，我们以之前介绍过的线性回归为例。设数据集的样本数为 1000，我们使用权重  $w$  为 [2, -3.4]，偏差 “ $b$ ” 为 4.2 的线性回归模型来生成数据集。该模型的平方损失函数即所需优化的目标函数，模型参数即目标函数自变量。

我们把梯度按元素平方的累加变量初始化为和模型参数形状相同的零张量。

```
In [3]: # 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)

# 初始化模型参数。
def init_params():
    w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    sqrs = []
    for param in params:
        param.attach_grad()
        # 把梯度按元素平方的累加变量初始化为和参数形状相同的零张量。
        sqrs.append(param.zeros_like())
    return params, sqrs
```

优化函数 `optimize` 与 “梯度下降和随机梯度下降” 一节中的类似。需要指出的是，这里的初始学习率  $lr$  无需自我衰减。

```
In [4]: net = gb.linreg
loss = gb.squared_loss

def optimize(batch_size, lr, num_epochs, log_interval):
    [w, b], sqrs = init_params()
    ls = [loss(net(features, w, b), labels).mean().asnumpy()]
    for epoch in range(1, num_epochs + 1):
        for batch_i, (X, y) in enumerate(
            gb.data_iter(batch_size, num_examples, features, labels)):
            with autograd.record():
                l = loss(net(X, w, b), y)
                l.backward()
                adagrad([w, b], sqrs, lr, batch_size)
            if batch_i * batch_size % log_interval == 0:
```

```

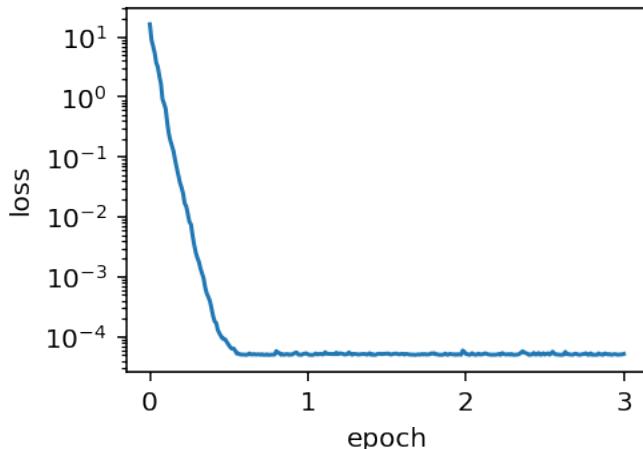
ls.append(loss(net(features, w, b), labels).mean().asnumpy())
print('w:', w, '\nb:', b, '\n')
es = np.linspace(0, num_epochs, len(ls), endpoint=True)
gb.semilogy(es, ls, 'epoch', 'loss')

```

最终，优化所得的模型参数值与它们的真实值较接近。

```
In [5]: optimize(batch_size=10, lr=0.9, num_epochs=3, log_interval=10)

w:
[[ 2.00155687]
 [-3.4009819 ]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.20012903]
<NDArray 1 @cpu(0)>
```



## 7.6.5 小结

- Adagrad 在迭代过程中不断调整学习率，并让目标函数自变量中每个元素都分别拥有自己的学习率。
- 使用 Adagrad 时，自变量中每个元素的学习率在迭代过程中一直在降低（或不变）。

## 7.6.6 练习

- 在介绍 Adagrad 的特点时，我们提到了它可能存在的问题。你能想到什么办法来应对这个问题？

## 7.6.7 扫码直达讨论区



## 7.6.8 参考文献

[1] Duchi, John, Elad Hazan, and Yoram Singer. “Adaptive subgradient methods for online learning and stochastic optimization.” Journal of Machine Learning Research 12.Jul (2011): 2121-2159.

## 7.7 Adagrad——使用 Gluon

在 Gluon 里，使用 Adagrad 很方便，我们无需重新实现该算法。

首先，导入本节中实验所需的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import gluon, init, nd
        from mxnet.gluon import nn
```

下面生成实验数据集并定义线性回归模型。

```
In [2]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
```

```

true_b = 4.2
features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)

# 线性回归模型。
net = nn.Sequential()
net.add(nn.Dense(1))

```

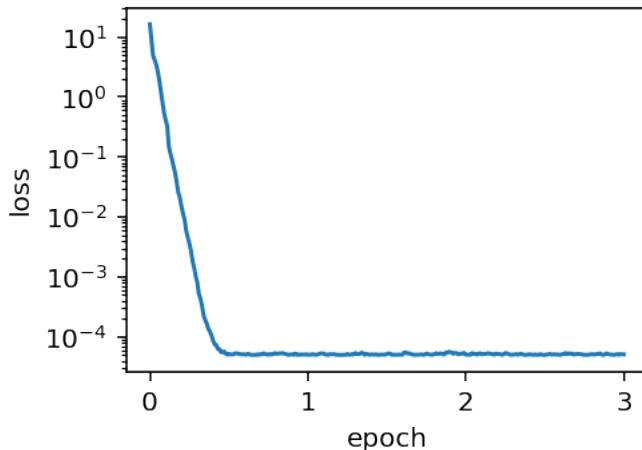
我们可以在 Trainer 中定义优化算法名称 adagrad。以下实验分别重现了“Adagrad——从零开始”一节中实验结果。

```

In [3]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'adagrad',
                                {'learning_rate': 0.9})
        gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=None,
                    log_interval=10, features=features, labels=labels, net=net)

w:
[[ 2.00077629 -3.39881825]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20059013]
<NDArray 1 @cpu(0)>

```



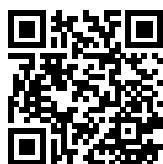
### 7.7.1 小结

- 使用 Gluon 的 `Trainer` 可以方便地使用 Adagrad。

### 7.7.2 练习

- 尝试使用其他的初始学习率，结果有什么变化？

### 7.7.3 扫码直达讨论区



## 7.8 RMSProp——从零开始

我们在“Adagrad——从零开始”一节里提到，由于调整学习率时分母上的变量  $s$  一直在累加按元素平方的小批量随机梯度，目标函数自变量每个元素的学习率在迭代过程中一直在降低（或不变）。所以，当学习率在迭代早期降得较快且当前解依然不佳时，Adagrad 在迭代后期由于学习率过小，可能较难找到一个有用的解。为了应对这一问题，RMSProp 算法对 Adagrad 做了一点小小的修改 [1]。

下面，我们来描述 RMSProp 算法。

### 7.8.1 RMSProp 算法

我们在“动量法——从零开始”一节里介绍过指数加权移动平均。事实上，RMSProp 算法使用了小批量随机梯度按元素平方的指数加权移动平均变量  $s$ ，并将其中每个元素初始化为 0。给定超参数  $\gamma$  且  $0 \leq \gamma < 1$ ，在每次迭代中，RMSProp 首先计算小批量随机梯度  $g$ ，然后对该梯度按元素平方项  $g \odot g$  做指数加权移动平均，记为  $s$ ：

$$s \leftarrow \gamma s + (1 - \gamma)g \odot g.$$

然后，和 Adagrad 一样，将目标函数自变量中每个元素的学习率通过按元素运算重新调整一下：

$$\mathbf{g}' \leftarrow \frac{\eta}{\sqrt{s + \epsilon}} \odot \mathbf{g},$$

其中  $\eta$  是初始学习率且  $\eta > 0$ ， $\epsilon$  是为了维持数值稳定性而添加的常数，例如  $10^{-8}$ 。和 Adagrad 一样，模型参数中每个元素都分别拥有自己的学习率。同样地，最后的自变量迭代步骤与小批量随机梯度下降类似：

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{g}'.$$

需要强调的是，RMSProp 只在 Adagrad 的基础上修改了变量  $s$  的更新方法：对平方项  $\mathbf{g} \odot \mathbf{g}$  从累加变成了指数加权移动平均。由于变量  $s$  可看作是最近  $1/(1 - \gamma)$  个时刻的平方项  $\mathbf{g} \odot \mathbf{g}$  的加权平均，自变量每个元素的学习率在迭代过程中避免了“直降不升”的问题。

## 7.8.2 RMSProp 的实现

RMSProp 的实现很简单。我们只需要把上面的数学公式翻译成代码。

```
In [1]: def rmsprop(params, sqrs, lr, gamma, batch_size):
    eps_stable = 1e-8
    for param, sqr in zip(params, sqrs):
        g = param.grad / batch_size
        sqr[:] = gamma * sqr + (1 - gamma) * g.square()
        param[:] -= lr * g / (sqr + eps_stable).sqrt()
```

## 7.8.3 实验

首先，导入本节中实验所需的包或模块。

```
In [2]: import sys
sys.path.append('..')
import gluonbook as gb
from mxnet import autograd, nd
import numpy as np
```

实验中，我们依然以线性回归为例。设数据集的样本数为 1000，我们使用权重  $w$  为  $[2, -3.4]$ ，偏差 “ $b$ ” 为 4.2 的线性回归模型来生成数据集。该模型的平方损失函数即所需优化的目标函数，模型参数即目标函数自变量。

我们把小批量随机梯度按元素平方的指数加权移动平均变量  $s$  初始化为和模型参数形状相同的零张量。

```
In [3]: # 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)

# 初始化模型参数。
def init_params():
    w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    sqrs = []
    for param in params:
        param.attach_grad()
    # 把梯度按元素平方的指数加权移动平均变量初始化为和参数形状相同的零张量。
    sqrs.append(param.zeros_like())
    return params, sqrs
```

优化函数 `optimize` 与“Adagrad——从零开始”一节中的类似。

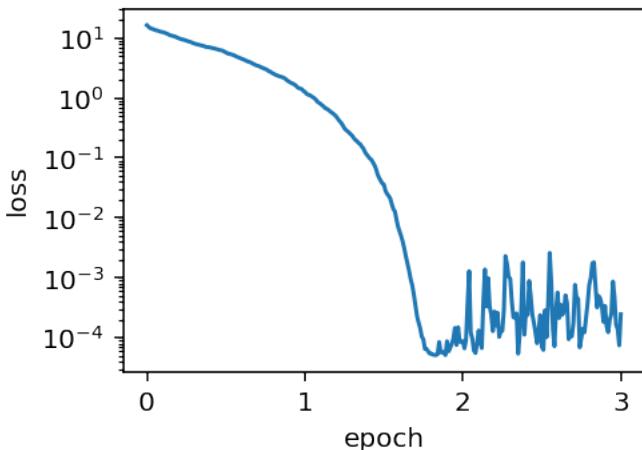
```
In [4]: net = gb.linreg
loss = gb.squared_loss

def optimize(batch_size, lr, gamma, num_epochs, log_interval):
    [w, b], sqrs = init_params()
    ls = [loss(net(features, w, b), labels).mean().asnumpy()]
    for epoch in range(1, num_epochs + 1):
        for batch_i, (X, y) in enumerate(
            gb.data_iter(batch_size, num_examples, features, labels)):
            with autograd.record():
                l = loss(net(X, w, b), y)
                l.backward()
                rmsprop([w, b], sqrs, lr, gamma, batch_size)
            if batch_i * batch_size % log_interval == 0:
                ls.append(loss(net(features, w, b), labels).mean().asnumpy())
    print('w:', w, '\nb:', b, '\n')
    es = np.linspace(0, num_epochs, len(ls), endpoint=True)
    gb.semilogy(es, ls, 'epoch', 'loss')
```

我们将初始学习率设为 0.03，并将  $\gamma$  (gamma) 设为 0.9。此时，变量  $s$  可看作是最近  $1/(1-0.9) = 10$  个时刻的平方项  $\mathbf{g} \odot \mathbf{g}$  的加权平均。我们观察到，损失函数在迭代后期较震荡。

```
In [5]: optimize(batch_size=10, lr=0.03, gamma=0.9, num_epochs=3, log_interval=10)

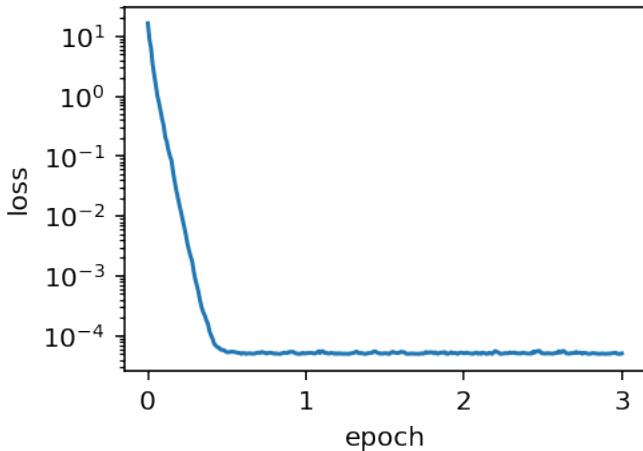
w:
[[ 1.99026108]
 [-3.41502833]]
<NDArray 2x1 @cpu(0)>
b:
b:
[ 4.19347048]
<NDArray 1 @cpu(0)>
```



我们将  $\gamma$  调大一点，例如 0.999。此时，变量  $s$  可看作是最近  $1/(1 - 0.999) = 1000$  个时刻的平方项  $\mathbf{g} \odot \mathbf{g}$  的加权平均。这时损失函数在迭代后期较平滑。

```
In [6]: optimize(batch_size=10, lr=0.03, gamma=0.999, num_epochs=3, log_interval=10)

w:
[[ 1.99943686]
 [-3.40103698]]
<NDArray 2x1 @cpu(0)>
b:
b:
[ 4.19961596]
<NDArray 1 @cpu(0)>
```



#### 7.8.4 小结

- RMSProp 和 Adagrad 的不同在于， RMSProp 使用了小批量随机梯度按元素平方的指数加权移动平均变量来调整学习率。
- 理解指数加权移动平均有助于我们调节 RMSProp 算法中的超参数，例如  $\gamma$ 。

#### 7.8.5 练习

- 把  $\gamma$  的值设为 0 或 1，观察并分析实验结果。

#### 7.8.6 扫码直达讨论区



## 7.8.7 参考文献

[1] Tieleman, Tijmen, and Geoffrey Hinton. “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude.” COURSERA: Neural networks for machine learning 4.2 (2012): 26-31.

## 7.9 RMSProp——使用 Gluon

在 Gluon 里，使用 RMSProp 很方便，我们无需重新实现该算法。

首先，导入本节中实验所需的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import gluon, init, nd
        from mxnet.gluon import nn
```

下面生成实验数据集并定义线性回归模型。

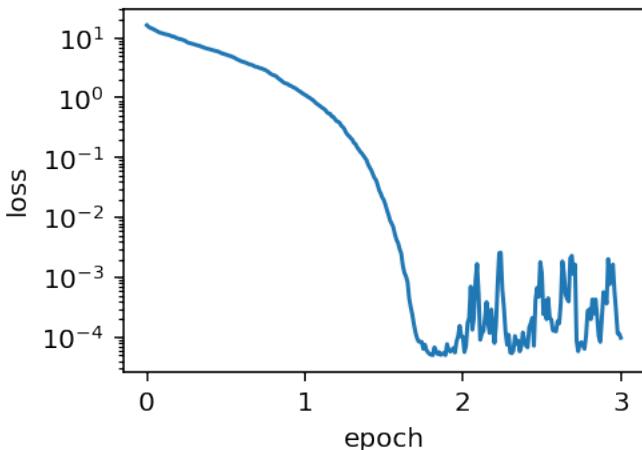
```
In [2]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)

        # 线性回归模型。
        net = nn.Sequential()
        net.add(nn.Dense(1))
```

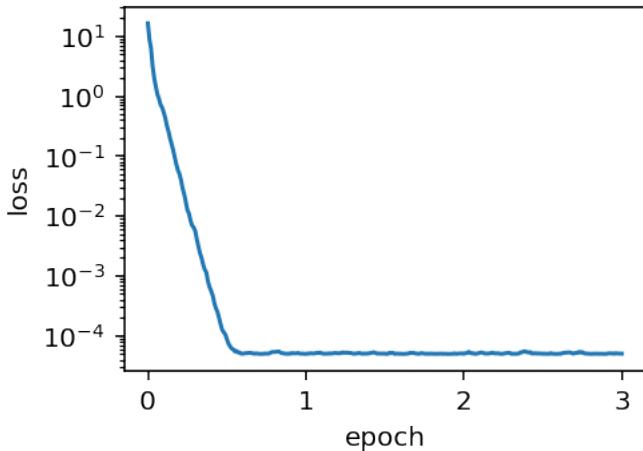
我们可以在 Trainer 中定义优化算法名称 `rmsprop` 并定义  $\gamma$  超参数 `gamma1`。以下几组实验分别重现了“RMSProp——从零开始”一节中实验结果。

```
In [3]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'rmsprop',
                               {'learning_rate': 0.03, 'gamma1': 0.9})
        gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=None,
                    log_interval=10, features=features, labels=labels, net=net)
```

```
w:  
[[ 2.00707316 -3.39467239]]  
<NDArray 1x2 @cpu(0)>  
b:  
[ 4.19605732]  
<NDArray 1 @cpu(0)>
```



```
In [4]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)  
      trainer = gluon.Trainer(net.collect_params(), 'rmsprop',  
                             {'learning_rate': 0.03, 'gamma1': 0.999})  
      gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=None,  
                  log_interval=10, features=features, labels=labels, net=net)  
  
w:  
[[ 2.00028229 -3.40018582]]  
<NDArray 1x2 @cpu(0)>  
b:  
[ 4.199296]  
<NDArray 1 @cpu(0)>
```



### 7.9.1 小结

- 使用 Gluon 的 `Trainer` 可以方便地使用 RMSProp。

### 7.9.2 练习

- 试着使用其他的初始学习率和  $\gamma$  超参数的组合，观察并分析实验结果。

### 7.9.3 扫码直达讨论区



## 7.10 Adadelta——从零开始

我们在“RMSProp——从零开始”一节中描述了，RMSProp 针对 Adagrad 在迭代后期可能较难找到有用解的问题，对小批量随机梯度按元素平方项做指数加权移动平均而不是累加。另一种应对

该问题的优化算法叫做 Adadelta [1]。有意思的是，它没有学习率超参数。

### 7.10.1 Adadelta 算法

Adadelta 算法也像 RMSProp 一样，使用了小批量随机梯度按元素平方的指数加权移动平均变量  $s$ ，并将其中每个元素初始化为 0。给定超参数  $\rho$  且  $0 \leq \rho < 1$ ，在每次迭代中，RMSProp 首先计算小批量随机梯度  $\mathbf{g}$ ，然后对该梯度按元素平方项  $\mathbf{g} \odot \mathbf{g}$  做指数加权移动平均，记为  $s$ ：

$$\mathbf{s} \leftarrow \rho \mathbf{s} + (1 - \rho) \mathbf{g} \odot \mathbf{g}.$$

然后，计算当前需要迭代的目标函数自变量的变化量  $\mathbf{g}'$ ：

$$\mathbf{g}' \leftarrow \frac{\sqrt{\Delta \mathbf{x} + \epsilon}}{\sqrt{\mathbf{s} + \epsilon}} \odot \mathbf{g},$$

其中  $\epsilon$  是为了维持数值稳定性而添加的常数，例如  $10^{-5}$ 。和 Adagrad 与 RMSProp 一样，目标函数自变量中每个元素都分别拥有自己的学习率。上式中  $\Delta \mathbf{x}$  初始化为零张量，并记录  $\mathbf{g}'$  按元素平方的指数加权移动平均：

$$\Delta \mathbf{x} \leftarrow \rho \Delta \mathbf{x} + (1 - \rho) \mathbf{g}' \odot \mathbf{g}'.$$

同样地，最后的自变量迭代步骤与小批量随机梯度下降类似：

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{g}'.$$

### 7.10.2 Adadelta 的实现

Adadelta 的实现很简单。我们只需要把上面的数学公式翻译成代码。

```
In [1]: def adadelta(params, sqrs, deltas, rho, batch_size):
    eps_stable = 1e-5
    for param, sqr, delta in zip(params, sqrs, deltas):
        g = param.grad / batch_size
        sqr[:] = rho * sqr + (1 - rho) * g.square()
        cur_delta = ((delta + eps_stable).sqrt() /
                     (sqr + eps_stable).sqrt() * g)
        delta[:] = rho * delta + (1 - rho) * cur_delta * cur_delta
        param[:] -= cur_delta
```

### 7.10.3 实验

首先，导入本节中实验所需的包或模块。

```
In [2]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, nd
        import numpy as np
```

实验中，我们依然以线性回归为例。设数据集的样本数为 1000，我们使用权重  $w$  为  $[2, -3.4]$ ，偏差 “ $b$ ” 为 4.2 的线性回归模型来生成数据集。该模型的平方损失函数即所需优化的目标函数，模型参数即目标函数自变量。

我们把算法中变量  $s$  和  $\Delta x$  初始化为和模型参数形状相同的零张量。

```
In [3]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)

        # 初始化模型参数。
        def init_params():
            w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
            b = nd.zeros(shape=(1,))
            params = [w, b]
            sqrs = []
            deltas = []
            for param in params:
                param.attach_grad()
            # 把算法中基于指数加权移动平均的变量初始化为和参数形状相同的零张量。
            sqrs.append(param.zeros_like())
            deltas.append(param.zeros_like())
            return params, sqrs, deltas
```

优化函数 `optimize` 与 “Adagrad——从零开始” 一节中的类似。

```
In [4]: net = gb.linreg
        loss = gb.squared_loss

        def optimize(batch_size, rho, num_epochs, log_interval):
```

```

[w, b], sqrs, deltas = init_params()
ls = [loss(net(features, w, b), labels).mean().asnumpy()]
for epoch in range(1, num_epochs + 1):
    for batch_i, (X, y) in enumerate(
        gb.data_iter(batch_size, num_examples, features, labels)):
        with autograd.record():
            l = loss(net(X, w, b), y)
            l.backward()
            adadelta([w, b], sqrs, deltas, rho, batch_size)
        if batch_i * batch_size % log_interval == 0:
            ls.append(loss(net(features, w, b), labels).mean().asnumpy())
print('w:', w, '\nb:', b, '\n')
es = np.linspace(0, num_epochs, len(ls), endpoint=True)
gb.semilogy(es, ls, 'epoch', 'loss')

```

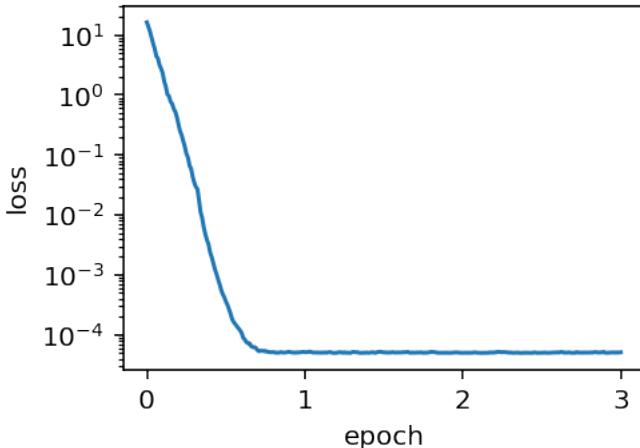
最终，优化所得的模型参数值与它们的真实值较接近。

In [5]: `optimize(batch_size=10, rho=0.9999, num_epochs=3, log_interval=10)`

```

w:
[[ 1.998445 ]
 [-3.40107441]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.20002794]
<NDArray 1 @cpu(0)>

```



#### 7.10.4 小结

- Adadelta 没有学习率参数。

#### 7.10.5 练习

- Adadelta 为什么不需要设置学习率超参数？它被什么代替了？

#### 7.10.6 扫码直达讨论区



#### 7.10.7 参考文献

[1] Zeiler, Matthew D. “ADADELTA: an adaptive learning rate method.” arXiv preprint arXiv:1212.5701 (2012).

### 7.11 Adadelta——使用 Gluon

在 Gluon 里，使用 Adadelta 很容易，我们无需重新实现该算法。

首先，导入本节中实验所需的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import gluon, init, nd
        from mxnet.gluon import nn
```

下面生成实验数据集并定义线性回归模型。

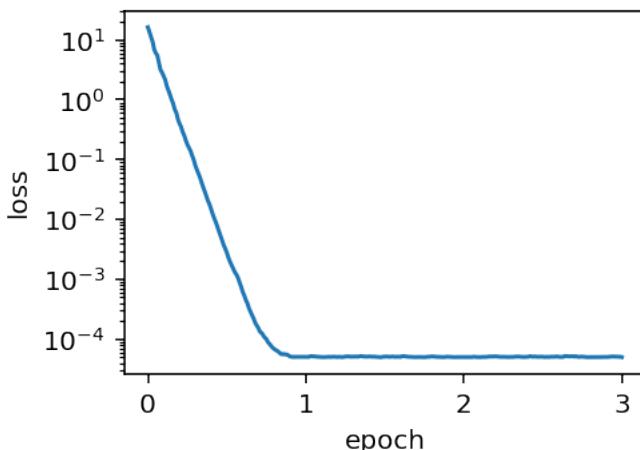
```
In [2]: # 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)

# 线性回归模型。
net = nn.Sequential()
net.add(nn.Dense(1))
```

我们可以在 Trainer 中定义优化算法名称 adadelta 并定义  $\rho$  超参数 rho。以下实验重现了“Adadelta——从零开始”一节中实验结果。

```
In [3]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
trainer = gluon.Trainer(net.collect_params(), 'adadelta', {'rho': 0.9999})
gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=None,
            log_interval=10, features=features, labels=labels, net=net)

w:
[[ 2.00022936 -3.39975238]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20018196]
<NDArray 1 @cpu(0)>
```



### 7.11.1 小结

- 使用 Gluon 的 `Trainer` 可以方便地使用 Adadelta。

### 7.11.2 练习

- 如果把试验中的参数  $\rho$  改小会怎样？观察并分析实验结果。

### 7.11.3 扫码直达讨论区



## 7.12 Adam——从零开始

Adam 是一个组合了动量法和 RMSProp 的优化算法 [1]。下面我们来介绍 Adam 算法。

### 7.12.1 Adam 算法

Adam 算法使用了动量变量  $v$  和 RMSProp 中小批量随机梯度按元素平方的指数加权移动平均变量  $s$ ，并将它们中每个元素初始化为 0。在每次迭代中，时刻  $t$  的小批量随机梯度记作  $\mathbf{g}_t$ 。

和动量法类似，给定超参数  $\beta_1$  且满足  $0 \leq \beta_1 < 1$ （算法作者建议设为 0.9），将小批量随机梯度的指数加权移动平均记作动量变量  $v$ ，并将它在时刻  $t$  的值记作  $\mathbf{v}_t$ ：

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

和 RMSProp 中一样，给定超参数  $\beta_2$  且满足  $0 \leq \beta_2 < 1$ （算法作者建议设为 0.999），将小批量随机梯度按元素平方后做指数加权移动平均得到  $s$ ，并将它在时刻  $t$  的值记作  $\mathbf{s}_t$ ：

$$\mathbf{s}_t \leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t \odot \mathbf{g}_t.$$

由于我们将  $v$  和  $s$  中的元素都初始化为 0, 在时刻  $t$  我们得到  $v_t = (1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} g_i$ 。将过去各时刻小批量随机梯度的权值相加, 得到  $(1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} = 1 - \beta_1^t$ 。需要注意的是, 当  $t$  较小时, 过去各时刻小批量随机梯度权值之和会较小。例如当  $\beta_1 = 0.9$  时,  $v_1 = 0.1g_1$ 。为了消除这样的影响, 对于任意时刻  $t$ , 我们可以将  $v_t$  再除以  $1 - \beta_1^t$ , 从而使得过去各时刻小批量随机梯度权值之和为 1。这也叫做偏差修正。在 Adam 算法中, 我们对变量  $v$  和  $s$  均作偏差修正:

$$\hat{v}_t \leftarrow \frac{v_t}{1 - \beta_1^t},$$

$$\hat{s}_t \leftarrow \frac{s_t}{1 - \beta_2^t}.$$

接下来, Adam 算法使用以上偏差修正后的变量  $\hat{v}_t$  和  $\hat{s}_t$ , 将模型参数中每个元素的学习率通过按元素运算重新调整:

$$g'_t \leftarrow \frac{\eta \hat{v}_t}{\sqrt{\hat{s}_t + \epsilon}},$$

其中  $\eta$  是初始学习率且  $\eta > 0$ ,  $\epsilon$  是为了维持数值稳定性而添加的常数, 例如  $10^{-8}$ 。和 Adagrad、RMSProp 以及 Adadelta 一样, 目标函数自变量中每个元素都分别拥有自己的学习率。

最后, 时刻  $t$  的自变量  $x_t$  的迭代步骤与小批量随机梯度下降类似:

$$x_t \leftarrow x_{t-1} - g'_t.$$

### 7.12.2 Adam 的实现

Adam 的实现很简单。我们只需要把上面的数学公式翻译成代码。

```
In [1]: def adam(params, vs, sqrs, lr, batch_size, t):
    beta1 = 0.9
    beta2 = 0.999
    eps_stable = 1e-8
    for param, v, sqr in zip(params, vs, sqrs):
        g = param.grad / batch_size
        v[:] = beta1 * v + (1 - beta1) * g
        sqr[:] = beta2 * sqr + (1 - beta2) * g.square()
        v_bias_corr = v / (1 - beta1 ** t)
        sqr_bias_corr = sqr / (1 - beta2 ** t)
        param[:] = param - lr * v_bias_corr / (
            sqr_bias_corr.sqrt() + eps_stable)
```

### 7.12.3 实验

首先，导入实验所需的包或模块。

```
In [2]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import autograd, nd
        import numpy as np
```

实验中，我们依然以线性回归为例。设数据集的样本数为 1000，我们使用权重  $w$  为  $[2, -3.4]$ ，偏差 “ $b$ ” 为 4.2 的线性回归模型来生成数据集。该模型的平方损失函数即所需优化的目标函数，模型参数即目标函数自变量。

我们把算法中变量  $v$  和  $s$  初始化为和模型参数形状相同的零张量。

```
In [3]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)

        # 初始化模型参数。
        def init_params():
            w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
            b = nd.zeros(shape=(1,))
            params = [w, b]
            vs = []
            sqrs = []
            for param in params:
                param.attach_grad()
                # 把算法中基于指数加权移动平均的变量初始化为和参数形状相同的零张量。
                vs.append(param.zeros_like())
                sqrs.append(param.zeros_like())
            return params, vs, sqrs
```

优化函数 `optimize` 与 “Adagrad——从零开始” 一节中的类似。

```
In [4]: net = gb.linreg
        loss = gb.squared_loss

        def optimize(batch_size, lr, num_epochs, log_interval):
```

```

[w, b], vs, sqrs = init_params()
ls = [loss(net(features, w, b), labels).mean().asnumpy()]
t = 0
for epoch in range(1, num_epochs + 1):
    for batch_i, (X, y) in enumerate(
        gb.data_iter(batch_size, num_examples, features, labels)):
        with autograd.record():
            l = loss(net(X, w, b), y)
            l.backward()
        # 必须在调用 Adam 前。
        t += 1
        adam([w, b], vs, sqrs, lr, batch_size, t)
        if batch_i * batch_size % log_interval == 0:
            ls.append(loss(net(features, w, b), labels).mean().asnumpy())
print('w:', w, '\nb:', b, '\n')
es = np.linspace(0, num_epochs, len(ls), endpoint=True)
gb.semilogy(es, ls, 'epoch', 'loss')

```

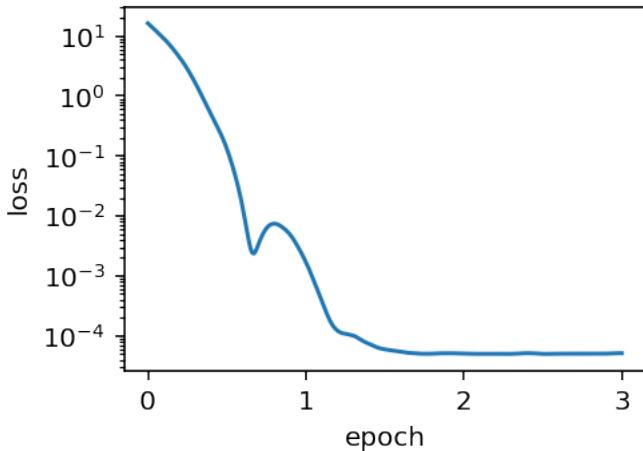
最终，优化所得的模型参数值与它们的真实值较接近。

In [5]: optimize(batch\_size=10, lr=0.1, num\_epochs=3, log\_interval=10)

```

w:
[[ 1.9996376 ]
 [-3.39861751]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.2010231]
<NDArray 1 @cpu(0)>

```



#### 7.12.4 小结

- Adam 组合了动量法和 RMSProp。
- Adam 使用了偏差修正。

#### 7.12.5 练习

- 使用其他初始学习率，观察并分析实验结果。

#### 7.12.6 扫码直达讨论区



#### 7.12.7 参考文献

[1] Kingma, Diederik P., and Jimmy Ba. “Adam: A method for stochastic optimization.” arXiv

preprint arXiv:1412.6980 (2014).

## 7.13 Adam——使用 Gluon

在 Gluon 里，使用 Adadelta 很容易，我们无需重新实现该算法。

首先，导入本节中实验所需的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import gluon, init, nd
        from mxnet.gluon import nn
```

下面生成实验数据集并定义线性回归模型。

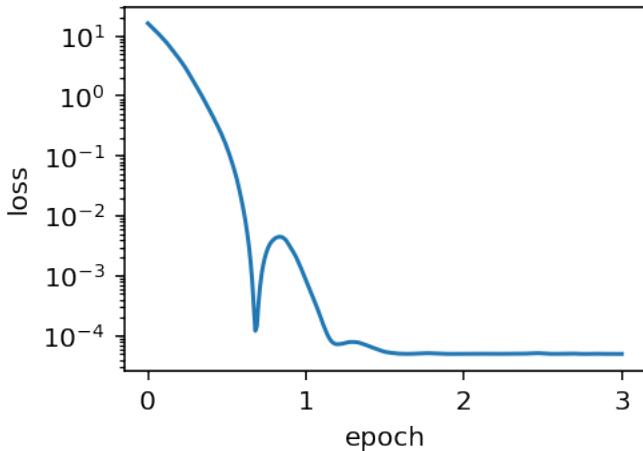
```
In [2]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)

        # 线性回归模型。
        net = nn.Sequential()
        net.add(nn.Dense(1))
```

我们可以在 Trainer 中定义优化算法名称 `adam` 并定义初始学习率。以下实验重现了“Adam——从零开始”一节中实验结果。

```
In [3]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': 0.1})
        gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=None,
                    log_interval=10, features=features, labels=labels, net=net)

w:
[[ 1.99994624 -3.40046382]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20040512]
<NDArray 1 @cpu(0)>
```



### 7.13.1 小结

- 使用 Gluon 的 Trainer 可以方便地使用 Adam。

### 7.13.2 练习

- 总结本章各个优化算法的异同。
- 回顾前面几章中你感兴趣的模型，将训练部分的优化算法替换成其他算法，观察并分析实验现象。

### 7.13.3 扫码直达讨论区



#### 7.13.4 本章回顾

梯度下降可沉甸，随机降低方差难。

引入动量别弯慢，Adagrad 梯方贪。

Adadelta 学率换，RMSProp 梯方权。

Adam 动量 RMS 伴，优化还需己调参。

注释：

- 梯方：梯度按元素平方。
- 贪：因贪婪故而不断累加。
- 学率：学习率。
- 换：这个参数被替换掉。
- 权：指数加权移动平均。



---

## 计算性能

---

无论是当数据集很大还是计算资源或应用有约束条件时，深度学习十分关注计算性能。本章将重点介绍影响计算性能的重要因子：命令式编程、符号式编程、异步计算、自动并行计算和多 GPU 计算。通过本章的学习，你将很可能进一步提升已有模型的计算性能，例如在不影响模型精度的前提下减少模型的训练时间。

### 8.1 命令式和符号式混合编程

其实，到目前为止我们一直都在使用命令式编程：使用编程语句改变程序状态。考虑下面这段简单的命令式编程代码。

```
In [1]: def add(a, b):
        return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
```

```
    return g

fancy_func(1, 2, 3, 4)

Out[1]: 10
```

和我们预期的一样，在运行 `e = add(a, b)` 时，Python 会做加法运算并将结果存储在变量 `e`，从而令程序的状态发生了改变。类似地，后面的两个语句 `f = add(c, d)` 和 `g = add(e, f)` 会依次做加法运算并存储变量。

虽然使用命令式编程很方便，但它的运行可能会慢。一方面，即使 `fancy_func` 函数中的 `add` 是被重复调用的函数，Python 也会逐一执行这三个函数调用语句。另一方面，我们需要保存变量 `e` 和 `f` 的值直到 `fancy_func` 中所有语句执行结束。这是因为在执行 `e = add(a, b)` 和 `f = add(c, d)` 之前我们并不知道变量 `e` 和 `f` 是否会被程序的其他部分使用。

与命令式编程不同，符号式编程通常在计算流程完全定义好后才被执行。大部分的深度学习框架，例如 Theano 和 TensorFlow，都使用了符号式编程。通常，符号式编程的程序需要下面三个步骤：

1. 定义计算流程；
2. 把计算流程编译成可执行的程序；
3. 给定输入，调用编译好的程序执行。

下面我们用符号式编程重新实现本节开头给出的命令式编程代码。

```
In [2]: def add_str():
    return ''

def add(a, b):
    return a + b
'''


def fancy_func_str():
    return ''
def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
'''


def evoke_str():
    return add_str() + fancy_func_str() + ''
print(fancy_func(1, 2, 3, 4))
'''
```

```
prog = evoke_str()
print(prog)
y = compile(prog, '', 'exec')
exec(y)

def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))
```

10

以上定义的三个函数都只是返回计算流程。最后，我们编译完整的计算流程并运行。由于在编译时系统能够完整地看到整个程序，因此有更多空间优化计算。例如，编译的时候可以将程序改写成 `print((1 + 2) + (3 + 4))`，甚至直接改写成 `print(10)`。这样不仅减少了函数调用，还节省了内存。

总结一下，

- 命令式编程更方便。当我们在 Python 里使用命令式编程时，大部分代码编写起来都符合直觉。同时，命令式编程更容易除错。这是因为我们可以很方便地拿到所有的中间变量值并打印，或者使用 Python 的除错工具。
- 符号式编程更高效并更容易移植。一方面，在编译的时候系统可以容易地做更多优化；另一方面，符号式编程可以将程序变成一个与 Python 无关的格式，从而可以使程序在非 Python 环境下运行。

### 8.1.1 混合式编程取两者之长

大部分的深度学习框架在命令式编程和符号式编程之间二选一。例如 Theano 和受其启发的后来者 TensorFlow 使用了符号式编程；Chainer 和它的追随者 PyTorch 使用了命令式编程。开发人员在设计 Gluon 时思考了这个问题：有没有可能既拿到命令式编程的好处，又享受符号式编程的

优势？开发者们认为，用户应该用纯命令式编程进行开发和调试；当需要产品级别的性能和部署时，用户可以将至少大部分程序转换成符号式来运行。

值得强调的是，Gluon 可以通过混合式编程做到这一点。在混合式编程中，我们可以通过使用 HybridBlock 或者 HybridSequential 类构建模型。默认情况下，它们和 Block 或者 Sequential 类一样依据命令式编程的方式执行。当我们调用 `hybridize` 函数后，Gluon 会转换成依据符号式编程的方式执行。事实上，绝大多数模型都可以享受符号式编程的优势。

本节将通过实验展示混合式编程的魅力。首先，导入本节中实验所需的包或模块。

```
In [3]: from mxnet import nd, sym  
      from mxnet.gluon import nn  
      from time import time
```

### 8.1.2 使用 HybridSequential 类构造模型

我们之前学习了如何使用 Sequential 类来串联多个层。为了使用混合式编程，下面我们将 Sequential 类替换成 HybridSequential 类。

```
In [4]: def get_net():  
    net = nn.HybridSequential()  
    net.add(  
        nn.Dense(256, activation="relu"),  
        nn.Dense(128, activation="relu"),  
        nn.Dense(2)  
    )  
    net.initialize()  
    return net  
  
x = nd.random.normal(shape=(1, 512))  
net = get_net()  
net(x)
```

Out[4]:

```
[[ 0.08827581  0.00505182]]  
<NDArray 1x2 @cpu(0)>
```

我们可以通过调用 `hybridize` 函数来编译和优化 HybridSequential 实例中串联的层的计算。模型的计算结果不变。

```
In [5]: net.hybridize()  
net(x)
```

```
Out[5]:  
[[ 0.08827581  0.00505182]]  
<NDArray 1x2 @cpu(0)>
```

需要注意的是，只有继承 HybridBlock 的层才会被优化。例如，HybridSequential 类和 Gluon 提供的 Dense 类都是 HybridBlock 的子类，它们都会被优化计算。如果一个层只是继承自 Block 而不是 HybridBlock 类，那么它将不会被优化。我们接下会讨论如何使用 HybridBlock 类。

## 性能

我们比较调用 hybridize 函数前后的计算时间来展示符号式编程的性能提升。这里我们计时 1000 次 net 模型计算。在 net 调用 hybridize 函数前后，它分别依据命令式编程和符号式编程做模型计算。

```
In [6]: def benchmark(net, x):  
    start = time()  
    for i in range(1000):  
        y = net(x)  
    # 等待所有计算完成。  
    nd.waitall()  
    return time() - start  
  
net = get_net()  
print('Before hybridizing: %.4f sec' % (benchmark(net, x)))  
net.hybridize()  
print('After hybridizing: %.4f sec' % (benchmark(net, x)))
```

Before hybridizing: 0.3258 sec  
After hybridizing: 0.1903 sec

由上面结果可见，在一个 HybridSequential 实例调用 hybridize 函数后，它可以通过符号式编程提升计算性能。

## 获取符号式程序

在模型 net 根据输入计算模型输出后，例如 benchmark 函数中的 net(x)，我们就可以通过 export 函数来保存符号式程序和模型参数到硬盘。

```
In [7]: net.export('my_mlp')
```

此时生成的.json 和.params 文件分别为符号式程序和模型参数。它们可以被 Python 或 MXNet 支持的其他前端语言读取，例如 C++。这样，我们就可以很方便地使用其他前端语言或在其他设

备上部署训练好的模型。同时，由于部署时使用的是基于符号式编程的程序，计算性能往往比基于命令式编程更好。

在 MXNet 中，符号式程序指的是 Symbol 类型的程序。我们知道，当给 net 提供 NDArray 类型的输入 x 后，net(x) 会根据 x 直接计算模型输出并返回结果。对于调用过 hybridize 函数后的模型，我们还可以给它输入一个 Symbol 类型的变量，net(x) 会返回同样是 Symbol 类型的程序。

```
In [8]: x = sym.var('data')
net(x)
```

```
Out[8]: <Symbol dense5_fwd>
```

### 8.1.3 使用 HybridBlock 类构造模型

和 Sequential 类与 Block 之间的关系一样，HybridSequential 类是 HybridBlock 的子类。跟 Block 实例需要实现 forward 函数不太一样的是，对于 HybridBlock 实例我们需要实现 hybrid\_forward 函数。

前面我们展示了调用 hybridize 函数后的模型可以获得更好的计算性能和移植性。另一方面，调用 hybridize 后的模型会影响灵活性。为了解释这一点，我们先使用 HybridBlock 构造模型。

```
In [9]: class HybridNet(nn.HybridBlock):
    def __init__(self, **kwargs):
        super(HybridNet, self).__init__(**kwargs)
        self.hidden = nn.Dense(10)
        self.output = nn.Dense(2)

    def hybrid_forward(self, F, x):
        print('F: ', F)
        print('x: ', x)
        x = F.relu(self.hidden(x))
        print('hidden: ', x)
        return self.output(x)
```

在继承 HybridBlock 类时，我们需要在 hybrid\_forward 函数中添加额外的输入 F。我们知道，MXNet 既有基于命令式编程的 NDArray 类，又有基于符号式编程的 Symbol 类。由于这两个类的函数基本一致，MXNet 会根据输入来决定 F 使用 NDArray 或 Symbol。

下面创建了一个 HybridBlock 实例。可以看到默认下 F 使用 NDArray。而且，我们打印出了输入 x 和使用 ReLU 激活函数的隐藏层的输出。

```
In [10]: net = HybridNet()
        net.initialize()
        x = nd.random.normal(shape=(1, 4))
        net(x)

F:  <module 'mxnet.ndarray' from '/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/py
    ↵  python3.6/site-packages/mxnet/ndarray/__init__.py'>
x:
[[ -0.12225834  0.5429998  -0.94693518  0.59643304]]
<NDArray 1x4 @cpu(0)>
hidden:
[[ 0.11134676  0.04770704  0.05341475  0.          0.08091211  0.          0.
  0.04143535  0.          0.          ]]
<NDArray 1x10 @cpu(0)>

Out[10]:
[[ 0.00370749  0.00134991]]
<NDArray 1x2 @cpu(0)>
```

再运行一次会得到同样的结果。

```
In [11]: net(x)

F:  <module 'mxnet.ndarray' from '/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/py
    ↵  python3.6/site-packages/mxnet/ndarray/__init__.py'>
x:
[[ -0.12225834  0.5429998  -0.94693518  0.59643304]]
<NDArray 1x4 @cpu(0)>
hidden:
[[ 0.11134676  0.04770704  0.05341475  0.          0.08091211  0.          0.
  0.04143535  0.          0.          ]]
<NDArray 1x10 @cpu(0)>

Out[11]:
[[ 0.00370749  0.00134991]]
<NDArray 1x2 @cpu(0)>
```

接下来看看调用 `hybridize` 函数后会发生什么。

```
In [12]: net.hybridize()
        net(x)

F:  <module 'mxnet.symbol' from '/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/py
    ↵  thon3.6/site-packages/mxnet/symbol/__init__.py'>
x:  <Symbol data>
hidden: <Symbol hybridnet0_relu0>
```

```
Out[12]:
```

```
[[ 0.00370749  0.00134991]]  
<NDArray 1x2 @cpu(0)>
```

可以看到，`F` 变成了 `Symbol`。而且，虽然输入数据还是 `NDArray`，但 `hybrid_forward` 函数里，相同输入和中间输出全部变成了 `Symbol`。

再运行一次看看。

```
In [13]: net(x)
```

```
Out[13]:
```

```
[[ 0.00370749  0.00134991]]  
<NDArray 1x2 @cpu(0)>
```

可以看到 `hybrid_forward` 函数里定义的三行打印语句都没有打印任何东西。这是因为上一次在调用 `hybridize` 函数后运行 `net(x)` 的时候，符号式程序已经得到。之后再运行 `net(x)` 的时候 MXNet 将不再访问 Python 代码，而是直接在 C++ 后端执行符号式程序。这也是调用 `hybridize` 后模型计算性能会提升的一个原因。但它可能的问题是我们损失了写程序的灵活性。在上面这个例子中，如果我们希望使用那三行打印语句调试代码，执行符号式程序时会跳过它们无法打印。此外，对于少数 `Symbol` 不支持的函数，例如 `asnumpy`，我们是无法在 `hybrid_forward` 函数中使用并在调用 `hybridize` 函数后进行模型计算的。

## 8.1.4 小结

- 命令式编程和符号式编程各有优劣。MXNet 通过混合式编程取两者之长。
- 通过 `HybridSequential` 类和 `HybridBlock` 构建的模型可以调用 `hybridize` 来将命令式程序转成符号式程序。我们建议大家使用这种方法获得计算性能的提升。

## 8.1.5 练习

- 在本节 `HybridNet` 类 `hybrid_forward` 函数中第一行添加 `x.asnumpy()`，运行本节全部代码，观察报错的位置和错误类型。
- 回顾前面几章中你感兴趣的模型，改用 `HybridBlock` 或 `HybridSequential` 类实现。

### 8.1.6 扫码直达讨论区



## 8.2 异步计算

MXNet 使用异步计算来提升计算性能。理解它的工作原理既有助于开发更高效的程序，又有助于在内存资源有限的情况下主动降低计算性能从而减小内存开销。

我们先导入本节中实验需要的包或模块。

```
In [1]: from mxnet import autograd, gluon, nd
        from mxnet.gluon import loss as gloss, nn
        import os
        import subprocess
        from time import time
```

### 8.2.1 MXNet 中的异步计算

广义上，MXNet 包括用户直接用来交互的前端和系统用来执行计算的后端。例如，用户可以使用不同的前端语言编写 MXNet 程序，像 Python、R、Scala 和 C++。无论使用何种前端编程语言，MXNet 程序的执行主要都发生在 C++ 实现的后端。换句话说，用户写好的前端 MXNet 程序会传给后端执行计算。后端有自己的线程在队列中不断收集任务并执行它们。

MXNet 通过前端线程和后端线程的交互实现异步计算。异步计算指，前端线程无需等待当前指令从后端线程返回结果就继续执行后面的指令。为了便于解释，假设 Python 前端线程调用以下四条指令。

```
In [2]: a = nd.ones((1, 2))
        b = nd.ones((1, 2))
        c = a * b + 2
        c
```

Out[2]:

```
[[ 3.  3.]]  
<NDArray 1x2 @cpu(0)>
```

在异步计算中，Python 前端线程执行前三条语句的时候，仅仅是把任务放进后端的队列里就返回了。当最后一条语句需要打印计算结果时，Python 前端线程会等待 C++ 后端线程把 c 的结果计算完。此设计的一个好处是，这里的 Python 前端线程不需要做实际计算。因此，无论 Python 的性能如何，它对整个程序性能的影响很小。只要 C++ 后端足够高效，那么不管前端语言性能如何，MXNet 都可以提供一致的高性能。

下面的例子通过计时来展示异步计算的效果。可以看到，当 `y = nd.dot(x, x)` 返回的时候并没有等待它真正被计算完。

```
In [3]: start = time()  
x = nd.random.uniform(shape=(2000, 2000))  
y = nd.dot(x, x)  
print('workloads are queued: %f sec' % (time() - start))  
print(y)  
print('workloads are completed: %f sec' % (time() - start))  
  
workloads are queued: 0.000519 sec  
  
[[ 501.15838623  508.29724121  495.65237427 ... ,  492.8470459   492.69091797  
  490.0480957 ]]  
[ 508.81057739  507.18218994  495.17428589 ... ,  503.10525513  
  497.29315186  493.6791687 ]]  
[ 489.565979    499.47015381  490.17721558 ... ,  490.99945068  
  488.05007935  483.2883606 ]]  
...,  
[ 484.00189209  495.71789551  479.92141724 ... ,  493.69952393  
  478.89193726  487.20739746 ]]  
[ 499.64932251  507.65093994  497.59381104 ... ,  493.0473938   500.74511719  
  495.82711792 ]]  
[ 516.01428223  519.17150879  506.35400391 ... ,  510.08877563  496.3560791  
  495.42523193 ]]  
<NDArray 2000x2000 @cpu(0)>  
workloads are completed: 0.149492 sec
```

的确，除非我们需要打印或者保存计算结果，我们基本无需关心目前结果在内存中是否已经计算好了。只要数据是保存在 NDArray 里并使用 MXNet 提供的运算符，MXNet 将默认使用异步计算来获取高计算性能。

## 8.2.2 用同步函数让前端等待计算结果

除了前面介绍的 `print` 外，我们还有其他方法让前端线程等待后端的计算结果完成。我们可以使用 `wait_to_read` 函数让前端等待某个的 `NDArray` 的计算结果完成，再执行前端中后面的语句。或者，我们可以用 `waitall` 函数令前端等待前面所有计算结果完成。后者是性能测试中常用的方法。

下面是使用 `wait_to_read` 的例子。输出用时包含了 `y` 的计算时间。

```
In [4]: start = time()
y = nd.dot(x, x)
y.wait_to_read()
time() - start
Out[4]: 0.12300324440002441
```

下面是使用 `waitall` 的例子。输出用时包含了 `y` 和 `z` 的计算时间。

```
In [5]: start = time()
y = nd.dot(x, x)
z = nd.dot(x, x)
nd.waitall()
time() - start
Out[5]: 0.24203109741210938
```

此外，任何将 `NDArray` 转换成其他不支持异步计算的数据结构的操作都会让前端等待计算结果。例如当我们调用 `asnumpy` 和 `asscalar` 函数时：

```
In [6]: start = time()
y = nd.dot(x, x)
y.asnumpy()
time() - start
Out[6]: 0.129072904586792
In [7]: start = time()
y = nd.dot(x, x)
y.norm().asscalar()
time() - start
Out[7]: 0.15854883193969727
```

上面介绍的 `wait_to_read`、`waitall`、`asnumpy`、`asscalar` 和 `print` 函数会触发让前端等待后端计算结果的行为，我们通常把这类函数称作同步函数。

### 8.2.3 使用异步计算提升计算性能

在下面例子中，我们用 for 循环不断对 `y` 赋值。当 for 循环内使用同步函数 `wait_to_read` 时，每次赋值不使用异步计算；当 for 循环外使用同步函数 `waitall` 时，则使用异步计算。

```
In [8]: start = time()
for _ in range(1000):
    y = x + 1
    y.wait_to_read()
print('synchronous: %f sec' % (time() - start))

start = time()
for _ in range(1000):
    y = x + 1
nd.waitall()
print('asynchronous: %f sec' % (time() - start))

synchronous: 1.042080 sec
asynchronous: 0.701175 sec
```

我们观察到，使用异步计算能提升一定的计算性能。为了解释这个现象，让我们对 Python 前端线程和 C++ 后端线程的交互稍作简化。在每一次循环中，前端和后端的交互大约可以分为三个阶段：

1. 前端令后端将计算任务 `y = x + 1` 放进队列；
2. 后端从队列中获取计算任务并执行真正的计算；
3. 后端将计算结果返回给前端。

我们将这三个阶段的耗时分别设为  $t_1, t_2, t_3$ 。如果不使用异步计算，执行 1000 次计算的总耗时大约为  $1000(t_1 + t_2 + t_3)$ ；如果使用异步计算，由于每次循环前端都无需等待后端返回计算结果，执行 1000 次计算的总耗时可以降为  $t_1 + 1000t_2 + t_3$ （假设  $1000t_2 > 999t_1$ ）。

### 8.2.4 异步计算对内存使用的影响

为了解释异步计算对内存使用的影响，让我们先回忆一下前面章节的内容。

在前面章节中实现的模型训练过程中，我们通常会在每个小批量上评测一下模型，例如模型的损失或者精度。细心的你也许发现了，这类评测常用到同步函数，例如 `asscalar` 或者 `asnumpy`。如果去掉这些同步函数，前端会将大量的小批量计算任务在极短的时间内丢给后端，从而可能导

致较大的内存开销。当我们在每个小批量上都使用同步函数时，前端在每次迭代时仅会将一个小批量的任务丢给后端执行计算，并通常会减小内存开销。

由于深度学习模型通常比较大，而内存资源通常有限，我们建议大家在训练模型时对每个小批量都使用同步函数，例如用 `asscalar` 或者 `asnumpy` 评价模型的表现。类似地，在使用模型预测时，为了减小内存开销，我们也建议大家对每个小批量预测时都使用同步函数，例如直接打印出当前小批量的预测结果。

下面我们来演示异步计算对内存使用的影响。我们先定义一个数据获取函数，它会从被调用时开始计时，并定期打印到目前为止获取数据批量总共耗时。

```
In [9]: num_batches = 41
def data_iter():
    start = time()
    batch_size = 1024
    for i in range(num_batches):
        if i % 10 == 0:
            print('batch %d, time %f sec' % (i, time() - start))
        X = nd.random.normal(shape=(batch_size, 512))
        y = nd.ones((batch_size,))
        yield X, y
```

以下定义多层感知机、优化器和损失函数。

```
In [10]: net = nn.Sequential()
net.add(
    nn.Dense(2048, activation='relu'),
    nn.Dense(512, activation='relu'),
    nn.Dense(1),
)
net.initialize()
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'learning_rate':0.005})
loss = gloss.L2Loss()
```

这里定义辅助函数来监测内存的使用。需要注意的是，这个函数只能在 Linux 或 MacOS 运行。

```
In [11]: def get_mem():
    res = subprocess.check_output(['ps', 'u', '-p', str(os.getpid())])
    return int(str(res).split()[15]) / 1e3
```

现在我们可以做测试了。我们先试运行一次让系统把 `net` 的参数初始化。相关内容请参见“模型参数的延后初始化”一节。

```
In [12]: for X, y in data_iter():
```

```

break
loss(y, net(X)).wait_to_read()

batch 0, time 0.000002 sec

对于训练 net 来说, 我们可以自然地使用同步函数 asscalar 将每个小批量的损失从 NDArray
格式中取出, 并打印每个迭代周期后的模型损失。此时, 每个小批量的生成间隔较长, 不过内存
开销较小。

In [13]: mem = get_mem()
for epoch in range(1, 3):
    l_sum = 0
    for X, y in data_iter():
        with autograd.record():
            l = loss(y, net(X))
            l_sum += l.mean().asscalar()
            l.backward()
            trainer.step(X.shape[0])
        print('epoch', epoch, ' loss:', l_sum / num_batches)
    nd.waitall()
    print('increased memory: %f MB' % (get_mem() - mem))

batch 0, time 0.000003 sec
batch 10, time 1.340160 sec
batch 20, time 2.759166 sec
batch 30, time 4.182007 sec
batch 40, time 5.604554 sec
epoch 1 loss: 0.15615208556
batch 0, time 0.000003 sec
batch 10, time 1.418581 sec
batch 20, time 2.834151 sec
batch 30, time 4.248881 sec
batch 40, time 5.663722 sec
epoch 2 loss: 0.0985021460347
increased memory: 6.776000 MB

```

如果去掉同步函数, 虽然每个小批量的生成间隔较短, 训练过程中可能会导致内存开销过大。这是因为默认异步计算下, 前端会将所有小批量计算在短时间内全部丢给后端。

```

In [14]: mem = get_mem()
for epoch in range(1, 3):
    for X, y in data_iter():
        with autograd.record():
            l = loss(y, net(X))
            l.backward()

```

```
        trainer.step(x.shape[0])
        nd.waitall()
        print('increased memory: %f MB' % (get_mem() - mem))

batch 0, time 0.000002 sec
batch 10, time 0.016301 sec
batch 20, time 0.030612 sec
batch 30, time 0.044745 sec
batch 40, time 0.058716 sec
batch 0, time 0.000003 sec
batch 10, time 0.013973 sec
batch 20, time 0.027900 sec
batch 30, time 0.042564 sec
batch 40, time 0.056754 sec
increased memory: 136.020000 MB
```

## 8.2.5 小结

- MXNet 包括用户直接用来交互的前端和系统用来执行计算的后端。
- MXNet 能够通过异步计算提升计算性能。
- 我们建议使用每个小批量训练或预测时至少使用一个同步函数，从而避免在短时间内将过多计算任务丢给后端。

## 8.2.6 练习

- 在“使用异步计算提升计算性能”一节中，我们提到使用异步计算可以使执行 1000 次计算的总耗时可以降为  $t_1 + 1000t_2 + t_3$ 。这里为什么要假设  $1000t_2 > 999t_1$ ？

## 8.2.7 扫码直达讨论区



## 8.3 自动并行计算

在“异步计算”一节里我们提到 MXNet 后端会自动构建计算图。通过计算图，系统可以知道所有计算的依赖关系，并可以选择将没有依赖关系的多个任务并行执行来获得性能的提升。以“异步计算”一节中的计算图（图 8.1）为例。其中 `a = nd.ones((1, 2))` 和 `b = nd.ones((1, 2))` 这两步计算之间并没有依赖关系。因此，系统可以选择并行执行它们。

通常一个运算符会用掉一个 CPU/GPU 上所有计算资源。例如，`dot` 操作符会用到所有 CPU（即使是有多个 CPU）或单个 GPU 上所有线程。因此在单 CPU/GPU 上并行运行多个运算符可能效果并不明显。本节中探讨的自动并行计算主要关注 CPU 和 GPU 的并行计算，以及计算和通讯的并行。

首先导入本节中实验所需的包或模块。注意，我们需要至少一个 GPU 才能运行本节实验。

```
In [1]: import mxnet as mx  
from mxnet import nd  
from time import time
```

### 8.3.1 CPU 和 GPU 的并行计算

我们先介绍 CPU 和 GPU 的并行计算，例如程序中的计算既发生在 CPU，又发生在 GPU 之上。

先定义一个函数，令它做 10 次矩阵乘法。

```
In [2]: def run(x):  
    return [nd.dot(x, x) for _ in range(10)]
```

接下来，分别在 CPU 和 GPU 上创建 NDArray。

```
In [3]: x_cpu = nd.random.uniform(shape=(2000, 2000))  
x_gpu = nd.random.uniform(shape=(6000, 6000), ctx=mx.gpu(0))
```

然后，分别使用它们在 CPU 和 GPU 上运行 `run` 函数并打印所需时间。

```
In [4]: run(x_cpu) # 预热开始。  
run(x_gpu)  
nd.waitall() # 预热结束。  
  
start = time()  
run(x_cpu)  
nd.waitall()  
print('run on CPU: %f sec' % (time()-start))
```

```
start = time()
run(x_gpu)
nd.waitall()
print('run on GPU: %f sec' % (time()-start))

run on CPU: 1.219543 sec
run on GPU: 1.207776 sec
```

我们去掉 `run(x_cpu)` 和 `run(x_gpu)` 两个计算任务之间的 `nd.waitall()`, 希望系统能自动并行这两个任务。

```
In [5]: start = time()
run(x_cpu)
run(x_gpu)
nd.waitall()
print('run on both CPU and GPU: %f sec' % (time()-start))

run on both CPU and GPU: 1.221883 sec
```

可以看到, 当两个计算任务一起执行时, 执行总时间小于它们分开执行的总和。这表示, MXNet 能有效地在 CPU 和 GPU 上自动并行计算。

### 8.3.2 计算和通讯的并行计算

在多 CPU/GPU 计算中, 我们经常需要在 CPU/GPU 之间复制数据, 造成数据的通讯。举个例子, 在下面例子中, 我们在 GPU 上计算, 然后将结果复制回 CPU。我们分别打印 GPU 上计算时间和 GPU 到 CPU 的通讯时间。

```
In [6]: def copy_to_cpu(x):
    return [y.copyto(mx.cpu()) for y in x]

    start = time()
    y = run(x_gpu)
    nd.waitall()
    print('run on GPU: %f sec' % (time() - start))

    start = time()
    copy_to_cpu(y)
    nd.waitall()
    print('copy to CPU: %f sec' % (time() - start))

run on GPU: 1.219378 sec
copy to CPU: 0.511516 sec
```

我们去掉计算和通讯之间的 `waitall` 函数，打印这两个任务完成的总时间。

```
In [7]: start = time()
y = run(x_gpu)
copy_to_cpu(y)
nd.waitall()
t = time() - start
print('run on GPU then copy to CPU: %f sec' % (time() - start))

run on GPU then copy to CPU: 1.262505 sec
```

可以看到，执行计算和通讯的总时间小于两者分别执行的耗时之和。需要注意的是，这个计算并通讯的任务不同于前面多 CPU/GPU 的并行计算中的任务。这里的运行和通讯之间有依赖关系： $y[i]$  必须先计算好才能复制到 CPU。所幸的是，在计算  $y[i]$  的时候系统可以复制  $y[i-1]$ ，从而减少计算和通讯的总运行时间。

### 8.3.3 小结

- MXNet 能够通过自动并行计算提升计算性能，例如 CPU 和 GPU 的并行以及计算和通讯的并行。

### 8.3.4 练习

- 本节中定义的 `run` 函数里做了 10 次运算。它们之间也没有依赖关系。看看 MXNet 有没有自动并行执行它们。
- 试试包含更加复杂的数据依赖的计算任务。MXNet 能不能得到正确结果并提升计算性能？

### 8.3.5 扫码直达讨论区



## 8.4 多 GPU 计算——从零开始

本教程我们将展示如何使用多个 GPU 计算，例如使用多个 GPU 训练模型。正如你期望的那样，运行本节中的程序需要至少两块 GPU。事实上，一台机器上安装多块 GPU 非常常见。这是因为主板上通常会有多个 PCIe 插槽。如果正确安装了 NVIDIA 驱动，我们可以通过 `nvidia-smi` 命令来查看当前机器上的全部 GPU。

```
In [1]: !nvidia-smi
```

```
Mon Jun  4 21:43:02 2018
```

NVIDIA-SMI 375.26				Driver Version: 375.26		
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr. ECC
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.
<hr/>						
0	Tesla M60	On	0000:00:1D.0	Off	0%	Default
N/A	46C	P0	39W / 150W	298MiB / 7612MiB	0%	Default
<hr/>						
1	Tesla M60	On	0000:00:1E.0	Off	0%	Default
N/A	57C	P0	39W / 150W	289MiB / 7612MiB	0%	Default
<hr/>						
Processes:				GPU Memory		
GPU	PID	Type	Process name	Usage		
<hr/>						

在“[自动并行计算](#)”一节里，我们介绍过，大部分的运算可以使用所有的 CPU 的全部计算资源，或者单个 GPU 的全部计算资源。但如果使用多个 GPU 训练模型，我们仍然需要实现相应的算法。这些算法中最常用的叫做数据并行。

### 8.4.1 数据并行

数据并行目前是深度学习里使用最广泛的将模型训练任务划分到多个 GPU 的办法。回忆一下我们在“[梯度下降和随机梯度下降——从零开始](#)”一节中介绍的使用优化算法训练模型的过程。下面我们就以小批量随机梯度下降为例来介绍数据并行是如何工作的。

假设一台机器上有  $k$  个 GPU。给定需要训练的模型，每个 GPU 将分别独立维护一份完整的模型

参数。在模型训练的任意一次迭代中，给定一个小批量，我们将该批量中的样本划分成  $k$  份并分给每个 GPU 一份。然后，每个 GPU 将分别根据自己分到的训练数据样本和自己维护的模型参数计算模型参数的梯度。接下来，我们把  $k$  个 GPU 上分别计算得到的梯度相加，从而得到当前的小批量梯度。之后，每个 GPU 都使用这个小批量梯度分别更新自己维护的那一份完整的模型参数。

为了从零开始实现多 GPU 训练中的数据并行，让我们先导入需要的包或模块。

```
In [2]: import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
        from mxnet import autograd, nd
        from mxnet.gluon import loss as gloss
        from time import time
```

## 8.4.2 定义模型

我们使用“卷积神经网络——从零开始”一节里介绍的 LeNet 来作为本节的样例模型。

```
In [3]: # 初始化模型参数。
        scale = 0.01
        W1 = nd.random.normal(scale=scale, shape=(20, 1, 3, 3))
        b1 = nd.zeros(shape=20)
        W2 = nd.random.normal(scale=scale, shape=(50, 20, 5, 5))
        b2 = nd.zeros(shape=50)
        W3 = nd.random.normal(scale=scale, shape=(800, 128))
        b3 = nd.zeros(shape=128)
        W4 = nd.random.normal(scale=scale, shape=(128, 10))
        b4 = nd.zeros(shape=10)
        params = [W1, b1, W2, b2, W3, b3, W4, b4]

        # 定义模型。
        def lenet(X, params):
            h1_conv = nd.Convolution(data=X, weight=params[0], bias=params[1],
                                    kernel=(3, 3), num_filter=20)
            h1_activation = nd.relu(h1_conv)
            h1 = nd.Pooling(data=h1_activation, pool_type="avg", kernel=(2, 2),
                            stride=(2, 2))
            h2_conv = nd.Convolution(data=h1, weight=params[2], bias=params[3],
                                    kernel=(5, 5), num_filter=50)
            h2_activation = nd.relu(h2_conv)
            h2 = nd.Pooling(data=h2_activation, pool_type="avg", kernel=(2, 2),
                            stride=(2, 2))
```

```

        h2 = nd.flatten(h2)
        h3_linear = nd.dot(h2, params[4]) + params[5]
        h3 = nd.relu(h3_linear)
        y_hat = nd.dot(h3, params[6]) + params[7]
        return y_hat

# 交叉熵损失函数。
loss = gloss.SoftmaxCrossEntropyLoss()

```

### 8.4.3 多 GPU 之间同步数据

我们需要实现一些多 GPU 之间同步数据的辅助函数。下面函数将模型参数复制到某个特定 GPU 并初始化梯度。

```
In [4]: def get_params(params, ctx):
    new_params = [p.copyto(ctx) for p in params]
    for p in new_params:
        p.attach_grad()
    return new_params
```

试一试把 `params` 复制到 `mx.gpu(0)` 上。

```
In [5]: new_params = get_params(params, mx.gpu(0))
print('b1 weight:', new_params[1])
print('b1 grad:', new_params[1].grad)

b1 weight:
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
 0.  0.]
<NDArray 20 @gpu(0)>
b1 grad:
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
 0.  0.]
<NDArray 20 @gpu(0)>
```

给定分布在多个 GPU 之间的数据。以下函数可以把各个 GPU 上的数据加起来，然后再广播到所有 GPU 上。

```
In [6]: def allreduce(data):
    for i in range(1, len(data)):
        data[0][:] += data[i].copyto(data[0].context)
    for i in range(1, len(data)):
        data[0].copyto(data[i])
```

简单测试一下 allreduce 函数。

```
In [7]: data = [nd.ones((1,2), ctx=mx.gpu(i)) * (i + 1) for i in range(2)]
    print('before allreduce:', data)
    allreduce(data)
    print('after allreduce:', data)

before allreduce: [
[[ 1.  1.]]
<NDArray 1x2 @gpu(0)>,
[[ 2.  2.]]
<NDArray 1x2 @gpu(1)>]
after allreduce: [
[[ 3.  3.]]
<NDArray 1x2 @gpu(0)>,
[[ 3.  3.]]
<NDArray 1x2 @gpu(1)>]
```

给定一个批量的数据样本，以下函数可以划分它们并复制到各个 GPU 上。

```
In [8]: def split_and_load(data, ctx):
    n, k = data.shape[0], len(ctx)
    m = n // k
    assert m * k == n, '# examples is not divided by # devices.'
    return [data[i * m: (i + 1) * m].as_in_context(ctx[i]) for i in range(k)]
```

让我们试着用 split\_and\_load 函数将 6 个数据样本平均分给 2 个 GPU。

```
In [9]: batch = nd.arange(24).reshape((6, 4))
    ctx = [mx.gpu(0), mx.gpu(1)]
    splitted = split_and_load(batch, ctx)
    print('input: ', batch)
    print('load into', ctx)
    print('output:', splitted)

input:
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9.  10. 11.]
 [ 12. 13. 14. 15.]
 [ 16. 17. 18. 19.]
 [ 20. 21. 22. 23.]]
<NDArray 6x4 @cpu(0)>
load into [gpu(0), gpu(1)]
output:
[[ 0.  1.  2.  3.]]
```

```
[ 4.  5.  6.  7.]  
[ 8.  9. 10. 11.]]  
<NDArray 3x4 @gpu(0)>,  
[[ 12. 13. 14. 15.]  
[ 16. 17. 18. 19.]  
[ 20. 21. 22. 23.]]  
<NDArray 3x4 @gpu(1)>]
```

#### 8.4.4 单个小批量上的多 GPU 训练

现在我们可以实现单个小批量上的多 GPU 训练了。它的实现主要依据本节介绍的数据并行方法。我们将使用刚刚定义的多 GPU 之间同步数据的辅助函数，例如 `split_and_load` 和 `allreduce`。

```
In [10]: def train_batch(X, y, gpu_params, ctx, lr):  
    # 划分小批量数据样本并复制到各个 GPU 上。  
    gpu_Xs = split_and_load(X, ctx)  
    gpu_ys = split_and_load(y, ctx)  
    # 在各个 GPU 上计算损失。  
    with autograd.record():  
        ls = [loss(lenet(gpu_X, gpu_W), gpu_y)  
              for gpu_X, gpu_y, gpu_W in zip(gpu_Xs, gpu_ys, gpu_params)]  
    # 在各个 GPU 上反向传播。  
    for l in ls:  
        l.backward()  
    # 把各个 GPU 上的梯度加起来，然后再广播到所有 GPU 上。  
    for i in range(len(gpu_params[0])):  
        allreduce([gpu_params[c][i].grad for c in range(len(ctx))])  
    # 在各个 GPU 上更新自己维护的那一份完整的模型参数。  
    for param in gpu_params:  
        gb.sgd(param, lr, X.shape[0])
```

#### 8.4.5 训练函数

现在我们可以定义训练函数。这里的训练函数和之前章节里的训练函数稍有不同。例如，在这里我们需要依据本节介绍的数据并行，将完整的模型参数复制到多个 GPU 上，并在每次迭代时对单个小批量上进行多 GPU 训练。

```
In [11]: def train(num_gpus, batch_size, lr):  
    train_iter, test_iter = gb.load_data_fashion_mnist(batch_size)  
    ctx = [mx.gpu(i) for i in range(num_gpus)]
```

```
print('running on:', ctx)
# 将模型参数复制到 num_gpus 个 GPU 上。
gpu_params = [get_params(params, c) for c in ctx]
for epoch in range(1, 6):
    start = time()
    for X, y in train_iter:
        # 对单个小批量上进行多 GPU 训练。
        train_batch(X, y, gpu_params, ctx, lr)
    nd.waitall()
    print('epoch %d, time: %.1f sec' % (epoch, time() - start))
    # 在 GPU0 上验证模型。
    net = lambda x: lenet(x, gpu_params[0])
    test_acc = gb.evaluate_accuracy(test_iter, net, ctx[0])
    print('validation accuracy: %.4f' % test_acc)
```

我们先使用一个 GPU 来训练。

```
In [12]: train(num_gpus=1, batch_size=256, lr=0.3)

running on: [gpu(0)]
epoch 1, time: 2.2 sec
validation accuracy: 0.2641
epoch 2, time: 1.8 sec
validation accuracy: 0.7129
epoch 3, time: 1.8 sec
validation accuracy: 0.8019
epoch 4, time: 1.8 sec
validation accuracy: 0.7988
epoch 5, time: 1.8 sec
validation accuracy: 0.8337
```

接下来，我们先使用 2 个 GPU 来训练。我们将批量大小也增加一倍，以使得 GPU 的计算资源能够得到较充分利用。

```
In [13]: train(num_gpus=2, batch_size=512, lr=0.3)

running on: [gpu(0), gpu(1)]
epoch 1, time: 1.2 sec
validation accuracy: 0.1467
epoch 2, time: 1.0 sec
validation accuracy: 0.2858
epoch 3, time: 1.0 sec
validation accuracy: 0.7265
epoch 4, time: 1.0 sec
validation accuracy: 0.7453
epoch 5, time: 1.0 sec
```

```
validation accuracy: 0.7964
```

由于批量大小增加了一倍，每个迭代周期的迭代次数减小了一半。因此，我们观察到每个迭代周期的耗时比单 GPU 训练时少了近一半。但由于总体迭代次数的减少，模型在验证数据集上的精度略有下降。这很可能是由于训练不够充分造成的。因此，多 GPU 训练时，我们可以适当增加迭代周期使训练较充分。

#### 8.4.6 小结

- 我们可以使用数据并行更充分地利用多个 GPU 的计算资源，实现多 GPU 训练模型。

#### 8.4.7 练习

- 在本节实验中，试一试不同的迭代周期、批量大小和学习率。
- 将本节实验的模型预测部分改为用多 GPU 预测。

#### 8.4.8 扫码直达讨论区



### 8.5 多 GPU 计算——使用 Gluon

在 Gluon 中，我们可以很方便地使用数据并行进行多 GPU 计算。比方说，我们并不需要自己实现“[多 GPU 计算——从零开始](#)”一节里介绍的多 GPU 之间同步数据的辅助函数。

先导入本节实验需要的包或模块。同上一节，运行本节中的程序需要至少两块 GPU。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
```

```
from mxnet import autograd, gluon, init, nd
from mxnet.gluon import loss as gloss, utils as gutils
from time import time
```

### 8.5.1 多 GPU 上初始化模型参数

我们使用 ResNet-18 来作为本节的样例模型。

```
In [2]: net = gb.resnet18(10)
```

之前我们介绍了如何使用 `initialize` 函数的 `ctx` 参数在 CPU 或单个 GPU 上初始化模型参数。事实上，`ctx` 可以接受一系列的 CPU/GPU，从而使初始化好的模型参数复制到 `ctx` 里所有的 CPU/GPU 上。

```
In [3]: ctx = [mx.gpu(0), mx.gpu(1)]
net.initialize(init=init.Normal(sigma=0.01), ctx=ctx)
```

Gluon 提供了上一节中实现的 `split_and_load` 函数。它可以划分一个小批量的数据样本并复制到各个 CPU/GPU 上。之后，根据输入数据所在的 CPU/GPU，模型计算会发生在相同的 CPU/GPU 上。

```
In [4]: x = nd.random.uniform(shape=(4, 1, 28, 28))
gpu_x = gutils.split_and_load(x, ctx)
net(gpu_x[0]), net(gpu_x[1])
```

```
Out[4]: (
    [[ 3.22364103e-05 -1.77604452e-05 -4.53473040e-05 -4.43779390e-06
       4.11345136e-05 -1.12404132e-05 -3.77615615e-05  4.36321279e-05
       2.28628483e-06 -1.48372719e-05]
     [ 3.10487958e-05 -1.89744533e-05 -4.50448642e-05 -5.33929597e-06
       4.11167130e-05 -1.23149039e-05 -3.99470700e-05  4.46575323e-05
       2.58294403e-06 -1.34151524e-05]]
<NDArray 2x10 @gpu(0)>,
    [[ 2.98842060e-05 -1.86083871e-05 -4.19585049e-05 -5.71156716e-06
       4.02533478e-05 -1.19904144e-05 -3.85621279e-05  4.39719333e-05
       6.29562464e-08 -1.58273233e-05]
     [ 3.08366398e-05 -1.81654450e-05 -4.28299536e-05 -5.13964960e-06
       3.97496297e-05 -1.23091431e-05 -3.68527180e-05  4.15482536e-05
       3.11583290e-06 -1.18256366e-05]]
<NDArray 2x10 @gpu(1)>)
```

回忆一下“模型参数的延后初始化”一节中介绍的延后的初始化。现在，我们可以通过 `data` 访问初始化好的模型参数值了。需要注意的是，默认下 `weight.data()` 会返回 CPU 上的参数值。

由于我们指定了 2 个 GPU 来初始化模型参数，我们需要指定 GPU 访问。我们看到，相同参数在不同的 GPU 上的值一样。

```
In [5]: weight = net[1].params.get('weight')
try:
    weight.data()
except:
    print('not initialized on', mx.cpu())
weight.data(ctx[0])[0], weight.data(ctx[1])[0]

not initialized on cpu(0)

Out[5]: (
[[[-0.01473444 -0.01073093 -0.01042483]
 [-0.01327885 -0.01474966 -0.00524142]
 [ 0.01266256  0.00895064 -0.00601594]]]
<NDArray 1x3x3 @gpu(0)>,
[[[-0.01473444 -0.01073093 -0.01042483]
 [-0.01327885 -0.01474966 -0.00524142]
 [ 0.01266256  0.00895064 -0.00601594]]]
<NDArray 1x3x3 @gpu(1)>)
```

## 8.5.2 多 GPU 训练模型

我们先定义交叉熵损失函数。

```
In [6]: loss = gloss.SoftmaxCrossEntropyLoss()
```

当我们使用多个 GPU 来训练模型时，`gluon.Trainer` 会自动做数据并行，例如划分小批量数据样本并复制到各个 GPU 上，对各个 GPU 上的梯度求和再广播到所有 GPU 上。这样，我们就很方便地实现训练函数了。

```
In [7]: def train(num_gpus, batch_size, lr):
    train_iter, test_iter = gb.load_data_fashion_mnist(batch_size)
    ctx = [mx.gpu(i) for i in range(num_gpus)]
    print('running on:', ctx)
    net.initialize(init=init.Normal(sigma=0.01), ctx=ctx, force_reinit=True)
    trainer = gluon.Trainer(
        net.collect_params(), 'sgd', {'learning_rate': lr})
    for epoch in range(1, 6):
        start = time()
        for X, y in train_iter:
            gpu_Xs = gutils.split_and_load(X, ctx)
            gpu_ys = gutils.split_and_load(y, ctx)
```

```
with autograd.record():
    ls = [loss(net(gpu_X), gpu_y) for gpu_X, gpu_y in zip(
        gpu_Xs, gpu_ys)]
for l in ls:
    l.backward()
    trainer.step(batch_size)
nd.waitall()
print('epoch %d, training time: %.1f sec'%(epoch, time() - start))
test_acc = gb.evaluate_accuracy(test_iter, net, ctx[0])
print('validation accuracy: %.4f'%(test_acc))
```

我们在 2 个 GPU 上训练模型。

```
In [8]: train(num_gpus=2, batch_size=512, lr=0.3)
```

```
running on: [gpu(0), gpu(1)]
epoch 1, training time: 7.8 sec
validation accuracy: 0.4411
epoch 2, training time: 7.0 sec
validation accuracy: 0.8188
epoch 3, training time: 7.0 sec
validation accuracy: 0.8790
epoch 4, training time: 7.0 sec
validation accuracy: 0.8999
epoch 5, training time: 7.0 sec
validation accuracy: 0.9024
```

### 8.5.3 小结

- 在 Gluon 中，我们可以很方便地进行多 GPU 计算，例如在多 GPU 上初始化模型参数和训练模型。

### 8.5.4 练习

- 本节使用了 ResNet-18。试试不同的迭代周期、批量大小和学习率。如果条件允许，使用更多 GPU 计算。
- 有时候，不同的 CPU/GPU 的计算能力不一样，例如同时使用 CPU 和 GPU，或者 GPU 之间型号不一样。这时候应该怎么办？

### 8.5.5 扫码直达讨论区





## 9.1 图片增广

在“深度卷积神经网络：AlexNet”小节里我们提到过大规模数据集是深度网络能成功的前提条件。在 AlexNet 当年能取得的成功中，图片增广（image augmentation）功不可没。本小节我们将讨论这个在计算机视觉里被广泛使用的技术。

图片增广是指通过对训练图片做一系列变化来产生相似但又有不同的训练样本，这样来模型训练的时候识别了难以泛化的模式。例如我们可以对图片进行不同的裁剪使得感兴趣的物体出现在不同的位置中，从而使得模型减小对物体出现位置的依赖性。也可以调整亮度色彩等因素来降低模型对色彩的敏感度。

### 9.1.1 常用增广方法

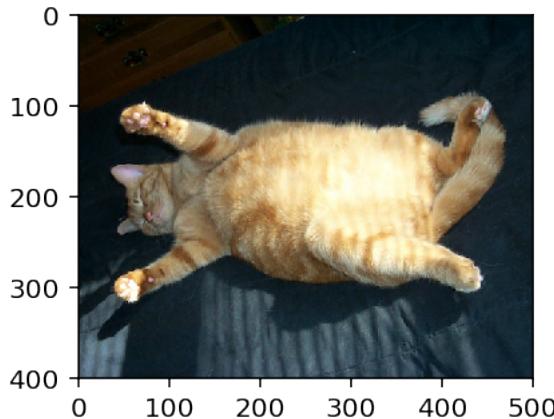
我们首先读取一张  $400 \times 500$  的图片作为样例解释常用的增广方法。

```
In [1]: %matplotlib inline  
import sys
```

```
sys.path.insert(0, '..')
import gluonbook as gb
from mxnet import nd, image, gluon, init
from mxnet.gluon.data.vision import transforms

img = image.imread('../img/cat1.jpg')
gb.plt.imshow(img.asnumpy())
```

Out[1]: <matplotlib.image.AxesImage at 0x7f735c750438>



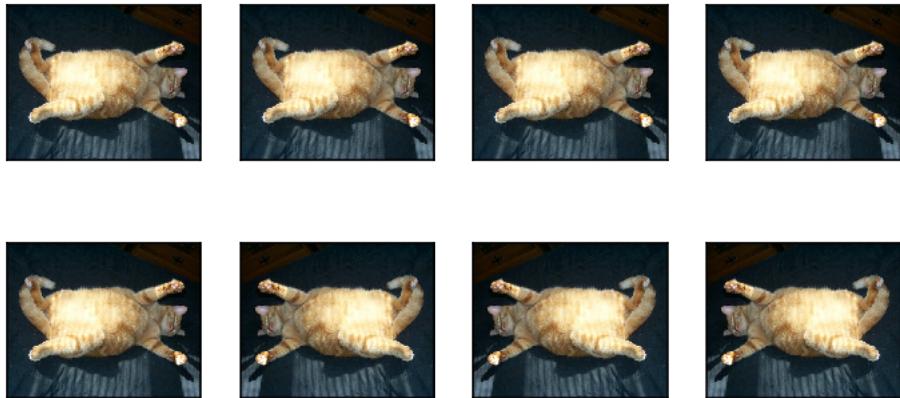
因为大部分的增广方法都有一定的随机性。接下来我们定义一个辅助函数，它对输入图片 `img` 运行多次增广方法 `aug` 并画出结果。

```
In [2]: def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
    Y = [aug(img) for _ in range(num_rows*num_cols)]
    gb.show_images(Y, num_rows, num_cols, scale)
```

## 变形

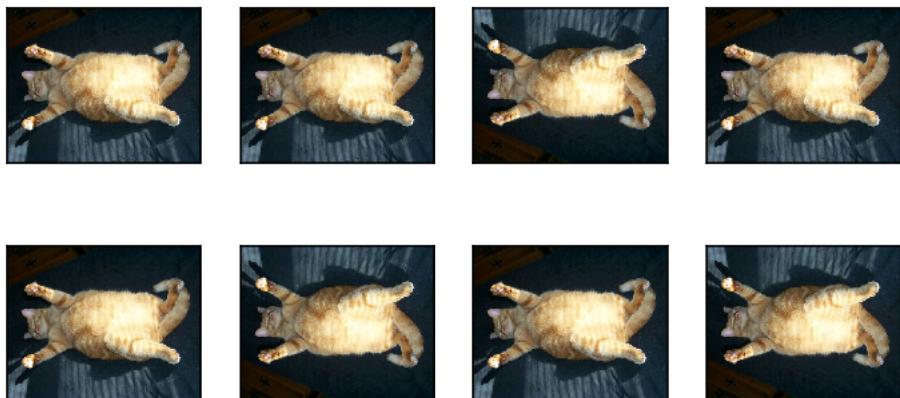
左右翻转图片通常不影响识别图片，它是最早也是最广泛使用的一种增广。下面我们使用 `transform` 模块里的 `RandomFlipLeftRight` 类来实现按 0.5 的概率左右翻转图片：

```
In [3]: apply(img, transforms.RandomFlipLeftRight())
```



当然有时候我们也使用上下翻转，至少对于我们使用的图片，上下翻转不会造成人的识别障碍。

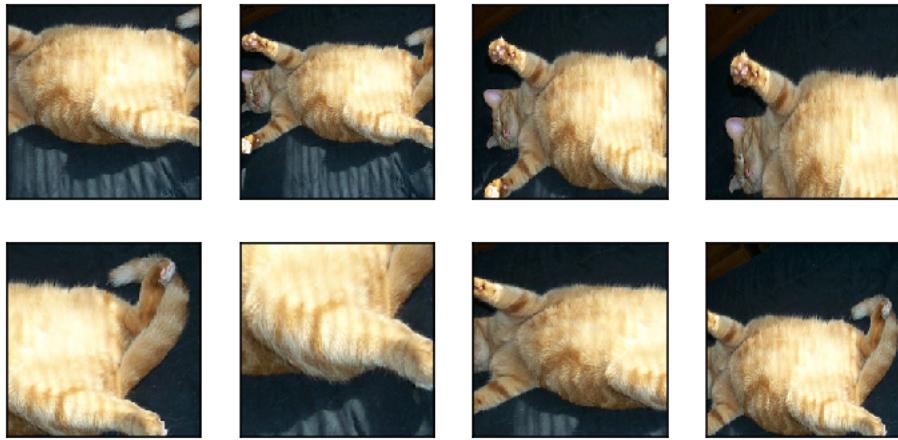
```
In [4]: apply(img, transforms.RandomFlipTopBottom())
```



我们使用的样例图片里猫在图片正中间，但一般情况下可能不是这样。“池化层”一节里我们解释了池化层能弱化卷积层对目标位置的敏感度，另一方面我们通过对图片随机剪裁来是的物体以不同的比例出现在不同位置。

下面代码里我们每次随机裁剪一片面积为原面积 10% 到 100% 的区域，其宽和高的比例在 0.5 和 2 之间，然后再将高宽缩放到 200 像素。

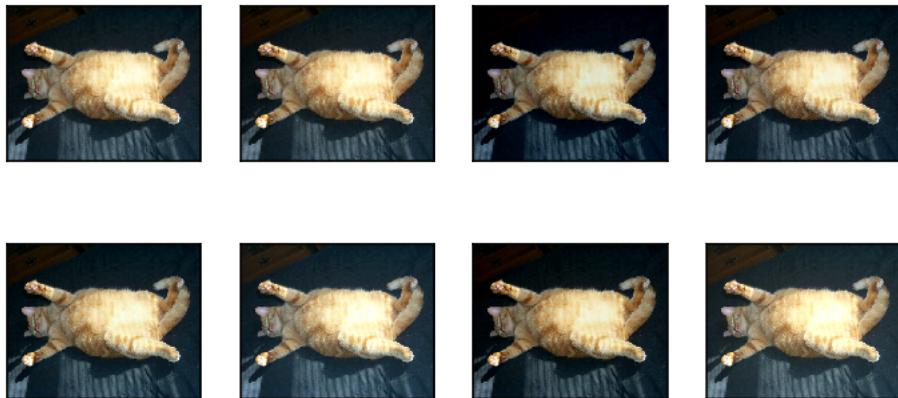
```
In [5]: shape_aug = transforms.RandomResizedCrop(  
    (200, 200), scale=(.1, 1), ratio=(.5, 2))  
apply(img, shape_aug)
```



## 颜色变化

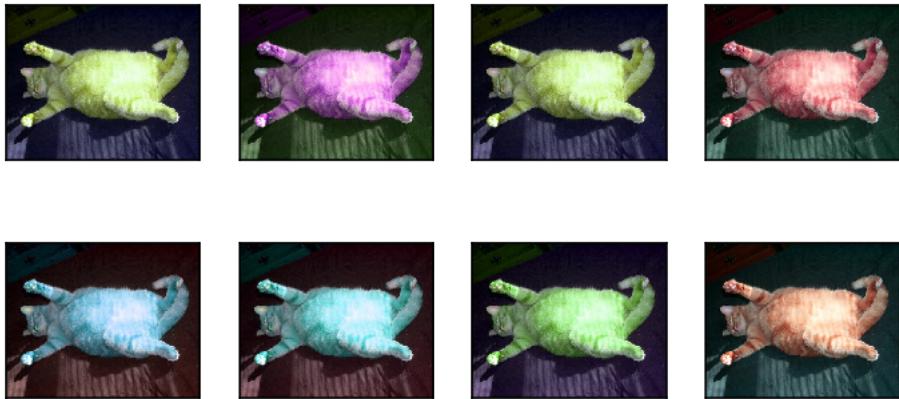
形状变化外的一个另一大类是变化颜色。颜色一般有四个可以调的参数：亮度、对比、饱和度和色相。下面例子里我们随机将亮度在当前值上增加或减小一个在 0 到 50% 之前的量。

```
In [6]: apply(img, transforms.RandomLighting(.5))
```



同样的修改色相。

```
In [7]: apply(img, transforms.RandomHue(.5))
```



或者用使用 `RandomColorJitter` 来一起使用。

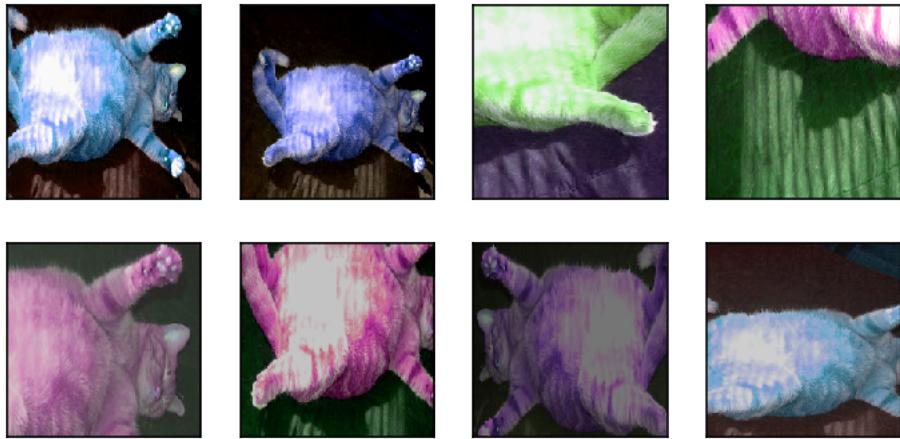
```
In [8]: color_aug = transforms.RandomColorJitter(  
    brightness=.5, contrast=.5, saturation=.5, hue=.5)  
apply(img, color_aug)
```



## 使用多个增广

实际应用中我们会将多个增广叠加使用。我们可以使用 `Compose` 类来将多个增广串联起来。

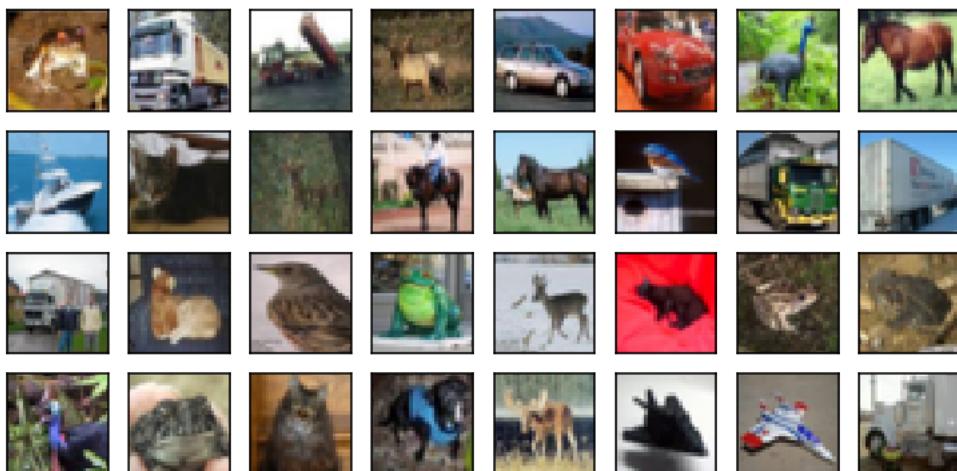
```
In [9]: augs = transforms.Compose([  
    transforms.RandomFlipLeftRight(), color_aug, shape_aug])  
apply(img, augs)
```



### 9.1.2 使用图片增广来训练

接下来我们来看一个将图片增广应用在实际训练的例子，并比较其与不使用时的区别。这里我们使用 CIFAR-10 数据集，而不是之前我们一直使用的 FashionMNIST。原因在于 FashionMNIST 中物体位置和尺寸都已经统一化了，而 CIFAR-10 中物体颜色和大小区别更加显著。下面我们展示 CIFAR-10 中的前 32 张训练图片。

```
In [10]: gb.show_images(gluon.data.vision.CIFAR10(train=True)[0:32][0], 4, 8,  
↪ scale=0.8)
```



在训练时，我们通常将图片增广作用在训练图片上，使得模型能识别出各种变化过后的版本。这里我们仅仅使用最简单的随机水平翻转。此外我们使用 `ToTensor` 变换来图片转成 MXNet 需要的格式，即格式为（批量，通道，高，宽）以及类型为 32 位浮点数。

```
In [11]: train_augs = transforms.Compose([
    transforms.RandomFlipLeftRight(),
    transforms.ToTensor(),
])

test_augs = transforms.Compose([
    transforms.ToTensor(),
])
```

接下来我们定义一个辅助函数来方便读取图片并应用增广。Gluon 的数据集提供 `transform_first` 函数来对数据里面的第一项图片（标签为第二项）来应用增广。另外图片增广将增加计算复杂度，我们使用两个额外 CPU 进程加来加速计算。

```
In [12]: def load_cifar10(is_train, augs, batch_size):
    return gluon.data.DataLoader(gluon.data.vision.CIFAR10(
        train=is_train).transform_first(augs),
        batch_size=batch_size, shuffle=is_train, num_workers=2)
```

## 模型训练

我们使用 ResNet 18 来训练 CIFAR-10。训练的的代码跟“残差网络：ResNet”一致，除了使用所有可用的 GPU 和不同的学习率外。

```
In [13]: def train(train_augs, test_augs, lr=.1):
    batch_size = 256
    ctx = gb.try_all_gpus()
    net = gb.resnet18(10)
    net.initialize(ctx=ctx, init=init.Xavier())
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate':lr})
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    train_data = load_cifar10(True, train_augs, batch_size)
    test_data = load_cifar10(False, test_augs, batch_size)
    gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=8)
```

首先我们看使用了图片增广的情况。

```
In [14]: train(train_augs, test_augs)

training on [gpu(0), gpu(1)]
epoch 1, loss 1.5049, train acc 0.460, test acc 0.417, time 11.9 sec
```

```
epoch 2, loss 1.0252, train acc 0.636, test acc 0.567, time 11.0 sec
epoch 3, loss 0.8094, train acc 0.716, test acc 0.611, time 10.9 sec
epoch 4, loss 0.6737, train acc 0.763, test acc 0.663, time 10.9 sec
epoch 5, loss 0.5820, train acc 0.796, test acc 0.695, time 10.9 sec
epoch 6, loss 0.5055, train acc 0.824, test acc 0.704, time 10.9 sec
epoch 7, loss 0.4397, train acc 0.847, test acc 0.640, time 10.9 sec
epoch 8, loss 0.3920, train acc 0.864, test acc 0.706, time 10.9 sec
```

作为对比，我们只对训练数据做中间剪裁。

```
In [15]: train(test_augs, test_augs)

training on [gpu(0), gpu(1)]
epoch 1, loss 1.4877, train acc 0.467, test acc 0.459, time 11.2 sec
epoch 2, loss 1.0011, train acc 0.644, test acc 0.429, time 10.9 sec
epoch 3, loss 0.7691, train acc 0.730, test acc 0.510, time 10.9 sec
epoch 4, loss 0.6145, train acc 0.785, test acc 0.558, time 10.9 sec
epoch 5, loss 0.4842, train acc 0.831, test acc 0.639, time 10.9 sec
epoch 6, loss 0.3681, train acc 0.872, test acc 0.651, time 10.9 sec
epoch 7, loss 0.2572, train acc 0.913, test acc 0.620, time 10.9 sec
epoch 8, loss 0.1750, train acc 0.942, test acc 0.682, time 10.9 sec
```

可以看到，即使是简单的随机翻转也会有明显效果。使用增广类似于增加了正则项话，它使得训练精度变低，但对提升测试精度有帮助。

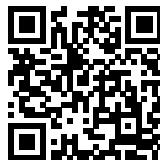
### 9.1.3 小结

图片增广对现有训练数据生成大量随机图片来有效避免过拟合。

### 9.1.4 练习

尝试在 CIFAR-10 训练中增加不同的增广方法。

### 9.1.5 扫码直达讨论区



## 9.2 微调

之前章节里我们通过大量样例演示了如何在只有 6 万张图片的 FashionMNIST 上训练模型。我们也介绍了 ImageNet 这个当下学术界使用最广的大数据集，它有超过一百万的图片和一千类的物体。但我们平常接触到数据集的规模通常在两者之间。

想象一下开发一个应用来从图片中识别里面的凳子然后提供购买链接给用户。一个可能的做法是先去找一百把常见的凳子，对每个凳子收集一千张不同的图片，然后在收集到的数据上训练一个分类器。这个数据集虽然可能比 FashionMNIST 要复杂，但仍然比 ImageNet 小 10 倍。这可能导致针对 ImageNet 提出的模型在这个数据上会过拟合。同时因为数据量有限，最终我们得到的分类器的模型精度也许达不到实用的要求。

一个解决办法是收集更多的数据。但注意到收集和标注数据均会花费大量的人力和财力。例如 ImageNet 这个数据集花费了数百万美元的研究经费。虽然目前的数据采集成本降低了十倍以上，但其成本仍然不可忽略。

另外一种解决办法是迁移学习 (transfer learning)，它通过将其他数据集来帮助学习当前数据集。例如，虽然 ImageNet 的图片基本跟椅子无关，但其上训练到的模型可能能做一些通用的图片特征抽取，例如识别边缘、纹理、形状和物体组成。这个对于识别椅子也可能同样有效。

本小节我们介绍迁移学习里面的一个常用技术：微调 (fine tuning)。它由下面四步构成：

1. 在源数据（例如 ImageNet）上训练一个神经网络  $A$ 。
2. 创建一个新的神经网络  $B$ ，它复制  $A$  上除了输出层外的所有模型参数。这里的假设是这些模型参数含有源数据上学习到的知识，这些知识同样适用于目标数据集。但最后的输出层跟源数据标注紧密相关，所以不被重用。
3. 为  $B$  添加一个输出大小为目标数据集类别数目（例如一百类椅子）的输出层，并将其权重初始化成随机值。

4. 在目标数据集（例如椅子数据集）上训练  $B$ 。我们将从头开始学习输出层，但其余层都是基于源数据上的模型参数进行微调。

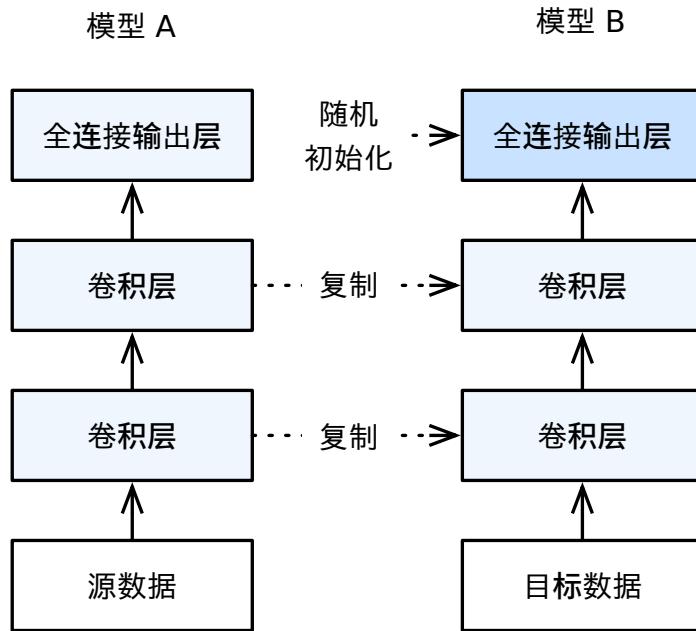


图 9.1: 微调。

接下来我们来看一个具体的例子，它使用 ImageNet 上训练好的 ResNet 来微调一个我们构造的小数据集：其含有数千张包含热狗和不包含热狗的图片。

### 9.2.1 热狗识别

#### 获取数据

我们使用的热狗数据集是从网上抓取的，它含有 1400 张包含热狗的正类图片，和同样多包含其他食品的负类图片。各类的 1000 张图片被用作训练，其余的作为测试。

我们首先将数据下载到`../data`。在当前目录解压后得到 `hotdog/train` 和 `hotdog/test` 这两个文件夹。每个下面有 `hotdog` 和 `not-hotdog` 这两个类别文件夹，里面是对应的图片文

件。

```
In [1]: %matplotlib inline
import sys
sys.path.insert(0, '...')
import zipfile
import gluonbook as gb
from mxnet import nd, image, gluon, init
from mxnet.gluon.data.vision import transforms

data_dir = '../data/'
base_url = 'https://apache-mxnet.s3-accelerate.amazonaws.com/'
fname = gluon.utils.download(
    base_url+'gluon/dataset/hotdog.zip',
    path=data_dir, sha1_hash='fba480ffa8aa7e0feb511d181409f899b9baa5')

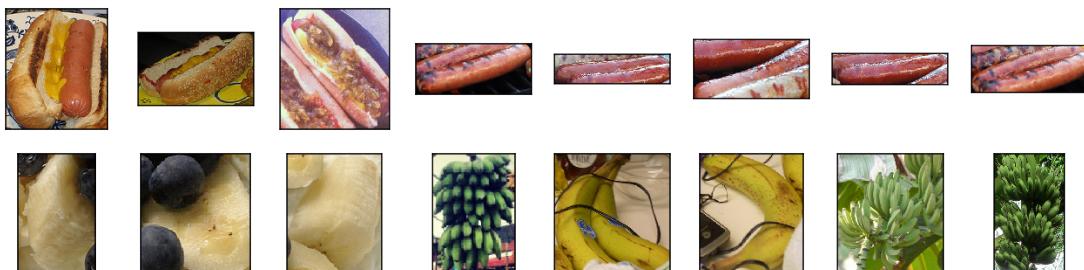
with zipfile.ZipFile(fname, 'r') as f:
    f.extractall(data_dir)
```

我们使用使用 `ImageFolderDataset` 类来读取数据。它将每个类别文件夹当做一个类，并读取下面所有的图片。

```
In [2]: train_imgs = gluon.data.vision.ImageFolderDataset(data_dir+'/hotdog/train')
test_imgs = gluon.data.vision.ImageFolderDataset(data_dir+'/hotdog/test')
```

下面画出前 8 张正例图片和最后的 8 张负例图片，可以看到他们性质和高宽各不相同。

```
In [3]: hotdogs = [train_imgs[i][0] for i in range(8)]
not_hotdogs = [train_imgs[-i-1][0] for i in range(8)]
gb.show_images(hotdogs+not_hotdogs, 2, 8, scale=1.4)
```



我们将训练图片首先扩大到高宽为 480，然后随机剪裁出高宽为 224 的输入。测试图片则是简单的中心剪裁。此外，我们对输入的 RGB 通道数值进行了归一化。

```
In [4]: # 指定 RGB 三个通道的均值和方差来将图片通道归一化。
normalize = transforms.Normalize(
```

```
[0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = transforms.Compose([
    transforms.Resize(480),
    transforms.RandomResizedCrop(224),
    transforms.RandomFlipLeftRight(),
    transforms.ToTensor(),
    normalize,
])
test_augs = transforms.Compose([
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    normalize
])
```

## 微调模型

我们用在 ImageNet 上训练好了 ResNet 18 来作为基础模型。这里指定 `pretrained=True` 来自动下载并加载训练好的权重。

```
In [5]: pretrained_net = gluon.model_zoo.vision.resnet18_v2(pretrained=True)
```

预训练好的模型由两块构成，一是 `features`，二是 `output`。前者包含从输入开始的大部分卷积和全连接层，后者主要包括最后一层全连接层。这样的划分的主要目的是为了更方便做微调。下面查看下 `output` 的内容：

```
In [6]: pretrained_net.output
```

```
Out[6]: Dense(512 -> 1000, linear)
```

它将 ResNet 最后的全局平均池化层输出转化成 1000 类的输出。

在微调中，我们新建一个网络，它的定义跟之前训练好的网络一样，除了最后的输出数等于当前数据的类别数。就是说新网络的 `features` 被初始化成前面训练好网络的权重，而 `output` 则是从头开始训练。

```
In [7]: finetune_net = gluon.model_zoo.vision.resnet18_v2(classes=2)
finetune_net.features = pretrained_net.features
finetune_net.output.initialize(init.Xavier())
```

## 9.2.2 训练

我们先定义一个可以重复使用的训练函数。

```
In [8]: def train(net, learning_rate, batch_size=128, epochs=5):
    train_data = gluon.data.DataLoader(
        train_imgs.transform_first(train_augs), batch_size, shuffle=True)
    test_data = gluon.data.DataLoader(
        test_imgs.transform_first(test_augs), batch_size)

    ctx = gb.try_all_gpus()
    net.collect_params().reset_ctx(ctx)
    net.hybridize()
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {
        'learning_rate': learning_rate, 'wd': 0.001})
    gb.train(train_data, test_data, net, loss, trainer, ctx, epochs)
```

因为微调的网络中的主要层的已经训练的足够好，所以一般采用比较小的学习率，防止过大的步长对训练好的层产生过多影响。

```
In [9]: train(finetune_net, 0.01)

training on [gpu(0), gpu(1)]
epoch 1, loss 0.4547, train acc 0.795, test acc 0.892, time 19.4 sec
epoch 2, loss 0.2506, train acc 0.896, test acc 0.919, time 17.0 sec
epoch 3, loss 0.1962, train acc 0.922, test acc 0.934, time 16.9 sec
epoch 4, loss 0.1676, train acc 0.932, test acc 0.930, time 16.9 sec
epoch 5, loss 0.1582, train acc 0.937, test acc 0.944, time 16.8 sec
```

为了对比起见，我们训练同样的一个模型，但所有参数都初始成随机值。我们使用较大的学习率来加速收敛。

```
In [10]: scratch_net = gluon.model_zoo.vision.resnet18_v2(classes=2)
scratch_net.initialize(init=init.Xavier())
train(scratch_net, 0.1)

training on [gpu(0), gpu(1)]
epoch 1, loss 0.7675, train acc 0.706, test acc 0.752, time 17.0 sec
epoch 2, loss 0.3918, train acc 0.833, test acc 0.771, time 16.9 sec
epoch 3, loss 0.4083, train acc 0.817, test acc 0.835, time 16.9 sec
epoch 4, loss 0.3704, train acc 0.840, test acc 0.848, time 16.9 sec
epoch 5, loss 0.3416, train acc 0.856, test acc 0.811, time 16.9 sec
```

可以看到，微调的模型因为初始值更好，它的收敛比从头开始训练要快很多。在很多情况下，微

调的模型最终的收敛到的结果也可能比非微调的模型更好。

### 9.2.3 小结

微调通过将模型部分权重初始化成在源数据集上预训练好的模型参数，从而将模型在源数据集上学到的知识迁移到目标数据上。

### 9.2.4 练习

- 对 `finetune_net` 试着增大学习率看看收敛变化。
- 多跑几个 `epochs` 直到收敛（你可以也需要调调参数），看看 `scratch_net` 和 `finetune_net` 最后的精度是不是有区别
- 这里 `finetune_net` 重用了 `pretrained_net` 除最后全连接外的所有权重，试试少重用些权重，有会有什么区别
- 事实上 ImageNet 里也有 `hotdog` 这个类，它对应的输出层参数可以如下拿到。试试如何使用它。

```
In [11]: weight = pretrained_net.output.weight  
hotdog_w = nd.split(weight.data(), 1000, axis=0)[713]  
hotdog_w.shape
```

```
Out[11]: (1, 512)
```

- 试试不让 `finetune_net` 里重用的权重参与训练，也就是不更新他们的权重。

### 9.2.5 扫码直达讨论区



## 9.3 物体检测：边界框和预测

前面小节里我们介绍了诸多用于图片分类的模型。在这个任务里，我们假设图片里只有一个主体物体，然后目标是识别这个物体的类别。但很多时候图片里有多个感兴趣的物体，我们不仅仅想知道它们是什么，而且想得到它们在图片中的具体位置。例如在无人驾驶任务里，我们需要识别拍摄到的图片里的车辆、行人、道路和障碍的位置来规划行进线路。在计算机视觉里，我们将这类任务称之为物体检测。

在接下来的数小节里我们将介绍物体检测里的多个深度学习模型。在此之前，让我们先讨论物体位置这个概念。首先我们加载本小节将使用的示例图片。

```
In [1]: %matplotlib inline
import sys
sys.path.insert(0, '..')
import gluonbook as gb
import numpy as np
from mxnet import image, nd, contrib

img = image.imread('../img/catdog.jpg').asnumpy()
gb.plt.imshow(img); # 用；使得不要显示它的输出。
```



可以看到图片左边是一只小狗，右边是一只小猫。跟前面使用的图片的主要不同点在于这里有两个主体物体。

### 9.3.1 边界框

在物体识别里，我们通常使用边界框（bounding box）来确定物体位置。它一个矩形框，可以由左上角的 x、y 轴位置与右下角 x、y 轴位置确定。我们根据上图坐标信息来定义图中小狗和小猫

的边界框。

```
In [2]: # 注意坐标轴原点是图片的左上角。bbox 是 bounding box 的缩写。
```

```
dog_bbox = [60, 0, 340, 365]  
cat_bbox = [360, 80, 580, 365]
```

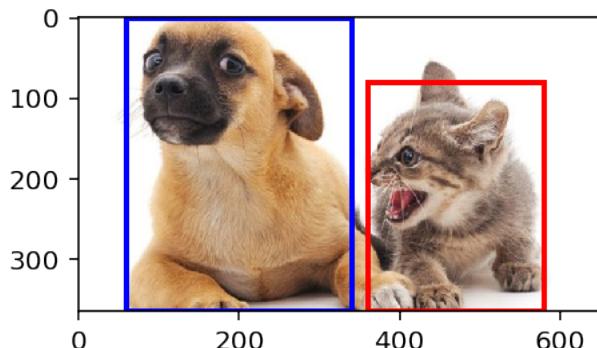
我们可以在图中将边框画出来检查其准确性。画之前我们定义一个辅助函数，它将边界框表示成 matplotlib 的边框格式，这个函数将保存在 GluonBook 里方便之后使用。

```
In [3]: # 将边界框 (左上 x, 左上 y, 右下 x, 右下 y) 格式转换成 matplotlib 格式:
```

```
# ((左上 x, 左上 y), 宽, 高)。  
def bbox_to_rect(bbox, color):  
    return gb.plt.Rectangle(  
        xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],  
        fill=False, edgecolor=color, linewidth=2)
```

我们将边界框加载在图上，可以看到物体的主体基本在框内。

```
In [4]: fig = gb.plt.imshow(img)  
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))  
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
```



### 9.3.2 预测边界框

相比于图片分类，物体识别的一个主要复杂点在于需要预测物体的边界框。一般这是通过下面两个步骤来完成：首先针对输入图片提出数个区域，然后对每个区域判断其是否包含感兴趣的物体，如果是则进一步预测其边界框。

### 9.3.3 锚框

不同的模型使用不同的区域生成方法，这里我们介绍其中常用的一种：它以每个输入像素为中心生成数个大小和比例不同的默认边界框，或者称之为锚框（anchor box）。假设输入高为  $h$ ，宽为  $w$ ，那么大小为  $s \in (0, 1]$  和比例为  $r > 0$  的锚框形状是

$$\left(ws\sqrt{r}, \frac{hs}{\sqrt{r}}\right).$$

注意到便利不同的  $s$  的  $r$  会生成大量的锚框，这样将使得计算很复杂。一般我们需要对其进行采样。一个例子是首先固定一个比例  $r_1$ ，然后采样  $n$  个不同的大小  $s_1, \dots, s_n$ 。然后固定一个大小  $s_1$ ，采样  $m$  个不同的比例  $r_1, \dots, r_m$ 。这样对每个像素我们一共生成  $n + m - 1$  个锚框。对于整个输入图片，我们将一共生成  $wh(n + m - 1)$  个锚框。

上面描述的采样方法实现在 `contrib.ndarray` 中的 `MultiBoxPrior` 函数。通过指定输入数据（我们只需要访问其形状），锚框的采样大小和比例，这个函数将返回所有采样到的锚框。

```
In [5]: h, w = img.shape[0:2]
x = nd.random.uniform(shape=(1, 3, h, w))
y = contrib.ndarray.MultiBoxPrior(x, sizes=[.75, .5, .25], ratios=[1, 2, .5])
('total #anchor boxes', y.shape[1])
```

```
Out[5]: ('total #anchor boxes', 1186250)
```

将返回结果变形成（高，宽， $n + m - 1$ , 4）后，我们可以方便的访问以任何一个像素为中心的所有锚框。下面例子里我们访问以（200, 200）为中心的第一个锚框。它有四个元素，同前一样是左上和右下的 x、y 轴坐标，但被分别除以了高和宽使得数值在 0 和 1 之间。

```
In [6]: boxes = y.reshape((h, w, 5, 4))
boxes[200, 200, 0, :]
```

```
Out[6]:
[ 0.09788463  0.17431509  0.51903844  0.92431509]
<NDArray 4 @cpu(0)>
```

在画出这些锚框的具体样子前我们先定义一个函数来图上画出多个边界框，它将被保存在 `GluonBook` 里以便后面使用。

```
In [7]: def show_bboxes(axes, bboxes, labels=None):
    colors = ['b', 'g', 'r', 'k', 'm']
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = bbox_to_rect(bbox.astype(np.int), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
```

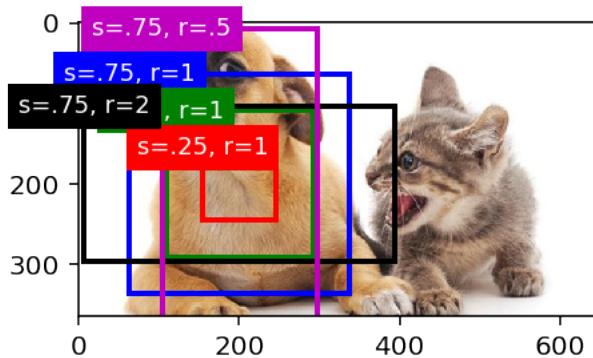
```

        axes.text(rect.xy[0], rect.xy[1], labels[i],
                    va="center", ha="center", fontsize=9, color='white',
                    bbox=dict(facecolor=color, lw=0))
    
```

然后我们画出以 (200, 200) 为中心的所有锚框。

```

In [8]: bbox_scale = nd.array((w, h, w, h)) # 需要乘以高和宽使得符合我们的画图格式。
fig = gb.plt.imshow(img)
show_bboxes(fig.axes, boxes[200, 200, :, :] * bbox_scale, [
    's=.75, r=1', 's=.5, r=1', 's=.25, r=1', 's=.75, r=2', 's=.75, r=.5'])
    
```



可以看到大小为 0.75 比例为 0.5 的洋红色锚框比较好的覆盖了图片中的小狗。

在预测的时候，我们对每个锚框预测一个到真实边界框的偏移。例如对于洋红锚框来说，我们希望其预测值是：

```

In [9]: boxes[200, 200, 3, :] - nd.array(dog_bbox) / bbox_scale
Out[9]:
[-0.08164687  0.28415003  0.08318532 -0.18551981]
<NDArray 4 @cpu(0)>
    
```

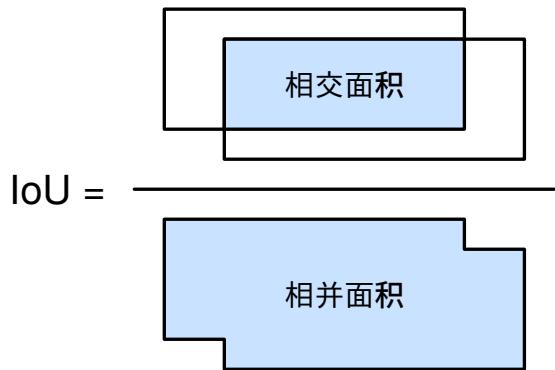
### 9.3.4 IoU：交集除并集

上面例子里我们生成了一百万以上的边界框，从而会有同样的预测边界框。而图片中只有两个物体，这导致大量的边界框都非常相似。我们需要去除冗余，使得预测结果更有可读性。

为了去除冗余，我们需要先定义如何判断两个边界框的相似性。我们知道集合相似度的最常用衡量标准叫做 Jaccard 距离。给定集合  $A$  和  $B$ ，它的定义是集合的交集除以集合的并集：

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

边界框指定了一块像素区域，其也可以看成是像素点的集合。因此我们可以定类似的距离，即将两个边界框的相交面积除以相并面积。



我们将这个测量方法称之为交集除并集（Intersection over Union, IoU）。它的取值范围在 0 和 1 之间。0 表示边界框不相关，1 则表示完全一样。

### 9.3.5 NMS：非最大抑制

对于相似的预测边界框，非最大抑制（Non-Maximum Suppression, NMS）只保留置信度最高的那个来去除冗余。它的工作机制如下：对于每个类别，我们首先拿到每个预测边界框被判断包含这个类别物体的概率。然后我们找到概率最大的那个边界框并保留它到输出，接下来移除掉（抑制）其它所有的跟这个边界框的 IoU 大于某个阈值的边界框。在剩下的边界框里我们再找出预测概率最大的边界框，重复前面的移除过程。直到我们要么保留或者移除了每个边界框。

非最大抑制的实现包含在 `contrib.ndarray` 的 `MultiBoxDetection` 里。这里锚框的格式是（批量大小， 锚框个数， 4），预测偏移则是（批量大小， 锚框个数  $\times$  4），这里偏移使用了更符合网络输出层的格式。假设锚框是 `A`，预测偏移是 `P`，那么预测边界框就是 `A+P.reshape(A.shape)`。类别预测的格式是（批量大小， 类别数 +1， 锚框个数），这里第 0 类预留给了背景，即不含有需要被识别的物体。

下面我们构造四个预测框。为了简单起见我们在锚框直接放置预测边界框内容，预测偏移则设成 0。

```
In [10]: anchors = nd.array([[.09, .00, .53, 1], # 每一行是一个预测框。
                           [.08, .09, .56, .95],
```

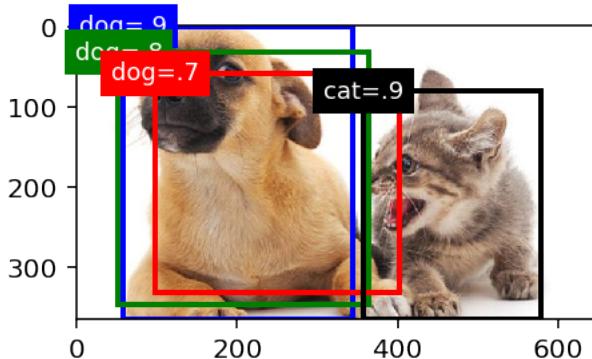
```

        [.15, .16, .62, .91],
        [.55, .22, .89, 1]]])
loc_preds = nd.array([[.0]*anchors.size])
cls_probs = nd.array([[0]*4, # 是背景的概率。
                     [.9, .8, .7, .1], # 是狗的概率。
                     [.1, .2, .3, .9]]]) # 是猫的概率。

```

在实际图片上看一下他们的位置和预测概率：

```
In [11]: fig = gb.plt.imshow(img)
show_bboxes(fig.axes, anchors.reshape(-1, 4) * bbox_scale,
            ['dog=.9', 'dog=.8', 'dog=.7', 'cat=.9'])
```



给定锚框、预测偏移、类别预测概率、和 IoU 阈值，`MultiBoxDetection` 返回格式为（批量大小，锚框个数，6）的结果。每一行对应一个预测边界框，其有六个元素，依次为预测类别（-1 表示被该边界框抑制了，其余物体类别从 0 开始，且移除了背景类）、预测物体属于此类的概率和预测边界框的位置。

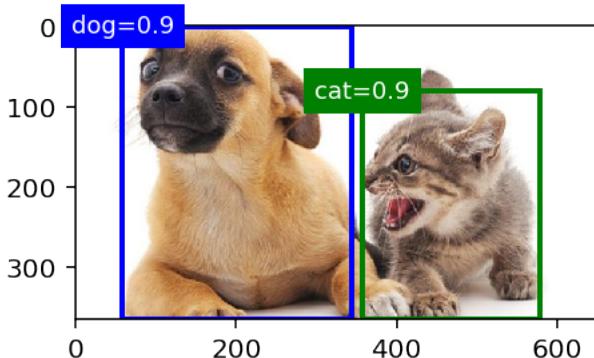
```
In [12]: np.set_printoptions(2) # 使得打印更加简洁。
ret = contrib.ndarray.MultiBoxDetection(
    cls_probs, loc_preds, anchors, nms_threshold=.5).asnumpy()
ret

Out[12]: array([[[ 0. ,  0.9 ,  0.09,  0. ,  0.53,  1. ],
   [ 1. ,  0.9 ,  0.55,  0.22,  0.89,  1. ],
   [-1. ,  0.8 ,  0.08,  0.09,  0.56,  0.95],
   [-1. ,  0.7 ,  0.15,  0.16,  0.62,  0.91]], dtype=float32)
```

我们移除掉-1 类的结果来可视化 NMS 保留的结果。

```
In [13]: bboxes = [nd.array(i[2:])*bbox_scale for i in ret[0] if i[0] != -1]
labels = [('dog=', 'cat=')[int(i[0])] + str(i[1]) for i in ret[0] if i[0] != -1]
```

```
fig = gb.plt.imshow(img)
show_bboxes(fig.axes, bboxes, labels)
```



NMS 在 70 年代提出后有数个变种。例如假设一张图片里出现的物体数远小于类别数，所以对每个类做抑制可能意义不大，因为绝大部分类别物体不会出现。因此我们可以忽略掉类别来做全局抑制，这样避免某些类即使最大的预测概率很低但仍然被输出。此外，我们也可以指定只返回预测值最高的固定数目的结果。

### 9.3.6 小节

在物体识别里我们不仅需要找出图片里面所有感兴趣的物体，而且要知道它们的位置。位置一般由方形边界框来表示。这一小节我们讨论了如何生产一系列的锚框来预测边界框，和在预测过程中如何用 IoU 来判断边界框的相似度并使用 NMS 来消除相似的输出使得预测边界框更加简洁。

### 9.3.7 练习

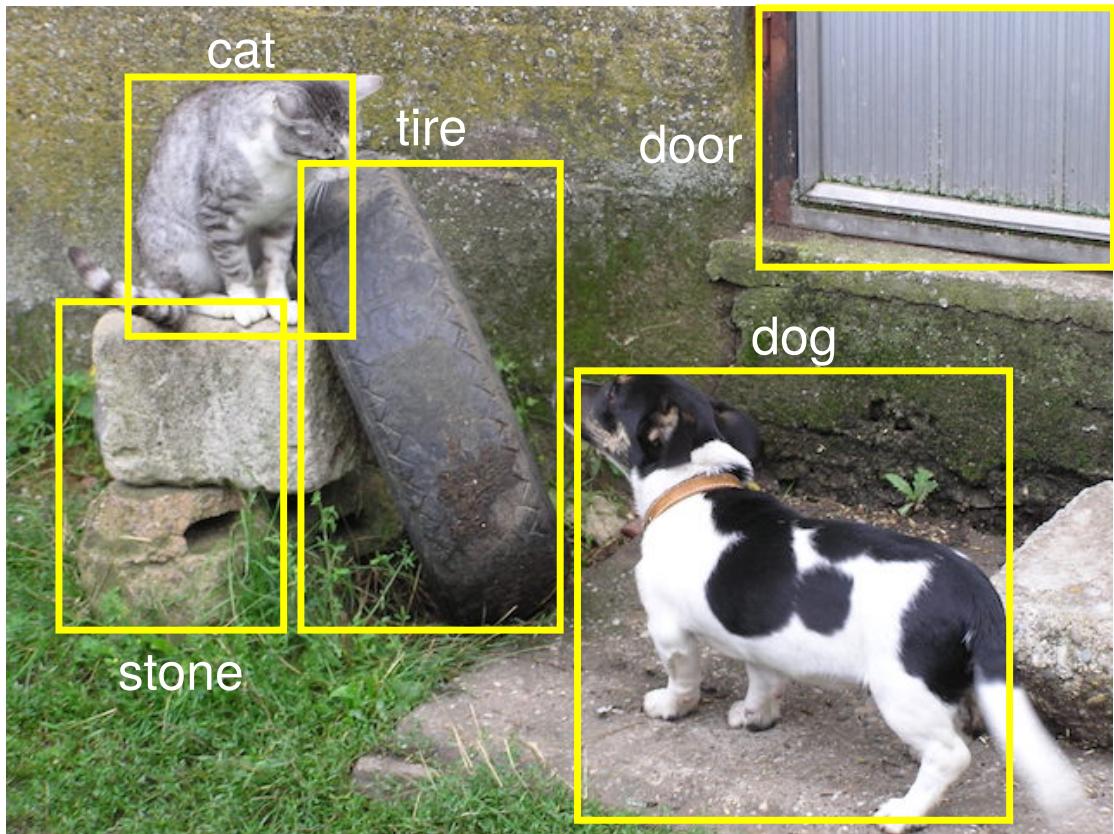
- 找一些图片，尝试标注下其中物体的边界框。比较下同图片分类标注所花时间的区别。
- 改变锚框生成里面的大小和比例采样来看看可视化时的区别。
- 修改 NMS 的阈值来看其对结果的影响。
- 试试 MultiBoxDetection 里的其他选项，例如 force\_suppress 和 nms\_topk。

## 9.4 目标检测

当我们讨论对图片进行预测时，到目前为止我们都是谈论分类。我们问过这个数字是 0 到 9 之间的哪一个，这个图片是鞋子还是衬衫，或者下面这张图片里面是猫还是狗。



但现实图片会更加复杂，它们可能不仅仅包含一个主体物体。物体检测就是针对这类问题提出，它不仅是要分析图片里有什么，而且需要识别它在什么位置。我们使用在[机器学习简介](#)那章讨论过的图片作为样例，并对它标上主要物体和位置。



可以看出物体检测跟图片分类有几个不同点：

1. 图片分类器通常只需要输出对图片中的主物体的分类。但物体检测必须能够识别多个物体，即使有些物体可能在图片中不是占主要版面。严格上来说，这个任务一般叫多类物体检测，但绝大部分研究都是针对多类的设置，所以我们这里为了简单去掉了“多类”
2. 图片分类器只需要输出将图片物体识别成某类的概率，但物体检测不仅需要输出识别概率，还需要识别物体在图片中的位置。这个通常是一个括住这个物体的方框，通常也称之为边界框（bounding box）。

但也看到物体检测跟图片分类有类似之处，都是对一块图片区域判断其包含的主要物体。因此可以想象我们在前面介绍的基于卷积神经网络的图片分类可以被应用到这里。

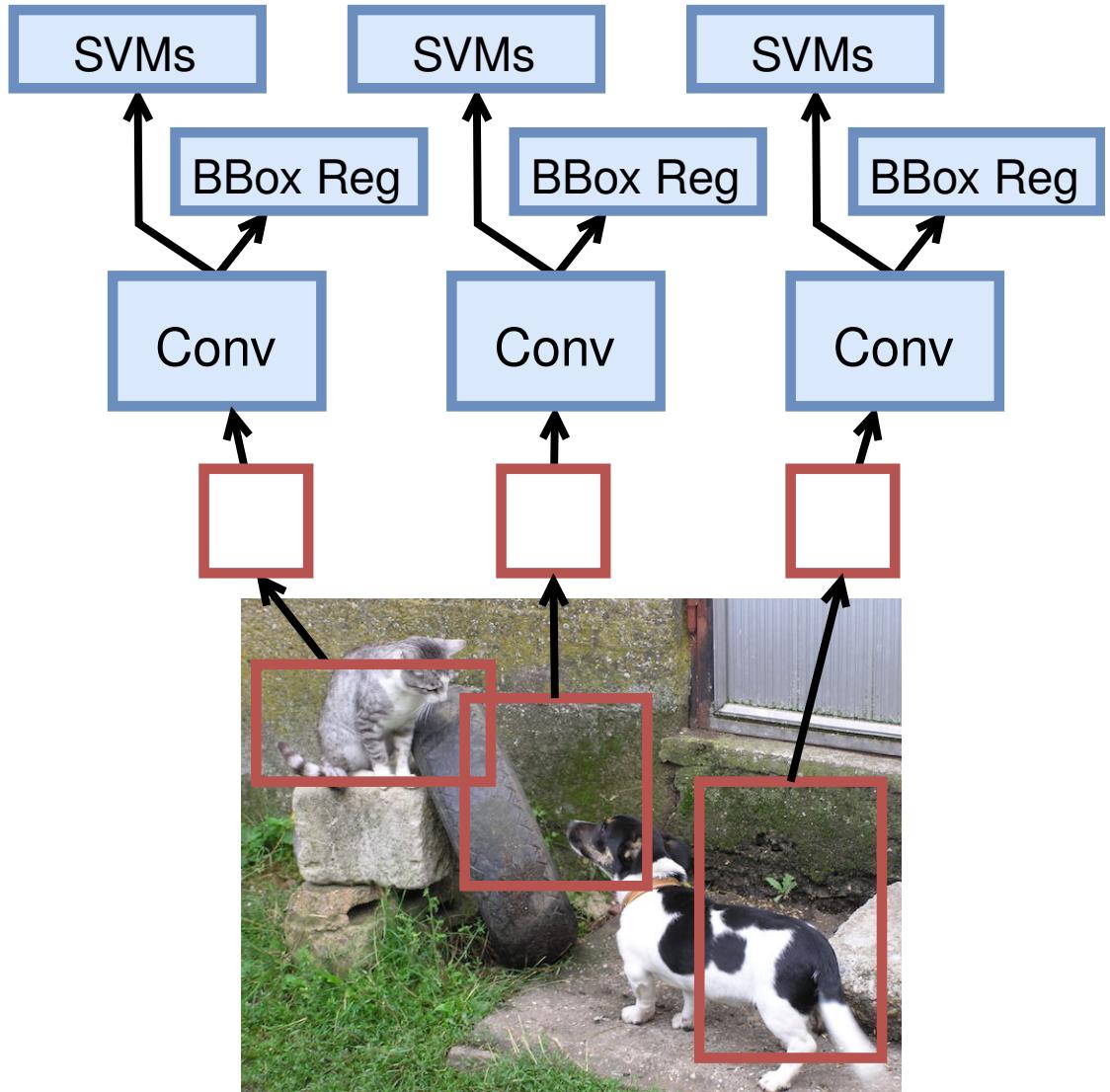
这一章我们将介绍数个基于卷积神经网络的物体检测算法的思想：

- R-CNN: <https://arxiv.org/abs/1311.2524>

- Fast R-CNN: <https://arxiv.org/abs/1504.08083>
- Faster R-CNN: <https://arxiv.org/abs/1506.01497>
- Mask R-CNN: <https://arxiv.org/abs/1703.06870>
- SSD: <https://arxiv.org/abs/1512.02325>
- YOLO: <https://arxiv.org/abs/1506.02640>
- YOLOv2: <https://arxiv.org/abs/1612.08242>

#### 9.4.1 R-CNN：区域卷积神经网络

这是基于卷积神经网络的物体检测的奠基之作。其核心思想是在对每张图片选取多个区域，然后每个区域作为一个样本进入一个卷积神经网络来抽取特征，最后使用分类器来对齐分类，和一个回归器来得到准确的边框。



## Selective Search

图 9.2: R-CNN

具体来说，这个算法有如下几个步骤：

1. 对每张输入图片使用一个基于规则的“选择性搜索”算法来选取多个提议区域

2. 跟微调迁移学习里那样，选取一个预先训练好的卷积神经网络并去掉最后一个输入层。每个区域被调整成这个网络要求的输入大小并计算输出。这个输出将作为这个区域的特征。
3. 使用这些区域特征来训练多个 SVM 来做物体识别，每个 SVM 预测一个区域是不是包含某个物体
4. 使用这些区域特征来训练线性回归器将提议区域

直观上 R-CNN 很好理解，但问题是它可能特别慢。一张图片我们可能选出上千个区域，导致一张图片需要做上千次预测。虽然跟微调不一样，这里训练可以不用更新用来抽特征的卷积神经网络，从而我们可以事先算好每个区域的特征并保存。但对于预测，我们无法避免这个。从而导致 R-CNN 很难实际中被使用。

#### 9.4.2 Fast R-CNN：快速的区域卷积神经网络

Fast R-CNN 对 R-CNN 主要做了两点改进来提升性能。

1. 考虑到 R-CNN 里面的大量区域可能是相互覆盖，每次重新抽取特征过于浪费。因此 Fast R-CNN 先对输入图片抽取特征，然后再选取区域
2. Fast R-CNN 不再使用多个 SVM 来做分类，而是用单个多类逻辑回归，这也是前面教程里默认使用的。

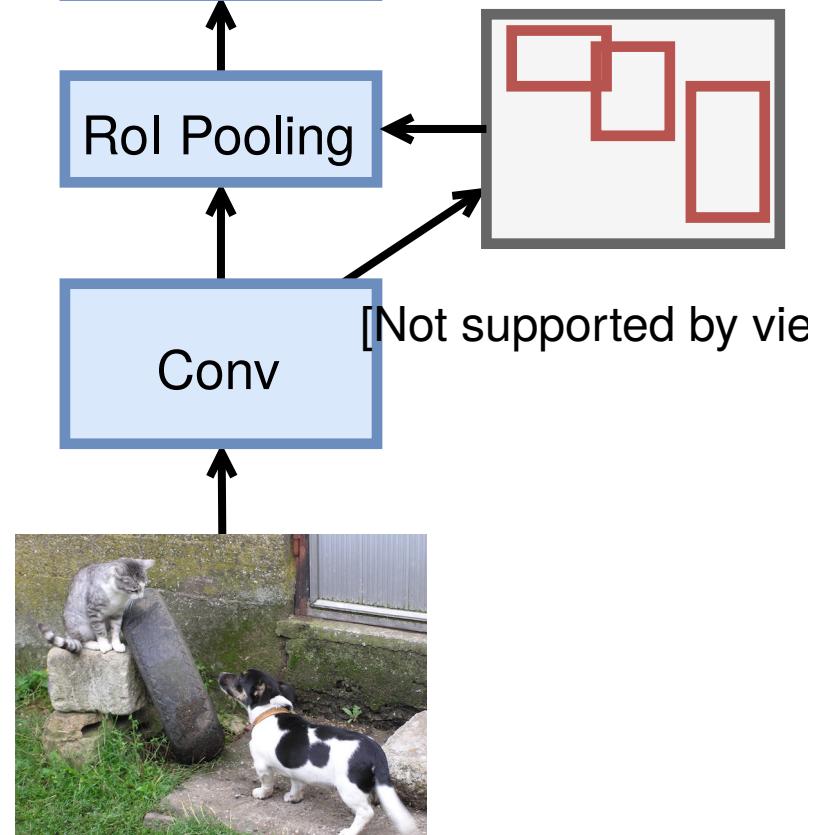


图 9.3: Fast R-CNN

从示意图可以看到，使用选择性搜索选取的区域是作用在卷积神经网络提取的特征上。这样我们只需要对原始的输入图片做一次特征提取即可，如此节省了大量重复计算。

Fast R-CNN 提出兴趣区域池化层（Region of Interest (RoI) pooling），它的输入为特征和一系列

的区域，对每个区域它将其均匀划分成  $n \times m$  的小区域，并对每个小区域做最大池化，从而得到一个  $n \times m$  的输出。因此不管输入区域的大小，RoI 池化层都将其池化成固定大小输出。

下面我们仔细看一下 RoI 池化层是如何工作的，假设对于一张图片我们提出了一个  $4 \times 4$  的特征，并且通道数为 1.

```
In [1]: from mxnet import nd

x = nd.arange(16).reshape((1,1,4,4))
x

Out[1]:
[[[ 0.   1.   2.   3.]
  [ 4.   5.   6.   7.]
  [ 8.   9.  10.  11.]
  [ 12.  13.  14.  15.]]]]
<NDArray 1x1x4x4 @cpu(0)>
```

然后我们创建两个区域，每个区域由一个长为 5 的向量表示。第一个元素是其对应的物体的标号，之后分别是  $x_{min}$ ,  $y_{min}$ ,  $x_{max}$ , 和  $y_{max}$ 。这里我们生成了  $3 \times 3$  和  $4 \times 3$  大小的两个区域。

RoI 池化层的输出大小是  $num\_regions \times num\_channels \times n \times m$ 。它可以当做一个样本个数是  $num\_regions$  的普通批量进入到其他层进行训练。

```
In [2]: rois = nd.array([[0,0,0,2,2], [0,0,1,3,3]])
nd.ROIPooling(x, rois, pooled_size=(2,2), spatial_scale=1)

Out[2]:
[[[ 5.   6.]
  [ 9.  10.]]

 [[ 9.  11.]
  [ 13.  15.]]]]
<NDArray 2x1x2x2 @cpu(0)>
```

### 9.4.3 Faster R-CNN：更快速的区域卷积神经网络

Fast R-CNN 沿用了 R-CNN 的选择性搜索方法来选择区域。这个通常很慢。Faster R-CNN 做的主要改进是提出了区域提议网络（region proposal network, RPN）来替代选择性搜索。它是这么工作的：

1. 在输入特征上放置一个  $1 \times 256 \times 3$  卷积。这样每个像素，连同它的周围 8 个像素，都被映射成一个长为 256 的向量。
2. 以每个像素为中心，生成  $k$  个大小和长宽比都预先设计好的默认边框，通常也叫锚框。
3. 对每个边框，使用其中心像素对应的 256 维向量作为特征，RPN 训练一个 2 类分类器来判断这个区域是不是含有任何感兴趣的物体还是只是背景，和一个 4 维输出的回归器来预测一个更准确的边框。
4. 对于所有的锚框，个数为  $nmk$  如果输入大小是  $n \times m$ ，选出被判断成还有物体的，然后前他们对应的回归器预测的边框作为输入放进接下来的 RoI 池化层



Region Proposal<div>Network

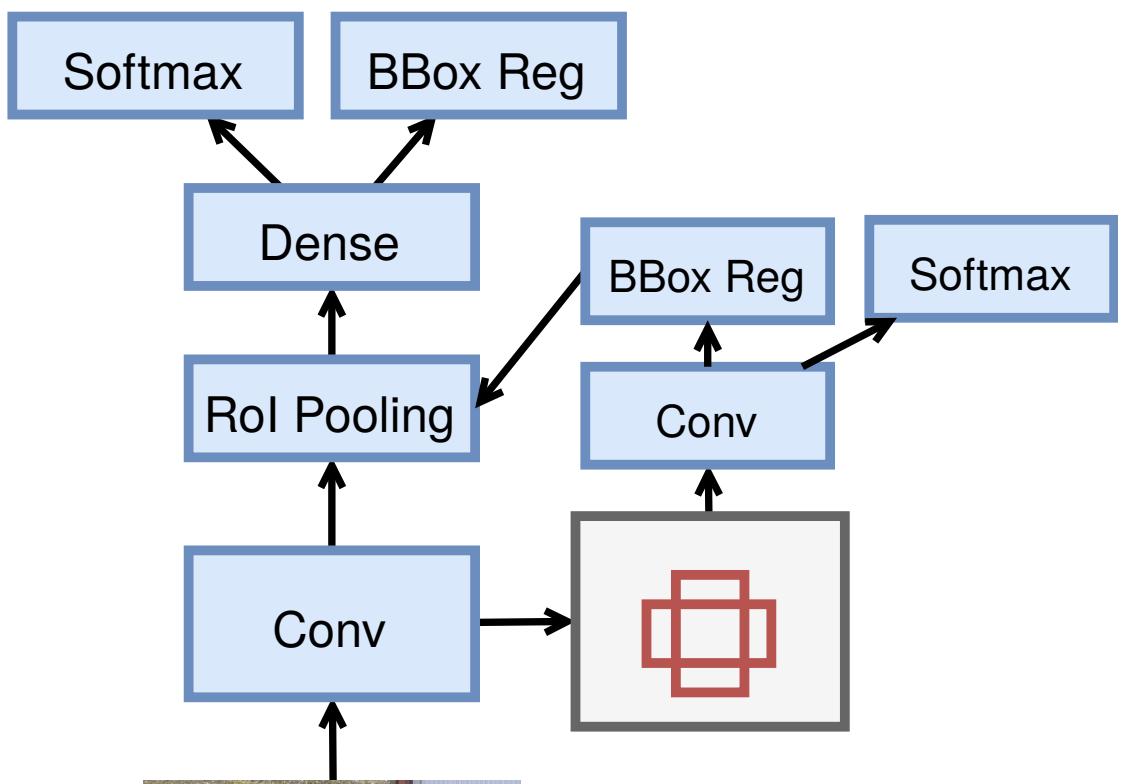


图 9.4: Faster R-CNN

虽然看上有些复杂，但 RPN 思想非常直观。首先提议预先配置好的一些区域，然后通过神经网络来判断这些区域是不是感兴趣的，如果是，那么再预测一个更加准确的边框。这样我们能有效降低搜索任何形状的边框的代价。

#### 9.4.4 Mask R-CNN

Mask R-CNN 在 Faster R-CNN 上加入了一个新的像素级别的预测层，它不仅预测一个锚框对应的类和真实的边框，而且会判断这个锚框内每个像素对应的是物体还是只是背景。后者是语义分割要解决的问题。Mask R-CNN 使用了之后我们将介绍的全连接卷积网络 (FCN) 来完成这个预测。当然这也意味这训练数据必须有像素级别的标注，而不是简单的边框。

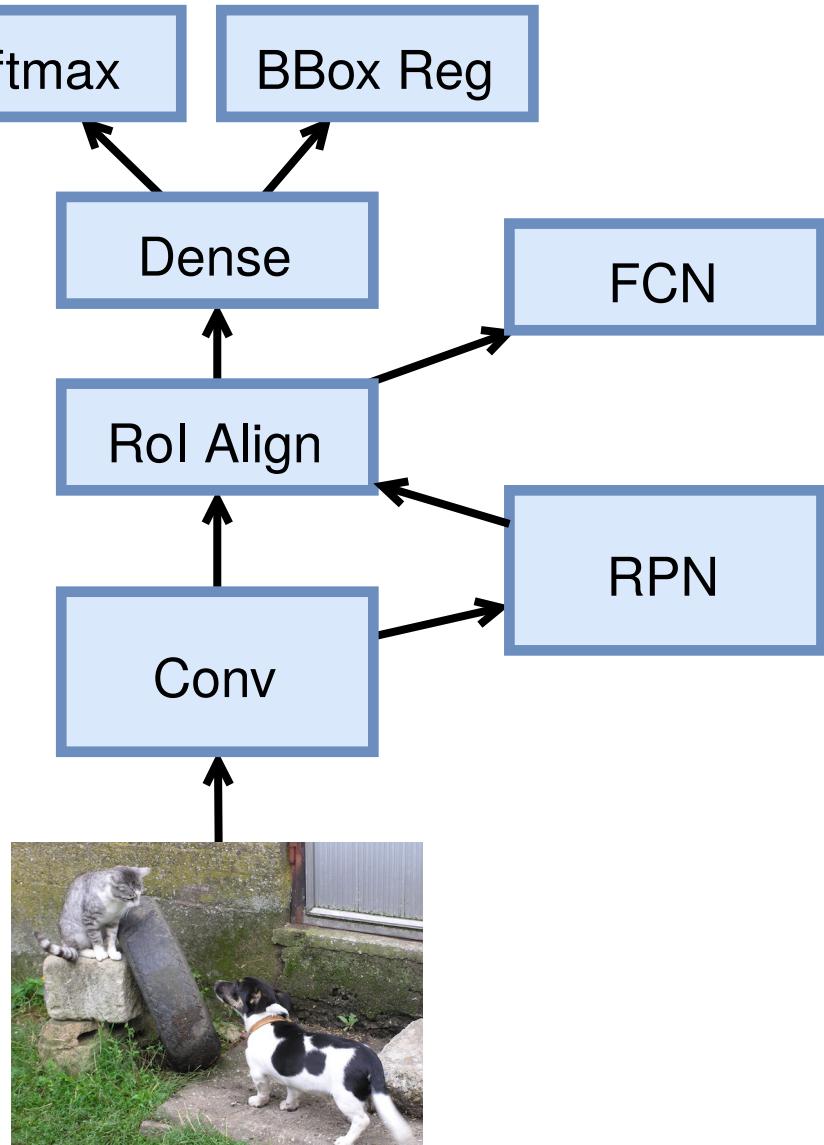


图 9.5: Mask R-CNN

因为 FCN 会精确预测每个像素的类别，就是输入图片中的每个像素都会在标注中对应一个类别。对于输入图片中的一个锚框，我们可以精确的匹配到像素标注中对应的区域。但是 ROI 池化是作用在卷积之后的特征上，其默认是将锚框做了定点化。例如假设选择的锚框是  $(x, y, w, h)$ ，且特征抽取将图片变小了 16 倍，就是如果原始图片是  $256 \times 256$ ，那么特征大小就是  $16 \times 16$ 。这时候

在特征上对应的锚框就是变成了  $(\lfloor x/16 \rfloor, \lfloor y/16 \rfloor, \lfloor w/16 \rfloor, \lfloor h/16 \rfloor)$ 。如果  $x, y, w, h$  中有任何一个不被 16 整除，那么就可能发生错位。同样道理，在上面的样例中我们看到，如果锚框的长宽不被池化大小整除，那么同样会定点化，从而带来错位。

通常这样的错位只是在几个像素之间，对于分类和边框预测影响不大。但对于像素级别的预测，这样的错位可能会带来大问题。Mask R-CNN 提出一个 RoI Align 层，它类似于 RoI 池化层，但是去除掉了定点化步骤，就是移除了所有  $\lfloor \cdot \rfloor$ 。如果计算得到的表框不是刚好在像素之间，那么我们就用四周的像素来线性插值得到这个点上的值。

对于一维情况，假设我们要计算  $x$  点的值  $f(x)$ ，那么我们可以用  $x$  左右的整点的值来插值：

$$f(x) = (\lfloor x \rfloor + 1 - x)f(\lfloor x \rfloor) + (x - \lfloor x \rfloor)f(\lfloor x \rfloor + 1)$$

我们实际要使用的是二维差值来估计  $f(x, y)$ ，我们首先在  $x$  轴上差值得到  $f(x, \lfloor y \rfloor)$  和  $f(x, \lfloor y \rfloor + 1)$ ，然后根据这两个值来差值得到  $f(x, y)$ 。

#### 9.4.5 SSD: 单发多框检测器

在 R-CNN 系列模型里。区域提议和分类是分作两块来进行的。SSD 则将其统一成一个步骤来使得模型更加简单并且速度更快，这也是为什么它被称之为单发的原因。

它跟 Faster R-CNN 主要有两点不一样

1. 对于锚框，我们不再首先判断它是不是含有感兴趣物体，再将正类锚框放入真正物体分类。SSD 里我们直接使用一个 `num_class+1` 类分类器来判断它对应的是哪类物体，还是只是背景。我们也不再有额外的回归器对边框再进一步预测，而是直接使用单个回归器来预测真实边框。
2. SSD 不只是对卷积神经网络输出的特征做预测，它会进一步将特征通过卷积和池化层变小来做预测。这样达到多尺度预测的效果。

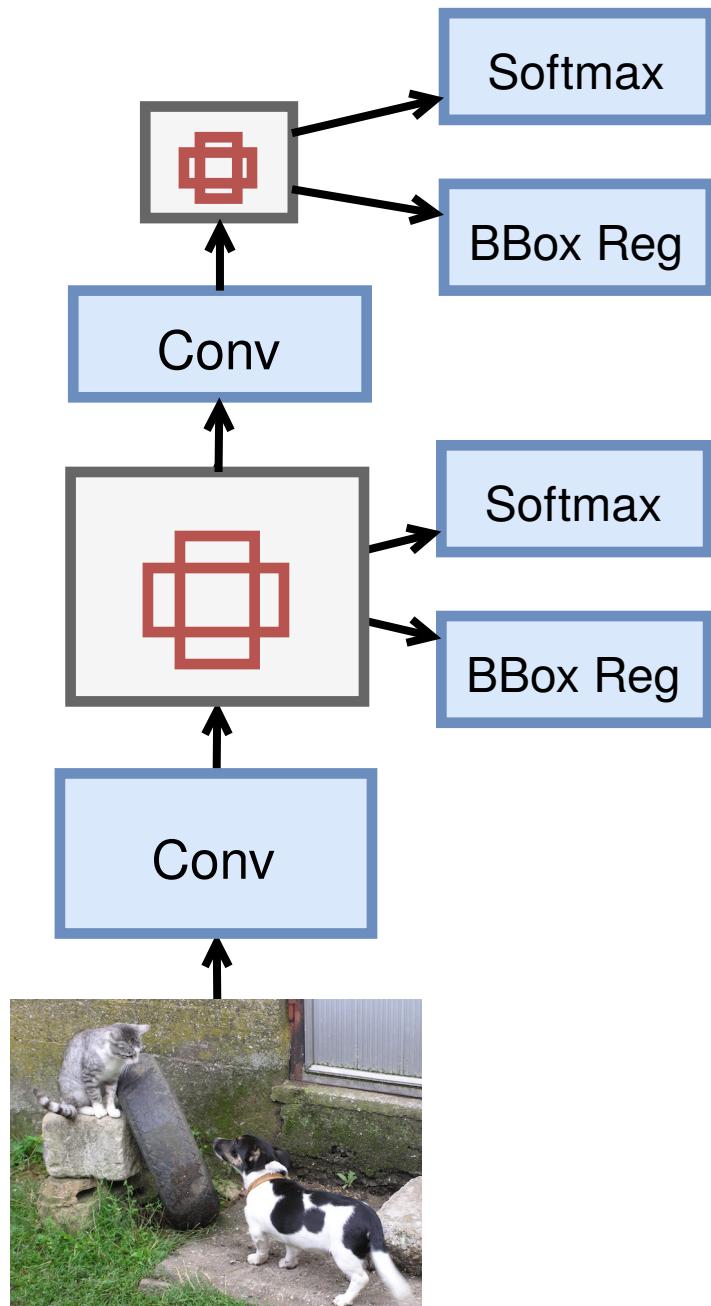


图 9.6: SSD

SSD 的具体实现将在[下一章](#)详细阐述。

#### 9.4.6 YOLO：只需要看一遍

不管是 Faster R-CNN 还是 SSD，它们生成的锚框仍然有大量是相互重叠的，从而导致仍然有大量的区域被重复计算了。YOLO 试图来解决这个问题。它将图片特征均匀的切成  $S \times S$  块，每一块当做一个锚框。每个锚框预测  $B$  个边框，以及这个锚框主要包含哪个物体。

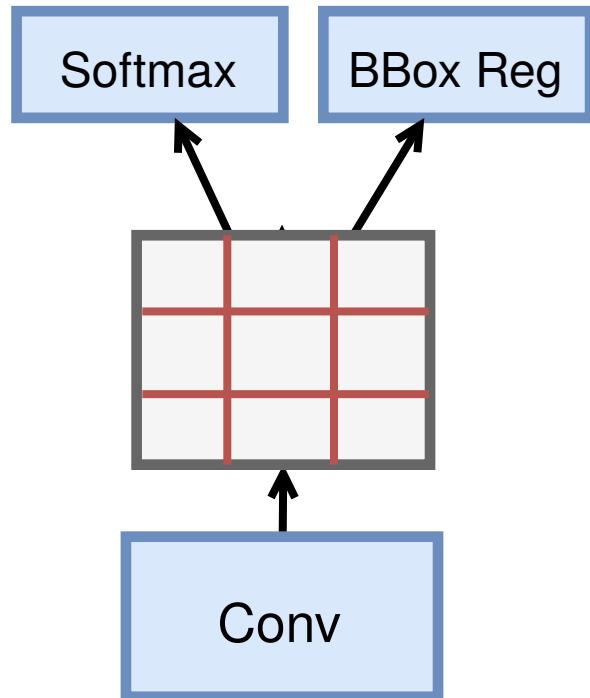
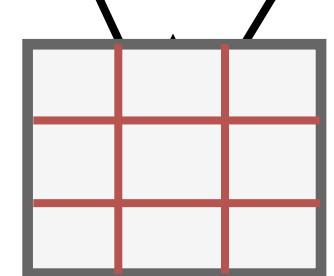


图 9.7: Yolo

#### 9.4.7 YOLO v2：更好，更快，更强

YOLO v2 对 YOLO 进行一些地方的改进，其主要包括：

1. 使用更好的卷积神经网络来做特征提取，使用更大输入图片  $448 \times 448$  使得特征输出大小增大到  $13 \times 13$

- 不再使用均匀切来的锚框，而是对训练数据里的真实锚框做聚类，然后使用聚类中心作为锚框。相对于 SSD 和 Faster R-CNN 来说可以大幅降低锚框的个数。
- 不再使用 YOLO 的全连接层来预测，而是同 SSD 一样使用卷积。例如假设使用 5 个锚框（聚类为 5 类），那么物体分类使用通道数是  $5 * (1 + \text{num\_classes})$  的  $1 \times 1$  卷积，边框回归使用通道数  $4 * 5$ .

#### 9.4.8 小结

- 我们描述了基于卷积神经网络的几个物体检测算法。他们之间的共同点在于首先提出锚框，使用卷积神经网络抽取特征后来预测其包含的主要物体和更准确的边框。但他们在锚框的选择和预测上各有不同，导致他们在计算实际和精度上也各有权衡。

#### 9.4.9 练习

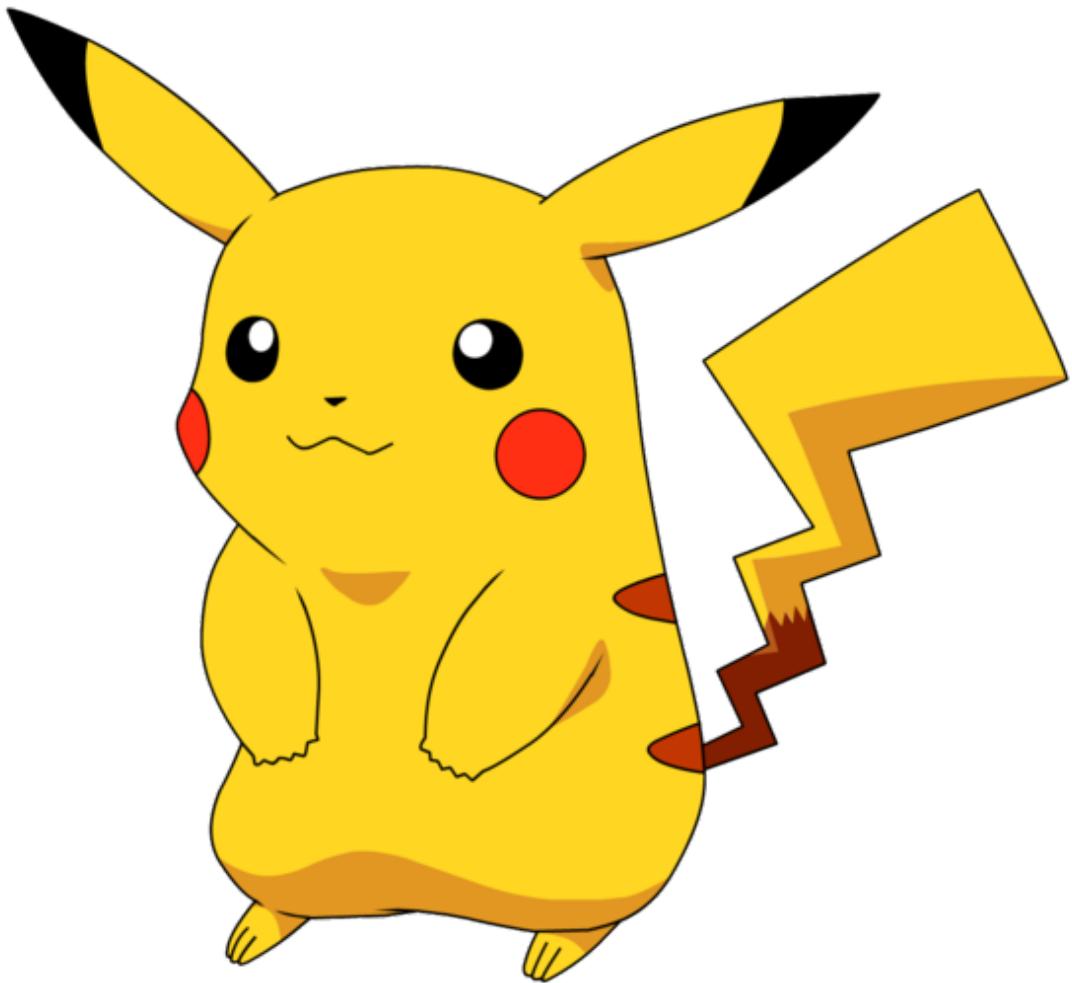
- TODO@mli

#### 9.4.10 扫码直达讨论区



### 9.5 目标检测模型：SSD

这一章下面我们将实现上一章介绍的 SSD 来检测野生皮卡丘。



### 9.5.1 数据集

为此我们合成了一个人工数据集。我们首先使用一个开源的皮卡丘 3D 模型，用其生成 1000 张不同角度和大小的照片。然后将其随机的放在背景图片里。我们将图片打包成 `rec` 文件，这是一个 MXNet 常用的二进制数据格式。我们可以使用 MXNet 下的 `tools/im2rec.py` 来将图片打包。(TODO(@mli) 加一个教程关于如何使用 `im2rec`)。

## 下载数据

打包好的数据可以直接在网上下载：

```
In [1]: from mxnet import gluon

root_url = ('https://apache-mxnet.s3-accelerate.amazonaws.com/'
            'gluon/dataset/pikachu/')
data_dir = '../data/pikachu'
dataset = {'train.rec': 'e6bcb6ffba1ac04ff8a9b1115e650af56ee969c8',
           'train.idx': 'dcf7318b2602c06428b9988470c731621716c393',
           'val.rec': 'd6c33f799b4d058e82f2cb5bd9a976f69d72d520'}
for k, v in dataset.items():
    gluon.utils.download(root_url+k, data_dir+k, sha1_hash=v)
```

## 读取数据集

我们使用 `image.ImageDetIter` 来读取数据。这是针对物体检测的迭代器，(Det 表示 Detection)。它跟 `image.ImageIter` 使用很类似。主要不同是它返回的标号不是单个图片标号，而是每个图片里所有物体的标号，以及其对用的边框。

```
In [2]: from mxnet import image
        from mxnet import nd

data_shape = 256
batch_size = 32
rgb_mean = nd.array([123, 117, 104])

def get_iterators(data_shape, batch_size):
    class_names = ['pikachu']
    num_class = len(class_names)
    train_iter = image.ImageDetIter(
        batch_size=batch_size,
        data_shape=(3, data_shape, data_shape),
        path_imgrec=data_dir+'train.rec',
        path_imgidx=data_dir+'train.idx',
        shuffle=True,
        mean=True,
        rand_crop=1,
        min_object_covered=0.95,
        max_attempts=200)
    val_iter = image.ImageIter(
```

```

batch_size=batch_size,
data_shape=(3, data_shape, data_shape),
path_imgrec=data_dir+'val.rec',
shuffle=False,
mean=True)
return train_iter, val_iter, class_names, num_class

train_data, test_data, class_names, num_class = get_iterators(
    data_shape, batch_size)

```

我们读取一个批量。可以看到标号的形状是 `batch_size x num_object_per_image x 5`。这里数据里每个图片里面只有一个标号。每个标号由长为 5 的数组表示，第一个元素是其对用物体的标号，其中 -1 表示非法物体，仅做填充使用。后面 4 个元素表示边框。

```

In [3]: batch = train_data.next()
print(batch)

DataBatch: data shapes: [(32, 3, 256, 256)] label shapes: [(32, 1, 5)]

```

## 图示数据

我们画出几张图片和其对应的标号。可以看到比卡丘的角度大小位置在每张图图片都不一样。不过也注意到这个数据集是直接将二次元动漫皮卡丘跟三次元背景相结合。可能通过简单判断区域的色彩直方图就可以有效的区别是不是有我们要的物体。我们用这个简单数据集来演示 SSD 是如何工作的。实际中遇到的数据集通常会复杂很多。

```

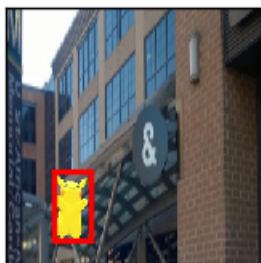
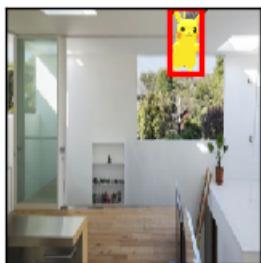
In [4]: %matplotlib inline
import matplotlib as mpl
mpl.rcParams['figure.dpi']= 120
from matplotlib import pyplot as plt

def box_to_rect(box, color, linewidth=3):
    """convert an anchor box to a matplotlib rectangle"""
    box = box.astype(np.int32)
    return plt.Rectangle(
        (box[0], box[1]), box[2]-box[0], box[3]-box[1],
        fill=False, edgecolor=color, linewidth=linewidth)

_, figs = plt.subplots(3, 3, figsize=(6,6))
for i in range(3):
    for j in range(3):
        img, labels = batch.data[0][3*i+j], batch.label[0][3*i+j]
        # (3L, 256L, 256L) => (256L, 256L, 3L)

```

```
img = img.transpose((1, 2, 0)) + rgb_mean
img = img.clip(0,255).asnumpy()/255
fig = figs[i][j]
fig.imshow(img)
for label in labels:
    rect = box_to_rect(label[1:5]*data_shape,'red',2)
    fig.add_patch(rect)
fig.axes.get_xaxis().set_visible(False)
fig.axes.get_yaxis().set_visible(False)
plt.show()
```



## 9.5.2 SSD 模型

### 锚框：默认的边界框

因为边框可以出现在图片中的任何位置，并且可以有任意大小。为了简化计算，SSD 跟 Faster R-CNN 一样使用一些默认的边界框，或者称之为锚框（anchor box），做为搜索起点。具体来说，对输入的每个像素，以其为中心采样数个有不同形状和不同比例的边界框。假设输入大小是  $w \times h$ ，

- 给定大小  $s \in (0, 1]$ ，那么生成的边界框形状是  $ws \times hs$
- 给定比例  $r > 0$ ，那么生成的边界框形状是  $w\sqrt{r} \times \frac{h}{\sqrt{r}}$

在采样的时候我们提供  $n$  个大小（`sizes`）和  $m$  个比例（`ratios`）。为了计算简单这里不生成  $nm$  个锚框，而是  $n + m - 1$  个。其中第  $i$  个锚框使用

- `sizes[i]` 和 `ratios[0]` 如果  $i \leq n$
- `sizes[0]` 和 `ratios[i-n]` 如果  $i > n$

我们可以使用 `contrib.ndarray` 里的 `MultiBoxPrior` 来采样锚框。这里锚框通过左下角和右上角两个点来确定，而且被标准化成了区间  $[0, 1]$  的实数。

```
In [5]: from mxnet import nd
        from mxnet.contrib.ndarray import MultiBoxPrior

        # shape: batch x channel x height x weight
        n = 40
        x = nd.random.uniform(shape=(1, 3, n, n))

        y = MultiBoxPrior(x, sizes=[.5,.25,.1], ratios=[1,2,.5])

        boxes = y.reshape((n, n, -1, 4))
        print(boxes.shape)
        # The first anchor box centered on (20, 20)
        # its format is (x_min, y_min, x_max, y_max)
        boxes[20, 20, 0, :]

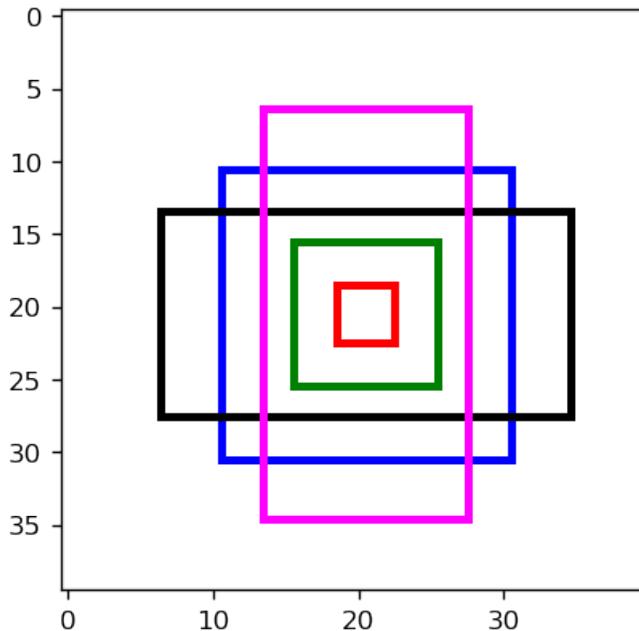
(40, 40, 5, 4)

Out[5]:
[ 0.26249999  0.26249999  0.76249999  0.76249999]
<NDArray 4 @cpu(0)>
```

我们可以画出以  $(20, 20)$  为中心的所有锚框：

```
In [6]: colors = ['blue', 'green', 'red', 'black', 'magenta']

plt.imshow(nd.ones((n, n, 3)).asnumpy())
anchors = boxes[20, 20, :, :]
for i in range(anchors.shape[0]):
    plt.gca().add_patch(box_to_rect(anchors[i,:]*n, colors[i]))
plt.show()
```



## 预测物体类别

对每一个锚框我们需要预测它是不是包含了我们感兴趣的物体，还是只是背景。这里我们使用一个  $3 \times 3$  的卷积层来做预测，加上 `pad=1` 使用它的输出和输入一样。同时输出的通道数是 `num_anchors*(num_classes+1)`，每个通道对应一个锚框对某个类的置信度。假设输出是  $Y$ ，那么对应输入中第  $n$  个样本的第  $(i, j)$  像素的置信值是在  $Y[n, :, i, j]$  里。具体来说，对于以  $(i, j)$  为中心的第  $a$  个锚框，

- 通道  $a * (\text{num\_class} + 1)$  是其只包含背景的分数
- 通道  $a * (\text{num\_class} + 1) + 1 + b$  是其包含第  $b$  个物体的分数

我们定义一个这样的类别分类器函数：

```
In [7]: from mxnet.gluon import nn
def class_predictor(num_anchors, num_classes):
    """return a layer to predict classes"""
    return nn.Conv2D(num_anchors * (num_classes + 1), 3, padding=1)

cls_pred = class_predictor(5, 10)
cls_pred.initialize()
x = nd.zeros((2, 3, 20, 20))
y = cls_pred(x)
y.shape
```

```
Out[7]: (2, 55, 20, 20)
```

## 预测边界框

因为真实的边界框可以是任意形状，我们需要预测如何从一个锚框转换成真正的边界框。这个变换可以由一个长为 4 的向量来描述。同上一样，我们用一个有 `num_anchors * 4` 通道的卷积。假设输出是 Y，那么对应输入中第  $n$  个样本的第  $(i, j)$  像素为中心的锚框的转换在  $Y[n, :, i, j]$  里。具体来说，对于第  $a$  个锚框，它的变换在  $a \times 4$  到  $a \times 4 + 3$  通道里。

```
In [8]: def box_predictor(num_anchors):
    """return a layer to predict delta locations"""
    return nn.Conv2D(num_anchors * 4, 3, padding=1)

box_pred = box_predictor(10)
box_pred.initialize()
x = nd.zeros((2, 3, 20, 20))
y = box_pred(x)
y.shape
```

```
Out[8]: (2, 40, 20, 20)
```

## 减半模块

我们定义一个卷积块，它将输入特征的长宽减半，以此来获取多尺度的预测。它由两个 Conv-BatchNorm-Relu 组成，我们使用填充为 1 的  $3 \times 3$  卷积使得输入和输出有同样的长宽，然后再通过跨度为 2 的最大池化层将长宽减半。

```
In [9]: def down_sample(num_filters):
    """stack two Conv-BatchNorm-Relu blocks and then a pooling layer
    to halve the feature size"""
    out = nn.HybridSequential()
```

```
for _ in range(2):
    out.add(nn.Conv2D(num_filters, 3, strides=1, padding=1))
    out.add(nn.BatchNorm(in_channels=num_filters))
    out.add(nn.Activation('relu'))
out.add(nn.MaxPool2D(2))
return out

blk = down_sample(10)
blk.initialize()
x = nd.zeros((2, 3, 20, 20))
y = blk(x)
y.shape

Out[9]: (2, 10, 10, 10)
```

## 合并来自不同层的预测输出

前面我们提到过 SSD 的一个重要性质是它会在多个层同时做预测。每个层由于长宽和锚框选择不一样，导致输出的数据形状会不一样。这里我们用物体类别预测作为样例，边框预测是类似的。我们首先创建一个特定大小的输入，然后对它输出类别预测。然后对输入减半，再输出类别预测。

```
In [10]: x = nd.zeros((2, 8, 20, 20))
print('x:', x.shape)

cls_pred1 = class_predictor(5, 10)
cls_pred1.initialize()
y1 = cls_pred1(x)
print('Class prediction 1:', y1.shape)

ds = down_sample(16)
ds.initialize()
x = ds(x)
print('x:', x.shape)

cls_pred2 = class_predictor(3, 10)
cls_pred2.initialize()
y2 = cls_pred2(x)
print('Class prediction 2:', y2.shape)

x: (2, 8, 20, 20)
Class prediction 1: (2, 55, 20, 20)
x: (2, 16, 10, 10)
Class prediction 2: (2, 33, 10, 10)
```

可以看到  $y_1$  和  $y_2$  形状不同。为了之后处理简单，我们将不同层的输入合并成一个输出。首先我们将通道移到最后的维度，然后将其展成 2D 数组。因为第一个维度是样本个数，所以不同输出之间是不变，我们可以将所有输出在第二个维度上拼接起来。

```
In [11]: def flatten_prediction(pred):
    return pred.transpose(axes=(0,2,3,1)).flatten()

def concat_predictions(preds):
    return nd.concat(*preds, dim=1)

flat_y1 = flatten_prediction(y1)
print('Flatten class prediction 1', flat_y1.shape)
flat_y2 = flatten_prediction(y2)
print('Flatten class prediction 2', flat_y2.shape)
y = concat_predictions([flat_y1, flat_y2])
print('Concat class predictions', y.shape)

Flatten class prediction 1 (2, 22000)
Flatten class prediction 2 (2, 3300)
Concat class predictions (2, 25300)
```

## 主体网络

主体网络用来从原始像素抽取特征。通常前面介绍的用来图片分类的卷积神经网络，例如 ResNet，都可以用来作为主体网络。这里为了示范，我们简单叠加几个减半模块作为主体网络。

```
In [12]: def body():
    out = nn.HybridSequential()
    for nfilters in [16, 32, 64]:
        out.add(down_sample(nfilters))
    return out

bnet = body()
bnet.initialize()
x = nd.random.uniform(shape=(2,3,256,256))
y = bnet(x)
y.shape

Out[12]: (2, 64, 32, 32)
```

## 创建一个玩具 SSD 模型

现在我们可以创建一个玩具 SSD 模型了。我们称之为玩具是因为这个网络不管是层数还是锚框个数都比较小，仅仅适合之后我们之后使用的一个小数据集。但这个模型不会影响我们介绍 SSD。这个网络包含四块。主体网络，三个减半模块，以及五个物体类别和边框预测模块。其中预测分别应用在在主体网络输出，减半模块输出，和最后的全局池化层上。

```
In [13]: def toy_ssd_model(num_anchors, num_classes):
    downsamplers = nn.Sequential()
    for _ in range(3):
        downsamplers.add(down_sample(128))

    class_predictors = nn.Sequential()
    box_predictors = nn.Sequential()
    for _ in range(5):
        class_predictors.add(class_predictor(num_anchors, num_classes))
        box_predictors.add(box_predictor(num_anchors))

    model = nn.Sequential()
    model.add(body(), downsamplers, class_predictors, box_predictors)
    return model
```

## 计算预测

给定模型和每层预测输出使用的锚框大小和形状，我们可以定义前向函数。

```
In [14]: def toy_ssd_forward(x, model, sizes, ratios, verbose=False):
    body, downsamplers, class_predictors, box_predictors = model
    anchors, class_preds, box_preds = [], [], []
    # feature extraction
    x = body(x)
    for i in range(5):
        # predict
        anchors.append(MultiBoxPrior(
            x, sizes=sizes[i], ratios=ratios[i]))
        class_preds.append(
            flatten_prediction(class_predictors[i](x)))
        box_preds.append(
            flatten_prediction(box_predictors[i](x)))
    if verbose:
        print('Predict scale', i, x.shape, 'with',
              anchors[-1].shape[1], 'anchors')
```

```

# down sample
if i < 3:
    x = downsamplers[i](x)
elif i == 3:
    x = nd.Pooling(
        x, global_pool=True, pool_type='max',
        kernel=(x.shape[2], x.shape[3]))
# concat data
return (concat_predictions(anchors),
        concat_predictions(class_preds),
        concat_predictions(box_preds))

```

## 完整的模型

```

In [15]: from mxnet import gluon
class ToySSD(gluon.Block):
    def __init__(self, num_classes, verbose=False, **kwargs):
        super(ToySSD, self).__init__(**kwargs)
        # anchor box sizes and ratios for 5 feature scales
        self.sizes = [[.2,.272], [.37,.447], [.54,.619],
                      [.71,.79], [.88,.961]]
        self.ratios = [[1,2,.5]]*5
        self.num_classes = num_classes
        self.verbose = verbose
        num_anchors = len(self.sizes[0]) + len(self.ratios[0]) - 1
        # use name_scope to guard the names
        with self.name_scope():
            self.model = toy_ssd_model(num_anchors, num_classes)

    def forward(self, x):
        anchors, class_preds, box_preds = toy_ssd_forward(
            x, self.model, self.sizes, self.ratios,
            verbose=self.verbose)
        # it is better to have class predictions reshaped for softmax
        ← computation
        class_preds = class_preds.reshape(shape=(0, -1, self.num_classes+1))
        return anchors, class_preds, box_preds

```

我们看一下输入图片的形状是如何改变的，已经输出的形状。

```

In [16]: net = ToySSD(num_classes=2, verbose=True)
net.initialize()

```

```
x = batch.data[0][0:1]
print('Input:', x.shape)
anchors, class_preds, box_preds = net(x)
print('Output anchors:', anchors.shape)
print('Output class predictions:', class_preds.shape)
print('Output box predictions:', box_preds.shape)

Input: (1, 3, 256, 256)
Predict scale 0 (1, 64, 32, 32) with 4096 anchors
Predict scale 1 (1, 128, 16, 16) with 1024 anchors
Predict scale 2 (1, 128, 8, 8) with 256 anchors
Predict scale 3 (1, 128, 4, 4) with 64 anchors
Predict scale 4 (1, 128, 1, 1) with 4 anchors
Output anchors: (1, 5444, 4)
Output class predictions: (1, 5444, 3)
Output box predictions: (1, 21776)
```

### 9.5.3 训练

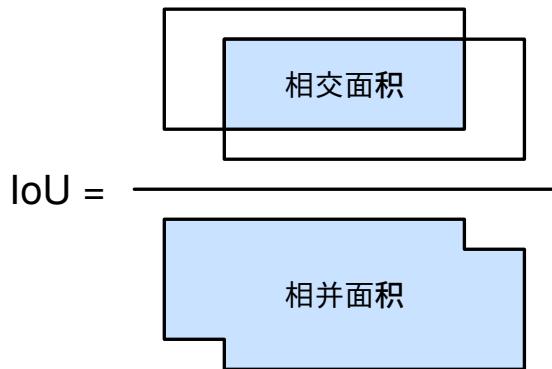
之前的教程我们主要是关注分类。对于分类的预测结果和真实的标号，我们通过交叉熵来计算他们的差异。但物体检测里我们需要预测边框。这里我们先引入一个概率来描述两个边框的距离。

#### IoU：交集除并集

我们知道判断两个集合的相似度最常用的衡量叫做 Jaccard 距离，给定集合  $A$  和  $B$ ，它的定义是

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

边框可以看成是像素的集合，我们可以类似的定义它。这个标准通常被称之为 Intersection over Union (IoU)。



大的值表示两个边框很相似，而小的值则表示不相似。

## 损失函数

虽然每张图片里面通常只有几个标注的边框，但 SSD 会生成大量的锚框。可以想象很多锚框都不会框住感兴趣的物体，就是说跟任何对应感兴趣物体的表框的 IoU 都小于某个阈值。这样就会产生大量的负类锚框，或者说对应标号为 0 的锚框。对于这类锚框有两点要考虑的：

1. 边框预测的损失函数不应该包括负类锚框，因为它们并没有对应的真实边框
2. 因为负类锚框数目可能远多于其他，我们可以只保留其中的一些。而且是保留那些目前预测最不确定它是负类的，就是对类 0 预测值排序，选取数值最小的哪一些困难的负类锚框。

我们可以使用 `MultiBoxTarget` 来完成上面这两个操作。

```
In [17]: from mxnet.contrib.ndarray import MultiBoxTarget
def training_targets(anchors, class_preds, labels):
    class_preds = class_preds.transpose(axes=(0, 2, 1))
    return MultiBoxTarget(anchors, labels, class_preds)

out = training_targets(anchors, class_preds, batch.label[0][0:1])
out
```

Out[17]: [

```
[[ 0.  0.  0. ...,  0.  0.  0.]]
<NDArray 1x21776 @cpu(0)>,
[[ 0.  0.  0. ...,  0.  0.  0.]]
<NDArray 1x21776 @cpu(0)>,
```

```
[[ 0.  0.  0. ...,  0.  0.  0.]]  
<NDArray 1x5444 @cpu(0)>]
```

它返回三个 NDArray，分别是

1. 预测的边框跟真实边框的偏移，大小是 batch\_size x (num\_anchors\*4)
2. 用来遮掩不需要的负类锚框的掩码，大小跟上面一致
3. 锚框的真实的标号，大小是 batch\_size x num\_anchors

我们可以计算这次只选中了多少个锚框进入损失函数：

```
In [18]: out[1].sum()/4
```

```
Out[18]:
```

```
[ 16.]  
<NDArray 1 @cpu(0)>
```

然后我们可以定义需要的损失函数了。

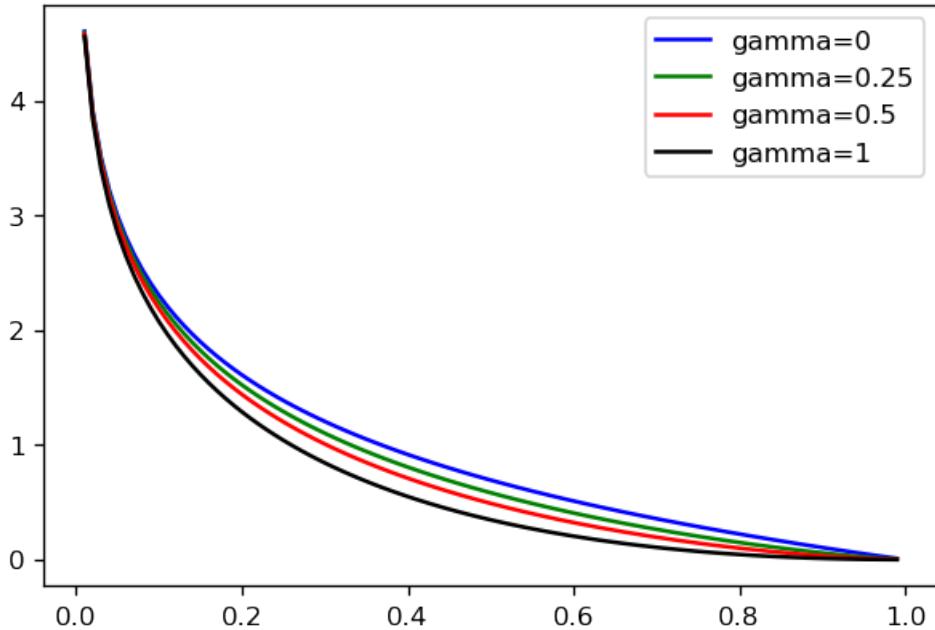
对于分类问题，最常用的损失函数是之前一直使用的交叉熵。这里我们定义一个类似于交叉熵的损失，不同于交叉熵的定义  $\log(p_j)$ ，这里  $j$  是真实的类别，且  $p_j$  是对于的预测概率。我们使用一个被称之为关注损失的函数，给定正的  $\gamma$  和  $\alpha$ ，它的定义是

$$-\alpha(1 - p_j)^\gamma \log(p_j)$$

下图我们演示不同  $\gamma$  导致的变化。可以看到，增加  $\gamma$  可以使得对正类预测值比较大时损失变小。

```
In [19]: import numpy as np
```

```
def focal_loss(gamma, x):  
    return - (1-x)**gamma*np.log(x)  
  
x = np.arange(0.01, 1, .01)  
gammas = [.0,.25,.5,1]  
for i,g in enumerate(gammas):  
    plt.plot(x, focal_loss(g,x), colors[i])  
  
plt.legend(['gamma=' + str(g) for g in gammas])  
plt.show()
```



这个自定义的损失函数可以简单通过继承 `gluon.loss.Loss` 来实现。

```
In [20]: class FocalLoss(gluon.loss.Loss):
    def __init__(self, axis=-1, alpha=0.25, gamma=2, batch_axis=0, **kwargs):
        super(FocalLoss, self).__init__(None, batch_axis, **kwargs)
        self._axis = axis
        self._alpha = alpha
        self._gamma = gamma

    def hybrid_forward(self, F, output, label):
        output = F.softmax(output)
        pj = output.pick(label, axis=self._axis, keepdims=True)
        loss = - self._alpha * ((1 - pj) ** self._gamma) * pj.log()
        return loss.mean(axis=self._batch_axis, exclude=True)

cls_loss = FocalLoss()
cls_loss
```

Out[20]: `FocalLoss(batch_axis=0, w=None)`

对于边框的预测是一个回归问题。通常可以选择平方损失函数（L2 损失） $f(x) = x^2$ 。但这个损失对于比较大的误差的惩罚很高。我们可以采用稍微缓和一点绝对损失函数（L1 损失） $f(x) = |x|$ ，它是随着误差线性增长，而不是平方增长。但这个函数在 0 点处导数不唯一，因此可能会影响收敛。

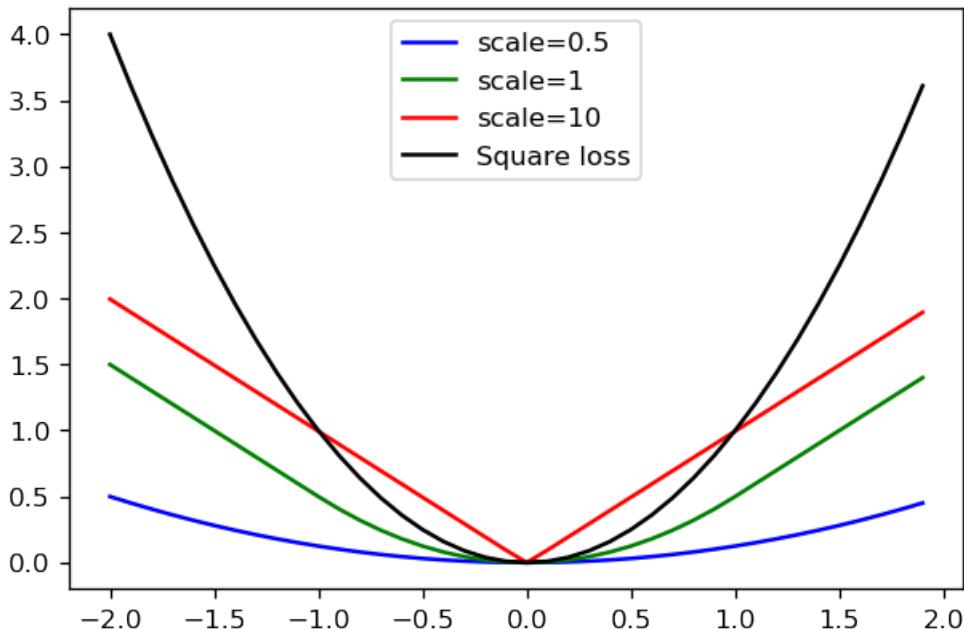
敛。一个通常的解决办法是在 0 点附近使用平方函数使得它更加平滑。它被称之为平滑 L1 损失函数。它通过一个参数  $\sigma$  来控制平滑的区域：

$$f(x) = \begin{cases} (\sigma x)^2 / 2, & \text{if } x < 1/\sigma^2 \\ |x| - 0.5/\sigma^2, & \text{otherwise} \end{cases}$$

我们图示不同的  $\sigma$  的平滑 L1 损失和 L2 损失的区别。

```
In [21]: scales = [.5, 1, 10]
x = nd.arange(-2, 2, 0.1)

for i,s in enumerate(scales):
    y = nd.smooth_l1(x, scalar=s)
    plt.plot(x.asnumpy(), y.asnumpy(), color=colors[i])
plt.plot(x.asnumpy(), (x**2).asnumpy(), color=colors[len(scales)])
plt.legend(['scale=' + str(s) for s in scales] + ['Square loss'])
plt.show()
```



我们同样通过继承 `Loss` 来定义这个损失。同时它接受一个额外参数 `mask`，这是用来屏蔽掉不需要被惩罚的负例样本。

```
In [22]: class SmoothL1Loss(gluon.loss.Loss):
    def __init__(self, batch_axis=0, **kwargs):
```

```
super(SmoothL1Loss, self).__init__(None, batch_axis, **kwargs)

def hybrid_forward(self, F, output, label, mask):
    loss = F.smooth_l1((output - label) * mask, scalar=1.0)
    return loss.mean(self._batch_axis, exclude=True)

box_loss = SmoothL1Loss()
box_loss

Out[22]: SmoothL1Loss(batch_axis=0, w=None)
```

## 评估测量

对于分类好坏我们可以沿用之前的分类精度。评估边框预测的好坏的一个常用是是平均绝对误差。记得在线性回归我们使用了平均平方误差。但跟上面对损失函数的讨论一样，平方误差对于大的误差给予过大的值，从而数值上过于敏感。平均绝对误差就是将二次项替换成绝对值，具体来说就是预测的边框和真实边框在 4 个维度上的差值的绝对值。

```
In [23]: from mxnet import metric

cls_metric = metric.Accuracy()
box_metric = metric.MAE()
```

## 初始化模型和训练器

```
In [24]: from mxnet import init
from mxnet import gpu

ctx = gpu(0)
# the CUDA implementation requires each image has at least 3 labels.
# Pad two -1 labels for each instance
train_data.reshape(label_shape=(3, 5))
train_data = test_data.sync_label_shape(train_data)

net = ToySSD(num_class)
net.initialize(init.Xavier(magnitude=2), ctx=ctx)
trainer = gluon.Trainer(net.collect_params(),
                        'sgd', {'learning_rate': 0.1, 'wd': 5e-4})
```

## 训练模型

训练函数跟前面的不一样在于网络会有多个输出，而且有两个损失函数。

```
In [25]: import time
from mxnet import autograd
for epoch in range(30):
    # reset data iterators and metrics
    train_data.reset()
    cls_metric.reset()
    box_metric.reset()
    tic = time.time()
    for i, batch in enumerate(train_data):
        x = batch.data[0].as_in_context(ctx)
        y = batch.label[0].as_in_context(ctx)
        with autograd.record():
            anchors, class_preds, box_preds = net(x)
            box_target, box_mask, cls_target = training_targets(
                anchors, class_preds, y)
            # losses
            loss1 = cls_loss(class_preds, cls_target)
            loss2 = box_loss(box_preds, box_target, box_mask)
            loss = loss1 + loss2
            loss.backward()
            trainer.step(batch_size)
            # update metrics
            cls_metric.update([cls_target], [class_preds.transpose((0,2,1))])
            box_metric.update([box_target], [box_preds * box_mask])

    print('Epoch %2d, train %s %.2f, %s %.5f, time %.1f sec' %
          epoch, *cls_metric.get(), *box_metric.get(), time.time()-tic
          ))
```

Epoch 0, train accuracy 0.95, mae 0.00408, time 13.0 sec  
Epoch 1, train accuracy 0.99, mae 0.00353, time 10.5 sec  
Epoch 2, train accuracy 0.99, mae 0.00339, time 10.5 sec  
Epoch 3, train accuracy 0.99, mae 0.00321, time 10.5 sec  
Epoch 4, train accuracy 0.99, mae 0.00316, time 10.5 sec  
Epoch 5, train accuracy 1.00, mae 0.00324, time 10.5 sec  
Epoch 6, train accuracy 1.00, mae 0.00306, time 10.5 sec  
Epoch 7, train accuracy 1.00, mae 0.00297, time 10.5 sec  
Epoch 8, train accuracy 1.00, mae 0.00314, time 10.5 sec  
Epoch 9, train accuracy 1.00, mae 0.00300, time 10.5 sec  
Epoch 10, train accuracy 1.00, mae 0.00301, time 10.5 sec

```
Epoch 11, train accuracy 1.00, mae 0.00310, time 10.4 sec
Epoch 12, train accuracy 1.00, mae 0.00296, time 10.5 sec
Epoch 13, train accuracy 1.00, mae 0.00305, time 10.4 sec
Epoch 14, train accuracy 1.00, mae 0.00291, time 10.5 sec
Epoch 15, train accuracy 1.00, mae 0.00282, time 10.5 sec
Epoch 16, train accuracy 1.00, mae 0.00285, time 10.5 sec
Epoch 17, train accuracy 1.00, mae 0.00274, time 10.5 sec
Epoch 18, train accuracy 1.00, mae 0.00287, time 10.4 sec
Epoch 19, train accuracy 1.00, mae 0.00276, time 10.5 sec
Epoch 20, train accuracy 1.00, mae 0.00284, time 10.4 sec
Epoch 21, train accuracy 1.00, mae 0.00265, time 10.5 sec
Epoch 22, train accuracy 1.00, mae 0.00267, time 10.5 sec
Epoch 23, train accuracy 1.00, mae 0.00280, time 10.5 sec
Epoch 24, train accuracy 1.00, mae 0.00259, time 10.5 sec
Epoch 25, train accuracy 1.00, mae 0.00259, time 10.5 sec
Epoch 26, train accuracy 1.00, mae 0.00268, time 10.5 sec
Epoch 27, train accuracy 1.00, mae 0.00267, time 10.5 sec
Epoch 28, train accuracy 1.00, mae 0.00255, time 10.5 sec
Epoch 29, train accuracy 1.00, mae 0.00268, time 10.4 sec
```

## 9.5.4 预测

在预测阶段，我们希望能把图片里面所有感兴趣的物体找出来。

我们先定一个数据读取和预处理函数。

```
In [26]: def process_image(fname):
    with open(fname, 'rb') as f:
        im = image.imread(f.read())
    # resize to data_shape
    data = image.imresize(im, data_shape, data_shape)
    # minus rgb mean
    data = data.astype('float32') - rgb_mean
    # convert to batch x channel x height xwidth
    return data.transpose((2,0,1)).expand_dims(axis=0), im
```

然后我们跟训练那样预测表框和其对应的物体。但注意到因为我们要对每个像素都会生成数个锚框，这样我们可能会预测出大量相似的表框，从而导致结果非常嘈杂。一个办法是对于 IoU 比较高的两个表框，我们只保留预测执行度比较高的那个。这个算法（称之为 non maximum suppression）在 `MultiBoxDetection` 里实现了。下面我们实现预测函数：

```
In [27]: from mxnet.contrib.ndarray import MultiBoxDetection
```

```

def predict(x):
    anchors, cls_preds, box_preds = net(x.as_in_context(ctx))
    cls_probs = nd.SoftmaxActivation(
        cls_preds.transpose((0,2,1)), mode='channel')

    return MultiBoxDetection(cls_probs, box_preds, anchors,
                             force_suppress=True, clip=False)

```

预测函数会输出所有边框，每个边框由 [class\_id, confidence, xmin, ymin, xmax, ymax] 表示。其中 class\_id=-1 表示要么这个边框被预测只含有背景，或者被去重掉了。

```
In [28]: x, im = process_image('../img/pikachu.jpg')
out = predict(x)
out.shape
```

```
Out[28]: (1, 5444, 6)
```

最后我们将预测出置信度超过某个阈值的边框画出来：

```

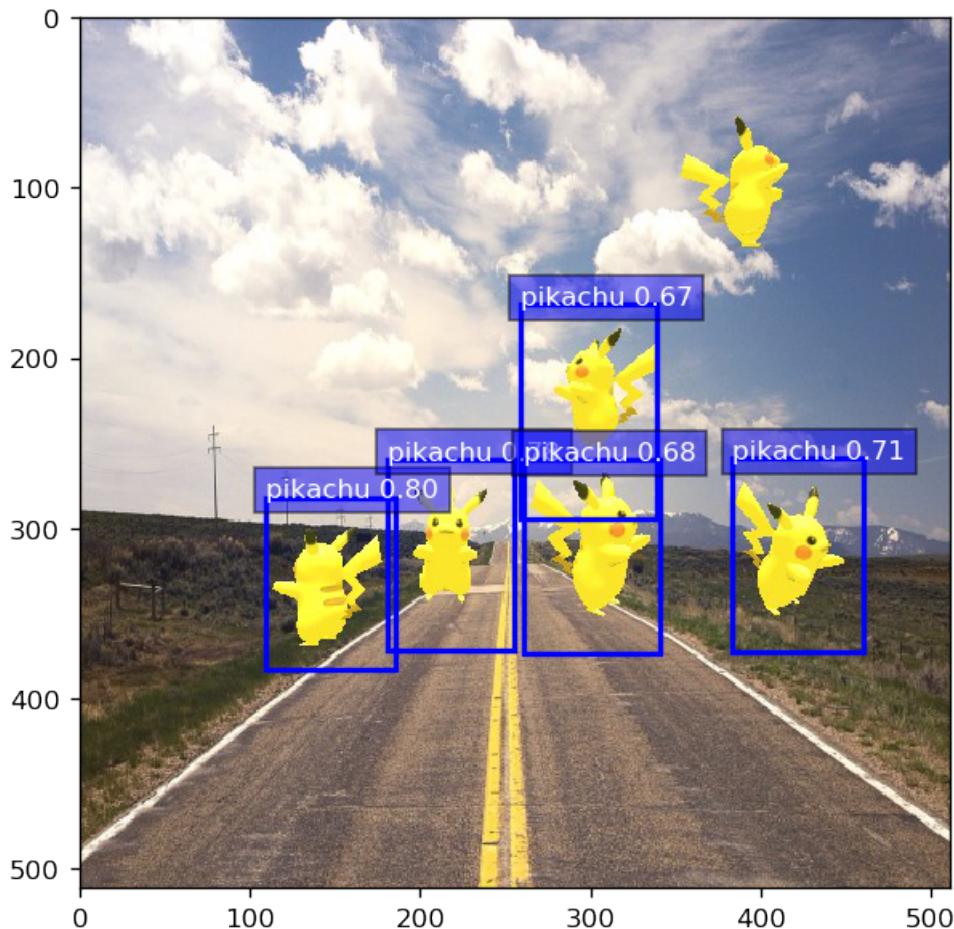
In [29]: mpl.rcParams['figure.figsize'] = (6,6)

def display(im, out, threshold=0.5):
    plt.imshow(im.asnumpy())
    for row in out:
        row = row.asnumpy()
        class_id, score = int(row[0]), row[1]
        if class_id < 0 or score < threshold:
            continue
        color = colors[class_id%len(colors)]
        box = row[2:6] * np.array([im.shape[0],im.shape[1]]*2)
        rect = box_to_rect(nd.array(box), color, 2)
        plt.gca().add_patch(rect)

        text = class_names[class_id]
        plt.gca().text(box[0], box[1],
                      '{:s} {:.2f}'.format(text, score),
                      bbox=dict(facecolor=color, alpha=0.5),
                      fontsize=10, color='white')
    plt.show()

display(im, out[0], threshold=0.5)

```



### 9.5.5 小结

- 物体检测比分类要困难很多。因为我们不仅要预测物体类别，还要找到它们的位置。这一章我们展示我们还是可以在合理篇幅里实现 SSD 算法。

### 9.5.6 练习

我们有很多细节并没有展开讨论。例如

- 锚框的大小和长宽比是如何选取的

- MultiBoxTarget 里我们没有采样负例
- 分类和回归损失我们直接加起来了，并没有给予权重
- 在展示的时候如何选取阈值 threshold

### 9.5.7 扫码直达讨论区



## 9.6 目标检测模型：YOLO

接着上一讲的 SSD，我们继续来实现一个目标检测的经典算法-YOLO2。YOLO2 是继 YOLO 之后使用纯卷积网络的作品。这里不再赘述数据集之类的琐事，直接进入算法实现的部分，推荐不熟悉的小伙伴先用 SSD 的例子热身一下。

```
In [1]: import numpy as np
import mxnet as mx
from mxnet import gluon
from mxnet import nd
from mxnet.gluon import nn
from mxnet.gluon import Block, HybridBlock
from mxnet.gluon.model_zoo import vision
```

### 9.6.1 YOLO v2

#### 原始卷积输出的转换

我们知道原始卷积输出的是一个  $(B, N, H, W)$  的矩阵，其中  $B$  是 batch-size， $H$  和  $W$  是特征层的空间维度， $N$  是卷积的输出通道数，相对比较复杂，是我们关注的维度。简单地说，我们需要对每个空间维度上的点  $(x, y)$  输出（类别数 + 1 + 4）个预测值。类别数很好理解，就是训练集里正类的数量，4 也很好理解，就是每个预测框的偏移量预测  $(x, y, w, h)$

## 中心点的转换

中心点是 sigmoid 函数的输出值，本身在 0 到 1 之间，表示的意义是每个格点内部的空间位置，我们需要把它们转换成图片上的相对位置。通过 arange 函数生成连续递增的数列，加上预测值，就是每个目标中心在图片上的相对坐标。

```
In [2]: def transform_center(xy):
    """Given x, y prediction after sigmoid(), convert to relative coordinates
    (0, 1) on image."""
    b, h, w, n, s = xy.shape
    offset_y = nd.tile(nd.arange(0, h, repeat=(w * n * 1),
    ctx=xy.context).reshape((1, h, w, n, 1)), (b, 1, 1, 1, 1))
    # print(offset_y[0].asnumpy()[:, :, 0, 0])
    offset_x = nd.tile(nd.arange(0, w, repeat=(n * 1),
    ctx=xy.context).reshape((1, 1, w, n, 1)), (b, h, 1, 1, 1))
    # print(offset_x[0].asnumpy()[:, :, 0, 0])
    x, y = xy.split(num_outputs=2, axis=-1)
    x = (x + offset_x) / w
    y = (y + offset_y) / h
    return x, y
```

## 长宽的转换

长宽是 exp 函数的输出，意义是相对于锚点长宽的比率，我们对预测值的 exp() 乘以相对锚点的长宽，除以图片格点的数量，得到的是目标长宽相对于图片的尺寸。

```
In [3]: def transform_size(wh, anchors):
    """Given w, h prediction after exp() and anchor sizes, convert to relative
    width/height (0, 1) on image"""
    b, h, w, n, s = wh.shape
    aw, ah = nd.tile(nd.array(anchors, ctx=wh.context).reshape((1, 1, 1, -1,
    2)), (b, h, w, 1, 1)).split(num_outputs=2, axis=-1)
    w_pred, h_pred = nd.exp(wh).split(num_outputs=2, axis=-1)
    w_out = w_pred * aw / w
    h_out = h_pred * ah / h
    return w_out, h_out
```

**yolo2\_forward** 作为一个方便使用的函数，会把卷积的通道分开，转换，最后转成我们需要的检测框

```
In [4]: def yolo2_forward(x, num_class, anchor_scales):
    """Transpose/reshape/organize convolution outputs."""
    stride = num_class + 5
    # transpose and reshape, 4th dim is the number of anchors
    x = x.transpose((0, 2, 3, 1))
    x = x.reshape((0, 0, 0, -1, stride))
    # now x is (batch, m, n, stride), stride = num_class + 1(object score) +
    ↵ 4(coordinates)
    # class probs
    cls_pred = x.slice_axis(begin=0, end=num_class, axis=-1)
    # object score
    score_pred = x.slice_axis(begin=num_class, end=num_class + 1, axis=-1)
    score = nd.sigmoid(score_pred)
    # center prediction, in range(0, 1) for each grid
    xy_pred = x.slice_axis(begin=num_class + 1, end=num_class + 3, axis=-1)
    xy = nd.sigmoid(xy_pred)
    # width/height prediction
    wh = x.slice_axis(begin=num_class + 3, end=num_class + 5, axis=-1)
    # convert x, y to positions relative to image
    x, y = transform_center(xy)
    # convert w, h to width/height relative to image
    w, h = transform_size(wh, anchor_scales)
    # cid is the argmax channel
    cid = nd.argmax(cls_pred, axis=-1, keepdims=True)
    # convert to corner format boxes
    half_w = w / 2
    half_h = h / 2
    left = nd.clip(x - half_w, 0, 1)
    top = nd.clip(y - half_h, 0, 1)
    right = nd.clip(x + half_w, 0, 1)
    bottom = nd.clip(y + half_h, 0, 1)
    output = nd.concat(*[cid, score, left, top, right, bottom], dim=4)
    return output, cls_pred, score, nd.concat(*[xy, wh], dim=4)
```

## 定义一个函数来生成 yolo2 训练目标

YOLO2 寻找真实目标的方法比较特殊，是在每个格点内各自比较，而不是使用全局的预设。而且我们不需要对生成的训练目标进行反向传播，为了简洁描述比较的方法，我们可以在这里转成

numpy 而且可以用 for 循环（切记转成 numpy 会破坏自动求导的记录，只有当反向传播不需要的时候才能使用这个技巧），实际使用中，如果遇到速度问题，我们可以用 mx.ndarray 矩阵的写法来加速。这里我们使用了一个技巧：sample\_weight（个体权重）矩阵，用于损失函数内部权重的调整，我们也可以通过权重矩阵来控制哪些个体需要被屏蔽，这一点在目标检测中尤其重要，因为往往大多数的背景区域不需要预测检测框。

```
In [5]: def corner2center(boxes, concat=True):
    """Convert left/top/right/bottom style boxes into x/y/w/h format"""
    left, top, right, bottom = boxes.split(axis=-1, num_outputs=4)
    x = (left + right) / 2
    y = (top + bottom) / 2
    width = right - left
    height = bottom - top
    if concat:
        last_dim = len(x.shape) - 1
        return nd.concat(*[x, y, width, height], dim=last_dim)
    return x, y, width, height

def center2corner(boxes, concat=True):
    """Convert x/y/w/h style boxes into left/top/right/bottom format"""
    x, y, w, h = boxes.split(axis=-1, num_outputs=4)
    w2 = w / 2
    h2 = h / 2
    left = x - w2
    top = y - h2
    right = x + w2
    bottom = y + h2
    if concat:
        last_dim = len(left.shape) - 1
        return nd.concat(*[left, top, right, bottom], dim=last_dim)
    return left, top, right, bottom

def yolo2_target(scores, boxes, labels, anchors, ignore_label=-1, thresh=0.5):
    """Generate training targets given predictions and labels."""
    b, h, w, n, _ = scores.shape
    anchors = np.reshape(np.array(anchors), (-1, 2))
    #scores = nd.slice_axis(outputs, begin=1, end=2, axis=-1)
    #boxes = nd.slice_axis(outputs, begin=2, end=6, axis=-1)
    gt_boxes = nd.slice_axis(labels, begin=1, end=5, axis=-1)
    target_score = nd.zeros((b, h, w, n, 1), ctx=scores.context)
    target_id = nd.ones_like(target_score, ctx=scores.context) * ignore_label
    target_box = nd.zeros((b, h, w, n, 4), ctx=scores.context)
    sample_weight = nd.zeros((b, h, w, n, 1), ctx=scores.context)
```

```

for b in range(output.shape[0]):
    # find the best match for each ground-truth
    label = labels[b].asnumpy()
    valid_label = label[np.where(label[:, 0] > -0.5)[0], :]
    # shuffle because multi gt could possibly match to one anchor, we keep
    → the last match randomly
    np.random.shuffle(valid_label)
    for l in valid_label:
        gx, gy, gw, gh = (l[1] + l[3]) / 2, (l[2] + l[4]) / 2, l[3] -
    → l[1], l[4] - l[2]
            ind_x = int(gx * w)
            ind_y = int(gy * h)
            tx = gx * w - ind_x
            ty = gy * h - ind_y
            gw = gw * w
            gh = gh * h
            # find the best match using width and height only, assuming centers
    → are identical
            intersect = np.minimum(anchors[:, 0], gw) * np.minimum(anchors[:, 1], gh)
            ovps = intersect / (gw * gh + anchors[:, 0] * anchors[:, 1] -
    → intersect)
            best_match = int(np.argmax(ovps))
            target_id[b, ind_y, ind_x, best_match, :] = l[0]
            target_score[b, ind_y, ind_x, best_match, :] = 1.0
            tw = np.log(gw / anchors[best_match, 0])
            th = np.log(gh / anchors[best_match, 1])
            target_box[b, ind_y, ind_x, best_match, :] = mx.nd.array([tx, ty,
    → tw, th])
            sample_weight[b, ind_y, ind_x, best_match, :] = 1.0
            # print('ind_y', ind_y, 'ind_x', ind_x, 'best_match', best_match,
    → 't', tx, ty, tw, th, 'ovp', ovps[best_match], 'gt', gx, gy, gw/w, gh/h, 'anchor',
    → anchors[best_match, 0], anchors[best_match, 1])
    return target_id, target_score, target_box, sample_weight

```

我们用 **YOLO2Output** 作为 **yolo2** 的输出层，其实本质就是一个 **HybridBlock**，内部包了一个卷积层作为最终的输出

```
In [6]: class YOLO2Output(HybridBlock):
    def __init__(self, num_class, anchor_scales, **kwargs):
        super(YOLO2Output, self).__init__(**kwargs)
```

```

    assert num_class > 0, "number of classes should > 0, given
↪  {}".format(num_class)
        self._num_class = num_class
        assert isinstance(anchor_scales, (list, tuple)), "list or tuple of
↪  anchor scales required"
        assert len(anchor_scales) > 0, "at least one anchor scale required"
        for anchor in anchor_scales:
            assert len(anchor) == 2, "expected each anchor scale to be (width,
↪  height), provided {}".format(anchor)
            self._anchor_scales = anchor_scales
        out_channels = len(anchor_scales) * (num_class + 1 + 4)
        with self.name_scope():
            self.output = nn.Conv2D(out_channels, 1, 1)

    def hybrid_forward(self, F, x, *args):
        return self.output(x)

```

## 接下来是下载并加载数据集

```

In [7]: from mxnet import gluon

root_url = ('https://apache-mxnet.s3-accelerate.amazonaws.com/'
            'gluon/dataset/pikachu/')
data_dir = '../data/pikachu/'
dataset = {'train.rec': 'e6bcb6ffba1ac04ff8a9b1115e650af56ee969c8',
           'train.idx': 'dcf7318b2602c06428b9988470c731621716c393',
           'val.rec': 'd6c33f799b4d058e82f2cb5bd9a976f69d72d520'}
for k, v in dataset.items():
    gluon.utils.download(root_url+k, data_dir+k, sha1_hash=v)

```

```

In [8]: from mxnet import image
from mxnet import nd

data_shape = 256
batch_size = 32
rgb_mean = nd.array([123, 117, 104])
rgb_std = nd.array([58.395, 57.12, 57.375])

def get_iterators(data_shape, batch_size):
    class_names = ['pikachu', 'dummy']
    num_class = len(class_names)
    train_iter = image.ImageDetIter(
        batch_size=batch_size,

```

```

        data_shape=(3, data_shape, data_shape),
        path_imgrec=data_dir+'train.rec',
        path_imgidx=data_dir+'train.idx',
        shuffle=True,
        mean=True,
        std=True,
        rand_crop=1,
        min_object_covered=0.95,
        max_attempts=200)
    val_iter = image.ImageDetIter(
        batch_size=batch_size,
        data_shape=(3, data_shape, data_shape),
        path_imgrec=data_dir+'val.rec',
        shuffle=False,
        mean=True,
        std=True)
    return train_iter, val_iter, class_names, num_class

train_data, test_data, class_names, num_class = get_iterators(
    data_shape, batch_size)

In [9]: batch = train_data.next()
print(batch)

DataBatch: data shapes: [(32, 3, 256, 256)] label shapes: [(32, 1, 5)]

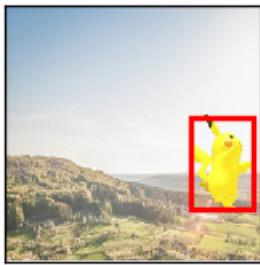
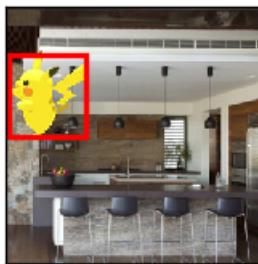
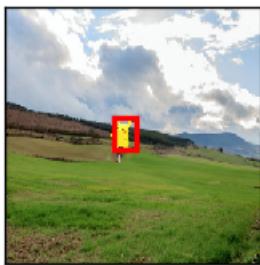
In [10]: %matplotlib inline
import matplotlib as mpl
mpl.rcParams['figure.dpi']= 120
from matplotlib import pyplot as plt

def box_to_rect(box, color, linewidth=3):
    """convert an anchor box to a matplotlib rectangle"""
    box = box.astype(np.int32)
    return plt.Rectangle(
        (box[0], box[1]), box[2]-box[0], box[3]-box[1],
        fill=False, edgecolor=color, linewidth=linewidth)

_, figs = plt.subplots(3, 3, figsize=(6,6))
for i in range(3):
    for j in range(3):
        img, labels = batch.data[0][3*i+j], batch.label[0][3*i+j]
        img = img.transpose((1, 2, 0)) * rgb_std + rgb_mean
        img = img.clip(0,255).astype(np.uint8)/255
        fig = figs[i][j]

```

```
fig.imshow(img)
for label in labels:
    rect = box_to_rect(label[1:5]*data_shape, 'red', 2)
    fig.add_patch(rect)
fig.axes.get_xaxis().set_visible(False)
fig.axes.get_yaxis().set_visible(False)
plt.show()
```



## 损失函数

```
In [11]: sce_loss = gluon.loss.SoftmaxCrossEntropyLoss(from_logits=False)
l1_loss = gluon.loss.L1Loss()
```

## 评估测量

这里我们取巧用一个自己定义的 metric 来记录纯损失值，有的时候当你想不出特别贴切的观测函数，不如直接看损失值有没有下降

```
In [12]: from mxnet import metric

class LossRecorder(mx.metric.EvalMetric):
    """LossRecorder is used to record raw loss so we can observe loss directly
    """
    def __init__(self, name):
        super(LossRecorder, self).__init__(name)

    def update(self, labels, preds=0):
        """Update metric with pure loss
        """
        for loss in labels:
            if isinstance(loss, mx.nd.NDArray):
                loss = loss.asnumpy()
            self.sum_metric += loss.sum()
            self.num_inst += 1

    obj_loss = LossRecorder('objectness_loss')
    cls_loss = LossRecorder('classification_loss')
    box_loss = LossRecorder('box_refine_loss')
```

## 粗粒度调控下每种损失相对的权重

```
In [13]: positive_weight = 5.0
negative_weight = 0.1
class_weight = 1.0
box_weight = 5.0
```

## 网络

加载模型园里训练好的 resnet 网络，取出中间的特征提取层，用合适的锚点尺寸新建我们的 YOLO2 输出层

```
In [14]: pretrained = vision.get_model('resnet18_v1', pretrained=True).features
net = nn.HybridSequential()
for i in range(len(pretrained) - 2):
```

```

        net.add(pretrained[i])

# anchor scales, try adjust it yourself
scales = [[3.3004, 3.59034],
           [9.84923, 8.23783]]


# use 2 classes, 1 as dummy class, otherwise softmax won't work
predictor = YOLO2Output(2, scales)
predictor.initialize()
net.add(predictor)

```

这里我们还是需要 GPU 来加速训练

```

In [15]: from mxnet import init
          from mxnet import gpu

          ctx = gpu(0)
          net.collect_params().reset_ctx(ctx)
          trainer = gluon.Trainer(net.collect_params(),
                                  'sgd', {'learning_rate': 1, 'wd': 5e-4})

In [16]: import time
          from mxnet import autograd
          for epoch in range(20):
              # reset data iterators and metrics
              train_data.reset()
              cls_loss.reset()
              obj_loss.reset()
              box_loss.reset()
              tic = time.time()
              for i, batch in enumerate(train_data):
                  x = batch.data[0].as_in_context(ctx)
                  y = batch.label[0].as_in_context(ctx)
                  with autograd.record():
                      x = net(x)
                      output, cls_pred, score, xywh = yolo2_forward(x, 2, scales)
                      with autograd.pause():
                          tid, tscore, tbox, sample_weight = yolo2_target(score, xywh,
→ y, scales, thresh=0.5)
                          # losses
                          loss1 = sce_loss(cls_pred, tid, sample_weight * class_weight)
                          score_weight = nd.where(sample_weight > 0,

```

```

                                nd.ones_like(sample_weight) *
↪  positive_weight,
                                nd.ones_like(sample_weight) *
↪  negative_weight)
            loss2 = l1_loss(score, tscore, score_weight)
            loss3 = l1_loss(xywh, bbox, sample_weight * box_weight)
            loss = loss1 + loss2 + loss3
            loss.backward()
            trainer.step(batch_size)
            # update metrics
            cls_loss.update(loss1)
            obj_loss.update(loss2)
            box_loss.update(loss3)

        print('Epoch %2d, train %s %.5f, %s %.5f, %s %.5f time %.1f sec' %
            epoch, *cls_loss.get(), *obj_loss.get(), *box_loss.get(),
↪  time.time()-tic))

Epoch 0, train classification_loss 0.00097, objectness_loss 0.03682, box_refine_loss
↪  0.00246 time 9.3 sec
Epoch 1, train classification_loss 0.00034, objectness_loss 0.01589, box_refine_loss
↪  0.00204 time 8.9 sec
Epoch 2, train classification_loss 0.00011, objectness_loss 0.00718, box_refine_loss
↪  0.00189 time 8.8 sec
Epoch 3, train classification_loss 0.00008, objectness_loss 0.00470, box_refine_loss
↪  0.00200 time 8.7 sec
Epoch 4, train classification_loss 0.00005, objectness_loss 0.00339, box_refine_loss
↪  0.00196 time 8.8 sec
Epoch 5, train classification_loss 0.00006, objectness_loss 0.00287, box_refine_loss
↪  0.00193 time 8.6 sec
Epoch 6, train classification_loss 0.00005, objectness_loss 0.00247, box_refine_loss
↪  0.00186 time 8.3 sec
Epoch 7, train classification_loss 0.00004, objectness_loss 0.00216, box_refine_loss
↪  0.00183 time 8.3 sec
Epoch 8, train classification_loss 0.00004, objectness_loss 0.00202, box_refine_loss
↪  0.00180 time 8.3 sec
Epoch 9, train classification_loss 0.00005, objectness_loss 0.00198, box_refine_loss
↪  0.00162 time 8.6 sec
Epoch 10, train classification_loss 0.00004, objectness_loss 0.00180, box_refine_loss
↪  0.00164 time 8.3 sec
Epoch 11, train classification_loss 0.00003, objectness_loss 0.00165, box_refine_loss
↪  0.00159 time 8.3 sec
Epoch 12, train classification_loss 0.00004, objectness_loss 0.00169, box_refine_loss
↪  0.00152 time 8.4 sec

```

```
Epoch 13, train classification_loss 0.00004, objectness_loss 0.00157, box_refine_loss
    ↵ 0.00150 time 8.7 sec
Epoch 14, train classification_loss 0.00004, objectness_loss 0.00154, box_refine_loss
    ↵ 0.00144 time 8.8 sec
Epoch 15, train classification_loss 0.00004, objectness_loss 0.00154, box_refine_loss
    ↵ 0.00137 time 8.8 sec
Epoch 16, train classification_loss 0.00003, objectness_loss 0.00145, box_refine_loss
    ↵ 0.00133 time 8.7 sec
Epoch 17, train classification_loss 0.00003, objectness_loss 0.00144, box_refine_loss
    ↵ 0.00129 time 8.7 sec
Epoch 18, train classification_loss 0.00004, objectness_loss 0.00143, box_refine_loss
    ↵ 0.00129 time 8.4 sec
Epoch 19, train classification_loss 0.00003, objectness_loss 0.00132, box_refine_loss
    ↵ 0.00129 time 8.2 sec
```

## 预处理和预测函数

```
In [17]: def process_image(fname):
    with open(fname, 'rb') as f:
        im = image.imdecode(f.read())
    # resize to data_shape
    data = image.imresize(im, data_shape, data_shape)
    # minus rgb mean, divide std
    data = (data.astype('float32') - rgb_mean) / rgb_std
    # convert to batch x channel x height xwidth
    return data.transpose((2,0,1)).expand_dims(axis=0), im

def predict(x):
    x = net(x)
    output, cls_prob, score, xywh = yolo2_forward(x, 2, scales)
    return nd.contrib.box_nms(output.reshape((0, -1, 6)))
```

## 继续读取皮卡丘来做测试

```
In [18]: x, im = process_image('../img/pikachu.jpg')
out = predict(x.as_in_context(ctx))
out.shape
out

Out[18]:
[[[ 0.          0.99391276  0.52771056  0.5253948   0.66244662  0.6800375 ]
 [ 0.          0.98003525  0.22776087  0.60798109  0.3495315   0.71839571]]
```

```
[ 0.          0.95139569  0.7615453   0.52984631  0.89384723  0.68914676]
...,
[-1.         -1.         -1.         -1.         -1.         -1.        ]
[-1.         -1.         -1.         -1.         -1.         -1.        ]
[-1.         -1.         -1.         -1.         -1.         -1.        ]

```

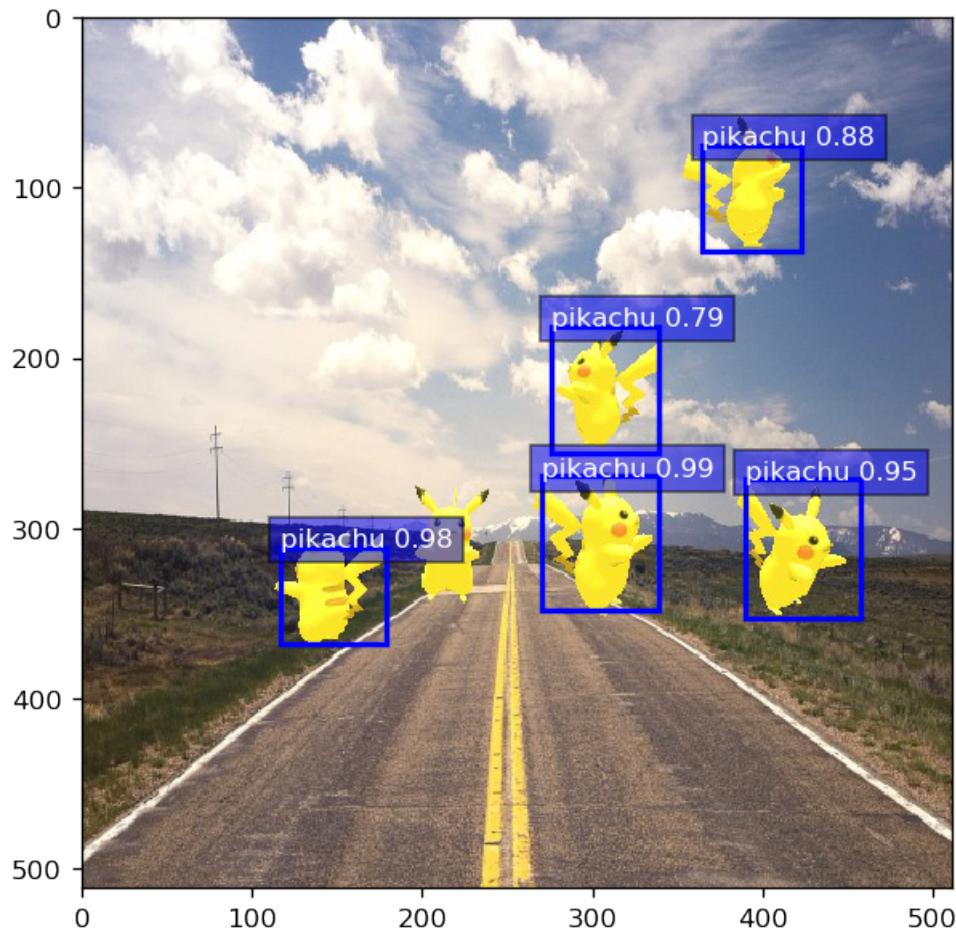
## 显示结果

```
In [19]: mpl.rcParams['figure.figsize'] = (6,6)

colors = ['blue', 'green', 'red', 'black', 'magenta']

def display(im, out, threshold=0.5):
    plt.imshow(im.asnumpy())
    for row in out:
        row = row.asnumpy()
        class_id, score = int(row[0]), row[1]
        if class_id < 0 or score < threshold:
            continue
        color = colors[class_id%len(colors)]
        box = row[2:6] * np.array([im.shape[0],im.shape[1]]*2)
        rect = box_to_rect(nd.array(box), color, 2)
        plt.gca().add_patch(rect)
        text = class_names[class_id]
        plt.gca().text(box[0], box[1],
                       '{:s} {:.2f}'.format(text, score),
                       bbox=dict(facecolor=color, alpha=0.5),
                       fontsize=10, color='white')
    plt.show()

display(im, out[0], threshold=0.5)
```



### 9.6.2 小结

- TODO(@mli)

### 9.6.3 练习

- 试试改变默认的 anchor scale，调整尺寸，增加或减少数量？
- 调整不同损失函数相对的权重，看看对于训练结果有什么影响？

- 在目标检测这种 batch 内部有效目标占比很少的情况下，损失函数内部取平均有什么问题，更好的方法？

#### 9.6.4 扫码直达讨论区



### 9.7 语义分割

我们已经学习了如何识别图片里面的主要物体，和找出里面物体的边框。语义分割则在之上更进一步，它对每个像素预测它是否只是背景，还是属于哪个我们感兴趣的物体。



图 9.8: Semantic Segmentation

可以看到，跟物体检测相比，语义分割预测的边框更加精细。

也许大家还听到过计算机视觉里的一个常用任务：图片分割。它跟语义分割类似，将每个像素划分到某个类。但它跟语义分割不同的时候，图片分割不需要预测每个类具体对应哪个物体。因此图片分割经常只需要利用像素之间的相似度即可，而语义分割则需要详细的类别标号。这也是为什么称其为语义的原因。

本章我们将介绍利用卷积神经网络解决语义分割的一个开创性工作之一：[全链接卷积网络](#)。在此之前我们先了解用来做语义分割的数据。

### 9.7.1 数据集

VOC2012是一个常用的语义分割数据集。输入图片跟之前的数据集类似，但标注也是保存称相应

大小的图片来方便查看。下面代码下载这个数据集并解压。注意到压缩包大小是 2GB，可以预先下好放置在 `data_root` 下。

```
In [1]: import os
import tarfile
from mxnet import gluon

data_root = '../data'
voc_root = data_root + '/VOCdevkit/VOC2012'
url = ('http://host.robots.ox.ac.uk/pascal/VOC/voc2012'
       '/VOCTrainval_11-May-2012.tar')
sha1 = '4e443f8a2eca6b1dac8a6c57641b67dd40621a49'

fname = gluon.utils.download(url, data_root, sha1_hash=sha1)

if not os.path.isfile(voc_root + '/ImageSets/Segmentation/train.txt'):
    with tarfile.open(fname, 'r') as f:
        f.extractall(data_root)
```

下面定义函数将训练图片和标注按序读进内存。

```
In [2]: from mxnet import image

def read_images(root=voc_root, train=True):
    txt_fname = root + '/ImageSets/Segmentation/' + (
        'train.txt' if train else 'val.txt')
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    n = len(images)
    data, label = [None] * n, [None] * n
    for i, fname in enumerate(images):
        data[i] = image.imread('%s/JPEGImages/%s.jpg' % (
            root, fname))
        label[i] = image.imread('%s/SegmentationClass/%s.png' % (
            root, fname))
    return data, label
```

我们画出前面三张图片和它们对应的标号。在标号中，白色代表边框黑色代表背景，其他不同的颜色对应不同物体。

```
In [3]: import sys
sys.path.append('..')
import gluonbook as gb

train_images, train_labels = read_images()
```

```
imgs = []
for i in range(3):
    imgs += [train_images[i], train_labels[i]]

# TODO(mli) temporarily disable show_images to avoid
# TeX capacity exceeded error.
#gb.show_images(imgs, 3, 2)
[im.shape for im in imgs]

Out[3]: [(281, 500, 3),
          (281, 500, 3),
          (375, 500, 3),
          (375, 500, 3),
          (375, 500, 3),
          (375, 500, 3)]
```

同时注意到图片的宽度基本是 500，但高度各不一样。为了能将多张图片合并成一个批量来加速计算，我们需要输入图片都是同样的大小。之前我们通过 `imresize` 来将他们调整成同样的大小。但在语义分割里，我们需要对标注做同样的变化来达到像素级别的匹配。但调整大小将改变像素颜色，使得再将它们映射到物体类别变得困难。

这里我们仅仅使用剪切来解决这个问题。就是说对于输入图片，我们随机剪切出一个固定大小的区域，然后对标号图片做同样位置的剪切。

```
In [4]: def rand_crop(data, label, height, width):
    data, rect = image.random_crop(data, (width, height))
    label = image.fixed_crop(label, *rect)
    return data, label

imgs = []
for _ in range(3):
    imgs += rand_crop(train_images[0], train_labels[0],
                      200, 300)

# TODO(mli) temporarily disable show_images to avoid
# TeX capacity exceeded error.
#gb.show_images(imgs, 3, 2)
```

接下来我们列出每个物体和背景对应的 RGB 值

```
In [5]: classes = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
                  'bottle', 'bus', 'car', 'cat', 'chair', 'cow', 'diningtable',
                  'dog', 'horse', 'motorbike', 'person', 'potted plant',
                  'sheep', 'sofa', 'train', 'tv/monitor']
```

```

# RGB color for each class
colormap = [[0,0,0],[128,0,0],[0,128,0], [128,128,0], [0,0,128],
            [128,0,128],[0,128,128],[128,128,128],[64,0,0],[192,0,0],
            [64,128,0],[192,128,0],[64,0,128],[192,0,128],
            [64,128,128],[192,128,128],[0,64,0],[128,64,0],
            [0,192,0],[128,192,0],[0,64,128]]
```

```
len(classes), len(colormap)
```

Out[5]: (21, 21)

这样给定一个标号图片，我们就可以将每个像素对应的物体标号找出来。

```

In [6]: import numpy as np
        from mxnet import nd

cm2lbl = np.zeros(256**3)
for i,cm in enumerate(colormap):
    cm2lbl[(cm[0]*256+cm[1])*256+cm[2]] = i

def image2label(im):
    data = im.astype('int32').asnumpy()
    idx = (data[:, :, 0]*256+data[:, :, 1])*256+data[:, :, 2]
    return nd.array(cm2lbl[idx])
```

可以看到第一张训练图片的标号里面属于飞机的像素被标记成了1.

```

In [7]: y = image2label(train_labels[0])
y[105:115, 130:140]
```

Out[7]:

```

[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  1.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  1.  1.]]
```

```
<NDArray 10x10 @cpu(0)>
```

现在我们可以定义数据读取了。每一次我们将图片和标注随机剪切到要求的形状，并将标注里每个像素转成对应的标号。简单起见我们将小于要求大小的图片全部过滤掉了。

```
In [8]: from mxnet import gluon
      from mxnet import nd

rgb_mean = nd.array([0.485, 0.456, 0.406])
rgb_std = nd.array([0.229, 0.224, 0.225])

def normalize_image(data):
    return (data.astype('float32') / 255 - rgb_mean) / rgb_std

class VOCSegDataset(gluon.data.Dataset):

    def _filter(self, images):
        return [im for im in images if (
            im.shape[0] >= self.crop_size[0] and
            im.shape[1] >= self.crop_size[1])]

    def __init__(self, train, crop_size):
        self.crop_size = crop_size
        data, label = read_images(train=train)
        data = self._filter(data)
        self.data = [normalize_image(im) for im in data]
        self.label = self._filter(label)
        print('Read '+str(len(self.data))+' examples')

    def __getitem__(self, idx):
        data, label = rand_crop(
            self.data[idx], self.label[idx],
            *self.crop_size)
        data = data.transpose((2, 0, 1))
        label = image2label(label)
        return data, label

    def __len__(self):
        return len(self.data)
```

我们采用  $320 \times 480$  的大小用来训练，注意到这个比前面我们使用的  $224 \times 224$  要大上很多。但是同样我们将长宽都定义成了 32 的整数倍。

```
In [9]: # height x width
input_shape = (320, 480)
voc_train = VOCSegDataset(True, input_shape)
voc_test = VOCSegDataset(False, input_shape)
```

```
Read 1114 examples
Read 1078 examples
```

最后定义批量读取。可以看到跟之前的不同是批量标号不再是一个向量，而是一个三维数组。

```
In [10]: batch_size = 64
train_data = gluon.data.DataLoader(
    voc_train, batch_size, shuffle=True, last_batch='discard')
test_data = gluon.data.DataLoader(
    voc_test, batch_size, last_batch='discard')

for data, label in train_data:
    print(data.shape)
    print(label.shape)
    break

(64, 3, 320, 480)
(64, 320, 480)
```

## 9.7.2 全连接卷积网络

在数据的处理过程我们看到语义分割跟前面介绍的应用的主要区别在于，预测的标号不再是一个或者几个数字，而是每个像素都需要有标号。在卷积神经网络里，我们通过卷积层和池化层逐渐减少数据长宽但同时增加通道数。例如 ResNet18 里，我们先将输入长宽减少 32 倍，由  $3 \times 224 \times 224$  的图片转成  $512 \times 7 \times 7$  的输出，应该全局池化层变成 512 长向量，然后最后通过全链接层转成一个长度为  $n$  的输出向量，这里  $n$  是类数，既 `num_classes`。但在这里，对于输出为  $3 \times 320 \times 480$  的图片，我们需要输出是  $n \times 320 \times 480$ ，就是每个输入像素都需要预测一个长度为  $n$  的向量。

全连接卷积网络（FCN）的提出是基于这样一个观察。假设  $f$  是一个卷积层，而且  $y = f(x)$ 。那么在反传求导时， $\partial f(y)$  会返回一个跟  $x$  一样形状的输出。卷积是一个对偶函数，就是  $\partial^2 f = f$ 。那么如果我们想得到跟输入一样的输入，那么定义  $g = \partial f$ ，这样  $g(f(x))$  就能达到我们想要的。

具体来说，我们定义一个卷积转置层（transposed convolutional，也经常被错误的叫做 deconvolutions），它就是将卷积层的 `forward` 和 `backward` 函数兑换。

下面例子里我们看到使用同样的参数，除了替换输入和输出通道数外，`Conv2DTranspose` 可以将 `nn.Conv2D` 的输出还原其输入大小。

```
In [11]: from mxnet.gluon import nn

conv = nn.Conv2D(10, kernel_size=4, padding=1, strides=2)
conv_trans = nn.Conv2DTranspose(3, kernel_size=4, padding=1, strides=2)
```

```
conv.initialize()
conv_trans.initialize()

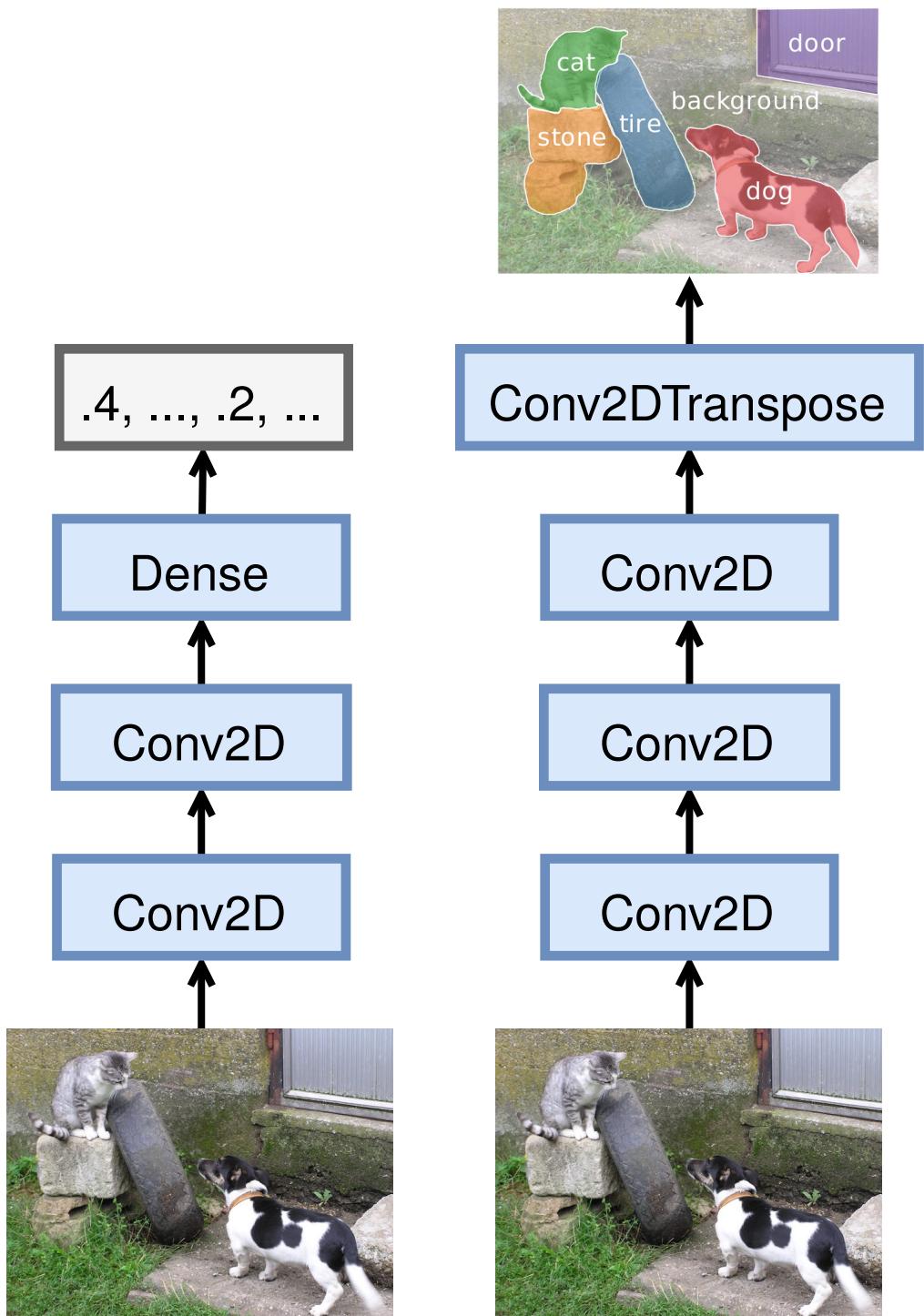
x = nd.random.uniform(shape=(1,3,64,64))
y = conv(x)
print('Input:', x.shape)
print('After conv:', y.shape)
print('After transposed conv', conv_trans(y).shape)

Input: (1, 3, 64, 64)
After conv: (1, 10, 32, 32)
After transposed conv (1, 3, 64, 64)
```

另外一点要注意的是，在最后的卷积层我们同样使用平化层（`nn.Flatten`）或者（全局）池化层来使得方便使用之后的全连接层作为输出。但是这样会损害空间信息，而这个对语义分割很重要。一个解决办法是去掉不需要的池化层，并将全连接层替换成 $1 \times 1$ 卷基层。

所以给定一个卷积网络，FCN 主要做下面的改动

- 替换全连接层成 $1 \times 1$ 卷基
- 去掉过于损失空间信息的池化层，例如全局池化
- 最后接上卷积转置层来得到需要大小的输出
- 为了训练更快，通常权重会初始化为预先训练好的权重



下面我们基于 Resnet18 来创建 FCN。首先我们下载一个预先训练好的模型。

```
In [12]: from mxnet.gluon.model_zoo import vision as models
pretrained_net = models.resnet18_v2(pretrained=True)

(pretrained_net.features[-4:], pretrained_net.output)

Out[12]: ([BatchNorm(axis=1, eps=1e-05, momentum=0.9, fix_gamma=False,
← use_global_stats=False, in_channels=512),
Activation(relu),
GlobalAvgPool2D(size=(1, 1), stride=(1, 1), padding=(0, 0), ceil_mode=True),
Flatten],
Dense(512 -> 1000, linear))
```

我们看到 feature 模块最后两层是 GlobalAvgPool2D 和 Flatten, 都是我们不需要的。所以我们定义一个新的网络，它复制除了最后两层的 features 模块的权重。

```
In [13]: net = nn.HybridSequential()
for layer in pretrained_net.features[:-2]:
    net.add(layer)

x = nd.random.uniform(shape=(1,3,*input_shape))
print('Input:', x.shape)
print('Output:', net(x).shape)

Input: (1, 3, 320, 480)
Output: (1, 512, 10, 15)
```

然后接上一个通道数等于类数的  $1 \times 1$  卷积层。注意到 net 已经将输入长宽减少了 32 倍。那么我们需要接入一个 strides=32 的卷积转置层。我们使用一个比 strides 大两倍的 kernel, 然后补上适当的填充。

```
In [14]: num_classes = len(classes)

with net.name_scope():
    net.add(
        nn.Conv2D(num_classes, kernel_size=1),
        nn.Conv2DTranspose(num_classes, kernel_size=64, padding=16,strides=32)
    )
```

### 9.7.3 训练

训练的时候我们需要初始化新添加的两层。我们可以随机初始化，但实际上发现将卷积转置层初始化成双线性差值函数可以使得训练更容易。

```
In [15]: def bilinear_kernel(in_channels, out_channels, kernel_size):
    factor = (kernel_size + 1) // 2
    if kernel_size % 2 == 1:
        center = factor - 1
    else:
        center = factor - 0.5
    og = np.ogrid[:kernel_size, :kernel_size]
    filt = (1 - abs(og[0] - center) / factor) * \
        (1 - abs(og[1] - center) / factor)
    weight = np.zeros(
        (in_channels, out_channels, kernel_size, kernel_size),
        dtype='float32')
    weight[range(in_channels), range(out_channels), :, :] = filt
    return nd.array(weight)
```

下面代码演示这样的初始化等价于对图片进行双线性差值放大。

```
In [16]: from matplotlib import pyplot as plt

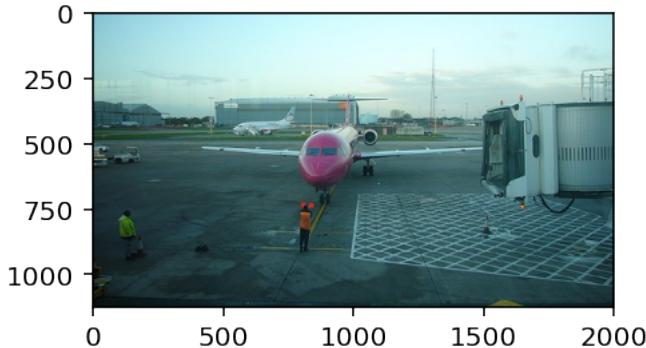
x = train_images[0]
print('Input', x.shape)
x = x.astype('float32').transpose((2, 0, 1)).expand_dims(axis=0)/255

conv_trans = nn.Conv2DTranspose(
    3, in_channels=3, kernel_size=8, padding=2, strides=4)
conv_trans.initialize()
conv_trans(x)
conv_trans.weight.set_data(bilinear_kernel(3, 3, 8))

y = conv_trans(x)
y = y[0].clip(0, 1).transpose((1, 2, 0))
print('Output', y.shape)

plt.imshow(y.asnumpy())
plt.show()

Input (281, 500, 3)
Output (1124, 2000, 3)
```



所以网络的初始化包括了三部分。主体卷积网络从训练好的 ResNet18 复制得来，替代 ResNet18 最后全连接的卷积层使用随机初始化。

最后的卷积转置层则使用双线性差值。对于卷积转置层，我们可以自定义一个初始化类。简单起见，这里我们直接通过权重的 `set_data` 函数改写权重。记得我们介绍过 Gluon 使用延后初始化来减少构造网络时需要制定输入大小。所以我们先随意初始化它，计算一次 `forward`，然后改写权重。

```
In [17]: from mxnet import init

conv_trans = net[-1]
conv_trans.initialize(init=init.Zero())
net[-2].initialize(init=init.Xavier())

x = nd.zeros((batch_size, 3, *input_shape))
net(x)

shape = conv_trans.weight.data().shape
conv_trans.weight.set_data(bilinear_kernel(*shape[0:3]))
```

这时候我们可以真正开始训练了。值得一提的是我们使用卷积转置层的通道来预测像素的类别。所以在做 `softmax` 和预测的时候我们需要使用通道这个维度，既维度 1。所以在 `SoftmaxCrossEntropyLoss` 里加入了额外了 `axis=1` 选项。其他的部分跟之前的训练一致。

```
In [18]: loss = gluon.loss.SoftmaxCrossEntropyLoss(axis=1)

ctx = gb.try_all_gpus()
net.collect_params().reset_ctx(ctx)

trainer = gluon.Trainer(net.collect_params(),
```

```
'sgd', {'learning_rate': .1, 'wd':1e-3})  
  
#gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=10)
```

### 9.7.4 预测

预测函数跟之前的图片分类预测类似，但跟上面一样，主要不同在于我们需要在 `axis=1` 上做 `argmax`。同时我们定义 `image2label` 的反函数，它将预测值转成图片。

```
In [19]: def predict(im):  
    data = normalize_image(im)  
    data = data.transpose((2,0,1)).expand_dims(axis=0)  
    yhat = net(data.as_in_context(ctx[0]))  
    pred = nd.argmax(yhat, axis=1)  
    return pred.reshape((pred.shape[1], pred.shape[2]))  
  
def label2image(pred):  
    x = pred.astype('int32').asnumpy()  
    cm = nd.array(colormap).astype('uint8')  
    return nd.array(cm[x,:])
```

我们读取前几张测试图片并对其进行预测。

```
In [20]: test_images, test_labels = read_images(train=False)  
  
n = 6  
imgs = []  
for i in range(n):  
    x = test_images[i]  
    #pred = label2image(predict(x))  
    #imgs += [x, pred, test_labels[i]]  
  
#gb.show_images(imgs, n, 3)  
# TODO(ml): fix vgg.md: out of memory error, too.
```

### 9.7.5 小结

- 通过使用卷积转置层，我们可以得到更大分辨率的输出。

### 9.7.6 练习

- 试着改改最后的卷积转置层的参数设定
- 看看双线性差值初始化是不是必要的
- 试着改改训练参数来使得收敛更好些
- FCN 论文中提到了不只是使用主体卷积网络输出，还可以将前面层的输出也加进来。试着实现。

### 9.7.7 扫码直达讨论区



## 9.8 样式迁移

喜欢拍照的同学可能都接触过滤镜，它们能改变照片的颜色风格，使得风景照更加锐利，或者人像更加美白。但一个滤镜通常只能改变照片的某个方面，要达到想要的风格，经常需要我们大量组合尝试多个滤镜。这个过程被通常称之为“PS一下”。对于简单的调整，例如拉一拉颜色曲线变成日系小清新风或者加点德味，通常都不难做到。但对于复杂的要求，例如将图片调成梵高风，则需要大量的专业技巧。例如 17 年上映的《挚爱梵高》由 115 名专业画师画了 65,000 张梵高风格的油画而得。

一个自然的想法是，我们能不能通过神经网络来自动化这个过程。具体来说，我们希望将一张指定的图片的风格，例如梵高的某张油画，应用到另外一张内容图片上。



图 9.10: Neural Style

Gatys 等人开创性的通过匹配卷积神经网络的中间层输出来训练出合成图片。它的流程如下所示：

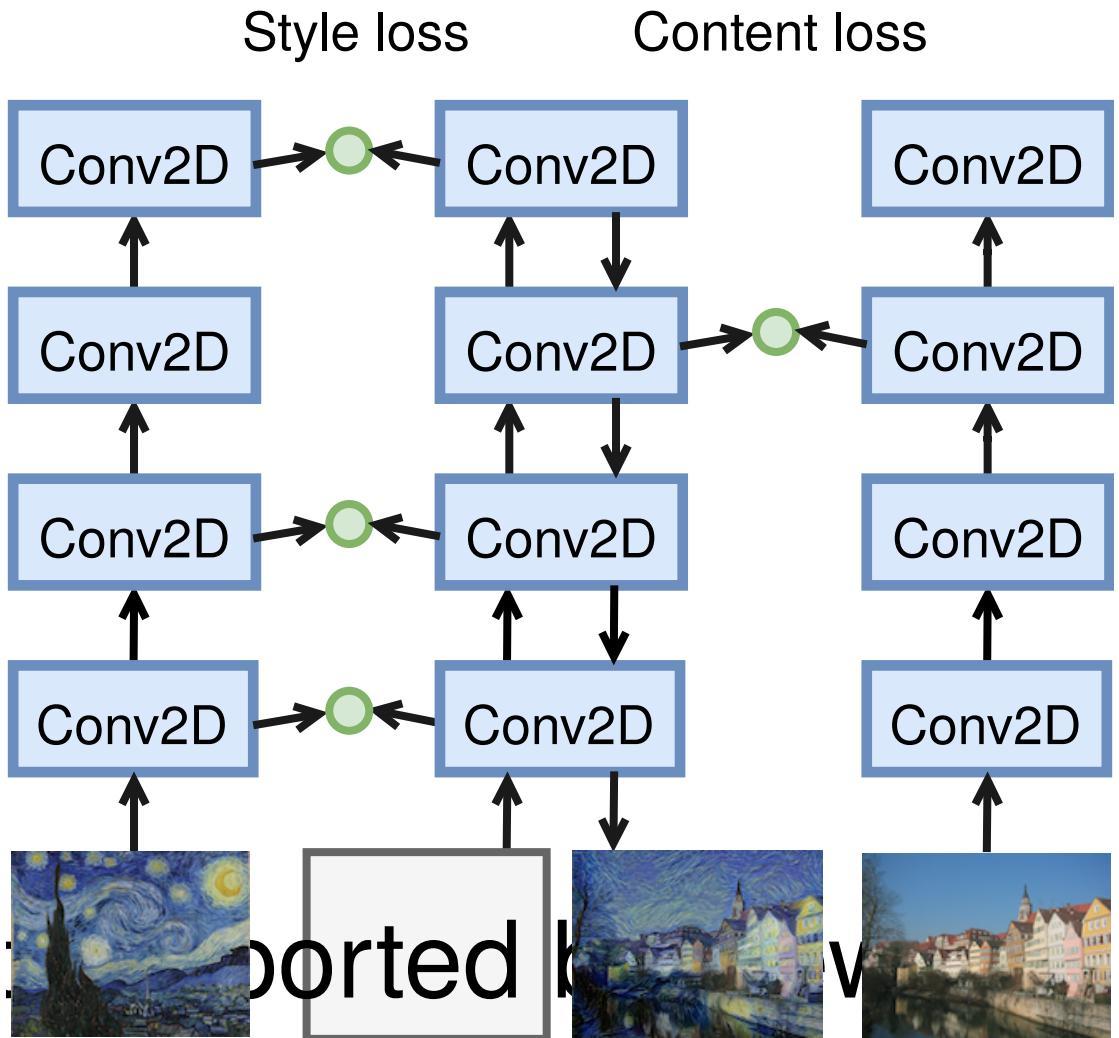


图 9.11: Neural Style Training

1. 我们首先挑选一个卷积神经网络来提取特征。我们选择它的特定层来匹配样式，特定层来匹配内容。示意图中我们选择层 1,2,4 作为样式层，层 3 作为内容层。
2. 输入样式图片并保存样式层输出，记第  $i$  层输出为  $s_i$
3. 输入内容图片并保存内容层输出，记第  $i$  层输出为  $c_i$

4. 初始化合成图片  $x$  为随机值或者其他更好的初始值。然后进行迭代使得用  $x$  抽取的特征能够匹配上  $s_i$  和  $c_i$ 。具体来说，我们如下迭代直到收敛。
5. 输入  $x$  计算样式层和内容层输出，记第  $i$  层输出为  $y_i$
6. 使用样式损失函数来计算  $y_i$  和  $s_i$  的差异
7. 使用内容损失函数来计算  $y_i$  和  $c_i$  的差异
8. 对损失求和并对输入  $x$  求导，记导数为  $g$
9. 更新  $x$ ，例如  $x = x - \eta g$

内容损失函数使用通常回归用的均方误差。对于样式，我们可以将它看成是像素点在每个通道的统计分布。例如要匹配两张图片的颜色，我们的一个做法是匹配这两张图片在 RGB 这三个通道上的直方图。更一般的，假设卷积层的输出格式是  $c \times h \times w$ ，既 `channels x height x width`。那么我们可以把它变形成  $c \times hw$  的 2D 数组，并将它看成是一个维度为  $c$  的随机变量采样到的  $hw$  个点。所谓的样式匹配就是使得两个  $c$  维随机变量统计分布一致。

匹配统计分布常用的做法是冲量匹配，就是说使得他们有一样的均值，协方差，和其他高维的冲量。为了计算简单起见，我们这里假设卷积输出已经是均值为 0 了，而且我们只匹配协方差。也就是说，样式损失函数就是对  $s_i$  和  $y_i$  计算 Gram 矩阵然后应用均方误差

$$\text{styleloss}(s_i, y_i) = \frac{1}{c^2 hw} \|s_i s_i^T - y_i y_i^T\|_F$$

这里假设我们已经将  $s_i$  和  $y_i$  变形成了  $c \times hw$  的 2D 矩阵了。

下面我们将实现这个算法来深入理解各个参数，例如样式层和内容层的选取，对实际结果的影响。

## 9.8.1 数据

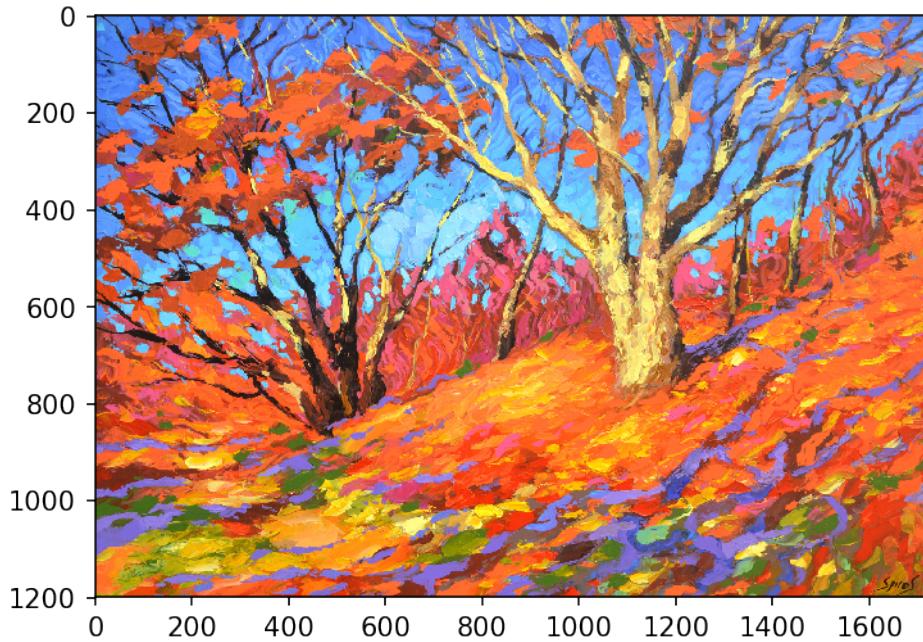
我们将尝试将下面的水粉橡树的样式应用到实拍的松树上。

```
In [1]: %matplotlib inline
import matplotlib as mpl
mpl.rcParams['figure.dpi']= 150
from matplotlib import pyplot as plt

from mxnet import image

style_img = image.imread('../img/autumn_oak.jpg')
content_img = image.imread('../img/pine-tree.jpg')
```

```
plt.imshow(style_img.asnumpy())
plt.show()
plt.imshow(content_img.asnumpy())
plt.show()
```





跟前面教程一样我们定义预处理和后处理函数，它们将原始图片进行归一化并转换成卷积网络接受的输入格式，和还原成能展示的图片格式。

```
In [2]: from mxnet import nd

rgb_mean = nd.array([0.485, 0.456, 0.406])
rgb_std = nd.array([0.229, 0.224, 0.225])

def preprocess(img, image_shape):
    img = image.imresize(img, *image_shape)
    img = (img.astype('float32')/255 - rgb_mean) / rgb_std
    return img.transpose((2,0,1)).expand_dims(axis=0)

def postprocess(img):
    img = img[0].as_in_context(rgb_std.context)
    return (img.transpose((1,2,0))*rgb_std + rgb_mean).clip(0,1)
```

## 9.8.2 模型

我们使用原论文使用的 VGG 19 模型。并下载在 Imagenet 上训练好的权重。

```
In [3]: from mxnet.gluon.model_zoo import vision as models
```

```
pretrained_net = models.vgg19(pretrained=True)
pretrained_net
```

```
Out[3]: VGG(
```

```
    (features): HybridSequential(
        (0): Conv2D(3 -> 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): Activation(relu)
        (2): Conv2D(64 -> 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): Activation(relu)
        (4): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
        (5): Conv2D(64 -> 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (6): Activation(relu)
        (7): Conv2D(128 -> 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (8): Activation(relu)
        (9): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
        (10): Conv2D(128 -> 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (11): Activation(relu)
        (12): Conv2D(256 -> 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (13): Activation(relu)
        (14): Conv2D(256 -> 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (15): Activation(relu)
        (16): Conv2D(256 -> 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (17): Activation(relu)
        (18): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0),
    ← ceil_mode=False)
        (19): Conv2D(256 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (20): Activation(relu)
        (21): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (22): Activation(relu)
        (23): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (24): Activation(relu)
        (25): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (26): Activation(relu)
        (27): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0),
    ← ceil_mode=False)
        (28): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (29): Activation(relu)
        (30): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (31): Activation(relu)
        (32): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (33): Activation(relu)
        (34): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```

        (35): Activation(relu)
        (36): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0),
→  ceil_mode=False)
        (37): Dense(25088 -> 4096, Activation(relu))
        (38): Dropout(p = 0.5, axes=())
        (39): Dense(4096 -> 4096, Activation(relu))
        (40): Dropout(p = 0.5, axes=())
    )
    (output): Dense(4096 -> 1000, linear)
)

```

回忆 VGG 这一章里，我们使用五个卷积块 `vgg_block` 来构建网络。快之间使用 `nn.MaxPool2D` 来做间隔。我们有很多种选择来使用某些层作为样式和内容的匹配层。通常越靠近输入层越容易匹配内容和样式的细节信息，越靠近输出则越倾向于语义的内容和全局的样式。这里我们按照原论文使用每个卷积块的第一个卷基层输出来匹配样式，和第四个块中的最后一个卷积层来匹配内容。根据 `pretrained_net` 的输出我们记录下这些层对应的位置。

```
In [4]: style_layers = [0,5,10,19,28]
content_layers = [25]
```

因为只需要使用中间层的输出，我们构建一个新的网络，它只保留我们需要的层。

```
In [5]: from mxnet.gluon import nn
```

```

def get_net(pretrained_net, content_layers, style_layers):
    net = nn.Sequential()
    for i in range(max(content_layers+style_layers)+1):
        net.add(pretrained_net.features[i])
    return net

net = get_net(pretrained_net, content_layers, style_layers)

```

给定输入 `x`，简单使用 `net(x)` 只能拿到最后的输出，而这里我们还需要 `net` 的中间层输出。因此我们逐层计算，并保留需要的输出。

```

In [6]: def extract_features(x, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        x = net[i](x)
        if i in style_layers:
            styles.append(x)
        if i in content_layers:
            contents.append(x)

```

```
    return contents, styles
```

### 9.8.3 损失函数

内容匹配是一个典型的回归问题，我们将使用均方误差来比较内容层的输出。

```
In [7]: def content_loss(yhat, y):
    return (yhat-y).square().mean()
```

样式匹配则是通过拟合 Gram 矩阵。我们先定义它的计算：

```
In [8]: def gram(x):
    c = x.shape[1]
    n = x.size / x.shape[1]
    y = x.reshape((c, int(n)))
    return nd.dot(y, y.T) / n

gram(nd.ones((1,3,4,4)))
```

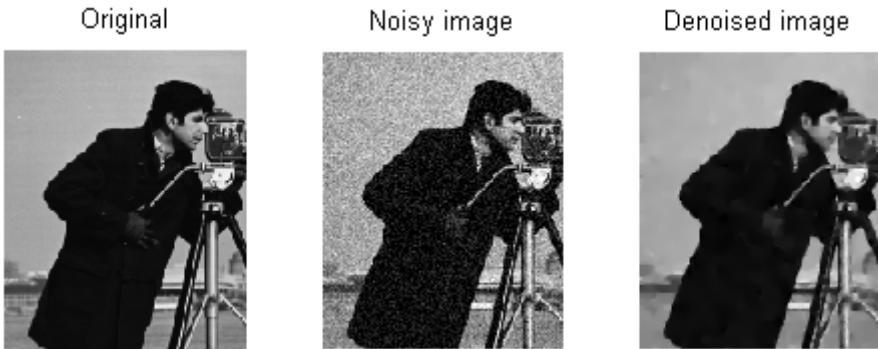
Out[8]:

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
<NDArray 3x3 @cpu(0)>
```

和对应的损失函数。对于要匹配的样式图片它的样式输出在训练中不会改变，我们将提前计算好它的 Gram 矩阵来作为输入使得计算加速。

```
In [9]: def style_loss(yhat, gram_y):
    return (gram(yhat) - gram_y).square().mean()
```

当使用靠近输出层的高层输出来拟合时，经常可以观察到学到的图片里面有大量高频噪音。这个有点类似老式天线电视机经常遇到的白噪音。有多种方法来降噪，例如可以加入模糊滤镜，或者使用总变差降噪（Total Variation Denoising）。



假设  $x_{i,j}$  表示像素  $(i, j)$ , 那么我们加入下面的损失函数, 它使得邻近的像素值相似:

$$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}|$$

```
In [10]: def tv_loss(yhat):
    return 0.5*((yhat[:, :, 1:, :] - yhat[:, :, :-1, :]).abs().mean() +
                 (yhat[:, :, :, 1:] - yhat[:, :, :, :-1]).abs().mean())
```

总损失函数是上述三个损失函数的加权和。通过调整权重值我们可以控制学到的图片是否保留更多样式, 更多内容, 还是更加干净。注意到样式匹配中我们使用了 5 个层的输出, 我们对靠近输入的层给予比较大的权重。

```
In [11]: channels = [net[l].weight.shape[0] for l in style_layers]
style_weights = [1e4/n**2 for n in channels]
content_weights = [1]
tv_weight = 10
```

我们可以使用 `nd.add_n` 来将多个损失函数的输出按权重加起来。

```
In [12]: def sum_loss(loss, preds, truths, weights):
    return nd.add_n(*[w*loss(yhat, y) for w, yhat, y in zip(
        weights, preds, truths)])
```

## 9.8.4 训练

首先我们定义两个函数, 他们分别对源内容图片和源样式图片提取特征。

```
In [13]: def get_contents(image_shape):
    content_x = preprocess(content_img, image_shape).copyto(ctx)
    content_y, _ = extract_features(content_x, content_layers, style_layers)
```

```

        return content_x, content_y

def get_styles(image_shape):
    style_x = preprocess(style_img, image_shape).copyto(ctx)
    _, style_y = extract_features(style_x, content_layers, style_layers)
    style_y = [gram(y) for y in style_y]
    return style_x, style_y

```

训练过程跟之前的主要的主要不同在于

1. 这里我们的损失函数更加复杂。
2. 我们只对输入进行更新，这个意味着我们需要对输入  $x$  预先分配了梯度。
3. 我们可能会替换匹配内容和样式的层，和调整他们之间的权重，来得到不同风格的输出。这里我们对梯度做了一般化，使得不同参数下的学习率不需要太大变化。
4. 仍然使用简单的梯度下降，但每  $n$  次迭代我们会减小一次学习率

```

In [14]: from time import time
          from mxnet import autograd

def train(x, max_epochs, lr, lr_decay_epoch=200):
    tic = time()
    for i in range(max_epochs):
        with autograd.record():
            content_py, style_py = extract_features(
                x, content_layers, style_layers)
            content_L = sum_loss(
                content_loss, content_py, content_y, content_weights)
            style_L = sum_loss(
                style_loss, style_py, style_y, style_weights)
            tv_L = tv_weight * tv_loss(x)
            loss = style_L + content_L + tv_L

            loss.backward()
            x.grad[:] /= x.grad.abs().mean() + 1e-8
            x[:] -= lr * x.grad
            # add sync to avoid large mem usage
            nd.waitall()

        if i and i % 20 == 0:
            print('batch %3d, content %.2f, style %.2f, '
                  'TV %.2f, time %.1f sec' % (
                      i, content_L.asscalar(), style_L.asscalar(),

```

```

        tv_L.asscalar(), time()-tic))
tic = time()

if i and i % lr_decay_epoch == 0:
    lr *= 0.1
    print('change lr to ', lr)

plt.imshow(postprocess(x).asnumpy())
plt.show()
return x

```

现在所有函数都定义好了，我们可以真正开始训练了。从性能上考虑，首先我们在一个大小为  $300 \times 200$  的输入上进行训练。因为我们不会更新源图片和模型参数，所以我们可以提前抽取好他们的特征。我们把要合成图片的初始值设成内容图片来加速收敛。

```

In [15]: import sys
sys.path.append('..')
import gluonbook as gb

image_shape = (300,200)

ctx = gb.try_gpu()
net.collect_params().reset_ctx(ctx)

content_x, content_y = get_contents(image_shape)
style_x, style_y = get_styles(image_shape)

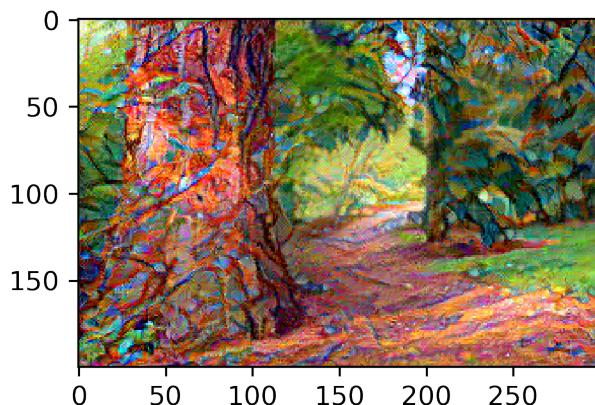
x = content_x.copyto(ctx)
x.attach_grad()

y = train(x, 500, 0.1)

batch 20, content 29.35, style 610.28, TV 4.61, time 3.7 sec
batch 40, content 31.20, style 294.64, TV 4.95, time 1.3 sec
batch 60, content 29.77, style 299.48, TV 5.13, time 1.3 sec
batch 80, content 30.15, style 233.02, TV 5.36, time 1.3 sec
batch 100, content 31.36, style 208.71, TV 5.47, time 1.3 sec
batch 120, content 28.98, style 218.05, TV 5.53, time 1.3 sec
batch 140, content 29.72, style 160.66, TV 5.72, time 1.3 sec
batch 160, content 30.19, style 153.45, TV 5.80, time 1.3 sec
batch 180, content 28.42, style 200.11, TV 5.83, time 1.3 sec
batch 200, content 29.08, style 153.43, TV 5.97, time 1.3 sec
change lr to  0.010000000000000002
batch 220, content 24.36, style 33.50, TV 5.72, time 1.3 sec

```

```
batch 240, content 21.88, style 28.71, TV 5.67, time 1.3 sec
batch 260, content 20.81, style 24.94, TV 5.65, time 1.3 sec
batch 280, content 20.17, style 22.60, TV 5.62, time 1.3 sec
batch 300, content 19.10, style 22.22, TV 5.58, time 1.3 sec
batch 320, content 18.86, style 20.59, TV 5.57, time 1.3 sec
batch 340, content 18.38, style 18.96, TV 5.55, time 1.3 sec
batch 360, content 17.81, style 19.18, TV 5.51, time 1.3 sec
batch 380, content 17.61, style 18.84, TV 5.49, time 1.3 sec
batch 400, content 17.21, style 18.63, TV 5.47, time 1.3 sec
change lr to 0.0010000000000000002
batch 420, content 16.77, style 12.59, TV 5.46, time 1.3 sec
batch 440, content 16.43, style 12.15, TV 5.45, time 1.3 sec
batch 460, content 16.17, style 11.82, TV 5.43, time 1.3 sec
batch 480, content 15.95, style 11.55, TV 5.42, time 1.3 sec
```



观察损失值的变化。因为我们使用了内容图片作为初始化，所以一开始看到样式损失比较大。但随着迭代的进行，它减少的非常迅速，尤其是在每次调整学习率后。噪音在训练中有略微的增加，但在后期还是控制在合理的范围。

最后的结果里可以看到明显的我们将样式图片里面的大的色块应用到了内容图片上。但是由于输入图片大小比较小，所以看到细节上比较模糊。

下面我们在更大的  $1200 \times 800$  的尺寸上训练，希望能得到更加清晰的合成图片。同样为了加速收敛，我们将前面得到的合成图片放大成我们要的尺寸做为初始值。

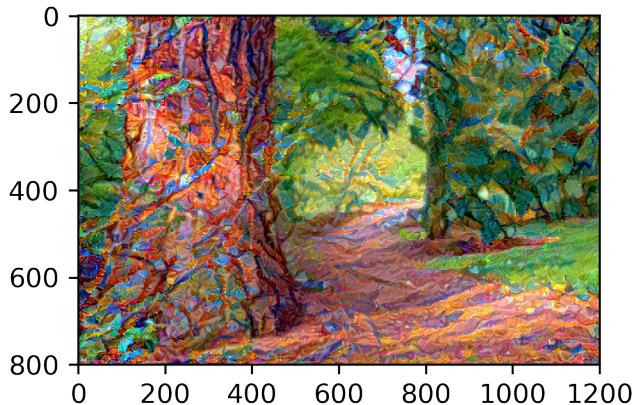
```
In [16]: image_shape = (1200,800)

content_x, content_y = get_contents(image_shape)
style_x, style_y = get_styles(image_shape)
```

```
x = preprocess(postprocess(y)*255, image_shape).copyto(ctx)
x.attach_grad()

z = train(x, 300, 0.1, 100)

batch 20, content 48.15, style 748.71, TV 3.29, time 25.8 sec
batch 40, content 45.85, style 699.52, TV 3.59, time 18.2 sec
batch 60, content 46.01, style 732.41, TV 3.83, time 18.2 sec
batch 80, content 46.57, style 762.81, TV 4.02, time 18.2 sec
batch 100, content 46.78, style 774.39, TV 4.19, time 18.2 sec
change lr to 0.010000000000000002
batch 120, content 29.30, style 61.53, TV 3.68, time 18.2 sec
batch 140, content 25.25, style 30.64, TV 3.48, time 18.2 sec
batch 160, content 23.23, style 44.91, TV 3.39, time 18.2 sec
batch 180, content 23.24, style 24.29, TV 3.34, time 18.3 sec
batch 200, content 21.72, style 18.58, TV 3.28, time 18.2 sec
change lr to 0.001000000000000002
batch 220, content 20.32, style 13.19, TV 3.24, time 18.2 sec
batch 240, content 19.49, style 12.07, TV 3.21, time 18.2 sec
batch 260, content 18.90, style 11.37, TV 3.19, time 18.2 sec
batch 280, content 18.43, style 10.85, TV 3.18, time 18.2 sec
```



可以看到由于初始值更加好，这次的收敛更加迅速，虽然每次迭代花时间更长。由于是图片更大，我们可以更清楚的看到细节。里面不仅有大块的色彩，色彩块里面也有细微的纹理。这是由于我们在匹配样式的时候使用了多层的输出。

最后我们可以把合成的图片保存下来。

```
In [17]: plt.imsave('result.png', postprocess(z).asnumpy())
```

### 9.8.5 小结

- 通过匹配神经网络的中间层输出，我们可以有效的融合不同图片的内容和样式。

### 9.8.6 练习

- 改变内容和样式层
- 使用不同的权重
- 换几张样式和内容图片

### 9.8.7 扫码直达讨论区



## 9.9 实战 Kaggle 比赛：对原始图像文件分类（CIFAR-10）

我们在监督学习中的一章里，以房价预测问题为例，介绍了如何使用 Gluon 来实战Kaggle 比赛。我们在本章中选择了 Kaggle 中著名的CIFAR-10 原始图像分类问题。我们以该问题为例，为大家提供使用 Gluon 对原始图像文件进行分类的示例代码。

### 9.9.1 Kaggle 中的 CIFAR-10 原始图像分类问题

Kaggle是一个著名的供机器学习爱好者交流的平台。为了便于提交结果，请大家注册Kaggle账号。然后请大家先点击[CIFAR-10 原始图像分类问题](#)了解有关本次比赛的信息。



**CIFAR-10 - Object Recognition in Images**

Identify the subject of 60,000 labeled images  
231 teams · 3 years ago

Overview Data Discussion Leaderboard Rules Team My Submissions **Late Submission**

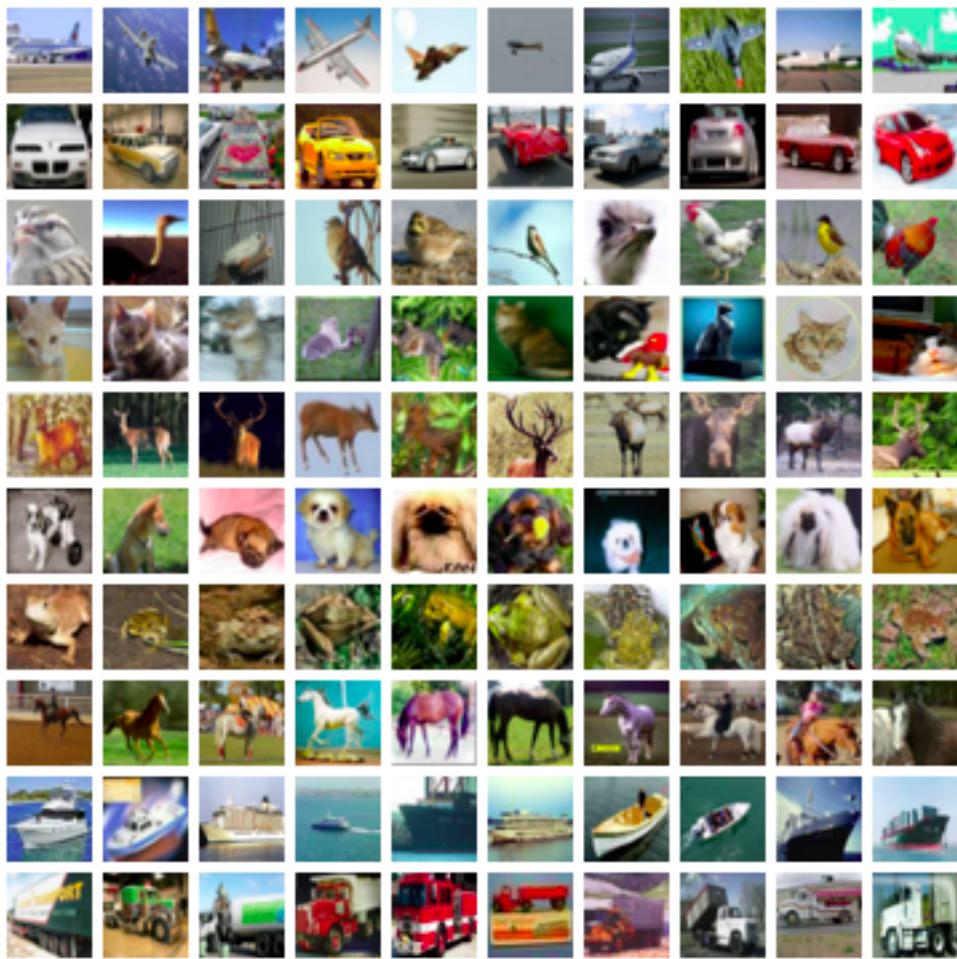
Overview

Description	CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the <a href="#">80 million tiny images dataset</a> and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.
Evaluation	Kaggle is hosting a CIFAR-10 leaderboard for the machine learning community to use for fun and practice. You can see how your approach compares to the latest research methods on <a href="#">Rodrigo Benenson's classification results page</a> .

## 9.9.2 整理原始数据集

比赛数据分为训练数据集和测试数据集。训练集包含 5 万张图片。测试集包含 30 万张图片：其中有 1 万张图片用来计分，但为了防止人工标注测试集，里面另加了 29 万张不计分的图片。

两个数据集都是 png 彩色图片，大小为  $32 \times 32 \times 3$ 。训练集一共有 10 类图片，分别为飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船和卡车。



## 下载数据集

登录 Kaggle 后，数据可以从[CIFAR-10 原始图像分类问题](#)中下载。

- 训练数据集 `train.7z` 下载地址
- 测试数据集 `test.7z` 下载地址
- 训练数据标签 `trainLabels.csv` 下载地址

## 解压数据集

训练数据集 train.7z 和测试数据集 test.7z 都是压缩格式，下载后请解压缩。解压缩后原始数据集的路径可以如下：

- ..../data/kaggle\_cifar10/train/[1-50000].png
- ..../data/kaggle\_cifar10/test/[1-300000].png
- ..../data/kaggle\_cifar10/trainLabels.csv

为了使网页编译快一点，我们在 git repo 里仅仅存放 100 个训练样本（'train\_tiny.zip'）和 1 个测试样本（'test\_tiny.zip'）。执行以下代码会从 git repo 里解压生成小样本训练和测试数据，文件夹名称分别为' train\_tiny' 和' test\_tiny'。训练数据标签的压缩文件将被解压成 trainLabels.csv。

```
In [1]: # 如果训练下载的 Kaggle 的完整数据集，把下面改 False
demo = True
if demo:
    import zipfile
    for fin in ['train_tiny.zip', 'test_tiny.zip', 'trainLabels.csv.zip']:
        with zipfile.ZipFile('../data/kaggle_cifar10/' + fin, 'r') as zin:
            zin.extractall('../data/kaggle_cifar10/')
```

## 整理数据集

我们定义下面的 reorg\_cifar10\_data 函数来整理数据集。整理后，同一类图片将出现在在同一个文件夹下，便于 Gluon 稍后读取。

函数中的参数如 data\_dir、train\_dir 和 test\_dir 对应上述数据存放路径及训练和测试的图片集文件夹名称。参数 label\_file 为训练数据标签的文件名称。参数 input\_dir 是整理后数据集文件夹名称。参数 valid\_ratio 是验证集占原始训练集的比重。以 valid\_ratio=0.1 为例，由于原始训练数据有 5 万张图片，调参时将有 4 万 5 千张图片用于训练（整理后存放在 input\_dir/train）而另外 5 千张图片为验证集（整理后存放在 input\_dir/valid）。

```
In [2]: import os
import shutil

def reorg_cifar10_data(data_dir, label_file, train_dir, test_dir, input_dir,
→ valid_ratio):
    # 读取训练数据标签。
    with open(os.path.join(data_dir, label_file), 'r') as f:
        # 跳过文件头行 (栏名称)。
```

```

lines = f.readlines()[1:]
tokens = [l.rstrip().split(',') for l in lines]
idx_label = dict(((int(idx), label) for idx, label in tokens))
labels = set(idx_label.values())

num_train = len(os.listdir(os.path.join(data_dir, train_dir)))
num_train_tuning = int(num_train * (1 - valid_ratio))
assert 0 < num_train_tuning < num_train
num_train_tuning_per_label = num_train_tuning // len(labels)
label_count = dict()

def mkdir_if_not_exist(path):
    if not os.path.exists(os.path.join(*path)):
        os.makedirs(os.path.join(*path))

# 整理训练和验证集。
for train_file in os.listdir(os.path.join(data_dir, train_dir)):
    idx = int(train_file.split('.')[0])
    label = idx_label[idx]
    mkdir_if_not_exist([data_dir, input_dir, 'train_valid', label])
    shutil.copy(os.path.join(data_dir, train_dir, train_file),
                os.path.join(data_dir, input_dir, 'train_valid', label))
    if label not in label_count or label_count[label] <
    → num_train_tuning_per_label:
        mkdir_if_not_exist([data_dir, input_dir, 'train', label])
        shutil.copy(os.path.join(data_dir, train_dir, train_file),
                    os.path.join(data_dir, input_dir, 'train', label))
        label_count[label] = label_count.get(label, 0) + 1
    else:
        mkdir_if_not_exist([data_dir, input_dir, 'valid', label])
        shutil.copy(os.path.join(data_dir, train_dir, train_file),
                    os.path.join(data_dir, input_dir, 'valid', label))

# 整理测试集。
mkdir_if_not_exist([data_dir, input_dir, 'test', 'unknown'])
for test_file in os.listdir(os.path.join(data_dir, test_dir)):
    shutil.copy(os.path.join(data_dir, test_dir, test_file),
                os.path.join(data_dir, input_dir, 'test', 'unknown'))

```

再次强调，为了使网页编译快一点，我们在这里仅仅使用 100 个训练样本和 1 个测试样本。训练和测试数据的文件夹名称分别为' train\_tiny' 和' test\_tiny'。相应地，我们仅将批量大小设为 1。实际训练和测试时应使用 Kaggle 的完整数据集。由于数据集较大，批量大小 batch\_size 大小可设为一个较大的整数，例如 128。

我们将 10% 的训练样本作为调参时的验证集。

```
In [3]: if demo:  
    # 注意：此处使用小训练集为便于网页编译。Kaggle 的完整数据集应包括 5 万训练样本。  
    train_dir = 'train_tiny'  
    # 注意：此处使用小测试集为便于网页编译。Kaggle 的完整数据集应包括 30 万测试样本。  
    test_dir = 'test_tiny'  
    # 注意：此处相应使用小批量。对 Kaggle 的完整数据集可设较大的整数，例如 128。  
    batch_size = 1  
  
else:  
    train_dir = 'train'  
    test_dir = 'test'  
    batch_size = 128  
  
    data_dir = '../data/kaggle_cifar10'  
    label_file = 'trainLabels.csv'  
    input_dir = 'train_valid_test'  
    valid_ratio = 0.1  
    reorg_cifar10_data(data_dir, label_file, train_dir, test_dir, input_dir,  
    ↪ valid_ratio)
```

### 9.9.3 使用 Gluon 读取整理后的数据集

为避免过拟合，我们在这里使用 `transforms` 来增广数据集。例如我们加入 `transforms.RandomFlipLeftRight()` 即可随机对每张图片做镜面反转。我们也通过 `transforms.Normalize()` 对彩色图像 RGB 三个通道分别做标准化。以下我们列举了所有可能用到的操作，这些操作可以根据需求来决定是否调用，它们的参数也都是可调的。

```
In [4]: from mxnet import autograd  
from mxnet import gluon  
from mxnet import init  
from mxnet import nd  
from mxnet.gluon.data import vision  
from mxnet.gluon.data.vision import transforms  
import numpy as np  
  
transform_train = transforms.Compose([  
    # transforms.CenterCrop(32)  
    # transforms.RandomFlipTopBottom(),  
    # transforms.RandomColorJitter(brightness=0.0, contrast=0.0,  
    ↪ saturation=0.0, hue=0.0),  
    # transforms.RandomLighting(0.0),
```

```

# transforms.Cast('float32'),
# transforms.Resize(32),

# 随机按照 scale 和 ratio 裁剪，并放缩为 32x32 的正方形
transforms.RandomResizedCrop(32, scale=(0.08, 1.0), ratio=(3.0/4.0,
↪ 4.0/3.0)),
    # 随机左右翻转图片
    transforms.RandomFlipLeftRight(),
    # 将图片像素值缩小到 (0,1) 内，并将数据格式从“高 * 宽 * 通道”改为“通道 * 高 * 宽”
    transforms.ToTensor(),
    # 对图片的每个通道做标准化
    transforms.Normalize([0.4914, 0.4822, 0.4465], [0.2023, 0.1994, 0.2010])
])

# 测试时，无需对图像做标准化以外的增强数据处理。
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.4914, 0.4822, 0.4465], [0.2023, 0.1994, 0.2010])
])

```

接下来，我们可以使用 Gluon 中的 `ImageFolderDataset` 类来读取整理后的数据集。注意，我们要在 `loader` 中调用刚刚定义好的图片增广函数。通过 `vision.ImageFolderDataset` 读入的数据是一个 `(image, label)` 组合，`transform_first()` 的作用便是对这个组合中的第一个成员（即读入的图像）做图片增广操作。

```

In [5]: input_str = data_dir + '/' + input_dir + '/'

# 读取原始图像文件。flag=1 说明输入图像有三个通道（彩色）。
train_ds = vision.ImageFolderDataset(input_str + 'train', flag=1)
valid_ds = vision.ImageFolderDataset(input_str + 'valid', flag=1)
train_valid_ds = vision.ImageFolderDataset(input_str + 'train_valid', flag=1)
test_ds = vision.ImageFolderDataset(input_str + 'test', flag=1)

loader = gluon.data.DataLoader
train_data = loader(train_ds.transform_first(transform_train),
                    batch_size, shuffle=True, last_batch='keep')
valid_data = loader(valid_ds.transform_first(transform_test),
                    batch_size, shuffle=True, last_batch='keep')
train_valid_data = loader(train_valid_ds.transform_first(transform_train),
                          batch_size, shuffle=True, last_batch='keep')
test_data = loader(test_ds.transform_first(transform_test),
                   batch_size, shuffle=False, last_batch='keep')

```

```
# 交叉熵损失函数。  
softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

## 9.9.4 设计模型

我们这里使用了ResNet-18模型。我们使用hybridizing来提升执行效率。

请注意：模型可以重新设计，参数也可以重新调整。

```
In [6]: from mxnet.gluon import nn  
from mxnet import nd  
  
class Residual(nn.HybridBlock):  
    def __init__(self, channels, same_shape=True, **kwargs):  
        super(Residual, self).__init__(**kwargs)  
        self.same_shape = same_shape  
        with self.name_scope():  
            strides = 1 if same_shape else 2  
            self.conv1 = nn.Conv2D(channels, kernel_size=3, padding=1,  
                                 strides=strides)  
            self.bn1 = nn.BatchNorm()  
            self.conv2 = nn.Conv2D(channels, kernel_size=3, padding=1)  
            self.bn2 = nn.BatchNorm()  
            if not same_shape:  
                self.conv3 = nn.Conv2D(channels, kernel_size=1,  
                                     strides=strides)  
  
    def hybrid_forward(self, F, x):  
        out = F.relu(self.bn1(self.conv1(x)))  
        out = self.bn2(self.conv2(out))  
        if not self.same_shape:  
            x = self.conv3(x)  
        return F.relu(out + x)  
  
class ResNet(nn.HybridBlock):  
    def __init__(self, num_classes, verbose=False, **kwargs):  
        super(ResNet, self).__init__(**kwargs)  
        self.verbose = verbose  
        with self.name_scope():  
            net = self.net = nn.HybridSequential()  
            # 模块 1
```

```

    net.add(nn.Conv2D(channels=32, kernel_size=3, strides=1,
↪ padding=1))
    net.add(nn.BatchNorm())
    net.add(nn.Activation(activation='relu'))
    # 模块 2
    for _ in range(3):
        net.add(Residual(channels=32))
    # 模块 3
    net.add(Residual(channels=64, same_shape=False))
    for _ in range(2):
        net.add(Residual(channels=64))
    # 模块 4
    net.add(Residual(channels=128, same_shape=False))
    for _ in range(2):
        net.add(Residual(channels=128))
    # 模块 5
    net.add(nn.AvgPool2D(pool_size=8))
    net.add(nn.Flatten())
    net.add(nn.Dense(num_classes))

    def hybrid_forward(self, F, x):
        out = x
        for i, b in enumerate(self.net):
            out = b(out)
            if self.verbose:
                print('Block %d output: %s'%(i+1, out.shape))
        return out

def get_net(ctx):
    num_outputs = 10
    net = ResNet(num_outputs)
    net.initialize(ctx=ctx, init=init.Xavier())
    return net

```

## 9.9.5 训练模型并调参

在过拟合中我们讲过，过度依赖训练数据集的误差来推断测试数据集的误差容易导致过拟合。由于图像分类训练时间可能较长，为了方便，我们这里不再使用 K 折交叉验证，而是依赖验证集的结果来调参。

我们定义模型训练函数。这里我们记录每个 epoch 的训练时间。这有助于我们比较不同模型设计

的时间成本。

```
In [7]: import datetime
      import sys
      sys.path.append('..')
      import gluonbook as gb

      def train(net, train_data, valid_data, num_epochs, lr, wd, ctx, lr_period,
→   lr_decay):
          trainer = gluon.Trainer(
              net.collect_params(), 'sgd', {'learning_rate': lr, 'momentum': 0.9,
→   'wd': wd})

          prev_time = datetime.datetime.now()
          for epoch in range(num_epochs):
              train_loss = 0.0
              train_acc = 0.0
              if epoch > 0 and epoch % lr_period == 0:
                  trainer.set_learning_rate(trainer.learning_rate * lr_decay)
              for data, label in train_data:
                  label = label.astype('float32').as_in_context(ctx)
                  with autograd.record():
                      output = net(data.as_in_context(ctx))
                      loss = softmax_cross_entropy(output, label)
                      loss.backward()
                      trainer.step(batch_size)
                      train_loss += nd.mean(loss).asscalar()
                      train_acc += gb.accuracy(output, label)
              cur_time = datetime.datetime.now()
              h, remainder = divmod((cur_time - prev_time).seconds, 3600)
              m, s = divmod(remainder, 60)
              time_str = "Time %02d:%02d:%02d" % (h, m, s)
              if valid_data is not None:
                  valid_acc = gb.evaluate_accuracy(valid_data, net, ctx)
                  epoch_str = ("Epoch %d. Loss: %f, Train acc %f, Valid acc %f, "
                               % (epoch, train_loss / len(train_data),
                                   train_acc / len(train_data), valid_acc))
              else:
                  epoch_str = ("Epoch %d. Loss: %f, Train acc %f, "
                               % (epoch, train_loss / len(train_data),
                                   train_acc / len(train_data)))
              prev_time = cur_time
              print(epoch_str + time_str + ', lr ' + str(trainer.learning_rate))
```

以下定义训练参数并训练模型。这些参数均可调。为了使网页编译快一点，我们这里将 epoch 数量有意设为 1。事实上，epoch 一般可以调大些，例如 100。

我们将依据验证集的结果不断优化模型设计和调整参数。依据下面的参数设置，优化算法的学习率将在每 80 个 epoch 自乘 0.1。

```
In [8]: ctx = gb.try_gpu()
num_epochs = 1
learning_rate = 0.1
weight_decay = 5e-4
lr_period = 80
lr_decay = 0.1

net = get_net(ctx)
net.hybridize()
train(net, train_data, valid_data, num_epochs, learning_rate,
      weight_decay, ctx, lr_period, lr_decay)

Epoch 0. Loss: 3.774123, Train acc 0.066667, Valid acc 0.100000, Time 00:00:01, lr 0.1
```

### 9.9.6 对测试集分类

当得到一组满意的模型设计和参数后，我们使用全部训练数据集（含验证集）重新训练模型，并对测试集分类。

```
In [9]: import numpy as np
import pandas as pd

net = get_net(ctx)
net.hybridize()
train(net, train_valid_data, None, num_epochs, learning_rate,
      weight_decay, ctx, lr_period, lr_decay)

preds = []
for data, label in test_data:
    output = net(data.as_in_context(ctx))
    preds.extend(output.argmax(axis=1).astype(int).asnumpy())

sorted_ids = list(range(1, len(test_ds) + 1))
sorted_ids.sort(key = lambda x:str(x))

df = pd.DataFrame({'id': sorted_ids, 'label': preds})
```

```
df['label'] = df['label'].apply(lambda x: train_valid_ds.synsets[x])
df.to_csv('submission.csv', index=False)

Epoch 0. Loss: 3.635287, Train acc 0.110000, Time 00:00:01, lr 0.1
```

执行完上述代码后，会生成一个“submission.csv”文件。这个文件符合 Kaggle 比赛要求的提交格式。这时我们可以在 Kaggle 上把对测试集分类的结果提交并查看分类准确率。你需要登录 Kaggle 网站，访问 CIFAR-10 比赛网页，并点击右侧“Submit Predictions”或“Late Submission”按钮 [1]。然后，点击页面下方“Upload Submission File”选择需要提交的分类结果文件。最后，点击页面最下方的“Make Submission”按钮就可以查看结果了。

### 9.9.7 小结

- CIFAR-10 是深度学习在计算机视觉领域的一个重要数据集。

### 9.9.8 练习

- 使用 Kaggle 完整 CIFAR-10 数据集，把 batch\_size 和 num\_epochs 分别改为 128 和 100，可以在 Kaggle 上拿到什么样的准确率和名次？
- 如果不使用增强数据的方法能拿到什么样的准确率？
- 扫码直达讨论区，在社区交流方法和结果。相信你一定会有收获。

### 9.9.9 扫码直达讨论区



### 9.9.10 参考文献

- [1] Kaggle CIFAR-10 比赛网址。<https://www.kaggle.com/c/cifar-10>

## 9.10 实战 Kaggle 比赛：识别 120 种狗 (ImageNet Dogs)

我们在本章中选择了 Kaggle 中的 120 种狗类识别问题。这是著名的 ImageNet 的子集数据集。与之前的 CIFAR-10 原始图像分类问题不同，本问题中的图片文件大小更接近真实照片大小，且大小不一。本问题的输出也变的更加通用：我们将输出每张图片对应 120 种狗的分别概率。

### 9.10.1 Kaggle 中的 CIFAR-10 原始图像分类问题

Kaggle 是一个著名的供机器学习爱好者交流的平台。为了便于提交结果，请大家注册 [Kaggle](#) 账号。然后请大家先点击 [120 种狗类识别问题](#) 了解有关本次比赛的信息。

Playground Prediction Competition

## Dog Breed Identification

Determine the breed of a dog in an image

Kaggle · 176 teams · 4 months to go

Overview Data Kernels Discussion Leaderboard Rules Team My Submissions Submit Predictions

Overview

Description Who's a good dog? Who likes ear scratches? Well, it seems those fancy deep neural networks don't have all the answers. However, maybe they can answer that ubiquitous question we all ask when meeting a four-legged stranger: what kind of good pup is that?

Evaluation In this playground competition, you are provided a strictly canine subset of [ImageNet](#) in order to practice fine-grained image categorization. How well you can tell your Norfolk Terriers from your Norwich Terriers? With 120 breeds of dogs and a limited number training images per class, you might find the problem more, err, ruff than you anticipated.



### 9.10.2 整理原始数据集

比赛数据分为训练数据集和测试数据集。训练集包含 10,222 张图片。测试集包含 10,357 张图片。两个数据集都是 jpg 彩色图片，大小接近真实照片大小，且大小不一。训练集一共有 120 类狗的图片。

#### 下载数据集

登录 Kaggle 后，数据可以从[120 种狗类识别问题](#)中下载。

- 训练数据集 train.zip 下载地址
- 测试数据集 test.zip 下载地址
- 训练数据标签 label.csv.zip 下载地址

## 解压数据集

训练数据集 train.zip 和测试数据集 test.zip 都是压缩格式，下载后它们的路径可以如下：

- ../data/kaggle\_dog/train.zip
- ../data/kaggle\_dog/test.zip
- ../data/kaggle\_dog/labels.csv.zip

为了使网页编译快一点，我们在 git repo 里仅仅存放小数据样本 ('train\_valid\_test\_tiny.zip')。执行以下代码会从 git repo 里解压生成小数据样本。

```
In [1]: # 如果训练下载的 Kaggle 的完整数据集，把 demo 改为 False。
demo = True
data_dir = '../data/kaggle_dog'

if demo:
    zipfiles= ['train_valid_test_tiny.zip']
else:
    zipfiles= ['train.zip', 'test.zip', 'labels.csv.zip']

import zipfile
for fin in zipfiles:
    with zipfile.ZipFile(data_dir + '/' + fin, 'r') as zin:
        zin.extractall(data_dir)
```

## 整理数据集

对于 Kaggle 的完整数据集，我们需要定义下面的 reorg\_dog\_data 函数来整理一下。整理后，同一类狗的图片将出现在在同一个文件夹下，便于 Gluon 稍后读取。

函数中的参数如 data\_dir、train\_dir 和 test\_dir 对应上述数据存放路径及原始训练和测试的图片集文件夹名称。参数 label\_file 为训练数据标签的文件名称。参数 input\_dir 是整理后数据集文件夹名称。参数 valid\_ratio 是验证集中每类狗的数量占原始训练集中数量最少一类的狗的数量(66)的比重。

```

In [2]: import math
        import os
        import shutil
        from collections import Counter

def reorg_dog_data(data_dir, label_file, train_dir, test_dir, input_dir,
                   valid_ratio):
    # 读取训练数据标签。
    with open(os.path.join(data_dir, label_file), 'r') as f:
        # 跳过文件头行 (栏名称)。
        lines = f.readlines()[1:]
        tokens = [l.rstrip().split(',') for l in lines]
        idx_label = dict(((idx, label) for idx, label in tokens))
        labels = set(idx_label.values())

    num_train = len(os.listdir(os.path.join(data_dir, train_dir)))
    # 训练集中数量最少一类的狗的数量。
    min_num_train_per_label = (
        Counter(idx_label.values()).most_common()[:-2:-1][0][1])
    # 验证集中每类狗的数量。
    num_valid_per_label = math.floor(min_num_train_per_label * valid_ratio)
    label_count = dict()

    def mkdir_if_not_exist(path):
        if not os.path.exists(os.path.join(*path)):
            os.makedirs(os.path.join(*path))

    # 整理训练和验证集。
    for train_file in os.listdir(os.path.join(data_dir, train_dir)):
        idx = train_file.split('.')[0]
        label = idx_label[idx]
        mkdir_if_not_exist([data_dir, input_dir, 'train_valid', label])
        shutil.copy(os.path.join(data_dir, train_dir, train_file),
                    os.path.join(data_dir, input_dir, 'train_valid', label))
        if label not in label_count or label_count[label] <
        num_valid_per_label:
            mkdir_if_not_exist([data_dir, input_dir, 'valid', label])
            shutil.copy(os.path.join(data_dir, train_dir, train_file),
                        os.path.join(data_dir, input_dir, 'valid', label))
            label_count[label] = label_count.get(label, 0) + 1
        else:
            mkdir_if_not_exist([data_dir, input_dir, 'train', label])
            shutil.copy(os.path.join(data_dir, train_dir, train_file),

```

```
        os.path.join(data_dir, input_dir, 'train', label))

# 整理测试集。
mkdir_if_not_exist([data_dir, input_dir, 'test', 'unknown'])
for test_file in os.listdir(os.path.join(data_dir, test_dir)):
    shutil.copy(os.path.join(data_dir, test_dir, test_file),
                os.path.join(data_dir, input_dir, 'test', 'unknown'))
```

再次强调，为了使网页编译快一点，我们在这里仅仅使用小数据样本。相应地，我们仅将批量大小设为 2。实际训练和测试时应使用 Kaggle 的完整数据集并调用 `reorg_dog_data` 函数整理便于 Gluon 读取的格式。由于数据集较大，批量大小 `batch_size` 大小可设为一个较大的整数，例如 128。

```
In [3]: if demo:
    # 注意：此处使用小数据集为便于网页编译。
    input_dir = 'train_valid_test_tiny'
    # 注意：此处相应使用小批量。对 Kaggle 的完整数据集可设较大的整数，例如 128。
    batch_size = 2
else:
    label_file = 'labels.csv'
    train_dir = 'train'
    test_dir = 'test'
    input_dir = 'train_valid_test'
    batch_size = 128
    valid_ratio = 0.1
    reorg_dog_data(data_dir, label_file, train_dir, test_dir, input_dir,
                   valid_ratio)
```

### 9.10.3 使用 Gluon 读取整理后的数据集

为避免过拟合，我们在这里使用 `transforms` 来增广数据集。例如我们加入 `transforms.RandomFlipLeftRight()` 即可随机对每张图片做镜面反转。以下我们列举了所有可能用到的操作，这些操作可以根据需求来决定是否调用，它们的参数也都是可调的。

```
In [4]: from mxnet import autograd
from mxnet import gluon
from mxnet import init
from mxnet import nd
from mxnet.gluon.data import vision
from mxnet.gluon.data.vision import transforms
import numpy as np
```

```

transform_train = transforms.Compose([
    # transforms.CenterCrop(32),
    # transforms.RandomFlipTopBottom(),
    # transforms.RandomColorJitter(brightness=0.0, contrast=0.0,
    ↵ saturation=0.0, hue=0.0),
    # transforms.RandomLighting(0.0),
    # transforms.Cast('float32'),

    # 将图片按比例放缩至短边为 256 像素
    transforms.Resize(256),
    # 随机按照 scale 和 ratio 裁剪，并放缩为 224x224 的正方形
    transforms.RandomResizedCrop(224, scale=(0.08, 1.0), ratio=(3.0/4.0,
    ↵ 4.0/3.0)),
    # 随机左右翻转图片
    transforms.RandomFlipLeftRight(),
    # 将图片像素值缩小到 (0,1) 内，并将数据格式从“高 * 宽 * 通道”改为“通道 * 高 * 宽”
    transforms.ToTensor(),
    # 对图片的每个通道做标准化
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# 去掉随机裁剪/翻转，保留确定性的图像预处理结果
transform_test = transforms.Compose([
    transforms.Resize(256),
    # 将图片中央的 224x224 正方形区域裁剪出来
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

```

接下来，我们可以使用 Gluon 中的 `ImageFolderDataset` 类来读取整理后的数据集。注意，我们要在 `loader` 中调用刚刚定义好的图片增广函数。通过 `vision.ImageFolderDataset` 读入的数据是一个 `(image, label)` 组合，`transform_first()` 的作用便是对这个组合中的第一个成员（即读入的图像）做图片增广操作。

```

In [5]: input_str = data_dir + '/' + input_dir + '/'

# 读取原始图像文件。flag=1 说明输入图像有三个通道（彩色）。
train_ds = vision.ImageFolderDataset(input_str + 'train', flag=1)
valid_ds = vision.ImageFolderDataset(input_str + 'valid', flag=1)
train_valid_ds = vision.ImageFolderDataset(input_str + 'train_valid', flag=1)
test_ds = vision.ImageFolderDataset(input_str + 'test', flag=1)

```

```
loader = gluon.data.DataLoader
train_data = loader(train_ds.transform_first(transform_train),
                    batch_size, shuffle=True, last_batch='keep')
valid_data = loader(valid_ds.transform_first(transform_test),
                     batch_size, shuffle=True, last_batch='keep')
train_valid_data = loader(train_valid_ds.transform_first(transform_train),
                          batch_size, shuffle=True, last_batch='keep')
test_data = loader(test_ds.transform_first(transform_test),
                   batch_size, shuffle=False, last_batch='keep')

# 交叉熵损失函数。
softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

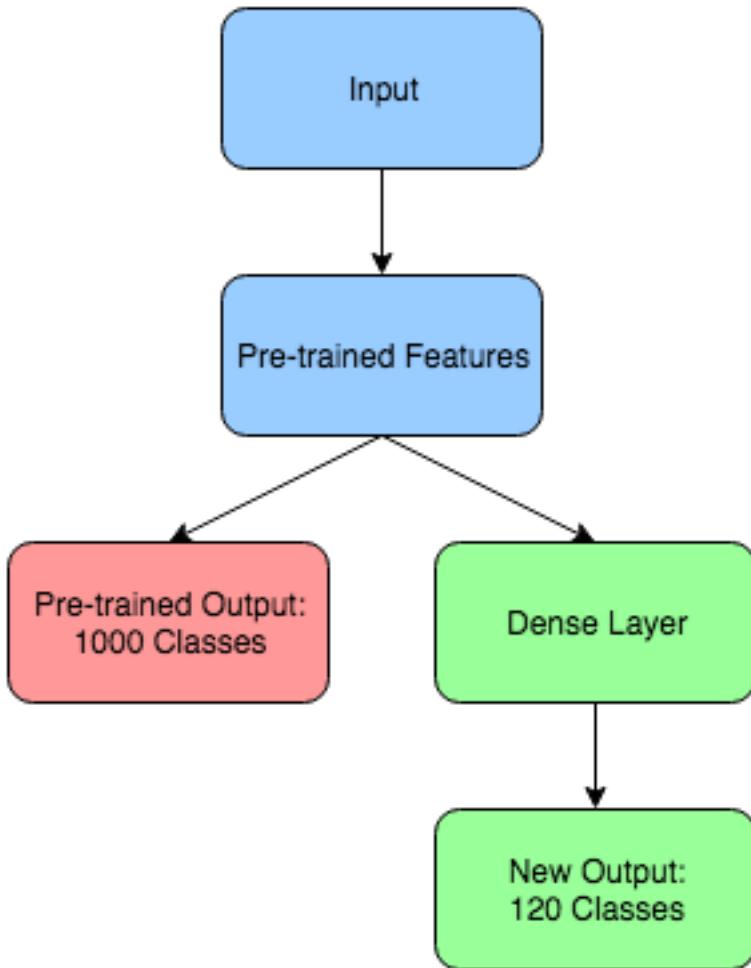
#### 9.10.4 设计模型

这个比赛的数据属于 ImageNet 数据集的子集，因此我们可以借助迁移学习的思想，选用在 ImageNet 全集上预训练过的模型，并通过微调在新数据集上进行训练。Gluon 提供了不少预训练模型，综合考虑模型大小与准确率，我们选择使用 ResNet-34。

这里，我们使用与前述教程略微不同的迁移学习方法。在新的训练数据与预训练数据相似的情况下，我们认为原有特征是可重用的。基于这个原因，在一个预训练好的新模型上，我们可以不去改变原已训练好的权重，而是在原网络结构上新加一个小的输出网络。

在训练过程中，我们让训练图片通过正向传播经过原有特征层与新定义的全连接网络，然后只在这个小网络上通过反向传播更新权重。这样的做法既能够节省在整个模型进行后向传播的时间，也能节省在特征层上储存梯度所需要的内存空间。

注意，我们在之前定义的数据预处理函数里用了 ImageNet 数据集上的均值和标准差做标准化，这样才能保证预训练模型能够捕捉正确的数据特征。



首先我们定义一个网络，并拿到预训练好的 ResNet-34 模型权重。接下来我们新定义一个两层的全连接网络作为输出层，并初始化其权重，为接下来的训练做准备。

```

In [6]: from mxnet.gluon import nn
        from mxnet import nd
        from mxnet.gluon.model_zoo import vision as models

def get_net(ctx):
    # 设置 pretrained=True 就能拿到预训练模型的权重，第一次使用需要联网下载
    finetune_net = models.resnet34_v2(pretrained=True)

```

```

# 定义新的输出网络
finetune_net.output_new = nn.HybridSequential(prefix='')

# 定义 256 个神经元的全连接层
finetune_net.output_new.add(nn.Dense(256, activation='relu'))
# 定义 120 个神经元的全连接层，输出分类预测
finetune_net.output_new.add(nn.Dense(120))
# 初始化这个输出网络
finetune_net.output_new.initialize(init.Xavier(), ctx=ctx)

# 把网络参数分配到即将用于计算的 CPU/GPU 上
finetune_net.collect_params().reset_ctx(ctx)
return finetune_net

```

### 9.10.5 训练模型并调参

在过拟合中我们讲过，过度依赖训练数据集的误差来推断测试数据集的误差容易导致过拟合。由于图像分类训练时间可能较长，为了方便，我们这里不再使用 K 折交叉验证，而是依赖验证集的结果来调参。

我们定义损失函数以便于计算验证集上的损失函数值。我们也定义了模型训练函数，其中的优化算法和参数都是可以调的。

注意，我们为了只更新新的输出层参数，做了两处修改：

1. 在 `gluon.Trainer` 里只对 `net.output_new.collect_params()` 定义了优化方法和参数。
2. 在训练时只在新输出层上记录自动求导的结果。

```

In [7]: import datetime
import sys
sys.path.append('..')
import gluonbook as gb

def get_loss(data, net, ctx):
    loss = 0.0
    for feas, label in data:
        label = label.as_in_context(ctx)
        # 计算特征层的结果
        output_features = net.features(feas.as_in_context(ctx))
        # 将特征层的结果作为输入，计算全连接网络的结果

```

```

        output = net.output_new(output_features)
        cross_entropy = softmax_cross_entropy(output, label)
        loss += nd.mean(cross_entropy).asscalar()
    return loss / len(data)

def train(net, train_data, valid_data, num_epochs, lr, wd, ctx, lr_period,
          lr_decay):
    # 只在新的全连接网络的参数上进行训练
    trainer = gluon.Trainer(net.output_new.collect_params(),
                            'sgd', {'learning_rate': lr, 'momentum': 0.9,
←   'wd': wd})
    prev_time = datetime.datetime.now()
    for epoch in range(num_epochs):
        train_loss = 0.0
        if epoch > 0 and epoch % lr_period == 0:
            trainer.set_learning_rate(trainer.learning_rate * lr_decay)
        for data, label in train_data:
            label = label.astype('float32').as_in_context(ctx)
            # 正向传播计算特征层的结果
            output_features = net.features(data.as_in_context(ctx))
            with autograd.record():
                # 将特征层的结果作为输入，计算全连接网络的结果
                output = net.output_new(output_features)
                loss = softmax_cross_entropy(output, label)
            # 反向传播与权重更新只发生在全连接网络上
            loss.backward()
            trainer.step(batch_size)
            train_loss += nd.mean(loss).asscalar()
        cur_time = datetime.datetime.now()
        h, remainder = divmod((cur_time - prev_time).seconds, 3600)
        m, s = divmod(remainder, 60)
        time_str = "Time %02d:%02d:%02d" % (h, m, s)
        if valid_data is not None:
            valid_loss = get_loss(valid_data, net, ctx)
            epoch_str = ("Epoch %d. Train loss: %f, Valid loss %f, "
                         % (epoch, train_loss / len(train_data), valid_loss))
        else:
            epoch_str = ("Epoch %d. Train loss: %f, "
                         % (epoch, train_loss / len(train_data)))
        prev_time = cur_time
        print(epoch_str + time_str + ', lr ' + str(trainer.learning_rate))

```

以下定义训练参数并训练模型。这些参数均可调。为了使网页编译快一点，我们这里将 epoch 数

量有意设为 1。事实上，epoch 一般可以调大些。我们将依据验证集的结果不断优化模型设计和调整参数。

另外，微调一个预训练模型往往不需要特别久的额外训练。依据下面的参数设置，优化算法的学习率设为 0.01，并将在每 10 个 epoch 自乘 0.1。

```
In [8]: ctx = gb.try_gpu()
num_epochs = 1
learning_rate = 0.01
weight_decay = 1e-4
lr_period = 10
lr_decay = 0.1

net = get_net(ctx)
net.hybridize()
train(net, train_data, valid_data, num_epochs, learning_rate,
      weight_decay, ctx, lr_period, lr_decay)

Epoch 0. Train loss: 5.442084, Valid loss 4.745717, Time 00:00:02, lr 0.01
```

## 9.10.6 对测试集分类

当得到一组满意的模型设计和参数后，我们使用全部训练数据集（含验证集）重新训练模型，并对测试集分类。注意，我们要用刚训练好的新输出层做预测。

```
In [9]: import numpy as np

net = get_net(ctx)
net.hybridize()
train(net, train_valid_data, None, num_epochs, learning_rate, weight_decay,
      ctx, lr_period, lr_decay)

outputs = []
for data, label in test_data:
    # 计算特征层的结果
    output_features = net.features(data.as_in_context(ctx))
    # 将特征层的结果作为输入，计算全连接网络的结果
    output = nd.softmax(net.output_new(output_features))
    outputs.extend(output.asnumpy())
ids = sorted(os.listdir(os.path.join(data_dir, input_dir, 'test/unknown')))
with open('submission.csv', 'w') as f:
    f.write('id,' + ','.join(train_valid_ds.synsets) + '\n')
    for i, output in zip(ids, outputs):
```

```
f.write(i.split('.')[0] + ',' + '.join(  
    [str(num) for num in output]) + '\n')  
  
Epoch 0. Train loss: 5.069541, Time 00:00:02, lr 0.01
```

执行完上述代码后，会生成一个“submission.csv”文件。这个文件符合 Kaggle 比赛要求的提交格式。这时我们可以在 Kaggle 上把对测试集分类的结果提交并查看分类准确率。你需要登录 Kaggle 网站，访问 ImageNet Dogs 比赛网页，并点击右侧“Submit Predictions”或“Late Submission”按钮 [1]。然后，点击页面下方“Upload Submission File”选择需要提交的分类结果文件。最后，点击页面最下方的“Make Submission”按钮就可以查看结果了。

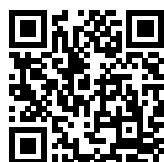
### 9.10.7 小结

- 我们可以利用在 ImageNet 数据集上预训练的模型对它的子集数据集做分类。

### 9.10.8 练习

- 使用 Kaggle 完整数据集，把 batch\_size 和 num\_epochs 分别调大些，可以在 Kaggle 上拿到什么样的准确率和名次？
- 扫码直达讨论区，在社区交流方法和结果。相信你一定会有收获。

### 9.10.9 扫码直达讨论区



### 9.10.10 参考文献

[1] Kaggle ImageNet Dogs 比赛网址。<https://www.kaggle.com/c/dog-breed-identification>



---

## 自然语言处理

---

自然语言处理关注计算机与人类自然语言的交互。在实际中，我们常常使用自然语言处理技术，例如“循环神经网络”一章中介绍的语言模型，来处理和分析大量的自然语言数据。

本章中，我们将先介绍如何用向量表示词，并应用这些词向量求近义词和类比词。接着，在文本分类任务中，我们进一步应用词向量分析文本情感。此外，自然语言处理任务中很多输出是不定长的，例如任意长度的句子。我们将描述应对这类问题的编码器—解码器模型以及注意力机制，并将它们应用于机器翻译中。

### 10.1 词向量：word2vec

自然语言是一套用来表达含义的复杂系统。在这套系统中，词是表义的基本单元。顾名思义，词向量是用来表示词的向量，通常也被认为是词的特征向量。近年来，词向量已逐渐成为自然语言处理的基础知识。

那么，我们应该如何使用向量表示词呢？

### 10.1.1 为何不采用 one-hot 向量

我们在“[循环神经网络——从零开始](#)”一节中使用 one-hot 向量表示字符，并以字符为词。回忆一下，假设词典中不同词的数量（词典大小）为  $N$ ，每个词可以和从 0 到  $N - 1$  的连续整数一一对应。这些与词对应的整数也叫词的索引。假设一个词的索引为  $i$ ，为了得到该词的 one-hot 向量表示，我们创建一个全 0 的长为  $N$  的向量，并将其第  $i$  位设成 1。

然而，使用 one-hot 词向量通常并不是一个好选择。一个主要的原因是，one-hot 词向量无法表达不同词之间的相似度，例如余弦相似度。由于任意一对向量  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$  的余弦相似度为

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \in [-1, 1],$$

任何一对词的 one-hot 向量的余弦相似度都为 0。

### 10.1.2 word2vec

2013 年，Google 团队发表了 word2vec 工具 [1]。word2vec 工具主要包含两个模型：跳字模型 (skip-gram) 和连续词袋模型 (continuous bag of words，简称 CBOW)，以及两种近似训练法：负采样 (negative sampling) 和层序 softmax (hierarchical softmax)。值得一提的是，word2vec 词向量可以较好地表达不同词之间的相似和类比关系。

word2vec 自提出后被广泛应用在自然语言处理任务中。它的模型和训练方法也启发了很多后续的词向量模型。本节将重点介绍 word2vec 的模型和训练方法。

### 10.1.3 模型

word2vec 工具主要包含跳字模型和连续词袋模型。下面将分别介绍它们。

#### 跳字模型

在跳字模型中，我们用一个词来预测它在文本序列周围的词。举个例子，假设文本序列是“the”、“man”、“hit”、“his”和“son”。跳字模型所关心的是，给定“hit”生成邻近词“the”、“man”、“his”和“son”的条件概率。在这个例子中，“hit”叫中心词，“the”、“man”、“his”和“son”叫背景词。由于“hit”只生成与它距离不超过 2 的背景词，该时间窗口的大小为 2。

我们来描述一下跳字模型。

假设词典索引集  $\mathcal{V}$  的大小为  $|\mathcal{V}|$ , 且  $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}| - 1\}$ 。给定一个长度为  $T$  的文本序列中, 时  
间步  $t$  的词为  $w^{(t)}$ 。当时间窗口大小为  $m$  时, 跳字模型需要最大化给定任一中心词生成所有背景  
词的概率

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} \mathbb{P}(w^{(t+j)} | w^{(t)}).$$

上式的最大似然估计与最小化以下损失函数等价:

$$-\frac{1}{T} \sum_{t=1}^T \sum_{-m \leq j \leq m, j \neq 0} \log \mathbb{P}(w^{(t+j)} | w^{(t)}).$$

我们可以用  $\mathbf{v}$  和  $\mathbf{u}$  分别表示中心词和背景词的向量。换言之, 对于词典中索引为  $i$  的词, 它在作为  
为中心词和背景词时的向量表示分别是  $\mathbf{v}_i$  和  $\mathbf{u}_i$ 。而词典中所有词的这两种向量正是跳字模型所  
要学习的模型参数。为了将模型参数植入损失函数, 我们需要使用模型参数表达损失函数中的给  
定中心词生成背景词的条件概率。给定中心词, 假设生成各个背景词是相互独立的。设中心词  $w_c$   
在词典中索引为  $c$ , 背景词  $w_o$  在词典中索引为  $o$ , 损失函数中的给定中心词生成背景词的条件概  
率可以通过 softmax 函数定义为

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

当序列长度  $T$  较大时, 我们通常在每次迭代时随机采样一个较短的子序列来计算有关该子序列的  
损失。然后, 根据该损失计算词向量的梯度并迭代词向量。具体算法可以参考 “[梯度下降和随  
机梯度下降——从零开始](#)” 一节。作为一个具体的例子, 下面我们看看如何计算随机采样的子序  
列的损失有关中心词向量的梯度。和上面提到的长度为  $T$  的文本序列的损失函数类似, 随机采  
样的子序列的损失实际上是对子序列中给定中心词生成背景词的条件概率的对数求平均。通过微  
分, 我们可以得到上式中条件概率的对数有关中心词向量  $\mathbf{v}_c$  的梯度

$$\frac{\partial \log \mathbb{P}(w_o | w_c)}{\partial \mathbf{v}_c} = \mathbf{u}_o - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \mathbf{u}_j.$$

该式也可写作

$$\frac{\partial \log \mathbb{P}(w_o | w_c)}{\partial \mathbf{v}_c} = \mathbf{u}_o - \sum_{j \in \mathcal{V}} \mathbb{P}(w_j | w_c) \mathbf{u}_j.$$

随机采样的子序列有关其他词向量的梯度同理可得。训练模型时, 每一次迭代实际上是用这些梯  
度来迭代子序列中出现过的中心词和背景词的向量。训练结束后, 对于词典中的任一索引为  $i$  的词,  
我们均得到该词作为中心词和背景词的两组词向量  $\mathbf{v}_i$  和  $\mathbf{u}_i$ 。在自然语言处理应用中, 我们  
会使用跳字模型的中心词向量。

## 连续词袋模型

连续词袋模型与跳字模型类似。与跳字模型最大的不同是，连续词袋模型用一个中心词在文本序列周围的词来预测该中心词。举个例子，假设文本序列为“the”、“man”、“hit”、“his”和“son”。连续词袋模型所关心的是，邻近词“the”、“man”、“his”和“son”一起生成中心词“hit”的概率。

假设词典索引集  $\mathcal{V}$  的大小为  $|\mathcal{V}|$ ，且  $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}| - 1\}$ 。给定一个长度为  $T$  的文本序列中，时间步  $t$  的词为  $w^{(t)}$ 。当时间窗口大小为  $m$  时，连续词袋模型需要最大化由背景词生成任一中心词的概率

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

上式的最大似然估计与最小化以下损失函数等价：

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

我们可以用  $\mathbf{v}$  和  $\mathbf{u}$  分别表示背景词和中心词的向量（注意符号和跳字模型中的不同）。换言之，对于词典中索引为  $i$  的词，它在作为背景词和中心词时的向量表示分别是  $\mathbf{v}_i$  和  $\mathbf{u}_i$ 。而词典中所有词的这两种向量正是连续词袋模型所要学习的模型参数。为了将模型参数植入损失函数，我们需要使用模型参数表达损失函数中的给定背景词生成中心词的概率。设中心词  $w_c$  在词典中索引为  $c$ ，背景词  $w_{o_1}, \dots, w_{o_{2m}}$  在词典中索引为  $o_1, \dots, o_{2m}$ ，损失函数中的给定背景词生成中心词的概率可以通过 softmax 函数定义为

$$\mathbb{P}(w_c | w_{o_1}, \dots, w_{o_{2m}}) = \frac{\exp(\mathbf{u}_c^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})/(2m))}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})/(2m))}.$$

和跳字模型一样，当序列长度  $T$  较大时，我们通常在每次迭代时随机采样一个较短的子序列来计算有关该子序列的损失。然后，根据该损失计算词向量的梯度并迭代词向量。通过微分，我们可以计算出上式中条件概率的对数有关任一背景词向量  $\mathbf{v}_{o_i}$  ( $i = 1, \dots, 2m$ ) 的梯度为：

$$\frac{\partial \log \mathbb{P}(w_c | w_{o_1}, \dots, w_{o_{2m}})}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left( \mathbf{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \mathbf{u}_j \right).$$

该式也可写作

$$\frac{\partial \log \mathbb{P}(w_c | w_{o_1}, \dots, w_{o_{2m}})}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left( \mathbf{u}_c - \sum_{j \in \mathcal{V}} \mathbb{P}(w_j | w_c) \mathbf{u}_j \right).$$

随机采样的子序列有关其他词向量的梯度同理可得。和跳字模型一样，训练结束后，对于词典中的任一索引为  $i$  的词，我们均得到该词作为背景词和中心词的两组词向量  $\mathbf{v}_i$  和  $\mathbf{u}_i$ 。在自然语言处理应用中，我们会使用连续词袋模型的背景词向量。

#### 10.1.4 近似训练法

我们可以看到，无论是跳字模型还是连续词袋模型，每一步梯度计算的开销与词典  $\mathcal{V}$  的大小相关。当词典较大时，例如几十万到上百万，这种训练方法的计算开销会较大。因此，我们将使用近似的方法来计算这些梯度，从而减小计算开销。常用的近似训练法包括负采样和层序 softmax。

##### 负采样

我们以跳字模型为例讨论负采样。

实际上，词典  $\mathcal{V}$  的大小之所以会在损失中出现，是因为给定中心词  $w_c$  生成背景词  $w_o$  的条件概率  $\mathbb{P}(w_o | w_c)$  使用了 softmax 运算，而 softmax 运算正是考虑了背景词可能是词典中的任一词，并体现在分母上。

不妨换个角度考虑给定中心词生成背景词的条件概率。假设中心词  $w_c$  生成背景词  $w_o$  由以下相互独立事件联合组成来近似：

- 中心词  $w_c$  和背景词  $w_o$  同时出现时间窗口。
- 中心词  $w_c$  和第 1 个噪声词  $w_1$  不同时出现在该时间窗口（噪声词  $w_1$  按噪声词分布  $\mathbb{P}(w)$  随机生成，且假设一定和  $w_c$  不同时出现在该时间窗口）。
- ...
- 中心词  $w_c$  和第  $K$  个噪声词  $w_K$  不同时出现在该时间窗口（噪声词  $w_K$  按噪声词分布  $\mathbb{P}(w)$  随机生成，且假设一定和  $w_c$  不同时出现在该时间窗口）。

下面，我们可以使用  $\sigma(x) = 1/(1 + \exp(-x))$  函数来表达中心词  $w_c$  和背景词  $w_o$  同时出现在该训练数据窗口的概率：

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

那么，给定中心词  $w_c$  生成背景词  $w_o$  的条件概率的对数可以近似为

$$\log \mathbb{P}(w_o | w_c) = \log \left( \mathbb{P}(D = 1 | w_o, w_c) \prod_{k=1, w_k \sim \mathbb{P}(w)}^K \mathbb{P}(D = 0 | w_k, w_c) \right).$$

假设噪声词  $w_k$  在词典中的索引为  $i_k$ , 上式可改写为

$$\log \mathbb{P}(w_o | w_c) = \log \frac{1}{1 + \exp(-\mathbf{u}_o^\top \mathbf{v}_c)} + \sum_{k=1, w_k \sim \mathbb{P}(w)}^K \log \left( 1 - \frac{1}{1 + \exp(-\mathbf{u}_{i_k}^\top \mathbf{v}_c)} \right).$$

因此, 有关给定中心词  $w_c$  生成背景词  $w_o$  的损失是

$$-\log \mathbb{P}(w_o | w_c) = -\log \frac{1}{1 + \exp(-\mathbf{u}_o^\top \mathbf{v}_c)} - \sum_{k=1, w_k \sim \mathbb{P}(w)}^K \log \frac{1}{1 + \exp(\mathbf{u}_{i_k}^\top \mathbf{v}_c)}.$$

假设词典  $\mathcal{V}$  很大, 每次迭代的计算开销由  $\mathcal{O}(|\mathcal{V}|)$  变为  $\mathcal{O}(K)$ 。当我们把  $K$  取较小值时, 负采样每次迭代的计算开销将较小。

当然, 我们也可以对连续词袋模型进行负采样。有关给定背景词  $w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}$  生成中心词  $w_c$  的损失

$$-\log \mathbb{P}(w^{(t)} | w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)})$$

在负采样中可以近似为

$$-\log \frac{1}{1 + \exp(-\mathbf{u}_c^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})/(2m))} - \sum_{k=1, w_k \sim \mathbb{P}(w)}^K \log \frac{1}{1 + \exp((\mathbf{u}_{i_k}^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})/(2m)))}.$$

同样, 当我们把  $K$  取较小值时, 负采样每次迭代的计算开销将较小。

### 10.1.5 层序 softmax

层序 softmax 是另一种常用的近似训练法。它利用了二叉树这一数据结构。树的每个叶子节点代表着词典  $\mathcal{V}$  中的每个词。我们以图 10.1 为例来描述层序 softmax 的工作机制。

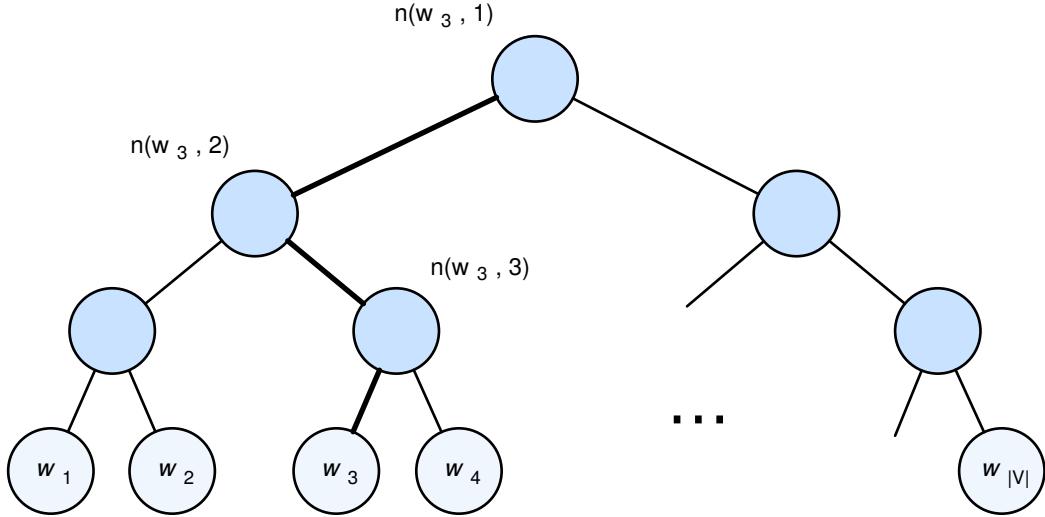


图 10.1: 层序 softmax。树的每个叶子节点代表着词典的每个词。

假设  $L(w)$  为从二叉树的根节点到词  $w$  的叶子节点的路径（包括根和叶子节点）上的节点数。设  $n(w, j)$  为该路径上第  $j$  个节点，并设该节点的向量为  $\mathbf{u}_{n(w,j)}$ 。以图 10.1 为例， $L(w_3) = 4$ 。设词典中的词  $w_i$  的词向量为  $\mathbf{v}_i$ 。那么，跳字模型和连续词袋模型所需要计算的给定词  $w_i$  生成词  $w$  的条件概率为：

$$\mathbb{P}(w \mid w_i) = \prod_{j=1}^{L(w)-1} \sigma \left( [n(w, j+1) = \text{leftChild}(n(w, j))] \cdot \mathbf{u}_{n(w,j)}^\top \mathbf{v}_i \right),$$

其中  $\sigma(x) = 1/(1 + \exp(-x))$ ， $\text{leftChild}(n)$  是节点  $n$  的左孩子节点，如果判断  $x$  为真， $[x] = 1$ ；反之  $[x] = -1$ 。由于  $\sigma(x) + \sigma(-x) = 1$ ，给定词  $w_i$  生成词典  $\mathcal{V}$  中任一词的条件概率之和为 1 这一条件也将满足：

$$\sum_{w \in \mathcal{V}} \mathbb{P}(w \mid w_i) = 1.$$

让我们计算图 10.1 中给定词  $w_i$  生成词  $w_3$  的条件概率。我们需要将  $w_i$  的词向量  $\mathbf{v}_i$  和根节点到  $w_3$  路径上的非叶子节点向量一一求内积。由于在二叉树中由根节点到叶子节点  $w_3$  的路径上需要向左、向右、再向左地遍历，我们得到

$$\mathbb{P}(w_3 \mid w_i) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_i) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_i) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_i).$$

在使用 softmax 的跳字模型和连续词袋模型中，词向量和二叉树中非叶子节点向量是需要学习的模型参数。假设词典  $\mathcal{V}$  很大，每次迭代的计算开销由  $\mathcal{O}(|\mathcal{V}|)$  下降至  $\mathcal{O}(\log_2 |\mathcal{V}|)$ 。

### 10.1.6 小结

- word2vec 工具中的跳字模型和连续词袋模型通常使用近似训练法，例如负采样和层序 softmax，从而减小训练的计算开销。

### 10.1.7 练习

- 噪声词采样概率  $\mathbb{P}(w)$  在实际中被建议设为  $w$  词频与总词频的比的  $3/4$  次方。这样做有什么好处？提示：想想  $0.99^{3/4}$  和  $0.01^{3/4}$  的大小。
- 一些“the”和“a”之类的英文高频词会对结果产生什么影响？该如何处理？提示：可参考 word2vec 论文第 2.3 节 [2]。
- 如何训练包括例如“new york”在内的词组向量？提示：可参考 word2vec 论文第 4 节 [2]。

### 10.1.8 扫码直达讨论区



### 10.1.9 参考文献

[1] word2vec 工具. <https://code.google.com/archive/p/word2vec/>

[2] Mikolov, Tomas, et al. “Distributed representations of words and phrases and their compositionality.” NIPS. 2013.

## 10.2 词向量：GloVe 和 fastText

在 word2vec 被提出以后，很多其他词向量模型也陆续发表出来。本节介绍其中比较有代表性的两个模型。它们分别是 2014 年由斯坦福团队发表的 GloVe 和 2017 年由 Facebook 团队发表的 fastText。

### 10.2.1 GloVe

GloVe 使用了词与词之间的共现 (co-occurrence) 信息。我们定义  $X$  为共现词频矩阵，其中元素  $x_{ij}$  为词  $j$  出现在词  $i$  的背景的次数。这里的“背景”有多种可能的定义。举个例子，在一段文本序列中，如果词  $j$  出现在词  $i$  左边或者右边不超过 10 个词的距离，我们可以认为词  $j$  出现在词  $i$  的背景一次。令  $x_i = \sum_k x_{ik}$  为任意词出现在词  $i$  的背景的次数。那么，

$$P_{ij} = \mathbb{P}(j | i) = \frac{x_{ij}}{x_i}$$

为词  $j$  在词  $i$  的背景中出现的条件概率。这一条件概率也称词  $i$  和词  $j$  的共现概率。

#### 共现概率比值

GloVe 论文里展示了以下一组词对的共现概率与比值 [1]:

- $\mathbb{P}(k | \text{ice})$ : 0.00019 ( $k = \text{solid}$ ), 0.000066 ( $k = \text{gas}$ ), 0.003 ( $k = \text{water}$ ), 0.000017 ( $k = \text{fashion}$ )
- $\mathbb{P}(k | \text{steam})$ : 0.000022 ( $k = \text{solid}$ ), 0.00078 ( $k = \text{gas}$ ), 0.0022 ( $k = \text{water}$ ), 0.000018 ( $k = \text{fashion}$ )
- $\mathbb{P}(k | \text{ice}) / \mathbb{P}(k | \text{steam})$ : 8.9 ( $k = \text{solid}$ ), 0.085 ( $k = \text{gas}$ ), 1.36 ( $k = \text{water}$ ), 0.96 ( $k = \text{fashion}$ )

我们可以观察到以下现象：

- 对于与 ice (冰) 相关而与 steam (蒸汽) 不相关的词  $k$ , 例如  $k = \text{solid}$  (固体), 我们期望共现概率比值  $P_{ik}/P_{jk}$  较大, 例如上面最后一行结果中的 8.9。
- 对于与 ice 不相关而与 steam 相关的词  $k$ , 例如  $k = \text{gas}$  (气体), 我们期望共现概率比值  $P_{ik}/P_{jk}$  较小, 例如上面最后一行结果中的 0.085。
- 对于与 ice 和 steam 都相关的词  $k$ , 例如  $k = \text{water}$  (水), 我们期望共现概率比值  $P_{ik}/P_{jk}$  接近 1, 例如上面最后一行结果中的 1.36。
- 对于与 ice 和 steam 都不相关的词  $k$ , 例如  $k = \text{fashion}$  (时尚), 我们期望共现概率比值  $P_{ik}/P_{jk}$  接近 1, 例如上面最后一行结果中的 0.96。

由此可见, 共现概率比值能比较直观地表达词与词之间的关系。GloVe 试图用有关词向量的函数来表达共现概率比值。

## 用词向量表达共现概率比值

GloVe 的核心思想在于使用词向量表达共现概率比值。而任意一个这样的比值需要三个词  $i$ 、 $j$  和  $k$  的词向量。对于共现概率  $P_{ij} = \mathbb{P}(j | i)$ ，我们称词  $i$  和词  $j$  分别为中心词和背景词。我们使用  $\mathbf{v}_i$  和  $\tilde{\mathbf{v}}_i$  分别表示词  $i$  作为中心词和背景词的词向量。

我们可以用有关词向量的函数  $f$  来表达共现概率比值：

$$f(\mathbf{v}_i, \mathbf{v}_j, \tilde{\mathbf{v}}_k) = \frac{P_{ik}}{P_{jk}}.$$

需要注意的是，函数  $f$  可能的设计并不唯一。下面我们考虑一种较为合理可能性。

首先，用向量之差来表达共现概率的比值，并将上式改写成

$$f(\mathbf{v}_i - \mathbf{v}_j, \tilde{\mathbf{v}}_k) = \frac{P_{ik}}{P_{jk}}.$$

由于共现概率比值是一个标量，我们可以使用向量之间的内积把函数  $f$  的自变量进一步改写，得到

$$f((\mathbf{v}_i - \mathbf{v}_j)^\top \tilde{\mathbf{v}}_k) = \frac{P_{ik}}{P_{jk}}.$$

由于任意一对词共现的对称性，我们希望以下两个性质可以同时被满足：

- 任意词作为中心词和背景词的词向量应该相等：对任意词  $i$ ， $\mathbf{v}_i = \tilde{\mathbf{v}}_i$ 。
- 词与词之间共现词频矩阵  $\mathbf{X}$  应该对称：对任意词  $i$  和  $j$ ， $x_{ij} = x_{ji}$ 。

为了满足以上两个性质，一方面，我们令

$$f((\mathbf{v}_i - \mathbf{v}_j)^\top \tilde{\mathbf{v}}_k) = \frac{f(\mathbf{v}_i^\top \tilde{\mathbf{v}}_k)}{f(\mathbf{v}_j^\top \tilde{\mathbf{v}}_k)},$$

并得到  $f(x) = \exp(x)$ 。以上两式的右边联立，

$$\exp(\mathbf{v}_i^\top \tilde{\mathbf{v}}_k) = P_{ik} = \frac{x_{ik}}{x_i}.$$

由上式可得

$$\mathbf{v}_i^\top \tilde{\mathbf{v}}_k = \log(x_{ik}) - \log(x_i).$$

另一方面，我们可以把上式中  $\log(x_i)$  替换成两个偏差项之和  $b_i + b_k$ ，得到

$$\mathbf{v}_i^\top \tilde{\mathbf{v}}_k = \log(x_{ik}) - b_i - b_k.$$

将  $i$  和  $k$  互换，我们可验证一对词共现的两个对称性质可以同时被上式满足。因此，对于任意一对词  $i$  和  $j$ ，我们可以用它们词向量表达共现词频的对数：

$$\mathbf{v}_i^\top \tilde{\mathbf{v}}_j + b_i + b_j = \log(x_{ij}).$$

## 损失函数

GloVe 中的共现词频是直接在训练数据上统计得到的。为了学习词向量和相应的偏差项，我们希望上式中的左边与右边尽可能接近。给定词典  $\mathcal{V}$  和权重函数  $h(x_{ij})$ ，GloVe 的损失函数为

$$\sum_{i \in \mathcal{V}, j \in \mathcal{V}} h(x_{ij}) (\mathbf{v}_i^\top \tilde{\mathbf{v}}_j + b_i + b_j - \log(x_{ij}))^2,$$

其中权重函数  $h(x)$  的一个建议选择是，当  $x < c$ （例如  $c = 100$ ），令  $h(x) = (x/c)^\alpha$ （例如  $\alpha = 0.75$ ），反之令  $h(x) = 1$ 。由于权重函数的存在，损失函数的计算复杂度与共现词频矩阵  $\mathbf{X}$  中非零元素的数目呈正相关。我们可以从  $\mathbf{X}$  中随机采样小批量非零元素，并使用优化算法迭代共现词频相关词的向量和偏差项。

需要注意的是，对于任意一对  $i, j$ ，损失函数中存在以下两项之和

$$h(x_{ij})(\mathbf{v}_i^\top \tilde{\mathbf{v}}_j + b_i + b_j - \log(x_{ij}))^2 + h(x_{ji})(\mathbf{v}_j^\top \tilde{\mathbf{v}}_i + b_j + b_i - \log(x_{ji}))^2.$$

由于上式中  $x_{ij} = x_{ji}$ ，对调  $\mathbf{v}$  和  $\tilde{\mathbf{v}}$  并不改变损失函数中这两项之和。也就是说，在损失函数所有项上对调  $\mathbf{v}$  和  $\tilde{\mathbf{v}}$  也不改变整个损失函数的值。因此，任意词的中心词向量和背景词向量是等价的。只是由于初始化值的不同，同一个词最终学习到的两组词向量可能不同。当学习得到所有词向量以后，GloVe 使用一个词的中心词向量与背景词向量之和作为该词的最终词向量。

### 10.2.2 fastText

我们在上一节介绍了 word2vec 的跳字模型和负采样。fastText 以跳字模型为基础，将每个中心词视为子词（subword）的集合，并使用负采样学习子词的词向量。

举个例子，设子词长度为 3 个字符，“where”的子词包括“<wh”、“whe”、“her”、“ere”、“re”和特殊子词（整词）“<where>”。这些子词中的“<”和“>”符号是为了将作为前后缀的子词区分出来。并且，这里的子词“her”与整词“<her>”也可被区分开。给定一个词  $w$ ，我们通常可以把字符长度在 3 到 6 之间的所有子词和特殊子词的并集  $\mathcal{G}_w$  取出。假设词典中任意子词  $g$  的子词向量为  $\mathbf{z}_g$ ，我们可以把使用负采样的跳字模型的损失函数

$$-\log \mathbb{P}(w_o | w_c) = -\log \frac{1}{1 + \exp(-\mathbf{u}_o^\top \mathbf{v}_c)} - \sum_{k=1, w_k \sim \mathbb{P}(w)}^K \log \frac{1}{1 + \exp(\mathbf{u}_{i_k}^\top \mathbf{v}_c)}$$

直接替换成

$$-\log \mathbb{P}(w_o | w_c) = -\log \frac{1}{1 + \exp(-\mathbf{u}_o^\top \sum_{g \in \mathcal{G}_{w_c}} \mathbf{z}_g)} - \sum_{k=1, w_k \sim \mathbb{P}(w)}^K \log \frac{1}{1 + \exp(\mathbf{u}_{i_k}^\top \sum_{g \in \mathcal{G}_{w_c}} \mathbf{z}_g)}.$$

可以看到，原中心词向量被替换了中心词的子词向量之和。与 word2vec 和 GloVe 不同，词典以外的新词的词向量可以使用 fastText 中相应的子词向量之和。

fastText 对于一些语言较重要，例如阿拉伯语、德语和俄语。例如，德语中有很多复合词，例如乒乓球（英文 table tennis）在德语中叫“Tischtennis”。fastText 可以通过子词表达两个词的相关性，例如“Tischtennis”和“Tennis”。

由于词是自然语言的基本表义单元，词向量广泛应用于各类自然语言处理的场景中。例如，我们可以在之前介绍的语言模型中使用在更大规模语料上预训练的词向量。我们将在接下来两节的实验中应用这些预训练的词向量，例如求近似词和类比词，以及文本分类。

### 10.2.3 小结

- GloVe 用词向量表达共现词频的对数。
- fastText 用子词向量之和表达整词。它能表示词典以外的新词。

### 10.2.4 练习

- GloVe 中，如果一个词出现在另一个词的背景中，是否可以利用它们之间在文本序列的距离重新设计词频计算方式？提示：可参考 Glove 论文 4.2 节 [1]。
- 如果丢弃 GloVe 中的偏差项，是否也可以满足任意一对词共现的对称性？
- 在 fastText 中，子词过多怎么办（例如，6 字英文组合数为  $26^6$ ）？提示：可参考 fastText 论文 3.2 节 [2]。

## 10.2.5 扫码直达讨论区



## 10.2.6 参考文献

- [1] Pennington, Jeffrey, Richard Socher, and Christopher Manning. “Glove: Global vectors for word representation.” EMNLP. 2014.
- [2] Bojanowski, Piotr, et al. “Enriching word vectors with subword information.” TACL vol. 5 (2017): 135–146.

## 10.3 求近似词和类比词

本节介绍如何应用在大规模语料上预训练的词向量，例如求近似词和类比词。这里使用的预训练的 GloVe 和 fastText 词向量分别来自它们的项目网站 [1,2]。

首先导入实验所需的包或模块。

```
In [1]: from mxnet import nd  
       from mxnet.contrib import text  
       from mxnet.gluon import nn
```

### 10.3.1 由数据集建立词典和载入词向量

下面，我们以 fastText 为例，由数据集建立词典并载入词向量。fastText 提供了基于不同语言的多套预训练的词向量。这些词向量是在大规模语料上训练得到的，例如维基百科语料。以下打印了其中的 10 套。

```
In [2]: print(text.embedding.get_pretrained_file_names('fasttext')[:10])  
['crawl-300d-2M.vec', 'wiki_aa.vec', 'wiki_ab.vec', 'wiki_ace.vec', 'wiki_ady.vec',  
 → 'wiki_af.vec', 'wiki_ak.vec', 'wiki_als.vec', 'wiki_am.vec', 'wiki_ang.vec']
```

## 访问词向量

为了演示方便，我们创建一个很小的文本数据集，并计算词频。

```
In [3]: text_data = " hello world \n hello nice world \n hi world \n"
    counter = text.utils.count_tokens_from_str(text_data)
```

我们先根据数据集建立词典，并为该词典中的词载入 fastText 词向量。这里使用 Simple English 的预训练词向量。

```
In [4]: my_vocab = text.vocab.Vocabulary(counter)
    my_embedding = text.embedding.create(
        'fasttext', pretrained_file_name='wiki.simple.vec', vocabulary=my_vocab)

/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/mxnet/contr_
↳ ib/text/embedding.py:278: UserWarning: At line 1 of the pre-trained text
↳ embedding file: token 111051 with 1-dimensional vector [300.0] is likely a header
↳ and is skipped.
'skipped.' % (line_num, token, elems))
```

词典除了包括数据集中四个不同的词语，还包括一个特殊的未知词符号。打印词典大小。

```
In [5]: len(my_embedding)
```

```
Out[5]: 5
```

默认情况下，任意一个词典以外词的词向量为零向量。

```
In [6]: my_embedding.get_vecs_by_tokens('beautiful')[:10]
```

```
Out[6]:
```

```
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
<NDArray 10 @cpu(0)>
```

fastText 中每个词均使用 300 维的词向量。打印数据集中两个词“hello”和“world”词向量的形状。

```
In [7]: my_embedding.get_vecs_by_tokens(['hello', 'world']).shape
```

```
Out[7]: (2, 300)
```

打印“hello”和“world”词向量前五个元素。

```
In [8]: my_embedding.get_vecs_by_tokens(['hello', 'world'])[:, :5]
```

```
Out[8]:
```

```
[[ 0.39567     0.21454     -0.035389    -0.24299     -0.095645   ]
 [ 0.10444     -0.10858      0.27212      0.13299     -0.33164999]]
<NDArray 2x5 @cpu(0)>
```

打印“hello”和“world”在词典中的索引。

```
In [9]: my_embedding.to_indices(['hello', 'world'])
```

```
Out[9]: [2, 1]
```

## 使用预训练词向量初始化 Embedding 实例

我们在“循环神经网络——使用 Gluon”一节中介绍了 Gluon 中的 Embedding 实例，并对其中每个词的向量做了随机初始化。实际上，我们可以使用预训练的词向量初始化 Embedding 实例。

```
In [10]: layer = nn.Embedding(len(my_embedding), my_embedding.vec_len)
layer.initialize()
layer.weight.set_data(my_embedding.idx_to_vec)
```

使用词典中“hello”和“world”两个词在词典中的索引，我们可以通过 Embedding 实例得到它们的预训练词向量，并向神经网络的下一层传递。

```
In [11]: layer(nd.array([2, 1]))[:, :5]
```

```
Out[11]:
[[ 0.39567    0.21454   -0.035389   -0.24299   -0.095645  ]
 [ 0.10444   -0.10858    0.27212    0.13299   -0.33164999]]
<NDArray 2x5 @cpu(0)>
```

### 10.3.2 由预训练词向量建立词典——以 GloVe 为例

除了使用数据集建立词典外，我们还可以直接由预训练词向量建立词典。

这一次我们使用 GloVe 的预训练词向量。以下打印了 GloVe 提供的各套预训练词向量。这些词向量是在大规模语料上训练得到的，例如维基百科语料和推特语料。

```
In [12]: print(text.embedding.get_pretrained_file_names('glove'))
['glove.42B.300d.txt', 'glove.6B.50d.txt', 'glove.6B.100d.txt', 'glove.6B.200d.txt',
 ↵ 'glove.6B.300d.txt', 'glove.840B.300d.txt', 'glove.twitter.27B.25d.txt',
 ↵ 'glove.twitter.27B.50d.txt', 'glove.twitter.27B.100d.txt',
 ↵ 'glove.twitter.27B.200d.txt']
```

我们使用 50 维的词向量。和之前不同，这里不再传入根据数据集建立的词典，而是直接使用预训练词向量中的词建立词典。

```
In [13]: glove_6b50d = text.embedding.create('glove',
                                             pretrained_file_name='glove.6B.50d.txt')
```

打印词典大小。注意其中包含一个特殊的未知词符号。

```
In [14]: print(len(glove_6b50d))
```

```
400001
```

我们可以访问词向量的属性。

```
In [15]: # 词到索引，索引到词。
```

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

```
Out[15]: (3367, 'beautiful')
```

### 10.3.3 应用预训练词向量

下面我们以 GloVe 为例，展示预训练词向量的应用。

首先，我们定义余弦相似度，并用它表示两个向量之间的相似度。

```
In [16]: def cos_sim(x, y):
    return nd.dot(x, y) / (x.norm() * y.norm())
```

余弦相似度的值域在-1 到 1 之间。两个余弦相似度越大的向量越相似。

```
In [17]: x = nd.array([1, 2])
y = nd.array([10, 20])
z = nd.array([-1, -2])
cos_sim(x, y), cos_sim(x, z)
```

```
Out[17]: (
    [ 1.]
    <NDArray 1 @cpu(0)>,
    [-1.]
    <NDArray 1 @cpu(0)>)
```

### 求近似词

给定任意词，我们可以从 GloVe 的整个词典（大小 40 万，不含未知词符号）中找出与它最接近的  $k$  个词。之前已经提到，词与词之间的相似度可以用两个词向量的余弦相似度表示。

```
In [18]: def norm_vecs_by_row(x):
    return x / nd.sum(x * x, axis=1).sqrt().reshape((-1, 1))

def get_knn(token_embedding, k, word):
    word_vec = token_embedding.get_vecs_by_tokens([word]).reshape((-1, 1))
```

```

vocab_vecs = norm_vecs_by_row(token_embedding.idx_to_vec)
dot_prod = nd.dot(vocab_vecs, word_vec)
indices = nd.topk(dot_prod.reshape((len(token_embedding), )), k=k+2,
                  ret_typ='indices')
indices = [int(i.asscalar()) for i in indices]
# 除去未知词符号和输入词。
return token_embedding.to_tokens(indices[2:])

```

查找词典中与“baby”最近似的5个词。

```

In [19]: get_knn(glove_6b50d, 5, 'baby')

Out[19]: ['babies', 'boy', 'girl', 'newborn', 'pregnant']

```

验证一下“baby”和“babies”两个词向量之间的余弦相似度。

```

In [20]: cos_sim(glove_6b50d.get_vecs_by_tokens('baby'),
                 glove_6b50d.get_vecs_by_tokens('babies'))

Out[20]:
[ 0.83871299]
<NDArray 1 @cpu(0)>

```

查找词典中与“computers”最近似的5个词。

```

In [21]: get_knn(glove_6b50d, 5, 'computers')

Out[21]: ['computer', 'phones', 'pcs', 'machines', 'devices']

```

查找词典中与“run”最近似的5个词。

```

In [22]: get_knn(glove_6b50d, 5, 'run')

Out[22]: ['running', 'runs', 'went', 'start', 'ran']

```

查找词典中与“beautiful”最近似的5个词。

```

In [23]: get_knn(glove_6b50d, 5, 'beautiful')

Out[23]: ['lovely', 'gorgeous', 'wonderful', 'charming', 'beauty']

```

## 求类比词

除近似词以外，我们还可以使用预训练词向量求词与词之间的类比关系。例如， $\text{man} : \text{woman} :: \text{son} : \text{daughter}$  是一个类比例子：“man”之于“woman”相当于“son”之于“daughter”。求类比词问题可以定义为：对于类比关系中的四个词  $a : b :: c : d$ ，给定前三个词  $a$ 、 $b$  和  $c$ ，求  $d$ 。设词  $w$  的词向量为  $\text{vec}(w)$ 。而解类比词的思路是，找到和  $\text{vec}(c) + \text{vec}(b) - \text{vec}(a)$  的结果向量最相似的词向量。

本例中，我们将从整个词典（大小 40 万，不含未知词符号）中搜索类比词。

```
In [24]: def get_top_k_by_analogy(token_embedding, k, word1, word2, word3):
    word_vecs = token_embedding.get_vecs_by_tokens([word1, word2, word3])
    word_diff = (word_vecs[1] - word_vecs[0] + word_vecs[2]).reshape((-1, 1))
    vocab_vecs = norm_vecs_by_row(token_embedding.idx_to_vec)
    dot_prod = nd.dot(vocab_vecs, word_diff)
    indices = nd.topk(dot_prod.reshape((len(token_embedding), )), ), k=k+1,
               ret_typ='indices')
    indices = [int(i.asscalar()) for i in indices]

    # 不考虑未知词为可能的类比词。
    if token_embedding.to_tokens(indices[0]) == token_embedding.unknown_token:
        return token_embedding.to_tokens(indices[1:])
    else:
        return token_embedding.to_tokens(indices[:-1])
```

“男-女”类比：“man”之于“woman”相当于“son”之于什么？

```
In [25]: get_top_k_by_analogy(glove_6b50d, 1, 'man', 'woman', 'son')
```

```
Out[25]: ['daughter']
```

验证一下  $\text{vec}(\text{son}) + \text{vec}(\text{woman}) - \text{vec}(\text{man})$  与  $\text{vec}(\text{daughter})$  两个向量之间的余弦相似度。

```
In [26]: def cos_sim_word_analogy(token_embedding, word1, word2, word3, word4):
    words = [word1, word2, word3, word4]
    vecs = token_embedding.get_vecs_by_tokens(words)
    return cos_sim(vecs[1] - vecs[0] + vecs[2], vecs[3])
```

```
cos_sim_word_analogy(glove_6b50d, 'man', 'woman', 'son', 'daughter')
```

```
Out[26]:
```

```
[ 0.96583432]
<NDArray 1 @cpu(0)>
```

“首都-国家”类比：“beijing”之于“china”相当于“tokyo”之于什么？

```
In [27]: get_top_k_by_analogy(glove_6b50d, 1, 'beijing', 'china', 'tokyo')
```

```
Out[27]: ['japan']
```

“形容词-形容词最高级”类比：“bad”之于“worst”相当于“big”之于什么？

```
In [28]: get_top_k_by_analogy(glove_6b50d, 1, 'bad', 'worst', 'big')
```

```
Out[28]: ['biggest']
```

“动词一般时-动词过去时”类比：“do”之于“did”相当于“go”之于什么？

```
In [29]: get_top_k_by_analogy(glove_6b50d, 1, 'do', 'did', 'go')  
Out[29]: ['went']
```

#### 10.3.4 小结

- 我们可以应用预训练的词向量求近似词和类比词。

#### 10.3.5 练习

- 将近似词和类比词应用中的  $k$  调大一些，观察结果。
- 测试一下 fastText 的中文词向量 (`pretrained_file_name='wiki.zh.vec'`)。
- 如果在“循环神经网络——使用 Gluon”一节中将 Embedding 实例里的参数初始化为预训练的词向量，效果如何？

#### 10.3.6 扫码直达讨论区



#### 10.3.7 参考文献

- [1] GloVe 项目网站. <https://nlp.stanford.edu/projects/glove/>
- [2] fastText 项目网站. <https://fasttext.cc/>

### 10.4 文本分类：情感分析

情感分析是非常重要的一项自然语言处理的任务。例如对于 Amazon 会对网站所销售的每个产品的评论进行情感分类，Netflix 或者 IMDb 会对每部电影的评论进行情感分类，从而帮助各个平

台改进产品，提升用户体验。本节介绍如何使用 Gluon 来创建一个情感分类模型，目标是给定一句话，判断这句话包含的是“正面”还是“负面”的情绪。为此，我们构造了一个简单的神经网络，其中包括 embedding 层，encoder（双向 LSTM），decoder，来判断 IMDb 上电影评论蕴含的情感。下面就让我们一起来构造这个情感分析模型吧。

### 10.4.1 准备工作

在开始构造情感分析模型之前，我们需要进行下面的一些准备工作。

#### 加载 MXNet 和 Gluon

首先，我们当然需要加载 MXNet 和 Gluon。

```
In [1]: import sys
        sys.path.append('..')
        import collections
        import gluonbook as gb
        import mxnet as mx
        from mxnet import autograd, gluon, init, metric, nd
        from mxnet.gluon import loss as gloss, nn, rnn
        from mxnet.contrib import text
        import os
        import random
        import zipfile
```

#### 读取 IMDb 数据集

接着需要下载情感分析时需要用的数据集。我们使用 Stanford's Large Movie Review Dataset[1] 作为数据集。

- 下载地址：[http://ai.stanford.edu/~amaas/data/sentiment/aclImdb\\_v1.tar.gz](http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz)。

这个数据集分为训练（train）和测试（test）数据集，分别都有 25,000 条从 IMDb 下载的关于电影的评论，其中 12,500 条被标注成“正面”的评论，另外 12,500 条被标注成“负面”的评论。我们使用 1 表示‘正面’评论，0 表示‘负面’评论。

下载好之后，将数据解压，放在教程的‘`../data/`’文件夹之下，最后文件位置如下：

```
'../data/aclImdb'
```

注意，如果读者已经下载了上述数据集，请略过此步，请设置 `demo = False`。

否则，如果读者想快速上手，并没有下载上述数据集，我们提供了上述数据集的一个小的采样，`../data/aclImdb_tiny.zip`，运行下面的代码解压这个小数据集。

```
In [2]: # 如果训练下载的 IMDB 的完整数据集，把下面改 False。
demo = True
if demo:
    with zipfile.ZipFile('../data/aclImdb_tiny.zip', 'r') as zin:
        zin.extractall('../data/')

In [3]: def readIMDB(dir_url, seg='train'):
    pos_or_neg = ['pos', 'neg']
    data = []
    for label in pos_or_neg:
        files = os.listdir('../data/' + dir_url + '/' + seg + '/' + label +
                           '/')
        for file in files:
            with open('../data/' + dir_url + '/' + seg + '/' + label + '/' +
                      file, 'r', encoding='utf8') as rf:
                review = rf.read().replace('\n', '')
            if label == 'pos':
                data.append([review, 1])
            elif label == 'neg':
                data.append([review, 0])
    return data

if demo:
    train_data = readIMDB('aclImdb_tiny/', 'train')
    test_data = readIMDB('aclImdb_tiny/', 'test')
else:
    train_data = readIMDB('aclImdb/', 'train')
    test_data = readIMDB('aclImdb/', 'test')

random.shuffle(train_data)
random.shuffle(test_data)
```

## 指定分词方式并且分词

接下来我们对每条评论分词，得到分好词的评论。我们使用最简单的基于空格进行分词（更好的分词工具我们留作练习）。运行下面的代码进行分词。

```
In [4]: def tokenizer(text):
    return [tok.lower() for tok in text.split(' ')]
```

通过执行下面的代码，我们能够获得训练和测试数据集的分好词的评论，并且得到相应的情感标签（1代表‘正面’，0代表‘负面’情绪）。

```
In [5]: train_tokenized = []
for review, score in train_data:
    train_tokenized.append(tokenizer(review))
test_tokenized = []
for review, score in test_data:
    test_tokenized.append(tokenizer(review))
```

## 创建词典

现在，先根据分好词的训练数据创建 counter，然后使用 mxnet.contrib 中的 vocab 创建词典。这里我们特别设置训练数据中没有的单词对应的符号‘’，所有不存在于词典中的词，未来都将对应到这个符号。

```
In [6]: token_counter = collections.Counter()
def count_token(train_tokenized):
    for sample in train_tokenized:
        for token in sample:
            if token not in token_counter:
                token_counter[token] = 1
            else:
                token_counter[token] += 1

count_token(train_tokenized)
vocab = text.vocab.Vocabulary(token_counter, unknown_token='<unk>',
                               reserved_tokens=None)
```

## 将分好词的数据转化成 NDArray

这小节我们介绍如果将数据转化成为 NDArray。

```
In [7]: # 根据词典，将数据转换成特征向量。
def encode_samples(tokenized_samples, vocab):
    features = []
    for sample in tokenized_samples:
        feature = []
        for token in sample:
            if token in vocab.token_to_idx:
                feature.append(vocab.token_to_idx[token])
            else:
```

```

        feature.append(0)
    features.append(feature)
    return features

def pad_samples(features, maxlen=500, padding=0):
    padded_features = []
    for feature in features:
        if len(feature) > maxlen:
            padded_feature = feature[:maxlen]
        else:
            padded_feature = feature
        # 添加 PAD 符号使每个序列等长 (长度为 maxlen )。
        while len(padded_feature) < maxlen:
            padded_feature.append(padding)
        padded_features.append(padded_feature)
    return padded_features

```

运行下面的代码将分好词的训练和测试数据转化成特征向量。

```
In [8]: train_features = encode_samples(train_tokenized, vocab)
        test_features = encode_samples(test_tokenized, vocab)
```

通过执行下面的代码将特征向量补成定长（我们使用 500），然后将特征向量转化为指定 context 上的 NDArray。这里我们假定我们有至少一块 gpu，context 被设置成 gpu。当然，也可以使用 cpu，运行速度可能稍微慢一点点。

```
In [9]: ctx = gb.try_gpu()
        train_features = nd.array(pad_samples(train_features, 500, 0), ctx=ctx)
        test_features = nd.array(pad_samples(test_features, 500, 0), ctx=ctx)
```

这里，我们将情感标签也转化成为了 NDArray。

```
In [10]: train_labels = nd.array([score for _, score in train_data], ctx=ctx)
        test_labels = nd.array([score for _, score in test_data], ctx=ctx)
```

## 加载预训练的词向量

这里我们使用之前创建的词典 vocab 以及 GloVe 词向量创建词典中每个词所对应的词向量。词向量将在后续的模型中作为每个词的初始权重加入模型，这样做有助于提升模型的结果。我们在里面使用‘glove.6B.100d.txt’作为预训练的词向量。

```
In [11]: glove_embedding = text.embedding.create(
        'glove', pretrained_file_name='glove.6B.100d.txt', vocabulary=vocab)
```

## 10.4.2 创建情感分析模型

情感分类模型是一种比较经典的能使用 LSTM 模型的应用。我们特别的使用预训练的词向量来初始化 `embedding layer` 的权重，然后使用双向 LSTM 抽取特征。具体地，输入的是一个句子即不定长的序列，然后通过 `embedding layer`，利用预训练的词向量表示句子，通过 LSTM 抽取句子的特征，然后输出是一个长度为 1 的标签。根据上述原理，我们设计如下神经网络结构，其结构比较简单，如下图所示。

模型包含四部分：1. `embedding layer`: 其将输入数据转化成为 TNC 的 NDArray，并且使用预先加载词向量作为该层的权重。2. `encoder`: 我们将重点介绍这一部分。`decoder` 是由一个两层的双向 LSTM 构成。这样做的好处是，我们能够利用 LSTM 的输出作为输入样本的特征，之后用于预测。3. `pooling layer`: 我们使用这个 `encoder` 在时刻 0 的输出，以及时刻最后一步的输出作为每个 batch 中 examples 的特征。4. `decoder`: 最后，我们利用上一步所生成的特征，通过一个 `dense` 层做预测。

```
In [12]: num_outputs = 2
        lr = 0.1
        num_epochs = 1
        batch_size = 10
        embed_size = 100
        num_hiddens = 100
        num_layers = 2
        bidirectional = True

In [13]: class SentimentNet(nn.Block):
            def __init__(self, vocab, embed_size, num_hiddens, num_layers,
                         bidirectional, **kwargs):
                super(SentimentNet, self).__init__(**kwargs)
                with self.name_scope():
                    self.embedding = nn.Embedding(
                        len(vocab), embed_size, weight_initializer=init.Uniform(0.1))
                    self.encoder = rnn.LSTM(num_hiddens, num_layers=num_layers,
                                          bidirectional=bidirectional,
                                          input_size=embed_size)
                    self.decoder = nn.Dense(num_outputs, flatten=False)
            def forward(self, inputs, begin_state=None):
                outputs = self.embedding(inputs)
                outputs = self.encoder(outputs)
                outputs = nd.concat(outputs[0], outputs[-1])
                outputs = self.decoder(outputs)
            return outputs
```

```

net = SentimentNet(vocab, embed_size, num_hiddens, num_layers, bidirectional)
net.initialize(init.Xavier(), ctx=ctx)
# 设置 embedding 层的 weight 为词向量。
net.embedding.weight.set_data(glove_embedding.idx_to_vec.as_in_context(ctx))
# 对 embedding 层不进行优化。
net.embedding.collect_params().setattr('grad_req', 'null')
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
loss = gloss.SoftmaxCrossEntropyLoss()

```

### 10.4.3 训练模型

这里我们训练模型。我们使用预先设置好的迭代次数和 batch\_size 训练模型。可以看到, Gluon 能极大的简化训练的代码量, 使得训练过程看起来非常简洁。另外, 我们使用交叉熵作为损失函数, 使用准确率来评价模型。

In [14]: # 使用准确率作为评价指标。

```

def eval_model(features, labels):
    l_sum = 0
    l_n = 0
    accuracy = metric.Accuracy()
    for i in range(features.shape[0] // batch_size):
        X = features[i*batch_size : (i+1)*batch_size].as_in_context(ctx).T
        y = labels[i*batch_size : (i+1)*batch_size].as_in_context(ctx).T
        output = net(X)
        l = loss(output, y)
        l_sum += l.sum().asscalar()
        l_n += l.size
        accuracy.update(preds=nd.argmax(output, axis=1), labels=y)
    return l_sum / l_n, accuracy.get()[1]

```

运行下面的代码开始训练模型。

In [15]:

```

for epoch in range(1, num_epochs + 1):
    for i in range(train_features.shape[0] // batch_size):
        X = train_features[i*batch_size : (i+1)*batch_size].as_in_context(
            ctx).T
        y = train_labels[i*batch_size : (i+1)*batch_size].as_in_context(
            ctx).T
        with autograd.record():
            l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
    train_loss, train_acc = eval_model(train_features, train_labels)

```

```
test_loss, test_acc = eval_model(test_features, test_labels)
print('epoch %d, train loss %.6f, acc %.2f; test loss %.6f, acc %.2f'
      % (epoch, train_loss, train_acc, test_loss, test_acc))

epoch 1, train loss 0.693590, acc 0.40; test loss 0.691725, acc 0.55
```

到这里，我们已经成功使用 Gluon 创建了一个情感分类模型。下面我们举了一个例子，来看看我们情感分类模型的效果（1 代表正面，0 代表负面）。如果希望在不同句子上得到较准确的分类，我们需要使用更大的数据集，并增大模型训练周期。

```
In [16]: review = ['this', 'movie', 'is', 'great']
nd.argmax(net(nd.reshape(
    nd.array([vocab.token_to_idx[token] for token in review], ctx=ctx),
    shape=(-1, 1))), axis=1).asscalar()

Out[16]: 1.0
```

#### 10.4.4 小结

这节，我们使用了之前学到的预训练的词向量以及双向 LSTM 来构建情感分类模型，通过使用 Gluon，我们可以很简单的就构造出一个还不错的情感模型。

#### 10.4.5 练习

大家可以尝试下面几个方向来得到更好的情感分类模型：

- 想要提高最后的准确率，有一个小方法，就是把迭代次数 (num\_epochs) 改成 3。最后在训练和测试数据上准确率大概能达到 0.82。
- 可以尝试使用更好的分词工具得到更好的分词效果，会对最终结果有帮助。例如可以使⽤ spacy 分词⼯具，先 pip 安装 spacy: pip install spacy，并且安装 spacy 的英⽂包: python -m spacy download en，然后运⾏下面的代码分词，先载⼊ spacy: import spacy，接着加载 spacy 英⽂包: spacy\_en = spacy.load('en')，最后定义基于 spacy 的分词函数: def tokenizer(text): return [tok.text for tok in spacy\_en.tokenizer(text)] 替换原来的基于空格的分词⼯具。注意，GloVe 的向量对于名词词组的存储⽅式是用'-'连接独立单词，例如'new york'为'new-york'。而使⽤ spacy 分词之后'new york'的存储可能是'new york'。所以为了得到更好的 embedding 效果，可以对于词组进⾏简单的后续处理。使⽤ spacy 作为分词⼯具，能使准确率上升到 0.85 以上。

- 使用更大的预训练词向量，例如 300 维的 GloVe 向量。
- 使用更加深层的 encoder，即使用更多数量的 layer。
- 使用更加有意思的 decoder，例如可以加上 LSTM，之后再加上 dense layer。

#### 10.4.6 扫码直达讨论区



#### 10.4.7 参考文献

[1] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). Learning Word Vectors for Sentiment Analysis. The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011).

### 10.5 编码器—解码器（seq2seq）

在很多应用中，输入和输出都可以是不定长序列。以机器翻译为例，输入是可以是一段不定长的英语文本序列，输出可以是一段不定长的法语文本序列，例如

英语：“They”、“are”、“watching”、“.”

法语：“Ils”、“regardent”、“.”

当输入输出都是不定长序列时，我们可以使用编码器—解码器(encoder-decoder)[1] 或者 seq2seq 模型[2]。这两个模型本质上都用到了两个循环神经网络，分别叫做编码器和解码器。编码器对应输入序列，解码器对应输出序列。下面我们来介绍编码器—解码器的设计。

### 10.5.1 编码器

编码器的把一个不定长的输入序列转换成一个定长的背景变量  $c$ ，并在该背景变量中编码输入序列信息。常用的编码器是循环神经网络。

让我们考虑批量大小为 1 的时序数据样本。在时间步  $t$ ，循环神经网络将输入  $x_t$  的特征向量  $\mathbf{x}_t$  和上个时间步的隐藏状态  $\mathbf{h}_{t-1}$  变换为当前时间步的隐藏状态  $\mathbf{h}_t$ 。其中，每个输入的特征向量可能是模型参数，例如“[循环神经网络——使用 Gluon](#)”一节中需要学习的每个词向量。我们可以用函数  $f$  表达循环神经网络隐藏层的变换：

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

假设输入序列的时间步数为  $T$ 。编码器通过自定义函数  $q$  将各个时间步的隐藏状态变换为背景变量

$$\mathbf{c} = q(\mathbf{h}_1, \dots, \mathbf{h}_T).$$

例如，当选择  $q(\mathbf{h}_1, \dots, \mathbf{h}_T) = \mathbf{h}_T$  时，背景变量是输入序列最终时间步的隐藏状态  $\mathbf{h}_T$ 。我们将这里的循环神经网络叫做编码器。

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

### 10.5.2 解码器

刚刚已经介绍，假设输入序列的时间步数为  $T$ ，编码器输出的背景变量  $\mathbf{c}$  编码了整个输入序列  $x_1, \dots, x_T$  的信息。给定训练样本中的输出序列  $y_1, y_2, \dots, y_{T'}$ 。假设其中每个时间步  $t'$  的输出同时取决于该时间步之前的输出序列和背景变量。那么，根据最大似然估计，我们可以最大化输出序列基于输入序列的条件概率

$$\begin{aligned}\mathbb{P}(y_1, \dots, y_{T'} \mid x_1, \dots, x_T) &= \prod_{t'=1}^{T'} \mathbb{P}(y_{t'} \mid y_1, \dots, y_{t'-1}, x_1, \dots, x_T) \\ &= \prod_{t'=1}^{T'} \mathbb{P}(y_{t'} \mid y_1, \dots, y_{t'-1}, \mathbf{c}),\end{aligned}$$

并得到该输出序列的损失

$$-\log \mathbb{P}(y_1, \dots, y_{T'}).$$

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

$$s_{t'} = g(y_{t'-1}, c, s_{t'-1}).$$

有了解码器的隐藏状态后，我们可以自定义输出层来计算损失中的  $\mathbb{P}(y_{t'} | y_1, \dots, y_{t'-1}, c)$ ，例如基于当前时间步的解码器隐藏状态  $s_{t'}$ 、上一时间步的输出  $y_{t'-1}$  以及背景变量  $c$  来计算当前时间步输出  $y_{t'}$  的概率分布。

在实际中，我们常使用深度循环神经网络作为编码器和解码器。我们将在本章稍后的“机器翻译”一节中实现含深度循环神经网络的编码器和解码器。

### 10.5.3 小结

- 编码器-解码器（seq2seq）可以输入并输出不定长的序列。
- 编码器-解码器使用了两个循环神经网络。

### 10.5.4 练习

- 除了机器翻译，你还能想到 seq2seq 的哪些应用？
- 有哪些方法可以设计解码器的输出层？

### 10.5.5 扫码直达讨论区



## 10.5.6 参考文献

- [1] Cho, Kyunghyun, et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation.” arXiv:1406.1078 (2014).
- [2] Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. “Sequence to sequence learning with neural networks.” NIPS. 2014.

## 10.6 注意力机制

在上一节里的解码器设计中，输出序列的各个时间步使用了相同的背景变量。如果解码器的不同时间步可以使用不同的背景变量呢？这样做有什么好处？

### 10.6.1 动机

以英语-法语翻译为例，给定一对输入序列“*They*”、“*are*”、“*watching*”、“*.*”和输出序列“*Ils*”、“*regardent*”、“*.*”。解码器可以在输出序列的时间步1使用更多编码了“*They*”、“*are*”信息的背景变量来生成“*Ils*”，在时间步2使用更多编码了“*watching*”信息的背景变量来生成“*regardent*”，在时间步3使用更多编码了“*.*”信息的背景变量来生成“*.*”。这看上去就像是在解码器的每一时间步对输入序列中不同时间步编码的信息分配不同的注意力。这也是注意力机制的由来。它最早由 Bahdanau 等提出 [1]。

### 10.6.2 设计

本节沿用上一节的符号。

我们对上一节的解码器稍作修改。在时间步  $t'$ ，设解码器的背景变量为  $\mathbf{c}_{t'}$ ，输出  $y_{t'}$  的特征向量为  $\mathbf{y}_{t'}$ 。和上一节输入的特征向量一样，这里每个输出的特征向量也可能是模型参数。解码器在时间步  $t'$  的隐藏状态

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

令编码器在时间步  $t$  的隐藏状态为  $\mathbf{h}_t$ ，且时间步数为  $T$ 。解码器在时间步  $t'$  的背景变量为

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

其中  $\alpha_{t't}$  是权值。也就是说，给定解码器的当前时间步  $t'$ ，我们需要对编码器中不同时间步  $t$  的隐藏状态求加权平均。这里的权值也称注意力权重。它的计算公式是

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

其中  $e_{t't} \in \mathbb{R}$  的计算为

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

上式中的函数  $a$  有多种设计方法。Bahanau 等使用了

$$e_{t't} = \mathbf{v}^\top \tanh(\mathbf{W}_s \mathbf{s}_{t'-1} + \mathbf{W}_h \mathbf{h}_t),$$

其中  $\mathbf{v}$ 、 $\mathbf{W}_s$ 、 $\mathbf{W}_h$  以及编码器与解码器中的各个权重和偏差都是模型参数 [1]。

Bahanau 等在编码器和解码器中分别使用了门控循环单元 [1]。在解码器中，我们需要对门控循环单元的设计稍作修改。解码器在  $t'$  时间步的隐藏状态为

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

其中的重置门、更新门和候选隐含状态分别为

$$\begin{aligned}\mathbf{r}_{t'} &= \sigma(\mathbf{W}_{yr} \mathbf{y}_{t'-1} + \mathbf{W}_{sr} \mathbf{s}_{t'-1} + \mathbf{W}_{cr} \mathbf{c}_{t'} + \mathbf{b}_r), \\ \mathbf{z}_{t'} &= \sigma(\mathbf{W}_{yz} \mathbf{y}_{t'-1} + \mathbf{W}_{sz} \mathbf{s}_{t'-1} + \mathbf{W}_{cz} \mathbf{c}_{t'} + \mathbf{b}_z), \\ \tilde{\mathbf{s}}_{t'} &= \tanh(\mathbf{W}_{ys} \mathbf{y}_{t'-1} + \mathbf{W}_{ss} (\mathbf{s}_{t'-1} \odot \mathbf{r}_{t'}) + \mathbf{W}_{cs} \mathbf{c}_{t'} + \mathbf{b}_s).\end{aligned}$$

我们将在下一节中实现含注意力机制的编码器和解码器。

### 10.6.3 小结

- 我们可以在解码器的每个时间步使用不同的背景变量，并对输入序列中不同时间步编码的信息分配不同的注意力。

### 10.6.4 练习

- 不修改“门控循环单元（GRU）——从零开始”一节中的 `gru_rnn` 函数，应如何用它实现本节介绍的解码器？
- 除了自然语言处理，注意力机制还可以应用在哪些地方？

## 10.6.5 扫码直达讨论区



## 10.6.6 参考文献

[1] Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate.” arXiv:1409.0473 (2014).

# 10.7 机器翻译

本节介绍编码器—解码器和注意力机制的应用。我们以神经机器翻译（neural machine translation）为例，介绍如何使用 Gluon 实现一个简单的编码器—解码器和注意力机制模型。

## 10.7.1 使用 Gluon 实现编码器—解码器和注意力机制

我们先载入需要的包。

```
In [1]: import collections
        import io
        import mxnet as mx
        from mxnet import autograd, gluon, init, nd
        from mxnet.contrib import text
        from mxnet.gluon import data as gdata, loss as gloss, nn, rnn
```

下面定义一些特殊字符。其中 PAD (padding) 符号使每个序列等长；BOS (beginning of sequence) 符号表示序列的开始；而 EOS (end of sequence) 符号表示序列的结束。

```
In [2]: PAD = '<pad>'
        BOS = '<bos>'
        EOS = '<eos>'
```

以下是一些可以调节的模型参数。我们在编码器和解码器中分别使用了一层和两层的循环神经网络。

```
In [3]: epochs = 40
epoch_period = 10
lr = 0.005
batch_size = 2
# 序列最大长度（含句末添加的 EOS 字符）。
max_seq_len = 5
# 测试时输出序列的最大长度（含句末添加的 EOS 字符）。
max_test_output_len = 20
encoder_num_layers = 1
decoder_num_layers = 2
encoder_drop_prob = 0.1
decoder_drop_prob = 0.1
encoder_embed_size = 256
encoder_num_hiddens = 256
decoder_num_hiddens = 256
alignment_size = 25
ctx = mx.cpu(0)
```

## 读取数据

我们定义函数读取训练数据集。为了减少运行时间，我们使用一个很小的法语——英语数据集。

这里使用了[之前章节](#)介绍的 `mxnet.contrib.text` 来创建法语和英语的词典。需要注意的是，我们会在句末附上 EOS 符号，并可能通过添加 PAD 符号使每个序列等长。

我们通常会在输入序列和输出序列后面分别附上一个特殊字符“<eos>”（end of sequence）表示序列的终止。在测试模型时，一旦输出“<eos>”就终止当前的输出序列。

```
In [4]: def read_data(max_seq_len):
    input_tokens = []
    output_tokens = []
    input_seqs = []
    output_seqs = []
    with io.open('../data/fr-en-small.txt') as f:
        lines = f.readlines()
        for line in lines:
            input_seq, output_seq = line.rstrip().split('\t')
            cur_input_tokens = input_seq.split(' ')
            cur_output_tokens = output_seq.split(' ')
            if len(cur_input_tokens) < max_seq_len and \

```

```

        len(cur_output_tokens) < max_seq_len:
    input_tokens.extend(cur_input_tokens)
    # 句末附上 EOS 符号。
    cur_input_tokens.append(EOS)
    # 添加 PAD 符号使每个序列等长 (长度为 max_seq_len )。
    while len(cur_input_tokens) < max_seq_len:
        cur_input_tokens.append(PAD)
    input_seqs.append(cur_input_tokens)
    output_tokens.extend(cur_output_tokens)
    cur_output_tokens.append(EOS)
    while len(cur_output_tokens) < max_seq_len:
        cur_output_tokens.append(PAD)
    output_seqs.append(cur_output_tokens)
fr_vocab = text.vocab.Vocabulary(collections.Counter(input_tokens),
                                  reserved_tokens=[PAD, BOS, EOS])
en_vocab = text.vocab.Vocabulary(collections.Counter(output_tokens),
                                  reserved_tokens=[PAD, BOS, EOS])
return fr_vocab, en_vocab, input_seqs, output_seqs

```

以下创建训练数据集。每一个样本包含法语的输入序列和英语的输出序列。

```

In [5]: input_vocab, output_vocab, input_seqs, output_seqs = read_data(max_seq_len)
fr = nd.zeros((len(input_seqs), max_seq_len), ctx=ctx)
en = nd.zeros((len(output_seqs), max_seq_len), ctx=ctx)
for i in range(len(input_seqs)):
    fr[i] = nd.array(input_vocab.to_indices(input_seqs[i]), ctx=ctx)
    en[i] = nd.array(output_vocab.to_indices(output_seqs[i]), ctx=ctx)
dataset = gdata.ArrayDataset(fr, en)

```

## 编码器、含注意力机制的解码器和解码器初始状态

以下定义了基于GRU的编码器。

```

In [6]: class Encoder(nn.Block):
    """ 编码器。"""
    def __init__(self, num_inputs, embed_size, num_hiddens, num_layers,
                 drop_prob, **kwargs):
        super(Encoder, self).__init__(**kwargs)
        with self.name_scope():
            self.embedding = nn.Embedding(num_inputs, embed_size)
            self.dropout = nn.Dropout(drop_prob)
            self.rnn = rnn.GRU(num_hiddens, num_layers, dropout=drop_prob,
                              input_size=embed_size)

```

```

def forward(self, inputs, state):
    embedding = self.embedding(inputs).swapaxes(0, 1)
    embedding = self.dropout(embedding)
    output, state = self.rnn(embedding, state)
    return output, state

def begin_state(self, *args, **kwargs):
    return self.rnn.begin_state(*args, **kwargs)

```

以下定义了基于GRU的解码器。它包含上一节里介绍的注意力机制的实现。

```

In [7]: class Decoder(nn.Block):
    """ 含注意力机制的解码器。 """
    def __init__(self, num_hiddens, num_outputs, num_layers, max_seq_len,
                 drop_prob, alignment_size, encoder_num_hiddens, **kwargs):
        super(Decoder, self).__init__(**kwargs)
        self.max_seq_len = max_seq_len
        self.encoder_num_hiddens = encoder_num_hiddens
        self.hidden_size = num_hiddens
        self.num_layers = num_layers
        with self.name_scope():
            self.embedding = nn.Embedding(num_outputs, num_hiddens)
            self.dropout = nn.Dropout(drop_prob)
            # 注意力机制。
            self.attention = nn.Sequential()
            with self.attention.name_scope():
                self.attention.add(
                    nn.Dense(alignment_size,
                            in_units=num_hiddens + encoder_num_hiddens,
                            activation="tanh", flatten=False))
            self.attention.add(nn.Dense(1, in_units=alignment_size,
                                       flatten=False))

            self.rnn = rnn.GRU(num_hiddens, num_layers, dropout=drop_prob,
                               input_size=num_hiddens)
            self.out = nn.Dense(num_outputs, in_units=num_hiddens,
                               flatten=False)
            self.rnn_concat_input = nn.Dense(
                num_hiddens, in_units=num_hiddens + encoder_num_hiddens,
                flatten=False)

    def forward(self, cur_input, state, encoder_outputs):
        # 当 RNN 为多层时，取最靠近输出层的单层隐藏状态。
        single_layer_state = [state[0][-1].expand_dims(0)]

```

```

encoder_outputs = encoder_outputs.reshape((self.max_seq_len, -1,
                                         self.encoder_num_hiddens))
hidden_broadcast = nd.broadcast_axis(single_layer_state[0], axis=0,
                                      size=self.max_seq_len)
encoder_outputs_and_hiddens = nd.concat(encoder_outputs,
                                         hidden_broadcast, dim=2)
energy = self.attention(encoder_outputs_and_hiddens)
batch_attention = nd.softmax(energy, axis=0).transpose((1, 2, 0))
batch_encoder_outputs = encoder_outputs.swapaxes(0, 1)
decoder_context = nd.batch_dot(batch_attention, batch_encoder_outputs)
input_and_context = nd.concat(
    nd.expand_dims(self.embedding(cur_input), axis=1),
    decoder_context, dim=2)
concat_input = self.rnn_concat_input(input_and_context).reshape(
    (1, -1, 0))
concat_input = self.dropout(concat_input)
state = [nd.broadcast_axis(single_layer_state[0], axis=0,
                           size=self.num_layers)]
output, state = self.rnn(concat_input, state)
output = self.dropout(output)
output = self.out(output).reshape((-3, -1))
return output, state

def begin_state(self, *args, **kwargs):
    return self.rnn.begin_state(*args, **kwargs)

```

为了初始化解码器的隐藏状态，我们通过一层全连接网络来转化编码器的输出隐藏状态。

```

In [8]: class DecoderInitState(nn.Block):
    """ 解码器隐藏状态的初始化。 """
    def __init__(self, encoder_num_hiddens, decoder_num_hiddens, **kwargs):
        super(DecoderInitState, self).__init__(**kwargs)
        with self.name_scope():
            self.dense = nn.Dense(decoder_num_hiddens,
                                 in_units=encoder_num_hiddens,
                                 activation="tanh", flatten=False)

    def forward(self, encoder_state):
        return [self.dense(encoder_state)]

```

## 训练和应用模型

我们定义 `translate` 函数来应用训练好的模型。这些模型通过该函数的前三个参数传递。解码器的最初时刻输入来自 BOS 字符。当任一时刻的输出为 EOS 字符时，输出序列即完成。

```
In [9]: def translate(encoder, decoder, decoder_init_state, fr_ens, ctx, max_seq_len):
    for fr_en in fr_ens:
        print('[input] ', fr_en[0])
        input_tokens = fr_en[0].split(' ') + [EOS]
        # 添加 PAD 符号使每个序列等长（长度为 max_seq_len）。
        while len(input_tokens) < max_seq_len:
            input_tokens.append(PAD)
        inputs = nd.array(output_vocab.to_indices(input_tokens), ctx=ctx)
        encoder_state = encoder.begin_state(func=nd.zeros, batch_size=1,
                                             ctx=ctx)
        encoder_outputs, encoder_state = encoder(inputs.expand_dims(0),
                                                encoder_state)
        encoder_outputs = encoder_outputs.flatten()
        # 解码器的第一个输入为 BOS 字符。
        decoder_input = nd.array([output_vocab.token_to_idx[BOS]], ctx=ctx)
        decoder_state = decoder_init_state(encoder_state[0])
        output_tokens = []

        for _ in range(max_test_output_len):
            decoder_output, decoder_state = decoder(
                decoder_input, decoder_state, encoder_outputs)
            pred_i = int(decoder_output.argmax(axis=1).asnumpy()[0])
            # 当任一时刻的输出为 EOS 字符时，输出序列即完成。
            if pred_i == output_vocab.token_to_idx[EOS]:
                break
            else:
                output_tokens.append(output_vocab.idx_to_token[pred_i])
            decoder_input = nd.array([pred_i], ctx=ctx)
            print('[output]', ' '.join(output_tokens))
            print('[expect]', fr_en[1], '\n')
```

下面定义模型训练函数。为了初始化解码器的隐藏状态，我们通过一层全连接网络来转化编码器最早时刻的输出隐藏状态。这里的解码器使用当前时刻的预测结果作为下一时刻的输入。

```
In [10]: loss = gloss.SoftmaxCrossEntropyLoss()
eos_id = output_vocab.token_to_idx[EOS]

def train(encoder, decoder, decoder_init_state, max_seq_len, ctx,
          eval_fr_ens):
```

```

encoder.initialize(init.Xavier(), ctx=ctx)
decoder.initialize(init.Xavier(), ctx=ctx)
decoder_init_state.initialize(init.Xavier(), ctx=ctx)
encoder_optimizer = gluon.Trainer(encoder.collect_params(), 'adam',
                                   {'learning_rate': lr})
decoder_optimizer = gluon.Trainer(decoder.collect_params(), 'adam',
                                   {'learning_rate': lr})
decoder_init_state_optimizer = gluon.Trainer(
    decoder_init_state.collect_params(), 'adam', {'learning_rate': lr})

data_iter = gdata.DataLoader(dataset, batch_size, shuffle=True)
l_sum = 0
for epoch in range(1, epochs + 1):
    for x, y in data_iter:
        cur_batch_size = x.shape[0]
        with autograd.record():
            l = nd.array([0], ctx=ctx)
            valid_length = nd.array([0], ctx=ctx)
            encoder_state = encoder.begin_state(
                func=nd.zeros, batch_size=cur_batch_size, ctx=ctx)
            encoder_outputs, encoder_state = encoder(x, encoder_state)
            encoder_outputs = encoder_outputs.flatten()
            # 解码器的第一个输入为 BOS 字符。
            decoder_input = nd.array(
                [output_vocab.token_to_idx[BOS]] * cur_batch_size,
                ctx=ctx)
            mask = nd.ones(shape=(cur_batch_size,), ctx=ctx)
            decoder_state = decoder_init_state(encoder_state[0])
            for i in range(max_seq_len):
                decoder_output, decoder_state = decoder(
                    decoder_input, decoder_state, encoder_outputs)
                # 解码器使用当前时刻的预测结果作为下一时刻的输入。
                decoder_input = decoder_output.argmax(axis=1)
                valid_length = valid_length + mask.sum()
                l = l + (mask * loss(decoder_output, y[:, i])).sum()
                mask = mask * (y[:, i] != eos_id)
            l = l / valid_length
            l.backward()
            encoder_optimizer.step(1)
            decoder_optimizer.step(1)
            decoder_init_state_optimizer.step(1)
            l_sum += l.asscalar() / max_seq_len

```

```

if epoch % epoch_period == 0 or epoch == 1:
    if epoch == 1:
        print('epoch %d, loss %f, ' % (epoch, l_sum / len(data_iter)))
    else:
        print('epoch %d, loss %f, '
              % (epoch, l_sum / epoch_period / len(data_iter)))
if epoch != 1:
    l_sum = 0
translate(encoder, decoder, decoder_init_state, eval_fr_ens, ctx,
          max_seq_len)

```

以下分别实例化编码器、解码器和解码器初始隐藏状态网络。

```

In [11]: encoder = Encoder(len(input_vocab), encoder_embed_size, encoder_num_hiddens,
                           encoder_num_layers, encoder_drop_prob)
decoder = Decoder(decoder_num_hiddens, len(output_vocab),
                  decoder_num_layers, max_seq_len, decoder_drop_prob,
                  alignment_size, encoder_num_hiddens)
decoder_init_state = DecoderInitState(encoder_num_hiddens,
                                       decoder_num_hiddens)

```

给定简单的法语和英语序列，我们可以观察模型的训练结果。打印的结果中，Input、Output 和 Expect 分别代表输入序列、输出序列和正确序列。

```

In [12]: eval_fr_ens =[['elle est japonaise .', 'she is japanese .'],
                     ['ils regardent .', 'they are watching .']]
train(encoder, decoder, decoder_init_state, max_seq_len, ctx, eval_fr_ens)

epoch 1, loss 0.539105,
[input] elle est japonaise .
[output] they . . . . . . . . . . .
[expect] she is japanese .

[input] ils regardent .
[output] they are . . . . . . . . . .
[expect] they are watching .

epoch 10, loss 0.198891,
[input] elle est japonaise .
[output] she is japanese .
[expect] she is japanese .

[input] ils regardent .
[output] they are watching .
[expect] they are watching .

```

```

epoch 20, loss 0.077858,
[input] elle est japonaise .
[output] she is old .
[expect] she is japanese .

[input] ils regardent .
[output] they are watching .
[expect] they are watching .

epoch 30, loss 0.022904,
[input] elle est japonaise .
[output] she is japanese .
[expect] she is japanese .

 ils regardent .
[output] they are watching .
[expect] they are watching .

epoch 40, loss 0.000890,
 elle est japonaise .
[output] she is japanese .
[expect] she is japanese .

 ils regardent .
[output] they are watching .
[expect] they are watching .

```

## 10.7.2 束搜索

在上一节里，我们提到编码器最终输出了一个背景变量  $\mathbf{c}$ ，该背景变量编码了输入序列  $x_1, x_2, \dots, x_T$  的信息。假设训练数据中的输出序列是  $y_1, y_2, \dots, y_{T'}$ ，输出序列的生成概率是

$$\mathbb{P}(y_1, \dots, y_{T'}) = \prod_{t'=1}^{T'} \mathbb{P}(y_{t'} \mid y_1, \dots, y_{t'-1}, \mathbf{c}).$$

对于机器翻译的输出来说，如果输出语言的词汇集合  $\mathcal{Y}$  的大小为  $|\mathcal{Y}|$ ，输出序列的长度为  $T'$ ，那么可能的输出序列种类是  $\mathcal{O}(|\mathcal{Y}|^{T'})$ 。为了找到生成概率最大的输出序列，一种方法是计算所有  $\mathcal{O}(|\mathcal{Y}|^{T'})$  种可能序列的生成概率，并输出概率最大的序列。我们将该序列称为最优序列。但是这种方法的计算开销过高（例如， $10000^{10} = 1 \times 10^{40}$ ）。

我们目前所介绍的解码器在每个时刻只输出生成概率最大的一个词汇。对于任一时刻  $t'$ , 我们从  $|\mathcal{Y}|$  个词中搜索出输出词

$$y_{t'} = \operatorname{argmax}_{y_{t'} \in \mathcal{Y}} \mathbb{P}(y_{t'} \mid y_1, \dots, y_{t'-1}, \mathbf{c})$$

因此, 搜索计算开销 ( $\mathcal{O}(|\mathcal{Y}| \times T')$ ) 显著下降 (例如,  $10000 \times 10 = 1 \times 10^5$ ), 但这并不能保证一定搜索到最优序列。

束搜索 (beam search) 介于上面二者之间。我们来看一个例子。

假设输出序列的词典中只包含五个词:  $\mathcal{Y} = \{A, B, C, D, E\}$ 。束搜索的一个超参数叫做束宽 (beam width)。以束宽等于 2 为例, 假设输出序列长度为 3, 假如时刻 1 生成概率  $\mathbb{P}(y_{t'} \mid \mathbf{c})$  最大的两个词为  $A$  和  $C$ , 我们在时刻 2 对于所有的  $y_2 \in \mathcal{Y}$  都分别计算  $\mathbb{P}(y_2 \mid A, \mathbf{c})$  和  $\mathbb{P}(y_2 \mid C, \mathbf{c})$ , 从计算出的 10 个概率中取最大的两个, 假设为  $\mathbb{P}(B \mid A, \mathbf{c})$  和  $\mathbb{P}(E \mid C, \mathbf{c})$ 。那么, 我们在时刻 3 对于所有的  $y_3 \in \mathcal{Y}$  都分别计算  $\mathbb{P}(y_3 \mid A, B, \mathbf{c})$  和  $\mathbb{P}(y_3 \mid C, E, \mathbf{c})$ , 从计算出的 10 个概率中取最大的两个, 假设为  $\mathbb{P}(D \mid A, B, \mathbf{c})$  和  $\mathbb{P}(D \mid C, E, \mathbf{c})$ 。

接下来, 我们可以在输出序列:  $A$ 、 $C$ 、 $AB$ 、 $CE$ 、 $ABD$ 、 $CED$  中筛选出以特殊字符 EOS 结尾的候选序列。再在候选序列中取以下分数最高的序列作为最终候选序列:

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

其中  $L$  为候选序列长度,  $\alpha$  一般可选为 0.75。分母上的  $L^\alpha$  是为了惩罚较长序列的分数中的对数相加项。

### 10.7.3 评价翻译结果

2002 年, IBM 团队提出了一种评价翻译结果的指标, 叫做 BLEU (Bilingual Evaluation Under-study)。

设  $k$  为我们希望评价的 n-gram 的最大长度, 例如  $k = 4$ 。n-gram 的精度  $p_n$  为模型输出中的 n-gram 匹配参考输出的数量与模型输出中的 n-gram 的数量的比值。例如, 参考输出 (真实值) 为 ABCDEF, 模型输出为 ABCD。那么  $p_1 = 4/5, p_2 = 3/4, p_3 = 1/3, p_4 = 0$ 。设  $len_{ref}$  和  $len_{MT}$  分别为参考输出和模型输出的词数。那么, BLEU 的定义为

$$\exp(\min(0, 1 - \frac{len_{ref}}{len_{MT}})) \prod_{i=1}^k p_n^{1/2^n}$$

需要注意的是，随着  $n$  的提高， $n$ -gram 的精度的权值随着  $p_n^{1/2^n}$  中的指数减小而提高。例如  $0.5^{1/2} \approx 0.7$ ,  $0.5^{1/4} \approx 0.84$ ,  $0.5^{1/8} \approx 0.92$ ,  $0.5^{1/16} \approx 0.96$ 。换句话说，匹配 4-gram 比匹配 1-gram 应该得到更多奖励。另外，模型输出越短往往越容易得到较高的  $n$ -gram 的精度。因此，BLEU 公式里连乘项前面的系数为了惩罚较短的输出。例如当  $k = 2$  时，参考输出为 ABCDEF，而模型输出为 AB，此时的  $p_1 = p_2 = 1$ ，而  $\exp(1 - 6/2) \approx 0.14$ ，因此 BLEU=0.14。当模型输出也为 ABCDEF 时，BLEU=1。

#### 10.7.4 小结

- 我们可以将编码器—解码器和注意力机制应用于神经机器翻译中。
- 束搜索有可能提高输出质量。
- BLEU 可以用来评价翻译结果。

#### 10.7.5 练习

- 试着使用更大的翻译数据集来训练模型，例如WMT和Tatoeba Project。调一调不同参数并观察实验结果。
- Teacher forcing：在模型训练中，试着让解码器使用当前时刻的正确结果（而不是预测结果）作为下一时刻的输入。结果会怎么样？

#### 10.7.6 扫码直达讨论区



## 11.1 数学基础

TODO(@astonzhang)。

### 11.1.1 线性代数

向量和 $L_p$ 范数

设  $d$  维向量  $\mathbf{x} = [x_1, x_2, \dots, x_d]^\top$ , 或

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}.$$

上式中  $x_1, \dots, x_d$  是向量的元素。我们用  $\mathbf{x} \in \mathbb{R}^d$  或  $\mathbf{x} \in \mathbb{R}^{d \times 1}$  表示， $\mathbf{x}$  是一个各元素均为实数的  $d$  维向量。

向量  $\mathbf{x}$  的  $L_p$  范数为

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

例如  $\mathbf{x}$  的  $L_1$  范数是该向量元素绝对值的和：

$$\|\mathbf{x}\|_1 = \sum_{i=1}^d |x_i|.$$

而  $\mathbf{x}$  的  $L_2$  范数是该向量元素平方和的平方根：

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^d x_i^2}.$$

我们通常用  $\|\mathbf{x}\|$  指代  $\|\mathbf{x}\|_2$ 。

## 矩阵和 Frobenius 范数

设  $m$  行  $n$  列矩阵

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix}.$$

上式中  $x_{11}, \dots, x_{mn}$  是矩阵的元素。我们用  $\mathbf{X} \in \mathbb{R}^{m \times n}$  表示， $\mathbf{X}$  是一个各元素均为实数的  $m$  行  $n$  列矩阵。

矩阵  $\mathbf{X}$  的 Frobenius 范数为该矩阵元素平方和的平方根：

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

## 特征向量和特征值

### 按元素运算

假设  $\mathbf{x} = [4, 9]^\top$ ，以下是一些按元素运算的例子：

- 按元素相加:  $\mathbf{x} + 1 = [5, 10]^\top$
- 按元素相乘:  $\mathbf{x} \odot \mathbf{x} = [16, 81]^\top$
- 按元素相除:  $72/\mathbf{x} = [18, 8]^\top$
- 按元素开方:  $\sqrt{\mathbf{x}} = [2, 3]^\top$

## 运算

转置、点乘（内积）、矩阵加法、减法和乘法。

### 11.1.2 导数和梯度

TODO(@astonzhang)。

假设函数  $f: \mathbb{R}^d \rightarrow \mathbb{R}$  的输入是一个  $d$  维向量  $\mathbf{x} = [x_1, x_2, \dots, x_d]^\top$ 。函数  $f(\mathbf{x})$  有关  $\mathbf{x}$  的梯度是一个由  $d$  个偏导数组成的向量:

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

为表示简洁，我们有时用  $\nabla f(\mathbf{x})$  代替  $\nabla_{\mathbf{x}} f(\mathbf{x})$ 。

## 常用梯度

假设  $\mathbf{x}$  是一个向量，那么

$$\begin{aligned}\nabla_{\mathbf{x}} \mathbf{A}^\top \mathbf{x} &= \mathbf{A} \\ \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} &= \mathbf{A} \\ \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} \mathbf{x} &= (\mathbf{A} + \mathbf{A}^\top) \mathbf{x} \\ \nabla_{\mathbf{x}} \|\mathbf{x}\|^2 &= \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}\end{aligned}$$

假设  $\mathbf{X}$  是一个矩阵，那么

$$\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}.$$

## 泰勒展开

函数  $f$  的泰勒展开式是

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!} (x-a)^n,$$

其中  $f^{(n)}$  为函数  $f$  的  $n$  阶导数。假设  $\epsilon$  是个足够小的数，如果将上式中  $x$  和  $a$  分别替换成  $x + \epsilon$  和  $x$ ，我们可以得到

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

由于  $\epsilon$  足够小，上式也可以简化成

$$f(x + \epsilon) \approx f(x) + f'(x)\epsilon.$$

## 黑塞矩阵（Hessian matrix）

### 11.1.3 概率和统计

全概率和条件概率

最大似然估计

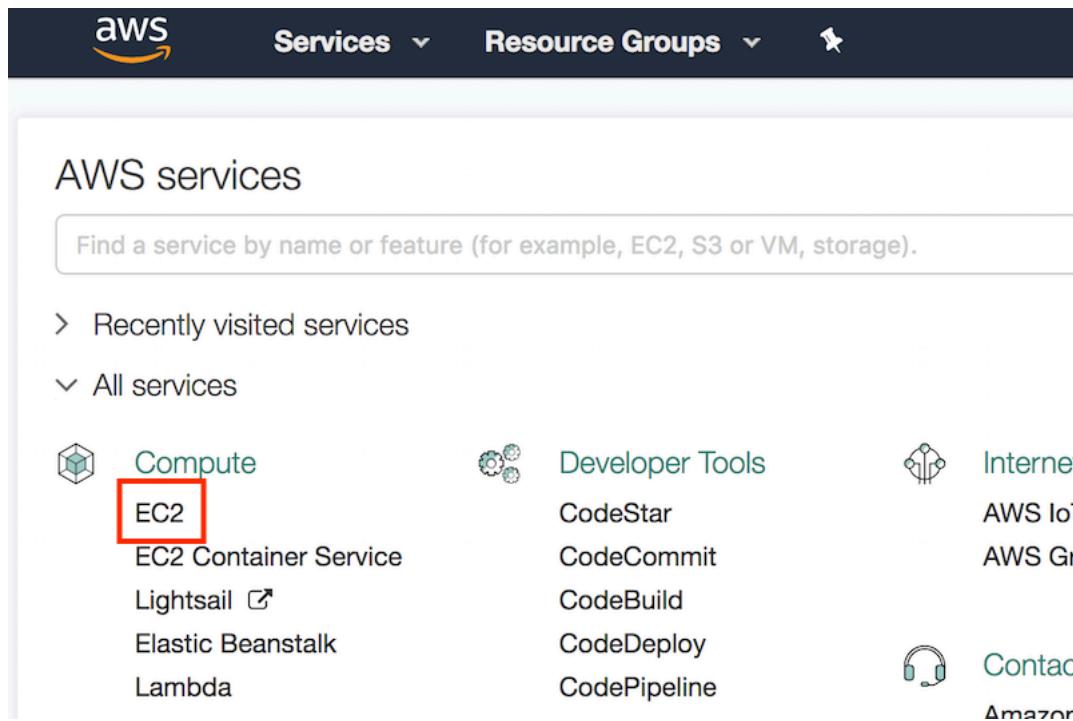
## 11.2 在 AWS 上运行教程

本节我们一步步讲解如何从 0 开始在 AWS 上申请 CPU 或者 GPU 机器并运行教程。

### 11.2.1 申请账号并登陆

首先我们需要在<https://aws.amazon.com/>上面创建账号，通常这个需要一张信用卡。不熟悉的同学可以自行搜索“如何注册 aws 账号”。【注意】AWS 中国需要公司实体才能注册，个人用户请注册 AWS 全球账号。

登陆后点击 EC2 进入 EC2 面板：



### 11.2.2 选择并运行 EC2 实例

【可选】进入面板后可以在右上角选择离我们较近的数据中心来减低延迟。例如国内用户可以选亚太地区数据中心。

【注意】有些数据中心可能没有 GPU 实例。

然后点击启动实例来选择操作系统和实例类型。

The screenshot shows the AWS EC2 Dashboard. On the left sidebar, there are several navigation items: EC2 Dashboard, Events, Tags, Reports, Limits, INSTANCES (with sub-options Instances, Spot Requests, Reserved Instances, Scheduled Instances, Dedicated Hosts), IMAGES (with sub-options AMIs, Bundle Tasks), ELASTIC BLOCK STORE (with sub-options Volumes, Snapshots), and NETWORK & SECURITY (with sub-option Security Groups). The main content area is titled 'Resources' and displays the following statistics for the US West (Oregon) region:

7 Running Instances	2 Elastic IPs
0 Dedicated Hosts	6 Snapshots
9 Volumes	0 Load Balancers
1 Key Pairs	39 Security Groups
0 Placement Groups	

Below these stats, a callout box says: "Just need a simple virtual private server? Get everything you need to jumpstart your project - compute, storage, and networking – for a low, predictable price. Try Amazon Lightsail for free." A large blue button labeled "Launch Instance" is prominently displayed.

The top right corner of the dashboard shows the region as "Oregon" with a red box around it. To the right of the dashboard, there's a sidebar titled "Account Attributed" with links to Supported Platforms (VPC), Default VPC (vpc-50265b34), Resource ID length, and Additional Information (Getting Started Guide, Documentation, All EC2 Resources, Forums, Pricing, Contact Us). Below this is the "AWS Marketplace" section, which includes a note about a free software trial and links to the Marketplace and EC2 Launch Wizard.

在接下来的操作系统里面选 Ubuntu 16.06:

This screenshot shows the "Step 1: Choose an Amazon Machine Image (AMI)" page of the EC2 instance creation wizard. The top navigation bar has tabs: 1. Choose AMI (highlighted in orange), 2. Choose Instance Type, 3. Configure Instance, 4. Add Storage, 5. Add Tags, 6. Configure Security Group, and 7. Review. A "Cancel and Exit" link is also present.

The main content area shows a list of AMIs. One item is highlighted with a red box: "Ubuntu Server 16.04 LTS (HVM), SSD Volume Type - ami-6e1a0117". This item is described as "Free tier eligible". To the right of this item is a "Select" button, which is also highlighted with a red box. Below the AMI list, there is some descriptive text about the selected AMI.

EC2 提供大量的有着不同配置的实例。这里我们选择了 p2.xlarge, 它有一个 K80 GPU。我们也可以选择有更多 GPU 的实例例如 p2.16xlarge, 或者有新一点 GPU 的 g3 系列。我们也可以选择只有 CPU 的实例, 例如 c4 系列。每个不同实例的机器配置和收费可以参考 [ec2instances.info](http://ec2instances.info).

## Step 2: Choose an Instance Type

	GPU instances	g2.8xlarge	32	60 2 x 120 (SSD)
	GPU compute	p2.xlarge	4	61 EBS only

【注意】选择某个类型的样例前我们需要在 `Limits` 里检查下这个是不是有数量限制。例如这个账号的 `p2.xlarge` 限制是最多一个区域开一个。如果需要更多，需要点右边来申请更多的实例容量（通常需要一个工作日来处理）。

EC2 Dashboard	Events	Tags	Reports	Limits	Instances
					Running On-Demand p2.16xlarge instances 1 Request limit increase
					Running On-Demand p2.8xlarge instances 1 Request limit increase
					Running On-Demand p2.xlarge instances 1 Request limit increase
					Running On-Demand r3.2xlarge instances 20 Request limit increase

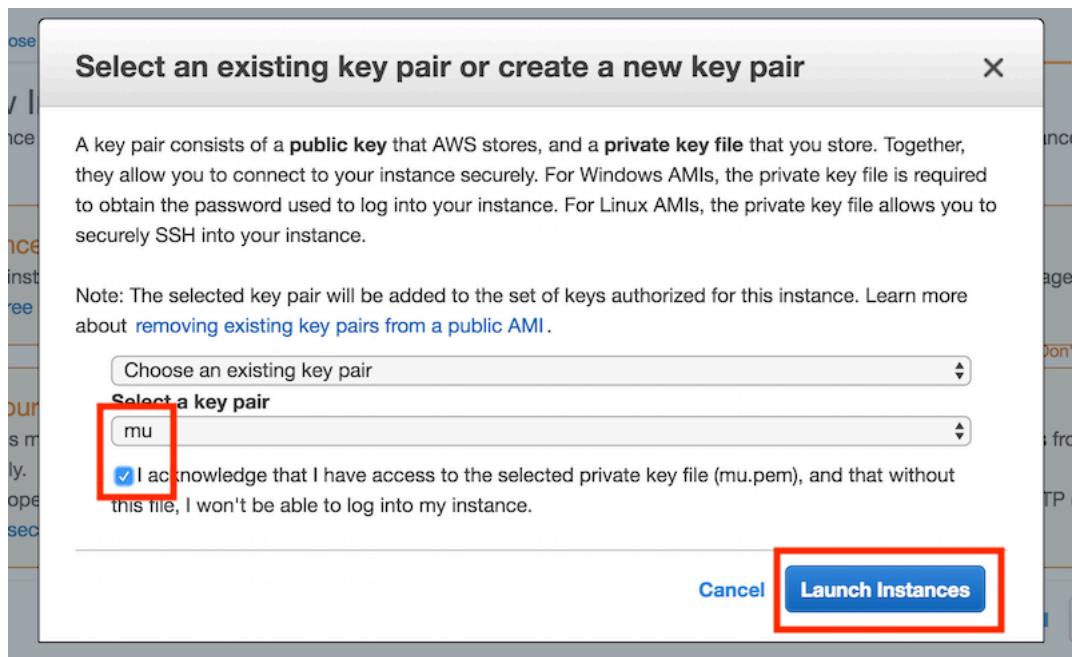
然后我们在存储那里将默认的硬盘从 8GB 增大的 40GB. 因为安装 CUDA 需要 4GB 空间，所以最小推荐 20GB（只有 CPU 的话不需要 CUDA，可以更少）。

## Step 4: Add Storage

Volume Type	Device	Snapshot	Size (GiB)	Volume Type	IOPS	Throughput (MiB/s)
Root	/dev/sda1	snap-02cc1e40d2e121683	40	General Purpose S	100 / 3000	N/A

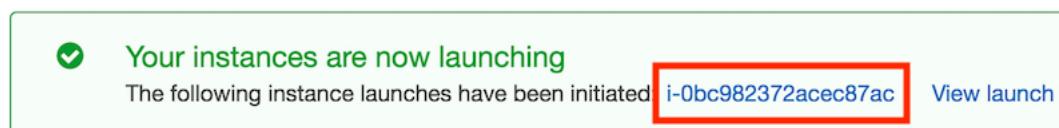
Add New Volume

其他的项我们都选默认，然后可以启动了。这时候会跳出选项选择 `key pair`，这是用来之后访问机器的秘钥（EC2 默认不支持密码访问）。如果没有的话可以选择生成一对秘钥。

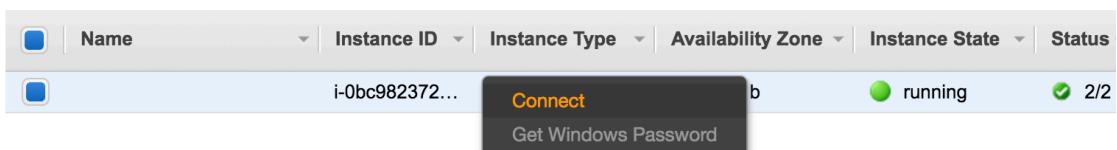


然后点击实例的 ID 可以查看当前实例的状态

## Launch Status



状态变绿后右击点 Connect 就可以看到如何访问这个实例的说明了



例如我们这里

```
Mus-MacBook-Pro:~ muli$ ssh ubuntu@ec2-34-203-223-63.compute-1.amazonaws.com
The authenticity of host 'ec2-34-203-223-63.compute-1.amazonaws.com (34.203.223.63)' can't be established.
ECDSA key fingerprint is SHA256:8aPhw5p745TdmsUxnWb+MpWF+vyMKddahBKB12eYi8I.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-34-203-223-63.compute-1.amazonaws.com' (RSA) to the list of known hosts.
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-1022-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

Get cloud support with Ubuntu Advantage Cloud Guest:
  http://www.ubuntu.com/business/services/cloud
```

### 11.2.3 安装依赖包

成功登陆后我们先更新并安装编译需要的包。

```
sudo apt-get update && sudo apt-get install -y build-essential git libgfortran3
```

#### 安装 CUDA

【注意】只有 CPU 的实例可以跳过步骤。

我们去 Nvidia 官网下载 CUDA 并安装。选择正确的版本并获取下载地址。

【注意】目前 CUDA 默认下载 9.0 版，但 mxnet-cu90 的 daily build 还不完善。建议使用下面命令安装 8.0 版。

## Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

### Operating System

Windows

Linux

Mac OSX

### Architecture

x86\_64

ppc64le

### Distribution

Fedora

OpenSUSE

RHEL

CentOS

SLES

Ubuntu

### Version

17.04

16.04

### Installer Type

runfile (local)

deb (local)

deb (network)

cluster (local)

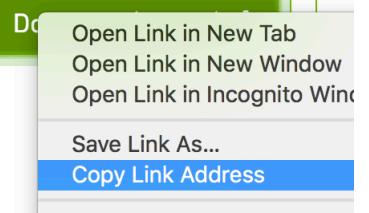
## Download Installer for Linux Ubuntu 16.04 x86\_64

The base installer is available for download below.

### › Base Installer

#### Installation Instructions:

1. Run `sudo sh cuda\_9.0.176\_384.81\_linux.run`
2. Follow the command-line prompts



然后使用 wget 下载并且安装

```
wget https://developer.nvidia.com/compute/cuda/8.0/Prod2/local_installers/cuda_8.0.61_375.26_linux-run
sudo sh cuda_8.0.61_375.26_linux-run
```

这里需要回答几个问题。

```
accept/decline/quit: accept
Install NVIDIA Accelerated Graphics Driver for Linux-x86_64 375.26?
(y)es/(n)o/(q)uit: y
Do you want to install the OpenGL libraries?
(y)es/(n)o/(q)uit [ default is yes ]: y
Do you want to run nvidia-xconfig?
(y)es/(n)o/(q)uit [ default is no ]: n
Install the CUDA 8.0 Toolkit?
(y)es/(n)o/(q)uit: y
Enter Toolkit Location
[ default is /usr/local/cuda-8.0 ]:
Do you want to install a symbolic link at /usr/local/cuda?
(y)es/(n)o/(q)uit: y
Install the CUDA 8.0 Samples?
(y)es/(n)o/(q)uit: n
```

安装完成后运行

```
nvidia-smi
```

就可以看到这个实例的 GPU 了。最后将 CUDA 加入到 library path 方便之后安装的库找到它。

```
echo "export LD_LIBRARY_PATH=\${LD_LIBRARY_PATH}:/usr/local/cuda-8.0/lib64" >>.bashrc
```

## 安装 MXNet

先安装 Miniconda

```
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
```

需要回答下面几个问题

```
Do you accept the license terms? [yes|no]
[no] >>> yes
Do you wish the installer to prepend the Miniconda3 install location
to PATH in your /home/ubuntu/.bashrc ? [yes|no]
[no] >>> yes
```

运行一次 bash 让 CUDA 和 conda 生效。

下载本教程，安装并激活 conda 环境

```
git clone https://github.com/mli/gluon-tutorials-zh
cd gluon-tutorials-zh
conda env create -f environment.yml
source activate gluon
```

默认环境里安装了只有 CPU 的版本。现在我们替换成 GPU 版本。

```
pip uninstall -y mxnet
pip install --pre mxnet-cu80
```

同时安装 notedown 插件来让 jupyter 读写 markdown 文件。

```
pip install https://github.com/mli/notedown/tarball/master
jupyter notebook --generate-config
echo "c.NotebookApp.contents_manager_class = 'notedown.NotedownContentsManager'" >>~/.jupyter/jupyter_notebook_config.py
```

## 11.2.4 运行

并运行 Jupyter notebook。

```
jupyter notebook
```

如果成功的话会看到类似的输出

```
(gluon) ubuntu@ip-172-31-0-171:~/gluon-tutorials-zh$ jupyter notebook
[I 22:10:29.383 NotebookApp] Writing notebook server cookie secret to /run/user/1000/jupyter_cookie
[I 22:10:29.404 NotebookApp] Serving notebooks from local directory: /home/ubuntu
[I 22:10:29.404 NotebookApp] 0 active kernels
[I 22:10:29.404 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888
[I 22:10:29.404 NotebookApp] Use Control-C to stop this server and shut down all
[W 22:10:29.404 NotebookApp] No web browser found: could not locate runnable browser
[C 22:10:29.404 NotebookApp]
```

Copy/paste this URL into your browser when you connect for the first time,  
to login with a token:

<http://localhost:8888/?token=c9c7b7d48caedc5c039535d777450f61dff7c0ad6fc1>

因为我们的实例没有暴露 8888 端口，所以我们可以在本地启动 ssh 从实例映射到本地

```
ssh -i "XXX.pem" -L8888:localhost:8888 ubuntu@XXXX.XXXX.compute.amazonaws.com
```

然后把 jupyter log 里的 URL 复制到本地浏览器就行了。

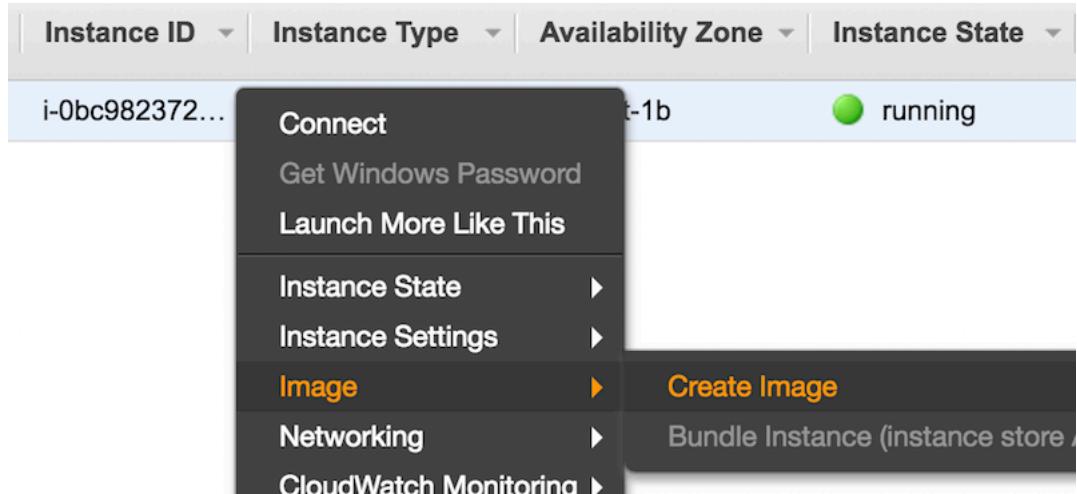
【注意】如果本地运行了 Jupyter notebook, 那么 8888 端口就可能被占用了。要么关掉本地 jupyter, 要么把端口映射改成别的。例如, 假设 aws 使用默认 8888 端口, 我们可以在本地启动 ssh 从实例映射到本地 8889 端口:

```
ssh -i "XXX.pem" -N -f -L localhost:8889:localhost:8888 ubuntu@XXXX.XXXX.compute.amazonaws.com
```

然后在本地浏览器打开 localhost:8889, 这时会提示需要 token 值。接下来, 我们将 aws 上 jupyter log 里的 token 值 (例如上图里:localhost:8888/?token=token 值) 复制粘贴即可。

### 11.2.5 后续

因为云服务按时间计费, 通常我们不用时需要把样例关掉, 到下次要用时再开。如果是停掉 (Stop), 下次可以直接继续用, 但硬盘空间会计费。如果是终结 (Termination), 我们一般会先把操作系统做镜像, 下次开始时直接使用镜像 (AMI) (上面的教程使用了 Ubuntu 16.06 AMI) 就行了, 不需要再把上面流程走一次。



每次重新开始后，我们建议升级下教程（记得保存自己的改动）

```
cd gluon-tutorials-zh  
git pull
```

和 MXNet 版本

```
source activate gluon  
pip install -U --pre mxnet-cu80
```

### 11.2.6 总结

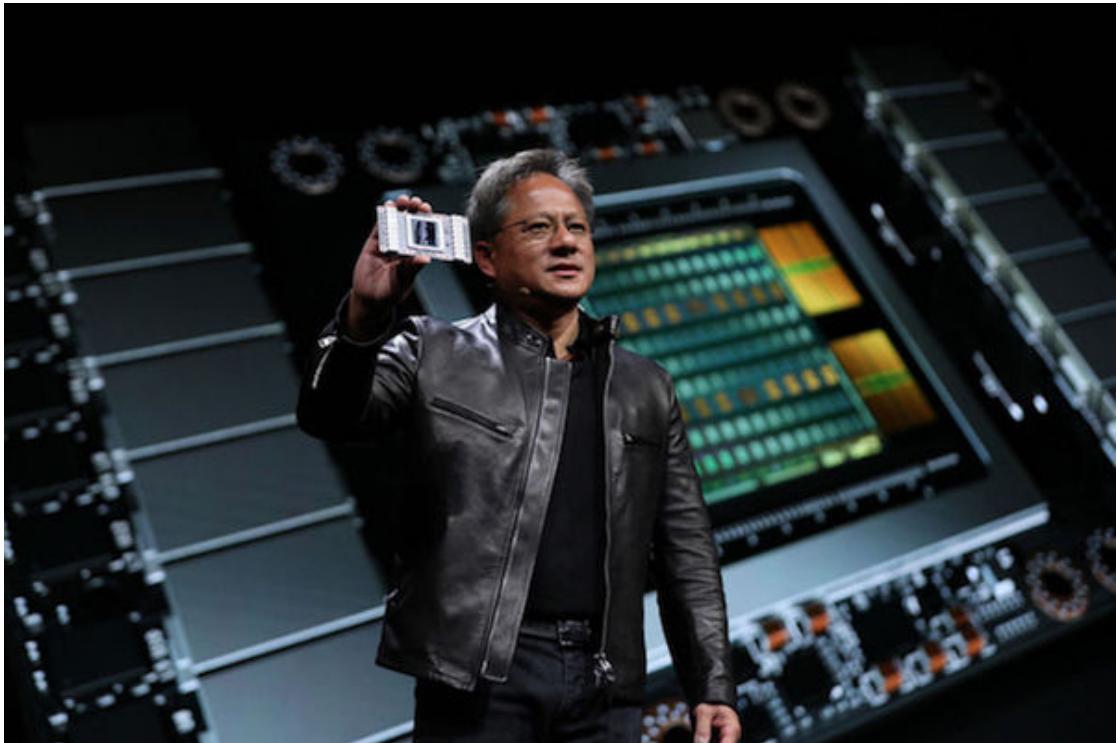
云上可以很方便的获取计算资源和配置环境

### 11.2.7 练习

云很方便，但不便宜。研究下它的价格，和看看如何节省开销。

### 11.3 GPU 购买指南

深度学习训练通常需要大量的计算资源。GPU 目前是深度学习最常使用的计算加速硬件。相对于 CPU 来说，GPU 更便宜（达到同样的计算能力 GPU 一般便宜 10 倍），而且计算更加密集（一台服务器可以搭配 8 块或者 16 块 GPU）。因此 GPU 数量通常是衡量深度学习计算能力的一个标准，同时 Nvidia 的创始人 Jensen Huang 也被人称深度学习教父。



(Nvidia CEO 黃教主和他的战术核武器)

本章我们简要介绍 GPU 的购买须知。这里主要针对个人用户购买一两台自用的 GPU 服务器。而不是针对需要购买

- 100+ 台机器的大公司用户。请咨询专业数据中心维护人员，通常你们会考虑 Nvidia Tesla P100 或者 V100。你可以完全跳过此节。
- 10+ 台机器的实验室和中小公司用户：不缺钱可以上 Nvidia DGX-1，不然可以考虑购买如 Supermicro 之类性价比较高的服务器。此节的一些内容可以做为参考。

### 11.3.1 选择 GPU

目前独立 GPU 主要有 AMD 和 Nvidia 两家厂商。其中 Nvidia 由于深度学习布局较早，深度学习框架支持更好，因此目前主要会选择 Nvidia 的卡。

Nvidia 卡有面向个人用户（例如 GTX 系列）和企业用户（例如 Tesla 系列）两种。企业用户卡通常使用被动散热和增加了内存校验从而更加适合数据中心。但计算能力上两者相当。企业卡通常要贵上 10 倍，因此个人用户通常选用 GTX 系列。

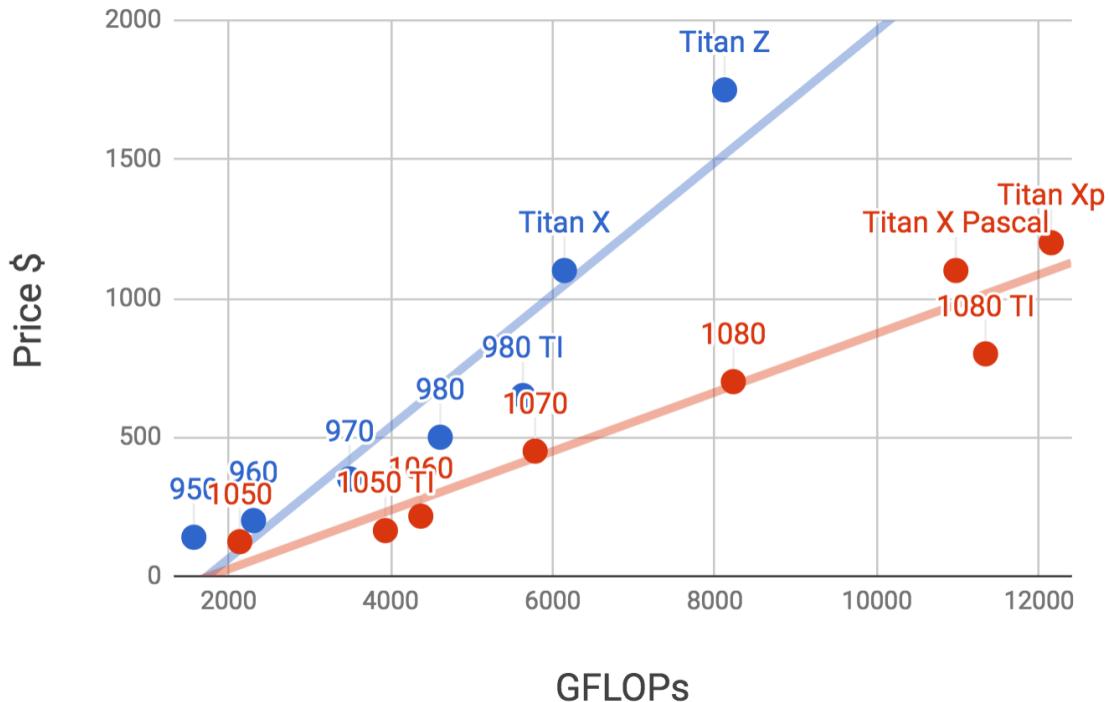
Nvidia 一般每一两年会更新一次大版本，例如目前最新的是 1000 系列。每个系列里面会有数个不同型号，对应不同的性能。

GPU 的性能主要由下面三个主要参数构成：

1. 计算能力。通常我们关心的是 32 位浮点计算能力。当然，对于高玩来说也可以考虑 16 位浮点用来训练，8 位整数来预测。
2. 内存大小。神经网络越深，或者训练时批量大小越大，所需要的 GPU 内存就越多。
3. 内存带宽。内存带宽要足够才能发挥出所有计算能力。

对于大部分用户来说，只要考虑计算能力就行了。内存不要太小就好，例如不要小于 4GB。如果显卡同时要用来显示图形界面，那么推荐 6G 内存。内存带宽可以让厂家来纠结。

下图画了 900 和 1000 系列里各个卡的 32 位浮点计算能力和价格的对比（价格是 wikipedia 的推荐价格，真实价格通常会有浮动）。



我们可以读出两点信息：

1. 在同一个系列里面，通常价格和性能成正比
2. 1000 系列性价比 900 高 2 倍左右。

如果大家继续比较 GTX 前面几代，也发现规律是类似的。根据这个我们推荐

1. 买新不买旧，因为目前看来 GPU 性能还是在快速迭代，贬值较快。
2. 量力购买。不缺钱直接上最好的，但入门的 1050TI 也不错。

### 11.3.2 整机配置

如果主要是用 GPU 来做计算，或者说主要是做深度学习训练，不需要购买高端的 CPU。可以将主要预算花费在 GPU 上。所以整机配置可以参考网上推荐的中高档就好。

不过由于 GPU 的功耗，散热和体积，需要一些额外考虑。

- 机箱体积。GPU 尺寸较大，通常不考虑太小的机箱。而且机箱自带的风扇要好。(下图里我们曾尝试在一个中等机箱里塞满 4 卡导致散热不好烧了 2 块 GPU。)



- 电源。购买 GPU 时需要查下 GPU 的功耗，50w 到 300w 不等。因此买电源时需要功率足够的。(我们倒是一开始就考虑了这个，但忘了不过载机房供电。下面是 5 台机器满负荷运行时烧掉了一个 30A 的电源接口。)



- 主板的 PCIe 卡槽。推荐使用 PCIe 3.0 16x 来保证足够的 GPU 到主内存带宽。如果是多卡的话，要仔细看主板说明，保证多卡一起使用时仍然是 16x 带宽。（有些主板插 4 卡时会降到 8x 甚至 4x）

对于更具体的配置可以参考我们走过的一些弯路，和来讨论区交流大家的机器配置。

## 11.4 gluonbook 函数索引

函数，定义所在章节

- `data_iter`, 线性回归——从零开始
- `linreg`, 线性回归——从零开始
- `squared_loss`, 线性回归——从零开始
- `sgd`, 线性回归——从零开始

- `plt`, 线性回归——从零开始
- `accuracy`, Softmax 回归——从零开始
- `evaluate_accuracy`, Softmax 回归——从零开始
- `load_data_fashion_mnist`, Softmax 回归——从零开始
- `train_cpu`, Softmax 回归——从零开始
- `semilogy`, 欠拟合和过拟合
- `to_onehot`, 循环神经网络——从零开始
- `data_iter_random`, 循环神经网络——从零开始
- `data_iter_consecutive`, 循环神经网络——从零开始
- `grad_clipping`, 循环神经网络——从零开始
- `predict_rnn`, 循环神经网络——从零开始
- `train_and_predict_rnn`, 循环神经网络——从零开始

本教程的[英文版本](#)（注意：中文版本根据社区的反馈做了比较大的更改，我们还在努力的将改动同步到英文版）