动手学深度学习

MXNet Community

2018 年 05 月 17 日

目录

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

1

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。当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。

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。

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

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

$\nabla_{\cdot} y$：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 $\mathbb{P}$

$\mathbb{P}(\cdot \mid \cdot)$：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）

2

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。

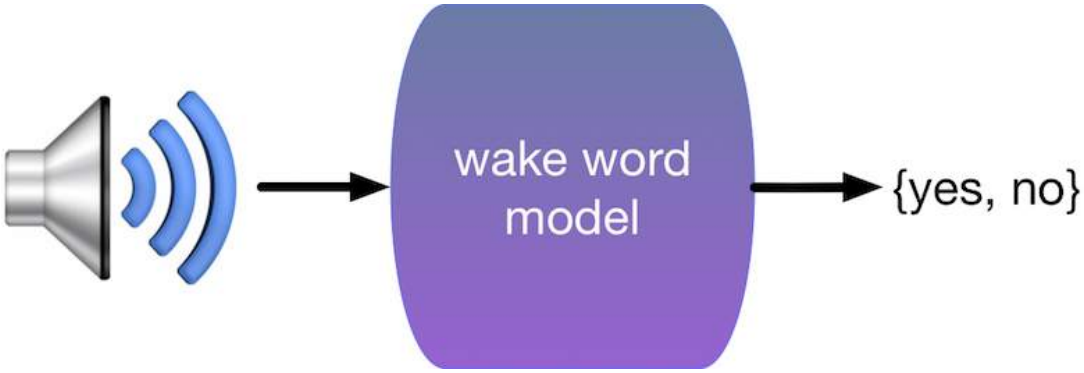
2.1.1 一个例子

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库表格中。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 == 'Okey, 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**

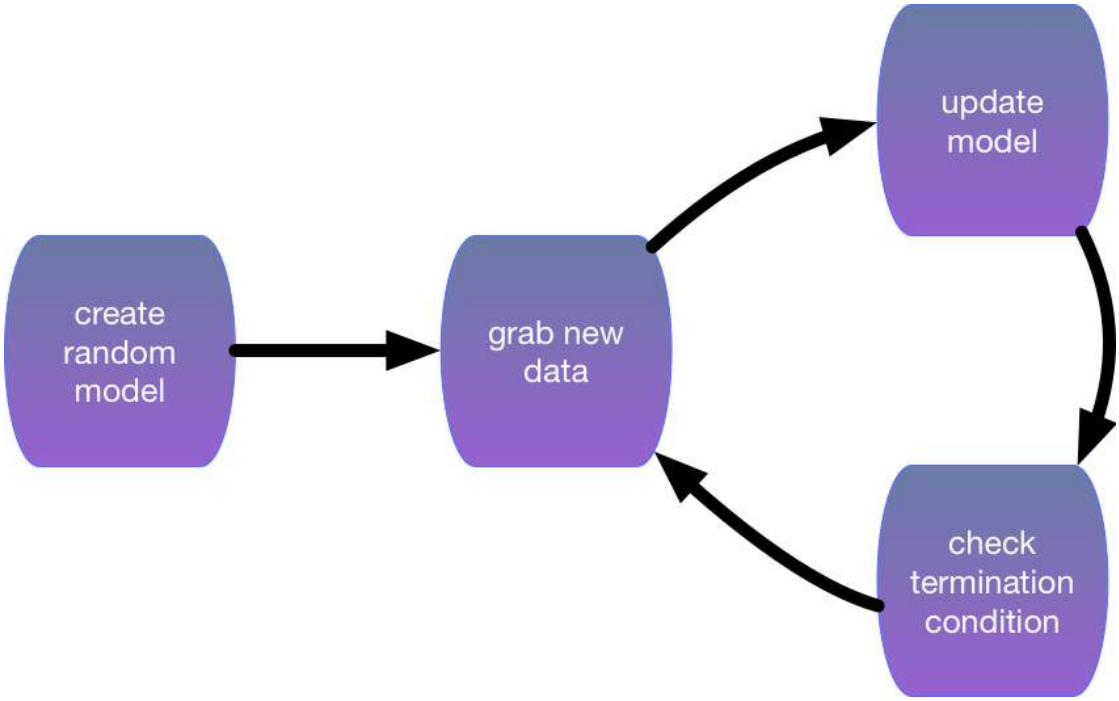
（**input**）是一段语音信号，它的输出则是一个回答 `{yes, no}`，告诉我们这段语音信号是否包含了唤醒词。

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：

			
喵	喵	汪	汪

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，它被广泛应用在了搜索引擎、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, ...]` 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, fizz), (15, fizzbuzz), (23, 23), (40, buzz)]`
- 从输入数据中抽取的特征，例如 `x -> [(x % 3), (x % 5), (x % 15)]`.

有了这些，Jeol 利用 TensorFlow 写了一个分类器。对于不按常理出牌的 Jeol，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。

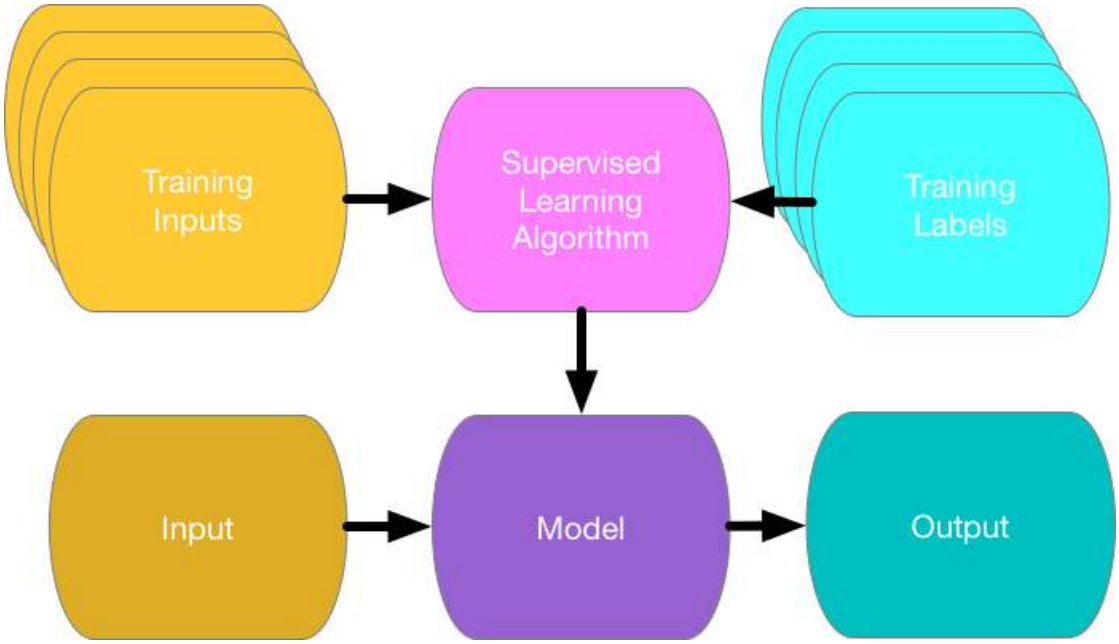
下面我们详细讨论一些不同的机器学习应用，以便理解我们具体会做什么。每个例子，我们都会介绍问题的解决方式，如训练方式、数据类型等等。这些内容仅仅是开胃小菜，并为大家提供一些共同的谈资。随着学习的推进，我们会介绍更多的类似问题。

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 $\{\text{cat}, \text{dog}\}$。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 $\Pr(y = \text{cat} \mid x) = 0.9$。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 $\Pr(y = \text{deathcap} \mid \text{image}) = 0.2$。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：

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

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。

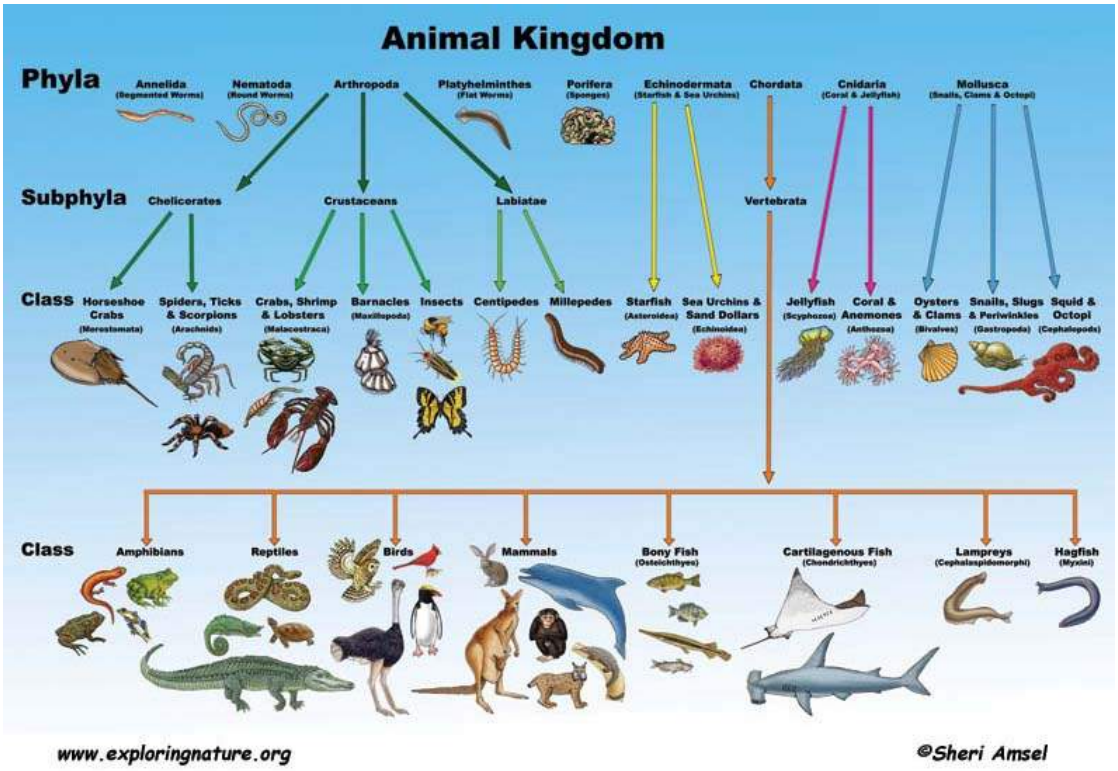


图 2.1: 动物的分类

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）

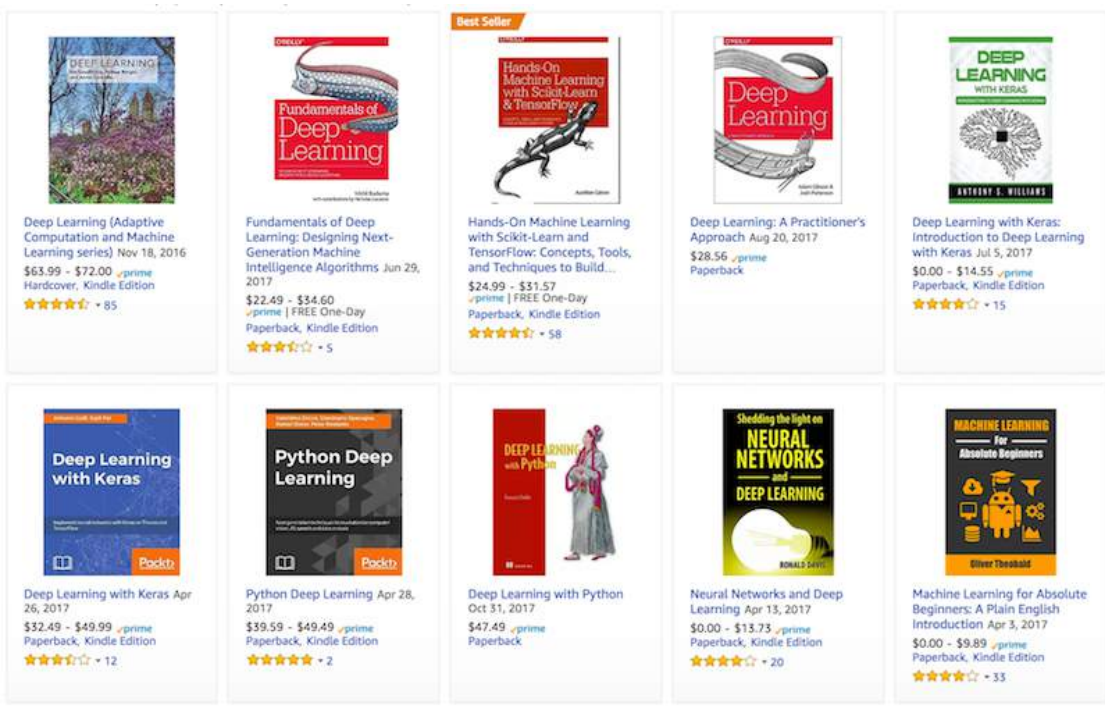
在信息检索领域，我们常常需要进行排序。以网络搜索为例，相较于判断页面是否与检索条目有关，我们更倾向于判断在浩如烟海的搜索结果中，应向用户显示哪个结果。这就要求我们的算法，能从较大的集合中生成一个有序子集。换句话说，如果要求生成字母表中的前 5 个字母，返回 A B C D E 和 C A B E D 是完全不同的。哪怕集合中的元素还是这些，但元素的顺序意义重大。

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。

----D----e--e--e-----p----------- L----ea-------r---------ni----- ng-----


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。

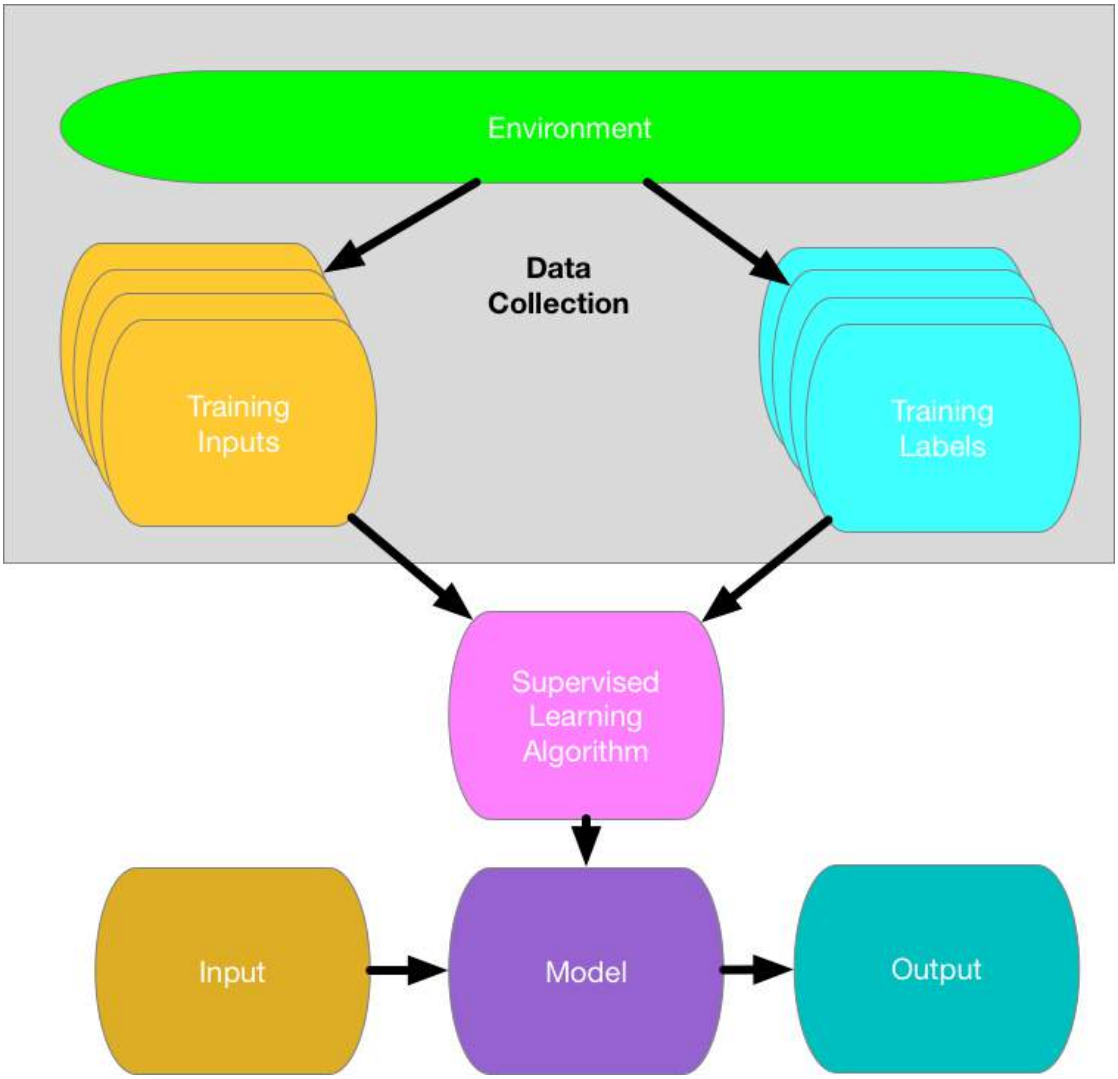
而相反的情形，给那种他们自己都不知道想让你做什么的老板打工也很糟心。不过，如果你打算成为一名数据科学家，你最好习惯这点。老板很可能就扔给你一堆数据，说，用上一点**数据科学**的方法吧。这种要求很模棱两可。我们称这类问题为**无监督学习（unsupervised learning）**。我们能提出的问题的类型和数量仅仅受创造力的限制。我们将在后面的章节中讨论一些无监督学习的技术。现在，介绍一些常见的非监督学习任务作为开胃小菜。

- 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：

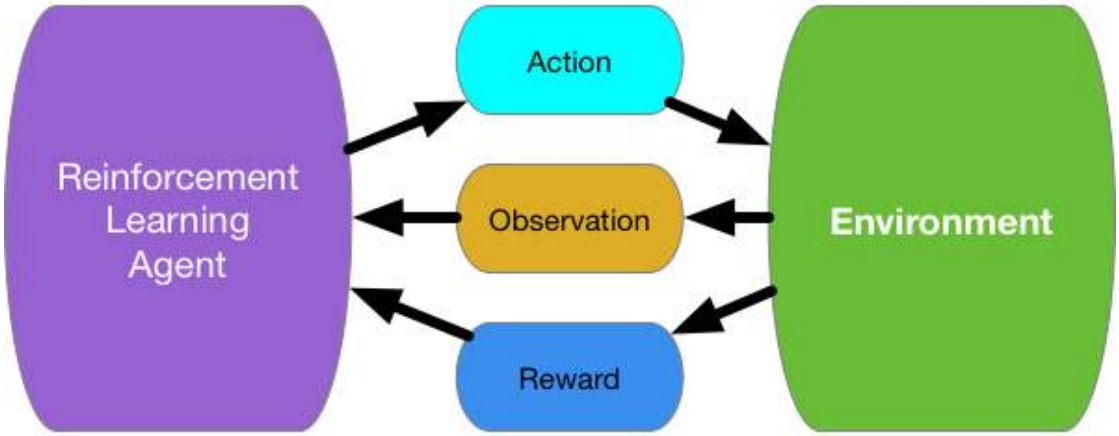
- 记得我们之前的行为吗？
- 愿意帮助我们吗？比如，一个能识别用户口述内容的语音识别器。
- 想要对抗我们？比如，一个对抗装置，类似垃圾邮件过滤（针对垃圾邮件发送者）或游戏玩家（针对对手）？
- 啥都不管（就像大多数情况）？
- 会动态地改变立场（随着时间表现稳定 vs 变化）？

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，将其传送回环境。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应目前的所有状态（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 ）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
```

Windows 用户可以使用如下命令。

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

第四步，打开 Juputer 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.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 muli/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 是存储和转换数据的主要工具。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)>
```

以下是按元素做指数运算。

```
In [13]: y.exp()
```

```
Out[13]:
[[  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 [14]: x == y
```

```
Out[14]:
[[ 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 [15]: nd.dot(x, y.T)
```

```
Out[15]:
         [[  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 [16]: x = nd.array([3, 4])
         x.sum()

Out[16]:
         [ 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 [17]: x.norm().asscalar()

Out[17]: 5.0
```

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 [18]: a = nd.arange(3).reshape((3, 1))
         b = nd.arange(2).reshape((1, 2))
         print('a:', a)
         print('b:', b)
         print('a + b:', a + b)
```

```
a:
[[ 0.]
 [ 1.]
 [ 2.]]
<NDArray 3x1 @cpu(0)>
b:
[[ 0.  1.]]
<NDArray 1x2 @cpu(0)>
a + b:
```

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

```
Out[19]: 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 [20]: z = nd.zeros_like(y)
         before = id(z)
         z[:] = x + y
         id(z) == before
```

```
Out[20]: 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 [21]: nd.elemwise_add(x, y, out=z)
         id(z) == before
```

```
Out[21]: 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 [22]: before = id(x)
         x += y
         id(x) == before
```

```
Out[22]: 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 [23]: x = nd.arange(9).reshape((3, 3))
         print('x:', x)
         x[1:3]
```

```
x:
[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]]
<NDArray 3x3 @cpu(0)>
```

```
Out[23]:
         [[ 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 [24]: x[1, 2] = 9
         x
```

```
Out[24]:
         [[ 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 [25]: x[1:2, 1:3] = 10
         x
```

```
Out[25]:
         [[  0.   1.   2.]
          [  3.  10.  10.]
          [  6.   7.   8.]]
         <NDArray 3x3 @cpu(0)>
```

2.3.6 NDArray 和 NumPy 相互转换

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 [26]: import numpy as np
         x = np.ones((2, 3))
         y = nd.array(x)  # NumPy 转换成 NDArray。
         z = y.asnumpy()  # NDArray 转换成 NumPy。
         print([z, y])
```

```
[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 中存储和转换数据的主要工具。
- 我们可以轻松地对 NDArray 进行创建、运算、指定索引和与 NumPy 之间的相互转换。

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 $y = 2\boldsymbol{x}^\top \boldsymbol{x}$ 求关于列向量 $\boldsymbol{x}$ 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 $y = 2\boldsymbol{x}^\top \boldsymbol{x}$ 关于 $\boldsymbol{x}$ 的梯度应为 $4\boldsymbol{x}$。现在我们来验证一下求出来的梯度是正确的。

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

```
x.grad:
[[  0.]
 [  4.]
 [  8.]
 [ 12.]]
<NDArray 4x1 @cpu(0)>
```

```
Out[6]:
        [[ 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

3

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 $\boldsymbol{w} = [w_1, w_2]^\top$ 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：

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

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

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

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），η 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 $x_1 w_1 + x_2 w_2 + b$ 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。

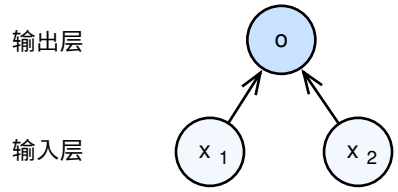


图 3.1: 线性回归是一个单层神经网络

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。由于该网络的输出为 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（dense layer 或 fully-connected 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]: 0.09804558753967285
```

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]: 0.0002799034118652344
```

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。如果我们对训练数据集中 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

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

对 3 个房屋样本估算价格的矢量计算表达式为

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

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 = nd.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

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

其中模型输出 $\boldsymbol{\hat{y}} \in \mathbb{R}^{n\times 1}$，批量数据样本特征 $\boldsymbol{X} \in \mathbb{R}^{n\times d}$，权重 $\boldsymbol{w} \in \mathbb{R}^{d\times 1}$，偏差 $b \in \mathbb{R}$。相应地，批量数据样本标签 $\boldsymbol{y} \in \mathbb{R}^{n\times 1}$。在矢量计算中，我们将两个向量 $\boldsymbol{\hat{y}}$ 和 $\boldsymbol{y}$ 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 $\boldsymbol{\theta} = [w_1, w_2, b]^\top$，本节中小批量随机梯度下降的迭代步骤将相应地改写为

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

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：

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

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，特征数为 d。给定批量数据样本的特征 $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ 和标签 $\boldsymbol{y} \in \mathbb{R}^{n \times 1}$，线性回归的批量输出 $\hat{\boldsymbol{y}} \in \mathbb{R}^{n \times 1}$ 的计算表达式为

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

其中 $\boldsymbol{w} \in \mathbb{R}^{d \times 1}$ 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 $\hat{\boldsymbol{y}}$ 和真实值 $\boldsymbol{y}$ 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]: %config InlineBackend.figure_format = 'retina'
        %matplotlib inline
        import matplotlib as mpl
        import matplotlib.pyplot as plt
        import mxnet as mx
        from mxnet import autograd, nd
        import numpy as np
        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（特征数）为 2。给定随机生成的批量样本特征 $\boldsymbol{X} \in \mathbb{R}^{1000 \times 2}$，我们使用线性回归模型真实权重 $\boldsymbol{w} = [2, -3.4]^\top$ 和偏差 $b = 4.2$，以及一个随机噪音项 ϵ 来生成标签

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

其中噪音项 ϵ 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)
```

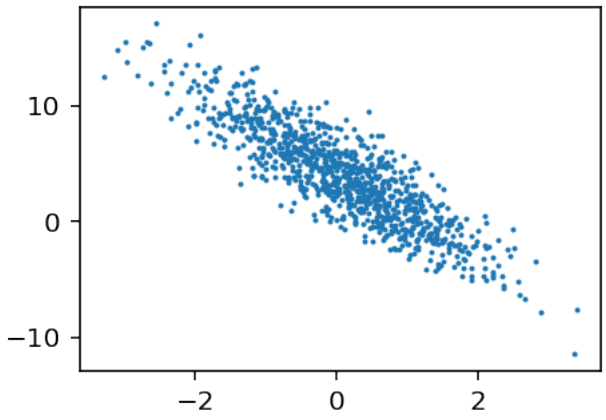
注意到 `features` 的每一行是一个长度为 2 的向量，而 `labels` 的每一行是一个长度为 1 的向量（标量）。

```
In [3]: print(features[0], labels[0])
```

```
[ 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]: def set_fig_size(mpl, figsize=(3.5, 2.5)):
            mpl.rcParams['figure.figsize'] = figsize

        set_fig_size(mpl)
        plt.scatter(features[:, 1].asnumpy(), labels.asnumpy(), 1)
        plt.show()
```



我们将该函数定义在 gluonbook 包中供后面章节调用。

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():
            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():
            print(X, y)
            break
```

```
[[-0.47335598 -1.10848832]
 [ 0.21522222  1.95657027]
 [ 0.5984059  -0.51465517]
 [-0.61270916 -2.81996584]
 [-0.86667937 -0.0170521 ]
 [ 1.61142027  0.59097415]
 [ 0.88062727  1.98851633]
 [-1.01891088 -0.55721617]
 [-1.37254357  0.0290854 ]
 [ 0.89070588  1.33772314]]
<NDArray 10x2 @cpu(0)>
[  7.02215433  -2.02247977   7.15676975  12.55443287   2.52690601
   5.41019535  -0.79267985   4.07729626   1.36435509   1.43572879]
<NDArray 10 @cpu(0)>
```

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=1, 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 net(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
```

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
         loss = squared_loss

         for epoch in range(1, num_epochs + 1):
             for X, y in data_iter():
                 with autograd.record():
                     y_hat = net(X, w, b)
                     l = loss(y_hat, 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: 0.034400
epoch 2, loss: 0.000137
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],
          [[ 1.99903178]
           [-3.3996675 ]]
          <NDArray 2x1 @cpu(0)>)
```

```
In [14]: true_b, b
```

```
Out[14]: (4.2,
          [ 4.19948435]
          <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]: %config InlineBackend.figure_format = 'retina'
        %matplotlib inline
        import mxnet as mx
        from mxnet import autograd, nd
        import numpy as np
```

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
```

```
[[-0.23041603  1.13859928]
 [-0.65207404 -1.76748502]
 [-0.05510042 -0.56671089]
 [ 0.06442814  1.63567686]
 [ 0.2963081   0.87118691]
 [ 0.3367998   0.79105479]
 [ 0.6360805  -0.58440769]
 [ 0.08167925  0.5635339 ]
 [-1.54575098  1.30501723]
 [ 2.17994285 -0.38305327]]
<NDArray 10x2 @cpu(0)>
[-0.11321159  8.89702225  6.01344442 -1.228127    1.83015716  2.19294548
  7.46077585  2.43111348 -3.32293463  9.86030197]
<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。这里我们使用默认的随机初始化方法：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。

```
In [7]: net.initialize()
```

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():
                     output = net(X)
                     l = loss(output, y)
                 l.backward()
                 trainer.step(batch_size)
             print("epoch %d, loss: %f"
                   % (epoch, loss(net(features), labels).mean().asnumpy()))
```

```
epoch 1, loss: 0.041553
epoch 2, loss: 0.000157
epoch 3, loss: 0.000050
```

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],
          [[ 1.99975872 -3.39958715]]
          <NDArray 1x2 @cpu(0)>)
```

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

```
Out[12]: (4.2,
          [ 4.19983339]
          <NDArray 1 @cpu(0)>)
```

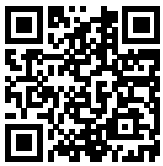
3.3.8 小结

- 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 2×2，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 2×2 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

$$
\begin{aligned}
o_1 &= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1, \\
o_2 &= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2, \\
o_3 &= x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3.
\end{aligned}
$$

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。

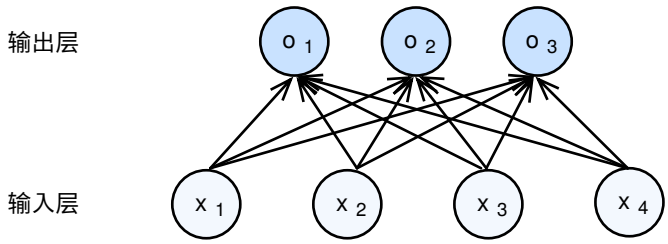


图 3.2: Softmax 回归是一个单层神经网络

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

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

设 2×2 图片样本 i 的特征为

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

输出层输出为

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

预测为狗、猫或鸡的概率分布为

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

我们对样本 i 分类的矢量计算表达式为

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

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 $\boldsymbol{X} \in \mathbb{R}^{n \times x}$，批量标签 $\boldsymbol{y} \in \mathbb{R}^{n \times 1}$。假设 Softmax 回归的权重和偏差参数分别为 $\boldsymbol{W} \in \mathbb{R}^{x \times y}, \boldsymbol{b} \in \mathbb{R}^{1 \times y}$。Softmax 回归的矢量计算表达式为

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

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

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_{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

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

其中 $\boldsymbol{\Theta}$ 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
        import matplotlib.pyplot as plt
        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:math: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。

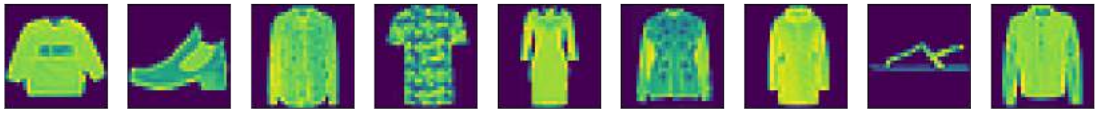
```
In [4]: def get_text_labels(labels):
            text_labels = [
                't-shirt', 'trouser', 'pullover', 'dress', 'coat',
                'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot'
            ]
            return [text_labels[int(i)] for i in labels]
```

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)
        get_text_labels(y)
```



```
Out[6]: ['pullover',
         'ankle boot',
         'shirt',
         't-shirt',
         'dress',
         'coat',
         'coat',
         'sandal',
         'coat']
```

3.5.2 读取数据

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，这里我们直接使用 `gluon.data.DataLoader`，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。

```
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)
```

注意到这里我们需要每次从训练数据里读取一个由随机样本组成的小批量，但测试数据则无需如此。

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 28×28 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 784×10 和 1×10 的矩阵。

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

        W = nd.random.normal(shape=(num_inputs, num_outputs))
        b = nd.random.normal(shape=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)>)
```

下面我们就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 = 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]: (
          [[ 0.07024596  0.38768554  0.06740031  0.21249865  0.26216954]
           [ 0.06539094  0.1500023   0.36792335  0.02470051  0.39198291]]
          <NDArray 2x5 @cpu(0)>,
          [ 1.  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_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_hat.argmax(axis=1)` 返回矩阵 `y_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_hat.argmax(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_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 / num_outputs = 0.1。

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

```
Out[19]: 0.098046875000000006
```

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)
```

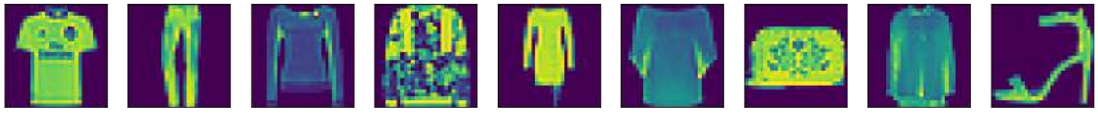
```
epoch 1, loss 3.5149, train acc 0.457, test acc 0.597
epoch 2, loss 1.8442, train acc 0.636, test acc 0.668
epoch 3, loss 1.5347, train acc 0.683, test acc 0.697
epoch 4, loss 1.3771, train acc 0.707, test acc 0.715
epoch 5, loss 1.2734, train acc 0.724, test acc 0.730
```

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.asnumpy()))
```



```
labels: ['t-shirt', 'trouser', 'pullover', 'pullover', 'dress', 'pullover', 'bag',
 ↪  'shirt', 'sandal']
predictions: ['shirt', 'trouser', 'shirt', 't-shirt', 'trouser', '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, nd
        from mxnet.gluon import nn, loss as gloss
```

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。

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

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 0.7974, train acc 0.742, test acc 0.803
epoch 2, loss 0.5747, train acc 0.810, test acc 0.820
epoch 3, loss 0.5322, train acc 0.824, test acc 0.828
epoch 4, loss 0.5069, train acc 0.830, test acc 0.834
epoch 5, loss 0.4914, train acc 0.834, test acc 0.837
```

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。

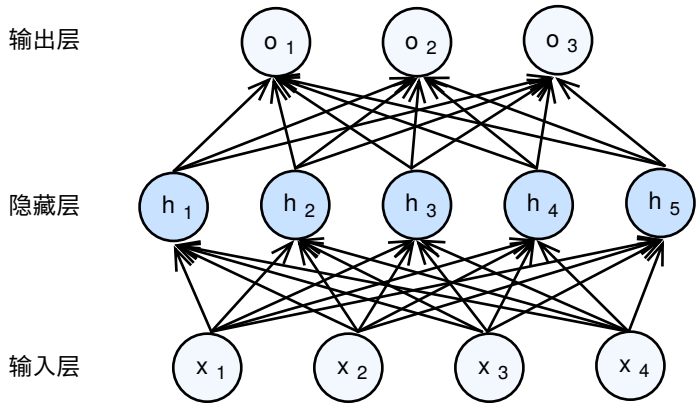


图 3.3: 带有隐藏层的多层感知机。它含有一个隐藏层，该层中有 5 个隐藏单元

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 线性转换

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 $\boldsymbol{H} \in \mathbb{R}^{n \times h}$。假设输出层的权重和偏差参数分别为 $\boldsymbol{W}_o \in \mathbb{R}^{h \times y}, \boldsymbol{b}_o \in \mathbb{R}^{1 \times y}$，多层感知机输出

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

其中的加法运算使用了广播机制，$\boldsymbol{O} \in \mathbb{R}^{n \times y}$。可见，多层感知机的输出 $\boldsymbol{O}$ 是对上一层的输出 $\boldsymbol{H}$ 的线性转换。

那么，如果隐藏层也对输入做线性转换会怎么样呢？为了便于描述这一问题，让我们暂时忽略每一层的偏差参数。设批量特征为 $\boldsymbol{X} \in \mathbb{R}^{n \times x}$，隐藏层的权重参数 $\boldsymbol{W}_h \in \mathbb{R}^{x \times h}$。假设 $\boldsymbol{H} = \boldsymbol{X}\boldsymbol{W}_h$ 且 $\boldsymbol{O} = \boldsymbol{H}\boldsymbol{W}_o$，联立两式可得 $\boldsymbol{O} = \boldsymbol{X}\boldsymbol{W}_h\boldsymbol{W}_o$：它等价于 $\boldsymbol{O} = \boldsymbol{X}\boldsymbol{W}'$，其中 $\boldsymbol{W}' = \boldsymbol{W}_h\boldsymbol{W}_o$。因此，使用线性转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添加非线性转换，这样才能使多层感知机变得有意义。我们将这些非线性转换称为激活函数（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转换。给定元素 x，该函数的输出是

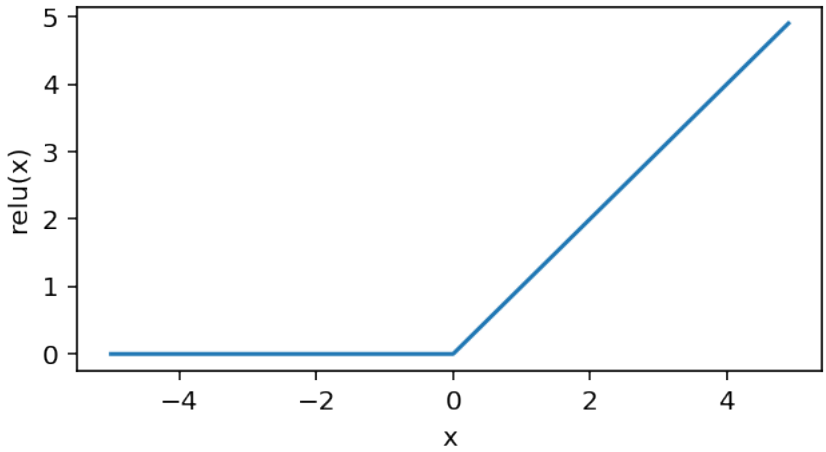
$$\text{relu}(x) = \max(x, 0).$$

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转换，让我们先导入一些包或模块。

```
In [1]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import matplotlib as mpl
        import matplotlib.pyplot as plt
        from mxnet import nd
```

下面，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转换。

```
In [2]: gb.set_fig_size(mpl, (5, 2.5))

        x = nd.arange(-5.0, 5.0, 0.1)
        plt.plot(x.asnumpy(), x.relu().asnumpy())
        plt.xlabel('x')
        plt.ylabel('relu(x)')
        plt.show()
```



Sigmoid 函数

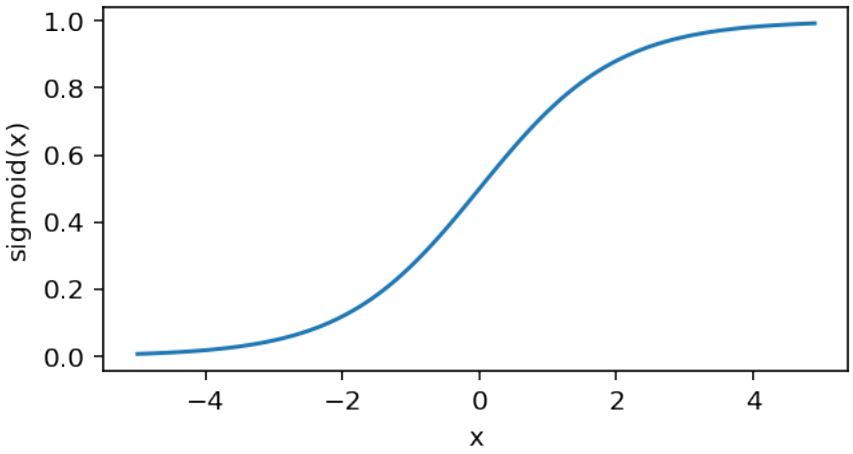
Sigmoid 函数可以将元素的值转换到 0 和 1 之间：

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

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 函数接近线性转换。

```
In [3]: plt.plot(x.asnumpy(), x.sigmoid().asnumpy())
        plt.xlabel('x')
        plt.ylabel('sigmoid(x)')
        plt.show()
```



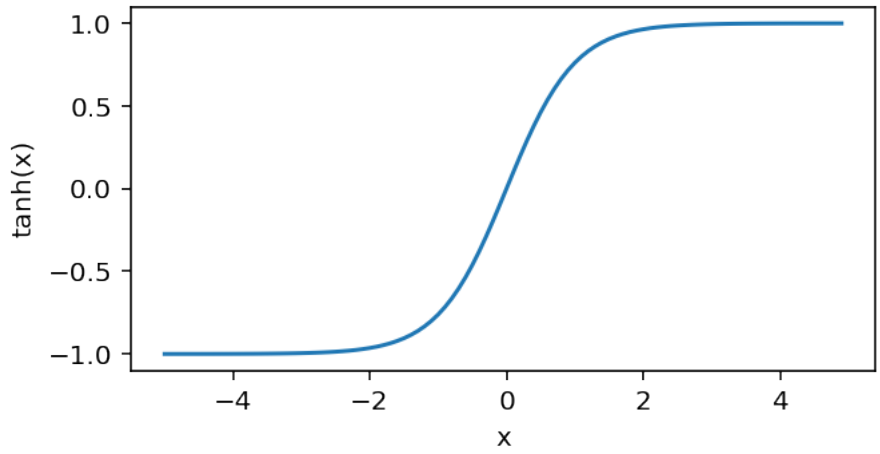
Tanh 函数

Tanh（双曲正切）函数可以将元素的值转换到-1 和 1 之间：

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

下面绘制了 tanh 函数。当元素值接近 0 时，tanh 函数接近线性转换。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]: plt.plot(x.asnumpy(), x.tanh().asnumpy())
        plt.xlabel('x')
        plt.ylabel('tanh(x)')
        plt.show()
```



下面，我们使用三种激活函数来转换输入。按元素操作后，输入和输出形状相同。

```
In [5]: X = nd.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)>,
         [[[  5.00000000e-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 $\boldsymbol{X} \in \mathbb{R}^{n \times x}$，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 ϕ。假设隐藏层的权重和偏差参数分别为 $\boldsymbol{W}_h \in \mathbb{R}^{x \times h}, \boldsymbol{b}_h \in \mathbb{R}^{1 \times h}$，输出层的权重和偏差参数分别为 $\boldsymbol{W}_o \in \mathbb{R}^{h \times y}, \boldsymbol{b}_o \in \mathbb{R}^{1 \times y}$。多层感知机的矢量计算表达式为

$$\boldsymbol{H} = \phi(\boldsymbol{X}\boldsymbol{W}_h + \boldsymbol{b}_h),$$
$$\boldsymbol{O} = \boldsymbol{H}\boldsymbol{W}_o + \boldsymbol{b}_o,$$

其中的加法运算使用了广播机制，$\boldsymbol{H} \in \mathbb{R}^{n \times h}, \boldsymbol{O} \in \mathbb{R}^{n \times y}$。在分类问题中，我们可以对输出 $\boldsymbol{O}$ 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 $\boldsymbol{O}$ 直接提供给线性回归中使用的平方损失函数。定义了损失函数后，我们使用优化算法迭代模型参数从而不断降低损失函数的值。

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 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。以图 3.3 为例，假设隐藏层使用相同的激活函数。为了便于描述，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）。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每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（我们将在后面章节中介绍迭代细节）。因此，我们需要通过随机初始化模型参数避免让每个隐藏单元做相同计算。

3.7.6 小结

- 多层感知机本质上是对输入做一系列线性和非线性的转换。
- 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。

3.7.7 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？

3.7.8 扫码直达讨论区

3.8 多层感知机——从零开始

前面我们介绍了包括线性回归和多类逻辑回归的数个模型，它们的一个共同点是全是只含有一个输入层，一个输出层。这一节我们将介绍多层神经网络，就是包含至少一个隐含层的网络。

3.8.1 数据获取

我们继续使用 FashionMNIST 数据集。

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

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

3.8.2 多层感知机

多层感知机与前面介绍的多类逻辑回归非常类似，主要的区别是我们在输入层和输出层之间插入了一个到多个隐含层。

这里我们定义一个只有一个隐含层的模型，这个隐含层输出 256 个节点。

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

        num_hiddens = 256

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

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

3.8.3 激活函数

如果我们就用线性操作符来构造多层神经网络，那么整个模型仍然只是一个线性函数。这是因为

$$\hat{y} = X \cdot W_1 \cdot W_2 = X \cdot W_3$$

这里 $W_3 = W_1 \cdot W_2$。为了让我们的模型可以拟合非线性函数，我们需要在层之间插入非线性的激活函数。这里我们使用 ReLU

$$\mathrm{rel}u(x) = \max(x, 0)$$

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

3.8.4 定义模型

我们的模型就是将层（全连接）和激活函数（Relu）串起来：

```
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 Softmax 和交叉熵损失函数

在多类 Logistic 回归里我们提到分开实现 Softmax 和交叉熵损失函数可能导致数值不稳定。这里我们直接使用 Gluon 提供的函数

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

3.8.6 训练

训练跟之前一样。

```
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 0.7820, train acc 0.710, test acc 0.818
epoch 2, loss 0.4863, train acc 0.819, test acc 0.827
epoch 3, loss 0.4289, train acc 0.841, test acc 0.848
epoch 4, loss 0.3900, train acc 0.856, test acc 0.864
epoch 5, loss 0.3732, train acc 0.862, test acc 0.859
```

3.8.7 小结

可以看到，加入一个隐含层后我们将精度提升了不少。

3.8.8 练习

- 尝试改变 num_hiddens 来控制模型的复杂度
- 尝试加入一个新的隐含层

3.8.9 扫码直达讨论区

3.9 多层感知机——使用 Gluon

我们只需要稍微改动多类 Logistic 回归来实现多层感知机。

3.9.1 定义模型

唯一的区别在这里，我们加了一行进来。

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

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

3.9.2 读取数据并训练

```
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 0.7192, train acc 0.737, test acc 0.822
epoch 2, loss 0.4659, train acc 0.827, test acc 0.853
epoch 3, loss 0.4106, train acc 0.849, test acc 0.865
epoch 4, loss 0.3828, train acc 0.858, test acc 0.873
epoch 5, loss 0.3565, train acc 0.869, test acc 0.870
```

3.9.3 小结

通过 Gluon 我们可以更方便地构造多层神经网络。

3.9.4 练习

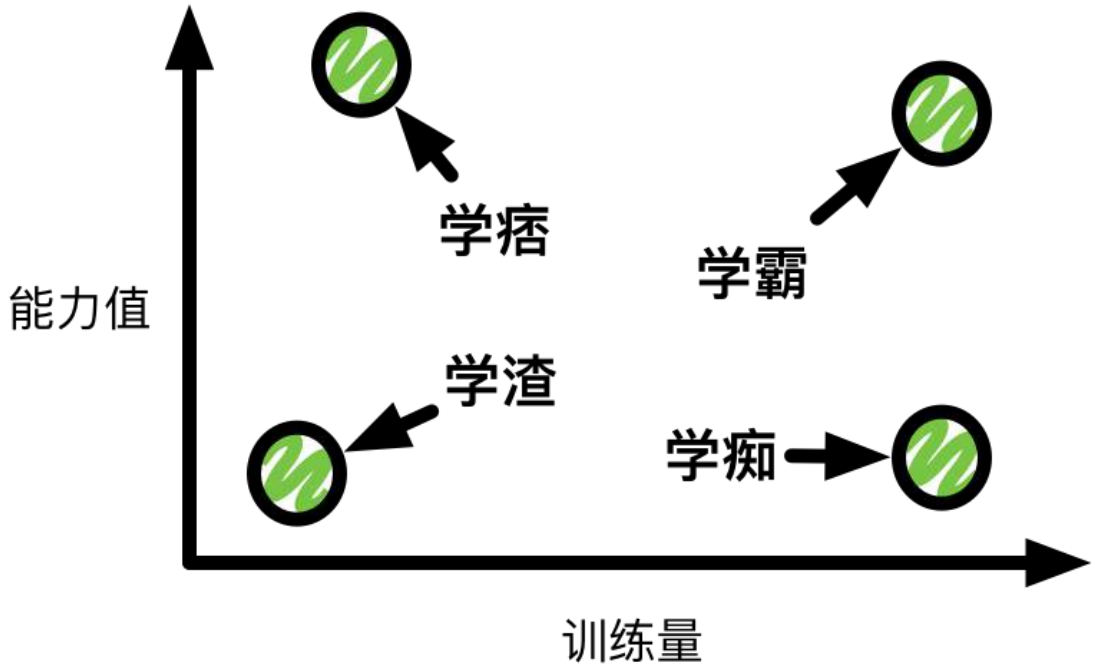
- 尝试多加入几个隐含层，对比从 0 开始的实现。
- 尝试使用一个另外的激活函数，可以使用 `help(nd.Activation)` 或者线上文档查看提供的选项。

3.9.5 扫码直达讨论区

3.10 欠拟合和过拟合

你有没有类似这样的体验？考试前突击背了模拟题的答案，模拟题随意秒杀。但考试时出的题即便和模拟题相关，只要不是原题依然容易考挂。换种情况来说，如果考试前通过自己的学习能力从模拟题的答案里总结出一个比较通用的解题套路，考试时碰到这些模拟题的变种更容易答对。

有人曾依据这种现象对学生群体简单粗暴地做了如下划分：



这里简要总结上图中学生的特点：

- 学渣：能力不行，也不认真做作业，容易考挂
- 学痞：能力不错，但喜欢裸奔，但还是可能考得比学渣好
- 学痴：能力不行，但贵在认真，考不赢学霸但秒掉学渣毫无压力
- 学霸：有能力，而且卖力，考完后喜大普奔

（现在问题来了，学酥应该在上图的哪里?）

学生的考试成绩和看起来与自身的训练量以及自身的学习能力有关。但即使是在科技进步的今天，我们依然没有完全知悉人类大脑学习的所有奥秘。的确，依赖数据训练的机器学习和人脑学习不一定完全相同。但有趣的是，机器学习模型也可能由于自身不同的训练量和不同的学习能力而产生不同的测试效果。为了科学地阐明这个现象，我们需要从若干机器学习的重要概念开始讲解。

3.10.1 训练误差（模考成绩）和泛化误差（考试成绩）

在实践中，机器学习模型通常在训练数据集上训练并不断调整模型里的参数。之后，我们通常把训练得到的模型在一个区别于训练数据集的测试数据集上测试，并根据测试结果评价模型的好坏。机器学习模型在训练数据集上表现出的误差叫做**训练误差**，在任意一个测试数据样本上表现出的误差的期望值叫做**泛化误差**。

训练误差和泛化误差的计算可以利用我们之前提到的损失函数，例如线性回归里用到的平方误差和多类逻辑回归里用到的交叉熵损失函数。

之所以要了解训练误差和泛化误差，是因为统计学习理论基于这两个概念可以科学解释本节教程一开始提到的模型不同的测试效果。我们知道，理论的研究往往需要基于一些假设。而统计学习理论的一个假设是：

> 训练数据集和测试数据集里的每一个数据样本都是从同一个概率分布中相互独立地生成出的（独立同分布假设）。

基于以上独立同分布假设，给定任意一个机器学习模型及其参数，它的训练误差的期望值和泛化误差都是一样的。然而从之前的章节中我们了解到，在机器学习的过程中，模型的参数并不是事先给定的，而是通过训练数据学习得出的：模型的参数在训练中使训练误差不断降低。所以，如果模型参数是通过训练数据学习得出的，那么训练误差的期望值无法高于泛化误差。换句话说，通常情况下，由训练数据学到的模型参数会使模型在训练数据上的表现不差于在测试数据上的表现。

因此，一个重要结论是：

> 训练误差的降低不一定意味着泛化误差的降低。机器学习既需要降低训练误差，又需要降低泛化误差。

3.10.2 欠拟合和过拟合

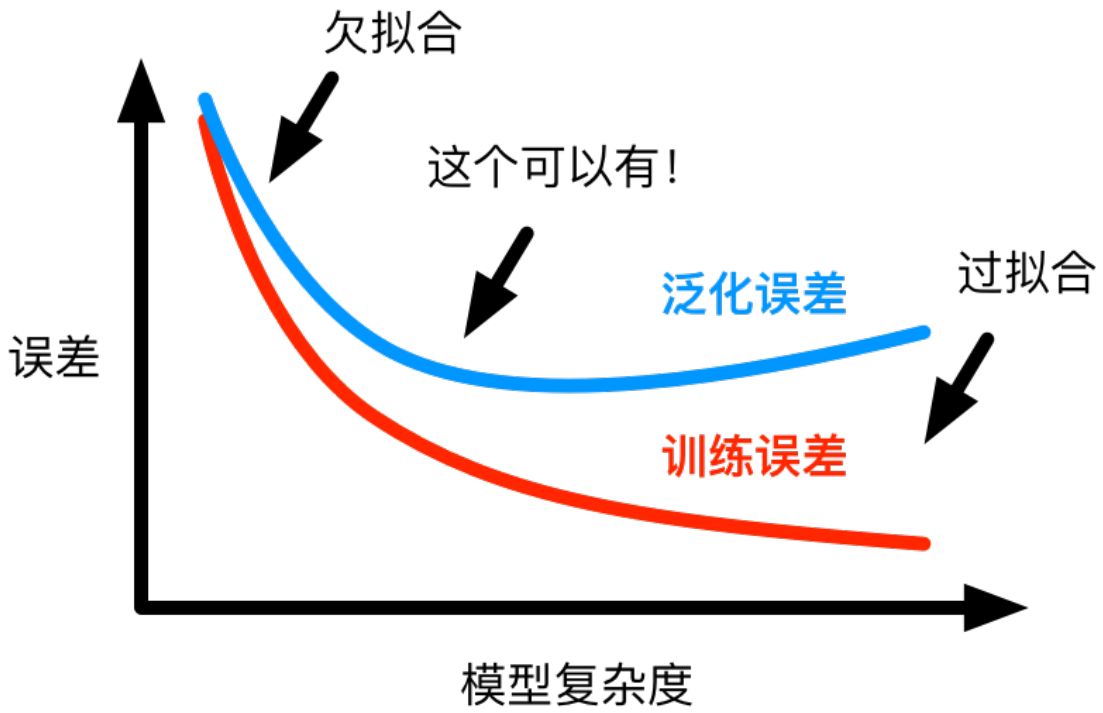
实践中，如果测试数据集是给定的，我们通常用机器学习模型在该测试数据集的误差来反映泛化误差。基于上述重要结论，以下两种拟合问题值得注意：

- **欠拟合**：机器学习模型无法得到较低训练误差。
- **过拟合**：机器学习模型的训练误差远小于其在测试数据集上的误差。

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：模型的选择和训练数据集的大小。

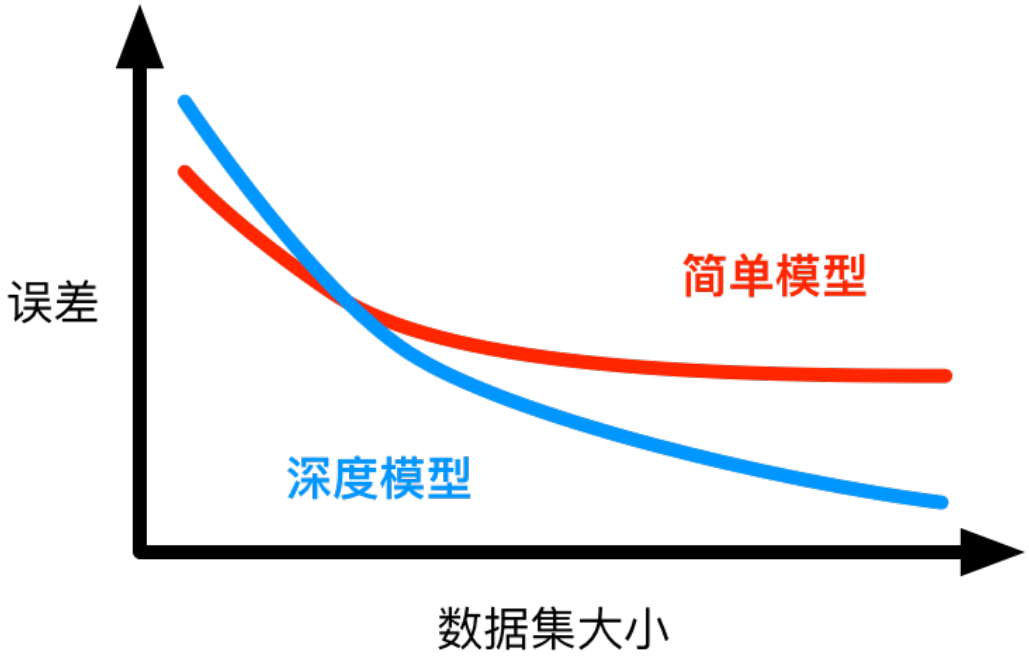
模型的选择

在本节的开头，我们提到一个学生可以有特定的学习能力。类似地，一个机器学习模型也有特定的拟合能力。拿多项式函数举例，一般来说，高阶多项式函数（拟合能力较强）比低阶多项式函数（拟合能力较弱）更容易在相同的训练数据集上得到较低的训练误差。需要指出的是，给定数据集，过低拟合能力的模型更容易欠拟合，而过高拟合能力的模型更容易过拟合。模型拟合能力和误差之间的关系如下图。



训练数据集的大小

在本节的开头，我们同样提到一个学生可以有特定的训练量。类似地，一个机器学习模型的训练数据集的样本数也可大可小。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。除此之外，泛化误差不会随训练数据集里样本数量增加而增大。



为了理解这两个因素对拟合和过拟合的影响，下面让我们来动手学习。

3.10.3 多项式拟合

我们以多项式拟合为例。给定一个**标量**数据点集合 `x` 和对应的标量目标值 `y`，多项式拟合的目标是找一个 K 阶多项式，其由向量 `w` 和位移 `b` 组成，来最好地近似每个样本 `x` 和 `y`。用数学符号来表示就是我们将学 `w` 和 `b` 来预测

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

并以平方误差为损失函数。特别地，一阶多项式拟合又叫线性拟合。

创建数据集

这里我们使用一个人工数据集来把事情弄简单些，因为这样我们将知道真实的模型是什么样的。具体来说我们使用如下的二阶多项式来生成每一个数据样本

$$y = 1.2x - 3.4x^2 + 5.6x^3 + 5.0 + \text{noise}$$

这里噪音服从均值 0 和标准差为 0.1 的正态分布。

需要注意的是，我们用以上相同的数据生成函数来生成训练数据集和测试数据集。两个数据集的样本数都是 100。

```
In [1]: from mxnet import ndarray as nd
        from mxnet import autograd
        from mxnet import gluon

        num_train = 100
        num_test = 100
        true_w = [1.2, -3.4, 5.6]
        true_b = 5.0
```

下面生成数据集。

```
In [2]: x = nd.random.normal(shape=(num_train + num_test, 1))
        X = nd.concat(x, nd.power(x, 2), nd.power(x, 3))
        y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_w[2] * X[:, 2] + true_b
        y += .1 * nd.random.normal(shape=y.shape)

        ('x:', x[:5], 'X:', X[:5], 'y:', y[:5])
```

```
Out[2]: ('x:',
         [[ 2.21220636]
          [ 0.7740038 ]
          [ 1.04344046]
          [ 1.18392551]
          [ 1.89171135]]
         <NDArray 5x1 @cpu(0)>, 'X:',
         [[  2.21220636   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)>, 'y:',
         [ 51.6748848    6.3585763    8.94907284  11.09345436  33.03696442]
         <NDArray 5 @cpu(0)>)
```

定义训练和测试步骤

我们定义一个训练和测试的函数，这样在跑不同的实验时不需要重复实现相同的步骤。

以下的训练步骤在使用 Gluon 的线性回归有过详细描述。这里不再赘述。

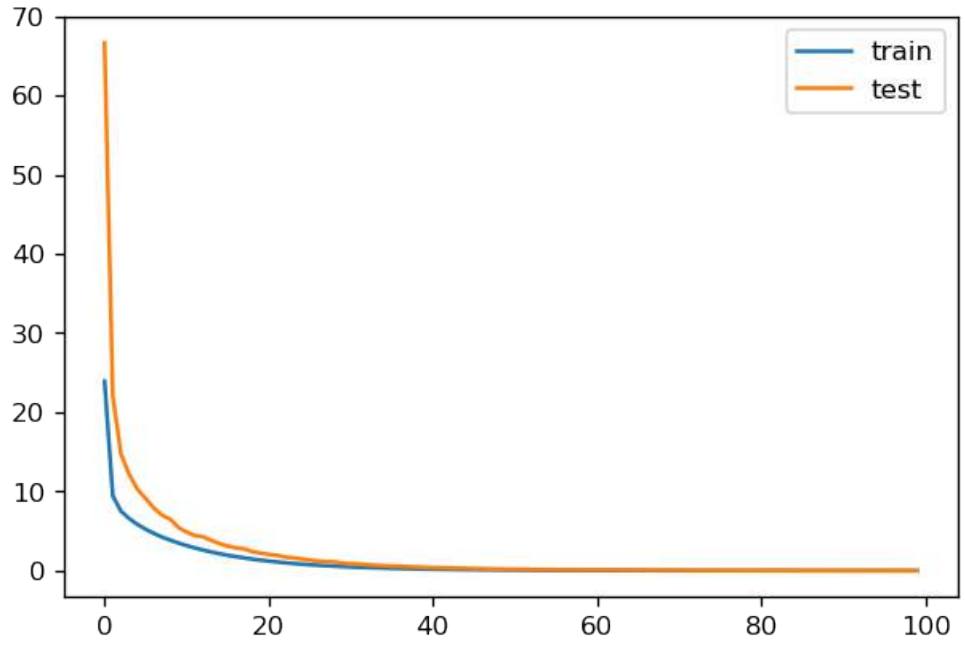
```
In [3]: %matplotlib inline
        import matplotlib as mpl
        mpl.rcParams['figure.dpi']= 120
        import matplotlib.pyplot as plt

        def train(X_train, X_test, y_train, y_test):
            # 线性回归模型
            net = gluon.nn.Sequential()
            with net.name_scope():
                net.add(gluon.nn.Dense(1))
            net.initialize()
            # 设一些默认参数
            learning_rate = 0.01
            epochs = 100
            batch_size = min(10, y_train.shape[0])
            dataset_train = gluon.data.ArrayDataset(X_train, y_train)
            data_iter_train = gluon.data.DataLoader(
                dataset_train, batch_size, shuffle=True)
            # 默认 SGD 和均方误差
            trainer = gluon.Trainer(net.collect_params(), 'sgd', {
                'learning_rate': learning_rate})
            square_loss = gluon.loss.L2Loss()
            # 保存训练和测试损失
            train_loss = []
            test_loss = []
            for e in range(epochs):
                for data, label in data_iter_train:
                    with autograd.record():
                        output = net(data)
                        loss = square_loss(output, label)
                    loss.backward()
                    trainer.step(batch_size)
                train_loss.append(square_loss(
                    net(X_train), y_train).mean().asscalar())
                test_loss.append(square_loss(
                    net(X_test), y_test).mean().asscalar())
            # 打印结果
            plt.plot(train_loss)
```

```
    plt.plot(test_loss)
    plt.legend(['train','test'])
    plt.show()
    return ('learned weight', net[0].weight.data(),
            'learned bias', net[0].bias.data())
```

三阶多项式拟合（正常）

我们先使用与数据生成函数同阶的三阶多项式拟合。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 [4]: train(X[:num_train, :], X[num_train:, :], y[:num_train], y[num_train:])
```

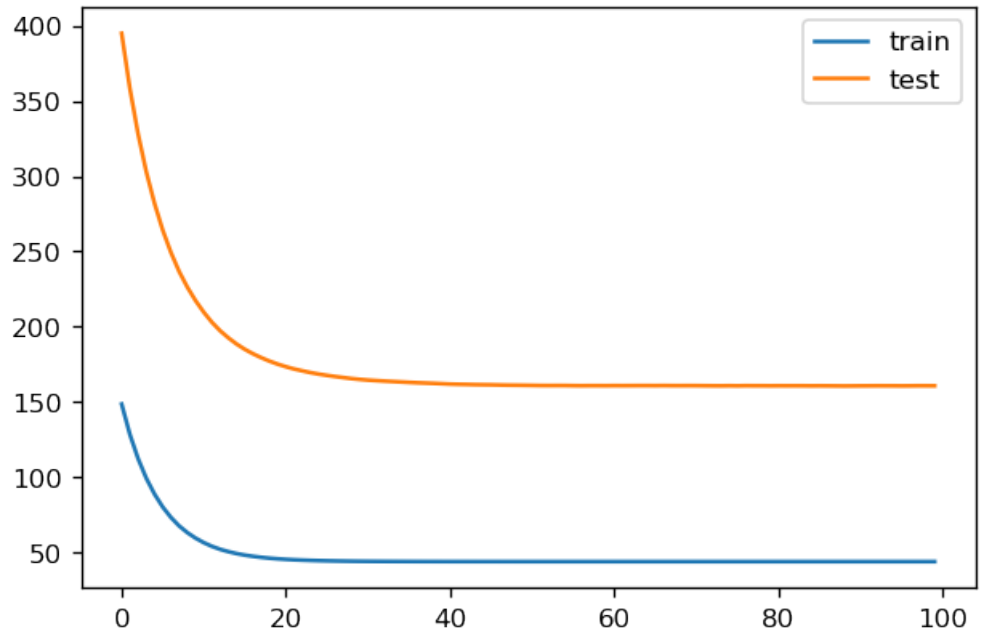



```
Out[4]: ('learned weight',
         [[ 1.32664704 -3.36352539  5.56294584]]
         <NDArray 1x3 @cpu(0)>, 'learned bias',
         [ 4.95119953]
         <NDArray 1 @cpu(0)>)
```

线性拟合（欠拟合）

我们再试试线性拟合。很明显，该模型的训练误差很高。线性模型在非线性模型（例如三阶多项式）生成的数据集上容易欠拟合。

```
In [5]: train(x[:num_train, :], x[num_train:, :], y[:num_train], y[num_train:])
```

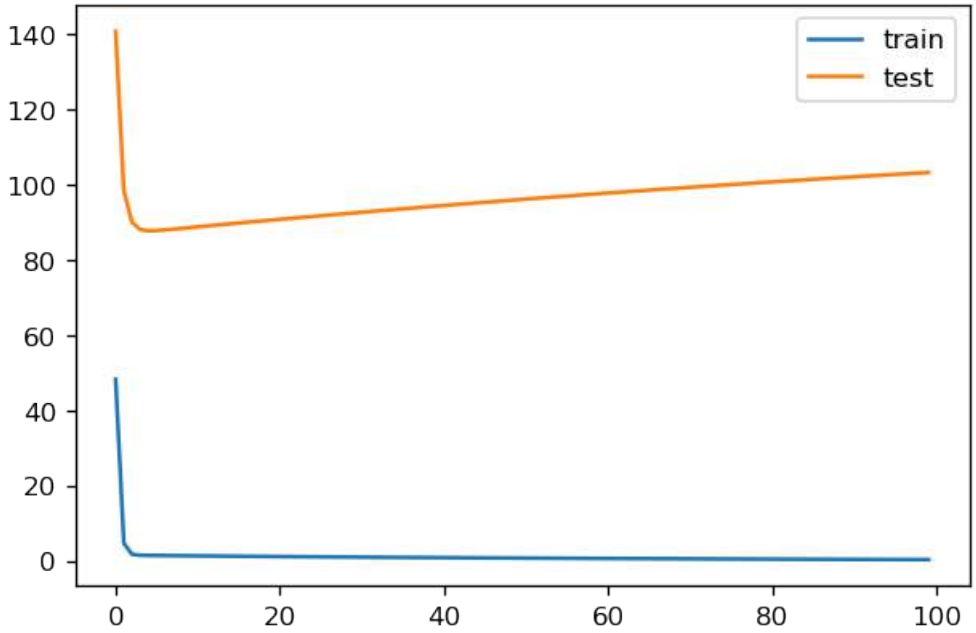



```
Out[5]: ('learned weight',
         [[ 15.56328011]]
         <NDArray 1x1 @cpu(0)>, 'learned bias',
         [ 2.29803348]
         <NDArray 1 @cpu(0)>)
```

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训练样本来训练。很显然，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集中的噪音影响。在机器学习过程中，即便训练误差很低，但是测试数据集上的误差很高。这是典型的过拟合现象。

```
In [6]: train(X[0:2, :], X[num_train:, :], y[0:2], y[num_train:])
```




```
Out[6]: ('learned weight',
         [[ 1.38723636  1.93765891  3.50859237]]
         <NDArray 1x3 @cpu(0)>, 'learned bias',
         [ 1.23128557]
         <NDArray 1 @cpu(0)>)
```

我们还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。

3.10.4 小结

- 训练误差的降低并不一定意味着泛化误差的降低。
- 欠拟合和过拟合都是需要尽量避免的。我们要注意模型的选择和训练量的大小。

3.10.5 练习

- 学渣、学痞、学痴和学霸对应的模型复杂度、训练量、训练误差和泛化误差分别是怎样的？
- 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 1000 个样本的训练误差的期望降到 0，为什么？

3.10.6 扫码直达讨论区

3.11 正则化——从零开始

本章从 0 开始介绍如何的正则化来应对过拟合问题。

3.11.1 高维线性回归

我们使用高维线性回归为例来引入一个过拟合问题。

具体来说我们使用如下的线性函数来生成每一个数据样本

$$y = 0.05 + \sum_{i=1}^{p} 0.01x_i + \text{noise}$$

这里噪音服从均值 0 和标准差为 0.01 的正态分布。

需要注意的是，我们用以上相同的数据生成函数来生成训练数据集和测试数据集。为了观察过拟合，我们特意把训练数据样本数设低，例如 $n = 20$，同时把维度升高，例如 $p = 200$。

```
In [1]: from mxnet import ndarray as nd
        from mxnet import autograd
        from mxnet import gluon
        import mxnet as mx

        num_train = 20
        num_test = 100
        num_inputs = 200
```

3.11.2 生成数据集

这里定义模型真实参数。

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

我们接着生成训练和测试数据集。

```
In [3]: X = nd.random.normal(shape=(num_train + num_test, num_inputs))
        y = nd.dot(X, true_w) + true_b
        y += .01 * nd.random.normal(shape=y.shape)

        X_train, X_test = X[:num_train, :], X[num_train:, :]
        y_train, y_test = y[:num_train], y[num_train:]
```

当我们开始训练神经网络的时候，我们需要不断读取数据块。这里我们定义一个函数它每次返回 `batch_size` 个随机的样本和对应的目标。我们通过 python 的 `yield` 来构造一个迭代器。

```
In [4]: import random
        batch_size = 1
        def data_iter(num_examples):
            idx = list(range(num_examples))
            random.shuffle(idx)
            for i in range(0, num_examples, batch_size):
                j = nd.array(idx[i:min(i+batch_size,num_examples)])
                yield X.take(j), y.take(j)
```

3.11.3 初始化模型参数

下面我们随机初始化模型参数。之后训练时我们需要对这些参数求导来更新它们的值，所以我们需要创建它们的梯度。

```
In [5]: 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
```

3.11.4 L_2 范数正则化

这里我们引入 L_2 范数正则化。不同于在训练时仅仅最小化损失函数 (Loss)，我们在训练时其实在最小化

$$\text{loss} + \lambda \sum_{p \in \text{params}} \|p\|_2^2$$

直观上，L_2 范数正则化试图惩罚较大绝对值的参数值。下面我们定义 L2 正则化。注意有些时候大家对偏移加罚，有时候不加罚。通常结果上两者区别不大。这里我们演示对偏移也加罚的情况：

```
In [6]: def L2_penalty(w, b):
            return ((w**2).sum() + b**2) / 2
```

3.11.5 定义训练和测试

下面我们定义剩下的所需要的函数。这个跟之前的教程大致一样，主要是区别在于计算 `loss` 的时候我们加上了 L2 正则化，以及我们将训练和测试损失都画了出来。

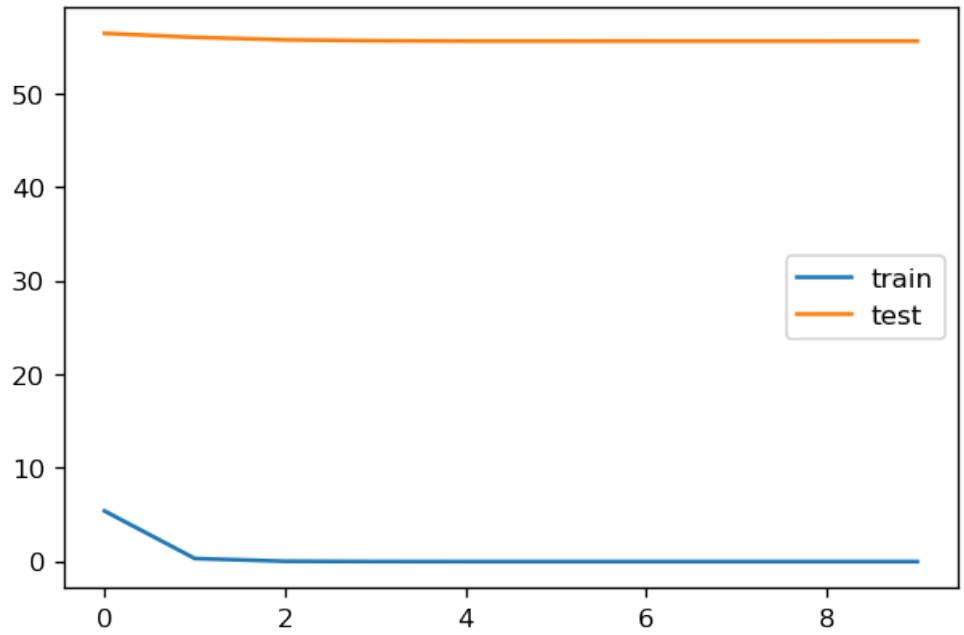
```
In [7]: %matplotlib inline
        import matplotlib as mpl
        mpl.rcParams['figure.dpi']= 120
        import matplotlib.pyplot as plt
        import numpy as np

        def net(X, w, b):
            return nd.dot(X, w) + b

        def square_loss(yhat, y):
            return (yhat - y.reshape(yhat.shape)) ** 2 / 2

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

        def test(net, params, X, y):
            return square_loss(net(X, *params), y).mean().asscalar()
            #return np.mean(square_loss(net(X, *params), y).asnumpy())

        def train(lambd):
            epochs = 10
            learning_rate = 0.005
```

```
    w, b = params = init_params()
    train_loss = []
    test_loss = []
    for e in range(epochs):
        for data, label in data_iter(num_train):
            with autograd.record():
                output = net(data, *params)
                loss = square_loss(
                    output, label) + lambd * L2_penalty(*params)
            loss.backward()
            sgd(params, learning_rate, batch_size)
        train_loss.append(test(net, params, X_train, y_train))
        test_loss.append(test(net, params, X_test, y_test))
    plt.plot(train_loss)
    plt.plot(test_loss)
    plt.legend(['train', 'test'])
    plt.show()
    return 'learned w[:10]:', w[:10].T, 'learned b:', b
```

3.11.6 观察过拟合

接下来我们训练并测试我们的高维线性回归模型。注意这时我们并未使用正则化。

```
In [8]: train(0)
```




```
Out[8]: ('learned w[:10]:',
         [[-0.81826591 -0.22133309 -0.74775428 -2.26074386 -0.10052045 -0.69787067
           -0.09946337  0.61101794 -1.10576057 -1.45006204]]
         <NDArray 1x10 @cpu(0)>, 'learned b:',
         [-0.38759825]
         <NDArray 1 @cpu(0)>)
```

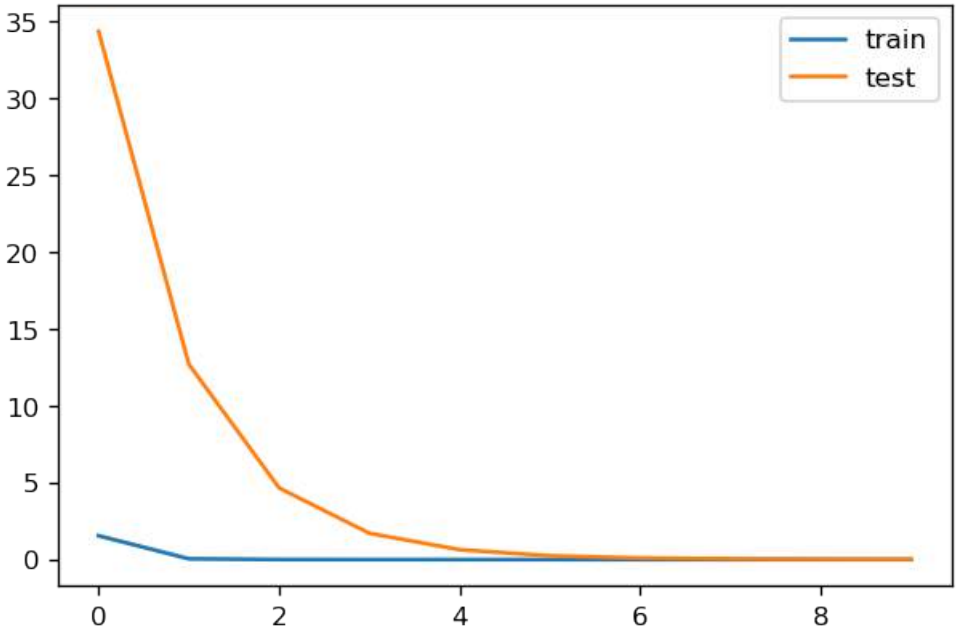
即便训练误差可以达到 0.000000，但是测试数据集上的误差很高。这是典型的过拟合现象。

观察学习的参数。事实上，大部分学到的参数的绝对值比真实参数的绝对值要大一些。

3.11.7 使用正则化

下面我们重新初始化模型参数并设置一个正则化参数。

```
In [9]: train(5)
```




```
Out[9]: ('learned w[:10]:',
         [[ 0.00606788  0.00212744 -0.00183699  0.00365135 -0.00137779  0.00371153
            0.00328525 -0.00041245  0.00434453  0.00162885]]
         <NDArray 1x10 @cpu(0)>, 'learned b:',
         [ 0.00741734]
         <NDArray 1 @cpu(0)>)
```

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。过拟合现象得到缓解。但打印出的学到的参数依然不是很理想，这主要是因为我们训练数据的样本相对维度来说太少。

3.11.8 小结

- 我们可以使用正则化来应对过拟合问题。

3.11.9 练习

- 除了正则化、增大训练量、以及使用合适的模型，你觉得还有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？

3.11.10 扫码直达讨论区

3.12 正则化——使用 Gluon

本章介绍如何使用 Gluon 的正则化来应对过拟合问题。

3.12.1 高维线性回归数据集

我们使用与上一节相同的高维线性回归为例来引入一个过拟合问题。

```
In [1]: from mxnet import ndarray as nd
        from mxnet import autograd
        from mxnet import gluon
        import mxnet as mx

        num_train = 20
        num_test = 100
        num_inputs = 200

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

        X = nd.random.normal(shape=(num_train + num_test, num_inputs))
        y = nd.dot(X, true_w) + true_b
        y += .01 * nd.random.normal(shape=y.shape)

        X_train, X_test = X[:num_train, :], X[num_train:, :]
        y_train, y_test = y[:num_train], y[num_train:]
```

3.12.2 定义训练和测试

跟前一样定义训练模块。你也许发现了主要区别，`Trainer` 有一个新参数 `wd`。我们通过优化算法的 `wd` 参数 (weight decay) 实现对模型的正则化。这相当于 L_2 范数正则化。

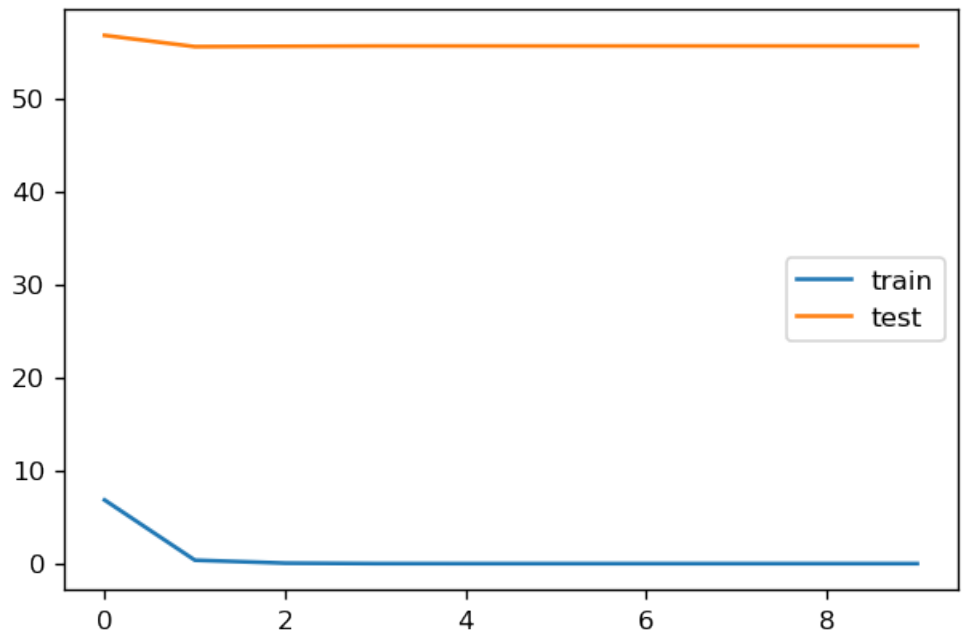
```
In [2]: %matplotlib inline
        import matplotlib as mpl
        mpl.rcParams['figure.dpi']= 120
        import matplotlib.pyplot as plt
        import numpy as np

        batch_size = 1
        dataset_train = gluon.data.ArrayDataset(X_train, y_train)
        data_iter_train = gluon.data.DataLoader(dataset_train, batch_size,
 ↪  shuffle=True)

        square_loss = gluon.loss.L2Loss()

        def test(net, X, y):
            return square_loss(net(X), y).mean().asscalar()

        def train(weight_decay):
            epochs = 10
            learning_rate = 0.005
            net = gluon.nn.Sequential()
            with net.name_scope():
                net.add(gluon.nn.Dense(1))
            net.collect_params().initialize(mx.init.Normal(sigma=1))

            # 注意到这里 'wd'
            trainer = gluon.Trainer(net.collect_params(), 'sgd', {
                'learning_rate': learning_rate, 'wd': weight_decay})

            train_loss = []
            test_loss = []
            for e in range(epochs):
                for data, label in data_iter_train:
                    with autograd.record():
                        output = net(data)
                        loss = square_loss(output, label)
                    loss.backward()
                    trainer.step(batch_size)
                train_loss.append(test(net, X_train, y_train))
```

```
            test_loss.append(test(net, X_test, y_test))
        plt.plot(train_loss)
        plt.plot(test_loss)
        plt.legend(['train','test'])
        plt.show()

        return ('learned w[:10]:', net[0].weight.data()[:,:10],
                'learned b:', net[0].bias.data())
```

训练模型并观察过拟合

接下来我们训练并测试我们的高维线性回归模型。

```
In [3]: train(0)
```




```
Out[3]: ('learned w[:10]:',
         [[-0.81826538 -0.22134091 -0.74775845 -2.26073837 -0.10052742 -0.69786811
           -0.09945718  0.61101633 -1.10575712 -1.45006013]]
         <NDArray 1x10 @cpu(0)>, 'learned b:',
         [-0.38759589]
         <NDArray 1 @cpu(0)>)
```

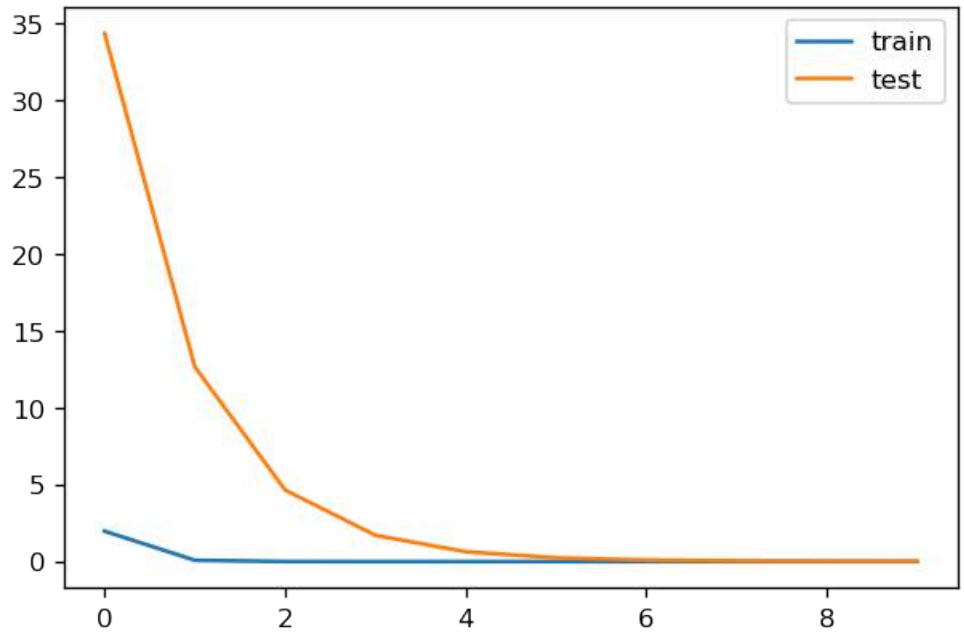
即便训练误差可以达到 0.000000，但是测试数据集上的误差很高。这是典型的过拟合现象。

观察学习的参数。事实上，大部分学到的参数的绝对值比真实参数的绝对值要大一些。

3.12.3 使用 Gluon 的正则化

下面我们重新初始化模型参数并在 Trainer 里设置一个 wd 参数。

In [4]: train(5)




```
Out[4]: ('learned w[:10]:',
         [[ 0.00574103  0.00315972 -0.00186996  0.00328838 -0.00029327  0.00284444
            0.00322134 -0.00050331  0.00435426  0.00218251]]
         <NDArray 1x10 @cpu(0)>, 'learned b:',
         [ 0.00740844]
         <NDArray 1 @cpu(0)>)
```

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。过拟合现象得到缓解。但打印出的学到的参数依然不是很理想，这主要是因为我们训练数据的样本相对维度来说太少。

3.12.4 小结

- 使用 Gluon 的 weight decay 参数可以很容易地使用正则化来应对过拟合问题。

3.12.5 练习

- 如何从字面正确理解 weight decay 的含义？它为何相当于 L_2 范式正则化？

3.12.6 扫码直达讨论区

3.13 丢弃法（Dropout）——从零开始

前面我们介绍了多层神经网络，就是包含至少一个隐含层的网络。我们也介绍了正则法来应对过拟合问题。在深度学习中，一个常用的应对过拟合问题的方法叫做丢弃法（Dropout）。本节以多层神经网络为例，从 0 开始介绍丢弃法。

由于丢弃法的概念和实现非常容易，在本节中，我们先介绍丢弃法的概念以及它在现代神经网络中是如何实现的。然后我们一起探讨丢弃法的本质。

3.13.1 丢弃法的概念

在现代神经网络中，我们所指的丢弃法，通常是对输入层或者隐含层做以下操作：

- 随机选择一部分该层的输出作为丢弃元素；
- 把丢弃元素乘以 0；
- 把非丢弃元素拉伸。

3.13.2 丢弃法的实现

丢弃法的实现很容易,例如像下面这样。这里的标量 drop_probability 定义了一个 X(NDArray 类）中任何一个元素被丢弃的概率。

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

        def dropout(X, drop_probability):
            keep_probability = 1 - drop_probability
            assert 0 <= keep_probability <= 1
            # 这种情况下把全部元素都丢弃。
            if keep_probability == 0:
                return X.zeros_like()

            # 随机选择一部分该层的输出作为丢弃元素。
            mask = nd.random.uniform(
                0, 1.0, X.shape, ctx=X.context) < keep_probability
            # 保证 E[dropout(X)] == X
            scale =  1 / keep_probability
            return mask * X * scale
```

我们运行几个实例来验证一下。

```
In [2]: A = nd.arange(20).reshape((5,4))
        dropout(A, 0.0)

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(A, 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(A, 1.0)
```

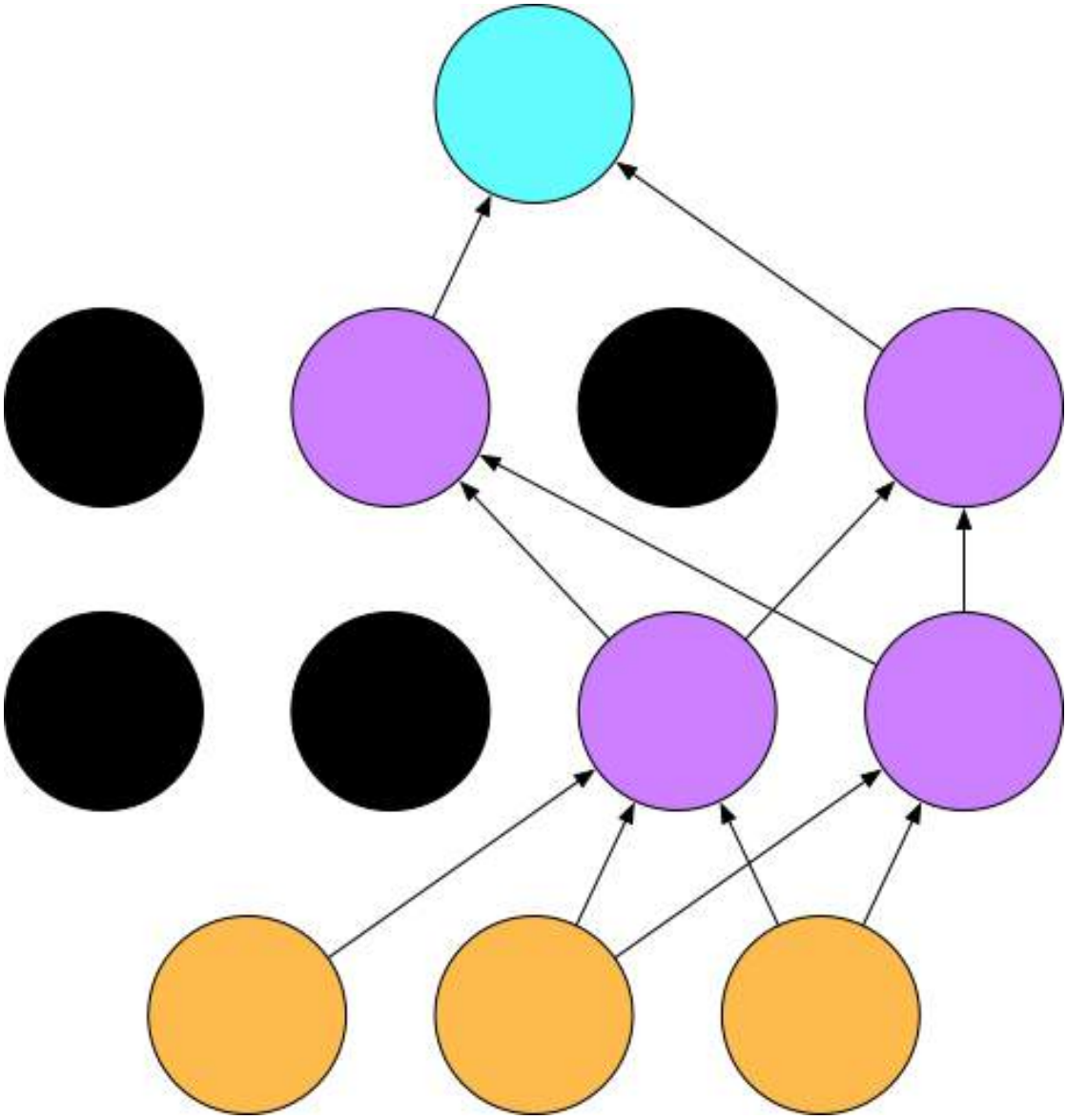
```
Out[4]:
        [[ 0.  0.  0.  0.]
         [ 0.  0.  0.  0.]
         [ 0.  0.  0.  0.]
         [ 0.  0.  0.  0.]
         [ 0.  0.  0.  0.]]
        <NDArray 5x4 @cpu(0)>
```

3.13.3 丢弃法的本质

了解了丢弃法的概念与实现，那你可能对它的本质产生了好奇。

如果你了解集成学习，你可能知道它在提升弱分类器准确率上的威力。一般来说，在集成学习里，我们可以对训练数据集有放回地采样若干次并分别训练若干个不同的分类器；测试时，把这些分类器的结果集成一下作为最终分类结果。

事实上，丢弃法在模拟集成学习。试想，一个使用了丢弃法的多层神经网络本质上是原始网络的子集（节点和边）。举个例子，它可能长这个样子。



我们在之前的章节里介绍过随机梯度下降算法：我们在训练神经网络模型时一般随机采样一个批量的训练数据。丢弃法实质上是对每一个这样的数据集分别训练一个原神经网络子集的分类器。与一般的集成学习不同，这里每个原神经网络子集的分类器用的是同一套参数。因此丢弃法只是在模拟集成学习。

我们刚刚强调了，原神经网络子集的分类器在不同的训练数据批量上训练并使用同一套参数。因

此，使用丢弃法的神经网络实质上是对输入层和隐含层的参数做了正则化：学到的参数使得原神经网络不同子集在训练数据上都尽可能表现良好。

下面我们动手实现一下在多层神经网络里加丢弃层。

3.13.4 数据获取

我们继续使用 FashionMNIST 数据集。

```
In [5]: import sys
        sys.path.append('..')
        import gluonbook as gb
        batch_size = 256
        train_iter, test_iter = gb.load_data_fashion_mnist(batch_size)
```

3.13.5 含两个隐藏层的多层感知机

多层感知机已经在之前章节里介绍。与之前章节不同，这里我们定义一个包含两个隐含层的模型，两个隐含层都输出 256 个节点。我们定义激活函数 Relu 并直接使用 Gluon 提供的交叉熵损失函数。

```
In [6]: num_inputs = 28*28
        num_outputs = 10

        num_hidden1 = 256
        num_hidden2 = 256
        weight_scale = .01

        W1 = nd.random_normal(shape=(num_inputs, num_hidden1), scale=weight_scale)
        b1 = nd.zeros(num_hidden1)

        W2 = nd.random_normal(shape=(num_hidden1, num_hidden2), scale=weight_scale)
        b2 = nd.zeros(num_hidden2)

        W3 = nd.random_normal(shape=(num_hidden2, num_outputs), scale=weight_scale)
        b3 = nd.zeros(num_outputs)

        params = [W1, b1, W2, b2, W3, b3]

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

3.13.6 定义包含丢弃层的模型

我们的模型就是将层（全连接）和激活函数（Relu）串起来，并在应用激活函数后添加丢弃层。每个丢弃层的元素丢弃概率可以分别设置。一般情况下，我们推荐把更靠近输入层的元素丢弃概率设的更小一点。这个试验中，我们把第一层全连接后的元素丢弃概率设为 0.2，把第二层全连接后的元素丢弃概率设为 0.5。

```
In [7]: drop_prob1 = 0.2
        drop_prob2 = 0.5

        def net(X):
            X = X.reshape((-1, num_inputs))
            # 第一层全连接。
            h1 = nd.relu(nd.dot(X, W1) + b1)
            # 在第一层全连接后添加丢弃层。
            h1 = dropout(h1, drop_prob1)
            # 第二层全连接。
            h2 = nd.relu(nd.dot(h1, W2) + b2)
            # 在第二层全连接后添加丢弃层。
            h2 = dropout(h2, drop_prob2)
            return nd.dot(h2, W3) + b3
```

3.13.7 训练

训练跟之前一样。

```
In [8]: from mxnet import autograd
        from mxnet import gluon

        loss = gluon.loss.SoftmaxCrossEntropyLoss()
        num_epochs = 5
        lr = 0.5

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

```
epoch 1, loss 1.1739, train acc 0.545, test acc 0.751
epoch 2, loss 0.5991, train acc 0.778, test acc 0.820
epoch 3, loss 0.4962, train acc 0.819, test acc 0.838
epoch 4, loss 0.4518, train acc 0.834, test acc 0.817
epoch 5, loss 0.4231, train acc 0.846, test acc 0.854
```

3.13.8 小结

我们可以通过使用丢弃法对神经网络正则化。

3.13.9 练习

- 尝试不使用丢弃法，看看这个包含两个隐含层的多层感知机可以得到什么结果。
- 我们推荐把更靠近输入层的元素丢弃概率设的更小一点。想想这是为什么？如果把本节教程中的两个元素丢弃参数对调会有什么结果?

3.13.10 扫码直达讨论区

3.14 丢弃法（Dropout）——使用 Gluon

本章介绍如何使用 `Gluon` 在训练和测试深度学习模型中使用丢弃法 (Dropout)。

3.14.1 定义模型并添加丢弃层

有了 `Gluon`, 我们模型的定义工作变得简单了许多。我们只需要在全连接层后添加 `gluon.nn.Dropout` 层并指定元素丢弃概率。一般情况下，我们推荐把更靠近输入层的元素丢弃概率设的更小一点。这个试验中，我们把第一层全连接后的元素丢弃概率设为 0.2，把第二层全连接后的元素丢弃概率设为 0.5。

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

        net = nn.Sequential()
        drop_prob1 = 0.2
        drop_prob2 = 0.5
```

```
        with net.name_scope():
            net.add(nn.Flatten())
            # 第一层全连接。
            net.add(nn.Dense(256, activation="relu"))
            # 在第一层全连接后添加丢弃层。
            net.add(nn.Dropout(drop_prob1))
            # 第二层全连接。
            net.add(nn.Dense(256, activation="relu"))
            # 在第二层全连接后添加丢弃层。
            net.add(nn.Dropout(drop_prob2))
            net.add(nn.Dense(10))
        net.initialize()
```

3.14.2 读取数据并训练

这跟之前没什么不同。

```
In [2]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import nd
        from mxnet import autograd
        from mxnet import gluon

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

        loss = gluon.loss.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 0.8081, train acc 0.704, test acc 0.822
epoch 2, loss 0.5137, train acc 0.812, test acc 0.838
epoch 3, loss 0.4519, train acc 0.834, test acc 0.857
epoch 4, loss 0.4184, train acc 0.847, test acc 0.847
epoch 5, loss 0.3951, train acc 0.856, test acc 0.858
```

3.14.3 小结

通过 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

- 尝试不同元素丢弃概率参数组合，看看结果有什么不同。

3.14.5 扫码直达讨论区

3.15 正向传播和反向传播

我们在线性回归—从 0 开始中使用了一个叫做随机梯度下降的优化算法来训练模型。在随机梯度下降每一次迭代中，模型参数的当前值将自减学习率与该参数梯度的乘积。注意到这里我们使用了模型参数的梯度。

和线性回归一样，通常情况下，我们需要使用优化算法来训练深度学习模型，而优化算法往往会依赖模型参数梯度的计算。因此，模型参数梯度的计算对模型的训练来说十分重要。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通常并不直观。虽然我们可以通过 MXNet 轻松获取模型参数的梯度，但是了解模型参数梯度的计算将有助于我们进一步了解深度学习模型训练的本质。

3.15.1 概念

反向传播（back-propagation）是计算深度学习模型参数梯度的方法。总的来说，反向传播中会依据微积分中的链式法则，按照输出层、靠近输出层的隐含层、靠近输入层的隐含层和输入层的次序，依次计算并存储模型损失函数有关模型各层的中间变量和参数的梯度。

反向传播对于各层中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。对深度学习模型按照输入层、靠近输入层的隐含层、靠近输出层的隐含层和输出层的次序，依次计算并存储模型的中间变量叫做正向传播（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。

模型定义

给定一个输入为 $\boldsymbol{x} \in \mathbb{R}^x$（每个样本输入向量长度为 x）和真实值为 $y \in \mathbb{R}$ 的训练数据样本，不考虑偏差项，我们可以得到中间变量

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

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

$$\boldsymbol{h} = \phi(\boldsymbol{z})$$

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

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

假设损失函数为 ℓ，损失项

$$L = \ell(\boldsymbol{o}, y)$$

根据 L_2 范数正则化的定义，带有提前设定的超参数 λ 的正则化项

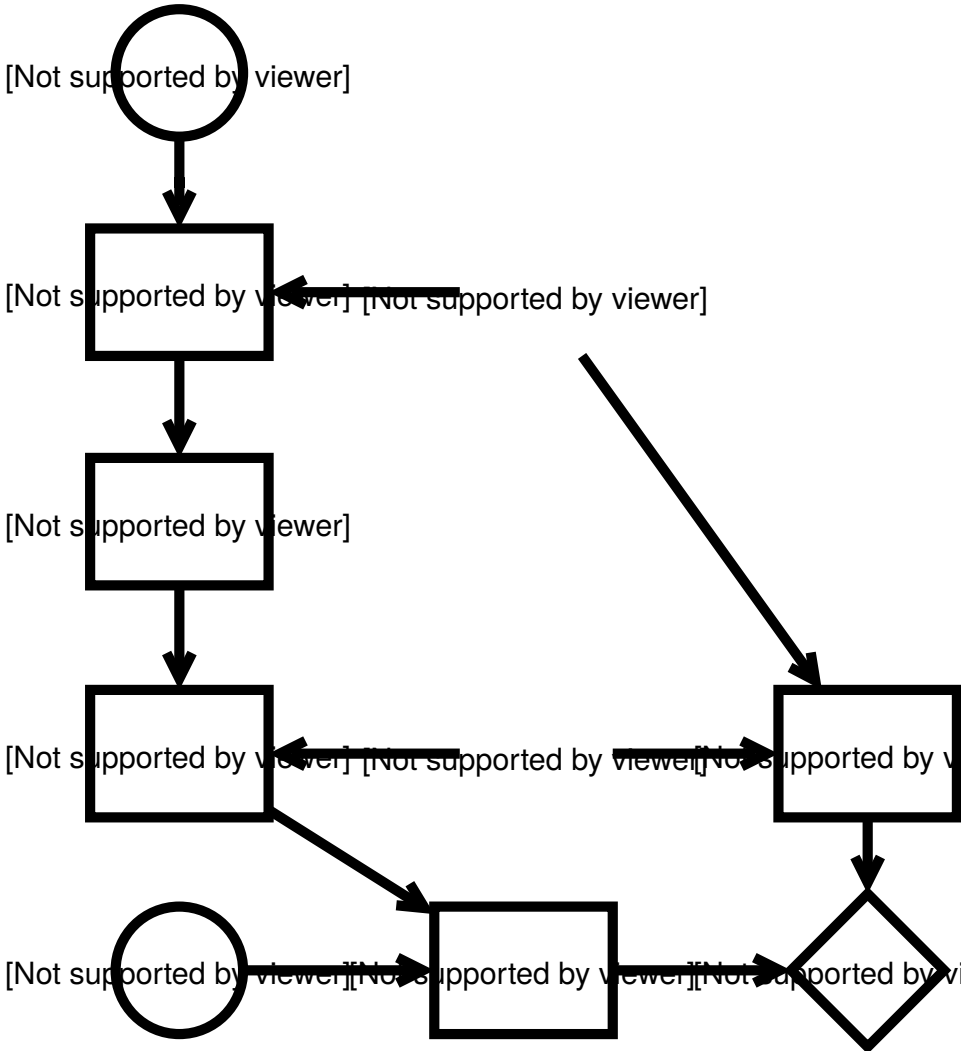
$$s = \frac{\lambda}{2}(\|\boldsymbol{W}^{(1)}\|_2^2 + \|\boldsymbol{W}^{(2)}\|_2^2)$$

模型最终需要被优化的目标函数

$$J = L + s$$

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，我们可以绘制计算图。



梯度的计算与存储

在上图中，模型的参数是 $\boldsymbol{W}^{(1)}$ 和 $\boldsymbol{W}^{(2)}$。为了在模型训练中学习这两个参数，以随机梯度下降为例，假设学习率为 η，我们可以通过

$$\boldsymbol{W}^{(1)} = \boldsymbol{W}^{(1)} - \eta \frac{\partial J}{\partial \boldsymbol{W}^{(1)}}$$

$$\boldsymbol{W}^{(2)} = \boldsymbol{W}^{(2)} - \eta \frac{\partial J}{\partial \boldsymbol{W}^{(2)}}$$

来不断迭代模型参数的值。因此我们需要模型参数梯度 $\partial J/\partial \boldsymbol{W}^{(1)}$ 和 $\partial J/\partial \boldsymbol{W}^{(2)}$。为此，我们可以按照反向传播的次序依次计算并存储梯度。

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

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

来表达链式法则。以下依次计算得到的梯度将依次被存储。

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 $\partial J/\partial \boldsymbol{o} \in \mathbb{R}^y$。

$$\frac{\partial J}{\partial \boldsymbol{o}} = \text{prod}(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \boldsymbol{o}}) = \frac{\partial L}{\partial \boldsymbol{o}}$$

正则项有关两个参数的梯度可以很直观地计算：

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

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

现在我们可以计算最靠近输出层的模型参数的梯度 $\partial J/\partial \boldsymbol{W}^{(2)} \in \mathbb{R}^{y \times h}$。在计算图中，$\boldsymbol{W}^{(2)}$ 可以经过 $\boldsymbol{o}$ 和 s 通向 J，依据链式法则，我们有

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

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

$$\frac{\partial J}{\partial \boldsymbol{h}} = \text{prod}(\frac{\partial J}{\partial \boldsymbol{o}}, \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}}) = \boldsymbol{W}^{(2)\top} \frac{\partial J}{\partial \boldsymbol{o}}$$

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

$$\frac{\partial J}{\partial \boldsymbol{z}} = \text{prod}(\frac{\partial J}{\partial \boldsymbol{h}}, \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}) = \frac{\partial J}{\partial \boldsymbol{h}} \odot \phi'(\boldsymbol{z})$$

最终，我们可以得到最靠近输入层的模型参数的梯度 $\partial J/\partial \boldsymbol{W}^{(1)} \in \mathbb{R}^{h \times x}$。在计算图中，$\boldsymbol{W}^{(1)}$ 可以经过 $\boldsymbol{z}$ 和 s 通向 J，依据链式法则，我们有

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

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 $\partial J/\partial \boldsymbol{o}$ 被计算存储，反向传播稍后的参数梯度 $\partial J/\partial \boldsymbol{W}^{(2)}$ 和隐含层变量梯度 $\partial J/\partial \boldsymbol{h}$ 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。

还有需要注意的是，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和参数的当前值。举例来说，参数梯度 $\partial J/\partial \boldsymbol{W}^{(2)}$ 的计算需要依赖隐含层变量的当前值 $\boldsymbol{h}$。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。

3.15.3 小结

正向传播和反向传播是深度学习模型训练的基石（目前是）。

3.15.4 练习

- 如果模型的层数特别多，梯度的计算会有什么问题？
- 1986 年，Rumelhart, Hinton, 和 Williams 提出了反向传播。然而 2017 年Hinton 表示他对反向传播“深刻怀疑”并发表了Capsule 论文。你对此有何思考？

3.15.5 扫码直达讨论区

3.16 实战 Kaggle 比赛：预测房价和 K 折交叉验证

本章介绍如何使用 `Gluon` 来实战Kaggle 比赛。我们以房价预测问题为例，为大家提供一整套实战中常常需要的工具，例如 **K 折交叉验证**。我们还以 `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

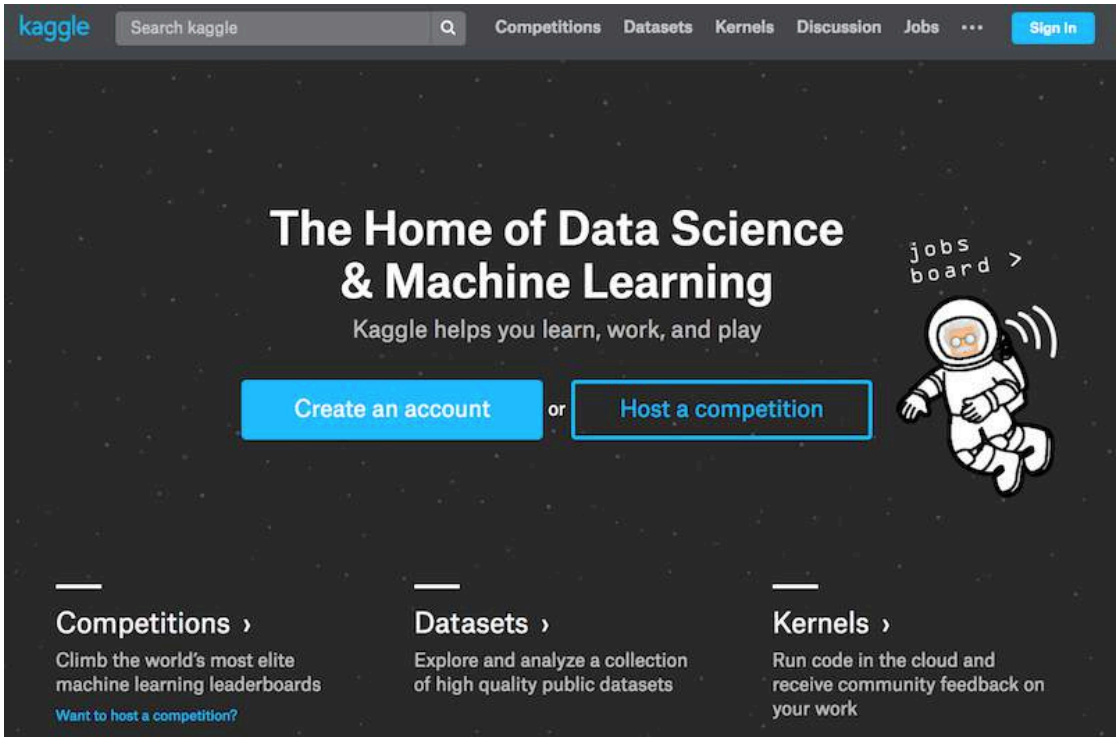
需要注意的是，本章仅提供一些基本实战流程供大家参考。对于数据的预处理、模型的设计和参数的选择等，我们特意只提供最基础的版本。希望大家一定要通过动手实战、仔细观察实验现象、认真分析实验结果并不断调整方法，从而得到令自己满意的结果。

这是一次宝贵的实战机会，我们相信你一定能从动手的过程中学到很多。

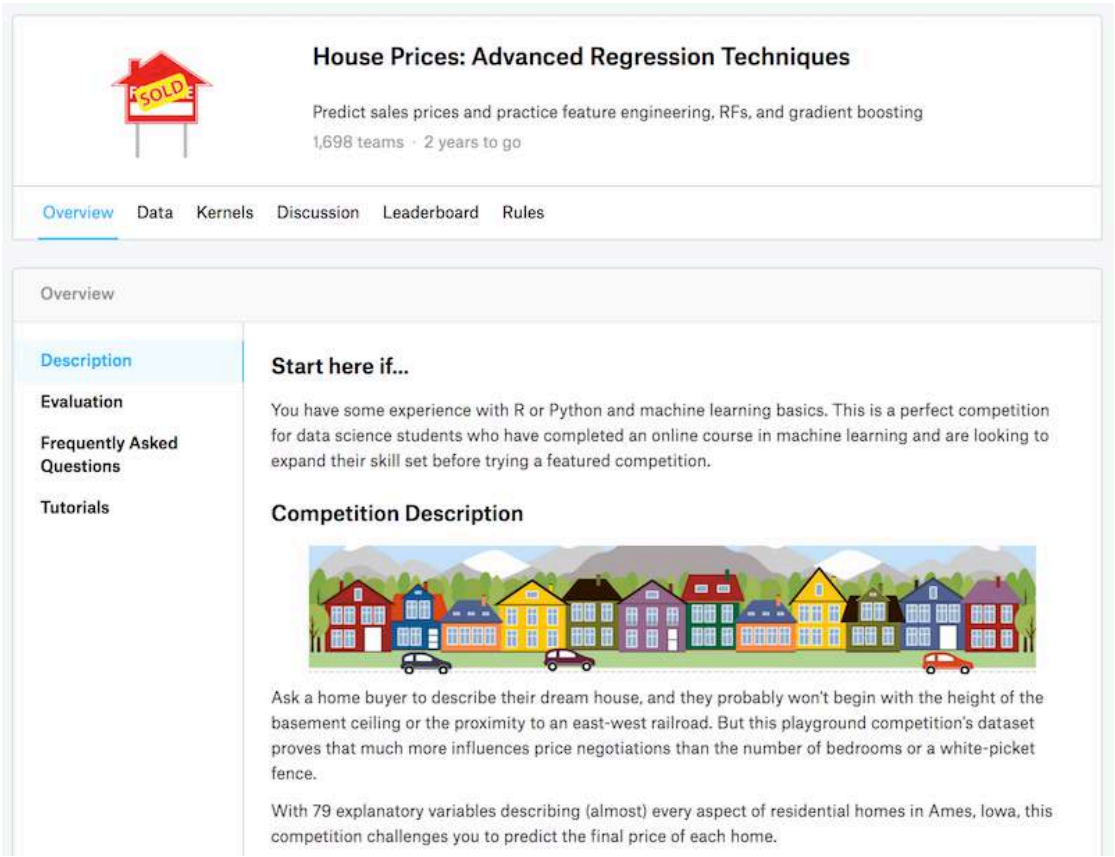
> Get your hands dirty。

3.16.1 Kaggle 中的房价预测问题

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。请注意，**目前 Kaggle 仅限每个账号一天以内 10 次提交结果的机会**。所以提交结果前务必三思。



我们以房价预测问题为例教大家如何实战一次 Kaggle 比赛。请大家在动手开始之前点击房价预测问题了解相关信息。



3.16.2 读入数据

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个房子的特征，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我们需要在测试数据集中预测的每个房子的价格。数据可以从房价预测问题中下载。

训练数据集下载地址 测试数据集下载地址

我们通过使用 `pandas` 读入数据。请确保安装了 `pandas`（`pip install pandas`）。

```
In [1]: import pandas as pd
        import numpy as np

        train = pd.read_csv("../data/kaggle_house_pred_train.csv")
```

```
test = pd.read_csv("../data/kaggle_house_pred_test.csv")
all_X = pd.concat((train.loc[:, 'MSSubClass':'SaleCondition'],
                   test.loc[:, 'MSSubClass':'SaleCondition']))
```

我们看看数据长什么样子。

```
In [2]: train.head()
```

```
Out[2]: Id  MSSubClass MSZoning  LotFrontage  LotArea Street Alley LotShape  \
        0   1          60       RL         65.0     8450   Pave   NaN      Reg
        1   2          20       RL         80.0     9600   Pave   NaN      Reg
        2   3          60       RL         68.0    11250   Pave   NaN      IR1
        3   4          70       RL         60.0     9550   Pave   NaN      IR1
        4   5          60       RL         84.0    14260   Pave   NaN      IR1

          LandContour Utilities    ...     PoolArea PoolQC Fence MiscFeature MiscVal  \
        0         Lvl    AllPub    ...            0    NaN   NaN         NaN       0
        1         Lvl    AllPub    ...            0    NaN   NaN         NaN       0
        2         Lvl    AllPub    ...            0    NaN   NaN         NaN       0
        3         Lvl    AllPub    ...            0    NaN   NaN         NaN       0
        4         Lvl    AllPub    ...            0    NaN   NaN         NaN       0

          MoSold YrSold  SaleType  SaleCondition  SalePrice
        0      2   2008        WD         Normal     208500
        1      5   2007        WD         Normal     181500
        2      9   2008        WD         Normal     223500
        3      2   2006        WD        Abnorml     140000
        4     12   2008        WD         Normal     250000

        [5 rows x 81 columns]
```

数据大小如下。

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

```
Out[3]: (1460, 81)
```

```
In [4]: test.shape
```

```
Out[4]: (1459, 80)
```

3.16.3 预处理数据

我们使用 pandas 对数值特征做标准化处理：

$$x_i = \frac{x_i - \mathbb{E}x_i}{\text{std}(x_i)}$$

```
In [5]: numeric_feats = all_X.dtypes[all_X.dtypes != "object"].index
        all_X[numeric_feats] = all_X[numeric_feats].apply(lambda x: (x - x.mean())
                                                                    / (x.std()))
```

现在把离散数据点转换成数值标签。

```
In [6]: all_X = pd.get_dummies(all_X, dummy_na=True)
```

把缺失数据用本特征的平均值估计。

```
In [7]: all_X = all_X.fillna(all_X.mean())
```

下面把数据转换一下格式。

```
In [8]: num_train = train.shape[0]

        X_train = all_X[:num_train].as_matrix()
        X_test = all_X[num_train:].as_matrix()
        y_train = train.SalePrice.as_matrix()
```

```
/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/ipykernel_l
 ↪  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_l
 ↪  auncher.py:4: FutureWarning: Method .as_matrix will be removed in a future
 ↪  version. Use .values instead.
  after removing the cwd from sys.path.
/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/ipykernel_l
 ↪  auncher.py:5: FutureWarning: Method .as_matrix will be removed in a future
 ↪  version. Use .values instead.
  """
```

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 [9]: from mxnet import ndarray as nd
        from mxnet import autograd
        from mxnet import gluon

        X_train = nd.array(X_train)
        y_train = nd.array(y_train)
        y_train.reshape((num_train, 1))
```

```
        X_test = nd.array(X_test)
```

我们把损失函数定义为平方误差。

```
In [10]: square_loss = gluon.loss.L2Loss()
```

我们定义比赛中测量结果用的函数。

```
In [11]: def get_rmse_log(net, X_train, y_train):
             num_train = X_train.shape[0]
             clipped_preds = nd.clip(net(X_train), 1, float('inf'))
             return np.sqrt(2 * nd.sum(square_loss(
                 nd.log(clipped_preds), nd.log(y_train))).asscalar() / num_train)
```

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。

```
In [12]: def get_net():
             net = gluon.nn.Sequential()
             with net.name_scope():
                 net.add(gluon.nn.Dense(1))
             net.initialize()
             return net
```

我们定义一个训练的函数，这样在跑不同的实验时不需要重复实现相同的步骤。

```
In [13]: %matplotlib inline
         import matplotlib as mpl
         mpl.rcParams['figure.dpi']= 120
         import matplotlib.pyplot as plt

         def train(net, X_train, y_train, X_test, y_test, epochs,
                   verbose_epoch, learning_rate, weight_decay):
             train_loss = []
             if X_test is not None:
                 test_loss = []
             batch_size = 100
             dataset_train = gluon.data.ArrayDataset(X_train, y_train)
             data_iter_train = gluon.data.DataLoader(
                 dataset_train, batch_size,shuffle=True)
             trainer = gluon.Trainer(net.collect_params(), 'adam',
                                     {'learning_rate': learning_rate,
                                      'wd': weight_decay})
```

```
    net.collect_params().initialize(force_reinit=True)
    for epoch in range(epochs):
        for data, label in data_iter_train:
            with autograd.record():
                output = net(data)
                loss = square_loss(output, label)
            loss.backward()
            trainer.step(batch_size)

            cur_train_loss = get_rmse_log(net, X_train, y_train)
        if epoch > verbose_epoch:
            print("Epoch %d, train loss: %f" % (epoch, cur_train_loss))
        train_loss.append(cur_train_loss)
        if X_test is not None:
            cur_test_loss = get_rmse_log(net, X_test, y_test)
            test_loss.append(cur_test_loss)
    plt.plot(train_loss)
    plt.legend(['train'])
    if X_test is not None:
        plt.plot(test_loss)
        plt.legend(['train','test'])
    plt.show()
    if X_test is not None:
        return cur_train_loss, cur_test_loss
    else:
        return cur_train_loss
```

3.16.6 K 折交叉验证

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。事实上，当我们调参时，往往需要基于 K 折交叉验证。

> 在 K 折交叉验证中，我们把初始采样分割成 K 个子样本，一个单独的子样本被保留作为验证模型的数据，其他 $K-1$ 个样本用来训练。

我们关心 K 次验证模型的测试结果的平均值和训练误差的平均值，因此我们定义 K 折交叉验证函数如下。

```
In [14]: def k_fold_cross_valid(k, epochs, verbose_epoch, X_train, y_train,
                                learning_rate, weight_decay):
             assert k > 1
             fold_size = X_train.shape[0] // k
```

```
    train_loss_sum = 0.0
    test_loss_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_loss, test_loss = train(
            net, X_val_train, y_val_train, X_val_test, y_val_test,
            epochs, verbose_epoch, learning_rate, weight_decay)
        train_loss_sum += train_loss
        print("Test loss: %f" % test_loss)
        test_loss_sum += test_loss
    return train_loss_sum / k, test_loss_sum / k
```

训练模型并交叉验证

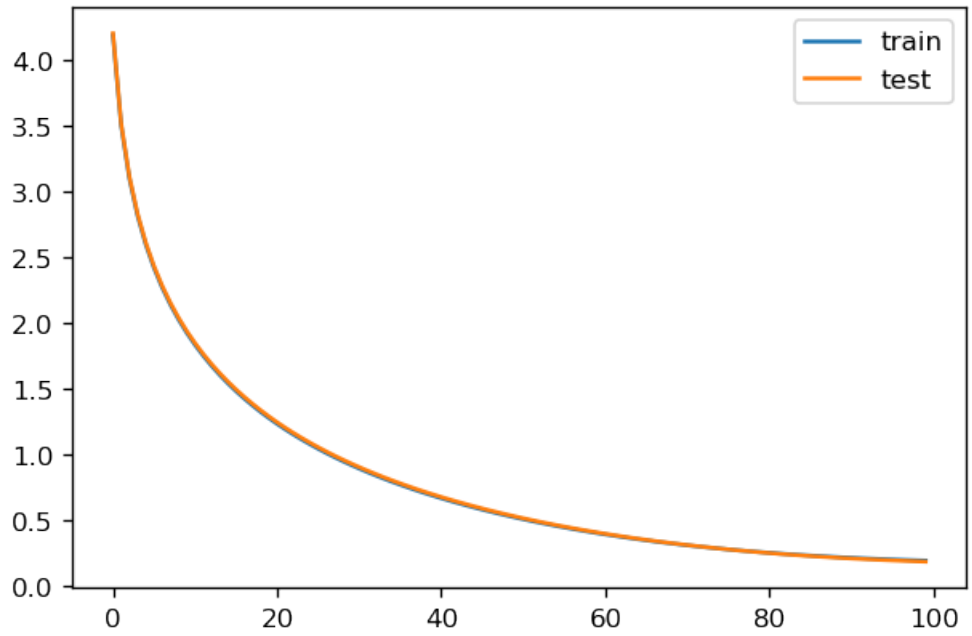
以下的模型参数都是可以调的。

```
In [15]: k = 5
         epochs = 100
         verbose_epoch = 95
         learning_rate = 5
         weight_decay = 0.0
```

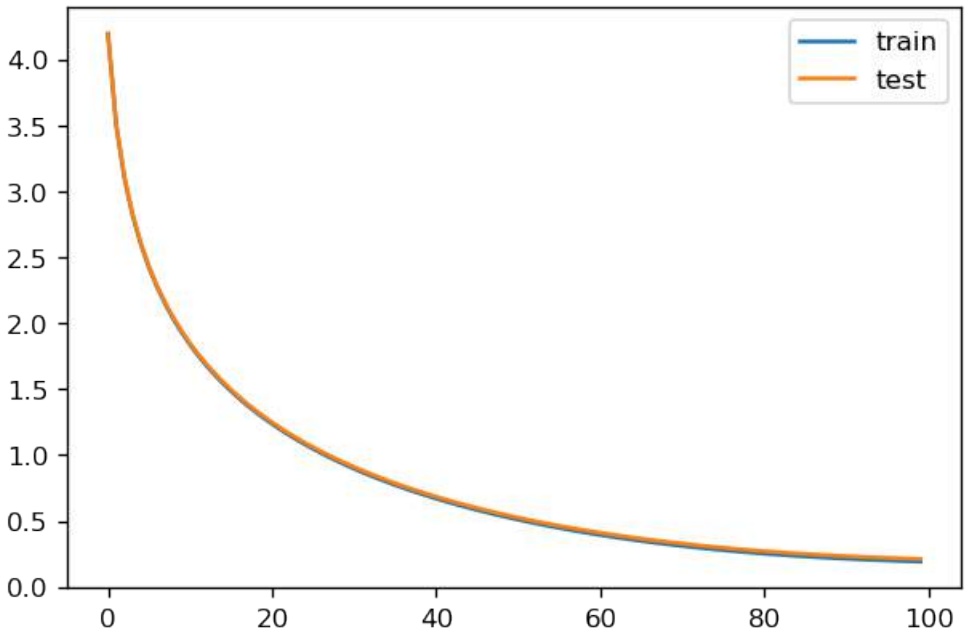
给定以上调好的参数，接下来我们训练并交叉验证我们的模型。

```
In [16]: train_loss, test_loss = k_fold_cross_valid(k, epochs, verbose_epoch, X_train,
                                                    y_train, learning_rate,
 ↪  weight_decay)
         print("%d-fold validation: Avg train loss: %f, Avg test loss: %f" %
               (k, train_loss, test_loss))
```

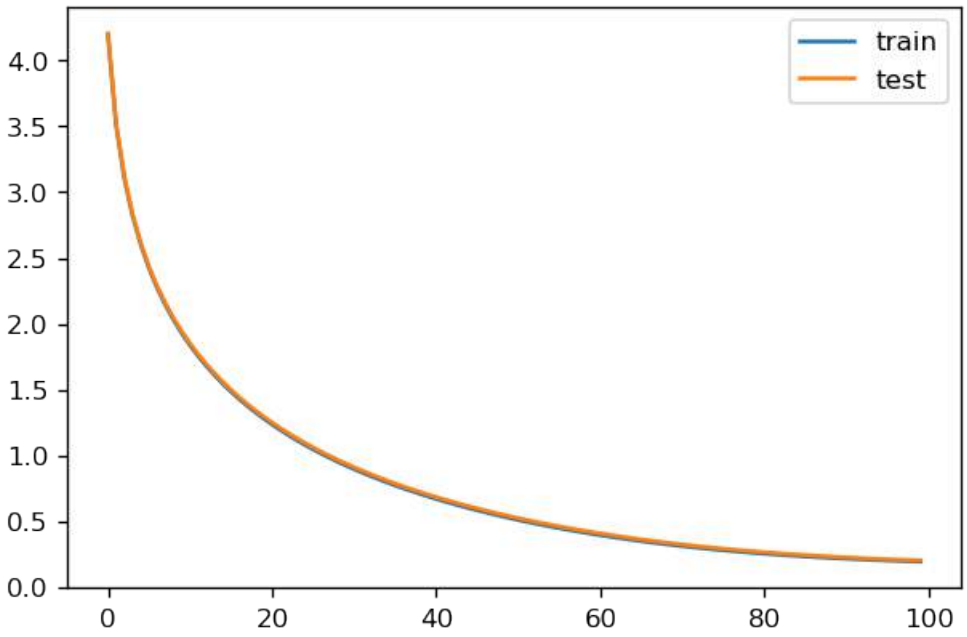
```
Epoch 96, train loss: 0.201935
Epoch 97, train loss: 0.199819
Epoch 98, train loss: 0.197814
Epoch 99, train loss: 0.195959
```



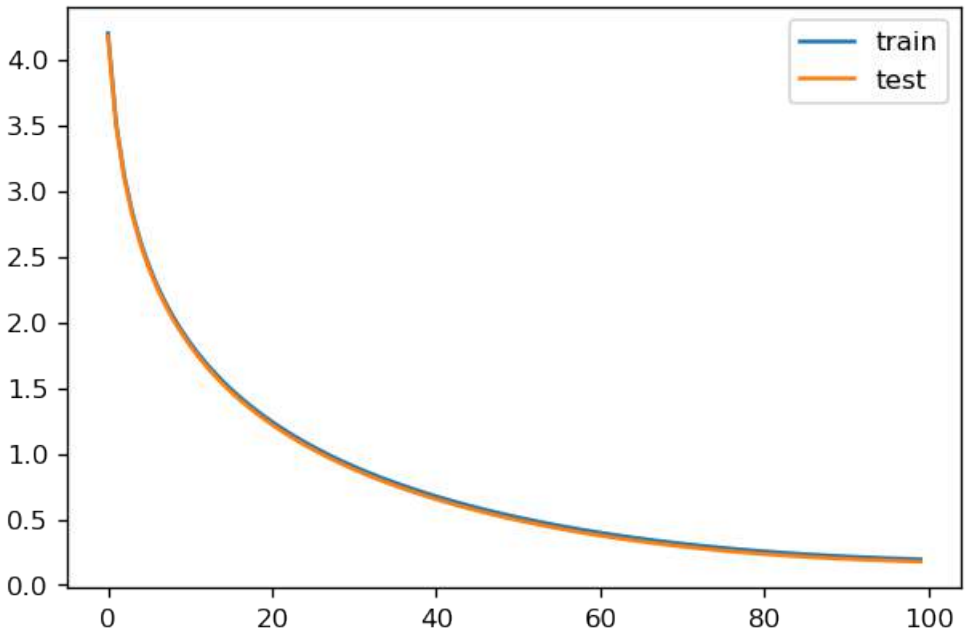

```
Test loss: 0.188436
Epoch 96, train loss: 0.198050
Epoch 97, train loss: 0.195858
Epoch 98, train loss: 0.193861
Epoch 99, train loss: 0.191910
```



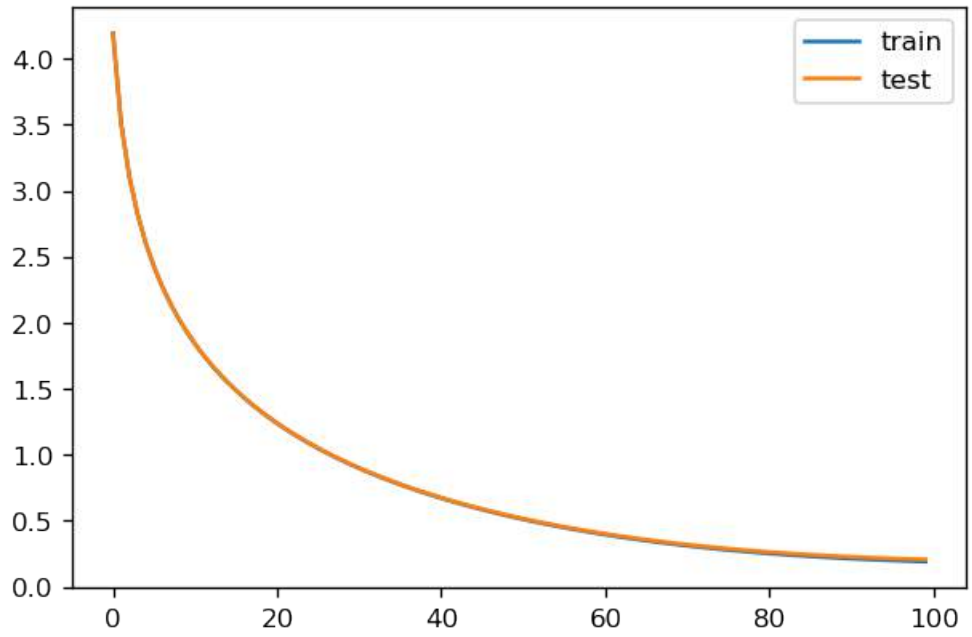

```
Test loss: 0.211722
Epoch 96, train loss: 0.199046
Epoch 97, train loss: 0.196912
Epoch 98, train loss: 0.194866
Epoch 99, train loss: 0.192946
```




```
Test loss: 0.201585
Epoch 96, train loss: 0.201766
Epoch 97, train loss: 0.199597
Epoch 98, train loss: 0.197527
Epoch 99, train loss: 0.195591
```




```
Test loss: 0.178410
Epoch 96, train loss: 0.196697
Epoch 97, train loss: 0.194487
Epoch 98, train loss: 0.192383
Epoch 99, train loss: 0.190363
```




```
Test loss: 0.206371
5-fold validation: Avg train loss: 0.193354, Avg test loss: 0.197305
```

即便训练误差可以达到很低（调好参数之后），但是 K 折交叉验证上的误差可能更高。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并小心过拟合。我们通常依赖 K 折交叉验证误差结果来调节参数。

3.16.7 预测并在 Kaggle 提交预测结果

本部分为选学内容。网络不好的同学可以通过上述 K 折交叉验证的方法来评测自己训练的模型。

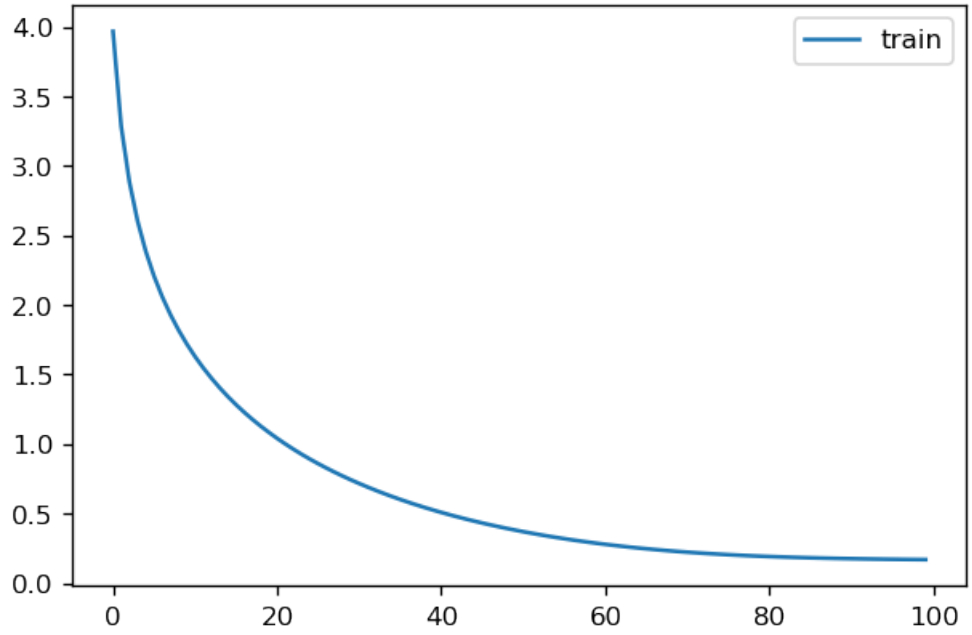
我们首先定义预测函数。

```
In [17]: def learn(epochs, verbose_epoch, X_train, y_train, test, learning_rate,
                   weight_decay):
             net = get_net()
             train(net, X_train, y_train, None, None, epochs, verbose_epoch,
                   learning_rate, weight_decay)
             preds = net(X_test).asnumpy()
             test['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
             submission = pd.concat([test['Id'], test['SalePrice']], axis=1)
             submission.to_csv('submission.csv', index=False)
```

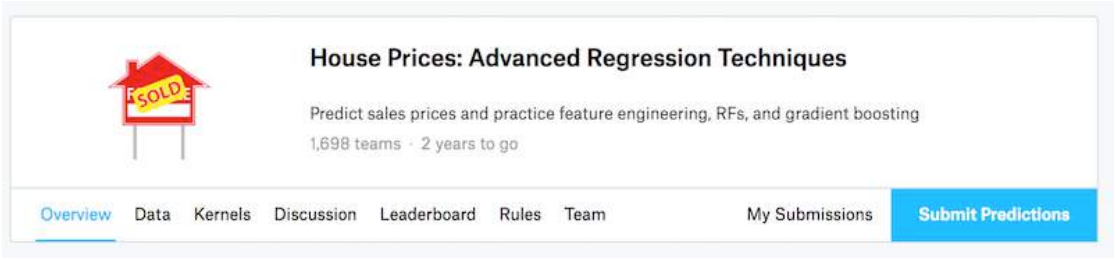
调好参数以后，下面我们预测并在 Kaggle 提交预测结果。

```
In [18]: learn(epochs, verbose_epoch, X_train, y_train, test, learning_rate,
              weight_decay)
```

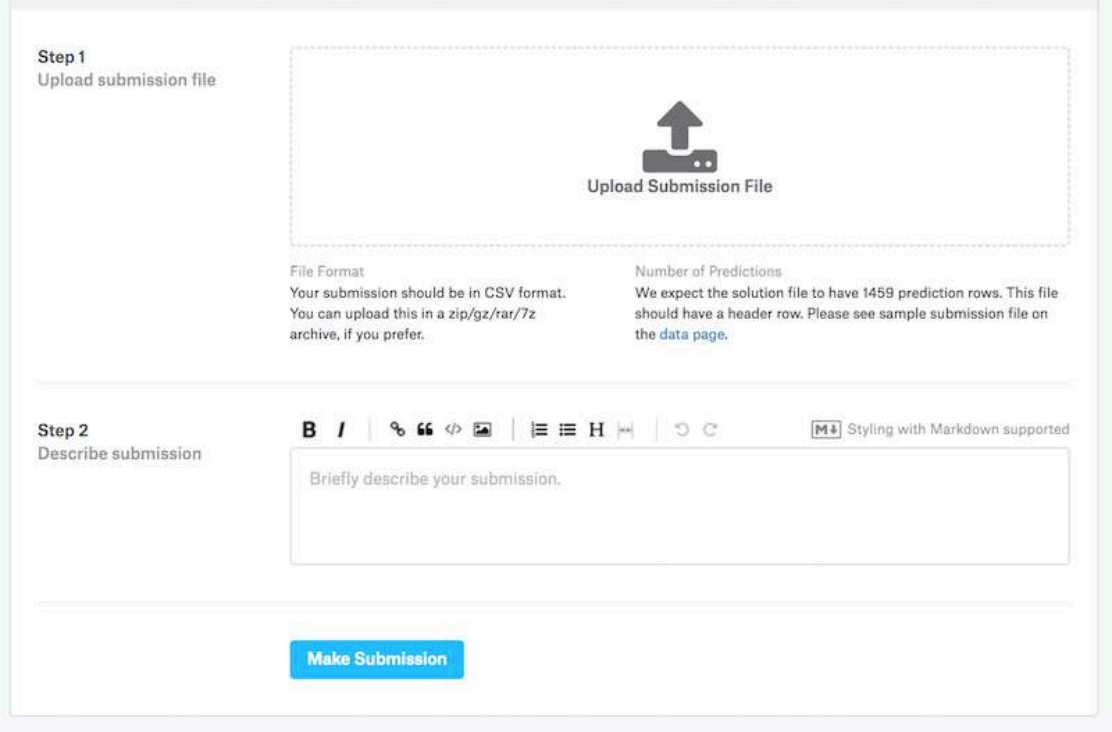
```
Epoch 96, train loss: 0.171242
Epoch 97, train loss: 0.170645
Epoch 98, train loss: 0.170112
Epoch 99, train loss: 0.169636
```



执行完上述代码后，会生成一个 `submission.csv` 文件。这是 Kaggle 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的误差。你需要登录 Kaggle 网站，打开房价预测问题地址，并点击下方右侧 `Submit Predictions` 按钮提交。



请点击下方 `Upload Submission File` 选择需要提交的预测结果。然后点击下方的 `Make Submission` 按钮就可以查看结果啦！



再次温馨提醒，**目前 Kaggle 仅限每个账号一天以内 10 次提交结果的机会**。所以提交结果前务必三思。

3.16.8 作业（汇报作业和查看其他小伙伴作业）：

- 运行本教程，目前的模型在 5 折交叉验证上可以拿到什么样的 loss?
- 如果网络条件允许，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 loss?
- 通过重新设计模型、调参并对照 K 折交叉验证结果，新模型是否比其他小伙伴的更好？除了调参，你可能发现我们之前学过的以下内容有些帮助：
  - 多层感知机—使用 Gluon

- 正则化—使用 Gluon

- 如果不使用对数值特征做标准化处理能拿到什么样的 loss?
- 你还有什么其他办法可以继续改进模型？小伙伴们都期待学习到你独特的富有创造力的解决方案。

3.16.9 扫码直达讨论区

4

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()`，来初始话模型参数。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.  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的参数，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话权重，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.         -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.  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 中这是不一定的。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。只有在当我们将形状是 `(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 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 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')
```

下面我们把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.  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
```

```
Wed May 16 23:41:29 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   40C    P0    38W / 150W |   1479MiB /  7612MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla M60           On   | 0000:00:1E.0     Off |                    0 |
| N/A   49C    P0    38W / 150W |    289MiB /  7612MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|    0    101705    C   /home/ubuntu/miniconda3/bin/python            1180MiB |
+-----------------------------------------------------------------------------+
```

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。如果有多块 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

5

卷积神经网络

本章我们介绍深度学习里面使用最广泛的一类神经网络：卷积神经网络（Convolutional neural network）。它深度学习应用在计算机视觉的基础，也正在渐渐被其他诸如自然语言处理、推荐系统和语音识别等应用采用。我们将详细讨论卷积，之后会介绍最近几年提出的数个重要的深度卷积神经网络。

5.1 卷积层

卷积层（Convolutional layer）是卷积神经网络里的基石。本节我们将介绍最简单形式的二维卷积层的是怎么工作的。虽然最早卷积层是使用卷积（Convolution）运算符，但目前的主流实现使用的是更加直观的相关（Correlation）运算符。

5.1.1 二维相关运算符

一个二维相关运算符将一个二维矩阵核（kernel）作用在一个二维输入数据上计算出一个二维矩阵输出。

下面例子里我们构造了一个 `(3, 3)` 形状的输入 `X` 和 `(2, 2)` 形状的核 `K` 来计算输出 `Y`。

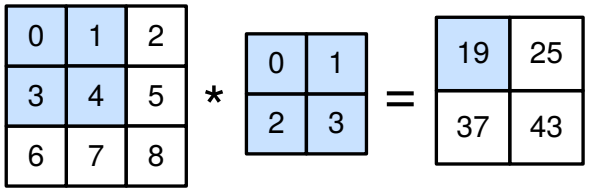


图 5.1: 二维相关运算符。

可以看到 `Y` 的形状是 `(2, 2)`，而且第一个元素是由 `X` 的左上的 `(2, 2)` 子矩阵与核做按元素乘法然后相加得来，即 `Y[0, 0] = (X[0:2, 0:2] * K).sum()`，这里我们使用假设数据类型是 NDArray。然后我们将 `X` 上子矩阵向左滑动一个元素来计算 `Y` 的第一行第二个元素。以此类推计算下面所有结果。

首先导入实验所需的包或模块。

```
In [1]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import matplotlib as mpl
        import matplotlib.pyplot as plt
        from mxnet import autograd, nd
        from mxnet.gluon import nn
```

我们将上面描述的这一过程实现在 `corr2d` 函数里。

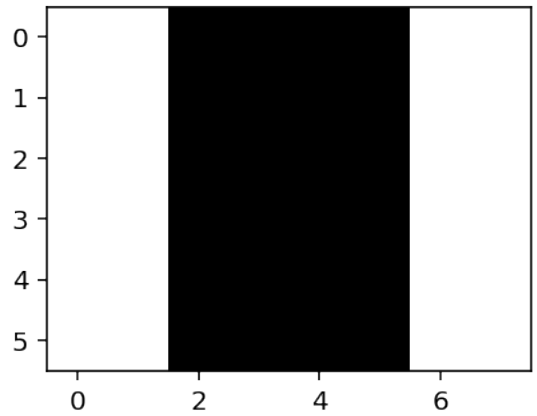
```
In [2]: def corr2d(X, K):
            n, m = K.shape
            Y = nd.zeros((X.shape[0]-n+1, X.shape[1]-m+1))
            for i in range(Y.shape[0]):
                for j in range(Y.shape[1]):
                    Y[i, j] = (X[i:i+n, j:j+m]*K).sum()
            return Y

        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 图片物体边缘检测

在这里让我们看一个如何构造一个核来进行图片里的物体边缘检测。假如我们定义一个 6×8 的图片，它中间 4 列为黑，其余为白。

```
In [3]: X = nd.ones((6, 8))
        X[:, 2:6] = 0
        gb.set_fig_size(mpl)
        plt.imshow(X.asnumpy(), cmap='gray')
        plt.show()
```



然后我们构造一个形状为 (1, 2) 的核，使得如果是作用在相同色块上时输出为 0，有颜色变化时为 1。因为这里颜色只在水平方向上变化，所以我们设计是第一个元素为 1，第二个元素为-1。

```
In [4]: K = nd.array([[1, -1]])
        K
```

```
Out[4]:
        [[ 1. -1.]]
        <NDArray 1x2 @cpu(0)>
```

对输入图片作用我们设计的核后可以发现，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 [5]: Y = corr2d(X, K)
        Y

Out[5]:
        [[ 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)>
```

5.1.3 二维卷积层

二维卷积层则将核作为权重，外加一个标量偏差。在前向计算时，它的输出就是输入数据和核的二维相关运算加上偏差。在训练的时候，我们同全连接一样将权重进行随机初始化，然后不断迭代优化权重和偏差来拟合数据。

下面的我们基于 `corr2d` 函数来实现一个自定义的二维卷基层。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 [6]: 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()
```

你也许会好奇为什么不使用二维卷积运算符呢？其实它的计算于与二维相关运算符类似，唯一的区别是我们将反向访问 `K`，即 `Y[0, 0] = (X[0:2, 0:2] * K[::-1, ::-1]).sum()`。因为在卷基层里我们会通过数据来学习 `K`，所以不管是正向还是反向访问，我们最后得到的结果是一样的。

5.1.4 学习核参数

最后我们来看一个例子，我们使用图片边缘检测例子里的 `X` 和 `Y` 来学习 `K`. 虽然我们之前定义了 Conv2D，但由于 `corr2d` 使用了对单个元素赋值（`[i, j]=`）的操作会导致系统无法对其自

动求导，所以我们使用 nn 模块里的 Conv2D 层来实现这个例子。它的使用跟我们定义的非常相似。

每一个迭代里，我们使用平方误差来比较 Y 和卷积层的输入，然后计算梯度来更新权重（为了简单起见这里忽略了偏差）。

```
In [7]: # 构造一个输出通道是 1 的二维卷基层，我们会在后面小节里解释什么是通道。
        conv2d = nn.Conv2D(1, kernel_size=(1, 2))
        conv2d.initialize()

        # 二维卷基层使用 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():
                pY = conv2d(X)
                loss = (pY - Y) ** 2
                print('batch %d, loss %.3f' % (i, loss.sum().asscalar()))
            loss.backward()
            conv2d.weight.data()[:] -= 3e-2 * conv2d.weight.grad()
```

```
batch 0, loss 12.080
batch 1, loss 4.949
batch 2, loss 2.028
batch 3, loss 0.831
batch 4, loss 0.341
batch 5, loss 0.140
batch 6, loss 0.057
batch 7, loss 0.024
batch 8, loss 0.010
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权重。

```
In [8]: conv2d.weight.data()
```

```
Out[8]:
        [[[[ 0.98949999 -0.98737049]]]]
        <NDArray 1x1x1x2 @cpu(0)>
```

如果忽略掉这里权重是一个 4 维数组（会在之后小节解释），我们学到的权重与我们之前定义的 K 已经非常接近。

5.1.5 小结

- 二维卷基层的核心计算是二维相关运算。在最简单的形式下，它将一个二维核矩阵作用在二维输入上。
- 我们可以设计核矩阵来检测图片中的边缘，同时也可以通过数据来学习这个核矩阵。

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。

5.1.7 扫码直达讨论区

5.2 填充和步幅

在同样形状的输入和核参数下，我们可以通过调节卷基层的填充和步幅来改变输出大小。这一节我们将解释这两个参数。

回忆上一节我们使用的例子，使用 `(3, 3)` 形状的输入和 `(2, 2)` 形状的核下我们可以得到 `(2, 2)` 形状的输出。输出在高和宽上均比输入小 1。一般来说，假设输入形状是 $n_h \times n_w$，核形状是 $k_h \times k_w$，那么输出形状将会是 $(n_h - k_h + 1) \times (n_h - k_w + 1)$。

5.2.1 填充

一个改变输出形状的做法是在输入的四周填充元素，其值通常设成 0。下图里我们在输入的左边和上边填充了 0 使得形状变成了 (4, 4)，从而导致输出形状增加到 (3, 3)。

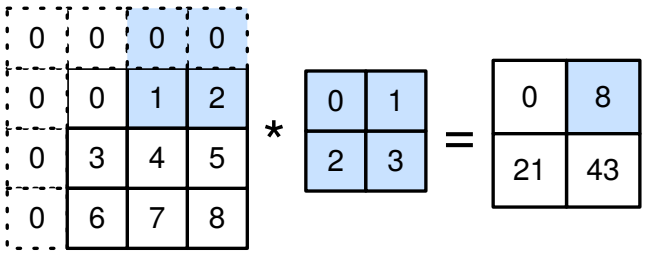


图 5.2: 在输入的左边和上边填充了 0 的二维相关计算。

一般来说，如果我们在高上填充了 p_h，在宽上填充了 p_w，那么输出形状将会是 $(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$.

在卷积神经网络里，我们经常会对输入进行填充来使得输出跟它有一样的高和宽。这样在推测网络中，尤其是深层网络，每个层的输出大小时更加容易。就是说我们会设置 $p_h = k_h - 1$ 和 $p_w = k_w - 1$。如果 k_h 是奇数，我们在输入的上下分别填充 $p_h/2$ 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。

目前的绝大部分卷积神经网络使用奇数高宽的核，例如 1，3，5，和 7。所以填充通常是对称的，而且我们知道输出 $Y[i, j]$ 是由输入以 $X[i, j]$ 为中心的区域同核进行相关计算得来。

下面例子里我们创建一个有形状为 (3, 3) 的核的二维卷积层，其中在高和宽的两侧分别填充 1。然后构造一个 (8, 8) 的输入，我们会发现输出的高宽也是 (8, 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)
```

当然我们可以使用有非方形核的卷积层，使用合适的填充可以同样得到同样大小是输出。

```
In [2]: 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

在上一节里我们实现的 corr2d 函数里，我们从左上元素开始计算输出，然后每次往左移一列或者往下移动一行。我们将每次移动的行数和列数称之为步幅。前面的例子里步幅在高和宽两个方向上均为 1。

自然我们可以使用更大步幅。下图展示了两个方向上均使步幅为 2 的情况。我们高亮了计算第二个输出元素时所使用的输入元素。跟图 5.2 相比，这里使用的输入区域向左移了一列，从而导致输出少了一行了一列。

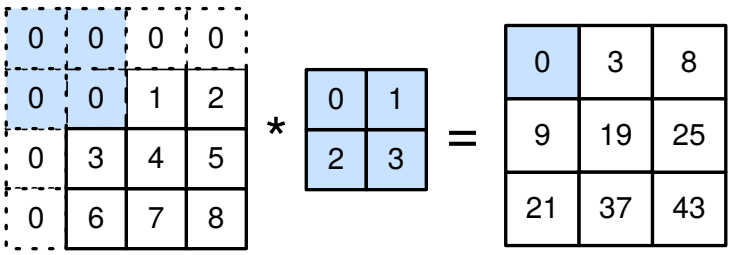


图 5.3: 同图 5.2，但使用步幅 2。

一般来说，如果在高上使用步幅 s_h，在宽上使用 s_w，那么输出大小将是 $\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor$。大致上来说，使用步幅 s，那么输出大概会减小 s 倍。具体数字则有核大小、填充大小、和是不是能整除决定。在卷积神经网络里，我们通常来使用大于 1 的步幅来成倍的减小输出的高和宽。

最后我们通过一个例子来解释如何处理不能乘除的情况。

```
In [3]: conv2d = nn.Conv2D(1, kernel_size=3, padding=1, strides=4)
        comp_conv2d(conv2d, X).shape
```

```
Out[3]: (2, 2)
```

这里 $n_h - k_h + p_h + s_h = 8 - 3 + 2 + 4 = 11$，核 $s_h = 4$，所以输出大小为 2。我们使用的输入元素如下图所示，当在横向上我们走了两步后，会发现只剩下不能被 (3, 3) 的核使用的两列。这里我们直接丢弃这两列，而不是为它们隐形的填充上 0。

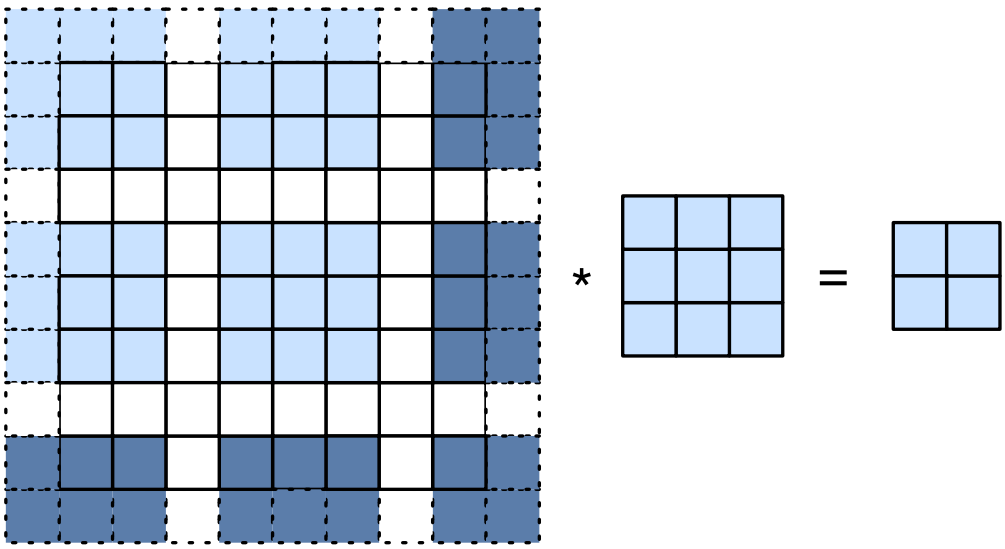


图 5.4: 使用步幅 4。

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出同高宽。
- 通过步幅可以成倍的减少输出的高宽。

5.2.4 练习

- 填充和步幅使得推断输出大小变得困难。尝试使用不同的组合来观察输出大小的变化。

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矩阵。但实际使用中我们的数据的维度经常更高。例如如果使用彩色图片作为输出，它可能有 RGB 这三个通道。假设它的高和宽分别是 h 和 w，那么内存中它可能会被表示成一个 $3 \times h \times w$ 的多维数组。我们将大小为 3 的这一维称之为通道（channel）。这一节我们将介绍输入和输出都是多通道的二维卷积层。

5.3.1 多输入通道

当输入通道数是 c_i，且使用 $k_h \times k_w$ 形状的核时，我们将会为每个输入通道分配一个单独的 $k_h \times k_w$ 形状的核参数。所以卷积层的核参数的形状将会是 $c_i \times k_h \times k_w$。下图展示了当输入通道是 2 的时候的情况。可以看到我们在每个通道里对各自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。

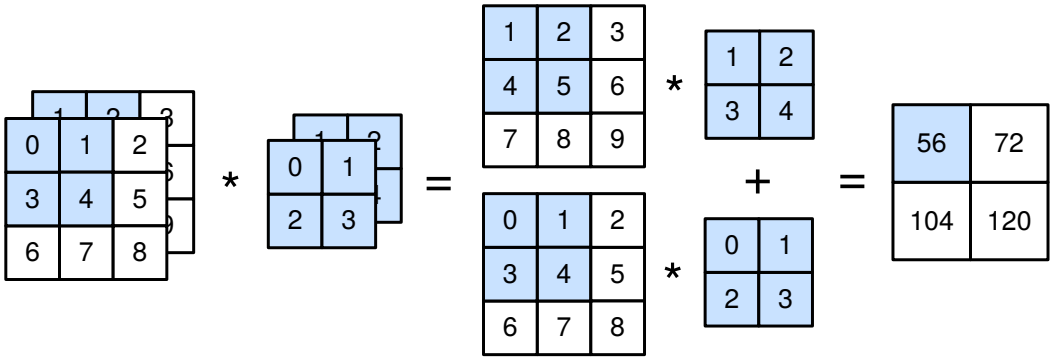


图 5.5: 输入通道为 2 的二维相关计算。

下面我们来实现它的计算。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)])
```

下面构造出上图的输入和核，然后验证结果的正确性。

```
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

由于我们对输入通道的结果做了累加，因此不论输入数据通道数的大小是多少，输入通道总是为1。如果我们想得到 c_o 通道数的输入，我们可以创建 c_o 个 $c_i \times k_h \times k_w$ 小大的核参数，然后每个核参数与输入做相关计算来得到输出的一个通道。这样，卷积层核的形状将是 $c_o \times c_i \times k_h \times k_w$。多输出通道的实现见下面代码。

```
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])
```

我们将三维的 K，K+1 和 K+2 拼在一起来构造一个输出通道为 3 的核参数，它将是一个四维数组。

```
In [5]: K = nd.stack(K, K+1, K+2)
        K.shape
```

```
Out[5]: (3, 2, 2, 2)
```

计算结果后验证我们输出有三个通道，其中第一个通道跟上例中输出一致。

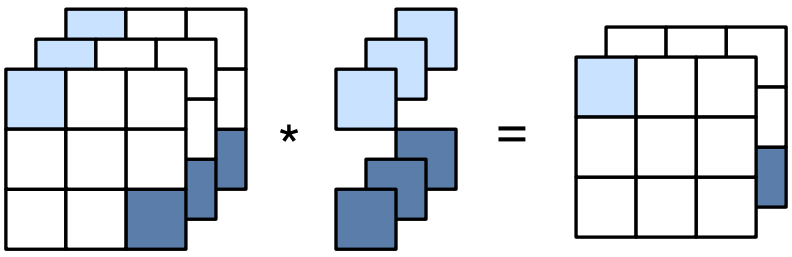
```
In [6]: corr2d_multi_in_out(X, K)
```

```
Out[6]:
        [[[  56.   72.]
          [ 104.  120.]]

         [[  76.  100.]
          [ 148.  172.]]

         [[  96.  128.]
          [ 192.  224.]]]
        <NDArray 3x2x2 @cpu(0)>
```

5.3.3 1×1 卷积层

最后我们讨论一下使用 1×1 形状核（$k_h = k_w = 1$）的特殊卷基层。它失去了卷基层可以识别相邻元素构成的模式的功能（例如回忆之前介绍的图片边缘识别），但仍然经常使用。

下图展示了输入通道为 3 和输出通道为 2 的情况。输出里的每个元素来自输入对应位置的元素在不同通道之间的按权重累加。这个情况下，它等价于一个输入大小为 2 和输出大小为 3 的全连接层。



下面代码我们直接使用矩阵乘法来来实现 1×1 卷积层。可以看到，这里的通道对应全连接的特征，而宽和高里的元素则对应之前的数据点。

```
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我们这个实现的正确性。

```
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
```

可以看到 1×1 卷积层虽然失去了识别空间模式的功能，但它能够混合输在不同通道之间的信息，因此被经常使用在调整网络的层之间的通道数，从而控制模型复杂度。

5.3.4 小结

- 使用多通道可以极大拓展卷基层的模型参数。
- 1×1 卷积层通常用来调节通道数。

5.3.5 练习

- 假设输入大小为 $c_i \times h \times w$，我们使用 $c_o \times c_i \times k_h \times k_w$ 的核，而且使用 (p_h, p_w) 填充和 (s_h, s_w) 步幅，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？
- `Y1` 和 `Y2` 结果完全一致吗？原因是什么？
- 对于非 1×1 卷积层，如果将其也表示成一个矩阵乘法。

5.3.6 扫码直达讨论区

5.4 池化层

在“卷积层”这一小节里我们介绍了检测图片中的物体边缘。我们通过一个手动构造了核的卷积层来精确的找到像素变化的位置。例如如果输出为 `Y` 且 `Y[i, j]=1`，那么边缘通过输入 `X` 的 `X[i, j]` 和 `X[i, j+1]` 中间。但在实际应用中，图片中我们感兴趣的物体不会总出现在固定位置。而卷积层这种对位置敏感的特性不能很好的处理这种情况。

5.4.1 二维最大、平均池化层

池化层（pooling layer）的提出就是针对这个问题，它同卷积层那样每次作用在输入数据的一个固定大小的窗口上。不同于卷积层里通过核来计算输出，池化层通常直接计算窗口内元素的最大值或者平均值。

下图展示了一个 2×2 窗口的最大池化层。其中输出的第一个元素由输入的左上 2×2 窗口里的四个元素的最大值构成。然后和卷积层一样依次向左或向下移动窗口来计算其余的输出。

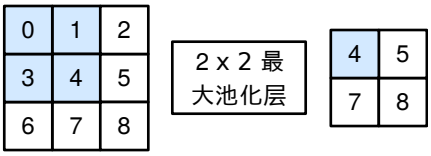


图 5.6: 2×2 最大池化层

如果这个池化层是输入是用于物体边缘的卷积层的输出。假设卷积层输入是 `X`，那么不管是物体边缘通过 `X[i, j]` 和 `X[i, j+1]`，还是 `X[i, j+1]` 和 `X[i, j+2]`，我们都有池化层输出 `Y[i, j]=1`。换句话说，使用 2×2 最大池化层，只要感兴趣物体的位置偏差在不超过两个元素，我们均可以检测出来。

下面我们通过 pool2d 函数来实现这个计算。它跟“卷积层”里 corr2d 函数的实现非常类似，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四周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的实现 MaxPool2D 类来说明它的工作机制。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，和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会混合输入通道。这个意味池化层的输出通道跟输入通道数相同。

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

- 池化层的一个主要作用是缓解卷积层的位置敏感性。
- 池化层的计算同卷积层类似通过滑动窗口计算结果。但窗口内计算更加简单，通常直接取输入窗口内元素的最大值或者平均值。

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多少次乘法，多少次加法?
- 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”这一节里我们构造了一个两层感知机模型来对 FashionMNIST 这个图片数据集进行分类。这个数据集的图片形状是 28×28。我们将二维图片展开成一个长为 784 的向量输入到模型里。这样的做法虽然简单，但有两个重要的局限性。

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，从而很难被模型察觉。
2. 对于大图片输入模型可能会过大。例如对于输入是 $1000 \times 1000 \times 3$ 的彩色照片，即使隐藏层输出仍为 256，这一层的模型形状是 $3,000,000 \times 256$，其占用将近 3GB 的内存，这带来过于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，使得有效的发掘水平和垂直两个方向上的数据关联。且通过滑动窗口将核参数重复作用在输入上，而得到更紧凑的参数表示。卷积神经网络就是主要由卷积层组成的网络，这一小节我们介绍一个著名的早期用来识别手写数字图片的卷积神经网络：LeNet [1]，名字来源于论文一作 Yann LeCun。它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 定义模型

LeNet 分为卷积层块和全连接层块两个部分。卷积层块里的基础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，例如笔画，之后的最大池化层则用来减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。每一块里的卷积层都使用 5×5 窗口，且在输出上作用 sigmoid 激活函数 $f(x) = \frac{1}{1+e^{-x}}$ 来将输入非线性变换到 $(0, 1)$ 区间。不同点在于第一个卷积层输出通道为 5，第二个则增加到 16。两个最大池化层的窗口大小均为 2×2，且步幅为 2。这意味着每个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，对应 10 类数字，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:\t', 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。（我们将下面这段代码保存在 GluonBook 的 `try_gpu` 函数里方便下次重复使用）。

```
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] 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区间，从而使得输入输出的方差相似来使得网络优化更加稳定。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]: loss = gluon.loss.SoftmaxCrossEntropyLoss()

        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 1})

        gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)
```

```
training on  gpu(0)
epoch 1, loss 2.3206, train acc 0.099, test acc 0.102, time 1.8 sec
epoch 2, loss 2.1758, train acc 0.160, test acc 0.390, time 1.6 sec
epoch 3, loss 1.0520, train acc 0.581, test acc 0.655, time 1.6 sec
epoch 4, loss 0.7794, train acc 0.699, test acc 0.714, time 1.6 sec
epoch 5, loss 0.6677, train acc 0.736, test acc 0.744, time 1.6 sec
```

5.5.3 小节

LeNet 使用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 是针对 MNIST 提出，但在我们这里的 FashionMNIST 上效果不是特别好。一个原因可能是 FashionMNIST 数据集更加复杂，可能需要更复杂的网络。尝试修改它来提升精度。可以考虑调整卷积窗口大小，输出层大小，激活函数，全连接层输出大小。当然在优化方面，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。

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

LeNet 提出后的将近二十年里，神经网络曾一度被许多其他方法超越。虽然 LeNet 可以在 MNIST 上得到好的成绩，在更大的真实世界的数据集上，神经网络表现并不佳。一方面神经网络计算慢，虽然 90 年代也有过一些针对神经网络的加速硬件，但并没有像目前 GPU 那样普及。因此训练一个多通道、多层和大量参数的在当年很难实现。另一方面，当时候研究者还没有大量深入研究权重的初始化和优化算法等话题，导致复杂的神经网络很难收敛。

跟神经网络从原始像素直接到最终标签，或者通常被称为端到端（end-to-end），不同。很长一段时间里流行的时研究者们通过勤劳、智慧和黑魔法生成了许多手工特征。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这一步）。

这样的局面一直维持到 2012 年。如果那时候如果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。这人会告诉你图像识别里”不可告人”的现实是，计算机视觉里的机器学习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

简单来说，给定一个数据集，当时流程里最重要的是特征表示这步。并且直到 2012 年，特征表示这步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。然而许多年来，研究者们由于种种原因并不能实现这一愿景。

缺失要素一：数据

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，这其中有诸多因素。第一，包含许多表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，虽然这些当时还不为人知。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 2009 年李飞飞团队贡献了 ImageNet 数据库后得以焕然一新。它包含了 1000 类，每类有 1000 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生的，尤其是为了大吞吐量的 4x4 矩阵和向量乘法，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开始使用。

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。它与 LeNet 的设计理念非常相似。但也有非常显著的特征。

第一，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 11×11，接着第二层中的是 5×5，之后都是 3×3。此外，第一，第二和第五个卷积层之后都跟了有重叠的窗口为 3×3，步幅为 2×2 的最大池化层。

紧接着卷积层，AlexNet 有每层大小为 4096 个节点的全连接层们。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 包括了双数据流的设计，以让网络中一半的节点能存入一个 GPU。这两个数据流，也就是说两个 GPU 只在一部分层进行通信，这样达到限制 GPU 同步时的额外开销的效果。有幸的是，GPU 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。

第二，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。

第三，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。

第四，引入了大量的图片增广，例如翻转、裁剪和颜色变化，来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(
            # 因为输入图片尺寸（224 x 224）比 LeNet（28 x 28）大很多，
            # 因此使用较大的卷积窗口来捕获物体。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:\t', X.shape)
```

```
conv0 output shape:      (1, 96, 54, 54)
pool0 output shape:      (1, 96, 26, 26)
```

```
conv1 output shape:      (1, 256, 26, 26)
pool1 output shape:      (1, 256, 12, 12)
conv2 output shape:      (1, 384, 12, 12)
conv3 output shape:      (1, 384, 12, 12)
conv4 output shape:      (1, 256, 12, 12)
pool2 output shape:      (1, 256, 5, 5)
dense0 output shape:     (1, 4096)
dropout0 output shape:   (1, 4096)
dense1 output shape:     (1, 4096)
dropout1 output shape:   (1, 4096)
dense2 output shape:     (1, 10)
```

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]: ctx = gb.try_gpu()
        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': .01})

        loss = gluon.loss.SoftmaxCrossEntropyLoss()
        gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=3)
```

```
training on  gpu(0)
epoch 1, loss 1.3105, train acc 0.510, test acc 0.743, time 99.5 sec
epoch 2, loss 0.6480, train acc 0.759, test acc 0.811, time 97.3 sec
epoch 3, loss 0.5313, train acc 0.803, test acc 0.837, time 97.3 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基层。但他们在卷积窗口上、通道数上和构造顺序上均有一定区别。如果构造更加深层的网络，我们需要更加简单的构造规则。VGG [1]，名字来源于论文作者所在实验室 Visual Geometry Group，提出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 网络

VGG 模型的最基础单位是连续数个相同的使用 1 填充的 3×3 卷基层后接上一个不重叠的 2×2 最大池化层。前者保持输出输出高宽，后者则对其减半。我们定义这个这个基础块叫 vgg_block，它可以指定连续的卷基层个数和卷基层的输出通道数。

```
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
```

VGG 网络同前一样由数个卷积层后接上数个全连接层构成。卷积层部分串联数个卷积块，其具体架构由 conv_arch，即每个块的卷基层个数和输出通道，定义。全连接层则跟 AlexNet 一样使用 3 个全连接层。

```
In [2]: def vgg(conv_arch, num_outputs):
            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(num_outputs))
            return net
```

现在我们构造一个 VGG 网络。它有 5 个卷积块，前三块使用单卷基层，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。然后我们打印每个卷积块的输出变化。

```
In [3]: conv_arch = ((1,64), (1,128), (2,256), (2,512), (2,512))
```

```
        net = vgg(conv_arch, 1000)
        net.initialize()

        X = nd.random.uniform(shape=(1,1,224,224))
        for blk in net:
            X = blk(X)
            print(blk.name, 'output shape:\t', X.shape)
```

```
sequential1 output shape:        (1, 64, 112, 112)
sequential2 output shape:        (1, 128, 56, 56)
sequential3 output shape:        (1, 256, 28, 28)
sequential4 output shape:        (1, 512, 14, 14)
sequential5 output shape:        (1, 512, 7, 7)
dense0 output shape:     (1, 4096)
dropout0 output shape:   (1, 4096)
dense1 output shape:     (1, 4096)
dropout1 output shape:   (1, 4096)
dense2 output shape:     (1, 1000)
```

可以看到每次我们将长宽减半，最后变成 7×7 后进入全连接层。这个模式在之后我们会常见到。同时输出通道数每次都翻倍。因为每个卷基层的窗口大小一样，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。

5.7.2 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 [4]: ratio = 4
        small_conv_arch = [(pair[0], int(pair[1]/ratio)) for pair in conv_arch]
        net = vgg(small_conv_arch, 10)
```

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 [5]: ctx = gb.try_gpu()
        net.initialize(ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': .05})

        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 1.1499, train acc 0.580, test acc 0.826, time 156.0 sec
epoch 2, loss 0.4245, train acc 0.843, test acc 0.878, time 153.5 sec
epoch 3, loss 0.3389, train acc 0.877, test acc 0.896, time 153.5 sec
```

5.7.3 小结

VGG 通过 5 个可以重复使用的卷积块来构造网络。根据卷积块里卷积层数目和输出通道不同可以定义出不同的 VGG 模型。

5.7.4 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。

5.7.5 扫码直达讨论区

5.7.6 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

回忆我们介绍的 LeNet、AlexNet 和 VGG 均由两个部分组成，输入图片首先进入由卷积层构成的部分充分抽取空间特征，然后再进入由全连接层构成的部分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部分。

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 网络中的网络

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再接上卷积层，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，我们介绍了 1×1 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 1×1 卷积层来替代全连接层使得空间信息能够传递到后面的层去。下图对比了 NiN 同 AlexNet 和 VGG 等网络的主要区别。

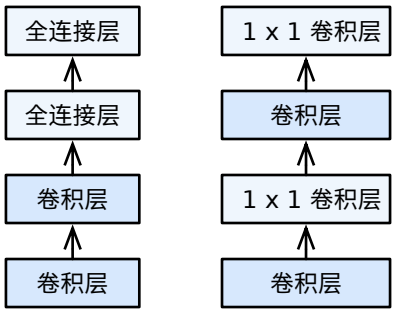


图 5.7: 对比 NiN（右）和其他（左）

NiN 中的一个构成块由一个卷积层外加两个充当全连接层的 1×1 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除了设置它们的输出通道与之前一致外，别的超参数的均使用了固定值。如果使用

```
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
```

NiN 的卷积层超参数跟 Alexnet 类似，使用窗口分别为 11×11、5×5 和 3×3 的卷积层。卷积层后跟窗口为 3×3 有重叠的最大池化层。但除了使用 NiN 块外，还有一点重要的跟 AlexNet 的不同在于去除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，然后在每一个通道使用一个平均池化层来将这个通道里的数值平均成一个标量作为输出。这个设计显著的减小模型参数大小，从而鞯更好的避免了过拟合。但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:\t', X.shape)
```

```
sequential1 output shape:        (1, 96, 54, 54)
```

```
pool0 output shape:      (1, 96, 26, 26)
sequential2 output shape:        (1, 256, 26, 26)
pool1 output shape:      (1, 256, 12, 12)
sequential3 output shape:        (1, 384, 12, 12)
pool2 output shape:      (1, 384, 5, 5)
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)
```

5.8.2 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]: ctx = gb.try_gpu()
        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': .1})

        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 2.2008, train acc 0.180, test acc 0.311, time 131.4 sec
epoch 2, loss 1.4991, train acc 0.441, test acc 0.575, time 129.8 sec
epoch 3, loss 0.9996, train acc 0.650, test acc 0.705, time 129.8 sec
```

5.8.3 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 1×1 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。

5.8.4 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 1×1 卷积层，去除一个看看？

5.8.5 扫码直达讨论区

5.8.6 参考文献

[1] Lin, M., Chen, Q., & Yan, S. (2013). Network in network. arXiv preprint arXiv:1312.4400.

5.9 含并行连结的网络：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，在此基础上做了很大的改进。在随后的几年里研究人员持续对它进行改进，提出了数个被广泛使用的版本。本小节将介绍这个模型系列的一个版本。

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。

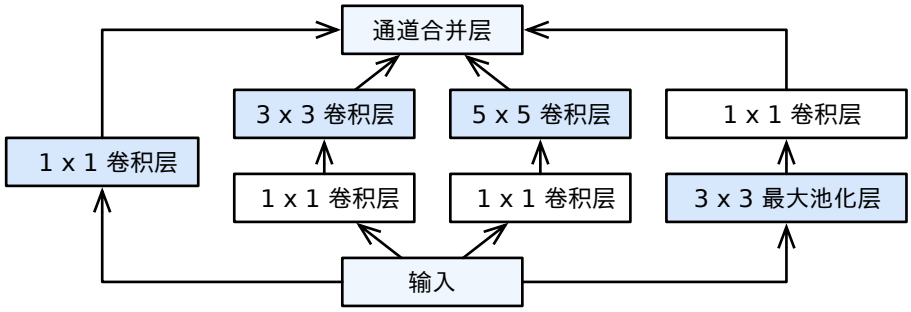


图 5.8: Inception 块。

由上图可以看出，Inception 里有四个并行的线路。前三个通道里使用窗口大小分别是 1×1、3×3 和 5×5 的卷基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 1×1 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 3×3 最大池化层，后接 1×1 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 x 1 卷积层。
                self.p1_1 = nn.Conv2D(c1, kernel_size=1, activation='relu')
                # 线路 2，1 x 1 卷积层后接 3 x 3 卷积层。
                self.p2_1 = nn.Conv2D(c2[0], kernel_size=1, activation='relu')
                self.p2_2 = nn.Conv2D(c2[1], kernel_size=3, padding=1,
                                      activation='relu')
                # 线路 3，1 x 1 卷积层后接 5 x 5 卷积层。
                self.p3_1 = nn.Conv2D(c3[0], kernel_size=1, activation='relu')
                self.p3_2 = nn.Conv2D(c3[1], kernel_size=5, padding=2,
                                      activation='relu')
                # 线路 4，3 x 3 最大池化层后接 1 x 1 卷积层。
```

```
        self.p4_1 = nn.MaxPool2D(pool_size=3, strides=1, padding=1)
        self.p4_2 = nn.Conv2D(c4, kernel_size=1, activation='relu')

    def forward(self, x):
        p1 = self.p1_1(x)
        p2 = self.p2_2(self.p2_1(x))
        p3 = self.p3_2(self.p3_1(x))
        p4 = self.p4_2(self.p4_1(x))
        # 在通道维上合并输出
        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 3×3 最大池化层来减小输出高宽。第一模块使用一个 64 通道的 7×7 卷基层。

```
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)
        )
```

第二模块使用两个卷基层，首先是 64 通道的 1×1 卷基层，然后是将通道增大 3 倍的 3×3 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)
        )
```

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基层。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)
        )
```

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，3×3 卷积层线路输出最多通道，其次是 1×1 卷基层线路，之后是 5×5 卷基层和 3×3 最大池化层线路。其中线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)
        )
```

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.Flatten(), 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, 23, 23)
sequential1 output shape:        (1, 192, 11, 11)
sequential2 output shape:        (1, 480, 5, 5)
```

```
sequential3 output shape:        (1, 832, 2, 2)
sequential4 output shape:        (1, 1024, 1, 1)
flatten0 output shape:   (1, 1024)
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]: ctx = gb.try_gpu()
        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': .1})

        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=3)
```

```
training on  gpu(0)
epoch 1, loss 1.7184, train acc 0.381, test acc 0.673, time 75.9 sec
epoch 2, loss 0.6003, train acc 0.776, test acc 0.830, time 74.2 sec
epoch 3, loss 0.4404, train acc 0.837, test acc 0.858, time 74.2 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基层和最大池化层来并行抽取信息, 使用 1×1 卷基层减低通道数来减少模型复杂度。GoogLeNet 则精细的将多个 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。这个使得 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

这一节我们介绍一个让深层卷积网络训练更加容易的层：批量归一化（batch normalization）[1]。回忆在‘“实战 Kaggle 比赛：预测房价和 K 折交叉验证” <../chapter_supervised-learning/kaggle-gluon-kfold.md>‘__ 这一节里，我们对输入数据做了归一化处理，就是将每个特征在所有样本上的值转换成均值为 0 方差为 1。这样所有数值都同样量级上，从而使得训练的时候数值更加稳定。

这个数据归一化预处理对于浅层模型来说通常足够了，因为通过几层网络的作用后，输出值通常不会出现剧烈变化。但对于深层神经网络来说，情况可能会比较复杂。因为每一层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，最终的输出可能极大或者极小，而且权重改变都可能带来输出的巨大变化。为了让训练稳定，通常我们每次只能对权重做很小的改动，即使用很小的学习率，进而导致收敛缓慢。

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。假设这个全连接层对一个批量数据输出 n 个向量数据点 $X = \{x_1, \ldots, x_n\}$，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，其均长度为 p 的向量：

$$\mu_X \leftarrow \frac{1}{n}\sum_{i=1}^{n} x_i,$$

$$\sigma_X^2 \leftarrow \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_B)^2.$$

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：

$$\hat{x_i} \leftarrow \frac{x_i - \mu_X}{\sqrt{\sigma_X^2 + \epsilon}},$$

这里 ϵ 是一个很小的常数保证不除以 0。在上面归一化的基础上，批量归一化层引入了两个可以学习的模型参数，拉升参数 γ 和偏移参数 β。它们均为 p 长向量，并作用在 $\hat{x_i}$ 上：

$$y_i \leftarrow \gamma\hat{x_i} + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i).$$

这里 $Y = \{y_1, \ldots, y_n\}$ 是批量归一化层的输出。

如果批量归一化层是放置在卷基层后面，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来计算（参考“多输入和输出通道”里我们对 1×1 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)
        # 接在二维卷基层后的情况，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。我们在所有的卷基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)
        )
```

使用更前同样的超参数，可以发现前面五个迭代周期的收敛有明显加速。

```
In [4]: ctx = gb.try_gpu()
        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 1})
        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 0.6516, train acc 0.769, test acc 0.754, time 3.4 sec
epoch 2, loss 0.3942, train acc 0.858, test acc 0.772, time 3.3 sec
epoch 3, loss 0.3478, train acc 0.875, test acc 0.810, time 3.3 sec
```

```
epoch 4, loss 0.3193, train acc 0.884, test acc 0.799, time 3.8 sec
epoch 5, loss 0.3016, train acc 0.890, test acc 0.880, time 3.8 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]: (
         [ 1.40176749  0.00727557 -0.00675547  0.56281644 -0.41581157 -1.52440953]
         <NDArray 6 @gpu(0)>,
         [ 2.27523398  1.41030586  1.9413203   1.5304116   1.04069507  1.62909055]
         <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
        import mxnet as mx
        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]: ctx = gb.try_gpu()
        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 1})
        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 0.6672, train acc 0.762, test acc 0.823, time 2.2 sec
epoch 2, loss 0.3994, train acc 0.856, test acc 0.805, time 1.9 sec
epoch 3, loss 0.3487, train acc 0.875, test acc 0.851, time 1.9 sec
epoch 4, loss 0.3256, train acc 0.882, test acc 0.853, time 1.9 sec
epoch 5, loss 0.3035, train acc 0.889, test acc 0.854, 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。当我们将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。

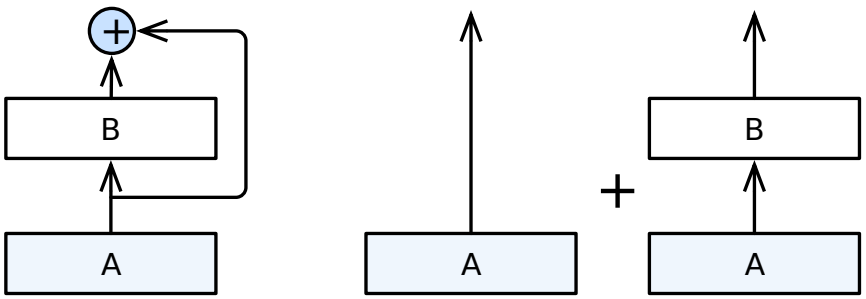


图 5.9: 残差快（左）和它的分解（右）

ResNet 沿用了 VGG 全 3×3 卷积层设计。残差块输入首先被连续作用两次同样输入通道的 3×3 卷积层，每个卷积层后跟一个批量归一化层，然后是 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。这样我们要求这两个卷积层的输出都保持跟输入形状一样来保证可以之后与输入相加。

如果我们想改变输入大小，意味着卷积层使用跟输入不一样的通道大小，同时我们让第一个卷积层使用步幅 2 来减半输入高宽。为了保证的相加操作还能进行，我们引入一个额外的 1×1 卷积层来将输入变换成需要的形状后再相加。

残差块的实现见下。它可以设定输出通道数，和是否保持输出形状和输入一致。

```
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

        class Residual(nn.Block):
            def __init__(self, num_channels, same_shape=True, **kwargs):
```

```
            super(Residual, self).__init__(**kwargs)
            if same_shape:
                self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
                self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
                self.conv3 = None
            else:
                self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                                       strides=2)
                self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
                self.conv3 = nn.Conv2D(num_channels, kernel_size=1, strides=2)

            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)
```

否则我们改变输出形状的同时减半输出高宽：

```
In [3]: blk2 = Residual(6, same_shape=False)
        blk2.initialize()
        blk2(x).shape
```

```
Out[3]: (4, 6, 3, 3)
```

5.12.2 ResNet 模型

ResNet 主体是由多个残差块构成。它的构建模式是首先一个减半高宽的残差块，然后接数个保持输入形状的残差块，然后再接一个通道翻倍但高宽减半的残差块。如此重复 4 次。我们先定义一个这样的模式，它在一个减半高宽的残差块后加数个保持形状的残差块：

```
In [4]: def resnet_block(num_channels, num_residuals):
            blk = nn.Sequential()
            for i in range(num_residuals):
                blk.add(Residual(num_channels, same_shape=(i is not 0)))
            return blk
```

下面我们构造一个 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 7×7 卷积层后接步幅为 2 的 3×3 的最大池化层。不同于 GoogLeNet 在后面接 4 个有 Inception 块组成的模块，这里我们使用输出通道数从 64 开始，每次翻倍的由 2 个残差块组成的模块。最后跟 GoogLeNet 一样使用全局平均池化层和全连接层来输出。

因为这里每个模块里有 4 个卷积层（1×1 卷积层不算），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。

```
In [5]: net = nn.Sequential()
        net.add(
            nn.Conv2D(64, kernel_size=7, strides=2, activation='relu'),
            nn.MaxPool2D(pool_size=3, strides=2),
            resnet_block(64, 2),
            resnet_block(128, 2),
            resnet_block(256, 2),
            resnet_block(512, 2),
            nn.GlobalAvgPool2D(),
            nn.Dense(10),
        )
```

主要到每个残差块里我们都将输入直接或者通过简单的 1×1 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 [6]: 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)
```

```
conv5 output shape:      (1, 64, 45, 45)
pool0 output shape:      (1, 64, 22, 22)
sequential1 output shape:        (1, 64, 11, 11)
sequential2 output shape:        (1, 128, 6, 6)
sequential3 output shape:        (1, 256, 3, 3)
sequential4 output shape:        (1, 512, 2, 2)
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。

```
In [7]: 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 0.7534, train acc 0.776, test acc 0.847, time 38.4 sec
epoch 2, loss 0.3296, train acc 0.877, test acc 0.859, time 37.0 sec
epoch 3, loss 0.2708, train acc 0.899, test acc 0.840, time 37.0 sec
epoch 4, loss 0.2296, train acc 0.914, test acc 0.899, time 37.0 sec
epoch 5, loss 0.1916, train acc 0.928, test acc 0.906, time 37.1 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

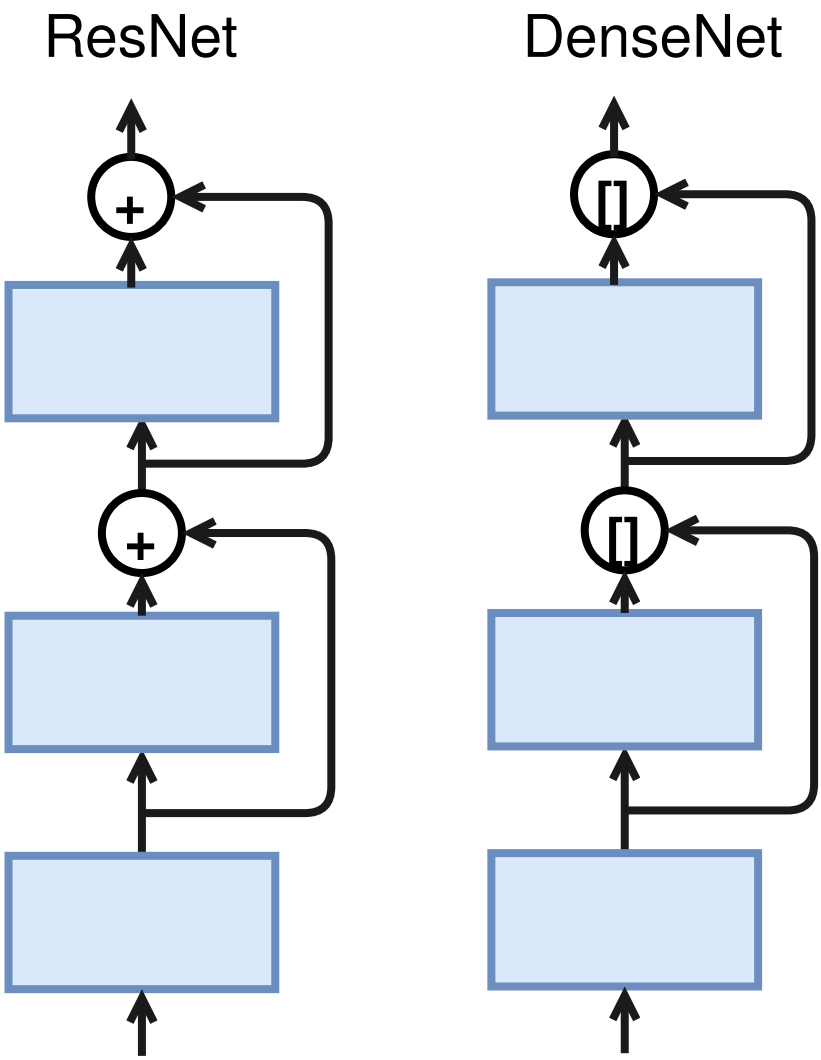
5.12.7 参考文献

[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 稠密连接的网络：DenseNet

ResNet 的跨层连接思想影响了接下来的众多工作。这里我们介绍其中的一个：DenseNet。下图展示了这两个的主要区别：



可以看到 DenseNet 里来自跳层的输出不是通过加法（+）而是拼接（concat）来跟目前层的输出合并。因为是拼接，所以底层的输出会保留的进入上面所有层。这是为什么叫“稠密连接”的原因

5.13.1 稠密块（Dense Block）

我们先来定义一个稠密连接块。DenseNet的卷积块使用ResNet改进版本的 `BN->Relu->Conv`。每个卷积的输出通道数被称之为 `growth_rate`，这是因为假设输出为 `in_channels`，而且有 `layers` 层，那么输出的通道数就是 `in_channels+growth_rate*layers`。

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

        def conv_block(channels):
            out = nn.Sequential()
            out.add(
                nn.BatchNorm(),
                nn.Activation('relu'),
                nn.Conv2D(channels, kernel_size=3, padding=1)
            )
            return out

        class DenseBlock(nn.Block):
            def __init__(self, layers, growth_rate, **kwargs):
                super(DenseBlock, self).__init__(**kwargs)
                self.net = nn.Sequential()
                for i in range(layers):
                    self.net.add(conv_block(growth_rate))

            def forward(self, x):
                for layer in self.net:
                    out = layer(x)
                    x = nd.concat(x, out, dim=1)
                return x
```

我们验证下输出通道数是不是符合预期。

```
In [2]: dblk = DenseBlock(2, 10)
        dblk.initialize()

        x = nd.random.uniform(shape=(4,3,8,8))
        dblk(x).shape

Out[2]: (4, 23, 8, 8)
```

5.13.2 过渡块（Transition Block）

因为使用拼接的缘故，每经过一次拼接输出通道数可能会激增。为了控制模型复杂度，这里引入一个过渡块，它不仅把输入的长宽减半，同时也使用 1×1 卷积来改变通道数。

```
In [3]: def transition_block(channels):
            out = nn.Sequential()
            out.add(
                nn.BatchNorm(),
                nn.Activation('relu'),
                nn.Conv2D(channels, kernel_size=1),
                nn.AvgPool2D(pool_size=2, strides=2)
            )
            return out
```

验证一下结果：

```
In [4]: tblk = transition_block(10)
        tblk.initialize()

        tblk(x).shape
```

```
Out[4]: (4, 10, 4, 4)
```

5.13.3 DenseNet

DenseNet 的主体就是交替串联稠密块和过渡块。它使用全局的 `growth_rate` 使得配置更加简单。过渡层每次都将通道数减半。下面定义一个 121 层的 DenseNet。

```
In [5]: init_channels = 64
        growth_rate = 32
        block_layers = [6, 12, 24, 16]
        num_classes = 10

        def dense_net():
            net = nn.Sequential()
            # add name_scope on the outermost Sequential
            with net.name_scope():
                # first block
                net.add(
                    nn.Conv2D(init_channels, kernel_size=7,
                              strides=2, padding=3),
                    nn.BatchNorm(),
```

```
            nn.Activation('relu'),
            nn.MaxPool2D(pool_size=3, strides=2, padding=1)
        )
        # dense blocks
        channels = init_channels
        for i, layers in enumerate(block_layers):
            net.add(DenseBlock(layers, growth_rate))
            channels += layers * growth_rate
            if i != len(block_layers)-1:
                net.add(transition_block(channels//2))
        # last block
        net.add(
            nn.BatchNorm(),
            nn.Activation('relu'),
            nn.AvgPool2D(pool_size=1),
            nn.Flatten(),
            nn.Dense(num_classes)
        )
    return net
```

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 32×32 来训练。

```
In [6]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import gluon
        from mxnet import init

        train_data, test_data = gb.load_data_fashion_mnist(
            batch_size=64, resize=32)

        ctx = gb.try_gpu()
        net = dense_net()
        net.initialize(ctx=ctx, init=init.Xavier())

        loss = gluon.loss.SoftmaxCrossEntropyLoss()
        trainer = gluon.Trainer(net.collect_params(),
                                'sgd', {'learning_rate': 0.1})
        gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=1)
```

```
training on  gpu(0)
epoch 1, loss 0.4979, train acc 0.825, test acc 0.840, time 87.0 sec
```

5.13.5 小结

- Desnet 通过将 ResNet 里的 `+` 替换成 `concat` 从而获得更稠密的连接。

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，想想为什么？
- 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 224×224（需要改最后的 `AvgPool2D` 大小），来看看实际（GPU）内存消耗。
- 这里的 FashionMNIST 有必要用 100+ 层的网络吗？尝试将其改简单看看效果。

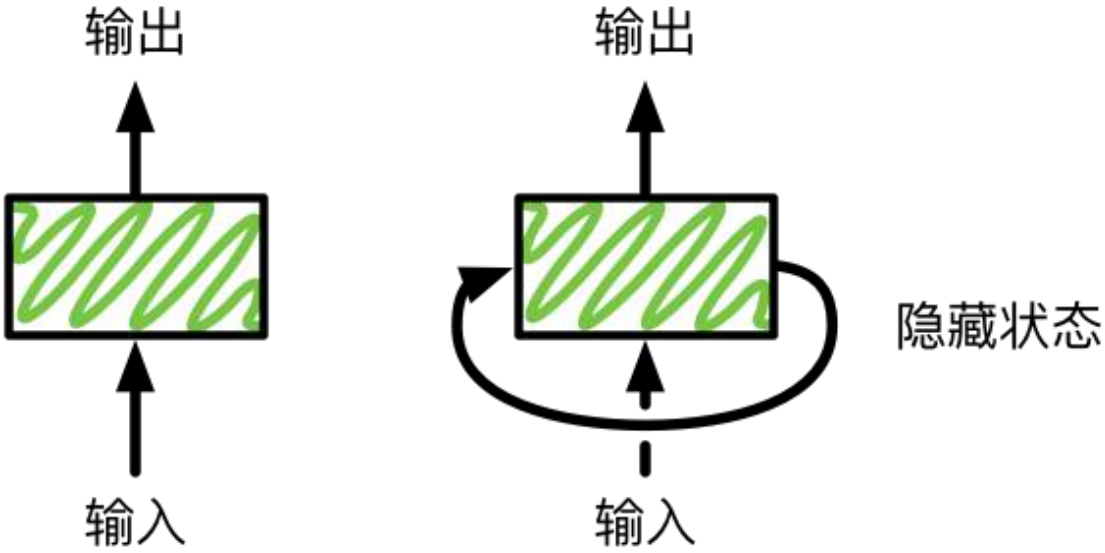
5.13.7 扫码直达讨论区

6

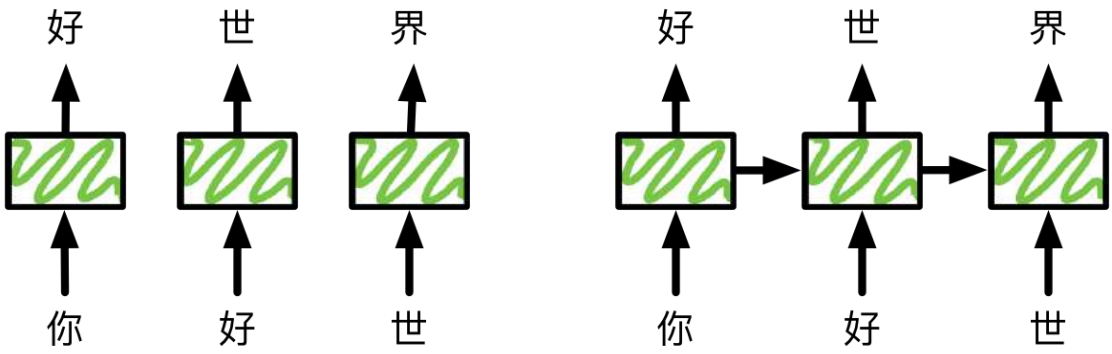
循环神经网络

6.1 循环神经网络——从零开始

前面的教程里我们使用的网络都属于**前馈神经网络**。之所以叫前馈，是因为整个网络是一条链（回想下 `gluon.nn.Sequential`），每一层的结果都是反馈给下一层。这一节我们介绍**循环神经网络**，这里每一层不仅输出给下一层，同时还输出一个**隐含状态**，给当前层在处理下一个样本时使用。下图展示这两种网络的区别。



循环神经网络的这种结构使得它适合处理前后有依赖关系数据样本。我们拿语言模型举个例子来解释这个是怎么工作的。语言模型的任务是给定句子的前 t 个字符，然后预测第 t+1 个字符。假设我们的句子是“你好世界”，使用前馈神经网络来预测的一个做法是，在时间 1 输入“你”，预测”好“，时间 2 向同一个网络输入“好”预测“世”。下图左边展示了这个过程。



注意到一个问题是，当我们预测“世”的时候只给了“好”这个输入，而完全忽略了“你”。直觉上“你”这个词应该对这次的预测比较重要。虽然这个问题通常可以通过 **n-gram** 来缓解，就是说预测第 t+1 个字符的时候，我们输入前 n 个字符。如果 n=1，那就是我们这里用的。我们可以增大 n 来使得输入含有更多信息。但我们不能任意增大 n，因为这样通常带来模型复杂度的增加

从而导致需要大量数据和计算来训练模型。

循环神经网络使用一个隐含状态来记录前面看到的数据来帮助当前预测。上图右边展示了这个过程。在预测“好”的时候，我们输出一个隐含状态。我们用这个状态和新的输入“好”来一起预测“世”，然后同时输出一个更新过的隐含状态。我们希望前面的信息能够保存在这个隐含状态里，从而提升预测效果。

6.1.1 循环神经网络

在对输入输出数据有了解后，我们来正式介绍循环神经网络。

首先回忆一下单隐含层的前馈神经网络的定义，例如多层感知机。假设隐含层的激活函数是 ϕ，对于一个样本数为 n 特征向量维度为 x 的批量数据 $\boldsymbol{X} \in \mathbb{R}^{n \times x}$（$\boldsymbol{X}$ 是一个 n 行 x 列的实数矩阵）来说，那么这个隐含层的输出就是

$$\boldsymbol{H} = \phi(\boldsymbol{X}\boldsymbol{W}_{xh} + \boldsymbol{b}_h)$$

假定隐含层长度为 h,其中的 $\boldsymbol{W}_{xh} \in \mathbb{R}^{x \times h}$ 是权重参数。偏移参数 $\boldsymbol{b}_h \in \mathbb{R}^{1 \times h}$ 在与前一项 $\boldsymbol{X}\boldsymbol{W}_{xh} \in \mathbb{R}^{n \times h}$ 相加时使用了广播。这个隐含层的输出的尺寸为 $\boldsymbol{H} \in \mathbb{R}^{n \times h}$。

把隐含层的输出 $\boldsymbol{H}$ 作为输出层的输入，最终的输出

$$\hat{\boldsymbol{Y}} = \text{softmax}(\boldsymbol{H}\boldsymbol{W}_{hy} + \boldsymbol{b}_y)$$

假定每个样本对应的输出向量维度为 y，其中 $\hat{\boldsymbol{Y}} \in \mathbb{R}^{n \times y}, \boldsymbol{W}_{hy} \in \mathbb{R}^{h \times y}, \boldsymbol{b}_y \in \mathbb{R}^{1 \times y}$ 且两项相加使用了广播。

将上面网络改成循环神经网络，我们首先对输入输出加上时间戳 t。假设 $\boldsymbol{X}_t \in \mathbb{R}^{n \times x}$ 是序列中的第 t 个批量输入（样本数为 n，每个样本的特征向量维度为 x），对应的隐含层输出是隐含状态 $\boldsymbol{H}_t \in \mathbb{R}^{n \times h}$（隐含层长度为 h），而对应的最终输出是 $\hat{\boldsymbol{Y}}_t \in \mathbb{R}^{n \times y}$（每个样本对应的输出向量维度为 y）。在计算隐含层的输出的时候，循环神经网络只需要在前馈神经网络基础上加上跟前一时间 $t-1$ 输入隐含层 $\boldsymbol{H}_{t-1} \in \mathbb{R}^{n \times h}$ 的加权和。为此，我们引入一个新的可学习的权重 $\boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$:

$$\boldsymbol{H}_t = \phi(\boldsymbol{X}_t\boldsymbol{W}_{xh} + \boldsymbol{H}_{t-1}\boldsymbol{W}_{hh} + \boldsymbol{b}_h)$$

输出的计算跟前面一致:

$$\hat{\boldsymbol{Y}}_t = \text{softmax}(\boldsymbol{H}_t\boldsymbol{W}_{hy} + \boldsymbol{b}_y)$$

一开始我们提到过，隐含状态可以认为是这个网络的记忆。该网络中，时刻 t 的隐含状态就是该时刻的隐含层变量 $\boldsymbol{H}_t$。它存储前面时间里面的信息。我们的输出是只基于这个状态。最开始的隐含状态里的元素通常会被初始化为 0。

6.1.2 周杰伦歌词数据集

为了实现并展示循环神经网络，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》所有歌曲的歌词。

下面我们读取这个数据并看看前面 49 个字符（char）是什么样的：

```
In [1]: 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()
```

```
        print(corpus_chars[0:49])
```

```
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
我每天每天每天在想想想想著你
```

我们看一下数据集里的字符数。

```
In [2]: len(corpus_chars)
```

```
Out[2]: 64925
```

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，然后截去后面一段使得接下来的训练会快一点。

```
In [3]: corpus_chars = corpus_chars.replace('\n', ' ').replace('\r', ' ')
        corpus_chars = corpus_chars[0:20000]
```

6.1.3 字符的数值表示

先把数据里面所有不同的字符拿出来做成一个字典：

```
In [4]: 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)

        print('vocab size:', vocab_size)
```

```
vocab size: 1465
```

然后可以把每个字符转成从 0 开始的索引 (index) 来方便之后的使用。

```
In [5]: 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:
 [227, 973, 216, 1275, 699, 105, 411, 227, 973, 196, 1386, 871, 1136, 933, 943, 155,
 ↪  411, 227, 973, 196, 1386, 130, 345, 1138, 1419, 595, 411, 130, 345, 1138, 933,
 ↪  943, 381, 411, 162, 809, 1183, 809, 1183, 809]
```

6.1.4 时序数据的批量采样

同之前一样我们需要每次随机读取一些（batch_size 个）样本和其对用的标号。这里的样本跟前面有点不一样，这里一个样本通常包含一系列连续的字符（前馈神经网络里可能每个字符作为一个样本）。

如果我们把序列长度（num_steps）设成 5，那么一个可能的样本是“想要有直升”。其对应的标号仍然是长为 5 的序列，每个字符是对应的样本里字符的后面那个。例如前面样本的标号就是“要有直升机”。

随机批量采样

下面代码每次从数据里随机采样一个批量。

```
In [6]: import random
        from mxnet import nd

        def data_iter_random(corpus_indices, batch_size, num_steps, ctx=None):
            # 减一是因为 label 的索引是相应 data 的索引加一
            num_examples = (len(corpus_indices) - 1) // num_steps
            epoch_size = num_examples // batch_size
            # 随机化样本
            example_indices = list(range(num_examples))
            random.shuffle(example_indices)

            # 返回 num_steps 个数据
            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]
                data = nd.array(
                    [_data(j * num_steps) for j in batch_indices], ctx=ctx)
                label = nd.array(
```

```
            [_data(j * num_steps + 1) for j in batch_indices], ctx=ctx)
        yield data, label
```

为了便于理解时序数据上的随机批量采样，让我们输入一个从 0 到 29 的人工序列，看下读出来长什么样：

```
In [7]: my_seq = list(range(30))

        for data, label in data_iter_random(my_seq, batch_size=2, num_steps=3):
            print('data: ', data, '\nlabel:', label, '\n')
```

```
data:
[[  9.  10.  11.]
 [  0.   1.   2.]]
<NDArray 2x3 @cpu(0)>
label:
[[ 10.  11.  12.]
 [  1.   2.   3.]]
<NDArray 2x3 @cpu(0)>

data:
[[  6.   7.   8.]
 [ 15.  16.  17.]]
<NDArray 2x3 @cpu(0)>
label:
[[  7.   8.   9.]
 [ 16.  17.  18.]]
<NDArray 2x3 @cpu(0)>

data:
[[ 18.  19.  20.]
 [ 12.  13.  14.]]
<NDArray 2x3 @cpu(0)>
label:
[[ 19.  20.  21.]
 [ 13.  14.  15.]]
<NDArray 2x3 @cpu(0)>

data:
[[ 24.  25.  26.]
 [ 21.  22.  23.]]
<NDArray 2x3 @cpu(0)>
label:
[[ 25.  26.  27.]
```

```
 [ 22.  23.  24.]]
<NDArray 2x3 @cpu(0)>
```

由于各个采样在原始序列上的位置是随机的时序长度为 num_steps 的连续数据点，相邻的两个随机批量在原始序列上的位置不一定相毗邻。因此，在训练模型时，读取每个随机时序批量前需要重新初始化隐含状态。

相邻批量采样

除了对原序列做随机批量采样之外，我们还可以使相邻的两个随机批量在原始序列上的位置相毗邻。

```
In [8]: 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 label 的索引是相应 data 的索引加一
            epoch_size = (batch_len - 1) // num_steps

            for i in range(epoch_size):
                i = i * num_steps
                data = indices[:, i: i + num_steps]
                label = indices[:, i + 1: i + num_steps + 1]
                yield data, label
```

相同地，为了便于理解时序数据上的相邻批量采样，让我们输入一个从 0 到 29 的人工序列，看下读出来长什么样：

```
In [9]: my_seq = list(range(30))

        for data, label in data_iter_consecutive(my_seq, batch_size=2, num_steps=3):
            print('data: ', data, '\nlabel:', label, '\n')
```

```
data:
[[  0.   1.   2.]
 [ 15.  16.  17.]]
<NDArray 2x3 @cpu(0)>
label:
[[  1.   2.   3.]
```

```
 [ 16.  17.  18.]]
<NDArray 2x3 @cpu(0)>

data:
[[  3.   4.   5.]
 [ 18.  19.  20.]]
<NDArray 2x3 @cpu(0)>
label:
[[  4.   5.   6.]
 [ 19.  20.  21.]]
<NDArray 2x3 @cpu(0)>

data:
[[  6.   7.   8.]
 [ 21.  22.  23.]]
<NDArray 2x3 @cpu(0)>
label:
[[  7.   8.   9.]
 [ 22.  23.  24.]]
<NDArray 2x3 @cpu(0)>

data:
[[  9.  10.  11.]
 [ 24.  25.  26.]]
<NDArray 2x3 @cpu(0)>
label:
[[ 10.  11.  12.]
 [ 25.  26.  27.]]
<NDArray 2x3 @cpu(0)>
```

由于各个采样在原始序列上的位置是毗邻的时序长度为 `num_steps` 的连续数据点，因此，使用相邻批量采样训练模型时，读取每个时序批量前，我们需要将该批量最开始的隐含状态设为上个批量最终输出的隐含状态。在同一个 epoch 中，隐含状态只需要在该 epoch 开始的时候初始化。

6.1.5 One-hot 向量

注意到每个字符现在是用一个整数来表示，而输入进网络我们需要一个定长的向量。一个常用的办法是使用 one-hot 来将其表示成向量。也就是说，如果一个字符的整数值是 i，那么我们创建一个全 0 的长为 `vocab_size` 的向量，并将其第 i 位设成 1。该向量就是对原字符的 one-hot 向量。

```
In [10]: nd.one_hot(nd.array([0, 2]), vocab_size)
```

```
Out[10]:
         [[ 1.  0.  0. ...,  0.  0.  0.]
          [ 0.  0.  1. ...,  0.  0.  0.]]
         <NDArray 2x1465 @cpu(0)>
```

记得前面我们每次得到的数据是一个 batch_size * num_steps 的批量。下面这个函数将其转换成 num_steps 个可以输入进网络的 batch_size * vocab_size 的矩阵。对于一个长度为 num_steps 的序列,每个批量输入 $\boldsymbol{X} \in \mathbb{R}^{n \times x}$,其中 $n =$ batch_size,而 $x =$ vocab_size (onehot 编码向量维度)。

```
In [11]: def get_inputs(data):
             return [nd.one_hot(X, vocab_size) for X in data.T]

         inputs = get_inputs(data)

         print('input length: ', len(inputs))
         print('input[0] shape: ', inputs[0].shape)
```

```
input length:  3
input[0] shape:  (2, 1465)
```

6.1.6 初始化模型参数

对于序列中任意一个时间戳，一个字符的输入是维度为 vocab_size 的 one-hot 向量，对应输出是预测下一个时间戳为词典中任意字符的概率，因而该输出是维度为 vocab_size 的向量。

当序列中某一个时间戳的输入为一个样本数为 batch_size (对应模型定义中的 n) 的批量，每个时间戳上的输入和输出皆为尺寸 batch_size * vocab_size (对应模型定义中的 $n \times x$) 的矩阵。假设每个样本对应的隐含状态的长度为 hidden_dim (对应模型定义中隐含层长度 h)，根据矩阵乘法定义，我们可以推断出模型隐含层和输出层中各个参数的尺寸。

```
In [12]: import mxnet as mx

         # 尝试使用 GPU
         import sys
         sys.path.append('..')
         import gluonbook as gb
         ctx = gb.try_gpu()
         print('Will use', ctx)

         input_dim = vocab_size
         # 隐含状态长度
```

```
        hidden_dim = 256
        output_dim = vocab_size
        std = .01

        def get_params():
            # 隐含层
            W_xh = nd.random_normal(scale=std, shape=(input_dim, hidden_dim), ctx=ctx)
            W_hh = nd.random_normal(scale=std, shape=(hidden_dim, hidden_dim),
↪   ctx=ctx)
            b_h = nd.zeros(hidden_dim, ctx=ctx)

            # 输出层
            W_hy = nd.random_normal(scale=std, shape=(hidden_dim, output_dim),
↪   ctx=ctx)
            b_y = nd.zeros(output_dim, 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)
```

6.1.7 定义模型

当序列中某一个时间戳的输入为一个样本数为 batch_size 的批量，而整个序列长度为 num_steps 时，以下 rnn 函数的 inputs 和 outputs 皆为 num_steps 个尺寸为 batch_size * vocab_size 的矩阵，隐含变量 $\boldsymbol{H}$ 是一个尺寸为 batch_size * hidden_dim 的矩阵。该隐含变量 $\boldsymbol{H}$ 也是循环神经网络的隐含状态 state。

我们将前面的模型公式翻译成代码。这里的激活函数使用了按元素操作的双曲正切函数

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

需要注意的是，双曲正切函数的值域是 $[-1, 1]$。如果自变量均匀分布在整个实域，该激活函数输出的均值为 0。

```
In [13]: def rnn(inputs, state, *params):
            # inputs: num_steps 个尺寸为 batch_size * vocab_size 矩阵。
            # H: 尺寸为 batch_size * hidden_dim 矩阵。
            # outputs: num_steps 个尺寸为 batch_size * vocab_size 矩阵。
            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_hy) + b_y
        outputs.append(Y)
    return (outputs, H)
```

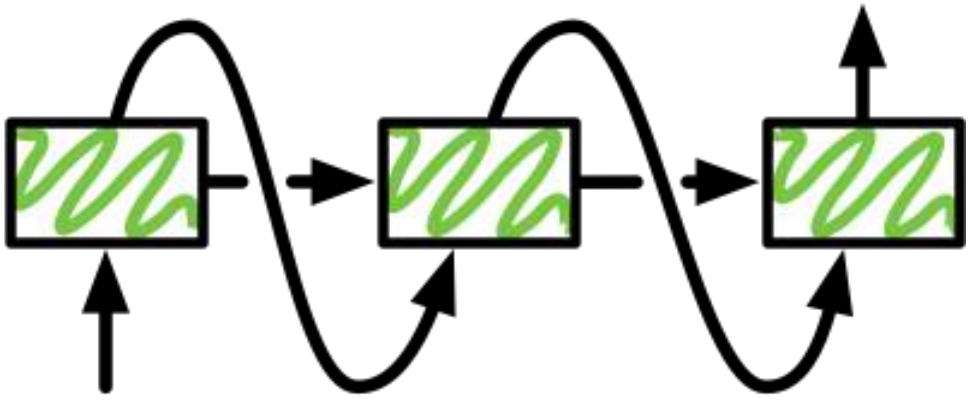
做个简单的测试:

```
In [14]: state = nd.zeros(shape=(data.shape[0], hidden_dim), ctx=ctx)

         params = get_params()
         outputs, state_new = rnn(get_inputs(data.as_in_context(ctx)), state, *params)

         print('output length: ',len(outputs))
         print('output[0] shape: ', outputs[0].shape)
         print('state shape: ', state_new.shape)
```

```
output length:  3
output[0] shape:  (2, 1465)
state shape:  (2, 256)
```

6.1.8 预测序列

在做预测时我们只需要给定时间 0 的输入和起始隐含变量。然后我们每次将上一个时间的输出作为下一个时间的输入。



```
In [15]: def predict_rnn(rnn, prefix, num_chars, params, hidden_dim, ctx, idx_to_char,
                         char_to_idx, get_inputs, is_lstm=False):
```

```
            # 预测以 prefix 开始的接下来的 num_chars 个字符。
            prefix = prefix.lower()
            state_h = nd.zeros(shape=(1, hidden_dim), ctx=ctx)
            if is_lstm:
                # 当 RNN 使用 LSTM 时才会用到，这里可以忽略。
                state_c = nd.zeros(shape=(1, hidden_dim), 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含变量。
                if is_lstm:
                    # 当 RNN 使用 LSTM 时才会用到，这里可以忽略。
                    Y, state_h, state_c = rnn(get_inputs(X), state_h, state_c,
↪  *params)
                else:
                    Y, state_h = rnn(get_inputs(X), 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])
```

6.1.9 梯度剪裁

我们在正向传播和反向传播中提到，训练神经网络往往需要依赖梯度计算的优化算法，例如我们之前介绍的随机梯度下降。而在循环神经网络的训练中，当每个时序训练数据样本的时序长度 num_steps 较大或者时刻 t 较小，目标函数有关 t 时刻的隐含层变量梯度较容易出现衰减（vanishing）或爆炸（explosion）。我们会在下一节详细介绍出现该现象的原因。

为了应对梯度爆炸，一个常用的做法是如果梯度特别大，那么就投影到一个比较小的尺度上。假设我们把所有梯度接成一个向量 $\boldsymbol{g}$，假设剪裁的阈值是 θ，那么我们这样剪裁使得 $\|\boldsymbol{g}\|$ 不会超过 θ：

$$\boldsymbol{g} = \min\left(\frac{\theta}{\|\boldsymbol{g}\|}, 1\right)\boldsymbol{g}$$

```
In [16]: def grad_clipping(params, theta, ctx):
            if theta is not None:
                norm = nd.array([0.0], ctx)
                for p in params:
                    norm += nd.sum(p.grad ** 2)
```

```
    norm = nd.sqrt(norm).asscalar()
    if norm > theta:
        for p in params:
            p.grad[:] *= theta / norm
```

6.1.10 训练模型

下面我们可以还是训练模型。跟前面前置网络的教程比，这里有以下几个不同。

1. 通常我们使用困惑度（Perplexity）这个指标。
2. 在更新前我们对梯度做剪裁。
3. 在训练模型时，对时序数据采用不同批量采样方法将导致隐含变量初始化的不同。

困惑度（Perplexity）

回忆以下我们之前介绍的交叉熵损失函数。在语言模型中，该损失函数即被预测字符的对数似然平均值的相反数：

$$\text{loss} = -\frac{1}{N}\sum_{i=1}^{N}\log p_{\text{target}_i}$$

其中 N 是预测的字符总数，p_{target_i} 是在第 i 个预测中真实的下个字符被预测的概率。

而这里的困惑度可以简单的认为就是对交叉熵做 exp 运算使得数值更好读。

为了解释困惑度的意义，我们先考虑一个完美结果：模型总是把真实的下个字符的概率预测为 1。也就是说，对任意的 i 来说，$p_{\text{target}_i} = 1$。这种完美情况下，困惑度值为 1。

我们再考虑一个基线结果：给定不重复的字符集合 W 及其字符总数 $|W|$，模型总是预测下个字符为集合 W 中任一字符的概率都相同。也就是说，对任意的 i 来说，$p_{\text{target}_i} = 1/|W|$。这种基线情况下，困惑度值为 $|W|$。

最后，我们可以考虑一个最坏结果：模型总是把真实的下个字符的概率预测为 0。也就是说，对任意的 i 来说，$p_{\text{target}_i} = 0$。这种最坏情况下，困惑度值为正无穷。

任何一个有效模型的困惑度值必须小于预测集中元素的数量。在本例中，困惑度必须小于字典中的字符数 $|W|$。如果一个模型可以取得较低的困惑度的值（更靠近 1），通常情况下，该模型预测更加准确。

```
In [17]: from mxnet import autograd
         from mxnet import gluon
         from math import exp

         def train_and_predict_rnn(rnn, is_random_iter, epochs, num_steps, hidden_dim,
                                   learning_rate, clipping_theta, batch_size,
                                   pred_period, pred_len, seqs, 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()

             softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()

             for e in range(1, epochs + 1):
                 # 如使用相邻批量采样，在同一个 epoch 中，隐含变量只需要在该 epoch 开始的时候初始
 ↪ 化。
                 if not is_random_iter:
                     state_h = nd.zeros(shape=(batch_size, hidden_dim), ctx=ctx)
                     if is_lstm:
                         # 当 RNN 使用 LSTM 时才会用到，这里可以忽略。
                         state_c = nd.zeros(shape=(batch_size, hidden_dim), ctx=ctx)
                 train_loss, num_examples = 0, 0
                 for data, label in data_iter(corpus_indices, batch_size, num_steps,
                                              ctx):
                     # 如使用随机批量采样，处理每个随机小批量前都需要初始化隐含变量。
                     if is_random_iter:
                         state_h = nd.zeros(shape=(batch_size, hidden_dim), ctx=ctx)
                         if is_lstm:
                             # 当 RNN 使用 LSTM 时才会用到，这里可以忽略。
                             state_c = nd.zeros(shape=(batch_size, hidden_dim),
 ↪ ctx=ctx)
                     with autograd.record():
                         # outputs 尺寸：(batch_size, vocab_size)
                         if is_lstm:
                             # 当 RNN 使用 LSTM 时才会用到，这里可以忽略。
                             outputs, state_h, state_c = rnn(get_inputs(data), state_h,
                                                             state_c, *params)
                         else:
                             outputs, state_h = rnn(get_inputs(data), state_h, *params)
```

```
                # 设 t_ib_j 为 i 时间批量中的 j 元素:
                # label 尺寸: (batch_size * num_steps)
                # label = [t_0b_0, t_0b_1, ..., t_1b_0, t_1b_1, ..., ]
                label = label.T.reshape((-1,))
                # 拼接 outputs, 尺寸: (batch_size * num_steps, vocab_size)。
                outputs = nd.concat(*outputs, dim=0)
                # 经上述操作, outputs 和 label 已对齐。
                loss = softmax_cross_entropy(outputs, label)
            loss.backward()

            grad_clipping(params, clipping_theta, ctx)
            gb.SGD(params, learning_rate)

            train_loss += nd.sum(loss).asscalar()
            num_examples += loss.size

        if e % pred_period == 0:
            print("Epoch %d. Perplexity %f" % (e,
                                               exp(train_loss/num_examples)))
            for seq in seqs:
                print(' - ', predict_rnn(rnn, seq, pred_len, params,
                      hidden_dim, ctx, idx_to_char, char_to_idx, get_inputs,
                      is_lstm))
            print()
```

以下定义模型参数和预测序列前缀。

```
In [18]: epochs = 200
         num_steps = 35
         learning_rate = 0.1
         batch_size = 32

         softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()

         seq1 = '分开'
         seq2 = '不分开'
         seq3 = '战争中部队'
         seqs = [seq1, seq2, seq3]
```

我们先采用随机批量采样实验循环神经网络谱写歌词。我们假定谱写歌词的前缀分别为“分开”、“不分开”和“战争中部队”。

```
In [19]: train_and_predict_rnn(rnn=rnn, is_random_iter=True, epochs=200, num_steps=35,
                               hidden_dim=hidden_dim, learning_rate=0.2,
                               clipping_theta=5, batch_size=32, pred_period=20,
```

```
                                pred_len=100, seqs=seqs, get_params=get_params,
                                get_inputs=get_inputs, ctx=ctx,
                                corpus_indices=corpus_indices, idx_to_char=idx_to_char,
                                char_to_idx=char_to_idx)
```

```
Epoch 20. Perplexity 225.467972
 - 分开 我不你我不 我不 我不你 一 我不你 我不 我不你 一 我不你 我不 我不你 一 我不你 我不 我不你
 ↪  一 我不你 我不 我不你 一 我不你 我不 我不你 一 我不你 我不 我不你 一 我不你 我不
 - 不分开 我不你我不 我不 我不你 一 我不你 我不 我不你 一 我不你 我不 我不你 一 我不你 我不 我不你
 ↪  一 我不你 我不 我不你 一 我不你 我不 我不你 一 我不你 我不 我不你 一 我不你 我不
 - 战争中部队 我不你我不 我不 我不你 一 我不你 我不 我不你 一 我不你 我不 我不你 一 我不你 我不
 ↪  我不你 一 我不你 我不 我不你 一 我不你 我不 我不你 一 我不你 我不 我不你 一 我不你 我不

Epoch 40. Perplexity 85.662196
 - 分开 一直两 一步两步三 有一种味 我要一直 你已会这样 是一种味 我要一直 你已会这样 是一种味
 ↪  我要一直 你已会这样 是一种味 我要一直 你已会这样 是一种味 我要一直 你已会这样 是一种味 我要一直
 - 不分开 你在我的茶 有一种味 我要一场 你已会这样 是一种味 我要一直 你已会这样 是一种味 我要一直
 ↪  你已会这样 是一种味 我要一直 你已会这样 是一种味 我要一直 你已会这样 是一种味 我要一直 你已会这
 - 战争中部队 (爷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 坏坏的让我疯狂的可

Epoch 60. Perplexity 30.795107
 - 分开 我不要再想你 不知不觉 你已经离开我 不知不觉 我已了这样我 不知不觉 我已了这样我 不知不觉
 ↪  我已了这样我 不知不觉 我已了这样我 不知不觉 我已了这样我 不知不觉 我已了这样我 不知不觉 我已了这
 - 不分开 走没有你的想 一天走剧 你已经美 我怎么这样我 不知不觉 我已了这样我 不知不觉 我已了这样我
 ↪  不知不觉 我已了这样我 不知不觉 我已了这样我 不知不觉 我已了这样我 不知不觉 我已了这样我 不知不觉
 - 战争中部队的可有 想不开 一兽两步三颗四步 连成线背 你只会感到我 我知想觉球我 不知不觉 我已了这样我
 ↪  不知不觉 我已了这样我 不知不觉 我已了这样我 不知不觉 我已了这样我 不知不觉 我已了这样我 不知不觉

Epoch 80. Perplexity 13.374274
 - 分开 我们拳打想 我不要再想 我不能再想 我不能再想 我不能再想 我不能再想 我不能再想 我不能再想
 ↪  我不能再想 我不能再想 我不能再想 我不能再想 我不能再想 我不能再想 我不能再想 我不能再想 我不能再
 - 不分开 走在我经不见 你只会我放说你 是因为我太爱你 是因为我太爱你 我只能你的微笑的天都就好著
 ↪  谁世界不想再你的天 不要走 别怪后一口气 不知记 剩最后一口气 不抹记 剩最后一口气 不抹记
 ↪  剩最后一口气
 - 战争中部队着 爷爷泡的茶 有一种味道叫做家 他牵泡的茶 有一种味道叫做家 他牵泡的茶 有一种味道叫做家
 ↪  他牵泡的茶 有一种味道叫做家 他牵泡的茶 有一种味道叫做家 他牵泡的茶 有一种味道叫做家 他牵泡的茶
 ↪  有一种

Epoch 100. Perplexity 7.174969
 - 分开 我不要你想你开多 别想你 说你怎么看着我 说你 是一场 周a 是你 从i 匚丫 冂一 xi xi xi xi xi
 ↪  xi xi xi xi xi xi xi xi xi xi xi xi xi xi
```

- 不分开 我会不知不好找 你身经你想离开 我也能你心微天 你手是你的离负 没人能说的人多 我会一直好多过
↪ 你已经远 离开 我想会你慢碎睡 我已一直好走过 你已经远远离开 我也会不想分开
↪ 为什么我连分开都迁就着你
- 战争中部队望 我想的你不是 不了我 印你是我都着你 看亮为的太 像一种味道叫做家 他爷泡的茶
↪ 有一种味道叫做家 他爷泡的茶 有一种味道叫做家 他爷泡的茶 有一种味道叫做家 他爷泡的茶
↪ 有一种味道叫做家 他爷泡的茶

```
Epoch 120. Perplexity 4.828117
```
- 分开 你是经 放悟我的想有 像单记容 在平止外的溪边 默默等待 娘你那双个会 一身正剧我的天手
↪ 当取被宽我的想女 没取是过去 我那么很抽护 你在我也见你 谁没有好球 我妈说我的证据 天晶莹的泪滴
↪ 闪烁成回忆
- 不分开 我经不知不好我要的没堂 象过些你 在小村外的溪边 默默等待 娘你那真个会 一身正气 快使用双截棍
↪ 哼哼哈兮 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈
- 战争中部队的石板 没有这过去 我将往事抽离 如果我遇见你是一场悲剧 我想我这辈子注定一个人演戏
↪ 最后再一个是慢在的天窗 我的你快我的想 你怎么雨我带 你 想是你怎单球 我该想要你 我忍着你想
↪ 我不要再想 我不 我

```
Epoch 140. Perplexity 3.481290
```
- 分开 你是经 心悟简的太快 马蹄管壁 在我们 半兽了 的灵魂 翻滚 停止古忍 回无止尽的战争 让我们
↪ 半兽人 的灵魂 单纯 对远古存在的神 用谦卑的身份 我都想 你爱我 想 简！简！单！单！ 爱~~~~~~
- 不分开着我嘲笑你那才难 想要开来你只能伞要 好人在战壕 这着不觉口 一朵一朵因 而底在角落 没爽就直事
↪ 我也不的爱情的那空小 眼地安的旧岸有像清晰风 离不开暴风圈来不及逃 我不能再想 我不要再想 我不 我不
- 战争中部队的流会的忧伤我 古到神 快给我抬起头 有话去对医药箱说 别怪我 别怪我 说你怎么每想日
↪ 这样刀种梦 我的世界将被摧毁 也许颓废也比 累不累 睡不睡 单什么人客） 干什么(客） 说什么(客）
↪ 干单都去我也

```
Epoch 160. Perplexity 2.871796
```
- 分开不好 在到 龙而过滤呀永 龙时 我想说通了鱼 这时 我想离水乀鱼 这时 我想离水乀鱼 这时 我想离水
↪ 不够说这样的坟 单谦卑红 一只会感截记 仁者哈兮 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈兮 快
- 不分开 我会不没有你离我 不是我们 我对多球到 我根你说的人 我知了这样难离下 爱念我 别怪我
↪ 说你怎么我想要 我要下的爱模有样 什么兵器最喜欢 双截棍柔中带刚 想越去河南嵩山 学少林跟武
↪ 我的世界将被摧毁
- 战争中部队的淡淡的古伤 消失的风旧在否褪色 我成在使只经这样的 爱景开 快脸儿斑之你的美
↪ 时间翼备之敌一句 爷牵泡的茶 听幅泼墨的山水画 唐朝千年的风沙 现在还在刮 爷爷泡的身边 然蹄在风琴
↪ 用静的人丽 你的完

```
Epoch 180. Perplexity 2.525899
```
- 分开 你说全 一步到步又步四步望著天 一只海 是颗四颗年江 暗是记传里 还灵魂 一果是 我给 老远 我不
↪ 我不 我不能再爱 我不 我不 我不能再想你 不知不觉 你已经离开我 不知不觉 我跟了这节奏 后知后觉
- 不分开期 然后是过去 就着人的在情 还是你说离很多 我 这样神强单曲 像这样里 别溪边乀不记 加字过去
↪ 像溪边乀田蝇 加字过去 像溪边乀田蝇 加字过去 像溪边乀田蝇 加字过去 像溪边乀田蝇 加字过去 像溪边

```
 - 战争中部队望才想想回你 从是在最时间大你丿字 你那全然龙没讯息 你那全然龙没讯息 像往南方燕子断翅
 ↪ 我说儿神 试颗心悬的半用 我都要你慢微没人帮好到 小灰在最我的左不要换黑 泪着远 不是 暗
 ↪ 连是你只念恨 我 不

Epoch 200. Perplexity 2.300685
 - 分开不再 回连泡木里 干时 在远的是我的红 相怎黑雨的路 印地安老斑鸠 平常话不多
 ↪ 除非是乌鸦抢了它的窝 它在灌木丛 它一双美 是何心前的出面一口 我亲揍使来看么 我会说陪我很糗
 ↪ 是身经 有多 是我讲
 - 不分开 已经 没发意义你的日老 闭著被宽 只晨下蜘蛛 辛辛苦苦 全家怕日出 白色蜡烛 温暖了空出
 ↪ 白色蜡你 全家是人屋地 我右拳起淋远 我就着我 说你有些犹豫 怎么我留出开了 就什么我已经你都迁就着
 ↪ 你和还没
 - 战争中部队家会好 睡故玩界著一句敌鸥 在谁谷 穿梭到间的画面的钟 从反方向开始移动 回到当午的点吻
 ↪ 现在还白刮 我学往很抽走 我爱你在你牵是我 这亮你笑手 一口 有因连 让过的旧 有你完过 我已好一条人
 ↪ 我知不起
```

我们再采用相邻批量采样实验循环神经网络谱写歌词。

```
In [20]: train_and_predict_rnn(rnn=rnn, is_random_iter=False, epochs=200, num_steps=35,
                               hidden_dim=hidden_dim, learning_rate=0.2,
                               clipping_theta=5, batch_size=32, pred_period=20,
                               pred_len=100, seqs=seqs, get_params=get_params,
                               get_inputs=get_inputs, ctx=ctx,
                               corpus_indices=corpus_indices, idx_to_char=idx_to_char,
                               char_to_idx=char_to_idx)
```

```
Epoch 20. Perplexity 213.182282
 - 分开 我不的你 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可
 ↪ 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可
 - 不分开 我不的你 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可
 ↪ 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可
 - 战争中部队 我不的你 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可
 ↪ 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可 我不的可
 ↪ 我不的可

Epoch 40. Perplexity 69.431232
 - 分开 你想是你 你不会 我想 这不 我不要再不 我不能再想 我不要再想 我不要我想 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
 - 不分开 我想你这样 我不要再你 我不要我想 我不能再想 我不要我想 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
 - 战争中部队 我不你再想 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
 ↪ 我不要再想 我不要再

Epoch 60. Perplexity 22.679852
```

- 分开 说静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 坏坏的让我疯狂的可
- 不分开你 这不是 小覆默 蜕变的公式我学不来 我想在这样龙 我就着我 说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
- 战争中部队（在你有不是我要的见堂景象 爱沦假象 你只会感到更加沮丧）（埋在这不是我要的天堂景象
  ↪ 沉沦假象 你只会感到更加沮丧）（埋在这不是我要的天堂景象 沉沦假象 你只会感到更加沮丧）
  ↪ （埋在这不是我要的天

```
Epoch 80. Perplexity 9.809241
```
- 分开 说你在这里 这没的手丽 你我的眼球 在我想 半景人 的灵魂 单纯 停止忿存 永忆止尽的战争 让我们
  ↪ 半兽人 的灵魂 单纯 对止忿存 永忆止尽的战争 让我们 半兽人 的灵魂 单纯 对止忿存 永忆止尽的战
- 不分开难的 这什么的你有在暗了卷风 离不开着我已无不可 我已不能你 我的有界将瘦 太不着你
  ↪ 不使经双截棍 哼哼哈兮 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈兮
  ↪ 快使用双截棍 哼哼哈
- 战争中部队家 这不到这样的我 永亲了没有 我的世界将瘦 如彻入秋 漫天黄双截棍 哼哼哈兮 快使用双截棍
  ↪ 哼哼哈兮 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈兮 快使用双截棍
  ↪ 哼哼哈兮 快

```
Epoch 100. Perplexity 5.343990
```
- 分开 快亮的没板 一直在气留 爱是我不想你 一场都你留你 经去着不多 你爱在我的证据 让晶莹的泪滴
  ↪ 闪烁成回忆 你手在我想 再人三着羞 我只不再你说的画道 我想揍你已经没人 你想躲过你 让我的 这是我
- 不分开难不 我心无你心碎没人帮你擦眼泪 别离开是边拥要我们感觉对 我害怕你心碎没人帮你擦眼泪
  ↪ 别离开是边拥要我们感觉对 我害怕你心碎没人帮你擦眼泪 别离开是边拥要我们感觉对
  ↪ 我害怕你心碎没人帮你擦眼泪 别离开
- 战争中部队家 我有你打开 一种一天事说 为不着我 你已经离我习 你在等 分不是 一场两步三步四步望著天
  ↪ 看星星 一颗两颗三颗四颗 连成线背著背默默许下心愿 看远方的星 有一种味道叫做家 他羽泡的茶
  ↪ 像说泼墨的山水

```
Epoch 120. Perplexity 3.643290
```
- 分开 如这不觉 经人在一个秋 我该好打开了天 化身为龙 那大地心脏汹涌 不安跳动 全世界
  ↪ 的表情只剩下一种 等待英雄 我的是口 你只不会我 印你那美你想你 想要好渐濡目的怒 时默 不容半张
  ↪ 你真会中的画争
- 不分开难不 我看不再心 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 我不要 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
- 战争中部队家相放 古 简血吹只单！单福 不到当没的画 篮框变著一高 爬过的那棵树 又何时变多都通
  ↪ 这样个人 我试我感已经猜了我 能好好暗 我想已难已我 陪你已觉不我 选 这不简友单走 这在古没有忧
  ↪ 我想念 分不是

```
Epoch 140. Perplexity 2.933824
```

- 分开 如这不觉 你的灵魂 又了不同 在不再 的日情 剩滚 收止忿恨 永无止尽的战争 让我们 半兽人 的灵魂
  ↪ 单纯 对远古存在的神 用谦卑的客 有一泼味道叫做家 他满头白发 喝茶时不准说话 陆羽泡的茶 像幅泼
- 不分开就走 我才就再心牵的你和你和那到 你只那这样开 你以着我 说你说 分数怎么停留 一直在停留
  ↪ 谁让它停留的 为什么我女你 经去按觉解的在 面想的真空边 我右等起生活 不知不觉不为 你爱你也难不是我
  ↪ 不想不
- 战争中部队 (说你 彷佛载 蜕变的公式我学不来 (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)
  ↪ (难道这不是我要的天堂景象 沉沦假象 你只会

```
Epoch 160. Perplexity 2.575358
```
- 分开了天在走小著 古 我有你想单！单！ 爱笑着说 其就是那安 真没配壶 让蟑螂蜘 辛对确在像动的想息
  ↪ 我不到你的世不重帮都能看到 心 开人简不要车单！ 爱心~~~~ 我不念起国 就什么不难过 景伤妙传出
- 不分开就走 这样慢天的那的爱言著 我要一个一步往你 我想怕这样牵着你 不想你这不觉分 一晚堂着孤
  ↪ 有太多人太 睡于 话我 说欢的字旧 喜欢天人潮中 只就去在的画剩下回忆 可爱还有 离来的钥匙我找不到
  ↪ 他在糖
- 战争中部队 爷说泡人） 如果我遇见你是一场悲剧 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

```
Epoch 180. Perplexity 2.242369
```
- 分开了暗在家戏放 古不 我们後喝难的真奏鼠到y 那我当谁世界看的边色像 低过可见多不会 让我忘了你是我
  ↪ 甩开球 快给我抬起头 有话去对医药箱说 别怪我 别怪我 说你怎么面对我 甩开球我满腔的怒火 我想揍你已
- 不分开就不 这手的最在天 周杰的老栈的 我在你已表现的非常明白 我懂我也知道 你没有舍不得
  ↪ 你说你没会难过我不相信 牵着你陪着我 也只是曾经 希望他是真的比我还要爱你 我才会逼自己天开
  ↪ 你要我说多难堪 我根本
- 战争中部队色戏淡 想不到 所覆开 蜕让我 别分怎么旧江 这抹记 剩抹记 一步两种 江一页动六页第
  ↪ 我满耍的让 有一种味道叫做家 他羽头剔它 口茶时不准 印人在角落 蜥蜴横著走 这里什么奇 有话你要说
  ↪ 别人在篮天

```
Epoch 200. Perplexity 2.077573
```
- 分开变停这家戏我看到声 消过不是我只要回 当真的我 将你那心 你一下受气 我们爱想有 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 我不能再想 我不 我不 我不能
- 不分开天走 我爱就这样牵着你的手不放开 爱可不可以永简单单没有伤害 你 靠着我的肩膀 你 在我胸口睡著
  ↪ 像这样的生活 我爱你 你爱我 想 简！简！单！ 这样抢我 全场了中 你只会功 我有多难忆 (的茶应 我想
- 战争中部队色身放 你知道想爸要 但一伦在我的寻场 换取被宽 清晨那安安意 我右拳打开了天 化身为龙
  ↪ 把山地心新汹涌 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
  ↪ 把山地心

可以看到一开始学到简单的字符，然后简单的词，接着是复杂点的词，然后看上去似乎像个句子了。

6.1.11 小结

- 通过隐含状态，循环神经网络适合处理前后有依赖关系时序数据样本。
- 对前后有依赖关系时序数据样本批量采样时，我们可以使用随机批量采样和相邻批量采样。
- 循环神经网络较容易出现梯度衰减和爆炸。

6.1.12 练习

- 调调参数（例如数据集大小、序列长度、隐含状态长度和学习率），看看对运行时间、perplexity和预测的结果造成的影响。
- 在随机批量采样中，如果在同一个 epoch 中只把隐含变量在该 epoch 开始的时候初始化会怎么样？

6.1.13 扫码直达讨论区

6.2 通过时间反向传播

在上一章循环神经网络的示例代码中，如果不使用梯度裁剪，模型将无法正常训练。为了深刻理解这一现象，并激发改进循环神经网络的灵感，本节我们将介绍循环神经网络中模型梯度的计算和存储，也即**通过时间反向传播**（back-propagation through time）。

我们在正向传播和反向传播中以 L_2 范数正则化的多层感知机为例，介绍了深度学习模型梯度的计算和存储。事实上，所谓通过时间反向传播只是反向传播在循环神经网络的具体应用。我们只需将循环神经网络按时间展开，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梯度。

为了解释通过时间反向传播，我们以一个简单的循环神经网络为例。

6.2.1 模型定义

给定一个输入为 $\boldsymbol{x}_t \in \mathbb{R}^x$（每个样本输入向量长度为 x）和对应真实值为 $y_t \in \mathbb{R}$ 的时序数据训练样本（$t = 1, 2, \ldots, T$ 为时刻），不考虑偏差项，我们可以得到隐含层变量的表达式

$$\boldsymbol{h}_t = \phi(\boldsymbol{W}_{hx}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1})$$

其中 $\boldsymbol{h}_t \in \mathbb{R}^h$ 是向量长度为 h 的隐含层变量，$\boldsymbol{W}_{hx} \in \mathbb{R}^{h\times x}$ 和 $\boldsymbol{W}_{hh} \in \mathbb{R}^{h\times h}$ 是隐含层模型参数。使用隐含层变量和输出层模型参数 $\boldsymbol{W}_{yh} \in \mathbb{R}^{y\times h}$，我们可以得到相应时刻的输出层变量 $\boldsymbol{o}_t \in \mathbb{R}^y$。不考虑偏差项，

$$\boldsymbol{o}_t = \boldsymbol{W}_{yh}\boldsymbol{h}_t$$

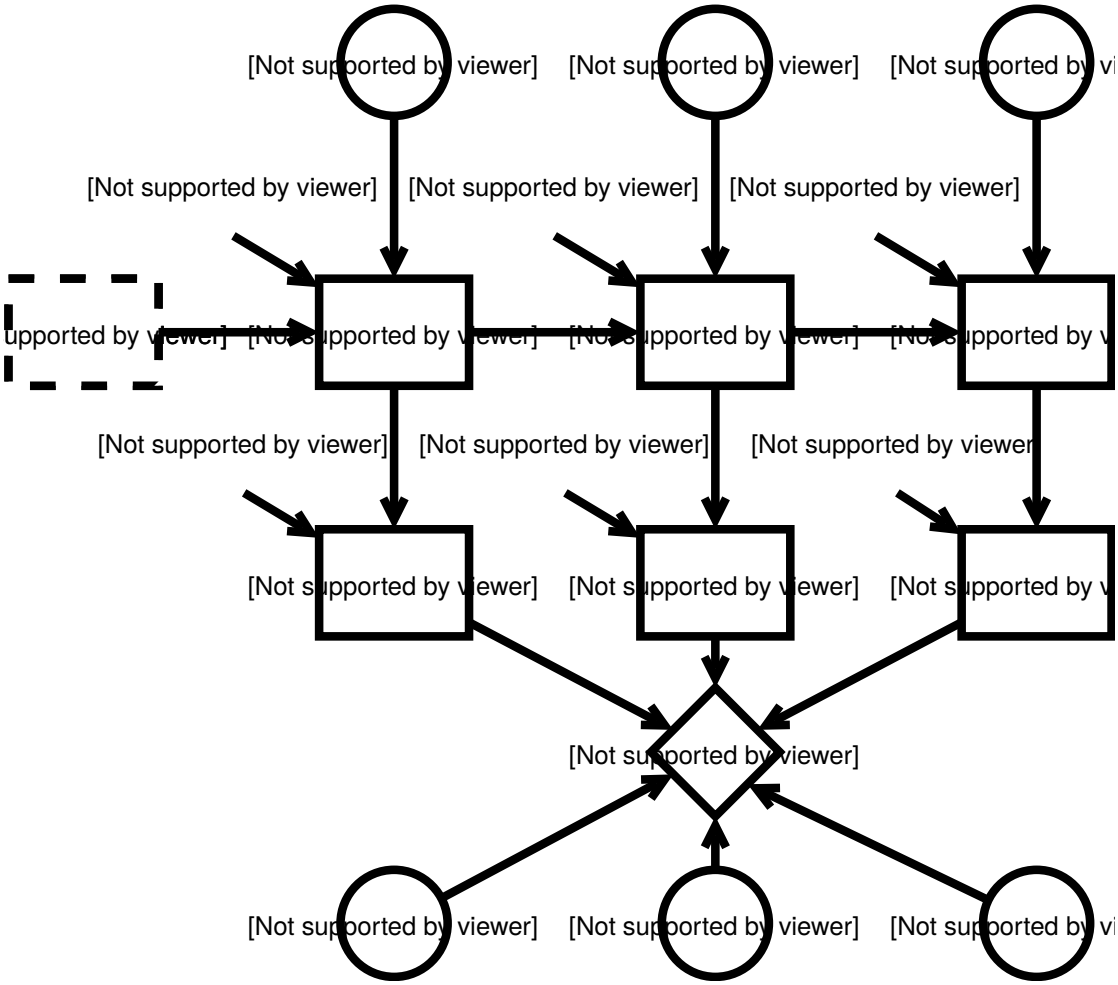
给定每个时刻损失函数计算公式 ℓ，长度为 T 的整个时序数据的损失函数 L 定义为

$$L = \frac{1}{T}\sum_{t=1}^{T} \ell(\boldsymbol{o}_t, y_t)$$

这也是模型最终需要被优化的目标函数。

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，我们可以绘制计算图。我们以时序长度 $T = 3$ 为例。



梯度的计算与存储

在上图中，模型的参数是 $\boldsymbol{W}_{hx}$、$\boldsymbol{W}_{hh}$ 和 $\boldsymbol{W}_{yh}$。为了在模型训练中学习这三个参数，以随机梯度下降为例，假设学习率为 η，我们可以通过

$$\boldsymbol{W}_{hx} = \boldsymbol{W}_{hx} - \eta \frac{\partial L}{\partial \boldsymbol{W}_{hx}}$$

$$\boldsymbol{W}_{hh} = \boldsymbol{W}_{hh} - \eta \frac{\partial L}{\partial \boldsymbol{W}_{hh}}$$

$$\boldsymbol{W}_{yh} = \boldsymbol{W}_{yh} - \eta \frac{\partial L}{\partial \boldsymbol{W}_{yh}}$$

来不断迭代模型参数的值。因此我们需要模型参数梯度 $\partial L/\partial \boldsymbol{W}_{hx}$、$\partial L/\partial \boldsymbol{W}_{hh}$ 和 $\partial L/\partial \boldsymbol{W}_{yh}$。为此，我们可以按照反向传播的次序依次计算并存储梯度。

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

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

来表达链式法则。以下依次计算得到的梯度将依次被存储。

首先，目标函数有关各时刻输出层变量的梯度 $\partial L/\partial \boldsymbol{o}_t \in \mathbb{R}^y$ 可以很容易地计算

$$\frac{\partial L}{\partial \boldsymbol{o}_t} = \frac{\partial \ell(\boldsymbol{o}_t, y_t)}{T \cdot \partial \boldsymbol{o}_t}$$

事实上，这时我们已经可以计算目标函数有关模型参数 $\boldsymbol{W}_{yh}$ 的梯度 $\partial L/\partial \boldsymbol{W}_{yh} \in \mathbb{R}^{y \times h}$。需要注意的是，在计算图中，$\boldsymbol{W}_{yh}$ 可以经过 $\boldsymbol{o}_1, \ldots, \boldsymbol{o}_T$ 通向 L，依据链式法则，

$$\frac{\partial L}{\partial \boldsymbol{W}_{yh}} = \sum_{t=1}^{T} \text{prod}(\frac{\partial L}{\partial \boldsymbol{o}_t}, \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{W}_{yh}}) = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{o}_t} \boldsymbol{h}_t^\top$$

其次，我们注意到隐含层变量之间也有依赖关系。对于最终时刻 T，在计算图中，隐含层变量 $\boldsymbol{h}_T$ 只经过 $\boldsymbol{o}_T$ 通向 L。因此我们先计算目标函数有关最终时刻隐含层变量的梯度 $\partial L/\partial \boldsymbol{h}_T \in \mathbb{R}^h$。依据链式法则，我们得到

$$\frac{\partial L}{\partial \boldsymbol{h}_T} = \text{prod}(\frac{\partial L}{\partial \boldsymbol{o}_T}, \frac{\partial \boldsymbol{o}_T}{\partial \boldsymbol{h}_T}) = \boldsymbol{W}_{yh}^\top \frac{\partial L}{\partial \boldsymbol{o}_T}$$

为了简化计算，我们假设激活函数 $\phi(x) = x$。接下来，对于时刻 $t < T$，在计算图中，由于 $\boldsymbol{h}_t$ 可以经过 $\boldsymbol{h}_{t+1}$ 和 $\boldsymbol{o}_t$ 通向 L，依据链式法则，目标函数有关隐含层变量的梯度 $\partial L/\partial \boldsymbol{h}_t \in \mathbb{R}^h$ 需要按照时刻从晚到早依次计算：

$$\frac{\partial L}{\partial \boldsymbol{h}_t} = \text{prod}(\frac{\partial L}{\partial \boldsymbol{h}_{t+1}}, \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t}) + \text{prod}(\frac{\partial L}{\partial \boldsymbol{o}_t}, \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{h}_t}) = \boldsymbol{W}_{hh}^\top \frac{\partial L}{\partial \boldsymbol{h}_{t+1}} + \boldsymbol{W}_{yh}^\top \frac{\partial L}{\partial \boldsymbol{o}_t}$$

将递归公式展开，对任意 $1 \leq t \leq T$，我们可以得到目标函数有关隐含层变量梯度的通项公式

$$\frac{\partial L}{\partial \boldsymbol{h}_t} = \sum_{i=t}^{T} (\boldsymbol{W}_{hh}^\top)^{T-i} \boldsymbol{W}_{yh}^\top \frac{\partial L}{\partial \boldsymbol{o}_{T+t-i}}$$

由此可见，当每个时序训练数据样本的时序长度 T 较大或者时刻 t 较小，目标函数有关隐含层变量梯度较容易出现**衰减**（valishing）和**爆炸**（explosion）。想象一下 2^{30} 和 0.5^{30} 会有多大。

有了各时刻隐含层变量的梯度之后，我们可以计算隐含层中模型参数的梯度 $\partial L/\partial \boldsymbol{W}_{hx} \in \mathbb{R}^{h \times x}$ 和 $\partial L/\partial \boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$。在计算图中，它们都可以经过 $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T$ 通向 L。依据链式法则，我们有

$$\frac{\partial L}{\partial \boldsymbol{W}_{hx}} = \sum_{t=1}^{T} \text{prod}(\frac{\partial L}{\partial \boldsymbol{h}_t}, \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}_{hx}}) = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{h}_t} \boldsymbol{x}_t^\top$$

$$\frac{\partial L}{\partial \boldsymbol{W}_{hh}} = \sum_{t=1}^{T} \text{prod}(\frac{\partial L}{\partial \boldsymbol{h}_t}, \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}_{hh}}) = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{h}_t} \boldsymbol{h}_{t-1}^\top$$

在正向传播和反向传播中我们解释过，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，由于输出层变量梯度 $\partial L/\partial \boldsymbol{h}_t$ 被计算存储，反向传播稍后的参数梯度 $\partial L/\partial \boldsymbol{W}_{hx}$ 和隐含层变量梯度 $\partial L/\partial \boldsymbol{W}_{hh}$ 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。

还有需要注意的是，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和参数的当前值。举例来说，参数梯度 $\partial L/\partial \boldsymbol{W}_{hh}$ 的计算需要依赖隐含层变量在时刻 $t = 1, \ldots, T-1$ 的当前值 $\boldsymbol{h}_t$（$\boldsymbol{h}_0$ 是初始化得到的）。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。

小结

- 所谓通过时间反向传播只是反向传播在循环神经网络的具体应用。
- 当每个时序训练数据样本的时序长度 T 较大或者时刻 t 较小，目标函数有关隐含层变量梯度较容易出现衰减和爆炸。

练习

- 在循环神经网络中，梯度裁剪是否对梯度衰减和爆炸都有效？
- 你还能想到别的什么方法可以应对循环神经网络中的梯度衰减和爆炸现象？

扫码直达讨论区

6.3 门控循环单元（GRU）——从零开始

上一节中，我们介绍了循环神经网络中的梯度计算方法。我们发现，循环神经网络的隐含层变量梯度可能会出现衰减或爆炸。虽然梯度裁剪可以应对梯度爆炸，但无法解决梯度衰减的问题。因此，给定一个时间序列，例如文本序列，循环神经网络在实际中其实较难捕捉两个时刻距离较大的文本元素（字或词）之间的依赖关系。

门控循环神经网络（gated recurrent neural networks）的提出，是为了更好地捕捉时序数据中间隔较大的依赖关系。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。它由 Cho、van Merrienboer、Bahdanau 和 Bengio 在 2014 年被提出。

6.3.1 门控循环单元

我们先介绍门控循环单元的构造。它比循环神经网络中的隐含层构造稍复杂一点。

重置门和更新门

门控循环单元的隐含状态只包含隐含层变量 $\boldsymbol{H}$。假定隐含状态长度为 h，给定时刻 t 的一个样本数为 n 特征向量维度为 x 的批量数据 $\boldsymbol{X}_t \in \mathbb{R}^{n\times x}$ 和上一时刻隐含状态 $\boldsymbol{H}_{t-1} \in \mathbb{R}^{n\times h}$，重置门（reset gate）$\boldsymbol{R}_t \in \mathbb{R}^{n\times h}$ 和更新门（update gate）$\boldsymbol{Z}_t \in \mathbb{R}^{n\times h}$ 的定义如下：

$$\boldsymbol{R}_t = \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xr} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hr} + \boldsymbol{b}_r)$$

$$\boldsymbol{Z}_t = \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xz} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hz} + \boldsymbol{b}_z)$$

其中的 $\boldsymbol{W}_{xr}, \boldsymbol{W}_{xz} \in \mathbb{R}^{x\times h}$ 和 $\boldsymbol{W}_{hr}, \boldsymbol{W}_{hz} \in \mathbb{R}^{h\times h}$ 是可学习的权重参数，$\boldsymbol{b}_r, \boldsymbol{b}_z \in \mathbb{R}^{1\times h}$ 是可学习的偏移参数。函数 σ 自变量中的三项相加使用了广播。

需要注意的是，重置门和更新门使用了值域为 $[0, 1]$ 的函数 $\sigma(x) = 1/(1+\exp(-x))$。因此，重置门 $\boldsymbol{R}_t$ 和更新门 $\boldsymbol{Z}_t$ 中每个元素的值域都是 $[0, 1]$。

候选隐含状态

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含状态中信息的流动：这通常可以应用按元素乘法符 $\odot$。门控循环单元中的候选隐含状态 $\tilde{\boldsymbol{H}}_t \in \mathbb{R}^{n\times h}$ 使用了值域在 $[-1, 1]$ 的双曲正

切函数 tanh 做激活函数：

$$\tilde{\boldsymbol{H}}_t = \tanh(\boldsymbol{X}_t \boldsymbol{W}_{xh} + \boldsymbol{R}_t \odot \boldsymbol{H}_{t-1} \boldsymbol{W}_{hh} + \boldsymbol{b}_h)$$

其中的 $\boldsymbol{W}_{xh} \in \mathbb{R}^{x \times h}$ 和 $\boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$ 是可学习的权重参数，$\boldsymbol{b}_h \in \mathbb{R}^{1 \times h}$ 是可学习的偏移参数。

需要注意的是，候选隐含状态使用了重置门来控制包含过去时刻信息的上一个隐含状态的流入。如果重置门近似 0，上一个隐含状态将被丢弃。因此，重置门提供了丢弃与未来无关的过去隐含状态的机制。

隐含状态

隐含状态 $\boldsymbol{H}_t \in \mathbb{R}^{n \times h}$ 的计算使用更新门 $\boldsymbol{Z}_t$ 来对上一时刻的隐含状态 $\boldsymbol{H}_{t-1}$ 和当前时刻的候选隐含状态 $\tilde{\boldsymbol{H}}_t$ 做组合，公式如下：

$$\boldsymbol{H}_t = \boldsymbol{Z}_t \odot \boldsymbol{H}_{t-1} + (1 - \boldsymbol{Z}_t) \odot \tilde{\boldsymbol{H}}_t$$

需要注意的是，更新门可以控制过去的隐含状态在当前时刻的重要性。如果更新门一直近似 1，过去的隐含状态将一直通过时间保存并传递至当前时刻。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。

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序数据中短期的依赖关系。
- 更新门有助于捕捉时序数据中长期的依赖关系。

输出层的设计可参照循环神经网络中的描述。

6.3.2 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中介绍。

数据处理

我们先读取并对数据集做简单处理。

```
In [1]: 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)
        print('vocab size:', vocab_size)
```

```
vocab size: 1465
```

我们使用 onehot 来将字符索引表示成向量。

```
In [2]: def get_inputs(data):
            return [nd.one_hot(X, vocab_size) for X in data.T]
```

初始化模型参数

以下部分对模型参数进行初始化。参数 `hidden_dim` 定义了隐含状态的长度。

```
In [3]: import mxnet as mx

        # 尝试使用 GPU
        import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import nd
        ctx = gb.try_gpu()
        print('Will use', ctx)

        input_dim = vocab_size
        # 隐含状态长度
        hidden_dim = 256
        output_dim = vocab_size
        std = .01
```

```
    def get_params():
        # 隐含层
        W_xz = nd.random_normal(scale=std, shape=(input_dim, hidden_dim), ctx=ctx)
        W_hz = nd.random_normal(scale=std, shape=(hidden_dim, hidden_dim), ctx=ctx)
        b_z = nd.zeros(hidden_dim, ctx=ctx)

        W_xr = nd.random_normal(scale=std, shape=(input_dim, hidden_dim), ctx=ctx)
        W_hr = nd.random_normal(scale=std, shape=(hidden_dim, hidden_dim), ctx=ctx)
        b_r = nd.zeros(hidden_dim, ctx=ctx)

        W_xh = nd.random_normal(scale=std, shape=(input_dim, hidden_dim), ctx=ctx)
        W_hh = nd.random_normal(scale=std, shape=(hidden_dim, hidden_dim), ctx=ctx)
        b_h = nd.zeros(hidden_dim, ctx=ctx)

        # 输出层
        W_hy = nd.random_normal(scale=std, shape=(hidden_dim, output_dim), ctx=ctx)
        b_y = nd.zeros(output_dim, 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
```

```
Will use gpu(0)
```

6.3.3 定义模型

我们将前面的模型公式翻译成代码。

```
In [4]: def gru_rnn(inputs, H, *params):
            # inputs: num_steps 个尺寸为 batch_size * vocab_size 矩阵
            # H: 尺寸为 batch_size * hidden_dim 矩阵
            # outputs: num_steps 个尺寸为 batch_size * vocab_size 矩阵
            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

下面我们开始训练模型。我们假定谱写歌词的前缀分别为“分开”、“不分开”和“战争中部队”。这里采用的是相邻批量采样实验门控循环单元谱写歌词。

```
In [5]: seq1 = '分开'
        seq2 = '不分开'
        seq3 = '战争中部队'
        seqs = [seq1, seq2, seq3]

        gb.train_and_predict_rnn(rnn=gru_rnn, is_random_iter=False, epochs=200,
                                 num_steps=35, hidden_dim=hidden_dim,
                                 learning_rate=0.2, clipping_norm=5,
                                 batch_size=32, pred_period=20, pred_len=100,
                                 seqs=seqs, get_params=get_params,
                                 get_inputs=get_inputs, ctx=ctx,
                                 corpus_indices=corpus_indices,
                                 idx_to_char=idx_to_char, char_to_idx=char_to_idx)
```

```
Epoch 20. Training perplexity 273.012025
 - 分开 我不的我的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我
 ↪ 的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我
 ↪ 的你的我的你
 - 不分开 我不的我的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你
 ↪ 我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你
 ↪ 我的你的我的你
 - 战争中部队 我不的我的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我
 ↪ 你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我的你的我
 ↪ 你的我的你的我的你

Epoch 40. Training perplexity 102.791829
 - 分开 我想要你的爱我 想不要 别兽人 的灵魂 xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi
 ↪  xi xi xi xi xi xi xi xi xi xi xi
 - 不分开 我想要你的爱我 想不要 别兽人 的灵魂 xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi
 ↪  xi xi xi xi xi xi xi xi xi xi xi xi
 - 战争中部队 我想你的爱我 想不要 别人 半丫 xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi xi
 ↪  xi xi xi xi xi xi xi xi xi xi xi xi

Epoch 60. Training perplexity 27.989152
 - 分开 一直一直 三炭 一直丶 三炭 一直走 三炭 一直筐 三炭 一直筐 三炭 一直筐 三炭 一直筐 三炭
 ↪  一直筐 三炭 一直筐 三炭 一直筐 三炭 一直筐 三炭 一直筐 三炭 一直筐 三炭 一直筐 三炭 一
 - 不分开了 为时了我们不开 让你在那里 的灵魂 翻滚 一直四 三炭 一直四 三炭 一直走 三炭 一直筐 三炭
 ↪  一直筐 三炭 一直筐 三炭 一直筐 三炭 一直筐 三炭 一直筐 三炭 一直筐 三炭 一直筐 三炭 一
```

- 战争中部队 他在风 被覆盖 蜕变的公式我学不来 (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) (难道这样子 我的天界被太摧 想要的脸 我想要的可爱女人
↪ 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱

Epoch 80. Training perplexity 7.883757
- 分开了口不来我 脑怪我 不不再想你 不知不觉 我跟了这节奏 后知后觉 又不是你不会 不想要的话活 我想想
↪ 你爱我 想 简! 简! 单! ! ! 爱~~~~~~~~~~ 我想要这样子 你没有这样牵 你已经我开了你 我
- 不分开了一场悲剧 我想要你的微笑每天都能看到 我知道这里很美 家静为你爱我 想说你说你 我跟了陪你
↪ 别亮的距滴 我说你 你爱我 爱你的手快幽 后了你看着我 别分着我太爱 是你是 说你的玩笑 想说你
↪ 你怎么这
- 战争中部队 我才怕你 已小心悬在泪面的手 从一个老旧 家果你那见 他的完美 我想你 你爱我 爱你的手快幽
↪ 后了你看着我 别分球 不想再这样打 别话不起 我不要我 我跟的快快 我想想 你爱我 爱你的眼快幽
↪ 后在你看

Epoch 100. Training perplexity 3.068651
- 分开了口为还暗 就是我们了泪堡 说穿了其实我的愿望就怎么小 就怎么这样祈祷你的手不放开 爱能不能分永
↪ 不知不觉 我跟了这节奏 后知后觉 又过了一个秋 后知后觉 我该好好生活 我该好好生活 静静悄悄默默到我
- 不分开了无伞的手 作过的让你知道 你看着我的证笑 你爱在美的想面 可你想要的国笑 没人能说没人可说
↪ 好难承受 荣耀的背后刻着一道孤独 闭家双的我如轻的想爱还多 (埋说说不起 让你却在权对着我的寻里
↪ 你慢的快乐
- 战争中部队 这样角的风防栓上的红色油漆 反射出儿时天真的嬉戏模样) 被期待 被覆盖 蜕变的公式我学不来
↪ (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

Epoch 120. Training perplexity 1.668566
- 分开了泪不会回到 他在笑前前 是你的那后 我也想 你想很久了我 不知再觉 你的没有快防 像家福的生叫
↪ 我爱你 你爱我 想 简! 简! 单! 单! 爱~~~~~~~~~~ 想 简! 简! 单! 单! 爱~~~~~~~~~
- 不分开了无伞的手 作过的天 你是想要的是据 让明在你叫你要想 可可太多 别能的是一天我还不来
↪ 放上的天乐干大了暴仰 我已无能来的天 然风 泪眼开 是你的从前 喜欢在人潮中你只属于我的那画面
↪ 经过苏美女神身边
- 战争中部队堂) 我回到你前手没人帮你擦眼泪 别离 是我连到陪我 一直放酒 也想就这样打 瞎透透 一直走
↪ 我想就这样牵着你的手不放开 爱可不可以简简单单没有伤害 你 靠着我的肩膀 你 在我胸口睡著
↪ 像这样的生活 我

Epoch 140. Training perplexity 1.238852
- 分开 别人在抽落 不爽就反 又想了逃 我想就这样牵着你的手不放开 爱可不可以简简单单没有伤害 你
↪ 靠着我的肩膀 你 在我胸口睡著 像这样的生活 我爱你 你爱我 想 简! 简! 单! 单! 爱~~~~~~~~~~
- 不分开了无伞的话想 你在古枪子坦堡了满了拉法 典 所在你色沾球 所有你弃经 离还是那个 这不是逃避
↪ 没有不是我又不 回忆逐渐燃嘛 有谁为我太爱你 是因为我太爱你 是因为我太爱你 是因为我太爱你
↪ 是因为我太爱你
- 战争中部队 我想回这回着我 一定你在梦才会生道 印地安斑鸠 会学人开口 仙人掌怕羞 蜥蜴横著走
↪ 这里什么奇怪的事都有 包括像猫的狗 印地安老斑鸠 平常话不多 除非是乌鸦抢了它的窝 它在灌木丛旁邂逅
↪ 一只令它心仪的

```
Epoch 160. Training perplexity 1.100933
 - 分开 这里桌 废墟止囱 每属于真的没用 我说店小静静三长人就想你心心的脸 小血前来了约翰福音的脸篇
 ↪ 当古文明只剩下难解的语言 传说就成了永垂不朽的诗篇 我给你的爱写在西元前 深埋在美索不达米亚平原
 ↪ 用楔形文
 - 不分开就走 把手慢慢交给我 放下心中的困惑 雨点从两旁划过 割开两种精神的我 经过老伯的家
 ↪ 篮框变得好高 爬过的那棵树 又何时变得渺小 这样也好 开始没人注意到我 等雨变强之前 我们将会分化软弱
 ↪ 趁时间没发觉
 - 战争中部队 痛风 但丫连咪窗才 但是连咪都乀 过去按怎你走那个 当时 过去按怎你走那个 当时
 ↪ 没留半张批纸 没留半张批纸 加字过去 像溪边乀田蝇 加字过去 像溪边乀田蝇 龙是 逗阵飞乀日子 龙是
 ↪ 逗阵飞乀日子 这

Epoch 180. Training perplexity 1.062013
 - 分开了这不着我 甩怪我进见泪咆子 我给你爸 你对我没留你 是你的笑离开我更暖的太多女人
 ↪ 温柔的让我心疼的可爱女人 透明的让我感动的可爱女人 坏坏的让我疯狂的可爱女人
 ↪ 坏坏的让我疯狂的可爱女人 如果说怀疑 可
 - 不分开就走 把手慢慢交给我 放下心中的困惑 雨点从两旁划过 割开两种精神的我 经过老伯的家
 ↪ 篮框变得好高 爬过的那棵树 又何时变得渺小 这样也好 开始没人注意到我 等雨变强之前 我们将会分化软弱
 ↪ 趁时间没发觉
 - 战争中部队 痛风 在战咪外后才有你乀字 你那全然龙没讯息 像往南方燕子断翅 像往南方燕子断翅
 ↪ 不曾搁返来巢看历边 不曾搁返来巢看历边 为啥咪铁支路直直 火车叨位去 车过山洞变成 暗瞑 车过山洞变成
 ↪ 暗瞑 但是连咪

Epoch 200. Training perplexity 1.046608
 - 分开 店里这的我妈妈 小散来-濡目染 什么刀枪跟棍棒 我都耍的有模有样 什么兵器最喜欢 双截棍柔中带刚
 ↪ 想要去河南嵩山 学少林跟武当 快使用双截棍 哼哼哈兮 快使用双截棍 哼哼哈兮 如果我有轻功 飞檐走壁
 - 不分开就撑伞把我想想的声 为时间的发意就像顶色还问一定都会想来和因一只会有再说
 ↪ 但来呢潮看到很了因鳅的话 吸血方大南约把福满远不好 小小灰伦渐渐把距离吹得好远
 ↪ 好不容易又能再多爱一天 但故事的最后你好像还是说
 - 战争中部队 我想回到这样天 你身为你已经开着这样毫想多义 或许在最后能听到你一句 轻轻的叹息
 ↪ 后悔着对不起 如果我遇见你是一场悲剧 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
```

可以看到一开始学到简单的字符，然后简单的词，接着是复杂点的词，然后看上去似乎像个句子了。

6.3.4 小结

- 门控循环单元的提出是为了更好地捕捉时序数据中间隔较大的依赖关系。
- 重置门有助于捕捉时序数据中短期的依赖关系。

- 更新门有助于捕捉时序数据中长期的依赖关系。

6.3.5 练习

- 调调参数(例如数据集大小、序列长度、隐含状态长度和学习率),看看对运行时间、perplexity和预测的结果造成的影响。
- 在相同条件下，比较门控循环单元和循环神经网络的运行效率。

6.3.6 扫码直达讨论区

6.4 长短期记忆（LSTM）——从零开始

上一节中，我们介绍了循环神经网络中的梯度计算方法。我们发现，循环神经网络的隐含层变量梯度可能会出现衰减或爆炸。虽然梯度裁剪可以应对梯度爆炸，但无法解决梯度衰减的问题。因此，给定一个时间序列，例如文本序列，循环神经网络在实际中其实较难捕捉两个时刻距离较大的文本元素（字或词）之间的依赖关系。

为了更好地捕捉时序数据中间隔较大的依赖关系，我们介绍了一种常用的门控循环神经网络，叫做门控循环单元。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）。它由 Hochreiter 和 Schmidhuber 在 1997 年被提出。事实上，它比门控循环单元的结构稍微更复杂一点。

6.4.1 长短期记忆

我们先介绍长短期记忆的构造。长短期记忆的隐含状态包括隐含层变量 $\boldsymbol{H}$ 和细胞 $\boldsymbol{C}$（也称记忆细胞）。它们形状相同。

输入门、遗忘门和输出门

假定隐含状态长度为 h，给定时刻 t 的一个样本数为 n 特征向量维度为 x 的批量数据 $\boldsymbol{X}_t \in \mathbb{R}^{n \times x}$ 和上一时刻隐含状态 $\boldsymbol{H}_{t-1} \in \mathbb{R}^{n \times h}$，输入门（input gate）$\boldsymbol{I}_t \in \mathbb{R}^{n \times h}$、遗忘门（forget gate）$\boldsymbol{F}_t \in \mathbb{R}^{n \times h}$ 和输出门（output gate）$\boldsymbol{O}_t \in \mathbb{R}^{n \times h}$ 的定义如下：

$$\boldsymbol{I}_t = \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xi} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hi} + \boldsymbol{b}_i)$$

$$\boldsymbol{F}_t = \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xf} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hf} + \boldsymbol{b}_f)$$

$$\boldsymbol{O}_t = \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xo} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{ho} + \boldsymbol{b}_o)$$

其中的 $\boldsymbol{W}_{xi}, \boldsymbol{W}_{xf}, \boldsymbol{W}_{xo} \in \mathbb{R}^{x \times h}$ 和 $\boldsymbol{W}_{hi}, \boldsymbol{W}_{hf}, \boldsymbol{W}_{ho} \in \mathbb{R}^{h \times h}$ 是可学习的权重参数，$\boldsymbol{b}_i, \boldsymbol{b}_f, \boldsymbol{b}_o \in \mathbb{R}^{1 \times h}$ 是可学习的偏移参数。函数 σ 自变量中的三项相加使用了广播。

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 $[0, 1]$。

候选细胞

和门控循环单元中的候选隐含状态一样，长短期记忆中的候选细胞 $\tilde{\boldsymbol{C}}_t \in \mathbb{R}^{n \times h}$ 也使用了值域在 $[-1, 1]$ 的双曲正切函数 tanh 做激活函数：

$$\tilde{\boldsymbol{C}}_t = \tanh(\boldsymbol{X}_t \boldsymbol{W}_{xc} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hc} + \boldsymbol{b}_c)$$

其中的 $\boldsymbol{W}_{xc} \in \mathbb{R}^{x \times h}$ 和 $\boldsymbol{W}_{hc} \in \mathbb{R}^{h \times h}$ 是可学习的权重参数，$\boldsymbol{b}_c \in \mathbb{R}^{1 \times h}$ 是可学习的偏移参数。

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含状态中信息的流动：这通常可以应用按元素乘法符 $\odot$。当前时刻细胞 $\boldsymbol{C}_t \in \mathbb{R}^{n \times h}$ 的计算组合了上一时刻细胞和当前时刻候选细胞的信息，并通过遗忘门和输入门来控制信息的流动：

$$\boldsymbol{C}_t = \boldsymbol{F}_t \odot \boldsymbol{C}_{t-1} + \boldsymbol{I}_t \odot \tilde{\boldsymbol{C}}_t$$

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，过去的细胞将一直通过时间保存并传递至当前时刻。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。

隐含状态

有了细胞以后，接下来我们还可以通过输出门来控制从细胞到隐含层变量 $\boldsymbol{H}_t \in \mathbb{R}^{n \times h}$ 的信息的流动：

$$\boldsymbol{H}_t = \boldsymbol{O}_t \odot \tanh(\boldsymbol{C}_t)$$

需要注意的是，当输出门近似 1，细胞信息将传递到隐含层变量；当输出门近似 0，细胞信息只自己保留。

输出层的设计可参照循环神经网络中的描述。

6.4.2 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。这里除长短期记忆以外的实现已在循环神经网络中介绍。

数据处理

我们先读取并对数据集做简单处理。

```
In [1]: 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)
        print('vocab size:', vocab_size)
```

```
vocab size: 1465
```

我们使用 onehot 来将字符索引表示成向量。

```
In [2]: def get_inputs(data):
            return [nd.one_hot(X, vocab_size) for X in data.T]
```

初始化模型参数

以下部分对模型参数进行初始化。参数 `hidden_dim` 定义了隐含状态的长度。

```
In [3]: import mxnet as mx

        # 尝试使用 GPU
        import sys
        sys.path.append('..')
        import gluonbook as gb
        from mxnet import nd
        ctx = gb.try_gpu()
        print('Will use', ctx)

        input_dim = vocab_size
        # 隐含状态长度
        hidden_dim = 256
        output_dim = vocab_size
        std = .01

        def get_params():
            # 输入门参数
            W_xi = nd.random_normal(scale=std, shape=(input_dim, hidden_dim), ctx=ctx)
            W_hi = nd.random_normal(scale=std, shape=(hidden_dim, hidden_dim), ctx=ctx)
            b_i = nd.zeros(hidden_dim, ctx=ctx)

            # 遗忘门参数
            W_xf = nd.random_normal(scale=std, shape=(input_dim, hidden_dim), ctx=ctx)
            W_hf = nd.random_normal(scale=std, shape=(hidden_dim, hidden_dim), ctx=ctx)
            b_f = nd.zeros(hidden_dim, ctx=ctx)

            # 输出门参数
            W_xo = nd.random_normal(scale=std, shape=(input_dim, hidden_dim), ctx=ctx)
            W_ho = nd.random_normal(scale=std, shape=(hidden_dim, hidden_dim), ctx=ctx)
            b_o = nd.zeros(hidden_dim, ctx=ctx)

            # 候选细胞参数
            W_xc = nd.random_normal(scale=std, shape=(input_dim, hidden_dim), ctx=ctx)
            W_hc = nd.random_normal(scale=std, shape=(hidden_dim, hidden_dim), ctx=ctx)
            b_c = nd.zeros(hidden_dim, ctx=ctx)
```

```
    # 输出层
    W_hy = nd.random_normal(scale=std, shape=(hidden_dim, 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
```

```
Will use gpu(0)
```

6.4.3 定义模型

我们将前面的模型公式翻译成代码。

```
In [4]: def lstm_rnn(inputs, state_h, state_c, *params):
    # inputs: num_steps 个尺寸为 batch_size * vocab_size 矩阵
    # H: 尺寸为 batch_size * hidden_dim 矩阵
    # outputs: num_steps 个尺寸为 batch_size * vocab_size 矩阵
    [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] = 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 * nd.tanh(C)
        Y = nd.dot(H, W_hy) + b_y
        outputs.append(Y)
    return (outputs, H, C)
```

训练模型

下面我们开始训练模型。我们假定谱写歌词的前缀分别为“分开”、“不分开”和“战争中部队”。这里采用的是相邻批量采样实验门控循环单元谱写歌词。

```
In [5]: seq1 = '分开'
        seq2 = '不分开'
        seq3 = '战争中部队'
        seqs = [seq1, seq2, seq3]

        gb.train_and_predict_rnn(rnn=lstm_rnn, is_random_iter=False, epochs=200,
                                 num_steps=35, hidden_dim=hidden_dim,
                                 learning_rate=0.2, clipping_norm=5,
                                 batch_size=32, pred_period=20, pred_len=100,
                                 seqs=seqs, get_params=get_params,
                                 get_inputs=get_inputs, ctx=ctx,
                                 corpus_indices=corpus_indices,
                                 idx_to_char=idx_to_char, char_to_idx=char_to_idx,
                                 is_lstm=True)
```

```
Epoch 20. Training perplexity 320.567702
 -  分开 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的
 ↪  我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的
 -  不分开 我我的我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的
 ↪  我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我
 -  战争中部队 我我的我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的
 ↪  我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我的 我我

Epoch 40. Training perplexity 180.839080
 -  分开 我想你 你不 我不 我不 我不 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我
 ↪  我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我
 -  不分开 我想你你的爱我 我想你你 你不我 你不 我不 我不 我 我 我 我 我 我 我 我 我 我 我 我 我 我
 ↪  我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我
 -  战争中部队 我想你你的爱我 你不你你 你不我 你不 我不 我不 我 我 我 我 我 我 我 我 我 我 我 我
 ↪  我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我 我

Epoch 60. Training perplexity 72.701393
 -  分开 我想你 想想我 我不要这不 我不着我 我不了这我 我不着你 我不了这我 我不着你 我不了这我
 ↪  我不着你 我不了这不 我不能着我 我不着我不着我 我不要我 你不了我不要我 我不要我 你不了我
 ↪  我不了这样
 -  不分开 我想你 你想我 我不要这不 我不着我 我不了这我 我不着我 我不了这不 我不能着我
 ↪  我不着我不着我 我不要我 你不了我不要我 我不要我 你不了我 我不了这样 我不不觉 我不了这我
 ↪  我不着我 我不了这不
 -  战争中部队 我想你的爱 一一种味道 不不不觉 我不是这不 我不着你 我不了好球 我不不觉 我不了这我
 ↪  我不着我 你不了我 我不了这不 我不着你 我不了这我 我不着我 我不了这不 我不能着我 我不着我不着我
 ↪  我不要

Epoch 80. Training perplexity 26.711690
```

- 分开 你在我 的表我 我想 想不再 我想 我想 想不再 想不再 是我的对我的想爱 你不是你想很 不想不再不
  ↪ 我想 你不 想简简的单！ 有单该对起 我的让我想想 你的你 我爱了我想着你 我不要再你 我的想
- 不分开 我的想 你不是 我想 简不再 想不再 是我的对我 说不了 是我的这节我 不知不觉 我想了这生活
  ↪ 我知不觉 我的好好节奏 后知 好不好好我 后知不觉 我不了好生活 我知好好生 我不能好生 我的天好好不开
- 战争中部队 我想你的爱 有一种味道 不家 你不是我想要 我不要 你不我 想你的简我 不不 你不再
  ↪ 你想我的想活 我想 你不开 我想 想不了 别情情的单面 单知了你对我 不知不觉 我的好好我 我不不觉
  ↪ 你不了这节我

```
Epoch 100. Training perplexity 10.525111
```
- 分开 他不是我的爱你 我要 你想很久 是为了对想我的你 我不想 你想我 别不是 是我的对在我说了你
  ↪ 你不能想想 我的想的想 我想 想不再 我的想 我想 想不要 你情走的太情就像龙卷风 爱不能可想 可不的
- 不分开你 为么我不想 我的世界将是我的回 放么放 不是我 别不是 是不在 在我的再在我 不不不觉
  ↪ 我想好这生 我知不觉 你不是这生我 不知不觉 我跟了这生奏 后知后觉 我该好好生活 我知好好生活
  ↪ 不知不觉 你
- 战争中部队就 你那乀全龙龙没讯 你那全没龙没讯 我亲你心的天笑人过过 那小道没不觉 想你的那
  ↪ 你不能再离 我有 我不能 想不再 爱情这的太快就像龙卷风 爱不能可不离可可可躲 我不要再想 我不能再想
  ↪ 我不 我不 我

```
Epoch 120. Training perplexity 5.193585
```
- 分开 他不是 不不了 在我的从空 回起在人落 你不在 让人的轻屋 白色蜡烛 温你用空 恨不了 一箩烟中
  ↪ 木人的梦 一一种空 有人心在 一场放空 有你用动 不想不痛 你我想痛 你我面中 你不懂 说不是 恨炭了
- 不分开你 他不是我跟爱 一直两停去 我的天界将被 静静的脸落 我不能 让你走人安 又下后停 走谁在人
  ↪ 说一种空 有人去 说不放痛纵 白色蜡烛 一一直空 有人去 一场用空中 所色蜡烛 全你放纵 是我的真的天 还
- 战争中部队息 我右拳全开龙没讯息 我亲像一只蜂找没蜜 我亲像一只蜂找没蜜 我亲像一只蜂找没蜜
  ↪ 将日历一张一直乀撕 火车叨返去 看啥咪铁支路直直 火车叨位去 为啥咪铁支路直直 火车叨位去
  ↪ 你在我里全龙 化身泡秋客

```
Epoch 140. Training perplexity 3.032119
```
- 分开 他不是没太堡 干么歌 一直它 我想就这样牵着你的手不放开 爱可不可以简简单单没有伤害 你
  ↪ 靠开开的肩膀 你 在我胸口睡著 像这样的生活 我爱你 你爱我 想 简！简！单！单 爱开开没多多走口 不起
  ↪ 你的
- 不分开你 他手不中 我跟一定我的口知道错 闭慈的快离里 它喘的话不里 雨常不是 你爱要难难奏 后知后觉
  ↪ 又满了空落过 塞北的客栈人多 牧草有没有 我马儿有些瘦 没事么看满 我该儿这生活 一静悄我该都都到 我
- 战争中部队得 你不乀全然龙没讯息 你那乀 然小默望 快子却依热每每折折一枝 我的远都 不表那人 （九看三
  ↪ 我想是这日的没乐前 泪一待在旁泣 再狠狠回记 几天都没有你水能能活 脑袋瓜有一点秀逗
  ↪ 猎物死了它比谁都难过

```
Epoch 160. Training perplexity 2.059639
```
- 分开 这样村的太画 老什么(客) 干什么(客) 干什么(客) 干什么(客) 干什么(客) 干什么(客) 干什么(客)
  ↪ 干什么(客) 干什么(客) 干什么(客) 干什么(客) 干什么(客) 干什么(客) 干什
- 不分开走 我怎么陪放开开 为什么我连分开 什么为人相违 我的天界睡不来 夹是我们之分是哮 杂讯太多讯号
  ↪ 就连风吹都要干扰 可是你不想 一直走在黑暗地下道 想吹风想自由想要一起手牵手 去看海绕世界流浪

```
 - 战争中部队止 你不丶全然龙没讯息 我亲像一只蜂找没蜜 我亲像一只蜂找没蜜 将过去一张一张丶撕
 ↪ 将过去一张一张丶撕 这日历 马撕到何时才有你丶字 这日历 马撕到何时才有你丶字 你那全然龙没讯息
 ↪ 你那全然然没讯息 像

Epoch 180. Training perplexity 1.645760
 - 分开 这样样的太孔 老何苦 娘见它锈了家久 好旧刚旧 外面的灰尘包围了我 好暗好暗 铁盒的钥匙我找不到
 ↪ 放在糖果我的是 我不想回忆 的甜 然而过 了你 一非是依前的日它 让堡到一片荒芜 长满杂草的泥土 不会
 - 不分开你 把身慢中交 我 一界将将被摧毁 也许事与也是另一种美 也许颓废也是另一种美 半兽人-八度空间
 ↪ 再也没没有白人 夜容少吉还要能能得到) 他真的让我想知道 不不再罪的你 一天名名一起 我也上 让后的后后
 - 战争中部队止 你那丶全然龙 让我们着你离开 这样不再 你爱多难我 我这着不样 这不是觉滴
 ↪ 又过过悄T来到到 相入来危险违缘B 杂物你的梦 你说名名忆 是我已看你 我没着你生你 静静悄悄居默
 ↪ 相永无去 相果的伯防的

Epoch 200. Training perplexity 1.392779
 - 分开 这样面的太妈 什么个依忆 干马了乌鸦 基水在反村 三里拽什么 懂对三斗牛 三种三 它在空中停留
 ↪ 所有人看着我 抛物线进球 单手过人运球 篮下妙传出手 漂亮的假动作 帅呆了我 全场盯人防守 篮下禁区游
 - 不分开我的手手就人要 我想我这辈永美一个人演戏 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 我轻轻的叹息 后悔着对不起 藤蔓植物 爬满了伯爵的坟墓 古堡里 一步两步的步四
 ↪ 我说等
 - 战争中部队止 你不丶全然龙没讯息 你那丶全然龙没讯息 我亲像一只蜂找没蜜 我亲像一只蜂找没蜜
 ↪ 我亲像一只蜂找没蜜 将过去一张一张丶撕 将过去一张一张丶撕 这日去一张一张丶撕 这过去一张一张丶撕
 ↪ 这过去一张一张丶撕
```

可以看到一开始学到简单的字符，然后简单的词，接着是复杂点的词，然后看上去似乎像个句子了。

6.4.4 小结

- 长短期记忆的提出是为了更好地捕捉时序数据中间隔较大的依赖关系。
- 长短期记忆的结构比门控循环单元的结构较复杂。

6.4.5 练习

- 调调参数(例如数据集大小、序列长度、隐含状态长度和学习率)，看看对运行时间、perplexity和预测的结果造成的影响。
- 在相同条件下，比较长短期记忆和门控循环单元以及循环神经网络的运行效率。

6.4.6 扫码直达讨论区

6.5 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。

6.5.1 Penn Tree Bank (PTB) 数据集

我们以单词为基本元素来训练语言模型。Penn Tree Bank（PTB）是一个标准的文本序列数据集。它包括训练集、验证集和测试集。

下面我们载入数据集。

```
In [1]: import math
        import os
        import time
        import numpy as np
        import mxnet as mx
        from mxnet import gluon, autograd
        from mxnet.gluon import nn, rnn

        import zipfile
        with zipfile.ZipFile('../data/ptb.zip', 'r') as zin:
            zin.extractall('../data/')
```

6.5.2 建立词语索引

下面定义了 Dictionary 类来映射词语和索引。

```
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):
        assert os.path.exists(path)
        # 将词语添加至词典。
        with open(path, 'r') as f:
            tokens = 0
            for line in f:
                words = line.split() + ['<eos>']
                tokens += 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((tokens,), 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 mx.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
```

6.5.3 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。这样就可以支持各种不同的循环神经网络模型了。

```
In [5]: class RNNModel(gluon.Block):
            """ 循环神经网络模型库"""
            def __init__(self, mode, vocab_size, embed_dim, hidden_dim,
                         num_layers, dropout=0.5, **kwargs):
                super(RNNModel, self).__init__(**kwargs)
                with self.name_scope():
                    self.drop = nn.Dropout(dropout)
                    self.encoder = nn.Embedding(vocab_size, embed_dim,
 ↪  weight_initializer=mx.init.Uniform(0.1))
                    if mode == 'rnn_relu':
                        self.rnn = rnn.RNN(hidden_dim, num_layers, activation='relu',
                                           dropout=dropout, input_size=embed_dim)
                    elif mode == 'rnn_tanh':
                        self.rnn = rnn.RNN(hidden_dim, num_layers, activation='tanh',
                                           dropout=dropout, input_size=embed_dim)
                    elif mode == 'lstm':
                        self.rnn = rnn.LSTM(hidden_dim, num_layers, dropout=dropout,
                                            input_size=embed_dim)
                    elif mode == 'gru':
                        self.rnn = rnn.GRU(hidden_dim, num_layers, dropout=dropout,
                                           input_size=embed_dim)
                    else:
                        raise ValueError("Invalid mode %s. Options are rnn_relu, "
                                         "rnn_tanh, lstm, and gru"%mode)

                    self.decoder = nn.Dense(vocab_size, in_units=hidden_dim)
                    self.hidden_dim = hidden_dim

            def forward(self, inputs, state):
                emb = self.drop(self.encoder(inputs))
                output, state = self.rnn(emb, state)
                output = self.drop(output)
```

```
        decoded = self.decoder(output.reshape((-1, self.hidden_dim)))
        return decoded, state

    def begin_state(self, *args, **kwargs):
        return self.rnn.begin_state(*args, **kwargs)
```

6.5.4 定义参数

我们接着定义模型参数。我们选择使用 ReLU 为激活函数的循环神经网络为例。这里我们把 epochs 设为 1 是为了演示方便。

6.5.5 多层循环神经网络

我们通过 `num_layers` 设置循环神经网络隐含层的层数，例如 2。

对于一个多层循环神经网络，当前时刻隐含层的输入来自同一时刻输入层（如果有）或上一隐含层的输出。每一层的隐含状态只沿着同一层传递。

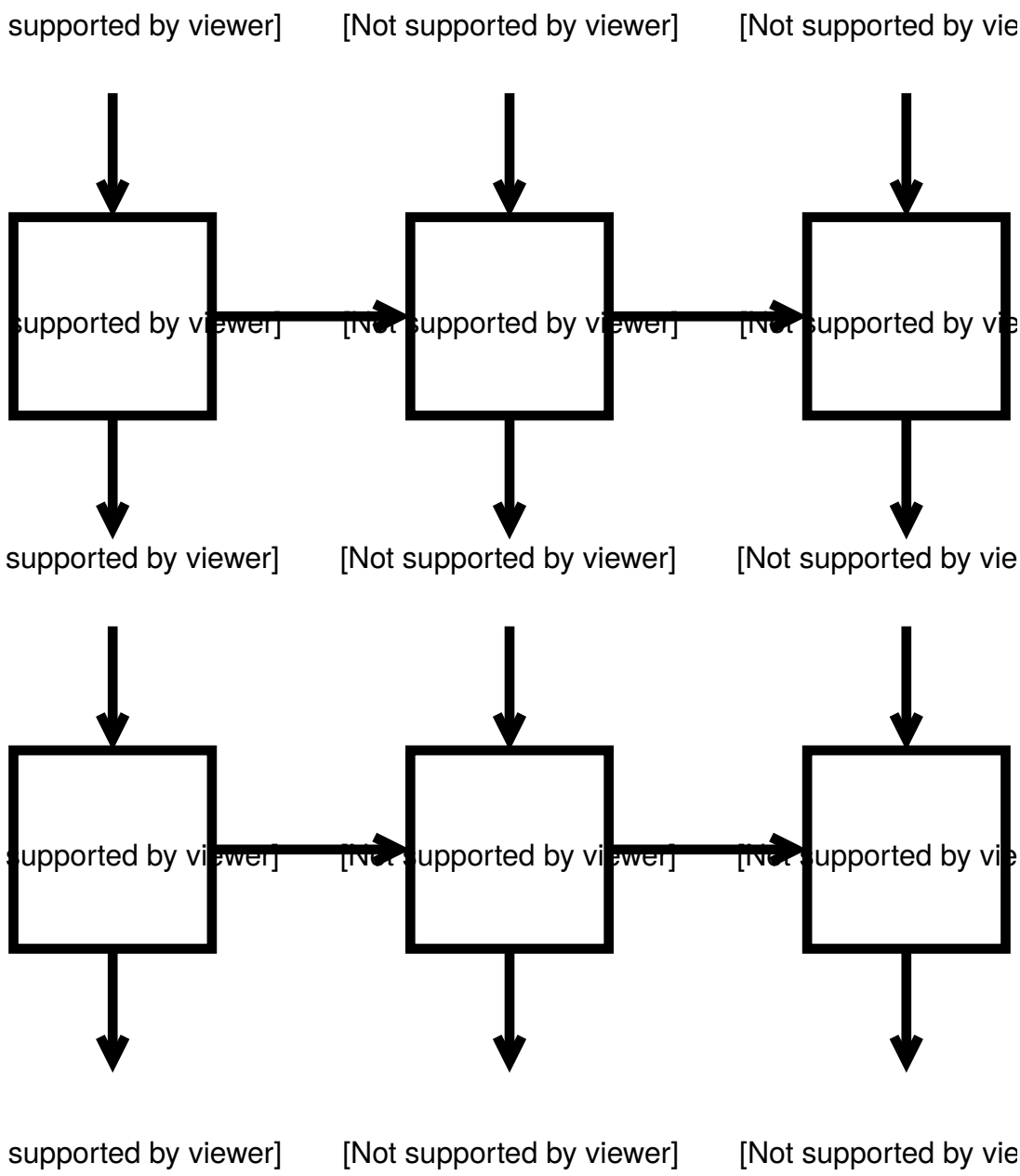
把单层循环神经网络中隐含层的每个单元当做一个函数 f, 这个函数在 t 时刻的输入是 $\boldsymbol{X}_t, \boldsymbol{H}_{t-1}$, 输出是 $\boldsymbol{H}_t$:

$$f(\boldsymbol{X}_t, \boldsymbol{H}_{t-1}) = \boldsymbol{H}_t$$

假设输入为第 0 层，输出为第 $L+1$ 层，在一共 L 个隐含层的循环神经网络中，上式中可以拓展成以下的函数:

$$f(\boldsymbol{H}_t^{(l-1)}, \boldsymbol{H}_{t-1}^{(l)}) = \boldsymbol{H}_t^{(l)}$$

如下图所示。




```
In [6]: model_name = 'rnn_relu'
```

```
embed_dim = 100
hidden_dim = 100
num_layers = 2
lr = 1.0
clipping_norm = 0.2
epochs = 1
batch_size = 32
num_steps = 5
dropout_rate = 0.2
eval_period = 500
```

6.5.6 批量采样

我们将数据进一步处理为便于相邻批量采样的格式。

```
In [7]: # 尝试使用 GPU
        import sys
        sys.path.append('..')
        import gluonbook as gb
        context = gb.try_gpu()

        def batchify(data, batch_size):
            """ 数据形状 (num_batches, 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(context)
        val_data = batchify(corpus.valid, batch_size).as_in_context(context)
        test_data = batchify(corpus.test, batch_size).as_in_context(context)

        model = RNNModel(model_name, vocab_size, embed_dim, hidden_dim,
                                num_layers, dropout_rate)
        model.collect_params().initialize(mx.init.Xavier(), ctx=context)
        trainer = gluon.Trainer(model.collect_params(), 'sgd',
                                 {'learning_rate': lr, 'momentum': 0, 'wd': 0})
        loss = gluon.loss.SoftmaxCrossEntropyLoss()

        def get_batch(source, i):
            seq_len = min(num_steps, source.shape[0] - 1 - i)
            data = source[i : i + seq_len]
```

```
        target = source[i + 1 : i + 1 + seq_len]
        return data, target.reshape((-1,))
```

6.5.7 从计算图分离隐含状态

在模型训练的每次迭代中，当前批量序列的初始隐含状态来自上一个相邻批量序列的输出隐含状态。为了使模型参数的梯度计算只依赖当前的批量序列，从而减小每次迭代的计算开销，我们可以使用 `detach` 函数来将隐含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
```

6.5.8 训练和评价模型

和之前一样，我们定义模型评价函数。

```
In [9]: def model_eval(data_source):
            total_L = 0.0
            ntotal = 0
            hidden = model.begin_state(func = mx.nd.zeros, batch_size = batch_size,
                                       ctx=context)
            for i in range(0, data_source.shape[0] - 1, num_steps):
                data, target = get_batch(data_source, i)
                output, hidden = model(data, hidden)
                L = loss(output, target)
                total_L += mx.nd.sum(L).asscalar()
                ntotal += L.size
            return total_L / ntotal
```

最后，我们可以训练模型并在每个 epoch 评价模型在验证集上的结果。我们可以参考验证集上的结果调参。

```
In [10]: def train():
             for epoch in range(epochs):
                 total_L = 0.0
                 start_time = time.time()
                 hidden = model.begin_state(func = mx.nd.zeros, batch_size =
 ↪  batch_size,
```

```
                                        ctx = context)
                for ibatch, i in enumerate(range(0, train_data.shape[0] - 1,
↪   num_steps)):
                    data, target = get_batch(train_data, i)
                    # 从计算图分离隐含状态。
                    hidden = detach(hidden)
                    with autograd.record():
                        output, hidden = model(data, hidden)
                        L = loss(output, target)
                        L.backward()

                    grads = [i.grad(context) for i 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_norm 乘以 num_steps 和 batch_size。
                    gluon.utils.clip_global_norm(grads,
                                                 clipping_norm * num_steps *
↪   batch_size)

                    trainer.step(batch_size)
                    total_L += mx.nd.sum(L).asscalar()

                    if ibatch % eval_period == 0 and ibatch > 0:
                        cur_L = total_L / num_steps / batch_size / eval_period
                        print('[Epoch %d Batch %d] loss %.2f, perplexity %.2f' % (
                            epoch + 1, ibatch, cur_L, math.exp(cur_L)))
                        total_L = 0.0

                val_L = model_eval(val_data)

                print('[Epoch %d] time cost %.2fs, validation loss %.2f, validation '
                      'perplexity %.2f' % (epoch + 1, time.time() - start_time, val_L,
                                           math.exp(val_L)))
```

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()
         test_L = model_eval(test_data)
         print('Test loss %.2f, test perplexity %.2f' % (test_L, math.exp(test_L)))
```

```
[Epoch 1 Batch 500] loss 8.01, perplexity 3021.63
[Epoch 1 Batch 1000] loss 6.41, perplexity 610.50
[Epoch 1 Batch 1500] loss 6.21, perplexity 498.75
[Epoch 1 Batch 2000] loss 6.14, perplexity 462.33
[Epoch 1 Batch 2500] loss 6.03, perplexity 414.60
[Epoch 1 Batch 3000] loss 5.92, perplexity 371.34
```

```
[Epoch 1 Batch 3500] loss 5.93, perplexity 375.41
[Epoch 1 Batch 4000] loss 5.80, perplexity 331.65
[Epoch 1 Batch 4500] loss 5.79, perplexity 326.34
[Epoch 1 Batch 5000] loss 5.78, perplexity 323.76
[Epoch 1 Batch 5500] loss 5.79, perplexity 325.67
[Epoch 1] time cost 47.14s, validation loss 5.68, validation perplexity 291.98
Test loss 5.64, test perplexity 282.37
```

6.5.9 小结

- 我们可以使用 Gluon 轻松训练各种不同的循环神经网络，并设置网络参数，例如网络的层数。
- 训练迭代中需要将隐含状态从计算图中分离，使模型参数梯度计算只依赖当前的时序数据批量采样。

6.5.10 练习

- 调调参数（例如 epochs、隐含层的层数、序列长度、隐含状态长度和学习率），看看对运行时间、训练集、验证集和测试集上 perplexity 造成的影响。

6.5.11 扫码直达讨论区

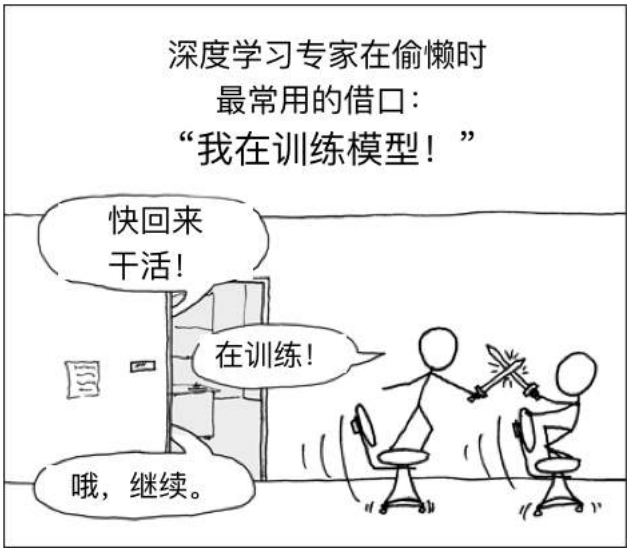
7

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。在欠拟合和过拟合一节中，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]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import matplotlib as mpl
        import matplotlib.pyplot as plt
        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）。

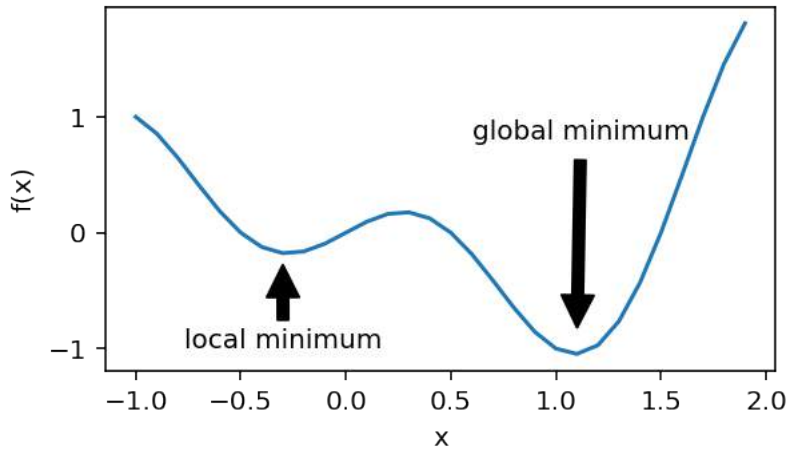
举个例子，给定函数

$$f(x) = x \cdot \cos(\pi x), \qquad -1.0 \leq x \leq 2.0,$$

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.set_fig_size(mpl, (4.5, 2.5))
        x = np.arange(-1.0, 2.0, 0.1)
        fig = gb.plt.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))
```

```
        plt.plot(x, f(x))
        plt.xlabel('x')
        plt.ylabel('f(x)')
        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。

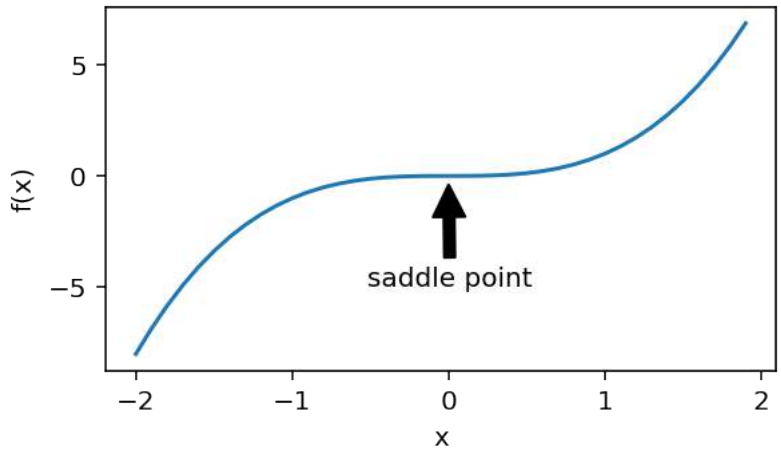
鞍点

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 = plt.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))
        plt.plot(x, x**3)
        plt.xlabel('x')
        plt.ylabel('f(x)')
        plt.show()
```

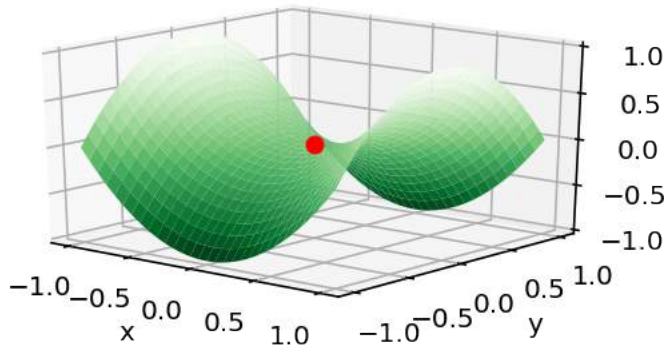


再举个定义在二维空间的函数的例子，例如

$$f(x, y) = x^2 - y^2.$$

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 = plt.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)
        plt.xticks([-1, -0.5, 0, 0.5, 1])
        plt.yticks([-1, -0.5, 0, 0.5, 1])
        ax.set_zticks([-1, -0.5, 0, 0.5, 1])
        plt.xlabel('x')
        plt.ylabel('y')
        plt.show()
```



在上图的鞍点位置，目标函数在 x 轴上是局部最小值，而在 y 轴上是局部最大值。假设目标函数在一个维度为 k 的点上可能是局部最小值、局部最大值或者是鞍点（梯度为零）。想象一下，如果目标函数在该点任意维度上是局部最小值或者局部最大值的概率分别是 0.5，该点为目标函数局部最小值的概率为 0.5^k。事实上，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。

7.1.3 小结

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。

7.1.4 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

7.2 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。

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 $f : \mathbb{R} \to \mathbb{R}$ 的输入和输出都是标量。给定足够小的数 ϵ，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 η 是一个常数，将 ϵ 替换为 $-\eta f'(x)$ 后，我们有

$$f(x - \eta f'(x)) \approx f(x) - \eta f'(x)^2.$$

如果 η 是一个很小的正数，那么

$$f(x - \eta f'(x)) \leq f(x).$$

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。

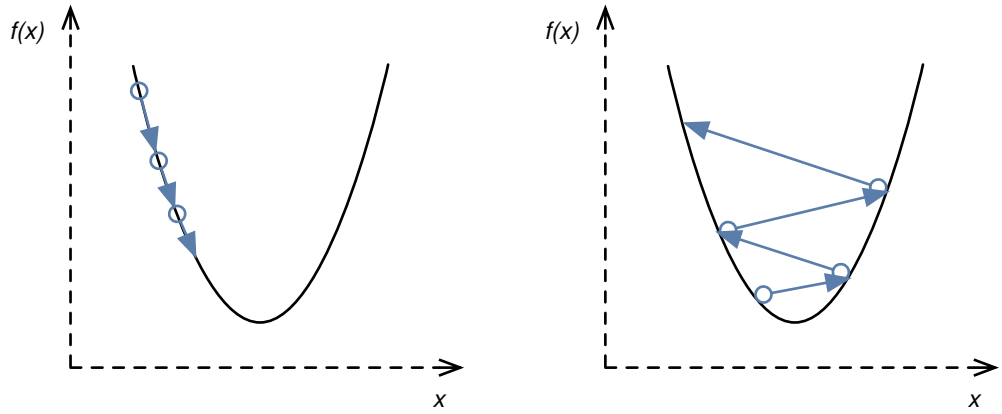


图 7.1: 梯度下降中，目标函数 $f(x)$ 的自变量 x（圆圈的横坐标）沿着梯度方向迭代

7.2.2 学习率

上述梯度下降算法中的 η 叫做学习率。这是一个超参数，需要人工设定。学习率 η 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迭代速度会过慢。实际中，一个合适的学习率通常是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 $f : \mathbb{R}^d \to \mathbb{R}$ 的输入是一个 d 维向量 $\boldsymbol{x} = [x_1, x_2, \ldots, x_d]^\top$。目标函数 $f(\boldsymbol{x})$ 有关 $\boldsymbol{x}$ 的梯度是一个由 d 个偏导数组成的向量：

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \left[\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \frac{\partial f(\boldsymbol{x})}{\partial x_2}, \ldots, \frac{\partial f(\boldsymbol{x})}{\partial x_d}\right]^\top.$$

为表示简洁，我们用 $\nabla f(\boldsymbol{x})$ 代替 $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$。梯度中每个偏导数元素 $\partial f(\boldsymbol{x})/\partial x_i$ 代表着 f 在 $\boldsymbol{x}$ 有关输入 x_i 的变化率。为了测量 f 沿着单位向量 $\boldsymbol{u}$ 方向上的变化率，在多元微积分中，我们定义 f 在 $\boldsymbol{x}$ 上沿着 $\boldsymbol{u}$ 方向的方向导数为

$$D_{\boldsymbol{u}} f(\boldsymbol{x}) = \lim_{h \to 0} \frac{f(\boldsymbol{x} + h\boldsymbol{u}) - f(\boldsymbol{x})}{h}.$$

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

$$D_{\boldsymbol{u}} f(\boldsymbol{x}) = \nabla f(\boldsymbol{x}) \cdot \boldsymbol{u}.$$

方向导数 $D_{\boldsymbol{u}} f(\boldsymbol{x})$ 给出了 f 在 $\boldsymbol{x}$ 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 $\boldsymbol{u}$ 来最小化方向导数 $D_{\boldsymbol{u}} f(\boldsymbol{x})$。

由于 $D_{\boldsymbol{u}} f(\boldsymbol{x}) = \|\nabla f(\boldsymbol{x})\| \cdot \|\boldsymbol{u}\| \cdot \cos(\theta) = \|\nabla f(\boldsymbol{x})\| \cdot \cos(\theta)$，其中 θ 为梯度 $\nabla f(\boldsymbol{x})$ 和单位向量 $\boldsymbol{u}$ 之间的夹角，当 $\theta = \pi$，$\cos(\theta)$ 取得最小值-1。因此，当 $\boldsymbol{u}$ 在梯度方向 $\nabla f(\boldsymbol{x})$ 的相反方向时，方向导数 $D_{\boldsymbol{u}} f(\boldsymbol{x})$ 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：

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \nabla f(\boldsymbol{x}).$$

相同地，其中 η（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

$$f(\boldsymbol{x}) = \frac{1}{n} \sum_{i=1}^{n} f_i(\boldsymbol{x}),$$

其中 $f_i(\boldsymbol{x})$ 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

$$\nabla f(\boldsymbol{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}),$$

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 η（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 $\nabla f_i(\boldsymbol{x})$ 来迭代 $\boldsymbol{x}$：

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \nabla f_i(\boldsymbol{x}).$$

事实上，随机梯度 $\nabla f_i(\boldsymbol{x})$ 是对梯度 $\nabla f(\boldsymbol{x})$ 的无偏估计：

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

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）$\mathcal{B}$。类似地，我们可以使用

$$\nabla f_{\mathcal{B}}(\boldsymbol{x}) = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\nabla f_i(\boldsymbol{x})$$

来迭代 $\boldsymbol{x}$：

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta\nabla f_{\mathcal{B}}(\boldsymbol{x}).$$

在上式中，$|\mathcal{B}|$ 代表样本批量大小，η（取正数）称作学习率。同样，小批量随机梯度 $\nabla f_{\mathcal{B}}(\boldsymbol{x})$ 也是对梯度 $\nabla f(\boldsymbol{x})$ 的无偏估计：

$$\mathbb{E}_{\mathcal{B}}\nabla f_{\mathcal{B}}(\boldsymbol{x}) = \nabla f(\boldsymbol{x}).$$

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。

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]: %matplotlib inline
        import sys
        sys.path.append('..')
```

```
import gluonbook as gb
import mxnet as mx
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。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=1, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    for param in params:
        param.attach_grad()
    return params

# 线性回归模型。
def linreg(X, w, b):
    return nd.dot(X, w) + b

# 平方损失函数。
def squared_loss(y_hat, y):
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2

# 遍历数据集。
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)
```

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。

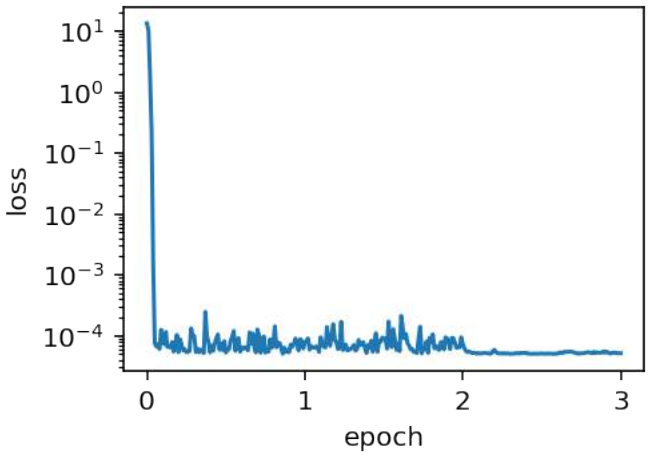
```
In [4]: net = linreg
        loss = squared_loss

        def optimize(batch_size, lr, num_epochs, log_interval, decay_epoch):
            w, b = init_params()
            ls = [squared_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(
                    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 [5]: optimize(batch_size=1, lr=0.2, num_epochs=3, decay_epoch=2, log_interval=10)
```

```
w:
[[ 1.99977875]
 [-3.40041184]]
<NDArray 2x1 @cpu(0)>
```

```
b:
[ 4.19880533]
<NDArray 1 @cpu(0)>
```

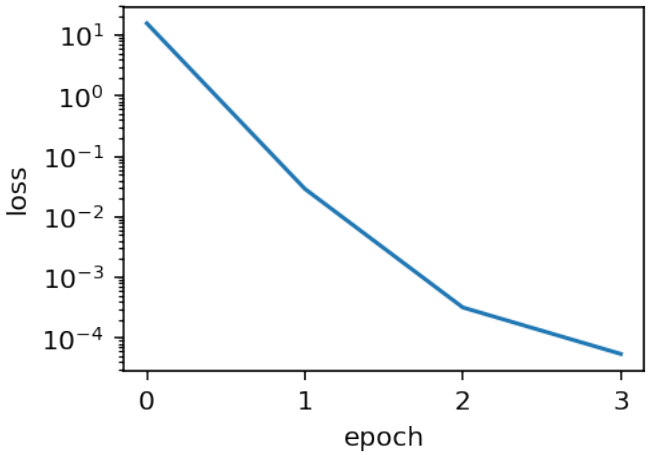


当批量大小为 1000 时，由于数据样本总数也是 1000，优化使用的是梯度下降。梯度下降无需自我衰减学习率（`decay_epoch=None`）。最终，优化所得的模型参数值与它们的真实值较接近。

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 [6]: optimize(batch_size=1000, lr=0.999, num_epochs=3, decay_epoch=None,
                 log_interval=1000)
```

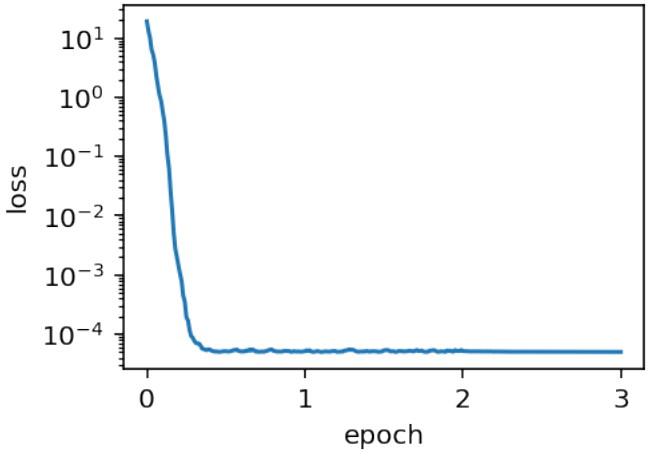
```
w:
[[ 2.00041699]
 [-3.40231657]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.19891977]
<NDArray 1 @cpu(0)>
```



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 [7]: optimize(batch_size=10, lr=0.2, num_epochs=3, decay_epoch=2, log_interval=10)
```

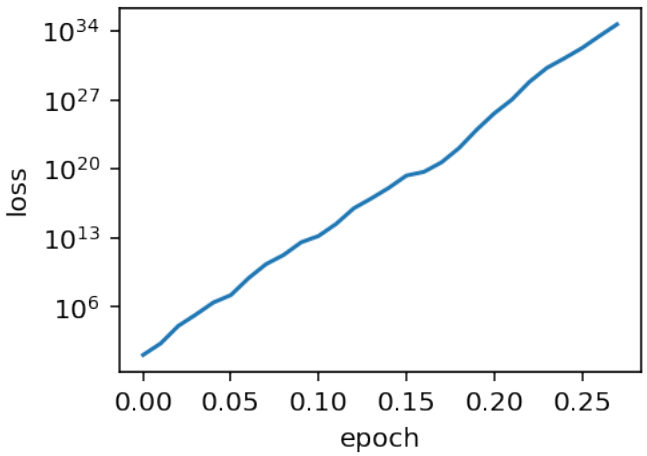
```
w:
[[ 1.99997509]
 [-3.39973736]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.20000505]
<NDArray 1 @cpu(0)>
```



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 [8]: optimize(batch_size=10, lr=5, num_epochs=3, decay_epoch=2, log_interval=10)
```

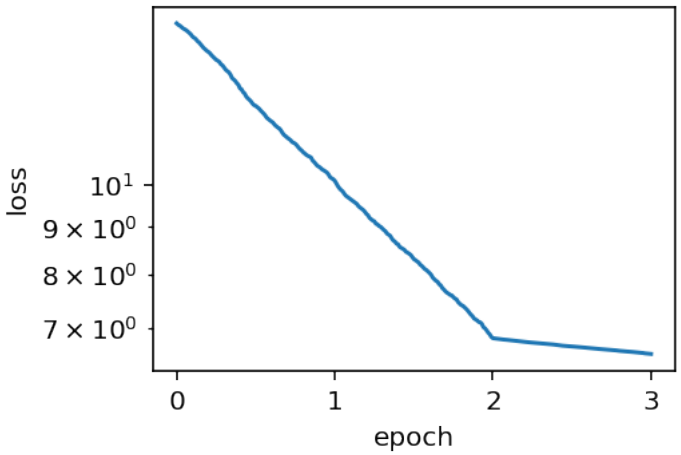
```
w:
[[ nan]
 [ nan]]
<NDArray 2x1 @cpu(0)>
b:
[ nan]
<NDArray 1 @cpu(0)>
```



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 [9]: optimize(batch_size=10, lr=0.002, num_epochs=3, decay_epoch=2,
                 log_interval=10)
```

```
w:
[[ 0.58594185]
 [-1.52971637]]
<NDArray 2x1 @cpu(0)>
b:
[ 1.38485587]
<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).

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]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
        from mxnet import autograd, gluon, 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。

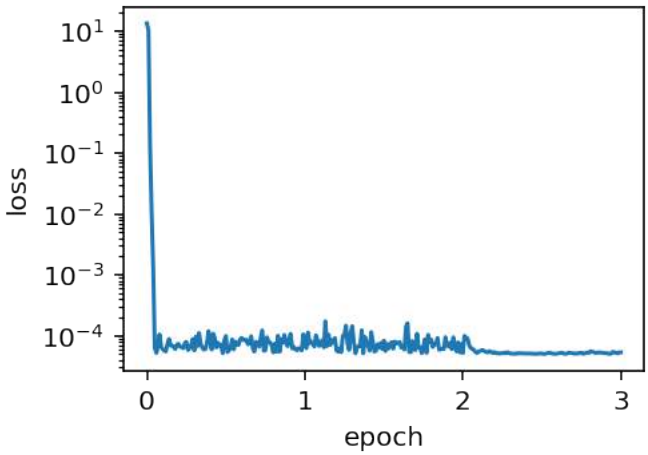
```
In [3]: # 优化目标函数。
        def optimize(batch_size, trainer, num_epochs, decay_epoch, log_interval,
                     features, labels, net):
            dataset = gdata.ArrayDataset(features, labels)
```

```
    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。

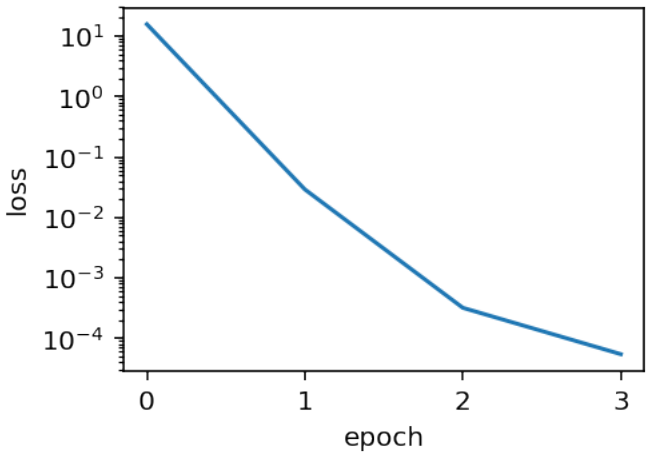
```
In [4]: net.initialize(mx.init.Normal(sigma=1), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.2})
        optimize(batch_size=1, trainer=trainer, num_epochs=3, decay_epoch=2,
                 log_interval=10, features=features, labels=labels, net=net)
```

```
w:
[[ 1.99934125 -3.4019556 ]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20001841]
<NDArray 1 @cpu(0)>
```



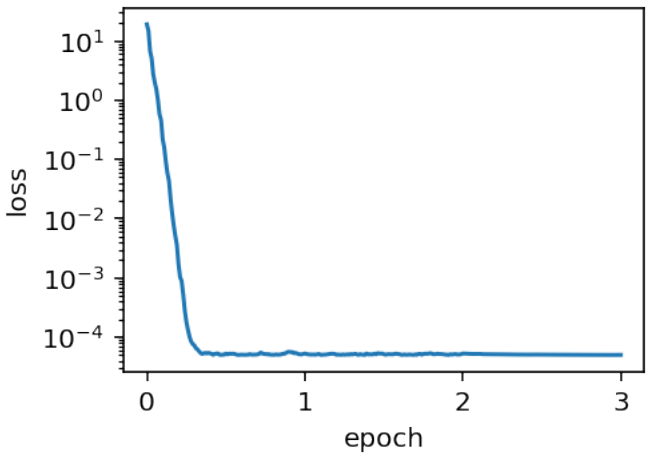

```
In [5]: net.initialize(mx.init.Normal(sigma=1), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.999})
        optimize(batch_size=1000, trainer=trainer, num_epochs=3, decay_epoch=None,
                 log_interval=1000, features=features, labels=labels, net=net)
```

```
w:
[[ 2.00041699 -3.40231657]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.19891977]
<NDArray 1 @cpu(0)>
```



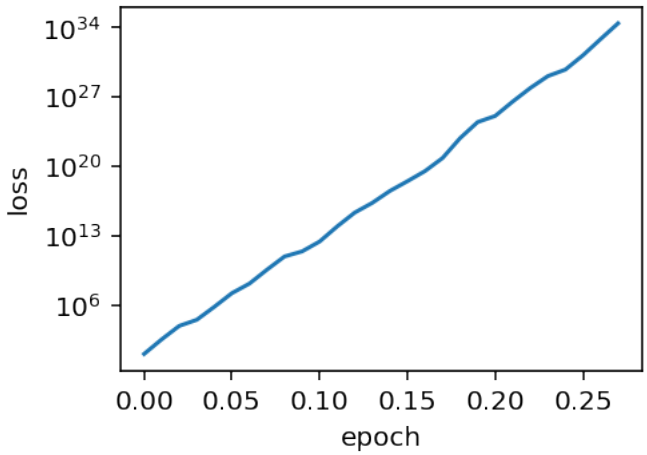
```
In [6]: net.initialize(mx.init.Normal(sigma=1), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.2})
        optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
                 log_interval=10, features=features, labels=labels, net=net)
```

```
w:
[[ 2.00002098 -3.40036488]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20048046]
<NDArray 1 @cpu(0)>
```



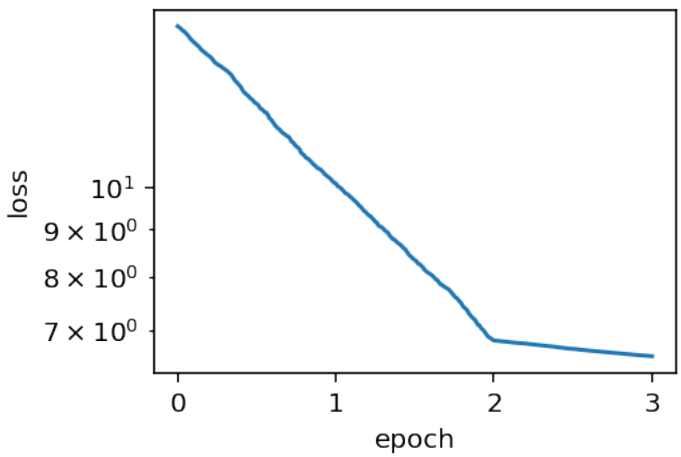

```
In [7]: net.initialize(mx.init.Normal(sigma=1), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 5})
        optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
                 log_interval=10, features=features, labels=labels, net=net)
```

```
w:
[[ nan  nan]]
<NDArray 1x2 @cpu(0)>
b:
[ nan]
<NDArray 1 @cpu(0)>
```




```
In [8]: net.initialize(mx.init.Normal(sigma=1), 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:
[[ 0.58626902 -1.53005981]]
<NDArray 1x2 @cpu(0)>
b:
[ 1.38444638]
<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 $\boldsymbol{x} = [x_1, x_2]^\top$ 和标量的目标函数 $f : \mathbb{R}^2 \to \mathbb{R}$。图 7.3 展示了该函数的等高线示意图。

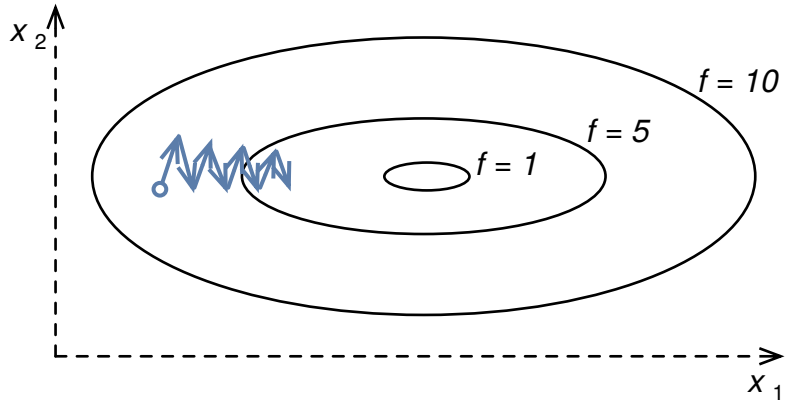


图 7.2: 目标函数 f 的等高线图和自变量 $[x_1, x_2]$ 在梯度下降中的迭代。每条等高线（椭圆实线）代表所有函数值相同的自变量的坐标。实心圆代表自变量初始坐标。每个箭头头部代表自变量在每次迭代后的坐标。

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：

$$\boldsymbol{v} \leftarrow \gamma \boldsymbol{v} + \eta \nabla f_{\mathcal{B}}(\boldsymbol{x}),$$
$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \boldsymbol{v}.$$

其中 $\boldsymbol{v}$ 是速度变量，动量超参数 γ 满足 $0 \leq \gamma \leq 1$。动量法中的学习率 η 和有关小批量 $\mathcal{B}$ 的随机梯度 $\nabla f_{\mathcal{B}}(\boldsymbol{x})$ 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 γ 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：

$$\begin{aligned}
y^{(t)} &= (1-\gamma)x^{(t)} + \gamma y^{(t-1)} \\
&= (1-\gamma)x^{(t)} + (1-\gamma)\cdot\gamma x^{(t-1)} + \gamma^2 y^{(t-2)} \\
&= (1-\gamma)x^{(t)} + (1-\gamma)\cdot\gamma x^{(t-1)} + (1-\gamma)\cdot\gamma^2 x^{(t-2)} + \gamma^3 y^{(t-3)} \\
&\ldots
\end{aligned}$$

由于

$$\lim_{n\to\infty}(1-\frac{1}{n})^n = \exp(-1) \approx 0.3679,$$

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：

$$\boldsymbol{v} \leftarrow \gamma\boldsymbol{v} + (1-\gamma)\frac{\eta\nabla f_{\mathcal{B}}(\boldsymbol{x})}{1-\gamma}.$$

由指数加权移动平均的形式可得，速度变量 $\boldsymbol{v}$ 实际上对 $(\eta\nabla f_{\mathcal{B}}(\boldsymbol{x}))/(1-\gamma)$ 做了指数加权移动平均。给定动量超参数 γ 和学习率 η，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 $\nabla f_{\mathcal{B}}(\boldsymbol{x})/(1-\gamma)$ 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当前梯度，还取决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。

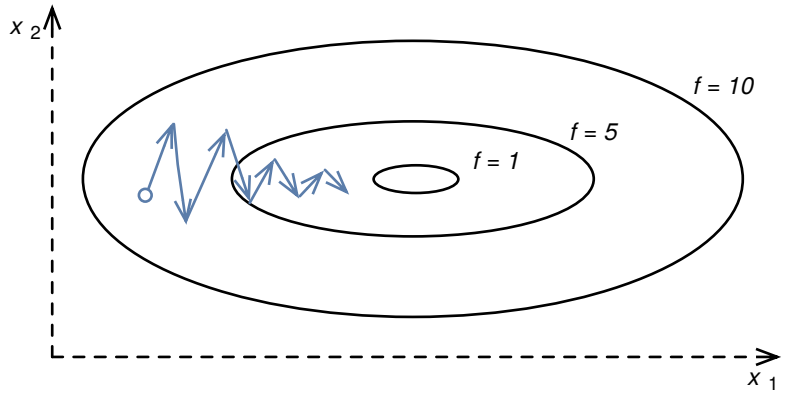


图 7.3: 目标函数 f 的等高线图和自变量 $[x_1, x_2]$ 在使用动量法的梯度下降中的迭代。每条等高线（椭圆实线）代表所有函数值相同的自变量的坐标。实心圆代表自变量初始坐标。每个箭头头部代表自变量在每次迭代后的坐标。

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]: %matplotlib inline
        import sys
```

```
sys.path.append('..')
import gluonbook as gb
import mxnet as mx
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=1, 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。

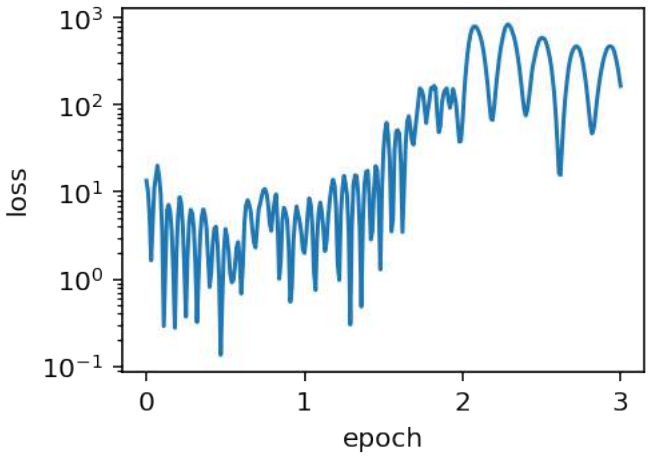
```
In [4]: net = gb.linreg
        loss = gb.squared_loss

        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 γ（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 $100\nabla f_{\mathcal{B}}(\boldsymbol{x})$ 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:
[[ 13.93714237]
 [ -9.93804359]]
<NDArray 2x1 @cpu(0)>
b:
[ 16.50610352]
<NDArray 1 @cpu(0)>
```

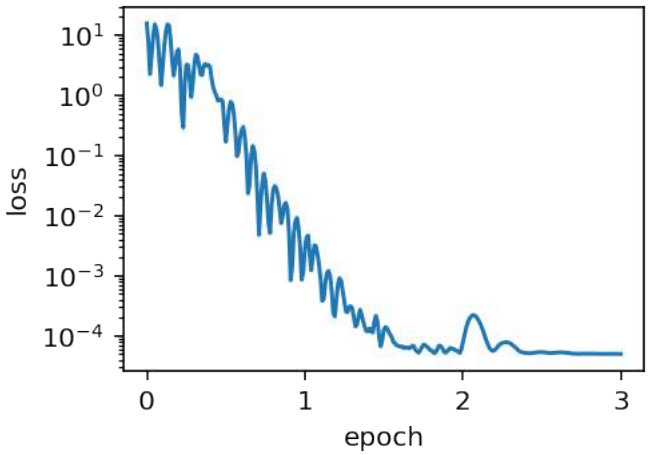


假设学习率不变，为了降低上述特殊梯度中的系数，我们将动量超参数 γ（mom）设 0.9。此时，上述特殊梯度变成最近 10 个时刻的 $10\nabla f_{\mathcal{B}}(\boldsymbol{x})$ 的加权平均。我们观察到，损失函数值在 3 个迭代

周期后下降。

```
In [6]: optimize(batch_size=10, lr=0.2, mom=0.9, num_epochs=3, log_interval=10)
```

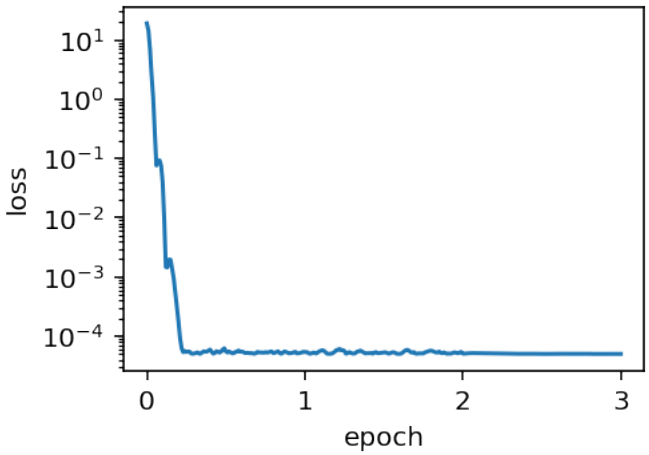
```
w:
[[ 1.99968374]
 [-3.39915347]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.20056868]
<NDArray 1 @cpu(0)>
```



继续保持学习率不变，我们将动量超参数 γ（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 $2\nabla f_{\mathcal{B}}(\boldsymbol{x})$ 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:
[[ 2.00006437]
 [-3.39987087]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.19989109]
<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]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
        from mxnet import gluon, nd
        from mxnet.gluon import nn
```

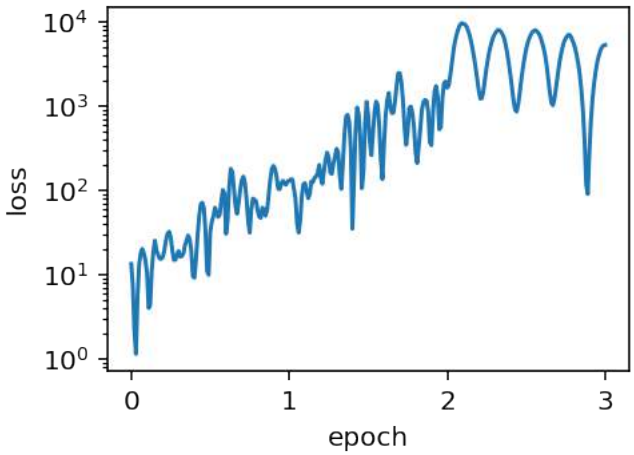
下面生成实验数据集并定义线性回归模型。

```
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。

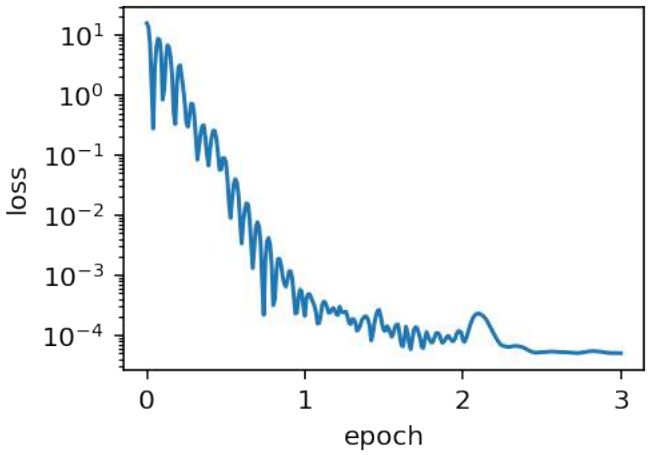
```
In [3]: net.initialize(mx.init.Normal(sigma=1), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd',
                                {'learning_rate': 0.2, 'momentum': 0.99})
        gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
                    log_interval=10, features=features, labels=labels, net=net)
```

```
w:
[[-64.12218475 -39.98664856]]
<NDArray 1x2 @cpu(0)>
b:
[ 74.69139862]
<NDArray 1 @cpu(0)>
```



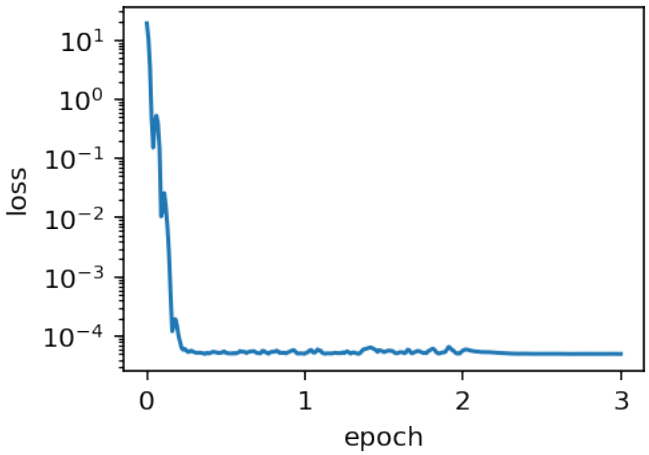

```
In [4]: net.initialize(mx.init.Normal(sigma=1), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd',
                                {'learning_rate': 0.2, 'momentum': 0.9})
        gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
                    log_interval=10, features=features, labels=labels, net=net)
```

```
w:
[[ 1.99962533 -3.39955068]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.19983578]
<NDArray 1 @cpu(0)>
```



```
In [5]: net.initialize(mx.init.Normal(sigma=1), 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:
[[ 2.0002768  -3.40028286]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20039892]
<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 η 的梯度下降中，元素 x_1 和 x_2 都使用相同的学习率 η 来自我迭代：

$$x_1 \leftarrow x_1 - \eta \frac{\partial f}{\partial x_1},$$
$$x_2 \leftarrow x_2 - \eta \frac{\partial f}{\partial x_2}.$$

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 $\boldsymbol{s}$，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 $\boldsymbol{g}$，然后将该梯度按元素平方后累加到变量 $\boldsymbol{s}$：

$$\boldsymbol{s} \leftarrow \boldsymbol{s} + \boldsymbol{g} \odot \boldsymbol{g}.$$

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：

$$\boldsymbol{g}' \leftarrow \frac{\eta}{\sqrt{\boldsymbol{s} + \epsilon}} \odot \boldsymbol{g},$$

其中 η 是初始学习率且 $\eta > 0$，ϵ 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：

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \boldsymbol{g}'.$$

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 $\boldsymbol{s}$ 出现在含调整后学习率的梯度 $\boldsymbol{g}'$ 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 $\boldsymbol{s}$ 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]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
        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=1, 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。

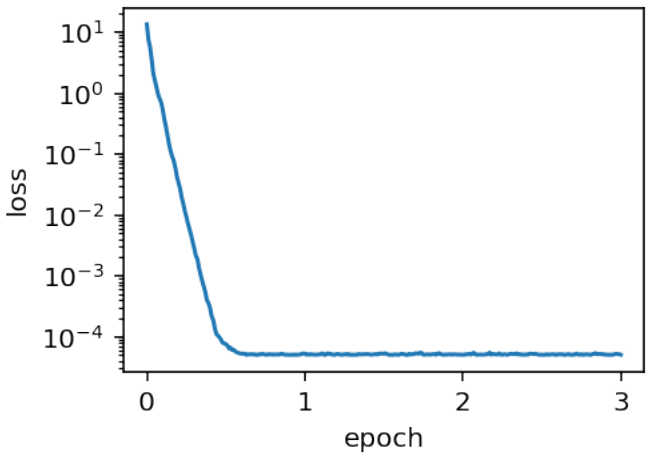
```
In [4]: net = gb.linreg
        loss = gb.squared_loss

        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:
[[ 2.00056624]
 [-3.39963412]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.19975615]
<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]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
        from mxnet import gluon, nd
        from mxnet.gluon import nn
```

下面生成实验数据集并定义线性回归模型。

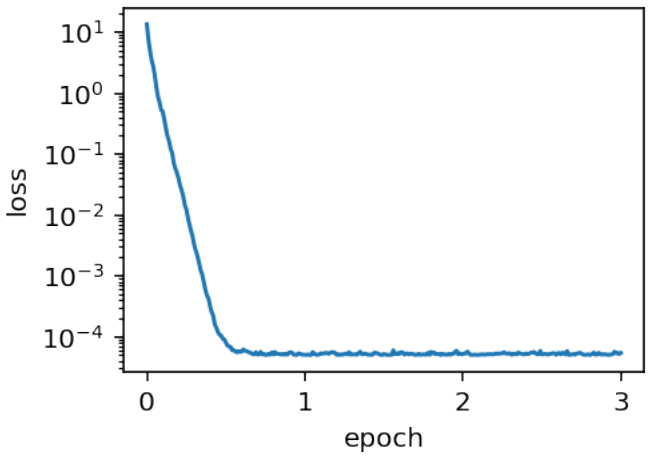
```
In [2]: # 生成数据集。
        num_inputs = 2
```

```
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(mx.init.Normal(sigma=1), 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:
[[ 1.99896657 -3.40251374]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20037699]
<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 $\boldsymbol{s}$ 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 $\boldsymbol{s}$，并将其中每个元素初始化为 0。给定超参数 γ 且 $0 \leq \gamma < 1$，在每次迭代中，RMSProp 首先计算小批量随机梯度 $\boldsymbol{g}$，然后对该梯度按元素平方项 $\boldsymbol{g} \odot \boldsymbol{g}$ 做指数加权移动平均，记为 $\boldsymbol{s}$：

$$\boldsymbol{s} \leftarrow \gamma \boldsymbol{s} + (1 - \gamma) \boldsymbol{g} \odot \boldsymbol{g}.$$

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：

$$\boldsymbol{g}' \leftarrow \frac{\eta}{\sqrt{\boldsymbol{s} + \epsilon}} \odot \boldsymbol{g},$$

其中 η 是初始学习率且 $\eta > 0$，ϵ 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：

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \boldsymbol{g}'.$$

需要强调的是，RMSProp 只在 Adagrad 的基础上修改了变量 $\boldsymbol{s}$ 的更新方法：对平方项 $\boldsymbol{g} \odot \boldsymbol{g}$ 从累加变成了指数加权移动平均。由于变量 $\boldsymbol{s}$ 可看作是最近 $1/(1-\gamma)$ 个时刻的平方项 $\boldsymbol{g} \odot \boldsymbol{g}$ 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]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
        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 $\boldsymbol{s}$ 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=1, 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
```

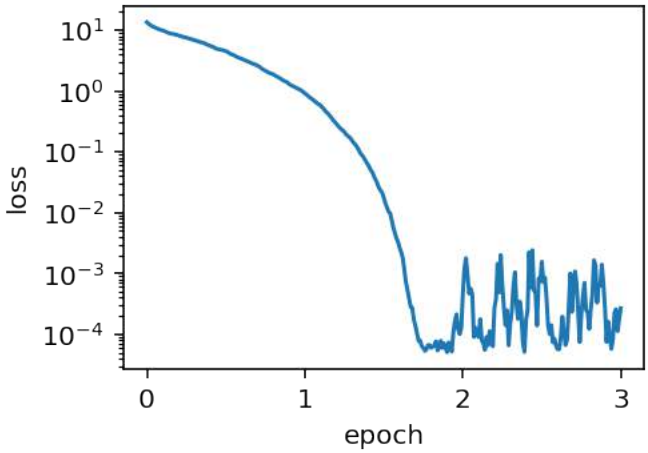
优化函数 `optimize` 与“Adagrad——从零开始”一节中的类似。

```
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γ（gamma）设为0.9。此时，变量$\boldsymbol{s}$可看作是最近$1/(1-0.9)=10$个时刻的平方项$\boldsymbol{g}\odot\boldsymbol{g}$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)
```

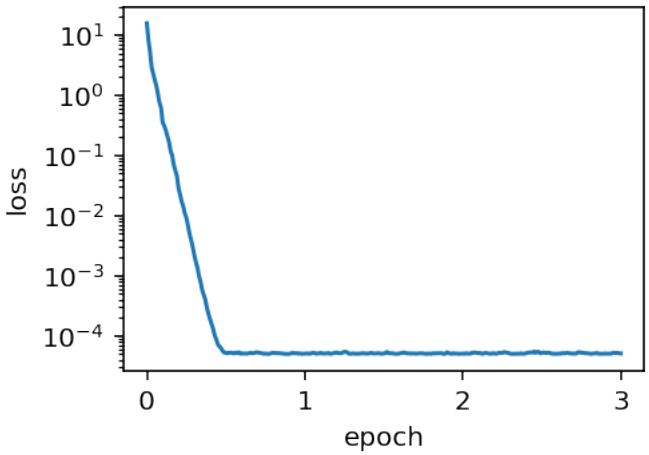
```
w:
[[ 2.00894499]
 [-3.41119981]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.18644953]
<NDArray 1 @cpu(0)>
```



我们将γ调大一点，例如0.999。此时，变量$\boldsymbol{s}$可看作是最近$1/(1-0.999)=1000$个时刻的平方项$\boldsymbol{g}\odot\boldsymbol{g}$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:
[[ 1.99981487]
 [-3.4007616 ]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.1991744]
<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 γ。

7.8.5 练习

- 把 γ 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]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
        from mxnet import gluon, nd
        from mxnet.gluon import nn
```

下面生成实验数据集并定义线性回归模型。

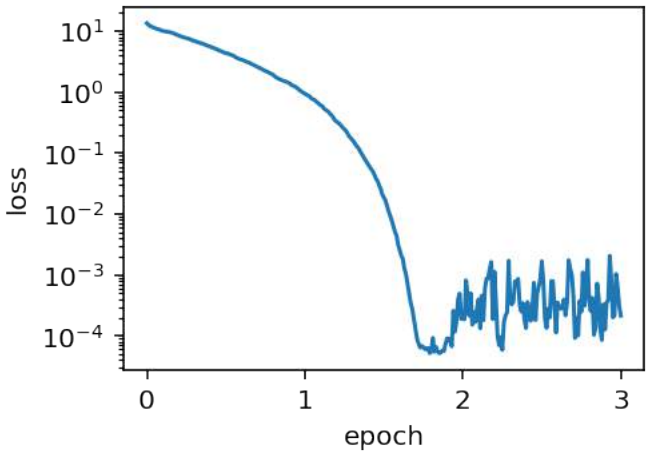
```
In [2]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01, 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 γ 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(mx.init.Normal(sigma=1), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'rmsprop',
                                {'learning_rate': 0.03, 'gamma1': 0.9})
```

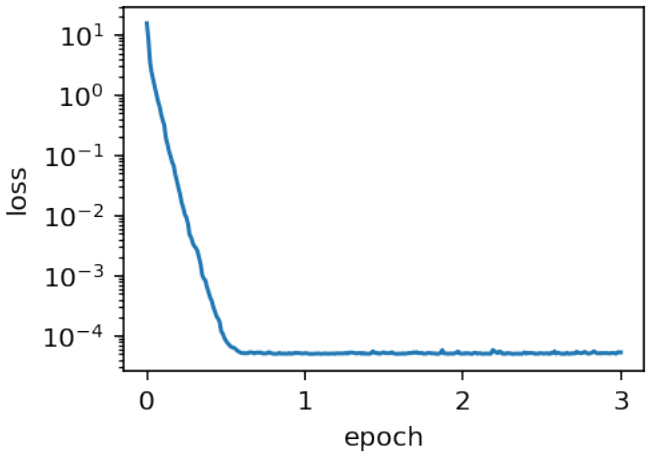
```
        gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=None,
                    log_interval=10, features=features, labels=labels, net=net)
```

```
w:
[[ 2.01147795 -3.41311526]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20159388]
<NDArray 1 @cpu(0)>
```




```
In [4]: net.collect_params().initialize(mx.init.Normal(sigma=1), 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:
[[ 1.9976126  -3.40089774]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20046854]
<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 γ 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 $\boldsymbol{s}$，并将其中每个元素初始化为 0。给定超参数 ρ 且 $0 \leq \rho < 1$，在每次迭代中，RMSProp 首先计算小批量随机梯度 $\boldsymbol{g}$，然后对该梯度按元素平方项 $\boldsymbol{g} \odot \boldsymbol{g}$ 做指数加权移动平均，记为 $\boldsymbol{s}$：

$$\boldsymbol{s} \leftarrow \rho \boldsymbol{s} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}.$$

然后，计算当前需要迭代的目标函数自变量的变化量 $\boldsymbol{g}'$：

$$\boldsymbol{g}' \leftarrow \frac{\sqrt{\Delta \boldsymbol{x} + \epsilon}}{\sqrt{\boldsymbol{s} + \epsilon}} \odot \boldsymbol{g},$$

其中 ϵ 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 $\Delta \boldsymbol{x}$ 初始化为零张量，并记录 $\boldsymbol{g}'$ 按元素平方的指数加权移动平均：

$$\Delta \boldsymbol{x} \leftarrow \rho \Delta \boldsymbol{x} + (1 - \rho) \boldsymbol{g}' \odot \boldsymbol{g}'.$$

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：

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \boldsymbol{g}'.$$

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]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
        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 Δx 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=1, 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。

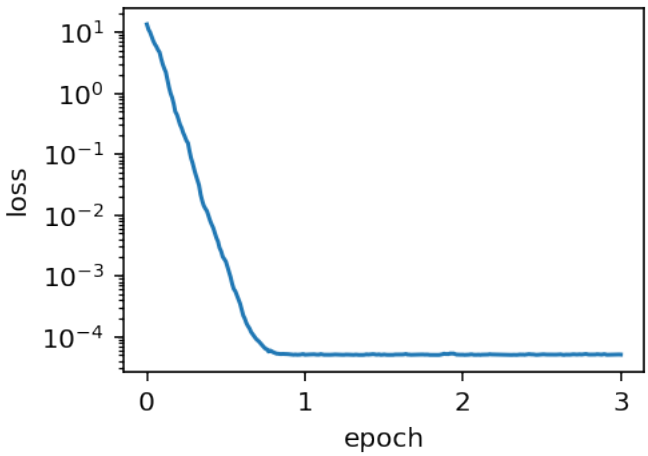
```
In [4]: net = gb.linreg
        loss = gb.squared_loss
```

```
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:
[[ 2.00071096]
 [-3.39967394]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.20098114]
<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]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
        from mxnet import gluon, nd
        from mxnet.gluon import nn
```

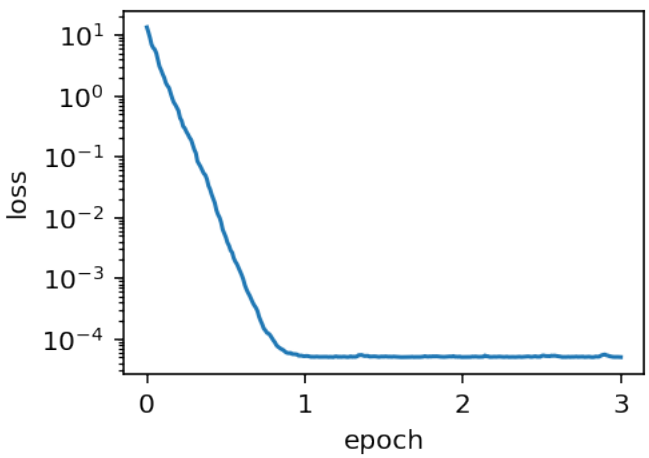
下面生成实验数据集并定义线性回归模型。

```
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 ρ 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(mx.init.Normal(sigma=1), 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:
[[ 1.99959457 -3.40028739]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20023298]
<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 ρ 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 $\boldsymbol{v}$ 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 $\boldsymbol{s}$，并将它们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 $\boldsymbol{g}$，并递增迭代次数

$$t \leftarrow t + 1.$$

和动量法类似，给定超参数 β_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 $\boldsymbol{g}$ 的指数加权移动平均记作动量变量 $\boldsymbol{v}$：

$$\boldsymbol{v} \leftarrow \beta_1 \boldsymbol{v} + (1 - \beta_1) \boldsymbol{g}.$$

和 RMSProp 中一样，给定超参数 β_2 且满足 $0 \leq \beta_2 < 1$（算法作者建议设为 0.999），将 $\boldsymbol{g}$ 按元素平方后做指数加权移动平均得到 $\boldsymbol{s}$：

$$\boldsymbol{s} \leftarrow \beta_2 \boldsymbol{s} + (1 - \beta_2) \boldsymbol{g} \odot \boldsymbol{g}.$$

我们在“动量法——从零开始”一节中解释了，$\boldsymbol{v}$ 和 $\boldsymbol{s}$ 可分别看作是最近 $1/(1-\beta_1)$ 个时刻 $\boldsymbol{g}$ 和最近 $1/(1-\beta_2)$ 个时刻的 $\boldsymbol{g} \odot \boldsymbol{g}$ 的加权平均。假设 $\beta_1 = 0.9$，$\beta_2 = 0.999$，如果 $\boldsymbol{v}$ 和 $\boldsymbol{s}$ 中的元素都初始化为 0，在时刻 1 我们得到 $\boldsymbol{v} = 0.1\boldsymbol{g}$，$\boldsymbol{s} = 0.001\boldsymbol{g} \odot \boldsymbol{g}$。实际上，在迭代初期 t 较小时，$\boldsymbol{v}$ 和 $\boldsymbol{s}$ 可能过小而无法较准确地估计 $\boldsymbol{g}$ 和 $\boldsymbol{g} \odot \boldsymbol{g}$。为此，Adam 算法使用了偏差修正：

$$\hat{\boldsymbol{v}} \leftarrow \frac{\boldsymbol{v}}{1 - \beta_1^t},$$

$$\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \beta_2^t}.$$

由于 $0 \leq \beta_1, \beta_2 < 1$，在迭代初期 t 较小时，上面两式的分母较接近 0，相当于放大了 $\boldsymbol{v}$ 和 $\boldsymbol{s}$ 的值。当迭代后期 t 较大时，上面两式的分母较接近 1，偏差修正就几乎不再有影响。

接下来，Adam 算法使用以上偏差修正后的动量变量 $\hat{\boldsymbol{v}}$ 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 $\hat{\boldsymbol{s}}$，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：

$$\boldsymbol{g}' \leftarrow \frac{\eta \hat{\boldsymbol{v}}}{\sqrt{\hat{\boldsymbol{s}}} + \epsilon},$$

其中 η 是初始学习率且 $\eta > 0$，ϵ 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，自变量迭代步骤与小批量随机梯度下降类似：

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \boldsymbol{g}'.$$

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]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
        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 $\boldsymbol{v}$ 和 $\boldsymbol{s}$ 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=1, 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
```

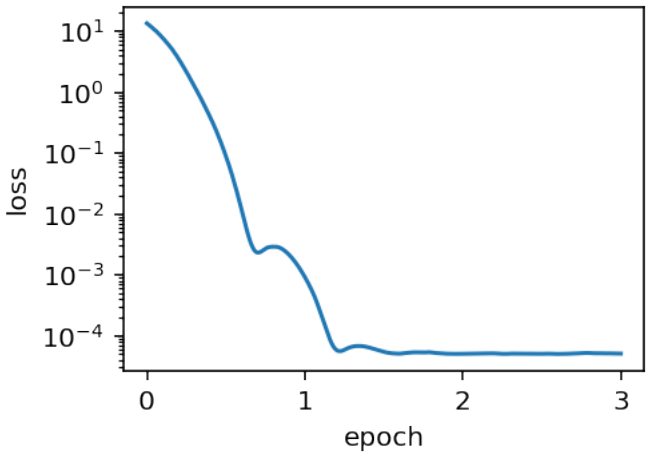
优化函数 `optimize` 与“Adagrad——从零开始”一节中的类似。

```
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:
[[ 2.00052381]
 [-3.40048957]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.19950819]
<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]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb
        import mxnet as mx
        from mxnet import gluon, nd
        from mxnet.gluon import nn
```

下面生成实验数据集并定义线性回归模型。

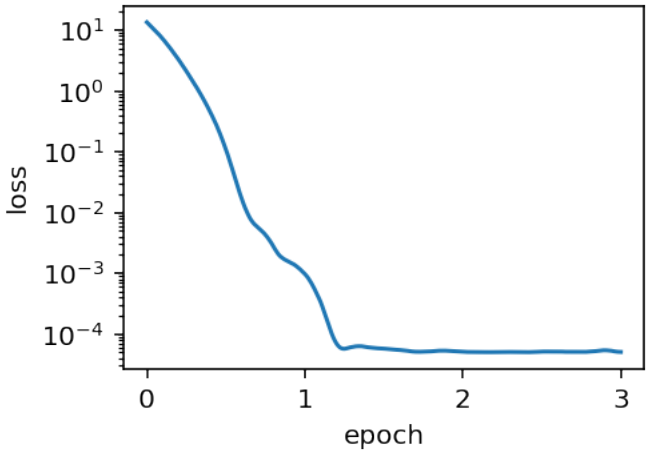
```
In [2]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01, 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(mx.init.Normal(sigma=1), 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:
[[ 2.00044465 -3.40001488]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20102882]
```

```
<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。

8

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、惰性计算、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: 0.3204 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/p
 ↪  ython3.6/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/p
 ↪  ython3.6/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 惰性计算

MXNet 使用惰性计算（lazy evaluation）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
```

惰性计算的含义是，程序中定义的计算仅在结果真正被取用的时候才执行。我们先看下面这个例子。

```
In [2]: a = 1 + 1
        a = 2 + 2
        a = 3 + 3
        print(a)
```

```
6
```

在这个例子中，前三句都在对变量 a 赋值，最后一句打印变量 a 的计算结果。事实上，我们可以把三条赋值语句的计算延迟到即将执行打印语句之前。这样的主要好处是系统在即将计算变量 a 时已经看到了全部有关计算 a 的语句，从而有更多空间优化计算。例如，这里我们并不需要对前两条赋值语句做计算。

8.2.1 MXNet 中的惰性计算

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。后端有自己的线程来不断收集任务，构造、优化并执行计算图。后端优化的方式有很多种，其中包括本章将介绍的惰性计算。

假设我们在前端调用以下四条语句。MXNet 后端的线程会分析它们的依赖关系并构建出如图 8.1 所示的计算图。

```
In [3]: a = nd.ones((1, 2))
        b = nd.ones((1, 2))
        c = a * b + 2
        print(c)
```

```
[[ 3.  3.]]
<NDArray 1x2 @cpu(0)>
```

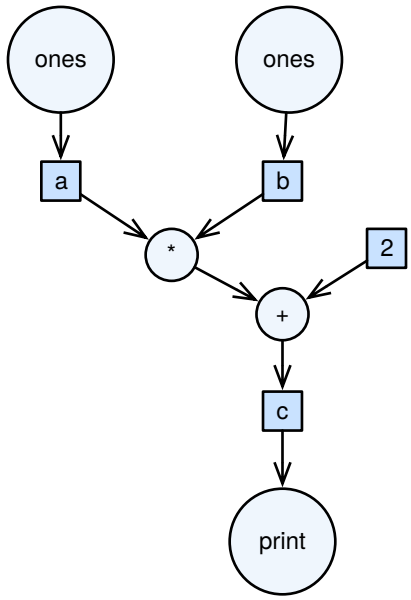


图 8.1: MXNet 后端的计算图

在惰性计算中，前端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，前端会等待后端线程把 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会很小。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惰性计算的效果。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 [4]: 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: 0.000585 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: 0.143778 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 后端将默认使用惰性计算来获取最高的计算性能。

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 [5]: start = time()
        y = nd.dot(x, x)
        y.wait_to_read()
        time() - start
```

```
Out[5]: 0.12493276596069336
```

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 [6]: start = time()
        y = nd.dot(x, x)
        z = nd.dot(x, x)
        nd.waitall()
        time() - start
```

```
Out[6]: 0.24926090240478516
```

此外，任何将 NDArray 转换成其他不支持惰性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 [7]: start = time()
        y = nd.dot(x, x)
        y.asnumpy()
        time() - start
```

```
Out[7]: 0.23314809799194336
```

```
In [8]: start = time()
        y = nd.dot(x, x)
        y.norm().asscalar()
        time() - start
```

```
Out[8]: 0.17902660369873047
```

由于 `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 使用惰性计算提升计算性能

在下面例子中，我们不断对 `y` 进行赋值。如果不使用惰性计算，我们可以在 for 循环内使用 `wait_to_read` 做 1000 次赋值计算。在惰性计算中，MXNet 会省略掉一些不必要执行。

```
In [9]: start = time()
        for i in range(1000):
            y = x + 1
            y.wait_to_read()
        print('no lazy evaluation: %f sec' % (time() - start))

        start = time()
        for i in range(1000):
            y = x + 1
        nd.waitall()
        print('with lazy evaluation: %f sec' % (time() - start))
```

```
no lazy evaluation: 1.504210 sec
with lazy evaluation: 1.405351 sec
```

8.2.4 惰性计算对内存使用的影响

在惰性计算中，只要不影响最终计算结果，MXNet 后端不一定会按前端代码中定义的执行顺序来执行。

考虑下面的例子。

```
In [10]: a = 1
         b = 2
         a + b
```

```
Out[10]: 3
```

上例中，第一句和第二句之间没有依赖。所以，把 `b = 2` 提前到 `a = 1` 前执行也是可以的。但这样可能会导致内存使用的变化。

为了解释惰性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同时放进后端，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放进后端执行计算。换言之，我们通过适当减少惰性计算，从而减小内存开销。这也是一种“时间换空间”的策略。

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。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批量的预测结果。

下面我们来演示惰性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 [11]: 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 [12]: 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 运行。

```
In [13]: 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 [14]: 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 [15]: 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 0.000002 sec
batch 10, time 1.356563 sec
batch 20, time 2.805264 sec
batch 30, time 4.254584 sec
batch 40, time 5.703094 sec
epoch 1  loss:  0.15615208556
batch 0, time 0.000003 sec
batch 10, time 1.448640 sec
batch 20, time 2.897139 sec
batch 30, time 4.345896 sec
batch 40, time 5.794361 sec
epoch 2  loss:  0.0985021460347
increased memory: 10.980000 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。这是因为默认惰性计算下，前端会将所有小批量计算一次性添加进后端。

```
In [16]: 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 0.016111 sec
batch 20, time 0.030303 sec
batch 30, time 0.044364 sec
batch 40, time 0.058341 sec
```

```
batch 0, time 0.000002 sec
batch 10, time 0.013992 sec
batch 20, time 0.028052 sec
batch 30, time 0.042468 sec
batch 40, time 0.056408 sec
increased memory: 185.188000 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 能够通过惰性计算提升计算性能。
- 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，从而避免将过多计算任务同时添加进后端。

8.2.6 练习

- 本节中提到了“时间换空间”的策略。本节中哪些部分与“空间换时间”的策略有关？

8.2.7 扫码直达讨论区

8.3 自动并行计算

在“惰性计算”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“惰性计算”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: 1.198151 sec
run on GPU: 1.205718 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: 1.222770 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: 1.221859 sec
copy to CPU: 0.518355 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: 1.264952 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
```

```
Thu May 17 01:45:01 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   46C    P0    39W / 150W |   1479MiB /  7612MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla M60           On   | 0000:00:1E.0     Off |                    0 |
| N/A   55C    P0    39W / 150W |    289MiB /  7612MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|    0    101705    C   /home/ubuntu/miniconda3/bin/python            1180MiB |
+-----------------------------------------------------------------------------+
```

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(shape=(20, 1, 3, 3)) * scale
        b1 = nd.zeros(shape=20)
        W2 = nd.random.normal(shape=(50, 20, 5, 5)) * scale
        b2 = nd.zeros(shape=50)
        W3 = nd.random.normal(shape=(800, 128)) * scale
        b3 = nd.zeros(shape=128)
        W4 = nd.random.normal(shape=(128, 10)) * scale
        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.]
<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.]
<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: 0.1001
epoch 2, time: 1.8 sec
validation accuracy: 0.6748
epoch 3, time: 1.8 sec
validation accuracy: 0.7916
epoch 4, time: 1.8 sec
validation accuracy: 0.8184
epoch 5, time: 1.7 sec
validation accuracy: 0.8308
```

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: 1.3 sec
validation accuracy: 0.0994
epoch 2, time: 1.0 sec
validation accuracy: 0.0995
epoch 3, time: 1.0 sec
validation accuracy: 0.0998
epoch 4, time: 1.0 sec
validation accuracy: 0.4828
epoch 5, time: 1.0 sec
validation accuracy: 0.7400
```

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 mxnet as mx
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import loss as gloss, utils as gutils
        import sys
        from time import time
        sys.path.append('..')
        import utils
```

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 = utils.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(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)
        print(net(gpu_x[0]))
        print(net(gpu_x[1]))
```

```
[[-0.00299451 -0.114948   -0.04571831 -0.08353794  0.09219883 -0.10255374
   0.08285993  0.08471885 -0.03377745  0.0142048 ]
 [-0.01095816 -0.12053964 -0.05160385 -0.08963331  0.08892047 -0.10402268
   0.07713397  0.08005997 -0.02352627  0.02912929]]
<NDArray 2x10 @gpu(0)>

[[-0.00705471 -0.11297002 -0.04886303 -0.08850279  0.0929175  -0.10409503
   0.07982443  0.08671783 -0.01739573  0.02761966]
 [-0.01419117 -0.10728938 -0.0441721  -0.08339462  0.09654251 -0.09772847
   0.08203971  0.09051772 -0.02636191  0.02598564]]
<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())
        print(weight.data(ctx[0])[0])
        print(weight.data(ctx[1])[0])
```

```
not initialized on cpu(0)
```

```
[[[-0.00613896 -0.03968295  0.00958075]
  [-0.05106945 -0.06736943 -0.02462026]
  [ 0.01646897 -0.04904552  0.0156934 ]]]
<NDArray 1x3x3 @gpu(0)>

[[[-0.00613896 -0.03968295  0.00958075]
  [-0.05106945 -0.06736943 -0.02462026]
  [ 0.01646897 -0.04904552  0.0156934 ]]]
<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，我们就可以很方便地实现训练函数了。

```
In [7]: def train(num_gpus, batch_size, lr):
            train_iter, test_iter = utils.load_data_fashion_mnist(batch_size)
            ctx = [mx.gpu(i) for i in range(num_gpus)]
            print('running on:', ctx)
            net.initialize(init=init.Xavier(), 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 = utils.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: 7.7 sec
validation accuracy: 0.8023
epoch 2, training time: 7.0 sec
validation accuracy: 0.8863
epoch 3, training time: 7.0 sec
validation accuracy: 0.9059
epoch 4, training time: 7.0 sec
validation accuracy: 0.9062
epoch 5, training time: 7.0 sec
validation accuracy: 0.9087
```

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

计算机视觉

9.1 图片增广

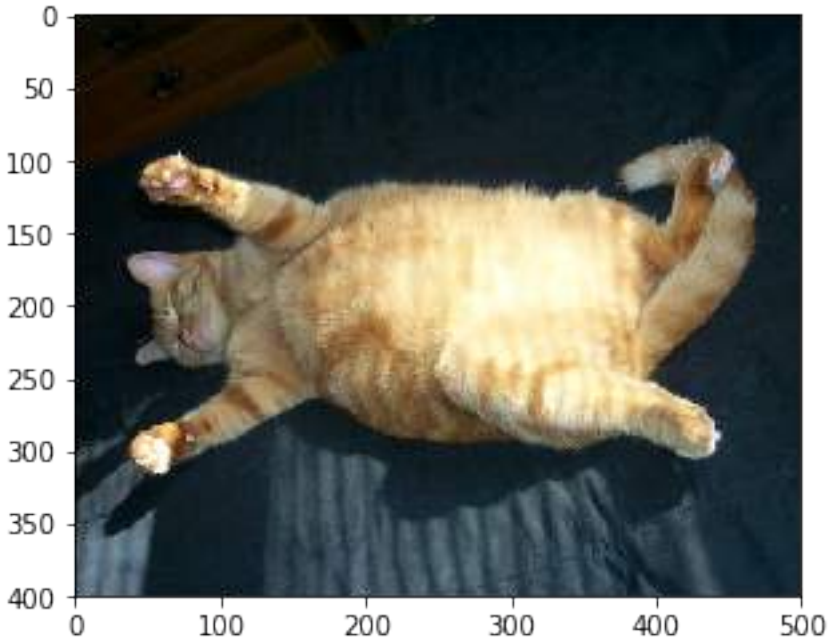
AlexNet 当年能取得巨大的成功，其中图片增广功不可没。图片增广通过一系列的随机变化生成大量“新”的样本，从而减低过拟合的可能。现在在深度卷积神经网络训练中，图片增广是必不可少的一部分。

9.1.1 常用增广方法

我们首先读取一张 400×500 的图片作为样例

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt
        from mxnet import image

        img = image.imdecode(open('../img/cat1.jpg', 'rb').read())
        plt.imshow(img.asnumpy())
```

```
Out[1]: <matplotlib.image.AxesImage at 0x7fed7c079048>
```



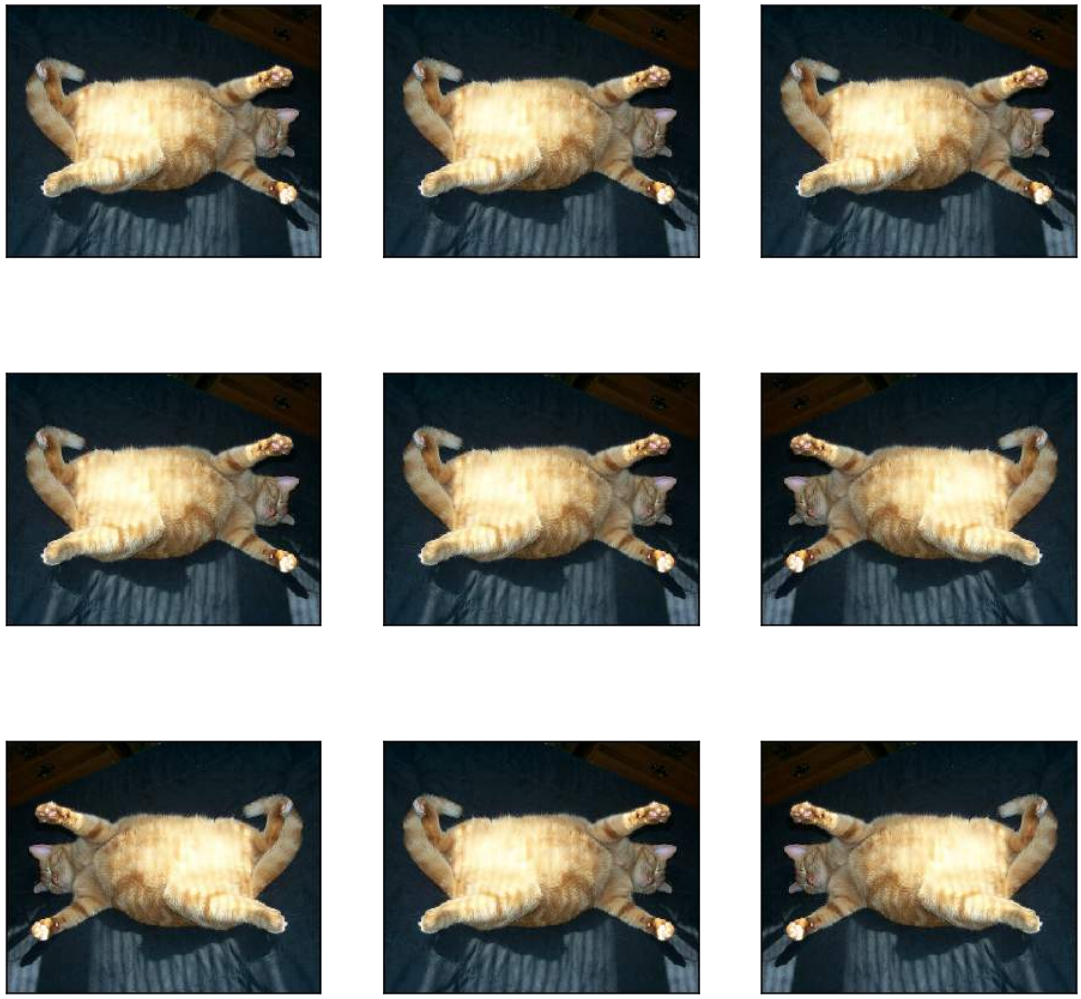
接下来我们定义一个辅助函数，给定输入图片 `img` 的增广方法 `aug`，它会运行多次并画出结果。

```
In [2]: from mxnet import nd
        import sys
        sys.path.append('..')
        import gluonbook as gb

        def apply(img, aug, n=3):
            # 转成 float，一是因为 aug 需要 float 类型数据来方便做变化。
            # 二是这里会有一次 copy 操作，因为有些 aug 直接通过改写输入
            #（而不是新建输出）获取性能的提升
            X = [aug(img.astype('float32')) for _ in range(n*n)]
            # 有些 aug 不保证输入是合法值，所以做一次 clip
            # 显示浮点图片时 imshow 要求输入在 [0,1] 之间
            Y = nd.stack(*X).clip(0,255)/255
            gb.show_images(Y, n, n, figsize=(8,8))
```

变形

水平方向翻转图片是最早也是最广泛使用的一种增广。

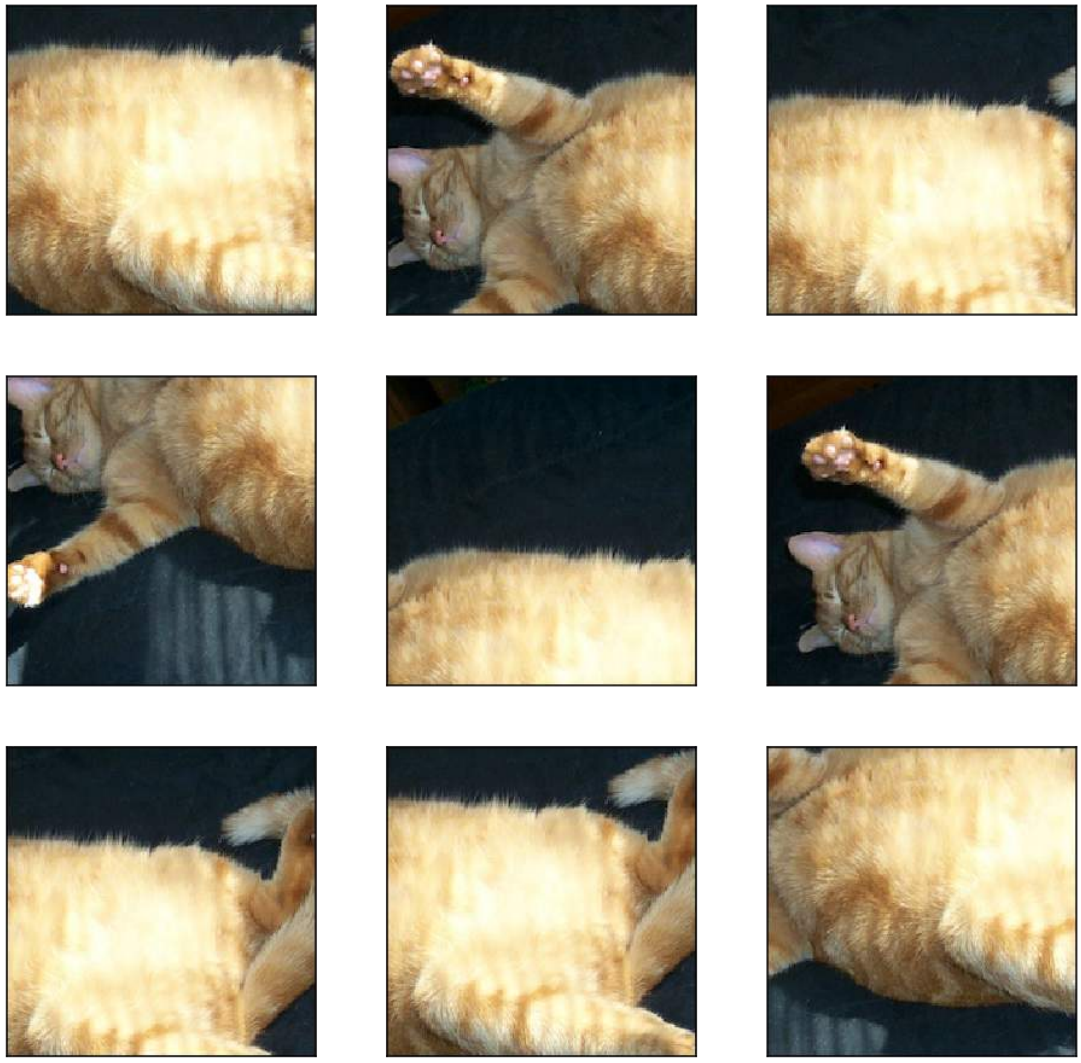
```
In [3]: # 以.5 的概率做翻转
        aug = image.HorizontalFlipAug(.5)
        apply(img, aug)
```



样例图片里我们关心的猫在图片正中间，但一般情况下可能不是这样。前面我们提到池化层能弱化卷积层对目标位置的敏感度，但也不能完全解决这个问题。一个常用增广方法是随机的截取其中的一块。

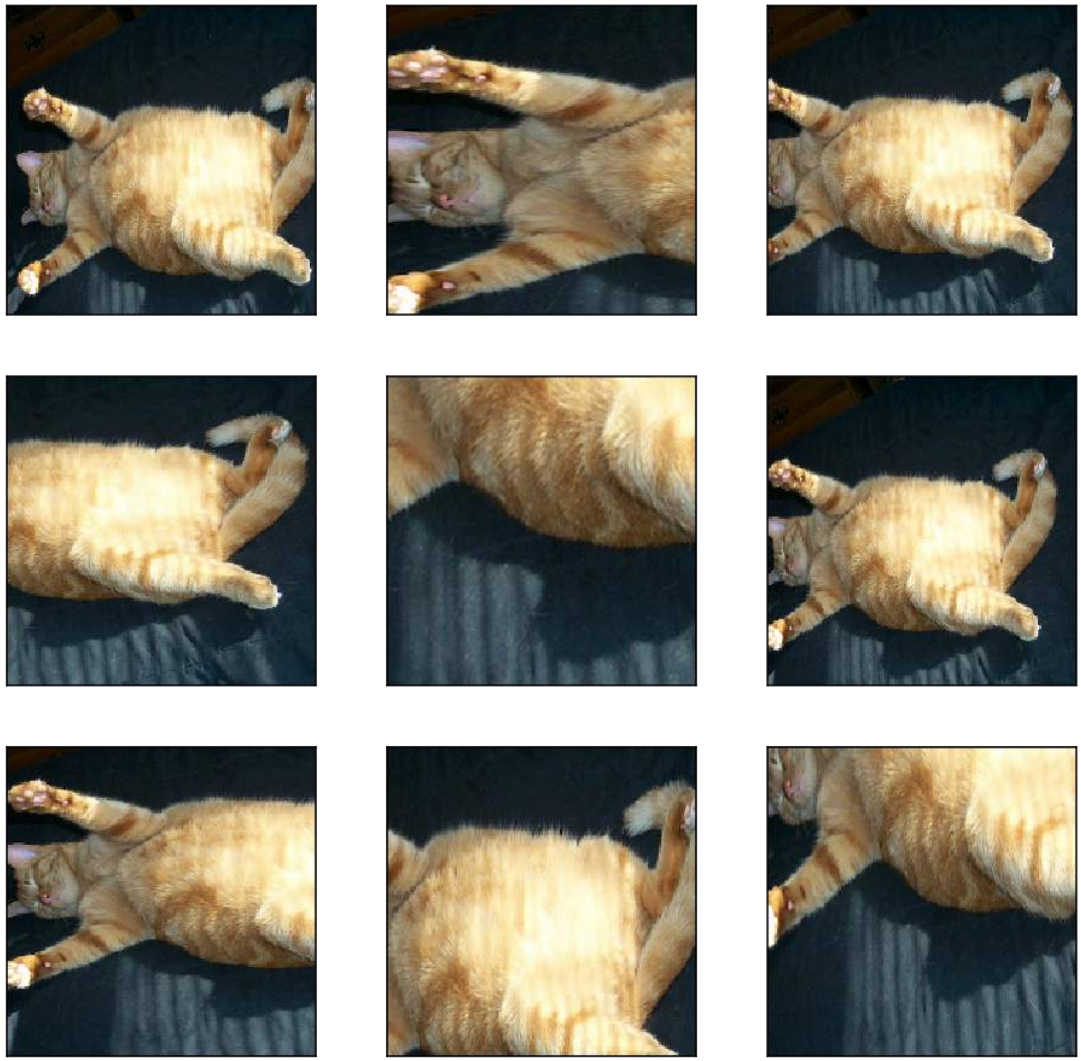
注意到随机截取一般会缩小输入的形状。如果原始输入图片过小，导致没有太多空间进行随机裁剪，通常做法是先将其放大的足够大的尺寸。所以如果你的原始图片足够大，建议不要事先将它们裁到网络需要的大小。

```
In [4]: # 随机裁剪一个块 200 x 200 的区域
        aug = image.RandomCropAug([200,200])
        apply(img, aug)
```



我们也可以随机裁剪一块随机大小的区域

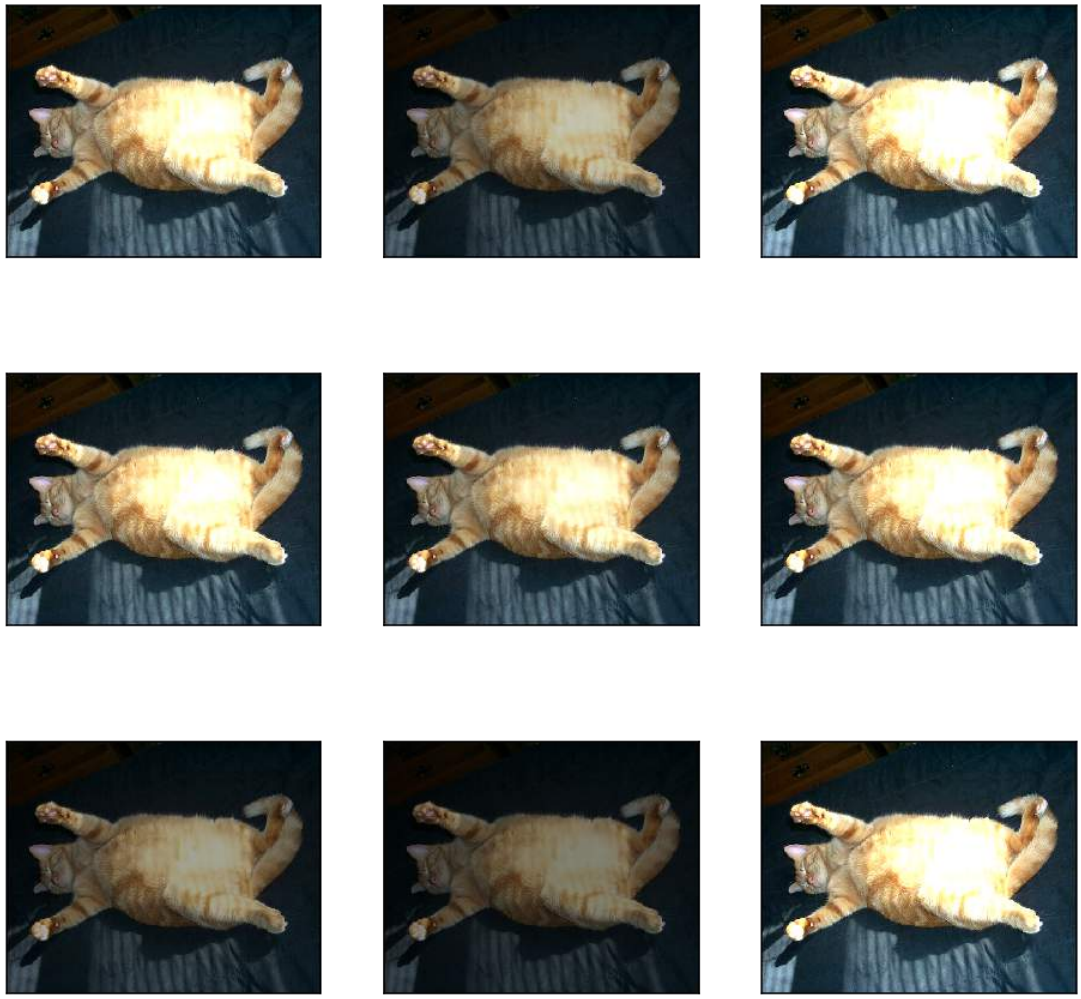
```
In [5]: # 随机裁剪，要求保留至少 0.1 的区域，随机长宽比在.5 和 2 之间。
        # 最后将结果 resize 到 200x200
        aug = image.RandomSizedCropAug((200,200), .1, (.5,2))
        apply(img, aug)
```



颜色变化

形状变化外的一个另一大类是变化颜色。

```
In [6]: # 随机将亮度增加或者减小在 0-50% 间的一个量
        aug = image.BrightnessJitterAug(.5)
        apply(img, aug)
```



```
In [7]: # 随机色调变化
        aug = image.HueJitterAug(.5)
        apply(img, aug)
```

9.1.2 如何使用

通常使用时我们会将数个增广方法一起使用。注意到图片增广通常只是针对训练数据，对于测试数据则用得较小。后者常用的是做 5 次随机剪裁，然后讲 5 张图片的预测结果做均值。

下面我们使用 CIFAR10 来演示图片增广对训练的影响。我们这里不使用前面一直用的 Fashion-MNIST，这是因为这个数据的图片基本已经对齐好了，而且是黑白图片，所以不管是变形还是变色增广效果都不会明显。

数据读取

我们首先定义一个辅助函数可以对图片按顺序应用数个增广：

```
In [8]: def apply_aug_list(img, augs):
            for f in augs:
                img = f(img)
            return img
```

对于训练图片我们随机水平翻转和剪裁。对于测试图片仅仅就是中心剪裁。CIFAR10 图片尺寸是 $32 \times 32 \times 3$，我们剪裁成 $28 \times 28 \times 3$.

```
In [9]: train_augs = [
            image.HorizontalFlipAug(.5),
            image.RandomCropAug((28,28))
        ]

        test_augs = [
            image.CenterCropAug((28,28))
        ]
```

然后定义数据读取，这里跟前面的 FashionMNIST 类似，但在 `transform` 中加入了图片增广：

```
In [10]: from mxnet import gluon
         from mxnet import nd
         import sys
         sys.path.append('..')
         import utils

         def get_transform(augs):
             def transform(data, label):
                 # data: sample x height x width x channel
                 # label: sample
                 data = data.astype('float32')
                 if augs is not None:
                     # apply to each sample one-by-one and then stack
                     data = nd.stack(*[
                         apply_aug_list(d, augs) for d in data])
                 data = nd.transpose(data, (0,3,1,2))
                 return data, label.astype('float32')
             return transform

         def get_data(batch_size, train_augs, test_augs=None):
             cifar10_train = gluon.data.vision.CIFAR10(
                 train=True, transform=get_transform(train_augs))
```

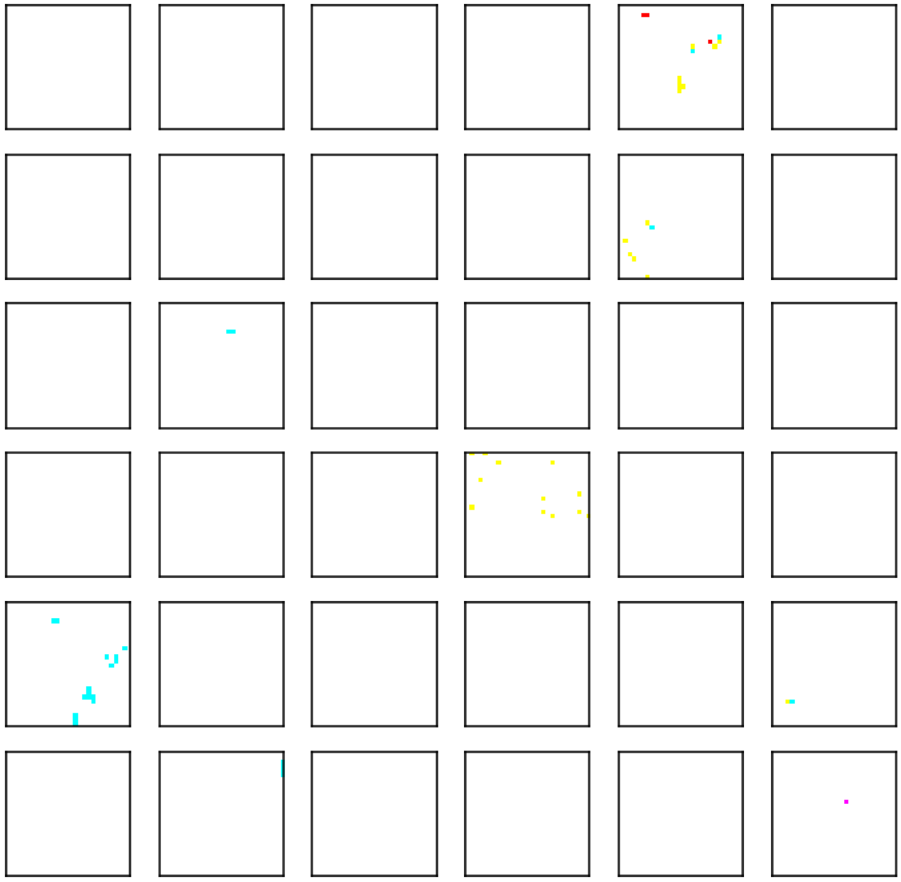
```
    cifar10_test = gluon.data.vision.CIFAR10(
        train=False, transform=get_transform(test_augs))
    train_data = gb.DataLoader(
        cifar10_train, batch_size, shuffle=True)
    test_data = gb.DataLoader(
        cifar10_test, batch_size, shuffle=False)
    return (train_data, test_data)
```

画出前几张看看

```
In [11]: train_data, _ = get_data(36, train_augs)
         for imgs, _ in train_data:
             break
         gb.show_images(imgs.transpose((0,2,3,1)), 6, 6)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
 ↪  [0..255] for integers).
```



In [12]: imgs.shape

Out[12]: (36, 3, 28, 28)

9.1.3 训练

我们使用 ResNet 18 训练。并且训练代码整理成一个函数使得可以重读调用：

In [13]:
```
from mxnet import init

def train(train_augs, test_augs, learning_rate=.1):
    batch_size = 128
    num_epochs = 10
    ctx = gb.try_all_gpus()
```

```
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    train_data, test_data = get_data(
        batch_size, train_augs, test_augs)
    net = gb.resnet18(10)
    net.initialize(ctx=ctx, init=init.Xavier())
    net.hybridize()
    trainer = gluon.Trainer(net.collect_params(),
                            'sgd', {'learning_rate': learning_rate})
    gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs)
```

使用增广：

```
In [14]: train(train_augs, test_augs)
```

```
training on  [gpu(0), gpu(1)]
epoch 1, loss 1.4918, train acc 0.468, test acc 0.534, time 19.6 sec
epoch 2, loss 1.0598, train acc 0.625, test acc 0.669, time 19.5 sec
epoch 3, loss 0.8877, train acc 0.690, test acc 0.721, time 19.3 sec
epoch 4, loss 0.7690, train acc 0.731, test acc 0.719, time 19.3 sec
epoch 5, loss 0.6938, train acc 0.759, test acc 0.746, time 19.5 sec
epoch 6, loss 0.6324, train acc 0.780, test acc 0.764, time 19.3 sec
epoch 7, loss 0.5855, train acc 0.796, test acc 0.783, time 19.0 sec
epoch 8, loss 0.5473, train acc 0.809, test acc 0.787, time 19.0 sec
epoch 9, loss 0.5154, train acc 0.822, test acc 0.746, time 18.9 sec
epoch 10, loss 0.4798, train acc 0.833, test acc 0.785, time 18.9 sec
```

不使用增广：

```
In [15]: train(test_augs, test_augs)
```

```
training on  [gpu(0), gpu(1)]
epoch 1, loss 1.4479, train acc 0.484, test acc 0.562, time 17.5 sec
epoch 2, loss 0.9969, train acc 0.650, test acc 0.630, time 17.1 sec
epoch 3, loss 0.7720, train acc 0.731, test acc 0.667, time 17.0 sec
epoch 4, loss 0.6123, train acc 0.785, test acc 0.670, time 17.1 sec
epoch 5, loss 0.4728, train acc 0.835, test acc 0.736, time 17.4 sec
epoch 6, loss 0.3563, train acc 0.877, test acc 0.737, time 18.4 sec
epoch 7, loss 0.2651, train acc 0.908, test acc 0.708, time 17.2 sec
epoch 8, loss 0.1872, train acc 0.936, test acc 0.710, time 17.0 sec
epoch 9, loss 0.1341, train acc 0.955, test acc 0.730, time 17.1 sec
epoch 10, loss 0.1033, train acc 0.965, test acc 0.711, time 17.2 sec
```

可以看到使用增广后，训练精度提升更慢，但测试精度比不使用更好。

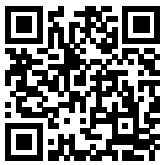
9.1.4 小结

- 图片增广可以有效避免过拟合。

9.1.5 练习

- 尝试换不同的增广方法试试。

9.1.6 扫码直达讨论区



9.2 迁移学习

在前面的章节里我们展示了如何训练神经网络来识别小图片里的问题。我们也介绍了 ImageNet 这个学术界默认的数据集，它有超过一百万的图片和一千类的物体。这个数据集很大的改变计算机视觉这个领域，展示了很多事情虽然在小的数据集上做不到，但在数 GB 的大数据上是可能的。事实上，我们目前还不知道有什么技术可以在类似的但小图片数据集上，例如一万张图片，训练出一个同样强大的模型。

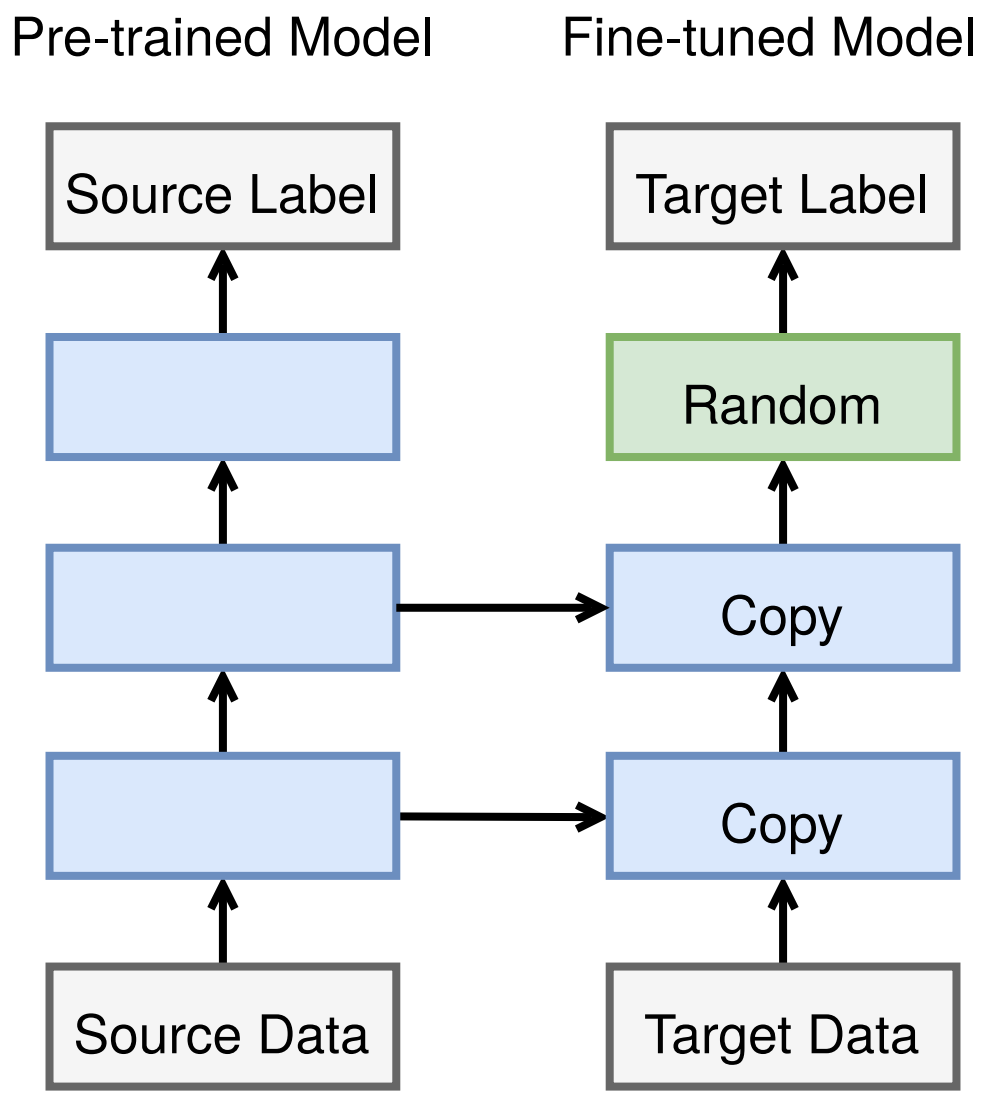
所以这是一个问题。尽管深度卷积神经网络在 ImageNet 上有了很惊讶的结果，但大部分人不关心 Imagenet 这个数据集本身。他们关心他们自己的问题。例如通过图片里面的人脸识别身份，或者识别图片里面的 10 种不同的珊瑚。通常大部分在非 BAT 类似大机构里的人在解决计算机视觉问题的时候，能获得的只是相对来说中等规模的数据。几百张图片很正常，找到几千张图片也有可能，但很难同 Imagenet 一样获得上百万张图片。

于是我们会有一个很自然的问题，如何使用在百万张图片上训练出来的强大的模型来帮助提升在小数据集上的精度呢？这种在源数据上训练，然后将学到的知识应用到目标数据集上的技术通常被叫做**迁移学习**。幸运的是，我们有一些有效的技术来解决这个问题。

对于深度神经网络来首，最为流行的一个方法叫做微调（fine-tuning）。它的想法很简单但有效：

- 在源数据 S 上训练一个神经网络。
- 砍掉它的头，将它的输出层改成适合目标数据 S 的大小
- 将输出层的权重初始化成随机值，但其它层保持跟原先训练好的权重一致
- 然后开始在目标数据集开始训练

下图图示了这个算法：



9.2.1 热狗识别

这一章我们将通过 ResNet 来演示如何进行微调。因为通常不会每次从 0 开始在 ImageNet 上训练模型，我们直接从 Gluon 的模型园下载已经训练好的。然后将其迁移到一个我们感兴趣的问题上：识别热狗。

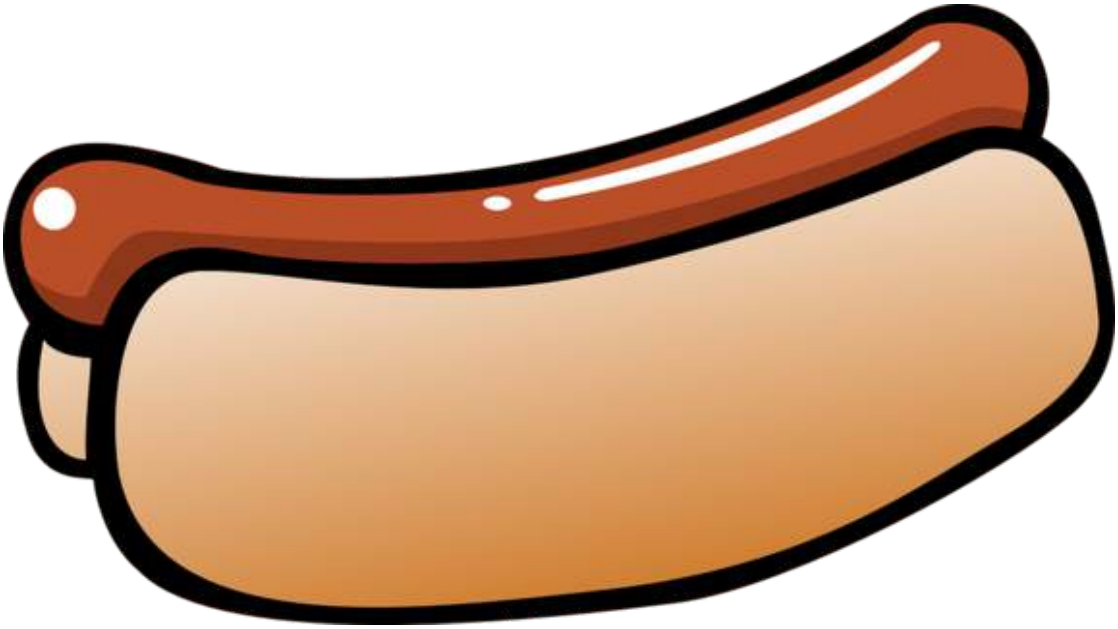


图 9.1: hot dog

热狗识别是一个二分类问题。我们这里使用的热狗数据集是从网上抓取的，它有 1400 张正类和同样多的负类，负类主要是食品相关图片。我们将各类的 1000 张作为训练集合，其余的作为测试集合。

获取数据

我们首先从网上下载数据并解压到`../data/hotdog`。每个文件夹下会有对应的 `png` 文件。

```
In [1]: from mxnet import gluon
        import zipfile

        data_dir = '../data'
```

```
        fname = gluon.utils.download(
    ↪  'https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/hotdog.zip',
            path=data_dir, sha1_hash='fba480ffa8aa7e0febbb511d181409f899b9baa5')

        with zipfile.ZipFile(fname, 'r') as f:
            f.extractall(data_dir)
```

我们使用图片增强里类似的方法来处理图片。

```
In [2]: from mxnet import nd
        from mxnet import image
        from mxnet import gluon

        train_augs = [
            image.HorizontalFlipAug(.5),
            image.RandomCropAug((224,224))
        ]

        test_augs = [
            image.CenterCropAug((224,224))
        ]

        def transform(data, label, augs):
            data = data.astype('float32')
            for aug in augs:
                data = aug(data)
            data = nd.transpose(data, (2,0,1))
            return data, nd.array([label]).asscalar().astype('float32')
```

读取文件夹下的图片，并且画出一些图片

```
In [3]: %matplotlib inline
        import sys
        sys.path.append('..')
        import gluonbook as gb

        train_imgs = gluon.data.vision.ImageFolderDataset(
            data_dir+'/hotdog/train',
            transform=lambda X, y: transform(X, y, train_augs))
        test_imgs = gluon.data.vision.ImageFolderDataset(
            data_dir+'/hotdog/test',
            transform=lambda X, y: transform(X, y, test_augs))

        data = gluon.data.DataLoader(train_imgs, 32, shuffle=True)
```

```
for X, _ in data:
    X = X.transpose((0,2,3,1)).clip(0,255)/255
    gb.show_images(X, 4, 8)
    break
```

模型和训练

这里我们将使用 Gluon 提供的 ResNet18 来训练。我们先从模型园里获取改良过 ResNet。使用 `pretrained=True` 将会自动下载并加载从 ImageNet 数据集上训练而来的权重。

```
In [4]: from mxnet.gluon.model_zoo import vision as models

        pretrained_net = models.resnet18_v2(pretrained=True)
```

通常预训练好的模型由两块构成，一是 `features`，二是 `output`。后者主要包括最后一层全连接层，前者包含从输入开始的大部分层。这样的划分的一个主要目的是为了更方便做微调。我们先看下 `output` 的内容：

```
In [5]: pretrained_net.output

Out[5]: Dense(512 -> 1000, linear)
```

我们可以看一下第一个卷积层的部分权重。

```
In [6]: pretrained_net.features[1].weight.data()[0][0]
```

```
Out[6]:
        [[-0.04693392  0.11487006 -0.13209556  0.16124195 -0.21484604  0.18044543
          -0.05956454]
         [-0.00242769 -0.03129578  0.01799692  0.15277492 -0.41541672  0.38176033
          -0.13370997]
         [ 0.10314132 -0.30472746  0.59482247 -0.52606624  0.0621427   0.25646785
          -0.12772678]
         [ 0.01783164 -0.21222414  0.58199424 -0.84664404  0.57027811 -0.20741715
           0.02784866]
         [ 0.01255781 -0.02931368  0.1608634  -0.33185521  0.31180814 -0.16463067
           0.05555796]
         [-0.0167121   0.03173966  0.00400858 -0.02572511 -0.02412852  0.08885808
          -0.04472235]
         [-0.05655501  0.08309566 -0.08147315  0.02597015 -0.03567177  0.0657132
          -0.03488606]]
        <NDArray 7x7 @cpu(0)>
```

在微调里，我们一般新建一个网络，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。新网络的 `features` 被初始化前面训练好网络的权重，而 `output` 则是从头开始训练。

```
In [7]: from mxnet import init

        finetune_net = models.resnet18_v2(classes=2)
        finetune_net.features = pretrained_net.features
        finetune_net.output.initialize(init.Xavier())
```

我们先定义一个可以重复使用的训练函数。

```
In [8]:
        def train(net, ctx, batch_size=64, epochs=10, learning_rate=0.01, wd=0.001):
            train_data = gluon.data.DataLoader(train_imgs, batch_size, shuffle=True)
            test_data = gluon.data.DataLoader(test_imgs, batch_size)

            # 确保 net 的初始化在 ctx 上
            net.collect_params().reset_ctx(ctx)
            net.hybridize()
            loss = gluon.loss.SoftmaxCrossEntropyLoss()
            # 训练
            trainer = gluon.Trainer(net.collect_params(), 'sgd', {
                'learning_rate': learning_rate, 'wd': wd})
            gb.train(train_data, test_data, net, loss, trainer, ctx, epochs)
```

现在我们可以训练了。

```
In [9]: ctx = gb.try_all_gpus()
        train(finetune_net, ctx)
```

```
training on  [gpu(0), gpu(1)]
epoch 1, loss 0.3559, train acc 0.841, test acc 0.927, time 16.4 sec
epoch 2, loss 0.1549, train acc 0.942, test acc 0.924, time 14.2 sec
epoch 3, loss 0.1335, train acc 0.950, test acc 0.906, time 14.3 sec
epoch 4, loss 0.1165, train acc 0.957, test acc 0.944, time 14.2 sec
epoch 5, loss 0.0978, train acc 0.962, test acc 0.949, time 14.4 sec
epoch 6, loss 0.0738, train acc 0.975, test acc 0.940, time 14.3 sec
epoch 7, loss 0.0622, train acc 0.979, test acc 0.949, time 14.3 sec
epoch 8, loss 0.0644, train acc 0.977, test acc 0.958, time 14.2 sec
epoch 9, loss 0.0717, train acc 0.975, test acc 0.943, time 14.3 sec
epoch 10, loss 0.0563, train acc 0.980, test acc 0.950, time 14.2 sec
```

对比起见我们尝试从随机初始值开始训练一个网络。

```
In [10]: scratch_net = models.resnet18_v2(classes=2)
         scratch_net.initialize(init=init.Xavier())
         train(scratch_net, ctx)
```

```
training on  [gpu(0), gpu(1)]
epoch 1, loss 0.4489, train acc 0.796, test acc 0.828, time 14.5 sec
epoch 2, loss 0.3633, train acc 0.842, test acc 0.789, time 14.3 sec
epoch 3, loss 0.3507, train acc 0.851, test acc 0.818, time 14.7 sec
epoch 4, loss 0.3200, train acc 0.862, test acc 0.843, time 14.2 sec
epoch 5, loss 0.3229, train acc 0.861, test acc 0.825, time 14.2 sec
epoch 6, loss 0.3177, train acc 0.867, test acc 0.835, time 14.3 sec
epoch 7, loss 0.3220, train acc 0.858, test acc 0.825, time 14.3 sec
epoch 8, loss 0.3053, train acc 0.870, test acc 0.775, time 14.3 sec
epoch 9, loss 0.2939, train acc 0.876, test acc 0.826, time 14.3 sec
epoch 10, loss 0.2763, train acc 0.878, test acc 0.843, time 14.2 sec
```

可以看到，微调版本收敛比从随机值开始的要快很多。

图片预测

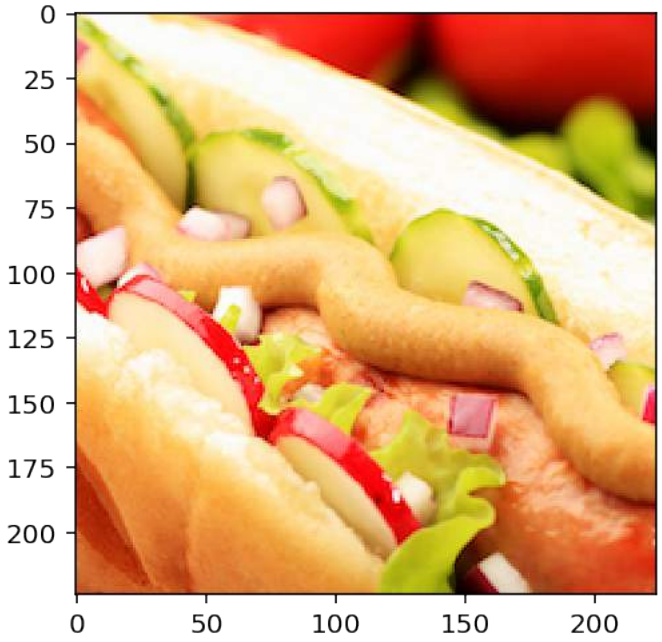
```
In [11]: import matplotlib.pyplot as plt

         def classify_hotdog(net, fname):
             with open(fname, 'rb') as f:
                 img = image.imdecode(f.read())
             data, _ = transform(img, -1, test_augs)
             plt.imshow(data.transpose((1,2,0)).asnumpy()/255)
```

```
    data = data.expand_dims(axis=0)
    out = net(data.as_in_context(ctx[0]))
    out = nd.SoftmaxActivation(out)
    pred = int(nd.argmax(out, axis=1).asscalar())
    prob = out[0][pred].asscalar()
    label = train_imgs.synsets
    return 'With prob=%f, %s'%(prob, label[pred])
```

接下来我们用训练好的模型来预测几张图片：

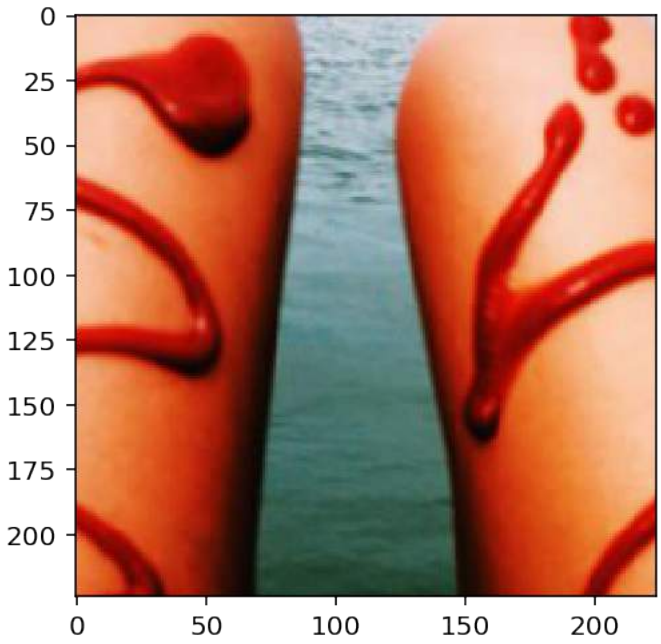
```
In [12]: classify_hotdog(finetune_net, '../img/real_hotdog.jpg')
```

```
Out[12]: 'With prob=0.999747, hotdog'
```



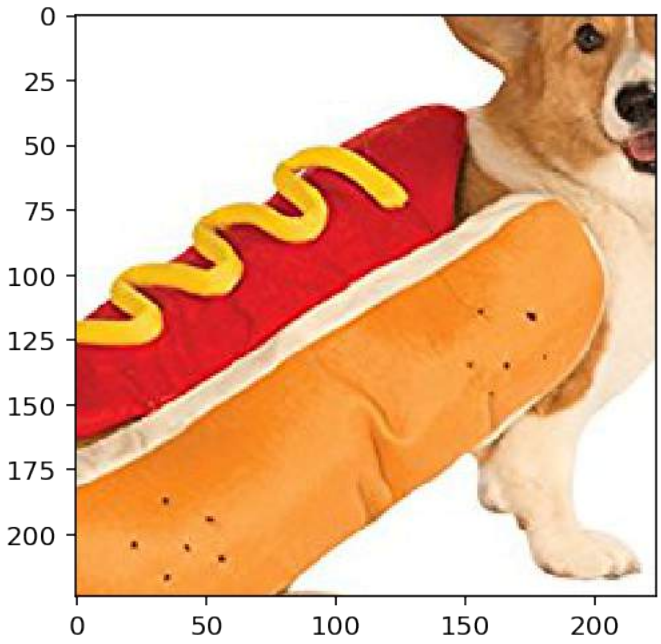

```
In [13]: classify_hotdog(finetune_net, '../img/leg_hotdog.jpg')
```

```
Out[13]: 'With prob=0.997494, hotdog'
```




```
In [14]: classify_hotdog(finetune_net, '../img/dog_hotdog.jpg')
```

```
Out[14]: 'With prob=0.737879, hotdog'
```



9.2.2 小结

- 我们看到，通过一个预先训练好的模型，我们可以在即使较小的数据集上训练得到很好的分类器。这是因为这两个任务里面的数据表示有很多共通性，例如都需要如何识别纹理、形状、边等等。而这些通常被在靠近数据的层有效的处理。因此，如果你有一个相对较小的数据在手，而且担心它可能不够训练出很好的模型，你可以寻找跟你数据类似的大数据集来先训练你的模型，然后再在你手上的数据集上微调。

9.2.3 练习

- 多跑几个 epochs 直到收敛（你可以也需要调调参数），看看 scratch_net 和 fine-tune_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，它的 index 是 713。例如它对应的 weight 可以这样拿到。试试如何重用这个权重

```
In [15]: weight = pretrained_net.output.weight
         hotdog_w = nd.split(weight.data(), 1000, axis=0)[713]
         hotdog_w.shape
```

```
Out[15]: (1, 512)
```

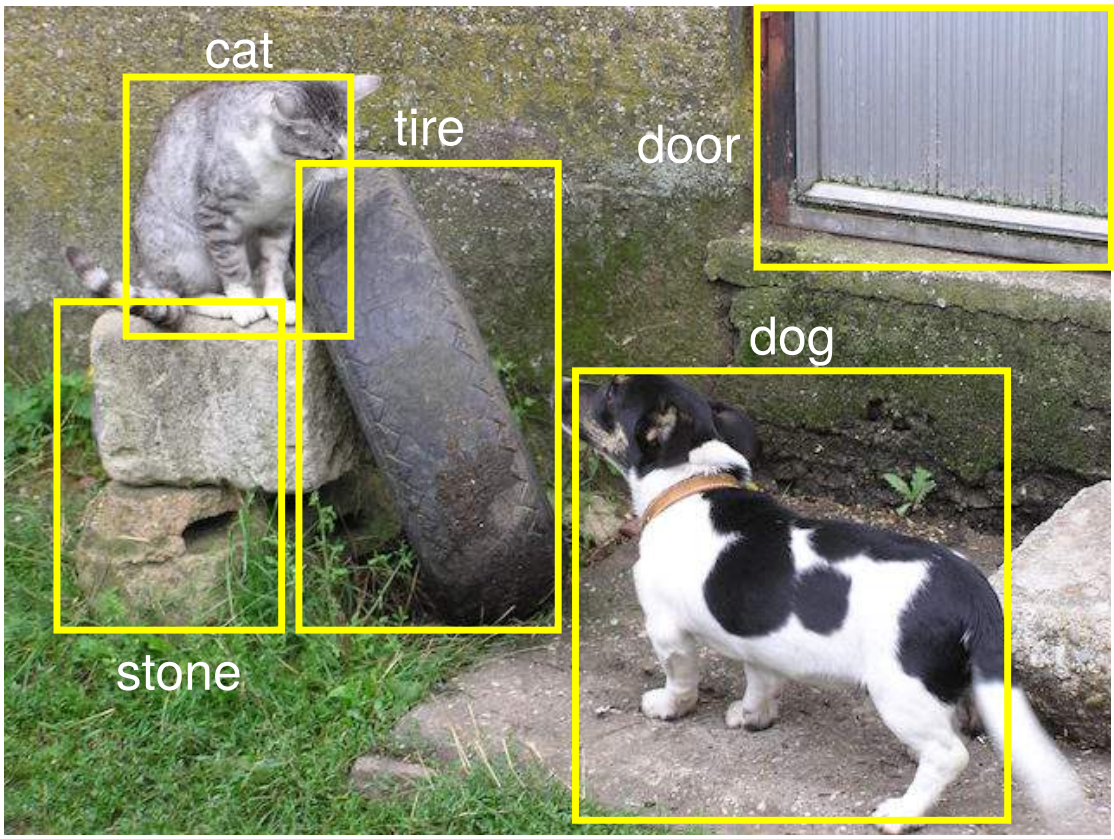
- 试试不让 `finetune_net` 里重用的权重参与训练，就是不更新权重
- 如果图片预测这一章里我们训练的模型没有分对所有的图片，如何改进？

9.2.4 扫码直达讨论区

9.3 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，通常也被称之为**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

9.3.1 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。

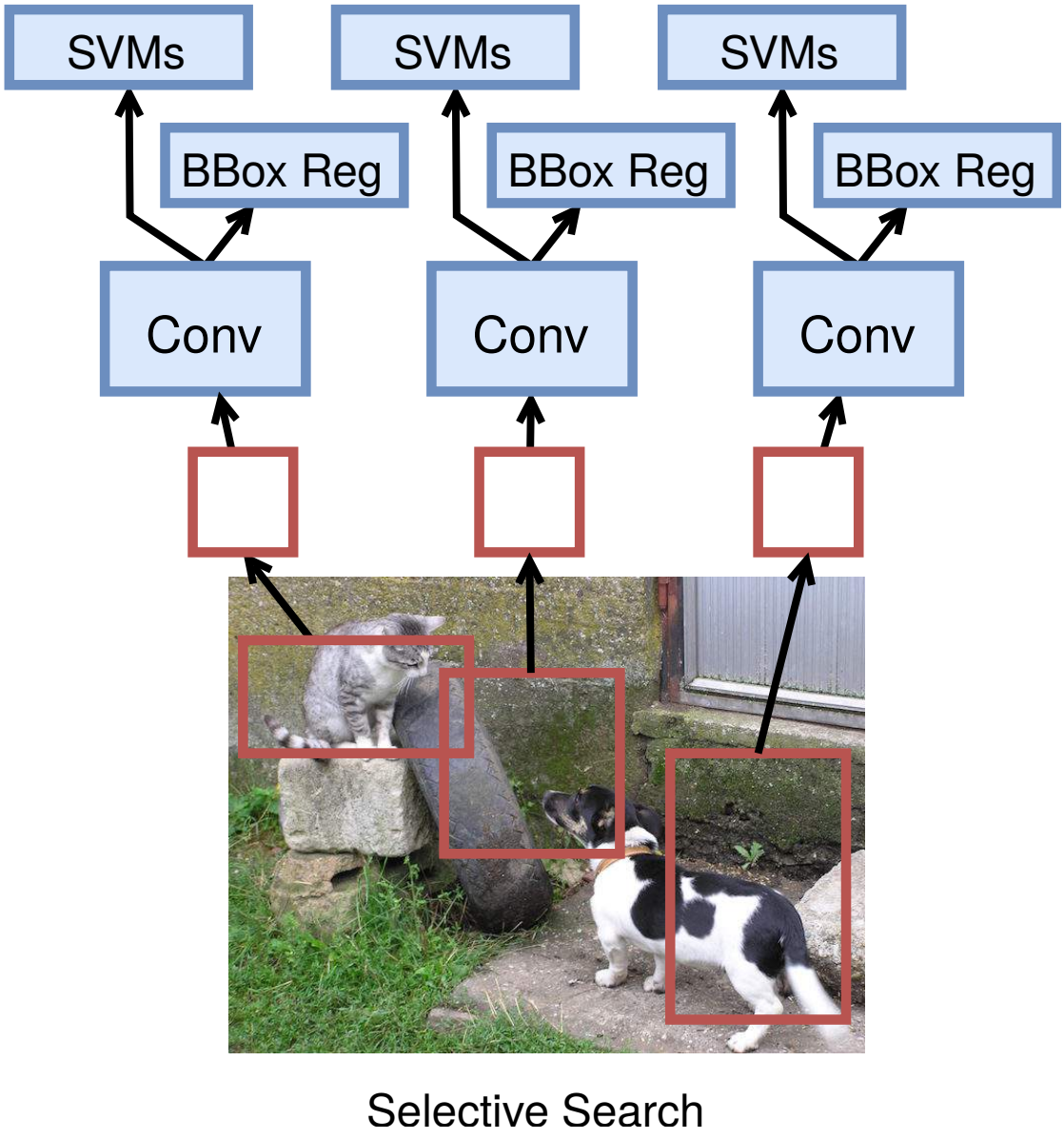


图 9.2: R-CNN

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。

9.3.2 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。

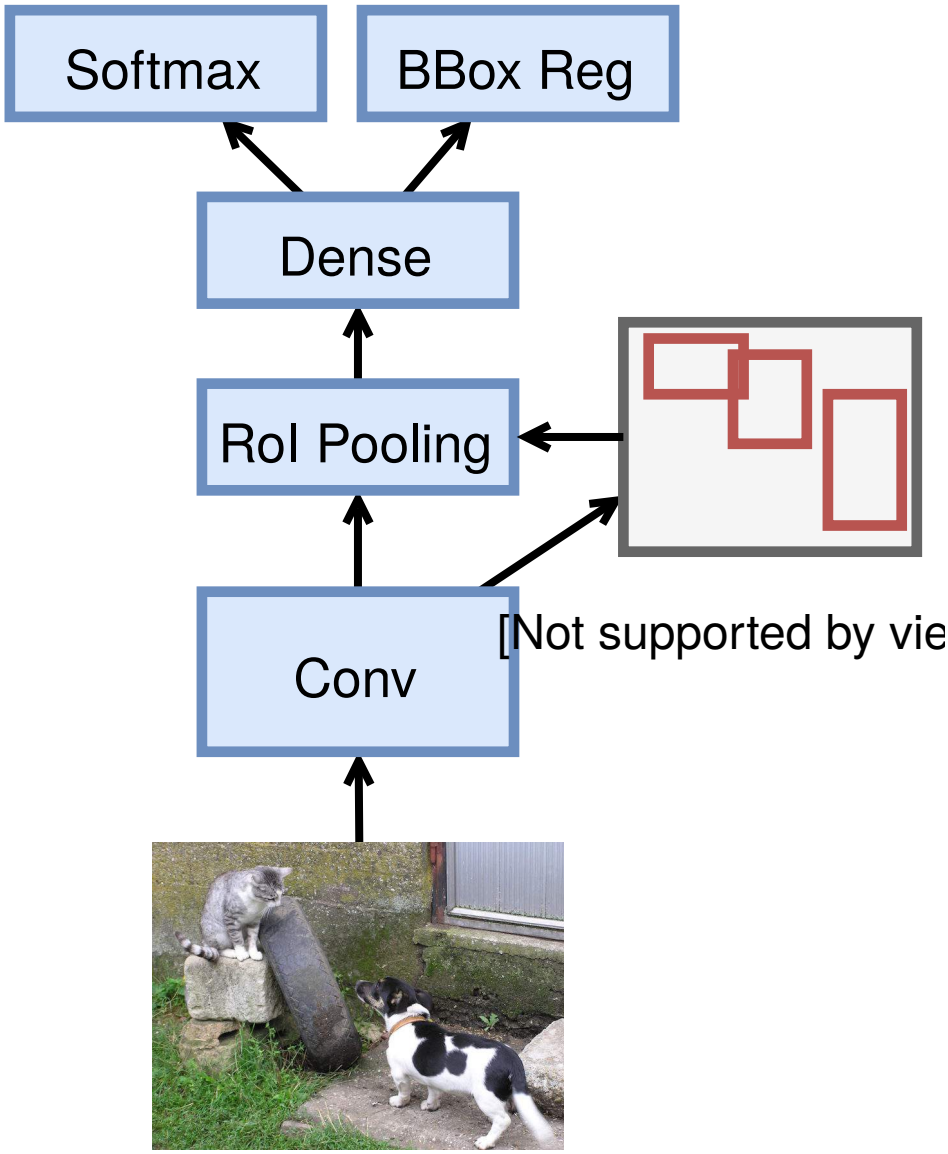


图 9.3: Fast R-CNN

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 4×4 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 3×3 和 4×3 大小的两个区域。

RoI 池化层的输出大小是 `num_regions x num_channels x n x 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)>
```

9.3.3 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⊠⊠⊠256⊠3×3 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

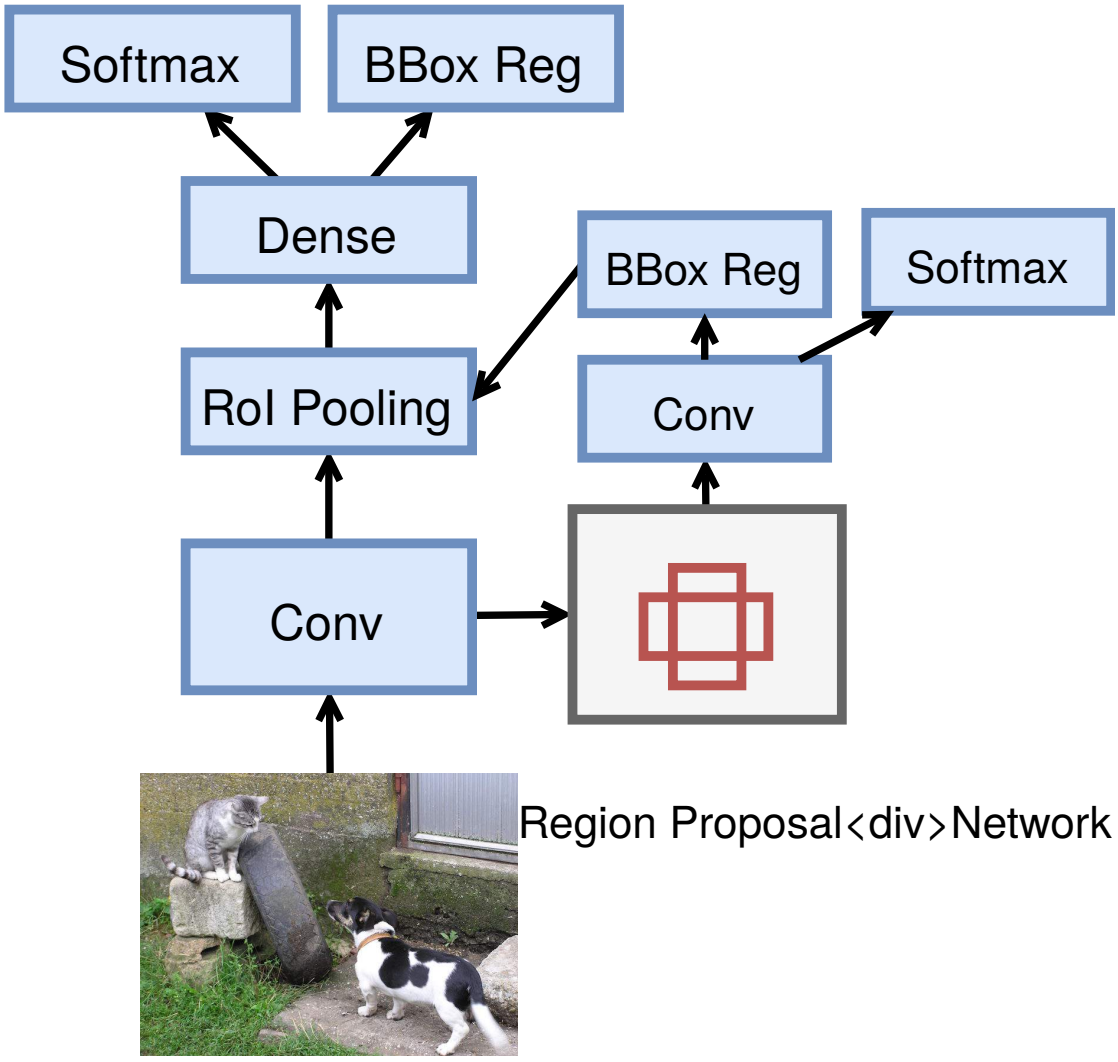


图 9.4: Faster R-CNN

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。

9.3.4 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。

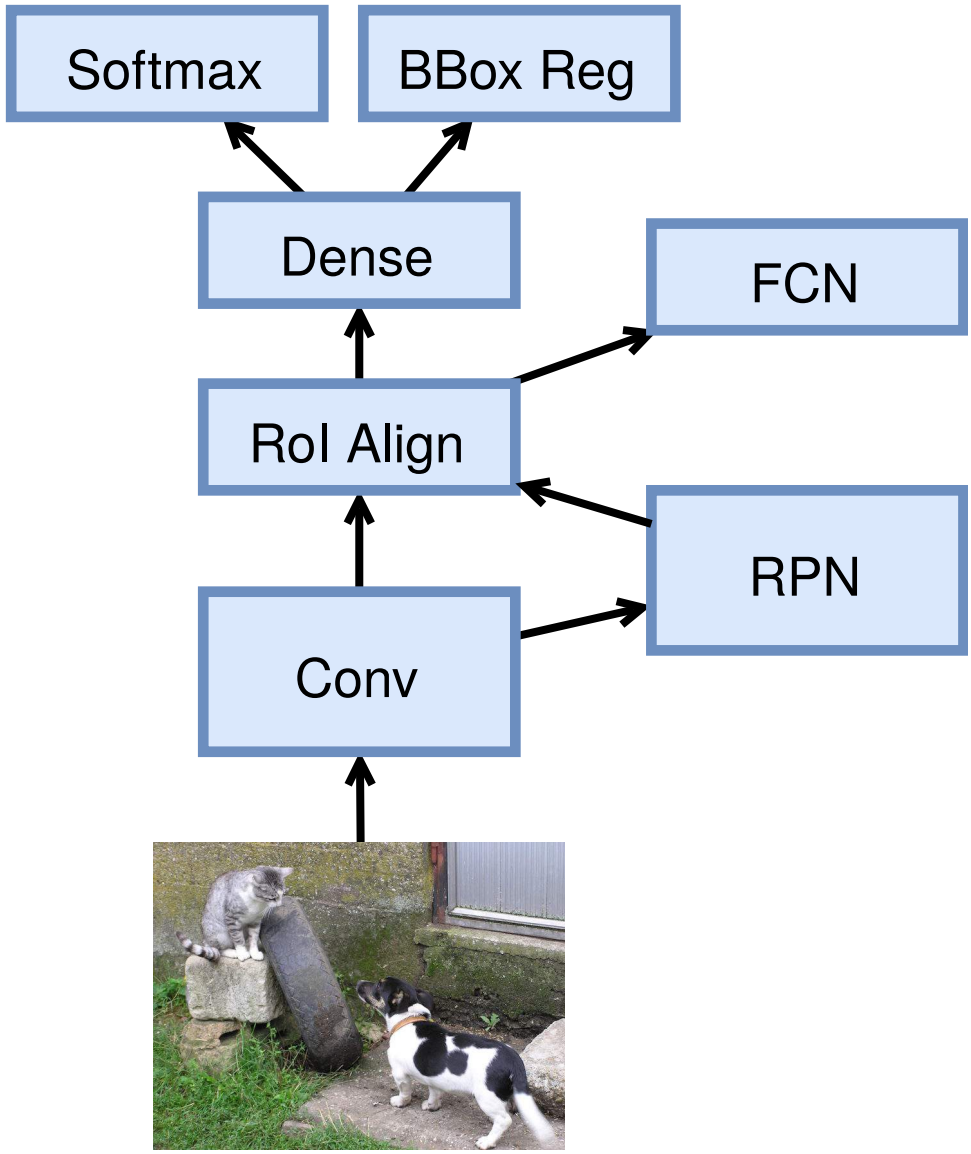


图 9.5: Mask R-CNN

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 256×256，那么特征大小就是 16×16。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 $f(x, y)$.

9.3.5 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。

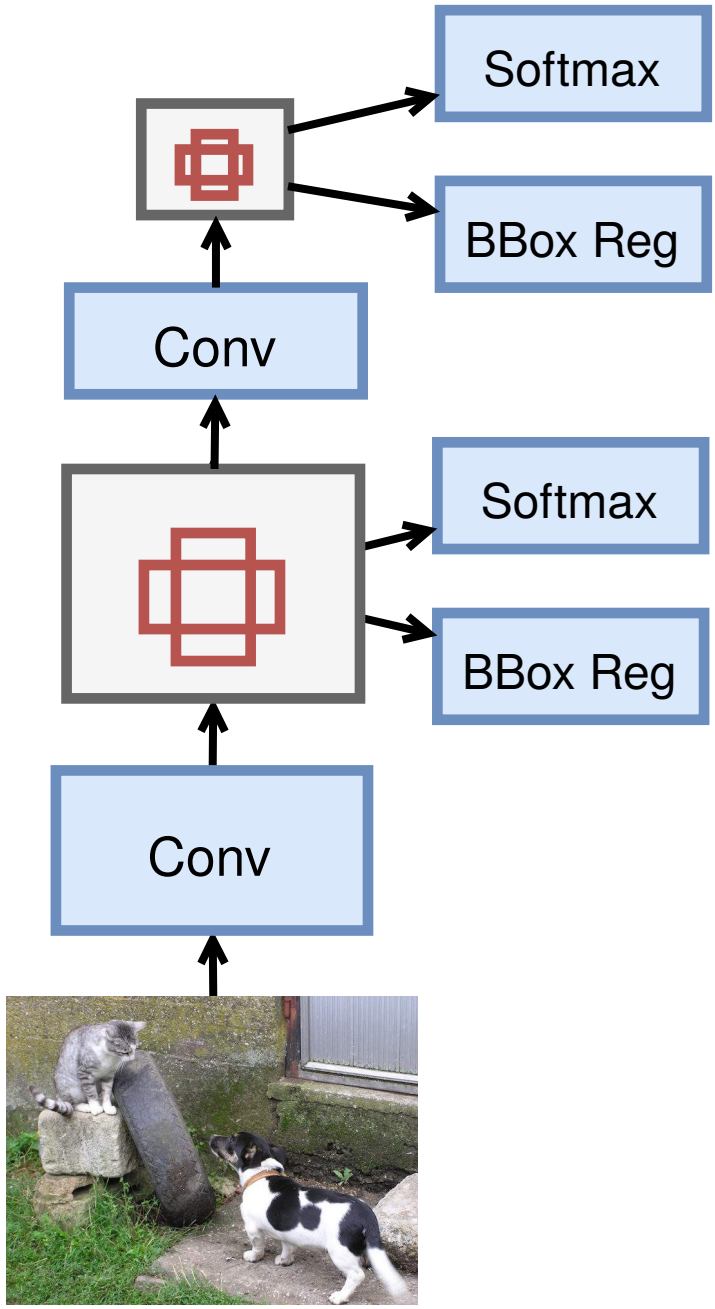


图 9.6: SSD

SSD 的具体实现将在下一章详细阐述。

9.3.6 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。

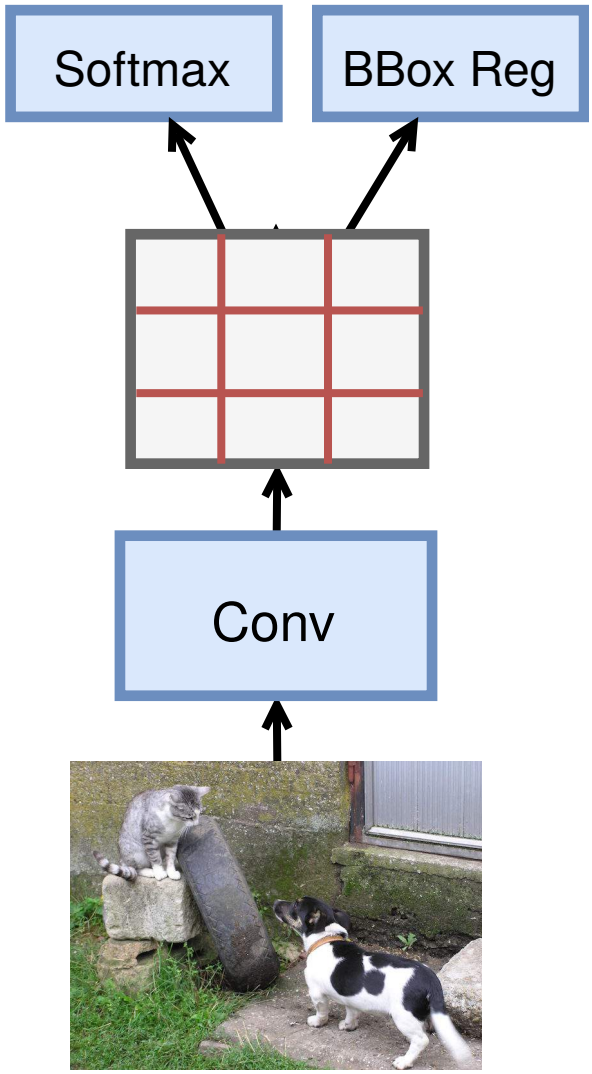


图 9.7: Yolo

9.3.7 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 448×448 使得特征输出大小增大到 13×13

2. 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。
3. 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+num_classes)` 的 1×1 卷积，边框回归使用通道数 `4*5`.

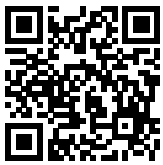
9.3.8 小结

- 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。

9.3.9 练习

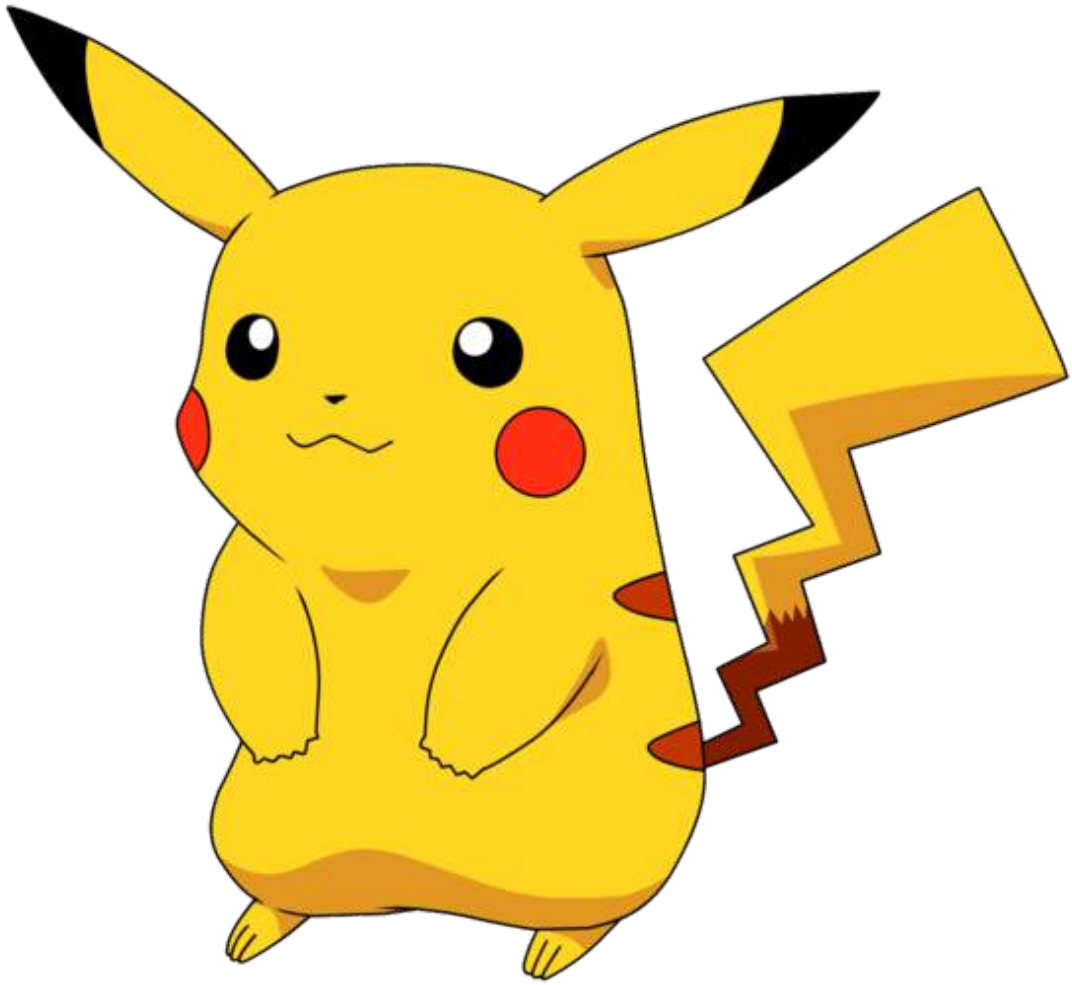
- TODO@mli

9.3.10 扫码直达讨论区



9.4 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。



9.4.1 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.ImageDetIter(
```

```
            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)
```

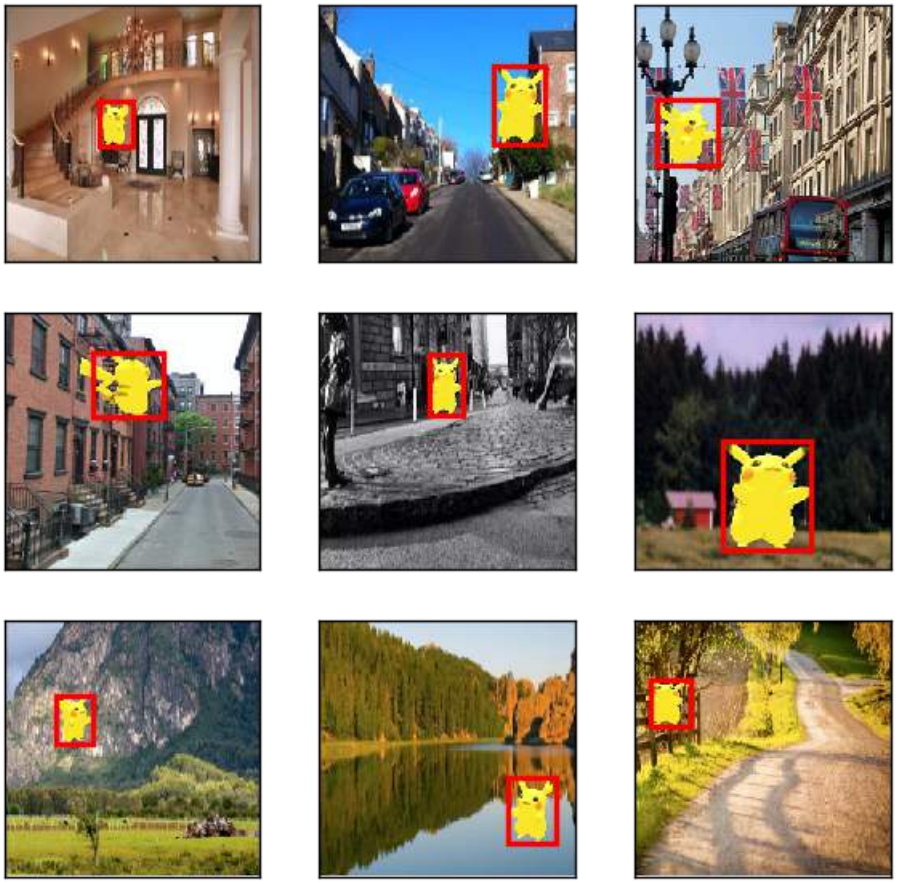
```
DataBatch: data shapes: [(32, 3, 256, 256)] label shapes: [(32, 1, 5)]
```

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
        import matplotlib.pyplot as plt

        def box_to_rect(box, color, linewidth=3):
            """convert an anchor box to a matplotlib rectangle"""
            box = box.asnumpy()
            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()
```



9.4.2 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 `contribe.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。

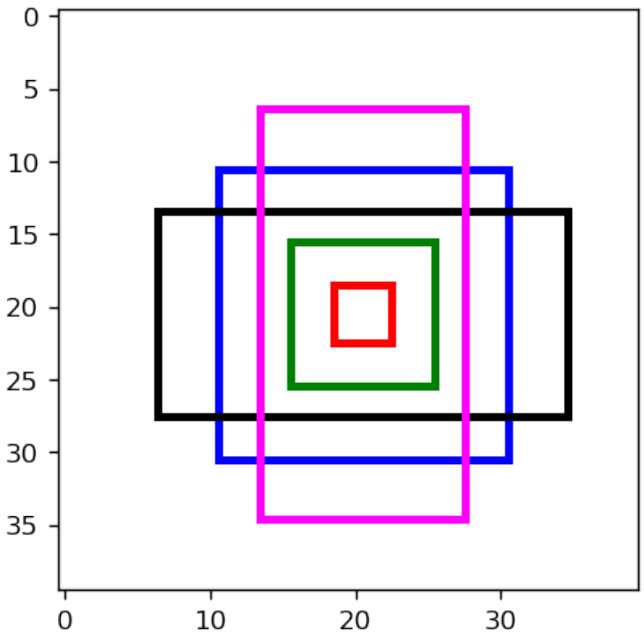
```
In [5]: from mxnet import nd
        from mxnet.contrib.ndarray import MultiBoxPrior

        # 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 3×3 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*(num_class+1)` 是其只包含背景的分数
- 通道 `a*(num_class+1)+1+b` 是其包含第 `b` 个物体的分数

我们定义个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变换成真正的边界框。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*4` 到 `a*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 3×3 卷积使得输入和输入有同样的长宽，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 y1 和 y2 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 achors:', 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 achors: (1, 5444, 4)
Output class predictions: (1, 5444, 3)
Output box predictions: (1, 21776)
```

9.4.3 训练

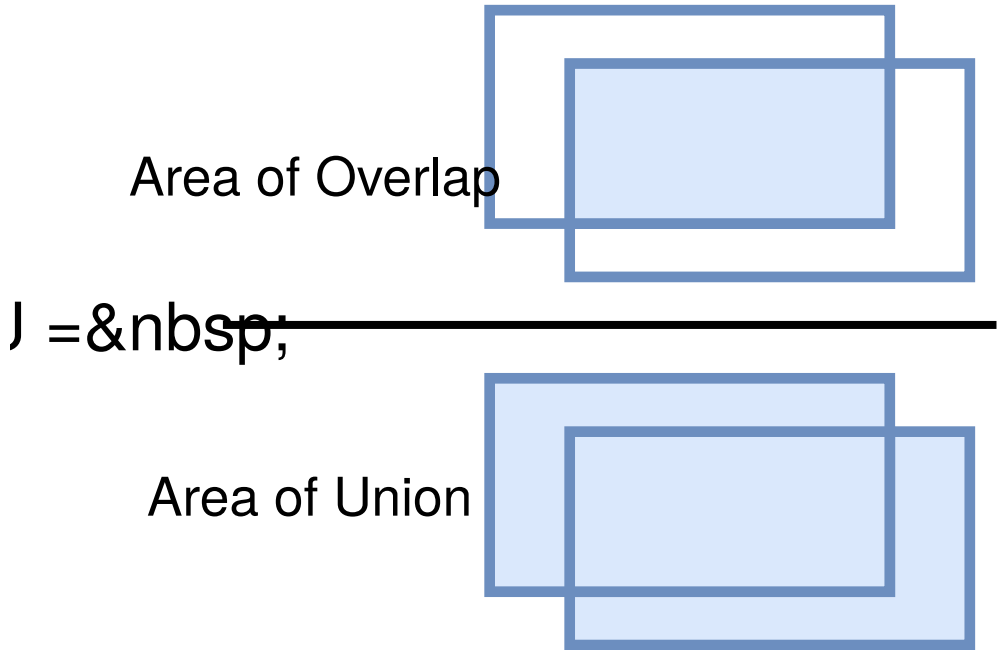
之前的教程我们主要是关注分类。对于分类的预测结果和真实的标号，我们通过交叉熵来计算他们的差异。但物体检测里我们需要预测边框。这里我们先引入一个概率来描述两个边框的距离。

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信它是负类的，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]:
[ 9.]
<NDArray 1 @cpu(0)>
```

然后我们可以定义需要的损失函数了。

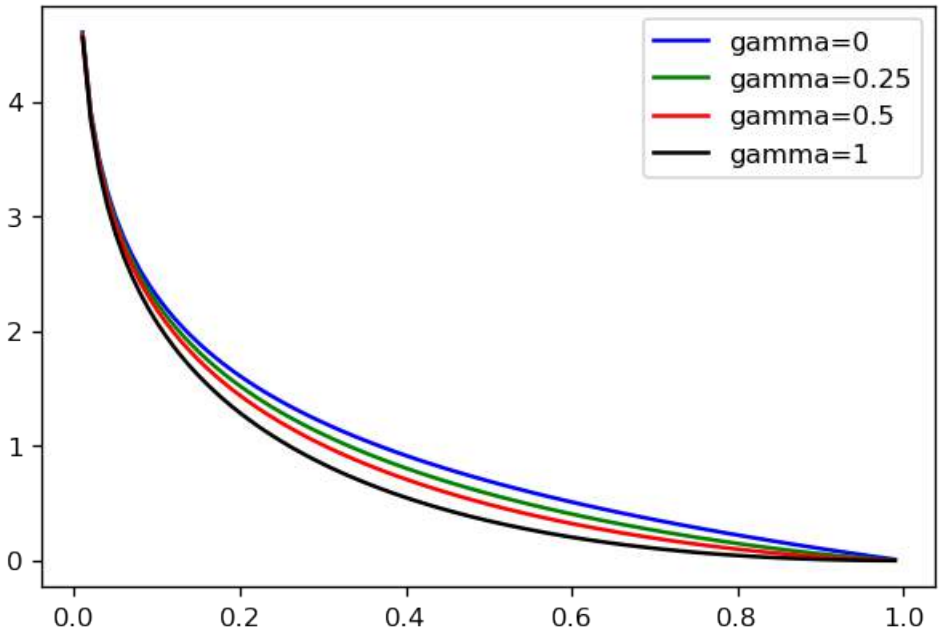
对于分类问题，最常用的损失函数是之前一直使用的交叉熵。这里我们定义一个类似于交叉熵的损失，不同于交叉熵的定义 $\log(p_j)$，这里 j 是真实的类别，且 p_j 是对于的预测概率。我们使用一个被称之为关注损失的函数，给定正的 γ 和 α，它的定义是

$$-\alpha(1-p_j)^\gamma \log(p_j)$$

下图我们演示不同 γ 导致的变化。可以看到，增加 γ 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)
```

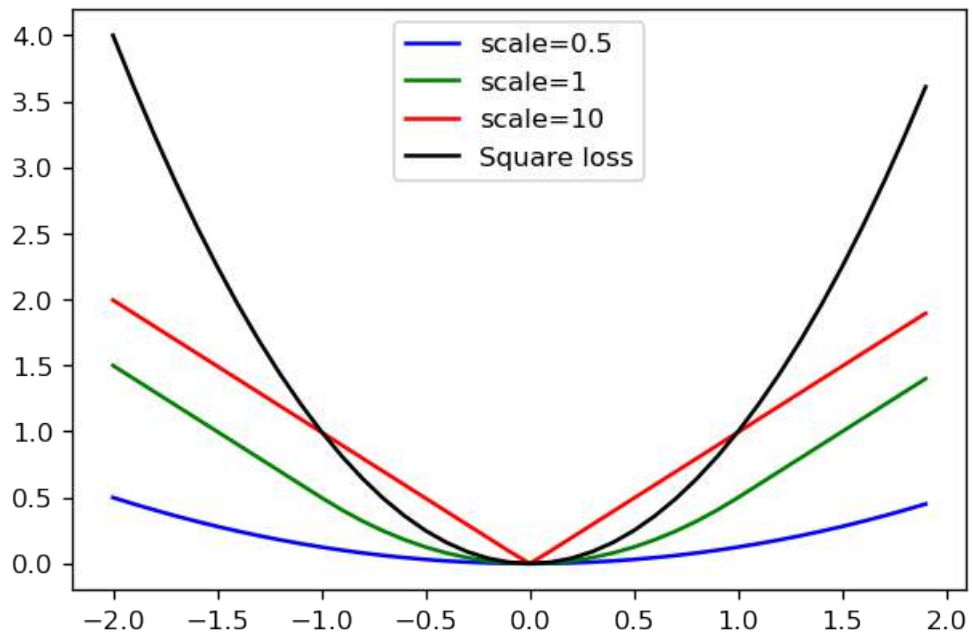
对于边框的预测是一个回归问题。通常可以选择平方损失函数（L2 损失）$f(x) = x^2$。但这个损失

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。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 σ 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 σ 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 requres each image has at least 3 lables.
         # Padd 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 0.00422, time 12.7 sec
Epoch  1, train accuracy 0.99, mae 0.00359, time 10.6 sec
Epoch  2, train accuracy 0.99, mae 0.00342, time 10.7 sec
Epoch  3, train accuracy 0.99, mae 0.00325, time 10.7 sec
Epoch  4, train accuracy 0.99, mae 0.00320, time 10.6 sec
Epoch  5, train accuracy 0.99, mae 0.00313, time 10.6 sec
Epoch  6, train accuracy 1.00, mae 0.00311, time 10.6 sec
Epoch  7, train accuracy 1.00, mae 0.00310, time 10.7 sec
Epoch  8, train accuracy 1.00, mae 0.00309, time 10.6 sec
Epoch  9, train accuracy 1.00, mae 0.00302, time 10.7 sec
Epoch 10, train accuracy 1.00, mae 0.00288, time 10.7 sec
```

```
Epoch 11, train accuracy 1.00, mae 0.00303, time 10.6 sec
Epoch 12, train accuracy 1.00, mae 0.00296, time 10.6 sec
Epoch 13, train accuracy 1.00, mae 0.00287, time 10.7 sec
Epoch 14, train accuracy 1.00, mae 0.00289, time 10.6 sec
Epoch 15, train accuracy 1.00, mae 0.00288, time 10.6 sec
Epoch 16, train accuracy 1.00, mae 0.00288, time 10.6 sec
Epoch 17, train accuracy 1.00, mae 0.00286, time 10.6 sec
Epoch 18, train accuracy 1.00, mae 0.00271, time 10.7 sec
Epoch 19, train accuracy 1.00, mae 0.00285, time 10.6 sec
Epoch 20, train accuracy 1.00, mae 0.00282, time 10.6 sec
Epoch 21, train accuracy 1.00, mae 0.00268, time 10.6 sec
Epoch 22, train accuracy 1.00, mae 0.00274, time 10.6 sec
Epoch 23, train accuracy 1.00, mae 0.00273, time 10.6 sec
Epoch 24, train accuracy 1.00, mae 0.00268, time 10.6 sec
Epoch 25, train accuracy 1.00, mae 0.00266, time 10.6 sec
Epoch 26, train accuracy 1.00, mae 0.00265, time 10.7 sec
Epoch 27, train accuracy 1.00, mae 0.00264, time 10.6 sec
Epoch 28, train accuracy 1.00, mae 0.00262, time 10.7 sec
Epoch 29, train accuracy 1.00, mae 0.00254, time 10.7 sec
```

9.4.4 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.imdecode(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)
```

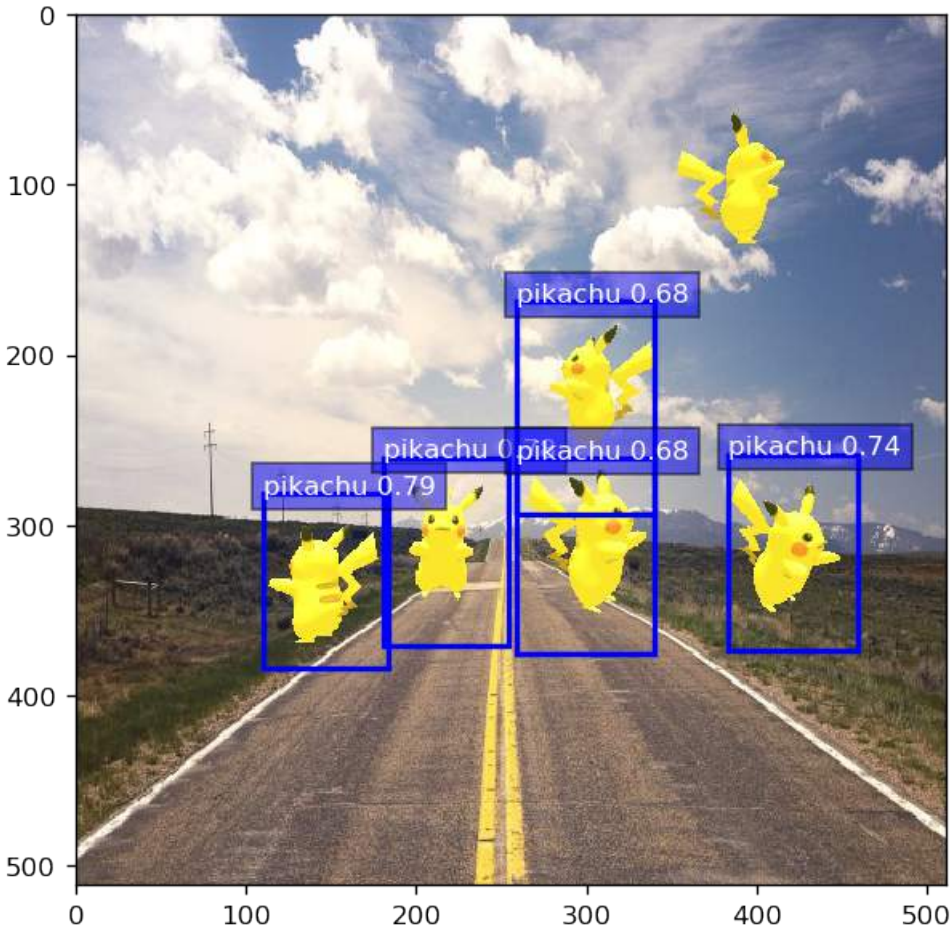
预测函数会输出所有边框，每个边框由 `[class_id, confidence, xmin, ymin, xmax, ymax]` 表示。其中 `class_id=-1` 表示要么这个边框被预测只含有背景，或者被去重掉了。

```
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)
```



9.4.5 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。

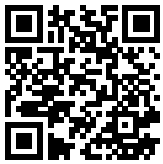
9.4.6 练习

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`

9.4.7 扫码直达讨论区



9.5 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
```

9.5.1 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,
```

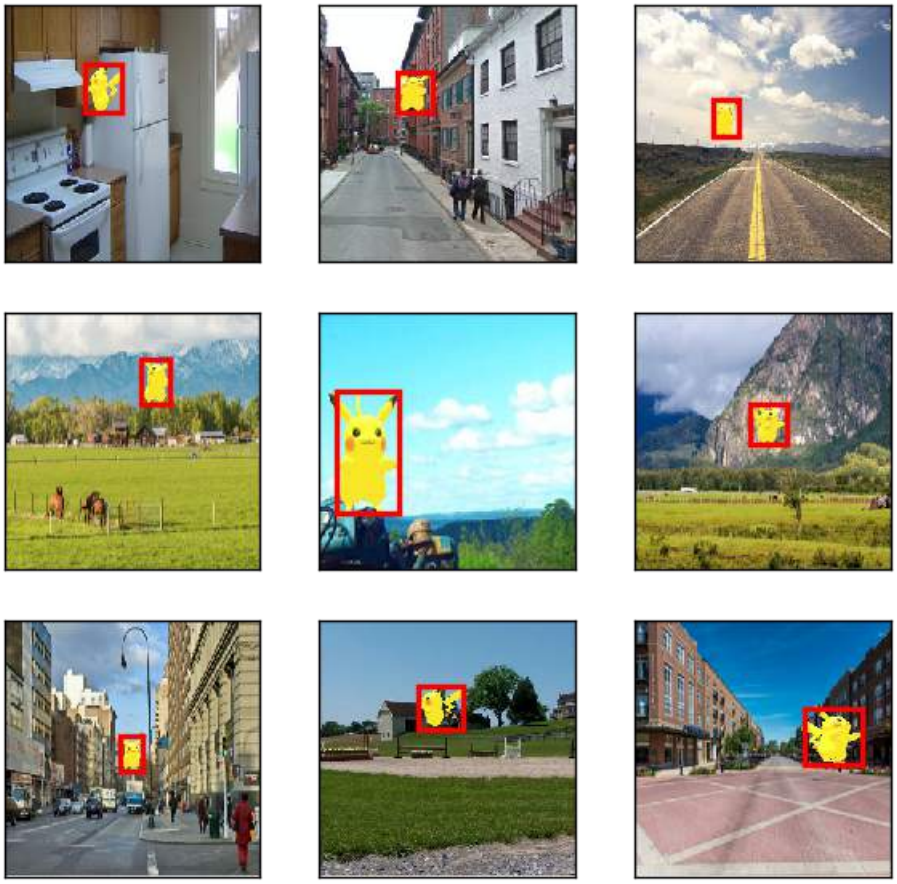
```
            data_shape=(3, data_shape, data_shape),
            path_imgrec=data_dir+'train.rec',
            path_imgidx=data_dir+'train.idx',
            shuffle=True,
            mean=True,
            std=True,
            rand_crop=1,
            min_object_covered=0.95,
            max_attempts=200)
        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
         import matplotlib.pyplot as plt

         def box_to_rect(box, color, linewidth=3):
             """convert an anchor box to a matplotlib rectangle"""
             box = box.asnumpy()
             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).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()
```



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, tbox, 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 0.03681, box_refine_loss
 ↪  0.00250 time 9.1 sec
Epoch  1, train classification_loss 0.00035, objectness_loss 0.01591, box_refine_loss
 ↪  0.00210 time 8.3 sec
Epoch  2, train classification_loss 0.00010, objectness_loss 0.00714, box_refine_loss
 ↪  0.00191 time 8.4 sec
Epoch  3, train classification_loss 0.00008, objectness_loss 0.00462, box_refine_loss
 ↪  0.00188 time 8.3 sec
Epoch  4, train classification_loss 0.00005, objectness_loss 0.00339, box_refine_loss
 ↪  0.00193 time 8.3 sec
Epoch  5, train classification_loss 0.00005, objectness_loss 0.00280, box_refine_loss
 ↪  0.00197 time 8.4 sec
Epoch  6, train classification_loss 0.00004, objectness_loss 0.00243, box_refine_loss
 ↪  0.00197 time 8.3 sec
Epoch  7, train classification_loss 0.00003, objectness_loss 0.00208, box_refine_loss
 ↪  0.00187 time 8.3 sec
Epoch  8, train classification_loss 0.00004, objectness_loss 0.00199, box_refine_loss
 ↪  0.00179 time 8.3 sec
Epoch  9, train classification_loss 0.00005, objectness_loss 0.00200, box_refine_loss
 ↪  0.00171 time 8.3 sec
Epoch 10, train classification_loss 0.00004, objectness_loss 0.00183, box_refine_loss
 ↪  0.00166 time 8.3 sec
Epoch 11, train classification_loss 0.00004, objectness_loss 0.00174, box_refine_loss
 ↪  0.00161 time 8.3 sec
Epoch 12, train classification_loss 0.00003, objectness_loss 0.00157, box_refine_loss
 ↪  0.00152 time 8.2 sec
```

```
Epoch 13, train classification_loss 0.00004, objectness_loss 0.00158, box_refine_loss
 ↪  0.00149 time 8.2 sec
Epoch 14, train classification_loss 0.00003, objectness_loss 0.00152, box_refine_loss
 ↪  0.00141 time 8.3 sec
Epoch 15, train classification_loss 0.00004, objectness_loss 0.00150, box_refine_loss
 ↪  0.00141 time 8.2 sec
Epoch 16, train classification_loss 0.00004, objectness_loss 0.00147, box_refine_loss
 ↪  0.00134 time 8.3 sec
Epoch 17, train classification_loss 0.00003, objectness_loss 0.00138, box_refine_loss
 ↪  0.00132 time 8.5 sec
Epoch 18, train classification_loss 0.00003, objectness_loss 0.00135, box_refine_loss
 ↪  0.00128 time 8.5 sec
Epoch 19, train classification_loss 0.00004, objectness_loss 0.00146, box_refine_loss
 ↪  0.00124 time 8.3 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)))
```

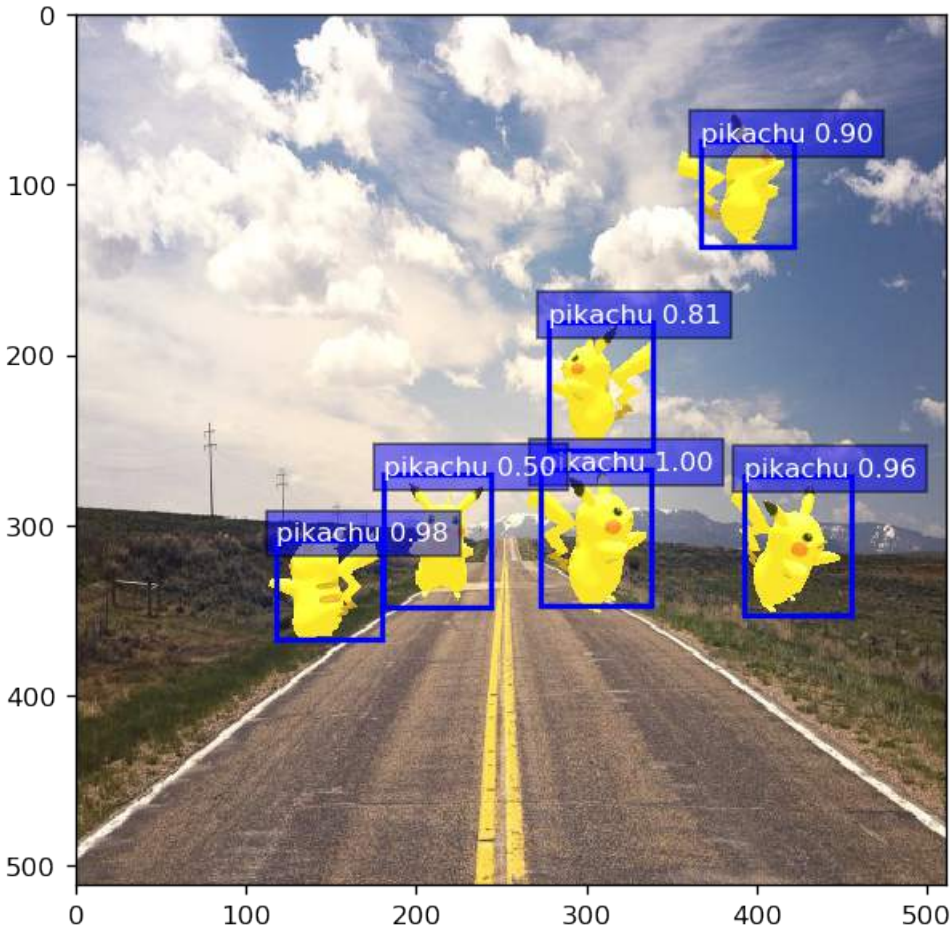
继续读取皮卡丘来做测试

```
In [18]: x, im = process_image('../img/pikachu.jpg')
         out = predict(x.as_in_context(ctx))
         out.shape
         out

Out[18]:
         [[[ 0.          0.99552453  0.53300101  0.52224678  0.66038698  0.67853349]
           [ 0.          0.98359966  0.23022327  0.60371518  0.35035852  0.71767378]
```

```
  [ 0.          0.96311462  0.76637644  0.52913278  0.89032966  0.68917173]
  ...,
  [-1.         -1.         -1.         -1.         -1.         -1.        ]
  [-1.         -1.         -1.         -1.         -1.         -1.        ]
  [-1.         -1.         -1.         -1.         -1.         -1.        ]]]
<NDArray 1x512x6 @gpu(0)>
```

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)
```



9.5.2 小结

- TODO(@mli)

9.5.3 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？

9.5.4 扫码直达讨论区

9.6 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，将每个像素划分的某个类。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。

9.6.1 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)
```

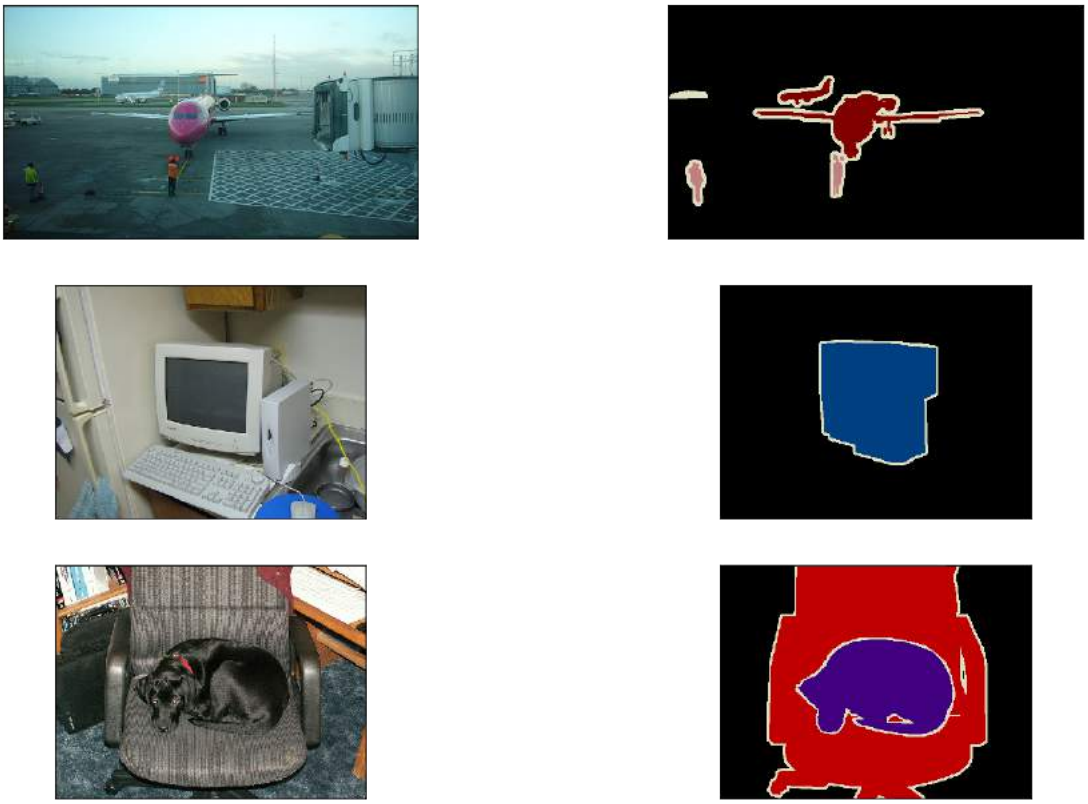
下面定义函数将训练图片和标注按序读进内存。

```
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]]

gb.show_images(imgs, nrows=3, ncols=2, figsize=(12,8))
[im.shape for im in imgs]
```



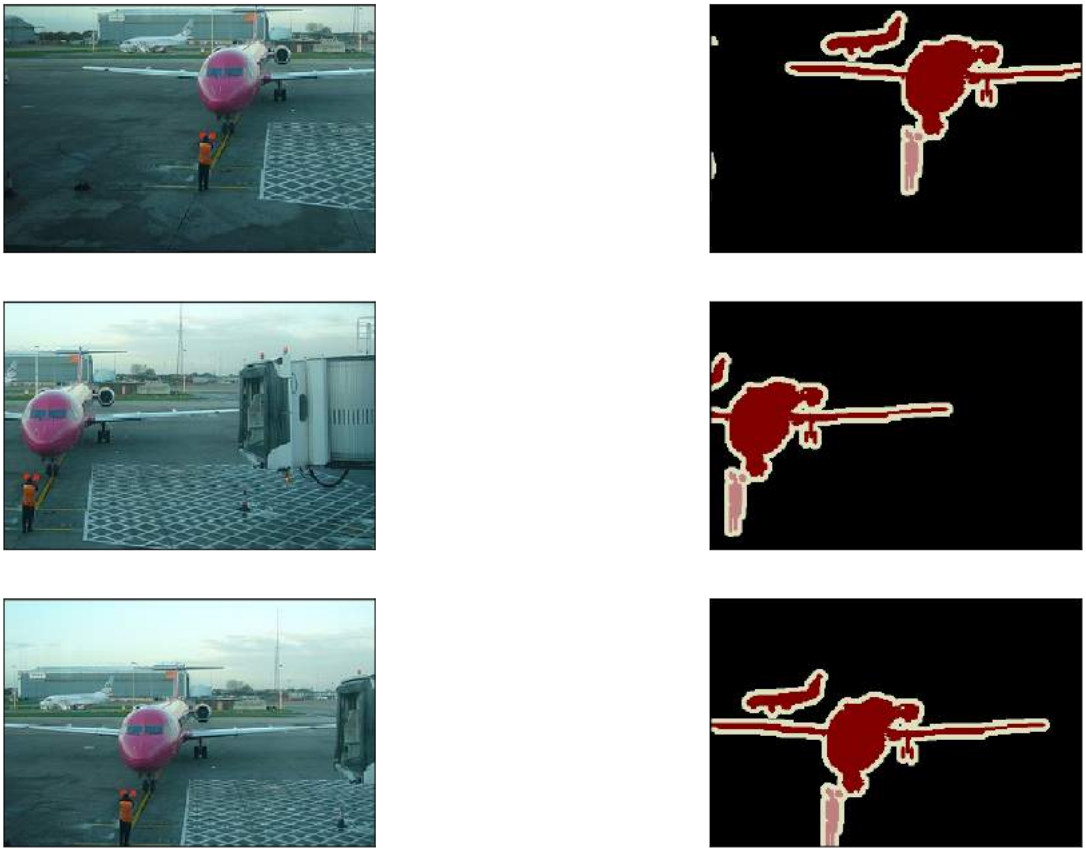
```
Out[3]: [(281, 500, 3),
         (281, 500, 3),
         (375, 500, 3),
         (375, 500, 3),
         (375, 500, 3),
         (375, 500, 3)]
```

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)

        gb.show_images(imgs, nrows=3, ncols=2, figsize=(12,8))
```



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

```
 [ 0.  0.  0.  0.  0.  0.  0.  0.  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 320×480 的大小用来训练，注意到这个比前面我们使用的 224×224 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)
```

9.6.2 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。

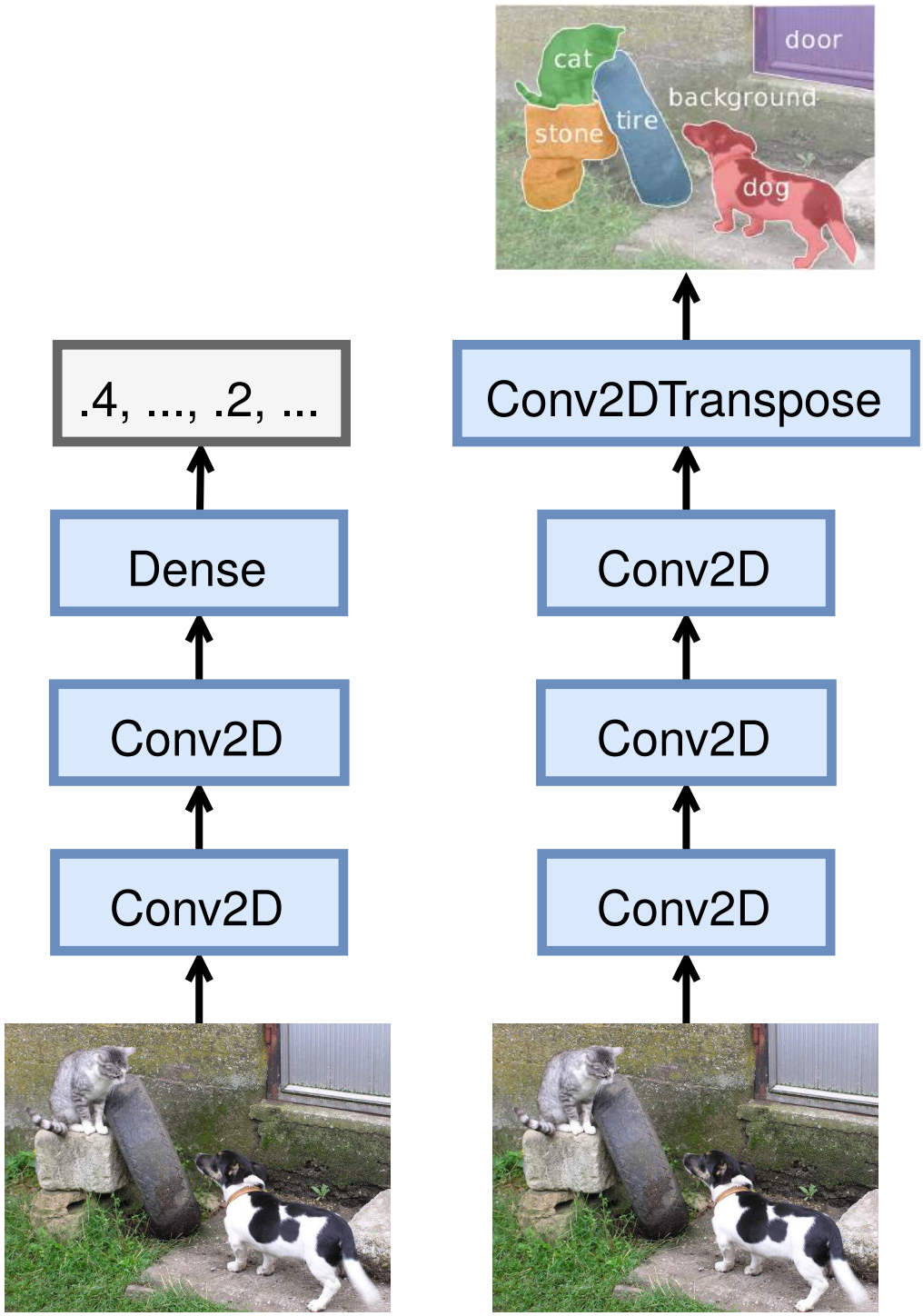
```
In [11]: from mxnet.gluon import nn

         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.Flattern）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 1×1 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 1×1 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，通常权重会初始化称预先训练好的权重


door
cat
background
tire
stone
dog
.4, ..., .2, ...
Conv2DTranspose
Dense
Conv2D
Conv2D
Conv2D
Conv2D

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]: (HybridSequential(
            (0): BatchNorm(axis=1, eps=1e-05, momentum=0.9, fix_gamma=False,
 ↪  use_global_stats=False, in_channels=512)
            (1): Activation(relu)
            (2): GlobalAvgPool2D(size=(1, 1), stride=(1, 1), padding=(0, 0),
 ↪  ceil_mode=True)
            (3): 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 1×1 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 `stides` 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)
             )
```

9.6.3 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，但实际中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`，然后再改写权重。

```
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 Softmax-CrossEntropyLoss 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)
```

```
training on  [gpu(0), gpu(1)]
epoch 1, loss 1.5610, train acc 0.730, test acc 0.809, time 72.5 sec
epoch 2, loss 0.5760, train acc 0.831, test acc 0.826, time 35.0 sec
epoch 3, loss 0.4858, train acc 0.849, test acc 0.841, time 35.1 sec
epoch 4, loss 0.4067, train acc 0.870, test acc 0.849, time 35.1 sec
epoch 5, loss 0.3500, train acc 0.886, test acc 0.849, time 35.0 sec
epoch 6, loss 0.3441, train acc 0.887, test acc 0.844, time 35.1 sec
epoch 7, loss 0.2949, train acc 0.903, test acc 0.849, time 35.1 sec
epoch 8, loss 0.2763, train acc 0.908, test acc 0.853, time 35.1 sec
epoch 9, loss 0.2505, train acc 0.916, test acc 0.857, time 35.1 sec
epoch 10, loss 0.2475, train acc 0.916, test acc 0.854, time 35.1 sec
```

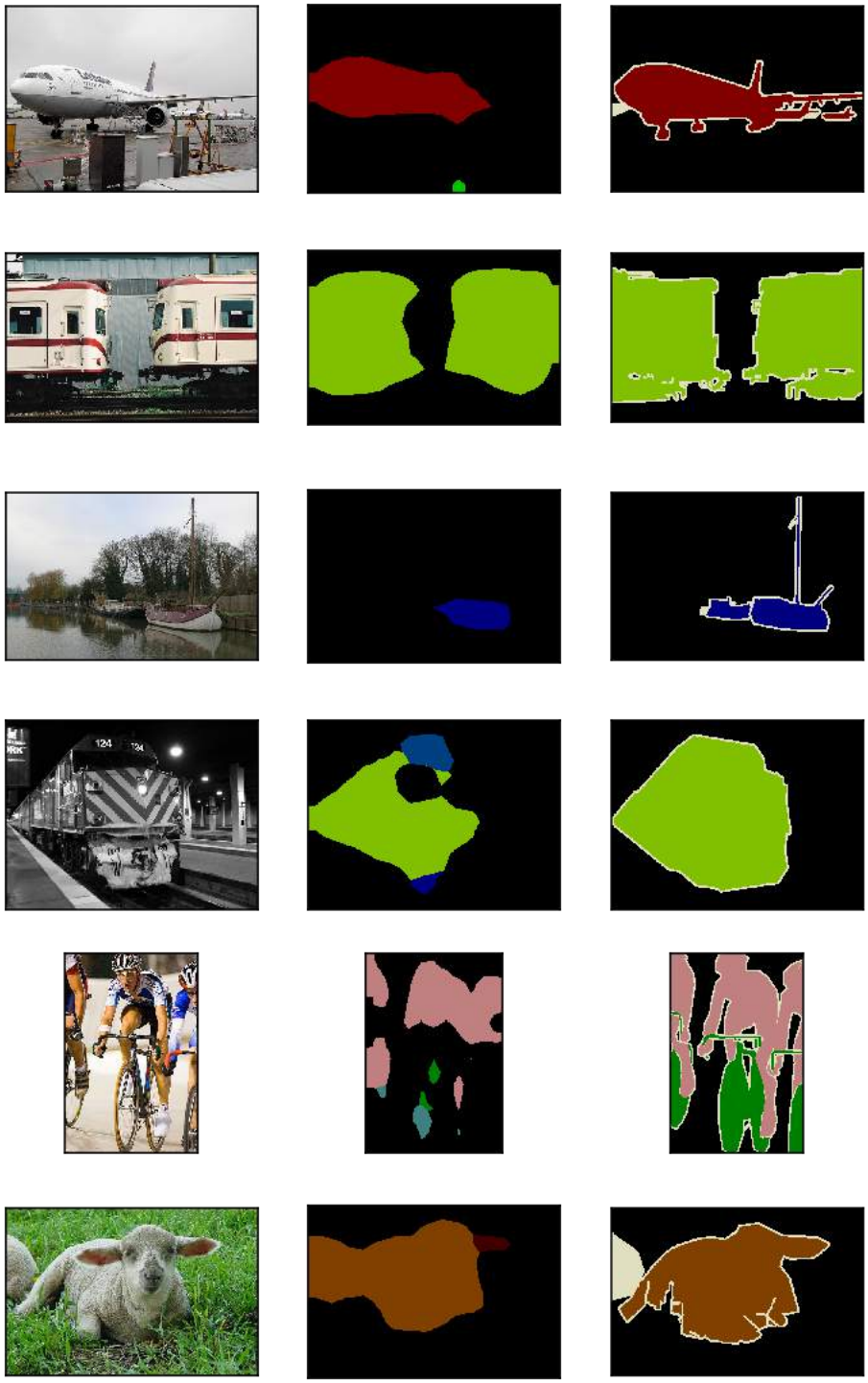
9.6.4 预测

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, nrows=n, ncols=3, figsize=(6,10))
```



9.6.5 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。

9.6.6 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。

9.6.7 扫码直达讨论区

9.7 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。

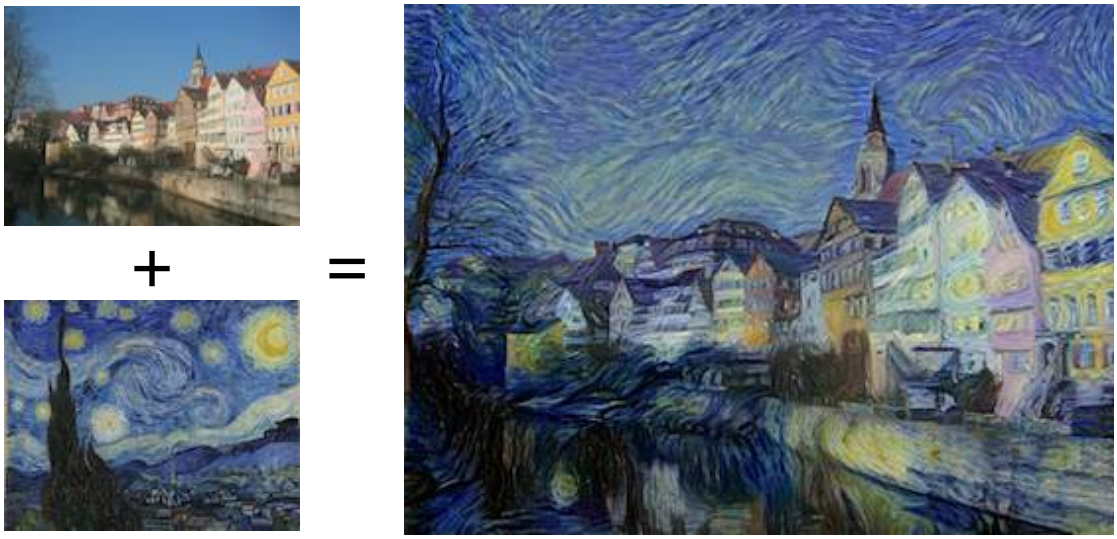


图 9.10: Neural Style

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：

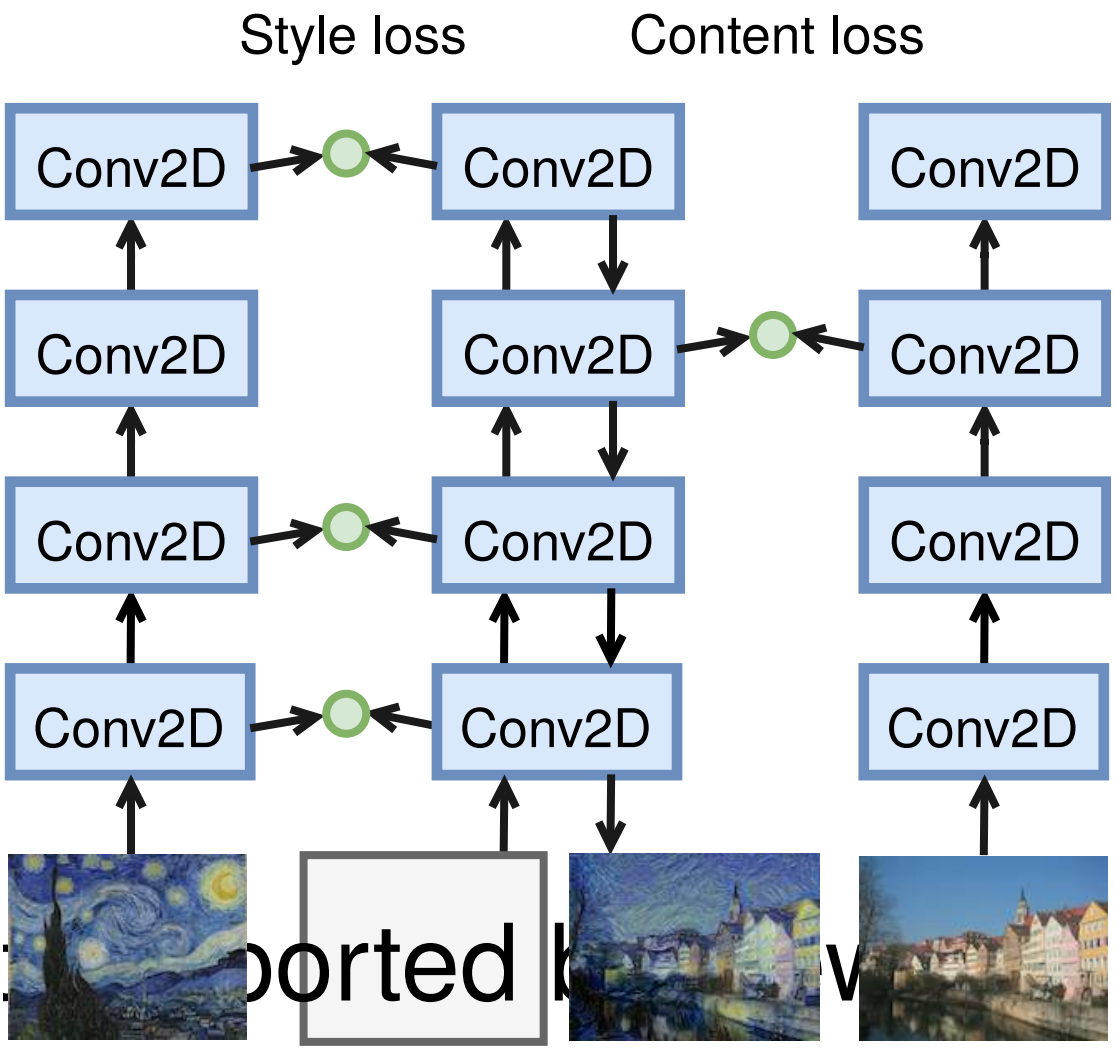


图 9.11: Neural Style Training

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。

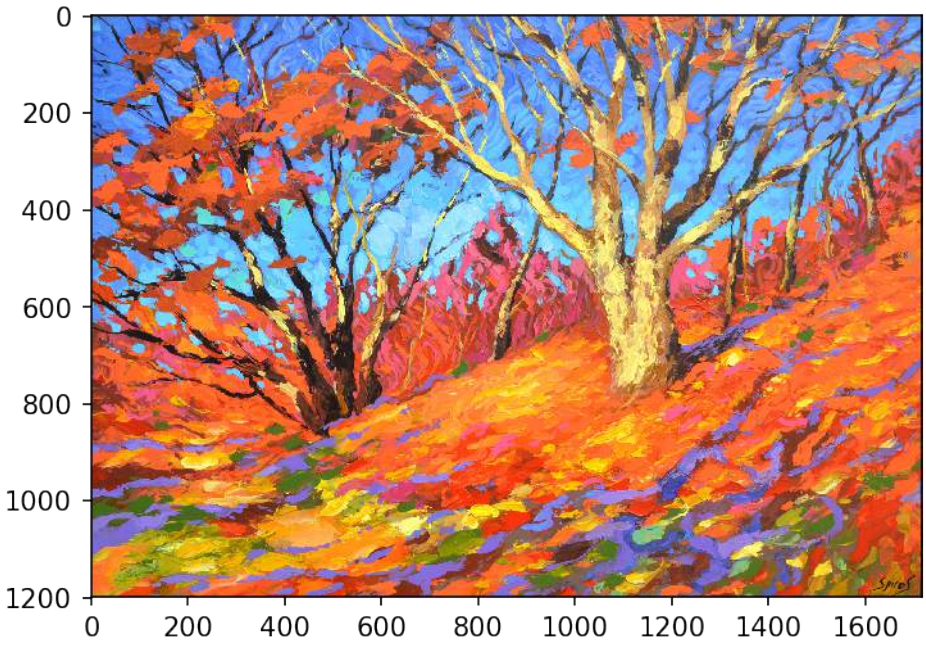
下面我们将实现这个算法来深入理解各个参数，例如样式层和内容层的选取，对实际结果的影响。

9.7.1 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
        import matplotlib.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)
```

9.7.2 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
```

9.7.3 损失函数

内容匹配是一个典型的回归问题，我们将来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)])
```

9.7.4 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(),
```

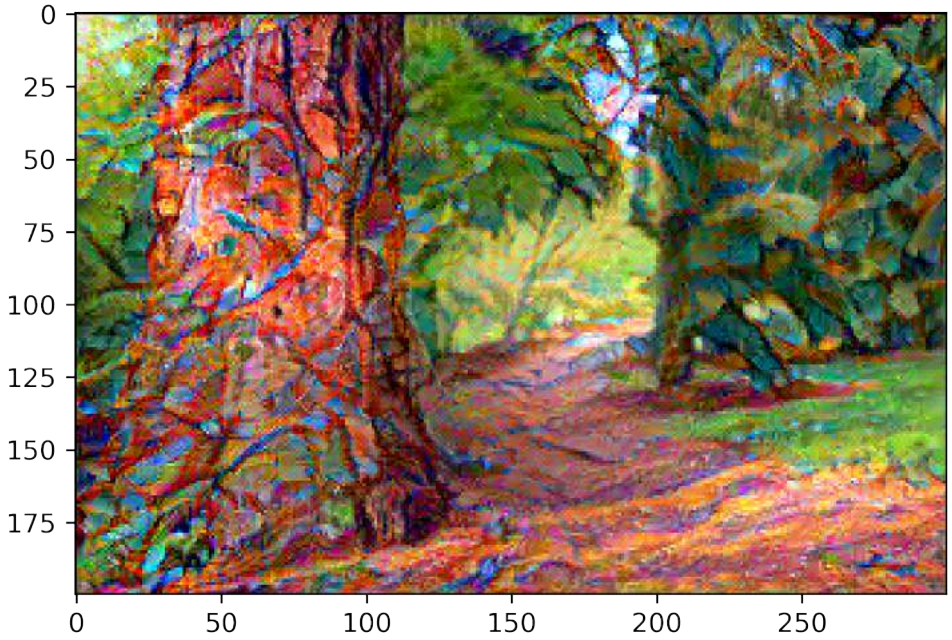
```
                    tv_L.asscalar(), time()-tic))
                tic = time()

            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，首先我们在一个小的 300 x 200 的输入大小上进行训练。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 29.11, style 603.43, TV 4.60, time 3.7 sec
batch  40, content 30.22, style 312.46, TV 4.92, time 1.3 sec
batch  60, content 30.95, style 238.68, TV 5.18, time 1.3 sec
batch  80, content 28.93, style 255.18, TV 5.27, time 1.3 sec
batch 100, content 29.72, style 226.97, TV 5.48, time 1.3 sec
batch 120, content 30.36, style 153.36, TV 5.59, time 1.3 sec
batch 140, content 29.09, style 190.19, TV 5.64, time 1.3 sec
batch 160, content 28.56, style 233.52, TV 5.79, time 1.3 sec
batch 180, content 29.02, style 145.63, TV 5.87, time 1.3 sec
batch 200, content 27.36, style 194.98, TV 5.85, time 1.3 sec
change lr to  0.010000000000000002
batch 220, content 23.86, style 33.09, TV 5.73, time 1.3 sec
```

```
batch 240, content 22.14, style 27.01, TV 5.69, time 1.3 sec
batch 260, content 20.66, style 25.03, TV 5.64, time 1.3 sec
batch 280, content 20.10, style 22.44, TV 5.62, time 1.3 sec
batch 300, content 19.10, style 22.14, TV 5.58, time 1.3 sec
batch 320, content 18.62, style 21.75, TV 5.56, time 1.3 sec
batch 340, content 18.30, style 20.54, TV 5.55, time 1.3 sec
batch 360, content 17.93, style 18.99, TV 5.53, time 1.3 sec
batch 380, content 17.38, style 18.86, TV 5.49, time 1.3 sec
batch 400, content 17.23, style 18.63, TV 5.47, time 1.3 sec
change lr to  0.0010000000000000002
batch 420, content 16.72, style 12.70, TV 5.46, time 1.3 sec
batch 440, content 16.38, style 12.26, TV 5.44, time 1.3 sec
batch 460, content 16.13, style 11.93, TV 5.43, time 1.3 sec
batch 480, content 15.92, style 11.65, TV 5.42, time 1.3 sec
```



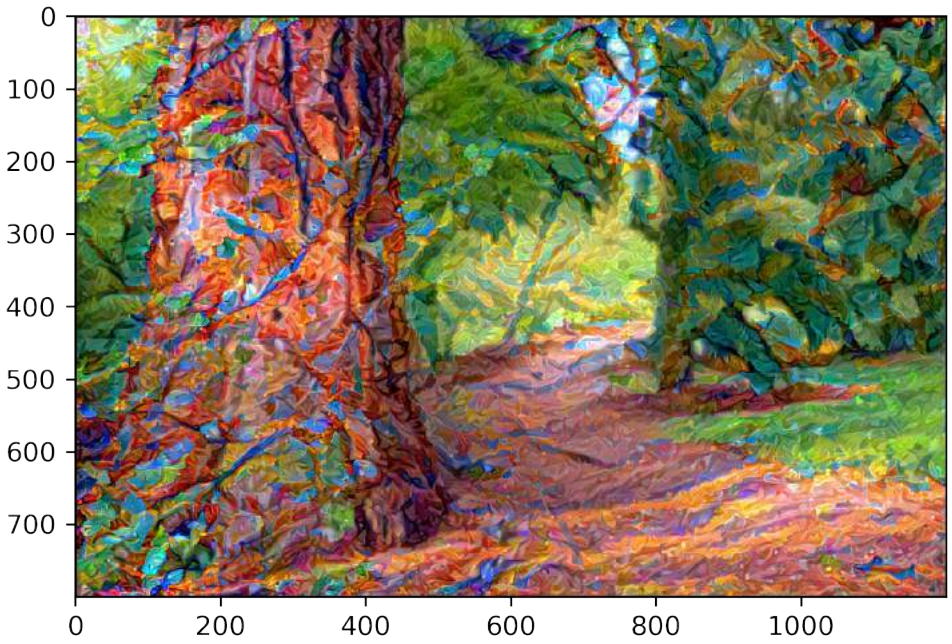
观察损失值的变化。因为我们使用了内容图片作为初始化，所以一开始看到样式损失比较大。但随着迭代的进行，它减少的非常迅速，尤其是在每次调整学习率后。噪音在训练中有略微的增加，但在后期还是控制在合理的范围。

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 x 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 47.79, style 757.34, TV 3.29, time 25.8 sec
batch  40, content 45.34, style 698.64, TV 3.59, time 18.3 sec
batch  60, content 45.46, style 716.08, TV 3.82, time 18.3 sec
batch  80, content 46.33, style 768.03, TV 4.01, time 18.3 sec
batch 100, content 46.51, style 792.28, TV 4.18, time 18.4 sec
change lr to  0.0100000000000000002
batch 120, content 29.30, style 62.32, TV 3.68, time 18.3 sec
batch 140, content 25.42, style 30.53, TV 3.47, time 18.3 sec
batch 160, content 23.50, style 20.66, TV 3.36, time 18.4 sec
batch 180, content 21.74, style 34.97, TV 3.30, time 18.3 sec
batch 200, content 20.66, style 41.28, TV 3.25, time 18.3 sec
change lr to  0.0010000000000000002
batch 220, content 19.88, style 12.25, TV 3.22, time 18.3 sec
batch 240, content 19.17, style 11.33, TV 3.19, time 18.3 sec
batch 260, content 18.65, style 10.71, TV 3.17, time 18.2 sec
batch 280, content 18.22, style 10.23, TV 3.16, time 18.3 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())
```

9.7.5 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。

9.7.6 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

9.7.7 扫码直达讨论区

9.8 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。

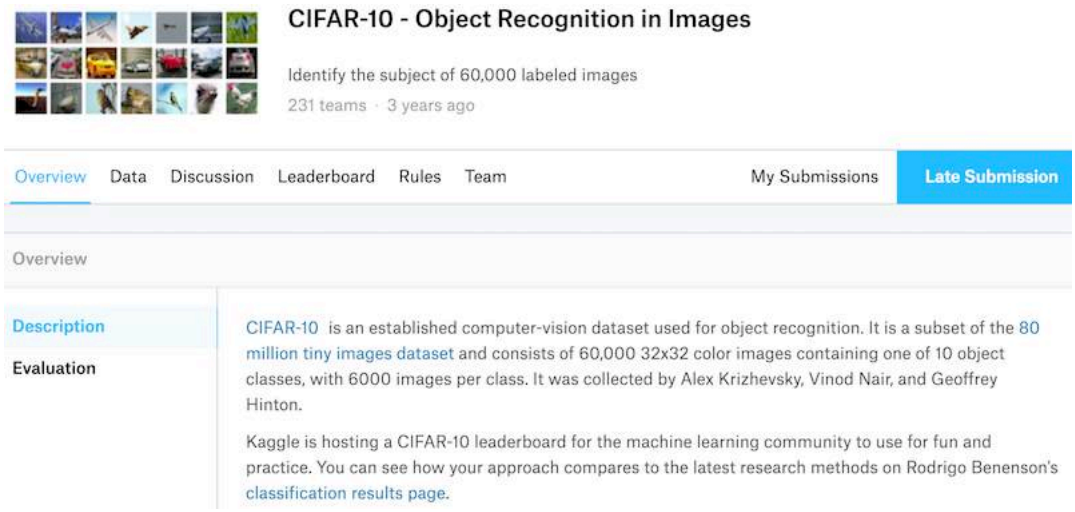
我们在本章中选择了 Kaggle 中著名的CIFAR-10 原始图像分类问题。我们以该问题为例，为大家提供使用 `Gluon` 对原始图像文件进行分类的示例代码。

计算机视觉一直是深度学习的主战场，请

> Get your hands dirty。

9.8.1 Kaggle 中的 CIFAR-10 原始图像分类问题

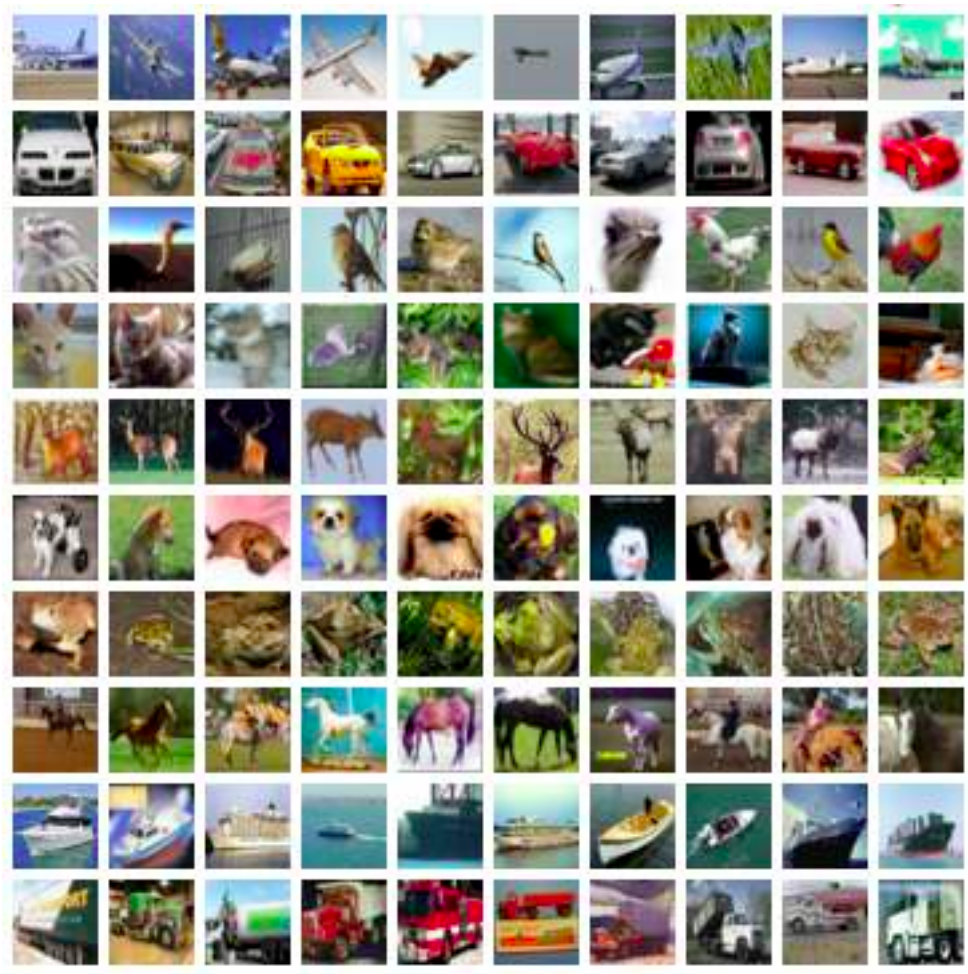
Kaggle是一个著名的供机器学习爱好者交流的平台。为了便于提交结果，请大家注册Kaggle账号。然后请大家先点击CIFAR-10 原始图像分类问题了解有关本次比赛的信息。



9.8.2 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)
```

9.8.3 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()
```

9.8.4 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
```

9.8.5 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: 3.691536, Train acc 0.033333, Valid acc 0.100000, Time 00:00:01, lr 0.1
```

9.8.6 对测试集分类

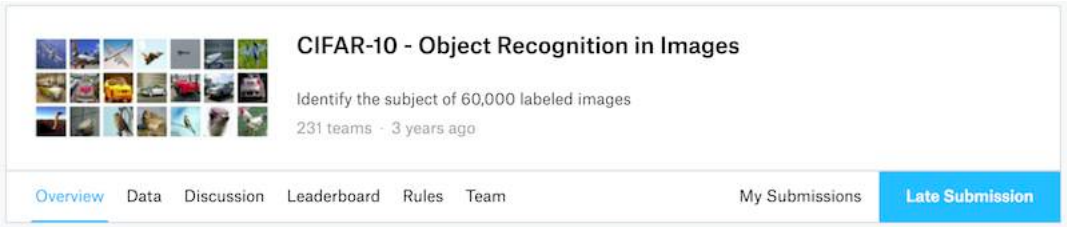
当得到一组满意的模型设计和参数后，我们使用全部训练数据集（含验证集）重新训练模型，并对测试集分类。

```
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})
```

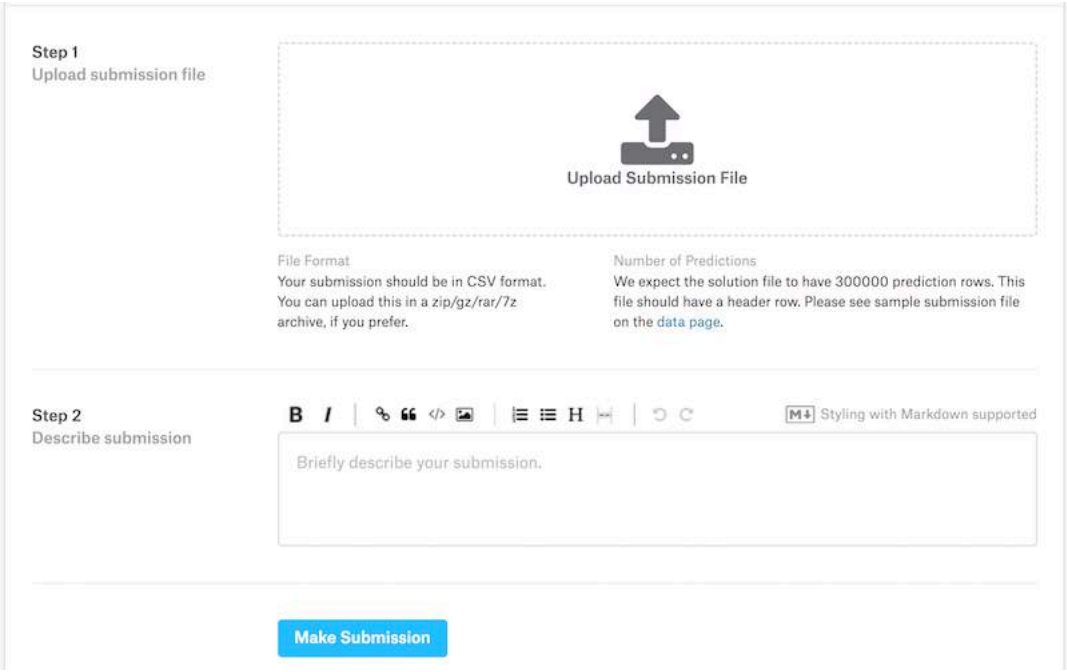
```
    df['label'] = df['label'].apply(lambda x: train_valid_ds.synsets[x])
    df.to_csv('submission.csv', index=False)
```

```
Epoch 0. Loss: 3.943431, Train acc 0.050000, Time 00:00:01, lr 0.1
```

上述代码执行完会生成一个 `submission.csv` 的文件用于在 Kaggle 上提交。这是 Kaggle 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，打开CIFAR-10 原始图像分类问题，并点击下方右侧 `Late Submission` 按钮。



请点击下方 `Upload Submission File` 选择需要提交的预测结果。然后点击下方的 `Make Submission` 按钮就可以查看结果啦！



9.8.7 作业（汇报作业和查看其他小伙伴作业）：

- 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？
- 你还有什么其他办法可以继续改进模型和参数？小伙伴们都期待你的分享。

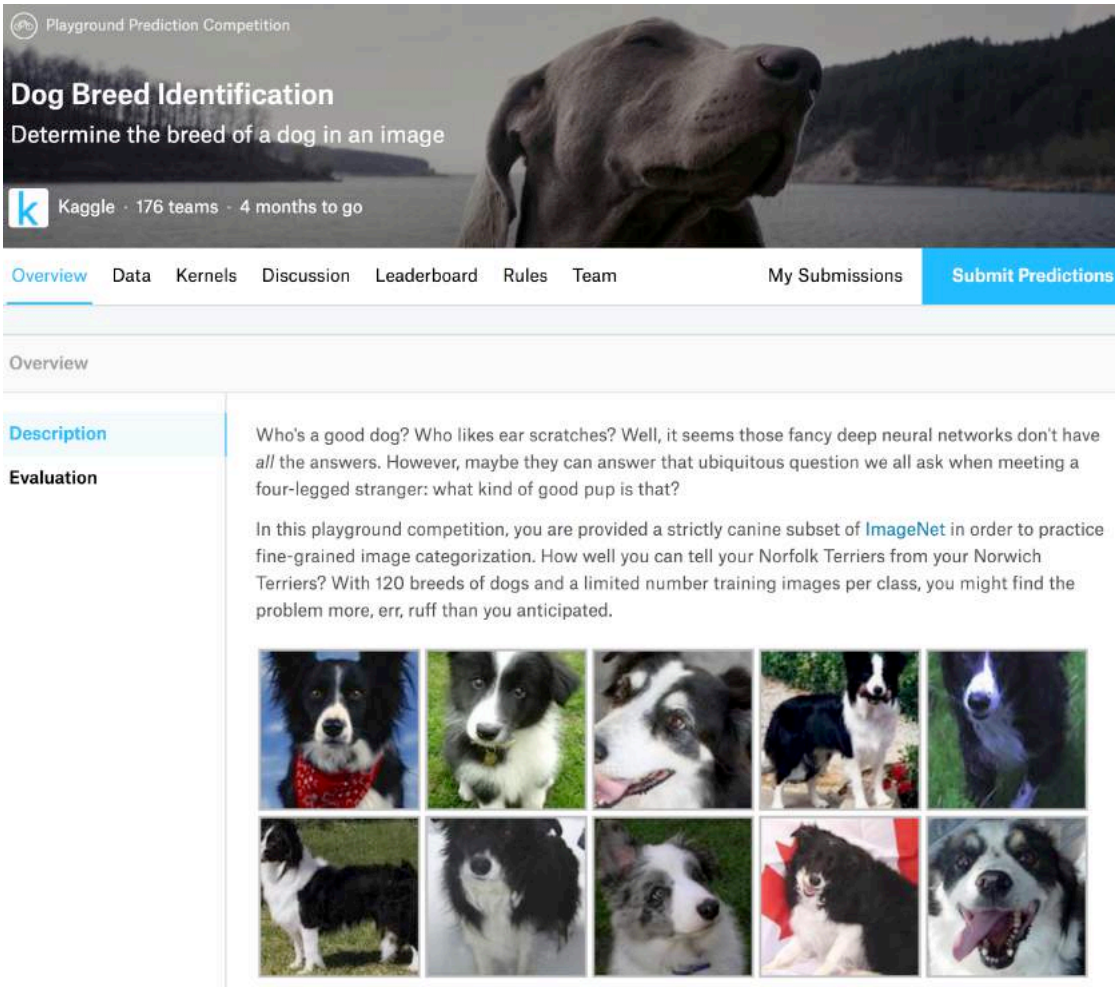
9.8.8 扫码直达讨论区

9.9 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。

9.9.1 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120 种狗类识别问题了解有关本次比赛的信息。



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 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)
```

9.9.3 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。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()
```

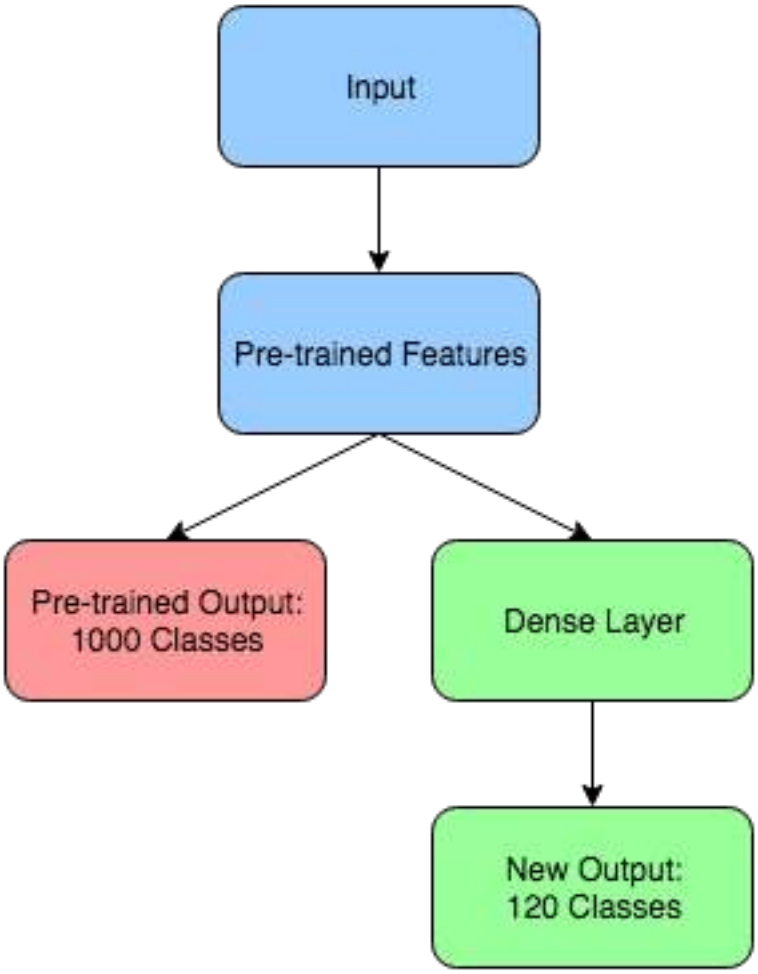
9.9.4 设计模型

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
```

9.9.5 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: 5.447030, Valid loss 4.756134, Time 00:00:01, lr 0.01
```

9.9.6 对测试集分类

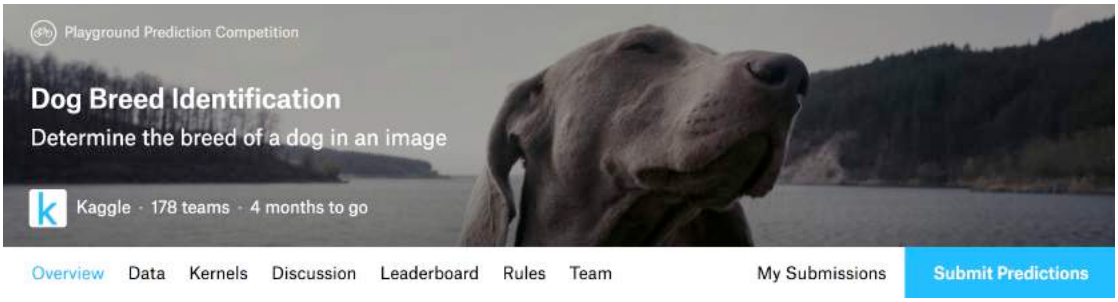
当得到一组满意的模型设计和参数后，我们使用全部训练数据集（含验证集）重新训练模型，并对测试集分类。注意，我们要用刚训练好的新输出层做预测。

```
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):
```

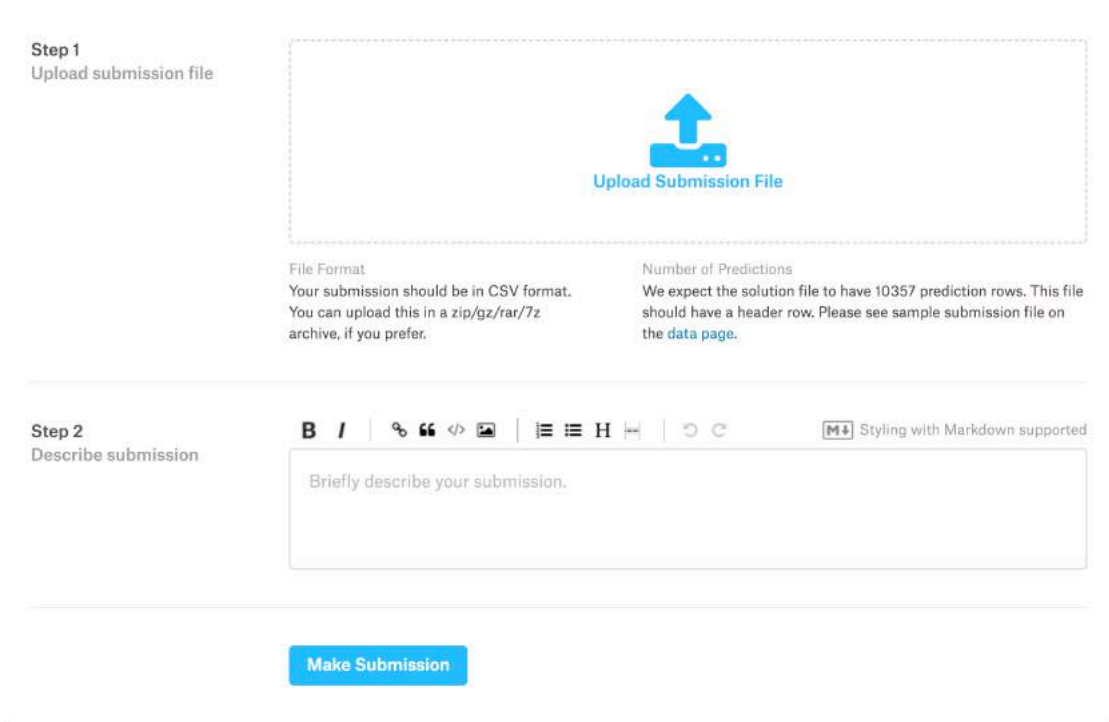
```
            f.write(i.split('.')[0] + ',' + ','.join(
                [str(num) for num in output]) + '\n')
```

```
Epoch 0. Train loss: 5.078403, Time 00:00:02, lr 0.01
```

上述代码执行完会生成一个 `submission.csv` 的文件用于在 Kaggle 上提交。这是 Kaggle 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，打开120 种狗类识别问题，并点击下方右侧 `Submit Predictions` 按钮。



请点击下方 `Upload Submission File` 选择需要提交的预测结果。然后点击下方的 `Make Submission` 按钮就可以查看结果啦！



温馨提醒，目前 **Kaggle 仅限每个账号一天以内 5 次提交结果的机会**。所以提交结果前务必三思。

9.9.7 作业（汇报作业和查看其他小伙伴作业）：

- 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?
- 你还有什么其他办法可以继续改进模型和参数？小伙伴们都期待你的分享。

9.9.8 扫码直达讨论区

10

自然语言处理

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。在机器学习中，如何使用向量表示词?

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。

10.1.1 为何不采用 one-hot 向量

我们在循环神经网络中介绍过 one-hot 向量来表示词。假设词典中不同词的数量为 N，每个词可以和从 0 到 $N-1$ 的连续整数一一对应。假设一个词的相应整数表示为 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，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。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），以及两种高效训练的方法：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

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。例如，给定文本序列”the”，“man”，“hit”，“his”，和”son”，跳字模型所关心的是，给定”hit”，生成它邻近词“the”，“man”，“his”，和”son”的概率。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。

假设词典大小为 $|\mathcal{V}|$，我们将词典中的每个词与从 0 到 $|\mathcal{V}| - 1$ 的整数一一对应：词典索引集 $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$。一个词在该词典中所对应的整数称为词的索引。给定一个长度为 T 的文本序列中，t 时刻的词为 $w^{(t)}$。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背景词的概率：

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)})$$

上式的最大似然估计与最小化以下损失函数等价

$$-\frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m, j \neq 0} \log \mathbb{P}(w^{(t+j)} \mid w^{(t)})$$

我们可以用 $\mathbf{v}$ 和 $\mathbf{u}$ 分别代表中心词和背景词的向量。换言之，对于词典中一个索引为 i 的词，它在作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中心词生成背景词的概率。假设中心词生成各个背景词的概率是相互独立的。给定中心词 w_c

在词典中索引为 c，背景词 w_o 在词典中索引为 o，损失函数中的中心词生成背景词的概率可以使用 softmax 函数定义为

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}$$

当序列长度 T 较大时，我们通常随机采样一个较小的子序列来计算损失函数并使用随机梯度下降优化该损失函数。通过微分，我们可以计算出上式生成概率的对数关于中心词向量 $\mathbf{v}_c$ 的梯度为：

$$\frac{\partial \log \mathbb{P}(w_o \mid w_c)}{\partial \mathbf{v}_c} = \mathbf{u}_o - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \mathbf{u}_j$$

而上式与下式等价：

$$\frac{\partial \log \mathbb{P}(w_o \mid w_c)}{\partial \mathbf{v}_c} = \mathbf{u}_o - \sum_{j \in \mathcal{V}} \mathbb{P}(w_j \mid w_c) \mathbf{u}_j$$

通过上面计算得到梯度后，我们可以使用随机梯度下降来不断迭代模型参数 $\mathbf{v}_c$。其他模型参数 $\mathbf{u}_o$ 的迭代方式同理可得。最终，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 $\mathbf{u}_i$。

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，连续词袋模型中用一个中心词在文本序列周围的词来预测该中心词。例如，给定文本序列”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。

假设词典大小为 $|\mathcal{V}|$，我们将词典中的每个词与从 0 到 $|\mathcal{V}| - 1$ 的整数一一对应：词典索引集 $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$。一个词在该词典中所对应的整数称为词的索引。给定一个长度为 T 的文本序列中，t 时刻的词为 $w^{(t)}$。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：

$$\prod_{t=1}^{T} \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)})$$

上式的最大似然估计与最小化以下损失函数等价

$$-\sum_{t=1}^{T} \log \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)})$$

我们可以用 $\mathbf{v}$ 和 $\mathbf{u}$ 分别代表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，对于词典中一个索引为 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中心词生成背景词的概率。给定中心词 w_c 在词典中索引为 c，背景词 $w_{o_1}, \ldots, w_{o_{2m}}$ 在词典中索引为 $o_1, \ldots, o_{2m}$，损失函数中的背景词生成中心词的概率可以使用 softmax 函数定义为

$$\mathbb{P}(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\exp[\mathbf{u}_c^\top (\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})/(2m)]}{\sum_{i \in \mathcal{V}} \exp[\mathbf{u}_i^\top (\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})/(2m)]}$$

当序列长度 T 较大时，我们通常随机采样一个较小的子序列来计算损失函数并使用随机梯度下降优化该损失函数。通过微分，我们可以计算出上式生成概率的对数关于任一背景词向量 $\mathbf{v}_{o_i}(i = 1, \ldots, 2m)$ 的梯度为：

$$\frac{\partial \log \mathbb{P}(w_c \mid w_{o_1}, \ldots, w_{o_{2m}})}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m}\left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \mathbf{u}_j\right)$$

而上式与下式等价：

$$\frac{\partial \log \mathbb{P}(w_c \mid w_{o_1}, \ldots, w_{o_{2m}})}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m}\left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} \mathbb{P}(w_j \mid w_c) \mathbf{u}_j\right)$$

通过上面计算得到梯度后，我们可以使用随机梯度下降来不断迭代各个模型参数 $\mathbf{v}_{o_i}(i = 1, \ldots, 2m)$。其他模型参数 $\mathbf{u}_c$ 的迭代方式同理可得。最终，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 $\mathbf{u}_i$。

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。显然，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。所以，使用上述训练方法在实践中是有难度的。

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。

词典 $\mathcal{V}$ 大小之所以会在目标函数中出现，是因为中心词 w_c 生成背景词 w_o 的概率 $\mathbb{P}(w_o \mid w_c)$ 使用了 softmax，而 softmax 正是考虑了背景词可能是词典中的任一词，并体现在 softmax 的分母上。

我们不妨换个角度，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 同时出现在该训练数据窗口
- 中心词 w_c 和第 1 个噪声词 w_1 不同时出现在该训练数据窗口（噪声词 w_1 按噪声词分布 $\mathbb{P}(w)$ 随机生成，假设一定和 w_c 不同时出现在该训练数据窗口）
- …
- 中心词 w_c 和第 K 个噪声词 w_K 不同时出现在该训练数据窗口（噪声词 w_K 按噪声词分布 $\mathbb{P}(w)$ 随机生成，假设一定和 w_c 不同时出现在该训练数据窗口）

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：

$$\mathbb{P}(D=1 \mid w_o, w_c) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c)$$

那么，中心词 w_c 生成背景词 w_o 的对数概率可以近似为

$$\log\mathbb{P}(w_o \mid w_c) = \log[\mathbb{P}(D=1 \mid w_o, w_c) \prod_{k=1, w_k \sim \mathbb{P}(w)}^{K} \mathbb{P}(D=0 \mid w_k, w_c)]$$

假设噪声词 w_k 在词典中的索引为 i_k，上式可改写为

$$\log\mathbb{P}(w_o \mid w_c) = \log\frac{1}{1+\exp(-\mathbf{u}_o^\top \mathbf{v}_c)} + \sum_{k=1, w_k \sim \mathbb{P}(w)}^{K} \log[1 - \frac{1}{1+\exp(-\mathbf{u}_{i_k}^\top \mathbf{v}_c)}]$$

因此，有关中心词 w_c 生成背景词 w_o 的损失函数是

$$-\log\mathbb{P}(w_o \mid w_c) = -\log\frac{1}{1+\exp(-\mathbf{u}_o^\top \mathbf{v}_c)} - \sum_{k=1, w_k \sim \mathbb{P}(w)}^{K} \log\frac{1}{1+\exp(\mathbf{u}_{i_k}^\top \mathbf{v}_c)}$$

当我们把 K 取较小值时，每次随机梯度下降的梯度计算开销将由 $\mathcal{O}(|\mathcal{V}|)$ 降为 $\mathcal{O}(K)$。

我们也可以对连续词袋模型进行负采样。有关背景词 $w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}$ 生成中心词 w_c 的损失函数

$$-\log\mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)})$$

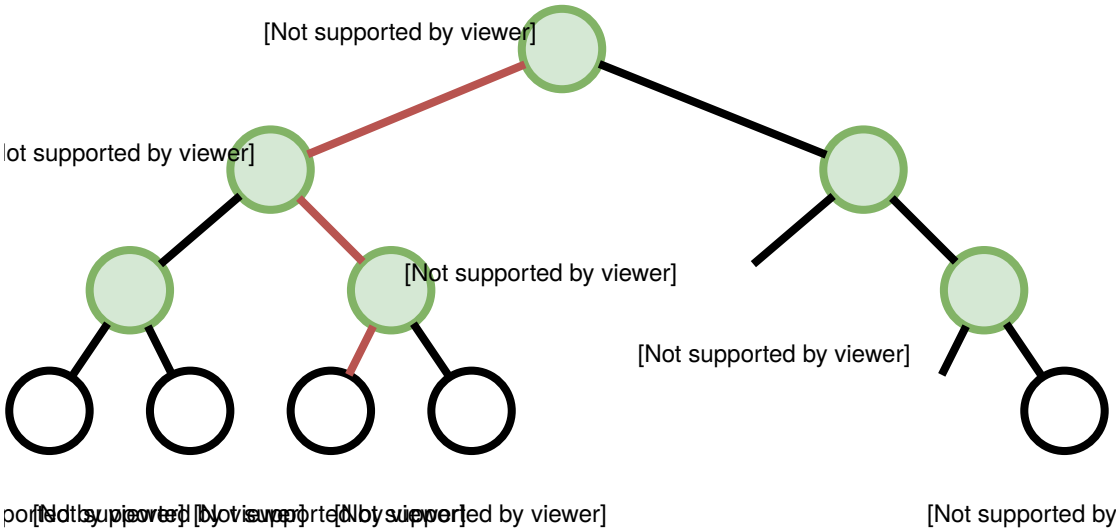
在负采样中可以近似为

$$-\log\frac{1}{1+\exp[-\mathbf{u}_c^\top(\mathbf{v}_{o_1}+\ldots+\mathbf{v}_{o_{2m}})/(2m)]}-\sum_{k=1,w_k\sim\mathbb{P}(w)}^{K}\log\frac{1}{1+\exp[(\mathbf{u}_{i_k}^\top(\mathbf{v}_{o_1}+\ldots+\mathbf{v}_{o_{2m}})/(2m)]}$$

同样地，当我们把 K 取较小值时，每次随机梯度下降的梯度计算开销将由 $\mathcal{O}(|\mathcal{V}|)$ 降为 $\mathcal{O}(K)$。

10.1.5 层序 softmax

层序 softmax 利用了二叉树。树的每个叶子节点代表着词典 $\mathcal{V}$ 中的每个词。每个词 w_i 相应的词向量为 $\mathbf{v}_i$。我们以下图为例，来描述层序 softmax 的工作机制。



假设 $L(w)$ 为从二叉树的根到代表词 w 的叶子节点的路径上的节点数，并设 $n(w,i)$ 为该路径上第 i 个节点，该节点的向量为 $\mathbf{u}_{n(w,i)}$。以上图为例，$L(w_3)=4$。那么，跳字模型和连续词袋模型所需要计算的任意词 w_i 生成词 w 的概率为：

$$\mathbb{P}(w\mid w_i)=\prod_{j=1}^{L(w)-1}\sigma([n(w,j+1)=\text{leftChild}(n(w,j))]\cdot\mathbf{u}_{n(w,j)}^\top\mathbf{v}_i)$$

其中 $\sigma(x)=1/(1+\exp(-x))$，如果 x 为真，$[x]=1$；反之 $[x]=-1$。

由于 $\sigma(x)+\sigma(-x)=1$，w_i 生成词典中任何词的概率之和为 1：

$$\sum_{w=1}^{\mathcal{V}} \mathbb{P}(w \mid w_i) = 1$$

让我们计算 w_i 生成 w_3 的概率，由于在二叉树中由根到 w_3 的路径上需要向左、向右、再向左地遍历，我们得到

$$\mathbb{P}(w_3 \mid w_i) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_i)) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_i)) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_i))$$

我们可以使用随机梯度下降在跳字模型和连续词袋模型中不断迭代计算字典中所有词向量 $\mathbf{v}$ 和非叶子节点的向量 $\mathbf{u}$。每次迭代的计算开销由 $\mathcal{O}(|\mathcal{V}|)$ 降为二叉树的高度 $\mathcal{O}(\log|\mathcal{V}|)$。

10.1.6 小结

word2vec 工具中的跳字模型和连续词袋模型可以使用两种负采样和层序 softmax 减小训练开销。

10.1.7 练习

- 噪声词 $\mathbb{P}(w)$ 在实际中被建议设为 w 的单字概率的 3/4 次方。为什么？（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？（可参考word2vec 论文第 2.3 节）
- 如何训练包括例如”new york”在内的词组向量？（可参考word2vec 论文第 4 节）。

10.1.8 扫码直达讨论区

10.2 词向量：GloVe 和 fastText

本节介绍两种更新一点的词向量。它们分别是 2014 年由 Stanford 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 $\boldsymbol{X}$ 为共现词频矩阵，其中元素 x_{ij} 为词 j 出现在词 i 的环境（context）的次数。这里的“环境”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 的环境一次。令 $x_i = \sum_k x_{ik}$ 为任意词出现在词 i 的环境的次数。那么，

$$P_{ij} = \mathbb{P}(j \mid i) = \frac{x_{ij}}{x_i}$$

为词 j 出现在词 i 的环境的概率。这一概率也称词 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：

- $\mathbb{P}(k \mid \text{ice})$：0.00019（$k$= solid），0.000066（$k$= gas），0.003（$k$= water），0.000017（$k$= fashion）
- $\mathbb{P}(k \mid \text{steam})$：0.000022（$k$= solid），0.00078（$k$= gas），0.0022（$k$= water），0.000018（$k$= fashion）
- $\mathbb{P}(k \mid \text{ice})/\mathbb{P}(k \mid \text{steam})$：8.9（$k$= solid），0.085（$k$= gas），1.36（$k$= water），0.96（$k$= fashion）

我们通过上表可以观察到以下现象：

- 对于与 ice 相关而与 steam 不相关的词 k，例如 k =solid，我们期望共现概率比值 P_{ik}/P_{jk} 较大，例如上面最后一栏的 8.9。
- 对于与 ice 不相关而与 steam 相关的词 k，例如 k =gas，我们期望共现概率比值 P_{ik}/P_{jk} 较小，例如上面最后一栏的 0.085。
- 对于与 ice 和 steam 都相关的词 k，例如 k =water，我们期望共现概率比值 P_{ik}/P_{jk} 接近 1，例如上面最后一栏的 1.36。
- 对于与 ice 和 steam 都不相关的词 k，例如 k =fashion，我们期望共现概率比值 P_{ik}/P_{jk} 接近 1，例如上面最后一栏的 0.96。

由此可见，共现概率比值能比较直观地表达词之间的关系。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 $P_{ij} = \mathbb{P}(j \mid i)$，我们称词 i 和词 j 分别为中心词和背景词。我们使用 $\boldsymbol{v}$ 和 $\tilde{\boldsymbol{v}}$ 分别表示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：

$$f(\boldsymbol{v}_i, \boldsymbol{v}_j, \tilde{\boldsymbol{v}}_k) = \frac{P_{ik}}{P_{jk}}$$

需要注意的是，函数 f 可能的设计并不唯一。首先，我们用向量之差来表达共现概率的比值，并将上式改写成

$$f(\boldsymbol{v}_i - \boldsymbol{v}_j, \tilde{\boldsymbol{v}}_k) = \frac{P_{ik}}{P_{jk}}$$

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。我们可以得到

$$f((\boldsymbol{v}_i - \boldsymbol{v}_j)^\top \tilde{\boldsymbol{v}}_k) = \frac{P_{ik}}{P_{jk}}$$

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，$\boldsymbol{v}_i = \tilde{\boldsymbol{v}}_i$
- 词与词之间共现次数矩阵 $\boldsymbol{X}$ 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

$$f((\boldsymbol{v}_i - \boldsymbol{v}_j)^\top \tilde{\boldsymbol{v}}_k) = \frac{f(\boldsymbol{v}_i^\top \tilde{\boldsymbol{v}}_k)}{f(\boldsymbol{v}_j^\top \tilde{\boldsymbol{v}}_k)}$$

并得到 $f(x) = \exp(x)$。以上两式的右边联立，

$$\exp(\boldsymbol{v}_i^\top \tilde{\boldsymbol{v}}_k) = P_{ik} = \frac{x_{ik}}{x_i}$$

由上式可得

$$\boldsymbol{v}_i^\top \tilde{\boldsymbol{v}}_k = \log(x_{ik}) - \log(x_i)$$

另一方面，我们可以把上式中 $\log(x_i)$ 替换成两个偏移项之和 $b_i + b_k$，得到

$$\boldsymbol{v}_i^\top \tilde{\boldsymbol{v}}_k = \log(x_{ik}) - b_i - b_k$$

将索引 i 和 k 互换，我们可验证对称性的两个性质可以同时被上式满足。

因此，对于任意一对词 i 和 j，用它们词向量表达共现概率比值最终可以被简化为表达它们共现词频的对数：

$$\boldsymbol{v}_i^\top \tilde{\boldsymbol{v}}_j + b_i + b_j = \log(x_{ij})$$

损失函数

上式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移项，我们希望上式中的左边与右边越接近越好。给定词典大小 V 和权重函数 $f(x_{ij})$，我们定义损失函数为

$$\sum_{i,j=1}^{V} f(x_{ij})(\boldsymbol{v}_i^\top \tilde{\boldsymbol{v}}_j + b_i + b_j - \log(x_{ij}))^2$$

对于权重函数 $f(x)$，一个建议的选择是，当 $x < c$（例如 $c = 100$），令 $f(x) = (x/c)^\alpha$（例如 $\alpha = 0.75$），反之令 $f(x) = 1$。需要注意的是，损失函数的计算复杂度与共现词频矩阵 $\boldsymbol{X}$ 中非零元素的数目呈线性关系。我们可以从 $\boldsymbol{X}$ 中随机采样小批量非零元素，使用随机梯度下降迭代词向量和偏移项。

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

$$f(x_{ij})(\boldsymbol{v}_i^\top \tilde{\boldsymbol{v}}_j + b_i + b_j - \log(x_{ij}))^2 + f(x_{ji})(\boldsymbol{v}_j^\top \tilde{\boldsymbol{v}}_i + b_j + b_i - \log(x_{ji}))^2$$

由于 $x_{ij} = x_{ji}$，对调 $\boldsymbol{v}$ 和 $\tilde{\boldsymbol{v}}$ 并不改变损失函数中这两项之和的值。也就是说，在损失函数所有项上对调 $\boldsymbol{v}$ 和 $\tilde{\boldsymbol{v}}$ 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。当所有词向量学习得到后，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

fastText 在使用负采样的跳字模型基础上，将每个中心词视为子词（subword）的集合，并学习子词的词向量。

以 where 这个词为例，设子词为 3 个字符，它的子词包括“<wh”、“whe”、“her”、“ere”、“re>”和特殊子词（整词）“<where>”。其中的“<”和“>”是为了将作为前后缀的子词区分出来。而且，这里的子词“her”与整词“<her>”也可被区分。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 $\boldsymbol{z}_g$，我们可以把使用负采样的跳字模型的损失函数

$$-\log \mathbb{P}(w_o \mid w_c) = -\log \frac{1}{1+\exp(-\boldsymbol{u}_o^\top \boldsymbol{v}_c)} - \sum_{k=1, w_k \sim \mathbb{P}(w)}^{K} \log \frac{1}{1+\exp(\boldsymbol{u}_{i_k}^\top \boldsymbol{v}_c)}$$

直接替换成

$$-\log \mathbb{P}(w_o \mid w_c) = -\log \frac{1}{1+\exp(-\boldsymbol{u}_o^\top \sum_{g \in \mathcal{G}_{w_c}} \boldsymbol{z}_g)} - \sum_{k=1, w_k \sim \mathbb{P}(w)}^{K} \log \frac{1}{1+\exp(\boldsymbol{u}_{i_k}^\top \sum_{g \in \mathcal{G}_{w_c}} \boldsymbol{z}_g)}$$

我们可以看到，原中心词向量被替换成了中心词的子词向量的和。与整词学习（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”。

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。

10.2.4 练习

- GloVe 中，如果一个词出现在另一个词的环境中，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？（可参考Glove 论文4.2 节）
- 如果丢弃 GloVe 中的偏移项，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）？（可参考fastText 论文3.2 节）

10.2.5 扫码直达讨论区

10.3 应用词向量：求近似词和类比词

本节介绍如何通过 `mxnet.contrib.text` 使用预训练的词向量。需要注意的是，`mxnet.contrib.text` 正在测试中并可能在未来有改动。如有改动，本节内容会作相应更新。

本节使用的预训练的 GloVe 和 fastText 词向量分别来自：

- GloVe 项目网站：https://nlp.stanford.edu/projects/glove/
- fastText 项目网站：https://fasttext.cc/

我们先载入需要的包。

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

10.3.1 由数据集建立词典和载入词向量——以 fastText 为例

看一下 fastText 前五个预训练的词向量。它们分别从不同语言的 Wikipedia 数据集训练得到。

```
In [2]: text.embedding.get_pretrained_file_names('fasttext')[:5]
```

```
Out[2]: ['crawl-300d-2M.vec',
         'wiki.aa.vec',
         'wiki.ab.vec',
         'wiki.ace.vec',
         'wiki.ady.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。看一下词典大小。

```
In [5]: len(my_embedding)
```

```
Out[5]: 5
```

任意一个词典以外词的词向量默认为零向量。

```
In [6]: my_embedding.get_vecs_by_tokens('beautiful')[:10]
```

```
Out[6]:
        [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
        <NDArray 10 @cpu(0)>
```

看一下数据集中两个词“hello”和“world”词向量的形状。fastText 中每个词均使用 300 维的词向量。

```
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)>
```

看一下“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 gluon.nn.Embedding

我们可以使用预训练的词向量初始化 `gluon.nn.Embedding`。

```
In [10]: layer = gluon.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 `gluon.nn.Embedding` 得到它们的词向量，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

看一下 GloVe 前五个预训练的词向量。

```
In [12]: text.embedding.get_pretrained_file_names('glove')[:5]
```

```
Out[12]: ['glove.42B.300d.txt',
          'glove.6B.50d.txt',
          'glove.6B.100d.txt',
          'glove.6B.200d.txt',
          'glove.6B.300d.txt']
```

为了演示简便，我们使用小一点的词向量，例如 50 维。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典。

```
In [13]: glove_6b50d = text.embedding.create('glove',
                                             pretrained_file_name='glove.6B.50d.txt')
```

看一下这个词典多大。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。
         print(glove_6b50d.token_to_idx['beautiful'])
         # 索引到词。
         print(glove_6b50d.idx_to_token[3367])
         # 词向量长度。
         print(glove_6b50d.vec_len)
```

```
3367
beautiful
50
```

10.3.3 预训练词向量的应用——以 GloVe 为例

为了应用预训练词向量，我们需要定义余弦相似度。它可以比较两个向量之间的相似度。

```
In [16]: from mxnet import nd
         def cos_sim(x, y):
             return nd.dot(x, y) / (nd.norm(x) * nd.norm(y))
```

余弦相似度的值域在-1 到 1 之间。余弦相似度值越大，两个向量越接近。

```
In [17]: x = nd.array([1, 2])
         y = nd.array([10, 20])
         z = nd.array([-1, -2])

         print(cos_sim(x, y))
         print(cos_sim(x, z))
```

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

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

求近似词

给定任意词，我们可以从整个词典（大小 40 万，不含未知词符号）中找出与它最接近的 k 个词（k nearest neighbors）。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.sqrt(nd.sum(x * x, axis=1)).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”最接近的 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”最接近的 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”最接近的 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”最接近的 5 个词。

```
In [23]: get_knn(glove_6b50d, 5, 'beautiful')
```

```
Out[23]: ['lovely', 'gorgeous', 'wonderful', 'charming', 'beauty']
```

求类比词

我们可以使用预训练词向量求词与词之间的类比关系。例如，man : woman :: son : 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, c，求 d。解类比词的思路是，找到和 c+(b-a) 的结果词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）中找类比词。

```
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 vec(“son”)+vec(“woman”)-vec(“man”) 与 vec(“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

- 使用 `mxnet.contrib.text` 可以轻松载入预训练的词向量。
- 我们可以应用预训练的词向量求相似词和类比词。

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’）。
- 如果在使用循环神经网络的语言模型中将 Embedding 层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

10.4 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]: from collections import Counter
        import os
        import random
        import time
        import zipfile

        import mxnet as mx
        from mxnet import autograd, gluon, init, nd
        from mxnet.contrib import text
```

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 。

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']
            dataset = []
            for lb in pos_or_neg:
                files = os.listdir('../data/' + dir_url + '/' + seg + '/' + lb + '/')
                for file in files:
                    with open('../data/' + dir_url + '/' + seg + '/' + lb + '/'
                              + file, 'r', encoding='utf8') as rf:
                        review = rf.read().replace('\n','')
                        if lb == 'pos':
                            dataset.append([review, 1])
                        elif lb == 'neg':
                            dataset.append([review, 0])
            return dataset

        if demo:
            train_dataset = readIMDB('aclImdb_tiny/', 'train')
            test_dataset = readIMDB('aclImdb_tiny/', 'test')
        else:
            train_dataset = readIMDB('aclImdb/', 'train')
            test_dataset = readIMDB('aclImdb/', 'test')

        # shuffle 数据集。
        random.shuffle(train_dataset)
        random.shuffle(test_dataset)
```

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 = []
        train_labels = []
        for review, score in train_dataset:
            train_tokenized.append(tokenizer(review))
            train_labels.append(score)
        test_tokenized = []
        test_labels = []
        for review, score in test_dataset:
            test_tokenized.append(tokenizer(review))
            test_labels.append(score)
```

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’<unk>‘，所有不存在在词典中的词，未来都将对应到这个符号。

```
In [6]: token_counter = 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(x_raw_samples, vocab):
            x_encoded_samples = []
            for sample in x_raw_samples:
                x_encoded_sample = []
                for token in sample:
                    if token in vocab.token_to_idx:
                        x_encoded_sample.append(vocab.token_to_idx[token])
                    else:
                        x_encoded_sample.append(0)
                x_encoded_samples.append(x_encoded_sample)
            return x_encoded_samples

        # 将特征向量补成定长。
        def pad_samples(x_encoded_samples, maxlen=500, val=0):
            x_samples = []
            for sample in x_encoded_samples:
                if len(sample) > maxlen:
                    new_sample = sample[:maxlen]
                else:
                    num_padding = maxlen - len(sample)
                    new_sample = sample
                    for i in range(num_padding):
                        new_sample.append(val)
                x_samples.append(new_sample)
            return x_samples
```

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]: x_encoded_train = encode_samples(train_tokenized, vocab)
        x_encoded_test = 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]: # 指定 context。
        context = mx.gpu(0)
        x_train = nd.array(pad_samples(x_encoded_train, 500, 0), ctx=context)
        x_test = nd.array(pad_samples(x_encoded_test, 500, 0), ctx=context)
```

这里，我们将情感标签也转化成为了 NDArray。

```
In [10]: y_train = nd.array([score for text, score in train_dataset], ctx=context)
         y_test = nd.array([score for text, score in test_dataset], ctx=context)
```

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。我们在这里使用’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]: nclass = 2
         lr = 0.1
         num_epochs = 1
         batch_size = 10
         emsize = 100
         num_hiddens = 100
         nlayers = 2
         bidirectional = True
```

```
In [13]: class SentimentNet(gluon.Block):
             def __init__(self, vocab, emsize, num_hiddens, nlayers, bidirectional,
```

```
                 **kwargs):
        super(SentimentNet, self).__init__(**kwargs)
        with self.name_scope():
            self.embedding = gluon.nn.Embedding(
                len(vocab), emsize, weight_initializer=init.Uniform(0.1))
            self.encoder = gluon.rnn.LSTM(num_hiddens, num_layers=nlayers,
                                          bidirectional=bidirectional,
                                          input_size=emsize)
            self.decoder = gluon.nn.Dense(nclass, 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, emsize, num_hiddens, nlayers, bidirectional)
net.initialize(mx.init.Xavier(), ctx=context)
# 设置 embedding 层的 weight 为词向量。
net.embedding.weight.set_data(
    glove_embedding.idx_to_vec.as_in_context(context))
# 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 = gluon.loss.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(x_samples, y_samples):
             total_L = 0
             ntotal = 0
             accuracy = mx.metric.Accuracy()
             for i in range(x_samples.shape[0] // batch_size):
                 data = x_samples[i * batch_size : (i+1) * batch_size]
                 target = y_samples[i * batch_size :(i+1) * batch_size]
                 data = data.as_in_context(context).T
```

```
            target = target.as_in_context(context).T
            output = net(data)
            L = loss(output, target)
            total_L += nd.sum(L).asscalar()
            ntotal += L.size
            predicts = nd.argmax(output, axis=1)
            accuracy.update(preds=predicts, labels=target)
        return total_L / ntotal, accuracy.get()[1]
```

运行下面的代码开始训练模型。我们每 800 个 batch，会输出一次当前的 loss。

```
In [15]: start_train_time = time.time()
         for epoch in range(num_epochs):
             start_epoch_time = time.time()
             total_L = 0
             ntotal = 0
             for i in range(x_train.shape[0] // batch_size):
                 data = x_train[i * batch_size : (i+1) * batch_size]
                 target = y_train[i * batch_size : (i+1) * batch_size]
                 data = data.as_in_context(context).T
                 target = target.as_in_context(context).T
                 with autograd.record():
                     output = net(data)
                     L = loss(output, target)
                 L.backward()
                 trainer.step(batch_size)
                 total_L += nd.sum(L).asscalar()
                 ntotal += L.size
                 if i % 800 == 0 and i != 0:
                     print('[epoch %d] batch %d. loss %.6f' % (epoch, i,
                                                               total_L / ntotal))
                     total_L = 0
                     ntotal = 0

             print('performing testing:')
             train_loss, train_acc = eval(x_train, y_train)
             test_loss, test_acc = eval(x_test, y_test)

             print('[epoch %d] train loss %.6f, train accuracy %.2f'
                   % (epoch, train_loss, train_acc))
             print('[epoch %d] test loss %.6f, test accuracy %.2f'
                   % (epoch, test_loss, test_acc))
             print('[epoch %d] throughput %.2f samples/s'
                   % (epoch, (batch_size * len(x_train))
```

```
                    / (time.time() - start_epoch_time)))
            print('[epoch %d] total time %.2f s'
                  % (epoch, (time.time() - start_epoch_time)))

        print('total training throughput %.2f samples/s'
              % ((batch_size * len(x_train) * num_epochs)
                 / (time.time() - start_train_time)))
        print('total training time %.2f s' % ((time.time() - start_train_time)))
```

```
performing testing:
[epoch 0] train loss 0.693508, train accuracy 0.45
[epoch 0] test loss 0.691830, test accuracy 0.55
[epoch 0] throughput 256.93 samples/s
[epoch 0] total time 0.78 s
total training throughput 256.76 samples/s
total training time 0.78 s
```

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。

```
In [16]: review = ['this', 'movie', 'is', 'great']
         print(review)
```

```
['this', 'movie', 'is', 'great']
```

上面这个句子的情感是（1 代表正面，0 代表负面）：

```
In [17]: nd.argmax(net(nd.reshape(
             nd.array([vocab.token_to_idx[token] for token in review], ctx=context),
             shape=(-1, 1))), axis=1).asscalar()
```

```
Out[17]: 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）和注意力机制

在基于词语的语言模型中，我们使用了循环神经网络。它的输入是一段不定长的序列，输出却是定长的，例如一个词语。然而，很多问题的输出也是不定长的序列。以机器翻译为例，输入是可以是英语的一段话，输出可以是法语的一段话，输入和输出皆不定长，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）或者 seq2seq。它们分别基于 2014 年的两个工作：

- Cho et al., Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation
- Sutskever et al., Sequence to Sequence Learning with Neural Networks

以上两个工作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—解码器

编码器和解码器是分别对应输入序列和输出序列的两个循环神经网络。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。

编码器

编码器的作用是把一个不定长的输入序列转化成一个定长的背景向量 $\boldsymbol{c}$。该背景向量包含了输入序列的信息。常用的编码器是循环神经网络。

我们回顾一下循环神经网络知识。假设循环神经网络单元为 f，在 t 时刻的输入为 $x_t, t = 1, \ldots, T$。假设 $\boldsymbol{x}_t$ 是单个输出 x_t 在嵌入层的结果，例如 x_t 对应的 one-hot 向量 $\boldsymbol{o} \in \mathbb{R}^x$ 与嵌入层参数矩阵 $\boldsymbol{E} \in \mathbb{R}^{x \times h}$ 的乘积 $\boldsymbol{o}^\top \boldsymbol{E}$。隐含层变量

$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})$$

编码器的背景向量

$$\boldsymbol{c} = q(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T)$$

一个简单的背景向量是该网络最终时刻的隐含层变量 $\boldsymbol{h}_T$。我们将这里的循环神经网络叫做编码器。

双向循环神经网络

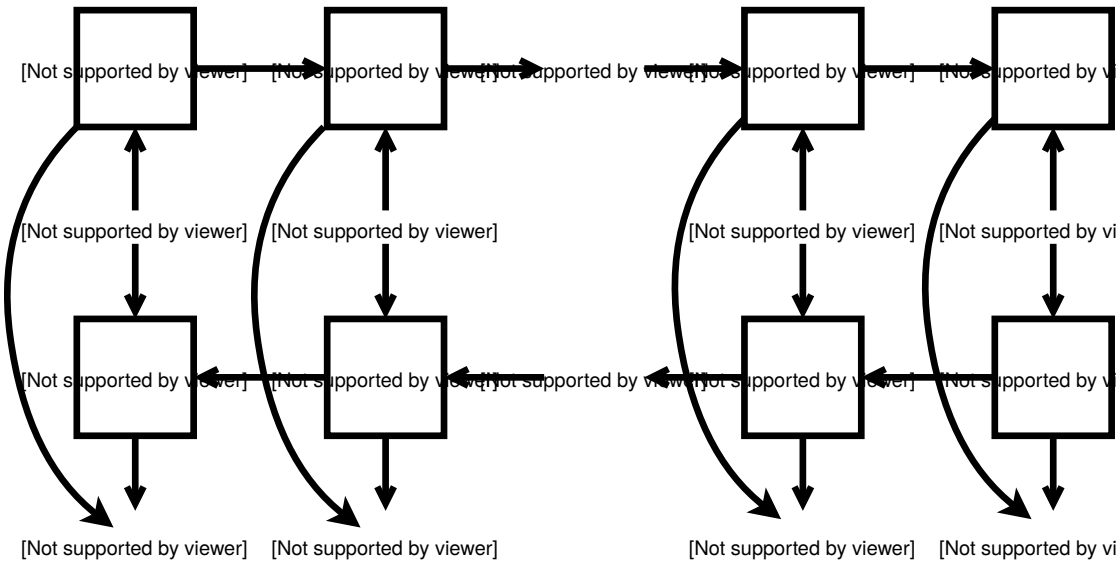
编码器的输入既可以是正向传递，也可以是反向传递。如果输入序列是 $x_1, x_2, \ldots, x_T$，在正向传递中，隐含层变量

$$\overrightarrow{\boldsymbol{h}}_t = f(\boldsymbol{x}_t, \overrightarrow{\boldsymbol{h}}_{t-1})$$

而反向传递中，隐含层变量的计算变为

$$\overleftarrow{\boldsymbol{h}}_t = f(\boldsymbol{x}_t, \overleftarrow{\boldsymbol{h}}_{t+1})$$

当我们希望编码器的输入既包含正向传递信息又包含反向传递信息时，我们可以使用双向循环神经网络。例如，给定输入序列 $x_1, x_2, \ldots, x_T$，按正向传递，它们在循环神经网络的隐含层变量分别是 $\overrightarrow{\boldsymbol{h}}_1, \overrightarrow{\boldsymbol{h}}_2, \ldots, \overrightarrow{\boldsymbol{h}}_T$；按反向传递，它们在循环神经网络的隐含层变量分别是 $\overleftarrow{\boldsymbol{h}}_1, \overleftarrow{\boldsymbol{h}}_2, \ldots, \overleftarrow{\boldsymbol{h}}_T$。在双向循环神经网络中，时刻 i 的隐含层变量可以把 $\overrightarrow{\boldsymbol{h}}_i$ 和 $\overleftarrow{\boldsymbol{h}}_i$ 连结起来。



```
In [1]: # 连结两个向量。
        from mxnet import nd
        h_forward = nd.array([1, 2])
        h_backward = nd.array([3, 4])
        h_bi = nd.concat(h_forward, h_backward, dim=0)
        h_bi
```

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

解码器

编码器最终输出了一个背景向量 $\boldsymbol{c}$，该背景向量编码了输入序列 $x_1, x_2, \ldots, x_T$ 的信息。

假设训练数据中的输出序列是 $y_1, y_2, \ldots, y_{T'}$，我们希望表示每个 t 时刻输出的既取决于之前的输出又取决于背景向量。之后，我们就可以最大化输出序列的联合概率

$$\mathbb{P}(y_1, \ldots, y_{T'}) = \prod_{t'=1}^{T'} \mathbb{P}(y_{t'} \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c})$$

并得到该输出序列的损失函数

$$-\log \mathbb{P}(y_1, \ldots, y_{T'})$$

为此，我们使用另一个循环神经网络作为解码器。解码器使用函数 p 来表示单个输出 $y_{t'}$ 的概率

$$\mathbb{P}(y_{t'} \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c}) = p(y_{t'-1}, \boldsymbol{s}_{t'}, \boldsymbol{c})$$

其中的 $\boldsymbol{s}_t$ 为 t' 时刻的解码器的隐含层变量。该隐含层变量

$$\boldsymbol{s}_{t'} = g(y_{t'-1}, \boldsymbol{c}, \boldsymbol{s}_{t'-1})$$

其中函数 g 是循环神经网络单元。

需要注意的是，编码器和解码器通常会使用多层循环神经网络。

10.5.2 注意力机制

在以上的解码器设计中，各个时刻使用了相同的背景向量。如果解码器的不同时刻可以使用不同的背景向量呢？

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”，解码器在时刻 1 可以使用更多编码了“they are”信息的背景向量来生成“Ils”，而在时刻 2 可以使用更多编码了“watching”信息的背景向量来生成“regardent”。这看上去就像是在解码器的每一时刻对输入序列中不同时刻分配不同的注意力。这也是注意力机制的由来。它最早由 Bahanau 等在 2015 年提出。

现在，对上面的解码器稍作修改。我们假设时刻 t' 的背景向量为 $\boldsymbol{c}_{t'}$。那么解码器在 t' 时刻的隐含层变量

$$\boldsymbol{s}_{t'} = g(\boldsymbol{y}_{t'-1}, \boldsymbol{c}_{t'}, \boldsymbol{s}_{t'-1})$$

令编码器在 t 时刻的隐含变量为 $\boldsymbol{h}_t$，解码器在 t' 时刻的背景向量为

$$\boldsymbol{c}_{t'} = \sum_{t=1}^{T} \alpha_{t't} \boldsymbol{h}_t$$

也就是说，给定解码器的当前时刻 t'，我们需要对编码器中不同时刻 t 的隐含层变量求加权平均。而权值也称注意力权重。它的计算公式是

$$\alpha_{t't} = \frac{\exp(e_{t't})}{\sum_{k=1}^{T} \exp(e_{t'k})}$$

而 $e_{t't} \in \mathbb{R}$ 的计算为：

$$e_{t't} = a(\boldsymbol{s}_{t'-1}, \boldsymbol{h}_t)$$

其中函数 a 有多种设计方法。在Bahanau 的论文中，

$$e_{t't} = \boldsymbol{v}^\top \tanh(\boldsymbol{W}_s \boldsymbol{s}_{t'-1} + \boldsymbol{W}_h \boldsymbol{h}_t)$$

其中的 $\boldsymbol{v}$、$\boldsymbol{W}_s$、$\boldsymbol{W}_h$ 和编码器与解码器两个循环神经网络中的各个权重和偏移项以及嵌入层参数等都是需要同时学习的模型参数。在Bahanau 的论文中，编码器和解码器分别使用了门控循环单元（GRU）。

在解码器中，我们需要对 GRU 的设计稍作修改。假设 $\boldsymbol{y}_t$ 是单个输出 y_t 在嵌入层的结果，例如 y_t 对应的 one-hot 向量 $\boldsymbol{o} \in \mathbb{R}^y$ 与嵌入层参数矩阵 $\boldsymbol{B} \in \mathbb{R}^{y \times s}$ 的乘积 $\boldsymbol{o}^\top \boldsymbol{B}$。假设时刻 t' 的背景向量为 $\boldsymbol{c}_{t'}$。那么解码器在 t' 时刻的单个隐含层变量

$$\boldsymbol{s}_{t'} = \boldsymbol{z}_{t'} \odot \boldsymbol{s}_{t'-1} + (1 - \boldsymbol{z}_{t'}) \odot \tilde{\boldsymbol{s}}_{t'}$$

其中的重置门、更新门和候选隐含状态分别为

$$\boldsymbol{r}_{t'} = \sigma(\boldsymbol{W}_{yr} \boldsymbol{y}_{t'-1} + \boldsymbol{W}_{sr} \boldsymbol{s}_{t'-1} + \boldsymbol{W}_{cr} \boldsymbol{c}_{t'} + \boldsymbol{b}_r)$$

$$\boldsymbol{z}_{t'} = \sigma(\boldsymbol{W}_{yz}\boldsymbol{y}_{t'-1} + \boldsymbol{W}_{sz}\boldsymbol{s}_{t'-1} + \boldsymbol{W}_{cz}\boldsymbol{c}_{t'} + \boldsymbol{b}_z)$$

$$\tilde{\boldsymbol{s}}_{t'} = \tanh(\boldsymbol{W}_{ys}\boldsymbol{y}_{t'-1} + \boldsymbol{W}_{ss}(\boldsymbol{s}_{t'-1} \odot \boldsymbol{r}_{t'}) + \boldsymbol{W}_{cs}\boldsymbol{c}_{t'} + \boldsymbol{b}_s)$$

10.5.3 小结

- 编码器-解码器（seq2seq）的输入和输出可以都是不定长序列。
- 在解码器上应用注意力机制可以在解码器的每个时刻使用不同的背景向量。每个背景向量相当于对输入序列的不同部分分配了不同的注意力。

10.5.4 练习

- 了解其他的注意力机制设计。例如论文Effective Approaches to Attention-based Neural Machine Translation。
- 在Bahanau 的论文中，我们是否需要重新实现解码器上的 GRU？
- 除了机器翻译，你还能想到 seq2seq 的哪些应用？
- 除了自然语言处理，注意力机制还可以应用在哪些地方？

10.5.5 扫码直达讨论区

10.6 应用编码器—解码器和注意力机制：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。

10.6.1 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 mxnet as mx
        from mxnet import autograd, gluon, nd
        from mxnet.gluon import nn, rnn, Block
        from mxnet.contrib import text

        from io import open
        import collections
        import datetime
```

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 = 50
        epoch_period = 10

        learning_rate = 0.005
        # 输入或输出序列的最大长度（含句末添加的 EOS 字符）。
        max_seq_len = 5

        encoder_num_layers = 1
        decoder_num_layers = 2

        encoder_drop_prob = 0.1
        decoder_drop_prob = 0.1

        encoder_hidden_dim = 256
        decoder_hidden_dim = 256
        alignment_dim = 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。

```
In [4]: def read_data(max_seq_len):
            input_tokens = []
            output_tokens = []
            input_seqs = []
            output_seqs = []

            with 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)
```

```
        X = nd.zeros((len(input_seqs), max_seq_len), ctx=ctx)
        Y = nd.zeros((len(output_seqs), max_seq_len), ctx=ctx)
        for i in range(len(input_seqs)):
            X[i] = nd.array(input_vocab.to_indices(input_seqs[i]), ctx=ctx)
            Y[i] = nd.array(output_vocab.to_indices(output_seqs[i]), ctx=ctx)

        dataset = gluon.data.ArrayDataset(X, Y)
```

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(Block):
            """ 编码器"""
            def __init__(self, input_dim, hidden_dim, num_layers, drop_prob,
                         **kwargs):
                super(Encoder, self).__init__(**kwargs)
                with self.name_scope():
                    self.embedding = nn.Embedding(input_dim, hidden_dim)
                    self.dropout = nn.Dropout(drop_prob)
                    self.rnn = rnn.GRU(hidden_dim, num_layers, dropout=drop_prob,
                                       input_size=hidden_dim)

            def forward(self, inputs, state):
                # inputs 尺寸: (batch_size, num_steps), emb 尺寸: (num_steps,
 ↪  batch_size, 256)
                emb = self.embedding(inputs).swapaxes(0, 1)
                emb = self.dropout(emb)
                output, state = self.rnn(emb, 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(Block):
            """ 含注意力机制的解码器"""
            def __init__(self, hidden_dim, output_dim, num_layers, max_seq_len,
                         drop_prob, alignment_dim, encoder_hidden_dim, **kwargs):
                super(Decoder, self).__init__(**kwargs)
                self.max_seq_len = max_seq_len
                self.encoder_hidden_dim = encoder_hidden_dim
                self.hidden_size = hidden_dim
```

```
        self.num_layers = num_layers
        with self.name_scope():
            self.embedding = nn.Embedding(output_dim, hidden_dim)
            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_dim, in_units=hidden_dim + encoder_hidden_dim,
                    activation="tanh", flatten=False))
                self.attention.add(nn.Dense(1, in_units=alignment_dim,
                                            flatten=False))

            self.rnn = rnn.GRU(hidden_dim, num_layers, dropout=drop_prob,
                               input_size=hidden_dim)
            self.out = nn.Dense(output_dim, in_units=hidden_dim, flatten=False)
            self.rnn_concat_input = nn.Dense(
                hidden_dim, in_units=hidden_dim + encoder_hidden_dim,
                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含状态。
        single_layer_state = [state[0][-1].expand_dims(0)]
        encoder_outputs = encoder_outputs.reshape((self.max_seq_len, -1,
                                                   self.encoder_hidden_dim))
        # single_layer_state 尺寸: [(1, batch_size, decoder_hidden_dim)]
        # hidden_broadcast 尺寸: (max_seq_len, batch_size, decoder_hidden_dim)
        hidden_broadcast = nd.broadcast_axis(single_layer_state[0], axis=0,
                                             size=self.max_seq_len)

        # encoder_outputs_and_hiddens 尺寸:
        # (max_seq_len, batch_size, encoder_hidden_dim + decoder_hidden_dim)
        encoder_outputs_and_hiddens = nd.concat(encoder_outputs,
                                                hidden_broadcast, dim=2)

        # energy 尺寸: (max_seq_len, batch_size, 1)
        energy = self.attention(encoder_outputs_and_hiddens)

        # batch_attention 尺寸: (batch_size, 1, max_seq_len)
        batch_attention = nd.softmax(energy, axis=0).transpose(
            (1, 2, 0))
```

```
                # batch_encoder_outputs 尺寸: (batch_size, max_seq_len,
↪  encoder_hidden_dim)
                batch_encoder_outputs = encoder_outputs.swapaxes(0, 1)

                # decoder_context 尺寸: (batch_size, 1, encoder_hidden_dim)
                decoder_context = nd.batch_dot(batch_attention, batch_encoder_outputs)

                # cur_input 尺寸: (batch_size,)
                # input_and_context 尺寸: (batch_size, 1, encoder_hidden_dim +
↪  decoder_hidden_dim)
                input_and_context =
↪  nd.concat(nd.expand_dims(self.embedding(cur_input), axis=1),
                                             decoder_context, dim=2)
                # concat_input 尺寸: (1, batch_size, decoder_hidden_dim)
                concat_input = self.rnn_concat_input(input_and_context).reshape((1,
↪  -1, 0))
                concat_input = self.dropout(concat_input)

                # 当 RNN 为多层时，用单层隐含状态初始化各个层的隐含状态。
                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))
                # output 尺寸: (batch_size, output_size)
                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含状态，我们通过一层全连接网络来转化编码器的输出隐含状态。

```
In [8]: class DecoderInitState(Block):
            """ 解码器隐含状态的初始化"""
            def __init__(self, encoder_hidden_dim, decoder_hidden_dim, **kwargs):
                super(DecoderInitState, self).__init__(**kwargs)
                with self.name_scope():
                    self.dense = nn.Dense(decoder_hidden_dim,
                                          in_units=encoder_hidden_dim,
                                          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(input_vocab.to_indices(input_tokens), ctx=ctx)
                encoder_state = encoder.begin_state(func=mx.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 = []

                while True:
                    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含状态，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含状态。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]: def train(encoder, decoder, decoder_init_state, max_seq_len, ctx,
 ↪   eval_fr_ens):
             # 对于三个网络，分别初始化它们的模型参数并定义它们的优化器。
             encoder.collect_params().initialize(mx.init.Xavier(), ctx=ctx)
```

```
    decoder.collect_params().initialize(mx.init.Xavier(), ctx=ctx)
    decoder_init_state.collect_params().initialize(mx.init.Xavier(), ctx=ctx)
    encoder_optimizer = gluon.Trainer(encoder.collect_params(), 'adam',
                                      {'learning_rate': learning_rate})
    decoder_optimizer = gluon.Trainer(decoder.collect_params(), 'adam',
                                      {'learning_rate': learning_rate})
    decoder_init_state_optimizer = gluon.Trainer(
        decoder_init_state.collect_params(), 'adam',
        {'learning_rate': learning_rate})

    softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()

    prev_time = datetime.datetime.now()
    data_iter = gluon.data.DataLoader(dataset, 1, shuffle=True)

    total_loss = 0.0
    for epoch in range(1, epochs + 1):
        for x, y in data_iter:
            with autograd.record():
                loss = nd.array([0], ctx=ctx)
                encoder_state = encoder.begin_state(
                    func=mx.nd.zeros, batch_size=1, ctx=ctx)
                encoder_outputs, encoder_state = encoder(x, encoder_state)

                # encoder_outputs 尺寸: (max_seq_len, encoder_hidden_dim)
                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])
                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)
                    loss = loss + softmax_cross_entropy(decoder_output[0],
↪   y[0][i])
                    if y[0][i].asscalar() == output_vocab.token_to_idx[EOS]:
                        break

                loss.backward()
                encoder_optimizer.step(1)
                decoder_optimizer.step(1)
```

```
            decoder_init_state_optimizer.step(1)
            total_loss += loss.asscalar() / max_seq_len

        if epoch % epoch_period == 0 or epoch == 1:
            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 epoch == 1:
                print_loss_avg = total_loss / len(data_iter)
            else:
                print_loss_avg = total_loss / epoch_period / len(data_iter)
            loss_str = 'Epoch %d, Loss %f, ' % (epoch, print_loss_avg)
            print(loss_str + time_str)
            if epoch != 1:
                total_loss = 0.0
            prev_time = cur_time

            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含状态网络。

```
In [11]: encoder = Encoder(len(input_vocab), encoder_hidden_dim, encoder_num_layers,
                           encoder_drop_prob)
         decoder = Decoder(decoder_hidden_dim, len(output_vocab),
                           decoder_num_layers, max_seq_len, decoder_drop_prob,
                           alignment_dim, encoder_hidden_dim)
         decoder_init_state = DecoderInitState(encoder_hidden_dim, decoder_hidden_dim)
```

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 2.546908, Time 00:00:00
Input : elle est japonaise .
Output: they is . . .
Expect: she is japanese .

Input : ils regardent .
Output: they are . .
Expect: they are watching .
```

```
Epoch 10, Loss 1.101315, Time 00:00:04
Input : elle est japonaise .
Output: she is japanese .
Expect: she is japanese .

Input : ils regardent .
Output: she is quiet .
Expect: they are watching .

Epoch 20, Loss 0.366322, Time 00:00:04
Input : elle est japonaise .
Output: she is japanese .
Expect: she is japanese .

Input : ils regardent .
Output: they are watching .
Expect: they are watching .

Epoch 30, Loss 0.017124, Time 00:00:04
Input : elle est japonaise .
Output: she is japanese .
Expect: she is japanese .

Input : ils regardent .
Output: they are watching .
Expect: they are watching .

Epoch 40, Loss 0.002681, Time 00:00:04
Input : elle est japonaise .
Output: she is japanese .
Expect: she is japanese .

Input : ils regardent .
Output: they are watching .
Expect: they are watching .

Epoch 50, Loss 0.001712, Time 00:00:04
Input : elle est japonaise .
Output: she is japanese .
Expect: she is japanese .

Input : ils regardent .
```

```
Output: they are watching .
Expect: they are watching .
```

10.6.2 束搜索

在上一节里，我们提到编码器最终输出了一个背景向量 $\boldsymbol{c}$，该背景向量编码了输入序列 $x_1, x_2, \ldots, x_T$ 的信息。假设训练数据中的输出序列是 $y_1, y_2, \ldots, y_{T'}$，输出序列的生成概率是

$$\mathbb{P}(y_1, \ldots, y_{T'}) = \prod_{t'=1}^{T'} \mathbb{P}(y_{t'} \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c}).$$

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

$$y_{t'} = \text{argmax}_{y_{t'} \in \mathcal{Y}} \mathbb{P}(y_{t'} \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c})$$

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 $\mathbb{P}(y_{t'} \mid \boldsymbol{c})$ 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 $\mathbb{P}(y_2 \mid A, \boldsymbol{c})$ 和 $\mathbb{P}(y_2 \mid C, \boldsymbol{c})$，从计算出的 10 个概率中取最大的两个，假设为 $\mathbb{P}(B \mid A, \boldsymbol{c})$ 和 $\mathbb{P}(E \mid C, \boldsymbol{c})$。那么，我们在时刻 3 对于所有的 $y_3 \in \mathcal{Y}$ 都分别计算 $\mathbb{P}(y_3 \mid A, B, \boldsymbol{c})$ 和 $\mathbb{P}(y_3 \mid C, E, \boldsymbol{c})$，从计算出的 10 个概率中取最大的两个，假设为 $\mathbb{P}(D \mid A, B, \boldsymbol{c})$ 和 $\mathbb{P}(D \mid C, E, \boldsymbol{c})$。

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：

$$\frac{1}{L^\alpha} \log \mathbb{P}(y_1, \ldots, y_L) = \frac{1}{L^\alpha} \sum_{t'=1}^{L} \log \mathbb{P}(y_{t'} \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c})$$

其中 L 为候选序列长度，α 一般可选为 0.75。分母上的 L^α 是为了惩罚较长序列的分数中的对数相加项。

10.6.3 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 Understudy）。

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 ABBCD。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 $\exp(1 - 6/3) \approx 0.37$，因此 BLEU=0.37。当模型输出也为 ABCDEF 时，BLEU=1。

10.6.4 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。

10.6.5 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？

10.6.6 扫码直达讨论区

11

附录

11.1 数学基础

TODO(@astonzhang)。

11.1.1 向量

设 d 维向量 $\boldsymbol{x} = [x_1, x_2, \ldots, x_d]^\top$，或

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}.$$

上式中 $x_1, \ldots, x_d$ 是向量的元素。我们用 $\boldsymbol{x} \in \mathbb{R}^d$ 或 $\boldsymbol{x} \in \mathbb{R}^{d \times 1}$ 表示，$\boldsymbol{x}$ 是一个各元素均为实数的 d 维向量。

11.1.2 矩阵

设 m 行 n 列矩阵

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix}.$$

上式中 $x_{11}, \dots, x_{mn}$ 是矩阵的元素。我们用 $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ 表示，$\boldsymbol{X}$ 是一个各元素均为实数的 m 行 n 列矩阵。

11.1.3 运算

按元素运算

假设 $\boldsymbol{x} = [4, 9]^\top$，以下是一些按元素运算的例子：

- 按元素相加：$\boldsymbol{x} + 1 = [5, 10]^\top$
- 按元素相乘：$\boldsymbol{x} \odot \boldsymbol{x} = [16, 81]^\top$
- 按元素相除：$72/\boldsymbol{x} = [18, 8]^\top$
- 按元素开方：$\sqrt{\boldsymbol{x}} = [2, 3]^\top$

11.1.4 导数和梯度

TODO(@astonzhang)。

假设函数 $f : \mathbb{R}^d \to \mathbb{R}$ 的输入是一个 d 维向量 $\boldsymbol{x} = [x_1, x_2, \dots, x_d]^\top$。函数 $f(\boldsymbol{x})$ 有关 $\boldsymbol{x}$ 的梯度是一个由 d 个偏导数组成的向量：

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \left[\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \frac{\partial f(\boldsymbol{x})}{\partial x_2}, \dots, \frac{\partial f(\boldsymbol{x})}{\partial x_d}\right]^\top.$$

为表示简洁，我们有时用 $\nabla f(\boldsymbol{x})$ 代替 $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$。

常用梯度

$$\nabla_{\boldsymbol{x}} \boldsymbol{A}^\top \boldsymbol{x} = \boldsymbol{A}$$
$$\nabla_{\boldsymbol{x}} \boldsymbol{x}^\top \boldsymbol{A} = \boldsymbol{A}$$
$$\nabla_{\boldsymbol{x}} \boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} = (\boldsymbol{A} + \boldsymbol{A}^\top)\boldsymbol{x}$$
$$\nabla_{\boldsymbol{x}} \|\boldsymbol{x}\|^2 = \nabla_{\boldsymbol{x}} \boldsymbol{x}^\top \boldsymbol{x} = 2\boldsymbol{x}$$

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 ϵ 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 ϵ 足够小，上式也可以简化成

$$f(x+\epsilon) \approx f(x) + f'(x)\epsilon.$$

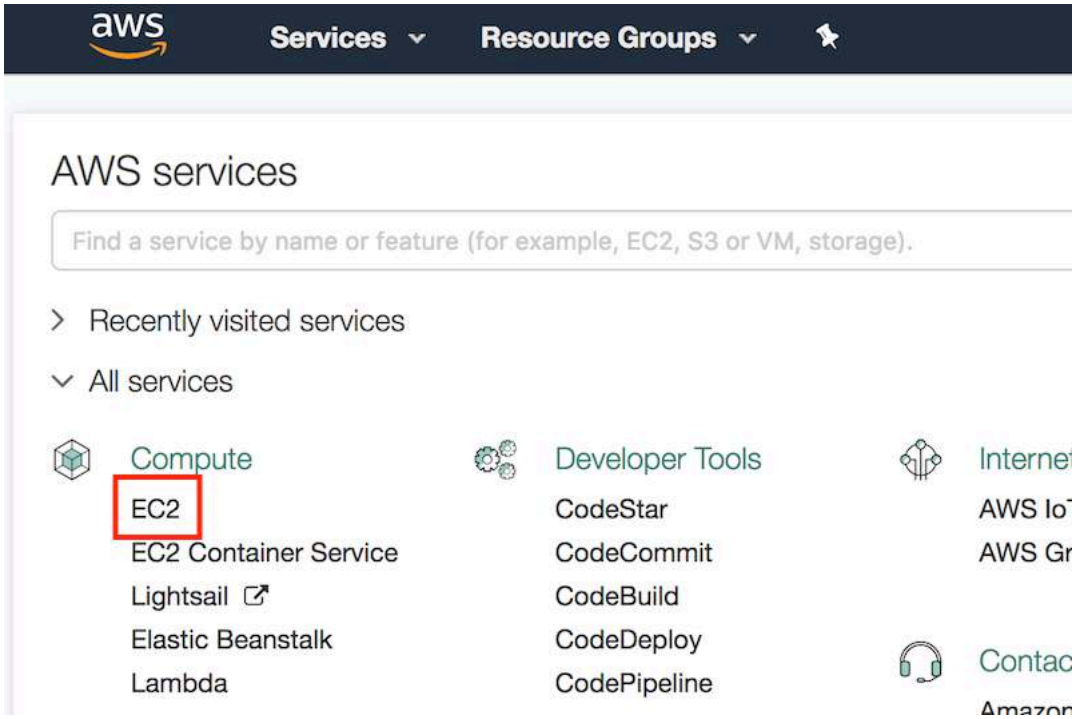
11.2 在 AWS 上运行教程

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：

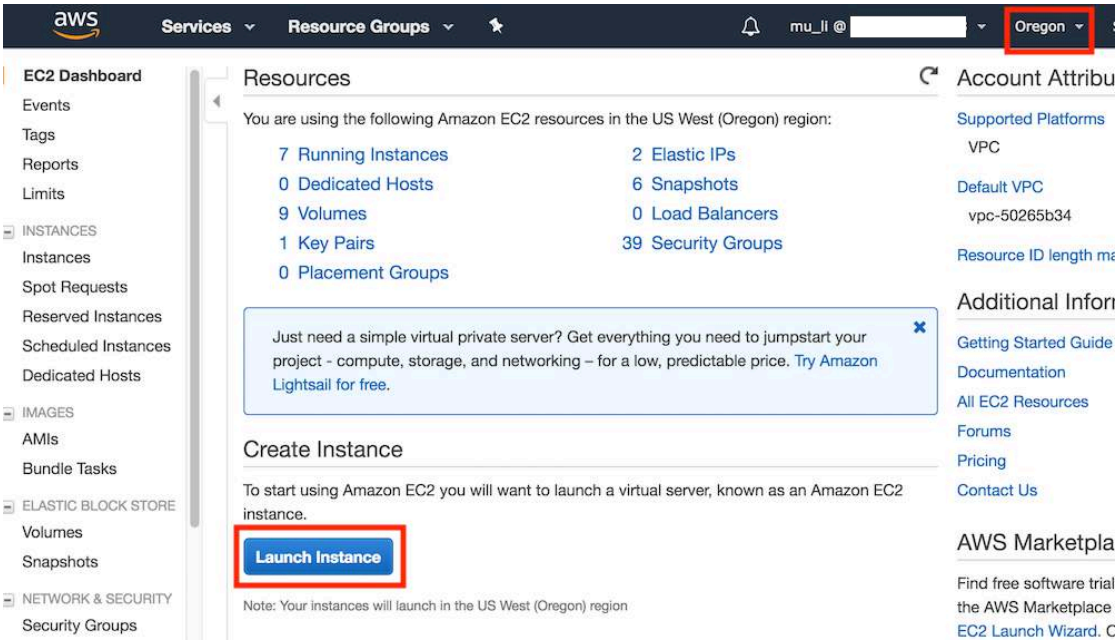


11.2.2 选择并运行 EC2 实例

【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。

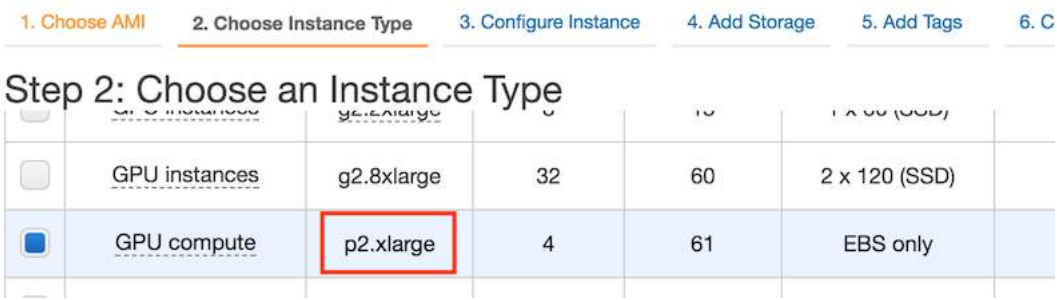
【注意】有些数据中心可能没有 GPU 实例。

然后点击启动实例来选择操作系统和实例类型。



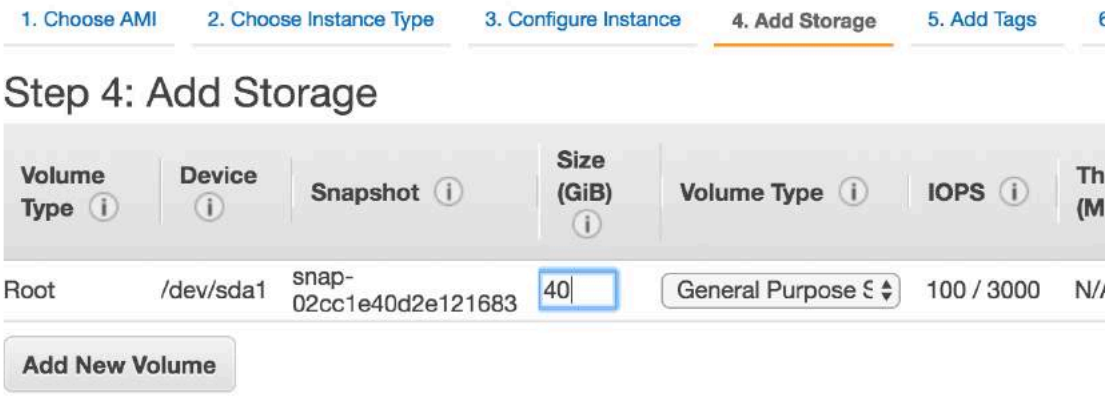
在接下来的操作系统里面选 Ubuntu 16.06:

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.

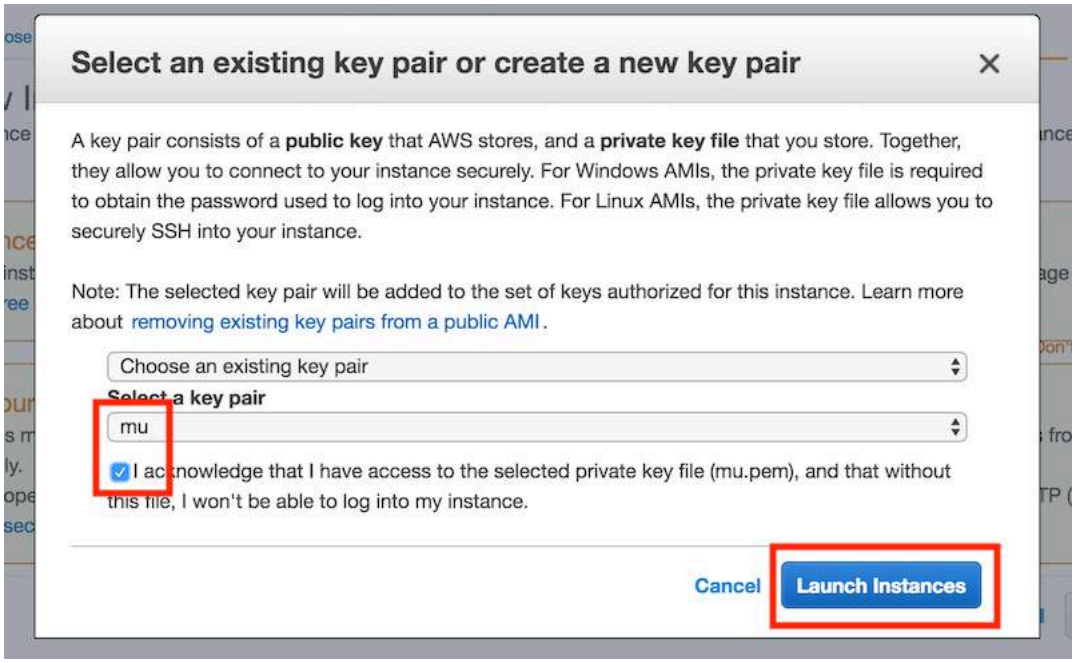


【注意】选择某个类型的样例前我们需要在 `Limits` 里检查下这个是不是有数量限制。例如这个账号的 `p2.xlarge` 限制是最多一个区域开一个。如果需要更多，需要点右边来申请更多的实例容量（通常需要一个工作日来处理）。

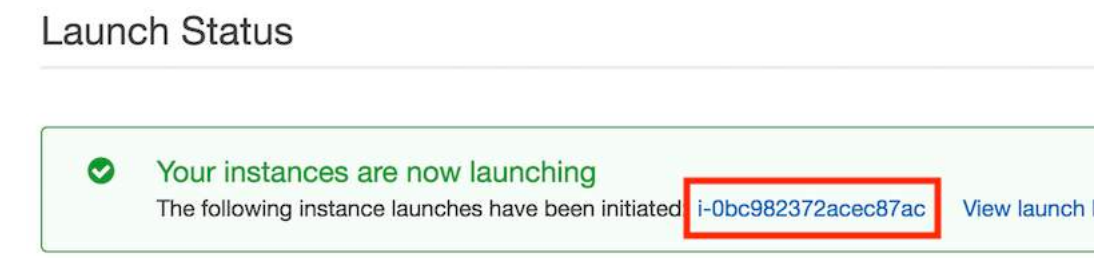
然后我们在存储那里将默认的硬盘从 8GB 增大的 40GB. 因为安装 CUDA 需要 4GB 空间，所以最小推荐 20GB（只有 CPU 的话不需要 CUDA，可以更少）。



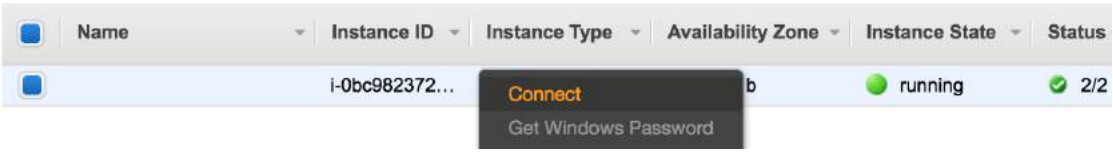
其他的项我们都选默认，然后可以启动了。这时候会跳出选项选择 `key pair`，这是用来之后访问机器的秘钥（EC2 默认不支持密码访问）。如果没有的话可以选择生成一对秘钥。



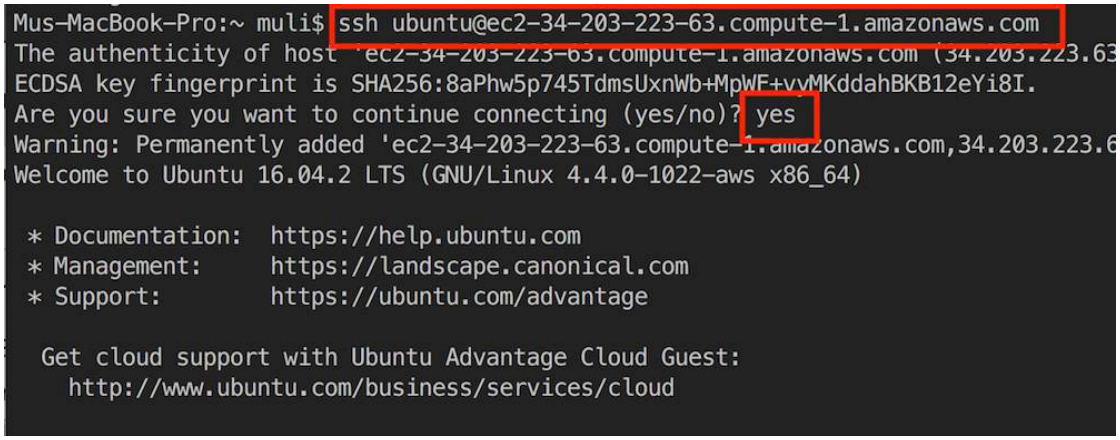
然后点击实例的 ID 可以查看当前实例的状态



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



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。

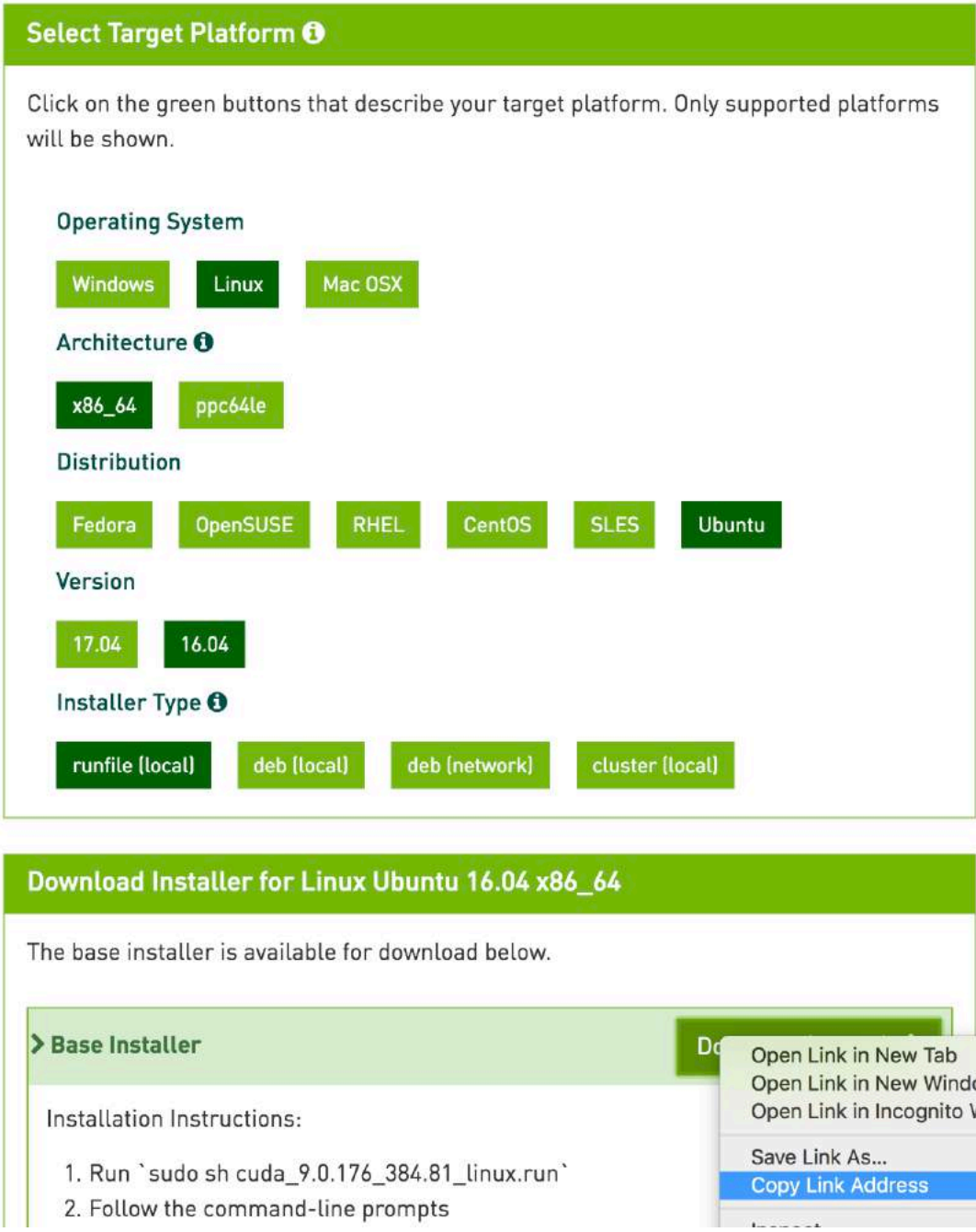
```
sudo apt-get update && sudo apt-get install -y build-essential git libgfortran3
```

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。



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 jupter 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/1
[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
[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 brow
[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:locallhost: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。

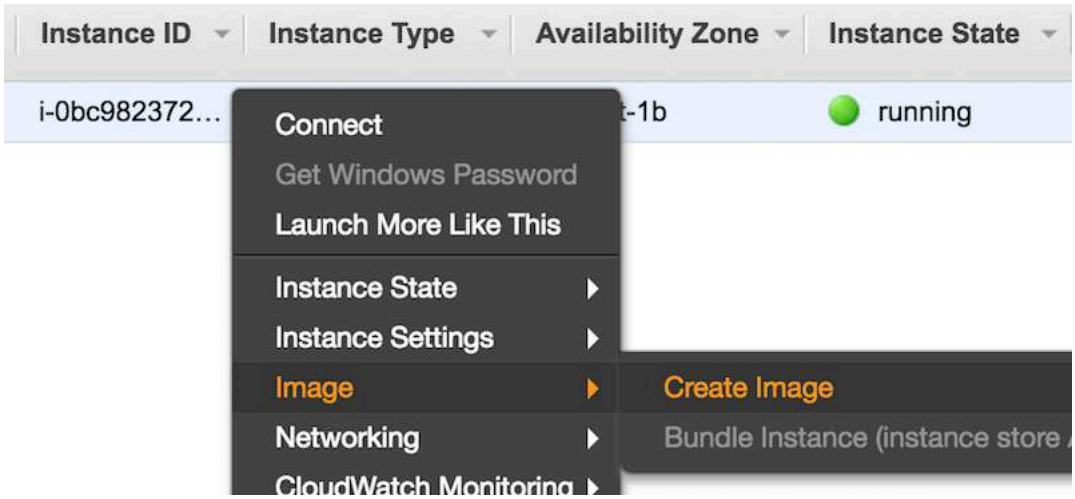
【注意】如果本地运行了 Jupyter notebook, 那么 8888 端口就可能被占用了。要么关掉本地 jupyter, 要么把端口映射改成别的。例如，假设 aws 使用默认 8888 端口，我们可以在本地启动 ssh 从实例映射到本地 8889 端口：

```
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

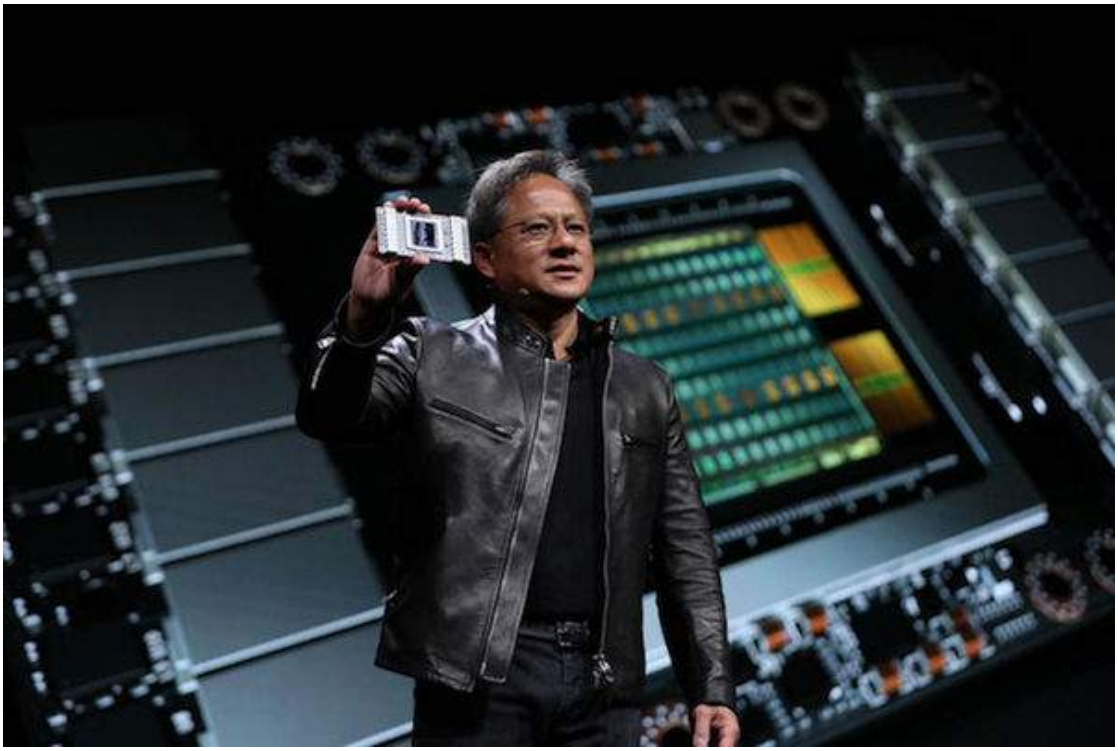
云上可以很方便的获取计算资源和配置环境

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 黄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。

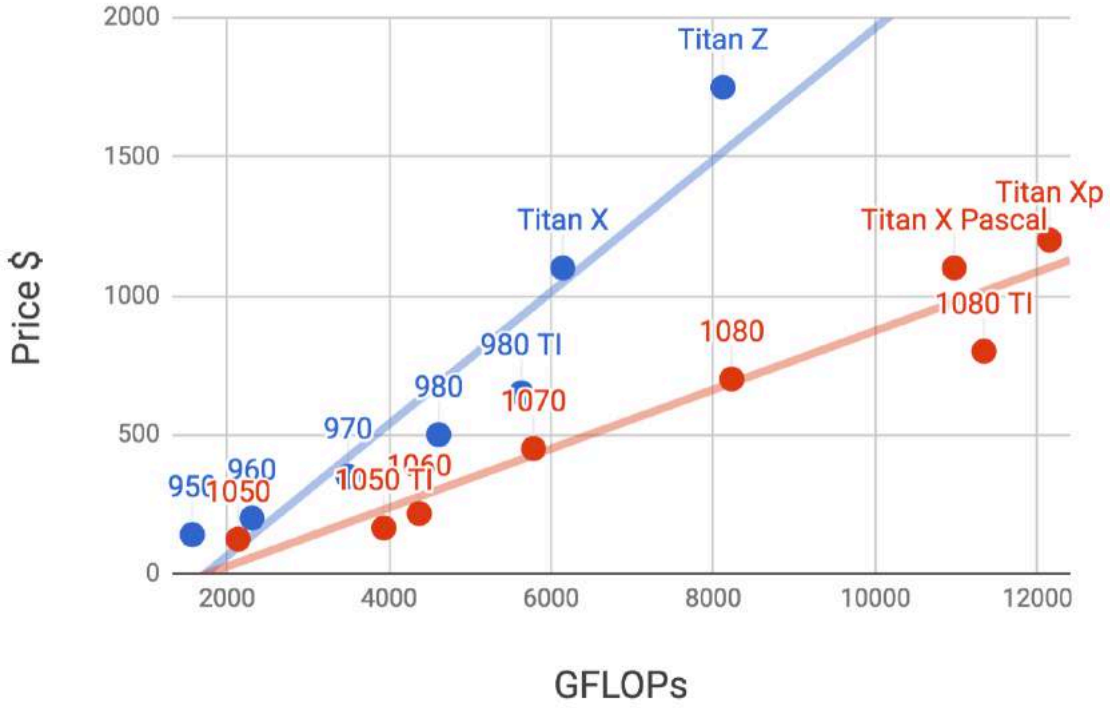
Nvidia 一般每一两年会更新一次大版本，例如目前最新的是 1000 系列。每个系列里面会有数个不同型号，对应不同的性能。

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

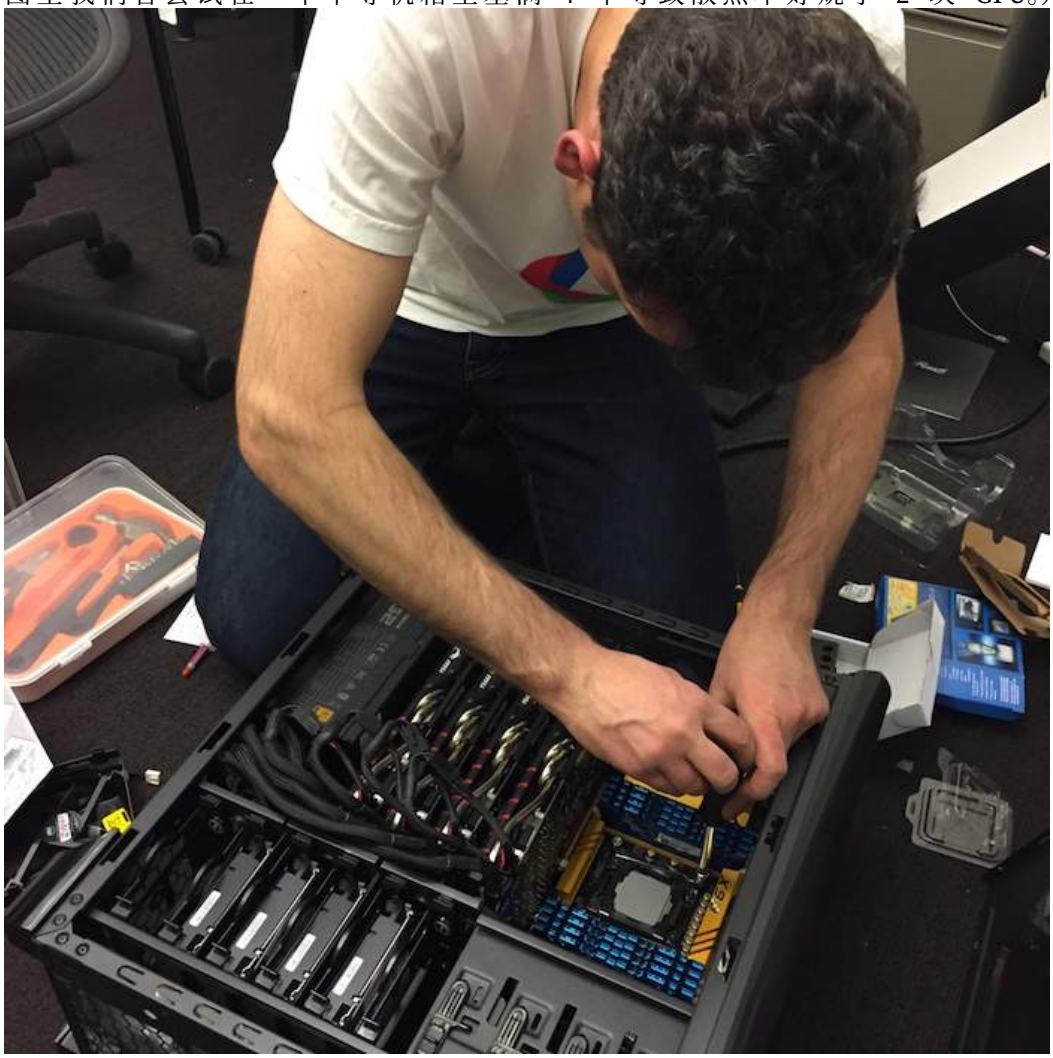
1. 买新不买旧，因为目前看来 GPU 性能还是在快速迭代，贬值较快。
2. 量力购买。不缺钱直接上最好的，但入门的 1050TI 也不错。

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。

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）