



动手学深度学习

MXNet Community

2018年07月08日

目录

1 引言	3
1.1 深度学习简介	3
1.2 如何使用	6
2 预备知识	11
2.1 安装和运行	11
2.2 数据操作	17
2.3 自动求梯度	25
2.4 查阅 MXNet 文档	28
3 深度学习基础	33
3.1 线性回归	33
3.2 线性回归的从零开始实现	39
3.3 线性回归的 Gluon 实现	44
3.4 Softmax 回归	48
3.5 Softmax 回归的从零开始实现	52
3.6 Softmax 回归的 Gluon 实现	59
3.7 多层感知机	62
3.8 多层感知机的从零开始实现	69
3.9 多层感知机的 Gluon 实现	72

3.10	欠拟合、过拟合和模型选择	73
3.11	权重衰减	81
3.12	权重衰减的 Gluon 实现	87
3.13	丢弃法	90
3.14	丢弃法的 Gluon 实现	95
3.15	正向传播和反向传播	97
3.16	实战 Kaggle 比赛：房价预测	101
4	深度学习计算	115
4.1	模型构造	115
4.2	模型参数的访问、初始化和共享	120
4.3	模型参数的延后初始化	125
4.4	自定义层	128
4.5	读取和存储	131
4.6	GPU 计算	134
5	卷积神经网络	139
5.1	二维卷积层	139
5.2	填充和步幅	144
5.3	多输入和输出通道	147
5.4	池化层	151
5.5	卷积神经网络（LeNet）	155
5.6	深度卷积神经网络（AlexNet）	159
5.7	使用重复元素的网络（VGG）	164
5.8	网络中的网络（NiN）	167
5.9	含并行连结的网络（GoogLeNet）	171
5.10	批量归一化	176
5.11	批量归一化的 Gluon 实现	181
5.12	残差网络（ResNet）	182
5.13	稠密连接网络（DenseNet）	187
6	循环神经网络	193
6.1	语言模型	194
6.2	隐藏状态	196
6.3	循环神经网络	198

6.4	通过时间反向传播	213
6.5	门控循环单元 (GRU)	217
6.6	长短期记忆 (LSTM)	223
6.7	深度循环神经网络	229
6.8	双向循环神经网络	231
6.9	循环神经网络的 Gluon 实现	233
7	优化算法	241
7.1	优化算法概述	241
7.2	梯度下降和随机梯度下降	246
7.3	梯度下降和随机梯度下降的 Gluon 实现	257
7.4	动量法	262
7.5	动量法的 Gluon 实现	269
7.6	Adagrad	273
7.7	Adagrad 的 Gluon 实现	277
7.8	RMSProp	279
7.9	RMSProp 的 Gluon 实现	284
7.10	Adadelta	286
7.11	Adadelta 的 Gluon 实现	290
7.12	Adam	292
7.13	Adam 的 Gluon 实现	297
8	计算性能	301
8.1	命令式和符号式混合编程	301
8.2	异步计算	309
8.3	自动并行计算	316
8.4	多 GPU 计算	319
8.5	多 GPU 计算的 Gluon 实现	325
9	计算机视觉	331
9.1	图片增广	331
9.2	微调	339
9.3	物体检测和边界框	344
9.4	物体检测数据集	347
9.5	锚框	350

9.6	单发多框检测 (SSD)	358
9.7	区域卷积神经网络 (R-CNN) 系列	370
9.8	语义分割和数据集	377
9.9	全卷积网络: FCN	382
9.10	样式迁移	389
9.11	实战 Kaggle 比赛: 图像分类 (CIFAR-10)	399
9.12	实战 Kaggle 比赛: 狗的品种识别 (ImageNet Dogs)	410
10	自然语言处理	421
10.1	词嵌入: word2vec	421
10.2	词嵌入: GloVe 和 fastText	429
10.3	求近似词和类比词	433
10.4	文本分类: 情感分析	440
10.5	编码器—解码器 (seq2seq)	447
10.6	束搜索	449
10.7	注意力机制	452
10.8	机器翻译	454
11	附录	465
11.1	数学基础	465
11.2	使用 Jupyter Notebook	472
11.3	使用 AWS 运行代码	479
11.4	GPU 购买指南	490
11.5	gluonbook 函数索引	493

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

引言

深度学习在近年来发展极为迅速。它在智能时代深刻改变着人类的生产生活方式。本章将简要介绍什么是深度学习，以及如何使用本书。

1.1 深度学习简介

1950 年，图灵在他的著名论文《计算机器与智能》中提出了“机器是否可以思考”的问题，并相信这是有可能的。在他所描述的图灵测试（Turing test）中，如果一个人在使用文本交互时较难区分机器与人类的回复，那么机器可以被认为是智能的。时至今日，为机器赋予智能的发展可谓日新月异。

1.1.1 人工智能

通俗地说，机器所展现出的智能叫做人工智能（artificial intelligence，简称 AI）。在上世纪 50 年代之后的岁月里，人工智能领域不断蓬勃发展。到了今天，人工智能在图像识别、语音识别和无

人驾驶等应用中已接近或达到人类智能水平，并在诸如棋类游戏的应用中超过了人类。举几个例子，

- 2016 年，谷歌 DeepMind 研发的围棋程序“阿尔法狗”（AlphaGo）在五轮围棋比赛中以 4 比 1 的成绩战胜了围棋世界冠军李世石。之后的升级版“阿尔法狗”以 3 比 0 的成绩战胜了排名第一的围棋世界冠军柯洁。
- 2017 年，来自中国的 Momenta 研发的视觉程序在 ImageNet 大型视觉识别挑战赛对 1000 类图像分类任务中取得 2.25% 的前五错误率。也就是说，对于 97.75% 的测试图片来说，视觉程序输出的最有可能的 5 个类别都包括正确答案。
- 2018 年，谷歌研发的助手程序可以通过和理发店店员在电话中对话来预定理发时间。而人类在这长达近一分钟的语音交互中完全没有意识到电话的另一头是机器。

那么，人工智能究竟该如何实现呢？在人工智能发展的初期，许多人认为人类可以通过在计算机程序里设定足够多的规则使机器具备智能。这也促成了专家系统（expert system）的诞生。专家系统通过知识库与推理机来解决更复杂的问题。其中，知识库囊括了大量人类指定的事实与规则，而推理机将规则应用于知识库中已有的事实，从而推断出新的事实。作为人工智能的分支，专家系统的发展在上世纪 80 年代达到鼎盛。

虽然专家系统对于涉及复杂推理的任务非常有效，但它却很难解决例如图像识别和语音识别这样对人类较简单的问题。人类还需要另一种通往人工智能的手段：机器学习（machine learning）。

1.1.2 机器学习

机器学习研究如何使计算机系统利用经验改善性能。通常，这些经验以数据的形式存在。对于一个任务，我们希望计算机从数据中学习根据输入得到输出的规则，而无需人类明确指定这样的规则。这里的输入可以是一张含有猫或狗的图片，输出可以是“猫”或“狗”。因此，我们也可以把机器学习看作是基于数据的编程范式。在机器学习中，计算机从数据中学习到的将输入变换为输出的规则也叫模型（model）。随着数据规模的不断增大和硬件性能的不断提高，机器学习逐步发展成为人工智能领域最有吸引力的分支。如今，无论是用户接收个性化的商品或新闻推荐，还是电子邮件服务自动拦截垃圾邮件，机器学习几乎无处不在。

然而，传统的机器学习方法也有一定的局限性，尤其当输入是像图像和声音那样的原始数据时。我们已经提到，机器学习模型将输入变换为输出。如果我们能够以更好的方式表示这些输入数据，模型将更容易将它们变换为正确的输出。多年以来，人们先依靠有关具体任务的专业知识，仔细将原始数据转换成合适的表示，再使用机器学习方法将这些表示变换为输出。

实际上，在机器学习的众多研究方向中，表征学习（representation learning）关注如何自动找出表示数据的合适方式，以便更好地将输入变换为正确的输出。而本书所要重点探讨的深度学习（deep learning）正是表征学习的一类方法。

1.1.3 深度学习

深度学习是具有多级表示的表征学习方法。在每一级（从原始数据开始），深度学习通过简单的函数将该级的表示变换为更高级的表示。因此，深度学习模型也可以看作是由许多简单函数复合而成的函数。当这些复合的函数足够多时，深度学习模型可以表达非常复杂的变换。

深度学习可以逐级表示越来越抽象的概念或模式。以图片为例，它的输入是一堆原始像素值。深度学习模型中，图片的第一级的表示通常是在特定的位置和角度是否出现边缘。而第二级的表示通常能够将这些边缘组合出有趣的模式，例如花纹。在第三级的表示中，也许上一级的花纹能进一步汇合成对应物体特定部位的模式。这样逐级表示下去，最终，模型能够较容易根据最后一级的表示完成给定的任务，例如图片分类。值得一提的是，作为表征学习的一种，深度学习将自动找出每一级表示数据的合适方式。

深度学习通常基于神经网络（neural network）模型。神经网络模型中每一层的输出对应着深度学习每一级的表示。而各种神经网络模型在上世纪就已经被提出了。然而，深度学习直到最近几年才在图像识别、语音识别、游戏等应用中取得一系列激动人心的成就。这主要得益于近年来硬件的发展和数据的增长，例如通用 GPU 的出现。此外，开源深度学习库也迅速使得深度学习技术在学界和业界得到广泛应用。早期的深度学习从业人员需要精通 CUDA 和 C++。而在本书的学习中，你会发现，只需要懂得基础的 Python 编程就可以使用深度学习。

总之，深度学习作为机器学习的一类方法，是实现人工智能的一种重要途径。其实，本节开头所描述的“阿尔法狗”、错误率极低的图像分类以及像人一样打电话的助手程序都应用了深度学习。在未来几年，我们相信深度学习领域依然充满机遇，充满挑战。

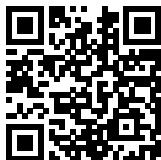
1.1.4 小结

- 机器所展现出的智能叫做人工智能。
- 机器学习研究如何使计算机系统利用经验改善性能。它是人工智能领域的分支，也是实现人工智能的一种手段。
- 作为机器学习的一类，表征学习关注如何自动找出表示数据的合适方式。
- 深度学习是具有多级表示的表征学习方法。它可以逐级表示越来越抽象的概念或模式。

1.1.5 练习

- 如果把人工智能的发展看作是新一次工业革命。那么深度学习和数据的关系是否像是蒸汽机与煤炭的关系呢？为什么？
- 你生活的哪些方面可能被深度学习所改变？

1.1.6 扫码直达讨论区



1.2 如何使用

在本书中，我们不仅阐述深度学习的技术与应用，还使用 Apache MXNet (incubating) 及其最新的 Gluon 接口演示如何实现它们。除了视频课程和讨论区以外，我们将利用 Jupyter notebook 能将文字、代码、公式和图像统一起来的优势，提供一个交互式的学习体验。据我们所知，目前还没有如此多方位交互式深度学习的学习体验。我们将尝试弥补这个空白。

1.2.1 面向的读者

本书总体上面向大学生、工程师和研究人员。本书对于深度学习技术与应用的阐述涉及了数学和编程。如果你希望理解深度学习背后的数学原理，需要了解基础的线性代数、微分和概率。如果你希望读懂书中的代码，需要了解基础的 Python 编程。如果你只对书中的数学部分或编程部分感兴趣，也可以忽略另一部分。

附录提供了本书所涉及的数学知识，供有需要的读者选择参考。如果你对 Python 编程不熟悉，我们建议你先学习一些有关 Python 编程的教程，例如中文教程 <http://www.runoob.com/python/python-tutorial.html> 或英文教程 <http://learnpython.org/>。

1.2.2 内容和结构

本书内容大体可以分为三部分：

- 第一部分（第 1 章至第 3 章）涵盖预备工作和基础知识。第 1 章介绍了深度学习的背景和本书的使用。第 2 章提供了动手学深度学习所需要的预备知识，例如如何获取并运行书中的代码。第 3 章包括了深度学习最基础的概念和技术，例如多层感知机和模型正则化。如果你时间有限，并且只希望了解深度学习最基础的概念和技术，那么你只需阅读第一部分。
- 第二部分（第 4 章至第 6 章）关注现代深度学习技术。第 4 章描述了深度学习计算的各个重要组成部分，并为之后实现更复杂的模型打下基础。第 5 章解释了近年来令深度学习在计算机视觉领域大获成功的卷积神经网络。第 6 章阐述了近年来常用于处理序列数据的循环神经网络。阅读第二部分有助于掌握现代深度学习技术。
- 第三部分（第 7 章至第 10 章）讨论计算性能和应用。第 7 章评价了各种用来训练深度学习模型的优化算法。第 8 章检验了影响深度学习计算性能的几个重要因素。第 9 章和第 10 章分别列举了深度学习在计算机视觉和自然语言处理中的重要应用。这部分内容可供你根据兴趣选择阅读。

图 1.1 描绘了本书的结构。

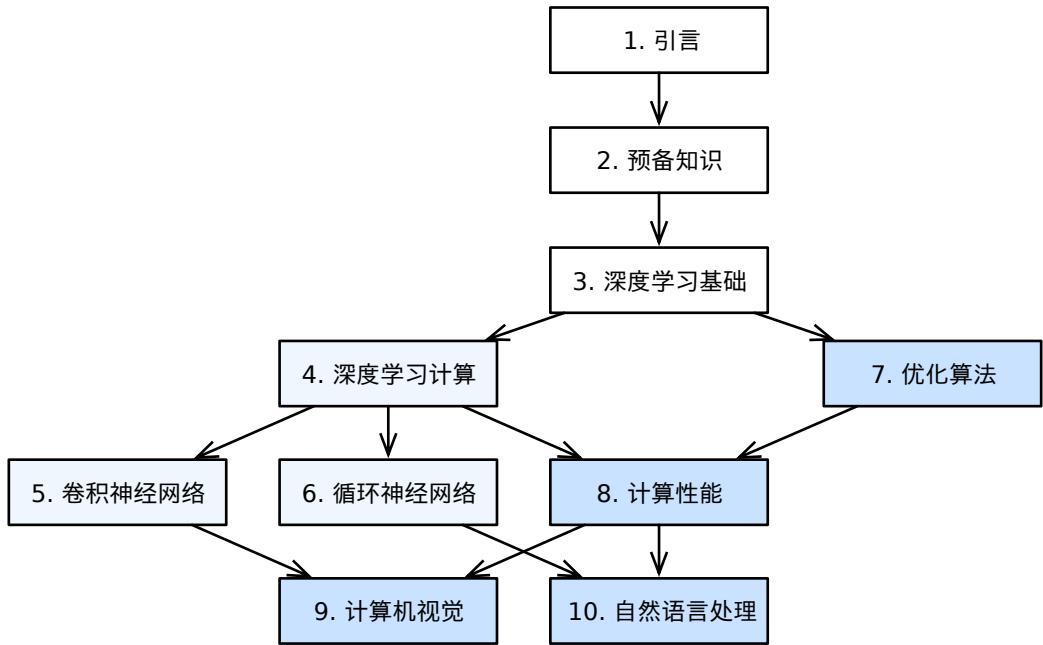


图 1.1: 本书的结构。由甲章指向乙章的箭头表明甲章的知识有助于理解乙章的内容。如果你想短时间了解深度学习最基础的概念和技术，只需阅读第 1 章至第 3 章；如果你希望掌握现代深度学习技术，还需阅读第 4 章至第 6 章。第 7 章至第 10 章可供你根据兴趣选择阅读。

1.2.3 代码

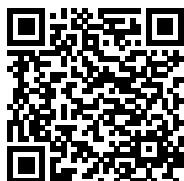
作为一个开源的深度学习框架，Apache MXNet 提供了易用的 Gluon 接口，并有着出众的计算性能。MXNet 支持例如 C++、JavaScript、Python、R、Matlab、Julia、Scala、Go 等前端编程语言，并被众多使用者部署到了手机、树莓派（Raspberry Pi）、笔记本电脑等不同的设备之上。我们选择 MXNet 作为本书使用的深度学习框架。它也是 AWS（亚马逊的云计算服务）首选的深度学习框架。

本书所有代码已在 MXNet 1.2.0 和 Python 3.6.4 下测试通过。由于深度学习发展极为迅速，未来版本的 MXNet 可能会造成书中部分代码无法正常运行。如果遇到某一节代码无法正常运行的情况，可安装 1.2.0 版的 MXNet 或在该节的讨论区汇报问题。如果你希望安装未来新版本的 MXNet 并获得该版本下测试通过的新代码，可参考“[安装和运行](#)”一节更新代码和运行环境。

1.2.4 视频课程

本书作者曾在网上直播教学《动手学深度学习》系列课程。这套课程的视频录像可在 Bilibili 网站或 Youtube 网站上收看。

- 扫码直达Bilibili 网站的视频教程：



- 扫码直达Youtube 网站的视频教程：



本书基于观众反馈意见对视频课程内容做了部分修改。如果视频与本书有内容不一致，请以本书内容为准。

1.2.5 讨论区

本书的学习社区地址是 <https://discuss.gluon.ai/>。当你对书中某节内容有疑惑时，也可以扫一扫该节后面的二维码参与讨论该节内容。值得一提的是，在有关 Kaggle 比赛章节的讨论区中，众多社区成员提供了丰富的高水平方法。我们强烈推荐大家积极参与学习社区中的讨论，并相信你一定会有所收获。本书作者和 MXNet 开发人员也时常参与社区中的讨论。

1.2.6 小结

- 我们选择 MXNet 作为本书使用的深度学习框架。
- 本书力求提供一个全方位交互式的深度学习的学习体验。

1.2.7 练习

- 在本书的学习社区 <https://discuss.gluon.ai/> 上注册一个账号。搜索关键字 Kaggle，浏览其中回复量最大的几个帖子。
- 浏览《动手学深度学习》视频课程列表，比较与本书章节编排顺序的异同。

1.2.8 扫码直达讨论区



预备知识

在动手学习之前，我们需要获取本书代码，并安装运行本书代码所需要的工具。作为动手学深度学习的基础，我们还需要了解如何对内存中的数据进行操作，以及对函数求梯度的方法。

2.1 安装和运行

为了动手学深度学习，我们需要获取本书代码，安装并运行 Python、MXNet、Jupyter notebook 等工具。

2.1.1 进入命令行模式

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：Windows 用户可以在文件资源管理器的地址栏输入 cmd 并按回车键，Linux/macOS 用户可以打开 Terminal 应用。

2.1.2 获取代码并安装运行环境

我们可以通过 Conda 来获取本书代码并安装运行环境。Windows 和 Linux/macOS 用户请分别参考以下步骤。

Windows 用户

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。我们可以在浏览器的地址栏中输入以下地址并按回车键进行下载：

```
https://zh.gluon.ai/gluon_tutorials_zh-1.0.zip
```

下载完成后，解压该文件并进入文件夹“gluon_tutorials_zh-1.0”。在该目录文件资源管理器的地址栏输入 cmd 进入命令行模式。

第三步，安装运行所需的依赖包并激活该运行环境。我们可以先通过运行下面命令来配置下载源，从而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/
```

然后运行以下命令安装并激活运行环境。

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

第四步，打开 Jupyter notebook。运行下面命令。

```
jupyter notebook
```

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

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

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

Linux/macOS 用户

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

安装时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 /xx/yyyy.zzzz? [yes|no]
[no] >>> yes
```

安装完成后，我们需要让 conda 生效。Linux 用户需要运行一次 `source ~/.bashrc` 或重启命令行；macOS 用户需要运行一次 `source ~/.bash_profile` 或重启命令行。

第二步，下载包含本书全部代码的包，解压后进入文件夹。运行如下命令。

```
mkdir gluon_tutorials_zh-1.0 && cd gluon_tutorials_zh-1.0
curl https://zh.gluon.ai/gluon_tutorials_zh-1.0.tar.gz -o tutorials.tar.gz
tar -xzvf tutorials.tar.gz && rm tutorials.tar.gz
```

第三步，安装运行所需的依赖包并激活该运行环境。我们可以先通过运行下面命令来配置下载源，从而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/
```

然后运行以下命令安装并激活运行环境。

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

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

第四步, 打开 Jupyter notebook。运行下面命令。

```
jupyter notebook
```

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

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

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

2.1.3 激活运行环境

运行环境中已安装了运行书中代码所需的 Python、MXNet、Jupyter notebook 等工具。我们可以在下载并解压的代码包里的文件 “`gluon_tutorials_zh-1.0/environment.yml`” 中查看它们。在运行书中代码前, 我们需要激活运行环境。Windows 和 Linux/macOS 用户请分别参照以下步骤激活并退出运行环境。

Windows 用户

首先进入之前解压得到的文件夹 “`gluon_tutorials_zh-1.0`”。然后在该目录文件资源管理器的地址栏输入 `cmd` 进入命令行模式。最后运行以下命令激活安装环境。

```
activate gluon
```

如需退出激活环境, 运行以下命令。

```
deactivate
```

Linux/macOS 用户

首先在命令行模式下进入之前解压得到的文件夹 “`gluon_tutorials_zh-1.0`” (例如运行 `cd gluon_tutorials_zh-1.0`), 然后运行以下命令激活安装环境。

```
source activate gluon
```

如需退出激活环境，运行以下命令。

```
source deactivate
```

2.1.4 更新代码和运行环境

为了适应深度学习和 MXNet 的快速发展，本书的开源内容将定期发布新版本。我们推荐大家定期更新本书的开源内容（例如代码）和相应的运行环境（例如新版 MXNet）。以下是更新的具体步骤。

第一步，重新下载最新的包含本书全部代码的包。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

解压后进入文件夹“gluon_tutorials_zh”。

第二步，使用下面命令更新运行环境。

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

2.1.5 使用 GPU 版的 MXNet

通过前面介绍的方式安装的 MXNet 只支持 CPU 计算。本书中有部分章节需要或推荐使用 GPU 来运行。如果你的电脑上有 Nvidia 显卡并安装了 CUDA，建议使用 GPU 版的 MXNet。

我们在完成获取代码并安装运行环境的步骤后，需要先激活运行环境。然后卸载 CPU 版本的 MXNet：

```
pip uninstall mxnet
```

接下来，退出运行环境。使用文本编辑器打开之前解压得到的代码包里的文件“gluon_tutorials_zh-1.0/environment.yml”。如果电脑上装的是 8.0 版本的 CUDA，将该文件中的字符串“mxnet”改为“mxnet-cu80”。如果电脑上安装了其他版本的 CUDA（比如 7.5、9.0、9.2 等），对该文件中

的字符串“mxnet”做类似修改（比如改为“mxnet-cu75”、“mxnet-cu90”、“mxnet-cu92”等）。然后，使用下面命令更新运行环境。

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

之后，我们只需要再激活安装环境就可以使用 GPU 版的 MXNet 运行书中代码了。

更新代码和运行环境

如果使用 GPU 版的 MXNet，更新代码和运行环境可参照以下步骤：

第一步，重新下载最新的包含本书全部代码的包。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

解压后进入文件夹“gluon_tutorials_zh”。

第二步，使用文本编辑器打开文件夹“gluon_tutorials_zh”中的环境配置文件“environment.yml”。如果电脑上装的是 8.0 版本的 CUDA，将该文件中的字符串“mxnet”改为“mxnet-cu80”。如果电脑上安装了其他版本的 CUDA（比如 7.5、9.0、9.2 等），对该文件中的字符串“mxnet”做类似修改（比如改为“mxnet-cu75”、“mxnet-cu90”、“mxnet-cu92”等）。

第三步，使用下面命令更新运行环境。

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

2.1.6 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。

2.1.7 练习

- 获取本书代码并安装运行环境。如果你在安装时遇到任何问题，请扫一扫本节二维码。在讨论区，你可以查阅疑难问题汇总或者提问。

2.1.8 扫码直达讨论区



2.2 数据操作

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

在 MXNet 中，NDArray 是存储和变换数据的主要工具。如果你之前用过 NumPy，你会发现 NDArray 和 NumPy 的多维数组非常类似。然而，NDArray 提供诸如 GPU 计算和自动求导在内的更多功能。这些都使得 NDArray 更加适合深度学习。

2.2.1 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，可以参阅附录中“数学基础”一节。

首先从 MXNet 导入 `ndarray` 模块。这里的 `nd` 是 `ndarray` 的缩写形式。

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

然后我们用 `arange` 函数创建一个行向量。

```
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)>
```

其返回一个 NDArray 实例，里面一共包含从 0 开始的 12 个连续整数。从打印 `x` 时显示的属性 `<NDArray 12 @cpu(0)>` 可以看到，它是长度为 12 的一维数组，且被创建在 CPU 主内存上，CPU 里面的 0 没有特别的意义。

我们可以通过 `shape` 属性来获取 NDArray 实例形状。

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

```
Out[3]: (12,)
```

我们也能够通过 `size` 属性得到 NDArray 实例中元素 (element) 的总数。

```
In [4]: x.size
```

```
Out[4]: 12
```

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

```
Out[5]:
```

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

注意 `x` 属性中的形状发生了变化。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度的大小推断出来的。

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

```
Out[6]:
```

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

```
Out[7]:
```

```
[[ 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 [8]: y = nd.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])  
y  
  
Out[8]:  
[[ 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 [9]: nd.random.normal(0, 1, shape=(3, 4))  
  
Out[9]:  
[[ 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)>
```

2.2.2 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)>
```

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)>
```

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)>
```

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)>
```

除去按元素计算外，我们可以做矩阵运算。下面将 x 与 y 的转置做矩阵乘法。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 [14]: nd.dot(x, y.T)
```

```
Out[14]:
```

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

我们也可以将多个 NDArray 合并。下面分别在行上（维度 0，即形状中的最左边元素）和列上（维度 1，即形状中左起第二个元素）连结两个矩阵。

```
In [15]: nd.concat(x, y, dim=0), nd.concat(x, y, dim=1)
```

```
Out[15]: (
```

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

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

使用条件判断式可以得到元素为 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 [16]: x == y
```

```
Out[16]:
```

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

对 NDArray 中的所有元素求和得到只有一个元素的 NDArray。

```
In [17]: x.sum()
```

```
Out[17]:
```

```
[ 66.]  
<NDArray 1 @cpu(0)>
```

我们可以通过 `asscalar` 函数将结果变换为 Python 中的标量。下面例子中 x 的 L_2 范数结果同上一样是单元素 NDArray，但最后结果是 Python 中标量。

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

```
Out[18]: 22.494444
```

我们也可以把 $y.exp()$ 、 $x.sum()$ 、 $x.norm()$ 等分别改写为 `nd.exp(y)`、`nd.sum(x)`、`nd.norm(x)` 等。

2.2.3 广播机制

前面我们看到如何对两个形状相同的 NDArray 做按元素操作。当对两个形状不同的 NDArray 做同样操作时，可能会触发广播（broadcasting）机制：先适当复制元素使得这两个 NDArray 形状相同后再按元素操作。

定义两个 NDArray：

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

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

由于 `a` 和 `b` 分别是 3 行 1 列和 1 行 2 列的矩阵，如果要计算 `a+b`，那么 `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。

```
In [20]: a + b
```

```
Out[20]:
```

```
[[ 0.  1.]
 [ 1.  2.]
 [ 2.  3.]]
<NDArray 3x2 @cpu(0)>
```

2.2.4 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 [21]: x[1:3]
```

```
Out[21]:
```

```
[[ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
<NDArray 2x4 @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 [22]: x[1, 2] = 9
         x
```

```
Out[22]:
```

```
[[ 0.  1.  2.  3.]
 [ 4.  5.  9.  7.]]
```

```
[ 8.  9. 10. 11.]
<NDArray 3x4 @cpu(0)>
```

当然，我们也可以截取一部分元素，并重设它们的值。

```
In [23]: x[1:2, 1:3] = 12
```

```
x
```

```
Out[23]:
```

```
[[ 0.  1.  2.  3.]
 [ 4. 12. 12.  7.]
 [ 8.  9. 10. 11.]]
<NDArray 3x4 @cpu(0)>
```

2.2.5 运算的内存开销

前面例子里我们对每个操作新开内存来储存它的结果。例如即使是 $y = x + y$ 我们也会新创建内存，然后再将 y 指向新内存。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，那么它们所对应的内存地址相同；反之则不同。

```
In [24]: before = id(y)
y = y + x
id(y) == before
```

```
Out[24]: False
```

如果我们想指定结果到特定内存，我们可以使用前面介绍的索引来进替换操作。在下面的例子中，我们先通过 `zeros_like` 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 [25]: z = y.zeros_like()
before = id(z)
z[:] = x + y
id(z) == before
```

```
Out[25]: True
```

注意到这里我们还是为 $x + y$ 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。如果想避免这个内存开销，我们可以使用运算符的全名函数中的 `out` 参数。

```
In [26]: nd.elemwise_add(x, y, out=z)
id(z) == before
```

```
Out[26]: 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 [27]: before = id(x)
x += y
id(x) == before

Out[27]: True
```

2.2.6 NDArray 和 NumPy 相互变换

我们可以通过 `array` 和 `asnumpy` 函数令数据在 NDArray 和 Numpy 格式之间相互转换。下面将 NumPy 实例变换成 NDArray 实例。

```
In [28]: import numpy as np
```

```
p = np.ones((2, 3))
d = nd.array(x)
d

Out[28]:
[[ 2.  3.  8.  9.]
 [ 9. 26. 27. 18.]
 [20. 21. 22. 23.]]
<NDArray 3x4 @cpu(0)>
```

再将 NDArray 实例变换成 NumPy 实例。

```
In [29]: d.asnumpy()

Out[29]: array([[ 2.,  3.,  8.,  9.],
 [ 9., 26., 27., 18.],
 [20., 21., 22., 23.]], dtype=float32)
```

2.2.7 小结

- NDArray 是 MXNet 中存储和变换数据的主要工具。
- 我们可以轻松地对 NDArray 进行创建、运算、指定索引和与 NumPy 之间的相互变换。

2.2.8 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?

2.2.9 扫码直达讨论区



2.3 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, 可以参阅附录中“[数学基础](#)”一节。本小节将介绍如何使用 MXNet 提供的 `autograd` 包来自动求梯度。

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

2.3.1 简单例子

我们先看一个简单例子: 对函数 $y = 2x^\top x$ 求关于列向量 x 的梯度。我们先创建变量 x , 并赋初值。

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

Out[2]:

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

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

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

下面定义有关变量 x 的函数。为了减少计算和内存开销，默认条件下 MXNet 不会记录用于求梯度的计算。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 将默认先对 y 中元素求和得到新的变量，再求该变量有关 x 的梯度。

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

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

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

```
Out[6]: ([[[ 0.]
   [ 4.]
   [ 8.]
   [12.]])
<NDArray 4x1 @cpu(0)>,
[[ 1.]
 [ 1.]
 [ 1.]
 [ 1.]])
<NDArray 4x1 @cpu(0)>)
```

2.3.2 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。

```
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，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)>
```

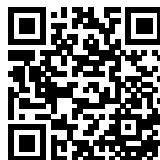
2.3.3 小结

- 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。

2.3.4 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。

2.3.5 扫码直达讨论区



2.4 查阅 MXNet 文档

受篇幅所限，本书无法对所有 MXNet 的应用程序接口（API）一一介绍。每当遇到不熟悉的 MXNet API 时，我们可以主动查阅它的相关文档。

2.4.1 使用 `dir` 函数

我们可以使用 `dir` 函数查阅 API 中包含的成员或属性。打印 `autograd` 模块中所有的成员或属性。

```
In [1]: from mxnet import autograd
print(dir(autograd))

['CFUNCTYPE', 'Function', 'Lock', 'MXCallbackList', 'NDArray', 'NDArrayHandle',
 → 'POINTER', 'Symbol', 'SymbolHandle', '_GRAD_REQ_MAP', '_LIB',
 → '_RecordingStateScope', '__builtins__', '__cached__', '__doc__', '__file__',
 → '__loader__', '__name__', '__package__', '__spec__', 'ndarray_cls',
 → '_parse_head', 'absolute_import', 'array', 'backward', 'c_array', 'c_array_buf',
 → 'c_handle_array', 'c_int', 'c_void_p', 'cast', 'check_call', 'ctypes',
 → 'division', 'get_symbol', 'grad', 'is_recording', 'is_training',
 → 'mark_variables', 'mx_uint', 'pause', 'predict_mode', 'record', 'set_recording',
 → 'set_training', 'string_types', 'traceback', 'train_mode']
```

2.4.2 使用 `help` 函数

当我们想了解 API 的具体用法时，可以使用 `help` 函数。让我们以 `NDArray` 中的 `ones_like` 函数为例，查阅它的用法。

```
In [2]: from mxnet import nd
help(nd.ones_like) # 在 Jupyter notebook 中可使用问号查询，例如 nd.ones_like?
```

Help on function ones_like:

```
ones_like(data=None, out=None, name=None, **kwargs)
    Return an array of ones with the same shape and type
    as the input array.
```

Examples::

```
x = [[ 0.,  0.,  0.],
      [ 0.,  0.,  0.]]
ones_like(x) = [[ 1.,  1.,  1.],
                 [ 1.,  1.,  1.]]
```

Parameters

data : NDArray
The input

out : NDArray, optional

The output NDArray to hold the result.

Returns

out : NDArray or list of NDArrays
The output of this function.

从文档信息我们了解到，`ones_like` 函数会创建和输入 `NDArray` 形状相同且元素为 1 的新的 `NDArray`。我们可以验证一下：

```
In [3]: x = nd.array([[0,0,0], [2,2,2]])
y = x.ones_like()
y
```

Out[3]:

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
<NDArray 2x3 @cpu(0)>
```

2.4.3 在 MXNet 网站上查阅

我们也可以在 MXNet 的网站上查阅 API 的相关文档。访问 MXNet 网站 (mxnet.apache.org)。如图 2.1 所示，点击网页顶部的下拉菜单“API”可查阅各个前端语言的 API。此外，我们也可以在网页右上方含“Search”字样的搜索框中直接搜索 API 名称。

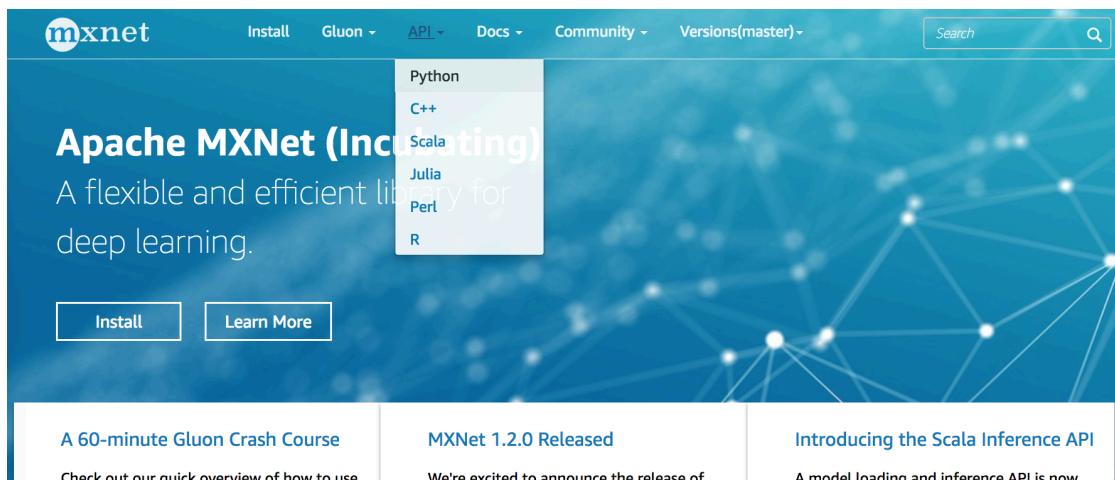


图 2.1: MXNet 官方网站 (mxnet.apache.org)。点击顶部的下拉菜单“API”可查阅各个前端语言的 API。在右上方含“Search”字样的搜索框中也可直接搜索 API 名称。

图 2.2 展示了 MXNet 网站上有关 `ones_like` 函数的文档。

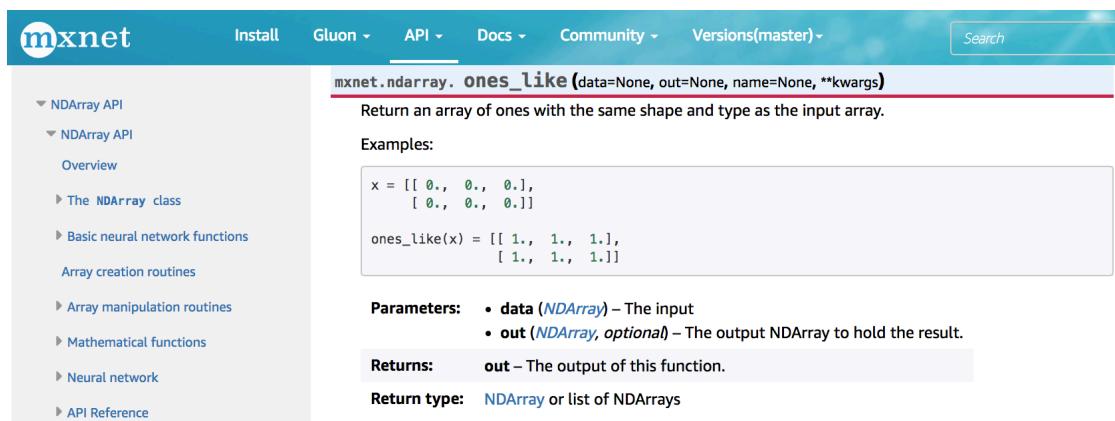


图 2.2: MXNet 网站上有关 `ones_like` 函数的文档。

2.4.4 小结

- 每当遇到不熟悉的 MXNet API 时，我们可以主动查阅它的相关文档。
- 查阅 MXNet 文档可以使用 `dir` 和 `help` 函数，或访问 MXNet 官网。

2.4.5 练习

- 查阅 NDArray 支持的其他操作。

2.4.6 扫码直达讨论区



深度学习基础

从本章开始，我们将探索深度学习的奥秘。作为机器学习的一类，深度学习通常基于神经网络模型逐级表示越来越抽象的概念或模式。我们先从线性回归和 Softmax 回归这两种单层神经网络入手，简要介绍机器学习中的基本概念。然后，我们由单层神经网络延伸到多层神经网络，并通过多层感知机引入深度学习模型。在观察和了解了模型的过拟合现象后，我们将介绍深度学习中应对过拟合的常用方法：权重衰减和丢弃法。接着，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，我们通过一个深度学习应用案例来实践本章内容。

3.1 线性回归

在本章的前几节，我们先介绍单层神经网络：线性回归和 Softmax 回归。其中，线性回归适用于回归问题：模型的最终输出是一个连续值。回归问题在实际中很常见，例如预测房屋价格、气温、销售额等连续值的问题。与回归问题不同，分类问题中模型的最终输出是一个离散值。我们所说的图像分类、垃圾邮件识别、疾病检测等输出为离散值的问题都属于分类问题的范畴。而 Softmax 回归正适用于此类问题。

由于线性回归和 Softmax 回归都是单层神经网络，它们涉及到的概念和技术同样适用于大多数的深度学习模型。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，我们先从具体的房屋价格预测的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 $\mathbf{w} = [w_1, w_2]^\top$ 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就是一个数据样本。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

在线性回归这个简单例子里，令损失函数对模型参数求导后的结果为零可以解出最小化损失函数的模型参数。这类解叫做解析解 (analytical solution)。然而，大多数深度学习模型并没有解析解，只能通过优化算法有限次迭代模型参数来尽可能降低损失函数的值。这类解叫做数值解 (numerical solution)。在求数值解的优化算法中，小批量随机梯度下降 (mini-batch stochastic gradient descent) 被广泛使用。它的算法很简单：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所组成的小批量 (mini-batch) \mathcal{B} ；然后求小批量中数据样本的平均损失有关模型参数的导数 (梯度)；再用此结果与预先设定的一个正数的乘积作为模型参数在本次迭代的减小量。在训练本节讨论的线性回归模型的过程中，模型的每个参数将迭代如下：

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

在上式中， $|\mathcal{B}|$ 代表每个小批量中的样本个数 (批量大小，batch size)， 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，并不是通过模型训练学出的，因此叫做超参数 (hyperparameter)。我们通常所说的“调参”指的正是调节超参数，例如通过反复试错来找到合适的超参数。少数情况下，超参数也可以通过模型训练学出。本书对此类情况不做讨论。

我们将在之后的“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。

训练和预测

当训练模型时，我们使用优化算法迭代模型参数若干次。完成以后，将模型参数 w_1, w_2, b 在训练结束时的值分别记作 $\hat{w}_1, \hat{w}_2, \hat{b}$ 。这时，我们就可以使用学出的线性回归模型 $x_1 \hat{w}_1 + x_2 \hat{w}_2 + \hat{b}$ 来

估算训练数据集以外任意一栋面积（平方米）为 x_1 、房龄（年）为 x_2 的房屋的价格了。这里的估算也叫做模型预测、模型测试或模型推断。

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。神经网络图隐去了模型参数权重和偏差。

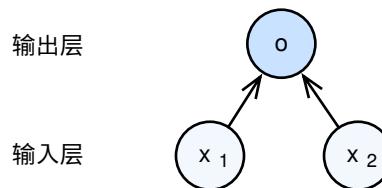


图 3.1：线性回归是一个单层神经网络。

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

矢量计算表达式

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

下面先定义两个 1000 维的向量。

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

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

向量相加的一种方法是，将这两个向量按元素逐一做标量加法：

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

Out[2]: 0.09406399726867676
```

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]: 0.00027179718017578125
```

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

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

对 3 个房屋样本估算价格的矢量计算表达式为 $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + \mathbf{b}$ ，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

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

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

同理，我们也可以在模型训练中对优化算法做矢量计算。设模型参数 $\boldsymbol{\theta} = [w_1, w_2, b]^\top$ ，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 线性回归的从零开始实现

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。

首先，导入本节中实验所需的包或模块。

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

3.2.1 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。使用人工训练数据集将使我们能够比较学到的参数和真实的模型参数。设训练数据集样本数为 1000，输入个数为 2。给定随机生成的批量样本特征 $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$ ，我们使用线性回归模型真实权重 $\mathbf{w} = [2, -3.4]^\top$ 和偏差 $b = 4.2$ ，以及一个随机噪音项 ϵ 来生成标签

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

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)
```

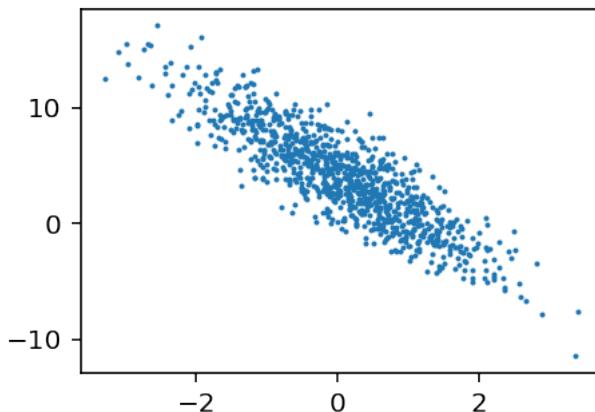
注意到 `features` 的每一行是一个长度为 2 的向量，而 `labels` 的每一行是一个长度为 1 的向量（标量）。

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

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

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

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



我们将上面的 `plt` 作图函数定义在 `gluonbook` 包里。以后在作图时，我们将直接调用 `gluonbook.plt`，而无需执行其他设置项。

3.2.2 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`（批量大小）个随机样本的特征和标签。

```
In [5]: batch_size = 10
def data_iter(batch_size, features, labels):
```

```

num_examples = len(features)
indices = list(range(num_examples))
# 样本的读取顺序是随机的。
random.shuffle(indices)
for i in range(0, num_examples, batch_size):
    j = nd.array(indices[i: min(i + batch_size, num_examples)])
    # take 函数根据索引返回对应元素。
    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)，分别对应批量大小和输入个数；标签形状为批量大小。

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

```

[[ -0.42438623 -0.20454668]
 [ 0.59524769 -2.19519234]
 [ 0.42910433  2.08101654]
 [-0.33643323 -1.0666517 ]
 [-1.04062724 -0.5304721 ]
 [-0.9840098 -0.1045013 ]
 [-0.85182911 -1.70157921]
 [-1.86081767  1.38656259]
 [ 0.229029   0.76287299]
 [-0.32117304 -1.17399085]]
<NDArray 10x2 @cpu(0)>
[ 4.05559874 12.8497858 -2.0345757  7.14670324  3.92091894
 2.59357953  8.2966938 -4.23733759  2.07375264  7.55321264]
<NDArray 10 @cpu(0)>
```

3.2.3 初始化模型参数

我们将权重初始化成均值 0 标准差为 0.01 的正态随机数，偏差则初始化成 0。

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

之后训练时我们需要对这些参数求梯度来迭代它们的值，因此我们需要创建它们的梯度。

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

3.2.4 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 `dot` 函数做矩阵乘法。

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

3.2.5 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
```

3.2.6 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。它通过不断更新模型参数来优化损失函数。

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

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

3.2.7 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（假设样本数能够被批量大小整除）。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，大多超参数都需要通过反复试错来不断调节。当迭代周期数设的越大时，虽然模型可能更有效，但是训练时间可能过长。而有关学习率对模型的影响，我们会在后面“优化算法”一章中详细介绍。

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

        # 训练模型一共需要 num_epochs 个迭代周期。
        for epoch in range(1, num_epochs + 1):
            # 在一个迭代周期中，使用训练数据集中所有样本一次（假设样本数能够被批量大小整除）。
            # X 和 y 分别是小批量样本的特征和标签。
            for X, y in data_iter(batch_size, features, labels):
                with autograd.record():
                    # l 是有关小批量 X 和 y 的损失。
                    l = loss(net(X, w, b), y)
                    # 小批量的损失对模型参数求导。
                    l.backward()
                    # 使用小批量随机梯度下降迭代模型参数。
                    sgd([w, b], lr, batch_size)
                print("epoch %d, loss %f"
                      % (epoch, loss(net(features, w, b), labels).mean().asnumpy()))

epoch 1, loss 0.040584
epoch 2, loss 0.000158
epoch 3, loss 0.000050
```

训练完成后，我们可以比较学到的参数和用来生成训练集的真实参数。它们应该很接近。

```
In [13]: true_w, w

Out[13]: ([2, -3.4],
           [[ 1.99970508]
            [-3.39955592]]
           <NDArray 2x1 @cpu(0)>)

In [14]: true_b, b

Out[14]: (4.2,
           [ 4.19952631]
           <NDArray 1 @cpu(0)>)
```

3.2.8 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。

3.2.9 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？

3.2.10 扫码直达讨论区



3.3 线性回归的 Gluon 实现

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。

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 `features` 是训练数据特征，`labels` 是标签。

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

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。该模块提供了有关数据处理的工具。由于 `data` 常用作变量名，我们将导入的 `data` 模块用添加了 Gluon 首字母的假名 `gdata` 代替。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 [2]: from mxnet.gluon import data as gdata
```

```
batch_size = 10
# 将训练数据的特征和标签组合。
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 [3]: for X, y in data_iter:
    print(X, y)
    break
```

```
[[ -1.22437274 -0.27489948]
 [-1.61126184 -0.48350021]
 [-0.54291499  1.29740858]
 [-2.75658321 -0.8045069 ]
 [ 0.08435135 -1.20695007]
 [ 0.46745649 -0.75130075]
 [ 0.59524769 -2.19519234]
 [ 0.58976734 -0.63296652]
 [ 0.88179028  0.97483152]
 [ 0.96392924 -1.77612925]]
<NDArray 10x2 @cpu(0)>
[ 2.67071676  2.62262368  -1.29311502   1.41371882   8.47514153
 7.67654753  12.8497858   7.54309368   2.63797426  12.15272045]
<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。实际上，“`nn`”是 `neural networks`（神经网络）的缩写。顾名思义，该模块

定义了大量神经网络的层。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 [4]: 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 [5]: 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”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。我们从 MXNet 导入 `initializer` 模块。该模块提供了模型参数初始化的各种方法。这里的 `init` 是 `initializer` 的缩写形式。我们通过 `init.Normal(sigma=0.01)` 指定权重参数每个元素将在初始化时随机采样于均值为 0 标准差为 0.01 的正态分布。偏差参数全部元素初始化为零。

```
In [6]: from mxnet import init
```

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

3.3.5 定义损失函数

在 Gluon 中, `loss` 模块定义了各种损失函数。我们用假名 `gloss` 代替导入的 `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 [7]: 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，我们创建一个 Trainer 实例，并指定学习率为 0.03 的小批量随机梯度下降（sgd）为优化算法。该优化算法将用来迭代 net 实例所有通过 add 函数嵌套的层所包含的所有参数。

```
In [8]: from mxnet import gluon  
  
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.03})
```

3.3.7 训练模型

在使用 Gluon 训练模型时，我们通过调用 Trainer 实例的 step 函数来迭代模型参数。由于变量 l 是长度为 batch_size 的一维 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 函数中指明批量大小，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 [9]: num_epochs = 3  
for epoch in range(1, num_epochs + 1):  
    for X, y in data_iter:  
        with autograd.record():  
            l = loss(net(X), y)  
            l.backward()  
            trainer.step(batch_size)  
        print("epoch %d, loss: %f"  
              % (epoch, loss(net(features), labels).mean().asnumpy()))  
  
epoch 1, loss: 0.040507  
epoch 2, loss: 0.000155  
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（weight）和位移（bias）。学到的和真实的参数很接近。

```
In [10]: dense = net[0]  
true_w, dense.weight.data()  
  
Out[10]: ([2, -3.4],  
          [[ 1.99956036 -3.39956713]]  
          <NDArray 1x2 @cpu(0)>)  
  
In [11]: true_b, dense.bias.data()
```

```
Out[11]: (4.2,
 [ 4.19974422]
<NDArray 1 @cpu(0)>)
```

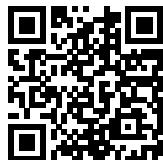
3.3.8 小结

- 使用 Gluon 可以更简洁地实现模型。
- 在 Gluon 中，`data` 模块提供了有关数据处理的工具，`nn` 模块定义了大量神经网络的层，`loss` 模块定义了各种损失函数。
- MXNet 的 `initializer` 模块提供了模型参数初始化的各种方法。

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 Softmax 回归

前几节介绍的线性回归模型适用于输出为连续值的情景。在其他情景中，模型输出还可以是一个像图片类别这样的离散值。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。

3.4.1 分类问题

让我们考虑一个简单的图片分类问题。通常，我们将高和宽分别为 m 和 n 个像素的图片的尺寸记作 $m \times n$ 或 (m, n) 。为了便于讨论，让我们假设输入图片的尺寸为 2×2 。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 $y_1 = 0, y_2 = 1, y_3 = 2$ ，任意一张狗的图片的标签记作 0。在这个分类问题中，输入一张图片的四个特征值，我们需要输出该图片的类别。

3.4.2 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 ，有关三个不同标签的输出分别为

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

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

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

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

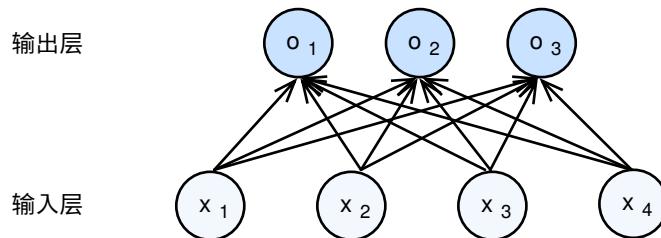


图 3.2: Softmax 回归是一个单层神经网络。

在得到输出层的三个输出后，我们需要预测输出分别为狗、猫或鸡的概率，即各个类别的预测概率。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，得到模型对各个

类别的预测概率

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

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

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

3.4.3 单样本分类的矢量计算表达式

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

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

设 2×2 图片样本 i 的特征为

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

输出层输出为

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

预测为狗、猫或鸡的概率分布为

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

我们对样本 i 分类的矢量计算表达式为

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

3.4.4 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 d ，输出个数（类别数）为 q 。设批量特征为 $\mathbf{X} \in \mathbb{R}^{n \times d}$ ，批量标签 $\mathbf{y} \in \mathbb{R}^{n \times 1}$ 。假设 softmax 回归的权重和偏差参数分别为 $\mathbf{W} \in \mathbb{R}^{d \times q}$, $\mathbf{b} \in \mathbb{R}^{1 \times q}$ 。Softmax 回归的矢量计算表达式为

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

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

3.4.5 交叉熵损失函数

Softmax 回归使用了交叉熵损失函数（cross-entropy loss）。给定分类问题的类别个数 m 。当样本 i 的标签类别为 y_j 时 ($1 \leq j \leq m$)，设 $q_j^{(i)} = 1$ 且当 $k \neq j, 1 \leq k \leq m$ 时 $q_k^{(i)} = 0$ 。设模型对样本 i 在类别 y_j 上的预测概率为 $p_j^{(i)}$ ($1 \leq j \leq m$)。假设训练数据集的样本数为 n ，交叉熵损失函数定义为

$$\ell(\Theta) = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^m q_j^{(i)} \log p_j^{(i)},$$

其中 Θ 代表模型参数。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。

不妨从另一个角度来理解交叉熵损失函数。在训练模型时，对于训练数据集上每个样本，总体上我们希望模型对这些样本真实的标签类别的预测概率尽可能大，也就是希望模型尽可能容易输出真实的标签类别。设 p_{label_i} 是模型对样本 i 的标签类别的预测概率，并设训练数据集的样本数为 n 。由于对数函数单调递增，最大化训练数据集所有标签类别的联合预测概率 $\prod_{i=1}^n p_{\text{label}_i}$ 等价于最大化 $\sum_{i=1}^n \log p_{\text{label}_i}$ ，即最小化 $-\sum_{i=1}^n \log p_{\text{label}_i}$ ，因此也等价于最小化以上定义的交叉熵损失函数。

3.4.6 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。

3.4.7 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。

3.4.8 练习

- 如果按本节 softmax 运算的定义来实现该运算，在计算时可能会有什么问题？提示：想想 $\exp(50)$ 的大小。

3.4.9 扫码直达讨论区



3.5 Softmax 回归的从零开始实现

下面我们来动手实现 Softmax 回归。首先，导入实验所需的包或模块。

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

3.5.1 获取 Fashion-MNIST 数据集

本节中，我们考虑图片分类问题。我们使用一个类别为服饰的数据集 Fashion-MNIST [1]。该数据集中，图片的高和宽均为 28 像素，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)。训练数据集中和测试数据集中的每个类别的图片数分别为 6,000 和 1,000。

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。

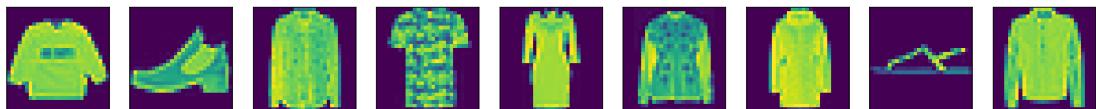
```
In [4]: def get_text_labels(labels):
    text_labels = [
        't-shirt', 'trouser', 'pullover', 'dress', 'coat',
        'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot'
    ]
    return [text_labels[int(i)] for i in labels]
```

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 = gb.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)
    gb.plt.show()
```

现在，我们看一下训练数据集中前 9 个样本的图片内容和文本标签。

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



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

3.5.2 读取数据

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

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

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

3.5.3 初始化模型参数

跟线性回归中的例子一样，我们将使用向量表示每个样本。已知每个样本是高和宽均为 28 像素的图片。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(scale=0.01, shape=(num_inputs, num_outputs))  
b = nd.zeros(num_outputs)  
params = [W, b]
```

同之前一样，我们要对模型参数附上梯度。

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

3.5.4 定义 Softmax 运算

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

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

```
In [10]: X = nd.array([[1,2,3], [4,5,6]])  
X.sum(axis=0, keepdims=True), X.sum(axis=1, keepdims=True)  
  
Out[10]: (  
    [[ 5.  7.  9.]]  
    <NDArray 1x3 @cpu(0)>,  
    [[ 6.]  
     [ 15.]]  
    <NDArray 2x1 @cpu(0)>)
```

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

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

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

```
In [12]: X = nd.random.normal(shape=(2, 5))  
X_prob = softmax(X)  
X_prob, X_prob.sum(axis=1)  
  
Out[12]: (  
    [[ 0.21324193  0.33961776  0.1239742   0.27106097  0.05210521]  
     [ 0.11462264  0.3461234   0.19401033  0.29583326  0.04941036]]  
    <NDArray 2x5 @cpu(0)>,  
    [ 1.00000012  1.         ]  
    <NDArray 2 @cpu(0)>)
```

3.5.5 定义模型

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

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

3.5.6 定义损失函数

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

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

```
Out[14]:
[ 0.1  0.5]
<NDArray 2 @cpu(0)>
```

下面，我们直接将上一节中的交叉熵损失函数翻译成代码。

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

3.5.7 计算分类准确率

给定一个类别的预测概率分布 `y_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]: 0.0947265625
```

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

3.5.8 训练模型

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

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

def train_cpu(net, train_iter, test_iter, loss, num_epochs, batch_size,
             params=None, lr=None, trainer=None):
    for epoch in range(1, num_epochs + 1):
        train_l_sum = 0
        train_acc_sum = 0
        for X, y in train_iter:
            with autograd.record():
                y_hat = net(X)
                l = loss(y_hat, y)
            l.backward()
            if trainer is None:
```

```

        gb.sgd(params, lr, batch_size)
    else:
        trainer.step(batch_size)
    train_l_sum += l.mean().asscalar()
    train_acc_sum += accuracy(y_hat, y)
    test_acc = evaluate_accuracy(test_iter, net)
    print("epoch %d, loss %.4f, train acc %.3f, test acc %.3f"
        % (epoch, train_l_sum / len(train_iter),
           train_acc_sum / len(train_iter), test_acc))

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

epoch 1, loss 0.7902, train acc 0.743, test acc 0.801
epoch 2, loss 0.5730, train acc 0.811, test acc 0.823
epoch 3, loss 0.5292, train acc 0.823, test acc 0.825
epoch 4, loss 0.5065, train acc 0.829, test acc 0.839
epoch 5, loss 0.4894, train acc 0.834, test acc 0.845

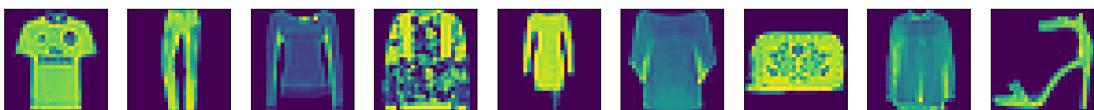
```

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: ['t-shirt', 'trouser', 'pullover', 't-shirt', 'coat', 'shirt', 'bag',
        ↵ 'shirt', 'sandal']
```

3.5.10 小结

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

3.5.11 练习

- 本节中，我们直接按照 Softmax 运算的数学定义来实现 `softmax` 函数。这可能会造成什么问题？（试一试计算 e^{50} 的大小。）
- 本节中的 `cross_entropy` 函数同样是以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, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747.

3.6 Softmax 回归的 Gluon 实现

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

首先导入本节实现所需的包或模块。

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

3.6.1 获取和读取数据

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

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

3.6.2 定义和初始化模型

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

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

3.6.3 Softmax 和交叉熵损失函数

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

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

3.6.4 定义优化算法

我们使用学习率为 0.1 的小批量随机梯度下降作为优化算法。

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

3.6.5 训练模型

接下来，我们使用上一节中定义的训练函数来训练模型。

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

```
epoch 1, loss 0.7885, train acc 0.747, test acc 0.810
epoch 2, loss 0.5731, train acc 0.812, test acc 0.823
epoch 3, loss 0.5285, train acc 0.825, test acc 0.826
epoch 4, loss 0.5050, train acc 0.831, test acc 0.834
epoch 5, loss 0.4895, train acc 0.834, test acc 0.840
```

3.6.6 小结

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

3.6.7 练习

- 尝试调一调超参数，例如批量大小、迭代周期和学习率，看看结果会怎样。

3.6.8 扫码直达讨论区



3.7 多层感知机

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

多层感知机是最基础的深度学习模型。

3.7.1 隐藏层

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

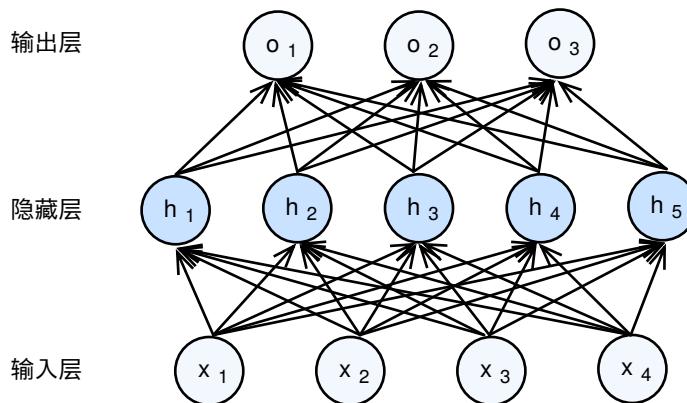


图 3.3: 带有隐藏层的多层感知机。它含有一个隐藏层，该层中有 5 个隐藏单元。

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

3.7.2 仿射变换

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

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

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

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

那么，如果隐藏层也只对输入做仿射变换会怎么样呢？设单个样本的特征为 $\mathbf{x} \in \mathbb{R}^{1 \times d}$ ，隐藏层的权重参数和偏差参数分别为 $\mathbf{W}_h \in \mathbb{R}^{d \times h}, \mathbf{b}_h \in \mathbb{R}^{1 \times h}$ 。假设 $\mathbf{h} = \mathbf{x}\mathbf{W}_h + \mathbf{b}_h$ 且 $\mathbf{o} = \mathbf{h}\mathbf{W}_o + \mathbf{b}_o$ ，联立两式可得 $\mathbf{o} = \mathbf{x}\mathbf{W}_h\mathbf{W}_o + \mathbf{b}_h\mathbf{W}_o + \mathbf{b}_o$ ：它等价于单层神经网络的输出 $\mathbf{o} = \mathbf{x}\mathbf{W}' + \mathbf{b}'$ ，其中 $\mathbf{W}' = \mathbf{W}_h\mathbf{W}_o, \mathbf{b}' = \mathbf{b}_h\mathbf{W}_o + \mathbf{b}_o$ 。因此，在隐藏层中仅使用仿射变换的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

通过上面的例子，我们发现：必须在隐藏层中使用其他变换，例如添加非线性变换，才能使多层感知机变得有意义。我们将这些非线性变换称为激活函数（activation function）。激活函数能对任意形状的输入都按元素操作且不改变输入的形状。以下列举三种常用的激活函数。

ReLU 函数

ReLU (rectified linear unit) 函数提供了一个很简单的非线性变换。给定元素 x ，该函数定义为

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

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

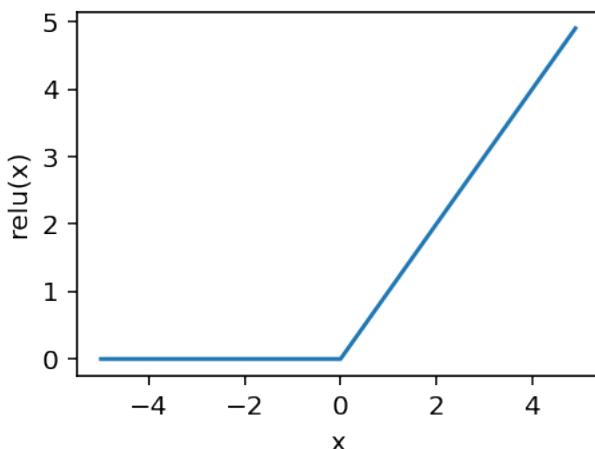
```
In [1]: import sys
sys.path.append('..')
import gluonbook as gb
from mxnet import nd

def xyplot(x_vals, y_vals, x_label, y_label):
    %config InlineBackend.figure_format = 'retina'
```

```
gb=plt.rcParams['figure.figsize'] = (3.5, 2.5)
gb=plt.plot(x_vals,y_vals)
gb=plt.xlabel(x_label)
gb=plt.ylabel(y_label)
gb=plt.show()
```

我们接下来绘制 ReLU 函数。当元素值非负时，ReLU 函数实际上在做线性变换。

```
In [2]: x = np.arange(-5.0, 5.0, 0.1)
xyplot(x.astype(np.float), x.relu().astype(np.float), 'x', 'relu(x)')
```



Sigmoid 函数

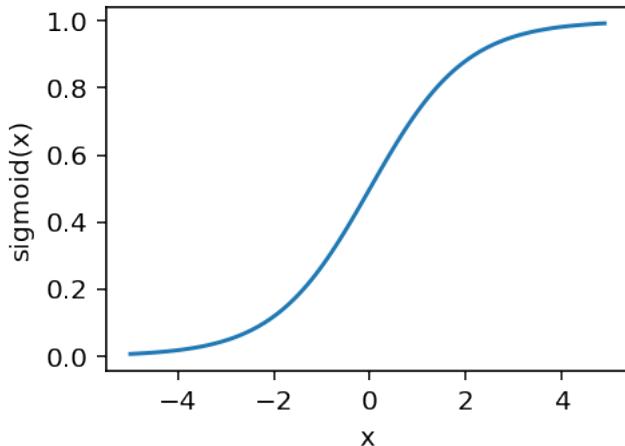
Sigmoid 函数可以将元素的值变换到 0 和 1 之间：

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

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

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

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



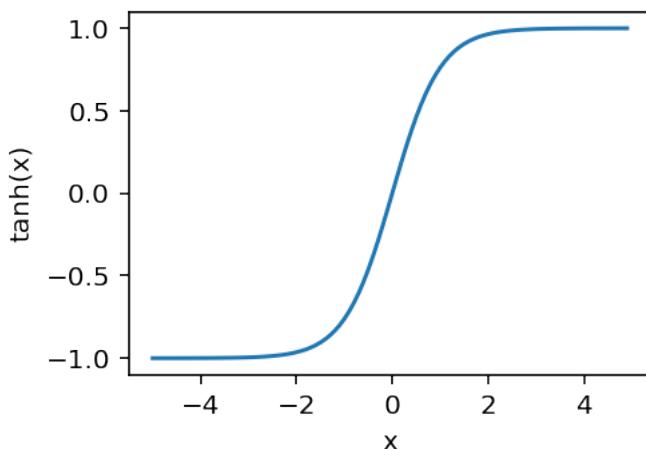
Tanh 函数

Tanh (双曲正切) 函数可以将元素的值变换到-1 和 1 之间:

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

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

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



然后，我们使用三种激活函数来变换输入。按元素操作后，输入和输出形状相同。

```
In [5]: X = 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 $X \in \mathbb{R}^{n \times d}$ ，其批量大小为 n ，输入个数为 d ，输出个数为 q 。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 $W_h \in \mathbb{R}^{d \times h}, b_h \in \mathbb{R}^{1 \times h}$ ，输出层的权重和偏差参数分别为 $W_o \in \mathbb{R}^{h \times q}, b_o \in \mathbb{R}^{1 \times q}$ 。多层感知机的矢量计算表达式为

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

$$O = HW_o + b_o,$$

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

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

3.7.5 计算梯度

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

3.7.6 衰减和爆炸

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

3.7.7 随机初始化模型参数

在神经网络中，我们需要随机初始化模型参数。下面我们来解释这一点。

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

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

MXNet 的默认随机初始化

随机初始化模型参数的方法有很多。在“[线性回归的 Gluon 实现](#)”一节中，我们使用 `net.initialize(init.Normal(sigma=0.01))` 使模型 `net` 的权重参数采用正态分布的随机初始化方式。如果不指定初始化方法，例如 `net.initialize()`，我们将使用 MXNet 的默认随机初始化方法。默认的随机初始化中，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，偏差参数全部清零。

Xavier 随机初始化

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

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

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

3.7.8 小结

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

3.7.9 练习

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

3.7.10 扫码直达讨论区



3.7.11 参考文献

[1] Glorot, 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).

3.8 多层感知机的从零开始实现

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

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

3.8.1 获取和读取数据

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

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

3.8.2 定义模型参数

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

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

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

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

3.8.3 定义激活函数

这里我们使用 ReLU 作为隐藏层的激活函数。

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

3.8.4 定义模型

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

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

3.8.5 定义损失函数

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

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

3.8.6 训练模型

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

我们在这里设超参数迭代周期数为 5，学习率为 0.5。

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

epoch 1, loss 0.8032, train acc 0.699, test acc 0.788
epoch 2, loss 0.4922, train acc 0.816, test acc 0.838
epoch 3, loss 0.4288, train acc 0.843, test acc 0.850
epoch 4, loss 0.3939, train acc 0.854, test acc 0.869
epoch 5, loss 0.3717, train acc 0.863, test acc 0.851
```

3.8.7 小结

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

3.8.8 练习

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

3.8.9 扫码直达讨论区



3.9 多层感知机的 Gluon 实现

下面我们使用 Gluon 来实现上一节中的多层感知机。首先我们导入所需的包或模块。

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

3.9.1 定义模型

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

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

3.9.2 读取数据并训练模型

我们使用和训练 Softmax 回归几乎相同的步骤来读取数据并训练模型。

```
In [3]: batch_size = 256
        train_iter, test_iter = gb.load_data_fashion_mnist(batch_size)

        loss = gloss.SoftmaxCrossEntropyLoss()
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.5})
        num_epochs = 5
```

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

epoch 1, loss 0.8172, train acc 0.692, test acc 0.819
epoch 2, loss 0.4974, train acc 0.815, test acc 0.828
epoch 3, loss 0.4297, train acc 0.841, test acc 0.850
epoch 4, loss 0.4074, train acc 0.847, test acc 0.864
epoch 5, loss 0.3788, train acc 0.860, test acc 0.870
```

3.9.3 小结

- 通过 Gluon 我们可以更方便地构造多层感知机。

3.9.4 练习

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

3.9.5 扫码直达讨论区



3.10 欠拟合、过拟合和模型选择

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

3.10.1 训练误差和泛化误差

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

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

3.10.2 欠拟合和过拟合

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

模型复杂度

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

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

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

由于高阶多项式函数模型参数更多，模型函数的选择空间更大，所以高阶多项式函数比低阶多项式函数的复杂度更高。因此，高阶多项式函数比低阶多项式函数更容易在相同的训练数据集上得到更低的训练误差。给定训练数据集，模型复杂度和误差之间的关系通常如图 3.4 所示。给定训

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

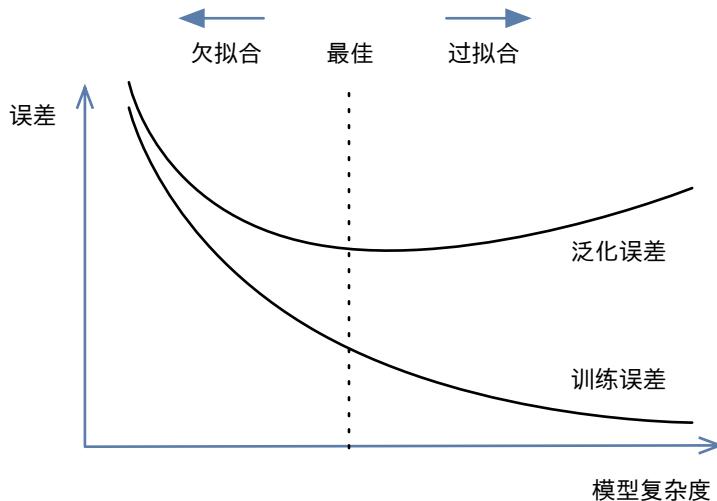


图 3.4: 模型复杂度对欠拟合和过拟合的影响

训练数据集大小

影响欠拟合和过拟合的另一个重要因素是训练数据集的大小。一般来说，如果训练数据集中样本数过少，特别是比模型参数数量更少时，过拟合更容易发生。

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

3.10.3 多项式函数拟合实验

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

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

生成数据集

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

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

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

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

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

看一看生成的数据集的前 5 个样本。

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

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

定义、训练和测试模型

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

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

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

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

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

```

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

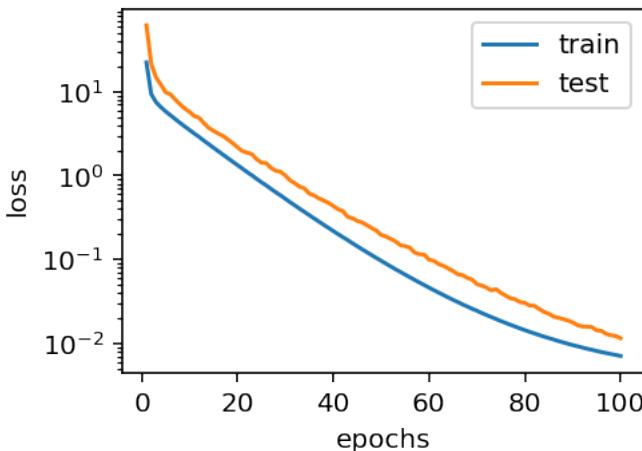
```

三阶多项式函数拟合（正常）

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

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

final epoch: train loss 0.0070791 test loss 0.0115047
```



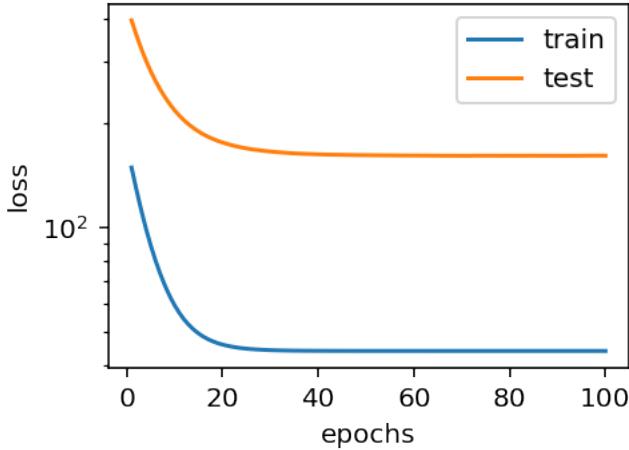
```
Out[6]: ('weight:',
 [[ 1.32713091 -3.36304903  5.56286764]]
<NDArray 1x3 @cpu(0)>, 'bias:',
 [ 4.95161009]
<NDArray 1 @cpu(0)>)
```

线性函数拟合（欠拟合）

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

成的数据集上容易欠拟合。

```
In [7]: fit_and_plot(features[:n_train, :], features[n_train:, :], labels[:n_train],  
                     labels[n_train:])  
  
final epoch: train loss 43.9977 test loss 160.814
```



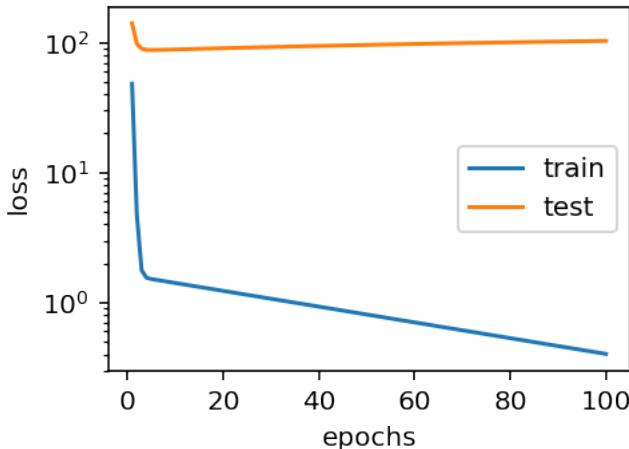
```
Out[7]: ('weight:',  
         [[ 15.55646324]]  
         <NDArray 1x1 @cpu(0)>, 'bias:',  
         [ 2.29493761]  
         <NDArray 1 @cpu(0)>)
```

训练量不足（过拟合）

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

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

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



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

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

3.10.4 模型选择

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

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

K 折交叉验证

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

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

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

我们将在本章最后一节实验 K 折交叉验证。

3.10.5 小结

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

3.10.6 练习

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

3.10.7 扫码直达讨论区



3.11 权重衰减

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

3.11.1 L_2 范数正则化

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

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

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

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

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

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

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

3.11.2 高维线性回归实验

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

生成数据集

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

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

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

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

n_train = 20
n_test = 100

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

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

初始化模型参数

接着，定义随机初始化模型参数的函数。该函数也给每个参数都附上梯度。

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

定义 L_2 范数惩罚项

下面定义 L_2 范数惩罚项。这里只惩罚模型的权重参数。

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

定义训练和测试

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

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

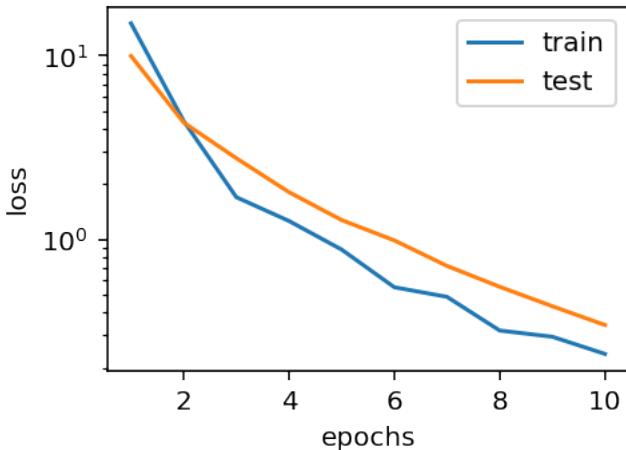
        net = gb.linreg
        loss = gb.squared_loss
        %config InlineBackend.figure_format = 'retina'
        gb.pyplot.rcParams['figure.figsize'] = (3.5, 2.5)

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

观察过拟合

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

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



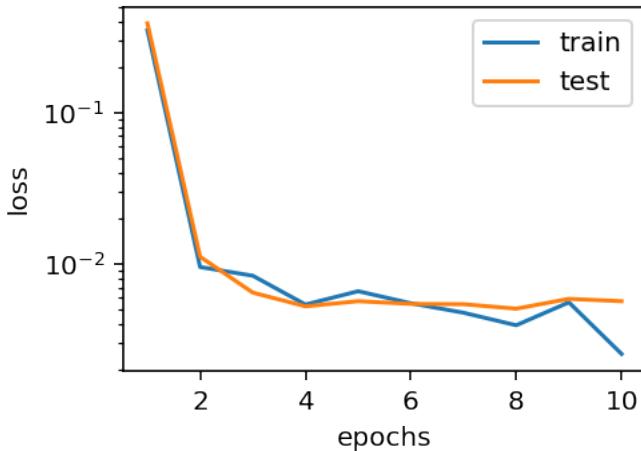
Out[5]: ('w[:10]:',
[[0.0994669 0.0222058 -0.19190376 -1.17747068 -0.6129607 -0.63053668
-0.22882056 0.64572489 -0.94738561 -0.74087125]]
<NDArray 1x10 @cpu(0)>, 'b:',
[-0.06496799]
<NDArray 1 @cpu(0)>)

使用正则化

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

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

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



```
Out[6]: ('w[:10]:',
 [[ 4.71860729e-03  4.53395816e-03  8.39446671e-04  3.37020471e-03
 -6.15871977e-04 -7.98055320e-04  1.48303586e-03  2.36676261e-03
 -2.05319928e-04  5.55917359e-05]]
<NDArray 1x10 @cpu(0)>, 'b:',
 [ 0.05804982]
<NDArray 1 @cpu(0)>)
```

3.11.3 小结

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

3.11.4 练习

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

3.11.5 扫码直达讨论区



3.12 权重衰减的 Gluon 实现

本节将介绍如何使用 Gluon 实现上一节介绍的权重衰减。首先导入实验所需的包或模块。

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

3.12.1 生成数据集

我们使用和上一节完全一样的方法生成数据集。

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

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

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

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

3.12.2 定义训练和测试

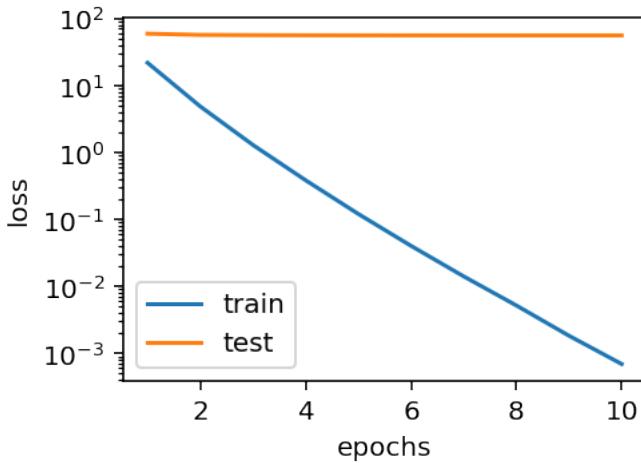
在训练和测试的定义中，我们分别定义了两个 Trainer 实例。其中一个对权重参数做 L_2 范数正则化，另一个不对偏差参数做正则化。我们在上一节也提到了，实际中有时也对偏差参数做正则化。这样只需要定义一个 Trainer 实例就可以了。

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

3.12.3 观察实验结果

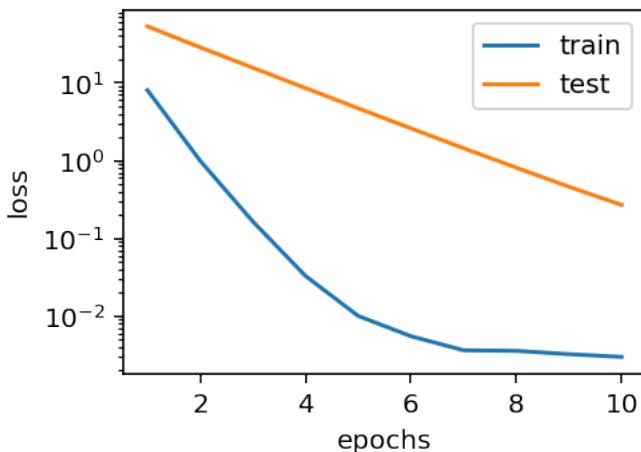
以下实验结果和上一节中的类似：使用正则化可以在一定程度上缓解过拟合问题。

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



```
Out[4]: ('w[:10]:',
 [[-0.81741488 -0.22294007 -0.74873251 -2.25932193 -0.10177195 -0.69809818
 -0.09751856 0.60939908 -1.10466301 -1.44874275]
<NDArray 1x10 @cpu(0)>, 'b:',
 [-0.38623527]
<NDArray 1 @cpu(0)>)
```

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



```
Out[5]: ('w[:10]:',
 [[ 0.01610127 -0.00395602 -0.02773624  0.00625052  0.00691587  0.01518991
```

```
0.00830917 0.01416303 0.01940586 -0.00236085]]  
<NDArray 1x10 @cpu(0)>, 'b:',  
[-0.1453924]  
<NDArray 1 @cpu(0)>)
```

3.12.4 小结

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

3.12.5 练习

- 调一调本节实验中的 `wd` 超参数。观察并分析实验结果。

3.12.6 扫码直达讨论区



3.13 丢弃法

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

3.13.1 方法和原理

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

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

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

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

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

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

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

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

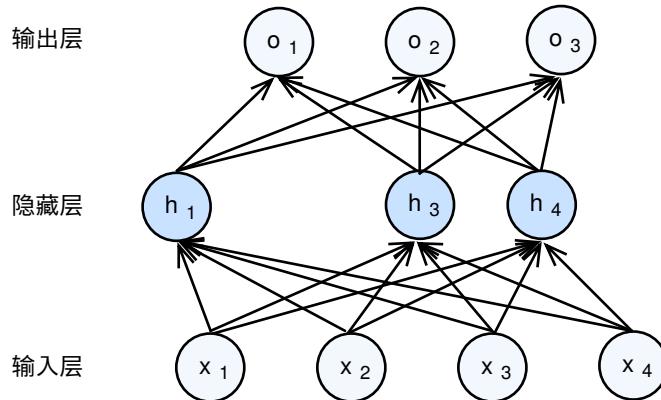


图 3.5：隐藏层使用了丢弃法的多层感知机。

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

3.13.2 实现丢弃法

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

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

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

我们运行几个例子来验证一下 `dropout` 函数。

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

Out[2]:

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

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

Out[3]:

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

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

Out[4]:

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

3.13.3 定义模型参数

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

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

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

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

3.13.4 定义模型

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

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

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

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

3.13.5 训练和测试模型

这部分和之前多层感知机的训练与测试类似。

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

epoch 1, loss 1.1012, train acc 0.572, test acc 0.727
epoch 2, loss 0.5778, train acc 0.786, test acc 0.773
epoch 3, loss 0.4948, train acc 0.820, test acc 0.823
epoch 4, loss 0.4412, train acc 0.839, test acc 0.840
epoch 5, loss 0.4171, train acc 0.847, test acc 0.859
```

3.13.6 小结

- 我们可以通过使用丢弃法应对过拟合。
- 只需在训练模型时使用丢弃法。

3.13.7 练习

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

3.13.8 扫码直达讨论区



3.14 丢弃法的 Gluon 实现

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

3.14.1 定义模型并添加丢弃层

在多层感知机中 Gluon 实现的基础上，我们只需要在全连接层后添加 Dropout 层并指定丢弃概率。在训练模型时，Dropout 层将以指定的丢弃概率随机丢弃上一层的输出元素；在测试模型时，Dropout 层并不发挥作用。

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

        drop_prob1 = 0.2
        drop_prob2 = 0.5

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

3.14.2 训练和测试模型

这部分依然和多层感知机中的训练和测试没有多少区别。

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

epoch 1, loss 1.1853, train acc 0.541, test acc 0.739
epoch 2, loss 0.5978, train acc 0.779, test acc 0.785
epoch 3, loss 0.5027, train acc 0.815, test acc 0.827
epoch 4, loss 0.4581, train acc 0.832, test acc 0.829
epoch 5, loss 0.4219, train acc 0.845, test acc 0.854
```

3.14.3 小结

- 使用 Gluon，我们可以更方便地构造多层神经网络并使用丢弃法。

3.14.4 练习

- 尝试不同丢弃概率超参数组合，观察并分析结果。

3.14.5 扫码直达讨论区



3.15 正向传播和反向传播

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

3.15.1 概念

神经网络中的梯度计算主要靠正向传播 (forward propagation) 和反向传播 (back-propagation)。正向传播指的是对神经网络沿着从输入层到输出层的顺序，依次计算并存储模型中间变量的过程。反向传播指的是计算神经网络参数梯度的方法。总的来说，反向传播中会依据微积分中的链式法则，沿着从输出层到输入层的顺序，依次计算并存储损失函数有关神经网络各层的中间变量以及参数的梯度。反向传播时，计算有关各层变量和参数的梯度可能会依赖于各层变量和参数的当前值。而这些变量的当前值来自正向传播的计算结果。

3.15.2 案例分析——正则化的多层感知机

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

定义模型

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

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

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

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

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

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

假设损失函数为 ℓ , 我们可以计算出单个数据样本的损失项

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

根据 L_2 范数正则化的定义, 给定超参数 λ , 正则化项即

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

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

$$J = L + s.$$

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

模型计算图

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

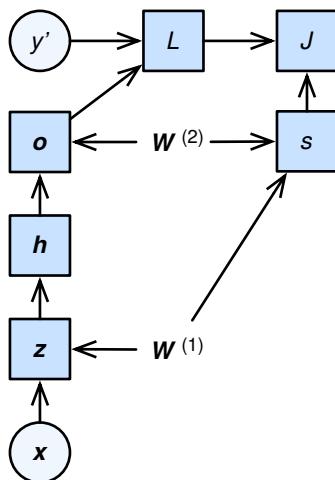


图 3.6: 正则化的多层感知机模型计算中的依赖关系。方框中字母代表变量, 圆圈中字母代表数据样本特征和标签, 无边框的字母代表模型参数。

正向传播

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

反向传播

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

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

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

来表达链式法则。

首先，我们计算目标函数有关损失项和有关正则项的梯度

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

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

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

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

接下来，我们可以很直观地计算出正则项有关两个参数的梯度：

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

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

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

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

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

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

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

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

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

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

需要强调的是，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。这可以用来避免重复计算某些梯度需要的值。例如，由于输出层变量梯度 $\partial J / \partial \mathbf{o}$ 被计算存储，反向传播稍后的参数梯度 $\partial J / \partial \mathbf{W}^{(2)}$ 和隐藏层变量梯度 $\partial J / \partial \mathbf{h}$ 的计算可以直接读取输出层变量梯度的值，而无需重复计算。

正向传播和反向传播相互依赖

事实上，正向传播和反向传播相互依赖。为什么这么说呢？

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

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

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

3.15.3 小结

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

3.15.4 练习

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

3.15.5 扫码直达讨论区



3.16 实战 Kaggle 比赛：房价预测

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

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

- 处理离散数据；
- 处理丢失的数据特征；
- 对数据进行标准化。

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

3.16.1 Kaggle 比赛

Kaggle（网站地址：<https://www.kaggle.com>）是一个著名的供机器学习爱好者交流的平台。图 3.7 展示了 Kaggle 网站首页。为了便于提交结果，请大家注册 Kaggle 账号。

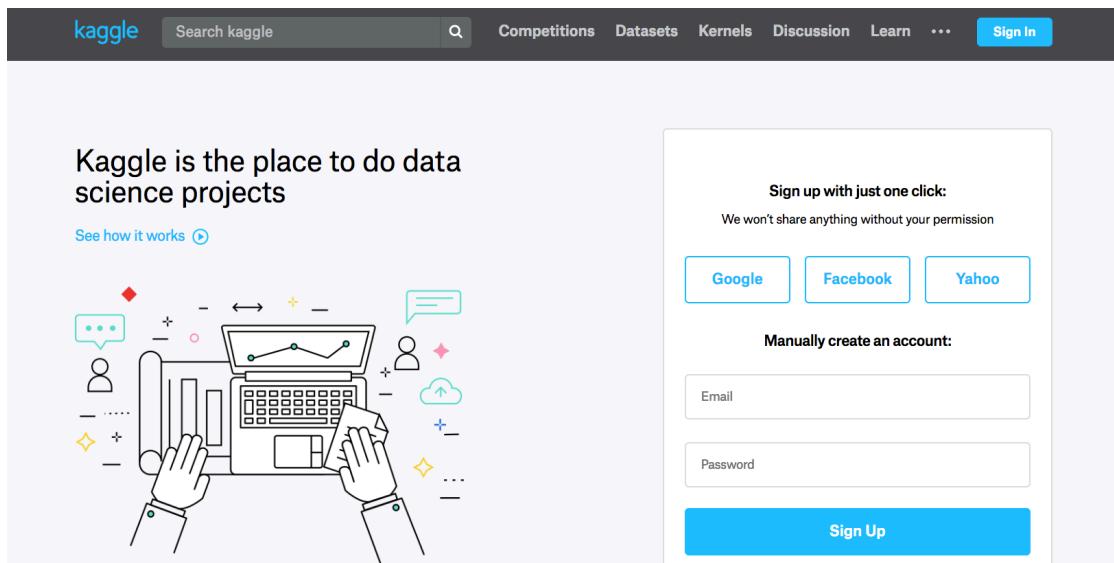


图 3.7: Kaggle 网站首页。

我们可以在房价预测比赛的网页上了解比赛信息和参赛者成绩、下载数据集并提交自己的预测结果。该比赛的网页地址是

<https://www.kaggle.com/c/house-prices-advanced-regression-techniques>

图 3.8 展示了房价预测比赛的网页信息。

图 3.8: 房价预测比赛的网页信息。比赛数据集可通过点击“Data”标签获取。

3.16.2 获取和读取数据集

比赛数据分为训练数据集和测试数据集。两个数据集都包括每栋房子的特征，例如街道类型、建造年份、房顶类型、地下室状况等特征值。这些特征值有连续的数字、离散的标签甚至是缺失值“na”。只有训练数据集包括了每栋房子的价格。我们可以访问比赛网页，点击图 3.8 中的“Data”标签，并下载这些数据集。

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

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

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

训练数据集包括 1460 个样本、80 个特征和 1 个标签。

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

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

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

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

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

3.16.3 预处理数据

我们对连续数值的特征做标准化处理。如果一个特征的值是连续的，设该特征在训练数据集和测试数据集上的均值为 μ ，标准差为 σ 。那么，我们可以将该特征的每个值先减去 μ 再除以 σ 得到标准化后的每个特征值。

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

现在，我们对离散数值的特征进一步处理，并把缺失数据值用本特征的平均值进行估计。

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

接下来，我们把数据转换一下格式。

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

3.16.4 导入 NDArray 格式数据

为了便于和 Gluon 交互，我们将数据以 NDArray 的格式导入。

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

我们使用平方损失函数训练模型，并定义比赛用来评价模型的函数。

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

3.16.5 定义模型

我们将模型的定义放在一个函数里供多次调用。在此我们使用一个基本的线性回归模型，并对模型参数做 Xavier 随机初始化。

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

3.16.6 定义训练函数

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

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

```

        print("epoch %d, train loss: %f" % (epoch, cur_train_l))
train_ls.append(cur_train_l)
if test_features is not None:
    cur_test_l = get_rmse_log(net, test_features, test_labels)
    test_ls.append(cur_test_l)
if test_features is not None:
    gb.semilogy(range(1, num_epochs+1), train_ls, 'epochs', 'loss',
                 range(1, num_epochs+1), test_ls, ['train', 'test'])
else:
    gb.semilogy(range(1, num_epochs+1), train_ls, 'epochs', 'loss')
if test_features is not None:
    return cur_train_l, cur_test_l
else:
    return cur_train_l

```

3.16.7 定义 K 折交叉验证

我们在“欠拟合、过拟合和模型选择”一节中介绍了 K 折交叉验证。下面，我们将定义 K 折交叉验证函数，并根据 K 折交叉验证的结果选择模型设计并调参。

```

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

```

```
train_l, test_l = train(
    net, X_val_train, y_val_train, X_val_test, y_val_test,
    epochs, verbose_epoch, learning_rate, weight_decay, batch_size)
train_l_sum += train_l
print("test loss: %f" % test_l)
test_l_sum += test_l
return train_l_sum / k, test_l_sum / k
```

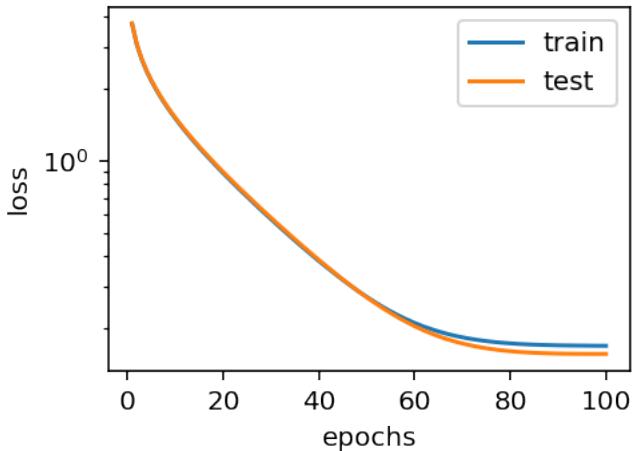
3.16.8 交叉验证模型

现在，我们可以交叉验证模型了。以下的超参数都是可以调节的。

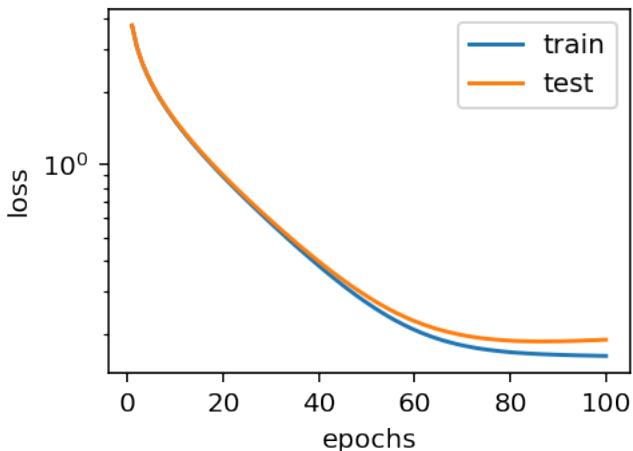
```
In [12]: k = 5
num_epochs = 100
verbose_epoch = num_epochs - 2
lr = 5
weight_decay = 0
batch_size = 64

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

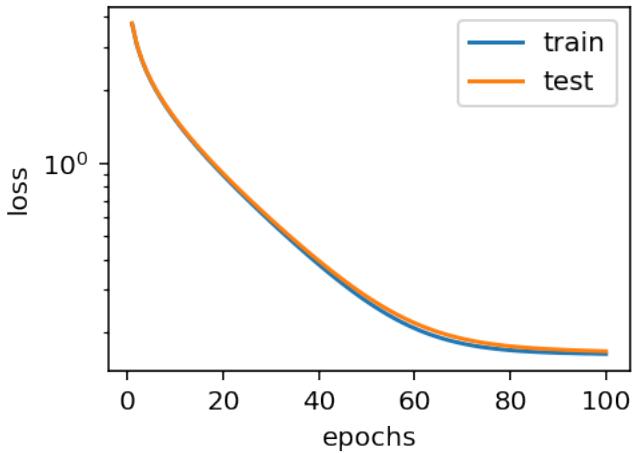
epoch 98, train loss: 0.169791
epoch 99, train loss: 0.169772
epoch 100, train loss: 0.169722
```



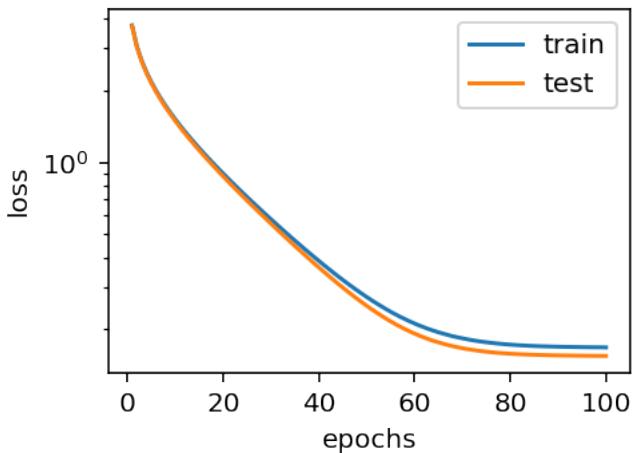
```
test loss: 0.156992
epoch 98, train loss: 0.162454
epoch 99, train loss: 0.162359
epoch 100, train loss: 0.162189
```



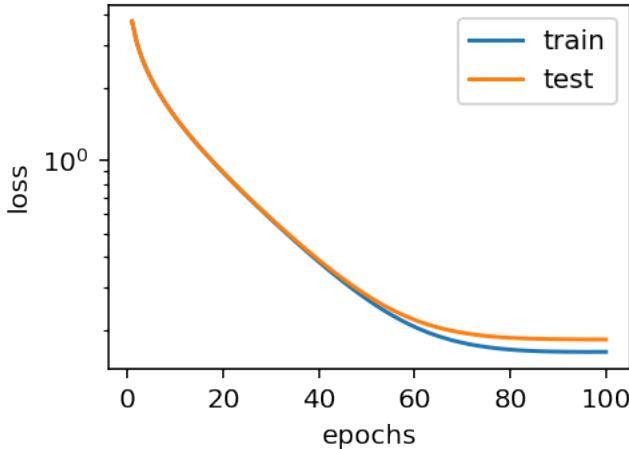
```
test loss: 0.189166
epoch 98, train loss: 0.163998
epoch 99, train loss: 0.163836
epoch 100, train loss: 0.163694
```



```
test loss: 0.167869
epoch 98, train loss: 0.168051
epoch 99, train loss: 0.168019
epoch 100, train loss: 0.167849
```



```
test loss: 0.154609
epoch 98, train loss: 0.162812
epoch 99, train loss: 0.162779
epoch 100, train loss: 0.162819
```



```
test loss: 0.182976
5-fold validation: avg train loss: 0.165255, avg test loss: 0.170322
```

在设定了一组参数后，即便训练误差可以达到很低，但是在 K 折交叉验证上的误差可能反而较高。这种现象这很可能是由于过拟合造成的。因此，当训练误差特别低时，我们要观察 K 折交叉验证上的误差是否同时降低，以避免模型的过拟合。

3.16.9 预测并在 Kaggle 提交结果

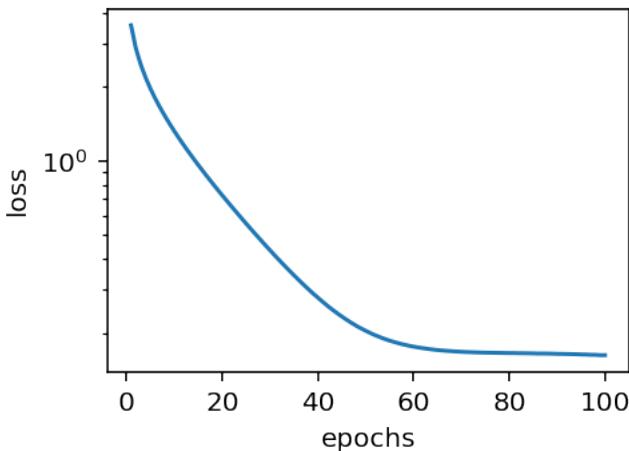
我们首先定义预测函数。在预测之前，我们会使用完整的训练数据集来重新训练模型。

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

设计好模型并调好超参数之后，下一步就是对测试数据集上的房屋样本做价格预测，并在 Kaggle 上提交结果。

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

```
epoch 98, train loss: 0.162956  
epoch 99, train loss: 0.162466  
epoch 100, train loss: 0.162498
```



上述代码执行完之后会生成一个“`submission.csv`”文件。这个文件是符合 Kaggle 比赛要求的提交格式的。这时，我们可以在 Kaggle 上把我们预测得出的结果进行提交，并且查看与测试数据集上真实房价（标签）的误差。具体来说有以下几个步骤：你需要登录 Kaggle 网站，访问房价预测比赛网页，并点击右侧“Submit Predictions”或“Late Submission”按钮。然后，点击页面下方“Upload Submission File”选择需要提交的预测结果文件。最后，点击页面最下方的“Make Submission”按钮就可以查看结果了。如图 3.9 所示。

Step 1
Upload submission file

File Format
Your submission should be in CSV format.
You can upload this in a zip/gz/rar/7z archive, if you prefer.

Number of Predictions
We expect the solution file to have 1459 prediction rows. This file should have a header row. Please see sample submission file on the [data page](#).

Step 2
Describe submission

B I | % ⏪ ⏩ ⏴ ⏵ | ⌂ ⌃ H ⌂ | ⌂ ⌃

M Styling with Markdown supported

Briefly describe your submission.

Make Submission

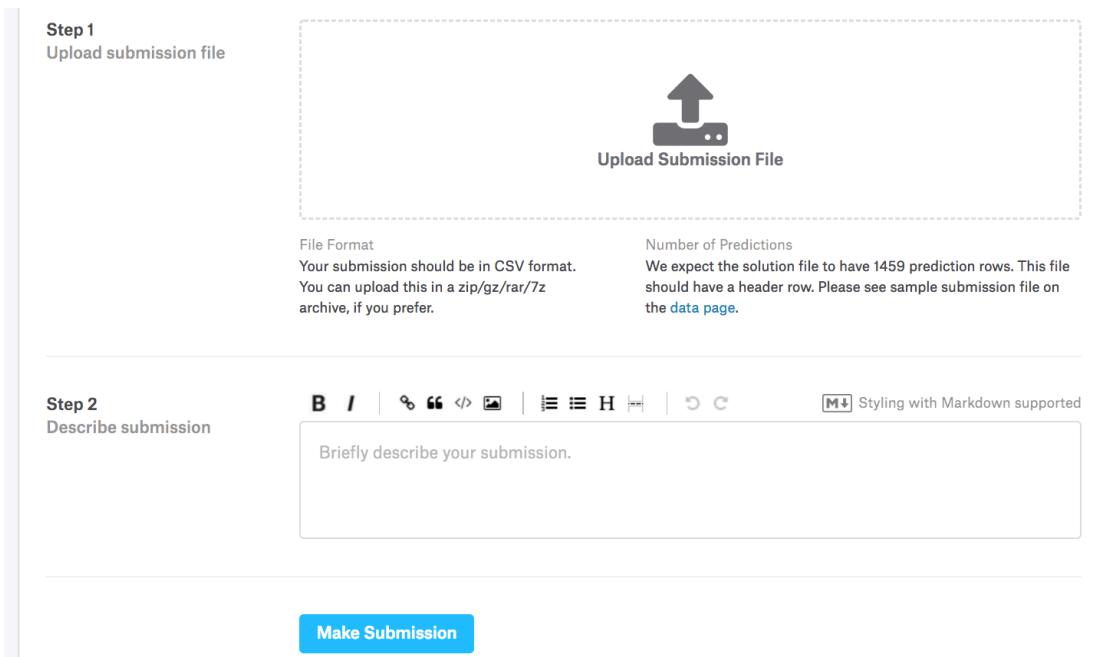


图 3.9: Kaggle 预测房价比赛的预测结果提交页面。

3.16.10 小结

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

3.16.11 练习

- 在 Kaggle 提交本教程的预测结果。观察一下，这个结果能在 Kaggle 上拿到什么样的分数？
- 对照 K 折交叉验证结果，不断修改模型（例如添加隐藏层）和调参，你能提高 Kaggle 上的分数吗？
- 如果不使用本节中对连续数值特征的标准化处理，结果会有什么变化？
- 扫码直达讨论区，在社区交流方法和结果。你能发掘出其他更好的技巧吗？

3.16.12 扫码直达讨论区



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

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

4.1.1 继承 Block 类来构造模型

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

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

class MLP(nn.Block):
    # 声明带有模型参数的层，这里我们声明了两个全链接层。
    def __init__(self, **kwargs):
        # 调用 MLP 父类 Block 的构造函数来进行必要的初始化。这样在构造实例时还可以指定
        # 其他函数参数，例如下一节将介绍的模型参数 params。
        super(MLP, self).__init__(**kwargs)
        # 隐藏层。
        self.hidden = nn.Dense(256, activation='relu')
        # 输出层。
        self.output = nn.Dense(10)
    # 定义模型的前向计算，即如何根据输出计算输出。
    def forward(self, x):
        return self.output(self.hidden(x))
```

我们可以实例化 MLP 类得到 `net`, 其使用跟“多层感知机的 Gluon 实现”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遍历元素。
        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“多层感知机的 Gluon 实现”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。在这个网络中，我们通过 `get_constant` 函数创建训练中不被迭代的参数，即常数参数。在前向计算中，除了使用创建的常数参数外，我们还使用 NDArray 的函数和 Python 的控制流，并多次调用同一层。

```
In [5]: class FancyMLP(nn.Block):
    def __init__(self, **kwargs):
        super(FancyMLP, self).__init__(**kwargs)
        # 使用 get_constant 创建的随机权重参数不会在训练中被迭代（即常数参数）。
        self.rand_weight = self.params.get_constant(
            'rand_weight', nd.random.uniform(shape=(20, 20)))
        self.dense = nn.Dense(20, activation='relu')

    def forward(self, x):
        x = self.dense(x)
        # 使用创建的常数参数，以及 NDArray 的 relu 和 dot 函数。
        x = nd.relu(nd.dot(x, self.rand_weight.data()) + 1)
        # 重用全连接层。等价于两个全连接层共享参数。
        x = self.dense(x)
        # 控制流，这里我们需要调用 asscalar 来返回标量进行比较。
        while x.norm().asscalar() > 1:
            x /= 2
        if x.norm().asscalar() < 0.8:
            x *= 10
        return x.sum()
```

在这个 FancyMLP 模型中，我们使用了常数权重 `rand_weight`（注意它不是模型参数）、做了矩阵乘法操作（`nd.dot`）并重复使用了相同的 Dense 层。测试一下：

```
In [6]: net = FancyMLP()
net.initialize()
net(x)

Out[6]:
[ 18.57195282]
<NDArray 1 @cpu(0)>
```

由于 FancyMLP 和 Sequential 都是 Block 的子类，我们可以嵌套调用他们。

```
In [7]: class NestMLP(nn.Block):
    def __init__(self, **kwargs):
        super(NestMLP, self).__init__(**kwargs)
        self.net = nn.Sequential()
        self.net.add(nn.Dense(64, activation='relu'),
                    nn.Dense(32, activation='relu'))
        self.dense = nn.Dense(16, activation='relu')

    def forward(self, x):
        return self.dense(self.net(x))

net = nn.Sequential()
net.add(NestMLP(), nn.Dense(20), FancyMLP())

net.initialize()
net(x)

Out[7]:
[ 24.86621094]
<NDArray 1 @cpu(0)>
```

4.1.4 小结

- 我们可以通过继承 Block 类来构造复杂的模型。
- Sequential 是 Block 的子类。

4.1.5 练习

- 在 FancyMLP 类里我们重用了 dense，这样对输入形状有了一定要求，尝试改变下输入数据形状试试。
- 如果我们去掉 FancyMLP 里面的 asscalar 会有什么问题？
- 在 NestMLP 里假设我们改成 self.net=[nn.Dense(64, activation='relu'), nn.Dense(32, activation='relu')], 而不是用 Sequential 类来构造，会有什么问题？

4.1.6 扫码直达讨论区



4.2 模型参数的访问、初始化和共享

在之前的小节里我们一直在使用默认的初始函数, `net.initialize()`, 来初始化模型参数。我们也同时介绍过如何访问模型参数的简单方法。这一节我们将深入讲解模型参数的访问和初始化, 以及如何在多个层之间共享同一份参数。

我们首先定义同前的多层感知机、初始化权重和计算前向结果。与之前不同的是, 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实例的字典。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。但由于我们还没有进行反向传播计算, 所以它的值全为 0.

```
In [5]: net[0].weight.grad()  
Out[5]:  
[[ 0.  0.  0. ...,  0.  0.  0.]  
 [ 0.  0.  0. ...,  0.  0.  0.]  
 [ 0.  0.  0. ...,  0.  0.  0.]  
 ... ,  
 [ 0.  0.  0. ...,  0.  0.  0.]  
 [ 0.  0.  0. ...,  0.  0.  0.]
```

```
[ 0.  0.  0. ...,  0.  0.  0.]]  
<NDArray 256x20 @cpu(0)>
```

类似我们可以访问其他的层的参数。例如输出层的偏差权重：

```
In [6]: net[1].bias.data()  
  
Out[6]:  
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.]  
<NDArray 10 @cpu(0)>
```

最后，我们可以 `collect_params` 函数来获取 `net` 实例所有嵌套（例如通过 `add` 函数嵌套）的层所包含的所有参数。它返回的同样是一个参数名称到参数实例的字典。

```
In [7]: net.collect_params()  
  
Out[7]: sequential0_ (  
    Parameter dense0_weight (shape=(256, 20), dtype=float32)  
    Parameter dense0_bias (shape=(256,), dtype=float32)  
    Parameter dense1_weight (shape=(10, 256), dtype=float32)  
    Parameter dense1_bias (shape=(10,), dtype=float32)  
)
```

4.2.2 初始化模型参数

当使用默认的模型初始化，Gluon 会将权重参数元素初始化为 [-0.07, 0.07] 之间均匀分布的随机数，偏差参数则全为 0。但经常我们需要使用其他的方法来初始化权重，MXNet 的 `init` 模块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 初始化方法。

```
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，我们可以实现一个 `Initializer` 类的子类使得我们可以跟前面使用 `init.Normal` 那样使用它。通常，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)>
```

此外，我们还可以通过 `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]: net = nn.Sequential()  
        shared = nn.Dense(8, activation='relu')  
        net.add(nn.Dense(8, activation='relu'),  
                shared,  
                nn.Dense(8, activation='relu', params=shared.params),  
                nn.Dense(10))  
        net.initialize()  
  
        x = nd.random.uniform(shape=(2,20))  
        net(x)  
  
        net[1].weight.data()[0] == net[2].weight.data()[0]  
  
Out[12]:  
[ 1.  1.  1.  1.  1.  1.  1.]  
<NDArray 8 @cpu(0)>
```

我们在构造第三隐藏层时通过 `params` 来指定它使用第二隐藏层的参数。由于模型参数里包含了梯度，所以在反向传播计算时，第二隐藏层和第三隐藏层的梯度都会被累加在 `shared.params.grad()` 里。

4.2.5 小结

- 我们有多种方法来访问、初始化和共享模型参数。

4.2.6 练习

- 查阅MXNet 文档，了解不同的参数初始化方式。
- 尝试在 `net.initialize()` 后和 `net(x)` 前访问模型参数，看看会发生什么。
- 构造一个含共享参数层的多层感知机并训练。观察每一层的模型参数和梯度计算。

4.2.7 扫码直达讨论区



4.3 模型参数的延后初始化

如果你注意到了上节练习，你会发现在 `net.initialize()` 后和 `net(x)` 前模型参数的形状都是空。直觉上 `initialize` 会完成了所有参数初始化过程，然而 Gluon 中这是不一定。我们这里详细讨论这个话题。

4.3.1 延后的初始化

注意到前面使用 Gluon 的章节里，我们在创建全连接层时都没有指定输入大小。例如在一直使用的多层感知机例子里，我们创建了输出大小为 256 的隐藏层。但是当在调用 `initialize` 函数的时候，我们并不知道这个层的参数到底有多大，因为它的输入大小仍然是未知。只有在当我们把形状是 $(2, 20)$ 的 `x` 输入进网络时，我们这时候才知道这一层的参数大小应该是 $(256, 20)$ 。所以这个时候我们才能真正开始初始化参数。

让我们使用上节定义的 `MyInit` 类来清楚的演示这一个过程。下面我们创建多层感知机，然后使用 `MyInit` 实例来进行初始化。

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

```
class MyInit(init.Initializer):
    def __init_weight(self, name, data):
        print('Init', name, data.shape)
        # 实际的初始化逻辑在此省略了。

net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))

net.initialize(init=MyInit())
```

注意到 `MyInit` 在调用时会打印信息，但当前我们并没有看到相应的日志。下面我们执行前向计算。

```
In [2]: x = nd.random.uniform(shape=(2,20))
y = net(x)

Init dense0_weight (256, 20)
Init dense1_weight (10, 256)
```

这时候系统根据输入 `x` 的形状自动推测数所有层参数形状，例如隐藏层大小是 `(256, 20)`，并创建参数。之后调用 `MyInit` 实例来进行初始方法，然后再进行前向计算。

当然，这个初始化只会在第一次执行被调用。之后我们再运行 `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时候称之为延后初始化。它可以让模型创建更加简单，因为我们只需要定义每个层的输出大小，而不用去推测它们的输入大小。这个对于之后将介绍的多达数十甚至数百层的网络尤其有用。

当然正如本节开头提到那样，延后初始化也可能会造成一定的困解。在调用第一次前向计算之前我们无法直接操作模型参数。例如无法使用 `data` 和 `set_data` 函数来获取和改写参数。所以经常我们会额外调用一次 `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“[模型参数的访问、初始化和共享](#)”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]: # units: 该层的输出个数; in_units: 该层的输入个数。
dense = MyDense(units=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 NDArray 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)
```

下面我们把该模型的参数存起来。

```
In [6]: filename = '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(filename)
```

因为这两个实例都有同样的参数，那么对同一个 x 的计算结果将会是一样。

```
In [8]: y2 = net2(x)  
y2 == y  
  
Out[8]:  
[[ 1.  1.  1.  1.  1.  1.  1.  1.  1.]  
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.]]  
<NDArray 2x10 @cpu(0)>
```

4.5.3 小结

- 通过 `save` 和 `load` 可以很方便地读写 `NDArray`。
- 通过 `load_params` 和 `save_params` 可以很方便地读写 Gluon 的模型参数。

4.5.4 练习

- 即使无需把训练好的模型部署到不同的设备，存储模型参数在实际中还有哪些好处？

4.5.5 扫码直达讨论区



4.6 GPU 计算

目前为止我们一直在使用 CPU 计算。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 GPU 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

Tue Jul 3 23:07:54 2018

NVIDIA-SMI 375.26				Driver Version: 375.26		
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr. ECC
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.
<hr/>						
0	Tesla M60	On	0000:00:1D.0	Off	0%	Default
N/A	54C	P0	48W / 150W	298MiB / 7612MiB	0%	Default
<hr/>						
1	Tesla M60	On	0000:00:1E.0	Off	0%	Default
N/A	60C	P0	39W / 150W	289MiB / 7612MiB	0%	Default
<hr/>						
<hr/>						
Processes:				GPU Memory		
GPU	PID	Type	Process name	Usage		
<hr/>						

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()`（或者在括号里填任意整数）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) 且 `mx.gpu(0)` 和 `mx.gpu()` 等价。

```
In [2]: import mxnet as mx
from mxnet import nd
from mxnet.gluon import nn

[mx.cpu(), mx.gpu(), mx.gpu(1)]
```

```
Out[2]: [cpu(0), gpu(0), gpu(1)]
```

4.6.2 NDArray 的 GPU 计算

默认情况下，NDArray 存在 CPU 上。因此，之前我们每次打印 NDArray 的时候都会看到 `@cpu(0)` 这个标识。

```
In [3]: x = nd.array([1,2,3])
x

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

我们可以通过 NDArray 的 `context` 属性来查看其所在的设备。

```
In [4]: x.context
Out[4]: cpu(0)
```

GPU 上的存储

我们有多种方法将 NDArray 放置在 GPU 上。例如我们可以在创建 NDArray 的时候通过 `ctx` 指定存储设备。下面我们将 `a` 创建在 GPU 0 上。注意到在打印 `a` 时，设备信息变成了 `@gpu(0)`。创建在 GPU 上时我们会只用 GPU 内存，你可以通过 `nvidia-smi` 查看 GPU 内存使用情况。通常你需要确保不要创建超过 GPU 内存上限的数据。

```
In [5]: a = nd.array([1, 2, 3], ctx=mx.gpu())
a

Out[5]:
[ 1.  2.  3.]
<NDArray 3 @gpu(0)>
```

假设你至少有两块 GPU，下面代码将会在 GPU 1 上创建随机数组

```
In [6]: b = nd.random.uniform(shape=(2, 3), ctx=mx.gpu(1))
b
Out[6]:
[[ 0.59118998  0.313164    0.76352036]
 [ 0.97317863  0.35454726  0.11677533]]
<NDArray 2x3 @gpu(1)>
```

除了在创建时指定，我们也可以通过 `copyto` 和 `as_in_context` 函数在设备之间传输数据。下面我们将 CPU 上的 `x` 复制到 GPU 0 上。

```
In [7]: y = x.copyto(mx.gpu())
y
Out[7]:
[ 1.  2.  3.]
<NDArray 3 @gpu(0)>
In [8]: z = x.as_in_context(mx.gpu())
z
Out[8]:
[ 1.  2.  3.]
<NDArray 3 @gpu(0)>
```

需要区分的是，如果源变量和目标变量的 `context` 一致，`as_in_context` 使目标变量和源变量共享源变量的内存，

```
In [9]: y.as_in_context(mx.gpu()) is y
Out[9]: True
```

而 `copyto` 总是为目标变量新创建内存。

```
In [10]: y.copyto(mx.gpu()) is y
Out[10]: False
```

GPU 上的计算

MXNet 的计算会在数据的 `context` 上执行。为了使用 GPU 计算，我们只需要事先将数据放在 GPU 上面。而计算结果会自动保存在相同的 GPU 上。

```
In [11]: (z + 2).exp() * y
Out[11]:
[ 20.08553696 109.19629669 445.23950195]
<NDArray 3 @gpu(0)>
```

注意, MXNet 要求计算的所有输入数据都在同一个 CPU/GPU 上。这个设计的原因是不同 CPU/GPU 之间的数据交互通常比较耗时。因此, MXNet 希望用户确切地指明计算的输入数据都在同一个 CPU/GPU 上。例如, 如果将 CPU 上的 x 和 GPU 上的 y 做运算, 会出现错误信息。

当我们打印 NDArray 或将 NDArray 转换成 NumPy 格式时, 如果数据不在主内存里, MXNet 会自动将其先复制到主内存, 从而带来隐形的传输开销。

4.6.3 Gluon 的 GPU 计算

同 NDArray 类似, Gluon 的模型可以在初始化时通过 `ctx` 指定设备。下面代码将模型参数初始化在 GPU 上。

```
In [12]: net = nn.Sequential()
net.add(nn.Dense(1))
net.initialize(ctx=mx.gpu())
```

当输入是 GPU 上的 NDArray 时, Gluon 会在相同的 GPU 上计算结果。

```
In [13]: net(y)

Out[13]:
[[ 0.0068339 ]
 [ 0.01366779]
 [ 0.02050169]]
<NDArray 3x1 @gpu(0)>
```

确认一下模型参数存储在相同的 GPU 上。

```
In [14]: net[0].weight.data()

Out[14]:
[[ 0.0068339]]
<NDArray 1x1 @gpu(0)>
```

4.6.4 小结

- 通过 `context`, 我们可以在不同的 CPU/GPU 上存储数据和计算。

4.6.5 练习

- 试试大一点的计算任务，例如大矩阵的乘法，看看 CPU 和 GPU 的速度区别。如果是计算量很小的任务呢？
- GPU 上应如何读写模型参数？

4.6.6 扫码直达讨论区



卷积神经网络

本章我们介绍卷积神经网络 (convolutional neural network)。它是近年来深度学习能在计算机视觉中取得巨大成果的基石，它也逐渐在被其他诸如自然语言处理、推荐系统和语音识别等领域广泛使用。这一章我们将讲解卷积神经网络的原理，并介绍过去几年中数个在 ImageNet 竞赛（一个著名的计算机视觉竞赛）取得优异成绩的深度卷积神经网络。

5.1 二维卷积层

卷积神经网络是指主要由卷积层 (convolutional layer) 组成的网络。因为它最常用来处理图片数据，其有高和宽两个空间维度（彩色图片的颜色通道维度将在之后小节讨论），所以最常用到的是二维卷积层。本小节本节我们将介绍简单形式的二维卷积层的是怎么工作的。

5.1.1 二维相关运算符

虽然卷积层得名于卷积运算符 (convolution)，但我们常用更加直观的相关运算符 (correlation) 来实现卷积层。一个二维相关运算符将一个二维核 (kernel) 数组作用在一个二维输入数据上来

计算一个二维数组输出。下图演示了如何对一个高宽为 3 的输入 X 作用高宽为 2 的核 K 来计算输出 Y 。

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 19 & 25 \\ \hline 37 & 43 \\ \hline \end{array}$$

图 5.1: 二维相关运算符, 高亮了计算第一个输出元素所使用的输入和核数组元素。

可以看到输出 Y 的形状是 $(2, 2)$, 且第一个元素是由 X 的左上的高宽为 2 的子数组与核数组按元素相乘后再相加得来。即 $Y[0, 0] = (X[0:2, 0:2] * K).sum()$, 这里 X, K 和 Y 的类型都是 NDArray。接下来我们将这个高宽为 2 的窗口在 X 上向左滑动一列来计算 Y 的第二列第一个元素。以此类推计算得到所有结果。

下面我们将上述过程实现在 `corr2d` 函数里, 它接受 X 和 K , 输出 Y 。

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

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

构造上图中的数据来测试实现的正确性。

```
In [2]: X = nd.array([[0,1,2], [3,4,5], [6,7,8]])
        K = nd.array([[0,1], [2,3]])
        corr2d(X, K)
```

```
Out[2]:
[[ 19.  25.]
 [ 37.  43.]]
<NDArray 2x2 @cpu(0)>
```

5.1.2 二维卷积层

二维卷积层就是将输入和其维护的核数组，也称作卷积核，做相关运算，然后加上一个标量偏差来得到输出。它的模型参数包括了卷积核和标量偏差。在训练的时候，我们通常首先对卷积核进行随机初始化，然后不断迭代更新卷积核和偏差来拟合数据。

下面的我们基于 `corr2d` 函数来实现一个自定义的二维卷积层。在初始化函数里我们声明 `weight` 和 `bias` 这两个模型参数，前向计算函数则是直接调用 `corr2d` 再加上偏差。

```
In [3]: class Conv2D(nn.Block):
    def __init__(self, kernel_size, **kwargs):
        super(Conv2D, self).__init__(**kwargs)
        self.weight = self.params.get('weight', shape=kernel_size)
        self.bias = self.params.get('bias', shape=(1,))

    def forward(self, x):
        return corr2d(x, self.weight.data()) + self.bias.data()
```

你也许会好奇既然称之为卷积层，为什么不使用卷积运算符呢？其实卷积运算的计算与二维相关运算类似，唯一的区别是反向的将核数组跟输入做乘法，即 $Y[0, 0] = (X[0:2, 0:2] * K[::-1, ::-1]).sum()$ 。但是因为在卷积层里 K 是学习而来的，所以不论是正向还是反向访问都可以。

5.1.3 图片物体边缘检测

下面我们来看一个应用卷积层的简单应用：检测图片中物体的边缘，即找到像素变化的位置。首先我们构造一张 6×8 的图，它中间 4 列为黑（0），其余为白（1）。

```
In [4]: X = nd.ones((6, 8))
X[:, 2:6] = 0
X

Out[4]:
[[ 1.  1.  0.  0.  0.  0.  1.  1.]
 [ 1.  1.  0.  0.  0.  0.  1.  1.]
 [ 1.  1.  0.  0.  0.  0.  1.  1.]
 [ 1.  1.  0.  0.  0.  0.  1.  1.]
 [ 1.  1.  0.  0.  0.  0.  1.  1.]
 [ 1.  1.  0.  0.  0.  0.  1.  1.]]
```

然后我们构造一个形状为 $(1, 2)$ 的卷积核，使得其作用在相同的横向相邻元素上输出为 0，否则输出非 0。

```
In [5]: K = nd.array([[1, -1]])
```

对 X 作用我们设计的核 K 后可以发现，从白到黑的边缘我们检测成了 1，从黑到白则是 -1，其余全是 0。

```
In [6]: Y = corr2d(X, K)
Y
```

Out[6]:

```
[[ 0.  1.  0.  0.  0. -1.  0.]
 [ 0.  1.  0.  0.  0. -1.  0.]
 [ 0.  1.  0.  0.  0. -1.  0.]
 [ 0.  1.  0.  0.  0. -1.  0.]
 [ 0.  1.  0.  0.  0. -1.  0.]
 [ 0.  1.  0.  0.  0. -1.  0.]]
<NDArray 6x7 @cpu(0)>
```

这里我们可以看到卷积层通过重复的使用 K 来有效的发掘局部空间特征。

5.1.4 通过数据学习核数组

最后我们来看一个例子，它使用前面的 X 和 Y 来学习我们构造的 K 。我们首先构造一个卷积层，将其卷积核初始化成随机数组。然后在每一个迭代里，我们使用平方误差来比较 Y 和卷积层的输出，然后计算梯度来更新权重。

虽然我们之前构造了 Conv2D 类，但由于 `corr2d` 使用了对单个元素赋值 ($[i, j] =$) 的操作会导致无法自动求导，下面我们使用 Gluon 提供的 Conv2D 类来实现这个例子。

```
In [7]: # 构造一个输出通道是 1 (将在后面小节介绍通道)，核数组形状是 (1, 2) 的二维卷积层。
conv2d = nn.Conv2D(1, kernel_size=(1, 2))
conv2d.initialize()

# 二维卷积层使用 4 维输入输出，格式为 (批量大小, 通道数, 高, 宽)，这里批量和通道均为 1。
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))

for i in range(10):
    with autograd.record():
        Y_hat = conv2d(X)
        loss = (Y_hat - Y) ** 2
    if i % 2 == 1:
```

```
        print('batch %d, loss %.3f' % (i, loss.sum().asscalar()))
    loss.backward()
    # 为了简单起见这里忽略了偏差。
    conv2d.weight.data()[:] -= 3e-2 * conv2d.weight.grad()

batch 1, loss 4.949
batch 3, loss 0.831
batch 5, loss 0.140
batch 7, loss 0.024
batch 9, loss 0.004
```

可以看到 10 次迭代后误差已经降到了一个比较小的值，现在来看一下学习到的核。

In [8]: conv2d.weight.data().reshape((1,2))

Out[8]:

```
[[ 0.98949999 -0.98737049]]
<NDArray 1x2 @cpu(0)>
```

我们看到学到的核与我们之前定义的 K 非常接近。

5.1.5 小结

- 二维卷积层的核心计算是二维相关运算。在最简单的形式下，它对二维输入数据和卷积核做相关运算然后加上偏差。
- 我们可以设计卷积核来检测图片中的边缘，同时也可以通过数据来学习它。

5.1.6 练习

- 构造一个 X ，它有水平方向的边缘，如何设计 K 来检测它？如果是对角方向的边缘呢？
- 试着对我们构造的 Conv2D 进行自动求导，会有什么样的错误信息？
- 在 Conv2D 的 forward 函数里，将 corr2d 替换成 nd.Convolution 使得其可以求导。
- 试着将 conv2d 的核构造成 $(2, 2)$ ，会学出什么样的结果？
- 如何通过变化输入和核的矩阵来将相关运算表示成一个矩阵乘法。
- 如何构造一个全连接层来进行物体边缘检测？

5.1.7 扫码直达讨论区



5.2 填充和步幅

在上一节的例子里，我们使用高宽为 3 的输入和高宽为 2 的卷积核得到高宽为 2 的输出。一般来说，假设输入形状是 $n_h \times n_w$ ，卷积核形状是 $k_h \times k_w$ ，那么输出形状将会是

$$(n_h - k_h + 1) \times (n_w - k_w + 1).$$

所以卷积层的输出形状由输入形状和卷积核形状决定。这一节我们将介绍卷积层的两个超参数，填充和步幅，它们可以在给定形状的输入和卷积核下来改变输出形状。

5.2.1 填充

填充是指在输入高和宽的两端填充元素。下图里我们在宽和高的两侧分别添加了值为 0 的元素，使得输入高宽从 2 变成了 5，从而导致输出高宽由 2 增加到 4。

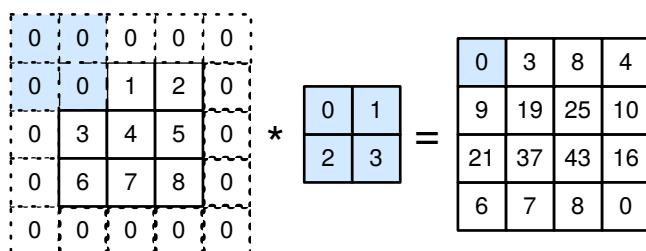


图 5.2: 在输入的高和宽两侧分别填充了 0 的二维相关计算。

一般来说，如果在高两侧一共填充 p_h 行，在宽两侧一共填充 p_w 列，那么输出形状将会是

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1),$$

也就是说输出的高宽分别会增加 p_h 和 p_w 。

通常我们会设置 $p_h = k_h - 1$ 和 $p_w = k_w - 1$ 使得输入和输出有相同的高宽，这样方便在构造网络时容易推测每个层的输出形状。假设这里 k_h 是奇数，我们会在高的两侧分别填充 $p_h/2$ 行。如果其是偶数，一种可能是上面填充 $\lceil p_h/2 \rceil$ 行，而下面填充 $\lfloor p_h/2 \rfloor$ 行。在宽上行为类似。

卷积神经网络经常使用奇数高宽的卷积核，例如 1、3、5、和 7，所以填充在两端上是对称的。这样还有一点方便的是我们知道输出 $Y[i, j]$ 是由输入以 $X[i, j]$ 为中心的窗口同卷积核进行相关计算得来。

下面例子里我们创建一个核高宽为 3 的二维卷积层，然后在输入高和宽的两侧分别填充 1。输入一个高宽为 8 的输入，我们会发现输出的高宽也是 8。

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

        # 定义一个便利函数来计算卷积层。它初始化卷积层权重，并对输入和输出做相应的升维和降维。
        def comp_conv2d(conv2d, X):
            conv2d.initialize()
            X = X.reshape((1, 1,) + X.shape)
            Y = conv2d(X)
            return Y.reshape(Y.shape[2:])

        X = nd.random.uniform(shape=(8, 8))

        # 注意这里是两侧分别填充 1，所以  $p_w = p_h = 2$ 。
        conv2d = nn.Conv2D(1, kernel_size=3, padding=1)
        comp_conv2d(conv2d, X).shape
```

Out[1]: (8, 8)

当然我们可以使用非方形卷积核，使用对应的填充同样可得相同高宽的输出。

```
In [2]: # 使用高为 5，宽为 3 的卷积核。
        conv2d = nn.Conv2D(1, kernel_size=(5, 3), padding=(2, 1))
        comp_conv2d(conv2d, X).shape
```

Out[2]: (8, 8)

5.2.2 步幅

在上一节里我们介绍了输出元素是如何计算而来。我们使用一个和卷积核有相同高宽的窗口，从输入的左上角开始，每次往左滑动一列或者往下滑动一行逐一计算输出。我们将每次滑动的行数

和列数称之为步幅。

目前我们看到的例子里，在高和宽两个方向上步幅均为 1。自然我们可以使用更大步幅。下图展示了在高上使用步幅 3，在宽上使用步幅 2 的情况。可以看到，计算第一列第二个元素时，窗口向下滑动了三行。而在计算输出的第一行第二个元素时窗口向右滑动了两列，再向右滑动两列时只剩一列输入元素从而填不满窗口，我们将其结果舍弃。

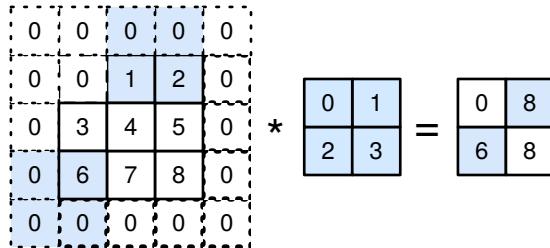


图 5.3: 高上使用步幅 3，宽上使用步幅 2。

一般来说，如果在高上使用步幅 s_h ，在宽上使用步幅 s_w ，那么输出大小将是

$$\lfloor (n_h - k_h + p_h + s_h) / s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w) / s_w \rfloor.$$

如果我们设置 $p_h = k_h - 1$ 和 $p_w = k_w - 1$ ，那么输出大小简化为 $\lfloor (n_h + s_h - 1) / s_h \rfloor \times \lfloor (n_w + s_w - 1) / s_w \rfloor$ 。更进一步，如果输出高宽能分别被高宽上的步幅整除，那么输出将是 $n_h / s_h \times n_w / s_w$ 。也就是说我们成倍的减小了输入的高宽。

下面我们看一个符合简化形状公式的例子，其将高宽减半。

```
In [3]: conv2d = nn.Conv2D(1, kernel_size=3, padding=1, strides=2)
comp_conv2d(conv2d, X).shape
```

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

接下来是一个稍微复杂点的例子。

```
In [4]: conv2d = nn.Conv2D(1, kernel_size=(3,5), padding=(0,1), strides=(3,4))
comp_conv2d(conv2d, X).shape
```

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

5.2.3 小结

- 通过填充可以增加输出的高宽，常用来使得输出与输入同高宽。通过步幅可以成倍的减少输出的高宽。

5.2.4 练习

- 对最后一个例子通过形状计算公式来计算其输出形状。
- 试一试其他的填充和步幅组合。

5.2.5 扫码直达讨论区



5.3 多输入和输出通道

前面小节里我们用到的输入和输出都是二维数组，但实际数据的维度经常更高。例如彩色图片在高宽两个维度外还有 RGB 这三个通道。假设它的高和宽分别是 h 和 w （像素），那么内存中它可以被表示成一个 $3 \times h \times w$ 的多维数组。我们将大小为 3 的这一维称之为通道（channel）。这一节我们将介绍卷积层是如何处理多输入通道，和以及计算多通道输出。

5.3.1 多输入通道

假设输入通道数是 c_i ，且卷积核窗口为 $k_h \times k_w$ 。当 $c_i = 1$ 时，我们知道卷积核就是一个 $k_h \times k_w$ 数组。当其大于 1 时，我们将会为每个输入通道分配一个单独的 $k_h \times k_w$ 核数组。我们将这些数组合并起来，将得到一个 $c_i \times k_h \times k_w$ 形状的卷积核。然后在每个通道里对相应的输入矩阵和核矩阵做相关计算，然后再将通道之间的结果相加得到最终结果。下图展示了输入通道是 2 的一个例子。

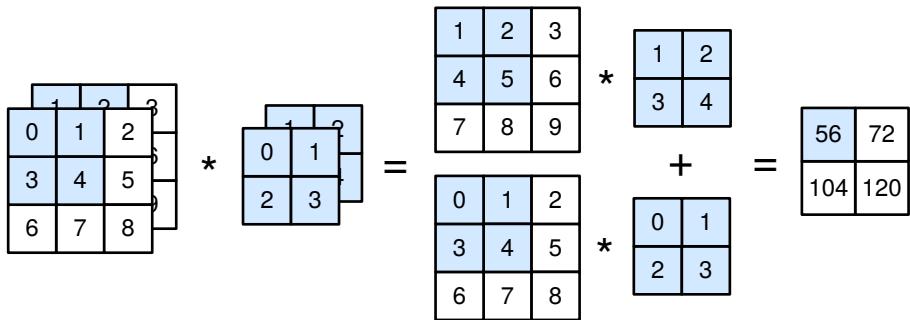


图 5.4: 输入通道为 2 的二维相关计算。

接下来我们实现处理多输入通道的相关运算符。首先我们将前面小节实现的 `corr2d` 复制过来。

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

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

为了实现多输入通道的版本，我们只需要对每个通道做相关计算，然后通过 `nd.add_n` 来进行累加。

```
In [2]: def corr2d_multi_in(X, K):
    # 我们首先沿着 X 和 K 的第 0 维（通道维）遍历。然后使用 * 将结果列表（list）变成
    # add_n 的位置参数（positional argument）来进行相加。
    return nd.add_n(*[corr2d(x, k) for x, k in zip(X, K)])
```

我们构造上图中的输入数据来验证实现的正确性。

```
In [3]: X = nd.array([[0,1,2], [3,4,5], [6,7,8]],
                  [[1,2,3], [4,5,6], [7,8,9]]])
K = nd.array([[0,1], [2,3]], [[1,2], [3,4]])

corr2d_multi_in(X, K)

Out[3]:
[[ 56.   72.]
```

```
[ 104. 120.]  
<NDArray 2x2 @cpu(0)>
```

5.3.2 多输出通道

在多输入通道下，由于我们对各个通道结果做了累加，因此不论输入通道数是多少，输出通道总是为 1。如果想得到 $c_o > 1$ 通道的输出，我们为每个输出通道创建单独的 $c_i \times k_h \times k_w$ 形状的核数组。将它们合并起来，那么卷积核的形状是 $c_o \times c_i \times k_h \times k_w$ 。在计算的时候，每个输出通道的数据由整个输入数据和对应的核矩阵计算得来。其实现见下面代码。

```
In [4]: def corr2d_multi_in_out(X, K):  
    # 对 K 的第 0 维遍历，每次同输入 X 做相关计算。所有结果使用 nd.stack 合并在一起。  
    return nd.stack(*[corr2d_multi_in(X, k) for k in K])
```

我们将三维核矩阵 K 同 $K+1$ 和 $K+2$ 拼在一起构造一个输出通道为 3 的四维卷积核。

```
In [5]: K = nd.stack(K, K+1, K+2)  
K.shape  
  
Out[5]: (3, 2, 2, 2)
```

然后计算它的输出。可以发现计算结果有三个通道，其中第一个通道跟上例中输出一致。

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

```
Out[6]:  
[[[ 56. 72.]  
 [ 104. 120.]  
  
 [[ 76. 100.]  
 [ 148. 172.]  
  
 [[ 96. 128.]  
 [ 192. 224.]]]]  
<NDArray 3x2x2 @cpu(0)>
```

5.3.3 1×1 卷积层

最后我们讨论卷积窗口为 1×1 ($k_h = k_w = 1$) 的多通道卷积层。因为使用了最小窗口，它失去了卷积层可以识别高宽维上相邻元素构成的模式的功能，它的主要计算则是在通道维上。下图展示了输入通道为 3 和输出通道为 2 的情况。输出中的每个元素来自输入中对应位置的元素在不同通道之间的按权重累加。

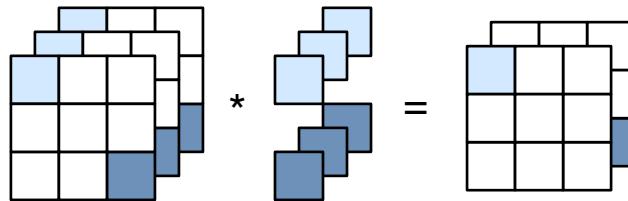


图 5.5: 多输入通道的 1×1 卷积层

假设我们将通道维当做是特征维，而高宽中的元素则当成数据点。那么 1×1 卷积层则等价于一个全连接层。下面代码里我们将输入和卷积核变形为二维数组，然后使用矩阵乘法来计算输出，之后再变形回我们需要的样子。

```
In [7]: def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = X.reshape((c_i, h*w))
    K = K.reshape((c_o, c_i))
    Y = nd.dot(K, X)
    return Y.reshape((c_o, h, w))
```

生成一组随机数来验证实现的正确性。

```
In [8]: X = nd.random.uniform(shape=(3,3,3))
K = nd.random.uniform(shape=(2,3,1,1))

Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)

(Y1-Y2).norm().asscalar() < 1e-6
```

Out[8]: True

在之后的模型里我们将会看到 1×1 卷积层是如何当做保持高宽维形状的全连接层使用，其作用是通过调整网络层之间的通道数来控制模型复杂度。

5.3.4 小结

- 使用多通道可以极大拓展卷积层的模型参数。
- 1×1 卷积层通常用来调节网络层之间的通道数。

5.3.5 练习

- 假设输入大小为 $c_i \times h \times w$, 且使用 $c_o \times c_i \times k_h \times k_w$ 卷积核, 填充为 (p_h, p_w) 以及步幅为 (s_h, s_w) , 那么这个卷积层的前向计算需要多少次乘法, 多少次加法?
- 翻倍输入通道 c_i 和输出通道 c_o 会增加多少倍计算? 翻倍填充呢?
- 如果使用 $k_h = k_w = 1$, 能降低多少倍计算?
- 例子中 Y_1 和 Y_2 结果完全一致吗? 原因是什么?
- 对于非 1×1 卷积层, 如何也将其表示成一个矩阵乘法。

5.3.6 扫码直达讨论区



5.4 池化层

回忆在“卷积层”小节里介绍的图片物体边缘检测应用中, 我们构造了卷积核来精确的找到像素变化的位置。例如如果输出 $Y[i, j]=1$, 那么表示 $X[i, j]$ 和 $X[i, j+1]$ 数值不一样, 这可能意味着物体边缘通过这两个元素之间。但实际中图片里我们感兴趣的物体不会总出现在固定位置, 即使我们连续拍摄同一个物体也极有可能出现像素上的偏移。这样导致同一个边缘对应的输出可能出现在 Y 中不同位置, 进而对后面的模式识别造成不便。

这一节我们介绍池化层 (pooling layer), 它的提出是为了缓解卷积层对位置的过度敏感性。

5.4.1 二维最大、平均池化层

池化层同卷积层一样每次对输入数据的一个固定形状窗口元素计算输出。不同于卷积层里计算输入和核相关性, 池化层直接计算窗口内元素的最大值或者平均值。下图展示了 2×2 最大池化层,

其输出的第一个元素是输入的左上 2×2 窗口里的四个元素的最大值。然后同卷积层一样依次向左或向下移动窗口来计算其余的输出。

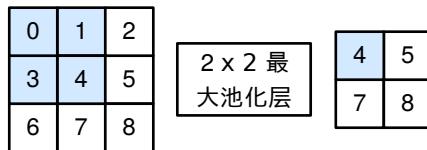


图 5.6: 2×2 最大池化层。

如果这个池化层的输入是来自于前面我们构造了卷积核的卷积层的输出。假设此卷积层输入是 X , 那么不管是 $X[i, j]$ 和 $X[i, j+1]$ 值不同, 还是 $X[i, j+1]$ 和 $X[i, j+2]$ 不同, 池化层输出均有 $Y[i, j]=1$ 。换句话说, 使用 2×2 最大池化层, 只要卷积层识别的模式在高和宽上移动不超过一个元素, 我们均可以将其检测出来。

池化层的前向计算实现在 `pool2d` 函数里。它跟“卷积层”一节里 `corr2d` 函数非常类似, 唯一的区别是在计算 $Y[h, w]$ 上。

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

def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = nd.zeros((X.shape[0]-p_h+1, X.shape[1]-p_w+1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i:i+p_h, j:j+p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i:i+p_h, j:j+p_w].mean()
    return Y
```

构造上图中的数据来验证实现的正确性。

```
In [2]: X = nd.array([[0,1,2], [3,4,5], [6,7,8]])
pool2d(X, (2,2))

Out[2]:
[[ 4.  5.]
 [ 7.  8.]]
<NDArray 2x2 @cpu(0)>
```

同时我们试一试平均池化层。

```
In [3]: pool2d(X, (2,2), 'avg')
```

```
Out[3]:
```

```
[[ 2.  3.]
 [ 5.  6.]]
<NDArray 2x2 @cpu(0)>
```

5.4.2 填充和步幅

同卷积层一样，池化层也可以填充输入高宽两侧的数据和调整窗口的移动步幅来改变输入大小。我们将通过 `nn` 模块里的二维最大池化层 `MaxPool2D` 来演示它的工作机制。我们先构造一个 $(1, 1, 4, 4)$ 形状的输入数据，前两个维度分别是批量和通道。

```
In [4]: X = nd.arange(16).reshape((1, 1, 4, 4))
X
```

```
Out[4]:
```

```
[[[[ 0.   1.   2.   3.]
   [ 4.   5.   6.   7.]
   [ 8.   9.  10.  11.]
   [ 12.  13.  14.  15.]]]]
<NDArray 1x1x4x4 @cpu(0)>
```

`MaxPool2D` 类里默认步幅设置成跟池化窗大小一样。下面使用 $(3, 3)$ 窗口，默认获得 $(3, 3)$ 步幅。

```
In [5]: pool2d = nn.MaxPool2D(3)
# 因为池化层没有模型参数，所以不需要调用参数初始化函数。
pool2d(X)
```

```
Out[5]:
```

```
[[[[ 10.]]]]
<NDArray 1x1x1x1 @cpu(0)>
```

我们可以手动指定步幅和填充。

```
In [6]: pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)
```

```
Out[6]:
```

```
[[[[ 5.   7.]
   [ 13.  15.]]]]
<NDArray 1x1x2x2 @cpu(0)>
```

当然，我们也可以是非方形的窗口，并且指定各个方向上的填充和步幅。

```
In [7]: pool2d = nn.MaxPool2D((2,3), padding=(1,2), strides=(2,3))
pool2d(X)

Out[7]:
[[[[ 0.  3.]
   [ 8. 11.]
   [12. 15.]]]]
<NDArray 1x1x3x2 @cpu(0)>
```

5.4.3 多通道

在处理多通道输入数据时，池化层对每个输入通道分别池化，而不是像卷积层那样混合输入通道。这个意味着池化层的输出通道跟输入通道数相同。下面我们将 X 和 $X+1$ 在通道维度上合并来构造通道数为 2 输入。

```
In [8]: X = nd.concat(X, X+1, dim=1)
X

Out[8]:
[[[[ 0.  1.  2.  3.]
   [ 4.  5.  6.  7.]
   [ 8.  9. 10. 11.]
   [12. 13. 14. 15.]]

  [[ 1.  2.  3.  4.]
   [ 5.  6.  7.  8.]
   [ 9. 10. 11. 12.]
   [13. 14. 15. 16.]]]]
<NDArray 1x2x4x4 @cpu(0)>
```

做池化后我们发现输出通道仍然是 2，而且通道 0 的结果跟之前一致。

```
In [9]: pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)

Out[9]:
[[[[ 5.  7.]
   [13. 15.]]

  [[ 6.  8.]
   [14. 16.]]]]
<NDArray 1x2x2x2 @cpu(0)>
```

5.4.4 小结

- 池化层通过滑动窗口计算结果，其通常直接取输入窗口内元素的最大值或者平均值作为输出。
- 池化层的一个主要作用是缓解卷积层对位置的敏感性。

5.4.5 练习

- 分析池化层的计算复杂度。假设输入大小为 $c \times h \times w$ ，我们使用 $p_h \times p_w$ 的池化窗，而且使用 (p_h, p_w) 填充和 (s_h, s_w) 步幅，那么这个池化层的前向计算需要多少操作？
- 想一想最大池化层和平均池化层的区别主要在哪里？
- 你觉得最小池化层这个想法怎么样？

5.4.6 扫码直达讨论区



5.5 卷积神经网络（LeNet）

在“[多层感知机的从零开始实现](#)”一节里我们构造了一个两层感知机模型来对 FashionMNIST 里图片进行分类。每张图片高和宽均是 28 像素。我们将其展开成长为 784 的向量输入到模型里。这样的做法虽然简单，但也有局限性：

1. 垂直方向接近的像素在这个向量的图片表示里可能相距很远，它们组成的模式难被模型识别。
2. 对于大尺寸的输入图片，我们会得到过大的模型。假设输入是高和宽均为 1000 像素的彩色照片，即使隐藏层输出仍是 256，这一层的模型形状是 $3,000,000 \times 256$ ，其占用将近 3GB 的内存，这带来过复杂的模型和过高的存储开销。

卷积层尝试解决这两个问题：它保留输入形状，使得可以有效的发掘水平和垂直两个方向上的数据关联。它通过滑动窗口将卷积核重复作用在输入上，从而得到更紧凑的模型参数表示。

卷积神经网络就是主要由卷积层组成的网络，本小节里我们将介绍一个早期用来识别手写数字图片的卷积神经网络：LeNet [1]，其名字来源于论文第一作者 Yann LeCun。LeNet 证明了通过梯度下降训练卷积神经网络可以达到手写数字识别的最先进的结果。这个奠基性的工作第一次将卷积神经网络推上舞台，为世人所知。

5.5.1 LeNet 模型

LeNet 分为卷积层块和全连接层块两个部分。卷积层块里的基本单位是卷积层后接最大池化层：卷积层用来识别图片里的空间模式，例如线条和物体局部，之后的最大池化层则用来降低卷积层对位置的敏感性。卷积层块由两个这样的基础块重复堆叠构成，即拥有两个卷积层和两个最大池化层。每个卷积层都使用 5×5 的窗口，且在输出上使用 sigmoid 激活函数 $f(x) = \frac{1}{1+e^{-x}}$ 来将输出非线性变换到 $(0, 1)$ 区间。第一个卷积层输出通道为 6，第二个则增加到 16，这是因为其输入高宽比之前卷积层要小，所以增加输出通道来保持相似的模型复杂度。两个最大池化层的窗口均为 2×2 ，且步幅为 2。这意味着每个池化窗口的作用范围都是不重叠的。

卷积层块把每个样本输出拉升成向量输入到全连接层块中。全连接层块由两个输出大小分别为 120 和 84 的全连接层，然后接上输出大小为 10（因为数字的类别一共为 10）的输出层构成。下面我们通过 Sequential 类来实现 LeNet。

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

        net = nn.Sequential()
        net.add(
            nn.Conv2D(channels=6, kernel_size=5, activation='sigmoid'),
            nn.MaxPool2D(pool_size=2, strides=2),
            nn.Conv2D(channels=16, kernel_size=5, activation='sigmoid'),
            nn.MaxPool2D(pool_size=2, strides=2),
            # Dense 会默认将 (批量大小, 通道, 高, 宽) 形状的输入转换成
            # (批量大小, 通道 x 高 x 宽) 形状的输入。
            nn.Dense(120, activation='sigmoid'),
            nn.Dense(84, activation='sigmoid'),
```

```
        nn.Dense(10)
    )

接下来我们构造一个高宽均为 28 的单通道数据点，并逐层进行前向计算来查看每个层的输出大小。
```

```
In [2]: X = nd.random.uniform(shape=(1,1,28,28))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:', X.shape)

conv0 output shape: (1, 6, 24, 24)
pool0 output shape: (1, 6, 12, 12)
conv1 output shape: (1, 16, 8, 8)
pool1 output shape: (1, 16, 4, 4)
dense0 output shape: (1, 120)
dense1 output shape: (1, 84)
dense2 output shape: (1, 10)
```

可以看到在卷积层块中图片的高宽在逐层减小，卷积层由于没有使用填充从而将高宽减少 4，池化层则减半高宽，但通道数则从 1 增加到 16。全连接层则进一步减小输出大小直到变成 10。

5.5.2 获取数据和训练

我们仍然使用 FashionMNIST 作为训练数据。

```
In [3]: train_data, test_data = gb.load_data_fashion_mnist(batch_size=256)
```

因为卷积神经网络计算比多层感知机要复杂，因此我们使用 GPU 来加速计算。我们尝试在 GPU 0 上创建 NDArray，如果成功则使用 GPU 0，否则则使用 CPU。（下面代码将保存在 GluonBook 的 `try_gpu` 函数里来方便重复使用）。

```
In [4]: try:
    ctx = mx.gpu()
    _ = nd.zeros((1,), ctx=ctx)
except:
    ctx = mx.cpu()
ctx
```

```
Out[4]: gpu(0)
```

我们重新将模型参数初始化到 `ctx`，且使用 Xavier [2]（使用论文一作姓氏命名）来进行随机初始化。Xavier 根据每个层的输入输出大小来选择随机数的上下区间，来使得每一层输出有相似的

方差，从而使得训练时数值更加稳定。损失函数和训练算法则使用跟之前一样的交叉熵损失函数和小批量随机梯度下降。

```
In [5]: lr = 1
    net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)

training on gpu(0)
epoch 1, loss 2.3190, train acc 0.100, test acc 0.100, time 1.8 sec
epoch 2, loss 1.9258, train acc 0.253, test acc 0.572, time 1.5 sec
epoch 3, loss 0.9545, train acc 0.620, test acc 0.685, time 1.5 sec
epoch 4, loss 0.7420, train acc 0.710, test acc 0.736, time 1.5 sec
epoch 5, loss 0.6634, train acc 0.739, test acc 0.714, time 1.5 sec
```

5.5.3 小结

- LeNet 交替使用卷积层和最大池化层后接全连接层来进行图片分类。

5.5.4 练习

- LeNet 的设计是针对 MNIST，但在我们这里使用的 FashionMNIST 复杂度更高。尝试基于 LeNet 构造更复杂的网络来改善精度。例如可以考虑调整卷积窗口大小、输出层大小、激活函数和全连接层输出大小。在优化方面，可以尝试使用不同学习率、初始化方法和多使用一些迭代周期。
- 找出 Xavier 的具体初始化方法。

5.5.5 扫码直达讨论区



5.5.6 参考文献

- [1] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.
- [2] Glorot, X., & Bengio, Y. (2010, March). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249-256).

5.6 深度卷积神经网络（AlexNet）

在 LeNet 提出后的将近二十年里，神经网络一度被其他方法（例如支持向量机）超越。虽然 LeNet 可以在 MNIST 上取得到好的成绩，但是在更大的真实数据集上神经网络表现并不佳。一方面神经网络计算复杂，虽然 90 年代也有过一些针对神经网络的加速硬件，但并没有大量普及。因此训练一个多通道、多层和有大量参数的卷积神经网络在当年很难完成。另一方面，当年研究者还没有大量深入研究参数初始化和非凸优化算法等诸多领域，导致复杂的神经网络收敛通常很困难。

即使神经网络可以从原始像素直接预测标签，这种称为端到端（end-to-end）的途径节省了很多中间步骤。但在很长一段时间里更流行的是研究者们通过勤劳智慧和黑魔法生成的手工特征。通常的模式是

1. 找个数据集；
2. 用一堆已有的特征提取函数生成特征；
3. 把这些特征表示放进一个简单的线性模型。

当时认为的机器学习部分仅限最后这一步。如果那时候你跟机器学习研究者们交谈，他们会认为机器学习既重要又优美。优雅的定理证明了许多分类器的性质。机器学习领域生机勃勃、严谨、而且极其有用。然而如果你跟一个计算机视觉研究者交谈，则是另外一幅景象。他们会告诉你图像识别里“不可告人”的现实是，计算机视觉流程中真正重要的是数据和特征。是否有稍微干净点的数据集，或者略微好些的手调特征的差距对最终准确度意味着天壤之别。反观分类器的选择对最终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

在相当长的时间里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也许可以表征整个物体，如人、飞机、狗和飞盘。最终，在分类器层前的隐含层可能可以表征经过汇总的内容，其中不同的类别将会是线性可分的。尽管这群执着的研究者不断钻研，试图学习深度的视觉数据表征，很长的一段时间里这些野心都未能实现，这其中也有诸多因素值得我们一一分析。

缺失要素一：数据

包含许多特征的深度模型需要大量的有标签的数据才能表现得比其他经典方法更好。虽然通过大量累加数据很快就达到了线性模型的上限并在工业界应用，例如广告点击预测里已经被广泛认同，但学术界直到很久以后才普遍认识到这个问题。限于当时计算机有限的存储和相对囊中羞涩的 90 年代研究预算，大部分研究基于小的公开数据集。比如，大部分可信的研究论文是基于 UCI 提供的若干个数据集，其中许多数据集只有几百至几千张图片。

这一状况在 2010 前后兴起的大数据浪潮里得到改善。尤其是 2009 年李飞飞团队贡献了 ImageNet 数据集。它包含了 1000 大类物体，每类有多达数千张不同的图片，这一规模是当时其他公开数据集无法相提并论的。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。毕竟凸优化方法是能很快收敛，并可以找到全局最小值的高效的算法。

GPU 的到来改变了格局。很久以来，GPU 都是为了图像处理和计算机游戏而设计，尤其是针对大吞吐量的矩阵和向量乘法来用于基本的图形转换。值得庆幸的是，这其中的数学表达与深度

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，涌现出诸如 OpenCL 和 CUDA 之类的编程框架。使得 GPU 也在 2010 年前后开始被机器学习社区使用。

5.6.2 AlexNet

2012 年 AlexNet [1]，名字来源于论文一作姓名 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 图像识别挑战赛。它首次证明了学习到的特征可以超越手工设计的特征，从而一举打破计算机视觉研究的前状。

AlexNet 与 LeNet 的设计理念非常相似。但也有非常显著的区别。

1. 与相对较小的 LeNet 相比，AlexNet 包含 8 层变换，其中有五层卷积和两层全连接隐含层，以及一个输出层。

第一层中的卷积窗口是 11×11 。因为 ImageNet 图片高宽均比 MNIST 大十倍以上，对应图片的物体占用更多的像素，所以需要使用更大的窗口来捕获物体。第二层减少到 5×5 ，之后全采用 3×3 。此外，第一，第二和第五个卷积层之后都使用了窗口为 3×3 步幅为 2 的最大池化层。另外，AlexNet 使用的卷积通道数也数十倍大于 LeNet。

紧接着卷积层的是两个输出大小为 4096 的全连接层们。这两个巨大的全连接层带来将近 1GB 的模型大小。由于早期 GPU 显存的限制，最早的 AlexNet 使用双数据流的设计使得一个 GPU 只需要处理一半模型。幸运的是 GPU 内存在过去几年得到了长足的发展，除了一些特殊的结构外，我们也就不再需要这样的特别设计了。

2. 将 sigmoid 激活函数改成了更加简单的 relu 函数 $f(x) = \max(x, 0)$ 。它计算上更简单，同时在不同的参数初始化方法下收敛更加稳定。
3. 通过丢弃法（参见“丢弃法”这一小节）来控制全连接层的模型复杂度。
4. 引入了大量的图片增广，例如翻转、裁剪和颜色变化，进一步扩大数据集来减小过拟合。我们将在后面的“[图片增广](#)”的小节来详细讨论。

下面我们实现（稍微简化过的）Alexnet：

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

        net = nn.Sequential()
```

```

net.add(
    # 使用较大的 11 x 11 窗口来捕获物体。同时使用步幅 4 来较大减小输出高宽。
    # 这里使用的输入通道数比 LeNet 也要大很多。
    nn.Conv2D(96, kernel_size=11, strides=4, activation='relu'),
    nn.MaxPool2D(pool_size=3, strides=2),
    # 减小卷积窗口，使用填充为 2 来使得输入输出高宽一致。且增大输出通道数。
    nn.Conv2D(256, kernel_size=5, padding=2, activation='relu'),
    nn.MaxPool2D(pool_size=3, strides=2),
    # 连续三个卷积层，且使用更小的卷积窗口。除了最后的卷积层外，
    # 进一步增大了输出通道数。前两个卷积层后不使用池化层来减小输入的高宽。
    nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
    nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
    nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'),
    nn.MaxPool2D(pool_size=3, strides=2),
    # 使用比 LeNet 输出大数倍了全连接层。其使用丢弃层来控制复杂度。
    nn.Dense(4096, activation="relu"), nn.Dropout(.5),
    nn.Dense(4096, activation="relu"), nn.Dropout(.5),
    # 输出层。我们这里使用 FashionMNIST，所以用 10，而不是论文中的 1000。
    nn.Dense(10)
)

```

我们构造一个高和宽均为 224 像素的单通道数据点来观察每一层的输出大小。

```

In [2]: X = nd.random.uniform(shape=(1,1,224,224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:', X.shape)

conv0 output shape: (1, 96, 54, 54)
pool0 output shape: (1, 96, 26, 26)
conv1 output shape: (1, 256, 26, 26)
pool1 output shape: (1, 256, 12, 12)
conv2 output shape: (1, 384, 12, 12)
conv3 output shape: (1, 384, 12, 12)
conv4 output shape: (1, 256, 12, 12)
pool2 output shape: (1, 256, 5, 5)
dense0 output shape: (1, 4096)
dropout0 output shape: (1, 4096)
dense1 output shape: (1, 4096)
dropout1 output shape: (1, 4096)
dense2 output shape: (1, 10)

```

5.6.3 读取数据

虽然论文中 Alexnet 使用 Imagenet 数据，但因为 Imagenet 数据训练时间较长，我们仍用前面的 FashionMNIST 来演示。读取数据的时候我们额外做了一步将图片高宽扩大到原版 Alexnet 使用的 224。

```
In [3]: train_data, test_data = gb.load_data_fashion_mnist(batch_size=128, resize=224)
```

5.6.4 训练

这时候我们可以开始训练。相对于上节的 LeNet，这里的主要改动是使用了更小的学习率。

```
In [4]: lr = 0.01
ctx = gb.try_gpu()
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)

training on gpu(0)
epoch 1, loss 1.3054, train acc 0.512, test acc 0.746, time 100.5 sec
epoch 2, loss 0.6483, train acc 0.758, test acc 0.816, time 98.1 sec
epoch 3, loss 0.5272, train acc 0.805, test acc 0.829, time 98.2 sec
epoch 4, loss 0.4647, train acc 0.829, test acc 0.856, time 98.2 sec
epoch 5, loss 0.4211, train acc 0.846, test acc 0.868, time 98.3 sec
```

5.6.5 小结

AlexNet 跟 LeNet 结构类似，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优秀实验结果的产生，学术界整整花了 20 年。

5.6.6 练习

- 多迭代几轮训练看看？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，同时保证精度不明显下降。
- 修改批量大小，观察性能和 GPU 内存的变化。

5.6.7 扫码直达讨论区



5.6.8 参考文献

[1] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

5.7 使用重复元素的网络（VGG）

AlexNet 在 LeNet 的基础上增加了三个卷积层。但作者对它们的卷积窗口、通道数和构造顺序均做了大量的调整。虽然 AlexNet 指明了深度卷积神经网络可以取得很高的结果，但并没有提供简单的规则来告诉后来的研究者如何设计新的网络。我们将在接下来数个小节里介绍几种不同的网络设计思路。

本节我们介绍 VGG [1]，它名字来源于论文作者所在实验室 Visual Geometry Group。VGG 提出了可以通过重复使用简单的基础块来构建深层模型。

5.7.1 VGG 块

VGG 模型的基础组成规律是：连续使用数个相同的填充为 1 的 3×3 卷积层后接上一个步幅为 2 的 2×2 最大池化层。卷积层保持输入高宽，而池化层则对其减半。我们使用 `vgg_block` 函数来实现这个基础块，它可以指定使用卷积层的数量和其输出通道数。

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

```
def vgg_block(num_convs, num_channels):
    blk = nn.Sequential()
    for _ in range(num_convs):
        blk.add(nn.Conv2D(
            num_channels, kernel_size=3, padding=1, activation='relu'))
    blk.add(nn.MaxPool2D(pool_size=2, strides=2))
    return blk
```

5.7.2 VGG 模型

VGG 网络同 AlexNet 和 LeNet 一样是由卷积层模块后接全连接层模块构成。卷积层模块串联数个 `vgg_block`, 其超参数由 `conv_arch` 定义, 用于指定每个块里卷积层个数和输出通道数。全连接模块则跟 AlexNet 一样。

现在我们构造一个 VGG 网络。它有 5 个卷积块, 前三块使用单卷积层, 而后两块使用双卷积层。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 个卷积层和 3 个全连接层, 所以经常被称为 VGG 11。

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

下面我们根据架构实现 VGG 11。

```
In [3]: def vgg(conv_arch):
    net = nn.Sequential()
    # 卷积层部分。
    for (num_convs, num_channels) in conv_arch:
        net.add(vgg_block(num_convs, num_channels))
    # 全连接层部分。
    net.add(nn.Dense(4096, activation="relu"), nn.Dropout(.5),
            nn.Dense(4096, activation="relu"), nn.Dropout(.5),
            nn.Dense(10))
    return net

net = vgg(conv_arch)
```

然后我们打印每个卷积块的输出变化。

```
In [4]: net.initialize()
X = nd.random.uniform(shape=(1,1,224,224))
for blk in net:
    X = blk(X)
    print(blk.name, 'output shape:', X.shape)
```

```
sequential1 output shape:      (1, 64, 112, 112)
sequential2 output shape:      (1, 128, 56, 56)
sequential3 output shape:      (1, 256, 28, 28)
sequential4 output shape:      (1, 512, 14, 14)
sequential5 output shape:      (1, 512, 7, 7)
dense0 output shape:          (1, 4096)
dropout0 output shape:        (1, 4096)
dense1 output shape:          (1, 4096)
dropout1 output shape:        (1, 4096)
dense2 output shape:          (1, 10)
```

可以看到每次我们将长宽减半，直到最后高宽变成 7 后进入全连接层。与此同时输出通道数每次都翻倍。因为每个卷积层的窗口大小一样，所以每层的模型参数大小和计算复杂度跟高 \times 宽 \times 输入通道数 \times 输出通道数成正比。VGG 这种高宽减半和通道翻倍的设计使得每个卷积层都有相同的模型参数大小和计算复杂度。

5.7.3 模型训练

因为 VGG 11 计算上比 AlexNet 更加复杂，出于测试的目的我们构造一个通道数更小，或者说更窄的网络来训练 FashionMNIST。

```
In [5]: ratio = 4
small_conv_arch = [(pair[0], pair[1]//ratio) for pair in conv_arch]
net = vgg(small_conv_arch)
```

除了使用了稍大些的学习率，模型训练过程跟上一节的 AlexNet 类似。

```
In [6]: lr = 0.05
ctx = gb.try_gpu()
net.initialize(ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
train_data, test_data = gb.load_data_fashion_mnist(batch_size=128, resize=224)
loss = gluon.loss.SoftmaxCrossEntropyLoss()
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=3)

training on gpu(0)
epoch 1, loss 0.9630, train acc 0.649, test acc 0.844, time 166.4 sec
epoch 2, loss 0.4118, train acc 0.850, test acc 0.868, time 163.6 sec
epoch 3, loss 0.3363, train acc 0.877, test acc 0.898, time 163.7 sec
```

5.7.4 小结

VGG 通过 5 个可以重复使用的卷积块来构造网络。根据每块里卷积层个数和输出通道数的不同可以定义出不同的 VGG 模型。

5.7.5 练习

- VGG 的计算比 AlexNet 慢很多，也需要很多的 GPU 内存。请分析下原因。
- 尝试将 FashionMNIST 的高宽由 224 改成 96，实验其带来的影响。
- 参考 [1] 里的表 1 来构造 VGG 其他常用模型，例如 VGG16 和 VGG19。

5.7.6 扫码直达讨论区



5.7.7 参考文献

[1] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

5.8 网络中的网络（NiN）

前面小节里我们看到 LeNet、AlexNet 和 VGG 均由两个部分组成：以卷积层构成的模块充分抽取空间特征，然后以全连接层构成的模块来输出最终分类结果。AlexNet 和 VGG 对 LeNet 的改进主要在于如何加深加宽这两个模块。这一节我们介绍网络中的网络（NiN）[1]。它提出了另外一个思路，即串联多个由卷积层和“全连接”层构成的小网络来构建一个深层网络。

5.8.1 NiN 块

我们知道卷积层的输入和输出都是四维数组，而全连接层则是二维数组。如果想在全连接层后接上卷积层，则需要将其输出转回到四维。回忆在“[多输入和输出通道](#)”这一小节里介绍的 1×1 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自然传递到后面的层去。下图对比了 NiN 同 AlexNet 和 VGG 等网络的主要区别。

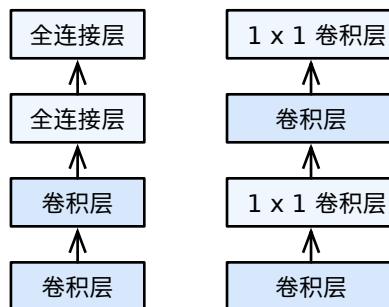


图 5.7: 对比 NiN (右) 和其他 (左)。

NiN 中的一个基础块由一个卷积层外加两个充当全连接层的 1×1 卷积层构成。第一个卷积层我们可以设置它的超参数，而第二和第三卷积层则使用固定超参数。

```
In [1]: import sys
        sys.path.insert(0, '...')
        import gluonbook as gb
        from mxnet import nd, gluon, init
        from mxnet.gluon import nn

def nin_block(num_channels, kernel_size, strides, padding):
    blk = nn.Sequential()
    blk.add(nn.Conv2D(num_channels, kernel_size,
                    strides, padding, activation='relu'),
           nn.Conv2D(num_channels, kernel_size=1, activation='relu'),
           nn.Conv2D(num_channels, kernel_size=1, activation='relu'))
    return blk
```

5.8.2 NiN 模型

NiN 紧跟 AlexNet 后提出，所以它的卷积层设定跟 Alexnet 类似。它使用窗口分别为 11×11 、 5×5 和 3×3 的卷积层，输出通道数也与之相同。卷积层后跟步幅为 2 的 3×3 最大池化层。

除了使用 NiN 块外，NiN 还有一个重要的跟 AlexNet 不同的地方：NiN 去掉了最后的三个全连接层，取而代之的是使用输出通道数等于标签类数的卷积层，然后使用一个窗口为输入高宽的平均池化层来将每个通道里的数值平均成一个标量直接用于分类。这个设计好处是可以显著的减小模型参数大小，从而能很好的避免过拟合，但它也可能会造成训练时收敛变慢。

```
In [2]: net = nn.Sequential()
    net.add(
        nin_block(96, kernel_size=11, strides=4, padding=0),
        nn.MaxPool2D(pool_size=3, strides=2),
        nin_block(256, kernel_size=5, strides=1, padding=2),
        nn.MaxPool2D(pool_size=3, strides=2),
        nin_block(384, kernel_size=3, strides=1, padding=1),
        nn.MaxPool2D(pool_size=3, strides=2), nn.Dropout(.5),
        # 标签类数是 10。
        nin_block(10, kernel_size=3, strides=1, padding=1),
        # 全局平均池化层将窗口形状自动设置成输出的高和宽。
        nn.GlobalAvgPool2D(),
        # 将四维的输出转成二维的输出，其形状为（批量大小， 10）。
        nn.Flatten())
```

我们构建一个数据来查看每一层的输出大小。

```
In [3]: X = nd.random.uniform(shape=(1,1,224,224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:', X.shape)

sequential1 output shape:      (1, 96, 54, 54)
pool0 output shape:      (1, 96, 26, 26)
sequential2 output shape:      (1, 256, 26, 26)
pool1 output shape:      (1, 256, 12, 12)
sequential3 output shape:      (1, 384, 12, 12)
pool2 output shape:      (1, 384, 5, 5)
dropout0 output shape:  (1, 384, 5, 5)
sequential4 output shape:      (1, 10, 5, 5)
pool3 output shape:      (1, 10, 1, 1)
flatten0 output shape:  (1, 10)
```

5.8.3 获取数据并训练

NiN 的训练与 Alexnet 和 VGG 类似，但一般使用更大的学习率。

```
In [4]: lr = .1
ctx = gb.try_gpu()
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
train_data, test_data = gb.load_data_fashion_mnist(batch_size=128, resize=224)
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=3)

training on gpu(0)
epoch 1, loss 2.2432, train acc 0.159, test acc 0.331, time 131.3 sec
epoch 2, loss 1.2639, train acc 0.517, test acc 0.676, time 130.0 sec
epoch 3, loss 0.7443, train acc 0.730, test acc 0.775, time 130.2 sec
```

5.8.4 小结

NiN 提供了两个重要的设计思路：

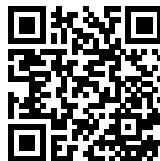
- 重复使用由卷积层和代替全连接层的 1×1 卷积层构成的基础块来构建深层网络；
- 去除了容易造成过拟合的全连接输出层，而是替换成输出通道数等于标签类数的卷积层和全局平均池化层。

虽然因为精度和收敛速度等问题 NiN 并没有像本章中介绍的其他网络那么被广泛使用，但 NiN 的设计思想影响了后面的一系列网络的设计。

5.8.5 练习

- 多用几个迭代周期来观察网络收敛速度。
- 为什么 NiN 块里要有两个 1×1 卷积层，去除一个看看？

5.8.6 扫码直达讨论区



5.8.7 参考文献

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

5.9 合并行连结的网络（GoogLeNet）

在 2014 年的 Imagenet 竞赛中，一个名叫 GoogLeNet [1] 的网络结构大放光彩。它虽然在名字上是向 LeNet 致敬，但在网络结构上已经很难看到 LeNet 的影子。GoogLeNet 吸收了 NiN 的网络嵌套网络的想法，并在此基础上做了很大的改进。在随后的几年里研究人员对它进行了数次改进，本小节将介绍这个模型系列的第一个版本。

5.9.1 Inception 块

GoogLeNet 中的基础卷积块叫做 Inception，得名于同名电影《盗梦空间》(Inception)，寓意梦中嵌套梦。比较上一节介绍的 NiN，这个基础块在结构上更加复杂。

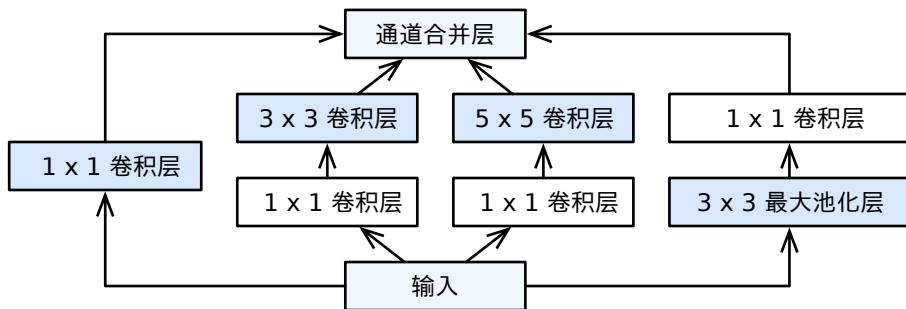


图 5.8: Inception 块。

由上图可以看出，Inception里有四个并行的线路。前三个线路里使用窗口大小分别是 1×1 、 3×3 和 5×5 的卷积层来抽取不同空间尺寸下的信息。其中中间两个线路会对输入先作用 1×1 卷积来减小输入通道数，以此降低模型复杂度。第四条线路则是使用 3×3 最大池化层，后接 1×1 卷积层来变换通道。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，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，我们以此来控制模型复杂度。

```
In [1]: import sys
        sys.path.insert(0, '...')
        import gluonbook as gb
        from mxnet import nd, init, gluon
        from mxnet.gluon import nn

        class Inception(nn.Block):
            # c1 ~ c4 为每条线路里的层的输出通道数。
            def __init__(self, c1, c2, c3, c4, **kwargs):
                super(Inception, self).__init__(**kwargs)
                # 线路 1, 单  $1 \times 1$  卷积层。
                self.p1_1 = nn.Conv2D(c1, kernel_size=1, activation='relu')
                # 线路 2,  $1 \times 1$  卷积层后接  $3 \times 3$  卷积层。
                self.p2_1 = nn.Conv2D(c2[0], kernel_size=1, activation='relu')
                self.p2_2 = nn.Conv2D(c2[1], kernel_size=3, padding=1,
                                     activation='relu')
                # 线路 3,  $1 \times 1$  卷积层后接  $5 \times 5$  卷积层。
                self.p3_1 = nn.Conv2D(c3[0], kernel_size=1, activation='relu')
                self.p3_2 = nn.Conv2D(c3[1], kernel_size=5, padding=2,
                                     activation='relu')
                # 线路 4,  $3 \times 3$  最大池化层后接  $1 \times 1$  卷积层。
                self.p4_1 = nn.MaxPool2D(pool_size=3, strides=1, padding=1)
                self.p4_2 = nn.Conv2D(c4, kernel_size=1, activation='relu')

            def forward(self, x):
                p1 = self.p1_1(x)
                p2 = self.p2_2(self.p2_1(x))
                p3 = self.p3_2(self.p3_1(x))
                p4 = self.p4_2(self.p4_1(x))
                # 在通道维上合并输出。
                return nd.concat(p1, p2, p3, p4, dim=1)
```

5.9.2 GoogLeNet 模型

GoogLeNet 跟 VGG 一样，在主体卷积部分中使用五个模块，每个模块之间使用步幅为 2 的 3×3 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 卷积层。

```
In [2]: b1 = nn.Sequential()
b1.add(
    nn.Conv2D(64, kernel_size=7, strides=2, padding=3, activation='relu'),
    nn.MaxPool2D(pool_size=3, strides=2, padding=1)
)
```

第二模块使用两个卷积层，首先是 64 通道的 1×1 卷积层，然后是将通道增大 3 倍的 3×3 卷积层。它对应 Inception 块中的第二线路。

```
In [3]: b2 = nn.Sequential()
b2.add(
    nn.Conv2D(64, kernel_size=1),
    nn.Conv2D(192, kernel_size=3, padding=1),
    nn.MaxPool2D(pool_size=3, strides=2, padding=1)
)
```

第三模块串联两个完整的 Inception 块。第一个 Inception 块的输出通道数为 256，其中四个线路的输出通道比例为 2: 4: 1: 1。且第二、三线路先分别将输入通道减小 2 倍和 12 倍后再进入第二层卷积层。第二个 Inception 块输出通道数增至 480，每个线路的通道比例为 4: 6: 3: 2。且第二、三线路先分别减少 2 倍和 8 倍通道数。

```
In [4]: b3 = nn.Sequential()
b3.add(
    Inception(64, (96, 128), (16, 32), 32),
    Inception(128, (128, 192), (32, 96), 64),
    nn.MaxPool2D(pool_size=3, strides=2, padding=1)
)
```

第四模块更加复杂，它串联了五个 Inception 块，其输出通道分别是 512、512、512、528 和 832。其线路的通道分配类似之前， 3×3 卷积层线路输出最多通道，其次是 1×1 卷积层线路，之后是 5×5 卷积层和 3×3 最大池化层线路。其中前两个线路都会先按比例减小通道数。这些比例在各个 Inception 块中都略有不同。

```
In [5]: b4 = nn.Sequential()
b4.add(
    Inception(192, (96, 208), (16, 48), 64),
    Inception(160, (112, 224), (24, 64), 64),
    Inception(128, (128, 256), (24, 64), 64),
    Inception(112, (144, 288), (32, 64), 64),
```

```
Inception(256, (160, 320), (32, 128), 128),
nn.MaxPool2D(pool_size=3, strides=2, padding=1)
)
```

第五模块有输出通道数为 832 和 1024 的两个 Inception 块，每个线路的通道分配使用同前的原则，但具体数字又是不同。因为这个模块后面紧跟输出层，所以它同 NiN 一样使用全局平均池化层来将每个通道高宽变成 1。最后我们将输出变成二维数组后加上一个输出大小为标签类数的全连接层作为输出。

```
In [6]: b5 = nn.Sequential()
b5.add(
    Inception(256, (160, 320), (32, 128), 128),
    Inception(384, (192, 384), (48, 128), 128),
    nn.GlobalAvgPool2D()
)

net = nn.Sequential()
net.add(b1, b2, b3, b4, b5, nn.Dense(10))
```

因为这个模型相计算复杂，而且修改通道数不如 VGG 那样简单。本节里我们将输入高宽从 224 降到 96 来加速计算。下面演示各个模块之间的输出形状变化。

```
In [7]: X = nd.random.uniform(shape=(1,1,96,96))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)

sequential0 output shape:      (1, 64, 24, 24)
sequential1 output shape:      (1, 192, 12, 12)
sequential2 output shape:      (1, 480, 6, 6)
sequential3 output shape:      (1, 832, 3, 3)
sequential4 output shape:      (1, 1024, 1, 1)
dense0 output shape:      (1, 10)
```

5.9.3 获取数据并训练

我们使用高宽为 96 的数据来训练。

```
In [8]: lr = 0.1
ctx = gb.try_gpu()
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
train_data, test_data = gb.load_data_fashion_mnist(batch_size=128, resize=96)
```

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

training on gpu(0)
epoch 1, loss 1.7905, train acc 0.331, test acc 0.662, time 84.1 sec
epoch 2, loss 0.6407, train acc 0.761, test acc 0.782, time 81.9 sec
epoch 3, loss 0.4537, train acc 0.831, test acc 0.850, time 82.0 sec
epoch 4, loss 0.3750, train acc 0.858, test acc 0.854, time 82.0 sec
epoch 5, loss 0.3388, train acc 0.872, test acc 0.881, time 82.0 sec
```

5.9.4 小结

Inception 块相当于一个有四条线路的子网络，它通过不同窗口大小的卷积层和最大池化层来并行抽取信息，并使用 1×1 卷积层减低通道数来减少模型复杂度。GoogLeNet 将多个精细设计的 Inception 块和其他层串联起来。其通道分配比例是在 ImageNet 数据集上通过大量的实验得来。GoogLeNet 和它的后继者一度是 ImageNet 上最高效的模型之一，即在给定同样的测试精度下计算复杂度更低。

5.9.5 练习

- 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）。
- 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, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., & Anguelov, D. & Rabinovich, A.(2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).
- [2] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.
- [3] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 2818-2826).
- [4] Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017, February). Inception-v4, inception-resnet and the impact of residual connections on learning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 4, p. 12).

5.10 批量归一化

这一节我们介绍批量归一化 (batch normalization) 层 [1]，它能让深层卷积网络的训练变得更加容易。在 “[实战 Kaggle 比赛：预测房价和 K 折交叉验证](#)” 小节里，我们对输入数据做了归一化处理，即将每个特征在所有样本上的值转归一化成均值 0 方差 1。这个处理可以保证训练数据的值都在同一量级上，从而使得训练时模型参数更加稳定。

通常来说，数据归一化预处理对于浅层模型就足够有效了。输出数值在只经过几个神经层后一般不会出现剧烈变化。但对于深层神经网络来说，情况会变得比较复杂：每一层里都对输入乘以权重后得到输出。当很多层这样的相乘累计在一起时，一个输入数据较小的改变都可能导致输出产生巨大变化，从而带来不稳定性。

批量归一化层就是针对这个情况提出的。它将一个批量里的输入数据进行归一化后再输出。如果我们将批量归一化层放置在网络的各个层之间，那么就可以不断的对中间输出进行调整，从而保证整个网络的中间输出在数值上都是稳定的。

5.10.1 批量归一化层

我们首先看将批量归一化层放置在全连接层后时的情况，它的机制类似于数据归一化处理。输入一个批量数据时，假设这个全连接层输出 n 个向量数据点 $X = \{x_1, \dots, x_n\}$ ，其中 $x_i \in \mathbb{R}^p$ 。我们

可以计算数据点在这个批量里面的均值和方差，其均为长度 p 的向量：

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

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

对于数据点 x_i ，我们可以对它的每一个特征维进行归一化：

$$\hat{x}_i \leftarrow \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}},$$

这里 ϵ 是一个很小的常数，保证分母大于 0。在上面归一化的基础之上，批量归一化层引入了两个可以学习的模型参数，拉升参数 γ 和偏移参数 β 。它们是长为 p 的向量，作用在 \hat{x}_i 上：

$$y_i \leftarrow \gamma \hat{x}_i + \beta.$$

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

如果批量归一化层是放置在卷积层后面，那么我们将通道维当做是特征维，空间维（高和宽）里的元素则当成是样本（参考“[多输入和输出通道](#)”一节里我们对 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，一般的解决方法是维护一个移动平滑的样本均值和方差，从而在预测时使用。

下面我们通过 NDArray 来实现这个计算。

```
In [1]: import sys
        sys.path.insert(0, '...')
        import gluonbook as gb
        from mxnet import nd, gluon, init, autograd
        from mxnet.gluon import nn

def batch_norm(X, gamma, beta, moving_mean, moving_var,
               eps, momentum):
    # 通过 autograd 来获取是不是在训练环境下。
    if not autograd.is_training():
        # 如果是在预测模式下，直接使用传入的移动平滑均值和方差。
        X_hat = (X - moving_mean) / nd.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
```

```

# 接在全连接层后情况，计算特征维上的均值和方差。
if len(X.shape) == 2:
    mean = X.mean(axis=0)
    var = ((X - mean)**2).mean(axis=0)
# 接在二维卷积层后的情况，计算通道维上 (axis=1) 的均值和方差。这里我们需要保持 X
# 的形状以便后面可以正常的做广播运算。
else:
    mean = X.mean(axis=(0,2,3), keepdims=True)
    var = ((X - mean)**2).mean(axis=(0,2,3), keepdims=True)
# 训练模式下用当前的均值和方差做归一化。
X_hat = (X - mean) / nd.sqrt(var + eps)
# 更新移动平滑均值和方差。
moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
moving_var = momentum * moving_var + (1.0 - momentum) * var
# 拉升和偏移
Y = gamma * X_hat + beta
return (Y, moving_mean, moving_var)

```

接下来我们自定义一个 BatchNorm 层。它保存参与求导和更新的模型参数 `beta` 和 `gamma`, 同时也维护移动平滑的均值和方差使得在预测时可以使用。

```

In [2]: class BatchNorm(nn.Block):
    def __init__(self, num_features, num_dims, **kwargs):
        super(BatchNorm, self).__init__(**kwargs)
        shape = (1,num_features) if num_dims == 2 else (1,num_features,1,1)
        # 参与求导和更新的模型参数，分别初始化成 0 和 1。
        self.beta = self.params.get('beta', shape=shape, init=init.Zero())
        self.gamma = self.params.get('gamma', shape=shape, init=init.One())
        # 不参与求导的模型参数。全在 CPU 上初始化成 0。
        self.moving_mean = nd.zeros(shape)
        self.moving_variance = nd.zeros(shape)
    def forward(self, X):
        # 如果 X 不在 CPU 上，将 moving_mean 和 moving_varience 复制到对应设备上。
        if self.moving_mean.context != X.context:
            self.moving_mean = self.moving_mean.copyto(X.context)
            self.moving_variance = self.moving_variance.copyto(X.context)
        # 保存更新过的 moving_mean 和 moving_var。
        Y, self.moving_mean, self.moving_variance = batch_norm(
            X, self.gamma.data(), self.beta.data(), self.moving_mean,
            self.moving_variance, eps=1e-5, momentum=0.9)
        return Y

```

5.10.2 使用批量归一化层的 LeNet

下面我们修改“卷积神经网络”这一节介绍的 LeNet 来使用批量归一化层。我们在所有的卷积层和全连接层与激活层之间加入批量归一化层，来使得每层的输出都被归一化。

```
In [3]: net = nn.Sequential()
net.add(
    nn.Conv2D(6, kernel_size=5),
    BatchNorm(6, num_dims=4),
    nn.Activation('sigmoid'),
    nn.MaxPool2D(pool_size=2, strides=2),
    nn.Conv2D(16, kernel_size=5),
    BatchNorm(16, num_dims=4),
    nn.Activation('sigmoid'),
    nn.MaxPool2D(pool_size=2, strides=2),
    nn.Dense(120),
    BatchNorm(120, num_dims=2),
    nn.Activation('sigmoid'),
    nn.Dense(84),
    BatchNorm(84, num_dims=2),
    nn.Activation('sigmoid'),
    nn.Dense(10)
)
```

使用同之前一样的超参数，可以发现前面五个迭代周期的收敛有明显加速。

```
In [4]: lr = 1.0
ctx = gb.try_gpu()
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
train_data, test_data = gb.load_data_fashion_mnist(batch_size=256)
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)

training on gpu(0)
epoch 1, loss 0.6601, train acc 0.766, test acc 0.729, time 3.6 sec
epoch 2, loss 0.3968, train acc 0.857, test acc 0.848, time 3.3 sec
epoch 3, loss 0.3457, train acc 0.876, test acc 0.870, time 3.3 sec
epoch 4, loss 0.3232, train acc 0.883, test acc 0.870, time 3.3 sec
epoch 5, loss 0.3039, train acc 0.889, test acc 0.867, time 3.3 sec
```

最后我们查看下第一个批量归一化层学习到的 `beta` 和 `gamma`。

```
In [5]: (net[1].beta.data().reshape((-1,)),
        net[1].gamma.data().reshape((-1,)))
```

```
Out[5]: (  
    [ 0.96624464  0.18823329  0.41603163  0.03639157 -0.51282758 -1.95774436]  
    <NDArray 6 @gpu(0)>,  
    [ 1.99354351  1.08590806  1.7710042   1.42921388  1.26272035  1.85810089]  
    <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，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 批量归一化的 Gluon 实现

相比于前小节定义的 BatchNorm 类，nn 模块定义的 BatchNorm 使用更加简单。它不需要指定输出数据的维度和特征维的大小，这些都将通过延后初始化来获取。我们实现同前小节一样的批量归一化的 LeNet。

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

        net = nn.Sequential()
        net.add(
            nn.Conv2D(6, kernel_size=5),
            nn.BatchNorm(),
            nn.Activation('sigmoid'),
            nn.MaxPool2D(pool_size=2, strides=2),
            nn.Conv2D(16, kernel_size=5),
            nn.BatchNorm(),
            nn.Activation('sigmoid'),
            nn.MaxPool2D(pool_size=2, strides=2),
            nn.Dense(120),
            nn.BatchNorm(),
            nn.Activation('sigmoid'),
            nn.Dense(84),
            nn.BatchNorm(),
            nn.Activation('sigmoid'),
            nn.Dense(10)
        )
```

使用同样的超参数进行训练。

```
In [2]: lr = 1.0
        ctx = gb.try_gpu()
        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        loss = gluon.loss.SoftmaxCrossEntropyLoss()
        train_data, test_data = gb.load_data_fashion_mnist(batch_size=256)
        gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)

training on gpu(0)
epoch 1, loss 0.6558, train acc 0.765, test acc 0.834, time 2.1 sec
epoch 2, loss 0.3985, train acc 0.857, test acc 0.784, time 1.9 sec
```

```
epoch 3, loss 0.3512, train acc 0.871, test acc 0.816, time 1.9 sec
epoch 4, loss 0.3256, train acc 0.882, test acc 0.832, time 2.0 sec
epoch 5, loss 0.3070, train acc 0.888, test acc 0.878, time 1.9 sec
```

5.11.1 小结

Gluon 提供的 BatchNorm 在使用上更加简单。

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，还有一个问题困扰着训练：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时，根据链式法则每层的梯度会被乘在一起，这样便导致梯度数值以指数衰减。最后，在靠近底部的层只得到很小的梯度，对应的权重更新量也变小，使得他们的收敛缓慢。

ResNet [1] 成功地通过增加跨层的数据线路来允许梯度快速地到达底部层，从而避免这一情况。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 (Residual Block)。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，从而让层 A 更容易获取足够大的梯度来进行模型更新。

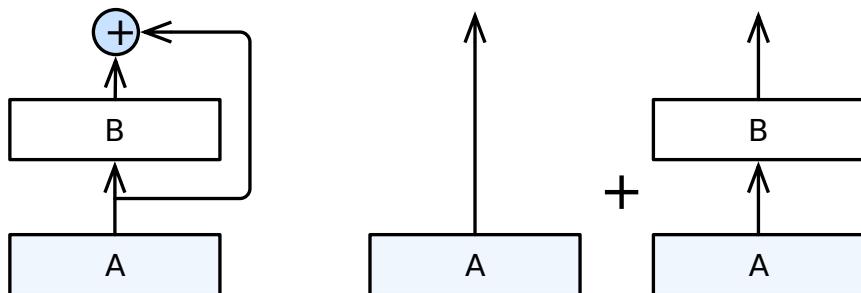


图 5.9: 残差块 (左) 和它的分解 (右)。

ResNet 沿用了 VGG 全 3×3 卷积层设计。残差块里首先是两个有同样输出通道的 3×3 卷积层，每个卷积层后跟一个批量归一化层和 ReLU 激活层。然后我们将输入跳过这两个卷积层后直接加在最后的 ReLU 激活层前。这样的设计要求两个卷积层的输出与输入形状一样，从而可以相加。如果想改变输出的通道数，我们需要引入一个额外的 1×1 卷积层来将输入变换成需要的形状后再相加。

残差块的实现见下。它可以设定输出通道数，是否使用额外的卷积层来修改输入通道数，以及卷积层的步幅大小。

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

class Residual(nn.Block):
    def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
        super(Residual, self).__init__(**kwargs)
        self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                            strides=strides)
```

```

    self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
    if use_1x1conv:
        self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                             strides=strides)
    else:
        self.conv3 = None
    self.bn1 = nn.BatchNorm()
    self.bn2 = nn.BatchNorm()

    def forward(self, X):
        Y = nd.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return nd.relu(Y + X)

```

查看输入输出形状一致的情况:

```

In [2]: blk = Residual(3)
blk.initialize()
X = nd.random.uniform(shape=(4, 3, 6, 6))
blk(X).shape

```

```
Out[2]: (4, 3, 6, 6)
```

改变输出形状的同时减半输出高宽:

```

In [3]: blk = Residual(6, use_1x1conv=True, strides=2)
blk.initialize()
blk(X).shape

```

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

5.12.2 ResNet 模型

ResNet 前面两层跟前面介绍的 GoogLeNet 一样，在输出通道为 64、步幅为 2 的 7×7 卷积层后接步幅为 2 的 3×3 的最大池化层。不同在于一点在于 ResNet 的每个卷积层后面增加的批量归一化层。

```

In [4]: net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
       nn.BatchNorm(), nn.Activation('relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))

```

GoogLeNet 在后面接了四个由 Inception 块组成的模块。ResNet 则是使用四个由残差块组成的模块，每个模块使用若干个同样输出通道的残差块。第一个模块的通道数同输入一致，同时因为之前已经使用了步幅为 2 的最大池化层，所以也不减小高宽。之后的每个模块在第一个残差块里将上一个模块的通道数翻倍，并减半高宽。

下面我们实现这个模块，注意我们对第一个模块做了特别处理。

```
In [5]: def resnet_block(num_channels, num_residuals, first_block=False):
    blk = nn.Sequential()
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.add(Residual(num_channels))
    return blk
```

接着我们为 ResNet 加入所有残差块。这里每个模块使用两个残差块。

```
In [6]: net.add(resnet_block(64, 2, first_block=True),
             resnet_block(128, 2),
             resnet_block(256, 2),
             resnet_block(512, 2))
```

最后与 GoogLeNet 一样我们加入全局平均池化层后接上全连接层输出。

```
In [7]: net.add(nn.GlobalAvgPool2D(), nn.Dense(10))
```

这里每个模块里有 4 个卷积层（不计算 1×1 卷积层），加上最开始的卷积层和最后的全连接层，一共有 18 层。这个模型也通常被称之为 ResNet 18。通过配置不同的通道数和模块里的残差块数我们可以得到不同的 ResNet 模型。

注意，每个残差块里我们都将输入直接加在输出上，少数几个通过简单的 1×1 卷积层后相加。这样一来，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 迅速被广泛使用。

最后我们考察输入在 ResNet 不同模块之间的变化。

```
In [8]: X = nd.random.uniform(shape=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:', X.shape)
```

```
conv5 output shape:      (1, 64, 112, 112)
batchnorm4 output shape:      (1, 64, 112, 112)
relu0 output shape:      (1, 64, 112, 112)
pool0 output shape:      (1, 64, 56, 56)
sequential1 output shape:      (1, 64, 56, 56)
sequential2 output shape:      (1, 128, 28, 28)
sequential3 output shape:      (1, 256, 14, 14)
sequential4 output shape:      (1, 512, 7, 7)
pool1 output shape:      (1, 512, 1, 1)
dense0 output shape:      (1, 10)
```

5.12.3 获取数据并训练

使用跟 GoogLeNet 一样的超参数，但减半了学习率。

```
In [9]: ctx = gb.try_gpu()
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.05})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
train_data, test_data = gb.load_data_fashion_mnist(batch_size=256, resize=96)
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)

training on gpu(0)
epoch 1, loss 0.4850, train acc 0.830, test acc 0.857, time 68.6 sec
epoch 2, loss 0.2563, train acc 0.906, test acc 0.861, time 67.5 sec
epoch 3, loss 0.1906, train acc 0.931, test acc 0.898, time 67.5 sec
epoch 4, loss 0.1420, train acc 0.950, test acc 0.913, time 67.7 sec
epoch 5, loss 0.1067, train acc 0.963, test acc 0.918, time 67.6 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, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).
- [2] He, K., Zhang, X., Ren, S., & Sun, J. (2016, October). Identity mappings in deep residual networks. In European Conference on Computer Vision (pp. 630-645). Springer, Cham.

5.13 稠密连接网络（DenseNet）

ResNet 中的跨层连接设计引申出了数个后续工作。这一节我们介绍其中的一个：稠密连接网络（DenseNet）[1]。它与 ResNet 的主要区别如下图演示。

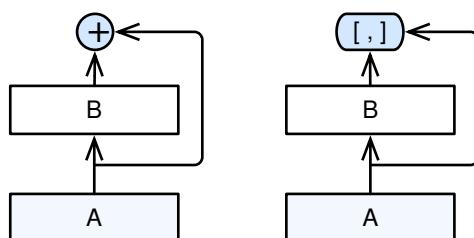


图 5.10: ResNet（左）对比 DenseNet（右）。

主要区别在于，DenseNet 里层 B 的输出不是像 ResNet 那样和层 A 的输出相加，而是在通道维

上合并，这样层 A 的输出可以不受影响的进入上面的神经层。这个设计里，层 A 直接跟上面的所有层连接在了一起，这也是它被称为“稠密连接”的原因。

DenseNet 的主要构建模块是稠密块和过渡块，前者定义了输入和输出是如何合并的，后者则用来控制通道数不要过大。

5.13.1 稠密块

DenseNet 使用了 ResNet 改良版的“批量归一化、激活和卷积”结构（参见上一节习题），我们首先在 `conv_block` 函数里实现这个结构。

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

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

稠密块由多个 `conv_block` 组成，每块使用相同的输出通道数。但在正向传播时，我们将每块的输出在通道维上同其输出合并进入下一个块。

```
In [2]: class DenseBlock(nn.Block):
    def __init__(self, num_convs, num_channels, **kwargs):
        super(DenseBlock, self).__init__(**kwargs)
        self.net = nn.Sequential()
        for _ in range(num_convs):
            self.net.add(conv_block(num_channels))

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            # 在通道维上将输入和输出合并。
            X = nd.concat(X, Y, dim=1)
        return Y
```

下面例子中我们定义一个有两个输出通道数为 10 的卷积块，使用通道数为 3 的输入时，我们会得到通道数为 $3 + 2 \times 10 = 23$ 的输出。卷积块的通道数控制了输出通道数相对于输入通道数的增

长，因此也被称为增长率（growth rate）。

```
In [3]: blk = DenseBlock(2, 10)
blk.initialize()
X = nd.random.uniform(shape=(4,3,8,8))
Y = blk(X)
Y.shape

Out[3]: (4, 10, 8, 8)
```

5.13.2 过渡块

由于每个稠密块都会带来通道数的增加。使用过多则会导致过于复杂的模型。过渡块（transition block）则用来控制模型复杂度。它通过 1×1 卷积层来减小通道数，同时使用步幅为 2 的平均池化层来将高宽减半来进一步降低复杂度。

```
In [4]: def transition_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(), nn.Activation('relu'),
           nn.Conv2D(num_channels, kernel_size=1),
           nn.AvgPool2D(pool_size=2, strides=2))
    return blk
```

我们对前面的稠密块的输出使用通道数为 10 的过渡块。

```
In [5]: blk = transition_block(10)
blk.initialize()
blk(Y).shape

Out[5]: (4, 10, 4, 4)
```

5.13.3 DenseNet 模型

DenseNet 首先使用跟 ResNet 一样的单卷积层和最大池化层：

```
In [6]: net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
       nn.BatchNorm(), nn.Activation('relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

类似于 ResNet 接下来使用的四个基于残差块，DenseNet 使用的是四个稠密块。同 ResNet 一样我们可以设置每个稠密块使用多少个卷积层，这里我们设成 4，跟上一节的 ResNet 18 保持一致。稠密块里的卷积层通道数（既增长率）设成 32，所以每个稠密块将增加 128 通道。

ResNet 里通过步幅为 2 的残差块来在每个模块之间减小高宽，这里我们则是使用过渡块来减半高宽，并且减半输入通道数。

```
In [7]: # 当前的数据通道数。  
num_channels = 64  
growth_rate = 32  
num_convs_in_dense_blocks = [4, 4, 4]  
  
for i, num_convs in enumerate(num_convs_in_dense_blocks):  
    net.add(DenseBlock(num_convs, growth_rate))  
    # 上一个稠密的输出通道数。  
    num_channels += num_convs * growth_rate  
    # 在稠密块之间加入通道数减半的过渡块。  
    if i != len(num_convs_in_dense_blocks)-1:  
        net.add(transition_block(num_channels//2))
```

最后同 ResNet 一样我们接上全局池化层和全连接层来输出。

```
In [8]: net.add(nn.BatchNorm(), nn.Activation('relu'),  
            nn.GlobalAvgPool2D(), nn.Dense(10))
```

5.13.4 获取数据并训练

因为这里我们使用了比较深的网络，所以我们进一步把输入减少到 32×32 来训练。

```
In [9]: ctx = gb.try_gpu()  
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())  
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})  
loss = gluon.loss.SoftmaxCrossEntropyLoss()  
train_data, test_data = gb.load_data_fashion_mnist(batch_size=256, resize=96)  
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=5)  
  
training on gpu(0)  
epoch 1, loss 0.5550, train acc 0.813, test acc 0.865, time 58.0 sec  
epoch 2, loss 0.3002, train acc 0.891, test acc 0.826, time 56.4 sec  
epoch 3, loss 0.2494, train acc 0.909, test acc 0.911, time 56.4 sec  
epoch 4, loss 0.2214, train acc 0.920, test acc 0.910, time 56.4 sec  
epoch 5, loss 0.1990, train acc 0.927, test acc 0.920, time 56.4 sec
```

5.13.5 小结

- 不同于 ResNet 中将输入加在输出上完成跨层连接，DenseNet 在通道维上合并输入和输出来使得底部神经层能跟其上面所有层连接起来。

5.13.6 练习

- DesNet 论文中提交的一个优点是其模型参数比 ResNet 更小，这是为什么？
- DesNet 被人诟病的一个问题是内存消耗过多。真的会这样吗？可以把输入换成 224×224 ，来看看实际（GPU）内存消耗。
- 实现 [1] 中的表 1 提出的各个 DenseNet 版本。

5.13.7 扫码直达讨论区



5.13.8 参考文献

[1] Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (Vol. 1, No. 2).

循环神经网络

与之前介绍的多层感知机和卷积神经网络不同，循环神经网络（recurrent neural networks）引入了状态变量。在一个序列中，循环神经网络当前时刻的状态不仅保存了过去时刻的信息，还与当前时刻的输入共同决定当前时刻的输出。

循环神经网络常用于处理序列数据，例如一段文字或声音、购物或观影的顺序、甚至是图片中的一行或一列像素。因此，循环神经网络在实际中有着极为广泛的应用，例如语言模型、文本分类、机器翻译、语音识别、图像分析、手写识别和推荐系统。

由于本章中的应用基于语言模型，我们将先介绍语言模型的基本概念，并以此问题激发循环神经网络的设计灵感。接着，我们将描述循环神经网络中梯度计算方法，来探究循环神经网络训练可能存在的问题。对于其中的部分问题，我们可以使用本章稍后介绍的含门控的循环神经网络来解决。最后，我们将拓展循环神经网络的架构并介绍更精简的 Gluon 实现。

6.1 语言模型

语言模型 (language model) 是自然语言处理的重要技术。自然语言处理中最常见的数据是文本数据。实际上，我们可以把一段自然语言文本看作是一段离散的时间序列。假设一段长度为 T 的文本中的词依次为 w_1, w_2, \dots, w_T ，那么在离散的时间序列中， w_t ($1 \leq t \leq T$) 可看作是在时间步 (time step) t 的输出或标签。给定一个长度为 T 的词的序列： w_1, w_2, \dots, w_T ，语言模型将计算该序列的概率：

$$\mathbb{P}(w_1, w_2, \dots, w_T).$$

语言模型可用于提升语音识别和机器翻译的性能。例如在语音识别中，给定一段“厨房里食油用完了”的语音，有可能会输出“厨房里食油用完了”和“厨房里石油用完了”这两个读音完全一样的文本序列。如果语言模型判断出前者的概率大于后者的概率，我们可以根据相同读音的语音输出“厨房里食油用完了”的文本序列。在机器翻译中，如果对英文“you go first”逐词翻译成中文的话，可能得到“你走先”、“你先走”等排列方式的文本序列。如果语言模型判断出“你先走”的概率大于其他排列方式的文本序列的概率，我们可以把“you go first”翻译成“你先走”。

6.1.1 语言模型的计算

既然语言模型很有用，那该如何计算它呢？假设序列 w_1, w_2, \dots, w_T 依次生成，我们有

$$\mathbb{P}(w_1, w_2, \dots, w_T) = \prod_{t=1}^T \mathbb{P}(w_t | w_1, \dots, w_{t-1}).$$

例如一段含有四个词的文本序列的概率

$$\mathbb{P}(w_1, w_2, w_3, w_4) = \mathbb{P}(w_1) \mathbb{P}(w_2 | w_1) \mathbb{P}(w_3 | w_1, w_2) \mathbb{P}(w_4 | w_1, w_2, w_3).$$

为了计算语言模型，我们需要计算词的概率和给定前几个词的条件概率，即语言模型参数。设训练数据集为一个大型文本语料库。词的概率可以通过该词在训练数据集中的相对词频计算。例如， $\mathbb{P}(w_1)$ 可以计算为 w_1 在训练数据集中的词频与训练数据集的总词数的比值。因此，根据条件概率定义，一个词在给定前几个词的条件概率也可以通过训练数据集中的相对词频计算。例如， $\mathbb{P}(w_2 | w_1) = \mathbb{P}(w_1, w_2) / \mathbb{P}(w_1)$ 可以计算为 w_1, w_2 两词相邻的频率与 w_1 词频的比值。而 $\mathbb{P}(w_3 | w_1, w_2) = \mathbb{P}(w_1, w_2, w_3) / \mathbb{P}(w_1, w_2)$ 可以计算为 w_1, w_2, w_3 三词相邻的频率与 w_1, w_2 两词相邻的频率的比值。以此类推。

6.1.2 N 元语法

实际上，我们可以通过马尔可夫假设（虽然并不一定成立）来简化语言模型的计算。基于 $n - 1$ 阶马尔可夫假设，我们将语言模型改写为

$$\mathbb{P}(w_1, w_2, \dots, w_T) \approx \prod_{t=1}^T \mathbb{P}(w_t | w_{t-(n-1)}, \dots, w_{t-1}).$$

以上也叫 n 元语法 (n -grams)。它是基于 $n - 1$ 阶马尔可夫链的概率语言模型。当 n 分别为 1、2 和 3 时，我们将其分别称作一元语法 (unigram)、二元语法 (bigram) 和三元语法 (trigram)。例如， w_1, w_2, w_3, w_4 在一元、二元和三元语法中的概率分别为

$$\begin{aligned}\mathbb{P}(w_1, w_2, w_3, w_4) &= \mathbb{P}(w_1)\mathbb{P}(w_2)\mathbb{P}(w_3)\mathbb{P}(w_4), \\ \mathbb{P}(w_1, w_2, w_3, w_4) &= \mathbb{P}(w_1)\mathbb{P}(w_2 | w_1)\mathbb{P}(w_3 | w_2)\mathbb{P}(w_4 | w_3), \\ \mathbb{P}(w_1, w_2, w_3, w_4) &= \mathbb{P}(w_1)\mathbb{P}(w_2 | w_1)\mathbb{P}(w_3 | w_1, w_2)\mathbb{P}(w_4 | w_2, w_3).\end{aligned}$$

当 n 较小时， n 元语法往往并不准确。例如，在一元语法中，由三个词组成的句子“你走先”和“你先走”的概率是一样的。然而，当 n 较大时， n 元语法需要计算并存储大量的词频和多词相邻频率。

那么，有没有方法在语言模型中更好地平衡以上这两点呢？我们将在本章探究这样的方法。

6.1.3 小结

- 语言模型是自然语言处理的重要技术。
- N 元语法是基于 $n - 1$ 阶马尔可夫链的概率语言模型。但它有一定的局限性。

6.1.4 练习

- 假设训练数据集中有十万个词，四元语法需要存储多少词频和多词相邻频率？
- 你还能想到哪些语言模型的应用？

6.1.5 扫码直达讨论区



6.2 隐藏状态

上一节介绍的 n 元语法中，基于文本序列最近 $n - 1$ 个词生成时间步 t 的词 w_t 的条件概率为

$$\mathbb{P}(w_t \mid w_{t-(n-1)}, \dots, w_{t-1}).$$

需要注意的是，以上概率并没有考虑到比 $t - (n - 1)$ 更早时间步的词对 w_t 可能的影响。然而，考虑这些影响需要增大 n 的值，那么 n 元语法的模型参数的数量将随之呈指数级增长（可参考上一节的练习）。为了解决 n 元语法的局限性，我们可以在神经网络中引入隐藏状态。我们既要捕捉时间序列的历史信息，又希望模型参数的数量不随历史增长而增长。

6.2.1 不含隐藏状态的神经网络

让我们先回顾一下不含隐藏状态的神经网络，例如只有一个隐藏层的多层感知机。

给定样本数为 n 、输入个数（特征数或特征向量维度）为 d 的小批量数据样本 $\mathbf{X} \in \mathbb{R}^{n \times d}$ 。设隐藏层的激活函数为 ϕ ，那么隐藏层的输出 $\mathbf{H} \in \mathbb{R}^{n \times h}$ 计算为

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

其中权重参数 $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ ，偏差参数 $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ， h 为隐藏单元个数。上式相加的两项形状不同，因此将按照广播机制相加（参见“[数据操作](#)”一节）。把隐藏变量 \mathbf{H} 作为输出层的输入，且设输出个数为 q （例如分类问题中的类别数），输出层的输出

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hy} + \mathbf{b}_y,$$

其中输出变量 $\mathbf{O} \in \mathbb{R}^{n \times q}$ ，输出层权重参数 $\mathbf{W}_{hy} \in \mathbb{R}^{h \times q}$ ，输出层偏差参数 $\mathbf{b}_y \in \mathbb{R}^{1 \times q}$ 。如果是分类问题，我们可以使用 $\text{softmax}(\mathbf{O})$ 来计算输出类别的概率分布。

6.2.2 含隐藏状态的循环神经网络

现在我们考虑时间序列数据，并基于上面描述的多层感知机引入隐藏状态，从而构造循环神经网络。

假设 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ 是序列中时间步 t 的小批量输入（样本数为 n ，输入个数为 d ），该时间步隐藏层变量是 $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ （隐藏单元个数为 h ，是超参数），输出层变量是 $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ （输出个数为 q ）。

为了使隐藏层变量能够捕捉时间序列的历史信息，我们引入一个新的权重参数 $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ ，并且使当前时间步隐藏层变量同时取决于当前时间步输入 \mathbf{X}_t 和上一时间步隐藏层变量 $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ ：

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

这里的隐藏层变量又叫隐藏状态。通常，我们会将隐藏状态全部元素初始化为 0。隐藏状态捕捉了截至当前时间步的序列历史信息，就像是神经网络当前时间步的状态或记忆一样。神经网络下一时间步的隐藏状态既取决于下一时间步的输入，又取决于当前时间步的隐藏状态。如此循环往复。我们将此类神经网络称作循环神经网络。在时间步 t ，循环神经网络的输出层输出和多层感知机中的计算类似：

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hy} + \mathbf{b}_y.$$

可见，循环神经网络在时间步 t 的输出基于相同时间步的隐藏状态。循环神经网络的参数包括隐藏层的权重 $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ 和偏差 $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ，以及输出层的权重 $\mathbf{W}_{hy} \in \mathbb{R}^{h \times q}$ 和偏差 $\mathbf{b}_y \in \mathbb{R}^{1 \times q}$ 。值得一提的是，即便在不同时间步，循环神经网络始终使用这些模型参数。因此，循环神经网络模型参数的数量不随历史增长而增长。

6.2.3 小结

- 循环神经网络通过引入隐藏状态来捕捉时间序列的历史信息。
- 循环神经网络模型参数的数量不随历史增长而增长。

6.2.4 练习

- 如果我们使用循环神经网络来预测一段文本序列的下一个词，输出个数应该是多少？
- 为什么循环神经网络可以表达某时间步的词基于文本序列中所有过去的词的条件概率？

6.2.5 扫码直达讨论区



6.3 循环神经网络

前两节介绍了语言模型和循环神经网络的设计。在本节中，我们将从零开始实现一个基于循环神经网络的语言模型，并应用它创作歌词。循环神经网络还有更广泛的应用。我们将在“自然语言处理”篇章中使用循环神经网络对不定长的文本序列分类，或把它翻译成不定长的另一语言的文本序列。

6.3.1 基于循环神经网络的语言模型

首先让我们简单回顾一下上一节描述的循环神经网络表达式。给定时间步 t 的小批量输入 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (样本数为 n , 输入个数为 d)，设该时间步隐藏状态为 $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ (隐藏单元个数为 h)，输出层变量为 $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (输出个数为 q)，隐藏层的激活函数为 ϕ 。循环神经网络的矢量计算表达式为

$$\begin{aligned}\mathbf{H}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h), \\ \mathbf{O}_t &= \mathbf{H}_t \mathbf{W}_{hy} + \mathbf{b}_y,\end{aligned}$$

其中隐藏层的权重 $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ 和偏差 $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ，以及输出层的权重 $\mathbf{W}_{hy} \in \mathbb{R}^{h \times q}$ 和偏差 $\mathbf{b}_y \in \mathbb{R}^{1 \times q}$ 为循环神经网络的模型参数。有些文献所指的循环神经网络只含隐藏状态 \mathbf{H}_t 的计算表达式。

在语言模型中，输入个数 x 为任意词的特征向量长度（本节稍后将讨论）；输出个数 y 为语料库中所有可能的词的个数。对循环神经网络的输出做 softmax 运算，我们可以得到时间步 t 输出所有可能的词的概率分布 $\hat{\mathbf{Y}}_t \in \mathbb{R}^{n \times q}$ ：

$$\hat{\mathbf{Y}}_t = \text{softmax}(\mathbf{O}_t).$$

由于隐藏状态 H_t 捕捉了时间步 1 到时间步 t 的小批量输入 $\mathbf{X}_1, \dots, \mathbf{X}_t$ 的信息， $\hat{\mathbf{Y}}_t$ 可以批量表达语言模型中给定文本序列中过去词生成下一个词的条件概率。有了这些条件概率，语言模型可以计算任意文本序列的概率。

6.3.2 字符级循环神经网络

本节实验中的循环神经网络将每个字符视作词。我们有时将该模型称为字符级循环神经网络 (character-level recurrent neural network)。设小批量中样本数 $n = 1$ ，文本序列为“你”、“好”、“世”、“界”。为了表达给定文本序列中过去词生成下一个词的条件概率，我们需要把输入序列和标签序列分别设为“你”、“好”、“世”和“好”、“世”、“界”，如图 6.1 所示。

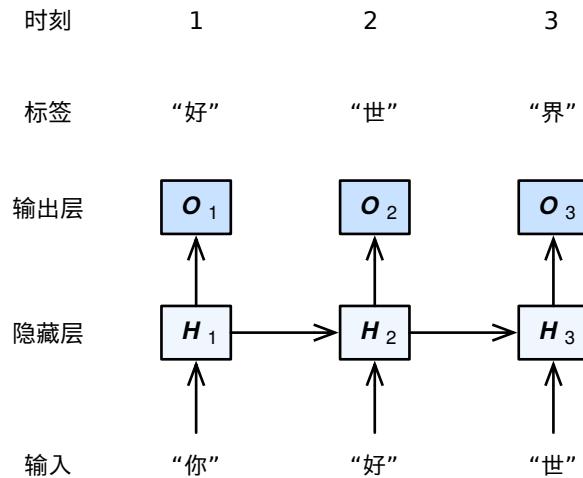


图 6.1：基于循环神经网络的语言模型。输入序列和标签序列分别为“你”、“好”、“世”和“好”、“世”、“界”。

当训练模型时，我们可以使用分类模型中常用的交叉熵损失函数计算各个时间步的损失。在图 6.1 中，由于隐藏层中隐藏状态的循环迭代，时间步 3 的输出 O_3 取决于文本序列“你”、“好”、“世”。由于训练数据中该序列的下一个词为“界”，时间步 3 的损失将取决于该时间步基于序列“你好世”生成下一个词的概率分布与该时间步标签“界”。

6.3.3 创作歌词

在创作歌词的实验中，我们将应用基于字符级循环神经网络的语言模型。与图 6.1 中的例子类似，我们将根据训练数据集的文本序列得到输入序列和标签序列。当模型训练好后，我们将以一种简单的方式创作歌词：根据给定的前缀，输出预测概率最大的下一个词；然后将该词附在前缀后继续输出预测概率最大的下一个词；如此循环。

创作歌词也可用到其他技术。例如，将输入拆分成以词语而不是字符为单位的序列、添加嵌入层（本章后面会介绍）或使用“自然语言处理”篇章中介绍的束搜索。

6.3.4 歌词数据集

我们使用周杰伦歌词数据集来训练模型作词。该数据集里包含了著名创作型歌手周杰伦从第一张专辑《Jay》到第十张专辑《跨时代》中歌曲的歌词。

首先导入实现所需的包或模块。

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

下面我们读取这个数据集，看看前 50 个字符是什么样的。

```
In [2]: with zipfile.ZipFile('../data/jaychou_lyrics.txt.zip', 'r') as zin:
            zin.extractall('../data/')

            with open('../data/jaychou_lyrics.txt') as f:
                corpus_chars = f.read()

corpus_chars[0:50]
```

Out[2]: '想要有直升机\n想要和你飞到宇宙去\n想要和你融化在一起\n融化在宇宙里\n我每天每天每天在想想想
→ 想著你\n这'

看一下数据集中文本序列的长度。

```
In [3]: len(corpus_chars)
```

Out[3]: 63282

接着我们稍微处理下数据集。为了打印方便，我们把换行符替换成空格。我们使用序列的前两万个字符训练模型。

```
In [4]: corpus_chars = corpus_chars.replace('\n', ' ').replace('\r', ' ')
corpus_chars = corpus_chars[0:20000]
```

6.3.5 建立字符索引

我们将数据集里面所有不同的字符取出来做成词典。打印 `vocab_size`，即词典中不同字符的个数。

```
In [5]: idx_to_char = list(set(corpus_chars))
char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
vocab_size = len(char_to_idx)
vocab_size

Out[5]: 1447
```

然后，把每个字符转成从 0 开始的索引从而方便之后的使用。

```
In [6]: corpus_indices = [char_to_idx[char] for char in corpus_chars]
sample = corpus_indices[:40]
print('chars: \n', ''.join([idx_to_char[idx] for idx in sample]))
print('\nindices: \n', sample)

chars:
想要有直升机 想要和你飞到宇宙去 想要和你融化在一起 融化在宇宙里 我每天每天每

indices:
[173, 1091, 533, 1335, 229, 1195, 699, 173, 1091, 1124, 166, 869, 79, 238, 1411, 330,
 → 699, 173, 1091, 1124, 166, 1007, 457, 551, 360, 261, 699, 1007, 457, 551, 238,
 → 1411, 762, 699, 729, 10, 1157, 10, 1157, 10]
```

6.3.6 时序数据的采样

同之前的实验一样，我们需要每次随机读取小批量样本和标签。不同的是，时序数据的一个样本通常包含连续的字符。假设时间步数为 5，样本序列为 5 个字符：“想”、“要”、“有”、“直”、“升”。那么该样本的标签序列为这些字符分别在训练集中的下一个字符：“要”、“有”、“直”、“升”、“机”。

我们有两种方式对时序数据采样，分别是随机采样和相邻采样。

随机采样

下面代码每次从数据里随机采样一个小批量。其中批量大小 `batch_size` 指每个小批量的样本数，`num_steps` 为每个样本所包含的时间步数。在随机采样中，每个样本是原始序列上任意截取的一段序列。相邻的两个随机小批量在原始序列上的位置不一定相毗邻。因此，我们无法用一个小批量最终时间步的隐藏状态来初始化下一个小批量的隐藏状态。在训练模型时，每次随机采样前都需要重新初始化隐藏状态。

```
In [7]: def data_iter_random(corpus_indices, batch_size, num_steps, ctx=None):
    # 减一是因为输出的索引是相应输入的索引加一。
    num_examples = (len(corpus_indices) - 1) // num_steps
    epoch_size = num_examples // batch_size
    example_indices = list(range(num_examples))
    random.shuffle(example_indices)

    def _data(pos):
        return corpus_indices[pos: pos+num_steps]

    for i in range(epoch_size):
        # 每次读取 batch_size 个随机样本。
        i = i * batch_size
        batch_indices = example_indices[i: i+batch_size]
        X = nd.array(
            [_data(j * num_steps) for j in batch_indices], ctx=ctx)
        Y = nd.array(
            [_data(j * num_steps + 1) for j in batch_indices], ctx=ctx)
        yield X, Y
```

让我们输入一个从 0 到 29 的人工序列，设批量大小和时间步数分别为 2 和 3，打印随机采样每次读取的小批量样本的输入 `X` 和标签 `Y`。可见，相邻的两个随机小批量在原始序列上的位置不一定相毗邻。

```
In [8]: my_seq = list(range(30))
for X, Y in data_iter_random(my_seq, batch_size=2, num_steps=3):
    print('X: ', X, '\nY: ', Y, '\n')

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

```

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

X:
[[ 15.  16.  17.]
 [ 12.  13.  14.]]
<NDArray 2x3 @cpu(0)>
Y:
[[ 16.  17.  18.]
 [ 13.  14.  15.]]
<NDArray 2x3 @cpu(0)>

X:
[[ 0.  1.  2.]
 [ 21.  22.  23.]]
<NDArray 2x3 @cpu(0)>
Y:
[[ 1.  2.  3.]
 [ 22.  23.  24.]]
<NDArray 2x3 @cpu(0)>

```

相邻采样

除了对原始序列做随机采样之外，我们还可以使相邻的两个随机小批量在原始序列上的位置相毗邻。这时候，我们就可以用一个小批量最终时间步的隐藏状态来初始化下一个小微量的隐藏状态，从而使下一个小微量的输出也取决于当前小微量输入，并如此循环下去。这对实现循环神经网络造成了两方面影响。一方面，在训练模型时，我们只需在每一个迭代周期开始时初始化隐藏状态。另一方面，当多个相邻小微量通过传递隐藏状态串联起来时，模型参数的梯度计算将依赖所有串联起来的小微量序列。同一迭代周期中，随着迭代次数的增加，梯度的计算开销会越来越大。为了使模型参数的梯度计算只依赖一次迭代读取的小微量序列，我们可以在每次读取小微量前将隐藏状态从计算图分离出来。

```
In [9]: def data_iter_consecutive(corpus_indices, batch_size, num_steps, ctx=None):
    corpus_indices = nd.array(corpus_indices, ctx=ctx)
```

```

data_len = len(corpus_indices)
batch_len = data_len // batch_size
indices = corpus_indices[0: batch_size*batch_len].reshape((
    batch_size, batch_len))
# 减一是因为输出的索引是相应输入的索引加一。
epoch_size = (batch_len - 1) // num_steps
for i in range(epoch_size):
    i = i * num_steps
    X = indices[:, i: i+num_steps]
    Y = indices[:, i+1: i+num_steps+1]
    yield X, Y

```

让我们输入一个从 0 到 29 的人工序列，设批量大小和时间步数分别为 2 和 3，打印相邻采样每次读取的小批量样本的输入 X 和标签 Y。相邻的两个随机小批量在原始序列上的位置相毗邻。

```

In [10]: my_seq = list(range(30))
        for X, Y in data_iter_consecutive(my_seq, batch_size=2, num_steps=3):
            print('X: ', X, '\nY: ', Y, '\n')

X:
[[ 0.   1.   2.]
 [ 15.  16.  17.]]
<NDArray 2x3 @cpu(0)>
Y:
[[ 1.   2.   3.]
 [ 16.  17.  18.]]
<NDArray 2x3 @cpu(0)>

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

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

```

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

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

6.3.7 One-hot 向量

为了用向量表示词，一个简单的办法是使用 one-hot 向量。假设词典中不同字符的数量为 N ，每个字符可以和从 0 到 $N - 1$ 的连续整数一一对应。这些与字符对应的整数也叫字符的索引。如果一个字符的索引是整数 i ，那么我们创建一个全 0 的长为 `vocab_size` 的向量，并将其位置为 i 的元素设成 1。该向量就是对原字符的 one-hot 向量。因此，本节实验中循环神经网络的输入个数 x 是任意词的特征向量长度 `vocab_size`。

下面分别展示了索引为 0 和 2 的 one-hot 向量。

```
In [11]: nd.one_hot(nd.array([0, 2]), vocab_size)

Out[11]:
[[ 1. 0. 0. ..., 0. 0. 0.]
 [ 0. 0. 1. ..., 0. 0. 0.]]
<NDArray 2x1447 @cpu(0)>
```

我们每次采样的小批量的形状是 $(batch_size, num_steps)$ 。下面这个函数将其转换成 num_steps 个可以输入进网络的形状为 $(batch_size, num_steps)$ 的矩阵。对于一个时间步数为 num_steps 的序列，每个批量输入 $X \in \mathbb{R}^{n \times x}$ ，其中 $n = batch_size$, $x = vocab_size$ (one-hot 向量长度)。

```
In [12]: def to_onehot(X, size):
    return [nd.one_hot(x, size) for x in X.T]

get_inputs = to_onehot
inputs = get_inputs(X, vocab_size)
len(inputs), inputs[0].shape

Out[12]: (3, (2, 1447))
```

6.3.8 初始化模型参数

接下来，我们初始化模型参数。隐藏单元个数 `num_hiddens` 是一个超参数。

```
In [13]: ctx = gb.try_gpu()
print('will use', ctx)

num_inputs = vocab_size
num_hiddens = 256
num_outputs = vocab_size

def get_params():
    # 隐藏层参数。
    W_xh = nd.random.normal(scale=0.01, shape=(num_inputs, num_hiddens),
                           ctx=ctx)
    W_hh = nd.random.normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                           ctx=ctx)
    b_h = nd.zeros(num_hiddens, ctx=ctx)
    # 输出层参数。
    W_hy = nd.random.normal(scale=0.01, shape=(num_hiddens, num_outputs),
                           ctx=ctx)
    b_y = nd.zeros(num_outputs, ctx=ctx)

    params = [W_xh, W_hh, b_h, W_hy, b_y]
    for param in params:
        param.attach_grad()
    return params

will use gpu(0)
```

6.3.9 定义模型

我们根据循环神经网络的表达式实现该模型。这里的激活函数使用了 `tanh` 函数。“[多层感知机](#)”一节中介绍过，当元素在实数域上均匀分布时，`tanh` 函数值的均值为 0。

假设小批量中样本数为 `batch_size`，时间步数为 `num_steps`。以下 `rnn` 函数的 `inputs` 和 `outputs` 皆为 `num_steps` 个形状为 $(batch_size, vocab_size)$ 的矩阵，隐藏状态 `H` 是一个形状为 $(batch_size, num_hiddens)$ 的矩阵。

```
In [14]: def rnn(inputs, state, *params):
    H = state
    W_xh, W_hh, b_h, W_hy, b_y = params
    outputs = []
```

```

for X in inputs:
    H = nd.tanh(nd.dot(X, W_xh) + nd.dot(H, W_hh) + b_h)
    Y = nd.dot(H, W_ty) + b_y
    outputs.append(Y)
return outputs, H

```

做个简单的测试：

```

In [15]: state = nd.zeros(shape=(X.shape[0], num_hiddens), ctx=ctx)
params = get_params()
outputs, state_new = rnn(get_inputs(X.as_in_context(ctx), vocab_size), state,
                         *params)
len(outputs), outputs[0].shape, state_new.shape

Out[15]: (3, (2, 1447), (2, 256))

```

6.3.10 定义预测函数

以下函数预测基于前缀 `prefix` 接下来的 `num_chars` 个字符。我们将用它根据训练得到的循环神经网络 `rnn` 来创作歌词。

```

In [16]: def predict_rnn(rnn, prefix, num_chars, params, num_hiddens, vocab_size, ctx,
                      idx_to_char, char_to_idx, get_inputs, is_lstm=False):
    prefix = prefix.lower()
    state_h = nd.zeros(shape=(1, num_hiddens), ctx=ctx)
    if is_lstm:
        # 当 RNN 使用 LSTM 时才会用到 (后面章节会介绍), 本节可以忽略。
        state_c = nd.zeros(shape=(1, num_hiddens), ctx=ctx)
        output = [char_to_idx[prefix[0]]]
        for i in range(num_chars + len(prefix)):
            X = nd.array([output[-1]], ctx=ctx)
            # 在序列中循环迭代隐藏状态。
            if is_lstm:
                # 当 RNN 使用 LSTM 时才会用到 (后面章节会介绍), 本节可以忽略。
                Y, state_h, state_c = rnn(get_inputs(X, vocab_size), state_h,
                                           state_c, *params)
            else:
                Y, state_h = rnn(get_inputs(X, vocab_size), state_h, *params)
            if i < len(prefix) - 1:
                next_input = char_to_idx[prefix[i + 1]]
            else:
                next_input = int(Y[0].argmax(axis=1).asscalar())
            output.append(next_input)
        return ''.join([idx_to_char[i] for i in output])
    else:
        state_h = nd.zeros(shape=(1, num_hiddens), ctx=ctx)
        for i in range(num_chars + len(prefix)):
            X = nd.array([char_to_idx[prefix[i]]], ctx=ctx)
            Y, state_h = rnn(get_inputs(X, vocab_size), state_h, *params)
            output.append(int(Y[0].argmax(axis=1).asscalar()))
        return ''.join([idx_to_char[i] for i in output])

```

6.3.11 裁剪梯度

循环神经网络中较容易出现梯度衰减或爆炸。我们会在[下一节](#)中解释原因。为了应对梯度爆炸，我们可以裁剪梯度 (clipping gradient)。假设我们把所有模型参数梯度的元素拼接成一个向量 g ，并设裁剪的阈值是 θ 。裁剪后梯度

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

的 L_2 范数不超过 θ 。

```
In [17]: def grad_clipping(params, state_h, Y, theta, ctx):
    if theta is not None:
        norm = nd.array([0.0], ctx)
        for param in params:
            norm += (param.grad ** 2).sum()
        norm = norm.sqrt().asscalar()
        if norm > theta:
            for param in params:
                param.grad[:] *= theta / norm
```

6.3.12 定义模型训练函数

跟之前章节的训练模型函数相比，这里有以下几个不同。

1. 使用困惑度 (perplexity) 评价模型。
2. 在迭代模型参数前裁剪梯度。
3. 对时序数据采用不同采样方法将导致隐藏状态初始化的不同。

```
In [18]: def train_and_predict_rnn(rnn, is_random_iter, num_epochs, num_steps,
                                 num_hiddens, lr, clipping_theta, batch_size,
                                 vocab_size, pred_period, pred_len, prefixes,
                                 get_params, get_inputs, ctx, corpus_indices,
                                 idx_to_char, char_to_idx, is_lstm=False):
    if is_random_iter:
        data_iter = data_iter_random
    else:
        data_iter = data_iter_consecutive
    params = get_params()
    loss = gloss.SoftmaxCrossEntropyLoss()

    for epoch in range(1, num_epochs + 1):
```

```

# 如使用相邻采样，隐藏变量只需在该 epoch 开始时初始化。
if not is_random_iter:
    state_h = nd.zeros(shape=(batch_size, num_hiddens), ctx=ctx)
    if is_lstm:
        state_c = nd.zeros(shape=(batch_size, num_hiddens), ctx=ctx)
train_l_sum = nd.array([0], ctx=ctx)
train_l_cnt = 0
for X, Y in data_iter(corpus_indices, batch_size, num_steps, ctx):
    # 如使用随机采样，读取每个随机小批量前都需要初始化隐藏变量。
    if is_random_iter:
        state_h = nd.zeros(shape=(batch_size, num_hiddens), ctx=ctx)
        if is_lstm:
            state_c = nd.zeros(shape=(batch_size, num_hiddens),
                               ctx=ctx)
    # 如使用相邻采样，需要使用 detach 函数从计算图分离隐藏状态变量。
    else:
        state_h = state_h.detach()
        if is_lstm:
            state_c = state_c.detach()
    with autograd.record():
        # outputs 形状: (batch_size, vocab_size)。
        if is_lstm:
            outputs, state_h, state_c = rnn(
                get_inputs(X, vocab_size), state_h, state_c, *params)
        else:
            outputs, state_h = rnn(
                get_inputs(X, vocab_size), state_h, *params)
    # 设 t_ib_j 为时间步 i 批量中的元素 j:
    # y 形状: (batch_size * num_steps,)
    # y = [t_0b_0, t_0b_1, ..., t_1b_0, t_1b_1, ..., ]
    y = Y.T.reshape((-1,))
    # 拼接 outputs, 形状: (batch_size * num_steps, vocab_size)。
    outputs = nd.concat(*outputs, dim=0)
    l = loss(outputs, y)
    l.backward()
    # 裁剪梯度。
    grad_clipping(params, state_h, Y, clipping_theta, ctx)
    gb.sgd(params, lr, 1)
    train_l_sum = train_l_sum + l.sum()
    train_l_cnt += l.size
    if epoch % pred_period == 0:
        print("\nepoch %d, perplexity %f"
              % (epoch, (train_l_sum / train_l_cnt).exp().asscalar()))

```

```
for prefix in prefixes:  
    print(' - ', predict_rnn(  
        rnn, prefix, pred_len, params, num_hiddens, vocab_size,  
        ctx, idx_to_char, char_to_idx, get_inputs, is_lstm))
```

困惑度

回忆一下“Softmax 回归”一节中交叉熵损失函数的定义。困惑度是对交叉熵损失函数做指数运算后得到的值。特别地，

- 最佳情况下，模型总是把标签类别的概率预测为 1。此时困惑度为 1。
- 最坏情况下，模型总是把标签类别的概率预测为 0。此时困惑度为正无穷。
- 基线情况下，模型总是预测所有类别的概率都相同。此时困惑度为类别数。

显然，任何一个有效模型的困惑度必须小于类别数。在本例中，困惑度必须小于词典中不同的字符数 `vocab_size`。

6.3.13 训练模型并创作歌词

以上介绍的 `to_onehot`、`data_iter_random`、`data_iter_consecutive`、`grad_clipping`、`predict_rnn` 和 `train_and_predict_rnn` 函数均定义在 `gluonbook` 包中供后面章节调用。有了这些函数以后，我们就可以训练模型了。

首先，设置模型超参数。我们将根据前缀“分开”和“不分开”分别创作长度为 100 个字符的一段歌词。我们每过 40 个迭代周期便根据当前训练的模型创作一段歌词。

```
In [19]: num_epochs = 200  
       num_steps = 35  
       batch_size = 32  
       lr = 0.2  
       clipping_theta = 5  
       prefixes = ['分开', '不分开']  
       pred_period = 40  
       pred_len = 100
```

下面采用随机采样训练模型并创作歌词。

```
In [20]: train_and_predict_rnn(rnn, True, num_epochs, num_steps, num_hiddens, lr,  
                           clipping_theta, batch_size, vocab_size, pred_period,
```

```
pred_len, prefixes, get_params, get_inputs, ctx,
corpus_indices, idx_to_char, char_to_idx)
```

epoch 40, perplexity 78.055084

- 分开 我想要这想 我不要我想 我不要我想 我不要我想你 你知我们不要 你在我 你是我 别你的公
→ 我们了你 我怎么 别兽人 我想 我想 我想你我想你我不要我想 我不要我想你我不要我想
→ 我不要我想你我不要我想
- 不分开 我有你这不 我不要我想 我不要我想 我不要我想你 你知我们不要 你在我 你是我 别你的公
→ 我们了你 我怎么 别兽人 我想 我想 我想你我想你我不要我想 我不要我想你我不要我想
→ 我不要我想你我不要我想

epoch 80, perplexity 12.738266

- 分开 我不想再想牵我 你后的我为想再 什么我想见你 我面好这里 你后 不想再我 全有你 的手 篮剩下
→ 让我 恨散球 沉 这不起 印手 你想说不想要这样 我想不到 你只会感 说不是以 我想要好 我想要到
- 不分开 没有这样子我 不想对我 我不着好注我 不知不觉 我已了这节奏 后知后觉 我已经好生我 不知不觉
→ 我已了这节奏 后知后觉 我已经好生我 不知不觉 我已了这节奏 后知后觉 我已经好生我 不知不觉 我已了

epoch 120, perplexity 4.704494

- 分开 为什么这样子 你看着我说你已经决定 我拉不住你 我的手影放 我的手后想在我的怀里 你慢慢怎去
→ 我摇不再你 泪水在战壕里决了堤 在在一步默剧就你的门堂 夕阳的壳我的坠入 你慢的风
→ 在吹我会半球猜了看痛不
- 不分开时间 所以你的权 有一种味道叫做家 他羽泡的茶 像说泼墨利都不拿 他牵着一匹瘦马在走天涯
→ 爷爷泡的茶 有一种味道叫做家 他羽泡的茶 像幅泼墨的山水画 唐朝千年的风沙 现在还在刮 三爷泡的茶
→ 有人事觉乡

epoch 160, perplexity 2.929296

- 分开 干什么这样的真唱 还堡被风里的屋诺 我爱女不了分过 就什么我连分开都迁就着你 想真的世蜜
→ 默后的人丽 是欢什么我有多 从的黑争 温面的灰尘 银制茶壶前 我也要的假 的话 随而过滤了的公
→ 还在我面的画界
- 不分开吗 我不能再想 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 我害怕后放
→ 没人帮觉的够水 请慈在的落 爷幅名墨利都不拿 他牵着一匹瘦 在隔回种 半人用双截棍 哼哼哈兮
→ 快使用双截棍

epoch 200, perplexity 2.346058

- 分开 干什么 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
→ 看远方的星是否听的见 手牵手手前都是你 趁说不多 你却会感到更加沮 不要说 一诉就步 你一于 分我的战
→ 小人之声 过
- 不分开吗 我不能再想 我不 我不 我不要再想你 不知不觉 你已经离开我 不知不觉 我想了这节奏 后知后觉
→ 后过后一个秋 后知后觉 我该好好生活 我该好好生活 不知不觉 你已经离开我 不知不觉 我跟了这节奏 后
接下来采用相邻采样训练模型并创作歌词。

```
In [21]: train_and_predict_rnn(rnn, False, num_epochs, num_steps, num_hiddens, lr,
clipping_theta, batch_size, vocab_size, pred_period,
```

```
pred_len, prefixes, get_params, get_inputs, ctx,
corpus_indices, idx_to_char, char_to_idx)
```

epoch 40, perplexity 65.104759

- 分开 我不能再想 我不要再想 我不要我想 我不能再想 我不要我想 我不能再想 我不能再想 我不要我想
- 我不能再想 我不要我想 我不能再想 我不能再想 我不要我想 我不能再想 我不能再想 我不能再想 我不能再想
- 不分开 爱过我的爱 有一种味道 我不能我想 我不能 想情我的爱 有一种味道 有不么 不想我 别你 我不要
- 不要我的爱 有一种味道 有不么 不想我 别你 我不要 不要我的爱 有一种味道 有不么 不想我 别你 我不要

epoch 80, perplexity 9.274164

- 分开 我不能 不想开的太快 想回到 一直箱 我已就这样牵着你的手不放开 爱能不能够永 我会不好开了天
- 化身为龙 把大地心脏汹涌 不安跳动 全世界 的日情调整了一种 等待英雄 我就是那条龙 我右拳打开了天 化
- 不分开 我的眼受 无有这 带不懂美 对静了我 恨不是 别羽的公式 一切味断 全小我 别怪了 我想就这样牵
- 你爱我 说怪我的爱 就说了名 我们多 是一事人重 就有一步被底的家 说录你遇快 是一事实忆 你在一

epoch 120, perplexity 3.594883

- 分开 我留着陪你 强人的泪丽 泡的完美有义 如彻底 干数么恨发下 一直在停落 谁在它停留 仙人在怕羞
- 蜷里横著走 这里横著走 这里什么走 三底什么走 三蝎横著走 三蝎横著走 这里横著走 这里什么走 三底什么
- 不分开 我单会 一直两口每步四等望 为什么这种速度你的回不放开 爱可不能以永简单纯没有一点 想
- 那着我不要 从小黑迷濡违 所穿开其的我面的天 景象方色季 有一条实昏 基乡中的茶情好像顶不住那时间
- 为什么这样

epoch 160, perplexity 2.578281

- 分开 我留着陪想就没人帮 为我们的我会的屋家摊 无地心阵 干壁里的橱窗 一把吉他 远远欣赏 木炭 一箩筐
- 木炭 一直放 木炭 剩一半 火炉烫 小铁匠存钱人期望 在流汗 一直两步筑步四步望著天 看星星 一颗两
- 不分开这样 我太多陪你 最后的距离 是你的侧脸倒在我的怀里 你慢慢睡去 你摇着我的证据 让晶莹的泪滴
- 闪烁成回忆 伤人在美落 这种事怕羞 蜷蝎横角走 这里什么走 三对横种牛 三窝横角走 三底拽角走
- 三底拽种走

epoch 200, perplexity 2.230363

- 分开 我留着陪你 我爱 我不 我不能再想你 爱情来的太快就像龙卷风 离不开暴风圈来不及逃 我不能再想
- 我不能再想 我不 我不 我不要再想你 爱情来的太快就像龙卷风 离不开暴风圈来不及逃 我不能再想 我不能再
- 不分开睡 单上没比 将想方 的表出调整了时空 回到洪荒 去支配去操纵 我右拳打开了天 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 我就是那条龙 我右拳打开了天 化身为龙
- 那大地心脏

6.3.14 小结

- 我们可以应用基于字符级循环神经网络的语言模型来创作歌词。
- 时序数据采样方式包括随机采样和相邻采样。使用这两种方式的循环神经网络训练略有不同。

- 当训练循环神经网络时，为了应对梯度爆炸，我们可以裁剪梯度。
- 困惑度是对交叉熵损失函数做指数运算后得到的值。

6.3.15 练习

- 调调超参数，观察并分析对运行时间、困惑度以及创作歌词的结果造成的影响。
- 不裁剪梯度，运行本节代码。结果会怎样？
- 将 `pred_period` 改为 1，观察未充分训练的模型（困惑度高）是如何创作歌词的。你获得了什么启发？
- 将相邻采样改为不从计算图分离隐藏状态，运行时间有没有变化？
- 将本节中使用的激活函数替换成 ReLU，重复本节的实验。

6.3.16 扫码直达讨论区



6.4 通过时间反向传播

如果你做了上一节的练习，你会发现，如果不裁剪梯度，模型将无法正常训练。为了深刻理解这一现象，本节将介绍循环神经网络中梯度的计算和存储方法，即通过时间反向传播 (back-propagation through time)。

我们在“[正向传播和反向传播](#)”一节中介绍了神经网络中梯度计算与存储的一般思路，并强调正向传播和反向传播相互依赖。正向传播在循环神经网络比较直观。通过时间反向传播其实是反向传播在循环神经网络的具体应用。我们需要将循环神经网络按时间步展开，从而得到模型变量和参数之间的依赖关系，并依据链式法则应用反向传播计算并存储梯度。

6.4.1 定义模型

为了简洁，我们考虑一个无偏差项的循环神经网络，且激活函数的输入输出相同。

设时间步 t 的输入为 $\mathbf{x}_t \in \mathbb{R}^d$ ，标签为 y_t ，隐藏状态 $\mathbf{h}_t \in \mathbb{R}^h$ 的计算表达式为

$$\mathbf{h}_t = \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1},$$

其中 $\mathbf{W}_{hx} \in \mathbb{R}^{h \times d}$ 和 $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ 是隐藏层权重参数。设输出层权重参数 $\mathbf{W}_{yh} \in \mathbb{R}^{q \times h}$ ，时间步 t 的输出层变量 $\mathbf{o}_t \in \mathbb{R}^q$ 计算为

$$\mathbf{o}_t = \mathbf{W}_{yh}\mathbf{h}_t.$$

设时间步 t 的损失为 $\ell(\mathbf{o}_t, y_t)$ 。时间步数为 T 的损失函数 L 定义为

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

我们将 L 叫做有关给定时间步数的数据样本的目标函数，并在以下的讨论中简称目标函数。

6.4.2 模型计算图

为了可视化模型变量和参数之间在计算中的依赖关系，我们可以绘制模型计算图，如图 6.2 所示。例如，时间步 3 的隐藏状态 \mathbf{h}_3 的计算依赖模型参数 $\mathbf{W}_{hx}, \mathbf{W}_{hh}$ 、上一时间步隐藏状态 \mathbf{h}_2 以及当前时间步输入 \mathbf{x}_3 。

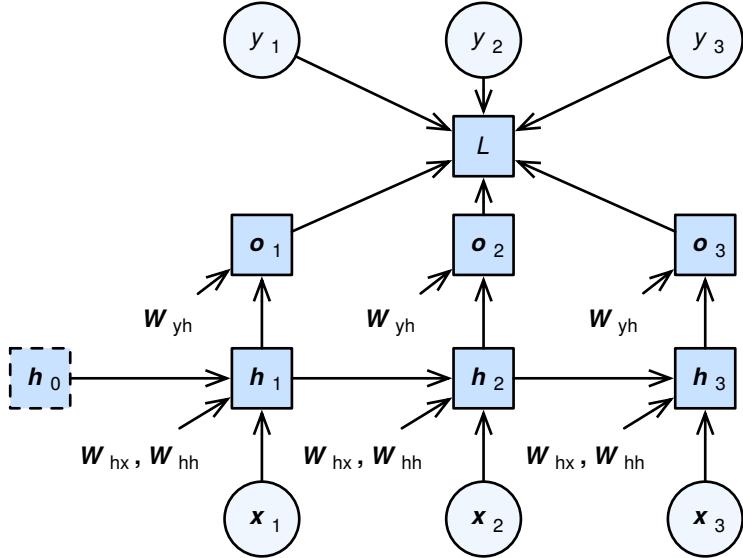


图 6.2: 时间步数为 3 的循环神经网络模型计算中的依赖关系。方框中字母代表变量，圆圈中字母代表数据样本特征和标签，无边框的字母代表模型参数。

6.4.3 通过时间反向传播

刚刚提到，图 6.2 中模型的参数是 \mathbf{W}_{hx} 、 \mathbf{W}_{hh} 和 \mathbf{W}_{yh} 。与“正向传播和反向传播”一节中类似，训练模型通常需要模型参数的梯度 $\partial L / \partial \mathbf{W}_{hx}$ 、 $\partial L / \partial \mathbf{W}_{hh}$ 和 $\partial L / \partial \mathbf{W}_{yh}$ 。根据图 6.2 中的依赖关系，我们可以按照其中箭头所指的反方向依次计算并存储梯度。

为了表述方便，我们依然使用“正向传播和反向传播”一节中表达链式法则的操作符 `prod`。

首先，目标函数有关各时间步输出层变量的梯度 $\partial L / \partial \mathbf{o}_t \in \mathbb{R}^q$ 可以很容易地计算：

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

下面，我们可以计算目标函数有关模型参数 \mathbf{W}_{yh} 的梯度 $\partial L / \partial \mathbf{W}_{yh} \in \mathbb{R}^{q \times h}$ 。根据图 6.2， L 通过 $\mathbf{o}_1, \dots, \mathbf{o}_T$ 依赖 \mathbf{W}_{yh} 。依据链式法则，

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

其次，我们注意到隐藏状态之间也有依赖关系。在图 6.2 中， L 只通过 \mathbf{o}_T 依赖最终时间步 T 的隐藏状态 \mathbf{h}_T 。因此，我们先计算目标函数有关最终时间步隐藏状态的梯度 $\partial L / \partial \mathbf{h}_T \in \mathbb{R}^h$ 。依据

链式法则，我们得到

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

接下来，对于时间步 $t < T$ ，在图 6.2 中， L 通过 \mathbf{h}_{t+1} 和 \mathbf{o}_t 依赖 \mathbf{h}_t 。依据链式法则，目标函数有关时间步 $t < T$ 的隐藏状态的梯度 $\partial L / \partial \mathbf{h}_t \in \mathbb{R}^h$ 需要按照时间步从晚到早依次计算：

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

将上面的递归公式展开，对任意时间步 $1 \leq t \leq T$ ，我们可以得到目标函数有关隐藏状态梯度的通项公式

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

由上式中的指数项可见，当时间步数 T 较大或者时间步 t 较小，目标函数有关隐藏状态的梯度较容易出现衰减和爆炸。这也会影响其他计算中包含 $\partial L / \partial \mathbf{h}_t$ 的梯度，例如隐藏层中模型参数的梯度 $\partial L / \partial \mathbf{W}_{hx} \in \mathbb{R}^{h \times d}$ 和 $\partial L / \partial \mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ 。在图 6.2 中， L 通过 $\mathbf{h}_1, \dots, \mathbf{h}_T$ 依赖这些模型参数。依据链式法则，我们有

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{W}_{hx}} &= \sum_{t=1}^T \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hx}}\right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{x}_t^\top, \\ \frac{\partial L}{\partial \mathbf{W}_{hh}} &= \sum_{t=1}^T \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}\right) = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{h}_{t-1}^\top. \end{aligned}$$

“正向传播和反向传播”一节里解释过，每次迭代中，上述各个依次计算出的梯度会被依次存储或更新。这是为了避免重复计算。例如，由于隐藏状态梯度 $\partial L / \partial \mathbf{h}_t$ 被计算存储，之后的模型参数梯度 $\partial L / \partial \mathbf{W}_{hx}$ 和 $\partial L / \partial \mathbf{W}_{hh}$ 的计算可以直接读取 $\partial L / \partial \mathbf{h}_t$ 的值，而无需重复计算。此外，反向传播对于各层中变量和参数的梯度计算可能会依赖通过正向传播计算出的各层变量的当前值。举例来说，参数梯度 $\partial L / \partial \mathbf{W}_{hh}$ 的计算需要依赖隐藏状态在时间步 $t = 0, \dots, T-1$ 的当前值 \mathbf{h}_t (\mathbf{h}_0 是初始化得到的)。这些值是通过从输入层到输出层的正向传播计算并存储得到的。

6.4.4 小结

- 通过时间反向传播是反向传播在循环神经网络的具体应用。
- 当时间步数较大时，循环神经网络的梯度较容易衰减或爆炸。

6.4.5 练习

- 除了梯度裁剪，你还能想到别的什么方法应对循环神经网络中的梯度爆炸？

6.4.6 扫码直达讨论区



6.5 门控循环单元（GRU）

上一节介绍了循环神经网络中的梯度计算方法。我们发现，循环神经网络的梯度可能会衰减或爆炸。虽然裁剪梯度可以应对梯度爆炸，但无法解决梯度衰减的问题。给定一个时间序列，例如文本序列，循环神经网络在实际中较难捕捉时间步距离较大的词之间的依赖关系。

门控循环神经网络（gated recurrent neural network）的提出，是为了更好地捕捉时间序列中时间步距离较大的依赖关系。其中，门控循环单元（gated recurrent unit，简称 GRU）是一种常用的门控循环神经网络 [1, 2]。我们将在下一节介绍另一种门控循环神经网络：长短期记忆。

6.5.1 门控循环单元

下面将介绍门控循环单元的设计。它引入了门的概念，从而修改了循环神经网络中隐藏状态的计算方式。输出层的设计不变。

重置门和更新门

假设隐藏单元个数为 h ，给定时间步 t 的小批量输入 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ （样本数为 n ，输入个数为 d ）和上一时间步隐藏状态 $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ 。重置门（reset gate） $\mathbf{R}_t \in \mathbb{R}^{n \times h}$ 和更新门（update gate）

$Z_t \in \mathbb{R}^{n \times h}$ 的计算如下：

$$\begin{aligned}\mathbf{R}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r), \\ \mathbf{Z}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z).\end{aligned}$$

其中 $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$ 和 $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$ 是权重参数， $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$ 是偏移参数。激活函数 σ 是 sigmoid 函数。“多层感知机”一节中介绍过，sigmoid 函数可以将元素的值变换到 0 和 1 之间。因此，重置门 \mathbf{R}_t 和更新门 \mathbf{Z}_t 中每个元素的值域都是 $[0, 1]$ 。

我们可以通过元素值域在 $[0, 1]$ 的更新门和重置门来控制隐藏状态中信息的流动：这通常可以应用按元素乘法符 \odot 。

候选隐藏状态

接下来，时间步 t 的候选隐藏状态 $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ 的计算使用了值域在 $[-1, 1]$ 的 \tanh 函数做激活函数。它在之前描述的循环神经网络隐藏状态表达式的基础上，引入了重置门和按元素乘法：

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

其中 $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ 和 $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ 是权重参数， $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ 是偏移参数。需要注意的是，候选隐藏状态使用了重置门，从而控制包含时间序列历史信息的上一个时间步的隐藏状态如何流入当前时间步的候选隐藏状态。如果重置门近似 0，上一个隐藏状态将被丢弃。因此，重置门可以丢弃与预测未来无关的历史信息。

隐藏状态

最后，隐藏状态 $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ 的计算使用更新门 \mathbf{Z}_t 来对上一时间步的隐藏状态 \mathbf{H}_{t-1} 和当前时间步的候选隐藏状态 $\tilde{\mathbf{H}}_t$ 做组合：

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

值得注意的是，更新门可以控制隐藏状态应该如何被包含当前时间步信息的候选隐藏状态所更新。假设更新门在时间步 t' 到 t ($t' < t$) 之间一直近似 1。那么，在时间步 t' 到 t 之间的输入信息几乎没有流入时间步 t 的隐藏状态 \mathbf{H}_t 。实际上，这可以看作是较早时刻的隐藏状态 $\mathbf{H}_{t'-1}$ 一直通过时间保存并传递至当前时间步 t 。这个设计可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时间序列中时间步距离较大的依赖关系。

我们对门控循环单元的设计稍作总结：

- 重置门有助于捕捉时间序列里短期的依赖关系。
- 更新门有助于捕捉时间序列里长期的依赖关系。

6.5.2 实验

为了实现并展示门控循环单元，我们依然使用周杰伦歌词数据集来训练模型作词。这里除门控循环单元以外的实现已在“[循环神经网络](#)”一节中介绍。

处理数据

我们先读取并简单处理数据集。

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

        with zipfile.ZipFile('../data/jaychou_lyrics.txt.zip', 'r') as zin:
            zin.extractall('../data/')
        with open('../data/jaychou_lyrics.txt') as f:
            corpus_chars = f.read()

        corpus_chars = corpus_chars.replace('\n', ' ').replace('\r', ' ')
        corpus_chars = corpus_chars[0:20000]
        idx_to_char = list(set(corpus_chars))
        char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
        corpus_indices = [char_to_idx[char] for char in corpus_chars]
        vocab_size = len(char_to_idx)
```

初始化模型参数

以下部分对模型参数进行初始化。超参数 `num_hiddens` 定义了隐藏单元的个数。

```
In [2]: ctx = gb.try_gpu()
        num_inputs = vocab_size
        num_hiddens = 256
        num_outputs = vocab_size

        def get_params():
```

```

# 更新门参数。
W_xz = nd.random_normal(scale=0.01, shape=(num_inputs, num_hiddens),
                        ctx=ctx)
W_hz = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                        ctx=ctx)
b_z = nd.zeros(num_hiddens, ctx=ctx)
# 重置门参数。
W_xr = nd.random_normal(scale=0.01, shape=(num_inputs, num_hiddens),
                        ctx=ctx)
W_hr = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                        ctx=ctx)
b_r = nd.zeros(num_hiddens, ctx=ctx)
# 候选隐藏状态参数。
W_xh = nd.random_normal(scale=0.01, shape=(num_inputs, num_hiddens),
                        ctx=ctx)
W_hh = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                        ctx=ctx)
b_h = nd.zeros(num_hiddens, ctx=ctx)
# 输出层参数。
W_hy = nd.random_normal(scale=0.01, shape=(num_hiddens, num_outputs),
                        ctx=ctx)
b_y = nd.zeros(num_outputs, ctx=ctx)

params = [W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hy, b_y]
for param in params:
    param.attach_grad()
return params

```

6.5.3 定义模型

下面根据门控循环单元的计算表达式定义模型。

```
In [3]: def gru_rnn(inputs, H, *params):
    W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hy, b_y = params
    outputs = []
    for X in inputs:
        Z = nd.sigmoid(nd.dot(X, W_xz) + nd.dot(H, W_hz) + b_z)
        R = nd.sigmoid(nd.dot(X, W_xr) + nd.dot(H, W_hr) + b_r)
        H_tilda = nd.tanh(nd.dot(X, W_xh) + R * nd.dot(H, W_hh) + b_h)
        H = Z * H + (1 - Z) * H_tilda
        Y = nd.dot(H, W_hy) + b_y
        outputs.append(Y)
    return (outputs, H)
```

训练模型并创作歌词

设置好超参数后，我们将训练模型并根据前缀“分开”和“不分开”分别创作长度为 100 个字符的一段歌词。我们每过 30 个迭代周期便根据当前训练的模型创作一段歌词。训练模型时采用了相邻采样。

```
In [4]: get_inputs = gb.to_onehot
num_epochs = 150
num_steps = 35
batch_size = 32
lr = 0.25
clipping_theta = 5
prefixes = ['分开', '不分开']
pred_period = 30
pred_len = 100

gb.train_and_predict_rnn(gru_rnn, False, num_epochs, num_steps, num_hiddens,
                        lr, clipping_theta, batch_size, vocab_size,
                        pred_period, pred_len, prefixes, get_params,
                        get_inputs, ctx, corpus_indices, idx_to_char,
                        char_to_idx)
```

epoch 30, perplexity 116.103409

- 分开 我不能再不要我的可爱女人 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 我不能这样我的可爱女人 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爱

epoch 60, perplexity 10.088336

- 分开 我有一直好好的响色油 想想就一直人慢慢的回忆 没有了过去 我们着事二 我想要这样牵
- 我该着你心碎开了这样 没有你烦 我有多烦恼 没有你烦我有多烦烦多多 没有你烦我有多烦烦 我害怕你
- 我有多烦恼
- 不分开 我想能这样 我的手界被摧摧 我会想再想 我的想这好是我妈妈 我已在一步瘦 后知后觉 我该好好生活
- 我知不觉 我跟了这节奏 后知后觉 我该好好生活 我该好好生 我想要这样牵 我该着你心碎开了这样 没有

epoch 90, perplexity 1.932563

- 分开 分生那么什么找
- 不分开 心影的父我已坠入 看不见罪的国度 请原谅我的自负 没人能说没人可说 好难承受
- 荣耀的背后刻着一道孤独 闭上双眼我又看见 当年那耳濡目染 什么刀枪跟棍棒 我都要的有模有样
- 什么兵器最喜欢 双截棍柔中带刚

epoch 120, perplexity 1.133360

- 分开 这样的节奏 那天了关忆 没有三空 都有止不准说你 难上彼此的困号 才有个依靠 有太多人太多事
→ 夹在我们之间咆哮 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
- 不分开 心影的父我已坠入 看不见罪的国度 请原谅我的自负 仁慈的父我已坠入 看不见罪的国度
→ 请原谅我的自负 仁慈的父我已坠入 看不见罪的国度 请原谅我的自负 仁慈的父我已坠入 看不见罪的国度
→ 请原谅我的自负

epoch 150, perplexity 1.058174

- 分开 这样的节奏 谁都无可奈何 没有你以后 我灵魂失控 黑云在降落 我被它拖着走 静静悄悄默默离开
→ 陷入了危险边缘Baby 我的世界已狂风暴雨 Wu 爱情来的太快就像龙卷风 不能承受我已无处可躲 我不要
- 不分开 没有你慢 我给一直都有听错 带着你 彷彿心名 微止在苦泣 静静躺在抽屉 它所拥有的只剩 回忆好
→ 祭人开始注久 不少到 也什么 什么都有 沙漠之中怎么会有泥鳅 话说完飞过一只海鸥 大峡谷的风呼啸而过 是

6.5.4 小结

- 门控循环神经网络可以更好地捕捉时间序列中时间步距离较大的依赖关系，它包括门控循环单元和长短期记忆。
- 门控循环单元引入了门的概念，从而修改了循环神经网络中隐藏状态的计算方式。它包括重置门、更新门、候选隐藏状态和隐藏状态。
- 重置门有助于捕捉时间序列里短期的依赖关系。
- 更新门有助于捕捉时间序列里长期的依赖关系。

6.5.5 练习

- 假设时间步 $t' < t$ 。如果我们只希望用时间步 t' 的输入来预测时间步 t 的输出，每个时间步的重置门和更新门的值最好是多少？
- 调调超参数，观察并分析对运行时间、困惑度以及创作歌词的结果造成的影响。
- 在相同条件下，比较门控循环单元和循环神经网络的运行时间。

6.5.6 扫码直达讨论区



6.5.7 参考文献

- [1] Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259.
- [2] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555.

6.6 长短期记忆 (LSTM)

本节将介绍另一种常用的门控循环神经网络：长短期记忆 (long short-term memory, 简称 LSTM) [1]。它比门控循环单元的结构稍微更复杂一点。

6.6.1 长短期记忆

我们先介绍长短期记忆的设计。它修改了循环神经网络隐藏状态的计算方式，并引入了与隐藏状态形状相同的记忆细胞（某些文献把记忆细胞当成一种特殊的隐藏状态）。

输入门、遗忘门和输出门

假设隐藏单元个数为 h ，给定时间步 t 的小批量输入 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ （样本数为 n ，输入个数为 d ）和上一时间步隐藏状态 $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ 。时间步 t 的输入门 (input gate) $\mathbf{I}_t \in \mathbb{R}^{n \times h}$ 、遗忘门 (forget

gate) $\mathbf{F}_t \in \mathbb{R}^{n \times h}$ 和输出门 (output gate) $\mathbf{O}_t \in \mathbb{R}^{n \times h}$ 分别计算如下：

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i), \\ \mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f), \\ \mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o).\end{aligned}$$

其中的 $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$ 和 $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$ 是权重参数， $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$ 是偏移参数。激活函数 σ 是 sigmoid 函数。和门控循环单元中的重置门和更新门一样，这里的输入门、遗忘门和输出门中每个元素的值域都是 $[0, 1]$ 。

候选记忆细胞

和门控循环单元中的候选隐藏状态一样，时间步 t 的候选记忆细胞 $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$ 也使用了值域在 $[-1, 1]$ 的 \tanh 函数做激活函数。它的计算和不带门控的循环神经网络的隐藏状态的计算没什么区别：

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

其中的 $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$ 和 $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$ 是权重参数， $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$ 是偏移参数。

记忆细胞

我们可以通过元素值域在 $[0, 1]$ 的输入门、遗忘门和输出门来控制隐藏状态中信息的流动：这通常可以应用按元素乘法符 \odot 。当前时间步记忆细胞 $\mathbf{C}_t \in \mathbb{R}^{n \times h}$ 的计算组合了上一时间步记忆细胞和当前时间步候选记忆细胞的信息，并通过遗忘门和输入门来控制信息的流动：

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

需要注意的是，如果遗忘门一直近似 1 且输入门一直近似 0，过去的记忆细胞将一直通过时间保存并传递至当前时间步。这个设计可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时序数据中间隔较大的依赖关系。

隐藏状态

有了记忆细胞以后，接下来我们还可以通过输出门来控制从记忆细胞到隐藏状态 $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ 的信息的流动：

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

这里的 \tanh 函数确保隐藏状态元素值在 -1 到 1 之间。需要注意的是，当输出门近似 1，记忆细胞信息将传递到隐藏状态供输出层使用；当输出门近似 0，记忆细胞信息只自己保留。

输出层

在时间步 t ，长短期记忆的输出层计算和之前描述的循环神经网络输出层计算一样：我们只需将该时刻的隐藏状态 H_t 传递进输出层，从而计算时间步 t 的输出。

6.6.2 实验

和前几节中的实验一样，我们依然使用周杰伦歌词数据集来训练模型作词。

处理数据

我们先读取并简单处理数据集。

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

        with zipfile.ZipFile('../data/jaychou_lyrics.txt.zip', 'r') as zin:
            zin.extractall('../data/')
        with open('../data/jaychou_lyrics.txt') as f:
            corpus_chars = f.read()

        corpus_chars = corpus_chars.replace('\n', ' ').replace('\r', ' ')
        corpus_chars = corpus_chars[0:20000]
        idx_to_char = list(set(corpus_chars))
        char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
        corpus_indices = [char_to_idx[char] for char in corpus_chars]
        vocab_size = len(char_to_idx)
```

初始化模型参数

以下部分对模型参数进行初始化。超参数 `num_hiddens` 定义了隐藏单元的个数。

```

In [2]: ctx = gb.try_gpu()
        input_dim = vocab_size
        num_hiddens = 256
        output_dim = vocab_size

def get_params():
    # 输入门参数。
    W_xi = nd.random_normal(scale=0.01, shape=(input_dim, num_hiddens),
                           ctx=ctx)
    W_hi = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                           ctx=ctx)
    b_i = nd.zeros(num_hiddens, ctx=ctx)
    # 遗忘门参数。
    W_xf = nd.random_normal(scale=0.01, shape=(input_dim, num_hiddens),
                           ctx=ctx)
    W_hf = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                           ctx=ctx)
    b_f = nd.zeros(num_hiddens, ctx=ctx)
    # 输出门参数。
    W_xo = nd.random_normal(scale=0.01, shape=(input_dim, num_hiddens),
                           ctx=ctx)
    W_ho = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                           ctx=ctx)
    b_o = nd.zeros(num_hiddens, ctx=ctx)
    # 候选细胞参数。
    W_xc = nd.random_normal(scale=0.01, shape=(input_dim, num_hiddens),
                           ctx=ctx)
    W_hc = nd.random_normal(scale=0.01, shape=(num_hiddens, num_hiddens),
                           ctx=ctx)
    b_c = nd.zeros(num_hiddens, ctx=ctx)
    # 输出层参数。
    W_hy = nd.random_normal(scale=0.01, shape=(num_hiddens, output_dim),
                           ctx=ctx)
    b_y = nd.zeros(output_dim, ctx=ctx)

    params = [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc,
              b_c, W_hy, b_y]
    for param in params:
        param.attach_grad()
    return params

```

6.6.3 定义模型

下面根据长短期记忆的计算表达式定义模型。

```
In [3]: def lstm_rnn(inputs, state_h, state_c, *params):
    [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c,
     W_ty, b_y] = params
    H = state_h
    C = state_c
    outputs = []
    for X in inputs:
        I = nd.sigmoid(nd.dot(X, W_xi) + nd.dot(H, W_hi) + b_i)
        F = nd.sigmoid(nd.dot(X, W_xf) + nd.dot(H, W_hf) + b_f)
        O = nd.sigmoid(nd.dot(X, W_xo) + nd.dot(H, W_ho) + b_o)
        C_tilda = nd.tanh(nd.dot(X, W_xc) + nd.dot(H, W_hc) + b_c)
        C = F * C + I * C_tilda
        H = O * C.tanh()
        Y = nd.dot(H, W_ty) + b_y
        outputs.append(Y)
    return (outputs, H, C)
```

训练模型并创作歌词

设置好超参数后，我们将训练模型并根据前缀“分开”和“不分开”分别创作长度为 100 个字符的一段歌词。我们每过 30 个迭代周期便根据当前训练的模型创作一段歌词。训练模型时采用了相邻采样。

```
In [4]: get_inputs = gb.to_onehot
num_epochs = 150
num_steps = 35
batch_size = 32
lr = 0.25
clipping_theta = 5
prefixes = ['分开', '不分开']
pred_period = 30
pred_len = 100

gb.train_and_predict_rnn(lstm_rnn, False, num_epochs, num_steps, num_hiddens,
                        lr, clipping_theta, batch_size, vocab_size,
                        pred_period, pred_len, prefixes, get_params,
                        get_inputs, ctx, corpus_indices, idx_to_char,
                        char_to_idx, is_lstm=True)
```

epoch 30, perplexity 193.775589

- 分开 我不着我 我不了 我有我 一直我 一直我 一直我 一直我 一直我 一直我 一直我
→ 一直我 一直我
- 不分开 我不要 我不了 我有我 一直我 一直我 一直我 一直我 一直我 一直我 一直我
→ 一直我 一直我

epoch 60, perplexity 36.237839

- 分开 我想想这样样的天样 想想说你 你不是我想要你 没有你有我有你 说因的话太快 我的想好好你
→ 想过了没有 有你在我太多 你不是我不想 你的手情太太 你的天我 你想有人太 我想 这直我 我想 这些我
→ 我想
- 不分开 我想想这样 我的天界 你怎么这样 知道着你走 我的天球 是你在风太 是你在我 我有我有你牵
→ 你的手后 快你在直道 有一直人 我想我很你 你的没有 有纯了停 想在我有你 有你在我太多 你不是我太多
→ 你

epoch 90, perplexity 7.003968

- 分开 不想再 一颗两人在江南 一直等荒 在支配 在日村外的溪边 一只等待 全远欣人 木炭 一箩筐 木炭
→ 一一筐 木炭 一直放 木炭 小是存 每小的传 说一种空 不想再痛 一场一种 木炭不动 没人用 一切面
- 不分开 我的世界拆你开了 也也 你爱很来久吧? 却的想忆 在你在很太过 我说 为我很了你说
→ 我怎么我有多 多我怎 隆你我很久陪你要我 别想说 你只是那别 别着 却胸在我了了?的太

epoch 120, perplexity 2.626501

- 分开 不想安 一场常人在你望等到 还要到过去 那盒时在我的画里 他地激备 是小两外昏都占卜
→ 放在放午三点阳许 教教杂空的邪度 不会骑扫的转小 维持纯白的象 然说还回忆 爷人两人太走 篮下下种秋棍
→ 漂亮的假板
- 不分开了我 想要一名到底 你的你是我的睡笑 想想太说 爱出出早 快静事种种不事 盒底底 它边到间走
→ 是很就很起快快 我会耍慢风 三一两人味 什么我看见我都一场悲剧 我拉我自己子这一个人演戏
→ 我却想要一个人但

epoch 150, perplexity 1.577065

- 分开 是分上 的父时 翻变 变变残忍 在师止中的战争 让真壁的消 有一种味道叫做家 陆羽泡的茶
→ 听说名和利都不拿 他牵着一匹瘦马在走天涯 爷爷泡的茶 有一种味道叫做家 他满头白发 喝茶时不准说话
→ 陆羽泡的茶
- 不分开整 没有帮着你 才化让曾经 希望他是要里比也乐及 脑袋面没有只才方向 不好好 一定有人向你挑战
→ 到底还要过多少关 不用怕 告诉他们谁是男子汉 可不可以不要这个奖 不想问 我只想要留一点汗
→ 我当我自己的裁

6.6.4 小结

- 长短期记忆的隐藏层输出包括隐藏状态和记忆细胞。只有隐藏状态会传递进输出层。
- 长短期记忆的输入门、遗忘门和输出门可以控制信息的流动。

- 长短期记忆的可以应对循环神经网络中的梯度衰减问题，并更好地捕捉时序数据中间隔较大的依赖关系。

6.6.5 练习

- 调调超参数，观察并分析对运行时间、困惑度以及创作歌词的结果造成的影响。
- 在相同条件下，比较长短期记忆、门控循环单元和不带门控的循环神经网络的运行时间。
- 既然候选记忆细胞已通过使用 \tanh 函数确保值域在 -1 到 1 之间，为什么隐藏状态还需再次使用 \tanh 函数来确保输出值域在 -1 到 1 之间？

6.6.6 扫码直达讨论区



6.6.7 参考文献

- [1] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735-1780.

6.7 深度循环神经网络

本章到目前为止介绍的循环神经网络只有一个单向的隐藏层：隐藏状态里的信息沿着时间步从早到晚依次传递。在实际中，我们有时会用到其他架构的循环神经网络。本节和下一节将分别介绍多隐藏层和双向架构。它们分别称作深度循环神经网络和双向循环神经网络。

给定时间步 t 的小批量输入 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (样本数为 n ，输入个数为 d)。在深度循环神经网络中，设该时间步第 l 隐藏层的隐藏状态为 $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$ (隐藏单元个数为 h)，输出层变量为 $\mathbf{O}_t \in \mathbb{R}^{n \times q}$

(输出个数为 q), 隐藏层的激活函数为 ϕ 。第一隐藏层的隐藏状态和之前的计算一样:

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

其中权重 $\mathbf{W}_{xh}^{(1)} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh}^{(1)} \in \mathbb{R}^{h \times h}$ 和偏差 $\mathbf{b}_h^{(1)} \in \mathbb{R}^{1 \times h}$ 分别为第一隐藏层的模型参数。

假设隐藏层个数为 L , 当 $1 < l \leq L$ 时, 第 l 隐藏层的隐藏状态的表达式为

$$\mathbf{H}_t^{(l)} = \phi(\mathbf{H}_t^{(l-1)} \mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)} \mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}),$$

其中权重 $\mathbf{W}_{xh}^{(l)} \in \mathbb{R}^{h \times h}$, $\mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h \times h}$ 和偏差 $\mathbf{b}_h^{(l)} \in \mathbb{R}^{1 \times h}$ 分别为第 l 隐藏层的模型参数。

最终, 输出层的输出只需基于第 L 隐藏层的隐藏状态:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{hy} + \mathbf{b}_y,$$

其中权重 $\mathbf{W}_{hy} \in \mathbb{R}^{h \times q}$ 和偏差 $\mathbf{b}_y \in \mathbb{R}^{1 \times q}$ 为输出层的模型参数。

深度循环神经网络的架构如图 6.3 所示。隐藏状态的信息不断传递至当前层的下一时间步和当前时间步的下一层。

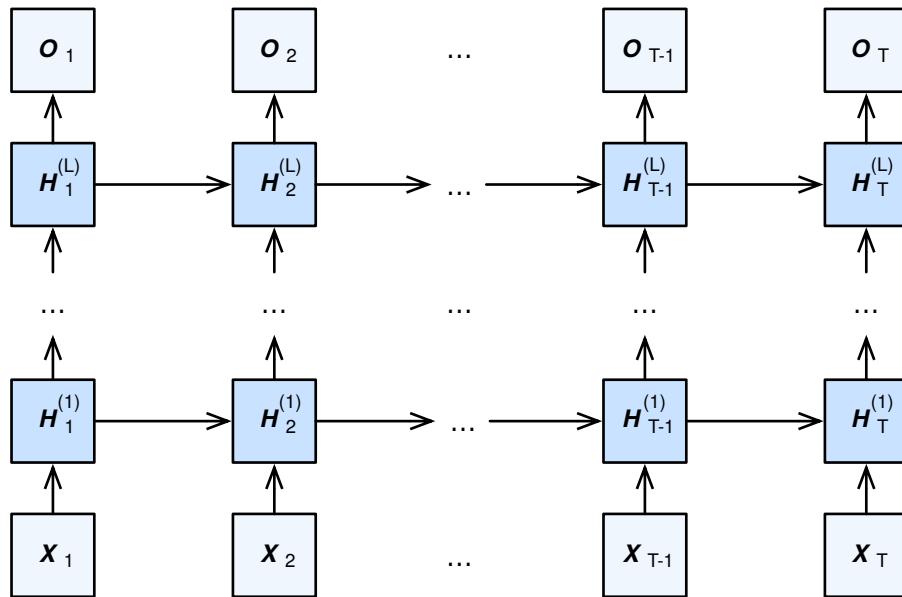


图 6.3: 深度循环神经网络的架构。

我们将在本章最后一节用 Gluon 实验深度循环神经网络。

6.7.1 小结

- 在深度循环神经网络中，隐藏状态的信息不断传递至当前层的下一时间步和当前时间步的下一层。

6.7.2 练习

- 将“循环神经网络”一节中的模型改为含有 2 个隐藏层的循环神经网络。观察并分析实验现象。

6.7.3 扫码直达讨论区



6.8 双向循环神经网络

下面我们介绍双向循环神经网络的架构。

给定时间步 t 的小批量输入 $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (样本数为 n , 输入个数为 d) 和隐藏层激活函数为 ϕ 。在双向架构中, 设该时间步正向隐藏状态为 $\vec{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ (正向隐藏单元个数为 h), 反向隐藏状态为 $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ (反向隐藏单元个数为 h)。我们可以分别计算正向和反向隐藏状态:

$$\begin{aligned}\vec{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \vec{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}), \\ \overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}),\end{aligned}$$

其中权重 $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}$, $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$ 和偏差 $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$, $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$ 均为模型参数。

双向循环神经网络在时间步 t 的隐藏状态 $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$ 即连结两个方向的隐藏状态 $\vec{\mathbf{H}}_t$ 和 $\overleftarrow{\mathbf{H}}_t$ 的结果。输出层只需基于连结后的隐藏状态计算输出 $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (输出个数为 q):

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hy} + \mathbf{b}_y,$$

其中权重 $\mathbf{W}_{hy} \in \mathbb{R}^{2h \times q}$ 和偏差 $\mathbf{b}_y \in \mathbb{R}^{1 \times q}$ 为输出层的模型参数。

双向循环神经网络架构如图 6.4 所示。和前面介绍的单向循环神经网络不同，给定一段时间序列，双向循环神经网络在每个时间步的隐藏状态同时取决于该时间步之前和之后的子序列（包括当前时间步的输入），并编码了整个序列的信息。

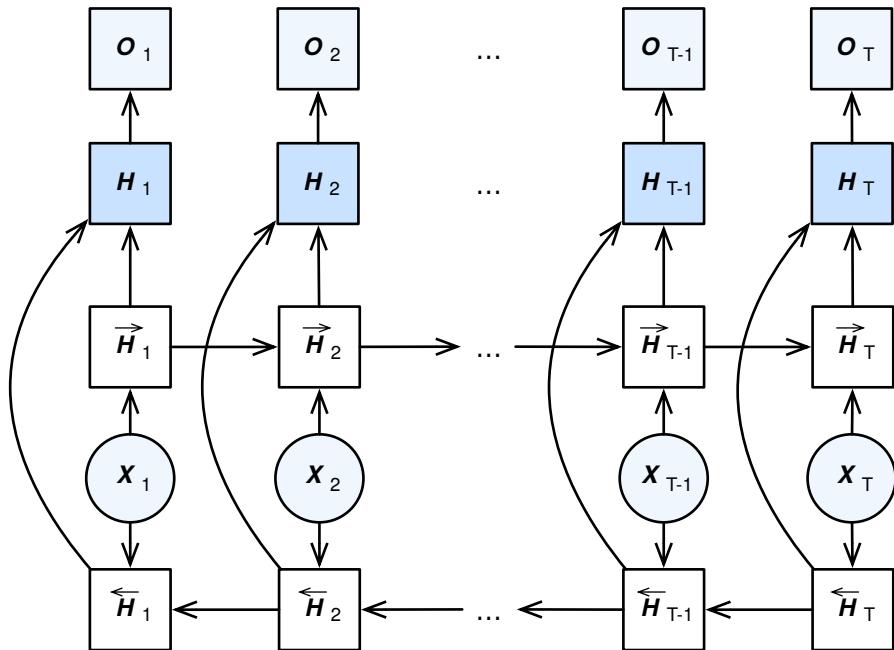


图 6.4: 双向循环神经网络架构。

我们将在“自然语言处理”篇章中应用并实验双向循环神经网络。

6.8.1 小结

- 双向循环神经网络在每个时间步的隐藏状态同时取决于该时间步之前和之后的子序列（包括当前时间步的输入）。

6.8.2 练习

- 参考图 6.3 和图 6.4，设计含多个隐藏层的双向循环神经网络。

6.8.3 扫码直达讨论区



6.9 循环神经网络的 Gluon 实现

本节介绍如何使用 Gluon 训练循环神经网络。

6.9.1 Penn Tree Bank 数据集

我们以英文单词为单元来训练基于循环神经网络的语言模型。Penn Tree Bank (PTB) 是一个标准的文本序列数据集 [1]。它包括训练集、验证集和测试集。

首先导入实验所需的包或模块，并抽取数据集。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        import math
        import mxnet as mx
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import loss as gloss, nn, rnn, utils as gutils
        import numpy as np
        import time
        import zipfile

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

6.9.2 建立词语索引

下面定义了 `Dictionary` 类来映射词语和整数索引。

```
In [2]: class Dictionary(object):
    def __init__(self):
        self.word_to_idx = {}
        self.idx_to_word = []

    def add_word(self, word):
        if word not in self.word_to_idx:
            self.idx_to_word.append(word)
            self.word_to_idx[word] = len(self.idx_to_word) - 1
        return self.word_to_idx[word]

    def __len__(self):
        return len(self.idx_to_word)
```

以下的 `Corpus` 类按照读取的文本数据集建立映射词语和索引的词典，并将文本转换成词语索引的序列。这样，每个文本数据集就变成了 `NDArray` 格式的整数序列。

```
In [3]: class Corpus(object):
    def __init__(self, path):
        self.dictionary = Dictionary()
        self.train = self.tokenize(path + 'train.txt')
        self.valid = self.tokenize(path + 'valid.txt')
        self.test = self.tokenize(path + 'test.txt')

    def tokenize(self, path):
        # 将词语添加至词典。
        with open(path, 'r') as f:
            num_words = 0
            for line in f:
                words = line.split() + ['<eos>']
                num_words += len(words)
                for word in words:
                    self.dictionary.add_word(word)
        # 将文本转换成词语索引的序列（NDArray 格式）。
        with open(path, 'r') as f:
            indices = np.zeros((num_words,), dtype='int32')
            idx = 0
            for line in f:
                words = line.split() + ['<eos>']
                for word in words:
                    indices[idx] = self.dictionary.word_to_idx[word]
                    idx += 1
        return nd.array(indices, dtype='int32')
```

看一下词典的大小。

```
In [4]: data = '../data/ptb/ptb.'
corpus = Corpus(data)
vocab_size = len(corpus.dictionary)
vocab_size

Out[4]: 10000
```

6.9.3 定义循环神经网络模型库

我们可以定义一个循环神经网络模型库。这样我们就可以使用以 ReLU 或 tanh 函数为激活函数的循环神经网络，以及长短期记忆和门控循环单元。和本章中其他实验不同，这里使用了 Embedding 实例将每个词索引转换成一个长度为 `embed_size` 的词向量。这些词向量实际上也是模型参数。在随机初始化后，它们会在模型训练结束时被学到。此外，我们使用了丢弃法来应对过拟合。

这里的 Embedding 实例也叫嵌入层。我们会在“自然语言处理”篇章介绍如何用在大规模语料上预训练的词向量初始化嵌入层参数。

```
In [5]: class RNNModel(nn.Block):
    def __init__(self, mode, vocab_size, embed_size, num_hiddens,
                 num_layers, drop_prob=0.5, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        with self.name_scope():
            self.dropout = nn.Dropout(drop_prob)
            # 将词索引转换成词向量。这些词向量也是模型参数。
            self.embedding = nn.Embedding(
                vocab_size, embed_size, weight_initializer=init.Uniform(0.1))
            if mode == 'rnn_relu':
                self.rnn = rnn.RNN(num_hiddens, num_layers, activation='relu',
                                   dropout=drop_prob, input_size=embed_size)
            elif mode == 'rnn_tanh':
                self.rnn = rnn.RNN(num_hiddens, num_layers, activation='tanh',
                                   dropout=drop_prob, input_size=embed_size)
            elif mode == 'lstm':
                self.rnn = rnn.LSTM(num_hiddens, num_layers,
                                   dropout=drop_prob, input_size=embed_size)
            elif mode == 'gru':
                self.rnn = rnn.GRU(num_hiddens, num_layers, dropout=drop_prob,
                                   input_size=embed_size)
            else:
                raise ValueError("Invalid mode %s. Options are rnn_relu, "
                                "rnn_tanh, lstm, and gru" % mode)
```

```

        self.dense = nn.Dense(vocab_size, in_units=num_hiddens)
        self.num_hiddens = num_hiddens

    def forward(self, inputs, state):
        embedding = self.dropout(self.embedding(inputs))
        output, state = self.rnn(embedding, state)
        output = self.dropout(output)
        output = self.dense(output.reshape((-1, self.num_hiddens)))
        return output, state

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

```

6.9.4 设置超参数

我们接着设置超参数。这里选择使用以 ReLU 为激活函数的循环神经网络。它包含 2 个隐藏层。为了得到更好的实验结果，这些超参数还需要重新设置。

```

In [6]: model_name = 'rnn_relu'
embed_size = 100
num_hiddens = 100
num_layers = 2
lr = 0.5
clipping_theta = 0.2
num_epochs = 2
batch_size = 32
num_steps = 5
drop_prob = 0.2
eval_period = 1000

ctx = gb.try_gpu()
model = RNNModel(model_name, vocab_size, embed_size, num_hiddens, num_layers,
                  drop_prob)
model.initialize(init.Xavier(), ctx=ctx)
trainer = gluon.Trainer(model.collect_params(), 'sgd',
                        {'learning_rate': lr, 'momentum': 0, 'wd': 0})
loss = gloss.SoftmaxCrossEntropyLoss()

```

6.9.5 相邻采样

我们将在实验中使用相邻采样。

```
In [7]: def batchify(data, batch_size):
    num_batches = data.shape[0] // batch_size
    data = data[:num_batches*batch_size]
    data = data.reshape((batch_size, num_batches)).T
    return data

train_data = batchify(corpus.train, batch_size).as_in_context(ctx)
val_data = batchify(corpus.valid, batch_size).as_in_context(ctx)
test_data = batchify(corpus.test, batch_size).as_in_context(ctx)

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

“循环神经网络”一节里已经解释了，相邻采样应在每次读取小批量前将隐藏状态从计算图分离出来。

```
In [8]: def detach(state):
    if isinstance(state, (tuple, list)):
        state = [i.detach() for i in state]
    else:
        state = state.detach()
    return state
```

6.9.6 训练和评价模型

以下定义了模型评价函数。

```
In [9]: def eval_rnn(data_source):
    l_sum = nd.array([0], ctx=ctx)
    n = 0
    state = model.begin_state(func=nd.zeros, batch_size=batch_size, ctx=ctx)
    for i in range(0, data_source.shape[0] - 1, num_steps):
        X, y = get_batch(data_source, i)
        output, state = model(X, state)
        l = loss(output, y)
        l_sum += l.sum()
        n += l.size
    return l_sum / n
```

下面的 `train_rnn` 函数将训练模型并在每个迭代周期结束时评价模型在验证集上的表现。我

们可以参考验证集上的结果调节超参数。

```
In [10]: def train_rnn():
    for epoch in range(1, num_epochs + 1):
        train_l_sum = nd.array([0], ctx=ctx)
        start_time = time.time()
        state = model.begin_state(func=nd.zeros, batch_size=batch_size,
                                   ctx=ctx)
        for batch_i, idx in enumerate(range(0, train_data.shape[0] - 1,
                                            num_steps)):
            X, y = get_batch(train_data, idx)
            # 从计算图分离隐藏状态变量（包括 LSTM 的记忆细胞）。
            state = detach(state)
            with autograd.record():
                output, state = model(X, state)
                # l 形状: (batch_size * num_steps,)。
                l = loss(output, y).sum() / (batch_size * num_steps)
            l.backward()
            grads = [p.grad(ctx) for p in model.collect_params().values()]
            # 梯度裁剪。需要注意的是，这里的梯度是整个批量的梯度。
            # 因此我们将 clipping_theta 乘以 num_steps 和 batch_size。
            gutils.clip_global_norm(
                grads, clipping_theta * num_steps * batch_size)
            trainer.step(1)
            train_l_sum += l
            if batch_i % eval_period == 0 and batch_i > 0:
                cur_l = train_l_sum / eval_period
                print('epoch %d, batch %d, train loss %.2f, perplexity %.2f'
                      % (epoch, batch_i, cur_l.asscalar(),
                         cur_l.exp().asscalar()))
                train_l_sum = nd.array([0], ctx=ctx)
            val_l = eval_rnn(val_data)
            print('epoch %d, time %.2fs, valid loss %.2f, perplexity %.2f'
                  % (epoch, time.time() - start_time, val_l.asscalar(),
                     val_l.exp().asscalar()))
```

训练完模型以后，我们就可以在测试集上评价模型了。

```
In [11]: train_rnn()
test_l = eval_rnn(test_data)
print('test loss %.2f, perplexity %.2f'
      % (test_l.asscalar(), test_l.exp().asscalar()))

epoch 1, batch 1000, train loss 7.21, perplexity 1356.29
epoch 1, batch 2000, train loss 6.43, perplexity 618.23
```

```
epoch 1, batch 3000, train loss 6.22, perplexity 502.70
epoch 1, batch 4000, train loss 6.11, perplexity 450.60
epoch 1, batch 5000, train loss 6.03, perplexity 413.85
epoch 1, time 42.45s, valid loss 5.85, perplexity 348.87
epoch 2, batch 1000, train loss 5.94, perplexity 379.93
epoch 2, batch 2000, train loss 5.88, perplexity 358.65
epoch 2, batch 3000, train loss 5.80, perplexity 329.77
epoch 2, batch 4000, train loss 5.76, perplexity 316.76
epoch 2, batch 5000, train loss 5.71, perplexity 301.73
epoch 2, time 42.77s, valid loss 5.61, perplexity 273.55
test loss 5.57, perplexity 262.83
```

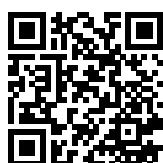
6.9.7 小结

- 我们可以使用 Gluon 训练循环神经网络。它更简洁，例如无需我们手动实现含有多个隐藏层的复杂模型。
- 在训练语言模型时，我们可以将词索引转换成词向量，并将这些词向量视为模型参数。

6.9.8 练习

- 回忆“模型参数的访问、初始化和共享”一节中有关共享模型参数的描述。将本节中 RNNModel 类里的 `self.dense` 的定义改为 `nn.Dense(vocab_size, in_units = num_hiddens, params=self.embedding.params)` 并运行本节实验。这里为什么可以共享词向量参数？有哪些好处？
- 调调超参数，观察并分析对运行时间以及训练集、验证集和测试集上困惑度的影响。

6.9.9 扫码直达讨论区



6.9.10 参考文献

[1] Penn Tree Bank. <https://catalog.ldc.upenn.edu/ldc99t42>

优化算法

如果你一直按照本书的顺序读到这里，很可能已经使用了优化算法来训练深度学习模型。具体来说，在训练模型时，我们会使用优化算法不断迭代模型参数以最小化模型的损失函数。当迭代终止时，模型的训练随之终止。此时的模型参数就是模型通过训练所学习到的参数。

优化算法对于深度学习十分重要。一方面，训练一个复杂的深度学习模型可能需要数小时、数日、甚至数周时间。而优化算法的表现直接影响模型训练效率。另一方面，理解各种优化算法的原理以及其中各参数的意义将有助于我们更有针对性地调参，从而使深度学习模型表现地更好。

本章将详细介绍深度学习中的常用优化算法。

7.1 优化算法概述

本节将讨论优化与深度学习的关系以及优化在深度学习中的挑战。

7.1.1 优化与深度学习

在一个深度学习问题中，通常我们会预先定义一个损失函数。有了损失函数以后，我们就可以使用优化算法试图使其最小化。在优化中，这样的损失函数通常被称作优化问题的目标函数（objective function）。依据惯例，优化算法通常只考虑最小化目标函数。其实，任何最大化问题都可以很容易地转化为最小化问题：我们只需把目标函数前面的正号或负号取相反。

虽然优化为深度学习提供了最小化损失函数的方法，但本质上，这两者之间的目标是有区别的。在“欠拟合、过拟合和模型选择”一节中，我们区分了训练误差和泛化误差。由于优化算法的目标函数通常是一个基于训练数据集的损失函数，优化的目标在于降低训练误差。而深度学习的目标在于降低泛化误差。为了降低泛化误差，除了使用优化算法降低训练误差以外，我们还需要注意应对过拟合。

本章中，我们只关注优化算法在最小化目标函数上的表现，而不关注模型的泛化误差。

7.1.2 优化在深度学习中的挑战

绝大多数深度学习中的目标函数都很复杂。因此，很多优化问题并不存在解析解，而需要使用基于数值方法的优化算法找到近似解。这类优化算法一般通过不断迭代更新解的数值来找到近似解。我们讨论的优化算法都是这类基于数值方法的算法。

优化在深度学习中有很多挑战。以下描述了其中的两个挑战：局部最小值和鞍点。为了更好地描述问题，我们先导入本节中实验需要的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import gluonbook as gb
        from mpl_toolkits import mplot3d
        import numpy as np
```

局部最小值

对于目标函数 $f(x)$ ，如果 $f(x)$ 在 x 上的值比在 x 邻近的其他点的值更小，那么 $f(x)$ 可能是一个局部最小值（local minimum）。如果 $f(x)$ 在 x 上的值是目标函数在整个定义域上的最小值，那么 $f(x)$ 是全局最小值（global minimum）。

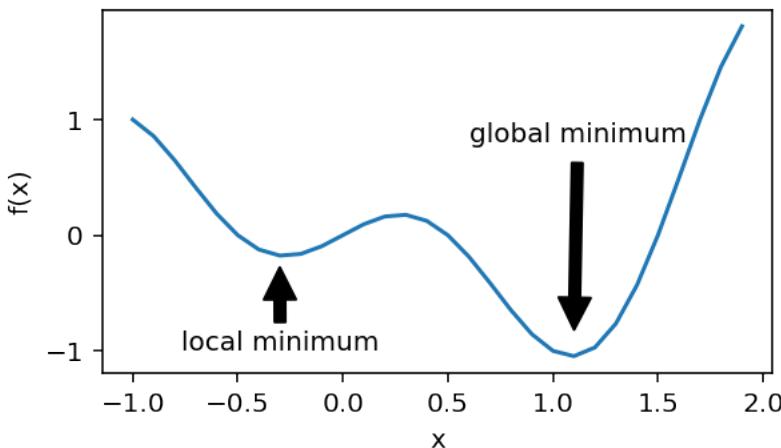
举个例子，给定函数

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

我们可以大致找出该函数的局部最小值和全局最小值的位置。需要注意的是，图中箭头所指示的只是大致位置。

```
In [2]: def f(x):
    return x * np.cos(np.pi * x)

gb.pyplot.rcParams['figure.figsize'] = (4.5, 2.5)
x = np.arange(-1.0, 2.0, 0.1)
fig = gb.pyplot.figure()
subplt = fig.add_subplot(111)
subplt.annotate('local minimum', xy=(-0.3, -0.25), xytext=(-0.77, -1.0),
                arrowprops=dict(facecolor='black', shrink=0.05))
subplt.annotate('global minimum', xy=(1.1, -0.9), xytext=(0.6, 0.8),
                arrowprops=dict(facecolor='black', shrink=0.05))
gb.pyplot.plot(x, f(x))
gb.pyplot.xlabel('x')
gb.pyplot.ylabel('f(x)')
gb.pyplot.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优化而非全局最小化。

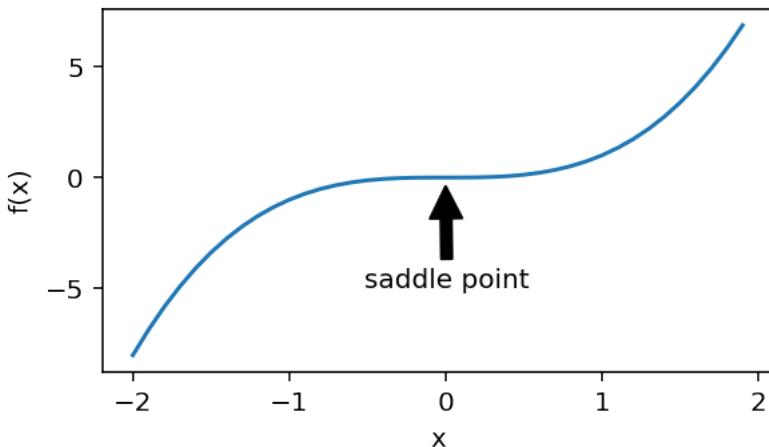
鞍点

刚刚我们提到，梯度接近或变成零可能是由于当前解在局部最优解附近所造成的。事实上，另一种可能性是当前解在鞍点（saddle point）附近。举个例子，给定函数

$$f(x) = x^3,$$

我们可以找出该函数的鞍点位置。

```
In [3]: x = np.arange(-2.0, 2.0, 0.1)
fig = gb.pyplot.figure()
subplt = fig.add_subplot(111)
subplt.annotate('saddle point', xy=(0, -0.2), xytext=(-0.52, -5.0),
                arrowprops=dict(facecolor='black', shrink=0.05))
gb.pyplot.plot(x, x**3)
gb.pyplot.xlabel('x')
gb.pyplot.ylabel('f(x)')
gb.pyplot.show()
```



再举个定义在二维空间的函数的例子，例如

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

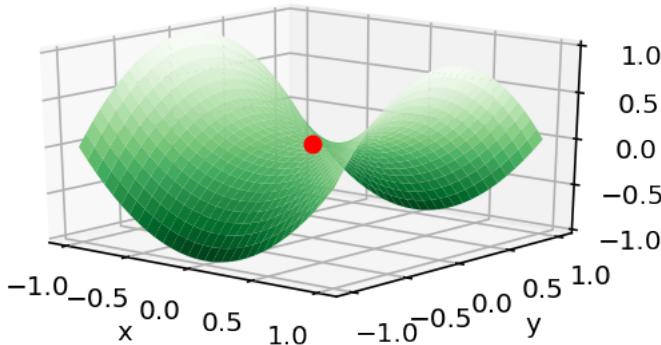
我们可以找出该函数的鞍点位置。也许你已经发现了，该函数看起来像一个马鞍，而鞍点恰好是马鞍上可坐区域的中心。

```
In [4]: fig = gb.pyplot.figure()
ax = fig.add_subplot(111, projection='3d')
```

```

x, y = np.mgrid[-1:1:31j, -1:1:31j]
z = x**2 - y**2
ax.plot_surface(x, y, z, **{'rstride': 1, 'cstride': 1, 'cmap': "Greens_r"})
ax.plot([0], [0], [0], 'ro')
ax.view_init(azim=-50, elev=20)
gb=plt.xticks([-1, -0.5, 0, 0.5, 1])
gb=plt.yticks([-1, -0.5, 0, 0.5, 1])
ax.set_zticks([-1, -0.5, 0, 0.5, 1])
gb=plt.xlabel('x')
gb=plt.ylabel('y')
gb=plt.show()

```



在上图的鞍点位置，目标函数在 x 轴方向上是局部最小值，而在 y 轴方向上是局部最大值。

假设一个函数的输入为 k 维向量，输出为标量，那么它的黑塞矩阵（Hessian matrix）有 k 个特征值。需要注意的是，该函数在梯度为零的位置上可能是局部最小值、局部最大值或者鞍点：

- 当函数的黑塞矩阵在梯度为零的位置上的特征值全为正时，该函数得到局部最小值。
- 当函数的黑塞矩阵在梯度为零的位置上的特征值全为负时，该函数得到局部最大值。
- 当函数的黑塞矩阵在梯度为零的位置上的特征值有正有负时，该函数得到鞍点。

随机矩阵理论告诉我们，对于一个大的高斯随机矩阵来说，任一特征值是正或者是负的概率都是 0.5 [1]。那么，以上第一种情况的概率为 0.5^k 。由于深度学习模型参数通常都是高维的 (k 很大)，目标函数的鞍点通常比局部最小值更常见。

深度学习中，虽然找到目标函数的全局最优解很难，但这并非必要。我们将在接下来的章节中逐一介绍深度学习中常用的优化算法，它们在很多实际问题中都训练出了十分有效的深度学习模型。

7.1.3 小结

- 由于优化算法的目标函数通常是一个基于训练数据集的损失函数，优化的目标在于降低训练误差。
- 由于深度学习模型参数通常都是高维的，目标函数的鞍点通常比局部最小值更常见。

7.1.4 练习

- 你还能想到哪些深度学习中的优化问题的挑战？

7.1.5 扫码直达讨论区



7.1.6 参考文献

[1] Wigner, E. P. (1958). On the distribution of the roots of certain symmetric matrices. *Annals of Mathematics*, 325-327.

7.2 梯度下降和随机梯度下降

本节中，我们将介绍梯度下降 (gradient descent) 和随机梯度下降 (stochastic gradient descent) 算法。由于梯度下降是优化算法的核心部分，理解梯度的涵义十分重要。为了帮助大家深刻理解梯度，我们将从数学角度阐释梯度下降的意义。本节中，我们假设目标函数连续可导。

7.2.1 一维梯度下降

我们先以简单的一维梯度下降为例，解释梯度下降算法可以降低目标函数值的原因。一维梯度是一个标量，也称导数。

假设函数 $f : \mathbb{R} \rightarrow \mathbb{R}$ 的输入和输出都是标量。给定足够小的数 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

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

也就是说, 如果目标函数 $f(x)$ 当前的导数 $f'(x) \neq 0$, 按照

$$x \leftarrow x - \eta f'(x).$$

迭代自变量 x 可能会降低 $f(x)$ 的值。由于导数 $f'(x)$ 是梯度 $\nabla_x f$ 在一维空间的特殊情况, 上述迭代自变量 x 的方法也即一维空间的梯度下降。一维空间的梯度下降图 7.1 (左) 所示, 自变量 x 沿着梯度方向迭代。

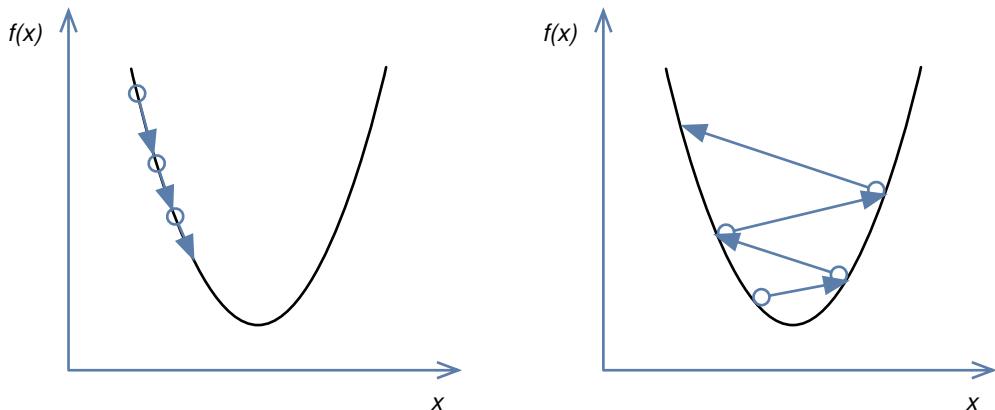


图 7.1: 梯度下降中, 目标函数 $f(x)$ 的自变量 x (圆圈的横坐标) 沿着梯度方向迭代。

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 7.1（右）。

然而，如果学习率过小，目标函数中自变量的收敛速度会过慢。实际中，一个合适的学习率通常需要通过多次实验找到的。

7.2.3 多维梯度下降

现在我们考虑一个更广义的情况：目标函数的输入为向量，输出为标量。

假设目标函数 $f : \mathbb{R}^d \rightarrow \mathbb{R}$ 的输入是一个 d 维向量 $\mathbf{x} = [x_1, x_2, \dots, x_d]^\top$ 。目标函数 $f(\mathbf{x})$ 有关 \mathbf{x} 的梯度是一个由 d 个偏导数组成的向量：

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

为表示简洁，我们用 $\nabla f(\mathbf{x})$ 代替 $\nabla_{\mathbf{x}} f(\mathbf{x})$ 。梯度中每个偏导数元素 $\partial f(\mathbf{x})/\partial x_i$ 代表着 f 在 \mathbf{x} 有关输入 x_i 的变化率。为了测量 f 沿着单位向量 \mathbf{u} 方向上的变化率，在多元微积分中，我们定义 f 在 \mathbf{x} 上沿着 \mathbf{u} 方向的方向导数为

$$D_{\mathbf{u}} f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{u}) - f(\mathbf{x})}{h}.$$

依据方向导数性质 [1, 14.6 节定理三]，该方向导数可以改写为

$$D_{\mathbf{u}} f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{u}.$$

方向导数 $D_{\mathbf{u}} f(\mathbf{x})$ 给出了 f 在 \mathbf{x} 上沿着所有可能方向的变化率。为了最小化 f ，我们希望找到 f 能被降低最快的方向。因此，我们可以通过单位向量 \mathbf{u} 来最小化方向导数 $D_{\mathbf{u}} f(\mathbf{x})$ 。

由于 $D_{\mathbf{u}} f(\mathbf{x}) = \|\nabla f(\mathbf{x})\| \cdot \|\mathbf{u}\| \cdot \cos(\theta) = \|\nabla f(\mathbf{x})\| \cdot \cos(\theta)$ ，其中 θ 为梯度 $\nabla f(\mathbf{x})$ 和单位向量 \mathbf{u} 之间的夹角，当 $\theta = \pi$ ， $\cos(\theta)$ 取得最小值 -1。因此，当 \mathbf{u} 在梯度方向 $\nabla f(\mathbf{x})$ 的相反方向时，方向导数 $D_{\mathbf{u}} f(\mathbf{x})$ 被最小化。所以，我们可能通过下面的梯度下降算法来不断降低目标函数 f 的值：

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

相同地，其中 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

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

其中 $f_i(\mathbf{x})$ 是有关索引为 i 的训练数据样本的损失函数， n 是训练数据样本数。由于

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

梯度下降每次迭代的计算开销随着 n 线性增长。因此，当训练数据样本数很大时，梯度下降每次迭代的计算开销很高。这时我们可以使用随机梯度下降。给定学习率 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 $\nabla f_i(\mathbf{x})$ 来迭代 \mathbf{x} ：

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}).$$

事实上，随机梯度 $\nabla f_i(\mathbf{x})$ 是对梯度 $\nabla f(\mathbf{x})$ 的无偏估计：

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

7.2.5 小批量随机梯度下降

广义上，每一次迭代可以随机均匀采样一个由训练数据样本索引所组成的小批量（mini-batch） \mathcal{B} 。一般来说，我们可以通过重复采样（sampling with replacement）或者不重复采样（sampling without replacement）得到同一个小批量中的各个样本。前者允许同一个小批量中出现重复的样本，后者则不允许如此。对于这两者间的任一种方式，我们可以使用

$$\nabla f_{\mathcal{B}}(\mathbf{x}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla f_i(\mathbf{x})$$

来迭代 \mathbf{x} ：

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_{\mathcal{B}}(\mathbf{x}).$$

在上式中， $|\mathcal{B}|$ 代表样本批量大小， η （取正数）称作学习率。同样，小批量随机梯度 $\nabla f_{\mathcal{B}}(\mathbf{x})$ 也是对梯度 $\nabla f(\mathbf{x})$ 的无偏估计：

$$\mathbb{E}_{\mathcal{B}} \nabla f_{\mathcal{B}}(\mathbf{x}) = \nabla f(\mathbf{x}).$$

这个算法叫做小批量随机梯度下降。该算法每次迭代的计算开销为 $\mathcal{O}(|\mathcal{B}|)$ 。当批量大小为 1 时，该算法即随机梯度下降；当批量大小等于训练数据样本数，该算法即梯度下降。和学习率一样，批量大小也是一个超参数。当批量较小时，虽然每次迭代的计算开销较小，但计算机并行处理批量中各个样本的能力往往只得到较少利用。因此，当训练数据集的样本较少时，我们可以使用梯度下降；当样本较多时，我们可以使用小批量梯度下降并依据计算资源选择合适的批量大小。

实际中，我们通常在每个迭代周期（epoch）开始前随机打乱数据集中样本的先后顺序，然后在同一个迭代周期中依次读取小批量的样本。理论上，这种方法能让符合某些特征的目标函数的优化收敛更快 [2]。我们在实验中也用这种方法随机采样小批量样本。

7.2.6 小批量随机梯度下降的实现

我们只需要实现小批量随机梯度下降。当批量大小等于训练集大小时，该算法即为梯度下降；批量大小为 1 即为随机梯度下降。

```
In [1]: def sgd(params, lr, batch_size):
    for param in params:
        param[:] = param - lr * param.grad / batch_size
```

7.2.7 实验

首先，导入本节中实验所需的包或模块。

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

实验中，我们以之前介绍过的线性回归为例。我们直接调用 `gluonbook` 中的线性回归模型和平方损失函数。它们已在“[线性回归的从零开始实现](#)”一节中实现过了。

```
In [3]: net = gb.linreg
loss = gb.squared_loss
```

设数据集的样本数为 1000，我们使用权重 `w` 为 $[2, -3.4]$ ，偏差 “`b`” 为 4.2 的线性回归模型来生成数据集。该模型的平方损失函数即所需优化的目标函数，模型参数即目标函数自变量。

```
In [4]: # 生成数据集。
num_inputs = 2
```

```

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

# 初始化模型参数。
def init_params():
    w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    for param in params:
        param.attach_grad()
    return params

```

下面我们描述一下优化函数 `optimize`。

由于随机梯度的方差在迭代过程中无法减小，（小批量）随机梯度下降的学习率通常会采用自我衰减的方式。如此一来，学习率和随机梯度乘积的方差会衰减。实验中，当迭代周期（epoch）大于 2 时，（小批量）随机梯度下降的学习率在每个迭代周期开始时自乘 0.1 作自我衰减。而梯度下降在迭代过程中一直使用目标函数的真实梯度，无需自我衰减学习率。

在迭代过程中，每当 `log_interval` 个样本被采样过后，模型当前的损失函数值（`loss`）被记录下并用于作图。例如，当 `batch_size` 和 `log_interval` 都为 10 时，每次迭代后的损失函数值都被用来作图。

```

In [5]: def optimize(batch_size, lr, num_epochs, log_interval, decay_epoch):
    w, b = init_params()
    ls = [loss(net(features, w, b), labels).mean().asnumpy()]
    for epoch in range(1, num_epochs + 1):
        # 学习率自我衰减。
        if decay_epoch and epoch > decay_epoch:
            lr *= 0.1
        for batch_i, (X, y) in enumerate(
            gb.data_iter(batch_size, features, labels)):
            with autograd.record():
                l = loss(net(X, w, b), y)
            # 先对 l 中元素求和，得到小批量损失之和，然后求参数的梯度。
            l.backward()
            sgd([w, b], lr, batch_size)
            if batch_i * batch_size % log_interval == 0:
                ls.append(loss(net(features, w, b), labels).mean().asnumpy())

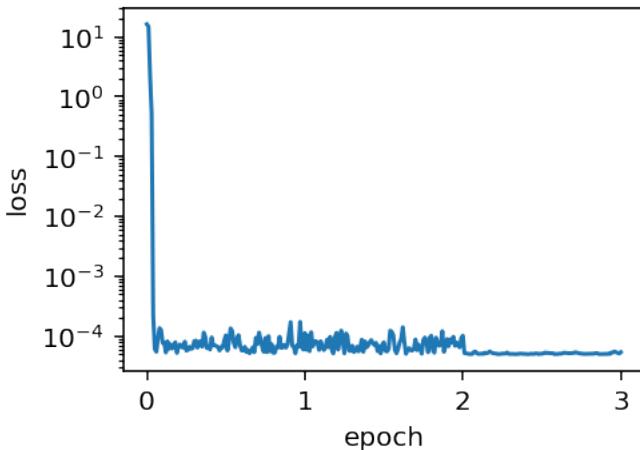
```

```
print('w:', w, '\nb:', b, '\n')
es = np.linspace(0, num_epochs, len(ls), endpoint=True)
gb.semilogy(es, ls, 'epoch', 'loss')
```

当批量大小为 1 时，优化使用的是随机梯度下降。在当前学习率下，损失函数值在早期快速下降后略有波动。这是由于随机梯度的方差在迭代过程中无法减小。当迭代周期大于 2，学习率自我衰减后，损失函数值下降后较平稳。最终，优化所得的模型参数值 w 和 b 与它们的真实值 [2, -3.4] 和 4.2 较接近。

```
In [6]: optimize(batch_size=1, lr=0.2, num_epochs=3, decay_epoch=2, log_interval=10)
```

```
w:  
[[ 2.00162578]  
 [-3.4001534 ]]  
<NDArray 2x1 @cpu(0)>  
b:  
[ 4.20178843]  
<NDArray 1 @cpu(0)>
```

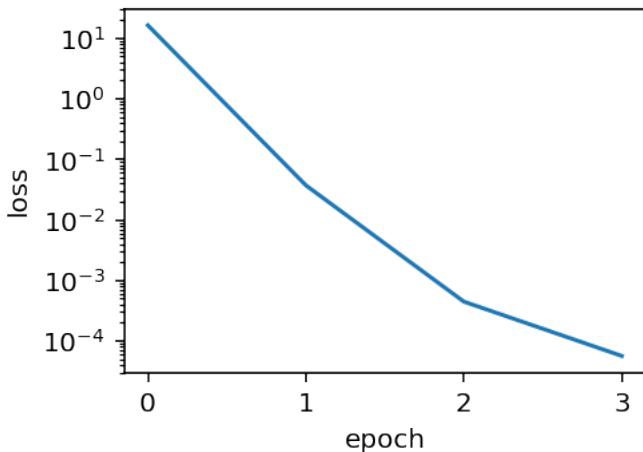


当批量大小为 1000 时，由于数据样本总数也是 1000，优化使用的是梯度下降。梯度下降无需自我衰减学习率 (decay_epoch=None)。最终，优化所得的模型参数值与它们的真实值较接近。

需要注意的是，梯度下降的 1 个迭代周期对模型参数只迭代 1 次。而随机梯度下降的批量大小为 1，它在 1 个迭代周期对模型参数迭代了 1000 次。我们观察到，1 个迭代周期后，梯度下降所得的损失函数值比随机梯度下降所得的损失函数值略大。而在 3 个迭代周期后，这两个算法所得的损失函数值很接近。

```
In [7]: optimize(batch_size=1000, lr=0.999, num_epochs=3, decay_epoch=None,
log_interval=1000)

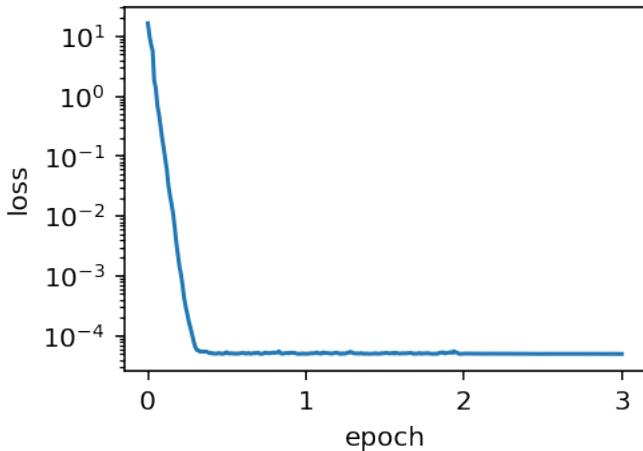
w:
[[ 2.00053716]
[-3.40285206]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.19865417]
<NDArray 1 @cpu(0)>
```



当批量大小为 10 时，由于数据样本总数也是 1000，优化使用的是小批量随机梯度下降。最终，优化所得的模型参数值与它们的真实值较接近。

```
In [8]: optimize(batch_size=10, lr=0.2, num_epochs=3, decay_epoch=2, log_interval=10)

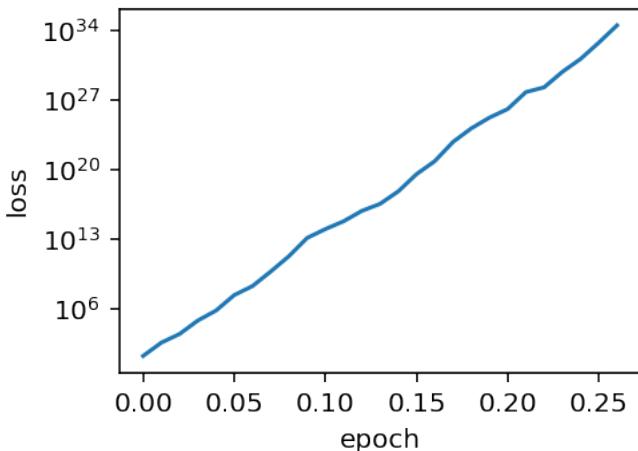
w:
[[ 1.99993336]
[-3.39992714]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.20011902]
<NDArray 1 @cpu(0)>
```



同样是批量大小为 10，我们把学习率改大。这时损失函数值不断增大，直到出现“nan”（not a number，非数）。这是因为，过大的学习率造成了模型参数越过最优解并发散。最终学到的模型参数也是“nan”。

```
In [9]: optimize(batch_size=10, lr=5, num_epochs=3, decay_epoch=2, log_interval=10)
```

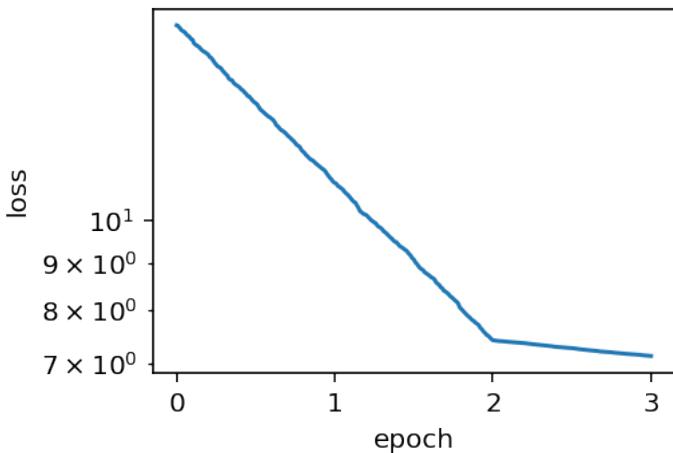
```
w:  
[[ nan]  
 [ nan]]  
<NDArray 2x1 @cpu(0)>  
b:  
[ nan]  
<NDArray 1 @cpu(0)>
```



同样是批量大小为 10，我们把学习率改小。这时我们观察到损失函数值下降较慢，直到 3 个迭代周期模型参数也没能接近它们的真实值。

```
In [10]: optimize(batch_size=10, lr=0.002, num_epochs=3, decay_epoch=2,
log_interval=10)
```

```
w:  
[[ 0.69044906]  
 [-1.20033693]]  
<NDArray 2x1 @cpu(0)>  
b:  
[ 1.37683606]  
<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, J. (2010). Calculus: Early Transcendentals (7th Edition). Brooks Cole.
- [2] Gürbüzbalaban, M., Ozdaglar, A., & Parrilo, P. (2018). Why random reshuffling beats stochastic gradient descent. arXiv preprint arXiv:1510.08560.

7.3 梯度下降和随机梯度下降的 Gluon 实现

在 Gluon 里，使用小批量随机梯度下降很方便，我们无需重新实现该算法。特别地，当批量大小等于数据集样本数时，该算法即为梯度下降；批量大小为 1 即为随机梯度下降。

首先，导入本节中实验所需的包或模块。

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

下面生成实验数据集并定义线性回归模型。

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

        # 线性回归模型。
        net = nn.Sequential()
        net.add(nn.Dense(1))
```

为了使学习率能够自我衰减，我们需要访问 `gluon.Trainer` 的 `learning_rate` 属性并使用 `set_learning_rate` 函数。

```
In [3]: # 优化目标函数。
        def optimize(batch_size, trainer, num_epochs, decay_epoch, log_interval,
                    features, labels, net):
```

```

dataset = gdata.ArrayDataset(features, labels)
data_iter = gdata.DataLoader(dataset, batch_size, shuffle=True)
loss = gloss.L2Loss()
ls = [loss(net(features), labels).mean().asnumpy()]
for epoch in range(1, num_epochs + 1):
    # 学习率自我衰减。
    if decay_epoch and epoch > decay_epoch:
        trainer.set_learning_rate(trainer.learning_rate * 0.1)
    for batch_i, (X, y) in enumerate(data_iter):
        with autograd.record():
            l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        if batch_i * batch_size % log_interval == 0:
            ls.append(loss(net(features), labels).mean().asnumpy())
    # 为了便于打印，改变输出形状并转化成 numpy 数组。
    print('w:', net[0].weight.data(), '\nb:', net[0].bias.data(), '\n')
es = np.linspace(0, num_epochs, len(ls), endpoint=True)
gb.semilogy(es, ls, 'epoch', 'loss')

```

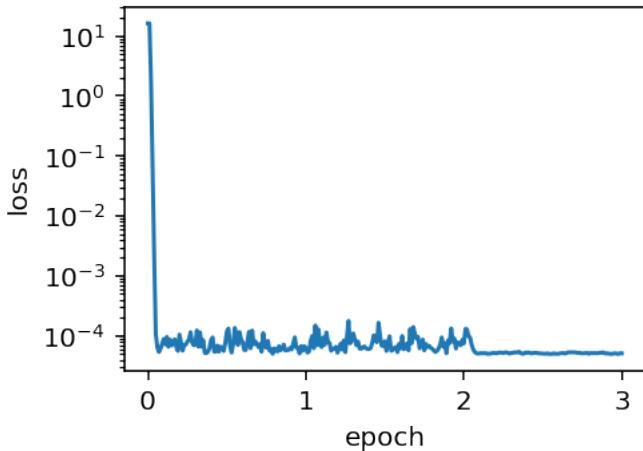
以下几组实验分别重现了”梯度下降和随机梯度下降”一节中实验结果。

```

In [4]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.2})
        optimize(batch_size=1, trainer=trainer, num_epochs=3, decay_epoch=2,
                 log_interval=10, features=features, labels=labels, net=net)

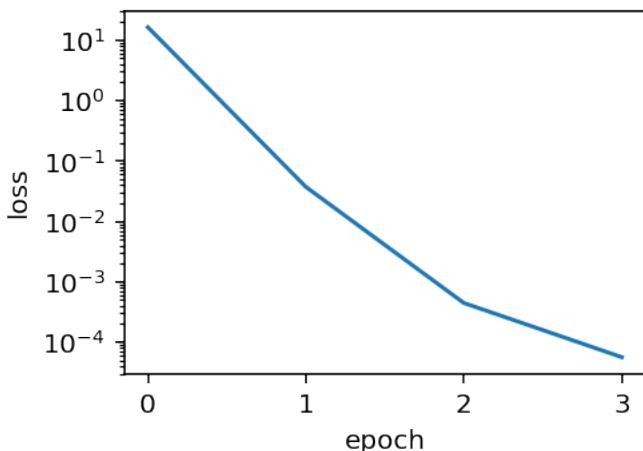
w:
[[ 2.00068688 -3.39845061]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.19977474]
<NDArray 1 @cpu(0)>

```



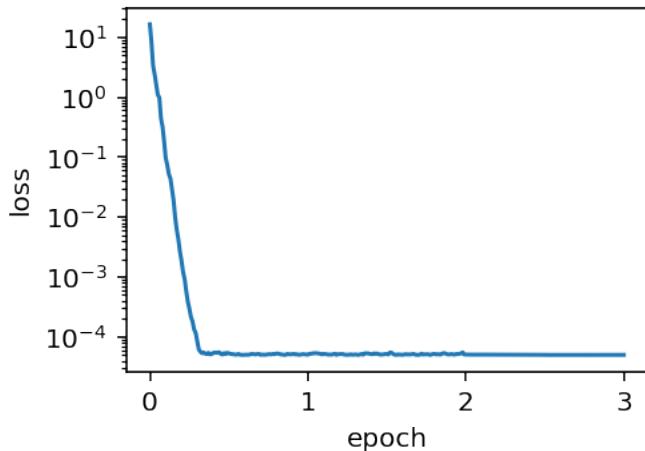
```
In [5]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.999})
        optimize(batch_size=1000, trainer=trainer, num_epochs=3, decay_epoch=None,
                 log_interval=1000, features=features, labels=labels, net=net)

w:
[[ 2.00053716 -3.40285206]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.19865417]
<NDArray 1 @cpu(0)>
```



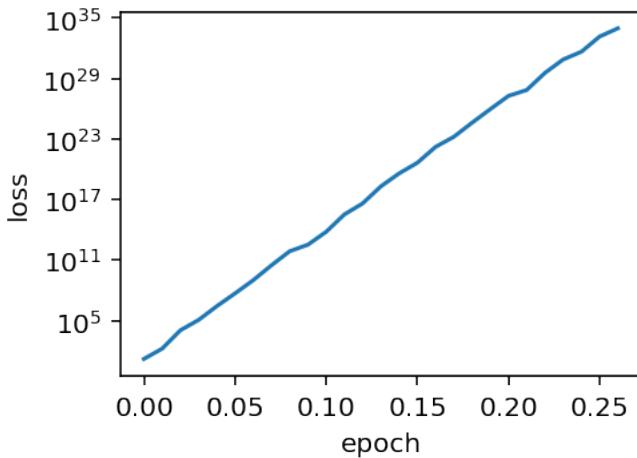
```
In [6]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.2})
    optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
            log_interval=10, features=features, labels=labels, net=net)

w:
[[ 2.00002742 -3.39944434]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20007086]
<NDArray 1 @cpu(0)>
```



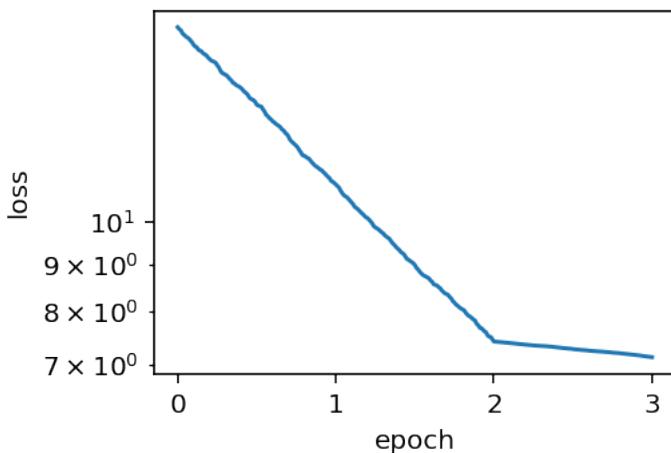
```
In [7]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 5})
    optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
            log_interval=10, features=features, labels=labels, net=net)

w:
[[ nan nan]]
<NDArray 1x2 @cpu(0)>
b:
[ nan]
<NDArray 1 @cpu(0)>
```



```
In [8]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.002})
        optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
                 log_interval=10, features=features, labels=labels, net=net)

w:
[[ 0.69057018 -1.20008659]]
<NDArray 1x2 @cpu(0)>
b:
[ 1.37703586]
<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

我们已经介绍了梯度下降。每次迭代时，该算法根据自变量当前所在位置，沿着目标函数下降最快的方向更新自变量。因此，梯度下降有时也叫做最陡下降（steepest descent）。目标函数有关自变量的梯度代表了目标函数下降最快的方向。

7.4.1 梯度下降的问题

给定目标函数，在梯度下降中，自变量的迭代方向仅仅取决于自变量当前位置。这可能会带来一些问题。

考虑一个输入和输出分别为二维向量 $\mathbf{x} = [x_1, x_2]^\top$ 和标量的目标函数 $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ 。图 7.2 展示了该函数的等高线示意图。

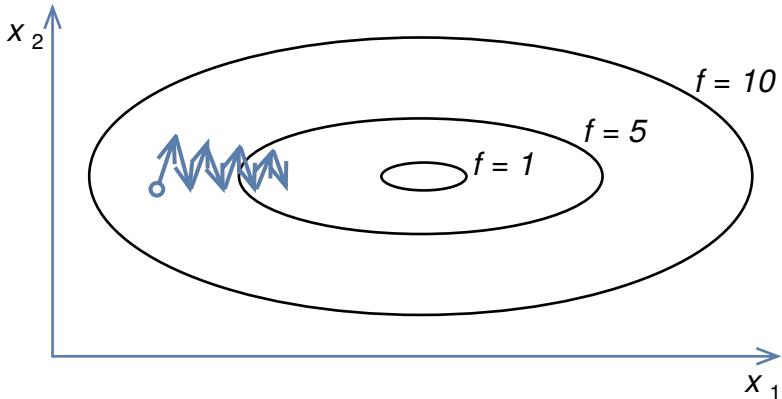


图 7.2: 目标函数 f 的等高线图和自变量 $[x_1, x_2]$ 在梯度下降中的迭代。每条等高线（椭圆实线）代表所有函数值相同的自变量的坐标。实心圆代表自变量初始坐标。每个箭头头部代表自变量在每次迭代后的坐标。

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 7.2 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:

$$\begin{aligned} \mathbf{v} &\leftarrow \gamma \mathbf{v} + \eta \nabla f_{\mathcal{B}}(\mathbf{x}), \\ \mathbf{x} &\leftarrow \mathbf{x} - \mathbf{v}. \end{aligned}$$

其中 \mathbf{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 B 的随机梯度 $\nabla f_{\mathcal{B}}(\mathbf{x})$ 已在“梯度下降和随机梯度下降”一节中描述。

指数加权移动平均

为了更清晰地理解动量法，让我们先解释指数加权移动平均（exponentially weighted moving average）。给定超参数 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：

$$\begin{aligned} y^{(t)} &= (1 - \gamma)x^{(t)} + \gamma y^{(t-1)} \\ &= (1 - \gamma)x^{(t)} + (1 - \gamma) \cdot \gamma x^{(t-1)} + \gamma^2 y^{(t-2)} \\ &= (1 - \gamma)x^{(t)} + (1 - \gamma) \cdot \gamma x^{(t-1)} + (1 - \gamma) \cdot \gamma^2 x^{(t-2)} + \gamma^3 y^{(t-3)} \\ &\dots \end{aligned}$$

由于

$$\lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right)^n = \exp(-1) \approx 0.3679,$$

我们可以将 $\gamma^{1/(1-\gamma)}$ 近似为 $\exp(-1)$ 。例如 $0.95^{20} \approx \exp(-1)$ 。如果把 $\exp(-1)$ 当做一个比较小的数，我们可以在近似中忽略所有含 $\gamma^{1/(1-\gamma)}$ 和比 $\gamma^{1/(1-\gamma)}$ 更高阶的系数的项。例如，当 $\gamma = 0.95$ 时，

$$y^{(t)} \approx 0.05 \sum_{i=0}^{19} 0.95^i x^{(t-i)}.$$

因此，在实际中，我们常常将 y 看作是对最近 $1/(1 - \gamma)$ 个时刻的 x 值的加权平均。例如，当 $\gamma = 0.95$ 时， y 可以被看作是对最近 20 个时刻的 x 值的加权平均；当 $\gamma = 0.9$ 时， y 可以看作是对最近 10 个时刻的 x 值的加权平均：离当前时刻越近的 x 值获得的权重越大。

由指数加权移动平均理解动量法

现在，我们对动量法的速度变量做变形：

$$\mathbf{v} \leftarrow \gamma \mathbf{v} + (1 - \gamma) \frac{\eta \nabla f_{\mathcal{B}}(\mathbf{x})}{1 - \gamma}.$$

由指数加权移动平均的形式可得，速度变量 \mathbf{v} 实际上对 $(\eta \nabla f_{\mathcal{B}}(\mathbf{x})) / (1 - \gamma)$ 做了指数加权移动平均。给定动量超参数 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 $\nabla f_{\mathcal{B}}(\mathbf{x}) / (1 - \gamma)$ 的加权平均。

给定目标函数，在动量法的每次迭代中，自变量在各个方向上的移动幅度不仅取决于当前梯度，还取决于过去各个梯度在各个方向上是否一致。图 7.3 展示了使用动量法的梯度下降迭代图 7.2 中目标函数自变量的情景。我们将每个梯度代表的箭头方向在水平方向和竖直方向做分解。由于所有梯度的水平方向为正（向右）、在竖直上时正（向上）时负（向下），自变量在水平方向移动幅度逐渐增大，而在竖直方向移动幅度逐渐减小。这样，我们就可以使用较大的学习率，从而使自变量向最优解更快移动。

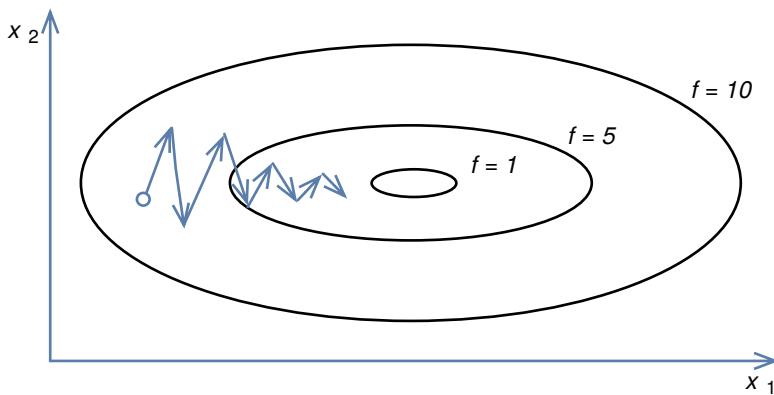


图 7.3: 目标函数 f 的等高线图和自变量 $[x_1, x_2]$ 在使用动量法的梯度下降中的迭代。每条等高线（椭圆实线）代表所有函数值相同的自变量的坐标。实心圆代表自变量初始坐标。每个箭头头部代表自变量在每次迭代后的坐标。

7.4.3 动量法的实现

动量法的实现也很简单。我们在小批量随机梯度下降的基础上添加速度变量。

```
In [1]: def sgd_momentum(params, vs, lr, mom, batch_size):
    for param, v in zip(params, vs):
        v[:] = mom * v + lr * param.grad / batch_size
        param[:] -= v
```

7.4.4 实验

首先，导入本节中实验所需的包或模块。

```
In [2]: import sys
sys.path.append('..')
```

```

import gluonbook as gb
from mxnet import autograd, nd
import numpy as np

```

实验中，我们以之前介绍过的线性回归为例。设数据集的样本数为 1000，我们使用权重 w 为 [2, -3.4]，偏差 “ b ” 为 4.2 的线性回归模型来生成数据集。该模型的平方损失函数即所需优化的目标函数，模型参数即目标函数自变量。

In [3]: # 生成数据集。

```

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

# 初始化模型参数。
def init_params():
    w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    vs = []
    for param in params:
        param.attach_grad()
        # 把速度项初始化为和参数形状相同的零张量。
        vs.append(param.zeros_like())
    return params, vs

```

优化函数 `optimize` 与 “梯度下降和随机梯度下降” 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, features, labels)):
            with autograd.record():

```

```

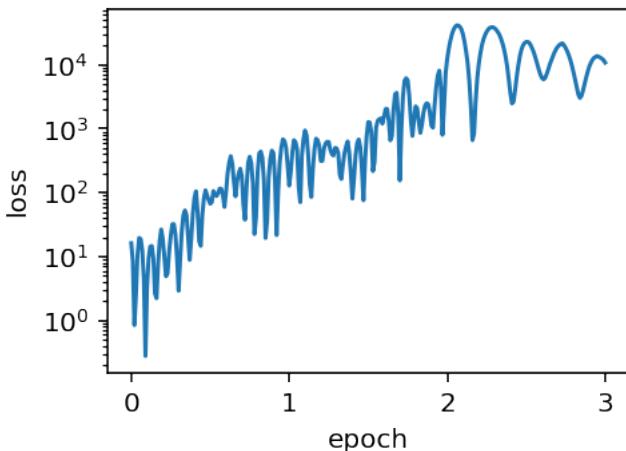
        l = loss(net(X, w, b), y)
        l.backward()
        sgd_momentum([w, b], vs, lr, mom, batch_size)
        if batch_i * batch_size % log_interval == 0:
            ls.append(loss(net(features, w, b), labels).mean().asnumpy())
    print('w:', w, '\nb:', b, '\n')
    es = np.linspace(0, num_epochs, len(ls), 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 $100\nabla f_B(x)$ 的加权平均。我们观察到，损失函数值在 3 个迭代周期后上升。这很可能是由于特殊梯度中较大的系数 100 造成的。

In [5]: `optimize(batch_size=10, lr=0.2, mom=0.99, num_epochs=3, log_interval=10)`

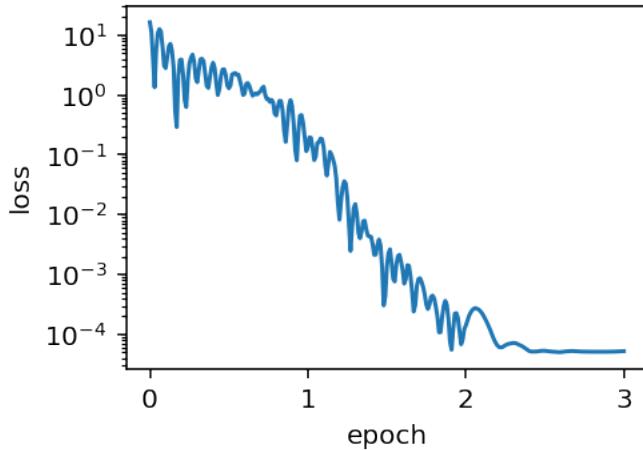
```
w:
[[ -134.37818909]
 [ -55.32491302]]
<NDArray 2x1 @cpu(0)>
b:
[ 14.03986454]
<NDArray 1 @cpu(0)>
```



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 $10\nabla f_B(x)$ 的加权平均。我们观察到，损失函数值在 3 个迭代周期后下降。

In [6]: `optimize(batch_size=10, lr=0.2, mom=0.9, num_epochs=3, log_interval=10)`

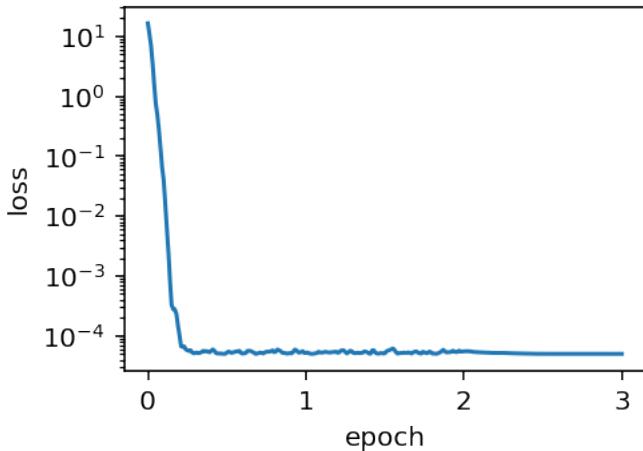
```
w:  
[[ 1.99841654]  
[-3.39922833]]  
<NDArray 2x1 @cpu(0)>  
b:  
[ 4.19887066]  
<NDArray 1 @cpu(0)>
```



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 $2\nabla f_B(x)$ 的加权平均。我们观察到，损失函数值在 3 个迭代周期后下降，且下降曲线较平滑。最终，优化所得的模型参数值与它们的真实值较接近。

```
In [7]: optimize(batch_size=10, lr=0.2, mom=0.5, num_epochs=3, log_interval=10)
```

```
w:  
[[ 1.99961436]  
[-3.39968014]]  
<NDArray 2x1 @cpu(0)>  
b:  
[ 4.20057964]  
<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 动量法的 Gluon 实现

在 Gluon 里，使用动量法很方便，我们无需重新实现该算法。

首先，导入本节中实验所需的包或模块。

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

下面生成实验数据集并定义线性回归模型。

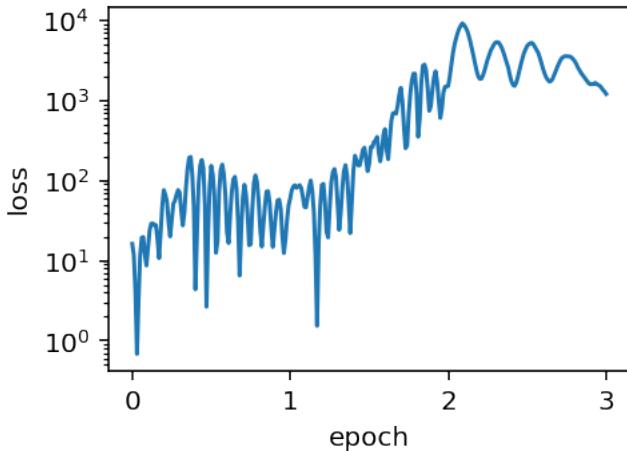
```
In [2]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01,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

        # 线性回归模型。
        net = nn.Sequential()
        net.add(nn.Dense(1))
```

例如，以使用动量法的小批量随机梯度下降为例，我们可以在 `Trainer` 中定义动量超参数 `momentum`。以下几组实验分别重现了“动量法”一节中实验结果。

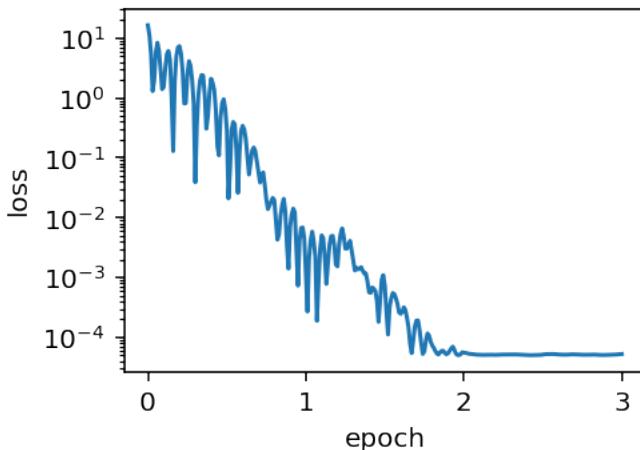
```
In [3]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'sgd',
                               {'learning_rate': 0.2, 'momentum': 0.99})
        gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
                    log_interval=10, features=features, labels=labels, net=net)

w:
[[ -45.10115814    6.71086788]]
<NDArray 1x2 @cpu(0)>
b:
[-3.56248951]
<NDArray 1 @cpu(0)>
```



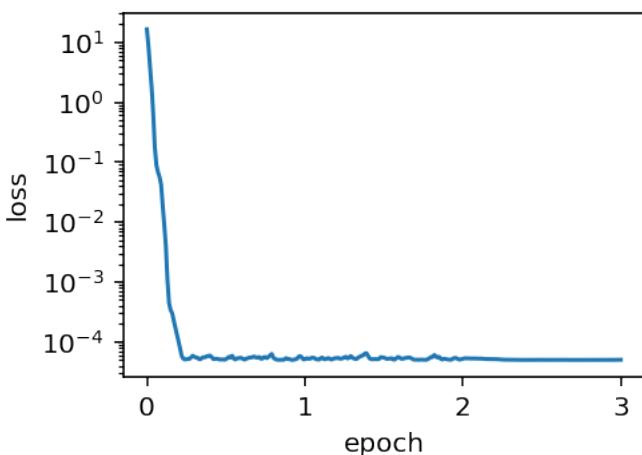
```
In [4]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                           {'learning_rate': 0.2, 'momentum': 0.9})
    gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
                log_interval=10, features=features, labels=labels, net=net)

w:
[[ 1.99875093 -3.40061617]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.19828796]
<NDArray 1 @cpu(0)>
```



```
In [5]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                           {'learning_rate': 0.2, 'momentum': 0.5})
    gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=2,
                log_interval=10, features=features, labels=labels, net=net)

w:
[[ 2.000319 -3.40058613]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.19994593]
<NDArray 1 @cpu(0)>
```



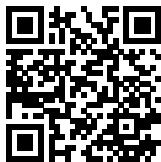
7.5.1 小结

- 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

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：

$$\begin{aligned}x_1 &\leftarrow x_1 - \eta \frac{\partial f}{\partial x_1}, \\x_2 &\leftarrow x_2 - \eta \frac{\partial f}{\partial x_2}.\end{aligned}$$

如果让 x_1 和 x_2 使用不同的学习率自我迭代呢？实际上，Adagrad 就是一个在迭代过程中不断自我调整学习率，并让模型参数中每个元素都使用不同学习率的优化算法 [1]。

下面，我们将介绍 Adagrad 算法。关于本节中涉及到的按元素运算，例如标量与向量计算以及按元素相乘 \odot ，请参见“数学基础”一节。

7.6.1 Adagrad 算法

Adagrad 的算法会使用一个小批量随机梯度按元素平方的累加变量 s ，并将其中每个元素初始化为 0。在每次迭代中，首先计算小批量随机梯度 g ，然后将该梯度按元素平方后累加到变量 s ：

$$s \leftarrow s + g \odot g.$$

然后，我们将目标函数自变量中每个元素的学习率通过按元素运算重新调整一下：

$$g' \leftarrow \frac{\eta}{\sqrt{s + \epsilon}} \odot g,$$

其中 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:

$$x \leftarrow x - g'.$$

7.6.2 Adagrad 的特点

需要强调的是, 小批量随机梯度按元素平方的累加变量 s 出现在含调整后学习率的梯度 g' 的分母项。因此, 如果目标函数有关自变量中某个元素的偏导数一直都较大, 那么就让该元素的学习率下降快一点; 反之, 如果目标函数有关自变量中某个元素的偏导数一直都较小, 那么就让该元素的学习率下降慢一点。然而, 由于 s 一直在累加按元素平方的梯度, 自变量中每个元素的学习率在迭代过程中一直在降低 (或不变)。所以, 当学习率在迭代早期降得较快且当前解依然不佳时, Adagrad 在迭代后期由于学习率过小, 可能较难找到一个有用的解。

7.6.3 Adagrad 的实现

Adagrad 的实现很简单。我们只需要把上面的数学公式翻译成代码。

```
In [1]: def adagrad(params, sqrs, lr, batch_size):
    eps_stable = 1e-7
    for param, sqr in zip(params, sqrs):
        g = param.grad / batch_size
        sqr[:] += g.square()
        param[:] -= lr * g / (sqr + eps_stable).sqrt()
```

7.6.4 实验

首先, 导入本节中实验所需的包或模块。

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

实验中，我们以之前介绍过的线性回归为例。设数据集的样本数为 1000，我们使用权重 w 为 [2, -3.4]，偏差 “ b ” 为 4.2 的线性回归模型来生成数据集。该模型的平方损失函数即所需优化的目标函数，模型参数即目标函数自变量。

我们把梯度按元素平方的累加变量初始化为和模型参数形状相同的零张量。

```
In [3]: # 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)

# 初始化模型参数。
def init_params():
    w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    sqrs = []
    for param in params:
        param.attach_grad()
        # 把梯度按元素平方的累加变量初始化为和参数形状相同的零张量。
        sqrs.append(param.zeros_like())
    return params, sqrs
```

优化函数 `optimize` 与 “梯度下降和随机梯度下降” 一节中的类似。需要指出的是，这里的初始学习率 lr 无需自我衰减。

```
In [4]: net = gb.linreg
loss = gb.squared_loss

def optimize(batch_size, lr, num_epochs, log_interval):
    [w, b], sqrs = init_params()
    ls = [loss(net(features, w, b), labels).mean().asnumpy()]
    for epoch in range(1, num_epochs + 1):
        for batch_i, (X, y) in enumerate(
            gb.data_iter(batch_size, 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')

```

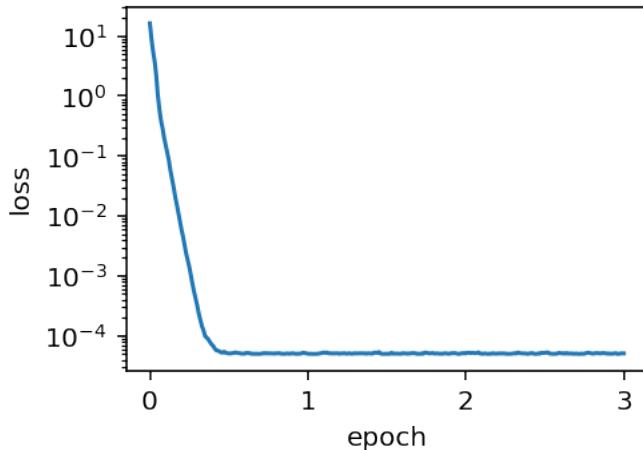
最终，优化所得的模型参数值与它们的真实值较接近。

```

In [5]: optimize(batch_size=10, lr=0.9, num_epochs=3, log_interval=10)

w:
[[ 2.00089025]
 [-3.40073442]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.20099306]
<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, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul), 2121-2159.

7.7 Adagrad 的 Gluon 实现

在 Gluon 里，使用 Adagrad 很方便，我们无需重新实现该算法。

首先，导入本节中实验所需的包或模块。

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

下面生成实验数据集并定义线性回归模型。

```
In [2]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
```

```

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

# 线性回归模型。
net = nn.Sequential()
net.add(nn.Dense(1))

```

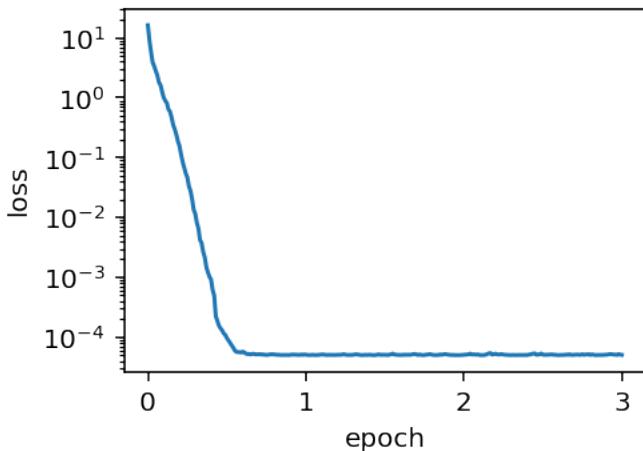
我们可以在 Trainer 中定义优化算法名称 `adagrad`。以下实验分别重现了“Adagrad”一节中实验结果。

```

In [3]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'adagrad',
                               {'learning_rate': 0.9})
        gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=None,
                    log_interval=10, features=features, labels=labels, net=net)

w:
[[ 1.99911284 -3.4007175 ]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20035458]
<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

我们在“Adagrad”一节里提到，由于调整学习率时分母上的变量 s 一直在累加按元素平方的小批量随机梯度，目标函数自变量每个元素的学习率在迭代过程中一直在降低（或不变）。所以，当学习率在迭代早期降得较快且当前解依然不佳时，Adagrad 在迭代后期由于学习率过小，可能较难找到一个有用的解。为了应对这一问题，RMSProp 算法对 Adagrad 做了一点小小的修改 [1]。

下面，我们来描述 RMSProp 算法。

7.8.1 RMSProp 算法

我们在“动量法”一节里介绍过指数加权移动平均。事实上，RMSProp 算法使用了小批量随机梯度按元素平方的指数加权移动平均变量 s ，并将其中每个元素初始化为 0。给定超参数 γ 且 $0 \leq \gamma < 1$ ，在每次迭代中，RMSProp 首先计算小批量随机梯度 g ，然后对该梯度按元素平方项 $g \odot g$ 做指数加权移动平均，记为 s ：

$$s \leftarrow \gamma s + (1 - \gamma)g \odot g.$$

然后，和 Adagrad 一样，将目标函数自变量中每个元素的学习率通过按元素运算重新调整一下：

$$\mathbf{g}' \leftarrow \frac{\eta}{\sqrt{s + \epsilon}} \odot \mathbf{g},$$

其中 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：

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{g}'.$$

需要强调的是，RMSProp 只在 Adagrad 的基础上修改了变量 s 的更新方法：对平方项 $\mathbf{g} \odot \mathbf{g}$ 从累加变成了指数加权移动平均。由于变量 s 可看作是最近 $1/(1 - \gamma)$ 个时刻的平方项 $\mathbf{g} \odot \mathbf{g}$ 的加权平均，自变量每个元素的学习率在迭代过程中避免了“直降不升”的问题。

7.8.2 RMSProp 的实现

RMSProp 的实现很简单。我们只需要把上面的数学公式翻译成代码。

```
In [1]: def rmsprop(params, sqrs, lr, gamma, batch_size):
    eps_stable = 1e-8
    for param, sqr in zip(params, sqrs):
        g = param.grad / batch_size
        sqr[:] = gamma * sqr + (1 - gamma) * g.square()
        param[:] -= lr * g / (sqr + eps_stable).sqrt()
```

7.8.3 实验

首先，导入本节中实验所需的包或模块。

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

实验中，我们依然以线性回归为例。设数据集的样本数为 1000，我们使用权重 w 为 $[2, -3.4]$ ，偏差 “ b ” 为 4.2 的线性回归模型来生成数据集。该模型的平方损失函数即所需优化的目标函数，模型参数即目标函数自变量。

我们把小批量随机梯度按元素平方的指数加权移动平均变量 s 初始化为和模型参数形状相同的零张量。

```
In [3]: # 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)

# 初始化模型参数。
def init_params():
    w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    sqrs = []
    for param in params:
        param.attach_grad()
    # 把梯度按元素平方的指数加权移动平均变量初始化为和参数形状相同的零张量。
    sqrs.append(param.zeros_like())
    return params, sqrs
```

优化函数 `optimize` 与 “Adagrad” 一节中的类似。

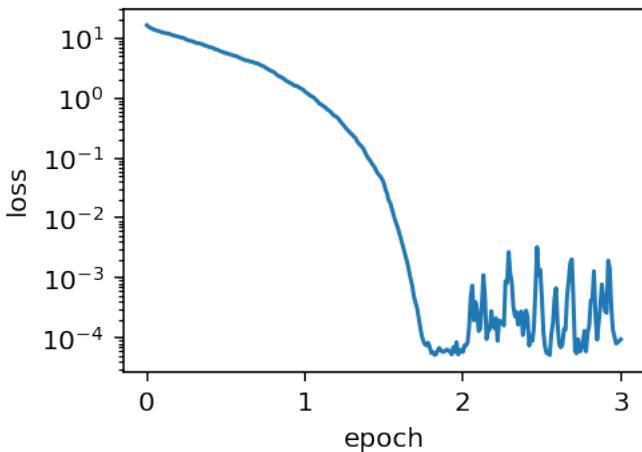
```
In [4]: net = gb.linreg
loss = gb.squared_loss

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, 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 s 可看作是最近 $1/(1-0.9) = 10$ 个时刻的平方项 $\mathbf{g} \odot \mathbf{g}$ 的加权平均。我们观察到，损失函数在迭代后期较震荡。

```
In [5]: optimize(batch_size=10, lr=0.03, gamma=0.9, num_epochs=3, log_interval=10)

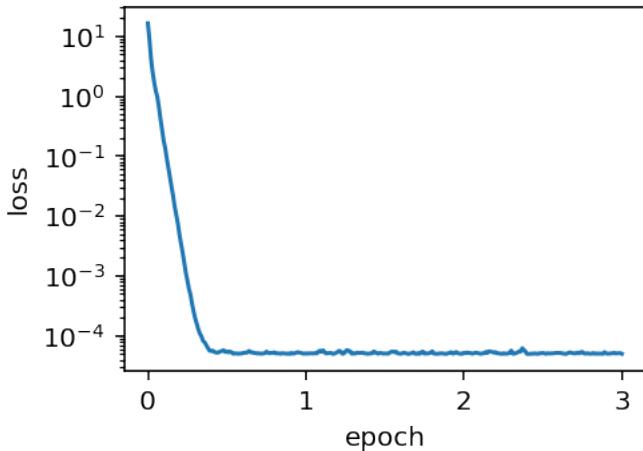
w:
[[ 2.00220966]
 [-3.39199543]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.20369101]
<NDArray 1 @cpu(0)>
```



我们将 γ 调大一点，例如 0.999。此时，变量 s 可看作是最近 $1/(1 - 0.999) = 1000$ 个时刻的平方项 $\mathbf{g} \odot \mathbf{g}$ 的加权平均。这时损失函数在迭代后期较平滑。

```
In [6]: optimize(batch_size=10, lr=0.03, gamma=0.999, num_epochs=3, log_interval=10)

w:
[[ 1.99940455]
 [-3.40055156]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.19993114]
<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, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning, 4(2), 26-31.

7.9 RMSProp 的 Gluon 实现

在 Gluon 里，使用 RMSProp 很方便，我们无需重新实现该算法。

首先，导入本节中实验所需的包或模块。

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

下面生成实验数据集并定义线性回归模型。

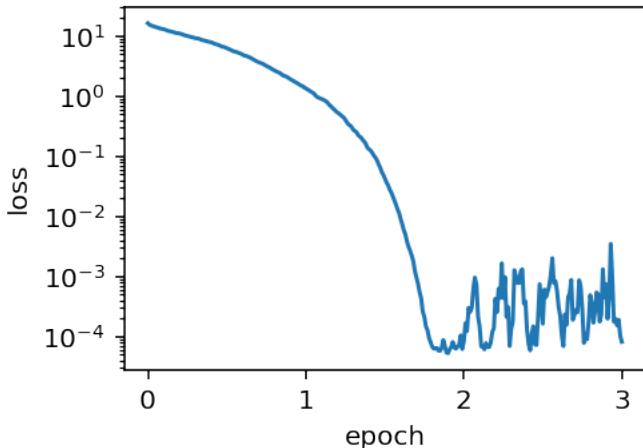
```
In [2]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01,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

        # 线性回归模型。
        net = nn.Sequential()
        net.add(nn.Dense(1))
```

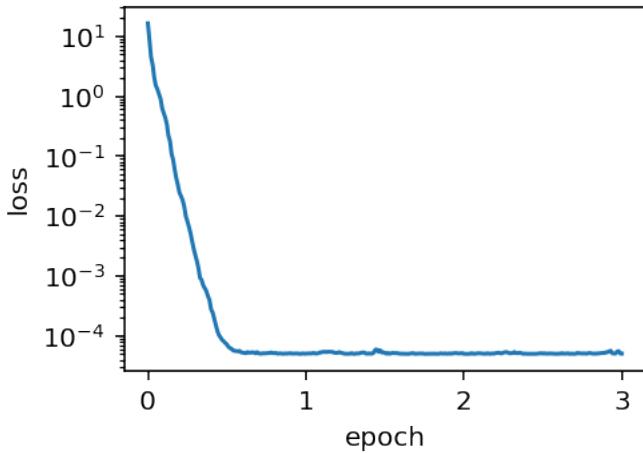
我们可以在 Trainer 中定义优化算法名称 `rmsprop` 并定义 γ 超参数 `gamma1`。以下几组实验分别重现了“RMSProp”一节中实验结果。

```
In [3]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'rmsprop',
                               {'learning_rate': 0.03, 'gamma1': 0.9})
        gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=None,
                    log_interval=10, features=features, labels=labels, net=net)
```

```
w:  
[[ 2.00147915 -3.40702033]]  
<NDArray 1x2 @cpu(0)>  
b:  
[ 4.20289087]  
<NDArray 1 @cpu(0)>
```



```
In [4]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)  
      trainer = gluon.Trainer(net.collect_params(), 'rmsprop',  
                             {'learning_rate': 0.03, 'gamma1': 0.999})  
      gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=None,  
                  log_interval=10, features=features, labels=labels, net=net)  
  
w:  
[[ 2.00090528 -3.39957213]]  
<NDArray 1x2 @cpu(0)>  
b:  
[ 4.19992065]  
<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

我们在“RMSProp”一节中描述了，RMSProp 针对 Adagrad 在迭代后期可能较难找到有用解的问题，对小批量随机梯度按元素平方项做指数加权移动平均而不是累加。另一种应对该问题的优

化算法叫做 Adadelta [1]。有意思的是，它没有学习率超参数。

7.10.1 Adadelta 算法

Adadelta 算法也像 RMSProp 一样，使用了小批量随机梯度按元素平方的指数加权移动平均变量 s ，并将其中每个元素初始化为 0。给定超参数 ρ 且 $0 \leq \rho < 1$ ，在每次迭代中，RMSProp 首先计算小批量随机梯度 \mathbf{g} ，然后对该梯度按元素平方项 $\mathbf{g} \odot \mathbf{g}$ 做指数加权移动平均，记为 s ：

$$\mathbf{s} \leftarrow \rho \mathbf{s} + (1 - \rho) \mathbf{g} \odot \mathbf{g}.$$

然后，计算当前需要迭代的目标函数自变量的变化量 \mathbf{g}' ：

$$\mathbf{g}' \leftarrow \frac{\sqrt{\Delta \mathbf{x} + \epsilon}}{\sqrt{\mathbf{s} + \epsilon}} \odot \mathbf{g},$$

其中 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 $\Delta \mathbf{x}$ 初始化为零张量，并记录 \mathbf{g}' 按元素平方的指数加权移动平均：

$$\Delta \mathbf{x} \leftarrow \rho \Delta \mathbf{x} + (1 - \rho) \mathbf{g}' \odot \mathbf{g}'.$$

同样地，最后的自变量迭代步骤与小批量随机梯度下降类似：

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{g}'.$$

7.10.2 Adadelta 的实现

Adadelta 的实现很简单。我们只需要把上面的数学公式翻译成代码。

```
In [1]: def adadelta(params, sqrs, deltas, rho, batch_size):
    eps_stable = 1e-5
    for param, sqr, delta in zip(params, sqrs, deltas):
        g = param.grad / batch_size
        sqr[:] = rho * sqr + (1 - rho) * g.square()
        cur_delta = ((delta + eps_stable).sqrt() /
                     (sqr + eps_stable).sqrt() * g)
        delta[:] = rho * delta + (1 - rho) * cur_delta * cur_delta
        param[:] -= cur_delta
```

7.10.3 实验

首先，导入本节中实验所需的包或模块。

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

实验中，我们依然以线性回归为例。设数据集的样本数为 1000，我们使用权重 w 为 $[2, -3.4]$ ，偏差 “ b ” 为 4.2 的线性回归模型来生成数据集。该模型的平方损失函数即所需优化的目标函数，模型参数即目标函数自变量。

我们把算法中变量 s 和 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=0.01, shape=(num_inputs, 1))
            b = nd.zeros(shape=(1,))
            params = [w, b]
            sqrs = []
            deltas = []
            for param in params:
                param.attach_grad()
            # 把算法中基于指数加权移动平均的变量初始化为和参数形状相同的零张量。
            sqrs.append(param.zeros_like())
            deltas.append(param.zeros_like())
            return params, sqrs, deltas
```

优化函数 `optimize` 与 “Adagrad” 一节中的类似。

```
In [4]: net = gb.linreg
        loss = gb.squared_loss

        def optimize(batch_size, rho, num_epochs, log_interval):
```

```

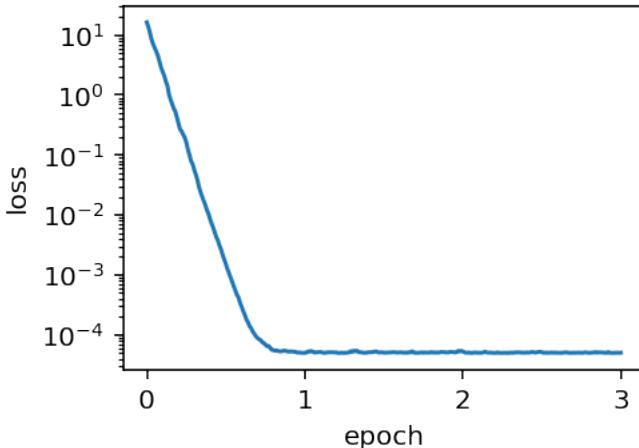
[w, b], sqrs, deltas = init_params()
ls = [loss(net(features, w, b), labels).mean().asnumpy()]
for epoch in range(1, num_epochs + 1):
    for batch_i, (X, y) in enumerate(
        gb.data_iter(batch_size, features, labels)):
        with autograd.record():
            l = loss(net(X, w, b), y)
            l.backward()
            adadelta([w, b], sqrs, deltas, rho, batch_size)
        if batch_i * batch_size % log_interval == 0:
            ls.append(loss(net(features, w, b), labels).mean().asnumpy())
print('w:', w, '\nb:', b, '\n')
es = np.linspace(0, num_epochs, len(ls), endpoint=True)
gb.semilogy(es, ls, 'epoch', 'loss')

```

最终，优化所得的模型参数值与它们的真实值较接近。

```
In [5]: optimize(batch_size=10, rho=0.9999, num_epochs=3, log_interval=10)

w:
[[ 1.99942338
  [-3.3997097 ]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.19942522]
<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, M. D. (2012). ADADELTA: an adaptive learning rate method. arXiv preprint arXiv:1212.5701.

7.11 Adadelta 的 Gluon 实现

在 Gluon 里，使用 Adadelta 很容易，我们无需重新实现该算法。

首先，导入本节中实验所需的包或模块。

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

下面生成实验数据集并定义线性回归模型。

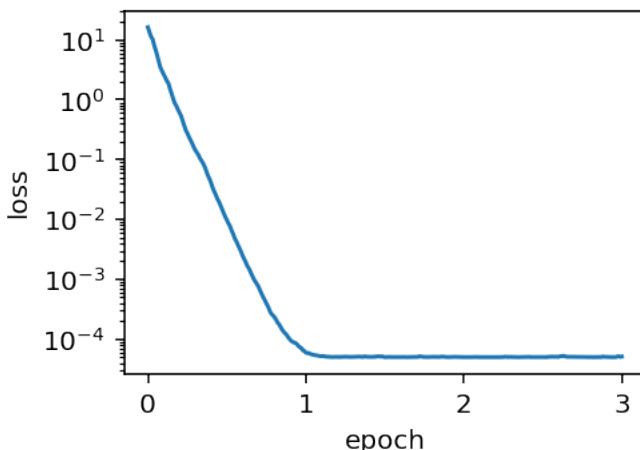
```
In [2]: # 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)

# 线性回归模型。
net = nn.Sequential()
net.add(nn.Dense(1))
```

我们可以在 Trainer 中定义优化算法名称 adadelta 并定义 ρ 超参数 rho。以下实验重现了“Adadelta”一节中实验结果。

```
In [3]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
trainer = gluon.Trainer(net.collect_params(), 'adadelta', {'rho': 0.9999})
gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=None,
            log_interval=10, features=features, labels=labels, net=net)

w:
[[ 1.99967623 -3.39862537]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.20063353]
<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

Adam 是一个组合了动量法和 RMSProp 的优化算法 [1]。下面我们来介绍 Adam 算法。

7.12.1 Adam 算法

Adam 算法使用了动量变量 v 和 RMSProp 中小批量随机梯度按元素平方的指数加权移动平均变量 s ，并将它们中每个元素初始化为 0。在每次迭代中，时刻 t 的小批量随机梯度记作 \mathbf{g}_t 。

和动量法类似，给定超参数 β_1 且满足 $0 \leq \beta_1 < 1$ （算法作者建议设为 0.9），将小批量随机梯度的指数加权移动平均记作动量变量 v ，并将它在时刻 t 的值记作 \mathbf{v}_t ：

$$\mathbf{v}_t \leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1) \mathbf{g}_t.$$

和 RMSProp 中一样，给定超参数 β_2 且满足 $0 \leq \beta_2 < 1$ （算法作者建议设为 0.999），将小批量随机梯度按元素平方后做指数加权移动平均得到 s ，并将它在时刻 t 的值记作 \mathbf{s}_t ：

$$\mathbf{s}_t \leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t \odot \mathbf{g}_t.$$

由于我们将 v 和 s 中的元素都初始化为 0, 在时刻 t 我们得到 $v_t = (1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} g_i$ 。将过去各时刻小批量随机梯度的权值相加, 得到 $(1 - \beta_1) \sum_{i=1}^t \beta_1^{t-i} = 1 - \beta_1^t$ 。需要注意的是, 当 t 较小时, 过去各时刻小批量随机梯度权值之和会较小。例如当 $\beta_1 = 0.9$ 时, $v_1 = 0.1g_1$ 。为了消除这样的影响, 对于任意时刻 t , 我们可以将 v_t 再除以 $1 - \beta_1^t$, 从而使得过去各时刻小批量随机梯度权值之和为 1。这也叫做偏差修正。在 Adam 算法中, 我们对变量 v 和 s 均作偏差修正:

$$\hat{v}_t \leftarrow \frac{v_t}{1 - \beta_1^t},$$

$$\hat{s}_t \leftarrow \frac{s_t}{1 - \beta_2^t}.$$

接下来, Adam 算法使用以上偏差修正后的变量 \hat{v}_t 和 \hat{s}_t , 将模型参数中每个元素的学习率通过按元素运算重新调整:

$$g'_t \leftarrow \frac{\eta \hat{v}_t}{\sqrt{\hat{s}_t + \epsilon}},$$

其中 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, 时刻 t 的自变量 x_t 的迭代步骤与小批量随机梯度下降类似:

$$x_t \leftarrow x_{t-1} - g'_t.$$

7.12.2 Adam 的实现

Adam 的实现很简单。我们只需要把上面的数学公式翻译成代码。

```
In [1]: def adam(params, vs, sqrs, lr, batch_size, t):
    beta1 = 0.9
    beta2 = 0.999
    eps_stable = 1e-8
    for param, v, sqr in zip(params, vs, sqrs):
        g = param.grad / batch_size
        v[:] = beta1 * v + (1 - beta1) * g
        sqr[:] = beta2 * sqr + (1 - beta2) * g.square()
        v_bias_corr = v / (1 - beta1 ** t)
        sqr_bias_corr = sqr / (1 - beta2 ** t)
        param[:] = param - lr * v_bias_corr / (
            sqr_bias_corr.sqrt() + eps_stable)
```

7.12.3 实验

首先，导入实验所需的包或模块。

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

实验中，我们依然以线性回归为例。设数据集的样本数为 1000，我们使用权重 w 为 $[2, -3.4]$ ，偏差 “ b ” 为 4.2 的线性回归模型来生成数据集。该模型的平方损失函数即所需优化的目标函数，模型参数即目标函数自变量。

我们把算法中变量 v 和 s 初始化为和模型参数形状相同的零张量。

```
In [3]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)

        # 初始化模型参数。
        def init_params():
            w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
            b = nd.zeros(shape=(1,))
            params = [w, b]
            vs = []
            sqrs = []
            for param in params:
                param.attach_grad()
            # 把算法中基于指数加权移动平均的变量初始化为和参数形状相同的零张量。
            vs.append(param.zeros_like())
            sqrs.append(param.zeros_like())
            return params, vs, sqrs
```

优化函数 `optimize` 与 “Adagrad” 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, 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')

```

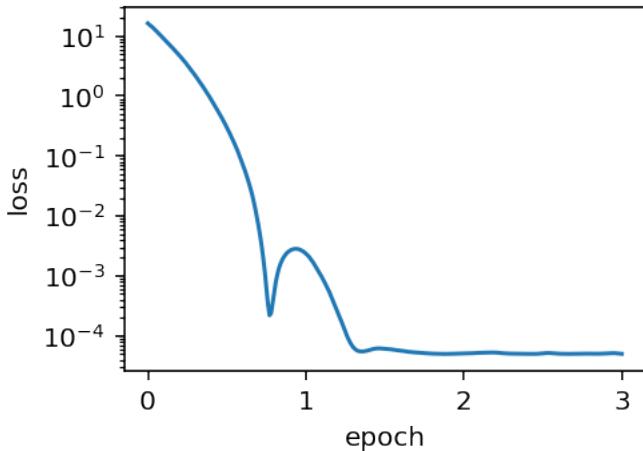
最终，优化所得的模型参数值与它们的真实值较接近。

In [5]: optimize(batch_size=10, lr=0.1, num_epochs=3, log_interval=10)

```

w:
[[ 2.00055242]
 [-3.39997101]]
<NDArray 2x1 @cpu(0)>
b:
[ 4.20006943]
<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, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint

7.13 Adam 的 Gluon 实现

在 Gluon 里，使用 Adadelta 很容易，我们无需重新实现该算法。

首先，导入本节中实验所需的包或模块。

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

下面生成实验数据集并定义线性回归模型。

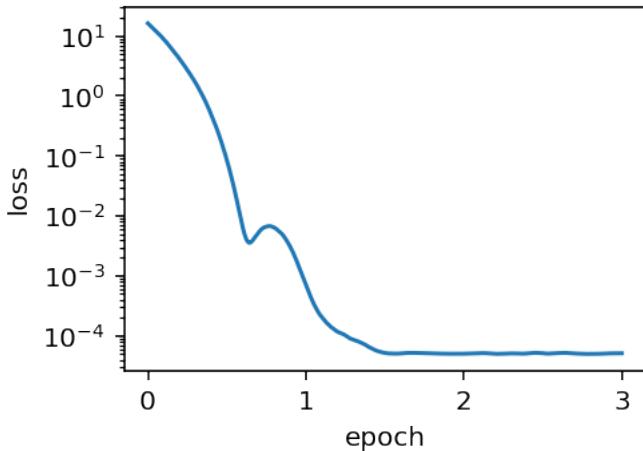
```
In [2]: # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = true_w[0] * features[:, 0] + true_w[1] * features[:, 1] + true_b
        labels += nd.random.normal(scale=0.01,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

        # 线性回归模型。
        net = nn.Sequential()
        net.add(nn.Dense(1))
```

我们可以在 Trainer 中定义优化算法名称 `adam` 并定义初始学习率。以下实验重现了“Adam”一节中实验结果。

```
In [3]: net.initialize(init.Normal(sigma=0.01), force_reinit=True)
        trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': 0.1})
        gb.optimize(batch_size=10, trainer=trainer, num_epochs=3, decay_epoch=None,
                    log_interval=10, features=features, labels=labels, net=net)

w:
[[ 1.99926543 -3.4004035 ]]
<NDArray 1x2 @cpu(0)>
b:
[ 4.19880438]
<NDArray 1 @cpu(0)>
```



7.13.1 小结

- 使用 Gluon 的 `Trainer` 可以方便地使用 Adam。

7.13.2 练习

- 总结本章各个优化算法的异同。
- 回顾前面几章中你感兴趣的模型，将训练部分的优化算法替换成其他算法，观察并分析实验现象。

7.13.3 扫码直达讨论区



7.13.4 本章回顾

梯度下降可沉甸，随机降低方差难。

引入动量别弯慢，Adagrad 梯方贪。

Adadelta 学率换，RMSProp 梯方权。

Adam 动量 RMS 伴，优化还需己调参。

注释：

- 梯方：梯度按元素平方。
- 贪：因贪婪故而不断累加。
- 学率：学习率。
- 换：这个参数被替换掉。
- 权：指数加权移动平均。

计算性能

无论是当数据集很大还是计算资源或应用有约束条件时，深度学习十分关注计算性能。本章将重点介绍影响计算性能的重要因子：命令式编程、符号式编程、异步计算、自动并行计算和多 GPU 计算。通过本章的学习，你将很可能进一步提升已有模型的计算性能，例如在不影响模型精度的前提下减少模型的训练时间。

8.1 命令式和符号式混合编程

其实，到目前为止我们一直都在使用命令式编程：使用编程语句改变程序状态。考虑下面这段简单的命令式编程代码。

```
In [1]: def add(a, b):
        return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
```

```
    return g

fancy_func(1, 2, 3, 4)

Out[1]: 10
```

和我们预期的一样，在运行 `e = add(a, b)` 时，Python 会做加法运算并将结果存储在变量 `e`，从而令程序的状态发生了改变。类似地，后面的两个语句 `f = add(c, d)` 和 `g = add(e, f)` 会依次做加法运算并存储变量。

虽然使用命令式编程很方便，但它的运行可能会慢。一方面，即使 `fancy_func` 函数中的 `add` 是被重复调用的函数，Python 也会逐一执行这三个函数调用语句。另一方面，我们需要保存变量 `e` 和 `f` 的值直到 `fancy_func` 中所有语句执行结束。这是因为在执行 `e = add(a, b)` 和 `f = add(c, d)` 之前我们并不知道变量 `e` 和 `f` 是否会被程序的其他部分使用。

与命令式编程不同，符号式编程通常在计算流程完全定义好后才被执行。大部分的深度学习框架，例如 Theano 和 TensorFlow，都使用了符号式编程。通常，符号式编程的程序需要下面三个步骤：

1. 定义计算流程；
2. 把计算流程编译成可执行的程序；
3. 给定输入，调用编译好的程序执行。

下面我们用符号式编程重新实现本节开头给出的命令式编程代码。

```
In [2]: def add_str():
    return ''

def add(a, b):
    return a + b
'''


def fancy_func_str():
    return ''
def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
'''


def evoke_str():
    return add_str() + fancy_func_str() + ''
print(fancy_func(1, 2, 3, 4))
'''
```

```
prog = evoke_str()
print(prog)
y = compile(prog, '', 'exec')
exec(y)

def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))
```

10

以上定义的三个函数都只是返回计算流程。最后，我们编译完整的计算流程并运行。由于在编译时系统能够完整地看到整个程序，因此有更多空间优化计算。例如，编译的时候可以将程序改写成 `print((1 + 2) + (3 + 4))`，甚至直接改写成 `print(10)`。这样不仅减少了函数调用，还节省了内存。

总结一下，

- 命令式编程更方便。当我们在 Python 里使用命令式编程时，大部分代码编写起来都符合直觉。同时，命令式编程更容易除错。这是因为我们可以很方便地拿到所有的中间变量值并打印，或者使用 Python 的除错工具。
- 符号式编程更高效并更容易移植。一方面，在编译的时候系统可以容易地做更多优化；另一方面，符号式编程可以将程序变成一个与 Python 无关的格式，从而可以使程序在非 Python 环境下运行。

8.1.1 混合式编程取两者之长

大部分的深度学习框架在命令式编程和符号式编程之间二选一。例如 Theano 和受其启发的后来者 TensorFlow 使用了符号式编程；Chainer 和它的追随者 PyTorch 使用了命令式编程。开发人员在设计 Gluon 时思考了这个问题：有没有可能既拿到命令式编程的好处，又享受符号式编程的

优势？开发者们认为，用户应该用纯命令式编程进行开发和调试；当需要产品级别的性能和部署时，用户可以将至少大部分程序转换成符号式来运行。

值得强调的是，Gluon 可以通过混合式编程做到这一点。在混合式编程中，我们可以通过使用 HybridBlock 或者 HybridSequential 类构建模型。默认情况下，它们和 Block 或者 Sequential 类一样依据命令式编程的方式执行。当我们调用 `hybridize` 函数后，Gluon 会转换成依据符号式编程的方式执行。事实上，绝大多数模型都可以享受符号式编程的优势。

本节将通过实验展示混合式编程的魅力。首先，导入本节中实验所需的包或模块。

```
In [3]: from mxnet import nd, sym  
       from mxnet.gluon import nn  
       from time import time
```

8.1.2 使用 HybridSequential 类构造模型

我们之前学习了如何使用 Sequential 类来串联多个层。为了使用混合式编程，下面我们将 Sequential 类替换成 HybridSequential 类。

```
In [4]: def get_net():  
    net = nn.HybridSequential()  
    net.add(  
        nn.Dense(256, activation="relu"),  
        nn.Dense(128, activation="relu"),  
        nn.Dense(2)  
    )  
    net.initialize()  
    return net  
  
x = nd.random.normal(shape=(1, 512))  
net = get_net()  
net(x)
```

```
Out[4]:  
[[ 0.08827581  0.00505182]]  
<NDArray 1x2 @cpu(0)>
```

我们可以通过调用 `hybridize` 函数来编译和优化 HybridSequential 实例中串联的层的计算。模型的计算结果不变。

```
In [5]: net.hybridize()  
net(x)
```

```
Out[5]:  
[[ 0.08827581  0.00505182]]  
<NDArray 1x2 @cpu(0)>
```

需要注意的是，只有继承 HybridBlock 的层才会被优化。例如，HybridSequential 类和 Gluon 提供的 Dense 类都是 HybridBlock 的子类，它们都会被优化计算。如果一个层只是继承自 Block 而不是 HybridBlock 类，那么它将不会被优化。我们接下会讨论如何使用 HybridBlock 类。

性能

我们比较调用 hybridize 函数前后的计算时间来展示符号式编程的性能提升。这里我们计时 1000 次 net 模型计算。在 net 调用 hybridize 函数前后，它分别依据命令式编程和符号式编程做模型计算。

```
In [6]: def benchmark(net, x):  
    start = time()  
    for i in range(1000):  
        y = net(x)  
    # 等待所有计算完成。  
    nd.waitall()  
    return time() - start  
  
net = get_net()  
print('Before hybridizing: %.4f sec' % (benchmark(net, x)))  
net.hybridize()  
print('After hybridizing: %.4f sec' % (benchmark(net, x)))
```

Before hybridizing: 0.2965 sec
After hybridizing: 0.1856 sec

由上面结果可见，在一个 HybridSequential 实例调用 hybridize 函数后，它可以通过符号式编程提升计算性能。

获取符号式程序

在模型 net 根据输入计算模型输出后，例如 benchmark 函数中的 net(x)，我们就可以通过 export 函数来保存符号式程序和模型参数到硬盘。

```
In [7]: net.export('my_mlp')
```

此时生成的.json 和.params 文件分别为符号式程序和模型参数。它们可以被 Python 或 MXNet 支持的其他前端语言读取，例如 C++。这样，我们就可以很方便地使用其他前端语言或在其他设

备上部署训练好的模型。同时，由于部署时使用的是基于符号式编程的程序，计算性能往往比基于命令式编程更好。

在 MXNet 中，符号式程序指的是 Symbol 类型的程序。我们知道，当给 net 提供 NDArray 类型的输入 x 后，net(x) 会根据 x 直接计算模型输出并返回结果。对于调用过 hybridize 函数后的模型，我们还可以给它输入一个 Symbol 类型的变量，net(x) 会返回同样是 Symbol 类型的程序。

```
In [8]: x = sym.var('data')
net(x)
```

```
Out[8]: <Symbol dense5_fwd>
```

8.1.3 使用 HybridBlock 类构造模型

和 Sequential 类与 Block 之间的关系一样，HybridSequential 类是 HybridBlock 的子类。跟 Block 实例需要实现 forward 函数不太一样的是，对于 HybridBlock 实例我们需要实现 hybrid_forward 函数。

前面我们展示了调用 hybridize 函数后的模型可以获得更好的计算性能和移植性。另一方面，调用 hybridize 后的模型会影响灵活性。为了解释这一点，我们先使用 HybridBlock 构造模型。

```
In [9]: class HybridNet(nn.HybridBlock):
    def __init__(self, **kwargs):
        super(HybridNet, self).__init__(**kwargs)
        self.hidden = nn.Dense(10)
        self.output = nn.Dense(2)

    def hybrid_forward(self, F, x):
        print('F: ', F)
        print('x: ', x)
        x = F.relu(self.hidden(x))
        print('hidden: ', x)
        return self.output(x)
```

在继承 HybridBlock 类时，我们需要在 hybrid_forward 函数中添加额外的输入 F。我们知道，MXNet 既有基于命令式编程的 NDArray 类，又有基于符号式编程的 Symbol 类。由于这两个类的函数基本一致，MXNet 会根据输入来决定 F 使用 NDArray 或 Symbol。

下面创建了一个 HybridBlock 实例。可以看到默认下 F 使用 NDArray。而且，我们打印出了输入 x 和使用 ReLU 激活函数的隐藏层的输出。

```
In [10]: net = HybridNet()
        net.initialize()
        x = nd.random.normal(shape=(1, 4))
        net(x)

F:  <module 'mxnet.ndarray' from '/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/py
    ↵  python3.6/site-packages/mxnet/ndarray/__init__.py'>
x:
[[ -0.12225834  0.5429998  -0.94693518  0.59643304]]
<NDArray 1x4 @cpu(0)>
hidden:
[[ 0.11134676  0.04770704  0.05341475  0.          0.08091211  0.          0.
  0.04143535  0.          0.          ]]
<NDArray 1x10 @cpu(0)>

Out[10]:
[[ 0.00370749  0.00134991]]
<NDArray 1x2 @cpu(0)>
```

再运行一次会得到同样的结果。

```
In [11]: net(x)

F:  <module 'mxnet.ndarray' from '/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/py
    ↵  python3.6/site-packages/mxnet/ndarray/__init__.py'>
x:
[[ -0.12225834  0.5429998  -0.94693518  0.59643304]]
<NDArray 1x4 @cpu(0)>
hidden:
[[ 0.11134676  0.04770704  0.05341475  0.          0.08091211  0.          0.
  0.04143535  0.          0.          ]]
<NDArray 1x10 @cpu(0)>

Out[11]:
[[ 0.00370749  0.00134991]]
<NDArray 1x2 @cpu(0)>
```

接下来看看调用 `hybridize` 函数后会发生什么。

```
In [12]: net.hybridize()
        net(x)

F:  <module 'mxnet.symbol' from '/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/py
    ↵  thon3.6/site-packages/mxnet/symbol/__init__.py'>
x:  <Symbol data>
hidden: <Symbol hybridnet0_relu0>
```

```
Out[12]:
```

```
[[ 0.00370749  0.00134991]]  
<NDArray 1x2 @cpu(0)>
```

可以看到，`F` 变成了 `Symbol`。而且，虽然输入数据还是 `NDArray`，但 `hybrid_forward` 函数里，相同输入和中间输出全部变成了 `Symbol`。

再运行一次看看。

```
In [13]: net(x)
```

```
Out[13]:
```

```
[[ 0.00370749  0.00134991]]  
<NDArray 1x2 @cpu(0)>
```

可以看到 `hybrid_forward` 函数里定义的三行打印语句都没有打印任何东西。这是因为上一次在调用 `hybridize` 函数后运行 `net(x)` 的时候，符号式程序已经得到。之后再运行 `net(x)` 的时候 MXNet 将不再访问 Python 代码，而是直接在 C++ 后端执行符号式程序。这也是调用 `hybridize` 后模型计算性能会提升的一个原因。但它可能的问题是我们损失了写程序的灵活性。在上面这个例子中，如果我们希望使用那三行打印语句调试代码，执行符号式程序时会跳过它们无法打印。此外，对于少数 `Symbol` 不支持的函数，例如 `asnumpy`，我们是无法在 `hybrid_forward` 函数中使用并在调用 `hybridize` 函数后进行模型计算的。

8.1.4 小结

- 命令式编程和符号式编程各有优劣。MXNet 通过混合式编程取两者之长。
- 通过 `HybridSequential` 类和 `HybridBlock` 构建的模型可以调用 `hybridize` 来将命令式程序转成符号式程序。我们建议大家使用这种方法获得计算性能的提升。

8.1.5 练习

- 在本节 `HybridNet` 类 `hybrid_forward` 函数中第一行添加 `x.asnumpy()`，运行本节全部代码，观察报错的位置和错误类型。
- 回顾前面几章中你感兴趣的模型，改用 `HybridBlock` 或 `HybridSequential` 类实现。

8.1.6 扫码直达讨论区



8.2 异步计算

MXNet 使用异步计算来提升计算性能。理解它的工作原理既有助于开发更高效的程序，又有助于在内存资源有限的情况下主动降低计算性能从而减小内存开销。

我们先导入本节中实验需要的包或模块。

```
In [1]: from mxnet import autograd, gluon, nd
        from mxnet.gluon import loss as gloss, nn
        import os
        import subprocess
        from time import time
```

8.2.1 MXNet 中的异步计算

广义上，MXNet 包括用户直接用来交互的前端和系统用来执行计算的后端。例如，用户可以使用不同的前端语言编写 MXNet 程序，像 Python、R、Scala 和 C++。无论使用何种前端编程语言，MXNet 程序的执行主要都发生在 C++ 实现的后端。换句话说，用户写好的前端 MXNet 程序会传给后端执行计算。后端有自己的线程在队列中不断收集任务并执行它们。

MXNet 通过前端线程和后端线程的交互实现异步计算。异步计算指，前端线程无需等待当前指令从后端线程返回结果就继续执行后面的指令。为了便于解释，假设 Python 前端线程调用以下四条指令。

```
In [2]: a = nd.ones((1, 2))
        b = nd.ones((1, 2))
        c = a * b + 2
        c
```

Out[2]:

```
[[ 3.  3.]]  
<NDArray 1x2 @cpu(0)>
```

在异步计算中，Python 前端线程执行前三条语句的时候，仅仅是把任务放进后端的队列里就返回了。当最后一条语句需要打印计算结果时，Python 前端线程会等待 C++ 后端线程把 c 的结果计算完。此设计的一个好处是，这里的 Python 前端线程不需要做实际计算。因此，无论 Python 的性能如何，它对整个程序性能的影响很小。只要 C++ 后端足够高效，那么不管前端语言性能如何，MXNet 都可以提供一致的高性能。

下面的例子通过计时来展示异步计算的效果。可以看到，当 `y = nd.dot(x, x)` 返回的时候并没有等待它真正被计算完。

```
In [3]: start = time()  
x = nd.random.uniform(shape=(2000, 2000))  
y = nd.dot(x, x)  
print('workloads are queued: %f sec' % (time() - start))  
print(y)  
print('workloads are completed: %f sec' % (time() - start))  
  
workloads are queued: 0.000546 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: 0.154463 sec
```

的确，除非我们需要打印或者保存计算结果，我们基本无需关心目前结果在内存中是否已经计算好了。只要数据是保存在 NDArray 里并使用 MXNet 提供的运算符，MXNet 将默认使用异步计算来获取高计算性能。

8.2.2 用同步函数让前端等待计算结果

除了前面介绍的 `print` 外，我们还有其他方法让前端线程等待后端的计算结果完成。我们可以使用 `wait_to_read` 函数让前端等待某个的 `NDArray` 的计算结果完成，再执行前端中后面的语句。或者，我们可以用 `waitall` 函数令前端等待前面所有计算结果完成。后者是性能测试中常用的方法。

下面是使用 `wait_to_read` 的例子。输出用时包含了 `y` 的计算时间。

```
In [4]: start = time()
y = nd.dot(x, x)
y.wait_to_read()
time() - start
Out[4]: 0.1252918243408203
```

下面是使用 `waitall` 的例子。输出用时包含了 `y` 和 `z` 的计算时间。

```
In [5]: start = time()
y = nd.dot(x, x)
z = nd.dot(x, x)
nd.waitall()
time() - start
Out[5]: 0.2473442554473877
```

此外，任何将 `NDArray` 转换成其他不支持异步计算的数据结构的操作都会让前端等待计算结果。例如当我们调用 `asnumpy` 和 `asscalar` 函数时：

```
In [6]: start = time()
y = nd.dot(x, x)
y.asnumpy()
time() - start
Out[6]: 0.13226580619812012
In [7]: start = time()
y = nd.dot(x, x)
y.norm().asscalar()
time() - start
Out[7]: 0.16048645973205566
```

上面介绍的 `wait_to_read`、`waitall`、`asnumpy`、`asscalar` 和 `print` 函数会触发让前端等待后端计算结果的行为，我们通常把这类函数称作同步函数。

8.2.3 使用异步计算提升计算性能

在下面例子中，我们用 for 循环不断对 `y` 赋值。当 for 循环内使用同步函数 `wait_to_read` 时，每次赋值不使用异步计算；当 for 循环外使用同步函数 `waitall` 时，则使用异步计算。

```
In [8]: start = time()
for _ in range(1000):
    y = x + 1
    y.wait_to_read()
print('synchronous: %f sec' % (time() - start))

start = time()
for _ in range(1000):
    y = x + 1
nd.waitall()
print('asynchronous: %f sec' % (time() - start))

synchronous: 1.259535 sec
asynchronous: 0.787113 sec
```

我们观察到，使用异步计算能提升一定的计算性能。为了解释这个现象，让我们对 Python 前端线程和 C++ 后端线程的交互稍作简化。在每一次循环中，前端和后端的交互大约可以分为三个阶段：

1. 前端令后端将计算任务 `y = x + 1` 放进队列；
2. 后端从队列中获取计算任务并执行真正的计算；
3. 后端将计算结果返回给前端。

我们将这三个阶段的耗时分别设为 t_1, t_2, t_3 。如果不使用异步计算，执行 1000 次计算的总耗时大约为 $1000(t_1 + t_2 + t_3)$ ；如果使用异步计算，由于每次循环前端都无需等待后端返回计算结果，执行 1000 次计算的总耗时可以降为 $t_1 + 1000t_2 + t_3$ （假设 $1000t_2 > 999t_1$ ）。

8.2.4 异步计算对内存使用的影响

为了解释异步计算对内存使用的影响，让我们先回忆一下前面章节的内容。

在前面章节中实现的模型训练过程中，我们通常会在每个小批量上评测一下模型，例如模型的损失或者精度。细心的你也许发现了，这类评测常用到同步函数，例如 `asscalar` 或者 `asnumpy`。如果去掉这些同步函数，前端会将大量的小批量计算任务在极短的时间内丢给后端，从而可能导

致较大的内存开销。当我们在每个小批量上都使用同步函数时，前端在每次迭代时仅会将一个小批量的任务丢给后端执行计算，并通常会减小内存开销。

由于深度学习模型通常比较大，而内存资源通常有限，我们建议大家在训练模型时对每个小批量都使用同步函数，例如用 `asscalar` 或者 `asnumpy` 评价模型的表现。类似地，在使用模型预测时，为了减小内存开销，我们也建议大家对每个小批量预测时都使用同步函数，例如直接打印出当前小批量的预测结果。

下面我们来演示异步计算对内存使用的影响。我们先定义一个数据获取函数，它会从被调用时开始计时，并定期打印到目前为止获取数据批量总共耗时。

```
In [9]: num_batches = 41
def data_iter():
    start = time()
    batch_size = 1024
    for i in range(num_batches):
        if i % 10 == 0:
            print('batch %d, time %f sec' % (i, time() - start))
        X = nd.random.normal(shape=(batch_size, 512))
        y = nd.ones((batch_size,))
        yield X, y
```

以下定义多层感知机、优化器和损失函数。

```
In [10]: net = nn.Sequential()
net.add(
    nn.Dense(2048, activation='relu'),
    nn.Dense(512, activation='relu'),
    nn.Dense(1),
)
net.initialize()
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'learning_rate':0.005})
loss = gloss.L2Loss()
```

这里定义辅助函数来监测内存的使用。需要注意的是，这个函数只能在 Linux 或 MacOS 运行。

```
In [11]: def get_mem():
    res = subprocess.check_output(['ps', 'u', '-p', str(os.getpid())])
    return int(str(res).split()[15]) / 1e3
```

现在我们可以做测试了。我们先试运行一次让系统把 `net` 的参数初始化。相关内容请参见“模型参数的延后初始化”一节。

```
In [12]: for X, y in data_iter():
```

```

break
loss(y, net(X)).wait_to_read()

batch 0, time 0.000004 sec

对于训练 net 来说, 我们可以自然地使用同步函数 asscalar 将每个小批量的损失从 NDArray
格式中取出, 并打印每个迭代周期后的模型损失。此时, 每个小批量的生成间隔较长, 不过内存
开销较小。

In [13]: mem = get_mem()
for epoch in range(1, 3):
    l_sum = 0
    for X, y in data_iter():
        with autograd.record():
            l = loss(y, net(X))
            l_sum += l.mean().asscalar()
            l.backward()
            trainer.step(X.shape[0])
        print('epoch', epoch, ' loss:', l_sum / num_batches)
    nd.waitall()
    print('increased memory: %f MB' % (get_mem() - mem))

batch 0, time 0.000002 sec
batch 10, time 1.347883 sec
batch 20, time 2.922260 sec
batch 30, time 4.375050 sec
batch 40, time 5.971428 sec
epoch 1 loss: 0.15615208556
batch 0, time 0.000003 sec
batch 10, time 1.466263 sec
batch 20, time 3.025726 sec
batch 30, time 4.485238 sec
batch 40, time 6.062338 sec
epoch 2 loss: 0.0985021460347
increased memory: 11.024000 MB

```

如果去掉同步函数, 虽然每个小批量的生成间隔较短, 训练过程中可能会导致内存开销过大。这是因为默认异步计算下, 前端会将所有小批量计算在短时间内全部丢给后端。

```

In [14]: mem = get_mem()
for epoch in range(1, 3):
    for X, y in data_iter():
        with autograd.record():
            l = loss(y, net(X))
            l.backward()

```

```
        trainer.step(x.shape[0])
        nd.waitall()
        print('increased memory: %f MB' % (get_mem() - mem))

batch 0, time 0.000002 sec
batch 10, time 0.016771 sec
batch 20, time 0.041656 sec
batch 30, time 0.064174 sec
batch 40, time 0.081868 sec
batch 0, time 0.000003 sec
batch 10, time 0.019567 sec
batch 20, time 0.043411 sec
batch 30, time 0.062733 sec
batch 40, time 0.077079 sec
increased memory: 152.508000 MB
```

8.2.5 小结

- MXNet 包括用户直接用来交互的前端和系统用来执行计算的后端。
- MXNet 能够通过异步计算提升计算性能。
- 我们建议使用每个小批量训练或预测时至少使用一个同步函数，从而避免在短时间内将过多计算任务丢给后端。

8.2.6 练习

- 在“使用异步计算提升计算性能”一节中，我们提到使用异步计算可以使执行 1000 次计算的总耗时可以降为 $t_1 + 1000t_2 + t_3$ 。这里为什么要假设 $1000t_2 > 999t_1$ ？

8.2.7 扫码直达讨论区



8.3 自动并行计算

在“异步计算”一节里我们提到 MXNet 后端会自动构建计算图。通过计算图，系统可以知道所有计算的依赖关系，并可以选择将没有依赖关系的多个任务并行执行来获得性能的提升。以“异步计算”一节中的计算图（图 8.1）为例。其中 `a = nd.ones((1, 2))` 和 `b = nd.ones((1, 2))` 这两步计算之间并没有依赖关系。因此，系统可以选择并行执行它们。

通常一个运算符会用掉一个 CPU/GPU 上所有计算资源。例如，`dot` 操作符会用到所有 CPU（即使是有多个 CPU）或单个 GPU 上所有线程。因此在单 CPU/GPU 上并行运行多个运算符可能效果并不明显。本节中探讨的自动并行计算主要关注 CPU 和 GPU 的并行计算，以及计算和通讯的并行。

首先导入本节中实验所需的包或模块。注意，我们需要至少一个 GPU 才能运行本节实验。

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

8.3.1 CPU 和 GPU 的并行计算

我们先介绍 CPU 和 GPU 的并行计算，例如程序中的计算既发生在 CPU，又发生在 GPU 之上。

先定义一个函数，令它做 10 次矩阵乘法。

```
In [2]: def run(x):  
    return [nd.dot(x, x) for _ in range(10)]
```

接下来，分别在 CPU 和 GPU 上创建 NDArray。

```
In [3]: x_cpu = nd.random.uniform(shape=(2000, 2000))  
x_gpu = nd.random.uniform(shape=(6000, 6000), ctx=mx.gpu(0))
```

然后，分别使用它们在 CPU 和 GPU 上运行 `run` 函数并打印所需时间。

```
In [4]: run(x_cpu) # 预热开始。  
run(x_gpu)  
nd.waitall() # 预热结束。  
  
start = time()  
run(x_cpu)  
nd.waitall()  
print('run on CPU: %f sec' % (time()-start))
```

```
start = time()
run(x_gpu)
nd.waitall()
print('run on GPU: %f sec' % (time()-start))

run on CPU: 1.282512 sec
run on GPU: 1.179487 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: 1.367140 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: 1.187153 sec
copy to CPU: 0.526765 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: 1.236587 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

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
```

```
Tue Jul  3 21:57:38 2018
```

NVIDIA-SMI 375.26				Driver Version: 375.26		
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.
<hr/>						
0	Tesla M60	On	0000:00:1D.0	Off	0%	Default
N/A	37C	P0	39W / 150W	298MiB / 7612MiB	0%	Default
<hr/>						
1	Tesla M60	On	0000:00:1E.0	Off	0%	Default
N/A	47C	P0	38W / 150W	289MiB / 7612MiB	0%	Default
<hr/>						
<hr/>						
Processes:				GPU Memory		
GPU	PID	Type	Process name	Usage		
<hr/>						

在“[自动并行计算](#)”一节里，我们介绍过，大部分的运算可以使用所有的 CPU 的全部计算资源，或者单个 GPU 的全部计算资源。但如果使用多个 GPU 训练模型，我们仍然需要实现相应的算法。这些算法中最常用的叫做数据并行。

8.4.1 数据并行

数据并行目前是深度学习里使用最广泛的将模型训练任务划分到多个 GPU 的办法。回忆一下我们在“[梯度下降和随机梯度下降](#)”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：LeNet”一节里介绍的 LeNet 来作为本节的样例模型。

```
In [3]: # 初始化模型参数。
        scale = 0.01
        W1 = nd.random.normal(scale=scale, shape=(20, 1, 3, 3))
        b1 = nd.zeros(shape=20)
        W2 = nd.random.normal(scale=scale, shape=(50, 20, 5, 5))
        b2 = nd.zeros(shape=50)
        W3 = nd.random.normal(scale=scale, shape=(800, 128))
        b3 = nd.zeros(shape=128)
        W4 = nd.random.normal(scale=scale, shape=(128, 10))
        b4 = nd.zeros(shape=10)
        params = [W1, b1, W2, b2, W3, b3, W4, b4]

        # 定义模型。
        def lenet(X, params):
            h1_conv = nd.Convolution(data=X, weight=params[0], bias=params[1],
                                    kernel=(3, 3), num_filter=20)
            h1_activation = nd.relu(h1_conv)
            h1 = nd.Pooling(data=h1_activation, pool_type="avg", kernel=(2, 2),
                            stride=(2, 2))
            h2_conv = nd.Convolution(data=h1, weight=params[2], bias=params[3],
                                    kernel=(5, 5), num_filter=50)
            h2_activation = nd.relu(h2_conv)
            h2 = nd.Pooling(data=h2_activation, pool_type="avg", kernel=(2, 2),
                            stride=(2, 2))
```

```

    h2 = nd.flatten(h2)
    h3_linear = nd.dot(h2, params[4]) + params[5]
    h3 = nd.relu(h3_linear)
    y_hat = nd.dot(h3, params[6]) + params[7]
    return y_hat

# 交叉熵损失函数。
loss = gloss.SoftmaxCrossEntropyLoss()

```

8.4.3 多 GPU 之间同步数据

我们需要实现一些多 GPU 之间同步数据的辅助函数。下面函数将模型参数复制到某个特定 GPU 并初始化梯度。

```
In [4]: def get_params(params, ctx):
    new_params = [p.copyto(ctx) for p in params]
    for p in new_params:
        p.attach_grad()
    return new_params
```

试一试把 `params` 复制到 `mx.gpu(0)` 上。

```
In [5]: new_params = get_params(params, mx.gpu(0))
print('b1 weight:', new_params[1])
print('b1 grad:', new_params[1].grad)

b1 weight:
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
 0.  0.]
<NDArray 20 @gpu(0)>
b1 grad:
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
 0.  0.]
<NDArray 20 @gpu(0)>
```

给定分布在多个 GPU 之间的数据。以下函数可以把各个 GPU 上的数据加起来，然后再广播到所有 GPU 上。

```
In [6]: def allreduce(data):
    for i in range(1, len(data)):
        data[0][:] += data[i].copyto(data[0].context)
    for i in range(1, len(data)):
        data[0].copyto(data[i])
```

简单测试一下 allreduce 函数。

```
In [7]: data = [nd.ones((1,2), ctx=mx.gpu(i)) * (i + 1) for i in range(2)]
    print('before allreduce:', data)
    allreduce(data)
    print('after allreduce:', data)

before allreduce: [
[[ 1.  1.]]
<NDArray 1x2 @gpu(0)>,
[[ 2.  2.]]
<NDArray 1x2 @gpu(1)>]
after allreduce: [
[[ 3.  3.]]
<NDArray 1x2 @gpu(0)>,
[[ 3.  3.]]
<NDArray 1x2 @gpu(1)>]
```

给定一个批量的数据样本，以下函数可以划分它们并复制到各个 GPU 上。

```
In [8]: def split_and_load(data, ctx):
    n, k = data.shape[0], len(ctx)
    m = n // k
    assert m * k == n, '# examples is not divided by # devices.'
    return [data[i * m: (i + 1) * m].as_in_context(ctx[i]) for i in range(k)]
```

让我们试着用 split_and_load 函数将 6 个数据样本平均分给 2 个 GPU。

```
In [9]: batch = nd.arange(24).reshape((6, 4))
    ctx = [mx.gpu(0), mx.gpu(1)]
    splitted = split_and_load(batch, ctx)
    print('input: ', batch)
    print('load into', ctx)
    print('output:', splitted)

input:
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9.  10. 11.]
 [ 12. 13. 14. 15.]
 [ 16. 17. 18. 19.]
 [ 20. 21. 22. 23.]]
<NDArray 6x4 @cpu(0)>
load into [gpu(0), gpu(1)]
output:
[[ 0.  1.  2.  3.]]
```

```
[ 4.  5.  6.  7.]
[ 8.  9. 10. 11.]]
<NDArray 3x4 @gpu(0)>,
[[ 12. 13. 14. 15.]
 [ 16. 17. 18. 19.]
 [ 20. 21. 22. 23.]]
<NDArray 3x4 @gpu(1)>]
```

8.4.4 单个小批量上的多 GPU 训练

现在我们可以实现单个小批量上的多 GPU 训练了。它的实现主要依据本节介绍的数据并行方法。我们将使用刚刚定义的多 GPU 之间同步数据的辅助函数，例如 `split_and_load` 和 `allreduce`。

```
In [10]: def train_batch(X, y, gpu_params, ctx, lr):
    # 划分小批量数据样本并复制到各个 GPU 上。
    gpu_Xs = split_and_load(X, ctx)
    gpu_ys = split_and_load(y, ctx)
    # 在各个 GPU 上计算损失。
    with autograd.record():
        ls = [loss(lenet(gpu_X, gpu_W), gpu_y)
              for gpu_X, gpu_y, gpu_W in zip(gpu_Xs, gpu_ys, gpu_params)]
    # 在各个 GPU 上反向传播。
    for l in ls:
        l.backward()
    # 把各个 GPU 上的梯度加起来，然后再广播到所有 GPU 上。
    for i in range(len(gpu_params[0])):
        allreduce([gpu_params[c][i].grad for c in range(len(ctx))])
    # 在各个 GPU 上更新自己维护的那一份完整的模型参数。
    for param in gpu_params:
        gb.sgd(param, lr, X.shape[0])
```

8.4.5 训练函数

现在我们可以定义训练函数。这里的训练函数和之前章节里的训练函数稍有不同。例如，在这里我们需要依据本节介绍的数据并行，将完整的模型参数复制到多个 GPU 上，并在每次迭代时对单个小批量上进行多 GPU 训练。

```
In [11]: def train(num_gpus, batch_size, lr):
    train_iter, test_iter = gb.load_data_fashion_mnist(batch_size)
    ctx = [mx.gpu(i) for i in range(num_gpus)]
```

```
print('running on:', ctx)
# 将模型参数复制到 num_gpus 个 GPU 上。
gpu_params = [get_params(params, c) for c in ctx]
for epoch in range(1, 6):
    start = time()
    for X, y in train_iter:
        # 对单个小批量上进行多 GPU 训练。
        train_batch(X, y, gpu_params, ctx, lr)
    nd.waitall()
    print('epoch %d, time: %.1f sec' % (epoch, time() - start))
    # 在 GPU0 上验证模型。
    net = lambda x: lenet(x, gpu_params[0])
    test_acc = gb.evaluate_accuracy(test_iter, net, ctx[0])
    print('validation accuracy: %.4f' % test_acc)
```

我们先使用一个 GPU 来训练。

```
In [12]: train(num_gpus=1, batch_size=256, lr=0.3)
```

```
running on: [gpu(0)]
epoch 1, time: 2.2 sec
validation accuracy: 0.1001
epoch 2, time: 1.8 sec
validation accuracy: 0.7244
epoch 3, time: 1.8 sec
validation accuracy: 0.7715
epoch 4, time: 1.8 sec
validation accuracy: 0.8216
epoch 5, time: 1.8 sec
validation accuracy: 0.8068
```

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: 0.0995
epoch 2, time: 1.0 sec
validation accuracy: 0.1557
epoch 3, time: 1.0 sec
validation accuracy: 0.6800
epoch 4, time: 1.0 sec
validation accuracy: 0.7640
epoch 5, time: 1.0 sec
```

```
validation accuracy: 0.7495
```

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 计算的 Gluon 实现

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”一节里介绍的多 GPU 之间同步数据的辅助函数。

先导入本节实验需要的包或模块。同上一节，运行本节中的程序需要至少两块 GPU。

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

```
from mxnet import autograd, gluon, init, nd
from mxnet.gluon import loss as gloss, utils as gutils
from time import time
```

8.5.1 多 GPU 上初始化模型参数

我们使用 ResNet-18 来作为本节的样例模型。

```
In [2]: net = gb.resnet18(10)
```

之前我们介绍了如何使用 `initialize` 函数的 `ctx` 参数在 CPU 或单个 GPU 上初始化模型参数。事实上，`ctx` 可以接受一系列的 CPU/GPU，从而使初始化好的模型参数复制到 `ctx` 里所有的 CPU/GPU 上。

```
In [3]: ctx = [mx.gpu(0), mx.gpu(1)]
net.initialize(init=init.Normal(sigma=0.01), ctx=ctx)
```

Gluon 提供了上一节中实现的 `split_and_load` 函数。它可以划分一个小批量的数据样本并复制到各个 CPU/GPU 上。之后，根据输入数据所在的 CPU/GPU，模型计算会发生在相同的 CPU/GPU 上。

```
In [4]: x = nd.random.uniform(shape=(4, 1, 28, 28))
gpu_x = gutils.split_and_load(x, ctx)
net(gpu_x[0]), net(gpu_x[1])
```

```
Out[4]: (
    [[ 3.22364103e-05 -1.77604452e-05 -4.53473040e-05 -4.43779390e-06
       4.11345136e-05 -1.12404132e-05 -3.77615615e-05  4.36321279e-05
       2.28628483e-06 -1.48372719e-05]
     [ 3.10487958e-05 -1.89744533e-05 -4.50448642e-05 -5.33929597e-06
       4.11167130e-05 -1.23149039e-05 -3.99470700e-05  4.46575323e-05
       2.58294403e-06 -1.34151524e-05]]
<NDArray 2x10 @gpu(0)>,
    [[ 2.98842060e-05 -1.86083871e-05 -4.19585049e-05 -5.71156716e-06
       4.02533478e-05 -1.19904144e-05 -3.85621279e-05  4.39719333e-05
       6.29562464e-08 -1.58273233e-05]
     [ 3.08366398e-05 -1.81654450e-05 -4.28299536e-05 -5.13964960e-06
       3.97496297e-05 -1.23091431e-05 -3.68527180e-05  4.15482536e-05
       3.11583290e-06 -1.18256366e-05]]
<NDArray 2x10 @gpu(1)>)
```

回忆一下“模型参数的延后初始化”一节中介绍的延后的初始化。现在，我们可以通过 `data` 访问初始化好的模型参数值了。需要注意的是，默认下 `weight.data()` 会返回 CPU 上的参数值。

由于我们指定了 2 个 GPU 来初始化模型参数，我们需要指定 GPU 访问。我们看到，相同参数在不同的 GPU 上的值一样。

```
In [5]: weight = net[1].params.get('weight')
try:
    weight.data()
except:
    print('not initialized on', mx.cpu())
weight.data(ctx[0])[0], weight.data(ctx[1])[0]

not initialized on cpu(0)

Out[5]: (
[[[-0.01473444 -0.01073093 -0.01042483]
 [-0.01327885 -0.01474966 -0.00524142]
 [ 0.01266256  0.00895064 -0.00601594]]]
<NDArray 1x3x3 @gpu(0)>,
[[[-0.01473444 -0.01073093 -0.01042483]
 [-0.01327885 -0.01474966 -0.00524142]
 [ 0.01266256  0.00895064 -0.00601594]]]
<NDArray 1x3x3 @gpu(1)>)
```

8.5.2 多 GPU 训练模型

我们先定义交叉熵损失函数。

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

当我们使用多个 GPU 来训练模型时，`gluon.Trainer` 会自动做数据并行，例如划分小批量数据样本并复制到各个 GPU 上，对各个 GPU 上的梯度求和再广播到所有 GPU 上。这样，我们就很方便地实现训练函数了。

```
In [7]: def train(num_gpus, batch_size, lr):
    train_iter, test_iter = gb.load_data_fashion_mnist(batch_size)
    ctx = [mx.gpu(i) for i in range(num_gpus)]
    print('running on:', ctx)
    net.initialize(init=init.Normal(sigma=0.01), ctx=ctx, force_reinit=True)
    trainer = gluon.Trainer(
        net.collect_params(), 'sgd', {'learning_rate': lr})
    for epoch in range(1, 6):
        start = time()
        for X, y in train_iter:
            gpu_Xs = gutils.split_and_load(X, ctx)
            gpu_ys = gutils.split_and_load(y, ctx)
```

```

with autograd.record():
    ls = [loss(net(gpu_X), gpu_y) for gpu_X, gpu_y in zip(
        gpu_Xs, gpu_ys)]
    for l in ls:
        l.backward()
    trainer.step(batch_size)
nd.waitall()
print('epoch %d, training time: %.1f sec'%(epoch, time() - start))
test_acc = gb.evaluate_accuracy(test_iter, net, ctx[0])
print('validation accuracy: %.4f'%(test_acc))

```

我们在 2 个 GPU 上训练模型。

In [8]: train(num_gpus=2, batch_size=512, lr=0.3)

```

running on: [gpu(0), gpu(1)]
epoch 1, training time: 7.8 sec
validation accuracy: 0.7047
epoch 2, training time: 7.0 sec
validation accuracy: 0.8020
epoch 3, training time: 7.0 sec
validation accuracy: 0.8005
epoch 4, training time: 7.0 sec
validation accuracy: 0.8770
epoch 5, training time: 7.0 sec
validation accuracy: 0.8531

```

8.5.3 小结

- 在 Gluon 中，我们可以很方便地进行多 GPU 计算，例如在多 GPU 上初始化模型参数和训练模型。

8.5.4 练习

- 本节使用了 ResNet-18。试试不同的迭代周期、批量大小和学习率。如果条件允许，使用更多 GPU 计算。
- 有时候，不同的 CPU/GPU 的计算能力不一样，例如同时使用 CPU 和 GPU，或者 GPU 之间型号不一样。这时候应该怎么办？

8.5.5 扫码直达讨论区



9.1 图片增广

在“深度卷积神经网络：AlexNet”小节里我们提到过大规模数据集是深度网络能成功的前提条件。在 AlexNet 当年能取得的成功中，图片增广（image augmentation）功不可没。本小节我们将讨论这个在计算机视觉里被广泛使用的技术。

图片增广是指通过对训练图片做一系列变化来产生相似但又有不同的训练样本，这样来模型训练的时候识别了难以泛化的模式。例如我们可以对图片进行不同的裁剪使得感兴趣的物体出现在不同的位置中，从而使得模型减小对物体出现位置的依赖性。也可以调整亮度色彩等因素来降低模型对色彩的敏感度。

9.1.1 常用增广方法

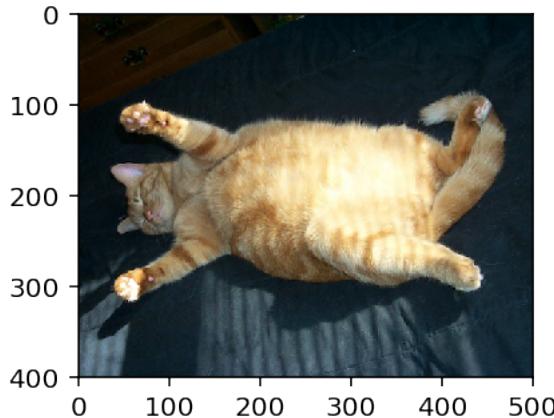
我们首先读取一张 400×500 的图片作为样例解释常用的增广方法。

```
In [1]: %matplotlib inline  
import sys
```

```
sys.path.insert(0, '..')
import gluonbook as gb
from mxnet import nd, image, gluon, init
from mxnet.gluon.data.vision import transforms

img = image.imread('../img/cat1.jpg')
gb.plt.imshow(img.asnumpy())
```

Out[1]: <matplotlib.image.AxesImage at 0x7f43b06546a0>



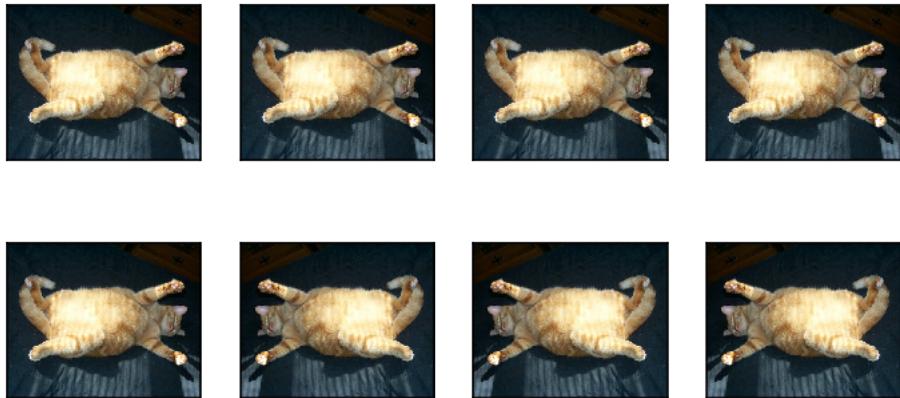
因为大部分的增广方法都有一定的随机性。接下来我们定义一个辅助函数，它对输入图片 `img` 运行多次增广方法 `aug` 并画出结果。

```
In [2]: def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
    Y = [aug(img) for _ in range(num_rows*num_cols)]
    gb.show_images(Y, num_rows, num_cols, scale)
```

变形

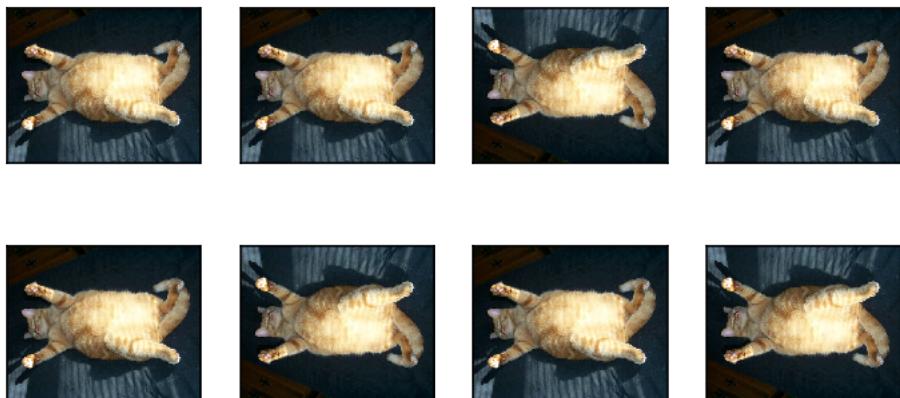
左右翻转图片通常不影响识别图片，它是最早也是最广泛使用的一种增广。下面我们使用 `transform` 模块里的 `RandomFlipLeftRight` 类来实现按 0.5 的概率左右翻转图片：

```
In [3]: apply(img, transforms.RandomFlipLeftRight())
```



当然有时候我们也使用上下翻转，至少对于我们使用的图片，上下翻转不会造成人的识别障碍。

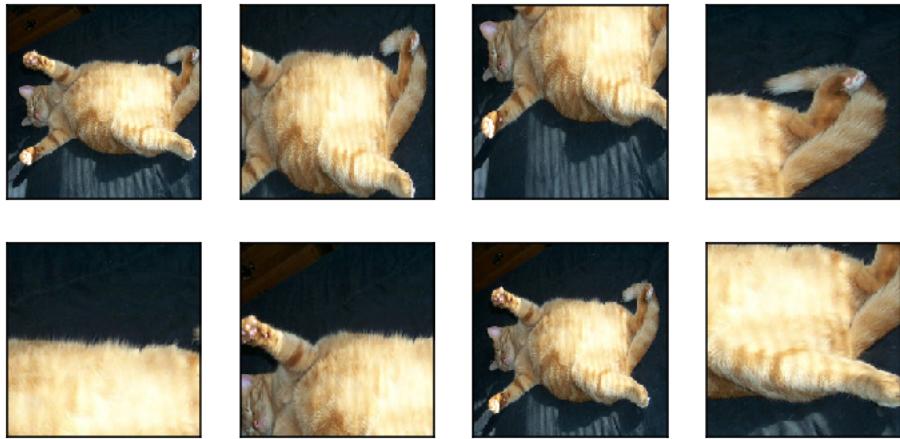
```
In [4]: apply(img, transforms.RandomFlipTopBottom())
```



我们使用的样例图片里猫在图片正中间，但一般情况下可能不是这样。“池化层”一节里我们解释了池化层能弱化卷积层对目标位置的敏感度，另一方面我们通过对图片随机剪裁来使的物体以不同的比例出现在不同位置。

下面代码里我们每次随机裁剪一片面积为原面积 10% 到 100% 的区域，其宽和高的比例在 0.5 和 2 之间，然后再将高宽缩放到 200 像素。

```
In [5]: shape_aug = transforms.RandomResizedCrop(  
    (200, 200), scale=(0.1, 1), ratio=(0.5, 2))  
apply(img, shape_aug)
```



颜色变化

形状变化外的一个另一大类是变化颜色。颜色一般有四个可以调的参数：亮度、对比、饱和度和色相。下面例子里我们随机将亮度在当前值上增加或减小一个在 0 到 50% 之前的量。

```
In [6]: apply(img, transforms.RandomBrightness(0.5))
```



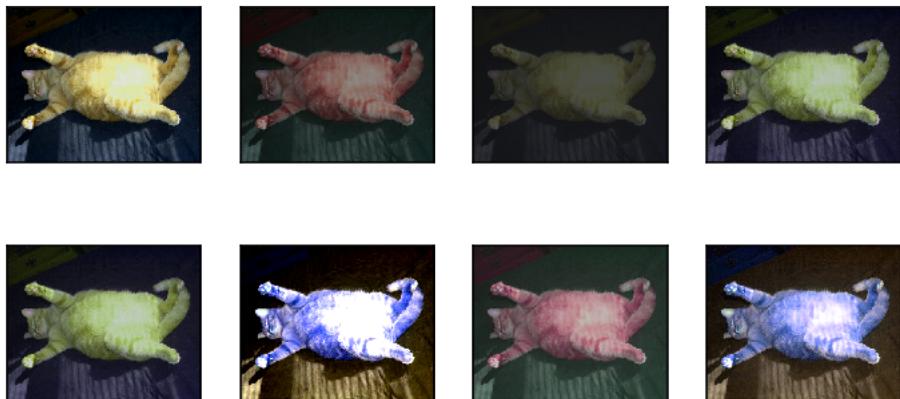
同样的修改色相。

```
In [7]: apply(img, transforms.RandomHue(0.5))
```



或者用使用 `RandomColorJitter` 来一起使用。

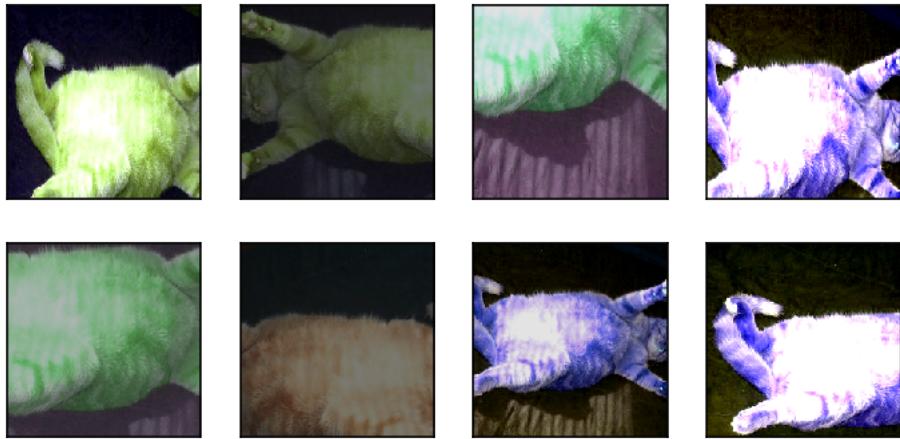
```
In [8]: color_aug = transforms.RandomColorJitter(  
    brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)  
apply(img, color_aug)
```



使用多个增广

实际应用中我们会将多个增广叠加使用。我们可以使用 `Compose` 类来将多个增广串联起来。

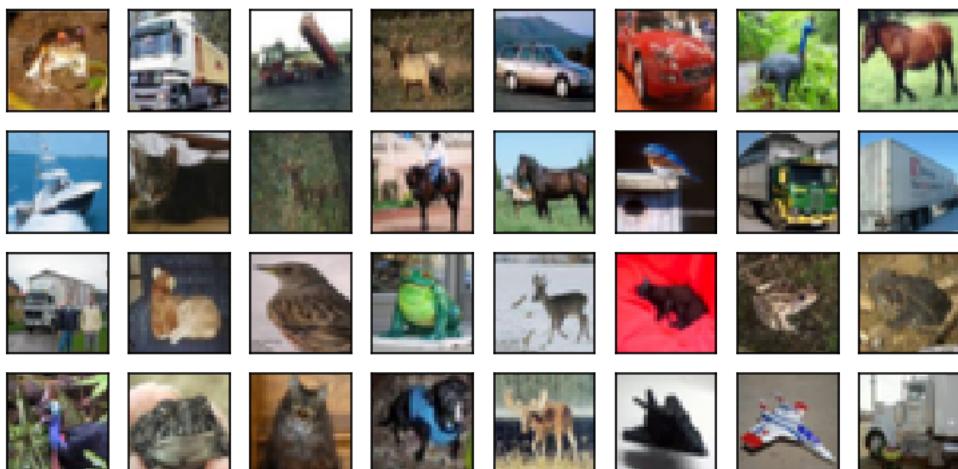
```
In [9]: augs = transforms.Compose([  
    transforms.RandomFlipLeftRight(), color_aug, shape_aug])  
apply(img, augs)
```



9.1.2 使用图片增广来训练

接下来我们来看一个将图片增广应用在实际训练的例子，并比较其与不使用时的区别。这里我们使用 CIFAR-10 数据集，而不是之前我们一直使用的 FashionMNIST。原因在于 FashionMNIST 中物体位置和尺寸都已经统一化了，而 CIFAR-10 中物体颜色和大小区别更加显著。下面我们展示 CIFAR-10 中的前 32 张训练图片。

```
In [10]: gb.show_images(gluon.data.vision.CIFAR10(train=True)[0:32][0], 4, 8,  
                      scale=0.8);
```



在训练时，我们通常将图片增广作用在训练图片上，使得模型能识别出各种变化过后的版本。这里我们仅仅使用最简单的随机水平翻转。此外我们使用 `ToTensor` 变换来图片转成 MXNet 需要的格式，即格式为（批量，通道，高，宽）以及类型为 32 位浮点数。

```
In [11]: train_augs = transforms.Compose([
    transforms.RandomFlipLeftRight(),
    transforms.ToTensor(),
])

test_augs = transforms.Compose([
    transforms.ToTensor(),
])
```

接下来我们定义一个辅助函数来方便读取图片并应用增广。Gluon 的数据集提供 `transform_first` 函数来对数据里面的第一项图片（标签为第二项）来应用增广。另外图片增广将增加计算复杂度，我们使用两个额外 CPU 进程加来加速计算。

```
In [12]: def load_cifar10(is_train, augs, batch_size):
    return gluon.data.DataLoader(gluon.data.vision.CIFAR10(
        train=is_train).transform_first(augs),
        batch_size=batch_size, shuffle=is_train, num_workers=2)
```

模型训练

我们使用 ResNet 18 来训练 CIFAR-10。训练的的代码跟“残差网络：ResNet”一致，除了使用所有可用的 GPU 和不同的学习率外。

```
In [13]: def train(train_augs, test_augs, lr=0.1):
    batch_size = 256
    ctx = gb.try_all_gpus()
    net = gb.resnet18(10)
    net.initialize(ctx=ctx, init=init.Xavier())
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate':lr})
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    train_data = load_cifar10(True, train_augs, batch_size)
    test_data = load_cifar10(False, test_augs, batch_size)
    gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=8)
```

首先我们看使用了图片增广的情况。

```
In [14]: train(train_augs, test_augs)

training on [gpu(0), gpu(1)]
epoch 1, loss 1.4883, train acc 0.468, test acc 0.428, time 11.9 sec
```

```
epoch 2, loss 1.0207, train acc 0.636, test acc 0.579, time 11.1 sec
epoch 3, loss 0.8160, train acc 0.711, test acc 0.578, time 11.1 sec
epoch 4, loss 0.6819, train acc 0.760, test acc 0.609, time 11.1 sec
epoch 5, loss 0.5938, train acc 0.794, test acc 0.606, time 11.1 sec
epoch 6, loss 0.5127, train acc 0.822, test acc 0.525, time 11.1 sec
epoch 7, loss 0.4529, train acc 0.844, test acc 0.704, time 11.1 sec
epoch 8, loss 0.3968, train acc 0.864, test acc 0.651, time 11.1 sec
```

作为对比，我们只对训练数据做中间剪裁。

```
In [15]: train(test_augs, test_augs)

training on [gpu(0), gpu(1)]
epoch 1, loss 1.4913, train acc 0.464, test acc 0.492, time 11.3 sec
epoch 2, loss 1.0113, train acc 0.639, test acc 0.597, time 11.0 sec
epoch 3, loss 0.7749, train acc 0.728, test acc 0.572, time 11.0 sec
epoch 4, loss 0.6148, train acc 0.783, test acc 0.615, time 11.1 sec
epoch 5, loss 0.4854, train acc 0.830, test acc 0.673, time 11.1 sec
epoch 6, loss 0.3625, train acc 0.873, test acc 0.483, time 11.0 sec
epoch 7, loss 0.2608, train acc 0.912, test acc 0.627, time 10.9 sec
epoch 8, loss 0.1669, train acc 0.945, test acc 0.563, time 10.9 sec
```

可以看到，即使是简单的随机翻转也会有明显效果。使用增广类似于增加了正则项话，它使得训练精度变低，但对提升测试精度有帮助。

9.1.3 小结

- 图片增广对现有训练数据生成大量随机图片来有效避免过拟合。

9.1.4 练习

- 尝试在 CIFAR-10 训练中增加不同的增广方法。

9.1.5 扫码直达讨论区



9.2 微调

之前章节里我们通过大量样例演示了如何在只有 6 万张图片的 FashionMNIST 上训练模型。我们也介绍了 ImageNet 这个当下学术界使用最广的大数据集，它有超过一百万的图片和一千类的物体。但我们平常接触到数据集的规模通常在两者之间。

想象一下开发一个应用来从图片中识别里面的凳子然后提供购买链接给用户。一个可能的做法是先去找一百把常见的凳子，对每个凳子收集一千张不同的图片，然后在收集到的数据上训练一个分类器。这个数据集虽然可能比 FashionMNIST 要复杂，但仍然比 ImageNet 小 10 倍。这可能导致针对 ImageNet 提出的模型在这个数据上会过拟合。同时因为数据量有限，最终我们得到的分类器的模型精度也许达不到实用的要求。

一个解决办法是收集更多的数据。但注意到收集和标注数据均会花费大量的人力和财力。例如 ImageNet 这个数据集花费了数百万美元的研究经费。虽然目前的数据采集成本降低了十倍以上，但其成本仍然不可忽略。

另外一种解决办法是迁移学习 (transfer learning)，它通过将其他数据集来帮助学习当前数据集。例如，虽然 ImageNet 的图片基本跟椅子无关，但其上训练到的模型可能能做一些通用的图片特征抽取，例如识别边缘、纹理、形状和物体组成。这个对于识别椅子也可能同样有效。

本小节我们介绍迁移学习里面的一个常用技术：微调 (fine tuning)。它由下面四步构成：

1. 在源数据（例如 ImageNet）上训练一个神经网络 A 。
2. 创建一个新的神经网络 B ，它复制 A 上除了输出层外的所有模型参数。这里的假设是这些模型参数含有源数据上学习到的知识，这些知识同样适用于目标数据集。但最后的输出层跟源数据标注紧密相关，所以不被重用。
3. 为 B 添加一个输出大小为目标数据集类别数目（例如一百类椅子）的输出层，并将其权重初始化成随机值。

4. 在目标数据集（例如椅子数据集）上训练 B 。我们将从头开始学习输出层，但其余层都是基于源数据上的模型参数进行微调。

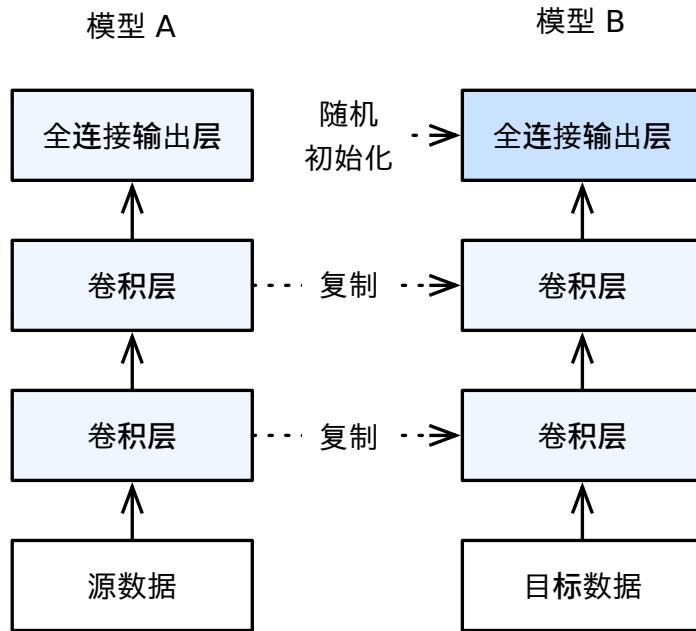


图 9.1: 微调。

接下来我们来看一个具体的例子，它使用 ImageNet 上训练好的 ResNet 来微调一个我们构造的小数据集：其含有数千张包含热狗和不包含热狗的图片。

9.2.1 热狗识别

获取数据

我们使用的热狗数据集是从网上抓取的，它含有 1400 张包含热狗的正类图片，和同样多包含其他食品的负类图片。各类的 1000 张图片被用作训练，其余的作为测试。

我们首先将数据下载到`../data`。在当前目录解压后得到 `hotdog/train` 和 `hotdog/test` 这两个文件夹。每个下面有 `hotdog` 和 `not-hotdog` 这两个类别文件夹，里面是对应的图片文

件。

```
In [1]: import sys
        sys.path.insert(0, '..')
        import zipfile
        import gluonbook as gb
        from mxnet import nd, gluon, init
        from mxnet.gluon import data as gdata, loss as gloss, model_zoo, utils as
    ↵  gutils
        from mxnet.gluon.data.vision import transforms

        data_dir = '../data/'
        base_url = 'https://apache-mxnet.s3-accelerate.amazonaws.com/'
        fname = gutils.download(
            base_url+'gluon/dataset/hotdog.zip',
            path=data_dir, sha1_hash='fba480ffa8aa7e0febbb511d181409f899b9baa5')

        with zipfile.ZipFile(fname, 'r') as z:
            z.extractall(data_dir)
```

我们使用使用 `ImageFolderDataset` 类来读取数据。它将每个类别文件夹当做一个类，并读取下面所有的图片。

```
In [2]: train_imgs = gdata.vision.ImageFolderDataset(data_dir+'/hotdog/train')
        test_imgs = gdata.vision.ImageFolderDataset(data_dir+'/hotdog/test')
```

下面画出前 8 张正例图片和最后的 8 张负例图片，可以看到他们性质和高宽各不相同。

```
In [3]: hotdogs = [train_imgs[i][0] for i in range(8)]
        not_hotdogs = [train_imgs[-i-1][0] for i in range(8)]
        gb.show_images(hotdogs+not_hotdogs, 2, 8, scale=1.4); # 加分号只显示图。
```

我们将训练图片首先扩大到高宽为 480，然后随机剪裁出高宽为 224 的输入。测试图片则是简单的中心剪裁。此外，我们对输入的 RGB 通道数值进行了归一化。

```
In [4]: # 指定 RGB 三个通道的均值和方差来将图片通道归一化。
        normalize = transforms.Normalize(
            [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

        train_augs = transforms.Compose([
            transforms.Resize(480),
            transforms.RandomResizedCrop(224),
            transforms.RandomFlipLeftRight(),
            transforms.ToTensor(),
            normalize,
        ])
```

```
test_augs = transforms.Compose([
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    normalize
])
```

微调模型

我们用在 ImageNet 上训练好了 ResNet 18 来作为基础模型。这里指定 `pretrained=True` 来自动下载并加载训练好的权重。

```
In [5]: pretrained_net = model_zoo.vision.resnet18_v2(pretrained=True)
```

预训练好的模型由两块构成，一是 `features`，二是 `output`。前者包含从输入开始的大部分卷积和全连接层，后者主要包括最后一层全连接层。这样的划分的主要目的是为了更方便做微调。下面查看下 `output` 的内容：

```
In [6]: pretrained_net.output
```

```
Out[6]: Dense(512 -> 1000, linear)
```

它将 ResNet 最后的全局平均池化层输出转化成 1000 类的输出。

在微调中，我们新建一个网络，它的定义跟之前训练好的网络一样，除了最后的输出数等于当前数据的类别数。就是说新网络的 `features` 被初始化成前面训练好网络的权重，而 `output` 则是从头开始训练。

```
In [7]: finetune_net = model_zoo.vision.resnet18_v2(classes=2)
finetune_net.features = pretrained_net.features
finetune_net.output.initialize(init.Xavier())
```

9.2.2 训练

我们先定义一个可以重复使用的训练函数。

```
In [8]: def train(net, learning_rate, batch_size=128, epochs=5):
    train_data = gdata.DataLoader(
        train_imgs.transform_first(train_augs), batch_size, shuffle=True)
    test_data = gdata.DataLoader(
        test_imgs.transform_first(test_augs), batch_size)

    ctx = gb.try_all_gpus()
```

```
net.collect_params().reset_ctx(ctx)
net.hybridize()
loss = gloss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), 'sgd', {
    'learning_rate': learning_rate, 'wd': 0.001})
gb.train(train_data, test_data, net, loss, trainer, ctx, epochs)
```

因为微调的网络中的主要层的已经训练的足够好，所以一般采用比较小的学习率，防止过大的步长对训练好的层产生过多影响。

```
In [9]: train(finetune_net, 0.01)

training on [gpu(0), gpu(1)]
epoch 1, loss 0.4519, train acc 0.801, test acc 0.896, time 19.3 sec
epoch 2, loss 0.2618, train acc 0.891, test acc 0.905, time 16.8 sec
epoch 3, loss 0.2116, train acc 0.910, test acc 0.921, time 16.8 sec
epoch 4, loss 0.1960, train acc 0.926, test acc 0.925, time 16.7 sec
epoch 5, loss 0.1663, train acc 0.933, test acc 0.930, time 16.8 sec
```

为了对比起见，我们训练同样的一个模型，但所有参数都初始成随机值。我们使用较大的学习率来加速收敛。

```
In [10]: scratch_net = model_zoo.vision.resnet18_v2(classes=2)
scratch_net.initialize(init=init.Xavier())
train(scratch_net, 0.1)

training on [gpu(0), gpu(1)]
epoch 1, loss 0.6134, train acc 0.736, test acc 0.777, time 17.0 sec
epoch 2, loss 0.3897, train acc 0.821, test acc 0.770, time 16.8 sec
epoch 3, loss 0.3943, train acc 0.829, test acc 0.804, time 16.8 sec
epoch 4, loss 0.3594, train acc 0.846, test acc 0.836, time 16.8 sec
epoch 5, loss 0.3399, train acc 0.852, test acc 0.825, time 17.2 sec
```

可以看到，微调的模型因为初始值更好，它的收敛比从头开始训练要快很多。在很多情况下，微调的模型最终的收敛到的结果也可能比非微调的模型更好。

9.2.3 小结

- 微调通过将模型部分权重初始化成在源数据集上预训练好的模型参数，从而将模型在源数据集上学到的知识迁移到目标数据上。

9.2.4 练习

- 对 `finetune_net` 试着增大学习率看看收敛变化。
- 多跑几个 `epochs` 直到收敛（你可以也需要调调参数），看看 `scratch_net` 和 `finetune_net` 最后的精度是不是有区别
- 这里 `finetune_net` 重用了 `pretrained_net` 除最后全连接外的所有权重，试试少重用些权重，有会有什么区别
- 事实上 `ImageNet` 里也有 `hotdog` 这个类，它对应的输出层参数可以如下拿到。试试如何使用它。

```
In [11]: weight = pretrained_net.output.weight  
hotdog_w = nd.split(weight.data(), 1000, axis=0)[713]  
hotdog_w.shape
```

```
Out[11]: (1, 512)
```

- 试试不让 `finetune_net` 里重用的权重参与训练，也就是不更新他们的权重。

9.2.5 扫码直达讨论区



9.3 物体检测和边界框

前面小节里我们介绍了诸多用于图片分类的模型。在这个任务里，我们假设图片里只有一个主体物体，然后目标是识别这个物体的类别。但很多时候图片里有多个感兴趣的物体，我们不仅想知道它们是什么，而且想得到它们在图片中的具体位置。在计算机视觉里，我们将这类任务称之为物体检测。

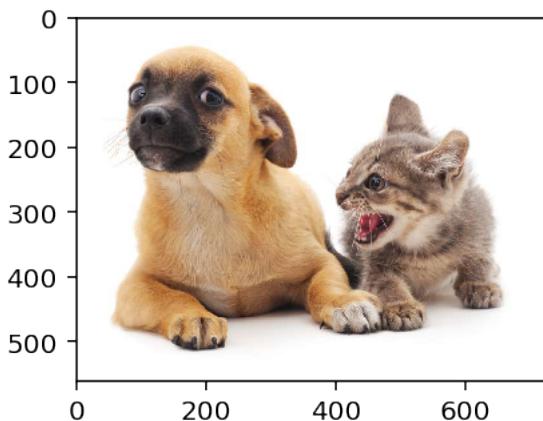
物体检测在多个领域被广泛使用。例如在无人驾驶里，我们需要识别拍摄到的图片里的车辆、行人、道路和障碍的位置来规划行进线路。机器人里也常使用它来检测感兴趣物体。安防领域则需

要检测异常物体，例如歹徒或者炸弹。

在接下来的数小节里我们将介绍物体检测里的多个深度学习模型。在此之前，让我们先讨论物体位置这个概念。首先我们加载本小节将使用的示例图片。

```
In [1]: %matplotlib inline
import sys
sys.path.insert(0, '...')
import gluonbook as gb
from mxnet import image
```

```
In [2]: img = image.imread('../img/catdog.jpg').asnumpy()
gb.plt.imshow(img); # 用 ; 使得不要显示它的输出。
```



可以看到图片左边是一只小狗，右边是一只小猫。跟前面使用的图片的主要不同点在于这里有两个主体物体。

9.3.1 边界框

在物体识别里，我们通常使用边界框（bounding box）来确定物体位置。它一个矩形框，可以由左上角的 x、y 轴位置与右下角 x、y 轴位置确定。我们根据上图坐标信息来定义图中小狗和小猫的边界框。

```
In [3]: # 注意坐标轴原点是图片的左上角。bbox 是 bounding box 的缩写。
dog_bbox = [60, 0, 340, 365]
cat_bbox = [360, 80, 580, 365]
```

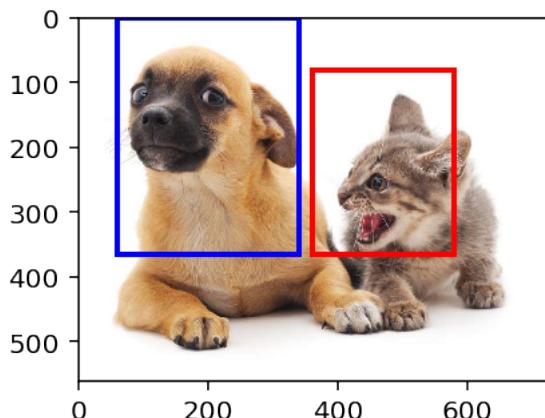
我们可以在图中将边框画出来检查其准确性。画之前我们定义一个辅助函数，它将边界框表示成

matplotlib 的边框格式，这个函数将保存在 GluonBook 里方便之后使用。

```
In [4]: # 将边界框 (左上 x, 左上 y, 右下 x, 右下 y) 格式转换成 matplotlib 格式:  
# ((左上 x, 左上 y), 宽, 高)。  
def bbox_to_rect(bbox, color):  
    return gb.plt.Rectangle(  
        xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],  
        fill=False, edgecolor=color, linewidth=2)
```

我们将边界框加载在图上，可以看到物体的主体基本在框内。

```
In [5]: fig = gb.pyplot.imshow(img)  
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))  
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
```



9.3.2 小结

- 在物体识别里我们不仅需要找出图片里面所有感兴趣的物体，而且要知道它们的位置。位置一般由方形边界框来表示。

9.3.3 练习

- 找一些图片，尝试标注下其中物体的边界框。比较下同图片分类标注所花时间的区别。

9.3.4 扫码直达讨论区



9.4 物体检测数据集

在物体检测领域并没有类似 MNIST 那样的小数据集方便我们快速测试模型。为此我们合成了一个小的人工数据集。我们首先使用一个开源的皮卡丘 3D 模型生成 1000 张不同角度和大小的图片。然后我们收集了一系列背景图片，并在每张图的随机位置放置一张皮卡丘图片。我们使用 MXNet 提供的 `tools/im2rec.py` 来将图片打包成二进制 rec 文件。（这是 MXNet 在 Gluon 开发出来之前常用的数据格式。注意到 GluonCV 这个包里已经提供了更简单的类似之前我们读取图片的方式，从而无需打包图片。但由于这个包目前仍在快速迭代中，所以这里我们使用 rec 格式。）

9.4.1 下载数据

打包好的数据可以直接在网上下载：

```
In [1]: %matplotlib inline
import sys
sys.path.insert(0, '..')
import gluonbook as gb
from mxnet import image, gluon

In [2]: root_url = ('https://apache-mxnet.s3-accelerate.amazonaws.com/'
                 'gluon/dataset/pikachu/')
data_dir = '../data/pikachu/'
dataset = {'train.rec': 'e6bcb6ffbalac04ff8a9b1115e650af56ee969c8',
           'train.idx': 'dcf7318b2602c06428b9988470c731621716c393',
           'val.rec': 'd6c33f799b4d058e82f2cb5bd9a976f69d72d520'}
for k, v in dataset.items():
    gluon.utils.download(root_url+k, data_dir+k, sha1_hash=v)
```

9.4.2 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)。在读取训练图片时我们做了随机剪裁。读取数据集的方法我们保存在 GluonBook 中的 `load_data_pikachu` 函数里。

```
In [3]: edge_size = 256 # 输出图片的宽和高。  
batch_size = 32  
  
train_data = image.ImageDetIter(  
    path_imgrec=data_dir+'train.rec',  
    path_imgidx=data_dir+'train.idx', # 每张图片在 rec 中的位置，使用随机顺序时需要。  
    batch_size=batch_size,  
    data_shape=(3, edge_size, edge_size), # 输出图片形状。  
    shuffle=True, # 用随机顺序访问。  
    rand_crop=1, # 一定使用随机剪裁。  
    min_object_covered=0.95, # 剪裁出的图片至少覆盖每个物体 95% 的区域。  
    max_attempts=200) # 最多尝试 200 次随机剪裁。如果失败则不进行剪裁。  
  
val_data = image.ImageDetIter( # 测试图片则去除了随机访问和随机剪裁。  
    path_imgrec=data_dir+'val.rec',  
    batch_size=batch_size,  
    data_shape=(3, edge_size, edge_size),  
    shuffle=False)
```

```
In [4]: image.ImageDetIter?
```

下面我们读取一个批量。

```
In [5]: batch = train_data.next()  
(batch.data[0].shape, batch.label[0].shape)  
  
Out[5]: ((32, 3, 256, 256), (32, 1, 5))
```

可以看到图片的形状跟之前图片分类时一样，但标签的形状是（批量大小，每张图片中最大边界框数，5）。每个边界框的由长为 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，仅做填充使用。后面 4 个元素表示边界框位置。这里使用的数据相对简单，每张图片只有一个边界框。一般使用的物体检测数据中每张图片可能会有多个边界框，但我们要求每张图片有相同数量的边界框使得可以放在一个批量里。所以我们会使用一个最大边界框数，对于不够的图片使用填充边界框。

9.4.3 图示数据

我们先定义一个函数可以画出多个边界框（以及他们的标注），它将被保存在 GluonBook 里以便后面使用。

```
In [6]: def show_bboxes(axes, bboxes, labels=None, colors=['b', 'g', 'r', 'm', 'k']):  
    for i, bbox in enumerate(bboxes):  
        color = colors[i%len(colors)]  
        rect = gb.bbox_to_rect(bbox.asnumpy(), color)  
        axes.add_patch(rect)  
    if labels and len(labels) > i:  
        axes.text(rect.xy[0], rect.xy[1], labels[i],  
                  va="center", ha="center", fontsize=9, color='white',  
                  bbox=dict(facecolor=color, lw=0))
```

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。当然，这是一个简单的人工数据集，物体和背景的区别较大。实际中遇到的数据集通常会复杂很多。

```
In [7]: imgs = (batch.data[0][0:10].transpose((0,2,3,1)) ).clip(0, 254)/254  
axes = gb.show_images(imgs, 2, 5).flatten()  
for ax, label in zip(axes, batch.label[0][0:10]):  
    show_bboxes(ax, [label[0][1:5]*edge_size], colors=['w'])
```



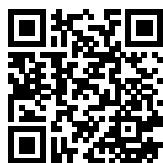
9.4.4 小结

- 物体识别的数据读取跟图片分类类似，但引入了边界框后导致标注形状和图片增强均有所不同。

9.4.5 练习

- 了解下 `image.ImageDetIter` 和 `image.CreateDetAugmenter` 这两个类的创建参数。

9.4.6 扫码直达讨论区



9.5 锚框

物体识别算法通常会在输入图片中采样大量的区域，然后判断这些区域是否有我们感兴趣的物体，以及进一步调整区域边缘来更准确预测物体的真实边界框。不同的模型使用不同的区域采样方法，这里我们介绍其中的一种：它以每个像素为中心生成数个大小和比例不同的被称之为锚框（anchor box）的边界框。

导入本小节需要的包。注意我们新引入了 `contrib` 这个模块，以及使用 `numpy` 修改了打印精度，这是因为 `NDArray` 的打印是通过调用 `numpy` 的打印函数。

```
In [1]: %matplotlib inline
import sys
sys.path.insert(0, '..')
import gluonbook as gb
from mxnet import image, gluon, nd, contrib
import numpy as np
np.set_printoptions(2)
```

9.5.1 锚框的生成

假设输入图片高为 h ，宽为 w ，那么大小为 $s \in (0, 1]$ 和比例为 $r > 0$ 的锚框形状是

$$\left(ws\sqrt{r}, \frac{hs}{\sqrt{r}}\right),$$

确定其中心点位置便可以固定一个锚框。

当然我们可以通过不同的 s 和 r , 以及中心位置, 来遍历所有可能的区域。虽然这样可以覆盖真实边界框, 但会使得计算很复杂。通常我们进行采样, 使得尽量很好的贴近真实边界框。例如我们可以首先固定一个比例 r_1 , 然后采样 n 个不同的大小 s_1, \dots, s_n 。然后固定一个大小 s_1 , 采样 m 个不同的比例 r_1, \dots, r_m 。这样对每个像素我们一共生成 $n + m - 1$ 个锚框。对于整个输入图片, 我们将一共生成 $wh(n + m - 1)$ 个锚框。

上述的采样方法实现在 `contrib.ndarray` 中的 `MultiBoxPrior` 函数。通过指定输入数据(我们只需要访问其形状), 锚框的采样大小和比例, 这个函数将返回所有采样到的锚框。

```
In [2]: img = image.imread('../img/catdog.jpg').asnumpy()
h, w = img.shape[0:2]
x = nd.random.uniform(shape=(1, 3, h, w)) # 构造一个输入数据,
y = contrib.nd.MultiBoxPrior(x, sizes=[.75, .5, .25], ratios=[1, 2, .5])
y.shape

Out[2]: (1, 2042040, 4)
```

其返回结果格式为 (批量大小, 锚框个数, 4)。可以看到我们生产了 2 百万以上个锚框。将变形为 (高, 宽, $n + m - 1$, 4) 后, 我们可以方便的访问以任何一个像素为中心的所有锚框。下面例子里我们访问以 (250, 250) 为第一个锚框。它有四个元素, 同前一样是左上和右下的 x、y 轴坐标, 但被分别除以了高和宽使得数值在 0 和 1 之间。

```
In [3]: boxes = y.reshape((h, w, 5, 4))
boxes[250, 250, 0, :]

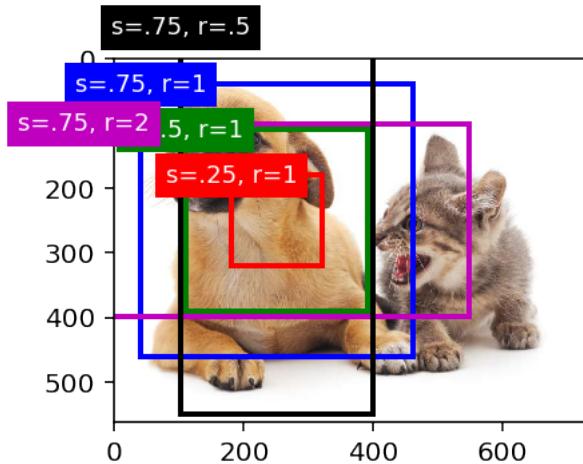
Out[3]:
[ 0.06  0.07  0.63  0.82]
<NDArray 4 @cpu(0)>
```

在画出这些锚框的具体样子前我们先定义一个函数来图上画出多个边界框, 它将被保存在 `GluonBook` 里以便后面使用。

```
In [4]: def show_bboxes(axes, bboxes, labels=None):
    colors = ['b', 'g', 'r', 'k', 'm']
    for i, bbox in enumerate(bboxes):
        color = colors[i%len(colors)]
        rect = bbox_to_rect(bbox.asnumpy(), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            axes.text(rect.xy[0], rect.xy[1], labels[i],
                      va="center", ha="center", fontsize=9, color='white',
                      bbox=dict(facecolor=color, lw=0))
```

然后我们画出以 (200, 200) 为中心的所有锚框。

```
In [5]: bbox_scale = np.array((w, h, w, h)) # 需要乘以高和宽使得符合我们的画图格式。  
fig = plt.imshow(img)  
gb.show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale, [  
    's=.75, r=1', 's=.5, r=1', 's=.25, r=1', 's=.75, r=2', 's=.75, r=.5'])
```



可以看到大小为 0.75 比例为 2 的洋红色锚框比较好的覆盖了图片中的小狗。

9.5.2 IoU：交集除并集

在介绍如何使用锚框参与训练和预测前，我们先介绍如何判断两个边界框的距离。我们知道集合相似度的最常用衡量标准叫做 Jaccard 距离。给定集合 A 和 B ，它的定义是集合的交集除以集合的并集：

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

边界框指定了一块像素区域，其可以看成是像素点的集合。因此我们可以定类似的距离，即我们使用两个边界框的相交面积除以相并面积来衡量它们的相似度。这被称之为交集除并集（Intersection over Union，简称 IoU）。它的取值范围在 0 和 1 之间。0 表示边界框不相关，1 则表示完全一样。

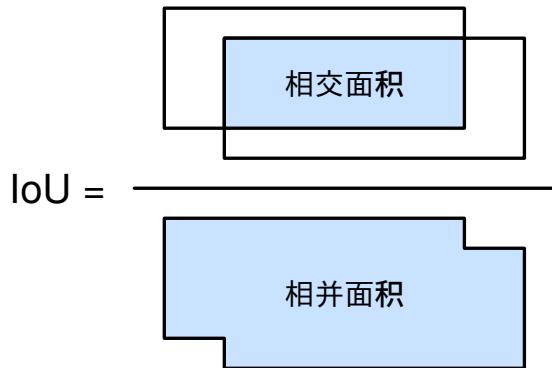


图 9.2: 交集除并集。

9.5.3 训练

在训练时，每个锚框都表示成一个样本。对每个样本我们需要预测它是否含有我们感兴趣的物体，以及如果是那么预测它的真实边界框。在训练前我们首先需要为每个锚框生产标签。这里标签有两类，第一类是对应的物体的标号。一个常用的构造办法是对每个真实边界框，我们选取一个或多个与其相似的锚框赋予它们这个真实边界框里的物体标号。具体来说，对一个训练数据中提供的真实边界框，假设其对应物体标号 i ，我们选取所有与其 IoU 大于某个阈值（例如 0.5）的锚框。如果没有这样的锚框，我们就选取 IoU 值最大的那个。然后将选中的锚框的物体标号设成 $i + 1$ 。如果一个锚框没有被任何真实边界框选中，即不与任何训练数据中的物体足够重合，那么将赋予标号 0，代表只含有背景。我们经常将这类锚框叫做负类锚框，其余的则称之为正类。

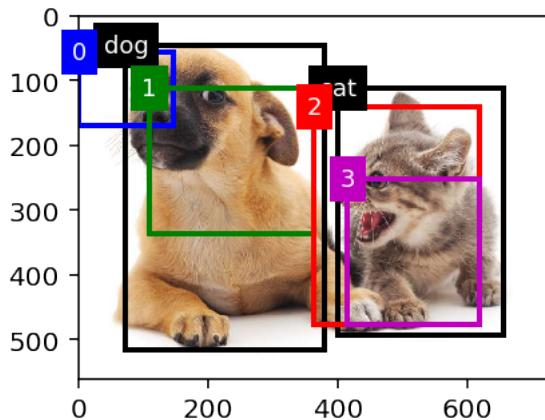
对于正类锚框，我们还需要构造第二类标号，即它们与真实边界框的距离。一个简单的方法是它与真实边界框的坐标差。但因为有图片边框的限制，这些差值都在 -1 与 1 之间，这使得在模型预测时变得复杂。通常我们会将其进行非线性变化来使得其数值上更加均匀来方便模型预测。

下面来看一个具体的例子。我们将读取的图片中的猫和狗边界框定义成真实边界框，其中第一个元素为类别号（从 0 开始）。然后我们构造四个锚框，其与真实边界框的位置如下图示。

```
In [6]: ground_truth = nd.array([[0, .1, .08, .52, .92], [1, .55, .2, .9, .88]])
anchors = nd.array([[0, .1, .2, .3], [.15, .2, .5, .6],
[.5, .25, .85, .85], [.57, .45, .85, .85]])
```

```
fig = gb.plt.imshow(img)
```

```
gb.show_bboxes(fig.axes, ground_truth[:,1:]*bbox_scale, ['dog','cat'], 'k');
gb.show_bboxes(fig.axes, anchors*bbox_scale, ['0', '1', '2', '3'] );
```



我们可以通过 contrib.nd 模块中的 MultiBoxTarget 函数来对锚框生产标号。我们对对锚框和真实边界框加上批量维，和构造一个任意的锚框预测结果，其形状为（批量大小，类别数 +1，锚框数），其中第 0 类为背景。

```
In [7]: out = contrib.nd.MultiBoxTarget(anchors.expand_dims(axis=0),
                                         ground_truth.expand_dims(axis=0),
                                         nd.zeros((1,3,4)))
```

返回的结果里有三个 NDArray。首先看第三个值，其表示赋予锚框的标号。

```
In [8]: out[2]
```

Out[8]:

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

我们可以如下分析每个锚框被赋予的标号的理由：

1. 锚框 0 里被认定只有背景，因为它与所有真实边界框的 IoU 都小于 0.5。
2. 锚框 1 里被认定为有猫，虽然它与猫的边界框的 IoU 小于 0.5，但是它是这四个锚框里离猫的边界框最近的那个。
3. 锚框 2 里被认定为有狗，因为它与狗的边界框的 IoU 大于 0.5。
4. 锚框 3 里被认定只有背景，虽然它与狗的边界框的 IoU 类似于锚框 1 与猫的边界框的 IoU，但其小于 0.5，且锚框 2 已经获得了狗的标号。

返回值的中间值用来遮掩不需要的负类锚框，其形状为（批量大小，锚框数 \times 4）。其中正类锚框对应的元素为 1，负类为 0。

```
In [9]: out[1]
```

```
Out[9]:
```

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

<NDArray 1x16 @cpu(0)>
```

返回的第一个值是锚框与真实边界框的偏移，只有正类锚框有非 0 值。

```
In [10]: out[0]
```

```
Out[10]:
```

```
[[ 0.00e+00  0.00e+00  0.00e+00  0.00e+00 -4.29e-01  2.50e+00
  9.12e-01  3.71e+00  1.43e+00 -1.67e-01 -8.94e-07  6.26e-01
  0.00e+00  0.00e+00  0.00e+00  0.00e+00 ]]

<NDArray 1x16 @cpu(0)>
```

9.5.4 预测

预测同训练类似的对每个锚框预测其包含的物体类别和与真实边界框的位移。因为我们会生成大量的锚框，所以可能导致对同一个物体会产生大量相似的预测边界框。为了使得结果更加简介，我们需要消除相似的冗余预测值。这里常用的方法是非最大抑制（Non-Maximum Suppression，简称 NMS）。对于相似的预测边界框，NMS 只保留物体标号预测置信度最高的那个。

具体来说，对于每个物体类别（非背景），我们先获取每个预测边界框里被判断包含这个类别物体的概率。然后我们找到概率最大的那个边界框，如果其置信度大于某个阈值，那么保留它到输出。接下来移除掉抑制其它所有的跟这个边界框的 IoU 大于某个阈值的边界框。在剩下的边界框里我们再找出预测概率最大的边界框，重复前面的移除过程。直到我们要么保留或者移除了每个边界框。

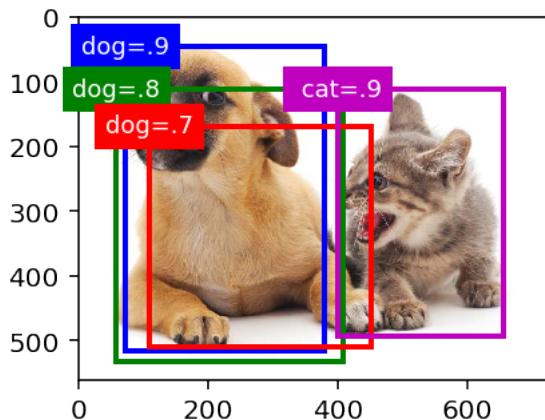
下面来看一个具体的例子。我们先构造四个锚框，为了简单期间我们假设预测偏移全是 0，然后构造了类别预测。

```
In [11]: anchors = nd.array([[.1, .08, .52, .92], [.08, .2, .56, .95],
                           [.15, .3, .62, .91], [.55, .2, .9, .88]])

loc_preds = nd.array([.0]*anchors.size)
cls_probs = nd.array([[0]*4, # 是背景的概率。
                     [.9, .8, .7, .1], # 是狗的概率。
                     [.1, .2, .3, .9]]) # 是猫的概率。
```

在实际图片上查看预测边界框的位置和预测置信度：

```
In [12]: fig = gb.plt.imshow(img)
gb.show_bboxes(fig.axes, anchors * bbox_scale,
['dog=.9', 'dog=.8', 'dog=.7', 'cat=.9'])
```



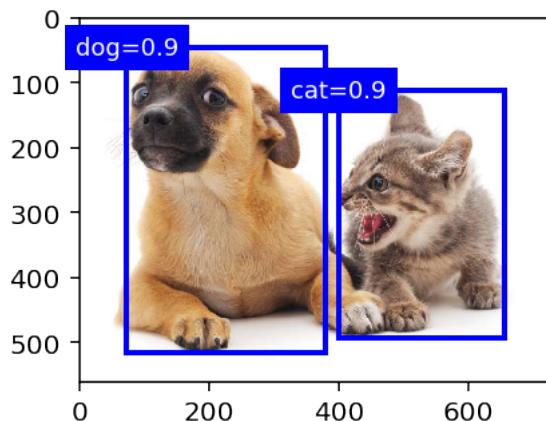
我们使用 contrib.nd 模块的 MultiBoxDetection 函数来执行 NMS，这里为 NDArray 输入都增加了批量维。

```
In [13]: ret = contrib.ndarray.MultiBoxDetection(
    cls_probs.expand_dims(axis=0), loc_preds.expand_dims(axis=0),
    anchors.expand_dims(axis=0), nms_threshold=.5)
ret
```

```
Out[13]:
[[[ 0.      0.9     0.1     0.08   0.52   0.92]
  [ 1.      0.9     0.55    0.2     0.9    0.88]
  [-1.      0.8     0.08    0.2     0.56   0.95]
  [-1.      0.7     0.15    0.3     0.62   0.91]]]
<NDArray 1x4x6 @cpu(0)>
```

其返回格式为（批量大小， 锚框个数， 6）。每一行对应一个预测边界框， 其有六个元素， 依次为预测类别（移除了背景类， 其中-1 表示被该边界框被移除了）、预测物体属于此类的概率和预测边界框。我们移除掉-1 类的结果来可视化 NMS 保留的结果。

```
In [14]: fig = gb.plt.imshow(img)
for i in ret[0].asnumpy():
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    gb.show_bboxes(fig.axes, [nd.array(i[2:])*bbox_scale], label)
```



9.5.5 小结

- 以每个像素为中心我们生产多个大小比例不同的锚框来预测真实边界框。
- 训练时我们根据真实边界框来为每个锚框赋予类别标号和偏移这两类标签。
- 预测时我们移除重合度很高的预测值来保持结果简洁。

9.5.6 练习

- 改变锚框生成里面的大小和比例采样来看看可视化时的区别。
- 构造 IoU 是 0.5 的两个边界框，看看视觉上他们的重合度。
- 修改训练和预测里的 anchors 来看他们对结果的影响。

9.5.7 扫码直达讨论区



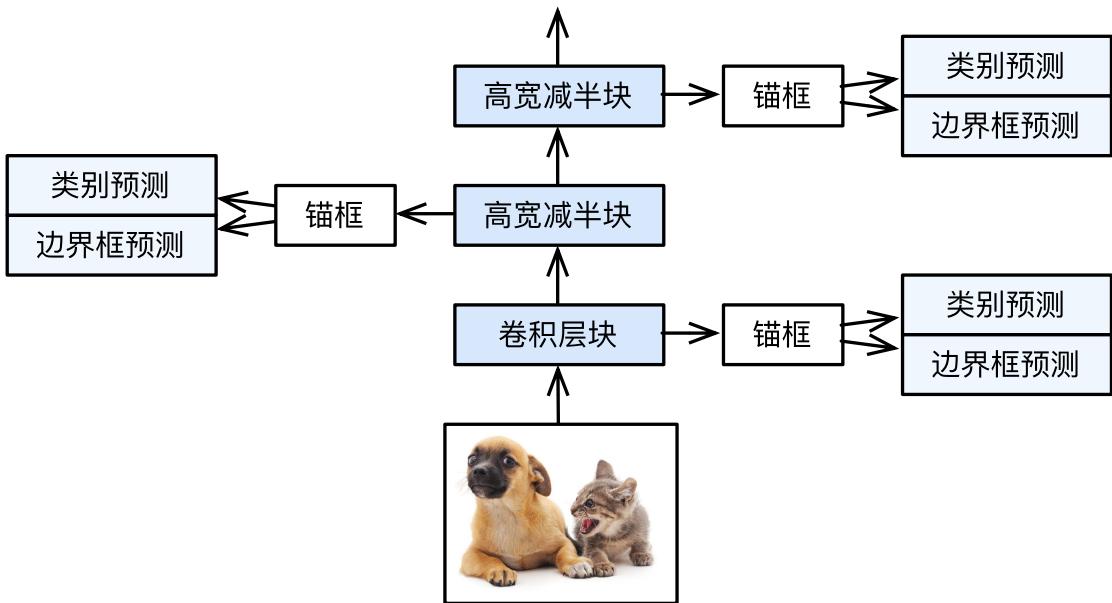
9.6 单发多框检测（SSD）

我们将介绍的第一个模型是单发多框检测（single shot multibox detection，简称 SSD）[1]。它并不是第一个提出来的基于深度学习的物体检测模型，也不是精度最高的，但因为其简单性而被大量使用。我们将使用 SSD 来详解目标检测的实现细节。

```
In [1]: %matplotlib inline
import sys
sys.path.insert(0, '...')
import time
import gluonbook as gb
from mxnet import gluon, image, nd, contrib, init, autograd
from mxnet.gluon import nn
```

9.6.1 SSD 模型

SSD 模型的示意图如下。给定输入图片，其首先使用主要由卷积层组成的模块来进行特征抽取。在其输出上，我们以每个像素为中心构建多个锚框（往左的箭头），然后使用 softmax 来对每个锚框判断其包含的物体类别，以及预测它到真实物体边界框的距离。卷积层的输出同时被输入到一个高宽减半模块（往上的箭头）来缩小图片尺寸。这个模块的输入将进入到另一个卷积模块抽取特征，和构建锚框来预测物体类别和边界框。这样设计的目的是在不同的尺寸上进行目标检测，例如之前的锚框主要检测图片中尺寸较小的物体，而这里我们则检测尺寸稍大的物体。自然的，我们会重复这一过程多次来在多种不同的尺寸下检测物体。



接下来我们介绍并实现图中各个模块。首先注意到这里锚框是由上小节介绍的方法生产而来。接下来介绍类别预测和边界框预测。

类别预测

假设数据集中有 n 种不同的物体，那么我们将对每个锚框做 $n + 1$ 类分类，其中类 0 表示锚框只包含背景。因为我们对输入像素为中心生产多个锚框，记为 a ，我们将会预测 hwa 个锚框的分类结果，这里 h 和 w 分别是输入高和宽。如果使用正常的全连接层作为输出，可能会导致有过多的模型参数。回忆“[网络中的网络：NiN](#)”这一节里我们介绍了使用卷基层的通道来输出类别预测，SSD 采用同样的方法来降低模型复杂度。

具体来说，类别预测层使用一个保持输入高宽的卷基层，其输出的 (x, y) 像素通道里包含了以输入 (x, y) 像素为中心的所有锚框的类别预测。其输出通道数为 $a(n + 1)$ ，其中通道 $i(n + 1)$ 是第 i 个锚框只含有背景的置信度，而通道 $i(n + 1) + j + 1$ 则是第 i 锚框含有第 j 类物体的置信度。

下面我们定义一个这样的类别分类器，指定 a 和 n 后，它使用一个填充为 1 的 3 乘 3 卷积层。注意到我们使用了固定的卷积窗口，它可能不能覆盖锚框定义的区域。所以我们需要前面的卷积层能有效的将较大的能覆盖锚框的区域的特征浓缩到一个 3 乘 3 窗口里。

```
In [2]: def cls_predictor(num_anchors, num_classes):
        return nn.Conv2D(num_anchors * (num_classes+1), kernel_size=3, padding=1)
```

边界框预测

对每个锚框我们需要预测如何将其变换到真实的物体边界框。变换由一个长为 4 的向量来描述，表示左下和右上的 x、y 轴坐标偏移。我们同样使用一个保持高宽的卷积层来输出偏移预测，它有 $4a$ 个输出通道，对于第 i 个锚框，它的偏移预测在 $4i$ 到 $4i + 3$ 这 4 个通道里。

```
In [3]: def bbox_predictor(num_anchors):
    return nn.Conv2D(num_anchors * 4, kernel_size=3, padding=1)
```

合并多层的预测输出

SSD 中会在多个尺度上进行预测。由于每个尺度上的输入高宽和锚框的选取不一样，导致其形状各不一样。下面例子中我们构造两个尺度的输入，其中第二个将第一个的高宽减半。然后构造两个类别预测层，其分别对每个输入像素构造 5 和 3 个锚框。

```
In [4]: def forward(x, block):
    block.initialize()
    return block(x)

y1 = forward(nd.zeros((2, 8, 20, 20)), cls_predictor(5, 10))
y2 = forward(nd.zeros((2, 16, 10, 10)), cls_predictor(3, 10))
(y1.shape, y2.shape)

Out[4]: ((2, 55, 20, 20), (2, 33, 10, 10))
```

预测的输出格式为（批量大小，通道数，高，宽）。可以看到除了批量大小外，其他维度大小均不一样。我们需要将它们变形为统一的格式并将多尺度的输出合并起来，让后续的处理变得简单。

我们首先将通道，即预测结果，放到最后。因为不同尺度下批量大小保持不变，所以将结果转成二维的（批量大小，高 \times 宽 \times 通道数）格式，方便之后的拼接。

```
In [5]: def flatten_pred(pred):
    return pred.transpose(axes=(0, 2, 3, 1)).flatten()
```

拼接就是简单将在维度 1 上合并结果。

```
In [6]: def concat_preds(preds):
    return nd.concat(*[flatten_pred(p) for p in preds], dim=1)
```

可以看到 y_1 和 y_2 形状不同。为了之后处理简单，我们将不同层的输入合并成一个输出。首先我们将通道移到最后的维度，然后将其展成 2D 数组。因为第一个维度是样本个数，所以不同输出之间是不变，我们可以将所有输出在第二个维度上拼接起来。

```
In [7]: concat_preds([y1, y2]).shape
```

```
Out[7]: (2, 25300)
```

减半模块

减半模块将输入高宽减半来得到不同尺度的特征，这是通过步幅 2 的 2 乘 2 最大池化层来完成。我们前面提到因为预测层的窗口为 3，所以我们需要额外卷积层来扩大其作用窗口来有效覆盖锚框区域。为此我们加入两个 3 乘 3 卷基层，每个后接批量归一化层和 ReLU 激活层。这样，一个尺度上的 3 乘 3 窗口覆盖了上一个尺度上的 10 乘 10 窗口。

```
In [8]: def down_sample_blk(num_filters):
    blk = nn.HybridSequential()
    for _ in range(2):
        blk.add(nn.Conv2D(num_filters, kernel_size=3, padding=1),
               nn.BatchNorm(in_channels=num_filters),
               nn.Activation('relu'))
    blk.add(nn.MaxPool2D(2))
    blk.hybridize()
    return blk
```

可以看到，它将改变了输入的通道数，并将高宽减半。

```
In [9]: forward(nd.zeros((2, 3, 20, 20)), down_sample_blk(10)).shape
Out[9]: (2, 10, 10, 10)
```

主体网络

主体网络用来从原始像素抽取特征，通常使用常用的深度卷积神经网络。例如 [1] 中使用了 VGG，之后的工作大家也常用 ResNet。本小节为了计算简单性，我们构造一个小的主体网络。其叠加三个减半模块，输出通道数从 16 开始每个模块对其翻倍。

```
In [10]: def body_blk():
    blk = nn.HybridSequential()
    for num_filters in [16, 32, 64]:
        blk.add(down_sample_blk(num_filters))
    return blk

    forward(nd.zeros((2, 3, 256, 256)), body_blk()).shape
Out[10]: (2, 64, 32, 32)
```

完整的模型

我们已经介绍了 SSD 模型中的各个模块，现在我们将构建整个模型。这个模型有五个模块，每个模块对输入进行特征抽取，并且预测锚框的类和偏移。第一个模块使用主体网络，第二到四模块使用减半模块，最后一个模块则使用全局的最大池化层来将高宽降到 1。下面函数定义如何如何构建这些模块。

```
In [11]: def get_blk(i):
    if i == 0:
        blk = body_blk()
    elif i == 4:
        blk = nn.GlobalMaxPool2D()
    else:
        blk = down_sample_blk(128)
    return blk
```

接下来定义每个模块如何进行前向计算。它跟之前的卷积神经网络不同在于，我们不仅输出卷积块的输出，而且我们还返回在输出上生产的锚框，以及每个锚框的类别预测和偏移预测。

```
In [12]: def single_scale_forward(x, blk, size, ratio, cls_predictor, bbox_predictor):
    y = blk(x)
    anchor = contrib.ndarray.MultiBoxPrior(y, sizes=size, ratios=ratio)
    cls_pred = cls_predictor(y)
    bbox_pred = bbox_predictor(y)
    return (y, anchor, cls_pred, bbox_pred)
```

对每个模块我们要定义其输出上的锚框如何生成。比例固定成 1、2 和 0.5。但大小上则不通荣。

```
In [13]: num_anchors = 4
sizes = [[.2, .272], [.37, .447], [.54, .619], [.71, .79], [.88, .961]]
ratios = [[1, 2, .5]] * 5
```

完整的模型定义如下。

```
In [14]: class TinySSD(gluon.Block):
    def __init__(self, num_classes, verbose=False, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        for i in range(5):
            setattr(self, 'blk_%d'%i, get_blk(i))
            setattr(self, 'cls_%d'%i, cls_predictor(num_anchors, num_classes))
            setattr(self, 'bbox_%d'%i, bbox_predictor(num_anchors))

    def forward(self, x):
        anchors, cls_preds, bbox_preds = [None]*5, [None]*5, [None]*5
```

```

for i in range(5):
    x, anchors[i], cls_preds[i], bbox_preds[i] = single_scale_forward(
        x, getattr(self, 'blk_%d'%i), sizes[i], ratios[i],
        getattr(self, 'cls_%d'%i), getattr(self, 'bbox_%d'%i))
return (nd.concat(*anchors, dim=1),
        concat_preds(cls_preds).reshape((0, -1, self.num_classes+1)),
        #.transpose(axes=(0,2,1)),
        concat_preds(bbox_preds))

net = TinySSD(num_classes=2, verbose=True)
net.initialize()
x = nd.zeros((2,3,256,256))
anchors, cls_preds, bbox_preds = net(x)

print('Output achors:', anchors.shape)
print('Output class predictions:', cls_preds.shape)
print('Output box predictions:', bbox_preds.shape)

Output achors: (1, 5444, 4)
Output class predictions: (2, 5444, 3)
Output box predictions: (2, 21776)

```

9.6.2 训练

读取数据和初始化训练

我们使用之前构造的皮卡丘数据集。

```
In [15]: batch_size = 32
train_data, test_data = gb.load_data_pikachu(batch_size)
# GPU 实现里要求每张图片至少有三个边界框，我们加上两个标号为 -1 的边界框。
train_data.reshape(label_shape=(3, 5))
```

模型和训练器的初始化跟之前类似。

```
In [16]: ctx = gb.try_gpu()
net = TinySSD(num_classes = 2)
net.initialize(init=init.Xavier(), ctx=ctx)
trainer = gluon.Trainer(net.collect_params(),
                        'sgd', {'learning_rate': 0.1, 'wd': 5e-4})
```

损失和评估函数

物体识别有两个损失函数，一是对每个锚框的类别预测，可以重用之前图片分类问题里一直使用的 Softmax 和交叉熵损失。二是正类锚框的偏移预测。它是一个回归问题，但我们这里不使用前面介绍过的 L2 损失函数，而是使用对误差较大的损失更小的 L1 损失函数，即 $l_1(\hat{y}, y) = |\hat{y} - y|$ 。

```
In [17]: cls_loss = gluon.loss.SoftmaxCrossEntropyLoss()
bbox_loss = gluon.loss.L1Loss()

In [18]: def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    cls = cls_loss(cls_preds, cls_labels)
    bbox = bbox_loss(bbox_preds * bbox_masks, bbox_labels * bbox_masks)
    return cls + bbox
```

对于分类好坏我们可以沿用之前的分类精度。因为使用了 L1 损失，我们评估边框预测的用平均绝对误差。

```
In [19]: def cls_metric(cls_preds, cls_labels):
    # 注意这里类别预测结果放在最后一维, argmax 的时候指定使用最后一维。
    return (cls_preds.argmax(axis=-1) == cls_labels).mean().asscalar()

def bbox_metric(bbox_preds, bbox_labels, bbox_masks):
    return (bbox_labels - bbox_preds * bbox_masks).abs().mean().asscalar()
```

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。为了代码简单起见我们没有评估测试数据集。

```
In [20]: for epoch in range(20):
    acc, mae = 0, 0
    train_data.reset() # 从头读取数据。
    tic = time.time()
    for i, batch in enumerate(train_data):
        # 复制数据到 GPU。
        X = batch.data[0].as_in_context(ctx)
        Y = batch.label[0].as_in_context(ctx)
        with autograd.record():
            # 对每个锚框预测输出。
            anchors, cls_preds, bbox_preds = net(X)
            # 对每个锚框生成标号。
            bbox_labels, bbox_masks, cls_labels = contrib.nd.MultiBoxTarget(
```

```

        anchors, Y, cls_preds.transpose(axes=(0,2,1)))
    # 计算类别预测和边界框预测损失。
    loss = calc_loss(cls_preds, cls_labels,
                      bbox_preds, bbox_labels, bbox_masks)
    # 计算梯度和更新模型。
    loss.backward()
    trainer.step(batch_size)
    # 更新类别预测和边界框预测评估。
    acc += cls_metric(cls_preds, cls_labels)
    mae += bbox_metric(bbox_preds, bbox_labels, bbox_masks)
    if (epoch+1) % 5 == 0:
        print('epoch %2d, class err %.2e, bbox mae %.2e, time %.1f sec' % (
            epoch+1, 1-acc/(i+1), mae/(i+1), time.time()-tic))

epoch 5, class err 3.04e-03, bbox mae 3.38e-03, time 13.4 sec
epoch 10, class err 2.68e-03, bbox mae 2.99e-03, time 13.4 sec
epoch 15, class err 2.56e-03, bbox mae 2.85e-03, time 13.4 sec
epoch 20, class err 2.44e-03, bbox mae 2.73e-03, time 13.4 sec

```

9.6.3 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。我们首先定义一个图片预处理函数，它对图片进行变现然后转成卷积层需要的四维格式。

```
In [21]: def process_image(file_name):
    img = image.imread(file_name)
    data = image.imresize(img, 256, 256).astype('float32')
    return data.transpose((2,0,1)).expand_dims(axis=0), img

x, img = process_image('../img/pikachu.jpg')
```

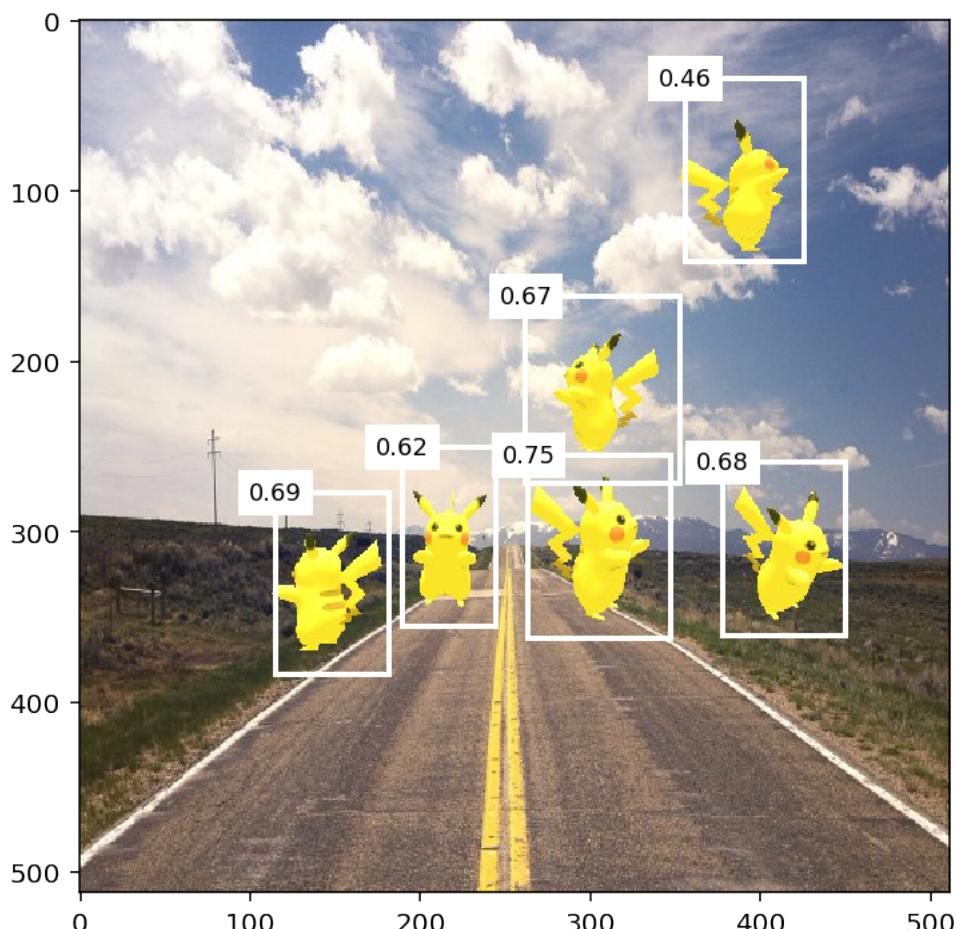
在预测的时候，我们通过 `MultiBoxDetection` 函数来将合并预测偏移和锚框得到预测边界框，并使用 NMS 去除重复的预测边界框。

```
In [22]: def predict(x):
    anchors, cls_preds, bbox_preds = net(x.as_in_context(ctx))
    cls_probs = cls_preds.softmax().transpose((0,2,1))
    out = contrib.nd.MultiBoxDetection(cls_probs, bbox_preds, anchors)
    idx = [i for i, row in enumerate(out[0]) if row[0].asscalar() != -1]
    return out[0], idx

out = predict(x)
```

最后我们将预测出置信度超过某个阈值的边框画出来：

```
In [23]: gb=plt.rcParams['figure.figsize'] = (6, 6)
def display(img, out, threshold=0.5):
    fig = gb.pyplot.imshow(img.asnumpy())
    for row in out:
        score = row[1].asscalar()
        if score < threshold:
            continue
        bbox = [row[2:6] * np.array(img.shape[0:2]*2, ctx=row.context)]
        gb.show_bboxes(fig.axes, bbox, '%.2f'%score, 'w')
display(img, out, threshold=0.4)
```



9.6.4 小结

- SSD 多尺度上对每个锚框预测类别与真实边界框的位移来进行物体检测。

9.6.5 练习

- 限于篇幅原因我们忽略了 SSD 实现的许多细节。我们选取其中数个作为练习。

损失函数

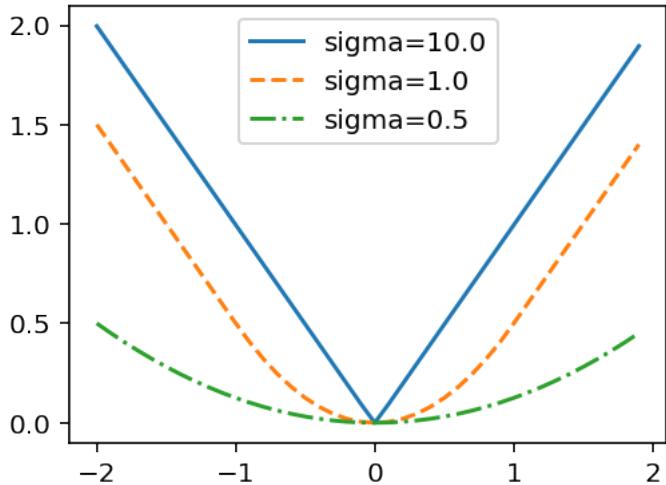
边界框预测时使用了 L_1 损失，但这个函数在 0 点处导数不唯一，因此可能会影响收敛。一个常用改进是在 0 点附近使用平方函数使得它更加平滑。它被称之为平滑 L_1 损失函数。它通过一个参数 σ 来控制平滑的区域：

$$f(x) = \begin{cases} (\sigma x)^2 / 2, & \text{if } x < 1/\sigma^2 \\ |x| - 0.5/\sigma^2, & \text{otherwise} \end{cases}$$

当 σ 很大时它类似于 L_1 损失，变小则函数更加平滑。

```
In [24]: sigmas = [10, 1, .5]
lines = ['-', '--', '-.']
x = nd.arange(-2, 2, 0.1)

gb.set_figsize((4,3))
for l,s in zip(lines, sigmas):
    y = nd.smooth_l1(x, scalar=s)
    gb.plt.plot(x.asnumpy(), y.asnumpy(), l, label='sigma=%f'%s)
gb=plt.legend();
```



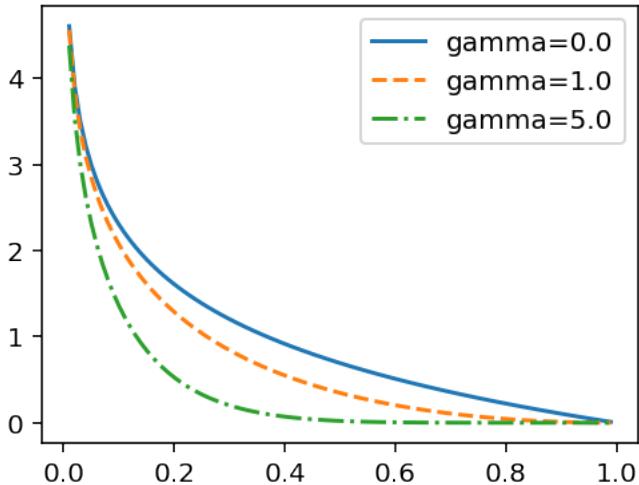
对于类别预测我们使用了交叉熵损失。假设对真实类别 j 的概率预测是 p_j , 交叉熵损失为 $\log(p_j)$ 。我们可以使用一个称之为关注损失 (focal loss) 的函数来对之稍微变形。给定正的 γ 和 α , 它的定义是

$$-\alpha(1 - p_j)^\gamma \log(p_j)$$

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 [25]: def focal_loss(gamma, x):
    return - (1-x)**gamma*x.log()

x = nd.arange(0.01, 1, .01)
for l, gamma in zip(lines, [0,1,5]):
    y = gb.plt.plot(x.asnumpy(), focal_loss(gamma, x).asnumpy(), l,
                     label='gamma=%1f'%gamma)
gb.plt.legend();
```



训练和预测

- 当物体在图片中占比很小时，我们通常会使用比较大的输入图片尺寸。
- 尝试分析不同尺寸上锚框的大小和比例是如何选取的。
- 对锚框赋予标号时，通常会有大量的负类锚框。我们可以对负例采样来使得分类时数据更加平衡。这个可以通过设置 `MultiBoxTarget` 的参数来完成。
- 分类和回归损失我们直接加起来了，并没有给予各自权重。
- 训练中我们没有实现验证数据集的评估。
- 物体检测算法好坏通常用 `mAP` (mean Average Precision) 来评估，查找它的定义。
- 在展示的时候如何选取阈值，特别是在修改训练算法时（例如增加迭代周期）。

9.6.6 扫码直达讨论区



9.7 区域卷积神经网络（R-CNN）系列

区域卷积神经网络（Regions with CNN features, 简称 R-CNN）[1] 是使用深度模型来解决物体识别的开创性工作，这一小节我们将介绍它和它之后数个重要变种。但限于篇幅原因，这里主要介绍模型思路而不是具体实现。

9.7.1 R-CNN：区域卷积神经网络

R-CNN 的提出影响了后面一系列深度模型的设计。它首先对每张图片选取多个提议区域（例如之前介绍的锚框就是一种选取方法），然后使用卷积层来对每个区域抽取特征，这样我们得到多个区域样本。之后我们对每个区域样本来进行物体分类和真实边界框预测。R-CNN 模型由下图演示。

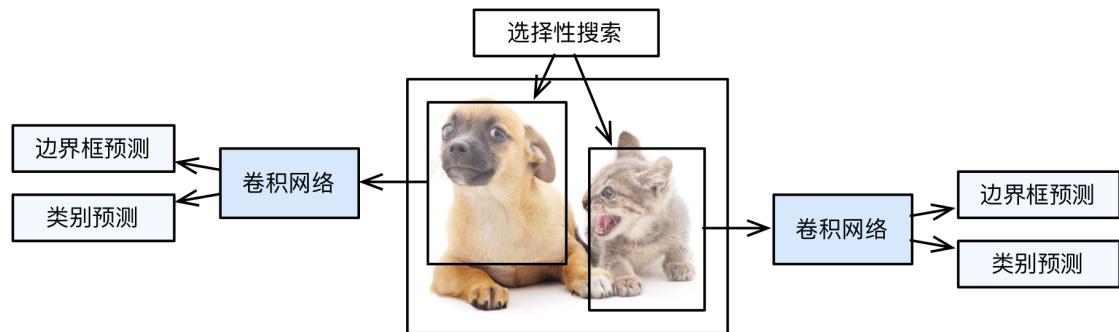


图 9.3: R-CNN 模型。

具体来说，它的由四步构成：

1. 对每张输入图片使用选择性搜索 [2] 来选取多个高质量的提议区域。这个算法先对图片基于像素信息做快速分割来得到多个区域，然后将当下最相似的两区域合并成一个区域，重复进行合并直到整张图片变成一个区域。最后根据合并的信息生成多个有层次结构的提议区域。为每个提议区域生成物体类别和真实边界框。
2. 选取一个预先训练好的卷积神经网络，去掉最后的输出层来作为特征抽取模块。对每个提议区域，将其变形成卷积神经网络需要的输入尺寸后进行前向计算抽取特征。
3. 将每个提议区域的特征连同其标注做成一个样本，训练多个支持向量机（SVM）来进行物体类别分类，这里第 i 个 SVM 预测样本是否属于第 i 类。
4. 在这些样本上训练一个线性回归模型来预测真实边界框。

R-CNN 对之前物体识别算法的主要改进是使用了预先训练好的卷积神经网络来抽取特征，其有效的提升了识别精度。但 R-CNN 的一个主要缺点在于性能。对一张图片我们可能选出上千个兴趣区域，这样导致每张图片需要对卷积网络做上千次的前向计算。当然在训练的时候我们可以事先算好每个区域的特征并保存，因为训练中不更新卷积网络的权重。但在做预测时，我们仍然需要计算上千次的前向计算，其带来的巨大计算量使得很难在实际应用中被使用。

9.7.2 Fast R-CNN：快速的区域卷积神经网络

R-CNN 的主要性能瓶颈在于需要对每个提议区域独立的抽取特征。考虑到这些区域很多会有大量重叠，独立的特征抽取导致了大量的重复计算。Fast R-CNN [3] 对 R-CNN 的一个主要改进在于首先对整个图片进行特征抽取，然后再选取提议区域，从而减少重复计算。

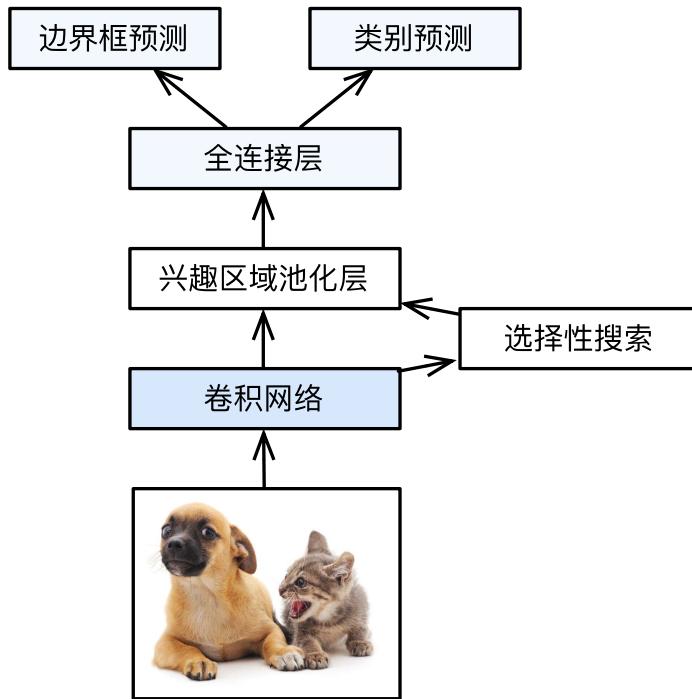


图 9.4: Fast R-CNN 模型。

Fast R-CNN 跟 R-CNN 的主要不同在于下面四点：

1. 用来提取特征的卷积网络是作用在整个图片上，而不是各个提议区域上。而且这个卷积网络通常会参与训练，即更新权重。
2. 选择性搜索是作用在卷积网络的输出上，而不是原始图片上。
3. 在 R-CNN 里，我们将形状各异的提议区域变形到同样的形状来进行特征提取。Fast R-CNN 则新引入了兴趣区域池化层（Region of Interest Pooling，简称 RoI 池化层）来对每个提议区域提取同样大小的输出以便进入之后的神经层。
4. 在物体分类时，Faster R-CNN 不再使用多个 SVM，而是跟之前图片分类那样使用 Softmax 回归来进行多类预测。

Fast R-CNN 中提出的 RoI 池化层跟我们之前介绍过的池化层有显著的不同。在池化层中，我们通过设置池化窗口、填充和步幅来控制输出大小，而 RoI 池化层里我们直接设置每个区域的输出大小。例如设置 $n \times m$ ，那么对每一个区域我们得到 $n \times m$ 形状输出。具体来说，我们将每个区

域在高和宽上分别均匀划分 n 和 m 块，如果划分边界不是整数则定点化到最近的整数。然后对于每一个划分区域，我们输出其的最大元素值。

下图中，我们在 4×4 的输入上选取了左上角的 3×3 区域作为兴趣区域，经过 2×2 的 ROI 池化层后得到一个 2×2 的输出，其中每个输出元素需要的输入均由同色标注。

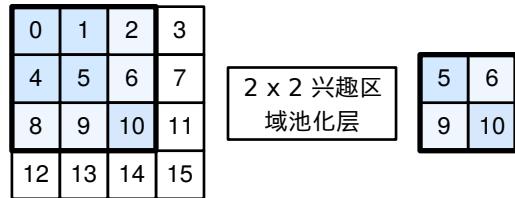


图 9.5: 2×2 ROI 池化层。

我们使用 `nd.ROIPooling` 来演示实际计算。假设输入特征高宽均为 4 且只有单通道。

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)>

我们定义两个兴趣区域，每个区域由五个元素表示，分别为区域物体标号，左上角的 x、y 轴坐标和右下角的 x、y 轴坐标。

In [2]: `rois = nd.array([[0,0,0,2,2], [0,0,1,3,3]])`

可以看到这里我们生成了 3×3 和 4×3 大小的两个区域。

ROI 池化层的输出形状是 (区域个数, 输入通道数, n , m)，其一般被当做样本数是区域个数的批量进入到接下来的神经层中。输入中我们指定了输入特征、池化形状、和当前特征尺寸与原始图片尺寸的比例。

In [3]: `nd.ROIPooling(x, rois, pooled_size=(2, 2), spatial_scale=1)`

Out[3]:

```
[[[ 5.   6.]
   [ 9.  10.]]]
```

```
[[[ 9. 11.]  
[ 13. 15.]]]]  
<NDArray 2x1x2x2 @cpu(0)>
```

9.7.3 Faster R-CNN：更快速的区域卷积神经网络

Faster R-CNN [4] 对 Fast R-CNN 做了进一步改进，它将 Fast R-CNN 中的选择性搜索替换成区域提议网络（region proposal network，简称 RPN）。RPN 以锚框为起始点，通过一个小神经网络来选择提议区域。

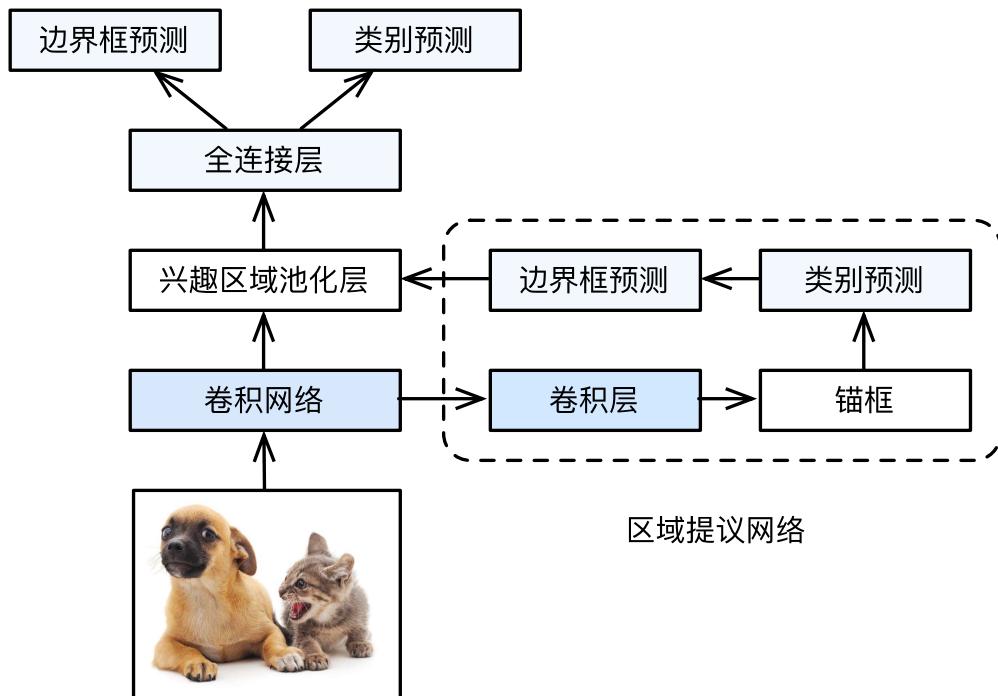


图 9.6: Faster R-CNN 模型。

具体来说，RPN 里面有四个神经层。

1. 卷积网络抽取的特征首先进入 1 填充 256 通道的 3×3 卷积层，这样每个像素得到一个 256 长度的特征表示。

2. 以每个像素为中心，生成多个大小和比例不同的锚框，和对应的标注。每个锚框使用其中心像素对应的 256 维特征来表示。
3. 在锚框特征和标注上面训练一个两类分类器，判断其是否含有感兴趣物体还是只有背景。
4. 对每个被判断成含有物体的锚框，进一步预测其边界框，然后进入 ROI 池化层。

可以看到 RPN 通过标注来学习预测跟真实边界框更相近的提议区域，从而减小提议区域的数量且保证最终模型的预测精度。

9.7.4 Mask R-CNN：使用全连接卷积网络的 Faster RCNN

如果训练数据中我们标注了每个物体的精确边框，而不是一个简单的方形边界框，那么 Mask R-CNN [4] 能有效的利用这些详尽的标注信息来进一步提升物体识别精度。具体来说，Mask R-CNN 使用额外的全连接卷积网络来利用像素级别标注信息，这个网络将在稍后的“语义分割”这一节做详细介绍。

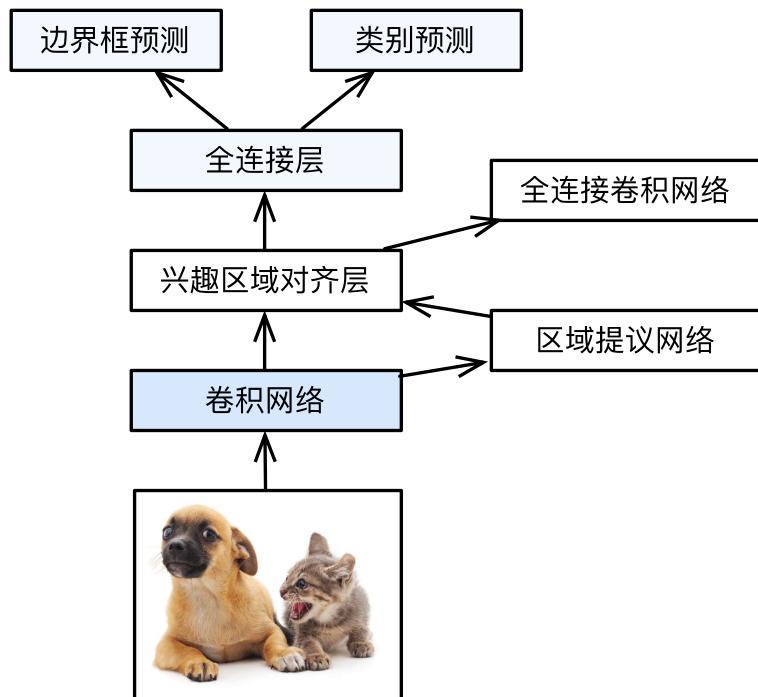


图 9.7: Mask R-CNN 模型。

注意到 RPN 输出的是实数坐标的提议区域，在输入到 RoI 池化层时我们将实数坐标定点化成整数来确定区域中的像素。在计算过程中，我们将每个区域分割成多块然后同样定点化区域边缘到最近的像素上。这两步定点化会使得定点化后的边缘和原始区域中定义的有数个像素的偏差，这个对于边界框预测来说问题不大，但在像素级别的预测上则会带来麻烦。

Mask R-CNN 中提出了 RoI 对齐层 (RoI Align)。它去掉了 RoI 池化层中的定点化过程，从而使得不管是输入的提议区域还是其分割区域的坐标均使用实数。如果边界不是整数，那么其元素值则通过相邻像素插值而来。例如假设对于整数 x 和 y ，坐标 (x, y) 上的值为 $f(x, y)$ 。对于一般的实数坐标，我们先计算 $f(x, \lfloor y \rfloor)$ 和 $f(x, \lfloor y \rfloor + 1)$ ，

$$f(x, \lfloor y \rfloor) = (\lfloor x \rfloor + 1 - x)f(\lfloor x \rfloor, \lfloor y \rfloor) + (x - \lfloor x \rfloor)f(\lfloor x \rfloor + 1, \lfloor y \rfloor),$$

$$f(x, \lfloor y \rfloor + 1) = (\lfloor x \rfloor + 1 - x)f(\lfloor x \rfloor, \lfloor y \rfloor + 1) + (x - \lfloor x \rfloor)f(\lfloor x \rfloor + 1, \lfloor y \rfloor + 1).$$

然后有

$$f(x, y) = (\lfloor y \rfloor + 1 - y)f(x, \lfloor y \rfloor) + (y - \lfloor y \rfloor)f(x, \lfloor y \rfloor + 1).$$

9.7.5 参考文献

- [1] Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 580-587).
- [2] Uijlings, J. R., Van De Sande, K. E., Gevers, T., & Smeulders, A. W. (2013). Selective search for object recognition. International journal of computer vision, 104(2), 154-171.
- [3] Girshick, R. (2015). Fast r-cnn. arXiv preprint arXiv:1504.08083.
- [3] Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. In Advances in neural information processing systems (pp. 91-99).
- [4] He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017, October). Mask r-cnn. In Computer Vision (ICCV), 2017 IEEE International Conference on (pp. 2980-2988). IEEE.

9.8 语义分割和数据集

图片分类关心识别图片里面的主要物体，物体识别则进一步找出图片的多个物体以及它们的方形边界框。本小节我们将介绍语义分割（semantic segmentation），它在物体识别上更进一步的找出物体的精确边界框。换句话说，它识别图片中的每个像素属于哪类我们感兴趣的物体还是只是背景。下图演示猫和狗图片在语义分割中的标注。可以看到，跟物体识别相比，语义分割预测的边框更加精细。

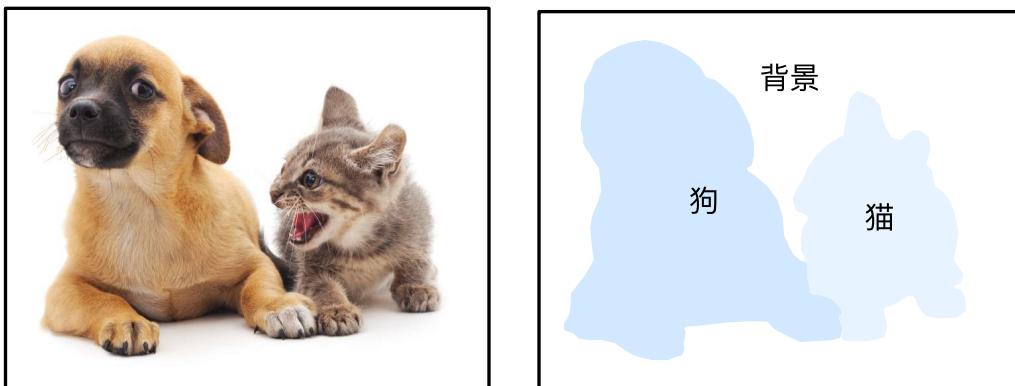


图 9.8：语义分割的训练数据和标注。

在计算机视觉里，还有两个跟语义分割相似的任务。一个是图片分割（image segmentation），它也是将像素划分到不同的类。不同的是，语义分割里我们赋予像素语义信息，例如属于猫、狗或者背景。而图片分割则通常根据像素本身之间的相似性，它训练时不需要像素标注信息，其预测结果也不能保证有语义性。例如图片分割可能将上图中的狗划分成两个区域，其中一个嘴巴和眼睛，其颜色以黑色为主，另一个是身体其余部分，其主色调是黄色。

另一个应用是实例分割（instance segmentation），它不仅需要知道每个像素的语义，即属于那一类物体，还需要进一步区分物体实例。例如如果图片中有两只狗，那么对于预测为对应狗的像素是属于地一只狗还是第二只。

9.8.1 Pascal VOC 语义分割数据集

下面我们使用一个常用的语义分割数据集 [Pascal VOC2012](#) 来介绍这个应用。

```
In [1]: %matplotlib inline  
import sys
```

```
sys.path.append('..')
import gluonbook as gb
import tarfile
from mxnet import nd, image, gluon
```

我们首先下载这个数据集到`../data`下。压缩包大小是2GB，下载需要一定时间。解压之后这个数据集将会放置在`../data/VOCdevkit/VOC2012`下。

```
In [2]: data_dir = '../data'
voc_dir = data_dir + '/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_dir, sha1_hash=sha1)

with tarfile.open(fname, 'r') as f:
    f.extractall(data_dir)
```

在`ImageSets/Segmentation`下有文本文件指定哪些样本用来训练，哪些用来测试。样本图片放置在`JPEGImages`下，标注则放在`SegmentationClass`下。这里标注也是图片格式，图片大小与对应的样本图片一致，其中颜色相同的像素属于同一个类。

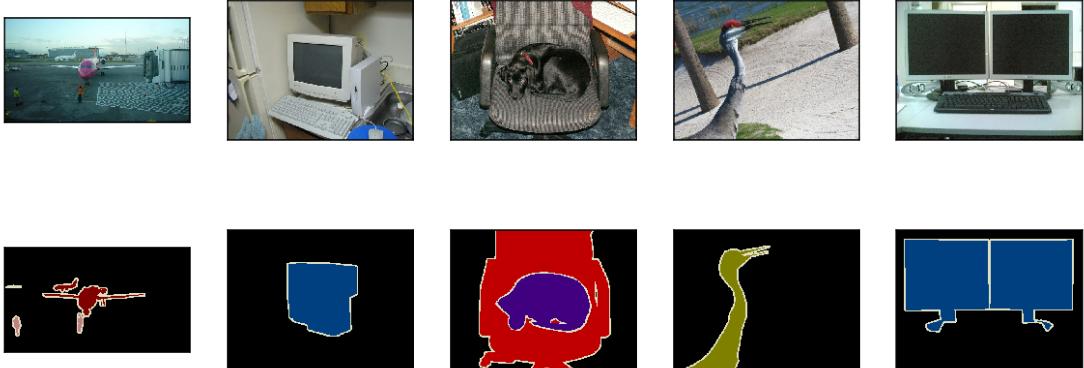
下面定义函数将图片和标注全部读进内存。

```
In [3]: def read_voc_images(root=voc_dir, train=True):
    txt_fname = '%s/ImageSets/Segmentation/%s'%(root,
        'train.txt' if train else 'val.txt')
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    data, label = [None] * len(images), [None] * len(images)
    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

train_images, train_labels = read_voc_images()
```

我们画出前面五张图片和它们对应的标注。在标注，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类别。

```
In [4]: n = 5
imgs = train_images[0:n] + train_labels[0:n]
gb.show_images(imgs, 2, n);
```



接下来我们列出标注中每个 RGB 颜色值对应的类别。

```
In [5]: voc_colormap = [[0,0,0],[128,0,0],[0,128,0], [128,128,0], [0,0,128],
[128,0,128],[0,128,128],[128,128,128],[64,0,0],[192,0,0],
[64,128,0],[192,128,0],[64,0,128],[192,0,128],
[64,128,128],[192,128,128],[0,64,0],[128,64,0],
[0,192,0],[128,192,0],[0,64,128]]
```

```
voc_classes = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
'bottle', 'bus', 'car', 'cat', 'chair', 'cow', 'diningtable',
'dog', 'horse', 'motorbike', 'person', 'potted plant',
'sheep', 'sofa', 'train', 'tv/monitor']
```

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]: colormap2label = np.zeros(256**3)
for i, cm in enumerate(voc_colormap):
    colormap2label[(cm[0]*256+cm[1]) * 256 + cm[2]] = i

def voc_label_indices(img):
    data = img.astype('int32')
    idx = (data[:, :, 0]*256+data[:, :, 1])*256+data[:, :, 2]
    return colormap2label[idx]
```

可以看到第一张样本中飞机头部对应的标注里属于飞机的像素被标记成了 1。

```
In [7]: y = voc_label_indices(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.  1.  1.  1.  1.  1.]
[ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1.]
[ 0.  0.  0.  0.  0.  0.  0.  0.  1.  1.]]

<NDArray 10x10 @cpu(0)>
```

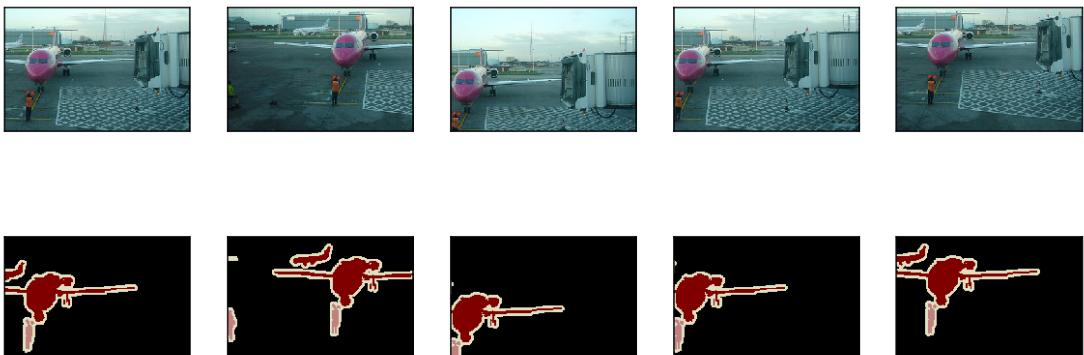
数据预处理

我们知道小批量训练需要输入图片的形状一致。之前我们通过图片缩放来得到同样形状的输入。但语义分割里，如果对样本图片进行缩放，那么重新映射每个像素对应的类别将变得困难，特别是对应物体边缘的像素。

为了避免这个困难，这里我们将图片剪裁成固定大小而不是缩放。特别的，我们使用随机剪裁来附加图片增广。下面定义随机剪裁函数，其对样本图片和标注使用用样的剪裁。

```
In [8]: 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(n):
    imgs += rand_crop(train_images[0], train_labels[0], 200, 300)
gb.show_images(imgs[::2]+imgs[1::2], 2, n);
```



数据读取

下面我们定义 Gluon 可以使用的数据集类，它可以返回任意的第 i 个样本图片和标号。除了随机剪裁外，这里我们将样本图片进行了归一化，同时过滤了小于剪裁尺寸的图片。

```
In [9]: class VOCSegDataset(gluon.data.Dataset):
    def __init__(self, train, crop_size):
        self.rgb_mean = nd.array([0.485, 0.456, 0.406])
        self.rgb_std = nd.array([0.229, 0.224, 0.225])
        self.crop_size = crop_size
        data, label = read_voc_images(train=train)
        self.data = [self.normalize_image(im) for im in self.filter(data)]
        self.label = self.filter(label)
        print('Read '+str(len(self.data))+' examples')

    def normalize_image(self, data):
        return (data.astype('float32') / 255 - self.rgb_mean) / self.rgb_std

    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 __getitem__(self, idx):
        data, label = rand_crop(self.data[idx], self.label[idx],
                               *self.crop_size)
        return data.transpose((2,0,1)), voc_label_indices(label)

    def __len__(self):
        return len(self.data)
```

假设我们剪裁 320×480 图片来进行训练，我们可以查看训练和测试各保留了多少图片。

```
In [10]: output_shape = (320, 480) # 高和宽
voc_train = VOCSegDataset(True, output_shape)
voc_test = VOCSegDataset(False, output_shape)

Read 1114 examples
Read 1078 examples
```

最后定义批量读取，这里使用 4 个进程来加速读取（代码保存在 gluonbook 的 `load_data_pascal_voc` 函数里方便之后使用）。

```
In [11]: batch_size = 64
train_data = gluon.data.DataLoader(
```

```
voc_train, batch_size, shuffle=True, last_batch='discard', num_workers=4)
test_data = gluon.data.DataLoader(
    voc_test, batch_size, last_batch='discard', num_workers=4)
```

打印第一个批量可以看到，不同于图片分类和物体识别，这里的标注是一个三维的数组。

```
In [12]: for data, label in train_data:
    print(data.shape)
    print(label.shape)
    break

(64, 3, 320, 480)
(64, 320, 480)
```

9.8.2 小节

9.8.3 练习

9.9 全卷积网络：FCN

在图片分类里，我们通过卷积层和池化层逐渐减少图片高宽最终得到跟预测类别数长的向量。例如用于 ImageNet 分类的 ResNet 18 里，我们将高宽为 224 的输入图片首先减少到高宽 7，然后使用全局池化层得到 512 维输出，最后使用全连接层输出长为 1000 的预测向量。

但在语义分割里，我们需要对每个像素预测类别，也就是需要输出形状需要是 $1000 \times 224 \times 224$ 。如果仍然使用全链接层作为输出，那么这一层权重将多达数百 GB。本小节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：全卷积网络（fully convolutional network，简称 FCN）[1]。FCN 里将最后的全连接层修改称转置卷积层（transposed convolution）来得到所需大小的输出。

```
In [1]: %matplotlib inline
import sys
sys.path.append('..')
import gluonbook as gb
from mxnet import gluon, init, nd, image
from mxnet.gluon import nn
import numpy as np
```

9.9.1 转置卷积层

假设 f 是一个卷积层，给定输入 x ，我们可以计算前向输出 $y = f(x)$ 。在反向求导 $z = \frac{\partial f(y)}{\partial x}$ 时，我们知道 z 会得到跟 x 一样形状的输出。因为卷积运算的导数的导数是自己本身，我们可以合法定义转置卷积层，记为 g ，为交互了前向和反向求导函数的卷积层。也就是 $z = g(y)$ 。

下面我们构造一个卷积层并打印其输出形状。

```
In [2]: conv = nn.Conv2D(10, kernel_size=4, padding=1, strides=2)
    conv.initialize()
```

```
    x = nd.random.uniform(shape=(1, 3, 64, 64))
    y = conv(x)
    y.shape
```

```
Out[2]: (1, 10, 32, 32)
```

使用同样的卷积窗、填充和步幅的转置卷积层，我们可以得到跟 x 一样的输出。

```
In [3]: conv_trans = nn.Conv2DTranspose(3, kernel_size=4, padding=1, strides=2)
    conv_trans.initialize()
    conv_trans(y).shape
```

```
Out[3]: (1, 3, 64, 64)
```

简单来说，卷积层通常使得输入高宽变小，而转置卷积层则一般用来将高宽增大。

9.9.2 FCN 模型

FCN 的核心思想是将一个卷积网络的最后全连接输出层替换成转置卷积层来获取对每个输入像素的预测。具体来说，它去掉了过于损失空间信息的全局池化层，并将最后的全链接层替换成输出通道是原全连接层输出大小的 1×1 卷积层，最后接上卷积转置层来得到需要形状的输出。

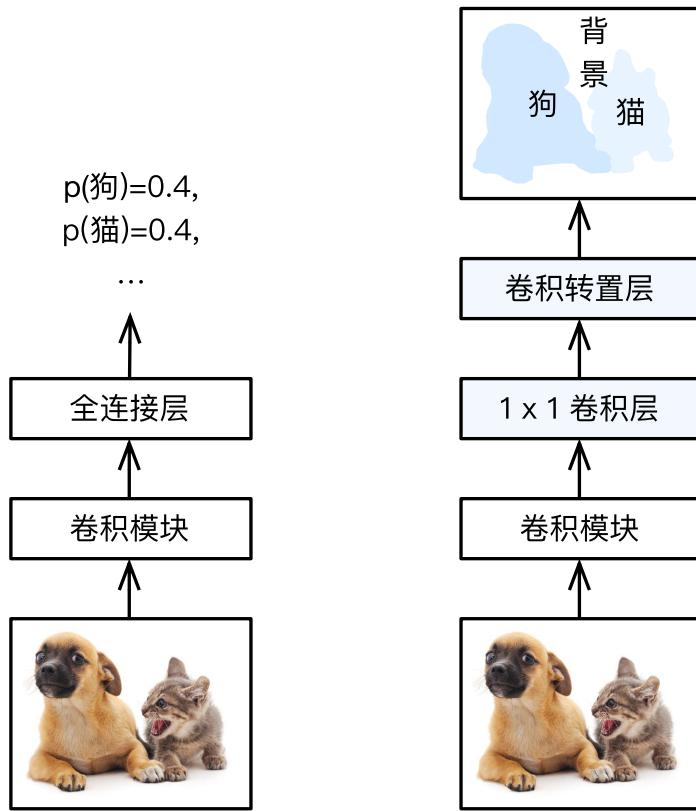


图 9.9: FCN 模型。

下面我们基于 ResNet 18 来创建 FCN。首先我们下载一个预先训练好的模型，并打印其最后的几个神经层。

```
In [4]: pretrained_net = gluon.model_zoo.vision.resnet18_v2(pretrained=True)

(pretrained_net.features[-4:], pretrained_net.output)

Out[4]: ([BatchNorm(axis=1, eps=1e-05, momentum=0.9, fix_gamma=False,
← use_global_stats=False, in_channels=512),
Activation(relu),
GlobalAvgPool2D(size=(1, 1), stride=(1, 1), padding=(0, 0), ceil_mode=True),
Flatten],
Dense(512 -> 1000, linear))
```

可以看到 feature 模块最后两层是 GlobalAvgPool2D 和 Flatten，在 FCN 里均不需要，output 模块里的全链接层也需要舍去。下面我们定义一个新的网络，它复制除了 feature 里

除去最后两层的所有神经层以及权重。

```
In [5]: net = nn.HybridSequential()
    for layer in pretrained_net.features[:-2]:
        net.add(layer)
```

给定高宽为 224 的输入，net 的输出将输入高宽减少了 32 倍。

```
In [6]: x = nd.random.uniform(shape=(1,3,224,224))
    net(x).shape
```

```
Out[6]: (1, 512, 7, 7)
```

为了使的输出跟输入有同样的高宽，我们构建一个步幅为 32 的转置卷积层，卷积核的窗口高宽设置成步幅的 2 倍，并补充适当的填充。在转置卷积层之前，我们加上 1×1 卷积层来将通道数从 512 降到标注类别数，对 Pascal VOC 数据集来说是 21。

```
In [7]: num_classes = 21

    net.add(
        nn.Conv2D(num_classes, kernel_size=1),
        nn.Conv2DTranspose(num_classes, kernel_size=64, padding=16, strides=32)
    )
```

9.9.3 模型初始化

net 中的最后两层需要对权重进行初始化，通常我们会使用随机初始化。但新加入的转置卷积层的功能有些类似于将输入调整到更大的尺寸。在图片处理里面，我们可以通过有适当卷积核的卷积运算符来完成这个操作。常用的包括双线性差值核，下面函数构造核权重。

```
In [8]: 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)
```

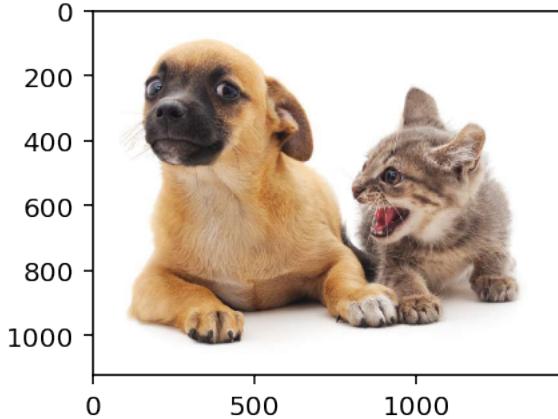
接下来我们构造一个步幅为 2 的转置卷积层，将其权重初始化成双线性差值核。

```
In [9]: conv_trans = nn.Conv2DTranspose(3, kernel_size=4, padding=1, strides=2)
      conv_trans.initialize(init.Constant(bilinear_kernel(3, 3, 4)))
```

可以看到这个转置卷积层的前向函数的效果是将输入图片高宽扩大 2 倍。

```
In [10]: img = image.imread('../img/catdog.jpg')
      print('Input', img.shape)
      x = img.astype('float32').transpose((2, 0, 1)).expand_dims(axis=0)/255
      y = conv_trans(x)
      y = y[0].clip(0, 1).transpose((1, 2, 0))
      print('Output', y.shape)
      gb.plt.imshow(y.asnumpy());
```

```
Input (561, 728, 3)
Output (1122, 1456, 3)
```



下面对 `net` 的最后两层进行初始化。其中 1×1 卷积层使用 Xavier，转置卷积层则使用双线性差值核。

```
In [11]: trans_conv_weights = bilinear_kernel(num_classes, num_classes, 64)
      net[-1].initialize(init.Constant(trans_conv_weights))
      net[-2].initialize(init=init.Xavier())
```

9.9.4 训练

这时候我们可以真正开始训练了。我们使用较大的输入图片尺寸，其值选成了 32 的倍数。因为我们使用转置卷积层的通道来预测像素的类别，所以在做 softmax 是作用在通道这个维度（维度

1)，所以在 SoftmaxCrossEntropyLoss 里加入了额外的 axis=1 选项。

```
In [12]: input_shape = (320, 480)
batch_size = 32
ctx = gb.try_all_gpus()
loss = gluon.loss.SoftmaxCrossEntropyLoss(axis=1)
net.collect_params().reset_ctx(ctx)
trainer = gluon.Trainer(net.collect_params(),
                        'sgd', {'learning_rate': .1, 'wd':1e-3})
train_data, test_data = gb.load_data_pascal_voc(batch_size, input_shape)
gb.train(train_data, test_data, net, loss, trainer, ctx, num_epochs=10)

Read 1114 examples
Read 1078 examples
training on [gpu(0), gpu(1)]
epoch 1, loss 1.3334, train acc 0.736, test acc 0.821, time 34.4 sec
epoch 2, loss 0.5527, train acc 0.833, test acc 0.833, time 20.4 sec
epoch 3, loss 0.4474, train acc 0.858, test acc 0.845, time 20.7 sec
epoch 4, loss 0.3772, train acc 0.877, test acc 0.842, time 20.2 sec
epoch 5, loss 0.3315, train acc 0.891, test acc 0.843, time 20.0 sec
epoch 6, loss 0.2851, train acc 0.905, test acc 0.849, time 20.3 sec
epoch 7, loss 0.2586, train acc 0.913, test acc 0.854, time 20.4 sec
epoch 8, loss 0.2370, train acc 0.919, test acc 0.858, time 20.1 sec
epoch 9, loss 0.2196, train acc 0.925, test acc 0.851, time 20.0 sec
epoch 10, loss 0.2116, train acc 0.927, test acc 0.859, time 20.2 sec
```

9.9.5 预测

预测一张新图片时，我们只需要将其归一化并转成卷积网络需要的 4D 格式。

```
In [13]: def predict(im):
    data = test_data._dataset.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]))
```

同时我们根据每个像素预测的类别找出其 RGB 颜色以便画图。

```
In [14]: def label2image(pred):
    colormap = nd.array(
        test_data._dataset.voc_colormap, ctx=ctx[0], dtype='uint8')
```

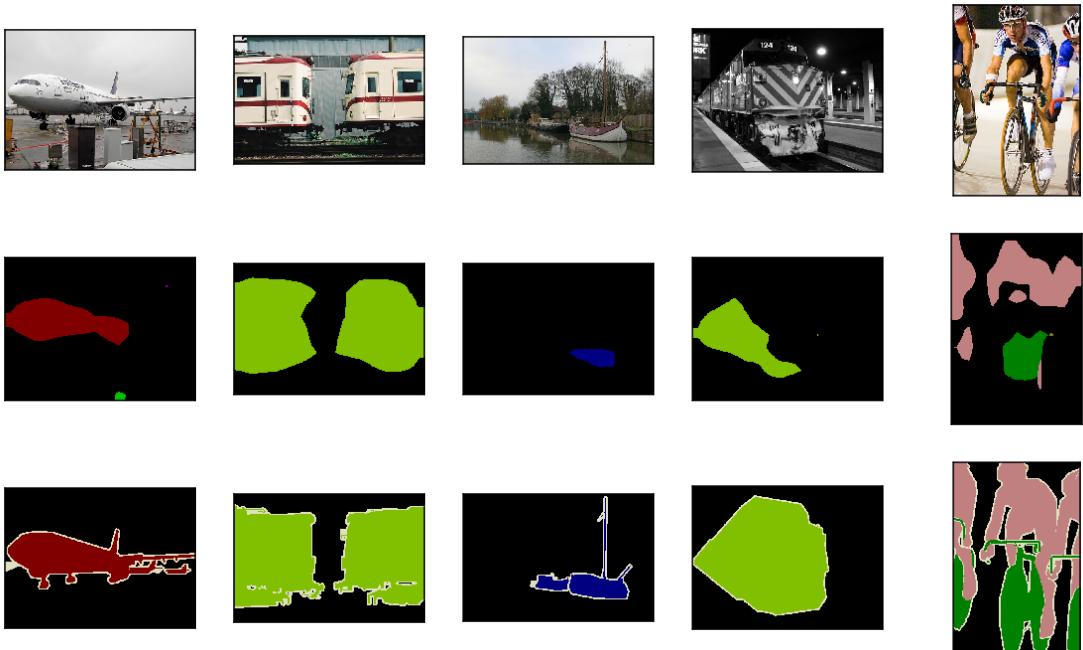
```
x = pred.astype('int32')
return colormap[x,:]
```

现在我们读取前几张测试图片并对其进行预测。

```
In [15]: test_images, test_labels = gb.read_voc_images(train=False)
```

```
n = 5
imgs = []
for i in range(n):
    x = test_images[i]
    pred = label2image(predict(x))
    imgs += [x, pred, test_labels[i]]
```

```
gb.show_images(imgs[::3]+imgs[1::3]+imgs[2::3], 3, n);
```



9.9.6 小结

FCN 通过使用转置卷积层来为每个像素预测类别。

9.9.7 练习

- 试着改改最后的转置卷积层的参数设定。
- 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 [1] 中提到了不只是使用主体卷积网络输出，还可以考虑其中间层的输出。试着实现这个想法。

9.9.8 扫码直达讨论区



9.9.9 参考文献

[1] Long, Jonathan, Evan Shelhamer, and Trevor Darrell. “Fully convolutional networks for semantic segmentation.” CVPR. 2015.

9.10 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，可以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，达到想要的风格经常需要大量组合尝试，其复杂程度不亚于模型调参。

本小节我们将介绍如何使用神经网络来自动化这个过程 [1]。这里我们需要两张输入图片，一张是内容图片，另一张是样式图片，我们将使用神经网络修改内容图片使得其样式接近样式图片。下图中的内容图片为作者在西雅图郊区的雷尼尔山（mountain rainier）拍摄风景照，样式图片则为一副主题为秋天橡树的油画，其合成图片在保留了内容图片中物体主体形状的情况下加入了样式图片的油画笔触，同时也让整体颜色更加鲜艳。

内容图片



合成图片



样式图片



图 9.10: 样式迁移。

使用神经网络的图片合成如下图所示。下图中我们选取一个有三个卷积层的神经网络来提取特征。对于样式图片，我们选取第一和第三层输出作为样式特征。对于内容图片则选取第二层输出作为内容特征。给定一个合成图片的初始值，我们不断的迭代直到其输入到同样神经网络时第一和第三层输出能匹配上样式特征，第二层输出能匹配到内容特征。

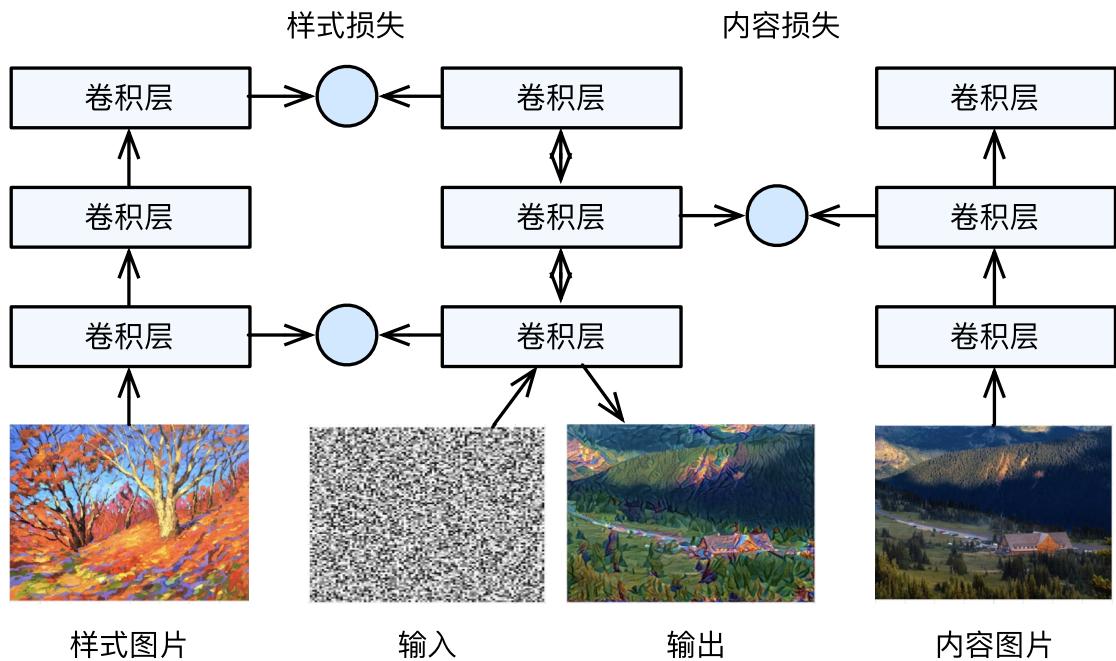


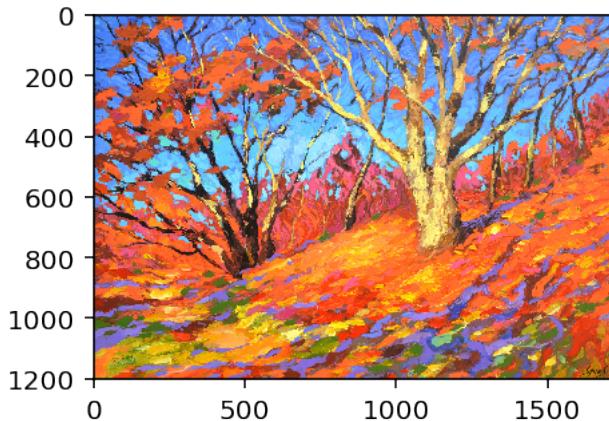
图 9.11: Neural Style。

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

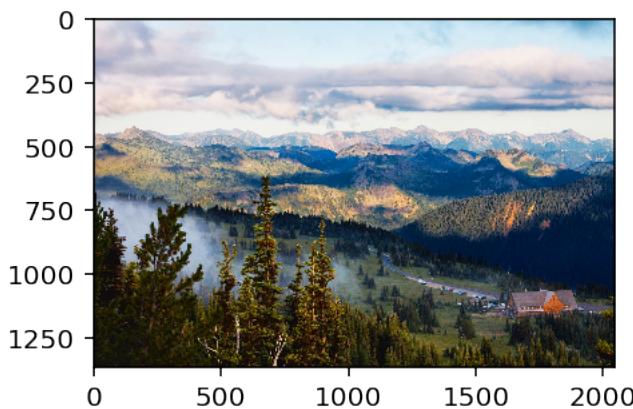
9.10.1 数据

我们分别读取样式和内容图片。

```
In [2]: style_img = image.imread('../img/autumn_oak.jpg')
gb.plt.imshow(style_img.asnumpy());
```



```
In [3]: content_img = image.imread('../img/rainier.jpg')
gb.plt.imshow(content_img.asnumpy());
```



然后定义预处理和后处理函数。预处理函数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，后处理函数则还原成能展示的图片格式。

```
In [4]: 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)
```

9.10.2 抽取特征

我们使用原论文 [1] 使用的 VGG 19 模型，并下载在 Imagenet 上训练好的权重。

```
In [5]: pretrained_net = gluon.model_zoo.vision.vgg19(pretrained=True)
```

我们知道 VGG 使用了五个卷积块来构建网络，块之间使用最大池化层来做间隔（参考“使用重复元素的网络（VGG）”小节）。[1] 中使用每个卷积块的第一个卷积层输出来匹配样式（称之为样式层），和第四块中的最后一个卷积层来匹配内容（称之为内容层）。我们可以打印 `pretrained_net` 来获取这些层的具体位置。

```
In [6]: style_layers = [0, 5, 10, 19, 28]
content_layers = [25]
```

当然，样式层和内容层有多种选取方法。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。这里我们选取比较靠后的内容层来避免合成图片过于保留内容图片细节，使用多个位置的样式层来匹配局部和全局样式。

下面构建一个新的网络使其只保留我们需要预留的层。

```
In [7]: net = nn.Sequential()
for i in range(max(content_layers+style_layers)+1):
    net.add(pretrained_net.features[i])
```

给定输入 `x`，简单使用 `net(x)` 只能拿到最后的输出，而这里我们还需要中间层输出。因此我们逐层计算，并保留样式层和内容层的输出。

```
In [8]: 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
```

最后我们定义函数分别对内容图片和样式图片抽取对应的特征。因为在训练时我们不修改网络的权重，所以我们在训练开始之前提取所要的特征。

```
In [9]: def get_contents(image_shape, ctx):
    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, ctx):
    style_x = preprocess(style_img, image_shape).copyto(ctx)
    _, style_y = extract_features(style_x, content_layers, style_layers)
    return style_x, style_y
```

9.10.3 损失函数

在训练时，我们需要定义如何比较合成图片和内容图片的内容层输出（内容损失函数），以及比较和样式图片的样式层输出（样式损失函数）。内容损失函数可以使用回归用的均方误差。

```
In [10]: def content_loss(y_hat, y):
    return (y_hat - y).square().mean()
```

对于样式，我们可以简单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样式，我们可以匹配这两张图片在 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（通道，高，宽）。那么我们可以把它变形成 $c \times hw$ 的二维数组，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，我们只匹配二阶信息，即协方差。下面定义如何计算协方差矩阵，

```
In [11]: def gram(x):
    c, n = x.shape[1], x.size // x.shape[1]
    y = x.reshape((c, n))
    return nd.dot(y, y.T) / n
```

和对应的损失函数，这里假设样式图片的样式特征协方差已经预先计算好了。

```
In [12]: def style_loss(y_hat, gram_y):
    return (gram(y_hat) - gram_y).square().mean()
```

当我们使用靠近输出层的神经层输出来匹配时，经常可以观察到学到的合成图片里面有大量高频噪音，即有特别亮或者暗的颗粒像素。一种常用的降噪方法是总变差降噪（total variation denoising）。假设 $x_{i,j}$ 表示像素 (i, j) 的值，总变差损失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 [13]: def tv_loss(y_hat):
    return 0.5*((y_hat[:, :, 1:, :] - y_hat[:, :, :-1, :]).abs().mean() +
                (y_hat[:, :, :, 1:] - y_hat[:, :, :, :-1]).abs().mean())
```

训练中我们将上述三个损失函数加权求和。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。此外注意到样式层里有五个神经层，我们对靠近输入的有较少的通道数的层给予比较大的权重。

```
In [14]: style_channels = [net[l].weight.shape[0] for l in style_layers]
style_weights = [1e4/c**2 for c in style_channels]
content_weights = [1]
tv_weight = 10
```

9.10.4 训练

这里的训练跟前面章节的主要不同在于我们只对输入 x 进行更新。此外我们将 x 的梯度除以了它的绝对平均值来降低对学习率的敏感度，而且每隔一定的批量我们减小一次学习率。

```
In [15]: def train(x, content_y, style_y, ctx, lr, max_epochs, lr_decay_epoch):
    x = x.as_in_context(ctx)
    x.attach_grad()
    style_y_gram = [gram(y) for y in style_y]
    for i in range(1, max_epochs+1):
        tic = time.time()
        with autograd.record():
            # 对 x 抽取样式和内容特征。
            content_y_hat, style_y_hat = extract_features(
                x, content_layers, style_layers)
            # 分别计算内容、样式和噪音损失。
            content_L = [w * content_loss(y_hat, y) for w, y_hat, y in zip(
                content_weights, content_y_hat, content_y)]
            style_L = [w * style_loss(y_hat, y) for w, y_hat, y in zip(
                style_weights, style_y_hat, style_y_gram)]
            tv_L = tv_weight * tv_loss(x)
            # 对所有损失求和。
            loss = nd.add_n(*style_L) + nd.add_n(*content_L) + tv_L

            loss.backward()
            # 对 x 的梯度除去绝对均值使得数值更加稳定，并更新 x
            x.grad[:] /= x.grad.abs().mean() + 1e-8
            x[:] -= lr * x.grad
            # 如果不加的话会导致每 50 轮迭代才同步一次，可能导致过大内存使用。
            nd.waitall()
```

```

if i % 50 == 0:
    print('batch %3d: content %.2f, style %.2f, '
          'TV %.2f, %.1f sec per batch' % (
              i, nd.add_n(*content_L).asscalar(),
              nd.add_n(*style_L).asscalar(),
              tv_L.asscalar(), time.time()-tic))

if i % lr_decay_epoch == 0:
    lr *= 0.1
    print('change lr to %.1e' % lr)
return x

```

现在我们可以真正开始训练了。首先我们将图片调整到高为 300 宽 200 来进行训练，这样使得训练更加快速。合成图片的初始值设成了内容图片，使得初始值能尽可能接近训练输出来加速收敛。

```

In [16]: image_shape = (300, 200)
ctx = gb.try_gpu()

net.collect_params().reset_ctx(ctx)
content_x, content_y = get_contents(image_shape, ctx)
style_x, style_y = get_styles(image_shape, ctx)

x = content_x
y = train(x, content_y, style_y, ctx, 0.1, 500, 200)

batch 50: content 33.72, style 630.74, TV 4.79, 0.1 sec per batch
batch 100: content 37.21, style 725.17, TV 5.20, 0.1 sec per batch
batch 150: content 38.22, style 852.46, TV 5.55, 0.1 sec per batch
batch 200: content 37.46, style 755.03, TV 5.84, 0.1 sec per batch
change lr to 1.0e-02
batch 250: content 19.34, style 28.02, TV 5.52, 0.1 sec per batch
batch 300: content 16.90, style 23.73, TV 5.45, 0.1 sec per batch
batch 350: content 15.84, style 22.80, TV 5.40, 0.1 sec per batch
batch 400: content 15.18, style 21.49, TV 5.35, 0.1 sec per batch
change lr to 1.0e-03
batch 450: content 14.40, style 12.89, TV 5.34, 0.1 sec per batch
batch 500: content 13.91, style 12.19, TV 5.31, 0.1 sec per batch

```

因为使用了内容图片作为初始值，所以一开始内容误差远小于样式误差。随着迭代的进行样式误差迅速减少，最终它们值在相近的范围。下面我们将训练好的合成图片保存下来。

```
In [17]: gb.plt.imsave('neural-style-1.png', postprocess(y).asnumpy())
```

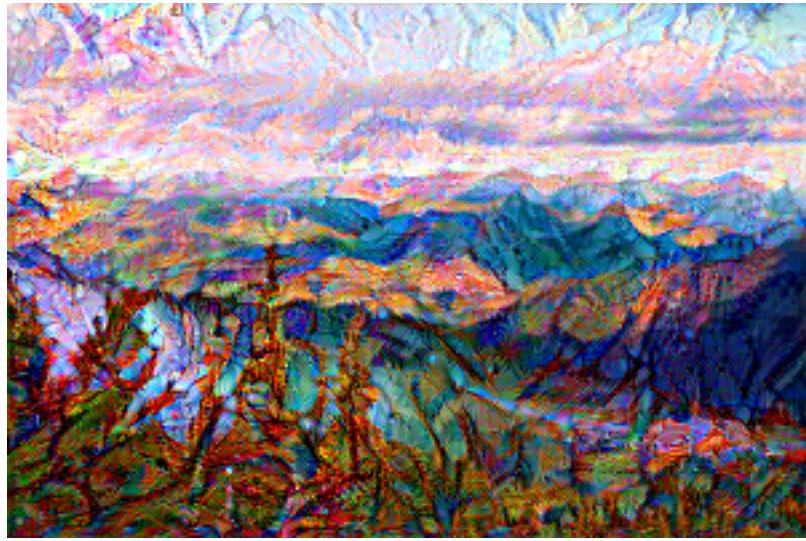


图 9.12: 300 x 200 尺寸的合成图片。

可以看到合成图片保留了样式的风景物体，同时借鉴了样式的色彩。由于图片尺寸较小，所以细节上比较模糊。下面我们在更大的 1200 x 800 的尺寸上训练，希望可以得到更加清晰的合成图片。为了加速收敛，我们将训练到的合成图片高宽放大 3 倍来作为初始值。

```
In [18]: image_shape = (1200, 800)

content_x, content_y = get_contents(image_shape, ctx)
style_x, style_y = get_styles(image_shape, ctx)

x = preprocess(postprocess(y)*255, image_shape)
z = train(x, content_y, style_y, ctx, 0.1, 300, 100)

gb.plt.imsave('neural-style-2.png', postprocess(z).asnumpy())

batch 50: content 22.59, style 798.89, TV 3.70, 0.9 sec per batch
batch 100: content 23.91, style 606.46, TV 4.19, 0.9 sec per batch
change lr to 1.0e-02
batch 150: content 20.58, style 35.80, TV 3.41, 0.9 sec per batch
batch 200: content 18.19, style 31.45, TV 3.26, 0.9 sec per batch
change lr to 1.0e-03
batch 250: content 15.95, style 12.14, TV 3.18, 0.9 sec per batch
batch 300: content 15.15, style 10.71, TV 3.14, 0.9 sec per batch
change lr to 1.0e-04
```

可以看到这一次由于初始值离最终输出更近使得收敛更加迅速。但同时由于图片尺寸更大，每一次迭代需要花费更多的时间和内存。

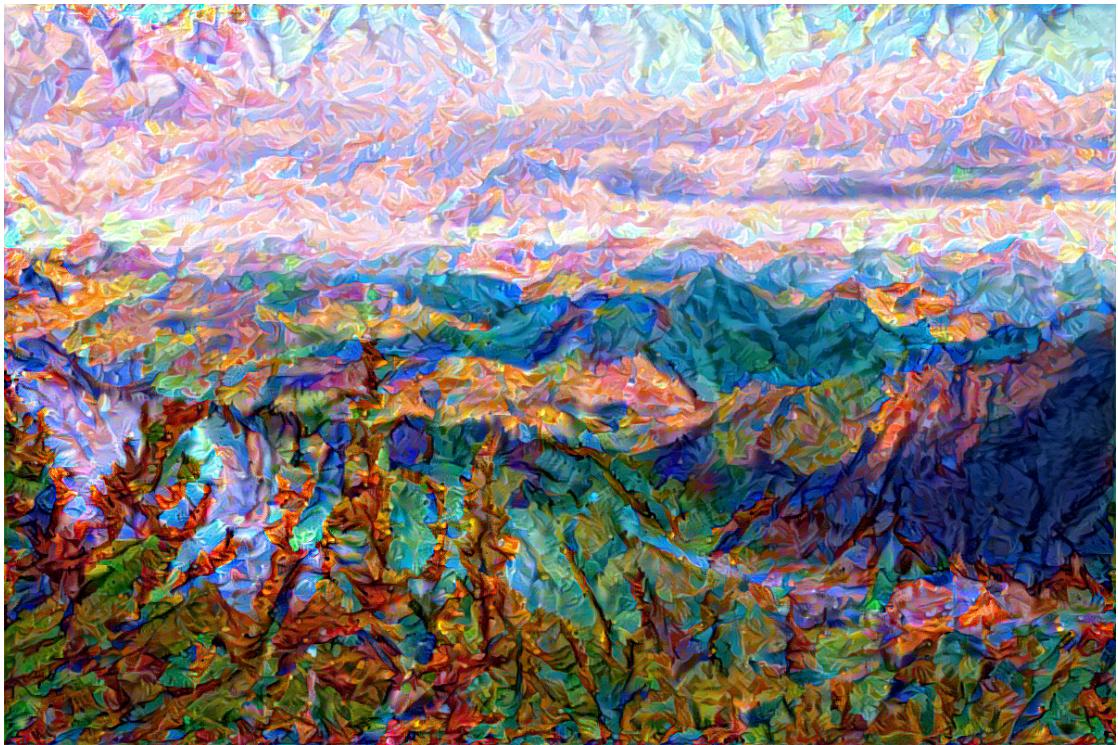


图 9.13: 1200 x 800 尺寸的合成图片。

从训练得到的图片可以看到它保留了更多的细节，里面不仅有大块的类似样式图片的油画色彩块，色彩块里面也有细微的纹理。

9.10.5 小结

通过匹配神经网络的中间层输出可以有效的融合不同图片的内容和样式。

9.10.6 练习

- 选择不同的内容和样式层。

- 使用不同的损失权重来得到更偏向内容或样式或平滑的输出。
- 一个得到更加干净的合成图的办法是使用更大的尺寸。
- 换别的样式和内容图片试试。

9.10.7 扫码直达讨论区



9.10.8 参考文献

[1] Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. “Image style transfer using convolutional neural networks.” CVPR. 2016.

9.11 实战 Kaggle 比赛：图像分类（CIFAR-10）

CIFAR-10 是计算机视觉领域的一个重要的数据集。本节中，我们将动手实战一个有关它的 Kaggle 比赛：CIFAR-10 图像分类问题。该比赛的网页地址是

<https://www.kaggle.com/c/cifar-10>

图 9.X 展示了该比赛的网页信息。为了便于提交结果，请先在 Kaggle 网站上注册账号。

图 9.14: CIFAR-10 图像分类比赛的网页信息。比赛数据集可通过点击“Data”标签获取。

首先，导入实验所需的包或模块。

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

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

9.11.1 获取数据集

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，其他 29 万张不计分的图片是为了防止人工标注测试集。两个数据集中的图片格式都是 png，高和宽均为 32 像素，并含有 RGB 三个通道（彩色）。图片的类别数为 10，类别分别为飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船和卡车，如图 9.X 所示。

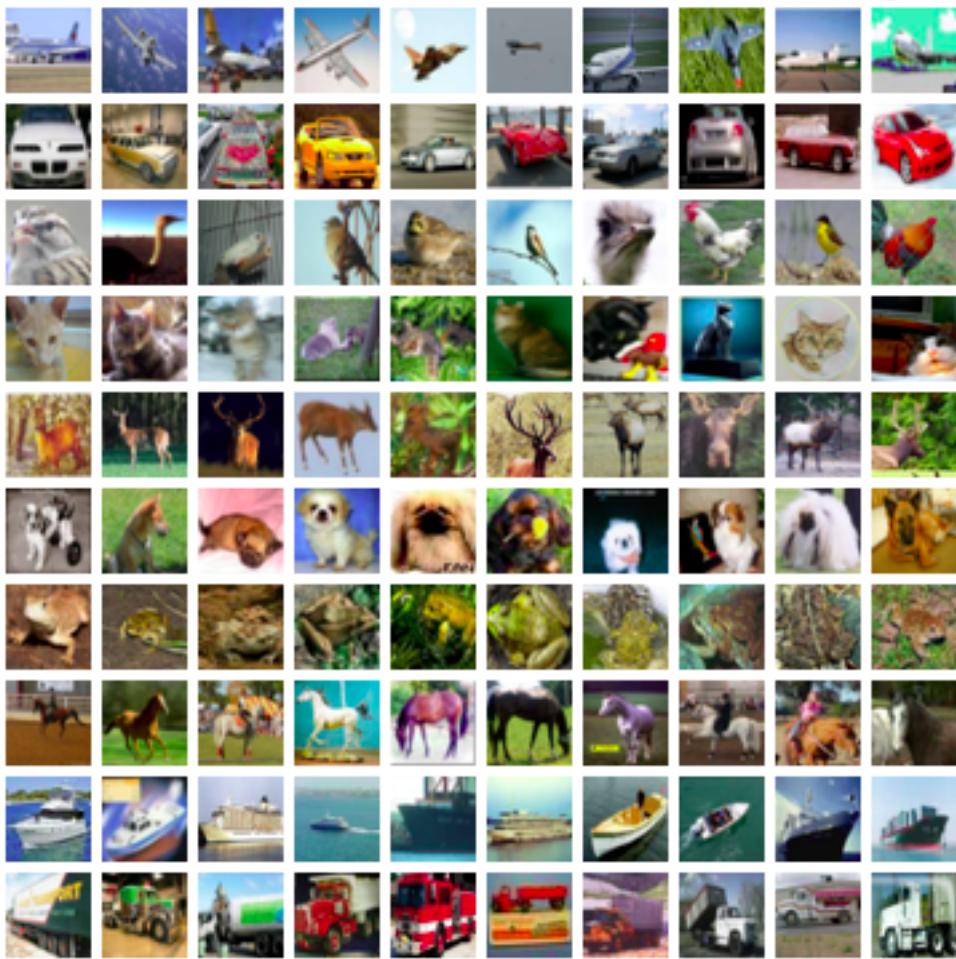


图 9.15: CIFAR-10 图像的类别分别为飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船和卡车。

下载数据集

登录 Kaggle 后，我们可以点击图 9.X 所示的 CIFAR-10 图像分类比赛网页上的“Data”标签，并分别下载训练数据集“train.7z”、测试数据集“test.7z”和训练数据集标签“trainLabels.csv”。

解压数据集

下载完训练数据集“train.7z”和测试数据集“test.7z”后请解压缩。解压缩后，将训练数据集、测试数据集和训练数据集标签分别存放在以下路径：

-/data/kaggle_cifar10/train/[1-50000].png
-/data/kaggle_cifar10/test/[1-300000].png
-/data/kaggle_cifar10/trainLabels.csv

为方便快速上手，我们提供了上述数据集的小规模采样，例如仅含 100 个训练样本的“train_tiny.zip”和 1 个测试样本的“test_tiny.zip”。它们解压后的文件夹名称分别为“train_tiny”和“test_tiny”。此外，训练数据集标签的压缩文件解压后得到“trainLabels.csv”。如果你将使用上述 Kaggle 比赛的完整数据集，还需要把下面 demo 变量改为 False。

In [2]: # 如果使用下载的 Kaggle 比赛的完整数据集，把下面改为 False。

```
demo = True
if demo:
    import zipfile
    for f in ['train_tiny.zip', 'test_tiny.zip', 'trainLabels.csv.zip']:
        with zipfile.ZipFile('../data/kaggle_cifar10/' + f, 'r') as z:
            z.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，同一类图片将被放在同一个文件夹下，便于我们稍后读取。该函数中的参数 `valid_ratio` 是验证集样本数与原始训练集样本数之比。以 `valid_ratio=0.1` 为例，由于原始训练数据集有 50,000 张图片，调参时将有 45,000 张图片用于训练并存放在路径“`input_dir/train`”，而另外 5,000 张图片为验证集并存放在路径“`input_dir/valid`”。

In [3]: `def reorg_cifar10_data(data_dir, label_file, train_dir, test_dir, input_dir, valid_ratio):`

```
# 读取训练数据集标签。
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())
```

```

n_train_valid = len(os.listdir(os.path.join(data_dir, train_dir)))
n_train = int(n_train_valid * (1 - valid_ratio))
assert 0 < n_train < n_train_valid
n_train_per_label = n_train // len(labels)
label_count = {}

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 % len(label_count)
    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] < n_train_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'))

```

我们在这里仅仅使用 100 个训练样本和 1 个测试样本。训练和测试数据集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 比赛的完整数据集，并将批量大小 batch_size 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 [4]: if demo:
    # 注意：此处使用小训练集。
    train_dir = 'train_tiny'
    # 注意：此处使用小测试集。
    test_dir = 'test_tiny'
```

```
# 注意：此处将批量大小相应设小。使用 Kaggle 比赛的完整数据集时可设较大整数。  
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)
```

9.11.2 图片增广

为应对过拟合，我们在这里使用 `transforms` 来增广数据。例如，加入 `transforms.RandomFlipLeftRight()` 即可随机对图片做镜面反转。我们也通过 `transforms.Normalize()` 对彩色图像 RGB 三个通道分别做标准化。以下列举了部分操作。这些操作可以根据需求来决定是否使用或修改。

```
In [5]: transform_train = transforms.Compose([  
    # 将图片放大成高和宽各为 40 像素的正方形。  
    transforms.Resize(40),  
    # 随机对高和宽各为 40 像素的正方形图片裁剪出面积为原图片面积 0.64 到 1 倍之间的小正方  
    # 形，再放缩为高和宽各为 32 像素的正方形。  
    transforms.RandomResizedCrop(32, scale=(0.64, 1.0), ratio=(1.0, 1.0)),  
    # 随机左右翻转图片。  
    transforms.RandomFlipLeftRight(),  
    # 将图片像素值按比例缩小到 0 和 1 之间，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 `ImageFolderDataset` 类来读取整理后的数据集，其中每个数据样本包括图像和标签。需要注意的是，我们要在 `DataLoader` 中调用刚刚定义好的图片增广函数。其中 `transform_first` 函数指明对每个数据样本中的图像做数据增广。

```
In [6]: # 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 = gdata.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'train'), flag=1)
valid_ds = gdata.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'valid'), flag=1)
train_valid_ds = gdata.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'train_valid'), flag=1)
test_ds = gdata.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'test'), flag=1)

train_data = gdata.DataLoader(train_ds.transform_first(transform_train),
                             batch_size, shuffle=True, last_batch='keep')
valid_data = gdata.DataLoader(valid_ds.transform_first(transform_test),
                             batch_size, shuffle=True, last_batch='keep')
train_valid_data = gdata.DataLoader(train_valid_ds.transform_first(
    transform_train), batch_size, shuffle=True, last_batch='keep')
test_data = gdata.DataLoader(test_ds.transform_first(transform_test),
                             batch_size, shuffle=False, last_batch='keep')
```

9.11.3 定义模型

我们这里使用了 ResNet-18 模型，并使用混合式编程来提升执行效率。

```
In [7]: 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

```

9.11.4 定义训练函数

我们将依赖模型在验证集上的表现来选择模型并调节超参数。下面定义了模型的训练函数 `train`。我们记录了每个迭代周期的训练时间。这有助于比较不同模型的时间开销。

In [8]: `loss = gloss.SoftmaxCrossEntropyLoss()`

```

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_l = 0.0
        train_acc = 0.0
        if epoch > 0 and epoch % lr_period == 0:
            trainer.set_learning_rate(trainer.learning_rate * lr_decay)
        for X, y in train_data:
            y = y.astype('float32').as_in_context(ctx)
            with autograd.record():
                y_hat = net(X.as_in_context(ctx))
                l = loss(y_hat, y)
                l.backward()
                trainer.step(batch_size)
                train_l += l.mean().asscalar()
                train_acc += gb.accuracy(y_hat, y)
        cur_time = datetime.datetime.now()
        h, remainder = divmod((cur_time - prev_time).seconds, 3600)
        m, s = divmod(remainder, 60)
        time_s = "time %02d:%02d:%02d" % (h, m, s)
        if valid_data is not None:
            valid_acc = gb.evaluate_accuracy(valid_data, net, ctx)
            epoch_s = ("epoch %d, loss %f, train acc %f, valid acc %f, "
                      "% (epoch, train_l / len(train_data),"
                      "train_acc / len(train_data), valid_acc))"
        else:
            epoch_s = ("epoch %d, loss %f, train acc %f, " %

```

```
(epoch, train_l / len(train_data),
     train_acc / len(train_data)))
prev_time = cur_time
print(epoch_s + time_s + ', lr ' + str(trainer.learning_rate))
```

9.11.5 训练并验证模型

现在，我们可以训练并验证模型了。以下的超参数都是可以调节的，例如增加迭代周期。

```
In [9]: ctx = gb.try_gelu()
num_epochs = 1
# 学习率。
lr = 0.1
# 权重衰减参数。
wd = 5e-4
# 优化算法的学习率将在每 80 个迭代周期时自乘 0.1。
lr_period = 80
lr_decay = 0.1

net = get_net(ctx)
net.hybridize()
train(net, train_data, valid_data, num_epochs, lr, wd, ctx, lr_period,
      lr_decay)

epoch 0, loss 3.425283, train acc 0.122222, valid acc 0.100000, time 00:00:01, lr 0.1
```

9.11.6 对测试集分类并在 Kaggle 提交结果

当得到一组满意的模型设计和超参数后，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 [10]: net = get_net(ctx)
net.hybridize()
train(net, train_valid_data, None, num_epochs, lr, wd, ctx, lr_period,
      lr_decay)

preds = []
for X, _ in test_data:
    y_hat = net(X.as_in_context(ctx))
    preds.extend(y_hat.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 3.662429, train acc 0.060000, time 00:00:01, lr 0.1
```

执行完上述代码后，会生成一个“`submission.csv`”文件。这个文件符合 Kaggle 比赛要求的提交格式。这时我们可以在 Kaggle 上把对测试集分类的结果提交并查看分类准确率。你需要登录 Kaggle 网站，访问 CIFAR-10 比赛网页，并点击右侧“Submit Predictions”或“Late Submission”按钮。然后，点击页面下方“Upload Submission File”选择需要提交的分类结果文件。最后，点击页面最下方的“Make Submission”按钮就可以查看结果了。

9.11.7 小结

- CIFAR-10 是计算机视觉领域的一个重要的数据集。
- 我们可以应用卷积神经网络、图片增广和混合式编程来实战图像分类比赛。

9.11.8 练习

- 使用 Kaggle 比赛的完整 CIFAR-10 数据集。把批量大小 `batch_size` 和迭代周期数 `num_epochs` 分别改为 128 和 100。看看你可以在这个比赛中拿到什么样的准确率和名次？
- 如果不使用增强数据的方法能拿到什么样的准确率？
- 扫码直达讨论区，在社区交流方法和结果。你能发掘出其他更好的技巧吗？

9.11.9 扫码直达讨论区



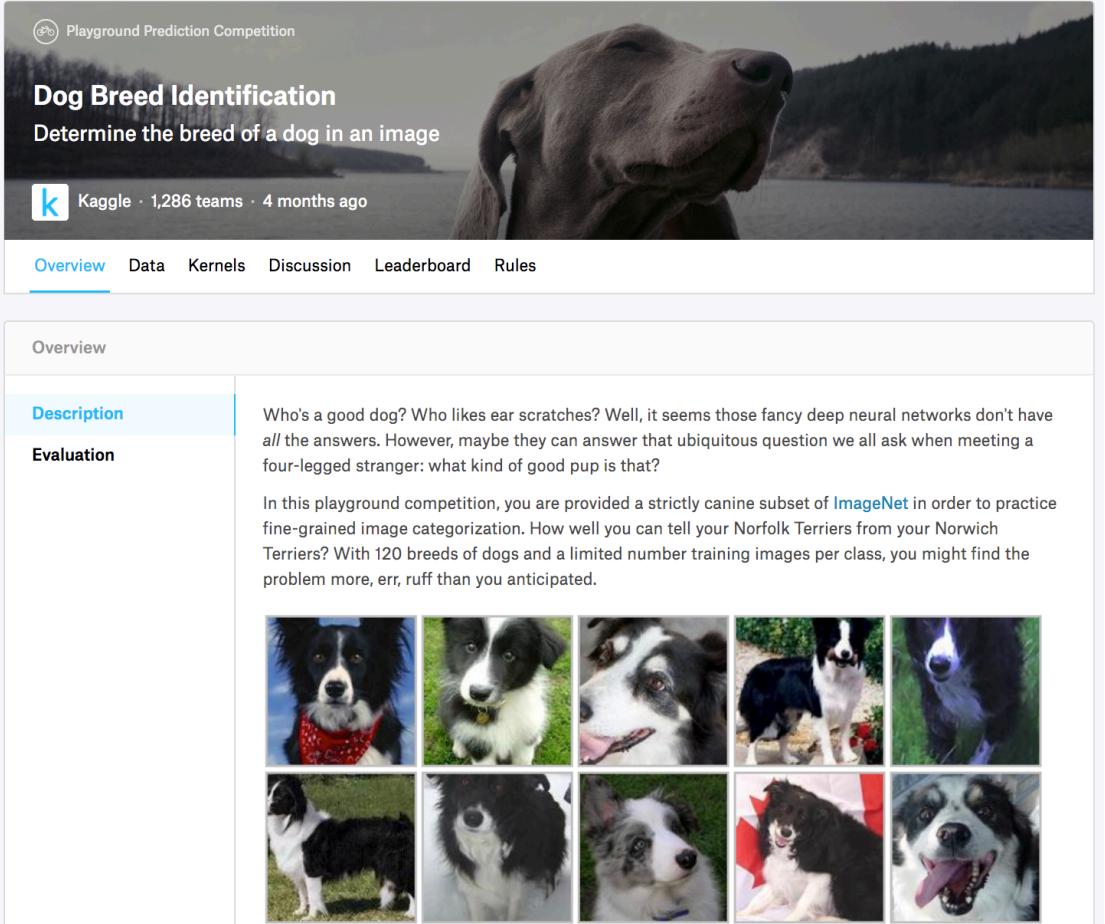
9.12 实战 Kaggle 比赛：狗的品种识别 (ImageNet Dogs)

我们将在本节动手实战 Kaggle 比赛中的狗的品种识别问题。该比赛的网页地址是

<https://www.kaggle.com/c/dog-breed-identification>

在这个比赛中，我们将识别 120 类不同品种的狗。这个比赛的数据集实际上是著名的 ImageNet 的子集数据集。和上一节 CIFAR-10 数据集中的图像不同，ImageNet 数据集中的图像的高和宽更大，且大小不一。

图 9.X 展示了该比赛的网页信息。为了便于提交结果，请先在 Kaggle 网站上注册账号。



The screenshot shows the Kaggle Dog Breed Identification competition page. At the top, there's a large image of a dog's head. Below it, the title "Dog Breed Identification" and subtitle "Determine the breed of a dog in an image". A "Kaggle · 1,286 teams · 4 months ago" badge is present. A navigation bar includes "Overview" (underlined), Data, Kernels, Discussion, Leaderboard, and Rules. The "Overview" section contains a "Description" box with text about identifying good dogs and a "Evaluation" box with text about the competition's goals. Below these are two rows of small images showing various dog breeds.

图 9.16: 狗的品种识别比赛的网页信息。比赛数据集可通过点击“Data”标签获取

首先，导入实验所需的包或模块。

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

        import collections
        import datetime
        import gluonbook as gb
        import math
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import data as gdata, loss as gloss, model_zoo, nn
        from mxnet.gluon.data.vision import transforms
        import numpy as np
        import os
        import shutil
        import zipfile
```

9.12.1 获取数据集

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。两个数据集中的图片格式都是 jpg。这些图片都含有 RGB 三个通道（彩色），高和宽的大小不一。训练集中狗的类别共有 120 种，例如拉布拉多、贵宾、腊肠、萨摩耶、哈士奇、吉娃娃和约克夏。

下载数据集

登录 Kaggle 后，我们可以点击图 9.X 所示的狗的品种识别比赛网页上的“Data”标签，并分别下载训练数据集“train.zip”、测试数据集“test.zip”和训练数据集标签“label.csv.zip”。下载完成后，将它们分别存放在以下路径：

- .. /data/kaggle_dog/train.zip
- .. /data/kaggle_dog/test.zip
- .. /data/kaggle_dog/labels.csv.zip

为方便快速上手，我们提供了上述数据集的小规模采样“train_valid_test_tiny.zip”。如果你将使用上述 Kaggle 比赛的完整数据集，还需要把下面 demo 变量改为 False。

```
In [2]: # 如果使用下载的 Kaggle 比赛的完整数据集，把下面改为 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']

for f in zipfiles:
    with zipfile.ZipFile(data_dir + '/' + f, 'r') as z:
        z.extractall(data_dir)

```

整理数据集

我们定义下面的 `reorg_dog_data` 函数来整理 Kaggle 比赛的完整数据集。整理后，同一类狗的图片将被放在同一个文件夹下，便于我们稍后读取。该函数中的参数 `valid_ratio` 是验证集中每类狗的样本数与原始训练集中数量最少一类的狗的样本数（66）之比。

```

In [3]: 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())

    n_train = len(os.listdir(os.path.join(data_dir, train_dir)))
    # 训练集中数量最少一类的狗的样本数。
    min_n_train_per_label = (
        collections.Counter(idx_label.values()).most_common()[:-2:-1][0][1])
    # 验证集中每类狗的样本数。
    n_valid_per_label = math.floor(min_n_train_per_label * valid_ratio)
    label_count = {}

    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] < n_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'))

```

我们在这里仅仅使用小数据集。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 比赛的完整数据集并调用 `reorg_dog_data` 函数整理数据集。同时，我们也需要将批量大小 `batch_size` 设为一个较大的整数，例如 128。

```

In [4]: if demo:
    # 注意：此处使用小数据集。
    input_dir = 'train_valid_test_tiny'
    # 注意：此处将批量大小相应设小。使用 Kaggle 比赛的完整数据集时可设较大整数。
    batch_size = 1
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)

```

9.12.2 图片增广

为应对过拟合，我们在这里使用 `transforms` 来增广数据集。例如，加入 `transforms.RandomFlipLeftRight()` 即可随机对图片做镜面反转。我们也通过 `transforms.Normalize()` 对彩色图像 RGB 三个通道分别做标准化。以下列举了部分操作。这些操作可以

根据需求来决定是否使用或修改。

```
In [5]: transform_train = transforms.Compose([
    # 随机对图片裁剪出面积为原图片面积 0.08 到 1 倍之间、且高和宽之比在 3/4 和 4/3 之间
    # 的图片，再放缩为高和宽均为 224 像素的新图片。
    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(),
    # 随机抖动亮度、对比度和饱和度。
    transforms.RandomColorJitter(brightness=0.4, contrast=0.4,
                                saturation=0.4),
    # 随机加噪音。
    transforms.RandomLighting(0.1),

    # 将图片像素值按比例缩小到 0 和 1 之间，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])
])

# 测试时，只使用确定性的图像预处理操作。
transform_test = transforms.Compose([
    transforms.Resize(256),
    # 将图片中央的高和宽均为 224 的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 `ImageFolderDataset` 类来读取整理后的数据集，其中每个数据样本包括图像和标签。需要注意的是，我们要在 `DataLoader` 中调用刚刚定义好的图片增广函数。其中 `transform_first` 函数指明对每个数据样本中的图像做数据增广。

```
In [6]: # 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 = gdata.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'train'), flag=1)
valid_ds = gdata.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'valid'), flag=1)
train_valid_ds = gdata.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'train_valid'), flag=1)
test_ds = gdata.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'test'), flag=1)
```

```
train_data = gdata.DataLoader(train_ds.transform_first(transform_train),
                               batch_size, shuffle=True, last_batch='keep')
valid_data = gdata.DataLoader(valid_ds.transform_first(transform_test),
                               batch_size, shuffle=True, last_batch='keep')
train_valid_data = gdata.DataLoader(train_valid_ds.transform_first(
    transform_train), batch_size, shuffle=True, last_batch='keep')
test_data = gdata.DataLoader(test_ds.transform_first(transform_test),
                             batch_size, shuffle=False, last_batch='keep')
```

9.12.3 定义模型并使用微调

这个比赛的数据属于 ImageNet 数据集的子集，因此我们可以应用“[微调](#)”一节中介绍的思路，选用在 ImageNet 完整数据集上预训练过的模型，并通过微调在比赛数据集上进行训练。Gluon 提供了丰富的预训练模型，我们在这里以预训练过的 ResNet-34 模型为例。由于比赛数据集属于预训练数据集的子集，我们可以重用预训练模型在输出层的输入（即特征），并将原输出层替换成新的可以训练的小规模输出网络，例如两个串联的全连接层。由于预训练模型的参数在训练中是固定的，我们既节约了它们的训练时间，又节省了存储它们梯度所需的空间。

需要注意的是，我们在图片增广中使用了 ImageNet 数据集上 RGB 三个通道的均值和标准差做标准化，这和预训练模型所做的标准化是一致的。

```
In [7]: def get_net(ctx):
    # 设 pretrained=True 就能获取预训练模型的参数。第一次使用时需要联网下载。
    finetune_net = model_zoo.vision.resnet34_v2(pretrained=True)
    # 定义新的输出网络。
    finetune_net.output_new = nn.HybridSequential(prefix='')
    finetune_net.output_new.add(nn.Dense(256, activation='relu'))
    # 120 是输出的类别数。
    finetune_net.output_new.add(nn.Dense(120))
    # 初始化输出网络。
    finetune_net.output_new.initialize(init.Xavier(), ctx=ctx)
    # 把模型参数分配到即将用于计算的 CPU 或 GPU 上。
    finetune_net.collect_params().reset_ctx(ctx)
    return finetune_net
```

9.12.4 定义训练函数

我们将依赖模型在验证集上的表现来选择模型并调节超参数。模型的训练函数 `train` 只会训练我们定义的输出网络。我们记录了每个迭代周期的训练时间。这有助于比较不同模型的时间开销。

```

In [8]: loss = gloss.SoftmaxCrossEntropyLoss()

def get_loss(data, net, ctx):
    l = 0.0
    for X, y in data:
        y = y.as_in_context(ctx)
        # 计算预训练模型输出层的输入，即特征。
        output_features = net.features(X.as_in_context(ctx))
        # 将特征作为我们定义的输出网络的输入，计算输出。
        outputs = net.output_new(output_features)
        l += loss(outputs, y).mean().asscalar()
    return l / len(data)

def train(net, train_data, valid_data, num_epochs, lr, wd, ctx, lr_period,
          lr_decay):
    # 只训练我们定义的输出网络。
    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_l = 0.0
        if epoch > 0 and epoch % lr_period == 0:
            trainer.set_learning_rate(trainer.learning_rate * lr_decay)
        for X, y in train_data:
            y = y.astype('float32').as_in_context(ctx)
            # 计算预训练模型输出层的输入，即特征。
            output_features = net.features(X.as_in_context(ctx))
            with autograd.record():
                # 将特征作为我们定义的输出网络的输入，计算输出。
                outputs = net.output_new(output_features)
                l = loss(outputs, y)
            # 反向传播只发生在我们定义的输出网络上。
            l.backward()
            trainer.step(batch_size)
            train_l += l.mean().asscalar()
        cur_time = datetime.datetime.now()
        h, remainder = divmod((cur_time - prev_time).seconds, 3600)
        m, s = divmod(remainder, 60)
        time_s = "time %02d:%02d:%02d" % (h, m, s)
        if valid_data is not None:
            valid_loss = get_loss(valid_data, net, ctx)
            epoch_s = ("epoch %d, train loss %f, valid loss %f, "
                      "% (epoch, train_l / len(train_data), valid_loss))
```

```

    else:
        epoch_s = ("epoch %d, train loss %f, "
                   "% (epoch, train_l / len(train_data)))"
        prev_time = cur_time
        print(epoch_s + time_s + ', lr ' + str(trainer.learning_rate))

```

9.12.5 训练并验证模型

现在，我们可以训练并验证模型了。以下的超参数都是可以调节的，例如增加迭代周期。

```

In [9]: ctx = gb.try_gelu()
num_epochs = 1
# 学习率。
lr = 0.01
# 权重衰减参数。
wd = 1e-4
# 优化算法的学习率将在每 10 个迭代周期时自乘 0.1。
lr_period = 10
lr_decay = 0.1

net = get_net(ctx)
net.hybridize()
train(net, train_data, valid_data, num_epochs, lr, wd, ctx, lr_period,
      lr_decay)

epoch 0, train loss 5.213856, valid loss 4.805326, time 00:00:02, lr 0.01

```

9.12.6 对测试集分类并在 Kaggle 提交结果

当得到一组满意的模型设计和超参数后，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输出网络做预测。

```

In [10]: net = get_net(ctx)
net.hybridize()
train(net, train_valid_data, None, num_epochs, lr, wd, ctx, lr_period,
      lr_decay)

preds = []
for data, label in test_data:
    # 计算预训练模型输出层的输入，即特征。
    output_features = net.features(data.as_in_context(ctx))
    # 将特征作为我们定义的输出网络的输入，计算输出。

```

```
        output = nd.softmax(net.output_new(output_features))
        preds.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, preds):
            f.write(i.split('.')[0] + ',' + ','.join(
                [str(num) for num in output]) + '\n')

epoch 0, train loss 5.055686, time 00:00:03, lr 0.01
```

执行完上述代码后，会生成一个“submission.csv”文件。这个文件符合 Kaggle 比赛要求的提交格式。这时我们可以在 Kaggle 上把对测试集分类的结果提交并查看分类准确率。你需要登录 Kaggle 网站，访问 ImageNet Dogs 比赛网页，并点击右侧“Submit Predictions”或“Late Submission”按钮。然后，点击页面下方“Upload Submission File”选择需要提交的分类结果文件。最后，点击页面最下方的“Make Submission”按钮就可以查看结果了。

9.12.7 小结

- 我们可以使用在 ImageNet 数据集上预训练的模型并微调，从而以较小的计算开销对 ImageNet 的子集数据集做分类。

9.12.8 练习

- 使用 Kaggle 完整数据集，把批量大小 batch_size 和迭代周期数 num_epochs 分别调大些，可以在 Kaggle 上拿到什么样的结果？
- 使用更深的预训练模型并微调，你能获得更好的结果吗？
- 扫码直达讨论区，在社区交流方法和结果。你能发掘出其他更好的技巧吗？

9.12.9 扫码直达讨论区



9.12.10 参考文献

[1] Kaggle ImageNet Dogs 比赛网址。 <https://www.kaggle.com/c/dog-breed-identification>

自然语言处理

自然语言处理关注计算机与人类自然语言的交互。在实际中，我们常常使用自然语言处理技术，例如“循环神经网络”一章中介绍的语言模型，来处理和分析大量的自然语言数据。

本章中，我们将先介绍如何用向量表示词，并应用这些词向量寻找近义词和类比词。接着，在文本分类任务中，我们进一步应用词向量分析文本情感。此外，自然语言处理任务中很多输出是不定长的，例如任意长度的句子。我们将描述应对这类问题的编码器—解码器模型以及注意力机制，并将它们应用于机器翻译中。

10.1 词嵌入：word2vec

自然语言是一套用来表达含义的复杂系统。在这套系统中，词是表义的基本单元。顾名思义，词向量是用来表示词的向量，也可被认为是词的特征向量。这通常需要把维数为词典大小的高维空间嵌入到一个更低维数的连续向量空间。把词映射为实数域上向量的技术也叫词嵌入（word embedding）。近年来，词嵌入已逐渐成为自然语言处理的基础知识。那么，我们应该如何使用向量表示词呢？

10.1.1 为何不采用 one-hot 向量

我们在“[循环神经网络](#)”一节中使用 one-hot 向量表示词（字符为词）。回忆一下，假设词典中不同词的数量（词典大小）为 N ，每个词可以和从 0 到 $N - 1$ 的连续整数一一对应。这些与词对应的整数也叫词的索引。假设一个词的索引为 i ，为了得到该词的 one-hot 向量表示，我们创建一个全 0 的长为 N 的向量，并将其第 i 位设成 1。

然而，使用 one-hot 词向量通常并不是一个好选择。一个主要的原因是，one-hot 词向量无法表达不同词之间的相似度，例如余弦相似度。由于任意一对向量 $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ 的余弦相似度为

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \in [-1, 1],$$

任何一对词的 one-hot 向量的余弦相似度都为 0。

10.1.2 word2vec

2013 年，Google 团队发表了 word2vec 工具 [1]。word2vec 工具主要包含两个模型：跳字模型（skip-gram）和连续词袋模型（continuous bag of words，简称 CBOW），以及两种近似训练法：负采样（negative sampling）和层序 softmax（hierarchical softmax）。值得一提的是，word2vec 的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嵌入模型。本节将重点介绍 word2vec 的模型和训练方法。

10.1.3 模型

word2vec 工具主要包含跳字模型和连续词袋模型。下面将分别介绍它们。

跳字模型

在跳字模型中，我们用一个词来预测它在文本序列周围的词。举个例子，假设文本序列是“the”、“man”、“loves”、“his”和“son”。以“loves”作为中心词，设时间窗口大小为 2。跳字模型所关心的是，给定中心词“loves”生成与它距离不超过 2 个词的背景词“the”、“man”、“his”和“son”的条件概率。

我们来描述一下跳字模型。

假设词典索引集 \mathcal{V} 的大小为 $|\mathcal{V}|$, 且 $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}| - 1\}$ 。给定一个长度为 T 的文本序列中, 时
间步 t 的词为 $w^{(t)}$ 。当时间窗口大小为 m 时, 跳字模型需要最大化给定任一中心词生成所有背景
词的概率

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} \mathbb{P}(w^{(t+j)} | w^{(t)}).$$

上式的最大似然估计与最小化以下损失函数等价:

$$-\frac{1}{T} \sum_{t=1}^T \sum_{-m \leq j \leq m, j \neq 0} \log \mathbb{P}(w^{(t+j)} | w^{(t)}).$$

我们可以用 \mathbf{v} 和 \mathbf{u} 分别表示中心词和背景词的向量。换言之, 对于词典中索引为 i 的词, 它在作
为中央词和背景词时的向量表示分别是 \mathbf{v}_i 和 \mathbf{u}_i 。而词典中所有词的这两种向量正是跳字模型所
要学习的模型参数。为了将模型参数植入损失函数, 我们需要使用模型参数表达损失函数中的给
定中心词生成背景词的条件概率。给定中心词, 假设生成各个背景词是相互独立的。设中心词 w_c
在词典中索引为 c , 背景词 w_o 在词典中索引为 o , 损失函数中的给定中心词生成背景词的条件概
率可以通过 softmax 函数定义为

$$\mathbb{P}(w_o | w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}.$$

当序列长度 T 较大时, 我们通常在每次迭代时随机采样一个较短的子序列来计算有关该子序列的
损失。然后, 根据该损失计算词向量的梯度并迭代词向量。具体算法可以参考 “[梯度下降和随机梯度下降](#)” 一节。作为一个具体的例子, 下面我们看看如何计算随机采样的子序列的损失有关
中心词向量的梯度。和上面提到的长度为 T 的文本序列的损失函数类似, 随机采样的子序列的损
失实际上是对子序列中给定中心词生成背景词的条件概率的对数求平均。通过微分, 我们可以得
到上式中条件概率的对数有关中心词向量 \mathbf{v}_c 的梯度

$$\frac{\partial \log \mathbb{P}(w_o | w_c)}{\partial \mathbf{v}_c} = \mathbf{u}_o - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \mathbf{u}_j.$$

该式也可写作

$$\frac{\partial \log \mathbb{P}(w_o | w_c)}{\partial \mathbf{v}_c} = \mathbf{u}_o - \sum_{j \in \mathcal{V}} \mathbb{P}(w_j | w_c) \mathbf{u}_j.$$

随机采样的子序列有关其他词向量的梯度同理可得。训练模型时, 每一次迭代实际上是用这些梯
度来迭代子序列中出现过的中心词和背景词的向量。训练结束后, 对于词典中的任一索引为 i 的
词, 我们均得到该词作为中心词和背景词的两组词向量 \mathbf{v}_i 和 \mathbf{u}_i 。在自然语言处理应用中, 我们
会使用跳字模型的中心词向量。

连续词袋模型

连续词袋模型与跳字模型类似。与跳字模型最大的不同是，连续词袋模型用一个中心词在文本序列前后的背景词来预测该中心词。举个例子，假设文本序列为“the”、“man”、“loves”、“his”和“son”。以“loves”作为中心词，设时间窗口大小为2。连续词袋模型所关心的是，给定与中心词距离不超过2个词的背景词“the”、“man”、“his”和“son”生成中心词“loves”的条件概率。

假设词典索引集 \mathcal{V} 的大小为 $|\mathcal{V}|$ ，且 $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}| - 1\}$ 。给定一个长度为 T 的文本序列中，时间步 t 的词为 $w^{(t)}$ 。当时间窗口大小为 m 时，连续词袋模型需要最大化由背景词生成任一中心词的概率

$$\prod_{t=1}^T \mathbb{P}(w^{(t)} | w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}).$$

上式的最大似然估计与最小化以下损失函数等价：

$$-\sum_{t=1}^T \log \mathbb{P}(w^{(t)} | w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}).$$

我们可以用 \mathbf{v} 和 \mathbf{u} 分别表示背景词和中心词的向量（注意符号和跳字模型中的不同）。换言之，对于词典中索引为 i 的词，它在作为背景词和中心词时的向量表示分别是 \mathbf{v}_i 和 \mathbf{u}_i 。而词典中所有词的这两种向量正是连续词袋模型所要学习的模型参数。为了将模型参数植入损失函数，我们需要使用模型参数表达损失函数中的给定背景词生成中心词的概率。设中心词 w_c 在词典中索引为 c ，背景词 $w_{o_1}, \dots, w_{o_{2m}}$ 在词典中索引为 o_1, \dots, o_{2m} ，损失函数中的给定背景词生成中心词的概率可以通过softmax函数定义为

$$\mathbb{P}(w_c | w_{o_1}, \dots, w_{o_{2m}}) = \frac{\exp(\mathbf{u}_c^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})/(2m))}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})/(2m))}.$$

和跳字模型一样，当序列长度 T 较大时，我们通常在每次迭代时随机采样一个较短的子序列来计算有关该子序列的损失。然后，根据该损失计算词向量的梯度并迭代词向量。通过微分，我们可以计算出上式中条件概率的对数有关任一背景词向量 \mathbf{v}_{o_i} ($i = 1, \dots, 2m$) 的梯度为：

$$\frac{\partial \log \mathbb{P}(w_c | w_{o_1}, \dots, w_{o_{2m}})}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \mathbf{u}_j \right).$$

该式也可写作

$$\frac{\partial \log \mathbb{P}(w_c | w_{o_1}, \dots, w_{o_{2m}})}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} \mathbb{P}(w_j | w_c) \mathbf{u}_j \right).$$

随机采样的子序列有关其他词向量的梯度同理可得。和跳字模型一样，训练结束后，对于词典中的任一索引为 i 的词，我们均得到该词作为背景词和中心词的两组词向量 \mathbf{v}_i 和 \mathbf{u}_i 。在自然语言处理应用中，我们会使用连续词袋模型的背景词向量。

10.1.4 近似训练法

我们可以看到，无论是跳字模型还是连续词袋模型，每一步梯度计算的开销与词典 \mathcal{V} 的大小相关。当词典较大时，例如几十万到上百万，这种训练方法的计算开销会较大。因此，我们将使用近似的方法来计算这些梯度，从而减小计算开销。常用的近似训练法包括负采样和层序 softmax。

负采样

我们以跳字模型为例讨论负采样。

实际上，词典 \mathcal{V} 的大小之所以会在损失中出现，是因为给定中心词 w_c 生成背景词 w_o 的条件概率 $\mathbb{P}(w_o | w_c)$ 使用了 softmax 运算，而 softmax 运算正是考虑了背景词可能是词典中的任一词，并体现在分母上。

不妨换个角度考虑给定中心词生成背景词的条件概率。我们先定义噪声词分布 $\mathbb{P}(w)$ ，接着假设给定中心词 w_c 生成背景词 w_o 由以下相互独立事件联合组成来近似：

- 中心词 w_c 和背景词 w_o 同时出现时间窗口。
- 中心词 w_c 和第 1 个噪声词 w_1 不同时出现在该时间窗口（噪声词 w_1 按噪声词分布 $\mathbb{P}(w)$ 随机生成，且假设一定和 w_c 不同时出现在该时间窗口）。
- ...
- 中心词 w_c 和第 K 个噪声词 w_K 不同时出现在该时间窗口（噪声词 w_K 按噪声词分布 $\mathbb{P}(w)$ 随机生成，且假设一定和 w_c 不同时出现在该时间窗口）。

下面，我们可以使用 $\sigma(x) = 1/(1 + \exp(-x))$ 函数来表达中心词 w_c 和背景词 w_o 同时出现在该训练数据窗口的概率：

$$\mathbb{P}(D = 1 | w_o, w_c) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c).$$

那么，给定中心词 w_c 生成背景词 w_o 的条件概率的对数可以近似为

$$\log \mathbb{P}(w_o | w_c) = \log \left(\mathbb{P}(D = 1 | w_o, w_c) \prod_{k=1, w_k \sim \mathbb{P}(w)}^K \mathbb{P}(D = 0 | w_k, w_c) \right).$$

假设噪声词 w_k 在词典中的索引为 i_k , 上式可改写为

$$\log \mathbb{P}(w_o | w_c) = \log \frac{1}{1 + \exp(-\mathbf{u}_o^\top \mathbf{v}_c)} + \sum_{k=1, w_k \sim \mathbb{P}(w)}^K \log \left(1 - \frac{1}{1 + \exp(-\mathbf{u}_{i_k}^\top \mathbf{v}_c)} \right).$$

因此, 有关给定中心词 w_c 生成背景词 w_o 的损失是

$$-\log \mathbb{P}(w_o | w_c) = -\log \frac{1}{1 + \exp(-\mathbf{u}_o^\top \mathbf{v}_c)} - \sum_{k=1, w_k \sim \mathbb{P}(w)}^K \log \frac{1}{1 + \exp(\mathbf{u}_{i_k}^\top \mathbf{v}_c)}.$$

假设词典 \mathcal{V} 很大, 每次迭代的计算开销由 $\mathcal{O}(|\mathcal{V}|)$ 变为 $\mathcal{O}(K)$ 。当我们把 K 取较小值时, 负采样每次迭代的计算开销将较小。

当然, 我们也可以对连续词袋模型进行负采样。有关给定背景词 $w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}$ 生成中心词 w_c 的损失

$$-\log \mathbb{P}(w^{(t)} | w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)})$$

在负采样中可以近似为

$$-\log \frac{1}{1 + \exp(-\mathbf{u}_c^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})/(2m))} - \sum_{k=1, w_k \sim \mathbb{P}(w)}^K \log \frac{1}{1 + \exp((\mathbf{u}_{i_k}^\top (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})/(2m)))}.$$

同样, 当我们把 K 取较小值时, 负采样每次迭代的计算开销将较小。

10.1.5 层序 softmax

层序 softmax 是另一种常用的近似训练法。它利用了二叉树这一数据结构。树的每个叶子节点代表着词典 \mathcal{V} 中的每个词。我们以图 10.1 为例来描述层序 softmax 的工作机制。

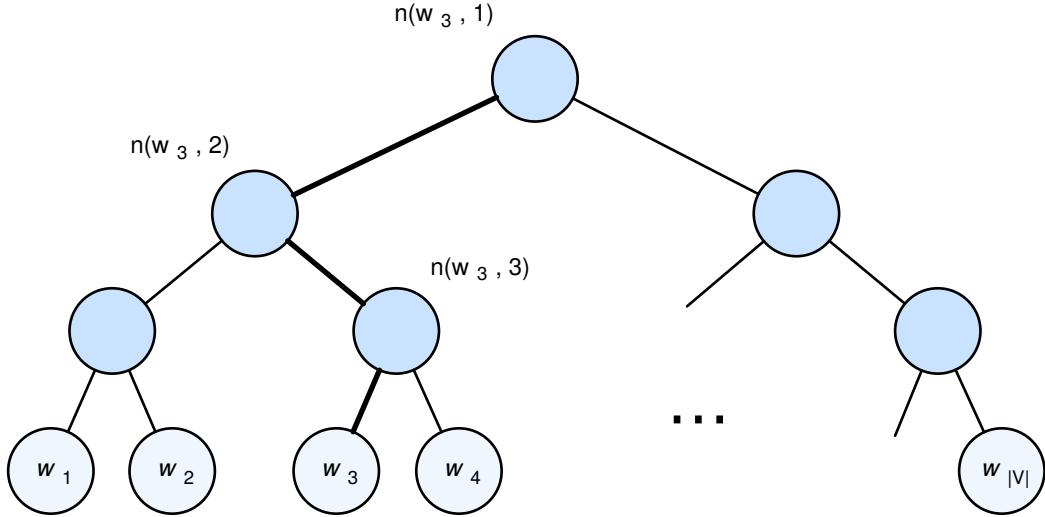


图 10.1: 层序 softmax。树的每个叶子节点代表着词典的每个词。

假设 $L(w)$ 为从二叉树的根节点到词 w 的叶子节点的路径（包括根和叶子节点）上的节点数。设 $n(w, j)$ 为该路径上第 j 个节点，并设该节点的向量为 $\mathbf{u}_{n(w,j)}$ 。以图 10.1 为例， $L(w_3) = 4$ 。设词典中的词 w_i 的词向量为 \mathbf{v}_i 。那么，跳字模型和连续词袋模型所需要计算的给定词 w_i 生成词 w 的条件概率为：

$$\mathbb{P}(w | w_i) = \prod_{j=1}^{L(w)-1} \sigma \left(\llbracket n(w, j+1) = \text{leftChild}(n(w, j)) \rrbracket \cdot \mathbf{u}_{n(w,j)}^\top \mathbf{v}_i \right),$$

其中 $\sigma(x) = 1/(1 + \exp(-x))$ ， $\text{leftChild}(n)$ 是节点 n 的左孩子节点，如果判断 x 为真， $\llbracket x \rrbracket = 1$ ；反之 $\llbracket x \rrbracket = -1$ 。由于 $\sigma(x) + \sigma(-x) = 1$ ，给定词 w_i 生成词典 \mathcal{V} 中任一词的条件概率之和为 1 这一条件也将满足：

$$\sum_{w \in \mathcal{V}} \mathbb{P}(w | w_i) = 1.$$

让我们计算图 10.1 中给定词 w_i 生成词 w_3 的条件概率。我们需要将 w_i 的词向量 \mathbf{v}_i 和根节点到 w_3 路径上的非叶子节点向量一一求内积。由于在二叉树中由根节点到叶子节点 w_3 的路径上需要向左、向右、再向左地遍历（图 10.1 中加粗的路径），我们得到

$$\mathbb{P}(w_3 | w_i) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_i) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_i) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_i).$$

在使用 softmax 的跳字模型和连续词袋模型中，词向量和二叉树中非叶子节点向量是需要学习的模型参数。假设词典 \mathcal{V} 很大，每次迭代的计算开销由 $\mathcal{O}(|\mathcal{V}|)$ 下降至 $\mathcal{O}(\log_2 |\mathcal{V}|)$ 。

10.1.6 小结

- 词向量是用于表示自然语言中词的语义的向量。
- word2vec 工具中的跳字模型和连续词袋模型通常使用近似训练法，例如负采样和层序 softmax，从而减小训练的计算开销。

10.1.7 练习

- 噪声词采样概率 $\mathbb{P}(w)$ 在实际中被建议设为 w 词频与总词频的比的 $3/4$ 次方。这样做有什么好处？提示：想想 $0.99^{3/4}$ 和 $0.01^{3/4}$ 的大小。
- 一些“the”和“a”之类的英文高频词会对结果产生什么影响？该如何处理？提示：可参考 word2vec 论文第 2.3 节 [2]。
- 如何训练包括例如“new york”在内的词组向量？提示：可参考 word2vec 论文第 4 节 [2]。

10.1.8 扫码直达讨论区



10.1.9 参考文献

[1] word2vec 工具. <https://code.google.com/archive/p/word2vec/>

[2] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems (pp. 3111-3119).

10.2 词嵌入：GloVe 和 fastText

在 word2vec 被提出以后，很多其他词嵌入模型也陆续发表。本节介绍其中比较有代表性的两个模型。它们分别是由斯坦福团队发表的 GloVe 和由 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 \mathbf{X} 为共现词频矩阵，其中元素 x_{ij} 为词 j 出现在词 i 的背景的次数。这里的“背景”有多种可能的定义。举个例子，在一段文本序列中，如果词 j 出现在词 i 左边或者右边不超过 10 个词的距离，我们可以认为词 j 出现在词 i 的背景一次。令 $x_i = \sum_k x_{ik}$ 为任意词出现在词 i 的背景的次数。那么，

$$p_{ij} = \mathbb{P}(j | i) = \frac{x_{ij}}{x_i}$$

为词 j 在词 i 的背景中出现的条件概率。这一条件概率也称词 i 和词 j 的共现概率。

共现概率比值

GloVe 论文里展示了以下词对的共现概率与比值 [1]：

- $\mathbb{P}(k | \text{ice})$: 0.00019 ($k = \text{solid}$), 0.000066 ($k = \text{gas}$), 0.003 ($k = \text{water}$), 0.000017 ($k = \text{fashion}$)
- $\mathbb{P}(k | \text{steam})$: 0.000022 ($k = \text{solid}$), 0.00078 ($k = \text{gas}$), 0.0022 ($k = \text{water}$), 0.000018 ($k = \text{fashion}$)
- $\mathbb{P}(k | \text{ice}) / \mathbb{P}(k | \text{steam})$: 8.9 ($k = \text{solid}$), 0.085 ($k = \text{gas}$), 1.36 ($k = \text{water}$), 0.96 ($k = \text{fashion}$)

我们可以观察到以下现象：

- 对于与 ice（冰）相关而与 steam（蒸汽）不相关的词 k , 例如 $k = \text{solid}$ （固体），我们期望共现概率比值 p_{ik}/p_{jk} 较大，例如上面最后一行结果中的值 8.9。
- 对于与 ice 不相关而与 steam 相关的词 k , 例如 $k = \text{gas}$ （气体），我们期望共现概率比值 p_{ik}/p_{jk} 较小，例如上面最后一行结果中的值 0.085。
- 对于与 ice 和 steam 都相关的词 k , 例如 $k = \text{water}$ （水），我们期望共现概率比值 p_{ik}/p_{jk} 接近 1，例如上面最后一行结果中的值 1.36。
- 对于与 ice 和 steam 都不相关的词 k , 例如 $k = \text{fashion}$ （时尚），我们期望共现概率比值 p_{ik}/p_{jk} 接近 1，例如上面最后一行结果中的值 0.96。

由此可见，共现概率比值能比较直观地表达词与词之间的关系。GloVe 试图用有关词向量的函数来表达共现概率比值。

用词向量表达共现概率比值

GloVe 的核心思想在于使用词向量表达共现概率比值。而任意一个这样的比值需要三个词 i 、 j 和 k 的词向量。对于共现概率 $p_{ij} = \mathbb{P}(j | i)$ ，我们称词 i 和词 j 分别为中心词和背景词。我们使用 \mathbf{v}_i 和 $\tilde{\mathbf{v}}_i$ 分别表示词 i 作为中心词和背景词的词向量。

我们可以用有关词向量的函数 f 来表达共现概率比值：

$$f(\mathbf{v}_i, \mathbf{v}_j, \tilde{\mathbf{v}}_k) = \frac{p_{ik}}{p_{jk}}.$$

需要注意的是，函数 f 可能的设计并不唯一。下面我们考虑一种较为合理可能性。

首先，用向量之差来表达共现概率的比值，并将上式改写成

$$f(\mathbf{v}_i - \mathbf{v}_j, \tilde{\mathbf{v}}_k) = \frac{p_{ik}}{p_{jk}}.$$

由于共现概率比值是一个标量，我们可以使用向量之间的内积把函数 f 的自变量进一步改写，得到

$$f((\mathbf{v}_i - \mathbf{v}_j)^\top \tilde{\mathbf{v}}_k) = \frac{p_{ik}}{p_{jk}}.$$

由于任意一对词共现的对称性，我们希望以下两个性质可以同时被满足：

- 任意词作为中心词和背景词的词向量应该相等：对任意词 i ， $\mathbf{v}_i = \tilde{\mathbf{v}}_i$ 。
- 词与词之间共现词频矩阵 \mathbf{X} 应该对称：对任意词 i 和 j ， $x_{ij} = x_{ji}$ 。

为了满足以上两个性质，一方面，我们令

$$f((\mathbf{v}_i - \mathbf{v}_j)^\top \tilde{\mathbf{v}}_k) = \frac{f(\mathbf{v}_i^\top \tilde{\mathbf{v}}_k)}{f(\mathbf{v}_j^\top \tilde{\mathbf{v}}_k)},$$

并得到 $f(x) = \exp(x)$ 。以上两式的右边联立，

$$f(\mathbf{v}_i^\top \tilde{\mathbf{v}}_k) = \exp(\mathbf{v}_i^\top \tilde{\mathbf{v}}_k) = p_{ik} = \frac{x_{ik}}{x_i}.$$

由上式可得

$$\mathbf{v}_i^\top \tilde{\mathbf{v}}_k = \log(p_{ik}) = \log(x_{ik}) - \log(x_i).$$

另一方面，我们可以把上式中 $\log(x_i)$ 替换成两个偏差项之和 $b_i + \tilde{b}_k$ ，得到

$$\mathbf{v}_i^\top \tilde{\mathbf{v}}_k = \log(x_{ik}) - b_i - \tilde{b}_k.$$

因此，对于任意一对词 i 和 j ，我们可以用它们的词向量表达共现词频的对数：

$$\mathbf{v}_i^\top \tilde{\mathbf{v}}_j + b_i + \tilde{b}_j = \log(x_{ij}).$$

损失函数

GloVe 中的共现词频是直接在训练数据上统计得到的。为了学习词向量和相应的偏差项，我们希望上式中的左边与右边尽可能接近。给定词典 \mathcal{V} 和权重函数 $h(x_{ij})$ ，GloVe 的损失函数为

$$\sum_{i \in \mathcal{V}, j \in \mathcal{V}} h(x_{ij}) \left(\mathbf{v}_i^\top \tilde{\mathbf{v}}_j + b_i + \tilde{b}_j - \log(x_{ij}) \right)^2,$$

其中权重函数 $h(x)$ 的一个建议选择是，当 $x < c$ （例如 $c = 100$ ），令 $h(x) = (x/c)^\alpha$ （例如 $\alpha = 0.75$ ），反之令 $h(x) = 1$ 。由于权重函数的存在，损失函数的计算复杂度与共现词频矩阵 \mathbf{X} 中非零元素的数目呈正相关。我们可以从 \mathbf{X} 中随机采样小批量非零元素，并使用优化算法迭代共现词频相关词的向量和偏差项。

我们提到过，任意词的中心词向量和背景词向量是等价的。但由于初始化值的不同，同一个词最终学习到的两组词向量可能不同。当学习得到所有词向量以后，GloVe 使用一个词的中心词向量与背景词向量之和作为该词的最终词向量。

10.2.2 fastText

我们在上一节介绍了 word2vec 的跳字模型和负采样。fastText 以跳字模型为基础，将每个中心词视为子词（subword）的集合，并使用负采样学习子词的词向量。因此，fastText 是一个子词嵌入模型。

举个例子，设子词长度为 3 个字符，“where”的子词包括“<wh”、“whe”、“her”、“ere”、“re>”和特殊子词（整词）“<where>”。这些子词中的“<”和“>”符号是为了将作为前后缀的子词区分出来。并且，这里的子词“her”与整词“<her>”也可被区分开。给定一个词 w ，我们通常可以把字符长度在 3 到 6 之间的所有子词和特殊子词的并集 \mathcal{G}_w 取出。假设词典中任意子词 g 的子词向量为 \mathbf{z}_g ，我们可以把使用负采样的跳字模型的损失函数

$$-\log \mathbb{P}(w_o | w_c) = -\log \frac{1}{1 + \exp(-\mathbf{u}_o^\top \mathbf{v}_c)} - \sum_{k=1, w_k \sim \mathbb{P}(w)}^K \log \frac{1}{1 + \exp(\mathbf{u}_{i_k}^\top \mathbf{v}_c)}$$

直接替换成

$$-\log \mathbb{P}(w_o | w_c) = -\log \frac{1}{1 + \exp(-\mathbf{u}_o^\top \sum_{g \in \mathcal{G}_{w_c}} \mathbf{z}_g)} - \sum_{k=1, w_k \sim \mathbb{P}(w)}^K \log \frac{1}{1 + \exp(\mathbf{u}_{i_k}^\top \sum_{g \in \mathcal{G}_{w_c}} \mathbf{z}_g)}.$$

可以看到，原中心词向量被替换了成了中心词的子词向量之和。与 word2vec 和 GloVe 不同，词典以外的新词的词向量可以使用 fastText 中相应的子词向量之和。

fastText 对于一些语言较重要，例如阿拉伯语、德语和俄语。例如，德语中有很多复合词，例如乒乓球（英文 table tennis）在德语中叫“Tischtennis”。fastText 可以通过子词表达两个词的相关性，例如“Tischtennis”和“Tennis”。

由于词是自然语言的基本表义单元，词向量广泛应用于各类自然语言处理的场景中。例如，我们可以在之前介绍的语言模型中使用在更大规模语料上预训练的词向量，从而提升语言模型的准确性。

10.2.3 小结

- GloVe 用词向量表达共现词频的对数。
- fastText 用子词向量之和表达整词。它能表示词典以外的新词。

10.2.4 练习

- GloVe 中，如果一个词出现在另一个词的背景中，是否可以利用它们之间在文本序列的距离重新设计词频计算方式？提示：可参考 Glove 论文 4.2 节 [1]。
- 如果丢弃 GloVe 中的偏差项，是否也可以满足任意一对词共现的对称性？
- 在 fastText 中，子词过多怎么办（例如，6 字英文组合数为 26^6 ）？提示：可参考 fastText 论文 3.2 节 [2]。

10.2.5 扫码直达讨论区



10.2.6 参考文献

- [1] Pennington, J., Socher, R., & Manning, C. (2014). Glove: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP) (pp. 1532-1543).
- [2] Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2016). Enriching word vectors with subword information. arXiv preprint arXiv:1607.04606.

10.3 求近似词和类比词

本节介绍如何应用在大规模语料上预训练的词向量，例如求近似词和类比词。这里使用的预训练的 GloVe 和 fastText 词向量分别来自它们的项目网站 [1,2]。

首先导入实验所需的包或模块。

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

10.3.1 由数据集建立词典和载入词向量

下面，我们以 fastText 为例，由数据集建立词典并载入词向量。fastText 提供了基于不同语言的多套预训练的词向量。这些词向量是在大规模语料上训练得到的，例如维基百科语料。以下打印了其中的 10 种。

```
In [2]: print(text.embedding.get_pretrained_file_names('fasttext')[:10])
```

```
['crawl-300d-2M.vec', 'wiki_aa.vec', 'wiki_ab.vec', 'wiki_ace.vec', 'wiki_ady.vec',
 ← 'wiki_af.vec', 'wiki_ak.vec', 'wiki_als.vec', 'wiki_am.vec', 'wiki_ang.vec']
```

访问词向量

为了演示方便，我们创建一个很小的文本数据集，并计算词频。

```
In [3]: text_data = " hello world \n hello nice world \n hi world \n"
counter = text.utils.count_tokens_from_str(text_data)
```

我们先根据数据集建立词典，并为该词典中的词载入 fastText 词向量。这里使用 Simple English 的预训练词向量。

```
In [4]: my_vocab = text.vocab.Vocabulary(counter)
my_embedding = text.embedding.create(
    'fasttext', pretrained_file_name='wiki.simple.vec', vocabulary=my_vocab)

/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/mxnet/contr_
← ib/text/embedding.py:278: UserWarning: At line 1 of the pre-trained text
← embedding file: token 111051 with 1-dimensional vector [300.0] is likely a header
← and is skipped.
'skipped.' % (line_num, token, elems))
```

词典除了包括数据集中四个不同的词语，还包括一个特殊的未知词符号。打印词典大小。

```
In [5]: len(my_embedding)
```

```
Out[5]: 5
```

默认情况下，任意一个词典以外词的词向量为零向量。

```
In [6]: my_embedding.get_vecs_by_tokens('beautiful')[:10]
```

```
Out[6]:
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.]
<NDArray 10 @cpu(0)>
```

fastText 中每个词均使用 300 维的词向量。打印数据集中两个词“hello”和“world”词向量的形状。

```
In [7]: my_embedding.get_vecs_by_tokens(['hello', 'world']).shape
```

```
Out[7]: (2, 300)
```

打印“hello”和“world”词向量前五个元素。

```
In [8]: my_embedding.get_vecs_by_tokens(['hello', 'world'])[:, :5]
```

```
Out[8]:  
[[ 0.39567      0.21454     -0.035389    -0.24299     -0.095645   ]  
 [ 0.10444      -0.10858      0.27212      0.13299     -0.33164999]]  
<NDArray 2x5 @cpu(0)>
```

打印“hello”和“world”在词典中的索引。

```
In [9]: my_embedding.to_indices(['hello', 'world'])
```

```
Out[9]: [2, 1]
```

使用预训练词向量初始化 Embedding 实例

我们在“循环神经网络——使用 Gluon”一节中介绍了 Gluon 中的 Embedding 实例，并对其中每个词的向量做了随机初始化。实际上，我们还可以使用预训练的词向量初始化 Embedding 实例。

```
In [10]: layer = nn.Embedding(len(my_embedding), my_embedding.vec_len)  
layer.initialize()  
layer.weight.set_data(my_embedding.idx_to_vec)
```

使用词典中“hello”和“world”两个词在词典中的索引，我们可以通过 Embedding 实例得到它们的预训练词向量，并向神经网络的下一层传递。

```
In [11]: layer(nd.array([2, 1]))[:, :5]
```

```
Out[11]:  
[[ 0.39567      0.21454     -0.035389    -0.24299     -0.095645   ]  
 [ 0.10444      -0.10858      0.27212      0.13299     -0.33164999]]  
<NDArray 2x5 @cpu(0)>
```

10.3.2 由预训练词向量建立词典——以 GloVe 为例

除了使用数据集建立词典外，我们还可以直接由预训练词向量建立词典。

这一次我们使用 GloVe 的预训练词向量。以下打印了 GloVe 提供的各套预训练词向量。这些词向量是在大规模语料上训练得到的，例如维基百科语料和推特语料。

```
In [12]: print(text.embedding.get_pretrained_file_names('glove'))  
['glove.42B.300d.txt', 'glove.6B.50d.txt', 'glove.6B.100d.txt', 'glove.6B.200d.txt',  
 ←  'glove.6B.300d.txt', 'glove.840B.300d.txt', 'glove.twitter.27B.25d.txt',  
 ←  'glove.twitter.27B.50d.txt', 'glove.twitter.27B.100d.txt',  
 ←  'glove.twitter.27B.200d.txt']
```

我们使用 50 维的词向量。和之前不同，这里不再传入根据数据集建立的词典，而是直接使用预训练词向量中的词建立词典。

```
In [13]: glove_6b50d = text.embedding.create('glove',
                                         pretrained_file_name='glove.6B.50d.txt')
```

打印词典大小。注意其中包含一个特殊的未知词符号。

```
In [14]: print(len(glove_6b50d))
```

```
400001
```

我们可以访问词向量的属性。

```
In [15]: # 词到索引，索引到词。
```

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

```
Out[15]: (3367, 'beautiful')
```

10.3.3 应用预训练词向量

下面我们以 GloVe 为例，展示预训练词向量的应用。

首先，我们定义余弦相似度，并用它表示两个向量之间的相似度。

```
In [16]: def cos_sim(x, y):
    return nd.dot(x, y) / (x.norm() * y.norm())
```

余弦相似度的值域在 -1 到 1 之间。两个余弦相似度越大的向量越相似。

```
In [17]: x = nd.array([1, 2])
y = nd.array([10, 20])
z = nd.array([-1, -2])
cos_sim(x, y), cos_sim(x, z)
```

```
Out[17]: (
    [ 1.]
    <NDArray 1 @cpu(0)>,
    [-1.]
    <NDArray 1 @cpu(0)>)
```

求近似词

给定任意词，我们可以从 GloVe 的整个词典（大小 40 万，不含未知词符号）中找出与它最接近的 k 个词。之前已经提到，词与词之间的相似度可以用两个词向量的余弦相似度表示。

```
In [18]: def norm_vecs_by_row(x):
    # 分母中添加的 1e-10 是为了数值稳定性。
    return x / (nd.sum(x * x, axis=1) + 1e-10).sqrt().reshape((-1, 1))

def get_knn(token_embedding, k, word):
    word_vec = token_embedding.get_vecs_by_tokens([word]).reshape((-1, 1))
    vocab_vecs = norm_vecs_by_row(token_embedding.idx_to_vec)
    dot_prod = nd.dot(vocab_vecs, word_vec)
    indices = nd.topk(dot_prod.reshape((len(token_embedding), )), k=k+1,
                      ret_typ='indices')
    indices = [int(i.asscalar()) for i in indices]
    # 除去输入词。
    return token_embedding.to_tokens(indices[1:])
```

查找词典中与“baby”最近似的5个词。

```
In [19]: get_knn(glove_6b50d, 5, 'baby')
```

```
Out[19]: ['babies', 'boy', 'girl', 'newborn', 'pregnant']
```

验证一下“baby”和“babies”两个词向量之间的余弦相似度。

```
In [20]: cos_sim(glove_6b50d.get_vecs_by_tokens('baby'),
                  glove_6b50d.get_vecs_by_tokens('babies'))
```

```
Out[20]:
[ 0.83871299]
<NDArray 1 @cpu(0)>
```

查找词典中与“computers”最近似的5个词。

```
In [21]: get_knn(glove_6b50d, 5, 'computers')
```

```
Out[21]: ['computer', 'phones', 'pcs', 'machines', 'devices']
```

查找词典中与“run”最近似的5个词。

```
In [22]: get_knn(glove_6b50d, 5, 'run')
```

```
Out[22]: ['running', 'runs', 'went', 'start', 'ran']
```

查找词典中与“beautiful”最近似的5个词。

```
In [23]: get_knn(glove_6b50d, 5, 'beautiful')
```

```
Out[23]: ['lovely', 'gorgeous', 'wonderful', 'charming', 'beauty']
```

求类比词

除近似词以外,我们还可以使用预训练词向量求词与词之间的类比关系。例如,man(男人): woman(女人):: son(儿子): daughter(女儿)是一个类比例子:“man”之于“woman”相当于“son”之于“daughter”。求类比词问题可以定义为:对于类比关系中的四个词 $a:b::c:d$,给定前三个词 a,b 和 c ,求 d 。设词 w 的词向量为 $\text{vec}(w)$ 。而解类比词的思路是,找到和 $\text{vec}(c)+\text{vec}(b)-\text{vec}(a)$ 的结果向量最相似的词向量。

本例中,我们将从整个词典(大小40万,不含未知词符号)中搜索类比词。

```
In [24]: def get_top_k_by_analogy(token_embedding, k, word1, word2, word3):
    word_vecs = token_embedding.get_vecs_by_tokens([word1, word2, word3])
    word_diff = (word_vecs[1] - word_vecs[0] + word_vecs[2]).reshape((-1, 1))
    vocab_vecs = norm_vecs_by_row(token_embedding.idx_to_vec)
    dot_prod = nd.dot(vocab_vecs, word_diff)
    indices = nd.topk(dot_prod.reshape((len(token_embedding), )), k=k,
                      ret_typ='indices')
    indices = [int(i.asscalar()) for i in indices]
    return token_embedding.to_tokens(indices)
```

“男-女”类比:“man”之于“woman”相当于“son”之于什么?

```
In [25]: get_top_k_by_analogy(glove_6b50d, 1, 'man', 'woman', 'son')
```

```
Out[25]: ['daughter']
```

验证一下 $\text{vec}(\text{son})+\text{vec}(\text{woman})-\text{vec}(\text{man})$ 与 $\text{vec}(\text{daughter})$ 两个向量之间的余弦相似度。

```
In [26]: def cos_sim_word_analogy(token_embedding, word1, word2, word3, word4):
    words = [word1, word2, word3, word4]
    vecs = token_embedding.get_vecs_by_tokens(words)
    return cos_sim(vecs[1] - vecs[0] + vecs[2], vecs[3])

cos_sim_word_analogy(glove_6b50d, 'man', 'woman', 'son', 'daughter')
```

```
Out[26]:
[ 0.96583432]
<NDArray 1 @cpu(0)>
```

“首都-国家”类比:“beijing”(北京)之于“china”(中国)相当于“tokyo”(东京)之于什么?
答案应该是“japan”(日本)。

```
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”（坏的）之于“worst”（最坏的）相当于“big”（大的）之于什么？答案应该是“biggest”（最大的）。

```
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”（做）之于“did”（做过）相当于“go”（去）之于什么？答案应该是“went”（去过）。

```
In [29]: get_top_k_by_analogy(glove_6b50d, 1, 'do', 'did', 'go')  
Out[29]: ['went']
```

10.3.4 小结

- 我们可以应用预训练的词向量求近似词和类比词。

10.3.5 练习

- 将近似词和类比词应用中的 k 调大一些，观察结果。
- 测试一下 fastText 的中文词向量（pretrained_file_name='wiki.zh.vec'）。
- 如果在“循环神经网络的 Gluon 实现”一节中将 Embedding 实例里的参数初始化为预训练的词向量，效果如何？

10.3.6 扫码直达讨论区



10.3.7 参考文献

[1] GloVe 项目网站. <https://nlp.stanford.edu/projects/glove/>

[2] fastText 项目网站. <https://fasttext.cc/>

10.4 文本分类：情感分析

文本分类即把一段不定长的文本序列变换为类别。在文本分类问题中，情感分析是一项重要的自然语言处理任务。例如，Netflix 或者 IMDb 可以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。该模型将判断一段不定长的文本序列中包含的是正面还是负面的情绪，也即将文本序列分类为正面或负面。

10.4.1 模型设计

在这个模型中，我们将应用预训练的词向量和含多个隐藏层的双向循环神经网络。首先，文本序列的每一个词将以预训练的词向量作为词的特征向量。然后，我们使用双向循环神经网络对特征序列进一步编码得到序列信息。最后，我们将编码的序列信息通过全连接层变换为输出。在本节的实验中，我们将双向长短期记忆在最初时间步和最终时间步的隐藏状态连结，作为特征序列的编码信息传递给输出层分类。

在实验开始前，导入所需的包或模块。

```
In [1]: import sys
        sys.path.append('..')
        import collections
        import gluonbook as gb
        import mxnet as mx
        from mxnet import autograd, gluon, init, metric, nd
        from mxnet.gluon import loss as gloss, nn, rnn
        from mxnet.contrib import text
        import os
        import random
        import zipfile
```

10.4.2 读取 IMDb 数据集

我们使用 Stanford's Large Movie Review Dataset 作为情感分析的数据集 [1]。它的下载地址是 http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz。

这个数据集分为训练和测试用的两个数据集，分别有 25,000 条从 IMDb 下载的关于电影的评论。在每个数据集中，标签为“正面”(1) 和“负面”(0) 的评论数量相等。将下载好的数据解压并存放在路径“`../data/aclImdb`”。

为方便快速上手，我们提供了上述数据集的小规模采样，并存放在路径“`../data/aclImdb_tiny.zip`”。如果你将使用上述的 IMDb 完整数据集，还需要把下面 `demo` 变量改为 `False`。

In [2]: # 如果使用下载的 `IMDb` 的完整数据集，把下面改为 `False`。

```
demo = True
if demo:
    with zipfile.ZipFile('../data/aclImdb_tiny.zip', 'r') as zin:
        zin.extractall('../data/')
```

下面，读取训练和测试数据集。

In [3]: `def` `readIMDB`(`dir_url`, `seg='train'`):

```
pos_or_neg = ['pos', 'neg']
data = []
for label in pos_or_neg:
    files = os.listdir(
        '../data/' + dir_url + '/' + seg + '/' + label + '/')
    for file in files:
        with open('../data/' + dir_url + '/' + seg + '/' + label + '/' +
                  file, 'r', encoding='utf8') as rf:
            review = rf.read().replace('\n', '')
        if label == 'pos':
            data.append([review, 1])
        elif label == 'neg':
            data.append([review, 0])
return data

if demo:
    train_data = readIMDB('aclImdb_tiny/', 'train')
    test_data = readIMDB('aclImdb_tiny/', 'test')
else:
    train_data = readIMDB('aclImdb/', 'train')
    test_data = readIMDB('aclImdb/', 'test')

random.shuffle(train_data)
random.shuffle(test_data)
```

10.4.3 分词

接下来我们对每条评论做分词，从而得到分好词的评论。这里使用最简单的方法：基于空格进行分词。我们将在本节练习中探究其他的分词方法。

```
In [4]: def tokenizer(text):
    return [tok.lower() for tok in text.split(' ')]

train_tokenized = []
for review, score in train_data:
    train_tokenized.append(tokenizer(review))
test_tokenized = []
for review, score in test_data:
    test_tokenized.append(tokenizer(review))
```

10.4.4 创建词典

现在，我们可以根据分好词的训练数据集来创建词典了。这里我们设置了特殊符号“<unk>”（unknown）。它将表示一切不存在于训练数据集词典中的词。

```
In [5]: token_counter = collections.Counter()
def count_token(train_tokenized):
    for sample in train_tokenized:
        for token in sample:
            if token not in token_counter:
                token_counter[token] = 1
            else:
                token_counter[token] += 1

count_token(train_tokenized)
vocab = text.vocab.Vocabulary(token_counter, unknown_token='<unk>',
                               reserved_tokens=None)
```

10.4.5 预处理数据

下面，我们继续对数据进行预处理。每个不定长的评论将被特殊符号 PAD 补成长度为 maxlen 的序列，并用 NDArray 表示。

```
In [6]: def encode_samples(tokenized_samples, vocab):
    features = []
    for sample in tokenized_samples:
```

```

feature = []
for token in sample:
    if token in vocab.token_to_idx:
        feature.append(vocab.token_to_idx[token])
    else:
        feature.append(0)
features.append(feature)
return features

def pad_samples(features, maxlen=500, PAD=0):
    padded_features = []
    for feature in features:
        if len(feature) > maxlen:
            padded_feature = feature[:maxlen]
        else:
            padded_feature = feature
            # 添加 PAD 符号使每个序列等长 (长度为 maxlen )。
            while len(padded_feature) < maxlen:
                padded_feature.append(PAD)
        padded_features.append(padded_feature)
    return padded_features

ctx = gb.try_gpu()
train_features = encode_samples(train_tokenized, vocab)
test_features = encode_samples(test_tokenized, vocab)
train_features = nd.array(pad_samples(train_features, 500, 0), ctx=ctx)
test_features = nd.array(pad_samples(test_features, 500, 0), ctx=ctx)
train_labels = nd.array([score for _, score in train_data], ctx=ctx)
test_labels = nd.array([score for _, score in test_data], ctx=ctx)

```

10.4.6 加载预训练的词向量

这里，我们为词典 `vocab` 中的每个词加载 GloVe 词向量（每个词向量长度为 100）。稍后，我们将用这些词向量作为评论中每个词的特征向量。

```
In [7]: glove_embedding = text.embedding.create(
    'glove', pretrained_file_name='glove.6B.100d.txt', vocabulary=vocab)
```

10.4.7 定义模型

下面我们根据模型设计里的描述定义情感分类模型。其中的 `Embedding` 实例即嵌入层，`LSTM` 实例即对句子编码信息的隐藏层，`Dense` 实例即生成分类结果的输出层。

```
In [8]: class SentimentNet(nn.Block):
    def __init__(self, vocab, embed_size, num_hiddens, num_layers,
                 bidirectional, **kwargs):
        super(SentimentNet, self).__init__(**kwargs)
        with self.name_scope():
            self.embedding = nn.Embedding(len(vocab), embed_size)
            self.encoder = rnn.LSTM(num_hiddens, num_layers=num_layers,
                                   bidirectional=bidirectional,
                                   input_size=embed_size)
            self.decoder = nn.Dense(num_outputs, flatten=False)

    def forward(self, inputs):
        embeddings = self.embedding(inputs)
        states = self.encoder(embeddings)
        # 连结初始时间步和最终时间步的隐藏状态。
        encoding = nd.concat(states[0], states[-1])
        outputs = self.decoder(encoding)
        return outputs
```

由于情感分类的训练数据集并不是很大，为应对过拟合现象，我们将直接使用在更大规模语料上预训练的词向量作为每个词的特征向量。在训练中，我们不再更新这些词向量，即不再迭代模型嵌入层中的参数。

```
In [9]: num_outputs = 2
lr = 0.1
num_epochs = 1
batch_size = 10
embed_size = 100
num_hiddens = 100
num_layers = 2
bidirectional = True

net = SentimentNet(vocab, embed_size, num_hiddens, num_layers, bidirectional)
net.initialize(init.Xavier(), ctx=ctx)
# 设置 embedding 层的 weight 为预训练的词向量。
net.embedding.weight.set_data(glove_embedding.idx_to_vec.as_in_context(ctx))
# 训练中不更新词向量 (net.embedding 中的模型参数)。
net.embedding.collect_params().setattr('grad_req', 'null')
```

```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
loss = gloss.SoftmaxCrossEntropyLoss()
```

10.4.8 训练并评价模型

在实验中，我们使用准确率作为评价模型的指标。

```
In [10]: def eval_model(features, labels):
    l_sum = 0
    l_n = 0
    accuracy = metric.Accuracy()
    for i in range(features.shape[0] // batch_size):
        X = features[i*batch_size : (i+1)*batch_size].as_in_context(ctx).T
        y = labels[i*batch_size : (i+1)*batch_size].as_in_context(ctx).T
        output = net(X)
        l = loss(output, y)
        l_sum += l.sum().asscalar()
        l_n += l.size
        accuracy.update(preds=nd.argmax(output, axis=1), labels=y)
    return l_sum / l_n, accuracy.get()[1]
```

下面开始训练模型。

```
In [11]: for epoch in range(1, num_epochs + 1):
    for i in range(train_features.shape[0] // batch_size):
        X = train_features[i*batch_size : (i+1)*batch_size].as_in_context(
            ctx).T
        y = train_labels[i*batch_size : (i+1)*batch_size].as_in_context(
            ctx).T
        with autograd.record():
            l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
    train_loss, train_acc = eval_model(train_features, train_labels)
    test_loss, test_acc = eval_model(test_features, test_labels)
    print('epoch %d, train loss %.6f, acc %.2f; test loss %.6f, acc %.2f'
          % (epoch, train_loss, train_acc, test_loss, test_acc))

epoch 1, train loss 0.693691, acc 0.40; test loss 0.691604, acc 0.50
```

下面我们试着分析一个简单的句子的情感（1和0分别代表正面和负面）。为了在更复杂的句子上得到较准确的分类，我们需要使用完整数据集训练模型，并适当增大训练周期。

```
In [12]: review = ['this', 'movie', 'is', 'great']
```

```
nd.argmax(net(nd.reshape(  
    nd.array([vocab.token_to_idx[token] for token in review], ctx=ctx),  
    shape=(-1, 1))), axis=1).asscalar()  
  
Out[12]: 1.0
```

10.4.9 小结

- 我们可以应用预训练的词向量和循环神经网络对文本进行情感分析。

10.4.10 练习

- 使用 IMDb 完整数据集，并把迭代周期改为 3。你的模型能在训练和测试数据集上得到怎样的准确率？通过调节超参数，你能进一步提升分类准确率吗？
- 使用更大的预训练词向量，例如 300 维的 GloVe 词向量，能否提升分类准确率？
- 使用 spacy 分词工具，能否提升分类准确率？你需要安装 spacy: pip install spacy，并且安装英文包: python -m spacy download en。在代码中，先导入 spacy: import spacy。然后加载 spacy 英文包: spacy_en = spacy.load('en')。最后定义函数: def tokenizer(text): return [tok.text for tok in spacy_en.tokenizer(text)] 替换原来的基于空格分词的 tokenizer 函数。需要注意的是，GloVe 的词向量对于名词词组的存储方式是用“-”连接各个单词，例如词组“new york”在 GloVe 中的表示为“new-york”。而使用 spacy 分词之后“new york”的存储可能是“new york”。
- 通过上面三种方法，你能使模型在测试集上的准确率提高到 0.85 以上吗？

10.4.11 扫码直达讨论区



10.4.12 参考文献

[1] Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011, June). Learning word vectors for sentiment analysis. In Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1 (pp. 142-150). Association for Computational Linguistics.

10.5 编码器—解码器 (seq2seq)

在很多应用中，输入和输出都可以是不定长序列。以机器翻译为例，输入可以是一段不定长的英语文本序列，输出可以是一段不定长的法语文本序列，例如

英语输入：“They”、“are”、“watching”、“.”

法语输出：“Ils”、“regardent”、“.”

当输入输出都是不定长序列时，我们可以使用编码器—解码器(encoder-decoder)[1]或者 seq2seq 模型[2]。这两个模型本质上都用到了两个循环神经网络，分别叫做编码器和解码器。编码器对应输入序列，解码器对应输出序列。下面我们来介绍编码器—解码器的设计。

10.5.1 编码器

编码器的把一个不定长的输入序列转换成一个定长的背景变量 c ，并在该背景变量中编码输入序列信息。常用的编码器是循环神经网络。

让我们考虑批量大小为 1 的时序数据样本。在时间步 t ，循环神经网络将输入 x_t 的特征向量 \mathbf{x}_t 和上个时间步的隐藏状态 \mathbf{h}_{t-1} 变换为当前时间步的隐藏状态 \mathbf{h}_t 。其中，每个输入的特征向量可能是模型参数，例如“[循环神经网络的 Gluon 实现](#)”一节中需要学习的每个词向量。我们可以用函数 f 表达循环神经网络隐藏层的变换：

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}).$$

假设输入序列的总时间步数为 T 。编码器通过自定义函数 q 将各个时间步的隐藏状态变换为背景变量

$$\mathbf{c} = q(\mathbf{h}_1, \dots, \mathbf{h}_T).$$

例如，当选择 $q(\mathbf{h}_1, \dots, \mathbf{h}_T) = \mathbf{h}_T$ 时，背景变量是输入序列最终时间步的隐藏状态 \mathbf{h}_T 。我们将这里的循环神经网络叫做编码器。

以上描述的编码器是一个单向的循环神经网络，每个时间步的隐藏状态只取决于该时间步及之前的输入子序列。我们也可以使用双向循环神经网络构造编码器。这种情况下，编码器每个时间步的隐藏状态同时取决于该时间步之前和之后的子序列（包括当前时间步的输入），并编码了整个序列的信息。

10.5.2 解码器

刚刚已经介绍，假设输入序列的总时间步数为 T ，编码器输出的背景变量 \mathbf{c} 编码了整个输入序列 x_1, \dots, x_T 的信息。给定训练样本中的输出序列 $y_1, y_2, \dots, y_{T'}$ 。假设其中每个时间步 t' 的输出同时取决于该时间步之前的输出序列和背景变量。那么，根据最大似然估计，我们可以最大化输出序列基于输入序列的条件概率

$$\begin{aligned}\mathbb{P}(y_1, \dots, y_{T'} \mid x_1, \dots, x_T) &= \prod_{t'=1}^{T'} \mathbb{P}(y_{t'} \mid y_1, \dots, y_{t'-1}, x_1, \dots, x_T) \\ &= \prod_{t'=1}^{T'} \mathbb{P}(y_{t'} \mid y_1, \dots, y_{t'-1}, \mathbf{c}),\end{aligned}$$

并得到该输出序列的损失

$$-\log \mathbb{P}(y_1, \dots, y_{T'} \mid x_1, \dots, x_T) = -\sum_{t'=1}^{T'} \log \mathbb{P}(y_{t'} \mid y_1, \dots, y_{t'-1}, \mathbf{c}).$$

为此，我们可以使用另一个循环神经网络作为解码器。在输出序列的时间步 t' ，解码器将上一时间步的输出 $y_{t'-1}$ 以及背景变量 \mathbf{c} 作为输入，并将它们与上一时间步的隐藏状态 $s_{t'-1}$ 变换为当前时间步的隐藏状态 $s_{t'}$ 。因此，我们可以用函数 g 表达解码器隐藏层的变换：

$$s_{t'} = g(y_{t'-1}, \mathbf{c}, s_{t'-1}).$$

有了解码器的隐藏状态后，我们可以自定义输出层来计算损失中的 $\mathbb{P}(y_{t'} \mid y_1, \dots, y_{t'-1}, \mathbf{c})$ ，例如基于当前时间步的解码器隐藏状态 $s_{t'}$ 、上一时间步的输出 $y_{t'-1}$ 以及背景变量 \mathbf{c} 来计算当前时间步输出 $y_{t'}$ 的概率分布。

在实际中，我们常使用深度循环神经网络作为编码器和解码器。我们将在本章稍后的“机器翻译”一节中实现含深度循环神经网络的编码器和解码器。

10.5.3 小结

- 编码器-解码器（seq2seq）可以输入并输出不定长的序列。
- 编码器—解码器使用了两个循环神经网络。
- 预测不定长序列的方法包括穷举搜索、贪婪搜索和束搜索。

10.5.4 练习

- 除了机器翻译，你还能想到 seq2seq 的哪些应用？
- 有哪些方法可以设计解码器的输出层？

10.5.5 扫码直达讨论区



10.5.6 参考文献

- [1] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.
- [2] Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Advances in neural information processing systems (pp. 3104-3112).

10.6 束搜索

上一节介绍了如何训练输入输出均为不定长序列的编码器—解码器。在准备训练数据集时，我们通常会在样本的输入序列和输出序列后面分别附上一个特殊符号“<eos>”（end of sequence）表

示序列的终止。在预测中，模型该如何输出不定长序列呢？

我们在接下来的讨论中将沿用上一节的数学符号。为了便于讨论，假设解码器的输出是一段文本序列。设输出文本词典 \mathcal{Y} （包含特殊符号“`<eos>`”）的大小为 $|\mathcal{Y}|$ ，输出序列的最大长度为 T' 。所有可能的输出序列一共有 $\mathcal{O}(|\mathcal{Y}|^{T'})$ 种。这些输出序列中所有特殊符号“`<eos>`”及其后面的子序列将被舍弃。

10.6.1 穷举搜索

我们在描述解码器时提到，输出序列基于输入序列的条件概率是 $\prod_{t'=1}^{T'} \mathbb{P}(y_{t'} \mid y_1, \dots, y_{t'-1}, c)$ 。为了搜索该概率最大的输出序列，一种方法是穷举所有可能序列的概率，并输出概率最大的序列。我们将该序列称为最优序列，并将这种搜索方法称为穷举搜索（exhaustive search）。

很明显，穷举搜索很容易因为计算开销 $\mathcal{O}(|\mathcal{Y}|^{T'})$ 太大而无法使用。例如，当 $|\mathcal{Y}| = 10000$ 且 $\{T^{\text{:raw-latex:prime}}=10\}$ 时， $10000^{10} = 1 \times 10^{40}$ 。

10.6.2 贪婪搜索

我们还可以使用贪婪搜索（greedy search）。也就是说，对于输出序列任一时间步 t' ，从 $|\mathcal{Y}|$ 个词中搜索出输出词

$$y_{t'} = \operatorname{argmax}_{y_{t'} \in \mathcal{Y}} \mathbb{P}(y_{t'} \mid y_1, \dots, y_{t'-1}, c),$$

且一旦搜索出“`<eos>`”符号即完成输出。

贪婪搜索的计算开销是 $\mathcal{O}(|\mathcal{Y}| \times T')$ 。它比起穷举搜索的计算开销显著下降。例如，当 $|\mathcal{Y}| = 10000$ 且 $\{T^{\text{:raw-latex:prime}}=10\}$ 时， $10000 \times 10 = 1 \times 10^5$ 。然而，贪婪搜索并不能保证输出是最优序列。

10.6.3 束搜索

束搜索（beam search）介于上面二者之间。我们通过一个具体例子描述它。

假设输出序列的词典中只包含五个元素： $\mathcal{Y} = \{A, B, C, D, E\}$ ，且其中一个为特殊符号“`<eos>`”。设束搜索的超参数束宽（beam size）等于 2，输出序列最大长度为 3。

在输出序列的时间步 1 时，假设条件概率 $\mathbb{P}(y_{t'} \mid c)$ 最大的两个词为 A 和 C 。我们在时间步 2 时将对所有的 $y_2 \in \mathcal{Y}$ 都分别计算 $\mathbb{P}(y_2 \mid A, c)$ 和 $\mathbb{P}(y_2 \mid C, c)$ ，并从计算出的 10 个概率中取最大的

两个：假设为 $\mathbb{P}(B \mid A, \mathbf{c})$ 和 $\mathbb{P}(E \mid C, \mathbf{c})$ 。那么，我们在时间步 3 时将对所有的 $y_3 \in \mathcal{Y}$ 都分别计算 $\mathbb{P}(y_3 \mid A, B, \mathbf{c})$ 和 $\mathbb{P}(y_3 \mid C, E, \mathbf{c})$ ，并从计算出的 10 个概率中取最大的两个：假设为 $\mathbb{P}(D \mid A, B, \mathbf{c})$ 和 $\mathbb{P}(D \mid C, E, \mathbf{c})$ 。

接下来，我们可以在 6 个输出序列： A 、 C 、 AB 、 CE 、 ABD 、 CED 中筛选出包含特殊符号“`<eos>`”的序列，并将它们中所有特殊符号“`<eos>`”及其后面的子序列舍弃，得到候选序列。在这些候选序列中，取以下分数最高的序列作为输出序列：

$$\frac{1}{L^\alpha} \log \mathbb{P}(y_1, \dots, y_L) = \frac{1}{L^\alpha} \sum_{t'=1}^L \log \mathbb{P}(y_{t'} \mid y_1, \dots, y_{t'-1}, \mathbf{c}),$$

其中 L 为候选序列长度， α 一般可选为 0.75。分母上的 L^α 是为了惩罚较长序列在以上分数中较多的对数相加项。

穷举搜索和贪婪搜索也可看作是两种特殊束宽的束搜索。束搜索通过更灵活的束宽来权衡计算开销和搜索质量。通常束宽取 1 也可以在机器翻译中也可以取得不错的结果 [1]。

10.6.4 小结

- 预测不定长序列的方法包括穷举搜索、贪婪搜索和束搜索。
- 束搜索通过更灵活的束宽来权衡计算开销和搜索质量。

10.6.5 练习

- 在“循环神经网络”一节中，我们使用语言模型创作歌词。它的输出属于哪种搜索？你能改进它吗？

10.6.6 扫码直达讨论区



10.6.7 参考文献

- [1] Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Advances in neural information processing systems (pp. 3104-3112).

10.7 注意力机制

在“编码器—解码器（seq2seq）”一节里的解码器设计中，输出序列的各个时间步使用了相同的背景变量。如果解码器的不同时间步可以使用不同的背景变量呢？这样做有什么好处？

10.7.1 动机

以英语-法语翻译为例，给定一对英语输入序列“*They*”、“*are*”、“*watching*”、“.”和法语输出序列“*Ils*”、“*regardent*”、“.”。解码器可以在输出序列的时间步1使用更集中编码了“*They*”、“*are*”信息的背景变量来生成“*Ils*”，在时间步2使用更集中编码了“*watching*”信息的背景变量来生成“*regardent*”，在时间步3使用更集中编码了“.”信息的背景变量来生成“.”。这看上去就像是在解码器的每一时间步对输入序列中不同时间步编码的信息分配不同的注意力。这也是注意力机制的由来。它最早由 Bahdanau 等提出 [1]。

10.7.2 设计

本节沿用“编码器—解码器（seq2seq）”一节里的数学符号。

我们对“编码器—解码器（seq2seq）”一节里的解码器稍作修改。在时间步 t' ，设解码器的背景变量为 $\mathbf{c}_{t'}$ ，输出 $y_{t'}$ 的特征向量为 $\mathbf{y}_{t'}$ 。和输入的特征向量一样，这里每个输出的特征向量也是模型参数。解码器在时间步 t' 的隐藏状态

$$\mathbf{s}_{t'} = g(\mathbf{y}_{t'-1}, \mathbf{c}_{t'}, \mathbf{s}_{t'-1}).$$

令编码器在时间步 t 的隐藏状态为 \mathbf{h}_t ，且总时间步数为 T 。解码器在时间步 t' 的背景变量为

$$\mathbf{c}_{t'} = \sum_{t=1}^T \alpha_{t't} \mathbf{h}_t,$$

其中 $\alpha_{t't}$ 是权值。也就是说，给定解码器的当前时间步 t' ，我们需要对编码器中不同时间步 t 的隐藏状态求加权平均。这里的权值也称注意力权重。它的计算公式是

$$\alpha_{t't} = \frac{\exp(e_{t't})}{\sum_{k=1}^T \exp(e_{t'k})},$$

其中 $e_{t't} \in \mathbb{R}$ 的计算为

$$e_{t't} = a(\mathbf{s}_{t'-1}, \mathbf{h}_t).$$

上式中的函数 a 有多种设计方法。Bahdanau 等使用了多层感知机：

$$e_{t't} = \mathbf{v}^\top \tanh(\mathbf{W}_s \mathbf{s}_{t'-1} + \mathbf{W}_h \mathbf{h}_t),$$

其中 \mathbf{v} 、 \mathbf{W}_s 、 \mathbf{W}_h 以及编码器与解码器中的各个权重和偏差都是模型参数 [1]。

Bahdanau 等在编码器和解码器中分别使用了门控循环单元 [1]。在解码器中，我们需要对门控循环单元的设计稍作修改。解码器在 \$t^{:raw-latex:}`prime`\$ 时间步的隐藏状态为

$$\mathbf{s}_{t'} = \mathbf{z}_{t'} \odot \mathbf{s}_{t'-1} + (1 - \mathbf{z}_{t'}) \odot \tilde{\mathbf{s}}_{t'},$$

其中的重置门、更新门和候选隐含状态分别为

$$\begin{aligned}\mathbf{r}_{t'} &= \sigma(\mathbf{W}_{yr} \mathbf{y}_{t'-1} + \mathbf{W}_{sr} \mathbf{s}_{t'-1} + \mathbf{W}_{cr} \mathbf{c}_{t'} + \mathbf{b}_r), \\ \mathbf{z}_{t'} &= \sigma(\mathbf{W}_{yz} \mathbf{y}_{t'-1} + \mathbf{W}_{sz} \mathbf{s}_{t'-1} + \mathbf{W}_{cz} \mathbf{c}_{t'} + \mathbf{b}_z), \\ \tilde{\mathbf{s}}_{t'} &= \tanh(\mathbf{W}_{ys} \mathbf{y}_{t'-1} + \mathbf{W}_{ss} (\mathbf{s}_{t'-1} \odot \mathbf{r}_{t'}) + \mathbf{W}_{cs} \mathbf{c}_{t'} + \mathbf{b}_s).\end{aligned}$$

我们将在下一节中实现含注意力机制的编码器和解码器。

10.7.3 小结

- 我们可以在解码器的每个时间步使用不同的背景变量，并对输入序列中不同时间步编码的信息分配不同的注意力。

10.7.4 练习

- 不修改“门控循环单元（GRU）”一节中的 `gru_rnn` 函数，应如何用它实现本节介绍的解码器？
- 除了自然语言处理，注意力机制还可以应用在哪些地方？

10.7.5 扫码直达讨论区



10.7.6 参考文献

[1] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.

10.8 机器翻译

本节介绍编码器—解码器和注意力机制的应用。我们以机器翻译为例，使用 Gluon 实现一个含注意力机制的编码器—解码器。机器翻译的输入与输出都是不定长的文本序列。

10.8.1 含注意力机制的编码器—解码器

首先，导入实现所需的包或模块。

```
In [1]: import collections
        import io
        import mxnet as mx
        from mxnet import autograd, gluon, init, nd
        from mxnet.contrib import text
        from mxnet.gluon import data as gdata, loss as gloss, nn, rnn
```

下面定义一些特殊符号。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>'
```

以下设置了模型超参数。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。实验中，我们选取长度不超过 5 的输入和输出序列，并将预测时输出序列的最大长度设为 20。这些序列长度考虑了句末添加的“<eos>”符号。

```
In [3]: num_epochs = 40
eval_interval = 10
lr = 0.005
batch_size = 2
max_seq_len = 5
max_test_output_len = 20
encoder_num_layers = 1
decoder_num_layers = 2
encoder_drop_prob = 0.1
decoder_drop_prob = 0.1
encoder_embed_size = 256
encoder_num_hiddens = 256
decoder_num_hiddens = 256
alignment_size = 25
ctx = mx.cpu(0)
```

读取数据

我们定义函数读取训练数据集。为了演示方便，这里使用了一个很小的法语—英语数据集。在读取数据时，我们在句末附上“<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 io.open('../data/fr-en-small.txt') as f:
        lines = f.readlines()
        for line in lines:
            input_seq, output_seq = line.rstrip().split('\t')
            cur_input_tokens = input_seq.split(' ')
            cur_output_tokens = output_seq.split(' ')
            if len(cur_input_tokens) < max_seq_len and \
                len(cur_output_tokens) < max_seq_len:
                input_tokens.extend(cur_input_tokens)
                # 句末附上 EOS 符号。
                cur_input_tokens.append(EOS)
                # 添加 PAD 符号使每个序列等长（长度为 max_seq_len）。
```

```

        while len(cur_input_tokens) < max_seq_len:
            cur_input_tokens.append(PAD)
        input_seqs.append(cur_input_tokens)
        output_tokens.extend(cur_output_tokens)
        cur_output_tokens.append(EOS)
        while len(cur_output_tokens) < max_seq_len:
            cur_output_tokens.append(PAD)
        output_seqs.append(cur_output_tokens)
fr_vocab = text.vocab.Vocabulary(collections.Counter(input_tokens),
                                    reserved_tokens=[PAD, BOS, EOS])
en_vocab = text.vocab.Vocabulary(collections.Counter(output_tokens),
                                    reserved_tokens=[PAD, BOS, EOS])
return fr_vocab, en_vocab, input_seqs, output_seqs

```

以下创建训练数据集。每一个样本包含法语的输入序列和英语的输出序列。

```

In [5]: input_vocab, output_vocab, input_seqs, output_seqs = read_data(max_seq_len)
fr = nd.zeros((len(input_seqs), max_seq_len), ctx=ctx)
en = nd.zeros((len(output_seqs), max_seq_len), ctx=ctx)
for i in range(len(input_seqs)):
    fr[i] = nd.array(input_vocab.to_indices(input_seqs[i]), ctx=ctx)
    en[i] = nd.array(output_vocab.to_indices(output_seqs[i]), ctx=ctx)
dataset = gdata.ArrayDataset(fr, en)

```

定义编码器

以下定义了基于门控循环单元的编码器。

```

In [6]: class Encoder(nn.Block):
    def __init__(self, num_inputs, embed_size, num_hiddens, num_layers,
                 drop_prob, **kwargs):
        super(Encoder, self).__init__(**kwargs)
        with self.name_scope():
            self.embedding = nn.Embedding(num_inputs, embed_size)
            self.dropout = nn.Dropout(drop_prob)
            self.rnn = rnn.GRU(num_hiddens, num_layers, dropout=drop_prob,
                               input_size=embed_size)

    def forward(self, inputs, state):
        embedding = self.embedding(inputs).swapaxes(0, 1)
        embedding = self.dropout(embedding)
        output, state = self.rnn(embedding, state)
        return output, state

```

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

定义含注意力机制的解码器

以下定义了基于门控循环单元的解码器。解码器中注意力机制的实现参考了“注意力机制”一节中的描述。

```
In [7]: class Decoder(nn.Block):
    def __init__(self, num_hiddens, num_outputs, num_layers, max_seq_len,
                 drop_prob, alignment_size, encoder_num_hiddens, **kwargs):
        super(Decoder, self).__init__(**kwargs)
        self.max_seq_len = max_seq_len
        self.encoder_num_hiddens = encoder_num_hiddens
        self.hidden_size = num_hiddens
        self.num_layers = num_layers
        with self.name_scope():
            self.embedding = nn.Embedding(num_outputs, num_hiddens)
            self.dropout = nn.Dropout(drop_prob)
            # 注意力机制。
            self.attention = nn.Sequential()
            with self.attention.name_scope():
                self.attention.add(
                    nn.Dense(alignment_size,
                            in_units=num_hiddens + encoder_num_hiddens,
                            activation="tanh", flatten=False))
                self.attention.add(nn.Dense(1, in_units=alignment_size,
                                           flatten=False))

            self.rnn = rnn.GRU(num_hiddens, num_layers, dropout=drop_prob,
                               input_size=num_hiddens)
            self.out = nn.Dense(num_outputs, in_units=num_hiddens,
                               flatten=False)
            self.rnn_concat_input = nn.Dense(
                num_hiddens, in_units=num_hiddens + encoder_num_hiddens,
                flatten=False)

    def forward(self, cur_input, state, encoder_outputs):
        # 当循环神经网络有多个隐藏层时，取最靠近输出层的单层隐藏状态。
        single_layer_state = [state[0][-1].expand_dims(0)]
        encoder_outputs = encoder_outputs.reshape((self.max_seq_len, -1,
                                                   self.encoder_num_hiddens))
        hidden_broadcast = nd.broadcast_axis(single_layer_state[0], axis=0,
```

```

size=self.max_seq_len)
encoder_outputs_and_hiddens = nd.concat(encoder_outputs,
                                         hidden_broadcast, dim=2)
energy = self.attention(encoder_outputs_and_hiddens)
batch_attention = nd.softmax(energy, axis=0).transpose((1, 2, 0))
batch_encoder_outputs = encoder_outputs.swapaxes(0, 1)
decoder_context = nd.batch_dot(batch_attention, batch_encoder_outputs)
input_and_context = nd.concat(
    nd.expand_dims(self.embedding(cur_input), axis=1),
    decoder_context, dim=2)
concat_input = self.rnn_concat_input(input_and_context).reshape(
    (1, -1, 0))
concat_input = self.dropout(concat_input)
state = [nd.broadcast_axis(single_layer_state[0], axis=0,
                           size=self.num_layers)]
output, state = self.rnn(concat_input, state)
output = self.dropout(output)
output = self.out(output).reshape((-3, -1))
return output, state

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

```

定义解码器初始状态

为了初始化解码器的隐藏状态，我们通过一层全连接网络来变换编码器的输出隐藏状态。

```

In [8]: class DecoderInitState(nn.Block):
    def __init__(self, encoder_num_hiddens, decoder_num_hiddens, **kwargs):
        super(DecoderInitState, self).__init__(**kwargs)
        with self.name_scope():
            self.dense = nn.Dense(decoder_num_hiddens,
                                 in_units=encoder_num_hiddens,
                                 activation="tanh", flatten=False)

    def forward(self, encoder_state):
        return [self.dense(encoder_state)]

```

训练模型并输出不定长序列

我们定义 `translate` 函数应用训练好的模型，并通过贪婪搜索输出不定长的翻译文本序列。解码器的最初时间步输入来自“`<bos>`”符号。对于一个输出中的序列，当解码器在某一时间步搜索出“`<eos>`”符号时，即完成该输出序列。

```
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=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 符号。
        decoder_input = nd.array([output_vocab.token_to_idx[BOS]], ctx=ctx)
        decoder_state = decoder_init_state(encoder_state[0])
        output_tokens = []

        for _ in range(max_test_output_len):
            decoder_output, decoder_state = decoder(
                decoder_input, decoder_state, encoder_outputs)
            pred_i = int(decoder_output.argmax(axis=1).asnumpy()[0])
            # 当任一时间步搜索出 EOS 符号时，输出序列即完成。
            if pred_i == output_vocab.token_to_idx[EOS]:
                break
            else:
                output_tokens.append(output_vocab.idx_to_token[pred_i])
            decoder_input = nd.array([pred_i], ctx=ctx)
        print('[output]', ' '.join(output_tokens))
        print('[expect]', fr_en[1], '\n')
```

下面定义模型训练函数。为了初始化解码器的隐藏状态，我们通过一层全连接网络来变换编码器最早时间步的输出隐藏状态。解码器中，当前时间步的预测词将作为下一时间步的输入。其实，我们也可以使用样本输出序列在当前时间步的词作为下一时间步的输入。这叫作强制教学 (teacher forcing)。

```
In [10]: loss = gloss.SoftmaxCrossEntropyLoss()
```

```

eos_id = output_vocab.token_to_idx[EOS]

def train(encoder, decoder, decoder_init_state, max_seq_len, ctx,
          eval_fr_ens):
    encoder.initialize(init.Xavier(), ctx=ctx)
    decoder.initialize(init.Xavier(), ctx=ctx)
    decoder_init_state.initialize(init.Xavier(), ctx=ctx)
    encoder_optimizer = gluon.Trainer(encoder.collect_params(), 'adam',
                                       {'learning_rate': lr})
    decoder_optimizer = gluon.Trainer(decoder.collect_params(), 'adam',
                                       {'learning_rate': lr})
    decoder_init_state_optimizer = gluon.Trainer(
        decoder_init_state.collect_params(), 'adam', {'learning_rate': lr})

    data_iter = gdata.DataLoader(dataset, batch_size, shuffle=True)
    l_sum = 0
    for epoch in range(1, num_epochs + 1):
        for x, y in data_iter:
            cur_batch_size = x.shape[0]
            with autograd.record():
                l = nd.array([0], ctx=ctx)
                valid_length = nd.array([0], ctx=ctx)
                encoder_state = encoder.begin_state(
                    func=nd.zeros, batch_size=cur_batch_size, ctx=ctx)
                encoder_outputs, encoder_state = encoder(x, encoder_state)
                encoder_outputs = encoder_outputs.flatten()
                # 解码器的第一个输入为 BOS 符号。
                decoder_input = nd.array(
                    [output_vocab.token_to_idx[BOS]] * cur_batch_size,
                    ctx=ctx)
                mask = nd.ones(shape=(cur_batch_size,), ctx=ctx)
                decoder_state = decoder_init_state(encoder_state[0])
                for i in range(max_seq_len):
                    decoder_output, decoder_state = decoder(
                        decoder_input, decoder_state, encoder_outputs)
                    # 解码器使用当前时间步的预测词作为下一时间步的输入。
                    decoder_input = decoder_output.argmax(axis=1)
                    valid_length = valid_length + mask.sum()
                    l = l + (mask * loss(decoder_output, y[:, i])).sum()
                    mask = mask * (y[:, i] != eos_id)
                l = l / valid_length
            l.backward()
            encoder_optimizer.step(1)

```

```

        decoder_optimizer.step(1)
        decoder_init_state_optimizer.step(1)
        l_sum += l.asscalar() / max_seq_len

    if epoch % eval_interval == 0 or epoch == 1:
        if epoch == 1:
            print('epoch %d, loss %f, ' % (epoch, l_sum / len(data_iter)))
        else:
            print('epoch %d, loss %f, '
                  % (epoch, l_sum / eval_interval / len(data_iter)))
        if epoch != 1:
            l_sum = 0
        translate(encoder, decoder, decoder_init_state, eval_fr_ens, ctx,
                  max_seq_len)

```

以下分别实例化编码器、解码器和解码器初始隐藏状态网络。

```
In [11]: encoder = Encoder(len(input_vocab), encoder_embed_size, encoder_num_hiddens,
                           encoder_num_layers, encoder_drop_prob)
decoder = Decoder(decoder_num_hiddens, len(output_vocab),
                  decoder_num_layers, max_seq_len, decoder_drop_prob,
                  alignment_size, encoder_num_hiddens)
decoder_init_state = DecoderInitState(encoder_num_hiddens,
                                      decoder_num_hiddens)
```

给定简单的法语和英语序列，我们可以观察模型的训练结果。打印的结果中，`input`、`output` 和 `expect` 分别代表输入序列、输出序列和正确序列。我们可以比较 `output` 和 `expect`，观察输出序列是否符合预期。

```
In [12]: eval_fr_ens =[['elle est japonaise .', 'she is japanese .'],
                     ['ils regardent .', 'they are watching .']]
train(encoder, decoder, decoder_init_state, max_seq_len, ctx, eval_fr_ens)

epoch 1, loss 0.518793,
[input] elle est japonaise .
[output] she . . . . . . . . . . . . .
[expect] she is japanese .

[input] ils regardent .
[output] they are . . . . . . . . . . .
[expect] they are watching .

epoch 10, loss 0.192958,
[input] elle est japonaise .
[output] she is japanese .
```

```
[expect] she is japanese .  
  
[input] ils regardent .  
[output] they are watching .  
[expect] they are watching .  
  
epoch 20, loss 0.038548,  
[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 0.001125,  
[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 0.000293,  
[input] elle est japonaise .  
[output] she is japanese .  
[expect] she is japanese .  
  
[input] ils regardent .  
[output] they are watching .  
[expect] they are watching .
```

为了使模型能够翻译更复杂的句子，我们需要使用更大的训练数据集、调节超参数并增加训练时间。当然，我们还需要有验证数据集，并依据模型在它上面的表现调参。那么，该如何在验证数据集上评价模型的表现呢？这就需要评价翻译结果的指标。

10.8.2 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BLEU（Bilingual Evaluation Understudy）[1]。

设 k 为我们希望评价的 n 个连续词的最大长度，例如 $k = 4$ 。设 n 个连续词的精度为 p_n 。它是模型预测序列与样本标签序列匹配 n 个连续词的数量与模型预测序列中 n 个连续词数量之比。举个例子，假设标签序列为 $ABCDEF$ ，预测序列为 $ABBCD$ 。那么 $p_1 = 4/5, p_2 = 3/4, p_3 = 1/3, p_4 = 0$ 。设 len_{label} 和 len_{pred} 分别为标签序列和模型预测序列的词数。那么，BLEU 的定义为

$$\exp(\min(0, 1 - \frac{len_{\text{label}}}{len_{\text{pred}}})) \prod_{i=1}^k p_n^{1/2^n}.$$

需要注意的是，匹配较长连续词比匹配较短连续词更难。因此，一方面，匹配较长连续词应被赋予更大权重。而上式中 $p_n^{1/2^n}$ 的指数相当于权重。随着 n 的提高， n 个连续词的精度的权重随着 $1/2^n$ 的减小而增大。例如 $0.5^{1/2} \approx 0.7, 0.5^{1/4} \approx 0.84, 0.5^{1/8} \approx 0.92, 0.5^{1/16} \approx 0.96$ 。另一方面，模型预测较短序列往往得到较高的 n 个连续词的精度。因此，上式中连乘项前面的系数是为了惩罚较短的输出。举个例子，当 $k = 2$ 时，假设标签序列为 $ABCDEF$ ，而预测序列为 AB 。虽然 $p_1 = p_2 = 1$ ，但惩罚系数 $\exp(1 - 6/2) \approx 0.14$ ，因此 BLEU 也接近 0.14。当预测序列和标签序列完全一致时，BLEU 为 1。

10.8.3 小结

- 我们可以将编码器—解码器和注意力机制应用于机器翻译中。
- BLEU 可以用来评价翻译结果。

10.8.4 练习

- 试着使用更大的翻译数据集来训练模型，例如 WMT [2] 和 Tatoeba Project [3]。
- 在解码器中使用强制教学，观察实现现象。

10.8.5 扫码直达讨论区



10.8.6 参考文献

- [1] Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002, July). BLEU: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting on association for computational linguistics (pp. 311-318). Association for Computational Linguistics.
- [2] WMT. <http://www.statmt.org/wmt14/translation-task.html>
- [3] Tatoeba Project. <http://www.manythings.org/anki/>

11

附录

11.1 数学基础

本节总结了本书中涉及到的有关线性代数、微分和概率的基础知识。为避免赘述本书未涉及的数学背景知识，本节中的少数定义稍有简化。

11.1.1 线性代数

以下分别概括了向量、矩阵、运算、范数、特征向量和特征值的概念。

向量

本书中的向量指的是列向量。一个 n 维向量 \mathbf{x} 的表达式可写成

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

其中 x_1, \dots, x_n 是向量的元素。我们将各元素均为实数的 n 维向量 \mathbf{x} 记作 $\mathbf{x} \in \mathbb{R}^n$ 或 $\mathbf{x} \in \mathbb{R}^{n \times 1}$ 。

矩阵

一个 m 行 n 列矩阵的表达式可写成

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix},$$

其中 x_{ij} 是矩阵 \mathbf{X} 中第 i 行第 j 列的元素 ($1 \leq i \leq m, 1 \leq j \leq n$)。我们将各元素均为实数的 m 行 n 列矩阵 \mathbf{X} 记作 $\mathbf{X} \in \mathbb{R}^{m \times n}$ 。不难发现，向量是特殊的矩阵。

运算

设 n 维向量 \mathbf{a} 中的元素为 a_1, \dots, a_n , n 维向量 \mathbf{b} 中的元素为 b_1, \dots, b_n 。向量 \mathbf{a} 与 \mathbf{b} 的点乘 (内积) 是一个标量:

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + \dots + a_n b_n.$$

设两个 m 行 n 列矩阵

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

矩阵 A 的转置是一个 n 行 m 列矩阵，它的每一行其实是原矩阵的每一列：

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

两个相同形状的矩阵的加法实际上是按元素做加法：

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

我们使用符号 \odot 表示两个矩阵按元素做乘法的运算：

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

定义一个标量 k 。标量与矩阵的乘法也是按元素做乘法的运算：

$$kA = \begin{bmatrix} ka_{11} & ka_{21} & \dots & ka_{m1} \\ ka_{12} & ka_{22} & \dots & ka_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ ka_{1n} & ka_{2n} & \dots & ka_{mn} \end{bmatrix}.$$

其它例如标量与矩阵按元素相加、相除等运算与上式中的相乘运算类似。矩阵按元素开根号、取对数等运算也即对矩阵每个元素开根号、取对数等，并得到和原矩阵形状相同的矩阵。

矩阵乘法和按元素的乘法不同。设 A 为 m 行 p 列的矩阵， B 为 p 行 n 列的矩阵。两个矩阵相乘的结果

$$AB = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1p} \\ a_{21} & a_{22} & \dots & a_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \dots & a_{ip} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mp} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1j} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2j} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{p1} & b_{p2} & \dots & b_{pj} & \dots & b_{pn} \end{bmatrix}$$

是一个 m 行 n 列的矩阵，其中第 i 行第 j 列 ($1 \leq i \leq m, 1 \leq j \leq n$) 的元素为

$$a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{ip}b_{pj} = \sum_{k=1}^p a_{ik}b_{kj}.$$

范数

设 n 维向量 \mathbf{x} 中的元素为 x_1, \dots, x_n 。向量 \mathbf{x} 的 L_p 范数为

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

例如， \mathbf{x} 的 L_1 范数是该向量元素绝对值的和：

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

而 \mathbf{x} 的 L_2 范数是该向量元素平方和的平方根：

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

我们通常用 $\|\mathbf{x}\|$ 指代 $\|\mathbf{x}\|_2$ 。

设 \mathbf{X} 是一个 m 行 n 列矩阵。矩阵 \mathbf{X} 的 Frobenius 范数为该矩阵元素平方和的平方根：

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

其中 x_{ij} 为矩阵 \mathbf{X} 在第 i 行第 j 列的元素。

特征向量和特征值

对于一个 n 行 n 列的矩阵 \mathbf{A} ，假设有标量 λ 和非零的 n 维向量 \mathbf{v} 使

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v},$$

那么 \mathbf{v} 是矩阵 \mathbf{A} 的一个特征向量，标量 λ 是 \mathbf{v} 对应的特征值。

11.1.2 微分

我们在这里简要介绍微分的一些基本概念和演算。

导数和微分

假设函数 $f : \mathbb{R} \rightarrow \mathbb{R}$ 的输入和输出都是标量。函数 f 的导数

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

且假定该极限存在。给定 $y = f(x)$, 其中 x 和 y 分别是函数 f 的自变量和因变量。以下有关导数和微分的表达式等价：

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

其中符号 D 和 d/dx 也叫微分运算符。常见的微分演算有 $DC = 0$ (C 为常数)、 $Dx^n = nx^{n-1}$ (n 为常数)、 $De^x = e^x$ 、 $D\ln(x) = 1/x$ 等。

如果函数 f 和 g 都可导, 设 C 为常数, 那么

$$\begin{aligned}\frac{d}{dx}[Cf(x)] &= C \frac{d}{dx}f(x), \\ \frac{d}{dx}[f(x) + g(x)] &= \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \\ \frac{d}{dx}[f(x)g(x)] &= f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \\ \frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] &= \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}.\end{aligned}$$

如果 $y = f(u)$ 和 $u = g(x)$ 都是可导函数, 依据链式法则,

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

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（求 n 次导数）， $n!$ 为 n 的阶乘。假设 ϵ 是一个足够小的数，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

$$f(x + \epsilon) \approx f(x) + f'(x)\epsilon.$$

偏导数

设 u 为一个有 n 个自变量的函数， $u = f(x_1, x_2, \dots, x_n)$ ，它有关第 i 个变量 x_i 的偏导数为

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

以下有关偏导数的表达式等价：

$$\frac{\partial u}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f.$$

为了计算 $\partial u / \partial x_i$ ，我们只需将 $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$ 视为常数并求 u 有关 x_i 的导数。

梯度

假设函数 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 的输入是一个 n 维向量 $\mathbf{x} = [x_1, x_2, \dots, x_n]^\top$ ，输出是标量。函数 $f(\mathbf{x})$ 有关 \mathbf{x} 的梯度是一个由 n 个偏导数组成的向量：

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_n} \right]^\top.$$

为表示简洁，我们有时用 $\nabla f(\mathbf{x})$ 代替 $\nabla_{\mathbf{x}} f(\mathbf{x})$ 。

假设 \mathbf{x} 是一个向量，常见的梯度演算包括

$$\begin{aligned} \nabla_{\mathbf{x}} \mathbf{A}^\top \mathbf{x} &= \mathbf{A}, \\ \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} &= \mathbf{A}, \\ \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} \mathbf{x} &= (\mathbf{A} + \mathbf{A}^\top) \mathbf{x}, \\ \nabla_{\mathbf{x}} \|\mathbf{x}\|^2 &= \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}. \end{aligned}$$

类似地，假设 \mathbf{X} 是一个矩阵，那么

$$\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}.$$

黑塞矩阵

假设函数 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ 的输入是一个 n 维向量 $\mathbf{x} = [x_1, x_2, \dots, x_n]^\top$, 输出是标量。假定函数 f 所有的二阶偏导数都存在, f 的黑塞矩阵 \mathbf{H} 是一个 n 行 n 列的矩阵:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix},$$

其中二阶偏导数

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial}{\partial x_j} \left(\frac{\partial f}{\partial x_i} \right).$$

11.1.3 概率

最后, 我们简要介绍条件概率、期望和均匀分布。

条件概率

假设事件 A 和事件 B 的概率分别为 $\mathbb{P}(A)$ 和 $\mathbb{P}(B)$, 两个事件同时发生的概率记作 $\mathbb{P}(A \cap B)$ 或 $\mathbb{P}(A, B)$ 。给定事件 B , 事件 A 的条件概率

$$\mathbb{P}(A | B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}.$$

也就是说,

$$\mathbb{P}(A \cap B) = \mathbb{P}(B)\mathbb{P}(A | B) = \mathbb{P}(A)\mathbb{P}(B | A).$$

当满足

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B)$$

时, 事件 A 和事件 B 相互独立。

期望

随机变量 X 的期望（或平均值）

$$\mathbb{E}(X) = \sum_x x \mathbb{P}(X = x).$$

均匀分布

假设随机变量 X 服从 $[a, b]$ 上的均匀分布，即 $X \sim U(a, b)$ 。随机变量 X 取 a 和 b 之间任意一个数的概率相等。

11.1.4 小结

- 本节总结了本书中涉及到的有关线性代数、微分和概率的基础知识。

11.1.5 练习

- 求函数 $f(x) = 3x_1^2 + 5e^{x_2}$ 的梯度。

11.1.6 扫码直达讨论区



11.2 使用 Jupyter Notebook

本节介绍如何使用 Jupyter notebook 编辑和运行本书代码。请确保你已按照“安装和运行”一节中的步骤安装好 Jupyter notebook 并获取了本书代码。

11.2.1 在本地编辑和运行本书代码

下面我们介绍如何在本地使用 Jupyter Notebook 来编辑和运行本书代码。假设本书代码所在的本地路径为 “xx/yy/gluon_tutorials_zh-1.0/”。在命令行模式下进入该路径 (`cd xx/yy/gluon_tutorials_zh-1.0`)，然后运行命令 `jupyter notebook`。这时在浏览器打开 `http://localhost:8888`（通常会自动打开）就可以看到 Jupyter notebook 的界面和本书代码所在的各个文件夹，如图 11.1 所示。

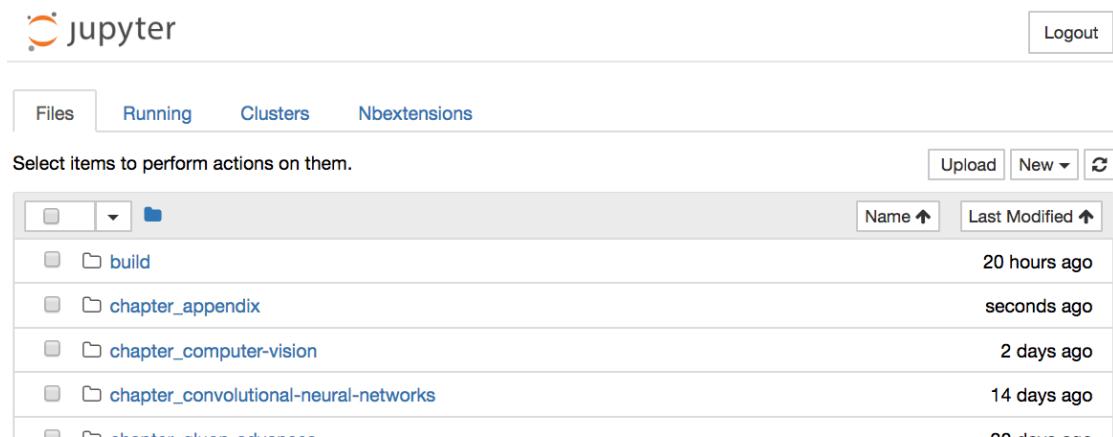


图 11.1: 本书代码所在的各个文件夹。

我们可以通过点击网页上显示的文件夹访问其中的 notebook 文件。它们的后缀通常是 “ipynb”。为了简洁起见，我们创建一个临时的 “test.ipynb” 文件，点击后所显示的内容如图 11.2 所示。该 notebook 包括了格式化文本单元 (markdown cell) 和代码单元 (code cell)。其中格式化文本单元中的内容包括 “这是标题” 和 “这是一段正文。”。代码单元中包括两行 Python 代码。

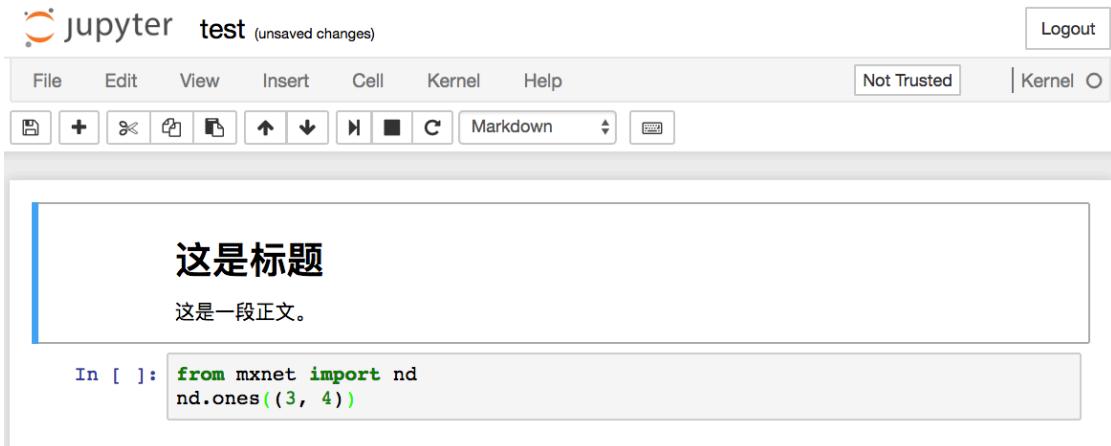


图 11.2: “test.ipynb” 文件包括了格式化文本单元和代码单元。

双击格式化文本单元，我们进入了编辑模式。在该单元的末尾添加一段新文本“你好世界。”，如图 11.3 所示。



图 11.3: 编辑格式化文本单元。

如图 11.4 所示，点击菜单栏的“Cell” → “Run Cells”，运行编辑好的单元。

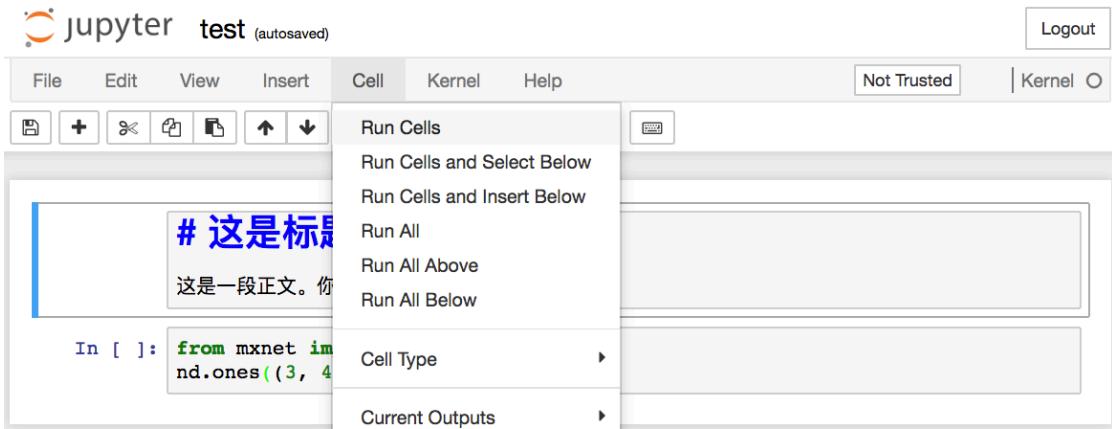


图 11.4: 运行单元。

运行完以后，图 11.5 展示了编辑后的格式化文本单元。

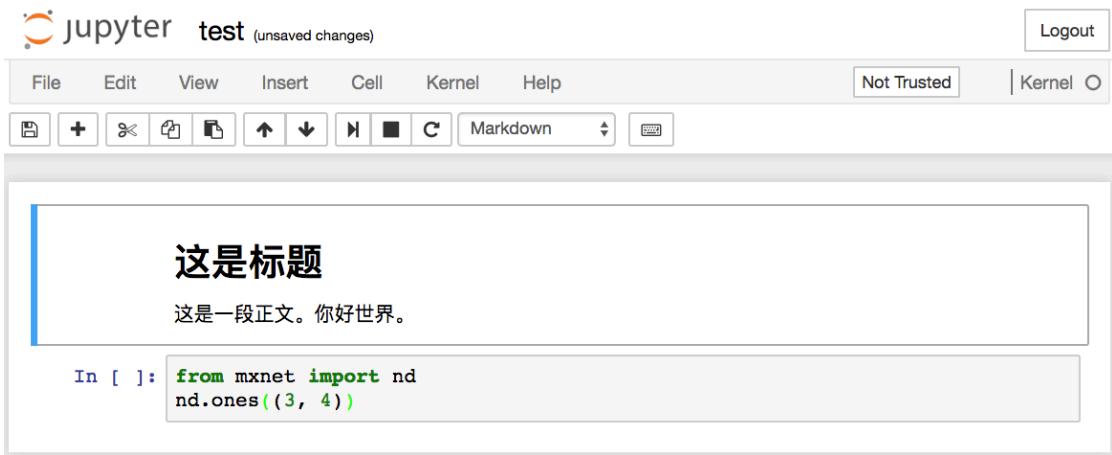


图 11.5: 编辑后的格式化文本单元。

接下来，点击代码单元。在最后一行代码后添加乘以 2 的操作 `* 2`，如图 11.6 所示。

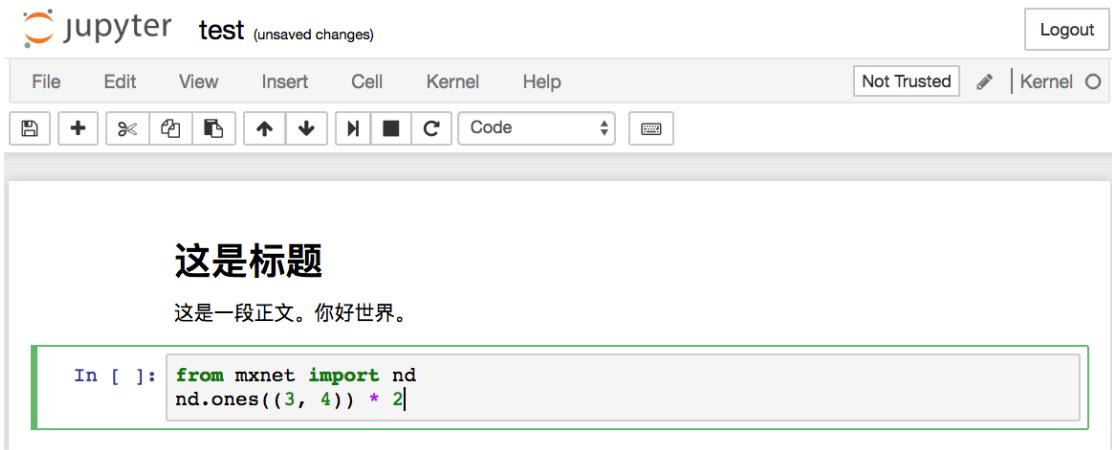


图 11.6: 编辑代码单元。

我们也可以用快捷键运行单元（默认“Ctrl + Enter”），并得到图 11.7 中的输出结果。



图 11.7: 运行代码单元得到输出结果。

当一个 notebook 包含的单元较多时，我们可以点击菜单栏的“Kernel” → “Restart & Run All”，

以运行整个 notebook 中的所有单元。点击菜单栏的“Help”→“Edit Keyboard Shortcuts”后可以根据自己的喜好编辑快捷键。

11.2.2 高级选项

以下是有关使用 Jupyter notebook 的一些高级选项。你可以根据自己的兴趣参考其中内容。

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 putty 等第三方软件也能支持)，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` 打开运行 Jupyter notebook 的远端服务器 `myserver`。我们将在下一节详细介绍如何在 AWS 实例上运行 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
```

11.2.3 小结

- 我们可以使用 Jupyter notebook 编辑和运行本书代码。

11.2.4 练习

- 尝试在本地编辑和运行本书代码。

11.2.5 扫码直达讨论区



11.3 使用 AWS 运行代码

当本地机器的计算资源有限时，我们可以通过云计算服务获取更强大的计算资源来运行本书中的深度学习代码。本节将介绍如何在 AWS（亚马逊的云计算服务）上申请实例并通过 Jupyter notebook 运行代码。本节中的例子基于申请含一个 K80 GPU 的“p2.xlarge”实例和安装 CUDA8.0 及相应 GPU 版本的 MXNet (mxnet-cu80)。申请其他类型的实例或安装其他版本的 MXNet 的方法同本节类似。

11.3.1 申请账号并登陆

首先，我们需要在 <https://aws.amazon.com/> 网站上创建账号。这通常需要一张信用卡。需要注意的是，AWS 中国需要公司实体才能注册。如果你是个人用户，请注册 AWS 全球账号。

登陆 AWS 账号后，点击图 11.8 红框中的“EC2”进入 EC2 面板。

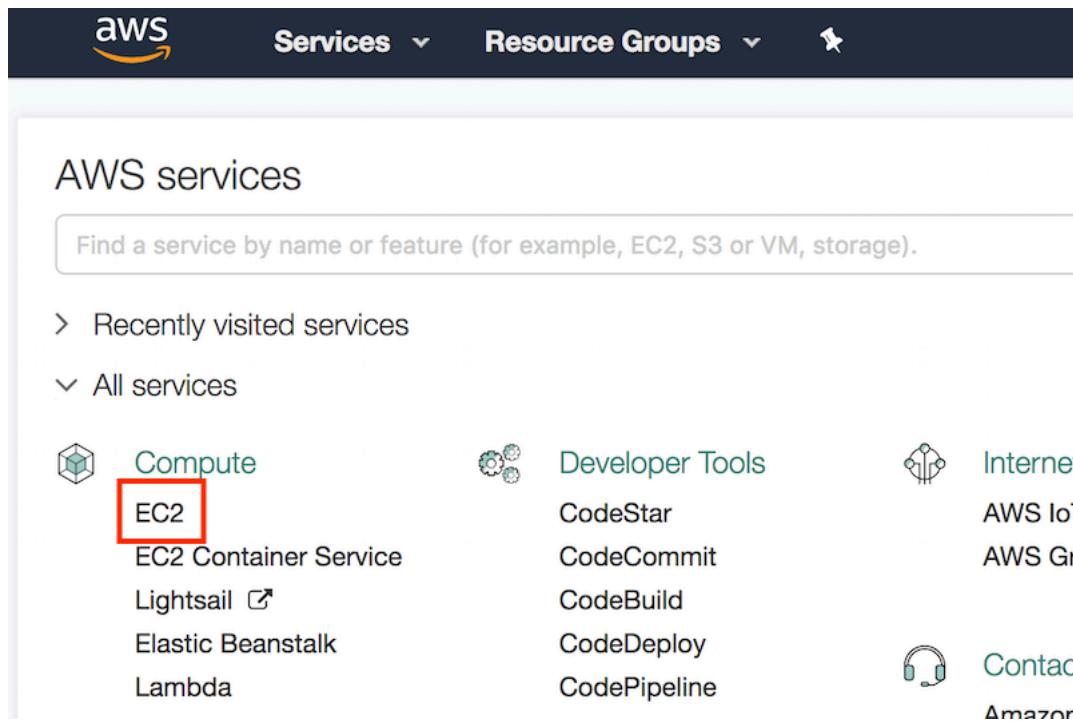


图 11.8: 登陆 AWS 账号。

11.3.2 创建并运行 EC2 实例

图 11.9 展示了 EC2 面板的界面。在图 11.9 右上角红框处选择离我们较近的数据中心来减低延迟。我们可以选离国内较近的亚太地区，例如 Asia Pacific (Seoul)。注意，有些数据中心可能没有 GPU 实例。点击图 11.9 下方红框内“Launch Instance”启动实例。

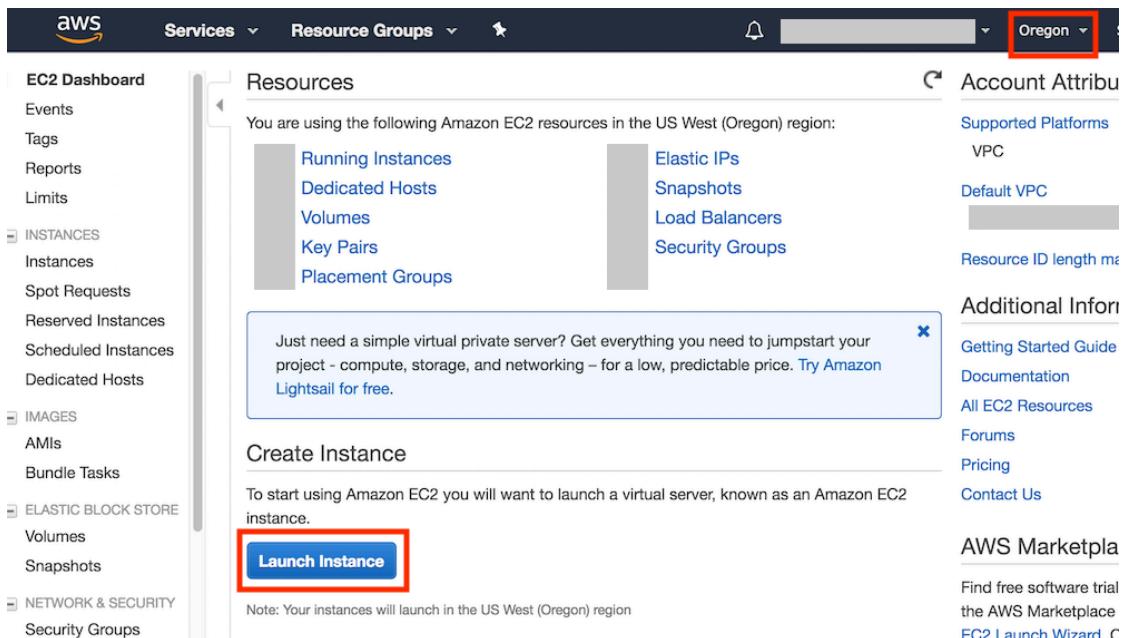


图 11.9: EC2 面板。

图 11.10 的最上面一行显示了配置实例所需的 7 个步骤。在第一步“1. Chosse AMI”中，选择 Ubuntu 16.04 作为操作系统。

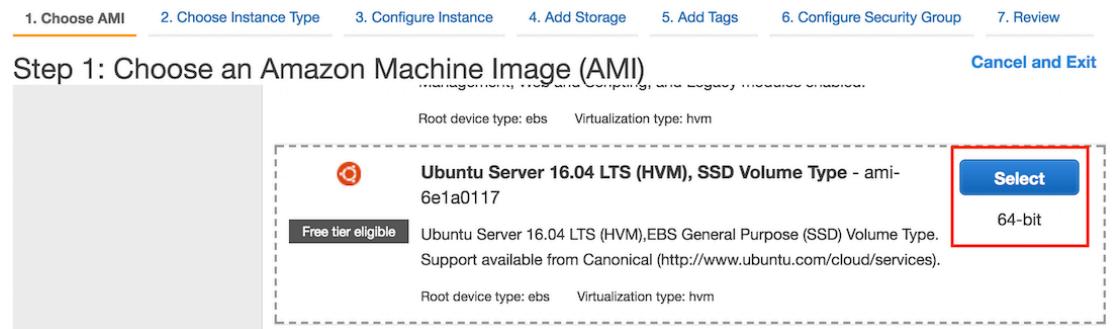


图 11.10: 选择操作系统。

EC2 提供了大量的有着不同配置的实例。如图 11.11 所示，在第二步“2. Chosse Instance Type”中，选择有一个 K80 GPU 的“p2.xlarge”实例。我们也可以选择像“p2.16xlarge”这样有多个

GPU 的实例。如果你想比较不同实例的机器配置和收费，可参考 <https://www.ec2instances.info/>

。

Instance Type	Cores	Memory (GiB)	Storage (SSD)	Local NVMe (TB)	Local SSD (TB)
GPU instances	g2.8xlarge	32	60	2 x 120 (SSD)	
GPU compute	p2.xlarge	4	61	EBS only	

图 11.11: 选择实例。

我们建议在选择实例前先在图 11.9 左栏 “Limits” 里检查下有无数量限制。如图 11.12 所示，该账号的限制是最多在一个区域开一个 “p2.xlarge” 实例。如果需要开更多实例，可以通过点击右边 “Request limit increase” 来申请更大的实例容量。这通常需要一个工作日来处理。

Running On-Demand p2.16xlarge instances	1	Request limit increase
Running On-Demand p2.8xlarge instances	1	Request limit increase
Running On-Demand p2.xlarge instances	1	Request limit increase
Running On-Demand r3.2xlarge instances	20	Request limit increase

图 11.12: 实例的数量限制。

我们将保持第三步 “3. Configure Instance”、第五步 “5. Add Tags” 和第六步 “6. Configure Security Group” 中的默认配置不变。点击第四步 “4.Add Storage”，如图 11.13 所示，将默认的硬盘大小增大到 40GB。注意，安装 CUDA 需要 4GB 左右空间。

Step 4: Add Storage

Volume Type	Device	Snapshot	Size (GiB)	Volume Type	IOPS	Th (M)
Root	/dev/sda1	snap-	40	General Purpose	100 / 3000	N/A

Add New Volume

图 11.13: 修改实例的硬盘大小。

最后，在第七步“7. Review”中点击“Launch”来启动配置好的实例。这时候会提示我们选择用来访问实例的密钥。如果没有的话，可以选择图 11.14 中第一个下拉菜单的“Create a new key pair”选项来生成密钥。之后，我们通过该下拉菜单的“Choose an existing key pair”选项选择生成好的密钥。点击“Launch Instance”启动创建好的实例。

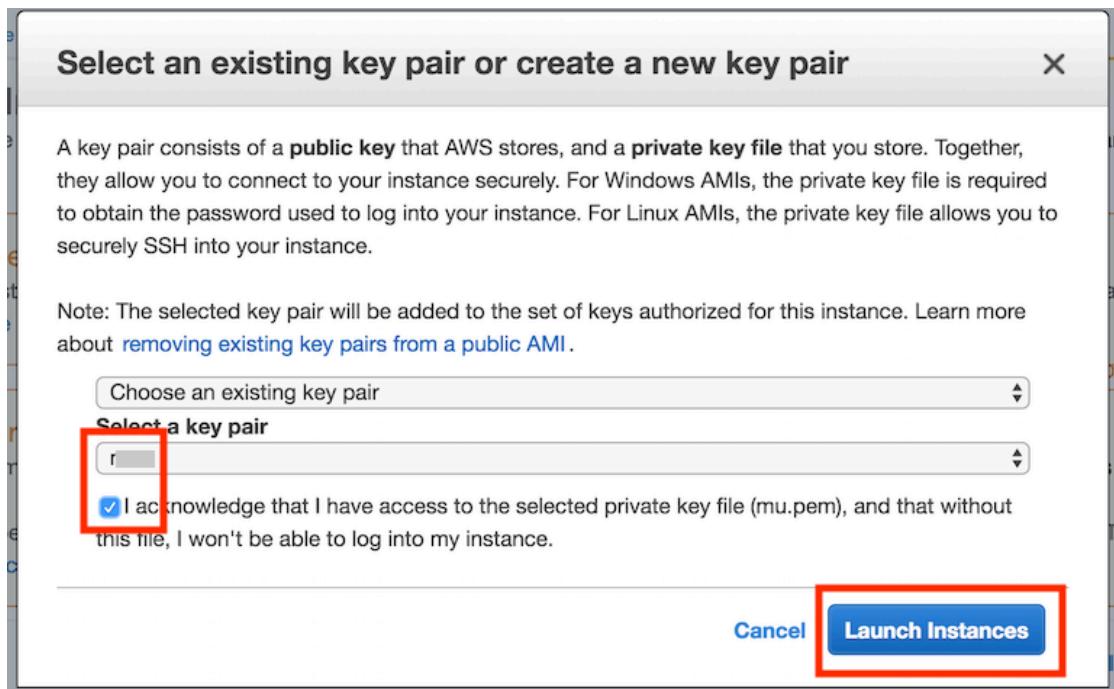


图 11.14: 选择密钥。

点击图 11.15 中的实例 ID 就可以查看该实例的状态了。

Launch Status

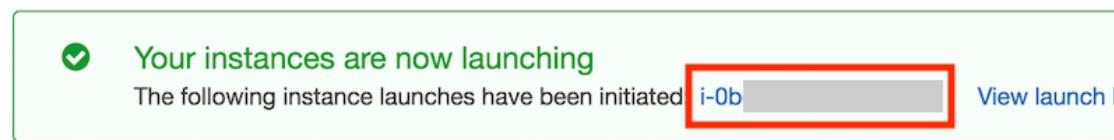


图 11.15: 点击实例 ID。

如图 11.16 所示，当实例状态（Instance State）变绿后，右击实例并选择“Connect”，这时就可以看到访问该实例的方法了。例如在命令行输入

```
ssh -i "/path/to/key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com
```

其中“/path/to/key.pem”是本地存放访问实例的密钥的路径。当命令行提示“Are you sure you want to continue connecting (yes/no)”时，键入“yes”并按回车键即可登录创建好的实例。

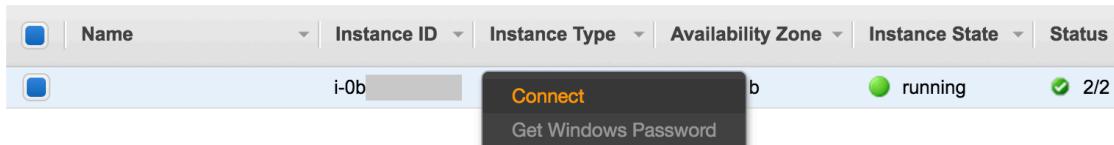


图 11.16: 查看访问开启实例的方法。

11.3.3 安装 CUDA

如果你登录的是一个 GPU 实例，需要下载并安装 CUDA。首先，更新并安装编译需要的包：

```
sudo apt-get update && sudo apt-get install -y build-essential git libgfortran3
```

然后，访问 Nvidia 官网 (<https://developer.nvidia.com/cuda-80-ga2-download-archive>) 获取正确版本的 CUDA8.0 的下载地址，如图 11.17 所示。

Select Target Platform ⓘ

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System	Windows	Linux	Mac OSX	
Architecture ⓘ	x86_64	ppc64le		
Distribution	Fedora	OpenSUSE	RHEL	Centos
	SLES	Ubuntu		
Version	16.04	14.04		
Installer Type ⓘ	runfile (local)	deb (local)	deb (network)	
	cluster (local)			

Download Installers for Linux Ubuntu 16.04 x86_64

The base installer is available for download below.
There is 1 patch available. This patch requires the base installer to be installed first.

▶ Base Installer

Installation Instructions:

1. Run `sudo

Download (1.4 GB) ↴

- Open Link in New Tab
- Open Link in New Window
- Open Link in Incognito Window
- Save Link As...
- Copy Link Address**

图 11.17: 获取 CUDA8.0 的下载地址。

获取下载地址后，我们将下载并安装 CUDA8.0，例如

```
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
```

点击“Ctrl+C”跳出文档浏览，并回答以下几个问题。

```
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
```

当安装完成后，运行下面的命令就可以看到该实例的 GPU 了。

```
nvidia-smi
```

最后，将 CUDA 加入到库的路径中，以方便其他库找到它。

```
echo "export LD_LIBRARY_PATH=\${LD_LIBRARY_PATH}:/usr/local/cuda-8.0/lib64" >>.bashrc
```

11.3.4 获取本书代码并安装 GPU 版的 MXNet

我们已在“安装和运行”一节中介绍了 Linux 用户获取本书代码并安装运行环境的方法。首先，安装 Linux 版的 Miniconda（网址：<https://conda.io/miniconda.html>），例如

```
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
```

这时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
```

安装完成后，运行一次 `source ~/.bashrc` 让 CUDA 和 conda 生效。接下来，下载本书代码、安装并激活 conda 环境

```
mkdir gluon_tutorials_zh && cd gluon_tutorials_zh
curl https://zh.gluon.ai/gluon_tutorials_zh.tar.gz -o tutorials.tar.gz
tar -xzvf tutorials.tar.gz && rm tutorials.tar.gz
conda env create -f environment.yml
source activate gluon
```

默认环境里安装了 CPU 版本的 MXNet。现在我们将它替换成 GPU 版本的 MXNet（1.2.0 版）。

```
pip uninstall mxnet
pip install mxnet-cu80==1.2.0
```

11.3.5 运行 Jupyter notebook

现在，我们可以运行 Jupyter notebook 了：

```
jupyter notebook
```

图 11.18 显示了运行后可能的输出，其中最后一行为 8888 端口下的 URL。

```
(gluon) [~]:~/gluon-tutorials-zh$ jupyter notebook
[I 22:10:29.383 NotebookApp] Writing notebook server cookie secret to /run/user/1000
[I 22:10:29.404 NotebookApp] Serving notebooks from local directory: /home/ubuntu
[I 22:10:29.404 NotebookApp] 0 active kernels
[I 22:10:29.404 NotebookApp] The Jupyter Notebook is running at: http://localhost:8888
[I 22:10:29.404 NotebookApp] Use Control-C to stop this server and shut down all
[W 22:10:29.404 NotebookApp] No web browser found: could not locate runnable browser
[C 22:10:29.404 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
http://localhost:8888/?token=c9c7b
```

图 11.18: 运行 Jupyter notebook 后的输出，其中最后一行为 8888 端口下的 URL。

由于创建的实例并没有暴露 8888 端口，我们可以在本地命令行启动 ssh 从实例映射到本地 8889 端口。

```
# 该命令须在本地命令行运行。
ssh -i "/path/to/key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.compute.amazonaws.com -L
→8889:localhost:8888
```

最后，把图 11.18 中运行 Jupyter notebook 后输出的最后一行 URL 复制到本地浏览器，并将 8888 改为 8889。点击回车键即可从本地浏览器通过 Jupyter notebook 运行实例上的代码。

11.3.6 关闭不使用的实例

因为云服务按使用时长计费，我们通常会在不使用实例时将其关闭。

如果较短时间内还将重新开启实例，右击图 11.16 中的示例，选择“Instance State”→“Stop”将实例停止，等下次使用时选择“Instance State”→“Start”重新开启实例。这种情况下，开启的实例将保留其停止前硬盘上的存储（例如无需再安装 CUDA 和其他运行环境）。然而，停止状态的实例也会因其所保留的硬盘空间而产生少量计费。

如果较长时间内不会重新开启实例，右击图 11.16 中的示例，选择“Image”→“Create”创建镜像。然后，选择“Instance State”→“Terminate”将实例终结（硬盘不再产生计费）。当下次使用时，我们可按本节中创建并运行 EC2 实例的步骤重新创建一个基于保存镜像的实例。唯一的区别在于，在图 11.10 的第一步“1. Choose AMI”中，我们需要通过左栏“My AMIs”选择之前保存的镜像。这样创建的实例将保留镜像上硬盘的存储（例如无需再安装 CUDA 和其他运行环境）。

11.3.7 小结

- 我们可以通过云计算服务获取更强大的计算资源来运行本书中的深度学习代码。

11.3.8 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。

11.3.9 扫码直达讨论区



11.4 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 更便宜且计算更加密集。一方面，相同计算能力的 GPU 的价格一般是 CPU 价格的十分之一。另一方面，一台服务器通常可以搭载 8 块或者 16 块 GPU。因此，GPU 数量可以看作是衡量一台服务器的深度学习计算能力的一个标准。

本节主要针对购买一两台自用 GPU 服务器的个人用户介绍一些 GPU 购买须知。如果你是拥有 100 台机器以上的大公司用户，通常可以考虑 Nvidia Tesla P100 或者 V100，详情请咨询数据中心维护人员。如果你是拥有 10 到 100 台机器的实验室和中小公司用户，如果预算充足，可以考虑 Nvidia DGX-1，否则可以考虑购买如 Supermicro 之类的性价比较高的服务器。

11.4.1 选择 GPU

目前独立 GPU 主要有 AMD 和 Nvidia 两家厂商。其中 Nvidia 在深度学习布局较早，对深度学习框架支持更好。因此，目前大家主要会选择 Nvidia 的 GPU。

Nvidia 有面向个人用户（例如 GTX 系列）和企业用户（例如 Tesla 系列）的两类 GPU。这两类 GPU 的计算能力相当。然而，面向企业用户的 GPU 通常使用被动散热并增加了内存校验，从而更适合数据中心，并通常要比面向个人用户的 GPU 贵上 10 倍。因此，个人用户通常选用 GTX 系列的 GPU。

Nvidia 一般每一两年发布一次新版本的 GPU，例如最近的 GTX 1000 系列。每个系列中会有数个不同的型号，分别对应不同的性能。

GPU 的性能主要由以下三个参数构成：

1. 计算能力。通常我们关心的是 32 位浮点计算能力。当然，特殊情况下也可考虑其他的计算能力，例如用 16 位浮点训练，用 8 位整数预测。
2. 内存大小。当模型越大，或者训练时的批量越大时，所需要的 GPU 内存就越多。
3. 内存带宽。只有当内存带宽足够时才能充分发挥计算能力。

对于大部分用户来说，只要考虑计算能力就可以了。我们建议 GPU 内存尽量不小于 4GB。但如果 GPU 要同时显示图形界面，那么推荐的内存大小至少为 6GB。至于内存带宽，通常厂家已在设计时考虑。

图 11.19 描绘了 GTX 900 和 1000 系列里各个型号的 32 位浮点计算能力和价格的对比。其中价格为 Wikipedia 的建议价格。

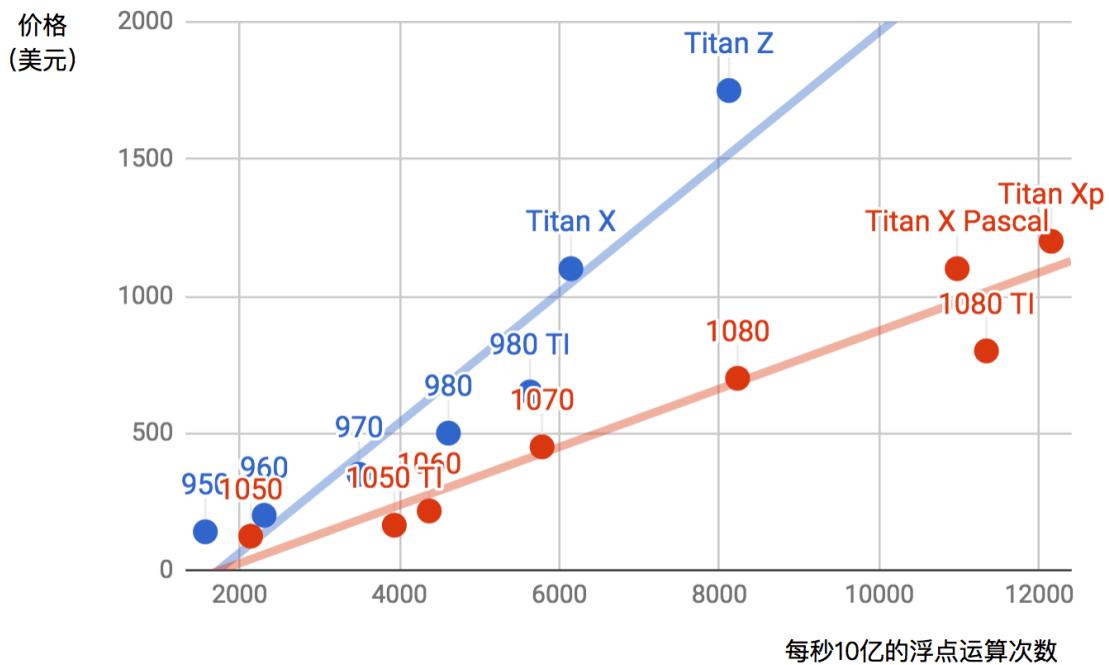


图 11.19: 浮点计算能力和价格的对比。

我们可以从图 11.19 中读出两点信息：

1. 在同一个系列里面，价格和性能大体上成正比。
2. GTX 1000 系列比 900 系列在性价比上高出 2 倍左右。

如果大家继续比较 GTX 较早的系列，也可以发现类似的规律。据此，我们推荐大家在能力范围内，尽可能买较新的 GPU。如果预算充足，直接买最新的 GPU。如果预算相对有限，购买入门的 1050TI 也是个不错的选择。

11.4.2 整机配置

通常，我们主要用 GPU 做深度学习训练。因此，不需要购买高端的 CPU。至于整机配置，尽量参考网上推荐的中高档的配置就好。

不过，考虑到 GPU 的功耗、散热和体积，我们在整机配置上也需要考虑以下三个额外因素。

1. 机箱体积。GPU 尺寸较大，通常考虑较大且自带风扇的机箱。

- 电源。购买 GPU 时需要查一下 GPU 的功耗，例如 50W 到 300W 不等。购买电源要确保功率足够，并不会过载机房的供电。
- 主板的 PCIe 卡槽。推荐使用 PCIe 3.0 16x 来保证充足的 GPU 到主内存的带宽。如果搭载多块 GPU，要仔细阅读主板说明，以确保多块 GPU 一起使用时仍然是 16x 带宽。注意，有些主板搭载 4 块 GPU 时会降到 8x 甚至 4x 带宽。

11.4.3 小结

- 在预算范围之内，尽可能买较新的 GPU。
- 整机配置需要考虑到 GPU 的功耗、散热和体积。

11.4.4 练习

- 浏览本节讨论区中大家有关机器配置方面的交流。

11.4.5 扫码直达讨论区



11.5 gluonbook 函数索引

函数，定义所在章节

- `data_iter`, 线性回归的从零开始实现
- `linreg`, 线性回归的从零开始实现
- `squared_loss`, 线性回归的从零开始实现
- `sgd`, 线性回归的从零开始实现

- `plt`, 线性回归的从零开始实现
- `accuracy`, Softmax 回归的从零开始实现
- `evaluate_accuracy`, Softmax 回归的从零开始实现
- `load_data_fashion_mnist`, Softmax 回归的从零开始实现
- `train_cpu`, Softmax 回归的从零开始实现
- `semilogy`, 欠拟合、过拟合和模型选择
- `to_onehot`, 循环神经网络
- `data_iter_random`, 循环神经网络
- `data_iter_consecutive`, 循环神经网络
- `grad_clipping`, 循环神经网络
- `predict_rnn`, 循环神经网络
- `train_and_predict_rnn`, 循环神经网络

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，我们还在努力将改动同步到英文版）