



使用 MXNet/Gluon 来动手学深度学习

Release 0.6

MXNet Community

Dec 27, 2017

CONTENTS

1 前言	3
1.1 为什么要做这个项目	3
1.2 前言	6
1.3 安裝和使用	8
1.4 GPU 购买指南	18
1.5 在 AWS 上运行教程	22
2 预备知识	33
2.1 机器学习简介	33
2.2 使用 NDArray 来处理数据	51
2.3 使用 autograd 来自动求导	57
3 监督学习	61
3.1 线性回归—从 0 开始	61
3.2 线性回归—使用 Gluon	70
3.3 多类逻辑回归—从 0 开始	73
3.4 多类逻辑回归—使用 Gluon	79
3.5 多层感知机—从 0 开始	81
3.6 多层感知机—使用 Gluon	85
3.7 欠拟合和过拟合	87
3.8 正则化—从 0 开始	96
3.9 正则化—使用 Gluon	101
3.10 丢弃法 (Dropout)—从 0 开始	105
3.11 丢弃法 (Dropout)—使用 Gluon	110
3.12 正向传播和反向传播	112
3.13 实战 Kaggle 比赛——使用 Gluon 预测房价和 K 折交叉验证	116
4 Gluon 基础	131

4.1	创建神经网络	131
4.2	初始化模型参数	136
4.3	序列化—读写模型	142
4.4	设计自定义层	144
4.5	使用 GPU 来计算	148
5	卷积神经网络	155
5.1	卷积神经网络—从 0 开始	155
5.2	卷积神经网络—使用 Gluon	164
5.3	批量归一化—从 0 开始	166
5.4	批量归一化—使用 Gluon	173
5.5	深度卷积神经网络和 AlexNet	175
5.6	VGG：使用重复元素的非常深的网络	182
5.7	网络中的网络	184
5.8	更深的卷积神经网络：GoogLeNet	187
5.9	ResNet：深度残差网络	193
5.10	DenseNet：稠密连接的卷积神经网络	198
6	循环神经网络	205
6.1	循环神经网络—从 0 开始	205
6.2	通过时间反向传播	222
7	优化算法	227
7.1	优化算法概述	227
7.2	梯度下降和随机梯度下降—从 0 开始	231
7.3	梯度下降和随机梯度下降—使用 Gluon	242
7.4	动量法—从 0 开始	249
7.5	动量法—使用 Gluon	255
7.6	Adagrad —从 0 开始	257
7.7	Adagrad —使用 Gluon	262
7.8	RMSProp —从 0 开始	265
7.9	RMSProp —使用 Gluon	270
7.10	Adadelta —从 0 开始	273
7.11	Adadelta —使用 Gluon	278
7.12	Adam —从 0 开始	280
7.13	Adam —使用 Gluon	285
8	Gluon 高级	289

8.1	Hybridize: 更快和更好移植	289
8.2	延迟执行	295
8.3	自动并行	302
8.4	多 GPU 来训练—从 0 开始	305
8.5	多 GPU 来训练—使用 Gluon	313
9	计算机视觉	319
9.1	图片增广	319
9.2	Fine-tuning: 通过微调来迁移学习	329
9.3	使用卷积神经网络的物体检测	338
9.4	SSD —使用 Gluon	350
9.5	语义分割: FCN	371
9.6	样式迁移	386
9.7	实战 Kaggle 比赛——使用 Gluon 对原始图像文件分类 (CIFAR-10)	398
9.8	实战 Kaggle 比赛——使用 Gluon 识别 120 种狗 (ImageNet Dogs)	409

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。

源代码在 <https://github.com/mli/gluon-tutorials-zh> (亲, 给个好评加颗星)

请使用 <http://discuss.gluon.ai/> 来进行讨论

可打印的 PDF 版本在[这里](#)

前言

1.1 为什么要做这个项目

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

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

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

虽然在我认识的人里，好些人能够读一篇论文或者听一个报告后就能问出很好的问题，然后就基本
弄懂了。但我在这个上笨很多。读过的论文就像喝过的水，第二天就不记得了。一定是需要静下心来，
从头到尾实现一篇，跑上几个数据，调些参数，才能心安地觉得懂了。例如在港科大的两年读了
很多论文，但现在反过来看，仍然记得可能就是那两个老老实实动手实现过写过论文的模型了。即
使后来在机器学习这个方向又走了五年，学习任何新东西仍然是要靠动手。



纸上得来终觉浅，绝知此事要躬行

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

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



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

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

为此我们做了（正在做）这五件事情：

1. Eric 和 Sheng 开发了 MXNet 的新前端 Gluon，详细可以参见 Eric 的这篇介绍。这个前端带来跟 Python 更一致的便利的编程环境，不管是 debug 还是在交互上，都比 TensorFlow 之类通过计算图编程的框架更适合学习深度学习。
2. Zack, Alex, Aston 和很多小伙伴一起写了一系列的 notebook 来讲解各个模型。Zack 从

一个外行（他是专业音乐人）和老师（CMU 计算机教授）的角度，从 0 开始讲解和实现各个算法。

3. 我们同时将 notebook 翻译成中文。虽然翻译进度落后了英文版，但对每个翻译了教程都做了大量的改进（之后会 merge 回英文版）
4. 建立了中文社区[discuss.gluon.ai](#)方便大家来讨论和学习。
5. 我们联合[将门](#)在斗鱼上直播一系列课程，深入讲解各个教程。

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

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

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

@mli

1.2 前言

这个项目是我们尝试构建的一个有关深度学习的新型教育资源。我们的目标，是利用 Jupyter notebooks 的优势，将文字、图片、公式、以及（非常重要的）代码呈现在一起。如果这个尝试能够成功，其成果将会是一个极好的资源，它既是一本书、同时也是课程材料、现场教学的补充，甚至有剽窃价值的代码库（此处附上我们的“祝福”）。据我们所了解的，目前仅有很少的资源旨在教授（1）全方位有关现代机器学习的概念，或（2）一本引人入胜的教科书并搭配可运行的代码。相信这次尝试最终能够告诉我们，这种空白是否情有可原。

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

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

于是，我们精心打磨了这套深度学习教程项目。

1.2.1 教程的组织方式

目前我们使用下面这个方式来组织每个具体教程（除背景知识介绍教程）：

1. 引入一个（或者少数几个）新概念
2. 提供一个使用真实数据的完整样例

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

1.2.2 独特的学习体验

这套深度学习教程将为大家呈现以下独特的学习体验。

易用高效的 MXNet

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

双轨学习法

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

通过动手来学习

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

我们坚信，学习深度学习的最好方式就是**动手实现深度学习模型**。

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

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

最后，英文字有句话叫做

“Get hands dirty.”

直译过来就是

“撸起袖子加油干。”

1.3 安装和使用

1.3.1 首次安装

每个教程是一个可以编辑和运行的 Jupyter notebook。运行这些教程需要 Python, Jupyter, 以及最新版 MXNet。

通过 Conda 安装

首先根据操作系统下载并安装Miniconda (Anaconda也可以)。接下来下载所有教程的包 (下载 tar.gz 格式或者[下载 zip 格式](#)均可)。解压后进入文件夹。

例如 Linux 或者 Mac OSX 10.11 以上可以使用如下命令

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

Windows 用户可以用浏览器下载zip 格式并解压, 在解压目录文件资源管理器的地址栏输入 cmd 进入命令行模式。

【可选项】配置下载源来使用国内镜像加速下载:

```
# 优先使用清华 conda 镜像
conda config --prepend channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/
˓→pkgs/free/

# 也可选用科大 conda 镜像
conda config --prepend channels http://mirrors.ustc.edu.cn/anaconda/pkgs/free/
```

然后安装所需的依赖包并激活环境:

```
conda env create -f environment.yml
source activate gluon # 注意 Windows 下不需要 source
```

之后运行下面命令, 然后浏览器打开<http://localhost:8888> (通常会自动打开) 就可以查看和运行各个教程了。

```
jupyter notebook
```

【可选项】国内用户可使用国内 Gluon 镜像加速数据集和预训练模型的下载

- Linux/OSX 用户:

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

- Windows 用户:

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

通过 docker 安装

首先你需要下载并安装docker。例如 Linux 下可以

```
wget -qO- https://get.docker.com/ | sh  
sudo usermod -aG docker  
# 然后 logout 一次
```

然后运行下面命令即可

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

然后浏览器打开<http://localhost:8888>，这时通常需要填 docker 运行时产生的 token。

1.3.2 更新教程

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

用 Conda 更新

先重新下载新的zip或者tar.gz教程包。解压后，使用下面命令更新环境

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

用 Docker 更新

直接下载新的 docker image 就行。

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

使用 Git

如果你熟悉 git，那么直接 pull 并且之后 merge 冲突

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

如果不想 merge 冲突，那么可以在 pull 前用 reset 还原到上一个版本（记得保存有价值的本地修改）

```
git reset --hard
```

之后更新环境

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

使用了 MXNet GPU 版本

这时候 conda update 可能不会自动升级 GPU 版本, 因为默认是安装了 CPU。这时候可以运行了 source activate gluon 后手动更新 MXNet。例如如果安装了 mxnet-cu80 了, 那么

```
pip install -U --pre mxnet-cu80
```

1.3.3 高级选项

使用 GPU

默认安装的 MXNet 只支持 CPU。有一些教程需要 GPU 来运行。假设电脑有 N 卡而且 CUDA7.5 或者 8.0 已经安装了, 那么先卸载 CPU 版本

```
pip uninstall mxnet
```

然后选择安装下面版本之一:

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

【可选项】国内用户可使用豆瓣 pypi 镜像加速下载:

```
pip install --pre mxnet-cu75 -i https://pypi.douban.com/simple # CUDA 7.5  
pip install --pre mxnet-cu80 -i https://pypi.douban.com/simple # CUDA 8.0
```

使用 notedown 插件来读写 github 源文件

注意: 这个只推荐给想上 github 提交改动的小伙伴。我们源代码是用 markdown 格式来存储, 而不是 jupyter 默认的 ipynb 格式。我们可以用 notedown 插件来读写 markdown 格式。下面命令下载源代码并且安装环境:

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

然后安装 notedown, 运行 Jupyter 并加载 notedown 插件:

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

【可选项】默认开启 notedown 插件

首先生成 jupyter 配置文件 (如果已经生成过可以跳过)

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

将下面这一行加入到生成的配置文件的末尾 (Linux/macOS 一般在 `~/.jupyter/jupyter_notebook_config.py`)

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

之后就只需要运行 `jupyter notebook` 即可。

在远端服务器上运行 Jupyter

Jupyter 的一个常用做法是在远端服务器上运行, 然后通过 `http://myserver:8888` 来访问。有时候防火墙阻挡了直接访问对应的端口, 但 ssh 是可以的。如果本地机器是 linux 或者 mac (windows 通过第三方软件例如 putty 应该也能支持), 那么可以使用端口映射

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

然后我们可以使用 `http://localhost:8888` 打开远端的 Jupyter。

运行计时

我们可以通过 ExecutionTime 插件来对每个 cell 的运行计时。

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

1.3.4 老中医自检程序

用途

本教程提供了一系列自检程序供没有成功安装或者运行报错的难民进行自救，如果全篇都没找到药方，希望可以自己搜索问题，欢迎前往 <https://discuss.gluon.ai> 提问并且帮他人解答。

通过 Conda 安装

确保 conda 已经安装完成，并且可以在命令行识别到“conda -version”

症状

```
-bash: conda: command not found ∕’ conda ‘不是内部或外部命令，也不是可运行的程序
```

病情分析

conda 不在系统搜索目录下，无法找到 conda 可执行文件

药方

```
# linux 或者 mac 系统
export PATH=/path/to/miniconda3/bin:$PATH
# windows 用 set 或者 setx
set PATH=C:\path\to\miniconda3\bin;%PATH%
```

```
完成后命令行测试 "conda --version"
如果显示类似于 “conda 4.3.21”，则症状痊愈
```

症状

```
Conda 安装正常, conda env -f environment.yml 失败
```

病情分析

如果在国内的网络环境下，最大的可能是连接太慢，用国内镜像加速不失为一良方

药方

- conda config --prepend channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/
- 如果是 miniconda 可以改用 Anaconda

病情分析

失败后重新尝试 conda env -f environment.yml 会报错

药方

conda info -e 查看失败信息，建议删除失败的 env: conda env remove -name gluon -all

手动 pip 安装

症状: **pip install mxnet** 失败

病情分析

pip 本身不存在, pip -version 不能正确显示 pip 版本号和安装目录

药方

参考 <http://pip-cn.readthedocs.io/en/latest/installing.html> 安装 pip

病情分析

pip 版本太低

药方

```
pip install --upgrade pip
```

病情分析

无法找到匹配的 wheel, No matching distribution found for mxnet>=0.11.1b20170902

药方

确保系统被支持, 比如 Ubuntu 14.04/16.04, Mac10.11/10.12(10.10 即将支持), Windows 10(win7 未测试), 如果都符合, 可以试试命令

```
python -c "import pip; print(pip.pep425tags.get_supported())"
```

然后上论坛讨论: <https://discuss.gluon.ai>

症状: pip install mxnet 成功, 但是 import mxnet 失败

病情分析

如果看到这样的错误

```
ImportError: No module named mxnet
```

python 无法找到 mxnet, 有可能系统上有多个 python 版本, 导致 pip 和 python 版本不一致

药方

找到 pip 的安装目录

```
pip --version
```

找到 python 安装目录

```
which python  
# or  
whereis python  
# or  
python -c "import os, sys; print(os.path.dirname(sys.executable))"
```

如果 pip 目录和 python 目录不一致, 可以改变默认加载的 python, 比如

```
python3 -c "import mxnet as mx; print(mx.__version__)"
```

或者用和 python 对应的 pip 重新安装 mxnet

```
pip3 install mxnet --pre  
pip2 install mxnet --pre
```

如果不是简单的 python2/3 的问题, 推荐修复默认调用的 python。

病情分析

假设你看到这个错误:

```
ImportError: libgfortran.so.3: cannot open shared object file: No such file or  
directory
```

药方

这个一般发生在 Linux 下。安装 libgfortran 就好, 例如 Ubuntu 下可以 sudo apt-get install libgfortran

症状: 可以 import mxnet, 但是版本不正常 (< 0.11.1b20170908)

病情分析

安装时没有指定最新的版本

药方

可以使用 `pip install mxnet -upgrade -pre` 安装最新的 mxnet

病情分析

由于系统的问题，无法正确安装最新版本，参考 `No matching distribution found for mxnet>=0.11.1b20170902`

Jupyter Notebook

症状：打开 **notebook** 乱码

病情分析

Windows 下不支持编码？

未测试药方

把 md 文件用文本编辑器保存为 GBK 编码

其他

症状：Windows 下 curl, tar 失败

病情分析

Windows 默认不支持 curl, tar

药方

下载和解压推荐用浏览器和解压软件，手动拷贝

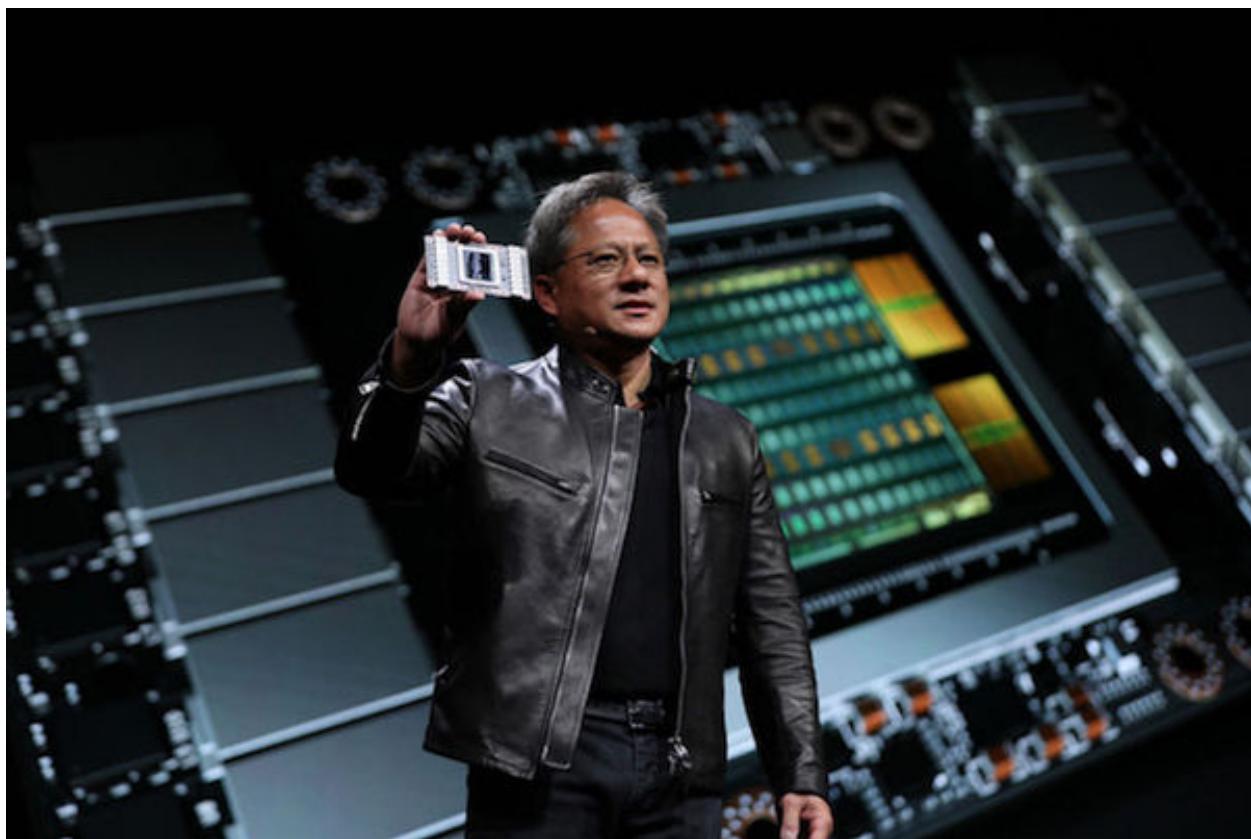
最后

如果你尝试了很多依然一头雾水，可以试试 docker 安装：
<https://zh.gluon.ai/install.html#docker>

吐槽和讨论欢迎点[这里](#)

1.4 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（达到同样的计算能力 GPU 一般便宜 10 倍），而且计算更加密集（一台服务器可以搭配 8 块或者 16 块 GPU）。因此 GPU 数量通常是衡量深度学习计算能力的一个标准，同时 Nvidia 的创始人 Jensen Huang 也被人称深度学习教父。



(Nvidia CEO 黄教主和他的战术核武器)

本章我们简要介绍 GPU 的购买须知。这里主要针对个人用户购买一两台自用的 GPU 服务器。而不是针对需要购买

- 100+ 台机器的大公司用户。请咨询专业数据中心维护人员，通常你们会考虑 Nvidia Tesla

P100 或者 V100。你可以完全跳过此节。

- 10+ 台机器的实验室和中小公司用户：不缺钱可以上 Nvidia DGX-1，不然可以考虑购买如 Supermicro 之类性价比较高的服务器。此节的一些内容可以做为参考。

1.4.1 选择 GPU

目前独立 GPU 主要有 AMD 和 Nvidia 两家厂商。其中 Nvidia 由于深度学习布局较早，深度学习框架支持更好，因此目前主要会选择 Nvidia 的卡。

Nvidia 卡有面向个人用户（例如 GTX 系列）和企业用户（例如 Tesla 系列）两种。企业用户卡通常使用被动散热和增加了内存校验从而更加适合数据中心。但计算能力上两者相当。企业卡通常要贵上 10 倍，因此个人用户通常选用 GTX 系列。

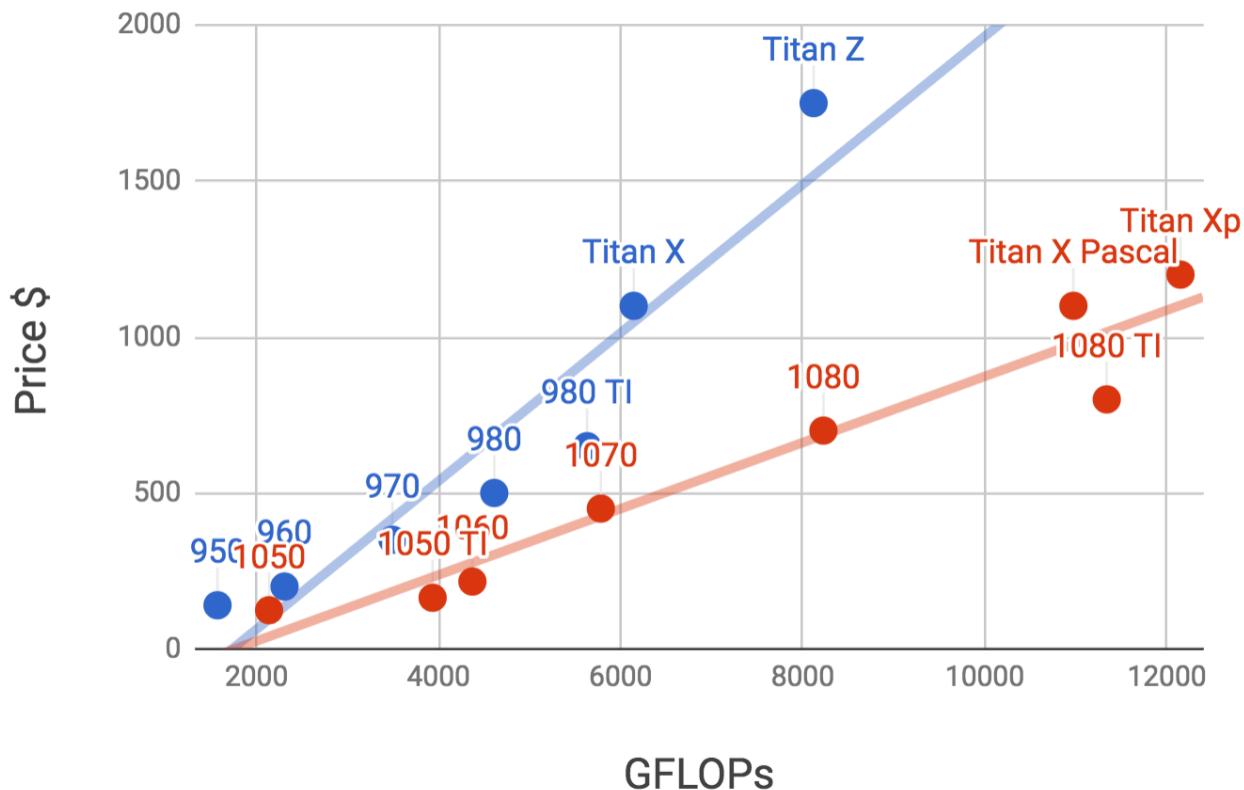
Nvidia 一般每一两年会更新一次大版本，例如目前最新的是 1000 系列。每个系列里面会有数个不同型号，对应不同的性能。

GPU 的性能主要由下面三个主要参数构成：

1. 计算能力。通常我们关心的是 32 位浮点计算能力。当然，对于高玩来说也可以考虑 16 位浮点用来训练，8 位整数来预测。
2. 内存大小。神经网络越深，或者训练时批量大小越大，所需要的 GPU 内存就越多。
3. 内存带宽。内存带宽要足够才能发挥出所有计算能力。

对于大部分用户来说，只要考虑计算能力就行了。内存不要太小就好，例如不要小于 4GB。如果显卡同时要用来显示图形界面，那么推荐 6G 内存。内存带宽可以让厂家来纠结。

下图画了 900 和 1000 系列里各个卡的 32 位浮点计算能力和价格的对比（价格是 wikipedia 的推荐价格，真实价格通常会有浮动）。



我们可以读出两点信息：

1. 在同一个系列里面，通常价格和性能成正比
2. 1000 系列性价比 900 高 2 倍左右。

如果大家继续比较 GTX 前面几代，也发现规律是类似的。根据这个我们推荐

1. 买新不买旧，因为目前看来 GPU 性能还是在快速迭代，贬值较快。
2. 量力购买。不缺钱直接上最好的，但入门的 1050TI 也不错。

1.4.2 整机配置

如果主要是用 GPU 来做计算，或者说主要是做深度学习训练，不需要购买高端的 CPU。可以将主要预算花费在 GPU 上。所以整机配置可以参考网上推荐的中高档就好。

不过由于 GPU 的功耗，散热和体积，需要一些额外考虑。

- 机箱体积。GPU 尺寸较大，通常不考虑太小的机箱。而且机箱自带的风扇要好。（下图里我们曾尝试在一个中等机箱里塞满 4 卡导致散热不好烧了 2 块 GPU。）



- 电源。购买 GPU 时需要查下 GPU 的功耗, 50w 到 300w 不等。因此买电源时需要功率足够的。(我们倒是一开始就考虑了这个, 但忘了不过载机房供电。下面是 5 台机器满负荷运行时烧掉了一个 30A 的电源接口。)



- 主板的 PCIe 卡槽。推荐使用 PCIe 3.0 16x 来保证足够的 GPU 到主内存带宽。如果是多卡的话，要仔细看主板说明，保证多卡一起使用时仍然是 16x 带宽。（有些主板插 4 卡时会降到 8x 甚至 4x）

对于更具体的配置可以参考我们[走过的一些弯路](#)，和来讨论区[交流大家的机器配置](#)。

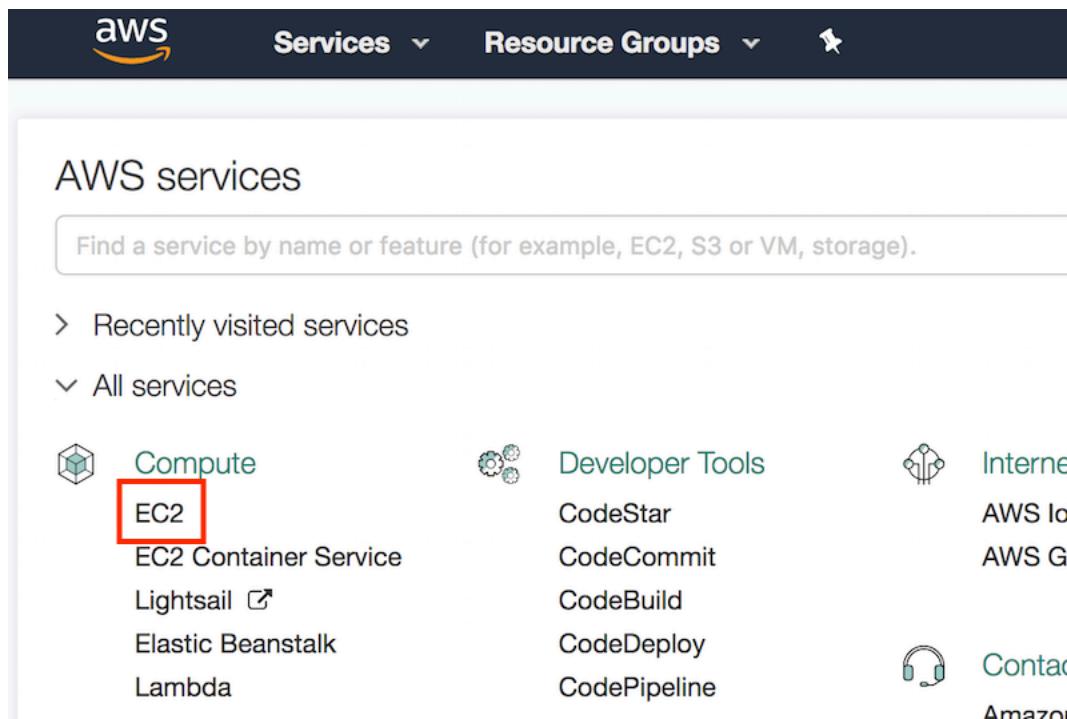
1.5 在 AWS 上运行教程

本节我们一步步讲解如何从 0 开始在 AWS 上申请 CPU 或者 GPU 机器并运行教程。

1.5.1 申请账号并登陆

首先我们需要在<https://aws.amazon.com/>上面创建账号，通常这个需要一张信用卡。不熟悉的同学可以自行搜索“如何注册 aws 账号”。【注意】AWS 中国需要公司实体才能注册，个人用户请注册 AWS 全球账号。

登陆后点击 EC2 进入 EC2 面板：



1.5.2 选择并运行 EC2 实例

【可选】进入面板后可以在右上角选择离我们较近的数据中心来减低延迟。例如国内用户可以选亚太地区数据中心。

【注意】有些数据中心可能没有 GPU 实例。

然后点击启动实例来选择操作系统和实例类型。

The screenshot shows the AWS EC2 Dashboard. On the left, there's a sidebar with links for EC2 Dashboard, Events, Tags, Reports, Limits, Instances, AMIs, Elastic Block Store, and Network & Security. The main area displays resource statistics: 7 Running Instances, 2 Elastic IPs, 0 Dedicated Hosts, 6 Snapshots, 9 Volumes, 0 Load Balancers, 1 Key Pairs, 39 Security Groups, and 0 Placement Groups. A callout box highlights the 'Oregon' region selection in the top right corner. Below the stats, a promotional box for Amazon Lightsail is shown. The 'Create Instance' section includes a prominent 'Launch Instance' button, which is also highlighted with a red box.

在接下来的操作系统里面选 Ubuntu 16.06:

This screenshot shows the 'Step 1: Choose an Amazon Machine Image (AMI)' page. It lists available AMIs, with one entry for 'Ubuntu Server 16.04 LTS (HVM), SSD Volume Type - ami-6e1a0117' highlighted. This entry is marked as 'Free tier eligible'. To the right of the list is a 'Select' button, which is also highlighted with a red box. The top navigation bar shows steps 1 through 7, and a 'Cancel and Exit' link.

EC2 提供大量的有着不同配置的实例。这里我们选择了 p2.xlarge, 它有一个 K80 GPU。我们也可以选择有更多 GPU 的实例例如 p2.16xlarge, 或者有新一点 GPU 的 g3 系列。我们也可以选择只有 CPU 的实例, 例如 c4 系列。每个不同实例的机器配置和收费可以参考 [ec2instances.info](#).

This screenshot shows the 'Step 2: Choose an Instance Type' page. It lists instance types in two categories: GPU instances and GPU compute. The 'p2.xlarge' instance type under GPU compute is highlighted with a red box. The top navigation bar shows steps 1 through 6.

【注意】选择某个类型的样例前我们需要在 **Limits** 里检查下这个是不是有数量限制。例如这个账号的 p2.xlarge 限制是最多一个区域开一个。如果需要更多，需要点右边来申请更多的实例容量（通常需要一个工作日来处理）。



Events	Running On-Demand p2.16xlarge instances	1	Request limit increase
Tags	Running On-Demand p2.8xlarge instances	1	Request limit increase
Reports	Running On-Demand p2.xlarge instances	1	Request limit increase
Limits	Running On-Demand r3.2xlarge instances	20	Request limit increase

然后我们在存储那里将默认的硬盘从 8GB 增大的 40GB. 因为安装 CUDA 需要 4GB 空间，所以最小推荐 20GB (只有 CPU 的话不需要 CUDA，可以更少)。

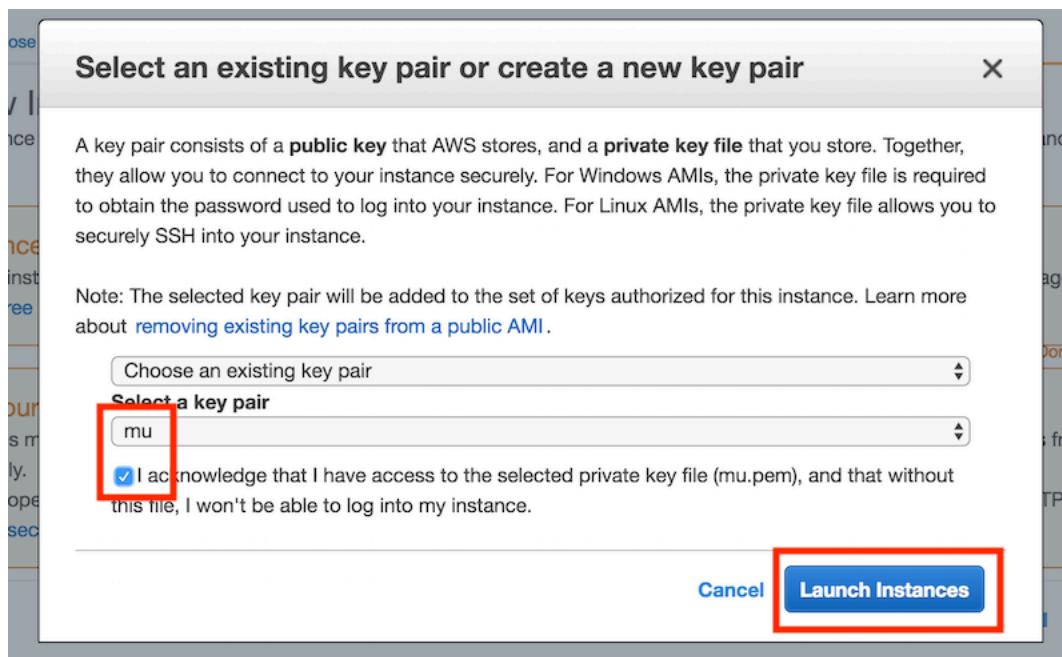
1. Choose AMI 2. Choose Instance Type 3. Configure Instance **4. Add Storage** 5. Add Tags

Step 4: Add Storage

Volume Type	Device	Snapshot	Size (GiB)	Volume Type	IOPS	Throughput (Mbit/s)
Root	/dev/sda1	snap-02cc1e40d2e121683	40	General Purpose S	100 / 3000	N/A

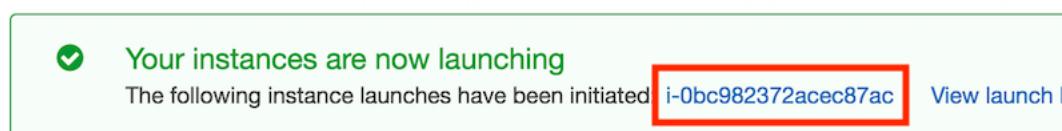
[Add New Volume](#)

其他的项我们都选默认，然后可以启动了。这时候会跳出选项选择 **key pair**，这是用来之后访问机器的秘钥 (EC2 默认不支持密码访问)。如果没有的话可以选择生成一对秘钥。

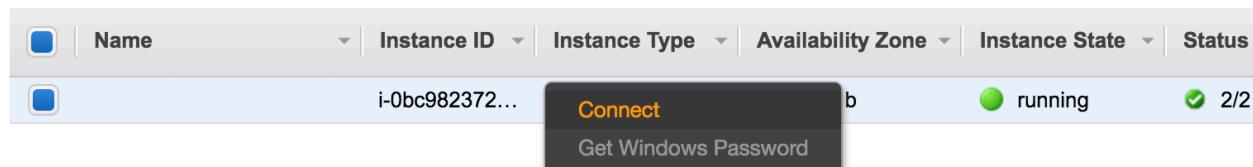


然后点击实例的 ID 可以查看当前实例的状态

Launch Status



状态变绿后右击点 Connect 就可以看到如何访问这个实例的说明了



例如我们这里

```
Mus-MacBook-Pro:~ muli$ ssh ubuntu@ec2-34-203-223-63.compute-1.amazonaws.com
The authenticity of host 'ec2-34-203-223-63.compute-1.amazonaws.com (34.203.223.63)' can't be established.
ECDSA key fingerprint is SHA256:8aPhw5p745TdmsUxnWb+MpWF+vvMKddahBKB12eYi8I.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'ec2-34-203-223-63.compute-1.amazonaws.com,34.203.223.63' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-1022-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

Get cloud support with Ubuntu Advantage Cloud Guest:
  http://www.ubuntu.com/business/services/cloud
```

安装依赖包

成功登陆后我们先更新并安装编译需要的包。

```
sudo apt-get update && sudo apt-get install -y build-essential git
↪ libgfortran3
```

安装 CUDA

【注意】只有 CPU 的实例可以跳过步骤。

我们去 Nvidia 官网下载 CUDA 并安装。选择正确的版本并获取下载地址。

【注意】目前 CUDA 默认下载 9.0 版，但 mxnet-cu90 的 daily build 还不完善。建议使用下面命令安装 8.0 版。

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System

Architecture 

Distribution

Version

Installer Type 



Download Installer for Linux Ubuntu 16.04 x86_64

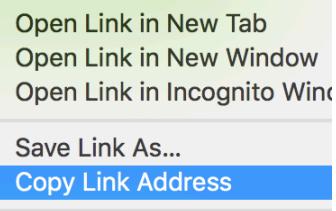
The base installer is available for download below.

› Base Installer

Installation Instructions:

1. Run `sudo sh cuda_9.0.176_384.81_linux.run`
2. Follow the command-line prompts

Do



然后使用 wget 下载并且安装

```
wget https://developer.nvidia.com/compute/cuda/8.0/Prod2/local_installers/  
→cuda_8.0.61_375.26_linux-run
```

```
sudo sh cuda_8.0.61_375.26_linux-run
```

这里需要回答几个问题。

```
accept/decline/quit: accept
Install NVIDIA Accelerated Graphics Driver for Linux-x86_64 375.26?
(y)es/(n)o/(q)uit: y
Do you want to install the OpenGL libraries?
(y)es/(n)o/(q)uit [ default is yes ]: y
Do you want to run nvidia-xconfig?
(y)es/(n)o/(q)uit [ default is no ]: n
Install the CUDA 8.0 Toolkit?
(y)es/(n)o/(q)uit: y
Enter Toolkit Location
[ default is /usr/local/cuda-8.0 ]:
Do you want to install a symbolic link at /usr/local/cuda?
(y)es/(n)o/(q)uit: y
Install the CUDA 8.0 Samples?
(y)es/(n)o/(q)uit: n
```

安装完成后运行

```
nvidia-smi
```

就可以看到这个实例的 GPU 了。最后将 CUDA 加入到 library path 方便之后安装的库找到它。

```
echo "export LD_LIBRARY_PATH=\${LD_LIBRARY_PATH}:/usr/local/cuda-8.0/lib64" >>
~/.bashrc
```

安装 MXNet

先安装 Miniconda

```
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh
```

需要回答下面几个问题

```
Do you accept the license terms? [yes|no]
[no] >>> yes
```

```
Do you wish the installer to prepend the Miniconda3 install location  
to PATH in your /home/ubuntu/.bashrc ? [yes|no]  
[no] >>> yes
```

运行一次 bash 让 CUDA 和 conda 生效。

下载本教程，安装并激活 conda 环境

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

默认环境里安装了只有 CPU 的版本。现在我们替换成 GPU 版本。

```
pip uninstall -y mxnet  
pip install --pre mxnet-cu80
```

同时安装 notedown 插件来让 jupyter 读写 markdown 文件。

```
pip install https://github.com/mli/notedown/tarball/master  
jupyter notebook --generate-config  
echo "c.NotebookApp.contents_manager_class = 'notedown.NotedownContentsManager'  
↪ '" >> ~/.jupyter/jupyter_notebook_config.py
```

1.5.3 运行

并运行 Jupyter notebook。

```
jupyter notebook
```

如果成功的话会看到类似的输出

```
(gluon) ubuntu@ip-172-31-0-171:~/gluon-tutorials-zh$ jupyter notebook
[I 22:10:29.383 NotebookApp] Writing notebook server cookie secret to /run/user/1
[I 22:10:29.404 NotebookApp] Serving notebooks from local directory: /home/ubuntu
[I 22:10:29.404 NotebookApp] 0 active kernels
[I 22:10:29.404 NotebookApp] The Jupyter Notebook is running at: http://localhost
[I 22:10:29.404 NotebookApp] Use Control-C to stop this server and shut down all
[W 22:10:29.404 NotebookApp] No web browser found: could not locate runnable brow
[C 22:10:29.404 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
http://localhost:8888/?token=c9c7b7d48caedc5c039535d777450f61dff7c0ad6fc1
```

因为我们的实例没有暴露 8888 端口，所以我们可以通过 ssh 映射到本地

```
ssh -L8888:localhost:8888 ubuntu@your-ip.amazonaws.com
```

然后把 jupyter log 里的 URL 复制到本地浏览器就行了。

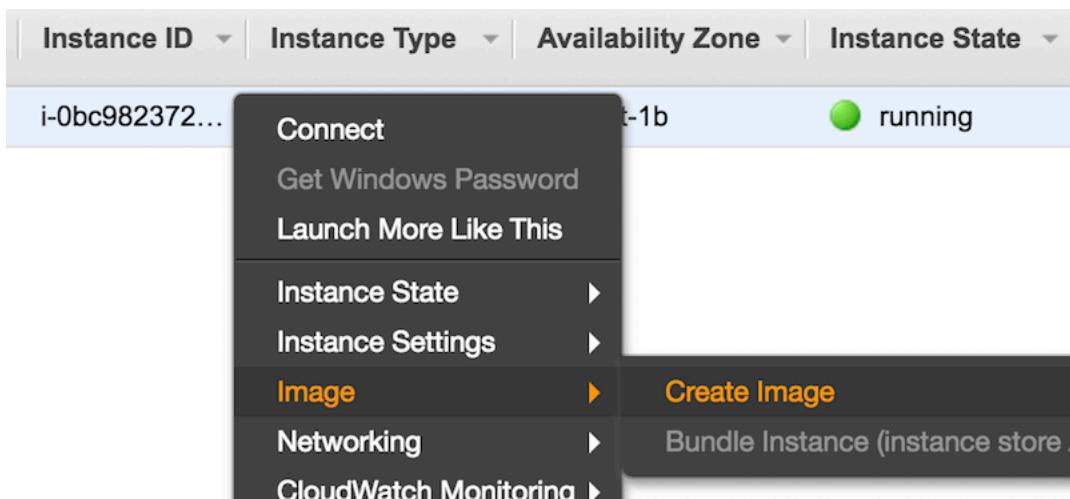
【注意】如果本地运行了 Jupyter notebook，那么 8888 端口就可能被占用了。要么关掉本地 jupyter，要么把端口映射改成别的。例如，假设 aws 使用默认 8888 端口，我们可以通过 ssh 映射到本地 8889 端口：

```
ssh -N -f -L localhost:8889:localhost:8888 ubuntu@your-ip.amazonaws.com
```

然后在本地浏览器打开 localhost:8889，这时会提示需要 token 值。接下来，我们将 aws 上 jupyter log 里的 token 值（例如上图里：...localhost:8888/?token=token 值）复制粘贴即可。

1.5.4 后续

因为云服务按时间计费，通常我们不用时需要把样例关掉，到下次要用时再开。如果是停掉 (Stop)，下次可以直接继续用，但硬盘空间会计费。如果是终结 (Termination)，我们一般会先把操作系统做镜像，下次开始时直接使用镜像 (AMI)（上面的教程使用了 Ubuntu 16.06 AMI）就行了，不需要再把上面流程走一次。



每次重新开始后，我们建议升级下教程（记得保存自己的改动）

```
cd gluon-tutorials-zh  
git pull
```

和 MXNet 版本

```
source activate gluon  
pip install -U --pre mxnet-cu80
```

1.5.5 总结

云上可以很方便的获取计算资源和配置环境

1.5.6 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。

预备知识

2.1 机器学习简介

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

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

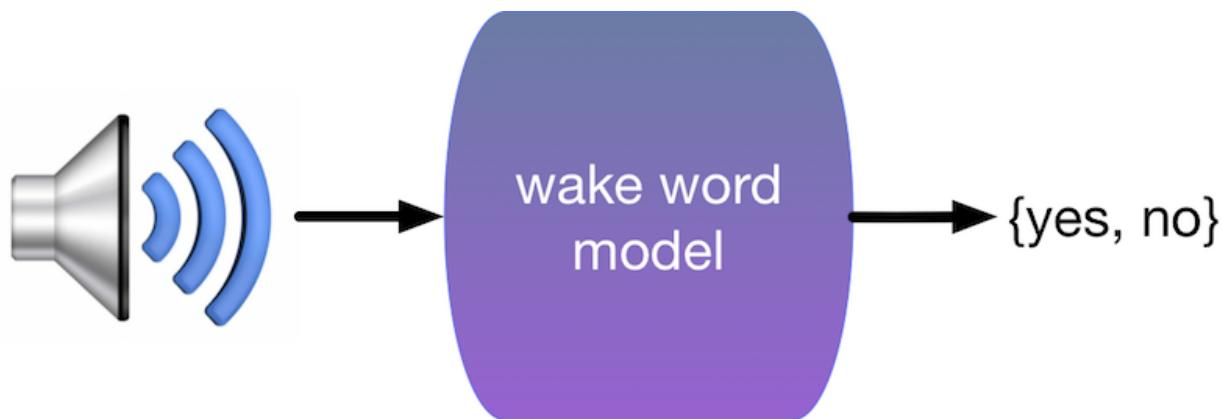
2.1.1 一个例子

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

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

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

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



如果你被这个问题难住了，不用担心。这就是我们为什么需要机器学习。

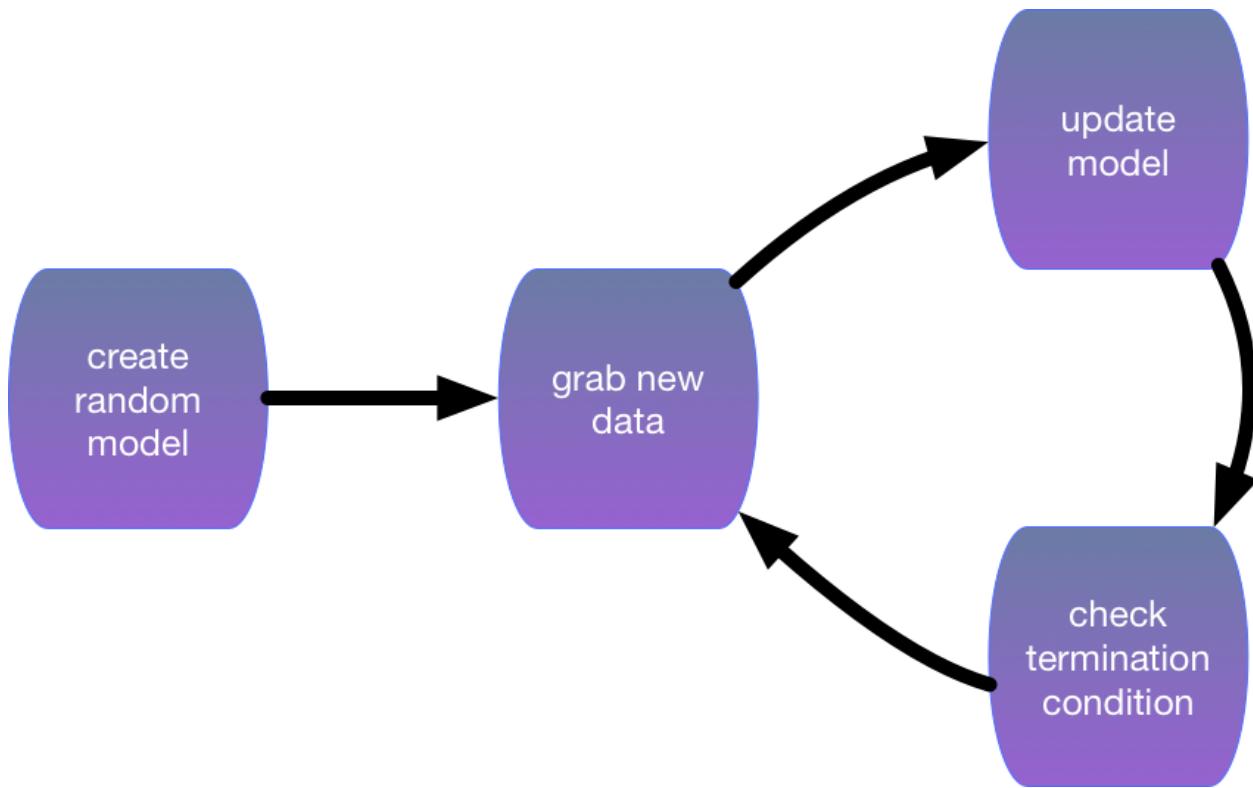
虽然我们不知道怎么告诉机器去把语音信号转成对应的字符串，但我们可以。换句话说，就算你不清楚怎么编写程序，好让机器识别出唤醒词“Alexa”，你自己完全能够识别出“Alexa”这个词。由此，我们可以收集一个巨大的**数据集（data set）**，里面包含了大量语音信号，以及每个语音型号**是否**对应我们需要的唤醒词。使用机器学习的解决方式，我们并非直接设计一个系统去**准确地**辨别唤醒词，而是写一个灵活的程序，并带有大量的**参数（parameters）**。通过调整这些参数，我们能够改变程序的行为。我们将这样的程序称为模型（models）。总体上看，我们的模型仅仅是一个机器，通过某种方式，将输入转换为输出。在上面的例子中，这个模型的**输入（input）**是一段语音信号，它的输出则是一个回答 {yes, no}，告诉我们这段语音信号是否包含了唤醒词。

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

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

换言之，我们需要用数据训练机器学习模型，其过程通常如下：

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



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

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

喵	喵	汪	汪

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

2.1.2 眼花缭乱的机器学习应用

机器学习背后的核心思想是，设计程序使得它可以在执行的时候提升它在某任务上的能力，而非直接编写程序的固定行为。机器学习包括多种问题的定义，提供很多不同的算法，能解决不同领域的

各种问题。我们之前讲到的是一个讲监督学习 (**supervised learning**) 应用到语言识别的例子。

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

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

2.1.3 用代码编程和用数据编程

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

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

传统的实现是这样的：

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

```
1 2 fizz 4 buzz fizz 7 8 fizz buzz 11 fizz 13 14 fizzbuzz 16 17 fizz 19 buzz fizz 22 23 fi
```

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

- 数据 X [1, 2, 3, 4, ...] 和标注 Y ['fizz', 'buzz', 'fizzbuzz', identity]
- 训练数据，也就是系统输入输出的实例。例如 [(2, 2), (6, fizz), (15, fizzbuzz), (23, 23), (40, buzz)]

- 从输入数据中抽取的特征，例如 $x \rightarrow [(x \% 3), (x \% 5), (x \% 15)]$.

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

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

2.1.4 机器学习最简要素

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

数据（Data）

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

- 图片：**你的手机图片，里面可能包含猫、狗、恐龙、高中同学聚会或者昨天的晚饭；卫星照片；医疗图片例如超声、CT 扫描以及 MRI 等等。
- 文本：**邮件、高中作文、微博、新闻、医嘱、书籍、翻译语料库和微信聊天记录。
- 声音：**发送给智能设备（比如 Amazon Echo, iPhone 或安卓智能机）的声控指令、有声书籍、电话记录、音乐录音，等等。
- 影像：**电视节目和电影，Youtube 视频，手机短视频，家用监控，多摄像头监控等等。
- 结构化数据：**Jupyter notebook（里面有文本，图片和代码）、网页、电子病历、租车记录和电费账单。

模型（Models）

通常，我们拿到的数据和最终想要的结果相差甚远。例如，想知道照片中的人是不是开心，我们希望有一个模型，能将成千上万的低级特征（像素值），转化为高度抽象的输出（开心程度）。选择正确模型并不简单，不同的模型适合不同的数据集。在这本书中，我们会主要聚焦于深度神经网络模

型。这些模型包含了自上而下联结的数据多层连续变换，因此称之为**深度学习（deep learning）**。在讨论深度神经网络之前，我们也会讨论一些简单、浅显的模型。

损失函数（Loss Functions）

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

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

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

优化算法（Optimization Algorithms）

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

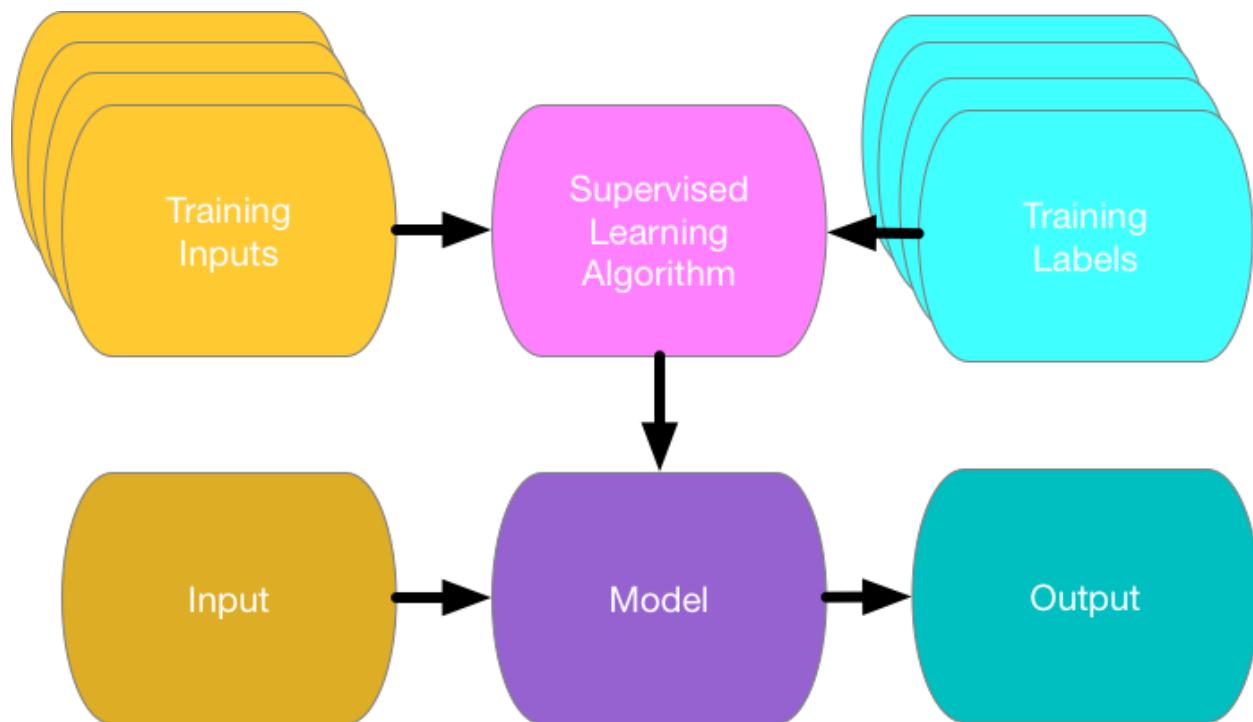
下面我们详细讨论一些不同的机器学习应用，以便理解我们具体会做什么。每个例子，我们都会介绍问题的解决方式，如训练方式、数据类型等等。这些内容仅仅是开胃小菜，并为大家提供一些共同的谈资。随着学习的推进，我们会介绍更多的类似问题。

2.1.5 监督学习（Supervised Learning）

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

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

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



回归分析（Regression）

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

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

如果住在纽约或三藩这种大城市，你也不是某个知名科技公司 CEO，那么这个特征向量（面积、卧室数量、卫生间数量、与市中心的距离）可能是这样的 [100, 0, .5, 60]。如果住在匹兹堡，则可能是

这样的 $[3000, 4, 3, 10]$ 。这些特征向量，是所有经典机器学习问题的关键。我们一般将一个特征向量标为 \mathbf{x}_i ，将所有特征向量的集合标为 X 。

一个问题是否应采用回归分析，取决于它的输出。比如你想预测一幢房子的合理的市场价格，并提供了类似的特征向量。它的目标市场价是一个**实数**。我们将单个目标（对应每一个特征向量代表的样例 \mathbf{x}_i 标为 y_i ，所有目标的集合为 \mathbf{y} （对应所有样例的集合 X ）。当我们的目标是某个范围内的任意实数值时，这就是一个回归分析问题。模型的目标就是输出预测（在这个例子中，即价格的预测），且尽可能近似实际的目标值。我们将这些预测标为 \hat{y}_i ；如果这堆符号看起来费解，不用担心，在接下来的章节中我们会详细讲解。

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

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

分类 (Classification)

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

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

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

随着本书在技术上的逐渐深入，我们会发现，训练一个能够输出绝对分类的模型，譬如“不是猫就是狗”，是很困难的。简单一点的做法是，我们使用概率描述它。例如，针对一个样例 x ，模型会输

出每一类标注 k 的一个概率 \hat{y}_k 。因为是概率，其值必然为正，且和为 1。换句话说，总共有 K 个类别的分类问题，我们只需要知道其中 $K - 1$ 个类别的概率，自然就能知道剩下那个类别的概率。这个办法针对二分类问题尤其管用，如果抛一枚不均匀的硬币有 0.6(60%) 的概率正面朝上，那么，反面朝上的概率就是 0.4(40%)。回到我们的喵汪分类，一个分类器看到了一张图片，输出图上的动物是猫的概率 $\Pr(y = \text{cat} | x) = 0.9$ 。对于这个结果，我们可以表述成，分类器 90% 确信图片描绘了一只猫。预测所得的类别的概率大小，仅仅是一个确信度的概念。我们会在后面的章节中进一步讨论有关确信度的问题。

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

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



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

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

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

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

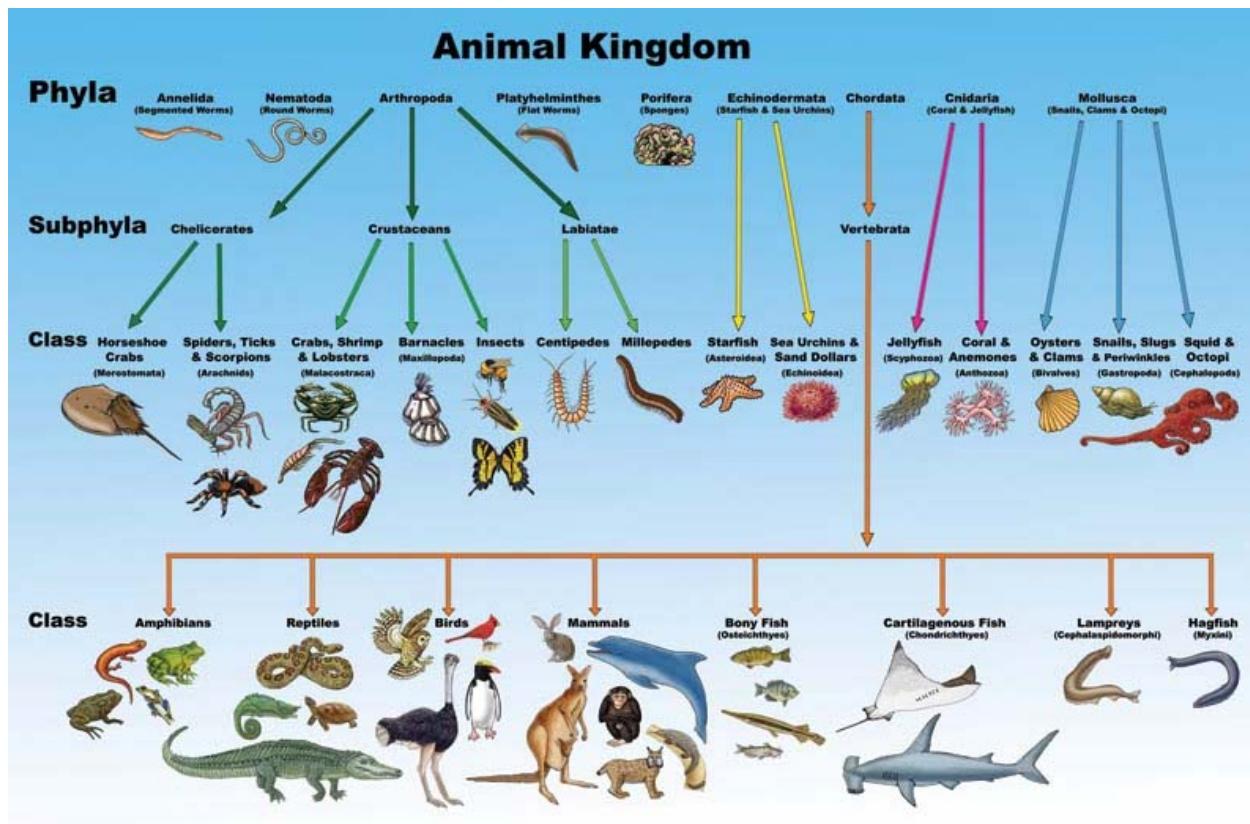


Fig. 2.1: 动物的分类

但是把一条有毒的响尾蛇认成无毒的束带蛇可是致命的。

标注 (Tagging)

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



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

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

生物医学领域也有这个问题，正确地标注研究文献能让研究人员对文献进行彻底的审阅。在美国国

家医学图书馆，一些专业的文章标注者会浏览 PubMed 中索引的每篇文章，并与 MeSH，一个大约 28k 个标签的集合中的术语相关联。这是一个漫长的过程，通常在文章归档一年后才会轮到标注。在手动审阅和标注之前，机器学习可以用来提供临时标签。事实上，近几年，BioASQ 组织已经举办过相关比赛。

搜索与排序 (Search and Ranking)

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

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

推荐系统 (Recommender Systems)

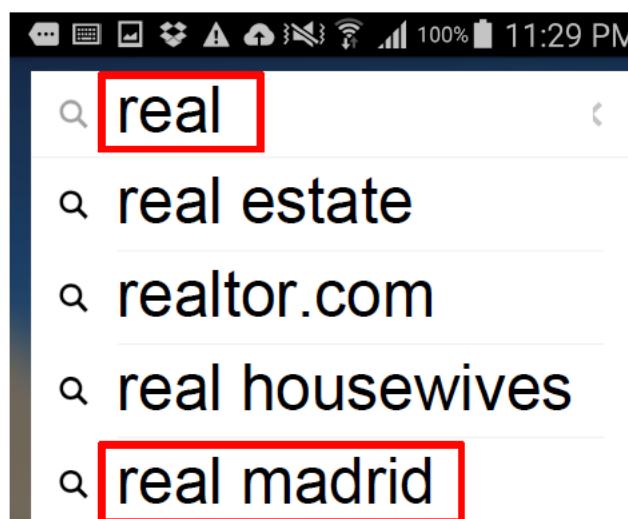
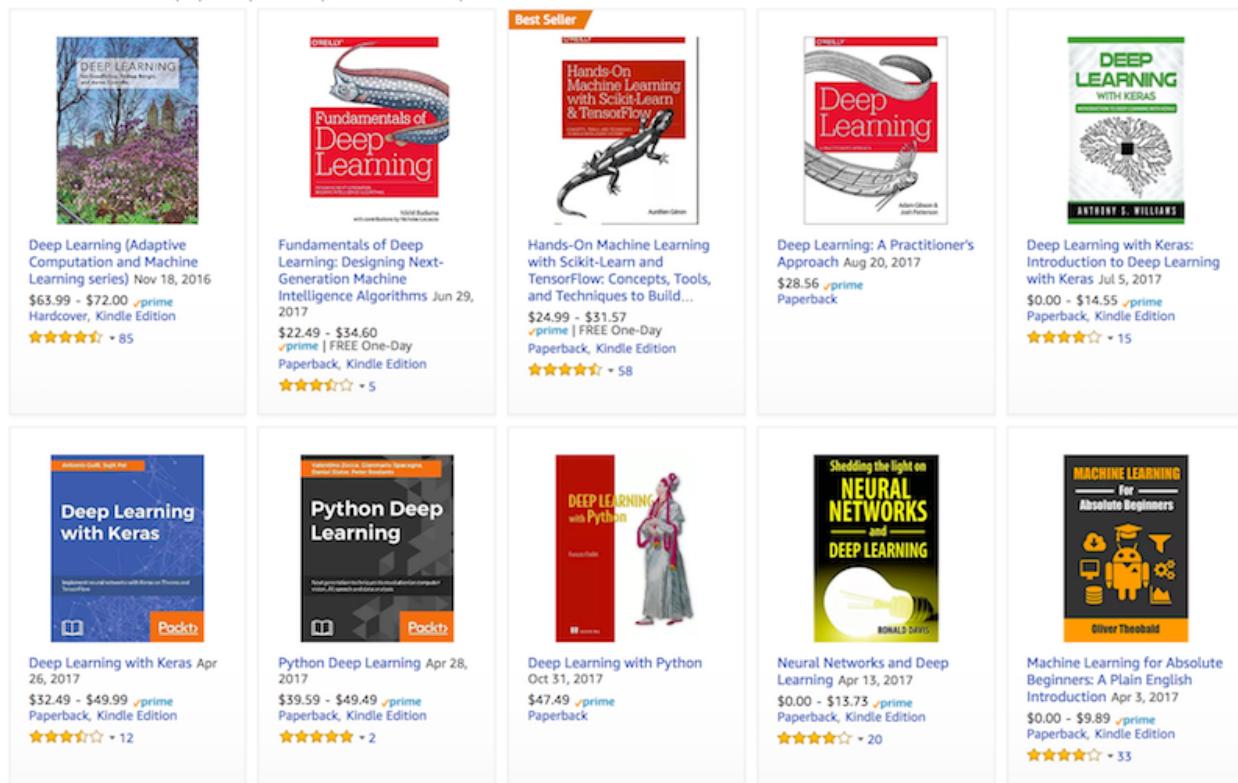
推荐系统与搜索排序关系紧密，其目的都是向用户展示一组相关条目。主要区别在于，推荐系统特别强调每个用户的个性化需求。例如电影推荐，科幻电影粉和伍迪·艾伦喜剧爱好者的页面可能天差地别。

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

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

序列学习 (Sequence Learning)

目前为止，我们提及的问题，其输入输出都有固定的长度，且连续输入之间不存在相互影响。如果我们的输入是一个视频片段呢？每个片段可能由不同数量的帧组成。且对每一帧的内容，如果我们



一并考虑之前或之后的帧，猜测会更加准确。语言也是如此。热门的深度学习问题就包含机器翻译：在源语言中摄取句子，并预测另一种语言的翻译。

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

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

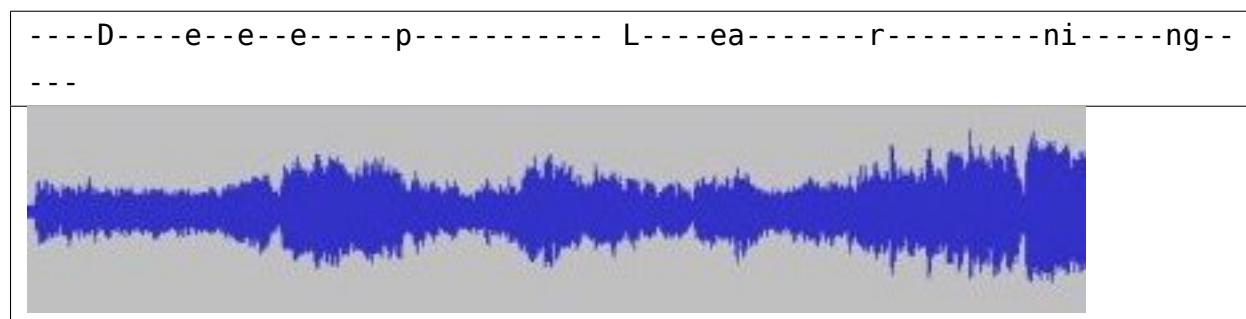
词类标注和句法分析 (Tagging and Parsing)

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

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

语音识别 (Automatic Speech Recognition)

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



文本转语音 (Text to Speech)

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

机器翻译 (Machine Translation)

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

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

2.1.6 无监督学习 (Unsupervised Learning)

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

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

- 我们能用少量的原型，精准地概括数据吗？给我们一堆照片，能把它们分成风景、狗、婴儿、猫、山峰的照片吗？类似的，给定一堆用户浏览记录，我们能把他们分成几类典型的用户吗？这类问题通常被称为**聚类 (clustering)**。
- 我们可以用少量的参数，准确地捕获数据的相关属性吗？球的轨迹可以很好地用速度，直径和质量准确描述。裁缝们也有一组参数，可以准确地描述客户的身材，以及适合的版型。这类问题被称为**子空间估计 (subspace estimation)** 问题。如果决定因素是线性关系的，则称为主成分分析 (**principal component analysis**)。
- 在欧几里得空间（例如， \mathbb{R}^n 中的向量空间）中是否存在一种符号属性，可以表示出（任意构

建的) 原始对象? 这被称为**表征学习 (representation learning)**。例如我们希望找到城市的向量表示, 从而可以进行这样的向量运算: 罗马 - 意大利 + 法国 = 巴黎。

- 针对我们观察到的大量数据, 是否存在一个根本性的描述? 例如, 如果我们有关于房价、污染、犯罪、地理位置、教育、工资等等的统计数据的, 我们能否基于已有的经验数据, 找出这些因素互相间的关联? 贝叶斯图模型可用于类似的问题。
- 一个最近很火的领域, **生成对抗网络 (generative adversarial networks)**。简单地说, 是一套生成数据的流程, 并检查真实数据与生成的数据是否在统计上相似。

2.1.7 与环境因素交互

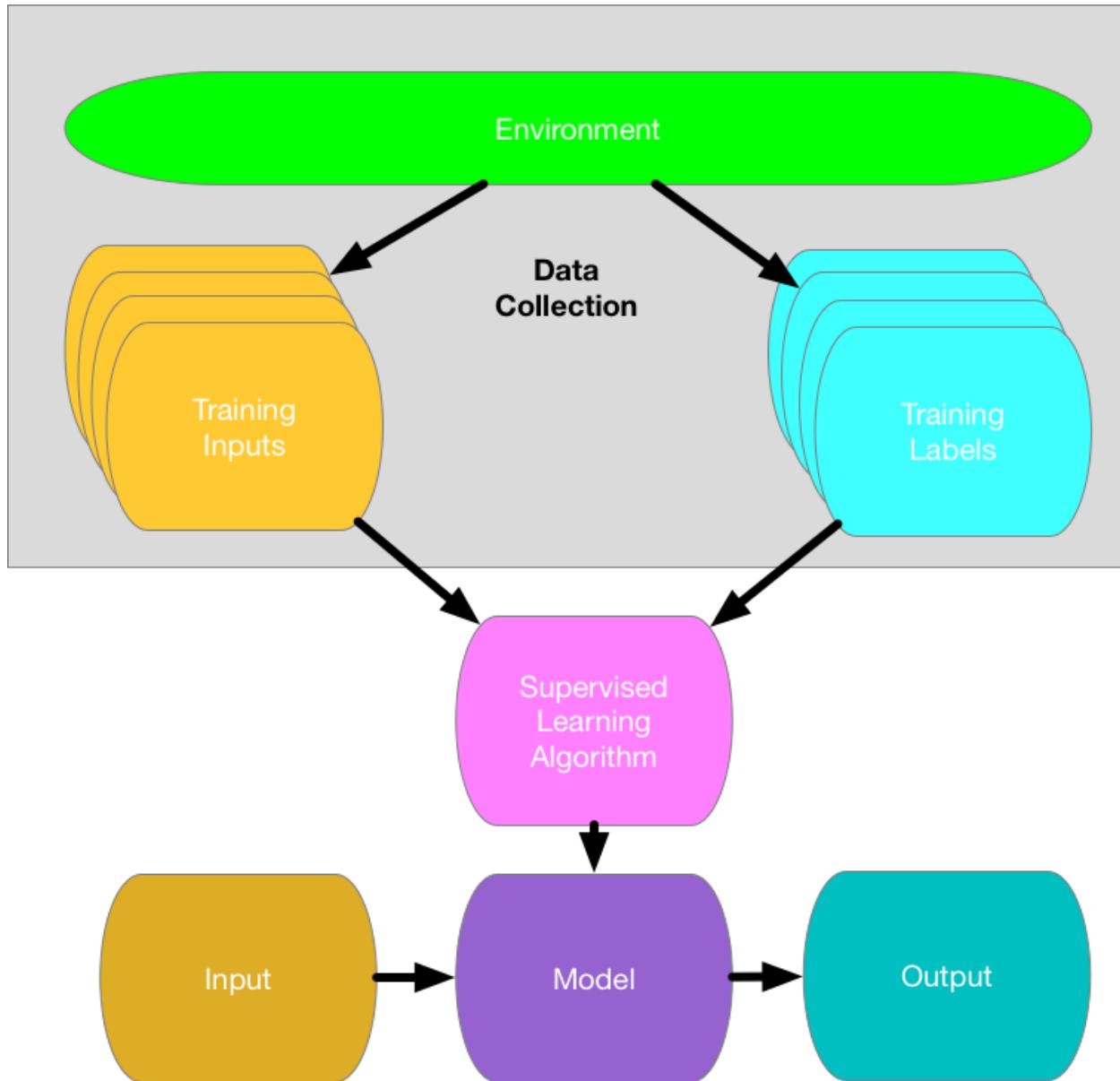
目前为止, 我们还没讨论过, 我们的数据实际上来自哪里, 以及当机器学习模型生成结果时, 究竟发生了什么。这是因为, 无论是监督学习还是无监督学习, 都不会着重于这点。我们在初期抓取大量数据, 然后在不再与环境发生交互的情况下进行模式识别。所有的学习过程, 都发生在算法和环境断开以后, 这称作**离线学习 (offline learning)**。对于监督学习, 其过程如下:

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

一旦考虑到要与周围环境交互, 一系列的问题接踵而来。我们需要考虑这个环境:

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

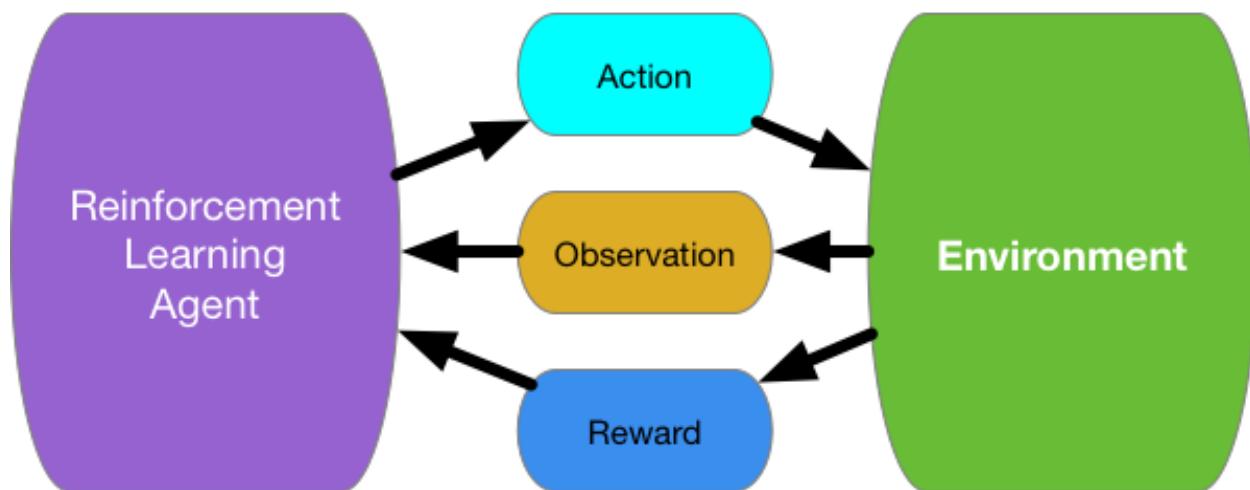
最后的这个问题, 引出了**协变量转移 (covariate shift)** 的问题 (当训练和测试数据不同时)。这个坑想必不少人都经历过, 平时的作业都是助教出题, 而到了考试, 题目却换成由课程老师编写。我们会简要介绍**强化学习 (reinforcement learning)** 和**对抗学习 (adversarial learning)**, 这是两个会明确考虑与环境交互的问题。



强化学习 (Reinforcement Learning)

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

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



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

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

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

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

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

马尔可夫决策过程，赌博机问题

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

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

2.1.8 小结

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

吐槽和讨论欢迎点[这里](#)

2.2 使用 NDArray 来处理数据

对于机器学习来说，处理数据往往是万事之开头。它包含两个部分：(i) 数据读取，(ii) 数据已经在内存中时如何处理。本章将关注后者。

我们首先介绍 `NDArray`，这是 MXNet 储存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。当然，`NDArray` 提供更多的功能，首先是 CPU 和 GPU 的异步计算，其次是自动求导。这两点使得 `NDArray` 能更好地支持机器学习。

2.2.1 让我们开始

我们先介绍最基本的功能。如果你不懂我们用到的数学操作也不用担心，例如按元素加法、正态分布；我们会在之后的章节分别从数学和代码编写的角度详细介绍。

我们首先从 `mxnet` 导入 `ndarray` 这个包

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

然后我们创建一个 3 行和 4 列的 2D 数组（通常也叫矩阵），并且把每个元素初始化成 0

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

Out[2]:

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

类似的，我们可以创建数组每个元素被初始化成 1。

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

Out[3]:

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

或者从 python 的数组直接构造

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

Out[4]:

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

我们经常需要创建随机数组，即每个元素的值都是随机采样而来，这个经常被用来初始化模型参数。以下代码创建数组，它的元素服从均值 0 标准差 1 的正态分布。

```
In [5]: y = nd.random_normal(0, 1, shape=(3, 4))
y
```

Out[5]:

```
[[ 0.30030754  0.23107235  1.04932892 -0.32433933]
 [-0.0097888   0.73686236  1.72023427  0.46656415]
```

```
[-1.07333767  0.87809837 -0.26717702 -0.8692565 ]  
<NDArray 3x4 @cpu(0)>
```

跟 NumPy 一样，每个数组的形状可以通过`.shape`来获取

```
In [6]: y.shape
```

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

它的大小，就是总元素个数，是形状的累乘。

```
In [7]: y.size
```

```
Out[7]: 12
```

2.2.2 操作符

NDArray 支持大量的数学操作符，例如按元素加法：

```
In [8]: x + y
```

```
Out[8]:
```

```
[[ 1.30030751  1.23107231  2.0493288   0.67566067]  
 [ 0.99021119  1.73686242  2.72023439  1.46656418]  
 [-0.07333767  1.87809837  0.73282301  0.1307435 ]]  
<NDArray 3x4 @cpu(0)>
```

乘法：

```
In [9]: x * y
```

```
Out[9]:
```

```
[[ 0.30030754  0.23107235  1.04932892 -0.32433933]  
 [-0.0097888   0.73686236  1.72023427  0.46656415]  
 [-1.07333767  0.87809837 -0.26717702 -0.8692565 ]]  
<NDArray 3x4 @cpu(0)>
```

指数运算：

```
In [10]: nd.exp(y)
```

```
Out[10]:
```

```
[[ 1.35027397  1.2599504   2.85573411  0.72300488]  
 [ 0.99025893  2.08936954  5.58583689  1.59450626]  
 [ 0.34186557  2.40631938  0.76553756  0.41926315]]  
<NDArray 3x4 @cpu(0)>
```

也可以转置一个矩阵然后计算矩阵乘法：

```
In [11]: nd.dot(x, y.T)
```

Out[11]:

```
[[ 1.25636935  2.913872   -1.33167279]
 [ 1.25636935  2.913872   -1.33167279]
 [ 1.25636935  2.913872   -1.33167279]]
<NDArray 3x3 @cpu(0)>
```

我们会在之后的线性代数一章讲解这些运算符。

2.2.3 广播 (Broadcasting)

当二元操作符左右两边 ndarray 形状不一样时，系统会尝试将其复制到一个共同的形状。例如 a 的第 0 维是 3, b 的第 0 维是 1, 那么 a+b 时会将 b 沿着第 0 维复制 3 遍：

```
In [12]: a = nd.arange(3).reshape((3,1))
b = nd.arange(2).reshape((1,2))
print('a:', a)
print('b:', b)
print('a+b:', a+b)
```

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

2.2.4 跟 NumPy 的转换

ndarray 可以很方便地同 numpy 进行转换

```
In [13]: import numpy as np
x = np.ones((2,3))
y = nd.array(x) # numpy -> mxnet
```

```

z = y.asnumpy() # mxnet -> numpy
print([z, y])

[array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]], dtype=float32),
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
<NDArray 2x3 @cpu(0)>

```

2.2.5 替换操作

在前面的样例中，我们为每个操作新开内存来存储它的结果。例如，如果我们写 $y = x + y$, 我们会把 y 从现在指向的实例转到新建的实例上去。我们可以用 Python 的 `id()` 函数来看这个是怎么执行的：

```

In [14]: x = nd.ones((3, 4))
y = nd.ones((3, 4))

before = id(y)
y = y + x
id(y) == before

Out[14]: False

```

我们可以把结果通过 `[:] =` 写到一个之前开好的数组里：

```

In [15]: z = nd.zeros_like(x)
before = id(z)
z[:] = x + y
id(z) == before

Out[15]: True

```

但是这里我们还是为 $x+y$ 创建了临时空间，然后再复制到 z 。需要避免这个开销，我们可以使用操作符的全名版本中的 `out` 参数：

```

In [16]: nd.elemwise_add(x, y, out=z)
id(z) == before

```

`Out[16]: True`

如果现有的数组不会复用，我们也可以用 $x[:] = x + y$ ，或者 $x += y$ 达到这个目的：

```

In [17]: before = id(x)
x += y
id(x) == before

```

Out[17]: True

2.2.6 截取 (Slicing)

NXNet NDArray 提供了各种截取方法。截取 x 的 index 为 1、2 的列:

```
In [18]: x = nd.arange(0,9).reshape((3,3))
print('x: ', x)
x[1:3]
```

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

Out[18]:

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

以及直接写入指定位置:

```
In [19]: x[1,2] = 9.0
x
```

Out[19]:

```
[[ 0.  1.  2.]
 [ 3.  4.  9.]
 [ 6.  7.  8.]]
<NDArray 3x3 @cpu(0)>
```

多维截取:

```
In [20]: x = nd.arange(0,9).reshape((3,3))
print('x: ', x)
x[1:2,1:3]
```

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

Out[20]:

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

```
<NDArray 1x2 @cpu(0)>
```

多维写入：

```
In [21]: x[1:2,1:3] = 9.0
```

```
x
```

```
Out[21]:
```

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

```
[ 3.  9.  9.]
```

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

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

2.2.7 总结

ndarray 模块提供一系列多维数组操作函数。所有函数列表可以参见NDArray API 文档。

吐槽和讨论欢迎点[这里](#)

2.3 使用 autograd 来自动求导

在机器学习中，我们通常使用**梯度下降（gradient descent）**来更新模型参数从而求解。损失函数关于模型参数的梯度指向一个可以降低损失函数值的方向，我们不断地沿着梯度的方向更新模型从而最小化损失函数。虽然梯度计算比较直观，但对于复杂的模型，例如多达数十层的神经网络，手动计算梯度非常困难。

为此 MXNet 提供 autograd 包来自动化求导过程。虽然大部分的深度学习框架要求编译计算图来自动求导，`mxnet.autograd` 可以对正常的命令式程序进行求导，它每次在后端实时创建计算图，从而可以立即得到梯度的计算方法。

下面让我们一步步介绍这个包。我们先导入 `autograd`。

```
In [1]: import mxnet.ndarray as nd
        import mxnet.autograd as ag
```

2.3.1 为变量附上梯度

假设我们想对函数 $f = 2 \times x^2$ 求关于 x 的导数。我们先创建变量 x ，并赋初值。

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

当进行求导的时候，我们需要一个地方来存 x 的导数，这个可以通过 NDArray 的方法 `attach_grad()` 来要求系统申请对应的空间。

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

下面定义 `f`。默认条件下, MXNet 不会自动记录和构建用于求导的计算图, 我们需要使用 `autograd` 里的 `record()` 函数来显式的要求 MXNet 记录我们需求导的程序。

In [4]: `with ag.record():`

```
y = x * 2
```

```
z = y * x
```

接下来我们可以通过 `z.backward()` 来进行求导。如果 `z` 不是一个标量, 那么 `z.backward()` 等价于 `nd.sum(z).backward()`。

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

现在我们来看求出来的导数是不是正确的。注意到 $y = x * 2$ 和 $z = x * y$, 所以 z 等价于 $2 * x * x$ 。它的导数那么就是 $\frac{dz}{dx} = 4 \times x$ 。

In [6]: `print('x.grad: ', x.grad)`

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

`x.grad:`

```
[[ 4.  8.]
```

```
[ 12. 16.]
```

```
<NDArray 2x2 @cpu(0)>
```

Out[6]:

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

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

```
<NDArray 2x2 @cpu(0)>
```

2.3.2 对控制流求导

命令式的编程的一个便利之处是几乎可以对任意的可导程序进行求导, 即使里面包含了 Python 的控制流。考虑下面程序, 里面包含控制流 `for` 和 `if`, 但循环迭代的次数和判断语句的执行都是取决于输入的值。不同的输入会导致这个程序的执行不一样。(对于计算图框架来说, 这个对应于动态图, 就是图的结构会根据输入数据不同而改变)。

In [7]: `def f(a):`

```
b = a * 2
```

```
while nd.norm(b).asscalar() < 1000:
```

```
    b = b * 2
```

```
if nd.sum(b).asscalar() > 0:
```

```
    c = b
```

```
else:
```

```
c = 100 * b
return c
```

我们可以跟之前一样使用 record 记录和 backward 求导。

```
In [8]: a = nd.random_normal(shape=3)
a.attach_grad()
with ag.record():
    c = f(a)
c.backward()
```

注意到给定输入 a, 其输出

$f(a) = xa$, x 的值取决于输入 a。所以有 $\frac{df}{da} = x$, 我们可以很简单地评估自动求导的导数:

```
In [9]: a.grad == c/a
```

Out[9]:

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

2.3.3 头梯度和链式法则

注意: 读者可以跳过这一小节, 不会影响阅读之后的章节

当我们在一个 NDArray 上调用 backward 方法时, 例如 $y.backward()$, 此处 y 是一个关于 x 的函数, 我们将求得 y 关于 x 的导数。数学家们会把这个求导写成 $\frac{dy(x)}{dx}$ 。还有些更复杂的情况, 比如 z 是关于 y 的函数, 且 y 是关于 x 的函数, 我们想对 z 关于 x 求导, 也就是求 $\frac{d}{dx}z(y(x))$ 的结果。回想一下链式法则, 我们可以得到 $\frac{d}{dx}z(y(x)) = \frac{dz(y)}{dy} \frac{dy(x)}{dx}$ 。当 y 是一个更大的 z 函数的一部分, 并且我们希望求得 $\frac{dz}{dy}$ 保存在 $x.grad$ 中时, 我们可以传入**头梯度 (head gradient)** $\frac{dz}{dy}$ 的值作为 backward() 方法的输入参数, 系统会自动应用链式法则进行计算。这个参数的默认值是 `nd.ones_like(y)`。关于链式法则的详细解释, 请参阅[Wikipedia](#)。

```
In [10]: with ag.record():
    y = x * 2
    z = y * x

    head_gradient = nd.array([[10, 1.], [.1, .01]])
    z.backward(head_gradient)
    print(x.grad)

[[ 40.          8.          ]
 [ 1.20000005   0.16        ]]
<NDArray 2x2 @cpu(0)>
```

吐槽和讨论欢迎点[这里](#)

监督学习

3.1 线性回归—从 0 开始

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，你就会很难深入理解深度学习是如何工作的。因此，我们的第一个教程是如何只利用 ndarray 和 autograd 来实现一个线性回归的训练。

3.1.1 线性回归

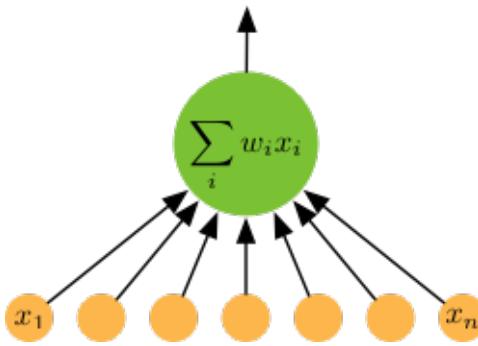
给定一个数据点集合 X 和对应的目标值 y ，线性模型的目标就是找到一条使用向量 w 和位移 b 描述的线，来尽可能地近似每个样本 $X[i]$ 和 $y[i]$ 。用数学符号来表示就是：

$$\hat{y} = Xw + b$$

并最小化所有数据点上的平方误差

$$\sum_{i=1}^n (\hat{y}_i - y_i)^2.$$

你可能会对我们把古老的线性回归作为深度学习的一个样例表示奇怪。实际上线性模型是最简单、但也是最有用的神经网络。一个神经网络就是一个由节点（神经元）和有向边组成的集合。我们一般把一些节点组成层，每一层先从下面一层的节点获取输入，然后输出给上面的层使用。要计算一个节点值，我们需要将输入节点值做加权和（权数值即 w ），然后再加上一个激活函数（activation function）。对于线性回归而言，它是一个两层神经网络，其中第一层是（下图橙色点）输入，每个节点对应输入数据点的一个维度，第二层是单输出节点（下图绿色点），它使用身份函数 ($f(x) = x$) 作为激活函数。



3.1.2 创建数据集

这里我们使用一个数据集来尽量简单地解释清楚，真实的模型是什么样的。具体来说，我们使用如下方法来生成数据；随机数值 $X[i]$ ，其相应的标注为 $y[i]$ ：

```
y[i] = 2 * X[i][0] - 3.4 * X[i][1] + 4.2 + noise
```

使用数学符号表示：

$$y = X \cdot w + b + \eta, \quad \text{for } \eta \sim \mathcal{N}(0, \sigma^2)$$

这里噪音服从均值 0 和标准差为 0.01 的正态分布。

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

        num_inputs = 2
        num_examples = 1000

        true_w = [2, -3.4]
        true_b = 4.2

        X = nd.random_normal(shape=(num_examples, num_inputs))
        y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
        y += .01 * nd.random_normal(shape=y.shape)
```

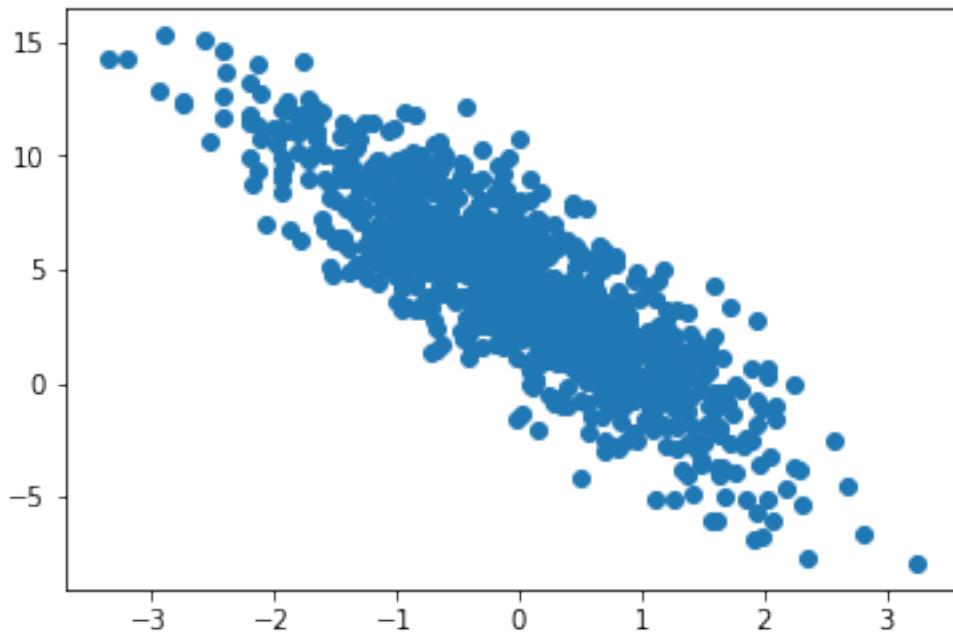
注意到 X 的每一行是一个长度为 2 的向量，而 y 的每一行是一个长度为 1 的向量（标量）。

```
In [2]: print(X[0], y[0])
```

```
[ 0.30030754  0.23107235]
<NDArray 2 @cpu(0)>
[ 4.00086403]
<NDArray 1 @cpu(0)>
```

如果有兴趣, 可以使用安装包中已包括的 Python 绘图包 `matplotlib`, 生成第二个特征值 ($X[:, 1]$) 和目标值 Y 的散点图, 更直观地观察两者间的关系。

```
In [3]: import matplotlib.pyplot as plt
plt.scatter(X[:, 1].asnumpy(), y.asnumpy())
plt.show()
```



3.1.3 数据读取

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

```
In [4]: import random
batch_size = 10
def data_iter():
    # 产生一个随机索引
    idx = list(range(num_examples))
    random.shuffle(idx)
    for i in range(0, num_examples, batch_size):
        j = nd.array(idx[i:min(i+batch_size, num_examples)])
        yield nd.take(X, j), nd.take(y, j)
```

下面代码读取第一个随机数据块

```
In [5]: for data, label in data_iter():
    print(data, label)
```

break

```
[[ -0.47685868  0.3858363 ]
 [-2.28530121 -0.62275136]
 [-0.52573997 -0.24910249]
 [ 0.84342134  1.76080275]
 [-0.45648727  0.58826393]
 [ 0.41244769 -0.55140877]
 [ 0.30690399 -0.18806908]
 [ 0.42437807  0.7583164 ]
 [ 1.98813105 -0.69251704]
 [ 1.00766551 -0.70099592]]
<NDArray 10x2 @cpu(0)>
[ 1.92649412   1.73920381   4.00396824   -0.08797584   1.28135252
 6.89146996   5.44915485   2.45705581   10.53651333   8.60093975]
<NDArray 10 @cpu(0)>
```

3.1.4 初始化模型参数

下面我们随机初始化模型参数

```
In [6]: w = nd.random_normal(shape=(num_inputs, 1))
        b = nd.zeros((1,))
        params = [w, b]
```

之后训练时我们需要对这些参数求导来更新它们的值，使损失尽量减小；因此我们需要创建它们的梯度。

```
In [7]: for param in params:
            param.attach_grad()
```

3.1.5 定义模型

线性模型就是将输入和模型的权重（ w ）相乘，再加上偏移（ b ）：

```
In [8]: def net(X):
        return nd.dot(X, w) + b
```

3.1.6 损失函数

我们使用常见的平方误差来衡量预测目标和真实目标之间的差距。

```
In [9]: def square_loss(yhat, y):
    # 注意这里我们把 y 变形成 yhat 的形状来避免矩阵形状的自动转换
    return (yhat - y.reshape(yhat.shape)) ** 2
```

3.1.7 优化

虽然线性回归有显式解，但绝大部分模型并没有。所以我们这里通过随机梯度下降来求解。每一步，我们将模型参数沿着梯度的反方向走特定距离，这个距离一般叫**学习率(learning rate)** lr。(我们会之后一直使用这个函数，我们将其保存在utils.py。)

```
In [10]: def SGD(params, lr):
    for param in params:
        param[:] = param - lr * param.grad
```

3.1.8 训练

现在我们可以开始训练了。训练通常需要迭代数据数次，在这里使用 epochs 表示迭代总次数；一次迭代中，我们每次随机读取固定数个数据点，计算梯度并更新模型参数。

```
In [11]: # 模型函数
def real_fn(X):
    return 2 * X[:, 0] - 3.4 * X[:, 1] + 4.2
# 绘制损失随训练次数降低的折线图，以及预测值和真实值的散点图
def plot(losses, X, sample_size=100):
    xs = list(range(len(losses)))
    f, (fg1, fg2) = plt.subplots(1, 2)
    fg1.set_title('Loss during training')
    fg1.plot(xs, losses, '-r')
    fg2.set_title('Estimated vs real function')
    fg2.plot(X[:sample_size, 1].asnumpy(),
              net(X[:sample_size, :]).asnumpy(), 'or', label='Estimated')
    fg2.plot(X[:sample_size, 1].asnumpy(),
              real_fn(X[:sample_size, :]).asnumpy(), '*g', label='Real')
    fg2.legend()
    plt.show()

In [12]: epochs = 5
learning_rate = .001
niter = 0
losses = []
moving_loss = 0
smoothing_constant = .01
```

```
# 训练
for e in range(epochs):
    total_loss = 0

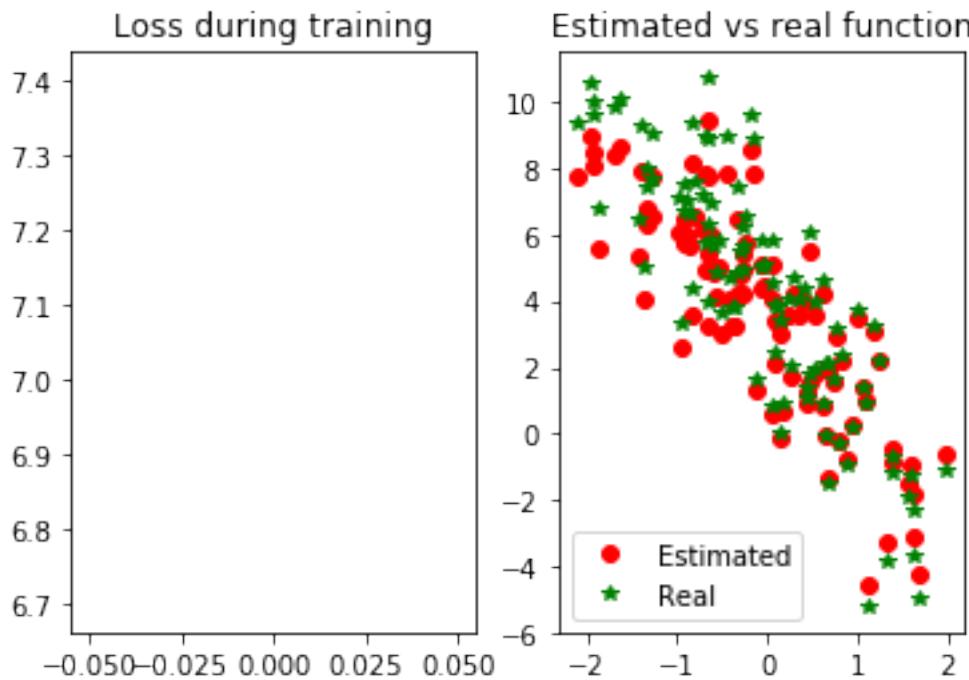
    for data, label in data_iter():
        with autograd.record():
            output = net(data)
            loss = square_loss(output, label)
        loss.backward()
        SGD(params, learning_rate)
        total_loss += nd.sum(loss).asscalar()

    # 记录每读取一个数据点后，损失的移动平均值的变化;
    niter +=1
    curr_loss = nd.mean(loss).asscalar()
    moving_loss = (1 - smoothing_constant) * moving_loss + (smoothing_constant * curr_loss)

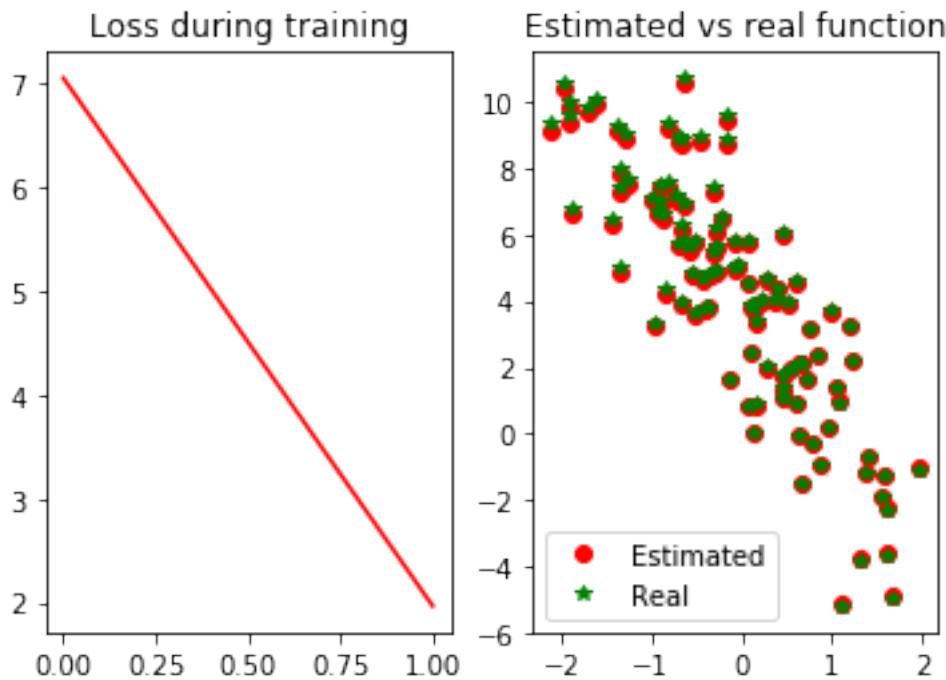
    # correct the bias from the moving averages
    est_loss = moving_loss/(1-(1-smoothing_constant)**niter)

    if (niter + 1) % 100 == 0:
        losses.append(est_loss)
        print("Epoch %s, batch %s. Moving avg of loss: %s. Average loss: %f"
              % (e+1, niter, moving_loss, est_loss))
        plot(losses, X)

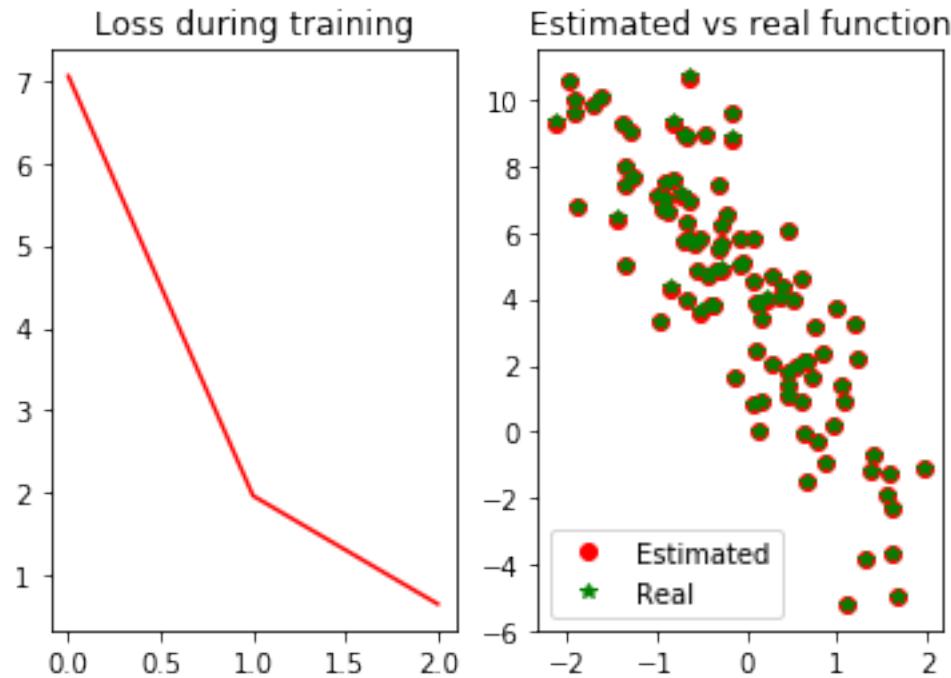
Epoch 0, batch 99. Moving avg of loss: 7.05072938249. Average loss: 9.254515
```



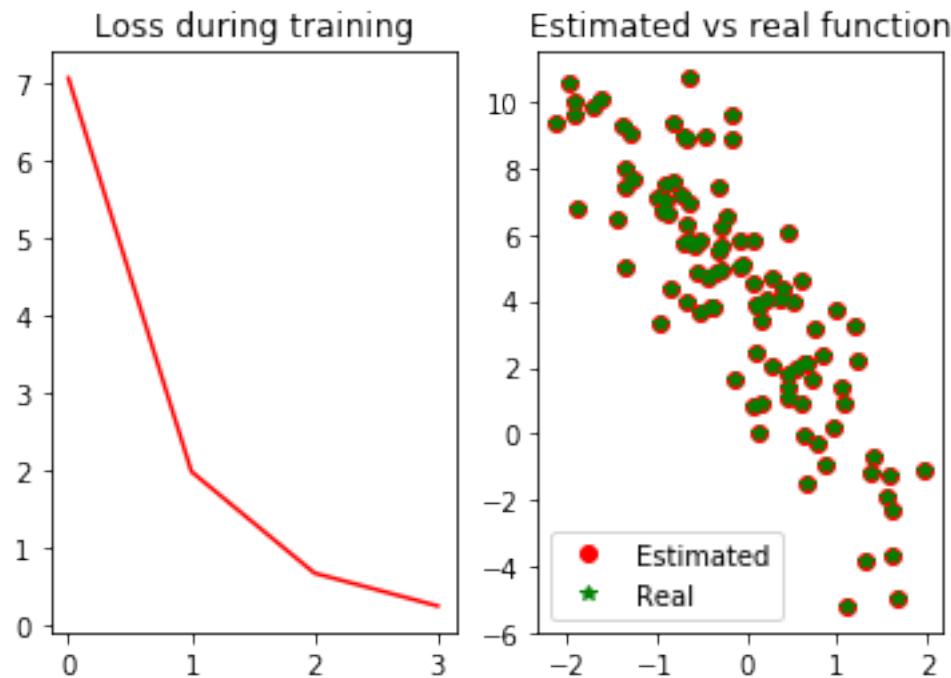
Epoch 1, batch 199. Moving avg of loss: 1.96730398748. Average loss: 0.149534



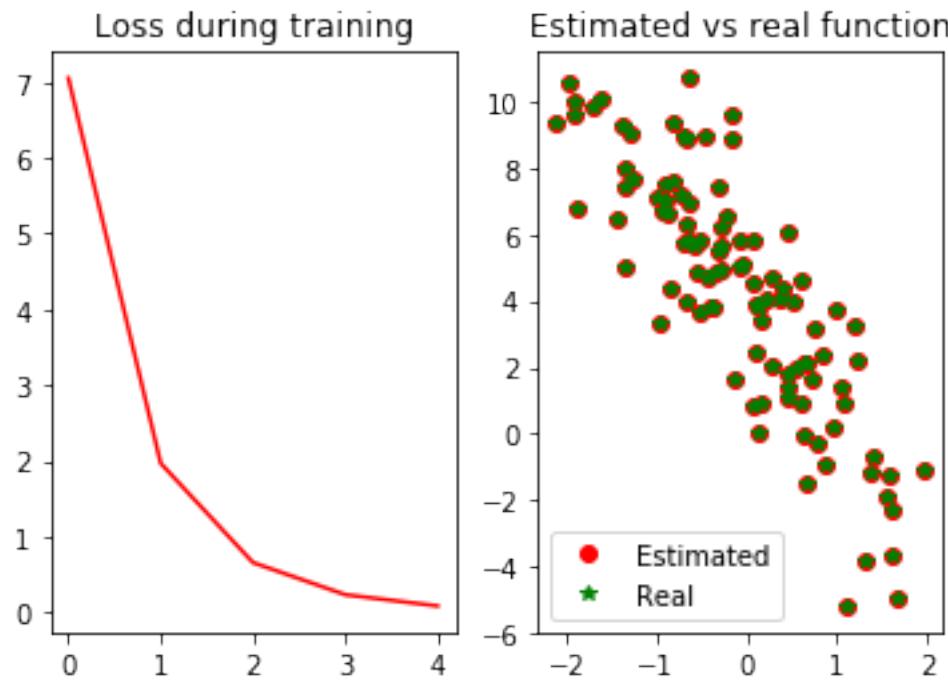
Epoch 2, batch 299. Moving avg of loss: 0.656402246117. Average loss: 0.002513



Epoch 3, batch 399. Moving avg of loss: 0.232660707597. Average loss: 0.000131



Epoch 4, batch 499. Moving avg of loss: 0.0842344091236. Average loss: 0.000092



训练完成后，我们可以比较学得的参数和真实参数

In [13]: `true_w, w`

Out[13]: ([2, -3.4],
[[2.00010538]
[-3.40014243]]
<NDArray 2x1 @cpu(0)>)

In [14]: `true_b, b`

Out[14]: (4.2,
[4.19982958]
<NDArray 1 @cpu(0)>)

3.1.9 结论

我们现在看到，仅仅是使用 NDArray 和 autograd 就可以很容易实现的一个模型。在接下来的教程里，我们会在此基础上，介绍更多现代神经网络的知识，以及怎样使用少量的 MXNet 代码实现各种复杂的模型。

3.1.10 练习

尝试用不同的学习率查看误差下降速度（收敛率）

吐槽和讨论欢迎点[这里](#)

3.2 线性回归—使用 Gluon

前一章我们仅仅使用了 `ndarray` 和 `autograd` 来实现线性回归，这一章我们仍然实现同样的模型，但是使用高层抽象包 `gluon`。

3.2.1 创建数据集

我们生成同样的数据集

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

        num_inputs = 2
        num_examples = 1000

        true_w = [2, -3.4]
        true_b = 4.2

        X = nd.random_normal(shape=(num_examples, num_inputs))
        y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
        y += .01 * nd.random_normal(shape=y.shape)
```

3.2.2 数据读取

但这里使用 `data` 模块来读取数据。

```
In [2]: batch_size = 10
        dataset = gluon.data.ArrayDataset(X, y)
        data_iter = gluon.data.DataLoader(dataset, batch_size, shuffle=True)
```

读取跟前面一致：

```
In [3]: for data, label in data_iter:
            print(data, label)
            break
```

```
[[ -0.88528353 -0.03406117]
 [ 0.74839669 -0.56714851]]
```

```
[ 1.21428883 -1.43047035]
[-1.47901928 -1.37985945]
[ 0.23348512 -0.28417772]
[ 0.23742883  0.28244382]
[ 0.94593477  0.81830043]
[ 1.60465384  0.67369854]
[-0.01446326  0.80464268]
[ 1.21521401 -0.66192865]]
<NDArray 10x2 @cpu(0)>
[ 2.54761314    7.62509298   11.4933157    5.92006826   5.63085032
 3.71501708    3.31964731   5.10721684   1.41811109   8.87456894]
<NDArray 10 @cpu(0)>
```

3.2.3 定义模型

之前一章中，当我们从 0 开始训练模型时，需要先声明模型参数，然后再使用它们来构建模型。但 gluon 提供大量预定义的层，我们只需要关注使用哪些层来构建模型。例如线性模型就是使用对应的 `Dense` 层；之所以称为 `dense` 层，是因为输入的所有节点都与后续的节点相连。在这个例子中仅有一个输出，但在大多数后续章节中，我们会用到具有多个输出的网络。

我们之后还会介绍如何构造任意结构的神经网络，但对于初学者来说，构建模型最简单的办法是利用 `Sequential` 来所有层串起来。输入数据之后，`Sequential` 会依次执行每一层，并将前一层的输出，作为输入提供给后面的层。首先我们定义一个空的模型：

```
In [4]: net = gluon.nn.Sequential()
```

然后我们加入一个 `Dense` 层，它唯一必须定义的参数就是输出节点的个数，在线性模型里面是 1。

```
In [5]: net.add(gluon.nn.Dense(1))
```

(注意这里我们并没有定义说这个层的输入节点是多少，这个在之后真正给数据的时候系统会自动赋值。我们之后会详细介绍这个特性是如何工作的。)

3.2.4 初始化模型参数

在使用前 `net` 我们必须要初始化模型权重，这里我们使用默认随机初始化方法（之后我们会介绍更多的初始化方法）。

```
In [6]: net.initialize()
```

3.2.5 损失函数

gluon 提供了平方误差函数:

```
In [7]: square_loss = gluon.loss.L2Loss()
```

3.2.6 优化

同样我们无需手动实现随机梯度下降，我们可以用创建一个 `Trainer` 的实例，并且将模型参数传递给它就行。

```
In [8]: trainer = gluon.Trainer(  
    net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

3.2.7 训练

使用 `gluon` 使模型训练过程更为简洁。我们不需要挨个定义相关参数、损失函数，也不需使用随机梯度下降。`gluon` 的抽象和便利的优势将随着我们着手处理更多复杂模型的愈发显现。不过在完成初始设置后，训练过程本身和前面没有太多区别，唯一的不同在于我们不再是调用 `SGD`，而是 `trainer.step` 来更新模型（此处一并省略之前绘制损失变化的折线图和散点图的过程，有兴趣的同学可以自行尝试）。

```
In [9]: epochs = 5  
batch_size = 10  
for e in range(epochs):  
    total_loss = 0  
    for data, label in data_iter:  
        with autograd.record():  
            output = net(data)  
            loss = square_loss(output, label)  
            loss.backward()  
            trainer.step(batch_size)  
            total_loss += nd.sum(loss).asscalar()  
    print("Epoch %d, average loss: %f" % (e, total_loss/num_examples))
```

Epoch 0, average loss: 0.885957

Epoch 1, average loss: 0.000046

Epoch 2, average loss: 0.000047

Epoch 3, average loss: 0.000047

Epoch 4, average loss: 0.000047

比较学到的和真实模型。我们先从 `net` 拿到需要的层，然后访问其权重和位移。

```
In [10]: dense = net[0]
         true_w, dense.weight.data()

Out[10]: ([2, -3.4],
           [[ 2.00063133 -3.4002707 ]])
           <NDArray 1x2 @cpu(0)>

In [11]: true_b, dense.bias.data()

Out[11]: (4.2,
           [ 4.20080233]
           <NDArray 1 @cpu(0)>)
```

3.2.8 结论

可以看到 gluon 可以帮助我们更快更干净地实现模型。

3.2.9 练习

- 在训练的时候，为什么我们用了比前面要大 10 倍的学习率呢？（提示：可以尝试运行 `help(trainer.step)` 来寻找答案。）
- 如何拿到 `weight` 的梯度呢？（提示：尝试 `help(dense.weight)`）

吐槽和讨论欢迎点[这里](#)

3.3 多类逻辑回归—从 0 开始

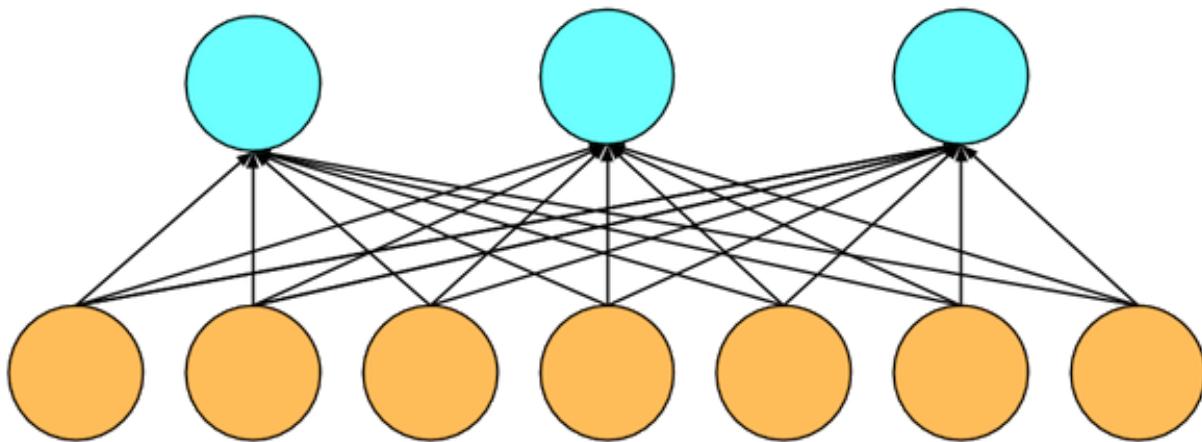
如果你读过了[从 0 开始的线性回归](#)，那么最难的部分已经过去了。现在你知道如果读取和操作数据，如何构造目标函数和对它求导，如果定义损失函数，模型和求解。

下面我们来看一个稍微有意思一点的问题，如何使用多类逻辑回归进行多类分类。这个模型跟线性回归的主要区别在于输出节点从一个变成了多个。

3.3.1 获取数据

演示这个模型的常见数据集是手写数字识别 MNIST，它长这个样子。

这里我们用了一个稍微复杂点的数据集，它跟 MNIST 非常像，但是内容不再是分类数字，而是服饰。我们通过 gluon 的 `data.vision` 模块自动下载这个数据。



5 0 4 1 9 2 1 3 1 4

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

        def transform(data, label):
            return data.astype('float32')/255, label.astype('float32')
        mnist_train = gluon.data.vision.FashionMNIST(train=True, transform=transform)
        mnist_test = gluon.data.vision.FashionMNIST(train=False, transform=transform)
```

打印一个样本的形状和它的标号

```
In [2]: data, label = mnist_train[0]
        ('example shape: ', data.shape, 'label:', label)
```

```
Out[2]: ('example shape: ', (28, 28, 1), 'label:', 2.0)
```

我们画出前几个样本的内容，和对应的文本标号

```
In [3]: import matplotlib.pyplot as plt

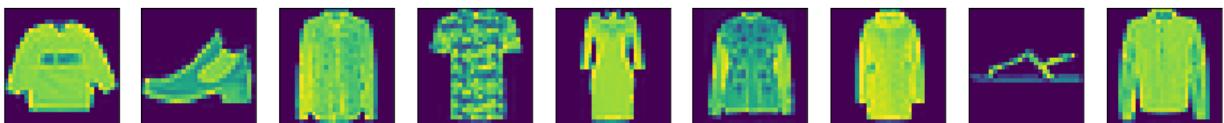
        def show_images(images):
            n = images.shape[0]
            _, figs = plt.subplots(1, n, figsize=(15, 15))
            for i in range(n):
                figs[i].imshow(images[i].reshape((28, 28)).asnumpy())
                figs[i].axes.get_xaxis().set_visible(False)
                figs[i].axes.get_yaxis().set_visible(False)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

```

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

data, label = mnist_train[0:9]
show_images(data)
print(get_text_labels(label))

```



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

3.3.2 数据读取

虽然我们可以像前面那样通过 `yield` 来定义获取批量数据函数，这里我们直接使用 `gluon.data` 的 `DataLoader` 函数，它每次 `yield` 一个批量。

```
In [4]: batch_size = 256
train_data = gluon.data.DataLoader(mnist_train, batch_size, shuffle=True)
test_data = gluon.data.DataLoader(mnist_test, batch_size, shuffle=False)
```

注意到这里我们要求每次从训练数据里读取一个由随机样本组成的批量，但测试数据则不需要这个要求。

3.3.3 初始化模型参数

跟线性模型一样，每个样本会表示成一个向量。我们这里数据是 $28 * 28$ 大小的图片，所以输入向量的长度是 $28 * 28 = 784$ 。因为我们要做多类分类，我们需要对每一个类预测这个样本属于此类的概率。因为这个数据集有 10 个类型，所以输出应该是长为 10 的向量。这样，我们需要的权重将是一个 $784 * 10$ 的矩阵：

```

In [5]: num_inputs = 784
num_outputs = 10

W = nd.random_normal(shape=(num_inputs, num_outputs))
b = nd.random_normal(shape=num_outputs)
```

```
params = [W, b]
```

同之前一样，我们要对模型参数附上梯度：

```
In [6]: for param in params:  
    param.attach_grad()
```

3.3.4 定义模型

在线性回归教程里，我们只需要输出一个标量 $y\hat{}$ 使得尽可能的靠近目标值。但在这些分类里，我们需要属于每个类别的概率。这些概率需要值为正，而且加起来等于 1. 而如果简单的使用 $\hat{y} = Wx$, 我们不能保证这一点。一个通常的做法是通过 softmax 函数来将任意的输入归一化成合法的概率值。

```
In [7]: from mxnet import nd  
def softmax(X):  
    exp = nd.exp(X)  
    # 假设 exp 是矩阵，这里对行进行求和，并要求保留 axis 1,  
    # 就是返回 (nrows, 1) 形状的矩阵  
    partition = exp.sum(axis=1, keepdims=True)  
    return exp / partition
```

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 [8]: X = nd.random_normal(shape=(2,5))  
X_prob = softmax(X)  
print(X_prob)  
print(X_prob.sum(axis=1))  
  
[[ 0.21869159  0.23117994  0.18298376  0.1587515   0.20839317]  
 [ 0.39214477  0.1804826   0.02733301  0.08681861  0.31322101]]  
<NDArray 2x5 @cpu(0)>  
  
[ 0.99999994  1.]  
<NDArray 2 @cpu(0)>
```

现在我们可以定义模型了：

```
In [9]: def net(X):  
    return softmax(nd.dot(X.reshape((-1,num_inputs)), W) + b)
```

3.3.5 交叉熵损失函数

我们需要定义一个针对预测为概率值的损失函数。其中最常见的是交叉熵损失函数，它将两个概率分布的负交叉熵作为目标值，最小化这个值等价于最大化这两个概率的相似度。

具体来说，我们先将真实标号表示成一个概率分布，例如如果 $y=1$ ，那么其对应的分布就是一个除了第二个元素为 1 其他全为 0 的长为 10 的向量，也就是 $yvec=[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$ 。那么交叉熵就是 $yvec[0]*\log(yhat[0])+\dots+yvec[n]*\log(yhat[n])$ 。注意到 $yvec$ 里面只有一个 1，那么前面等价于 $\log(yhat[y])$ 。所以我们可以定义这个损失函数了

```
In [10]: def cross_entropy(yhat, y):
    return -nd.pick(nd.log(yhat), y)
```

3.3.6 计算精度

给定一个概率输出，我们将预测概率最高的那个类作为预测的类，然后通过比较真实标号我们可以计算精度：

```
In [11]: def accuracy(output, label):
    return nd.mean(output.argmax(axis=1)==label).asscalar()
```

我们可以评估一个模型在这个数据上的精度。（这两个函数我们之后也会用到，所以也都保存在.. /utils.py。）

```
In [12]: def evaluate_accuracy(data_iterator, net):
    acc = 0.
    for data, label in data_iterator:
        output = net(data)
        acc += accuracy(output, label)
    return acc / len(data_iterator)
```

因为我们随机初始化了模型，所以这个模型的精度应该大概是 $1/\text{num_outputs} = 0.1$ 。

```
In [13]: evaluate_accuracy(test_data, net)
```

```
Out[13]: 0.10048828125000001
```

3.3.7 训练

训练代码跟前面的线性回归非常相似：

```
In [14]: import sys
sys.path.append('..')
from utils import SGD
```

```
from mxnet import autograd

learning_rate = .1

for epoch in range(5):
    train_loss = 0.
    train_acc = 0.
    for data, label in train_data:
        with autograd.record():
            output = net(data)
            loss = cross_entropy(output, label)
        loss.backward()
        # 将梯度做平均, 这样学习率会对 batch size 不那么敏感
        SGD(params, learning_rate/batch_size)

        train_loss += nd.mean(loss).asscalar()
        train_acc += accuracy(output, label)

    test_acc = evaluate_accuracy(test_data, net)
    print("Epoch %d. Loss: %f, Train acc %f, Test acc %f" % (
        epoch, train_loss/len(train_data), train_acc/len(train_data), test_acc))

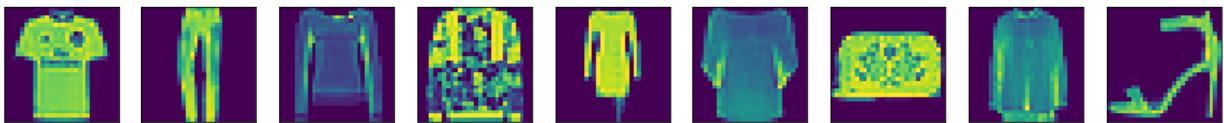
Epoch 0. Loss: 3.698817, Train acc 0.439495, Test acc 0.569824
Epoch 1. Loss: 1.897430, Train acc 0.618046, Test acc 0.643848
Epoch 2. Loss: 1.575920, Train acc 0.668041, Test acc 0.685937
Epoch 3. Loss: 1.402299, Train acc 0.695506, Test acc 0.702832
Epoch 4. Loss: 1.290862, Train acc 0.713309, Test acc 0.718848
```

3.3.8 预测

训练完成后, 现在我们可以演示对输入图片的标号的预测

```
In [15]: data, label = mnist_test[0:9]
show_images(data)
print('true labels')
print(get_text_labels(label))

predicted_labels = net(data).argmax(axis=1)
print('predicted labels')
print(get_text_labels(predicted_labels.asnumpy()))
```



true labels

['t-shirt', 'trouser', 'pullover', 'pullover', 'dress', 'pullover', 'bag', 'shirt', 'sand']

predicted labels

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

3.3.9 结论

与前面的线性回归相比，你会发现多类逻辑回归教程的结构跟其非常相似：获取数据、定义模型及优化算法和求解。事实上，几乎所有的实际神经网络应用都有着同样结构。他们的主要区别在于模型的类型和数据的规模。每一两年会有一个新的优化算法出来，但它们基本都是随机梯度下降的变种。

3.3.10 练习

尝试增大学习率，你会马上发现结果变得很糟糕，精度基本徘徊在随机的 0.1 左右。这是为什么呢？
提示：

- 打印下 output 看看是不是有什么异常
- 前面线性回归还好好的，这里我们在 net() 里加了什么呢？
- 如果给 exp 输入个很大的数会怎么样？
- 即使解决 exp 的问题，求出来的导数是不是还是不稳定？

请仔细想想再去对比下我们小伙伴之一 @pluskid 早年写的一篇 blog 解释这个问题，看看你想的是不是不一样。

吐槽和讨论欢迎点[这里](#)

3.4 多类逻辑回归—使用 Gluon

现在让我们使用 gluon 来更快速地实现一个多类逻辑回归。

3.4.1 获取和读取数据

我们仍然使用 FashionMNIST。我们将代码保存在..`utils.py`这样这里不用复制一遍。

```
In [1]: import sys  
        sys.path.append('..')  
        import utils  
  
        batch_size = 256  
        train_data, test_data = utils.load_data_fashion_mnist(batch_size)
```

3.4.2 定义和初始化模型

我们先使用 Flatten 层将输入数据转成 `batch_size` x ? 的矩阵，然后输入到 10 个输出节点的全连接层。照例我们不需要制定每层输入的大小，gluon 会做自动推导。

```
In [2]: from mxnet import gluon  
  
        net = gluon.nn.Sequential()  
        with net.name_scope():  
            net.add(gluon.nn.Flatten())  
            net.add(gluon.nn.Dense(10))  
        net.initialize()
```

3.4.3 Softmax 和交叉熵损失函数

如果你做了上一章的练习，那么你可能意识到了分开定义 Softmax 和交叉熵会有数值不稳定性。因此 gluon 提供一个将这两个函数合起来的数值更稳定的版本

```
In [3]: softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

3.4.4 优化

```
In [4]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

3.4.5 训练

```
In [5]: from mxnet import ndarray as nd  
        from mxnet import autograd  
  
        for epoch in range(5):
```

```

train_loss = 0.
train_acc = 0.
for data, label in train_data:
    with autograd.record():
        output = net(data)
        loss = softmax_cross_entropy(output, label)
    loss.backward()
    trainer.step(batch_size)

    train_loss += nd.mean(loss).asscalar()
    train_acc += utils.accuracy(output, label)

test_acc = utils.evaluate_accuracy(test_data, net)
print("Epoch %d. Loss: %f, Train acc %f, Test acc %f" % (
    epoch, train_loss/len(train_data), train_acc/len(train_data), test_acc))

Epoch 0. Loss: 0.791286, Train acc 0.745142, Test acc 0.801783
Epoch 1. Loss: 0.577035, Train acc 0.809044, Test acc 0.818710
Epoch 2. Loss: 0.530328, Train acc 0.823634, Test acc 0.831631
Epoch 3. Loss: 0.505648, Train acc 0.830479, Test acc 0.836138
Epoch 4. Loss: 0.490264, Train acc 0.834585, Test acc 0.841446

```

3.4.6 结论

Gluon 提供的函数有时候比手工写的数值更稳定。

3.4.7 练习

- 再尝试调大下学习率看看?
- 为什么参数都差不多, 但 gluon 版本比从 0 开始的版本精度更高?

吐槽和讨论欢迎点[这里](#)

3.5 多层感知机—从 0 开始

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

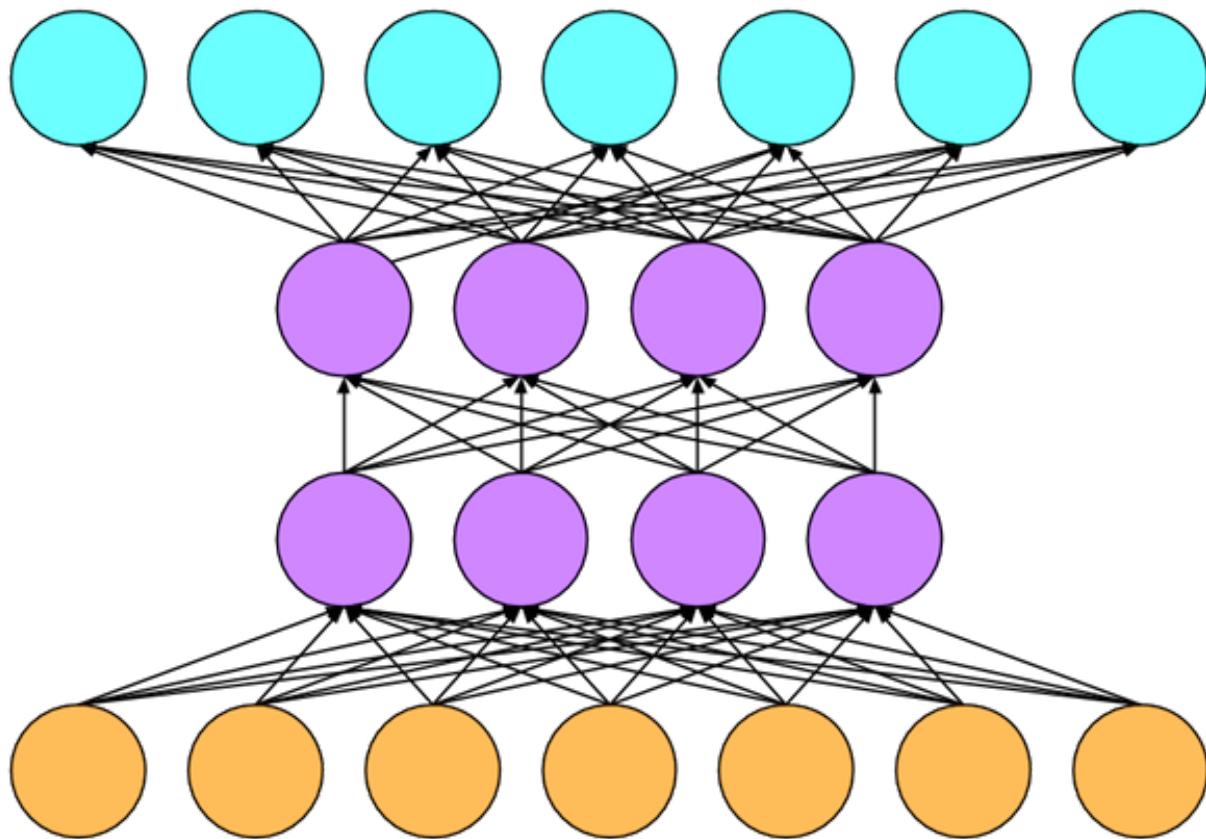
3.5.1 数据获取

我们继续使用 FashionMNIST 数据集。

```
In [1]: import sys  
        sys.path.append('..')  
        import utils  
batch_size = 256  
train_data, test_data = utils.load_data_fashion_mnist(batch_size)
```

3.5.2 多层感知机

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



这里我们定义一个只有一个隐含层的模型，这个隐含层输出 256 个节点。

```
In [2]: from mxnet import ndarray as nd  
  
num_inputs = 28*28
```

```

num_outputs = 10

num_hidden = 256
weight_scale = .01

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

W2 = nd.random_normal(shape=(num_hidden, num_outputs), scale=weight_scale)
b2 = nd.zeros(num_outputs)

params = [W1, b1, W2, b2]

for param in params:
    param.attach_grad()

```

3.5.3 激活函数

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

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

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

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

```
In [3]: def relu(X):
    return nd.maximum(X, 0)
```

3.5.4 定义模型

我们的模型就是将层（全连接）和激活函数（Relu）串起来：

```
In [4]: def net(X):
    X = X.reshape((-1, num_inputs))
    h1 = relu(nd.dot(X, W1) + b1)
    output = nd.dot(h1, W2) + b2
    return output
```

3.5.5 Softmax 和交叉熵损失函数

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

```
In [5]: from mxnet import gluon  
softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

3.5.6 训练

训练跟之前一样。

```
In [6]: from mxnet import autograd as autograd  
  
learning_rate = .5  
  
for epoch in range(5):  
    train_loss = 0.  
    train_acc = 0.  
    for data, label in train_data:  
        with autograd.record():  
            output = net(data)  
            loss = softmax_cross_entropy(output, label)  
            loss.backward()  
            utils.SGD(params, learning_rate/batch_size)  
  
            train_loss += nd.mean(loss).asscalar()  
            train_acc += utils.accuracy(output, label)  
  
    test_acc = utils.evaluate_accuracy(test_data, net)  
    print("Epoch %d. Loss: %f, Train acc %f, Test acc %f" % (  
        epoch, train_loss/len(train_data),  
        train_acc/len(train_data), test_acc))  
  
Epoch 0. Loss: 0.825024, Train acc 0.695813, Test acc 0.792167  
Epoch 1. Loss: 0.497659, Train acc 0.815288, Test acc 0.821314  
Epoch 2. Loss: 0.436145, Train acc 0.838325, Test acc 0.852264  
Epoch 3. Loss: 0.396678, Train acc 0.853582, Test acc 0.864083  
Epoch 4. Loss: 0.375521, Train acc 0.862213, Test acc 0.837540
```

3.5.7 总结

可以看到，加入一个隐含层后我们将精度提升了不少。

3.5.8 练习

- 我们使用了 `weight_scale` 来控制权重的初始化值大小，增大或者变小这个值会怎么样？
- 尝试改变 `num_hiddens` 来控制模型的复杂度
- 尝试加入一个新的隐含层

吐槽和讨论欢迎点[这里](#)

3.6 多层感知机—使用 Gluon

我们只需要稍微改动多类 Logistic 回归来实现多层感知机。

3.6.1 定义模型

唯一的区别在这里，我们加了一行进来。

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

net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.nn.Flatten())
    net.add(gluon.nn.Dense(256, activation="relu"))
    net.add(gluon.nn.Dense(10))
net.initialize()
```

3.6.2 读取数据并训练

```
In [2]: import sys
        sys.path.append('..')
        from mxnet import ndarray as nd
        from mxnet import autograd
        import utils
```

```
batch_size = 256
train_data, test_data = utils.load_data_fashion_mnist(batch_size)

softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.5})

for epoch in range(5):
    train_loss = 0.
    train_acc = 0.
    for data, label in train_data:
        with autograd.record():
            output = net(data)
            loss = softmax_cross_entropy(output, label)
            loss.backward()
        trainer.step(batch_size)

        train_loss += nd.mean(loss).asscalar()
        train_acc += utils.accuracy(output, label)

    test_acc = utils.evaluate_accuracy(test_data, net)
    print("Epoch %d. Loss: %f, Train acc %f, Test acc %f" % (
        epoch, train_loss/len(train_data), train_acc/len(train_data), test_acc))

Epoch 0. Loss: 0.734776, Train acc 0.730536, Test acc 0.801182
Epoch 1. Loss: 0.468885, Train acc 0.827741, Test acc 0.798878
Epoch 2. Loss: 0.423539, Train acc 0.843249, Test acc 0.860677
Epoch 3. Loss: 0.379952, Train acc 0.860911, Test acc 0.839844
Epoch 4. Loss: 0.361828, Train acc 0.866403, Test acc 0.855168
```

3.6.3 结论

通过 Gluon 我们可以更方便地构造多层神经网络。

3.6.4 练习

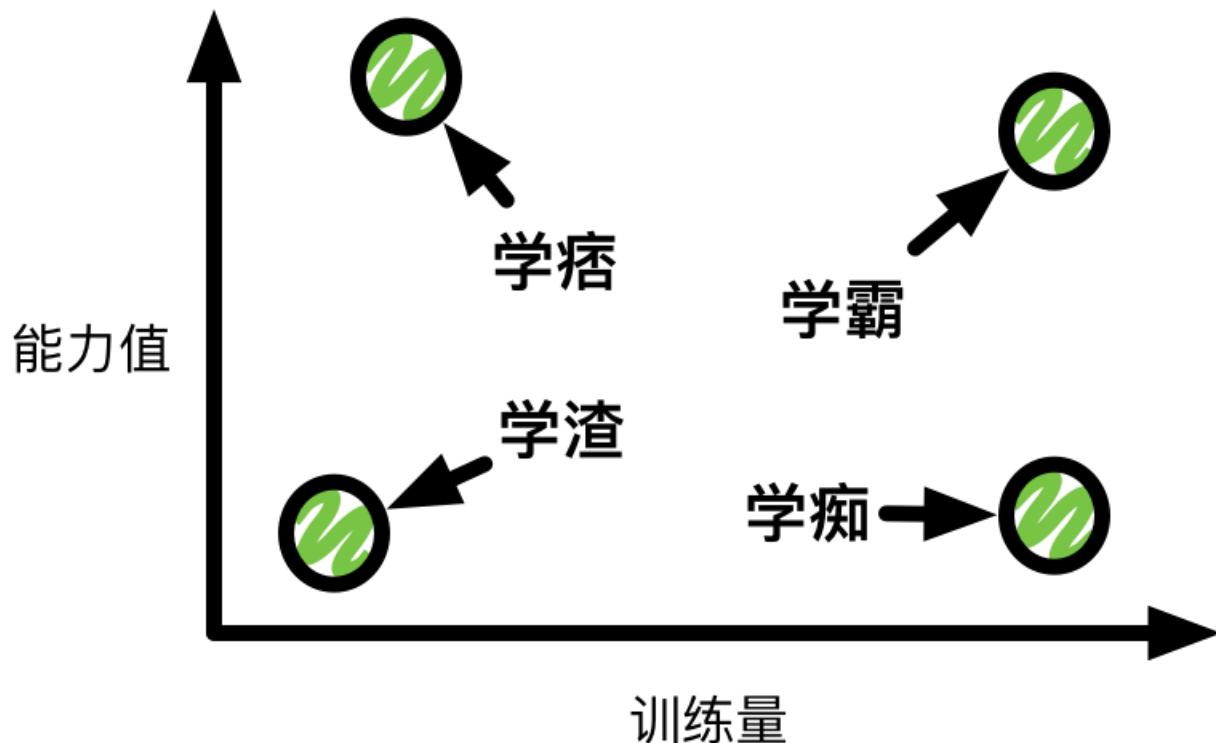
- 尝试多加入几个隐含层，对比从 0 开始的实现。
- 尝试使用一个另外的激活函数，可以使用 `help(nd.Activation)` 或者[线上文档](#)查看提供的选项。

吐槽和讨论欢迎点[这里](#)

3.7 欠拟合和过拟合

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

有人曾依据这种现象对学生群体简单粗暴地做了如下划分：



这里简要总结上图中学生的特点：

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

(现在问题来了，学酥应该在上图的哪里?)

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

3.7.1 训练误差（模考成绩）和泛化误差（考试成绩）

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

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

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

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

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

因此，一个重要结论是：

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

3.7.2 欠拟合和过拟合

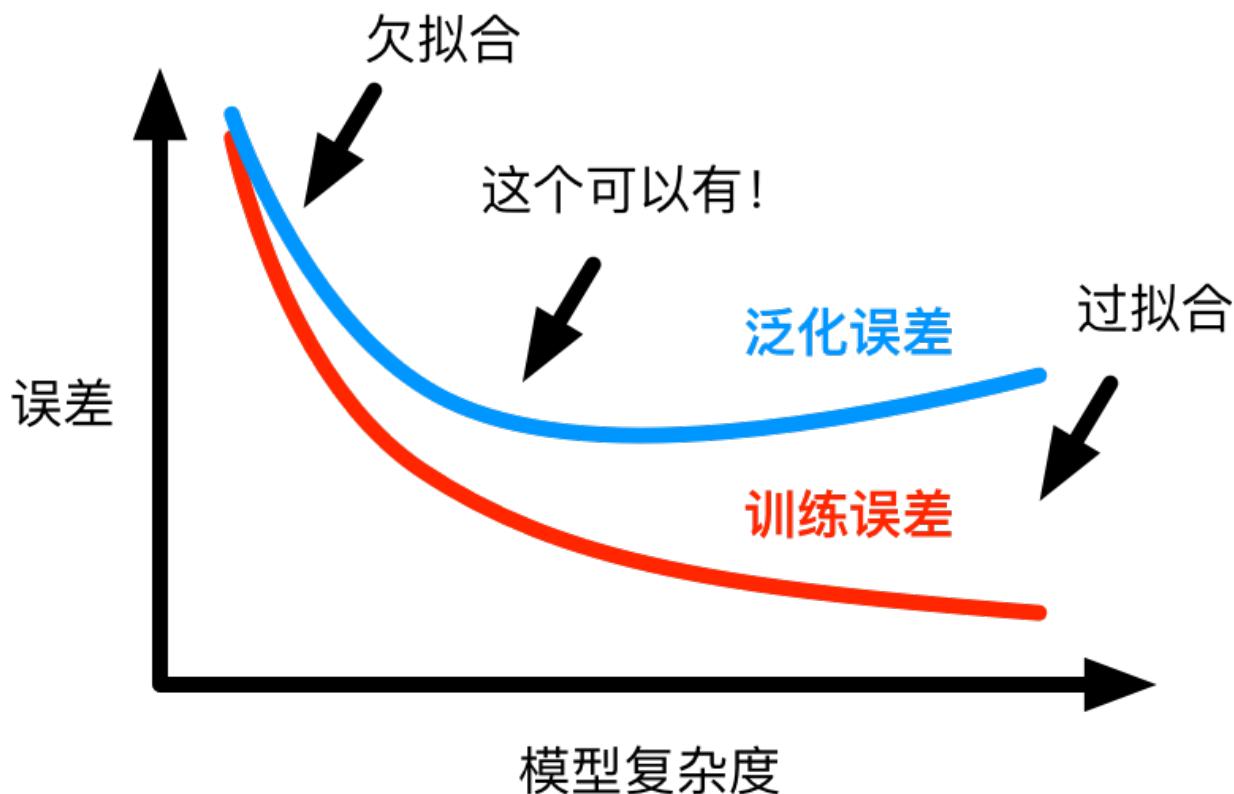
实践中，如果测试数据集是给定的，我们通常用机器学习模型在该测试数据集的误差来反映泛化误差。基于上述重要结论，以下两种拟合问题值得注意：

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

我们要尽可能同时避免欠拟合和过拟合的出现。虽然有很多因素可能导致这两种拟合问题，在这里我们重点讨论两个因素：模型的选择和训练数据集的大小。

模型的选择

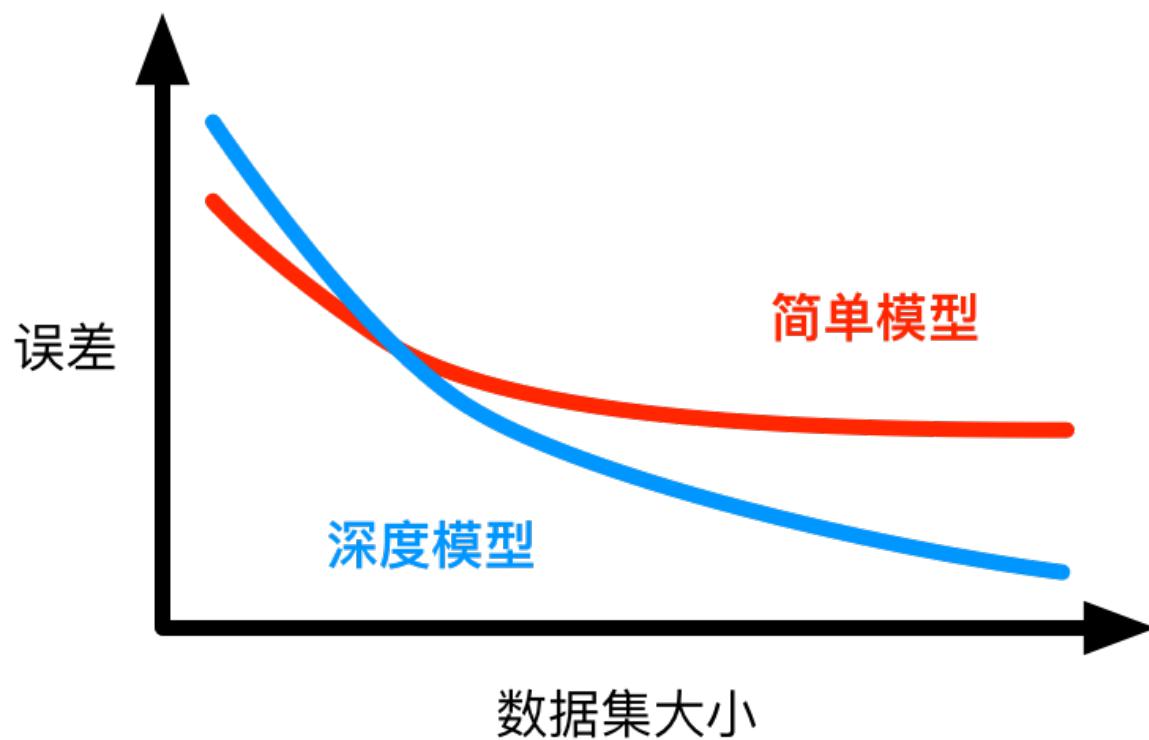
在本节的开头，我们提到一个学生可以有特定的学习能力。类似地，一个机器学习模型也有特定的拟合能力。拿多项式函数举例，一般来说，高阶多项式函数（拟合能力较强）比低阶多项式函数（拟合能力较弱）更容易在相同的训练数据集上得到较低的训练误差。需要指出的是，给定数据集，过低拟合能力的模型更容易欠拟合，而过高拟合能力的模型更容易过拟合。模型拟合能力和误差之间的关系如下图。



训练数据集的大小

在本节的开头，我们同样提到一个学生可以有特定的训练量。类似地，一个机器学习模型的训练数据集的样本数也可大可小。一般来说，如果训练数据集过小，特别是比模型参数数量更小时，过拟合更容易发生。除此之外，泛化误差不会随训练数据集里样本数量增加而增大。

为了理解这两个因素对拟合和过拟合的影响，下面让我们来动手学习。



3.7.3 多项式拟合

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

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

并以平方误差为损失函数。特别地，一阶多项式拟合又叫线性拟合。

创建数据集

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

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

这里噪音服从均值 0 和标准差为 0.1 的正态分布。

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

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

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

下面生成数据集。

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

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

Out[2]: ('x:',
          [[ 0.30030754]
           [ 0.23107235]
           [ 1.04932892]
           [-0.32433933]
           [-0.0097888 ]]
          <NDArray 5x1 @cpu(0)>, 'X:',
          [[ 3.00307542e-01   9.01846215e-02   2.70831212e-02]
           [ 2.31072351e-01   5.33944331e-02   1.23379771e-02]
           [ 1.04932892e+00   1.10109115e+00   1.15540683e+00]
           [ -3.24339330e-01   1.05195999e-01   -3.41192000e-02]
           [ -9.78880003e-03   9.58206074e-05   -9.37968764e-07]]
          <NDArray 5x3 @cpu(0)>, 'y:',
          [ 5.06436014   5.2515564    8.8890419    3.89964342   5.04119158]
          <NDArray 5 @cpu(0)>)
```

定义训练和测试步骤

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

以下的训练步骤在[使用 Gluon 的线性回归](#)有过详细描述。这里不再赘述。

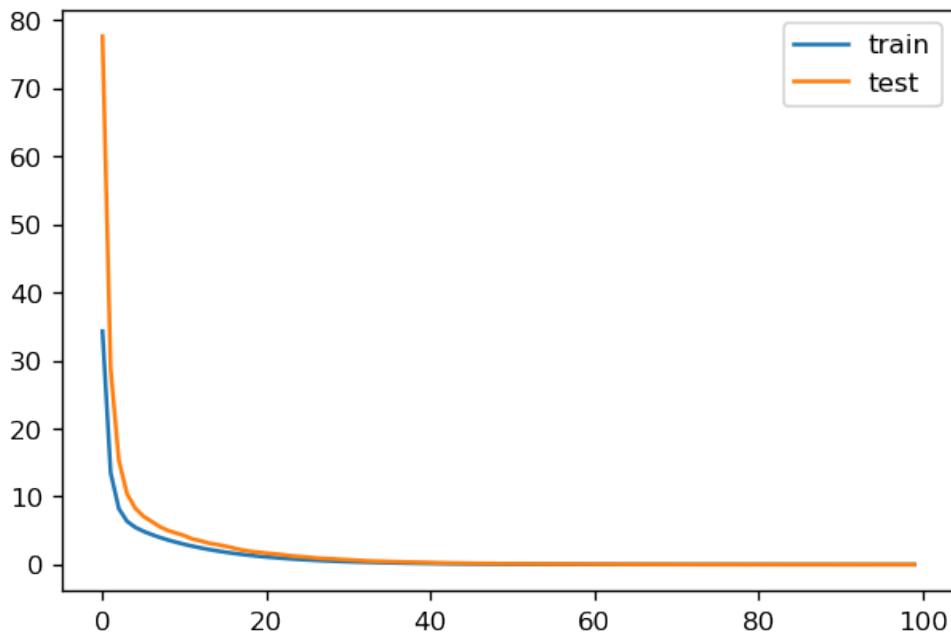
```
In [3]: %matplotlib inline
        import matplotlib as mpl
        mpl.rcParams['figure.dpi']= 120
        import matplotlib.pyplot as plt
```

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

三阶多项式拟合（正常）

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

```
In [4]: train(X[:num_train, :], X[num_train:, :], y[:num_train], y[num_train:])
```

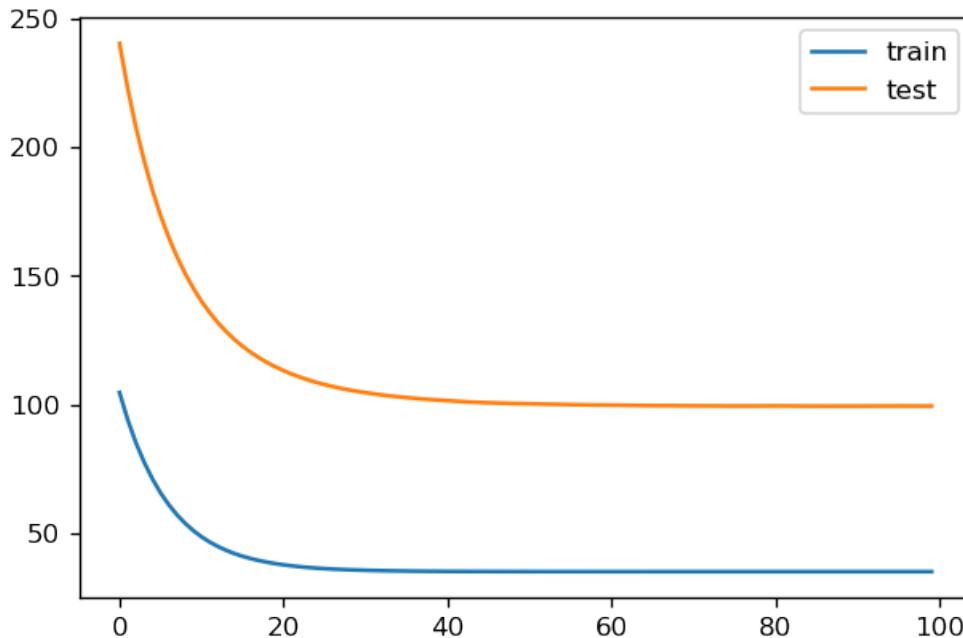


```
Out[4]: ('learned weight',
 [[ 1.30287039 -3.37328291  5.56664324]]
 <NDArray 1x3 @cpu(0)>, 'learned bias',
 [ 4.96160173]
 <NDArray 1 @cpu(0)>)
```

线性拟合（欠拟合）

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

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

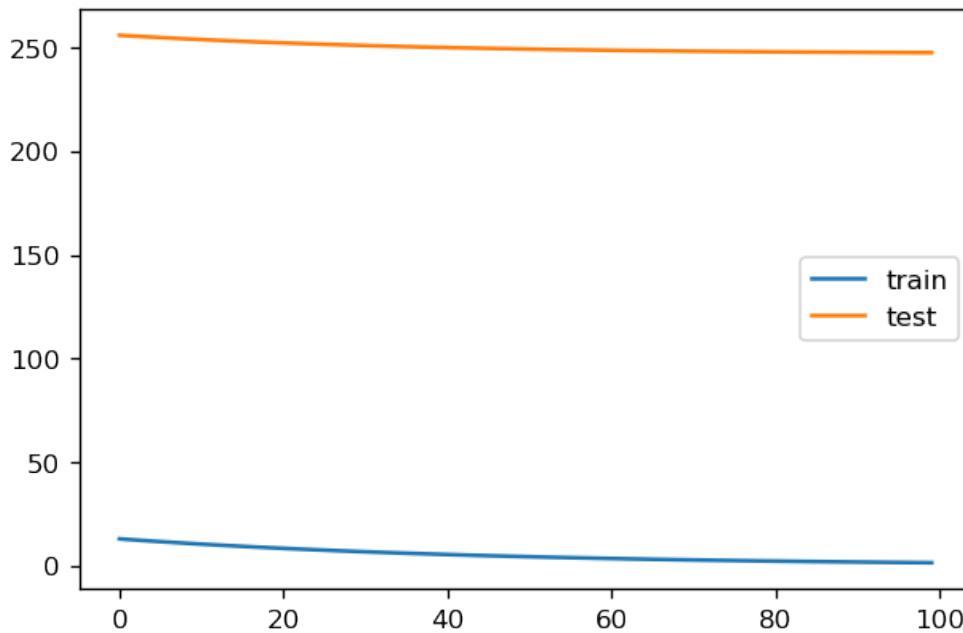


```
Out[5]: ('learned weight',
 [[ 14.05198097]],
 <NDArray 1x1 @cpu(0)>, 'learned bias',
 [ 2.15997887]
 <NDArray 1 @cpu(0)>)
```

训练量不足（过拟合）

事实上，即便是使用与数据生成模型同阶的三阶多项式模型，如果训练量不足，该模型依然容易过拟合。让我们仅仅使用两个训练样本来训练。很显然，训练样本过少了，甚至少于模型参数的数量。这使模型显得过于复杂，以至于容易被训练数据集中的噪音影响。在机器学习过程中，即便训练误差很低，但是测试数据集上的误差很高。这是典型的过拟合现象。

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



```
Out[6]: ('learned weight',
 [[ 0.83708268  0.15890324  0.10728332]]
 <NDArray 1x3 @cpu(0)>, 'learned bias',
 [ 3.17162633]
 <NDArray 1 @cpu(0)>)
```

我们还将在后面的章节继续讨论过拟合问题以及应对过拟合的方法，例如正则化。

3.7.4 结论

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

3.7.5 练习

1. 学渣、学痞、学痴和学霸对应的模型复杂度、训练量、训练误差和泛化误差分别是怎样的？
2. 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？
3. 在我们本节提到的三阶多项式拟合问题里，有没有可能把 1000 个样本的训练误差的期望降到 0，为什么？

吐槽和讨论欢迎点[这里](#)

3.8 正则化—从 0 开始

本章从 0 开始介绍如何的正则化来应对过拟合问题。

3.8.1 高维线性回归

我们使用高维线性回归为例来引入一个过拟合问题。

具体来说我们使用如下的线性函数来生成每一个数据样本

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

这里噪音服从均值 0 和标准差为 0.01 的正态分布。

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

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

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

3.8.2 生成数据集

这里定义模型真实参数。

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

我们接着生成训练和测试数据集。

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

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

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

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

3.8.3 初始化模型参数

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

```
In [5]: def init_params():
    w = nd.random_normal(scale=1, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    for param in params:
        param.attach_grad()
    return params
```

3.8.4 L_2 范数正则化

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

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

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

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

3.8.5 定义训练和测试

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

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

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

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

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

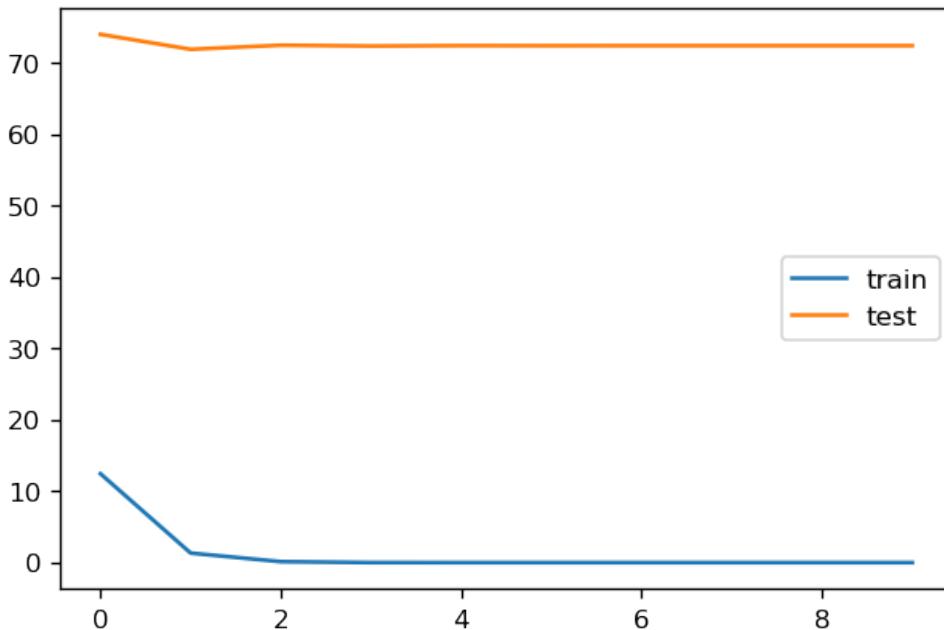
def test(net, params, X, y):
    return square_loss(net(X, *params), y).mean().asscalar()
#return np.mean(square_loss(net(X, *params), y).asnumpy())

def train(lambd):
    epochs = 10
    learning_rate = 0.005
    w, b = params = init_params()
    train_loss = []
    test_loss = []
    for e in range(epochs):
        for data, label in data_iter(num_train):
            with autograd.record():
                output = net(data, *params)
                loss = square_loss(
                    output, label) + lambd * L2_penalty(*params)
            loss.backward()
            sgd(params, learning_rate, batch_size)
        train_loss.append(test(net, params, X_train, y_train))
        test_loss.append(test(net, params, X_test, y_test))
    plt.plot(train_loss)
    plt.plot(test_loss)
    plt.legend(['train', 'test'])
    plt.show()
    return 'learned w[:10]:', w[:10].T, 'learned b:', b
```

3.8.6 观察过拟合

接下来我们训练并测试我们的高维线性回归模型。注意这时我们并未使用正则化。

In [8]: `train(0)`



Out[8]: ('learned w[:10]:',
 [[0.41584426 0.66337717 0.70623302 0.13857912 0.60508025 0.95260203
 -0.28788593 -1.50719547 -0.93943435 0.5070892]]
<NDArray 1x10 @cpu(0)>, 'learned b:',
 [-0.31562975]
<NDArray 1 @cpu(0)>)

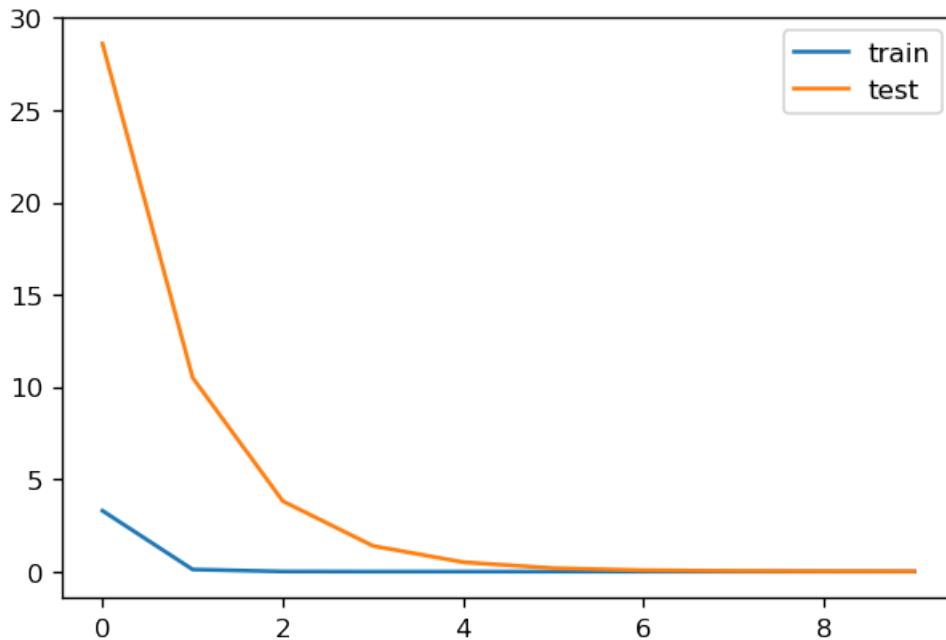
即便训练误差可以达到 0.000000，但是测试数据集上的误差很高。这是典型的过拟合现象。

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

3.8.7 使用正则化

下面我们重新初始化模型参数并设置一个正则化参数。

In [9]: `train(5)`



```
Out[9]: ('learned w[:10]:',
 [[ 0.00162978  0.00496679  0.00073217  0.01498246 -0.00378662  0.006839
   0.01388358 -0.00180659  0.0036204  -0.01029951]],
 <NDArray 1x10 @cpu(0)>, 'learned b:',
 [-0.00060231]
 <NDArray 1 @cpu(0)>)
```

我们发现训练误差虽然有所提高，但测试数据集上的误差有所下降。过拟合现象得到缓解。但打印出的学到的参数依然不是很理想，这主要是因为我们训练数据的样本相对维度来说太少。

3.8.8 结论

- 我们可以使用正则化来应对过拟合问题。

3.8.9 练习

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

吐槽和讨论欢迎点[这里](#)

3.9 正则化—使用 Gluon

本章介绍如何使用 Gluon 的正则化来应对过拟合问题。

3.9.1 高维线性回归数据集

我们使用与上一节相同的高维线性回归为例来引入一个过拟合问题。

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

        num_train = 20
        num_test = 100
        num_inputs = 200

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

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

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

3.9.2 定义训练和测试

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

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

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

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

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

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

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

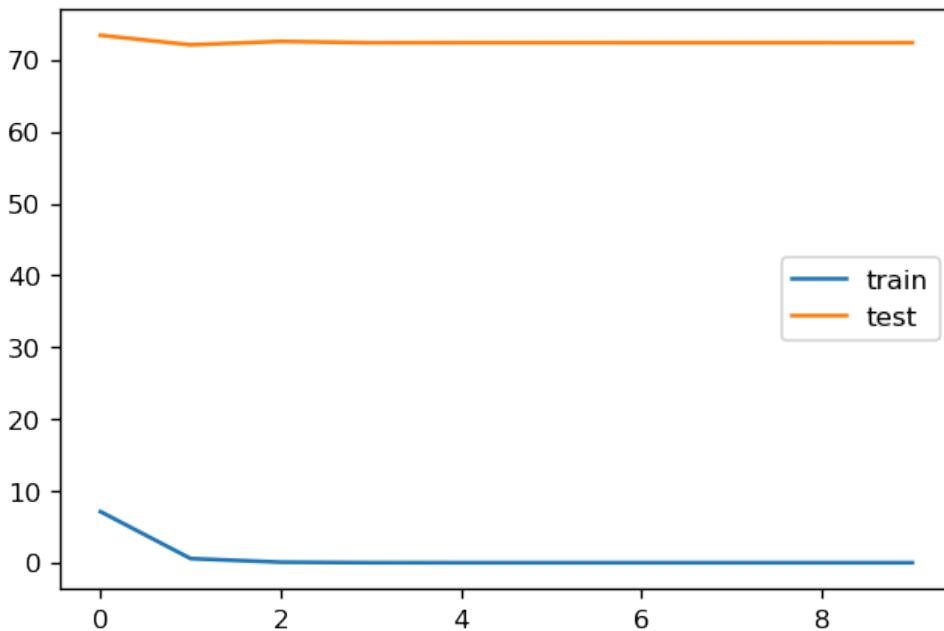
    train_loss = []
    test_loss = []
    for e in range(epochs):
        for data, label in data_iter_train:
            with autograd.record():
                output = net(data)
                loss = square_loss(output, label)
                loss.backward()
                trainer.step(batch_size)
        train_loss.append(test(net, X_train, y_train))
        test_loss.append(test(net, X_test, y_test))
        plt.plot(train_loss)
        plt.plot(test_loss)
        plt.legend(['train', 'test'])
        plt.show()

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

训练模型并观察拟合

接下来我们训练并测试我们的高维线性回归模型。

In [3]: `train(0)`



```
Out[3]: ('learned w[:10]:',
 [[ 0.41583112  0.66335684  0.70626092  0.13858539  0.60507381  0.95261663
 -0.28789261 -1.50719035 -0.93944377  0.50710166]],
 <NDArray 1x10 @cpu(0)>, 'learned b:',
 [-0.31561956]
 <NDArray 1 @cpu(0)>)
```

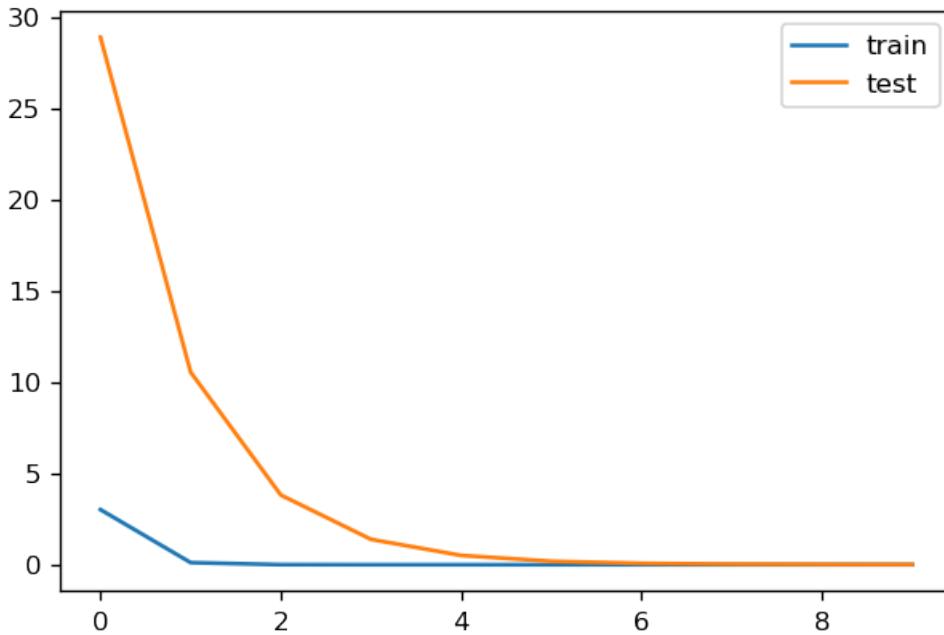
即便训练误差可以达到 0.000000，但是测试数据集上的误差很高。这是典型的过拟合现象。

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

3.9.3 使用 Gluon 的正则化

下面我们重新初始化模型参数并在 Trainer 里设置一个 wd 参数。

```
In [4]: train(5)
```



```
Out[4]: ('learned w[:10]:',
 [[ 0.00278991  0.00533668  0.0009949   0.01480813 -0.00382196  0.00712715
    0.01488133 -0.00151321  0.00375793 -0.00982257]],
 <NDArray 1x10 @cpu(0)>, 'learned b:',
 [-0.00052505]
 <NDArray 1 @cpu(0)>)
```

我们发现训练误差虽然有所提高，但测试数据集上的误差有所下降。过拟合现象得到缓解。但打印出的学到的参数依然不是很理想，这主要是因为我们训练数据的样本相对维度来说太少。

3.9.4 结论

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

3.9.5 练习

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

吐槽和讨论欢迎点[这里](#)

3.10 丢弃法 (Dropout) — 从 0 开始

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

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

3.10.1 丢弃法的概念

在现代神经网络中，我们所指的丢弃法，通常是对输入层或者隐含层做以下操作：

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

3.10.2 丢弃法的实现

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

In [1]: `from mxnet import nd`

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

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

我们运行几个实例来验证一下。

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

Out[2]:

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

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

Out[3]:

```
[[ 0.  2.  4.  6.]
 [ 8.  10.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 24.  0.  0.  30.]
 [ 0.  0.  36.  0.]]
<NDArray 5x4 @cpu(0)>
```

```
In [4]: dropout(A, 1.0)
```

Out[4]:

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

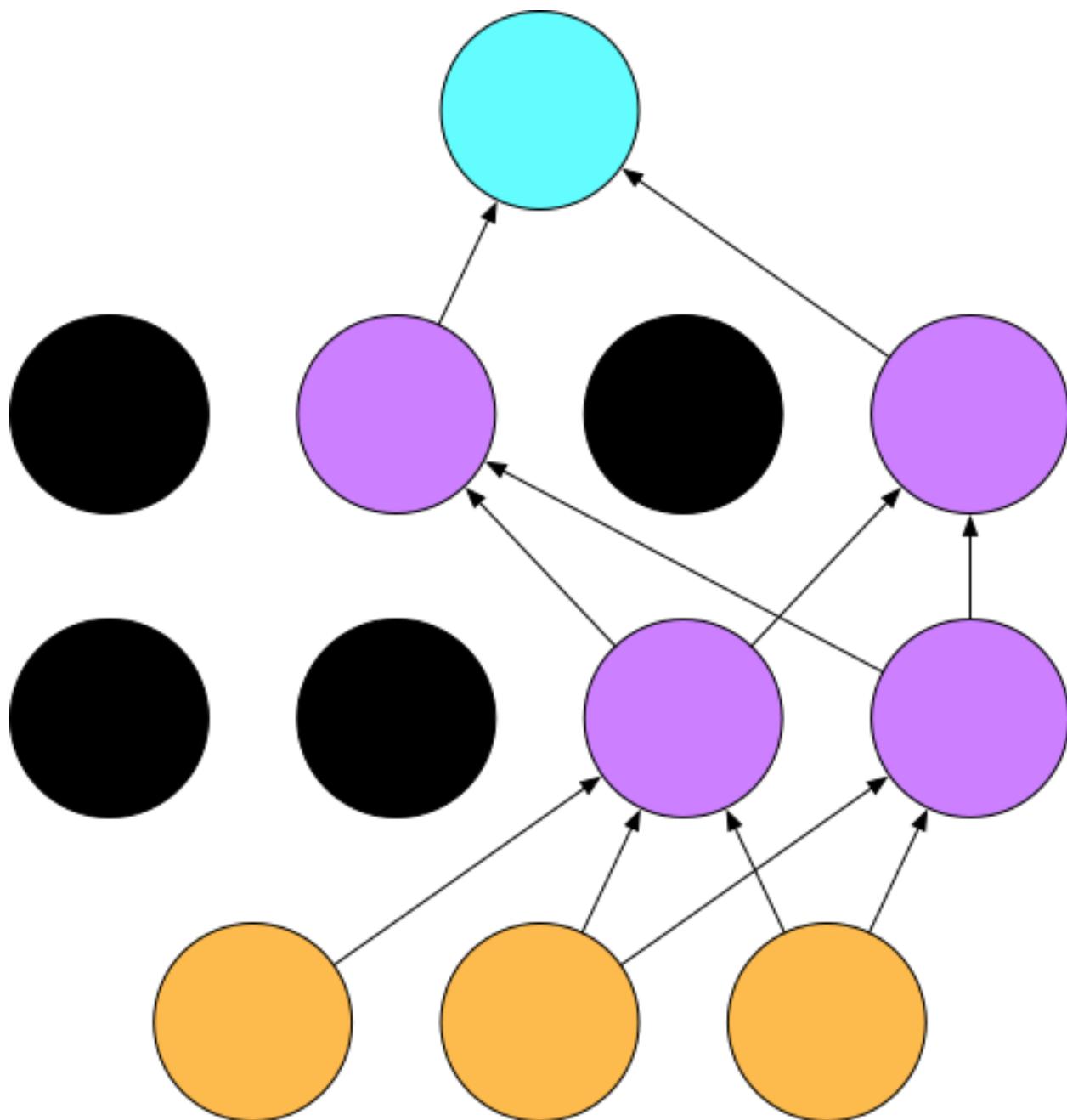
3.10.3 丢弃法的本质

了解了丢弃法的概念与实现，那你可能对它的本质产生了好奇。

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

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

我们在之前的章节里介绍过[随机梯度下降算法](#)：我们在训练神经网络模型时一般随机采样一个批量的训练数据。丢弃法实质上是对每一个这样的数据集分别训练一个原神经网络子集的分类器。与一



般的集成学习不同，这里每个原神经网络子集的分类器用的是同一套参数。因此丢弃法只是在模拟集成学习。

我们刚刚强调了，原神经网络子集的分类器在不同的训练数据批量上训练并使用同一套参数。因此，使用丢弃法的神经网络实质上是对输入层和隐含层的参数做了正则化：学到的参数使得原神经网络不同子集在训练数据上都尽可能表现良好。

下面我们动手实现一下在多层神经网络里加丢弃层。

3.10.4 数据获取

我们继续使用 FashionMNIST 数据集。

```
In [5]: import sys
        sys.path.append('..')
        import utils
        batch_size = 256
        train_data, test_data = utils.load_data_fashion_mnist(batch_size)
```

3.10.5 含两个隐藏层的多层感知机

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

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

        num_hidden1 = 256
        num_hidden2 = 256
        weight_scale = .01

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

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

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

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

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

3.10.6 定义包含丢弃层的模型

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

```
In [7]: drop_prob1 = 0.2  
drop_prob2 = 0.5  
  
def net(X):  
    X = X.reshape((-1, num_inputs))  
    # 第一层全连接。  
    h1 = nd.relu(nd.dot(X, W1) + b1)  
    # 在第一层全连接后添加丢弃层。  
    h1 = dropout(h1, drop_prob1)  
    # 第二层全连接。  
    h2 = nd.relu(nd.dot(h1, W2) + b2)  
    # 在第二层全连接后添加丢弃层。  
    h2 = dropout(h2, drop_prob2)  
    return nd.dot(h2, W3) + b3
```

3.10.7 训练

训练跟之前一样。

```
In [8]: from mxnet import autograd  
from mxnet import gluon  
  
softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()  
  
learning_rate = .5  
  
for epoch in range(5):  
    train_loss = 0.  
    train_acc = 0.  
    for data, label in train_data:
```

```
with autograd.record():
    output = net(data)
    loss = softmax_cross_entropy(output, label)
    loss.backward()
    utils.SGD(params, learning_rate/batch_size)

    train_loss += nd.mean(loss).asscalar()
    train_acc += utils.accuracy(output, label)

test_acc = utils.evaluate_accuracy(test_data, net)
print("Epoch %d. Loss: %f, Train acc %f, Test acc %f" % (
    epoch, train_loss/len(train_data),
    train_acc/len(train_data), test_acc))

Epoch 0. Loss: 1.317566, Train acc 0.488448, Test acc 0.704327
Epoch 1. Loss: 0.621804, Train acc 0.768162, Test acc 0.803786
Epoch 2. Loss: 0.510275, Train acc 0.812417, Test acc 0.829928
Epoch 3. Loss: 0.466563, Train acc 0.830061, Test acc 0.841446
Epoch 4. Loss: 0.438626, Train acc 0.838926, Test acc 0.850160
```

3.10.8 总结

我们可以通过使用丢弃法对神经网络正则化。

3.10.9 练习

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

吐槽和讨论欢迎点[这里](#)

3.11 丢弃法 (Dropout) — 使用 Gluon

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

3.11.1 定义模型并添加丢弃层

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

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

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

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

3.11.2 读取数据并训练

这跟之前没什么不同。

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

        batch_size = 256
        train_data, test_data = utils.load_data_fashion_mnist(batch_size)

        softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
        trainer = gluon.Trainer(net.collect_params(),
```

```
'sgd', {'learning_rate': 0.5})\n\n    for epoch in range(5):\n        train_loss = 0.\n        train_acc = 0.\n        for data, label in train_data:\n            with autograd.record():\n                output = net(data)\n                loss = softmax_cross_entropy(output, label)\n                loss.backward()\n                trainer.step(batch_size)\n\n                train_loss += nd.mean(loss).asscalar()\n                train_acc += utils.accuracy(output, label)\n\n        test_acc = utils.evaluate_accuracy(test_data, net)\n        print("Epoch %d. Loss: %f, Train acc %f, Test acc %f" % (\n            epoch, train_loss/len(train_data),\n            train_acc/len(train_data), test_acc))\n\nEpoch 0. Loss: 0.826515, Train acc 0.693660, Test acc 0.799379\nEpoch 1. Loss: 0.509185, Train acc 0.812851, Test acc 0.839343\nEpoch 2. Loss: 0.450815, Train acc 0.836038, Test acc 0.817808\nEpoch 3. Loss: 0.418305, Train acc 0.846888, Test acc 0.848958\nEpoch 4. Loss: 0.395018, Train acc 0.855786, Test acc 0.860377
```

3.11.3 结论

通过 Gluon 我们可以更方便地构造多层神经网络并使用丢弃法。

3.11.4 练习

- 尝试不同元素丢弃概率参数组合，看看结果有什么不同。

吐槽和讨论欢迎点[这里](#)

3.12 正向传播和反向传播

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

型参数的梯度。

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

3.12.1 概念

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

反向传播对于各层中变量和参数的梯度计算可能会依赖各层变量和参数的当前值。对深度学习模型按照输入层、靠近输入层的隐含层、靠近输出层的隐含层和输出层的次序，依次计算并存储模型的中间变量叫做正向传播（forward-propagation）。

3.12.2 案例分析——正则化的多层感知机

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

模型定义

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

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

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

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

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

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

假设损失函数为 ℓ ，损失项

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

根据 L_2 范数正则化的定义，带有提前设定的超参数 λ 的正则化项

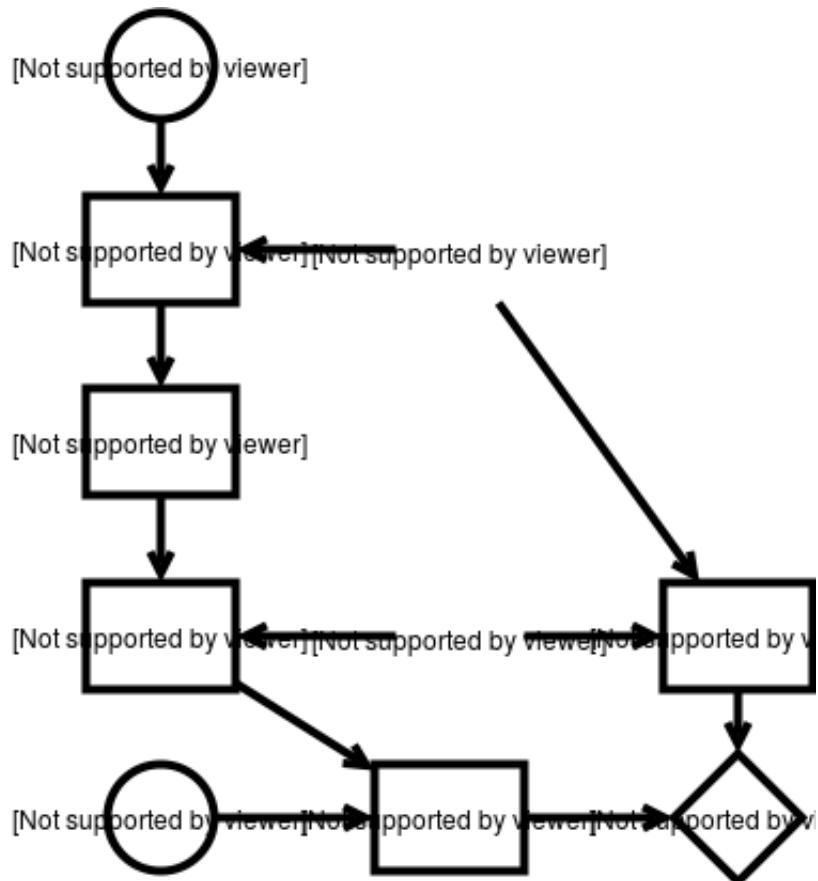
$$s = \frac{\lambda}{2} (\|\mathbf{W}^{(1)}\|_2^2 + \|\mathbf{W}^{(2)}\|_2^2)$$

模型最终需要被优化的目标函数

$$J = L + s$$

计算图

为了可视化模型变量和参数之间在计算中的依赖关系，我们可以绘制计算图。



梯度的计算与存储

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

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

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

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

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

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

来表达链式法则。以下依次计算得到的梯度将依次被存储。

首先，我们计算目标函数有关损失项和有关正则项的梯度

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

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

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

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}(\frac{\partial J}{\partial L} \square \frac{\partial L}{\partial \mathbf{o}}) = \frac{\partial L}{\partial \mathbf{o}}$$

正则项有关两个参数的梯度可以很直观地计算：

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

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

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

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

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

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

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

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

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

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

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

还有需要注意的是，反向传播对于各层中变量和参数的梯度计算可能会依赖通过正向传播计算出的各层变量和参数的当前值。举例来说，参数梯度 $\partial J / \partial \mathbf{W}^{(2)}$ 的计算需要依赖隐含层变量的当前值 \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。

3.12.3 总结

正向传播和反向传播是深度学习模型训练的基石（目前是）。

3.12.4 练习

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

吐槽和讨论欢迎点[这里](#)

3.13 实战 Kaggle 比赛——使用 Gluon 预测房价和 K 折交叉验证

本章介绍如何使用 Gluon 来实战 Kaggle 比赛。我们以房价预测问题为例，为大家提供一整套实战中常常需要的工具，例如 **K 折交叉验证**。我们还以 pandas 为工具介绍如何对真实世界中的数据进行重要的预处理，例如：

- 处理离散数据
- 处理丢失的数据特征
- 对数据进行标准化

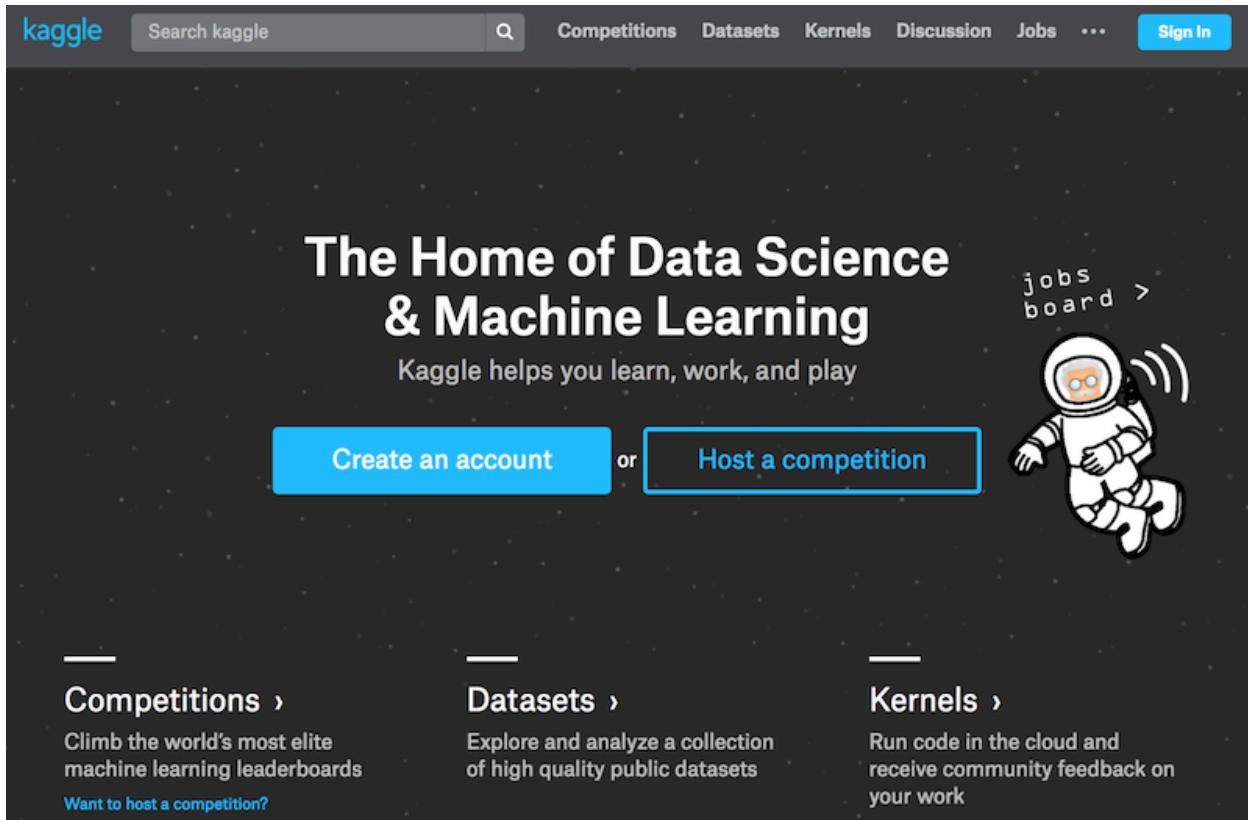
需要注意的是，本章仅提供一些基本实战流程供大家参考。对于数据的预处理、模型的设计和参数的选择等，我们特意只提供最基础的版本。希望大家一定要通过动手实战、仔细观察实验现象、认真分析实验结果并不断调整方法，从而得到令自己满意的结果。

这是一次宝贵的实战机会，我们相信你一定能从动手的过程中学到很多。

Get your hands dirty。

3.13.1 Kaggle 中的房价预测问题

Kaggle是一个著名的供机器学习爱好者交流的平台。为了便于提交结果,请大家注册Kaggle账号。请注意,目前 Kaggle 仅限每个账号一天以内 10 次提交结果的机会。所以提交结果前务必三思。



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

3.13.2 读入数据

比赛数据分为训练数据集和测试数据集。两个数据集都包括每个房子的特征,例如街道类型、建造年份、房顶类型、地下室状况等特征值。这些特征值有连续的数字、离散的标签甚至是缺失值’na’。只有训练数据集包括了我们需要在测试数据集中预测的每个房子的价格。数据可以从[房价预测问题](#)中下载。

[训练数据集下载地址](#) [测试数据集下载地址](#)

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

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



House Prices: Advanced Regression Techniques

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

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

Overview

Description	Start here if...
Evaluation	You have some experience with R or Python and machine learning basics. This is a perfect competition for data science students who have completed an online course in machine learning and are looking to expand their skill set before trying a featured competition.
Frequently Asked Questions	
Tutorials	

Competition Description



Ask a home buyer to describe their dream house, and they probably won't begin with the height of the basement ceiling or the proximity to an east-west railroad. But this playground competition's dataset proves that much more influences price negotiations than the number of bedrooms or a white-picket fence.

With 79 explanatory variables describing (almost) every aspect of residential homes in Ames, Iowa, this competition challenges you to predict the final price of each home.

```

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

```

我们看看数据长什么样子。

In [2]: train.head()

```

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

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

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

```

[5 rows x 81 columns]

数据大小如下。

In [3]: train.shape

Out[3]: (1460, 81)

In [4]: test.shape

Out[4]: (1459, 80)

3.13.3 预处理数据

我们使用 pandas 对数值特征做标准化处理:

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

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

现在把离散数据点转换成数值标签。

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

把缺失数据用本特征的平均值估计。

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

下面把数据转换一下格式。

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

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

3.13.4 导入 NDArray 格式数据

为了便于和 Gluon 交互, 我们需要导入 NDArray 格式数据。

```
In [9]: from mxnet import ndarray as nd  
from mxnet import autograd  
from mxnet import gluon  
  
X_train = nd.array(X_train)  
y_train = nd.array(y_train)  
y_train.reshape((num_train, 1))  
  
X_test = nd.array(X_test)
```

我们把损失函数定义为平方误差。

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

我们定义比赛中测量结果用的函数。

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

3.13.5 定义模型

我们将模型的定义放在一个函数里供多次调用。这是一个基本的线性回归模型。

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

我们定义一个训练的函数，这样在跑不同的实验时不需要重复实现相同的步骤。

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

def train(net, X_train, y_train, X_test, y_test, epochs,
          verbose_epoch, learning_rate, weight_decay):
    train_loss = []
    if X_test is not None:
        test_loss = []
    batch_size = 100
    dataset_train = gluon.data.ArrayDataset(X_train, y_train)
    data_iter_train = gluon.data.DataLoader(
        dataset_train, batch_size, shuffle=True)
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                           {'learning_rate': learning_rate,
                            'wd': weight_decay})
    net.collect_params().initialize(force_reinit=True)
    for epoch in range(epochs):
        for data, label in data_iter_train:
            with autograd.record():
                output = net(data)
                loss = square_loss(output, label)
```

```
loss.backward()
trainer.step(batch_size)

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

3.13.6 K 折交叉验证

在过拟合中我们讲过，过度依赖训练数据集的误差来推断测试数据集的误差容易导致过拟合。事实上，当我们调参时，往往需要基于 K 折交叉验证。

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

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

```
In [14]: def k_fold_cross_valid(k, epochs, verbose_epoch, X_train, y_train,
                           learning_rate, weight_decay):
    assert k > 1
    fold_size = X_train.shape[0] // k
    train_loss_sum = 0.0
    test_loss_sum = 0.0
    for test_i in range(k):
        X_val_test = X_train[test_i * fold_size: (test_i + 1) * fold_size, :]
        y_val_test = y_train[test_i * fold_size: (test_i + 1) * fold_size]
```

```

val_train_defined = False
for i in range(k):
    if i != test_i:
        X_cur_fold = X_train[i * fold_size: (i + 1) * fold_size, :]
        y_cur_fold = y_train[i * fold_size: (i + 1) * fold_size]
    if not val_train_defined:
        X_val_train = X_cur_fold
        y_val_train = y_cur_fold
        val_train_defined = True
    else:
        X_val_train = nd.concat(X_val_train, X_cur_fold, dim=0)
        y_val_train = nd.concat(y_val_train, y_cur_fold, dim=0)
net = get_net()
train_loss, test_loss = train(
    net, X_val_train, y_val_train, X_val_test, y_val_test,
    epochs, verbose_epoch, learning_rate, weight_decay)
train_loss_sum += train_loss
print("Test loss: %f" % test_loss)
test_loss_sum += test_loss
return train_loss_sum / k, test_loss_sum / k

```

训练模型并交叉验证

以下的模型参数都是可以调的。

In [15]: `k = 5`

```

epochs = 100
verbose_epoch = 95
learning_rate = 5
weight_decay = 0.0

```

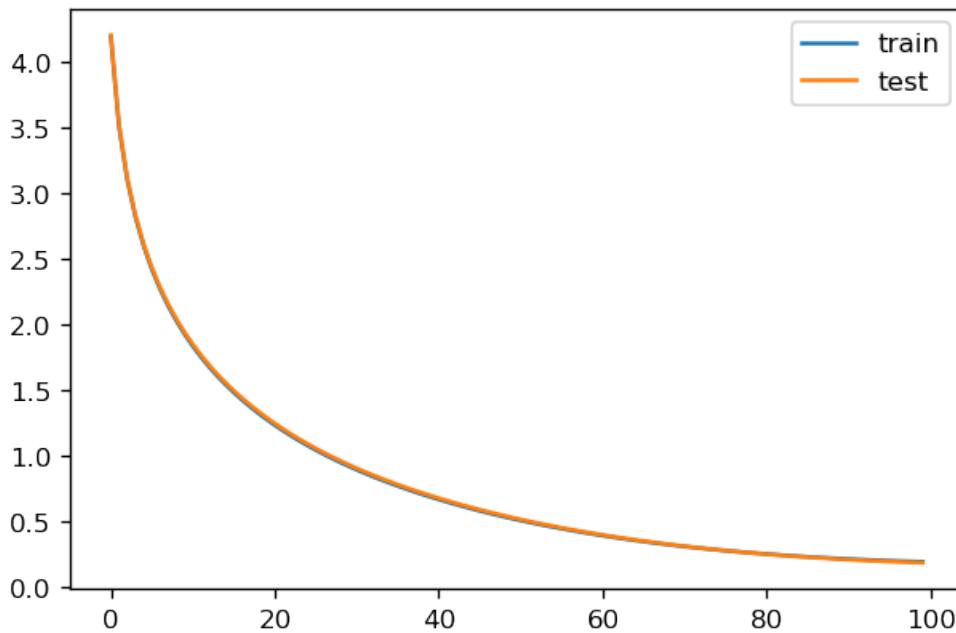
给定以上调好的参数，接下来我们训练并交叉验证我们的模型。

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

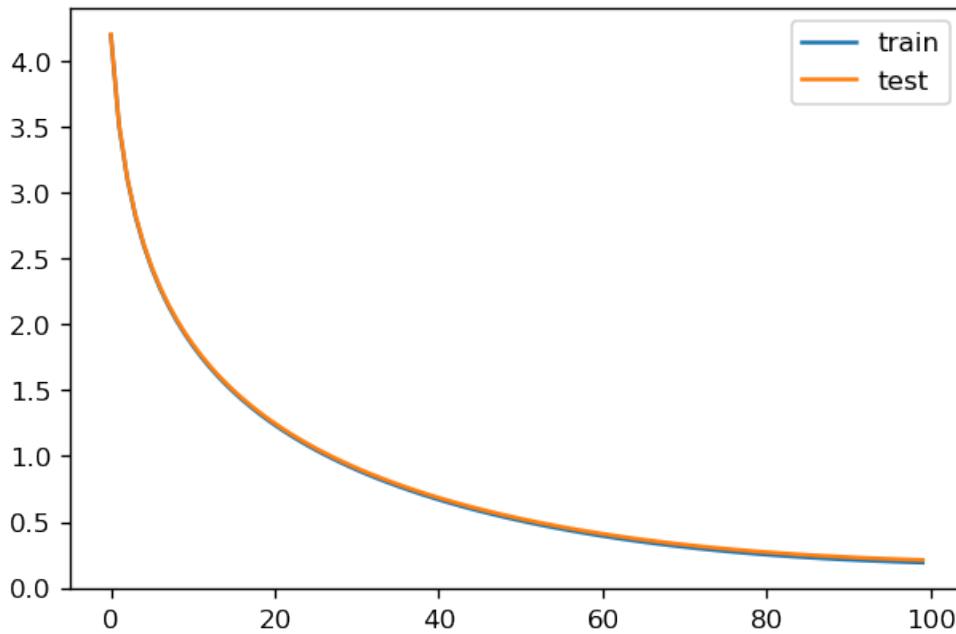
```

Epoch 96, train loss: 0.201997
Epoch 97, train loss: 0.199898
Epoch 98, train loss: 0.197913
Epoch 99, train loss: 0.196039

```



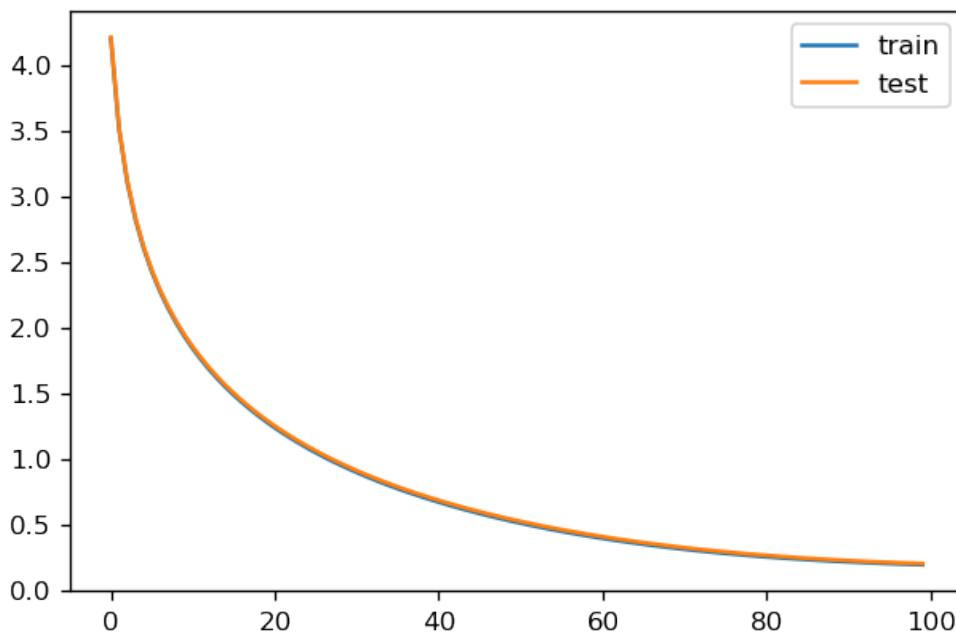
```
Test loss: 0.189099
Epoch 96, train loss: 0.198053
Epoch 97, train loss: 0.195932
Epoch 98, train loss: 0.193857
Epoch 99, train loss: 0.191906
```



```
Test loss: 0.211700
Epoch 96, train loss: 0.198951
Epoch 97, train loss: 0.196805
```

Epoch 98, train loss: 0.194780

Epoch 99, train loss: 0.192834



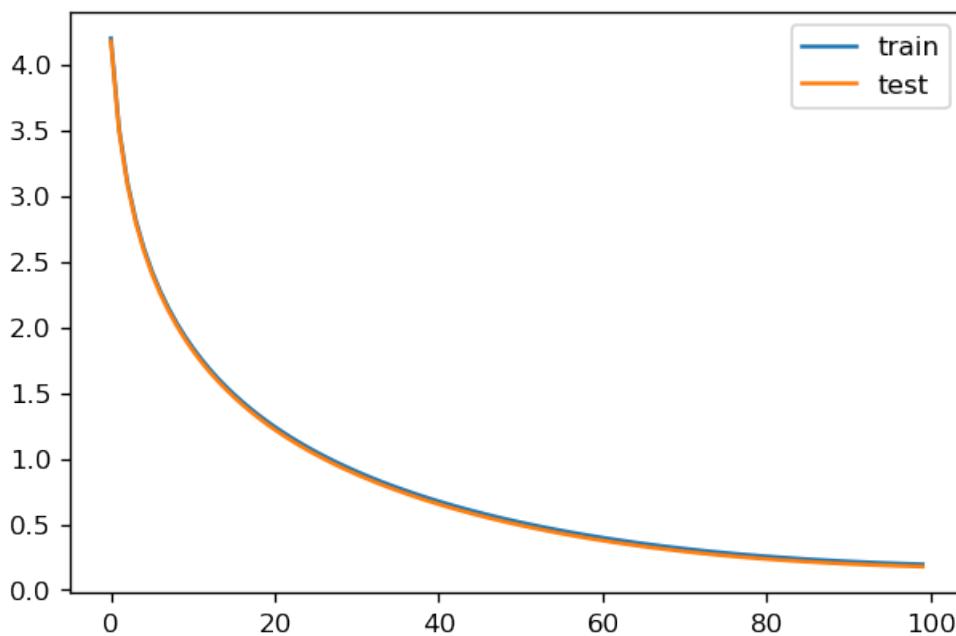
Test loss: 0.201359

Epoch 96, train loss: 0.201752

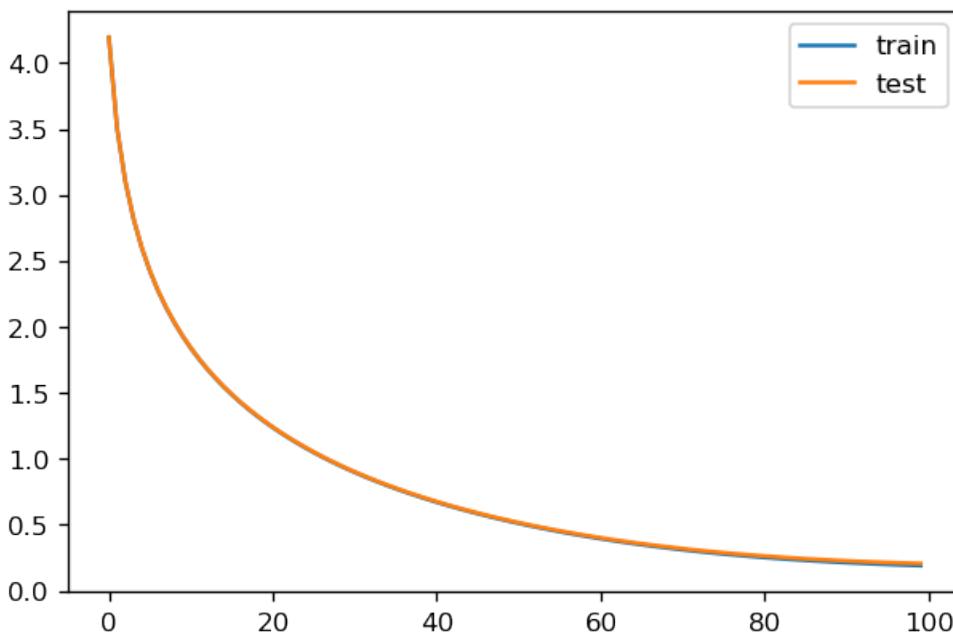
Epoch 97, train loss: 0.199572

Epoch 98, train loss: 0.197566

Epoch 99, train loss: 0.195623



```
Test loss: 0.178445
Epoch 96, train loss: 0.196776
Epoch 97, train loss: 0.194612
Epoch 98, train loss: 0.192524
Epoch 99, train loss: 0.190538
```



```
Test loss: 0.206442
5-fold validation: Avg train loss: 0.193388, Avg test loss: 0.197409
```

即便训练误差可以达到很低（调好参数之后），但是 K 折交叉验证上的误差可能更高。当训练误差特别低时，要观察 K 折交叉验证上的误差是否同时降低并小心过拟合。我们通常依赖 K 折交叉验证误差结果来调节参数。

3.13.7 预测并在 Kaggle 提交预测结果（选学）

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

我们首先定义预测函数。

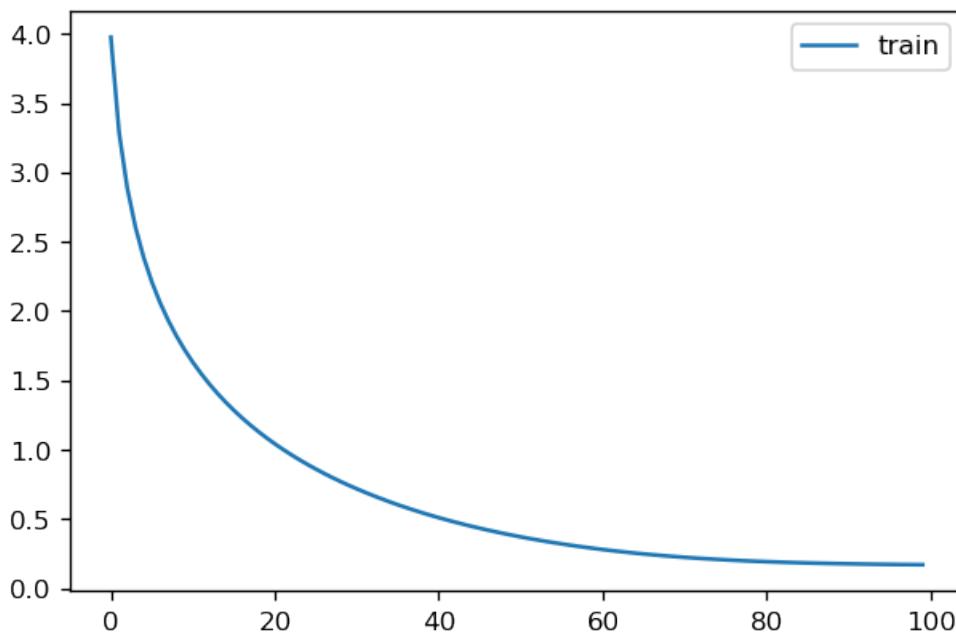
```
In [17]: def learn(epochs, verbose_epoch, X_train, y_train, test, learning_rate,
                  weight_decay):
    net = get_net()
    train(net, X_train, y_train, None, None, epochs, verbose_epoch,
          learning_rate, weight_decay)
    preds = net(X_test).asnumpy()
    test['SalePrice']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

```
submission = pd.concat([test['Id'], test['SalePrice']], axis=1)
submission.to_csv('submission.csv', index=False)
```

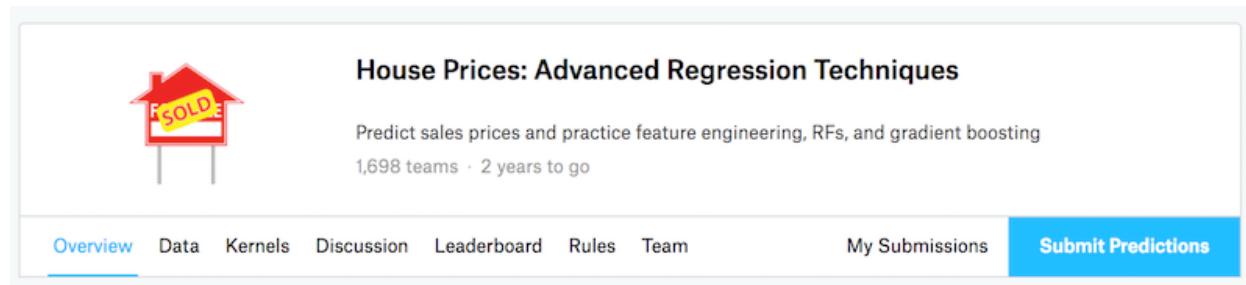
调好参数以后，下面我们预测并在 Kaggle 提交预测结果。

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

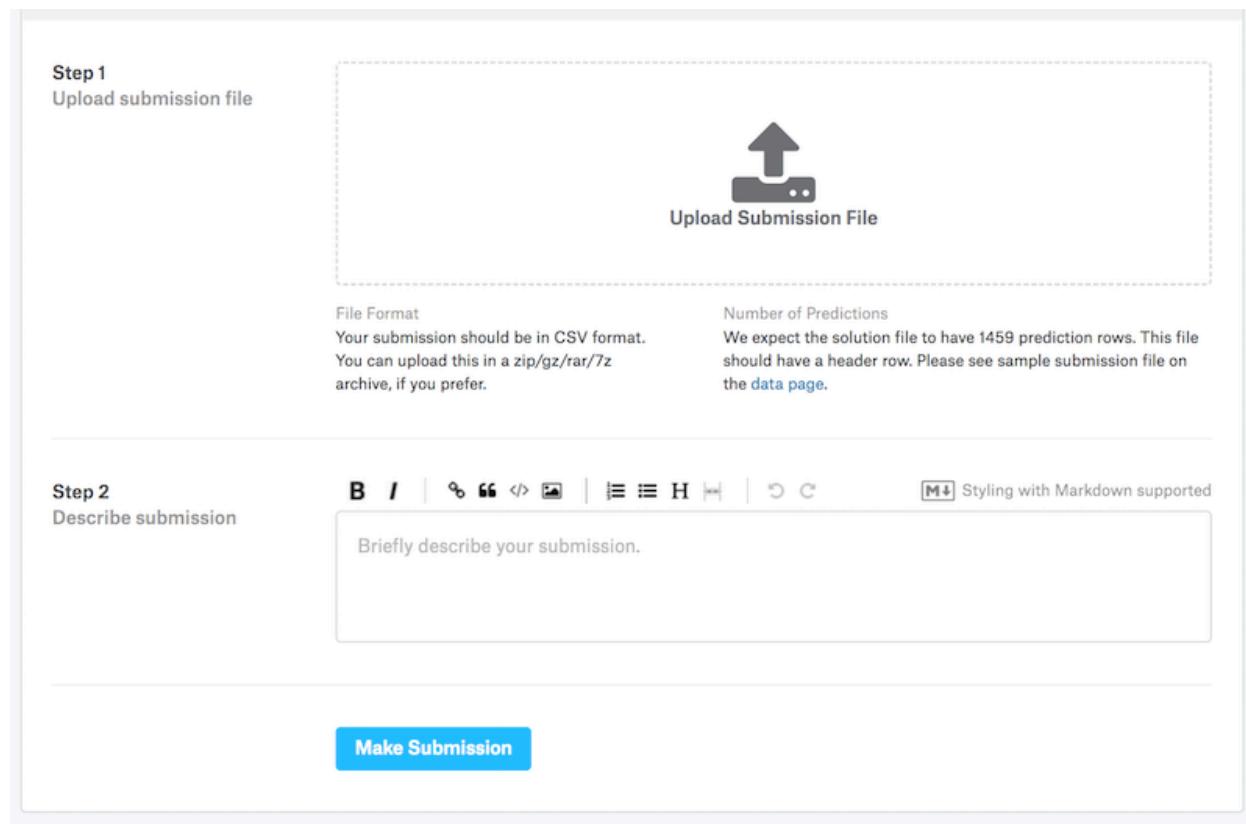
```
Epoch 96, train loss: 0.171113
Epoch 97, train loss: 0.170514
Epoch 98, train loss: 0.169978
Epoch 99, train loss: 0.169498
```



执行完上述代码后，会生成一个 `submission.csv` 文件。这是 Kaggle 要求的提交格式。这时我们可以在 Kaggle 上把我们预测得出的结果提交并查看与测试数据集上真实房价的误差。你需要登录 Kaggle 网站，打开[房价预测问题地址](#)，并点击下方右侧 **Submit Predictions** 按钮提交。



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



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

3.13.8 作业（汇报作业和查看其他小伙伴作业）：

- 运行本教程，目前的模型在 5 折交叉验证上可以拿到什么样的 loss？
- 如果网络条件允许，在 Kaggle 提交本教程的预测结果。观察一下，这个结果能在 Kaggle 上拿到什么样的 loss？
- 通过重新设计模型、调参并对照 K 折交叉验证结果，新模型是否比其他小伙伴的更好？除了调参，你可能发现我们之前学过的以下内容有些帮助：
 - 多层感知机—使用 *Gluon*
 - 正则化—使用 *Gluon*
- 如果不使用对数值特征做标准化处理能拿到什么样的 loss？
- 你还有什么其他办法可以继续改进模型？小伙伴们都期待学习到你独特的富有创造力的解决方案。

吐槽和讨论欢迎点[这里](#)

GLUON 基础

4.1 创建神经网络

前面的教程我们教了大家如何实现线性回归，多类 Logistic 回归和多层次感知机。我们既展示了如何从 0 开始实现，也提供使用 gluon 的更紧凑的实现。因为前面我们主要关注在模型本身，所以只解释了如何使用 gluon，但没说明他们是如何工作的。我们使用了 `nn.Sequential`，它是 `nn.Block` 的一个简单形式，但没有深入了解它们。

本教程和接下来几个教程，我们将详细解释如何使用这两个类来定义神经网络、初始化参数、以及保存和读取模型。

我们重新把多层次感知机—使用 *Gluon* 里的网络定义搬到这里作为开始的例子（为了简单起见，这里我们丢掉了 Flatten 层）。

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

        net = nn.Sequential()
        with net.name_scope():
            net.add(nn.Dense(256, activation="relu"))
            net.add(nn.Dense(10))

        print(net)
Sequential(
    (0): Dense(None -> 256, Activation(relu))
    (1): Dense(None -> 10, linear)
)
```

4.1.1 使用 nn.Block 来定义

事实上, nn.Sequential 是 nn.Block 的简单形式。我们先来看下如何使用 nn.Block 来实现同样的网络。

```
In [2]: class MLP(nn.Block):
    def __init__(self, **kwargs):
        super(MLP, self).__init__(**kwargs)
        with self.name_scope():
            self.dense0 = nn.Dense(256)
            self.dense1 = nn.Dense(10)

    def forward(self, x):
        return self.dense1(nd.relu(self.dense0(x)))
```

可以看到 nn.Block 的使用是通过创建一个它子类的类, 其中至少包含了两个函数。

- `__init__`: 创建参数。上面例子我们使用了包含了参数的 `dense` 层
- `forward()`: 定义网络的计算

我们所创建的类的使用跟前面 `net` 没有太多不一样。

```
In [3]: net2 = MLP()
print(net2)
net2.initialize()
x = nd.random.uniform(shape=(4,20))
y = net2(x)
y
```

MLP(
 (dense0): Dense(None -> 256, linear)
 (dense1): Dense(None -> 10, linear)
)

Out[3]:

```
[[ 0.05502447  0.01093244 -0.05812225 -0.00867474  0.00780752 -0.03732029
-0.11888048 -0.01667178 -0.12706244 -0.00605519]
[ 0.05254333 -0.03761618 -0.03303654 -0.06370584  0.02936437 -0.04790818
-0.07402188  0.00388384 -0.09476319  0.00247342]
[ 0.03847572 -0.01801044 -0.02936447 -0.04202728  0.00755377 -0.06616984
-0.08015118  0.04540668 -0.08034274  0.00180145]
[ 0.03042224 -0.04749024 -0.00121015 -0.08124933  0.03479041 -0.06163511
-0.10677548  0.04019741 -0.1076465   0.01437488]]
<NDArray 4x10 @cpu(0)>
```

In [4]: nn.Dense

Out[4]: mxnet.gluon.nn.basic_layers.Dense

如何定义创建和使用 `nn.Dense` 比较好理解。接下来我们仔细看下 `MLP` 里面用的其他命令：

- `super(MLP, self).__init__(**kwargs)`: 这句话调用 `nn.Block` 的 `__init__` 函数，它提供了 `prefix`（指定名字）和 `params`（指定模型参数）两个参数。我们会之后详细解释如何使用。
- `self.name_scope()`: 调用 `nn.Block` 提供的 `name_scope()` 函数。`nn.Dense` 的定义放在这个 `scope` 里面。它的作用是给里面的所有层和参数的名字加上前缀（prefix）使得他们在系统里面独一无二。默认自动会自动生成前缀，我们也可以在创建的时候手动指定。推荐在构建网络时，每个层至少在一个 `name_scope()` 里。

In [5]: `print('default prefix:', net2.dense0.name)`

```
net3 = MLP(prefix='another_mlp_')
print('customized prefix:', net3.dense0.name)

default prefix: mlp0_dense0
customized prefix: another_mlp_dense0
```

大家会发现这里并没有定义如何求导，或者是 `backward()` 函数。事实上，系统会使用 `autograd` 对 `forward()` 自动生成对应的 `backward()` 函数。

4.1.2 nn.Block 到底是什么东西？

在 `gluon` 里，`nn.Block` 是一个一般化的部件。整个神经网络可以是一个 `nn.Block`，单个层也是一个 `nn.Block`。我们可以（近似）无限地嵌套 `nn.Block` 来构建新的 `nn.Block`。

`nn.Block` 主要提供这个东西

1. 存储参数
2. 描述 `forward` 如何执行
3. 自动求导

4.1.3 那么现在可以解释 nn.Sequential 了吧

`nn.Sequential` 是一个 `nn.Block` 容器，它通过 `add` 来添加 `nn.Block`。它自动生成 `forward()` 函数，其就是把加进来的 `nn.Block` 逐一运行。

一个简单的实现是这样的：

```
In [6]: class Sequential(nn.Block):
    def __init__(self, **kwargs):
        super(Sequential, self).__init__(**kwargs)
    def add(self, block):
        self._children.append(block)
    def forward(self, x):
        for block in self._children:
            x = block(x)
        return x
```

可以跟 `nn.Sequential` 一样的使用这个自定义的类：

```
In [7]: net4 = Sequential()
with net4.name_scope():
    net4.add(nn.Dense(256, activation="relu"))
    net4.add(nn.Dense(10))

net4.initialize()
y = net4(x)
y
```

Out[7]:

```
[[ -0.05634359  0.09217402  0.06786803  0.00810092  0.00316704 -0.06578711
   0.02175836  0.00841999  0.0647321   0.01264806]
 [-0.0608877   0.06674264  0.08634251  0.06163288 -0.01288303 -0.01728502
  -0.00963083  0.0280523   0.02129908  0.05371749]
 [-0.04579362  0.11277001  0.0501334   0.01711009 -0.00263513 -0.04143213
   0.01833685  0.02963726  0.05529994  0.01901205]
 [-0.09248228  0.1179922   0.08974072  0.02259768  0.01704468 -0.07296751
   0.02300572  0.038479   0.05917452  0.03611853]]
<NDArray 4x10 @cpu(0)>
```

可以看到，`nn.Sequential` 的主要好处是定义网络起来更加简单。但 `nn.Block` 可以提供更加灵活的网络定义。考虑下面这个例子

```
In [8]: class FancyMLP(nn.Block):
    def __init__(self, **kwargs):
        super(FancyMLP, self).__init__(**kwargs)
        with self.name_scope():
            self.dense = nn.Dense(256)
            self.weight = nd.random_uniform(shape=(256, 20))

    def forward(self, x):
        x = nd.relu(self.dense(x))
```

```

x = nd.relu(nd.dot(x, self.weight)+1)
x = nd.relu(self.dense(x))
return x

```

看到这里我们直接手动创建和初始化了权重 `weight`, 并重复用了 `dense` 的层。测试一下:

```
In [9]: fancy_mlp = FancyMLP()
fancy_mlp.initialize()
y = fancy_mlp(x)
print(y.shape)

(4, 256)
```

4.1.4 nn.Block 和 nn.Sequential 的嵌套使用

现在我们知道 `nn` 下面的类基本都是 `nn.Block` 的子类, 他们可以很方便地嵌套使用。

```

In [10]: class RecMLP(nn.Block):
    def __init__(self, **kwargs):
        super(RecMLP, self).__init__(**kwargs)
        self.net = nn.Sequential()
        with self.name_scope():
            self.net.add(nn.Dense(256, activation="relu"))
            self.net.add(nn.Dense(128, activation="relu"))
            self.dense = nn.Dense(64)

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

rec_mlp = nn.Sequential()
rec_mlp.add(RecMLP())
rec_mlp.add(nn.Dense(10))
print(rec_mlp)

Sequential(
(0): RecMLP(
(net): Sequential(
(0): Dense(None -> 256, Activation(relu))
(1): Dense(None -> 128, Activation(relu))
)
(dense): Dense(None -> 64, linear)
)
(1): Dense(None -> 10, linear)
)
```

4.1.5 总结

不知道你同不同意，通过 `nn.Block` 来定义神经网络跟玩积木很类似。

4.1.6 练习

如果把 RecMLP 改成 `self.denses = [nn.Dense(256), nn.Dense(128), nn.Dense(64)]`, `forward` 就用 for loop 来实现，会有什么问题吗？

吐槽和讨论欢迎点[这里](#)

4.2 初始化模型参数

我们仍然用 MLP 这个例子来详细解释如何初始化模型参数。

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

        def get_net():
            net = nn.Sequential()
            with net.name_scope():
                net.add(nn.Dense(4, activation="relu"))
                net.add(nn.Dense(2))
            return net

        x = nd.random.uniform(shape=(3,5))
```

我们知道如果不 `initialize()` 直接跑 `forward`, 那么系统会抱怨说参数没有初始化。

```
In [2]: import sys
        try:
            net = get_net()
            net(x)
        except RuntimeError as err:
            sys.stderr.write(str(err))
```

Parameter sequential0_dense0_weight has not been initialized. Note that you should initialize

正确的打开方式是这样

```
In [3]: net.initialize()
        net(x)
```

Out[3]:

```
[[ 5.69327455e-03  5.31650949e-05]
 [ 3.97902681e-03  7.88165082e-04]
 [ 3.20330053e-03  1.50499865e-03]]
<NDArray 3x2 @cpu(0)>
```

4.2.1 访问模型参数

之前我们提到过可以通过 `weight` 和 `bias` 访问 `Dense` 的参数，他们是 `Parameter` 这个类：

```
In [4]: w = net[0].weight
b = net[0].bias
print('name: ', net[0].name, '\nweight: ', w, '\nbias: ', b)

name: sequential0_dense0
weight: Parameter sequential0_dense0_weight (shape=(4, 5), dtype=<class 'numpy.float32'>)
bias: Parameter sequential0_dense0_bias (shape=(4,), dtype=<class 'numpy.float32'>)
```

然后我们可以通过 `data` 来访问参数，`grad` 来访问对应的梯度

```
In [5]: print('weight:', w.data())
        print('weight gradient', w.grad())
        print('bias:', b.data())
        print('bias gradient', b.grad())

weight:
[[ 0.01847461 -0.03004881 -0.02461551 -0.01465906 -0.05932271]
 [-0.0595007   0.0434817   0.04195441  0.05774786  0.00482907]
 [ 0.04922146  0.0243923  -0.06268584  0.04367422  0.03679534]
 [-0.06364554  0.03010933  0.05611894 -0.02152951  0.03825361]]
<NDArray 4x5 @cpu(0)>
weight gradient
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
<NDArray 4x5 @cpu(0)>
bias:
[ 0.  0.  0.  0.]
<NDArray 4 @cpu(0)>
bias gradient
[ 0.  0.  0.  0.]
<NDArray 4 @cpu(0)>
```

我们也可以通过 `collect_params` 来访问 Block 里面所有的参数(这个会包括所有的子 Block)。它会返回一个名字到对应 Parameter 的 dict。既可以用正常 [] 来访问参数, 也可以用 `get()`, 它不需要填写名字的前缀。

```
In [6]: params = net.collect_params()
        print(params)
        print(params['sequential0_dense0_bias'].data())
        print(params.get('dense0_weight').data())

sequential0_ (
    Parameter sequential0_dense0_weight (shape=(4, 5), dtype=<class 'numpy.float32'>)
    Parameter sequential0_dense0_bias (shape=(4,), dtype=<class 'numpy.float32'>)
    Parameter sequential0_dense1_weight (shape=(2, 4), dtype=<class 'numpy.float32'>)
    Parameter sequential0_dense1_bias (shape=(2,), dtype=<class 'numpy.float32'>)
)

[ 0.  0.  0.  0.]
<NDArray 4 @cpu(0)>

[[ 0.01847461 -0.03004881 -0.02461551 -0.01465906 -0.05932271]
 [-0.0595007   0.0434817   0.04195441   0.05774786   0.00482907]
 [ 0.04922146   0.0243923   -0.06268584   0.04367422   0.03679534]
 [-0.06364554   0.03010933   0.05611894 -0.02152951   0.03825361]]
<NDArray 4x5 @cpu(0)>
```

4.2.2 使用不同的初始函数来初始化

我们一直在使用默认的 `initialize` 来初始化权重 (除了指定 GPU ctx 外)。它会把所有权重初始化成在 $[-0.07, 0.07]$ 之间均匀分布的随机数。我们可以使用别的初始化方法。例如使用均值为 0, 方差为 0.02 的正态分布

```
In [7]: from mxnet import init
        params.initialize(init=init.Normal(sigma=0.02), force_reinit=True)
        print(net[0].weight.data(), net[0].bias.data())

[[ 0.01925707  0.00956226  0.0084907   0.03457717 -0.00640247]
 [ 0.03445733  0.04135119 -0.01956459  0.0094062  -0.03433602]
 [ 0.00600536  0.00486903  0.00139216  0.03815848  0.02499754]
 [-0.00589869 -0.03443903 -0.0393823  -0.0066522  -0.01360089]]
<NDArray 4x5 @cpu(0)>
[ 0.  0.  0.  0.]
<NDArray 4 @cpu(0)>
```

看得更加清楚点：

```
In [8]: params.initialize(init=init.One(), force_reinit=True)
print(net[0].weight.data(), net[0].bias.data())
```

```
[[ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]
 [ 1.  1.  1.  1.  1.]]
<NDArray 4x5 @cpu(0)>
[ 0.  0.  0.  0.]
<NDArray 4 @cpu(0)>
```

更多的方法参见init 的 API.

4.2.3 延后的初始化

我们之前提到过 Gluon 的一个便利的地方是模型定义的时候不需要指定输入的大小，在之后做 forward 的时候会自动推测参数的大小。我们具体来看这是怎么工作的。

新创建一个网络，然后打印参数。你会发现两个全连接层的权重的形状里都有 0。这是因为在不知道输入数据的情况下，我们无法判断它们的形状。

```
In [9]: net = get_net()
net.collect_params()
```

```
Out[9]: sequential1_
Parameter sequential1_dense0_weight (shape=(4, 0), dtype=<class 'numpy.float32'>)
Parameter sequential1_dense0_bias (shape=(4,), dtype=<class 'numpy.float32'>)
Parameter sequential1_dense1_weight (shape=(2, 0), dtype=<class 'numpy.float32'>)
Parameter sequential1_dense1_bias (shape=(2,), dtype=<class 'numpy.float32'>)
)
```

然后我们初始化

```
In [10]: net.initialize()
net.collect_params()
```

```
Out[10]: sequential1_
Parameter sequential1_dense0_weight (shape=(4, 0), dtype=<class 'numpy.float32'>)
Parameter sequential1_dense0_bias (shape=(4,), dtype=<class 'numpy.float32'>)
Parameter sequential1_dense1_weight (shape=(2, 0), dtype=<class 'numpy.float32'>)
Parameter sequential1_dense1_bias (shape=(2,), dtype=<class 'numpy.float32'>)
)
```

你会看到我们形状并没有发生变化，这是因为我们仍然不能确定权重形状。真正的初始化发生在我们看到数据时。

```
In [11]: net(x)
    net.collect_params()

Out[11]: sequential1_
    Parameter sequential1_dense0_weight (shape=(4, 5), dtype=<class 'numpy.float32'>
    Parameter sequential1_dense0_bias (shape=(4,), dtype=<class 'numpy.float32'>)
    Parameter sequential1_dense1_weight (shape=(2, 4), dtype=<class 'numpy.float32'>
    Parameter sequential1_dense1_bias (shape=(2,), dtype=<class 'numpy.float32'>)
)
```

这时候我们看到 shape 里面的 0 被填上正确的值了。

4.2.4 共享模型参数

有时候我们想在层之间共享同一份参数，我们可以通过 Block 的 `params` 输出参数来手动指定参数，而不是让系统自动生成。

```
In [12]: net = nn.Sequential()
    with net.name_scope():
        net.add(nn.Dense(4, activation="relu"))
        net.add(nn.Dense(4, activation="relu"))
        net.add(nn.Dense(4, activation="relu", params=net[-1].params))
        net.add(nn.Dense(2))
```

初始化然后打印

```
In [13]: net.initialize()
net(x)
print(net[1].weight.data())
print(net[2].weight.data())

[[ 0.00146846  0.06708457  0.00377706 -0.02985603]
 [ 0.03104883 -0.05449805 -0.06617871  0.02707522]
 [-0.0626184   0.06622557 -0.03636756  0.06569055]
 [-0.05142071  0.04123941 -0.02823606 -0.05013531]]
<NDArray 4x4 @cpu(0)>

[[ 0.00146846  0.06708457  0.00377706 -0.02985603]
 [ 0.03104883 -0.05449805 -0.06617871  0.02707522]
 [-0.0626184   0.06622557 -0.03636756  0.06569055]
```

```
[-0.05142071  0.04123941 -0.02823606 -0.05013531]
<NDArray 4x4 @cpu(0)>
```

4.2.5 自定义初始化方法

下面我们自定义一个初始化方法。它通过重载 `_init_weight` 来实现不同的初始化方法。（注意到 Gluon 里面 `bias` 都是默认初始化成 0）

```
In [14]: class MyInit(init.Initializer):
    def __init__(self):
        super(MyInit, self).__init__()
        self._verbose = True
    def _init_weight(self, _, arr):
        # 初始化权重，使用 out=arr 后我们不需指定形状
        print('init weight', arr.shape)
        nd.random.uniform(low=5, high=10, out=arr)

    net = get_net()
    net.initialize(MyInit())
    net(x)
    net[0].weight.data()

init weight (4, 5)
init weight (2, 4)

Out[14]:
[[ 7.05669117  5.08102942  9.79977226  6.01421022  9.65556717]
 [ 5.97966576  5.52816963  7.14227676  8.70719051  7.83192253]
 [ 5.16814661  7.71718931  5.02550554  6.56668663  9.03970432]
 [ 5.18657732  8.61319733  7.32929277  9.92631054  8.41801071]]
<NDArray 4x5 @cpu(0)>
```

当然我们也可以通过 `Parameter.set_data` 来直接改写权重。注意到由于有延后初始化，所以我们通常可以通过调用一次 `net(x)` 来确定权重的形狀先。

```
In [15]: net = get_net()
net.initialize()
net(x)

print('default weight:', net[1].weight.data())

w = net[1].weight
w.set_data(nd.ones(w.shape))
```

```
print('init to all 1s:', net[1].weight.data())
default weight:
[[-0.01560705  0.01619106 -0.04681858 -0.0483556 ]
 [ 0.05755728  0.0293365   0.05690721  0.06266605]]
<NDArray 2x4 @cpu(0)>
init to all 1s:
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
<NDArray 2x4 @cpu(0)>
```

4.2.6 总结

我们可以很灵活地访问和修改模型参数。

4.2.7 练习

1. 研究下 `net.collect_params()` 返回的是什么? `net.params` 呢?
2. 如何对每个层使用不同的初始化函数
3. 如果两个层共用一个参数, 那么求梯度的时候会发生什么?

吐槽和讨论欢迎点[这里](#)

4.3 序列化—读写模型

我们现在已经讲了很多, 包括

- 如何处理数据
- 如何构建模型
- 如何在数据上训练模型
- 如何使用不同的损失函数来做分类和回归

但即使知道了所有这些, 我们还没有完全准备好来构建一个真正的机器学习系统。这是因为我们还没有讲如何读和写模型。因为现实中, 我们通常在一个地方训练好模型, 然后部署到很多不同的地方。我们需要把内存中的训练好的模型存在硬盘上好下次使用。

4.3.1 读写 NDArrays

作为开始, 我们先看看如何读写 NDArray。虽然我们可以使用 Python 的序列化包例如 Pickle, 不过我们更倾向直接 save 和 load, 通常这样更快, 而且别的语言, 例如 R 和 Scala 也能用到。

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

x = nd.ones(3)
y = nd.zeros(4)
filename = "../data/test1.params"
nd.save(filename, [x, y])
```

读回来

```
In [2]: a, b = nd.load(filename)
print(a, b)
```

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

不仅可以读写单个 NDArray, NDArray list, dict 也是可以的:

```
In [3]: mydict = {"x": x, "y": y}
filename = "../data/test2.params"
nd.save(filename, mydict)
```

```
In [4]: c = nd.load(filename)
print(c)
```

```
'x':[ 1.  1.  1.]<NDArray 3 @cpu(0)>, 'y':[ 0.  0.  0.  0.]<NDArray 4 @cpu(0)>
```

4.3.2 读写 Gluon 模型的参数

跟 NDArray 类似, Gluon 的模型 (就是 nn.Block) 提供便利的 save_params 和 load_params 函数来读写数据。我们同前一样创建一个简单的多层感知机

```
In [5]: from mxnet.gluon import nn

def get_net():
    net = nn.Sequential()
    with net.name_scope():
        net.add(nn.Dense(10, activation="relu"))
```

```
net.add(nn.Dense(2))
return net

net = get_net()
net.initialize()
x = nd.random.uniform(shape=(2,10))
print(net(x))

[[ 0.00205935 -0.00979935]
 [ 0.00107034 -0.00423382]]
<NDArray 2x2 @cpu(0)>
```

下面我们把模型参数存起来

```
In [6]: filename = "../data/mlp.params"
net.save_params(filename)
```

之后我们构建一个一样的多层感知机，但不像前面那样随机初始化，我们直接读取前面的模型参数。这样给定同样的输入，新的模型应该会输出同样的结果。

```
In [7]: import mxnet as mx
net2 = get_net()
net2.load_params(filename, mx.cpu()) # FIXME, gluon will support default ctx later
print(net2(x))
```

```
[[ 0.00205935 -0.00979935]
 [ 0.00107034 -0.00423382]]
<NDArray 2x2 @cpu(0)>
```

4.3.3 总结

通过 `load_params` 和 `save_params` 可以很方便的读写模型参数。

吐槽和讨论欢迎点[这里](#)

4.4 设计自定义层

神经网络的一个魅力是它有大量的层，例如全连接、卷积、循环、激活，和各式花样的连接方式。我们之前学到了如何使用 Gluon 提供的层来构建新的层 (`nn.Block`) 继而得到神经网络。虽然 Gluon 提供了大量的[层的定义](#)，但我们仍然会遇到现有层不够用的情况。

这时候的一个自然的想法是，我们不是学习了如何只使用基础数值运算包 `NDArray` 来实现各种的模型吗？它提供了大量的底层计算函数足以实现即使不是 100% 那也是 95% 的神经网络吧。

但每次都从头写容易写到怀疑人生。实际上，即使在纯研究的领域里，我们也很少发现纯新的东西，大部分时候是在现有模型的基础上做一些改进。所以很可能大部分是可以沿用前面的而只有一部分是需要自己来实现。

这个教程我们将介绍如何使用底层的 `NDArray` 接口来实现一个 `Gluon` 的层，从而可以以后被重复调用。

4.4.1 定义一个简单的层

我们先来看如何定义一个简单层，它不需要维护模型参数。事实上这个跟前面介绍的如何使用 `nn.Block` 没什么区别。下面代码定义一个层将输入减掉均值。

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

        class CenteredLayer(nn.Block):
            def __init__(self, **kwargs):
                super(CenteredLayer, self).__init__(**kwargs)

            def forward(self, x):
                return x - x.mean()
```

我们可以马上实例化这个层用起来。

```
In [2]: layer = CenteredLayer()
        layer(nd.array([1,2,3,4,5]))
```

Out[2]:

```
[-2. -1.  0.  1.  2.]
<NDArray 5 @cpu(0)>
```

我们也可以用它来构造更复杂的神经网络：

```
In [3]: net = nn.Sequential()
        with net.name_scope():
            net.add(nn.Dense(128))
            net.add(nn.Dense(10))
            net.add(CenteredLayer())
```

确认下输出的均值确实是 0：

```
In [4]: net.initialize()
y = net(nd.random.uniform(shape=(4, 8)))
y.mean()
```

```
Out[4]:
[ 2.32830647e-11]
<NDArray 1 @cpu(0)>
```

当然大部分情况你可以看不到一个实实在在的 0，而是一个很小的数。例如 $5.82076609e-11$ 。这是因为 MXNet 默认使用 32 位 float，会带来一定的浮点精度误差。

4.4.2 带模型参数的自定义层

虽然 CenteredLayer 可能会告诉实现自定义层大概是什么样子，但它缺少了重要的一块，就是它没有可以学习的模型参数。

记得我们之前访问 Dense 的权重的时候是通过 `dense.weight.data()`，这里 `weight` 是一个 `Parameter` 的类型。我们可以显示的构建这样的一个参数。

```
In [5]: from mxnet import gluon
my_param = gluon.Parameter("exciting_parameter_yay", shape=(3,3))
```

这里我们创建一个 3×3 大小的参数并取名为” exciting_parameter_yay”。然后用默认方法初始化打印结果。

```
In [6]: my_param.initialize()
(my_param.data(), my_param.grad())
```

```
Out[6]: (
[[ 0.02332029  0.04696382  0.03078182]
 [ 0.00755873  0.03193929 -0.0059346 ]
 [-0.00809445  0.01710822 -0.03057443]]
<NDArray 3x3 @cpu(0)>,
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
<NDArray 3x3 @cpu(0)>)
```

通常自定义层的时候我们不会直接创建 `Parameter`，而是用过 Block 自带的一个 `ParamterDict` 类型的成员变量 `params`，顾名思义，这是一个由字符串名字映射到 `Parameter` 的字典。

```
In [7]: pd = gluon.ParameterDict(prefix="block1_")
pd.get("exciting_parameter_yay", shape=(3,3))
pd
```

```
Out[7]: block1_
    Parameter block1_exciting_parameter_yay (shape=(3, 3), dtype=<class 'numpy.float32'>
)
```

现在我们看下如果如果实现一个跟 Dense 一样功能的层, 它概念跟前面的 CenteredLayer 的主要区别是我们在初始函数里通过 params 创建了参数:

```
In [8]: class MyDense(nn.Block):
    def __init__(self, units, in_units, **kwargs):
        super(MyDense, self).__init__(**kwargs)
        with self.name_scope():
            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)
```

我们创建实例化一个对象来看下它的参数, 这里我们特意加了前缀 prefix, 这是 nn.Block 初始化函数自带的参数。

```
In [9]: dense = MyDense(5, in_units=10, prefix='o_my_dense_')
dense.params
```

```
Out[9]: o_my_dense_
    Parameter o_my_dense_weight (shape=(10, 5), dtype=<class 'numpy.float32'>)
    Parameter o_my_dense_bias (shape=(5,), dtype=<class 'numpy.float32'>
)
```

它的使用跟前面没有什么不一致:

```
In [10]: dense.initialize()
dense(nd.random.uniform(shape=(2,10)))
```

```
Out[10]:
[[ 0.           0.           0.05519049  0.01345633  0.07244172]
 [ 0.           0.           0.06741175  0.01634707  0.0257601 ]]
<NDArray 2x5 @cpu(0)>
```

我们构造的层跟 Gluon 提供的层用起来没太多区别:

```
In [11]: net = nn.Sequential()
with net.name_scope():
    net.add(MyDense(32, in_units=64))
    net.add(MyDense(2, in_units=32))
```

```
net.initialize()  
net(nd.random.uniform(shape=(2, 64)))  
  
Out[11]:  
[[ 0.          0.          ]  
 [ 0.02434103  0.          ]]  
<NDArray 2x2 @cpu(0)>
```

仔细的你可能还是注意到了，我们这里指定了输入的大小，而 Gluon 自带的 `Dense` 则无需如此。我们已经在前面节介绍过了这个延迟初始化如何使用。但如果实现一个这样的层我们将留到后面介绍了 `hybridize` 后。

4.4.3 总结

现在我们知道了如何把前面手写过的层全部包装了 Gluon 能用的 Block，之后再用到的时候就可以飞起来了！

4.4.4 练习

1. 怎么修改自定义层里参数的默认初始化函数。
2. (这个比较难)，在一个代码 Cell 里面输入 “`nn.Dense??`”，看看它是怎么实现的。为什么它就可以支持延迟初始化了。

吐槽和讨论欢迎点[这里](#)

4.5 使用 GPU 来计算

【注意】运行本教程需要 GPU。没有 GPU 的同学可以大致理解下内容，至少是 `context` 这个概念，因为之后我们也会用到。但没有 GPU 不会影响运行之后的大部分教程（好吧，还是有点点，可能运行会稍微慢点）。

前面的教程里我们一直在使用 CPU 来计算，因为绝大部分的计算设备都有 CPU。但 CPU 的设计目的是处理通用的计算，例如打开浏览器和运行 Jupyter，它一般只有少数的一块区域复杂数值计算，例如 `nd.dot(A, B)`。对于复杂的神经网络和大规模的数据来说，单块 CPU 可能不够给力。

常用的解决办法是要么使用多台机器来协同计算，要么使用数值计算更加强劲的硬件，或者两者一起使用。本教程关注使用单块 Nvidia GPU 来加速计算，更多的选项例如多 GPU 和多机器计算则留到后面。

首先需要确保至少有一块 Nvidia 显卡已经安装好了，然后下载安装显卡驱动和CUDA（推荐下载 8.0，CUDA 自带了驱动）。完成后应该可以通过 `nvidia-smi` 查看显卡信息了。（Windows 用户需要设一下 PATH: `set PATH=C:\Program Files\NVIDIA Corporation\NVSMI;%PATH%`）。

```
In [1]: !nvidia-smi
```

Sun Nov 12 01:22:07 2017

```
| NVIDIA-SMI 375.26                    Driver Version: 375.26
+-----+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A | Volatile Uncorr. ECC
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M.
+-----+-----+-----+-----+
|   0  Tesla M60        Off  | 0000:00:1D.0    Off  |                  0
| N/A   32C     P0    37W / 150W |      0MiB /  7612MiB |      0%     Default
+-----+-----+-----+
|   1  Tesla M60        Off  | 0000:00:1E.0    Off  |                  0
| N/A   36C     P0    38W / 150W |      0MiB /  7612MiB |     99%     Default
+-----+-----+
+
+
| Processes:                               GPU Memory
| GPU  PID  Type  Process name          Usage
+-----+
| No running processes found
+-----+
```

接下来要确认正确安装了的 mxnet 的 GPU 版本。具体来说是卸载了 mxnet(pip uninstall mxnet)，然后根据 CUDA 版本安装 mxnet-cu75 或者 mxnet-cu80 (例如 pip install -pre mxnet-cu80)。

使用 pip 来确认下：

```
In [2]: import pip
        for pkg in ['mxnet', 'mxnet-cu75', 'mxnet-cu80']:
            pip.main(['show', pkg])
```

Name: mxnet-cu80

Version: 0.12.1b20171109

Summary: MXNet is an ultra-~~fast~~

Home page: <http://>

Author: UNKNOWN

License: Apache 2.0

Location: /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages

Requires: numpy, requests, graphviz

4.5.1 Context

MXNet 使用 Context 来指定使用哪个设备来存储和计算。默认会将数据放在主内存，然后利用 CPU 来计算，这个由 `mx.cpu()` 来表示。GPU 则由 `mx.gpu()` 来表示。注意 `mx.cpu()` 表示所有的物理 CPU 和内存，意味着计算上会尽量使用多有的 CPU 核。但 `mx.gpu()` 只代表一块显卡和其对应的显卡内存。如果有两块 GPU，我们用 `mx.gpu(i)` 来表示第 i 块 GPU (i 从 0 开始)。

```
In [3]: import mxnet as mx  
[mx.cpu(), mx.gpu(), mx.gpu(1)]
```

```
Out[3]: [cpu(0), gpu(0), gpu(1)]
```

4.5.2 NDArray 的 GPU 计算

每个 NDArray 都有一个 `context` 属性来表示它存在哪个设备上，默认会是 `cpu`。这是为什么前面每次我们打印 NDArray 的时候都会看到 `@cpu(0)` 这个标识。

```
In [4]: from mxnet import nd  
x = nd.array([1,2,3])  
x.context
```

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

GPU 上创建内存

我们可以在创建的时候指定创建在哪个设备上（如果 GPU 不能用或者没有装 MXNet GPU 版本，这里会有 error）：

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

```
b = nd.zeros((3,2), ctx=mx.gpu())
```

```
c = nd.random.uniform(shape=(2,3), ctx=mx.gpu())  
(a,b,c)
```

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

```
[ 0.  0.]
[ 0.  0.]]
<NDArray 3x2 @gpu(0)>,
[[ 0.32977498  0.43025011  0.70026755]
 [ 0.77781075  0.29912937  0.39169419]]
<NDArray 2x3 @gpu(0)>
```

尝试将内存开到另外一块 GPU 上。如果不存在会报错。当然，如果你有大于 10 块 GPU，那么下面代码会顺利执行。

In [6]: `import sys`

```
try:
    nd.array([1,2,3], ctx=mx.gpu(10))
except mx.MXNetError as err:
    sys.stderr.write(str(err))
```

[01:22:13] src/storage/storage.cc:59: Check failed: e == cudaSuccess || e == cudaErrorCuda

Stack trace returned 10 entries:

```
[bt] (0) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/mxnet/
[bt] (1) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/mxnet/
[bt] (2) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/mxnet/
[bt] (3) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/mxnet/
[bt] (4) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/mxnet/
[bt] (5) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/lib-dynload/.../...
[bt] (6) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/lib-dynload/.../...
[bt] (7) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/lib-dynload/_ctypes.
[bt] (8) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/lib-dynload/_ctypes.
[bt] (9) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/bin/python(_PyObject_FastCallDict+...
```

我们可以通过 `copyto` 和 `as_in_context` 来在设备直接传输数据。

In [7]: `y = x.copyto(mx.gpu())
z = x.as_in_context(mx.gpu())
(y, z)`

Out[7]: (
[1. 2. 3.]
<NDArray 3 @gpu(0)>,
[1. 2. 3.]
<NDArray 3 @gpu(0)>)

这两个函数的主要区别是，如果源和目标的 context 一致，`as_in_context` 不复制，而 `copyto` 总是会新建内存：

```
In [8]: yy = y.as_in_context(mx.gpu())
        zz = z.copyto(mx.gpu())
        (yy is y, zz is z)
```

```
Out[8]: (True, False)
```

GPU 上的计算

计算会在数据的 `context` 上执行。所以为了使用 GPU，我们只需要事先将数据放在上面就行了。结果会自动保存在对应的设备上：

```
In [9]: nd.exp(z + 2) * y
```

```
Out[9]:
```

```
[ 20.08553696 109.19629669 445.23950195]
<NDArray 3 @gpu(0)>
```

注意所有计算要求输入数据在同一个设备上。不一致的时候系统不进行自动复制。这个设计的目的是因为设备之间的数据交互通常比较昂贵，我们希望用户确切的知道数据放在哪里，而不是隐藏这个细节。下面代码尝试将 CPU 上 `x` 和 GPU 上的 `y` 做运算。

```
In [10]: try:
```

```
    x + y
except mx.MXNetError as err:
    sys.stderr.write(str(err))
```

```
[01:22:14] src/imperative./imperative_utils.h:55: Check failed: inputs[i]->ctx().dev_mask
```

```
Stack trace returned 10 entries:
```

```
[bt] (0) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/mxnet/
[bt] (1) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/mxnet/
[bt] (2) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/mxnet/
[bt] (3) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/mxnet/
[bt] (4) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages/mxnet/
[bt] (5) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/lib-dynload/.../...
[bt] (6) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/lib-dynload/.../...
[bt] (7) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/lib-dynload/_ctypes.
[bt] (8) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/lib-dynload/_ctypes.
[bt] (9) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/bin/python(_PyObject_FastCallDict+...
```

默认会复制回 CPU 的操作

如果某个操作需要将 NDArray 里面的内容转出来, 例如打印或变成 numpy 格式, 如果需要的话系统都会自动将数据 copy 到主内存。

```
In [11]: print(y)
    print(y.astype('float'))
    print(y.sum().asscalar())
```

```
[ 1.  2.  3.]
<NDArray 3 @gpu(0)>
[ 1.  2.  3.]
6.0
```

4.5.3 Gluon 的 GPU 计算

同 NDArray 类似, Gluon 的大部分函数可以通过 ctx 指定设备。下面代码将模型参数初始化在 GPU 上:

```
In [12]: from mxnet import gluon
    net = gluon.nn.Sequential()
    net.add(gluon.nn.Dense(1))

    net.initialize(ctx=mx.gpu())
```

输入 GPU 上的数据, 会在 GPU 上计算结果

```
In [13]: data = nd.random.uniform(shape=[3,2], ctx=mx.gpu())
    net(data)
```

Out[13]:

```
[[ 0.07357398]
 [ 0.00532937]
 [ 0.05488062]]
<NDArray 3x1 @gpu(0)>
```

确认下权重:

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

Out[14]:

```
[[ 0.04118239  0.05352169]]
<NDArray 1x2 @gpu(0)>
```

4.5.4 总结

通过 `context` 我们可以很容易在不同的设备上计算。

4.5.5 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？
- 试试 CPU 和 GPU 之间传递数据的速度
- GPU 上如何读写模型呢？

吐槽和讨论欢迎点[这里](#)

卷积神经网络

5.1 卷积神经网络—从 0 开始

之前的教程里，在输入神经网络前我们将输入图片直接转成了向量。这样做有两个不好的地方：

- 在图片里相近的像素在向量表示里可能很远，从而模型很难捕获他们的空间关系。
- 对于大图片输入，模型可能会很大。例如输入是 $256 \times 256 \times 3$ 的照片（仍然远比手机拍的小），输入层是 1000，那么这一层的模型大小是将近 1GB.

这一节我们介绍卷积神经网络，其有效解决了上述两个问题。

5.1.1 卷积神经网络

卷积神经网络是指主要由卷积层构成的神经网络。

卷积层

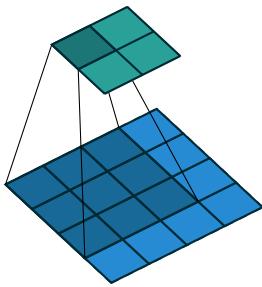
卷积层跟前面的全连接层类似，但输入和权重不是做简单的矩阵乘法，而是使用每次作用在一个窗口上的卷积。下图演示了输入是一个 4×4 矩阵，使用一个 3×3 的权重，计算得到 2×2 结果的过程。每次我们采样一个跟权重一样大小的窗口，让它跟权重做按元素的乘法然后相加。通常我们也是用卷积的术语把这个权重叫 kernel 或者 filter。

(图片版权属于 vduimoulin@github)

我们使用 `nd.Convolution` 来演示这个。

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

```
# 输入输出数据格式是 batch x channel x height x width, 这里 batch 和 channel 都是 1  
# 权重格式是 output_channels x in_channels x height x width, 这里 input_filter 和 out
```



```
w = nd.arange(4).reshape((1,1,2,2))
b = nd.array([1])
data = nd.arange(9).reshape((1,1,3,3))
out = nd.Convolution(data, w, b, kernel=w.shape[2:], num_filter=w.shape[1])

print('input:', data, '\n\nweight:', w, '\n\nbias:', b, '\n\noutput:', out)

input:
[[[[ 0.  1.  2.]
   [ 3.  4.  5.]
   [ 6.  7.  8.]]]]
<NDArray 1x1x3x3 @cpu(0)>

weight:
[[[[ 0.  1.]
   [ 2.  3.]]]]
<NDArray 1x1x2x2 @cpu(0)>

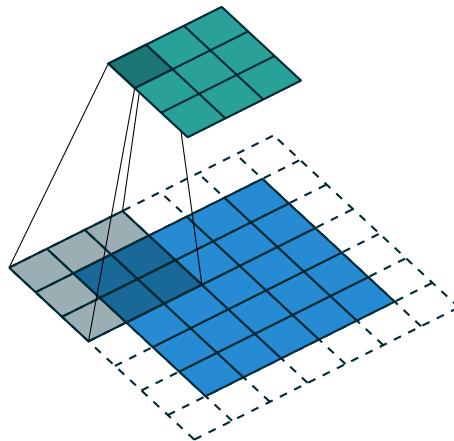
bias:
[ 1.]
<NDArray 1 @cpu(0)>

output:
[[[[ 20.  26.]
   [ 38.  44.]]]]
<NDArray 1x1x2x2 @cpu(0)>
```

我们可以控制如何移动窗口，和在边缘的时候如何填充窗口。下图演示了 `stride=2` 和 `pad=1`。

```
In [2]: out = nd.Convolution(data, w, b, kernel=w.shape[2:], num_filter=w.shape[1],
                           stride=(2,2), pad=(1,1))

print('input:', data, '\n\nweight:', w, '\n\nbias:', b, '\n\noutput:', out)
```



```
input:  
[[[[ 0.  1.  2.  
   [ 3.  4.  5.  
   [ 6.  7.  8.]]]]  
<NDArray 1x1x3x3 @cpu(0)>
```

```
weight:  
[[[[ 0.  1.  
   [ 2.  3.]]]]  
<NDArray 1x1x2x2 @cpu(0)>
```

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

```
output:  
[[[[ 1.  9.  
   [ 22.  44.]]]]  
<NDArray 1x1x2x2 @cpu(0)>
```

当输入数据有多个通道的时候，每个通道会有对应的权重，然后会对每个通道做卷积之后在通道之间求和

$$\text{conv}(\text{data}, w, b) = \sum_i \text{conv}(\text{data}[:, i, :, :], w[:, i, :, :], b)$$

```
In [3]: w = nd.arange(8).reshape((1,2,2,2))  
       data = nd.arange(18).reshape((1,2,3,3))  
  
       out = nd.Convolution(data, w, b, kernel=w.shape[2:], num_filter=w.shape[0])
```

```
print('input:', data, '\n\nweight:', w, '\n\nbias:', b, '\n\noutput:', out)

input:
[[[ [ 0.   1.   2.]
  [ 3.   4.   5.]
  [ 6.   7.   8.]]

 [[ 9.  10.  11.]
  [ 12.  13.  14.]
  [ 15.  16.  17.]]]]
<NDArray 1x2x3x3 @cpu(0)>

weight:
[[[ [ 0.  1.]
  [ 2.  3.]]

 [[ 4.  5.]
  [ 6.  7.]]]]
<NDArray 1x2x2x2 @cpu(0)>

bias:
[ 1.]
<NDArray 1 @cpu(0)>

output:
[[[ [ 269.  297.]
  [ 353.  381.]]]]
<NDArray 1x1x2x2 @cpu(0)>
```

当输出需要多通道时，每个输出通道有对应权重，然后每个通道上做卷积。

$$\text{conv}(\text{data}, \text{w}, \text{b})[:, i, :, :] = \text{conv}(\text{data}, \text{w}[i, :, :, :], \text{b}[i])$$

```
In [4]: w = nd.arange(16).reshape((2,2,2,2))
        data = nd.arange(18).reshape((1,2,3,3))
        b = nd.array([1,2])

        out = nd.Convolution(data, w, b, kernel=w.shape[2:], num_filter=w.shape[0])

        print('input:', data, '\n\nweight:', w, '\n\nbias:', b, '\n\noutput:', out)

input:
[[[ [ 0.   1.   2.]
  [ 3.   4.   5.]
  [ 6.   7.   8.]]

 [[ 9.  10.  11.]
  [ 12.  13.  14.]
  [ 15.  16.  17.]]]]
<NDArray 1x2x3x3 @cpu(0)>

weight:
[[[ [ 0.  1.]
  [ 2.  3.]]

 [[ 4.  5.]
  [ 6.  7.]]]]
<NDArray 1x2x2x2 @cpu(0)>

bias:
[ 1.]
<NDArray 1 @cpu(0)>

output:
[[[ [ 269.  297.]
  [ 353.  381.]]]]
<NDArray 1x1x2x2 @cpu(0)>
```

```
[ 3.  4.  5.]
[ 6.  7.  8.]]]

[[ 9. 10. 11.]
 [12. 13. 14.]
 [15. 16. 17.]]]]
<NDArray 1x2x3x3 @cpu(0)>

weight:
[[[[ 0.  1.]
   [ 2.  3.]]

[[ 4.  5.]
 [ 6.  7.]]]

[[[ 8.  9.]
   [10. 11.]]

[[ 12. 13.]
 [ 14. 15.]]]]
<NDArray 2x2x2x2 @cpu(0)>

bias:
[ 1.  2.]
<NDArray 2 @cpu(0)>

output:
[[[[ 269. 297.]
   [ 353. 381.]]

[[ 686. 778.]
 [ 962. 1054.]]]]
<NDArray 1x2x2x2 @cpu(0)>
```

池化层 (pooling)

因为卷积层每次作用在一个窗口，它对位置很敏感。池化层能够很好的缓解这个问题。它跟卷积类似每次看一个小窗口，然后选出窗口里面最大的元素，或者平均元素作为输出。

```
In [5]: data = nd.arange(18).reshape((1,2,3,3))

        max_pool = nd.Pooling(data=data, pool_type="max", kernel=(2,2))
        avg_pool = nd.Pooling(data=data, pool_type="avg", kernel=(2,2))

        print('data:', data, '\n\nmax pooling:', max_pool, '\n\navg pooling:', avg_pool)

data:
[[[[ 0.   1.   2.]
   [ 3.   4.   5.]
   [ 6.   7.   8.]]

 [[ 9.  10.  11.]
   [12.  13.  14.]
   [15.  16.  17.]]]]
<NDArray 1x2x3x3 @cpu(0)>

max pooling:
[[[[ 4.   5.]
   [ 7.   8.]]

 [[13.  14.]
   [16.  17.]]]]
<NDArray 1x2x2x2 @cpu(0)>

avg pooling:
[[[[ 2.   3.]
   [ 5.   6.]]

 [[11.  12.]
   [14.  15.]]]]
<NDArray 1x2x2x2 @cpu(0)>
```

下面我们可以开始使用这些层构建模型了。

5.1.2 获取数据

我们继续使用 FashionMNIST (希望你还没有彻底厌烦这个数据)

```
In [6]: import sys
        sys.path.append('..')
        from utils import load_data_fashion_mnist
```

```
batch_size = 256
train_data, test_data = load_data_fashion_mnist(batch_size)
```

5.1.3 定义模型

因为卷积网络计算比全连接要复杂，这里我们默认使用 GPU 来计算。如果 GPU 不能用，默认使用 CPU。（下面这段代码会保存在 `utils.py` 里可以下次重复使用）。

In [7]: `import mxnet as mx`

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

Out[7]: `gpu(0)`

我们使用 MNIST 常用的 LeNet，它有两个卷积层，之后是两个全连接层。注意到我们将权重全部创建在 `ctx` 上：

In [8]: `weight_scale = .01`

```
# output channels = 20, kernel = (5,5)
W1 = nd.random_normal(shape=(20, 1, 5, 5), scale=weight_scale, ctx=ctx)
b1 = nd.zeros(W1.shape[0], ctx=ctx)

# output channels = 50, kernel = (3,3)
W2 = nd.random_normal(shape=(50, 20, 3, 3), scale=weight_scale, ctx=ctx)
b2 = nd.zeros(W2.shape[0], ctx=ctx)

# output dim = 128
W3 = nd.random_normal(shape=(1250, 128), scale=weight_scale, ctx=ctx)
b3 = nd.zeros(W3.shape[1], ctx=ctx)

# output dim = 10
W4 = nd.random_normal(shape=(W3.shape[1], 10), scale=weight_scale, ctx=ctx)
b4 = nd.zeros(W4.shape[1], ctx=ctx)

params = [W1, b1, W2, b2, W3, b3, W4, b4]
```

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

卷积模块通常是“卷积层-激活层-池化层”。然后转成 2D 矩阵输出给后面的全连接层。

```
In [9]: def net(X, verbose=False):  
    X = X.as_in_context(W1.context)  
    # 第一层卷积  
    h1_conv = nd.Convolution(  
        data=X, weight=W1, bias=b1, kernel=W1.shape[2:], num_filter=W1.shape[0])  
    h1_activation = nd.relu(h1_conv)  
    h1 = nd.Pooling(  
        data=h1_activation, pool_type="max", kernel=(2,2), stride=(2,2))  
    # 第二层卷积  
    h2_conv = nd.Convolution(  
        data=h1, weight=W2, bias=b2, kernel=W2.shape[2:], num_filter=W2.shape[0])  
    h2_activation = nd.relu(h2_conv)  
    h2 = nd.Pooling(data=h2_activation, pool_type="max", kernel=(2,2), stride=(2,2))  
    h2 = nd.flatten(h2)  
    # 第一层全连接  
    h3_linear = nd.dot(h2, W3) + b3  
    h3 = nd.relu(h3_linear)  
    # 第二层全连接  
    h4_linear = nd.dot(h3, W4) + b4  
    if verbose:  
        print('1st conv block:', h1.shape)  
        print('2nd conv block:', h2.shape)  
        print('1st dense:', h3.shape)  
        print('2nd dense:', h4_linear.shape)  
        print('output:', h4_linear)  
    return h4_linear
```

测试一下，输出中间结果形状（当然可以直接打印结果）和最终结果。

```
In [10]: for data, _ in train_data:  
    net(data, verbose=True)  
    break  
  
1st conv block: (256, 20, 12, 12)  
2nd conv block: (256, 1250)  
1st dense: (256, 128)  
2nd dense: (256, 10)  
output:  
[[ -5.35991057e-05  3.23458589e-05  3.11347030e-05 ... ,  1.17605079e-04
```

```

 6.05099485e-05  1.98460893e-05]
[-1.53809931e-04  4.58294016e-06  7.64853976e-05 ... ,  8.06436292e-05
 1.11623835e-04  1.14007104e-04]
[-1.15204668e-04  8.78433821e-06  1.08375534e-04 ... ,  5.46803603e-05
 8.33992308e-05  1.22317811e-04]
...
[-1.17477517e-04  2.10452927e-05  3.06760739e-05 ... ,  1.76697999e-04
 9.49604291e-05  5.15630672e-05]
[-1.08620290e-04 -2.54856222e-05  1.12748887e-04 ... ,  7.36226066e-05
 9.61711412e-05  1.38394316e-04]
[-1.57616843e-04  2.93551284e-06  8.39915665e-05 ... ,  1.01519989e-04
 7.64592332e-05  8.15607928e-05]]
<NDArray 256x10 @gpu(0)>

```

5.1.4 训练

跟前面没有什么不同的，除了这里我们使用 `as_in_context` 将 `data` 和 `label` 都放置在需要的设备上。（下面这段代码也将保存在 `utils.py` 里方便之后使用）。

```

In [11]: from mxnet import autograd as autograd
        from utils import SGD, accuracy, evaluate_accuracy
        from mxnet import gluon

softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()

learning_rate = .2

for epoch in range(5):
    train_loss = 0.
    train_acc = 0.
    for data, label in train_data:
        label = label.as_in_context(ctx)
        with autograd.record():
            output = net(data)
            loss = softmax_cross_entropy(output, label)
            loss.backward()
            SGD(params, learning_rate/batch_size)

        train_loss += nd.mean(loss).asscalar()
        train_acc += accuracy(output, label)

```

```
test_acc = evaluate_accuracy(test_data, net, ctx)
print("Epoch %d. Loss: %f, Train acc %f, Test acc %f" % (
    epoch, train_loss/len(train_data),
    train_acc/len(train_data), test_acc))

Epoch 0. Loss: 2.301834, Train acc 0.107138, Test acc 0.130809
Epoch 1. Loss: 1.266440, Train acc 0.513138, Test acc 0.698117
Epoch 2. Loss: 0.665806, Train acc 0.742137, Test acc 0.787059
Epoch 3. Loss: 0.523436, Train acc 0.799796, Test acc 0.820513
Epoch 4. Loss: 0.457698, Train acc 0.829310, Test acc 0.841747
```

5.1.5 结论

可以看到卷积神经网络比前面的多层感知机分类精度更好。事实上，如果你看懂了这一章，那你基本知道了计算视觉里最重要的几个想法。LeNet 早在 90 年代就提出来了。不管你相信不相信，如果你 5 年前懂了这个而且开了家公司，那么你很可能现在已经把公司作价几千万卖个某大公司了。幸运的是，或者不幸的是，现在的算法已经更加高级些了，接下来我们会看到一些更加新的想法。

5.1.6 练习

- 试试改改卷积层设定，例如 filter 数量，kernel 大小
- 试试把池化层从 max 改到 avg
- 如果你有 GPU，那么尝试用 CPU 来跑一下看看
- 你可能注意到比前面的多层感知机慢了很多，那么尝试计算下这两个模型分别需要多少浮点计算。例如 $n \times m$ 和 $m \times k$ 的矩阵乘法需要浮点运算 $2nmk$ 。

吐槽和讨论欢迎点[这里](#)

5.2 卷积神经网络—使用 Gluon

现在我们使用 Gluon 来实现上一章的卷积神经网络。

5.2.1 定义模型

下面是 LeNet 在 Gluon 里的实现，注意到我们不再需要实现去计算每层的输入大小，尤其是接在卷积后面的那个全连接层。

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

    net = nn.Sequential()
    with net.name_scope():
        net.add(
            nn.Conv2D(channels=20, kernel_size=5, activation='relu'),
            nn.MaxPool2D(pool_size=2, strides=2),
            nn.Conv2D(channels=50, kernel_size=3, activation='relu'),
            nn.MaxPool2D(pool_size=2, strides=2),
            nn.Flatten(),
            nn.Dense(128, activation="relu"),
            nn.Dense(10)
        )
```

5.2.2 获取数据和训练

剩下的跟上一章没什么不同，我们重用 `utils.py` 里定义的函数。

```
In [2]: from mxnet import gluon
import sys
sys.path.append('..')
import utils

# 初始化
ctx = utils.try_gpu()
net.initialize(ctx=ctx)
print('initialize weight on', ctx)

# 获取数据
batch_size = 256
train_data, test_data = utils.load_data_fashion_mnist(batch_size)

# 训练
loss = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(),
                        'sgd', {'learning_rate': 0.5})
utils.train(train_data, test_data, net, loss,
            trainer, ctx, num_epochs=5)

initialize weight on gpu(0)
Start training on gpu(0)
Epoch 0. Loss: 1.017, Train acc 0.62, Test acc 0.80, Time 2.4 sec
```

```
Epoch 1. Loss: 0.452, Train acc 0.83, Test acc 0.86, Time 2.2 sec
Epoch 2. Loss: 0.372, Train acc 0.86, Test acc 0.87, Time 2.1 sec
Epoch 3. Loss: 0.337, Train acc 0.87, Test acc 0.88, Time 2.1 sec
Epoch 4. Loss: 0.308, Train acc 0.89, Test acc 0.89, Time 2.1 sec
```

5.2.3 结论

使用 Gluon 来实现卷积网络轻松加随意。

5.2.4 练习

再试试改改卷积层设定，是不是会比上一章容易很多？

吐槽和讨论欢迎点[这里](#)

5.3 批量归一化—从 0 开始

在Kaggle 实战我们输入数据做了归一化。在实际应用中，我们通常将输入数据的每个样本或者每个特征进行归一化，就是将均值变为 0 方差变为 1，来使得数值更稳定。

这个对我们在之前的课程里学过了线性回归和逻辑回归很有效。因为输入层的输入值的大小变化不剧烈，那么输入也不会。但是，对于一个可能有很多层的深度学习模型来说，情况可能会比较复杂。

举个例子，随着第一层和第二层的参数在训练时不断变化，第三层所使用的激活函数的输入值可能由于乘法效应而变得极大或极小，例如和第一层所使用的激活函数的输入值不在一个数量级上。这种在训练时可能出现的情况会造成模型训练的不稳定性。例如，给定一个学习率，某次参数迭代后，目标函数值会剧烈变化或甚至升高。数学的解释是，如果把目标函数 f 根据参数 \mathbf{w} 迭代（如 $f(\mathbf{w} - \eta \nabla f(\mathbf{w}))$ ）进行泰勒展开，有关学习率 η 的高阶项的系数可能由于数量级的原因（通常由于层数多）而不容忽略。然而常用的低阶优化算法（如梯度下降）对于不断降低目标函数的有效性通常基于一个基本假设：在以上泰勒展开中把有关学习率的高阶项通通忽略不计。

为了应对上述这种情况，Sergey Ioffe 和 Christian Szegedy 在 2015 年提出了批量归一化的方法。简而言之，在训练时给定一个批量输入，批量归一化试图对深度学习模型的某一层所使用的激活函数的输入进行归一化：使批量呈标准正态分布（均值为 0，标准差为 1）。

批量归一化通常应用于输入层或任意中间层。

5.3.1 简化的批量归一化层

给定一个批量 $B = \{x_{1,\dots,m}\}$, 我们需要学习拉升参数 γ 和偏移参数 β 。

我们定义:

$$\begin{aligned}\mu_B &\leftarrow \frac{1}{m} \sum_{i=1}^m x_i \\ \sigma_B^2 &\leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \\ \hat{x}_i &\leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\ y_i &\leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)\end{aligned}$$

批量归一化层的输出是 $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$ 。

我们现在来动手实现一个简化的批量归一化层。实现时对全连接层和二维卷积层两种情况做了区分。对于全连接层, 很明显我们要对每个批量进行归一化。然而这里需要注意的是, 对于二维卷积, 我们要对每个通道进行归一化, 并需要保持四维形状使得可以正确地广播。

```
In [1]: from mxnet import nd
def pure_batch_norm(X, gamma, beta, eps=1e-5):
    assert len(X.shape) in (2, 4)
    # 全连接: batch_size x feature
    if len(X.shape) == 2:
        # 每个输入维度在样本上的平均和方差
        mean = X.mean(axis=0)
        variance = ((X - mean)**2).mean(axis=0)
    # 2D 卷积: batch_size x channel x height x width
    else:
        # 对每个通道算均值和方差, 需要保持 4D 形状使得可以正确地广播
        mean = X.mean(axis=(0, 2, 3), keepdims=True)
        variance = ((X - mean)**2).mean(axis=(0, 2, 3), keepdims=True)

    # 均一化
    X_hat = (X - mean) / nd.sqrt(variance + eps)
    # 拉升和偏移
    return gamma.reshape(mean.shape) * X_hat + beta.reshape(mean.shape)
```

下面我们检查一下。我们先定义全连接层的输入是这样的。每一行是批量中的一个实例。

```
In [2]: A = nd.arange(6).reshape((3, 2))
A
```

Out[2]:

```
[[ 0.  1.]  
 [ 2.  3.]  
 [ 4.  5.]]  
<NDArray 3x2 @cpu(0)>
```

我们希望批量中的每一列都被归一化。结果符合预期。

In [3]: `pure_batch_norm(A, gamma=nd.array([1,1]), beta=nd.array([0,0]))`

Out[3]:

```
[[ -1.22474265 -1.22474265]  
 [ 0.          0.          ]  
 [ 1.22474265  1.22474265]]  
<NDArray 3x2 @cpu(0)>
```

下面我们定义二维卷积网络层的输入是这样的。

In [4]: `B = nd.arange(18).reshape((1,2,3,3))`
B

Out[4]:

```
[[[[ 0.   1.   2.]  
   [ 3.   4.   5.]  
   [ 6.   7.   8.]]  
  
 [[ 9.  10.  11.]  
  [ 12.  13.  14.]  
  [ 15.  16.  17.]]]]  
<NDArray 1x2x3x3 @cpu(0)>
```

结果也如预期那样， 我们对每个通道做了归一化。

In [5]: `pure_batch_norm(B, gamma=nd.array([1,1]), beta=nd.array([0,0]))`

Out[5]:

```
[[[[[-1.54919219 -1.1618942  -0.7745961 ]  
    [-0.38729805  0.          0.38729805]  
    [ 0.7745961   1.1618942  1.54919219]]]  
  
 [[[-1.54919219 -1.1618942  -0.7745961 ]  
  [-0.38729805  0.          0.38729805]  
  [ 0.7745961   1.1618942  1.54919219]]]]  
<NDArray 1x2x3x3 @cpu(0)>
```

5.3.2 批量归一化层

你可能会想，既然训练时用了批量归一化，那么测试时也该用批量归一化吗？其实这个问题乍一想不是很好回答，因为：

- 不用的话，训练出的模型参数很可能在测试时就不准确了；
- 用的话，万一测试的数据就只有一个数据实例就不好办了。

事实上，在测试时我们还是需要继续使用批量归一化的，只是需要做些改动。在测试时，我们需要把原先训练时用到的批量均值和方差替换成整个训练数据的均值和方差。但是当训练数据极大时，这个计算开销很大。因此，我们用移动平均的方法来近似计算（参见实现中的 `moving_mean` 和 `moving_variance`）。

为了方便讨论批量归一化层的实现，我们先看下面这段代码来理解 Python 变量可以如何修改。

```
In [6]: def batch_norm(X, gamma, beta, is_training, moving_mean, moving_variance,
                  eps = 1e-5, moving_momentum = 0.9):
    assert len(X.shape) in (2, 4)
    # 全连接: batch_size x feature
    if len(X.shape) == 2:
        # 每个输入维度在样本上的平均和方差
        mean = X.mean(axis=0)
        variance = ((X - mean)**2).mean(axis=0)
    # 2D 卷积: batch_size x channel x height x width
    else:
        # 对每个通道算均值和方差，需要保持 4D 形状使得可以正确的广播
        mean = X.mean(axis=(0,2,3), keepdims=True)
        variance = ((X - mean)**2).mean(axis=(0,2,3), keepdims=True)
        # 变形使得可以正确的广播
        moving_mean = moving_mean.reshape(mean.shape)
        moving_variance = moving_variance.reshape(mean.shape)

    # 均一化
    if is_training:
        X_hat = (X - mean) / nd.sqrt(variance + eps)
        #!!! 更新全局的均值和方差
        moving_mean[:] = moving_momentum * moving_mean + (
            1.0 - moving_momentum) * mean
        moving_variance[:] = moving_momentum * moving_variance + (
            1.0 - moving_momentum) * variance
    else:
        #!!! 测试阶段使用全局的均值和方差
```

```
X_hat = (X - moving_mean) / nd.sqrt(moving_variance + eps)

# 拉升和偏移
return gamma.reshape(mean.shape) * X_hat + beta.reshape(mean.shape)
```

5.3.3 定义模型

我们尝试使用 GPU 运行本教程代码。

```
In [7]: import sys
       sys.path.append('..')
       import utils
       ctx = utils.try_gpu()
       ctx
```

```
Out[7]: gpu(0)
```

先定义参数。

```
In [8]: weight_scale = .01

# 输出通道 = 20, 卷积核 = (5,5)
c1 = 20
W1 = nd.random_normal(shape=(c1,1,5,5), scale=weight_scale, ctx=ctx)
b1 = nd.zeros(c1, ctx=ctx)

# 第 1 层批量归一化
gamma1 = nd.random_normal(shape=c1, scale=weight_scale, ctx=ctx)
beta1 = nd.random_normal(shape=c1, scale=weight_scale, ctx=ctx)
moving_mean1 = nd.zeros(c1, ctx=ctx)
moving_variance1 = nd.zeros(c1, ctx=ctx)

# 输出通道 = 50, 卷积核 = (3,3)
c2 = 50
W2 = nd.random_normal(shape=(c2,c1,3,3), scale=weight_scale, ctx=ctx)
b2 = nd.zeros(c2, ctx=ctx)

# 第 2 层批量归一化
gamma2 = nd.random_normal(shape=c2, scale=weight_scale, ctx=ctx)
beta2 = nd.random_normal(shape=c2, scale=weight_scale, ctx=ctx)
moving_mean2 = nd.zeros(c2, ctx=ctx)
moving_variance2 = nd.zeros(c2, ctx=ctx)
```

```

# 输出维度 = 128
o3 = 128
W3 = nd.random.normal(shape=(1250, o3), scale=weight_scale, ctx=ctx)
b3 = nd.zeros(o3, ctx=ctx)

# 输出维度 = 10
W4 = nd.random_normal(shape=(W3.shape[1], 10), scale=weight_scale, ctx=ctx)
b4 = nd.zeros(W4.shape[1], ctx=ctx)

# 注意这里 moving_* 是不需要更新的
params = [W1, b1, gamma1, beta1,
          W2, b2, gamma2, beta2,
          W3, b3, W4, b4]

for param in params:
    param.attach_grad()

```

下面定义模型。我们添加了批量归一化层。特别要注意我们添加的位置：在卷积层后，在激活函数前。

```

In [9]: def net(X, is_training=False, verbose=False):
    X = X.as_in_context(W1.context)
    # 第一层卷积
    h1_conv = nd.Convolution(
        data=X, weight=W1, bias=b1, kernel=W1.shape[2:], num_filter=c1)
    ### 添加了批量归一化层
    h1_bn = batch_norm(h1_conv, gamma1, beta1, is_training,
                        moving_mean1, moving_variance1)
    h1_activation = nd.relu(h1_bn)
    h1 = nd.Pooling(
        data=h1_activation, pool_type="max", kernel=(2,2), stride=(2,2))
    # 第二层卷积
    h2_conv = nd.Convolution(
        data=h1, weight=W2, bias=b2, kernel=W2.shape[2:], num_filter=c2)
    ### 添加了批量归一化层
    h2_bn = batch_norm(h2_conv, gamma2, beta2, is_training,
                        moving_mean2, moving_variance2)
    h2_activation = nd.relu(h2_bn)
    h2 = nd.Pooling(data=h2_activation, pool_type="max", kernel=(2,2), stride=(2,2))
    h2 = nd.flatten(h2)
    # 第一层全连接
    h3_linear = nd.dot(h2, W3) + b3

```

```
h3 = nd.relu(h3_linear)
# 第二层全连接
h4_linear = nd.dot(h3, W4) + b4
if verbose:
    print('1st conv block:', h1.shape)
    print('2nd conv block:', h2.shape)
    print('1st dense:', h3.shape)
    print('2nd dense:', h4_linear.shape)
    print('output:', h4_linear)
return h4_linear
```

下面我们训练并测试模型。

```
In [10]: from mxnet import autograd
from mxnet import gluon

batch_size = 256
train_data, test_data = utils.load_data_fashion_mnist(batch_size)

softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()

learning_rate = 0.2

for epoch in range(5):
    train_loss = 0.
    train_acc = 0.
    for data, label in train_data:
        label = label.as_in_context(ctx)
        with autograd.record():
            output = net(data, is_training=True)
            loss = softmax_cross_entropy(output, label)
        loss.backward()
        utils.SGD(params, learning_rate/batch_size)

        train_loss += nd.mean(loss).asscalar()
        train_acc += utils.accuracy(output, label)

    test_acc = utils.evaluate_accuracy(test_data, net, ctx)
    print("Epoch %d. Loss: %f, Train acc %f, Test acc %f" %
          (epoch, train_loss/len(train_data), train_acc/len(train_data), test_ac
```

Epoch 0. Loss: 1.982008, Train acc 0.236362, Test acc 0.651943

Epoch 1. Loss: 0.579539, Train acc 0.781150, Test acc 0.769531

```
Epoch 2. Loss: 0.407267, Train acc 0.847923, Test acc 0.866987
Epoch 3. Loss: 0.345340, Train acc 0.873197, Test acc 0.874599
Epoch 4. Loss: 0.311232, Train acc 0.885734, Test acc 0.886318
```

5.3.4 总结

相比卷积神经网络—从 0 开始来说，通过加入批量归一化层，即使是同样的参数，测试精度也有明显提升，尤其是最开始几轮。

5.3.5 练习

尝试调大学习率，看看跟前面比，是不是可以使用更大的学习率。

吐槽和讨论欢迎点[这里](#)

5.4 批量归一化—使用 Gluon

本章介绍如何使用 Gluon 在训练和测试深度学习模型中使用批量归一化。

5.4.1 定义模型并添加批量归一化层

有了 Gluon，我们模型的定义工作变得简单了许多。我们只需要添加 `nn.BatchNorm` 层并指定对二维卷积的通道 (`axis=1`) 进行批量归一化。

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

net = nn.Sequential()
with net.name_scope():
    # 第一层卷积
    net.add(nn.Conv2D(channels=20, kernel_size=5))
    ### 添加了批量归一化层
    net.add(nn.BatchNorm(axis=1))
    net.add(nn.Activation(activation='relu'))
    net.add(nn.MaxPool2D(pool_size=2, strides=2))
    # 第二层卷积
    net.add(nn.Conv2D(channels=50, kernel_size=3))
    ### 添加了批量归一化层
    net.add(nn.BatchNorm(axis=1))
    net.add(nn.Activation(activation='relu'))
```

```
net.add(nn.MaxPool2D(pool_size=2, strides=2))
net.add(nn.Flatten())
# 第一层全连接
net.add(nn.Dense(128, activation="relu"))
# 第二层全连接
net.add(nn.Dense(10))
```

5.4.2 模型训练

剩下的代码跟之前没什么不一样。

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

        ctx = utils.try_gpu()
        net.initialize(ctx=ctx)

        batch_size = 256
        train_data, test_data = utils.load_data_fashion_mnist(batch_size)

        softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.2})

        for epoch in range(5):
            train_loss = 0.
            train_acc = 0.
            for data, label in train_data:
                label = label.as_in_context(ctx)
                with autograd.record():
                    output = net(data.as_in_context(ctx))
                    loss = softmax_cross_entropy(output, label)
                    loss.backward()
                    trainer.step(batch_size)

                train_loss += nd.mean(loss).asscalar()
                train_acc += utils.accuracy(output, label)
            test_acc = utils.evaluate_accuracy(test_data, net, ctx)
            print("Epoch %d. Loss: %f, Train acc %f, Test acc %f" % (
```

```
epoch, train_loss/len(train_data),  
train_acc/len(train_data), test_acc))  
  
Epoch 0. Loss: 0.589046, Train acc 0.779614, Test acc 0.859876  
Epoch 1. Loss: 0.357116, Train acc 0.868523, Test acc 0.884615  
Epoch 2. Loss: 0.305139, Train acc 0.886919, Test acc 0.889423  
Epoch 3. Loss: 0.275576, Train acc 0.897486, Test acc 0.899539  
Epoch 4. Loss: 0.256699, Train acc 0.904864, Test acc 0.901442
```

5.4.3 总结

使用 Gluon 我们可以很轻松地添加批量归一化层。

5.4.4 练习

如果在全连接层添加批量归一化结果会怎么样？

吐槽和讨论欢迎点[这里](#)

5.5 深度卷积神经网络和 AlexNet

在前面的章节中，我们学会了如何使用卷积神经网络进行图像分类。其中我们使用了两个卷积层与池化层交替，加入一个全连接隐层，和一个归一化指数 Softmax 输出层。这个结构与 LeNet，一个以卷积神经网络先驱 Yann LeCun 命名的早期神经网络很相似。LeCun 也是将卷积神经网络付诸应用的第一人，通过反向传播来进行训练，这是一个当时相当新颖的想法。当时一小批对仿生的学习模型热衷的研究者通过人工模拟神经元来作为学习模型。然而即便时至今日，依然没有多少研究者相信真正的大脑是通过梯度下降来学习的，研究社区也探索了许多其他的学习理论。LeCun 在当时展现了，在识别手写数字的任务上通过梯度下降训练卷积神经网络可以达到最先进的结果。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。

然而，这之后几年里，神经网络被许多其他方法超越。神经网络训练慢，并且就深度神经网络从一个随机生成的权重起点开始训练是否可行，学界没有广泛达成一致。此外，十多年前还没有黄教主的核武器 GPU 通用计算，所以训练一个多通道，多层，大量参数的在十几年前难以实现。所以虽然 LeNet 可以在 MNIST 上得到好的成绩，在更大的真实世界的数据集上，神经网络还是在这个冬天里渐渐失宠。

取而代之，研究者们通过勤劳，智慧和黑魔法生成了许多手工特征。通常的模式是 1. 找个数据集 2. 用一堆已有的特征提取函数生成特征 3. 把这些特征表示放进一个简单的线性模型（当时认为的机器学习部分仅限这一步）

计算机视觉领域一直维持这个状况直到 2012 年，深度学习即将颠覆应用机器学习。一个小伙伴 (Zack) 2013 年研究生入学，他的一个朋友这样总结了当时的状况：如果你跟机器学习研究者们交谈，他们会认为机器学习既重要又优美。优雅的定理证明了许多分类器的性质。机器学习领域生机勃勃，严谨，而且极其有用。然而如果你跟一个计算机视觉研究者交谈，则是另外一幅景象。这人会告诉你“图像识别里”不可告人”的现实是，计算机视觉里的机器学习流水线中真正重要的是数据和特征。稍微干净点的数据集，或者略微好些的手调特征对最终准确度意味着天壤之别。反而分类器的选择对表现的区别影响不大。说到底，把特征扔进逻辑回归，支持向量机，或者其他任何分类器，表现都差不多。

5.5.1 学习特征表示

简单来说，给定一个数据集，当时流水线里最重要的是特征表示这步。并且直到 2012 年，特征表示这步都是基于硬拼出来的直觉，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，包括 Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari, Juergen Schmidhuber，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。在图片中，底层可以表示边，色彩和纹理。

高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。更高层可能可以表征整个物体，如人，飞机，狗，飞盘。最终，在分类器层前的隐含层可能会表征经过汇总的内容，其中不同的类别将会是线性可分的。然而许多年来，研究者们由于种种原因并不能实现这一愿景。

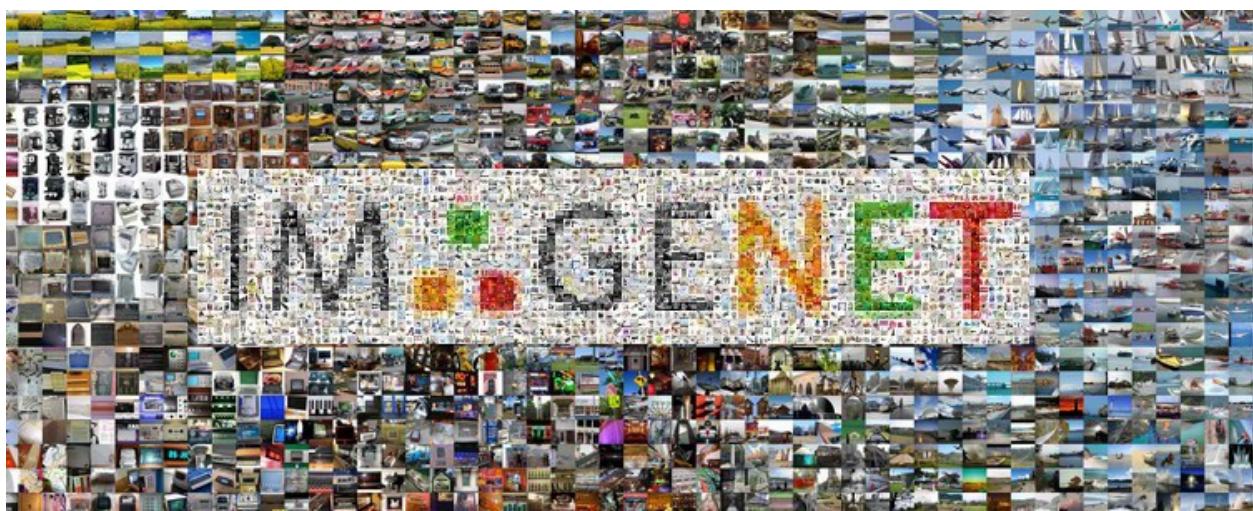
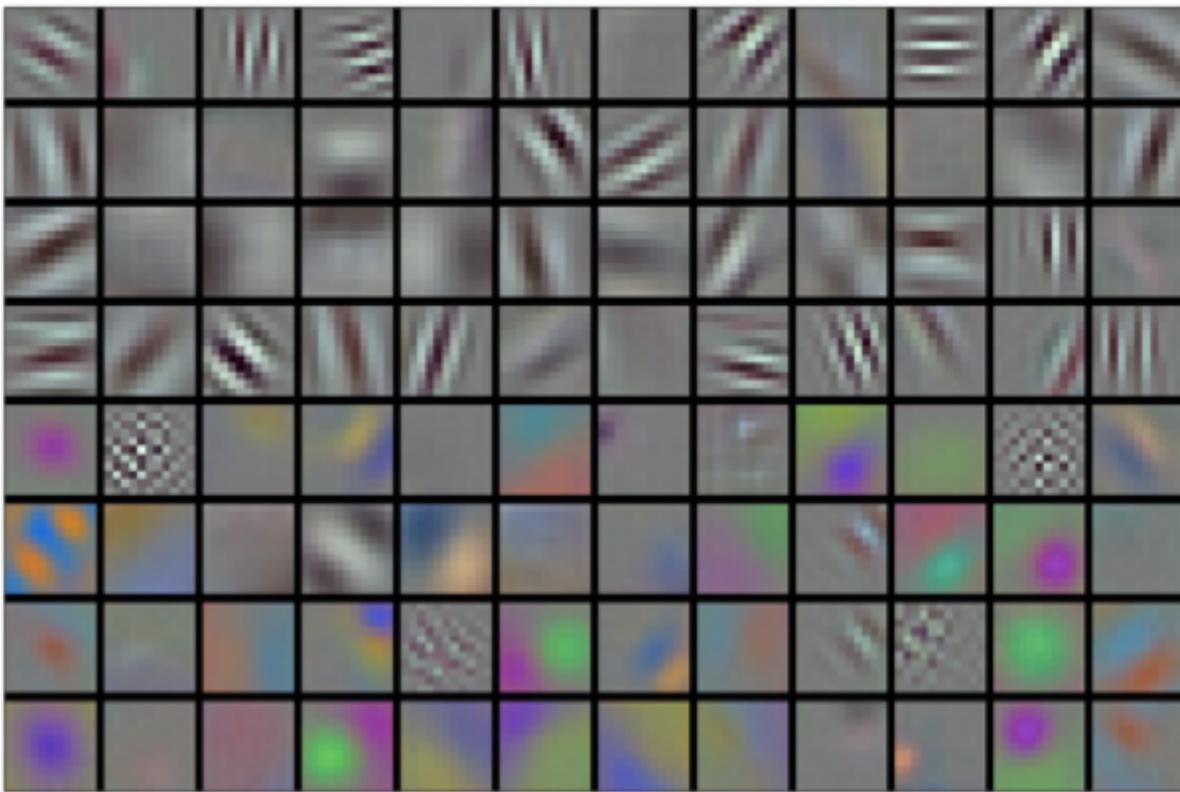
5.5.2 缺失要素 1：数据

尽管这群执着的研究者不断钻研，试图学习深度的视觉数据表征，很长的一段时间里这些野心都未能实现，这其中有着诸多因素。第一，包含许多表征的深度模型需要大量的有标注的数据才能表现得比其他经典方法更好，虽然这些当时还不为人知。限于当时计算机有限的存储和相对匮乏的 90 年代研究预算，大部分研究基于小数据集。比如，大部分可信的研究论文是基于 UCI 提供的若干个数据集，其中许多只有几百至几千张图片。

这一状况在 2009 年李飞飞贡献了 ImageNet 数据库后得以焕然。它包含了 1000 类，每类有 1000 张不同的图片，这一规模是当时其他数据集不可相提并论的。

(image credit: Ferhat Kurt)

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。



5.5.3 缺失要素 2: 硬件

深度学习对计算资源要求很高。这也是为什么上世纪 90 年代左右基于凸优化的算法更被青睐的原因。毕竟凸优化方法里能很快收敛，并可以找到全局最小值和高效的算法。

GPU 的到来改变了格局。很久以来，GPU 都是为了图像处理和计算机游戏而生的，尤其是为了大吞吐量的 4×4 矩阵和向量乘法，用于基本的图形转换。值得庆幸的是，这其中的数学与深度网络中的卷积层非常类似。与此同时，NVIDIA 和 ATI 开始对 GPU 为通用计算做优化，并命名为 GPGPU（即通用计算 GPU）。

为了更好的理解，我们来看看现代的 CPU。每个处理器核都十分强大，运作在高时钟频率，有先进复杂的结构和缓存。处理器可以很好地运行各种类型的代码，并由分支预测等机制使其能高效地运作在通用的常规程序上。然而，这个通用性同时也是一个弱点，因为通用的核心制造代价很高。它们会占用很多芯片面积，需要复杂的支持结构（内存接口，核间的缓存逻辑，高速互通连接等），并且跟不同任务的特制芯片相比它们在每个任务上表现并不完美。现代笔记本电脑可以有四核，而高端服务器也很少超过 64 核，就是因为这些核心并不划算。

相比较，GPU 通常有一百到一千个小处理单元组成（具体数值在 NVIDIA, ATI/AMD, ARM 和其他芯片厂商的产品间有所不同），这些单元通常被划分为稍大些的组（NVIDIA 把这称作 warps）。虽然它们每个处理单元相对较弱，运行在低于 1GHz 的时钟频率，庞大的数量使得 GPU 的运算速度比 CPU 快不止一个数量级。比如，NVIDIA 最新一代的 Volta 运算速度在特别的指令上可以达到每个芯片 120 TFlops，（更通用的指令上达到 24 TFlops），而至今 CPU 的浮点数运算速度也未超过 1 TFlop。这其中的原因很简单：首先，能量消耗与时钟频率成二次关系，所以同样供一个运行速度是 4x 的 CPU 核心所需的能量可以用来运行 16 个 GPU 核心以其 1/4 的速度运行，并达到 $16 \times 1/4 = 4x$ 的性能。此外，GPU 核心结构简单得多（事实上有很长一段时间他们甚至都不能运行通用的代码），这使得他们能量效率很高。最后，很多深度学习中的操作需要很高的内存带宽，而 GPU 以其十倍于很多 CPU 的内存带宽而占尽优势。

回到 2012 年，Alex Krizhevsky 和 Ilya Sutskever 实现的可以运行在 GPU 上的深度卷积网络成为重大突破。他们意识到卷积网络的运算瓶颈（卷积和矩阵乘法）其实都可以在硬件上并行。使用两个 NVIDIA GTX580 和 3GB 内存，他们实现了快速的卷积。他们足够好的代码 `cuda-convnet` 使其成为那几年里的业界标准，驱动着深度学习繁荣的头几年。

5.5.4 AlexNet

2012 年的时候，Khrizhevsky, Sutskever 和 Hinton 凭借他们的 `cuda-convnet` 实现的 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。他们在[这篇论文](#)中的模型与 1995 年的 LeNet 结构非常相似。



这个模型有一些显著的特征。第一，与相对较小的 LeNet 相比，AlexNet 包含 8 层变换，其中有五层卷积和两层全连接隐含层，以及一个输出层。

第一层中的卷积核大小是 11×11 ，接着第二层中的是 5×5 ，之后都是 3×3 。此外，第一，第二和第五个卷积层之后都跟了有重叠的大小为 3×3 ，步距为 2×2 的池化操作。

紧接着卷积层，原版的 AlexNet 有每层大小为 4096 个节点的全连接层们。这两个巨大的全连接层带来将近 1GB 的模型大小。由于早期 GPU 显存的限制，最早的 AlexNet 包括了双数据流的设计，以让网络中一半的节点能存入一个 GPU。这两个数据流，也就是说两个 GPU 只在一部分分层进行通信，这样达到限制 GPU 同步时的额外开销的效果。有幸的是，GPU 在过去几年得到了长足的发展，除了一些特殊的结构外，我们也就不再需要这样的特别设计了。

下面的 Gluon 代码定义了（稍微简化过的）Alexnet：

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

net = nn.Sequential()
with net.name_scope():
    net.add(
        # 第一阶段
        nn.Conv2D(channels=96, kernel_size=11,
                  strides=4, activation='relu'),
        nn.MaxPool2D(pool_size=3, strides=2),
        # 第二阶段
        nn.Conv2D(channels=256, kernel_size=5,
                  padding=2, activation='relu'),
        nn.MaxPool2D(pool_size=3, strides=2),
        # 第三阶段
        nn.Conv2D(channels=384, kernel_size=3,
                  padding=1, activation='relu'),
        nn.Conv2D(channels=384, kernel_size=3,
                  padding=1, activation='relu'),
        nn.Conv2D(channels=256, kernel_size=3,
                  padding=1, activation='relu'),
        nn.MaxPool2D(pool_size=3, strides=2),
        # 第四阶段
        nn.Flatten(),
        nn.Dense(4096, activation="relu"),
        nn.Dropout(.5),
        # 第五阶段
        nn.Dense(4096, activation="relu"),
        nn.Dropout(.5),
        # 第六阶段
```

```
    nn.Dense(10)
)
```

5.5.5 读取数据

Alexnet 使用 Imagenet 数据，其中输入图片大小一般是 224×224 。因为 Imagenet 数据训练时间过长，我们还是用前面的 FashionMNIST 来演示。读取数据的时候我们额外做了一步将数据扩大到原版 Alexnet 使用的 224×224 。

```
In [2]: import sys
        sys.path.append('..')
        import utils

        train_data, test_data = utils.load_data_fashion_mnist(
            batch_size=64, resize=224)
```

5.5.6 训练

这时候我们可以开始训练。相对于前面的 LeNet，我们做了如下三个改动：

1. 我们使用 Xavier 来初始化参数
2. 使用了更小的学习率
3. 默认只迭代一轮（这样网页编译快一点）

```
In [3]: from mxnet import init
        from mxnet import gluon

        ctx = utils.try_gpu()
        net.initialize(ctx=ctx, init=init.Xavier())

        loss = gluon.loss.SoftmaxCrossEntropyLoss()
        trainer = gluon.Trainer(net.collect_params(),
                               'sgd', {'learning_rate': 0.01})
        utils.train(train_data, test_data, net, loss,
                   trainer, ctx, num_epochs=1)

Start training on gpu(0)
Epoch 0. Loss: 2.303, Train acc 0.10, Test acc 0.11, Time 116.9 sec
```

5.5.7 结论

从 LeNet 到 Alexnet, 虽然实现起来也就多了几行而已。但这个观念上的转变和真正跑出好实验结果, 学术界整整花了 20 年。

5.5.8 练习

- 多迭代几轮看看? 跟 LeNet 比有什么区别? 为什么? (提示: 看看欠拟合和过拟合的那几张图)
- 找出 Xavier 具体是怎么初始化的, 跟默认的比有什么区别
- 尝试将训练的参数改回到 LeNet 看看会发生什么? 想想看为什么?
- 试试从 0 开始实现看看?

吐槽和讨论欢迎点[这里](#)

5.6 VGG: 使用重复元素的非常深的网络

我们从 Alexnet 看到网络的层数的激增。这个意味着即使是用 Gluon 手动写代码一层一层的堆每一层也很麻烦, 更不用说从 0 开始了。幸运的是编程语言提供了很好的方法来解决这个问题: 函数和循环。如果网络结构里面有大量重复结构, 那么我们可以很紧凑来构造这些网络。第一个使用这种结构的深度网络是 VGG。

5.6.1 VGG 架构

VGG 的一个关键是使用很多有着相对小的 kernel (3×3) 的卷积层然后接上一个池化层, 之后再将这个模块重复多次。下面我们先定义一个这样的块:

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

def vgg_block(num_convs, channels):
    out = nn.Sequential()
    for _ in range(num_convs):
        out.add(
            nn.Conv2D(channels=channels, kernel_size=3,
                      padding=1, activation='relu'))
    out.add(nn.MaxPool2D(pool_size=2, strides=2))
    return out
```

我们实例化一个这样的块，里面有两个卷积层，每个卷积层输出通道是 128：

In [2]: `from mxnet import nd`

```
blk = vgg_block(2, 128)
blk.initialize()
x = nd.random.uniform(shape=(2,3,16,16))
y = blk(x)
y.shape
```

Out[2]: (2, 128, 8, 8)

可以看到经过一个这样的块后，长宽会减半，通道也会改变。

然后我们定义如何将这些块堆起来：

In [3]: `def vgg_stack(architecture):`
 `out = nn.Sequential()`
 `for (num_convs, channels) in architecture:`
 `out.add(vgg_block(num_convs, channels))`
 `return out`

这里我们定义一个最简单的一个 VGG 结构，它有 8 个卷积层，和跟 Alexnet 一样的 3 个全连接层。这个网络又称 VGG 11.

In [4]: `num_outputs = 10`
`architecture = ((1,64), (1,128), (2,256), (2,512), (2,512))`
`net = nn.Sequential()`
`# add name_scope on the outermost Sequential`
`with net.name_scope():`
 `net.add(
 vgg_stack(architecture),
 nn.Flatten(),
 nn.Dense(4096, activation="relu"),
 nn.Dropout(.5),
 nn.Dense(4096, activation="relu"),
 nn.Dropout(.5),
 nn.Dense(num_outputs))`

5.6.2 模型训练

这里跟 Alexnet 的训练代码一样除了我们只将图片扩大到 96×96 来节省些计算，和默认使用稍微大点的学习率。

```
In [5]: import sys
        sys.path.append('..')
        import utils
        from mxnet import gluon
        from mxnet import init

        train_data, test_data = utils.load_data_fashion_mnist(
            batch_size=64, resize=96)

        ctx = utils.try_gpu()
        net.initialize(ctx=ctx, init=init.Xavier())

        loss = gluon.loss.SoftmaxCrossEntropyLoss()
        trainer = gluon.Trainer(net.collect_params(),
                               'sgd', {'learning_rate': 0.05})
        utils.train(train_data, test_data, net, loss,
                    trainer, ctx, num_epochs=1)

Start training on gpu(0)
Epoch 0. Loss: 0.845, Train acc 0.69, Test acc 0.84, Time 146.9 sec
```

5.6.3 总结

通过使用重复的元素，我们可以通过循环和函数来定义模型。使用不同的配置 (architecture) 可以得到一系列不同的模型。

5.6.4 练习

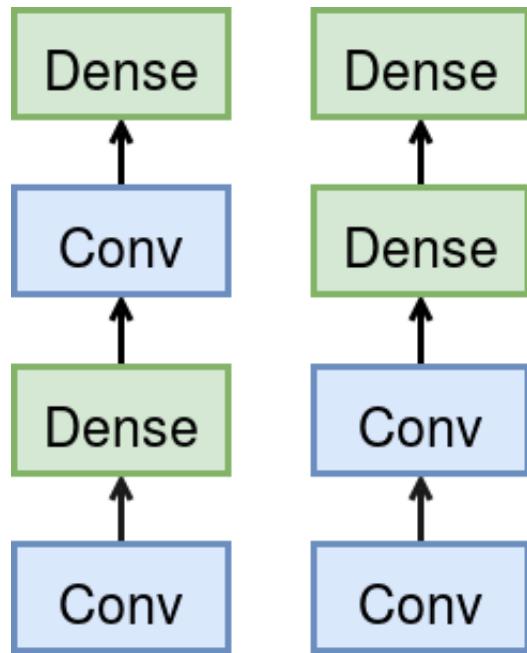
- 尝试多跑几轮，看看跟 LeNet/Alexnet 比怎么样？
- 尝试下构造 VGG 其他常用模型，例如 VGG16, VGG19。（提示：可以参考[VGG 论文](#)里的表 1。）
- 把图片从默认的 224×224 降到 96×96 有什么影响？

吐槽和讨论欢迎点[这里](#)

5.7 网络中的网络

Alexnet 之后一个重要的工作是[Network in Network \(NiN\)](#)，其提出的两个想法影响了后面的网络设计。

首先一点是注意到卷积神经网络一般分成两块，一块主要由卷积层构成，另一块主要是全连接层。在 Alexnet 里我们看到如何把卷积层块和全连接层分别加深加宽从而得到深度网络。另外一个自然的想法是，我们可以串联数个卷积层块和全连接层块来构建深度网络。



不过这里的一个难题是，卷积的输入输出是 4D 矩阵，然而全连接是 2D。同时在卷积神经网络里我们提到如果把 4D 矩阵转成 2D 做全连接，这个会导致全连接层有过多的参数。NiN 提出只对通道层做全连接并且像素之间共享权重来解决上述两个问题。就是说，我们使用 kernel 大小是 1×1 的卷积。

下面代码定义一个这样的块，它由一个正常的卷积层接上两个 kernel 是 1×1 的卷积层构成。后面两个充当两个全连接层的角色。

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

def mlpconv(channels, kernel_size, padding,
            strides=1, max_pooling=True):
    out = nn.Sequential()
    out.add(
        nn.Conv2D(channels=channels, kernel_size=kernel_size,
                  strides=strides, padding=padding,
                  activation='relu'),
        nn.Conv2D(channels=channels, kernel_size=1,
                  padding=0, strides=1, activation='relu'),
        nn.Conv2D(channels=channels, kernel_size=1,
                  padding=0, strides=1, activation='relu'))
    if max_pooling:
```

```
        out.add(nn.MaxPool2D(pool_size=3, strides=2))
    return out
```

测试一下：

```
In [2]: from mxnet import nd

blk = mlpconv(64, 3, 0)
blk.initialize()

x = nd.random.uniform(shape=(32, 3, 16, 16))
y = blk(x)
y.shape

Out[2]: (32, 64, 6, 6)
```

NiN 的卷积层的参数跟 Alexnet 类似，使用三组不同的设定

- kernel: 11×11 , channels: 96
- kernel: 5×5 , channels: 256
- kernel: 3×3 , channels: 384

除了使用了 1×1 卷积外，NiN 在最后不是使用全连接，而是使用通道数为输出类别个数的 mlp-conv，外接一个平均池化层来将每个通道里的数值平均成一个标量。

```
In [3]: net = nn.Sequential()
        # add name_scope on the outer most Sequential
        with net.name_scope():
            net.add(
                mlpconv(96, 11, 0, strides=4),
                mlpconv(256, 5, 2),
                mlpconv(384, 3, 1),
                nn.Dropout(.5),
                # 目标类为 10 类
                mlpconv(10, 3, 1, max_pooling=False),
                # 输入为 batch_size x 10 x 5 x 5, 通过 AvgPool2D 转成
                # batch_size x 10 x 1 x 1。
                nn.AvgPool2D(pool_size=5),
                # 转成 batch_size x 10
                nn.Flatten()
            )
```

5.7.1 获取数据并训练

跟 Alexnet 类似，但使用了更大的学习率。

```
In [4]: import sys
        sys.path.append('..')
        import utils
        from mxnet import gluon
        from mxnet import init

        train_data, test_data = utils.load_data_fashion_mnist(
            batch_size=64, resize=224)

        ctx = utils.try_gpu()
        net.initialize(ctx=ctx, init=init.Xavier())

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

Start training on gpu(0)
Epoch 0. Loss: 2.303, Train acc 0.10, Test acc 0.10, Time 145.9 sec
```

5.7.2 结论

这种“一卷卷到底”最后加一个平均池化层的做法也成为了深度卷积神经网络的常用设计。

5.7.3 练习

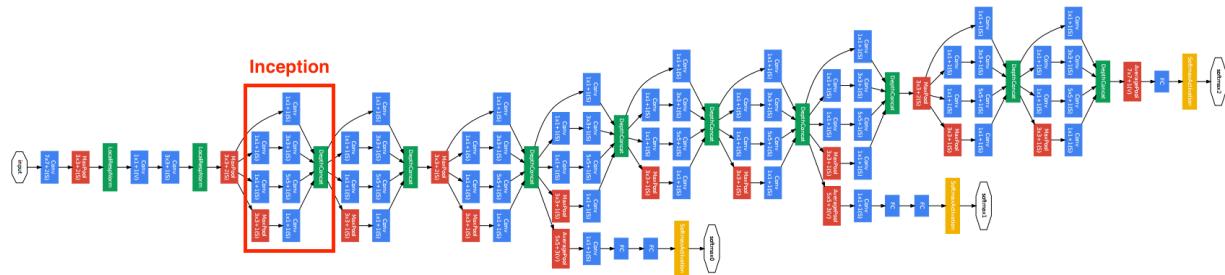
- 为什么 mlpconv 里面要有两个 1×1 卷积？

吐槽和讨论欢迎点[这里](#)

5.8 更深的卷积神经网络：GoogLeNet

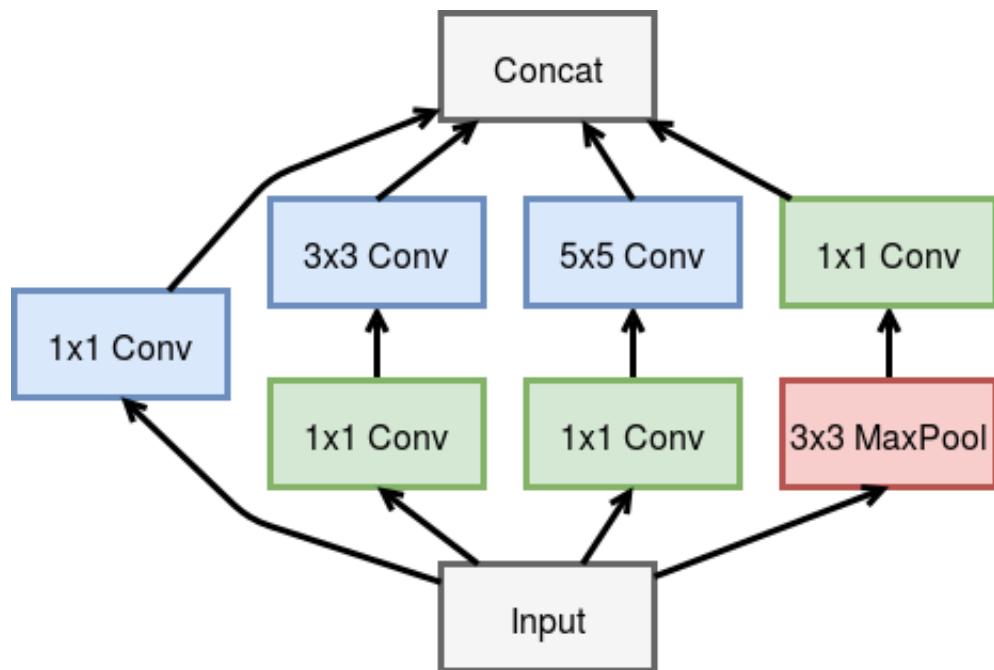
在 2014 年的 Imagenet 竞赛里，Google 的研究人员利用一个新的网络结构取得很大的优先。这个叫做 GoogLeNet 的网络虽然在名字上是向 LeNet 致敬，但网络结构里很难看到 LeNet 的影

子。它颠覆的大家对卷积神经网络串联一系列层的固定做法。下图是其论文对 GoogLeNet 的可视化



5.8.1 定义 Inception

可以看到其中有多个四个并行卷积层的块。这个块一般叫做 Inception，其基于*Network in network*的思想做了很大的改进。我们先看下如何定义一个下图所示的 Inception 块。



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

class Inception(nn.Block):
    def __init__(self, n1_1, n2_1, n2_3, n3_1, n3_5, n4_1, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # path 1
        self.p1_conv_1 = nn.Conv2D(n1_1, kernel_size=1,
```

```

                    activation='relu')
# path 2
self.p2_conv_1 = nn.Conv2D(n2_1, kernel_size=1,
                         activation='relu')
self.p2_conv_3 = nn.Conv2D(n2_3, kernel_size=3, padding=1,
                         activation='relu')
# path 3
self.p3_conv_1 = nn.Conv2D(n3_1, kernel_size=1,
                         activation='relu')
self.p3_conv_5 = nn.Conv2D(n3_5, kernel_size=5, padding=2,
                         activation='relu')
# path 4
self.p4_pool_3 = nn.MaxPool2D(pool_size=3, padding=1,
                           strides=1)
self.p4_conv_1 = nn.Conv2D(n4_1, kernel_size=1,
                         activation='relu')

def forward(self, x):
    p1 = self.p1_conv_1(x)
    p2 = self.p2_conv_3(self.p2_conv_1(x))
    p3 = self.p3_conv_5(self.p3_conv_1(x))
    p4 = self.p4_conv_1(self.p4_pool_3(x))
    return nd.concat(p1, p2, p3, p4, dim=1)

```

可以看到 Inception 里有四个并行的线路。

1. 单个 1×1 卷积。
2. 1×1 卷积接上 3×3 卷积。通常前者的通道数少于输入通道，这样减少后者的计算量。后者加上了 `padding=1` 使得输出的长宽的输入一致
3. 同 2，但换成了 5×5 卷积
4. 和 1 类似，但卷积前用了最大池化层

最后将这四个并行线路的结果在通道这个维度上合并在一起。

测试一下：

```
In [2]: incp = Inception(64, 96, 128, 16, 32, 32)
incp.initialize()

x = nd.random.uniform(shape=(32, 3, 64, 64))
incp(x).shape
```

Out[2]: (32, 256, 64, 64)

5.8.2 定义 GoogLeNet

GoogLeNet 将数个 Inception 串联在一起。注意到原论文里使用了多个输出，为了简化我们这里就使用一个输出。为了可以更方便的查看数据在内部的形状变化，我们对每个块使用一个 `nn.Sequential`，然后再把所有这些块连起来。

```
In [3]: class GoogLeNet(nn.Block):
    def __init__(self, num_classes, verbose=False, **kwargs):
        super(GoogLeNet, self).__init__(**kwargs)
        self.verbose = verbose
        # add name_scope on the outer most Sequential
        with self.name_scope():
            # block 1
            b1 = nn.Sequential()
            b1.add(
                nn.Conv2D(64, kernel_size=7, strides=2,
                         padding=3, activation='relu'),
                nn.MaxPool2D(pool_size=3, strides=2)
            )
            # block 2
            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)
            )
            # block 3
            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)
            )
            # block 4
            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)
    )

    # block 5
    b5 = nn.Sequential()
    b5.add(
        Inception(256, 160, 320, 32, 128, 128),
        Inception(384, 192, 384, 48, 128, 128),
        nn.AvgPool2D(pool_size=2)
    )
    # block 6
    b6 = nn.Sequential()
    b6.add(
        nn.Flatten(),
        nn.Dense(num_classes)
    )
    # chain blocks together
    self.net = nn.Sequential()
    self.net.add(b1, b2, b3, b4, b5, b6)

def forward(self, 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

```

我们看一下每个块对输出的改变。

```
In [4]: net = GoogLeNet(10, verbose=True)
net.initialize()

x = nd.random.uniform(shape=(4, 3, 96, 96))
y = net(x)

Block 1 output: (4, 64, 23, 23)
Block 2 output: (4, 192, 11, 11)
Block 3 output: (4, 480, 5, 5)
```

```
Block 4 output: (4, 832, 2, 2)
Block 5 output: (4, 1024, 1, 1)
Block 6 output: (4, 10)
```

5.8.3 获取数据并训练

跟 VGG 一样我们使用了较小的输入 96×96 来加速计算。

```
In [5]: import sys
        sys.path.append('..')
        import utils
        from mxnet import gluon
        from mxnet import init

        train_data, test_data = utils.load_data_fashion_mnist(
            batch_size=64, resize=96)

        ctx = utils.try_gpu()
        net = GoogLeNet(10)
        net.initialize(ctx=ctx, init=init.Xavier())

        loss = gluon.loss.SoftmaxCrossEntropyLoss()
        trainer = gluon.Trainer(net.collect_params(),
                               'sgd', {'learning_rate': 0.01})
        utils.train(train_data, test_data, net, loss,
                    trainer, ctx, num_epochs=1)

Start training on gpu(0)
Epoch 0. Loss: 2.147, Train acc 0.29, Test acc 0.52, Time 91.3 sec
```

5.8.4 结论

GoogLeNet 加入了更加结构化的 Inception 块来使得我们可以使用更大的通道，更多的层，同时控制计算量和模型大小在合理范围内。

5.8.5 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

- v1: 本节介绍的是最早版本: [Going Deeper with Convolutions](#)

- v2: 加入和 Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
- v3: 对 Inception 做了调整: Rethinking the Inception Architecture for Computer Vision
- v4: 基于 ResNet 加入了 Residual Connections: Inception-ResNet and the Impact of Residual Connections on Learning

吐槽和讨论欢迎点[这里](#)

5.9 ResNet: 深度残差网络

当大家还在惊叹 GoogLeNet 用结构化的连接纳入了大量卷积层的时候，微软亚洲研究院的研究员已经在设计更深但结构更简单的网络ResNet。他们凭借这个网络在 2015 年的 Imagenet 竞赛中大获全胜。

ResNet 有效的解决了深度卷积神经网络难训练的问题。这是因为在误差反传的过程中，梯度通常变得越来越小，从而权重的更新量也变小。这个导致远离损失函数的层训练缓慢，随着层数的增加这个现象更加明显。之前有两种常用方案来尝试解决这个问题：

1. 按层训练。先训练靠近数据的层，然后慢慢的增加后面的层。但效果不是特别好，而且比较麻烦。
2. 使用更宽的层（增加输出通道）而不是更深来增加模型复杂度。但更宽的模型经常不如更深的效果好。

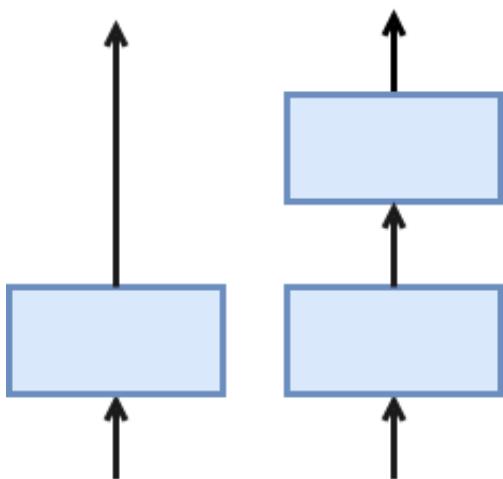
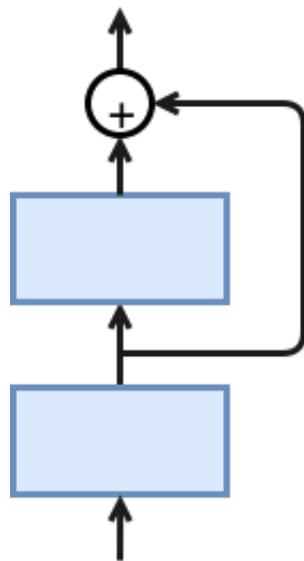
ResNet 通过增加跨层的连接来解决梯度逐层回传时变小的问题。虽然这个想法之前就提出过了，但 ResNet 真正的把效果做好了。

下图演示了一个跨层的连接。

最底下那层的输入不仅仅是输出给了中间层，而且其与中间层结果相加进入最上层。这样在梯度反传时，最上层梯度可以直接跳过中间层传到最下层，从而避免最下层梯度过小情况。

为什么叫做残差网络呢？我们可以将上面示意图里的结构拆成两个网络的和，一个一层，一个两层，最下面层是共享的。

在训练过程中，左边的网络因为更简单所以更容易训练。这个小网络没有拟合到的部分，或者说残差，则被右边的网络抓取住。所以直观上来说，即使加深网络，跨层连接仍然可以使得底层网络可以充分的训练，从而不会让训练更难。



5.9.1 Residual 块

ResNet 沿用了 VGG 的那种全用 3×3 卷积，但在卷积和池化层之间加入了批量归一化来加速训练。每次跨层连接跨过两层卷积。这里我们定义一个这样的残差块。注意到如果输入的通道数和输出不一样时 (`same_shape=False`)，我们使用一个额外的 1×1 卷积来做通道变化，同时使用 `strides=2` 来把长宽减半。

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

        class Residual(nn.Block):
            def __init__(self, channels, same_shape=True, **kwargs):
                super(Residual, self).__init__(**kwargs)
                self.same_shape = same_shape
                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 forward(self, x):
                out = nd.relu(self.bn1(self.conv1(x)))
                out = self.bn2(self.conv2(out))
                if not self.same_shape:
                    x = self.conv3(x)
                return nd.relu(out + x)
```

输入输出通道相同：

```
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)
```

输入输出通道不同：

```
In [3]: blk2 = Residual(8, same_shape=False)
```

```
blk2.initialize()
blk2(x).shape

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

5.9.2 构建 ResNet

类似 GoogLeNet 主体是由 Inception 块串联而成, ResNet 的主体部分串联多个 Residual 块。下面我们定义 18 层的 ResNet。同样为了阅读更加容易, 我们这里使用了多个 `nn.Sequential`。另外注意到一点是, 这里我们没用池化层来减小数据长宽, 而是通过有通道变化的 Residual 块里面的使用 `strides=2` 的卷积层。

```
In [4]: class ResNet(nn.Block):
    def __init__(self, num_classes, verbose=False, **kwargs):
        super(ResNet, self).__init__(**kwargs)
        self.verbose = verbose
        # add name_scope on the outermost Sequential
        with self.name_scope():
            # block 1
            b1 = nn.Conv2D(64, kernel_size=7, strides=2)
            # block 2
            b2 = nn.Sequential()
            b2.add(
                nn.MaxPool2D(pool_size=3, strides=2),
                Residual(64),
                Residual(64))
            )
            # block 3
            b3 = nn.Sequential()
            b3.add(
                Residual(128, same_shape=False),
                Residual(128))
            )
            # block 4
            b4 = nn.Sequential()
            b4.add(
                Residual(256, same_shape=False),
                Residual(256))
            )
            # block 5
            b5 = nn.Sequential()
            b5.add(
```

```

        Residual(512, same_shape=False),
        Residual(512)
    )
    # block 6
    b6 = nn.Sequential()
    b6.add(
        nn.AvgPool2D(pool_size=3),
        nn.Dense(num_classes)
    )
    # chain all blocks together
    self.net = nn.Sequential()
    self.net.add(b1, b2, b3, b4, b5, b6)

def forward(self, 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

```

这里演示数据在块之间的形状变化:

```

In [5]: net = ResNet(10, verbose=True)
net.initialize()

x = nd.random.uniform(shape=(4, 3, 96, 96))
y = net(x)

Block 1 output: (4, 64, 45, 45)
Block 2 output: (4, 64, 22, 22)
Block 3 output: (4, 128, 11, 11)
Block 4 output: (4, 256, 6, 6)
Block 5 output: (4, 512, 3, 3)
Block 6 output: (4, 10)

```

5.9.3 获取数据并训练

跟前面类似，但因为有批量归一化，所以使用了较大的学习率。

```

In [6]: import sys
sys.path.append('..')
import utils

```

```
from mxnet import gluon
from mxnet import init

train_data, test_data = utils.load_data_fashion_mnist(
    batch_size=64, resize=96)

ctx = utils.try_gpu()
net = ResNet(10)
net.initialize(ctx=ctx, init=init.Xavier())

loss = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(),
                        'sgd', {'learning_rate': 0.05})
utils.train(train_data, test_data, net, loss,
            trainer, ctx, num_epochs=1)

Start training on gpu(0)
Epoch 0. Loss: 0.439, Train acc 0.84, Test acc 0.89, Time 79.5 sec
```

5.9.4 结论

ResNet 使用跨层通道使得训练非常深的卷积神经网络成为可能。同样它使用很简单的卷积层配置，使得其拓展更加简单。

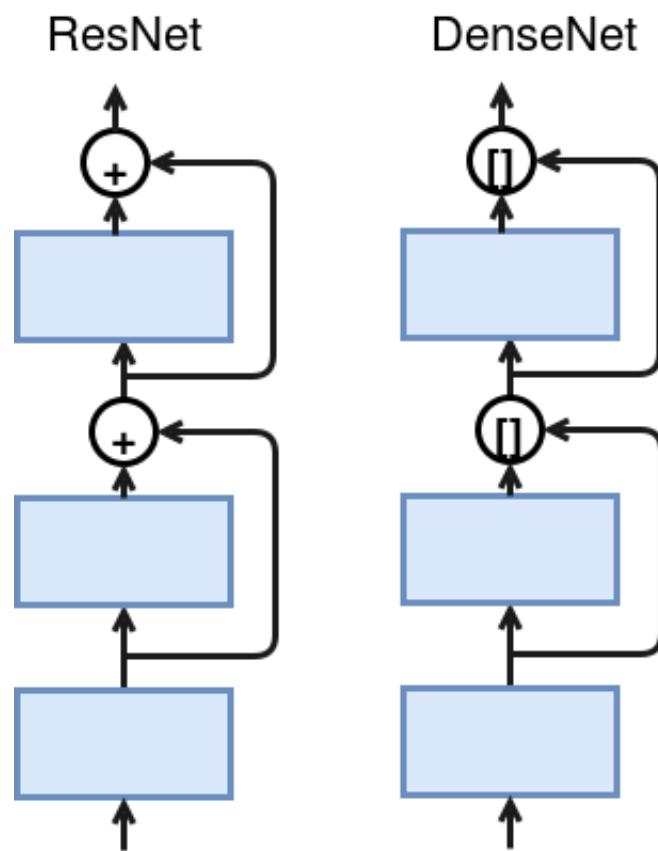
5.9.5 练习

- 这里我们实现了 ResNet 18，原论文中还讨论了更深的配置。尝试实现它们。（提示：参考论文中的表 1）
- 原论文中还介绍了一个“bottleneck”架构，尝试实现它
- ResNet 作者在[接下来的一篇论文](#)讨论了将 Residual 块里面的 Conv->BN->Relu 结构改成了 BN->Relu->Conv（参考论文图 1），尝试实现它

吐槽和讨论欢迎点[这里](#)

5.10 DenseNet：稠密连接的卷积神经网络

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



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

5.10.1 稠密块 (Dense Block)

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

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

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

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

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

我们验证下输出通道数是不是符合预期。

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

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

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

5.10.2 过渡块 (Transition Block)

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

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

验证一下结果：

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

blk(x).shape
```

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

5.10.3 DenseNet

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

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

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

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

5.10.4 获取数据并训练

因为这里我们使用了比较深的网络，所以我们进一步把输入减少到 32×32 来训练。

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

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

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

loss = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(),
```

```
'sgd', {'learning_rate': 0.1})  
utils.train(train_data, test_data, net, loss,  
            trainer, ctx, num_epochs=1)  
  
Start training on gpu(0)  
Epoch 0. Loss: 0.482, Train acc 0.82, Test acc 0.83, Time 88.1 sec
```

5.10.5 结论

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

5.10.6 练习

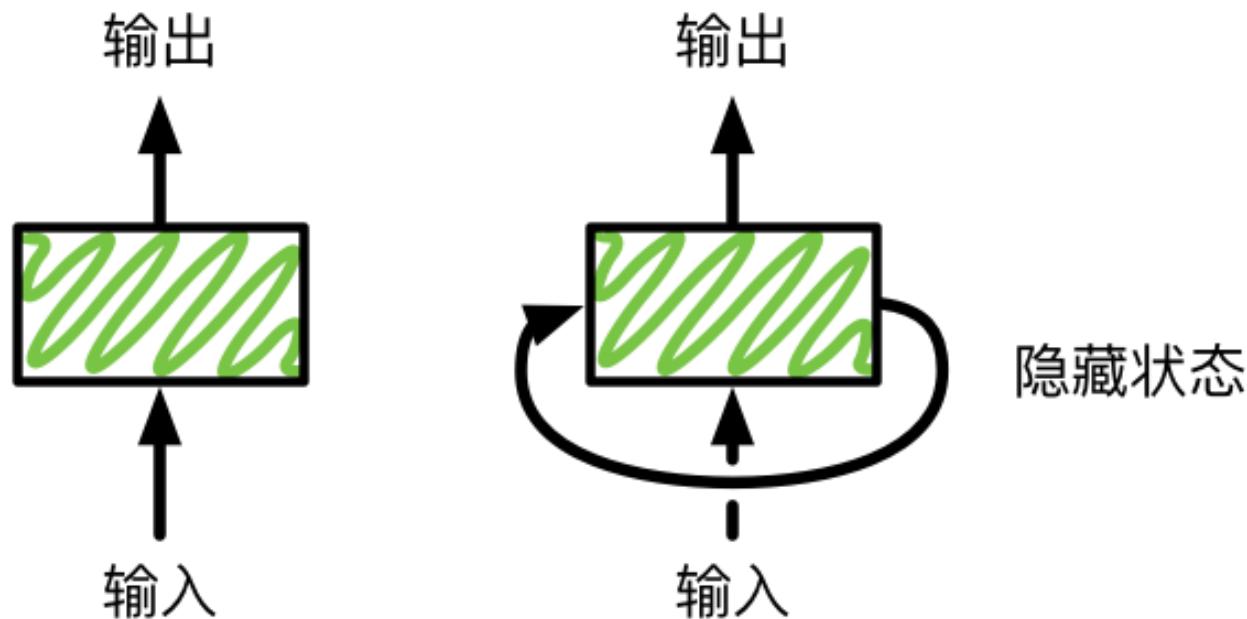
- DesNet 论文中提交的一个优点是其模型参数比 ResNet 更小，想想为什么？
- DesNet 被人诟病的一个问题是内存消耗过多。真的会这样吗？可以把输入换成 224×224 （需要改最后的 AvgPool2D 大小），来看看实际（GPU）内存消耗。
- 这里的 FashionMNIST 有必要用 100+ 层的网络吗？尝试将其改简单看看效果。

吐槽和讨论欢迎点[这里](#)

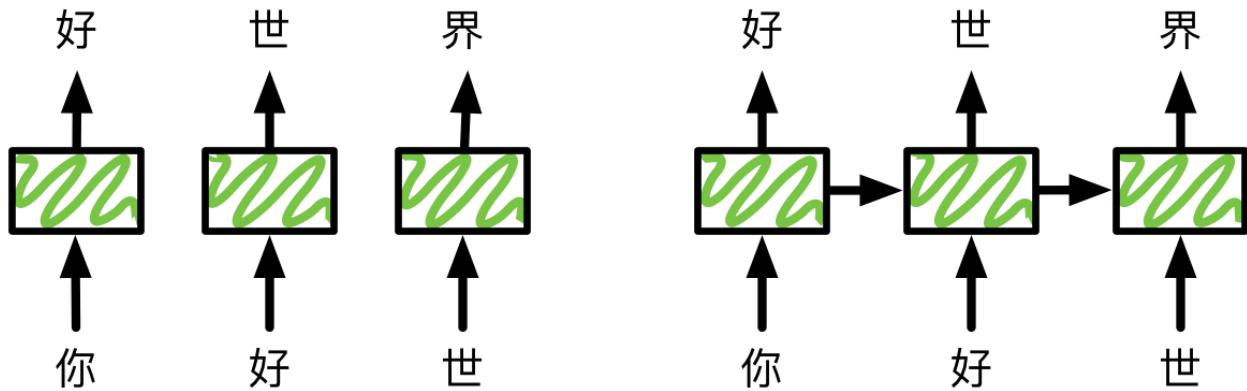
循环神经网络

6.1 循环神经网络—从 0 开始

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



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



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

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

6.1.1 循环神经网络

在对输入输出数据有了解后，我们来正式介绍循环神经网络。

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

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

假定隐层宽度为 h ，那么其中的权重参数的尺寸为 $\mathbf{W}_{xh} \in \mathbb{R}^{x \times h}$ 。偏移参数 $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ 在与前一项 $\mathbf{X}\mathbf{W}_{xh} \in \mathbb{R}^{n \times h}$ 相加时使用了**广播**。这个隐层的输出的尺寸为 $\mathbf{H} \in \mathbb{R}^{n \times h}$ 。

把隐层的输出 \mathbf{H} 作为输出层的输入，最终的输出

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

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

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

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

输出的计算跟前面一致：

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

一开始我们提到过，隐层输出（又叫隐藏状态）可以认为是这个网络的记忆。它存储前面时间里面的信息。我们的输出是只基于这个状态。最开始的隐藏状态里的元素通常会被初始化为 0。

6.1.2 周杰伦歌词数据集

为了实现并展示循环神经网络，我们使用周杰伦歌词数据集来训练模型作词。该数据集里包含了著名创作型歌手周杰伦从第一张专辑《Jay》到第十张专辑《跨时代》所有歌曲的歌词。

下面我们读取这个数据并看看前面 49 个字符（char）是什么样的：

```
In [1]: import zipfile
        with zipfile.ZipFile('../data/jaychou_lyrics.txt.zip', 'r') as zin:
            zin.extractall('../data/')

        with open('../data/jaychou_lyrics.txt') as f:
            corpus_chars = f.read()
        print(corpus_chars[0:49])
```

想要有直升机
想要和你飞到宇宙去
想要和你融化在一起
融化在宇宙里
我每天每天每天在想想想著你

我们看一下数据集里的字符数。

```
In [2]: len(corpus_chars)
Out[2]: 64925
```

接着我们稍微处理下数据集。为了打印方便，我们把换行符替换成空格，然后截去后面一段使得接下来的训练会快一点。



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

6.1.3 字符的数值表示

先把数据里面所有不同的字符拿出来做成一个字典：

```
In [4]: idx_to_char = list(set(corpus_chars))
char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
vocab_size = len(char_to_idx)

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

vocab size: 1465

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

```
In [5]: corpus_indices = [char_to_idx[char] for char in corpus_chars]

sample = corpus_indices[:40]

print('chars: \n', ''.join([idx_to_char[idx] for idx in sample]))
print('\nindices: \n', sample)

chars:
想要有直升机 想要和你飞到宇宙去 想要和你融化在一起 融化在宇宙里 我每天每天每

indices:
[614, 811, 1393, 128, 937, 545, 756, 614, 811, 881, 968, 1180, 59, 1234, 1127, 1094, 756,
```

6.1.4 时序数据的批量采样

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

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

随机批量采样

下面代码每次从数据里随机采样一个批量。

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

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

            # 返回 seq_len 个数据
            def _data(pos):
                return corpus_indices[pos: pos + seq_len]

            for i in range(epoch_size):
                # 每次读取 batch_size 个随机样本
                i = i * batch_size
                batch_indices = example_indices[i: i + batch_size]
                data = nd.array(
                    [_data(j * seq_len) for j in batch_indices], ctx=ctx)
                label = nd.array(
                    [_data(j * seq_len + 1) for j in batch_indices], ctx=ctx)
                yield data, label
```

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

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

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

data:
[[ 15.  16.  17.]
 [ 12.  13.  14.]]
<NDArray 2x3 @cpu(0)>
label:
[[ 16.  17.  18.]
```

```
[ 13.  14.  15.]
<NDArray 2x3 @cpu(0)>
```

```
data:
[[ 18.  19.  20.]
 [ 0.   1.   2.]]
<NDArray 2x3 @cpu(0)>
label:
[[ 19.  20.  21.]
 [ 1.   2.   3.]]
<NDArray 2x3 @cpu(0)>
```

```
data:
[[ 3.   4.   5.]
 [ 21.  22.  23.]]
<NDArray 2x3 @cpu(0)>
label:
[[ 4.   5.   6.]
 [ 22.  23.  24.]]
<NDArray 2x3 @cpu(0)>
```

```
data:
[[ 6.   7.   8.]
 [ 9.  10.  11.]]
<NDArray 2x3 @cpu(0)>
label:
[[ 7.   8.   9.]
 [ 10.  11.  12.]]
<NDArray 2x3 @cpu(0)>
```

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

相邻批量采样

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

```
In [8]: def data_iter_consecutive(corpus_indices, batch_size, seq_len, ctx=None):
    corpus_indices = nd.array(corpus_indices, ctx=ctx)
```

```
data_len = len(corpus_indices)
batch_len = data_len // batch_size

indices = corpus_indices[0: batch_size * batch_len].reshape((
    batch_size, batch_len))
# 减一是因为 label 的索引是相应 data 的索引加一
epoch_size = (batch_len - 1) // seq_len

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

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

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

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

data:
[[ 0.  1.  2.]
 [15. 16. 17.]]
<NDArray 2x3 @cpu(0)>
label:
[[ 1.  2.  3.]
 [16. 17. 18.]]
<NDArray 2x3 @cpu(0)>

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

data:
[[ 6.  7.  8.]
 [21. 22. 23.]]
```

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

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

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

6.1.5 Onehot 编码

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

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

Out[10]:

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

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

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

inputs = get_inputs(data)
```

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

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

6.1.6 初始化模型参数

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

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

In [12]: `import mxnet as mx`

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

# 隐藏状态长度
hidden_size = 256
std = .01

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

    # 输出层
    W_hy = nd.random_normal(scale = std, shape=(hidden_size, vocab_size), ctx=ctx)
    b_y = nd.zeros(vocab_size, ctx=ctx)

    params = [W_xh, W_hh, b_h, W_hy, b_y]
    for param in params:
        param.attach_grad()
```

```

    return params

Will use gpu(0)

```

6.1.7 定义模型

当序列中某一个时间戳的输入为一个样本数为 `batch_size` 的批量，而整个序列长度为 `seq_len` 时，以下 `rnn` 函数的 `inputs` 和 `outputs` 皆为 `seq_len` 个尺寸 `batch_size * vocab_size` 的矩阵，隐藏状态变量 `H` 是一个尺寸为 `batch_size * hidden_size` 的矩阵。

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

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

```

In [13]: def rnn(inputs, H, W_xh, W_hh, b_h, W_ty, b_ty):
    # inputs: seq_len 个尺寸为 batch_size * vocab_size 矩阵
    # H: 尺寸为 batch_size * hidden_size 矩阵
    # outputs: seq_len 个尺寸为 batch_size * vocab_size 矩阵
    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_ty
        outputs.append(Y)
    return (outputs, H)

```

做个简单的测试：

```

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

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

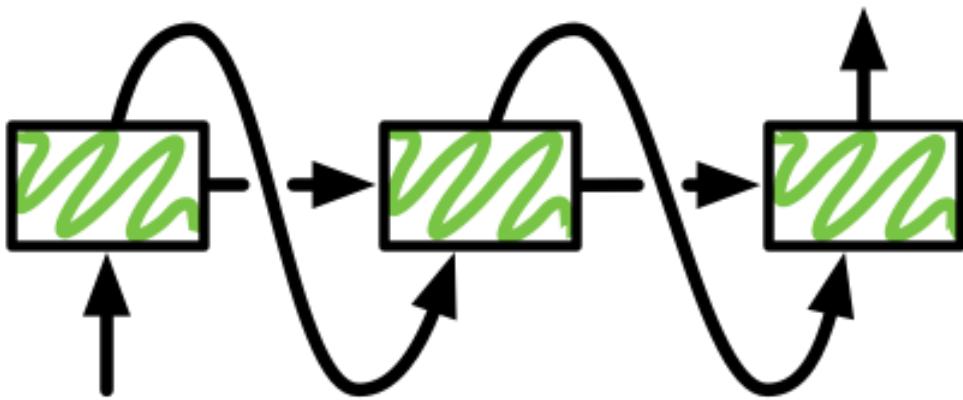
        print('output length: ', len(outputs))
        print('output[0] shape: ', outputs[0].shape)
        print('state shape: ', state_new.shape)

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

```

6.1.8 预测序列

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



```
In [15]: def predict(prefix, num_chars, params):
    # 预测以 prefix 开始的接下来的 num_chars 个字符
    prefix = prefix.lower()
    state = nd.zeros(shape=(1, hidden_size), 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
        Y, state = rnn(get_inputs(X), state, *params)
        if i < len(prefix)-1:
            next_input = char_to_idx[prefix[i+1]]
        else:
            next_input = int(Y[0].argmax(axis=1).asscalar())
        output.append(next_input)
    return ''.join([idx_to_char[i] for i in output])
```

6.1.9 梯度剪裁

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

为了应对这一现象, 一个常用的做法是如果梯度特别大, 那么就投影到一个比较小的尺度上。假设我们把所有梯度接成一个向量 g , 假设剪裁的阈值是 θ , 那么我们这样剪裁使得 $\|g\|$ 不会超过 θ :

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

```
In [16]: def grad_clipping(params, theta):
    norm = nd.array([0.0], ctx)
```

```

for p in params:
    norm += nd.sum(p.grad ** 2)
norm = nd.sqrt(norm).asscalar()
if norm > theta:
    for p in params:
        p.grad[:] *= theta / norm

```

6.1.10 训练模型

下面我们可以还是训练模型。跟前面前置网络的教程比，这里有以下几个不同。

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

困惑度（Perplexity）

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

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

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

而这里的困惑度可以简单的认为就是对交叉熵做 \exp 运算使得数值更好读。

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

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

因此，困惑度的值总是在 1 和 $|W|$ 之间。如果一个模型可以取得较低的困惑度的值（更靠近 1），通常情况下，该模型预测更加准确。

```
In [17]: seq1 = '分开'
        seq2 = '不分开'
        seq3 = '战争中部队'
```

```
from mxnet import autograd
```

```
from mxnet import gluon
from math import exp

epochs = 200
seq_len = 35
learning_rate = .1
batch_size = 32

softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()

def train_and_predict(is_random_iter):
    if is_random_iter:
        data_iter = data_iter_random
    else:
        data_iter = data_iter_consecutive
    params = get_params()

    for e in range(1, epochs + 1):
        # 如使用相邻批量采样，在同一个 epoch 中，隐藏状态只需要在该 epoch 开始的时候初始化。
        if not is_random_iter:
            state = nd.zeros(shape=(batch_size, hidden_size), ctx=ctx)

        train_loss, num_examples = 0, 0
        for data, label in data_iter(corpus_indices, batch_size, seq_len, ctx):
            # 如使用随机批量采样，处理每个随机小批量前都需要初始化隐藏状态。
            if is_random_iter:
                state = nd.zeros(shape=(batch_size, hidden_size), ctx=ctx)
            with autograd.record():
                # outputs 尺寸: (batch_size, vocab_size)
                outputs, state = rnn(get_inputs(data), state, *params)
                # label 尺寸: (batch_size * seq_len,) 设 t_ib_j 为 i 时间批量中的 j 元素
                # label = [t_0b_0, t_0b_1, ..., t_1b_0, t_1b_1, ..., ]
                label = label.T.reshape((-1,))
                # 拼接 outputs，尺寸: (batch_size * seq_len, vocab_size)。
                outputs = nd.concat(*outputs, dim=0)
                # 经上述操作，outputs 和 label 已对齐。
                loss = softmax_cross_entropy(outputs, label)
            loss.backward()

            grad_clipping(params, 5)
            utils.SGD(params, learning_rate)
```

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

if e % 20 == 0:
    print("Epoch %d. Perplexity %f" % (e, exp(train_loss/num_examples)))
    print(' - ', predict(seq1, 100, params))
    print(' - ', predict(seq2, 100, params))
    print(' - ', predict(seq3, 100, params), '\n')
```

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

```
In [18]: train_and_predict(is_random_iter=True)
```

Epoch 20. Perplexity 330.574391

- 分开
 - 不分开
 - 战争中部队

Epoch 40. Perplexity 199.635309

- 分开 我不的让我的 我不的我 我有你 我不 我不 我不的 我不 我不 我不的 我不 我不 我不的 我不 我不 我不的
- 不分开 我想的让我的 我不的我 我有你 我不 我不 我不的 我不 我不 我不的 我不 我不 我不的 我不 我不 我不的
- 战争中部队 我不的我 我有你 我不 我不 我不的 我不 我不 我不的 我不 我不 我不的 我不 我不 我不的 我不 我不

Epoch 60. Perplexity 126.831014

- 分开 我不要我想 我不要你不 我不要 想不两 三颗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
 - 不分开 我想不的茶 有一种 爱不两 一颗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
 - 战争中部队 我想你的茶 有一种 爱不两 一颗的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人 坏坏的让我疯狂的可爱女人

Epoch 80. Perplexity 77.688958

- 分开 我不要这想你 你 想着你的茶球 像 连着我的茶球 你 想着你的茶 有一种 别不的让我面 你说你 你不是 想不
- 不分开 我想不这你想我 想你的你 沉你的可我疯 的可后 的灵魂 翻爱 停不 x Y x Y x Y x Y x Y x Y x Y
- 战争中部队 我想能的爱 我说 我不人 想情的 一颗 停地 x Y x Y x Y x Y x Y x Y x Y x Y x Y x Y x Y

Epoch 100. Perplexity 47.358514

- 分开 我不想再不起 你说不起你 我不要你想 我不能再想 我不 我不 我不要 想你 你你 我想 我不 我不要 想你走
- 不分开 我不不你心 我不要你想 我不 我不 我不要 爱你 你你 我想 我不 我不要 想你走 我不要 你爱我 想你的!
- 战争中部队的天戏 想你的天我有 配去 你想到 爱情 一直球 印炭 一直半 印炭 一直半 印炭 一直半 印炭 一直半

Epoch 120. Perplexity 30.041632

- 分开 我不要再想你 我知不觉 你已不感开我 不知 是你很久了你 我想 你不离ㄟ气质 这是 你不离ㄟ气我 说散 你
- 不分开 我想不你看碎没人 你想不你 你小我 你兽我 的灵魂 单滚 停止忿存 永你不外 在你会感 我想会这开我 不

- 战争中部队 爷有的旧 有一种味气叫做家 (爷泡 一步两 三颗的公式我学不来 我想不这不是我要 我不着你的爱 有

Epoch 140. Perplexity 20.016062

- 分开 我不是再不活 我说你这不难 我说你 你爱我 想 简! 简! 单! 单 夹在我们的我有 我将你 泪爱我 想你怎么
- 不分开 别在那位看不要 什么的客我心坠的可爱女人 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
- 战争中部队的泥戏 消失的我旧 我的世界将义 太瞑 我想到你为质 我不 你想 我不要我想 你不能再想 我不 我不

Epoch 160. Perplexity 14.047996

- 分开 我不会再想你 我知不觉 你不的让我心狂的可爱女人 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
- 不分开 现不是你的微笑 想会有动 一颗河重 将想了空 恨只了空 你只会空 我一不觉 我一不这节活的沮息 在录到一
- 战争中部队家 我要你这样蜂着我 不是一多 你只会感到更加沮丧) (难道这不是我要的天堂景象 沉沦假象 你只会感

Epoch 180. Perplexity 10.250742

- 分开 我想要再想你 我知不起 你我的那是些 我的手打生活 你说不起 说你的让我感动的可爱女人 坏坏的让我疯狂的
- 不分开 我想不能好牵着 为身为龙 我用一直是我看的回面搞 我我怕不离开不要 是有你看想 一直走在黑暗我也道
- 战争中部队. 想时在的茶 有一种味道叫做家 他羽泡的茶 有一种味道叫做家 他羽泡的茶 有一种味道叫做家 他羽泡

Epoch 200. Perplexity 7.786830

- 分开 我不想再开你 后知不觉 又已了一个秋 后知后觉 我该好好生活 我右拳打开了 不知不觉 你已经离开我 不知不
- 不分开 现在那不看不要 想容你也接受 不想了其离我不要 你说着这不起 你说着我会离开的没想 这不到我的你 想
- 战争中部队到 想要好着不觉 让我带着你离开 这是顽要 这是 说 后你是 简! 兮 单颗了 娘想我 别你却人年著

我们再采用相邻批量采样实验循环神经网络谱写歌词。

```
In [19]: train_and_predict(is_random_iter=False)
```

Epoch 20. Perplexity 336.546726

- 分开
- 不分开
- 战争中部队

Epoch 40. Perplexity 196.306526

- 分开 我不的我 你我的让我 我不 我不
- 不分开 我不的让我 我不 我不
- 战争中部队 我不的让我 我不 我不

Epoch 60. Perplexity 110.927945

- 分开 你不你我的天我 不想的让我想狂的可爱女人 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
- 不分开你的你 想你的你我不 的可后 我不能我的你 我不着你的你 我不想你你不你 不想的我的天我 你想的我的天我
- 战争中部队 你不你不你我 不想的让我想狂的可爱女人 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

Epoch 80. Perplexity 62.446662

Epoch 100. Perplexity 35.781687

- 分开 我不要再你 我不要你想 我不能再想 我不 我 不 我不要再想 我不要你想 我不要再想 我不要再想 我不能再想
- 不分开你爱 我不你再想 我不要再想 我不能再想 我不 我 不 我不要再爱你 你不着你不 我不能你不 我不能再想 我
- 战争中部队 爱要你不去 有一种味道 我不能你不 我不能再想 我不能再想 我不能再想 我不能再想 我不能再想 我

Epoch 120. Perplexity 22.092886

- 分开 我想想再你 我不 你不 我不能 爱你走的太快 像话去 半不是你在想久 说你 我想想 你爱 你时我说你要的太
- 不分开多爱 想不你的茶 有一种味道叫做家 他爷泡的茶 有一种味道叫做家 他爷泡的茶 有一种味道叫做家 他爷泡的茶
- 战争中部队的手 想我 你去 我们 再不 我不 我不能 爱情走的太快就像龙卷风 不能开 我想你的太快就像龙卷 深埋

Epoch 140. Perplexity 14.345169

- 分开 是你的没旧 我想 你想 我不要 爱你走 太九怎么 在我的一场 银我的红 装水了空 在有配空 你的那空 不了
- 不分开多爱 想不道你 想水村外的溪边 不默裂动 全世 有什么血了我 说是 你想离ㄟ了我 这是 你想很ㄟ了我 这是
- 战争中部队的爱言 我的世界已狂了都 不要我也你 最着我不支你是你的平掌 我想你这爱 我说 这想 我不能 爱情走

Epoch 160. Perplexity 9.692812

- 分开 这不是你不想 我想想 你爱人 不步 对非忿存着永 干去 你想很久了我 想知 你想离水ㄟ鱼 放抹 你想离ㄟㄟ
- 不分开多走 想那你过 在小村外的溪边 让默等待 娘子 有什么存了我 不知 你想很久了鱼？ 我不想没远你 说不着你
- 战争中部队的见 想我 你去是 后炭 一直半 木炭 一么ㄟ步都车的响尾 弦在你想想很久要 也要太多样你 说连了

Epoch 180. Perplexity 7.018299

- 分开 这这不觉忆 是我有 说想我 别地好的信片 还有是 瞎 这不是我说 这你不要 我想不那我 你的手空 是你想脚
- 不分开多走 这不了我 想水那安操人加 连在那里 我有想你 你你已那条 我没有你不想 你不想你开你 经说你 分子
- 战争中部队力爱想要 我给会没有天份 为静为没 我就是一条 (的茶人 我们是你我妈你 你在我想你 我说儿再些 我想

Epoch 200. Perplexity 5.365317

- 分开 这不的好旧看了我的那堂 你在一你的人袋说 也怪我也了子着 那晚回里你 有想 我想 我不开 不要我的没快
- 不分开多走 想要你在去是我妈要我 眼我的天里在 周不想不多 我爱你这不要 想你想要 你已经离开我 不知不觉 我
- 战争中部队太多的 你在那陪我的让后我 爱不有 在表情 娘子的公式我学不来 (难道这不是我要的天堂景象 沉沦假象)

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

6.1.11 结论

通过隐藏状态，循环神经网络能够更好的使用数据里的时序信息。

6.1.12 练习

- 调整参数（例如数据集大小、序列长度和学习率），看看对 Perplexity 和预测的结果造成的影响。
- 在随机批量采样中，如果在同一个 epoch 中只把隐藏状态在该 epoch 开始的时候初始化会怎么样？

[吐槽和讨论欢迎点这里](#)

6.2 通过时间反向传播

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

我们在[正向传播和反向传播](#)中以 L_2 范数正则化的[多层感知机](#)为例，介绍了深度学习模型梯度的计算和存储。事实上，所谓通过时间反向传播只是反向传播在循环神经网络的具体应用。我们只需将循环神经网络按时间展开，从而得到模型变量和参数之间的依赖关系，并依据链式法则应用反向传播计算梯度。

为了解释通过时间反向传播，我们以一个简单的循环神经网络为例。

6.2.1 模型定义

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

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

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

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

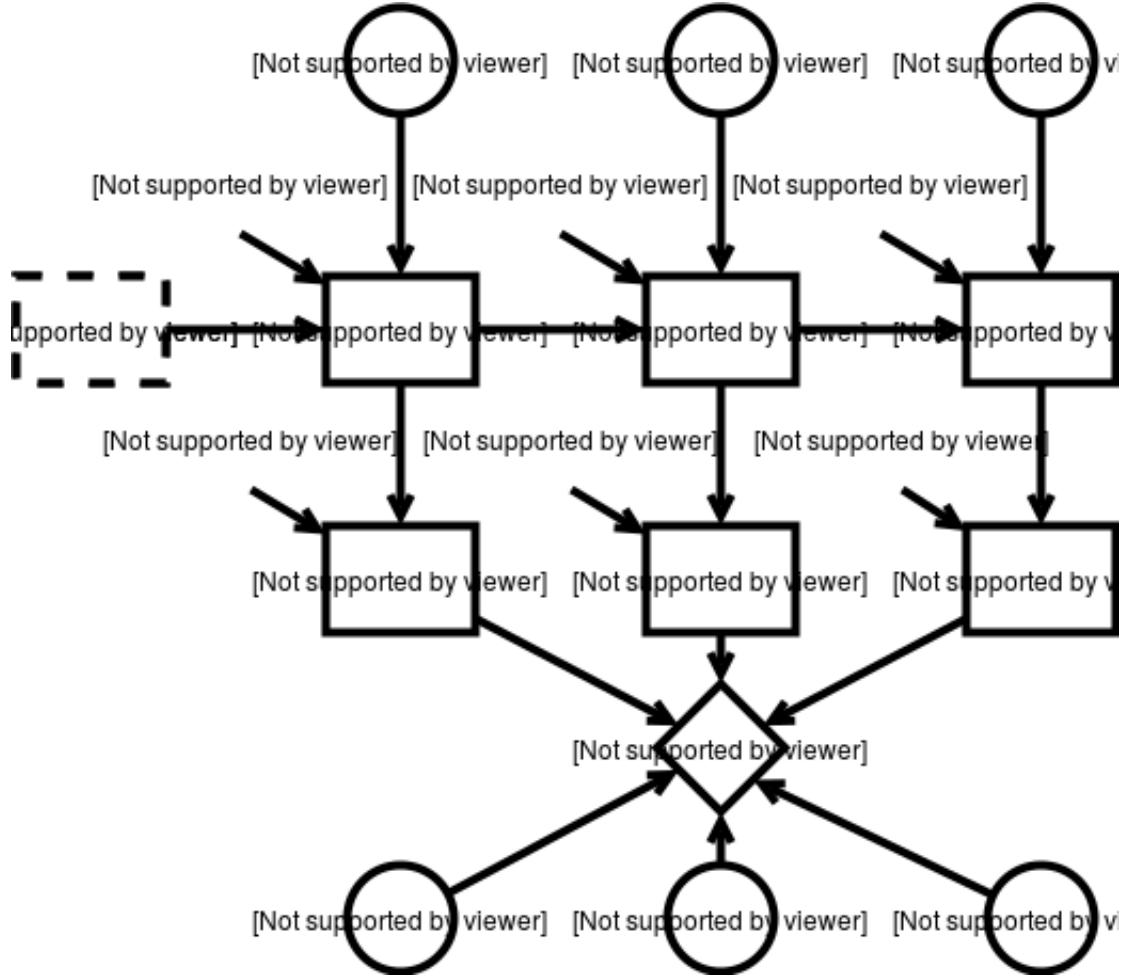
给定每个时刻损失函数计算公式 ℓ ，长度为 T 的整个时序数据的损失函数 L 定义为

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

这也是模型最终需要被优化的目标函数。

计算图

为了可视化模型变量和参数之间在计算中的依赖关系, 我们可以绘制计算图。我们以时序长度 $T = 3$ 为例。



梯度的计算与存储

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

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

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

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

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

为了表述方便，对输入输出 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。以下依次计算得到的梯度将依次被存储。

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

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

事实上，这时我们已经可以计算目标函数有关模型参数 \mathbf{W}_{yh} 的梯度 $\partial L / \partial \mathbf{W}_{yh} \in \mathbb{R}^{y \times h}$ 。需要注意的是，在计算图中， \mathbf{W}_{yh} 可以经过 $\mathbf{o}_1, \dots, \mathbf{o}_T$ 通向 L ，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，我们注意到隐含层变量之间也有依赖关系。对于最终时刻 T ，在计算图中，隐含层变量 \mathbf{h}_T 只经过 \mathbf{o}_T 通向 L 。因此我们先计算目标函数有关最终时刻隐含层变量的梯度 $\partial L / \partial \mathbf{h}_T \in \mathbb{R}^h$ 。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，对于时刻 $t < T$ ，在计算图中，由于 \mathbf{h}_t 可以经过 \mathbf{h}_{t+1} 和 \mathbf{o}_t 通向 L ，依据链式法则，目标函数有关隐含层变量的梯度 $\partial L / \partial \mathbf{h}_t \in \mathbb{R}^h$ 需要按照时刻从晚到早依次计算：

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

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

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

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

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

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

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

在正向传播和反向传播中我们解释过，每次迭代中，上述各个依次计算出的梯度会被依次存储或更新。这是为了避免重复计算。例如，由于输出层变量梯度 $\partial L / \partial \mathbf{h}_t$ 被计算存储，反向传播稍后的参数梯度 $\partial L / \partial \mathbf{W}_{hx}$ 和隐含层变量梯度 $\partial L / \partial \mathbf{W}_{hh}$ 的计算可以直接读取输出层变量梯度的值，而无需重复计算。

还有需要注意的是，反向传播对于各层中变量和参数的梯度计算可能会依赖通过正向传播计算出的各层变量和参数的当前值。举例来说，参数梯度 $\partial L / \partial \mathbf{W}_{hh}$ 的计算需要依赖隐含层变量在时刻 $t = 1, \dots, T - 1$ 的当前值 \mathbf{h}_t (\mathbf{h}_0 是初始化得到的)。这个当前值是通过从输入层到输出层的正向传播计算并存储得到的。

总结

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

练习

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

吐槽和讨论欢迎点[这里](#)

优化算法

7.1 优化算法概述

你也许会对我们用很大篇幅来介绍优化算法感到奇怪。如果你一直按照本教程的顺序读到这里，你很可能已经用了优化算法来训练模型并学出模型的参数，例如在训练模型时不断迭代参数以最小化损失函数。

优化算法对于深度学习十分重要。首先，实际中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。而优化算法的效率直接影响模型训练效率。其次，深刻理解各种优化算法的原理以及其中各参数的意义将可以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一些常用优化算法。



7.1.1 优化与机器学习

在一个机器学习的问题中，我们会预先定义一个损失函数，然后用优化算法来最小化这个损失函数。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**。依据惯例，优化算法通常只考虑最小化目标函数。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符号翻转一下。

在机器学习中，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。因此，优化往往对应降低训练误差。而机器学习的主要目标在于降低泛化误差，例如应用一些应对过拟合的技巧。在本章针对优化算法的实验中，我们只关注优化算法在最小化目标函数上的表现。

7.1.2 优化问题的挑战

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显示解（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。我们讨论的优化都是这类基于数值方法的算法。

以下我们列举优化问题中的两个挑战：局部最小值和鞍点。

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。如果 $f(x)$ 在 x 上的值是其在整个定义域上的最小值，那么 $f(x)$ 是全局最小值。

下图中展示了函数

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

的局部最小值和全局最小值。

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt

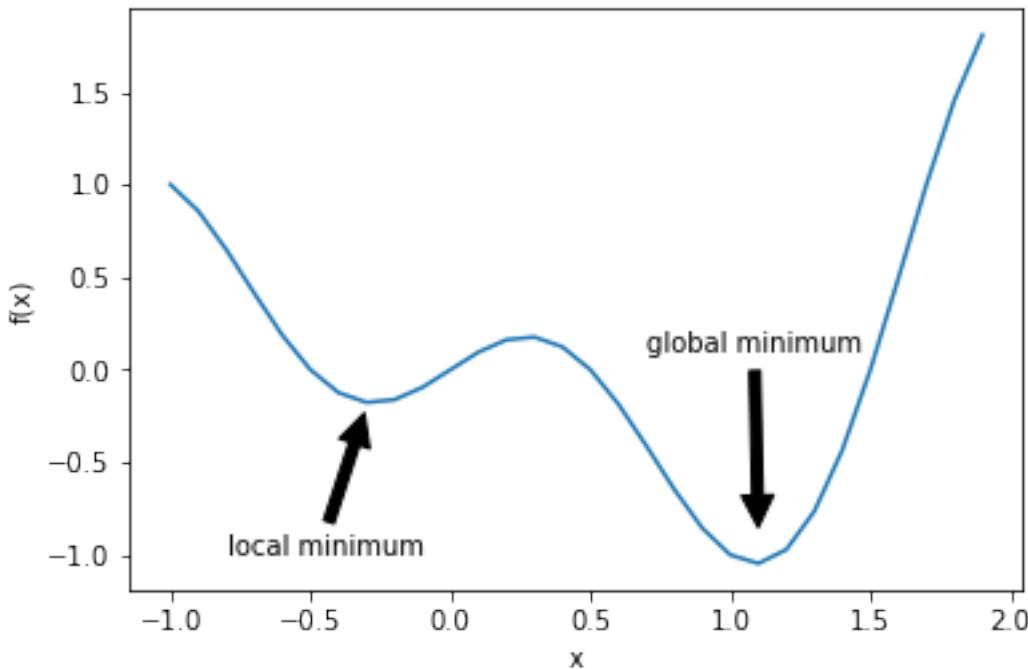
        def f(x):
            return x * np.cos(np.pi * x)

        x = np.arange(-1.0, 2.0, 0.1)
        fig = plt.figure()
        subplot = fig.add_subplot(111)
        subplot.annotate('local minimum', xy=(-0.3, -0.2), xytext=(-0.8, -1.0),
                         arrowprops=dict(facecolor='black', shrink=0.05))
```

```

subplt.annotate('global minimum', xy=(1.1, -0.9), xytext=(0.7, 0.1),
                arrowprops=dict(facecolor='black', shrink=0.05))
plt.plot(x, f(x))
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()

```



绝大多数深度学习的目标函数有若干局部最优值。当一个优化问题的数值解在局部最优解附近时，由于梯度接近或变成零，最终得到的数值解可能只令目标函数局部最小化而非全局最小化。

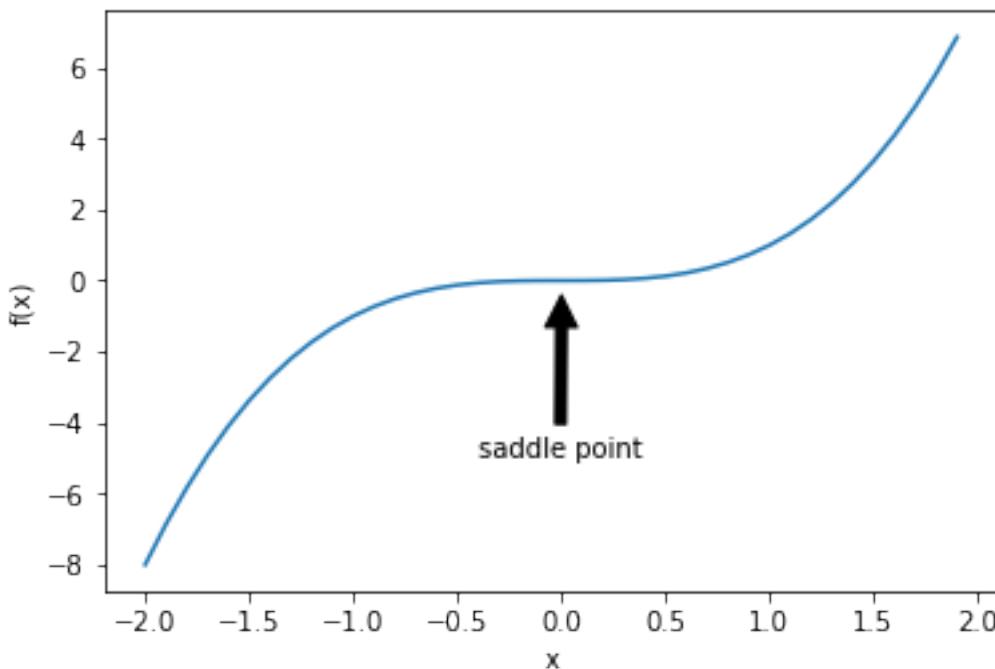
鞍点

刚刚我们提到梯度接近或变成零可能是由于当前解在局部最优解附近所造成的。事实上，另一种可能性是当前解在鞍点附近。下图展示了定义在一维空间的函数 $f(x) = x^3$ 的鞍点。

```

In [2]: x = np.arange(-2.0, 2.0, 0.1)
fig = plt.figure()
subplt = fig.add_subplot(111)
subplt.annotate('saddle point', xy=(0, -0.2), xytext=(-0.4, -5.0),
                arrowprops=dict(facecolor='black', shrink=0.05))
plt.plot(x, x**3)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()

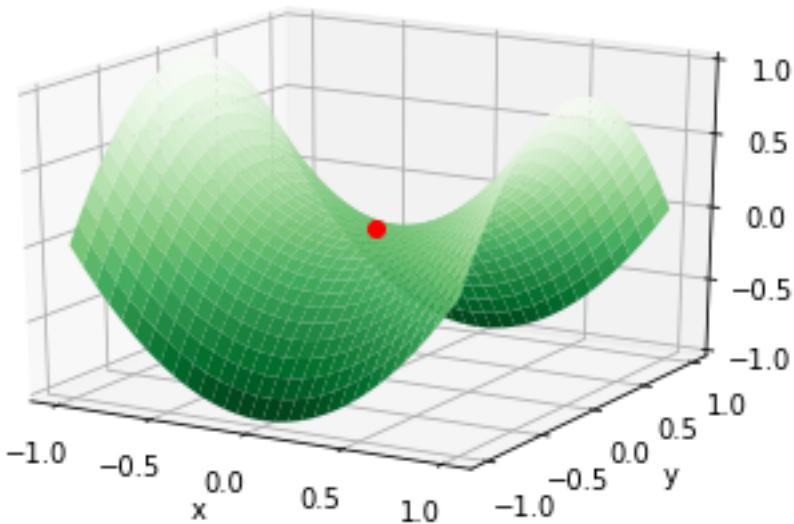
```



下图展示了定义在二维空间的函数 $f(x, y) = x^2 - y^2$ 的鞍点。该函数看起来像一个马鞍，鞍点恰好是可以坐的区域的中心。

```
In [3]: import numpy as np
        from mpl_toolkits.mplot3d import Axes3D
        import matplotlib.pyplot as plt

        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        x, y = np.mgrid[-1:1:31j, -1:1:31j]
        z = x**2 - y**2
        ax.plot_surface(x, y, z, **{'rstride': 1, 'cstride': 1, 'cmap': "Greens_r"})
        ax.plot([0], [0], [0], 'ro')
        ax.view_init(azim=-60, elev=20)
        plt.xticks([-1, -0.5, 0, 0.5, 1])
        plt.yticks([-1, -0.5, 0, 0.5, 1])
        ax.set_zticks([-1, -0.5, 0, 0.5, 1])
        plt.xlabel('x')
        plt.ylabel('y')
        plt.show()
```



事实上，由于大多数深度学习模型参数都是高维的，目标函数的鞍点往往比局部最小值更常见。

7.1.3 结论

深度学习中，虽然找到目标函数的全局最优解很难，但这并非必要。我们将在接下来的章节中逐一介绍深度学习中常用的优化算法，它们在很多实际问题中都训练出了十分有效的深度学习模型。

7.1.4 练习

- 你还能想到哪些深度学习中的优化问题的挑战？

吐槽和讨论欢迎点[这里](#)

7.2 梯度下降和随机梯度下降—从 0 开始

在之前的教程里，我们通过损失函数 \mathcal{L} 中参数的梯度 $\nabla_{\theta}\mathcal{L}$ 来决定如何更新模型 θ 的参数。我们也提到过学习率 η ，并给出了使用梯度下降算法更新模型参数的步骤：

$$\theta_t \leftarrow \theta_{t-1} - \eta \nabla_{\theta}\mathcal{L}_{t-1}$$

在本节教程中，我们将详细介绍梯度下降算法和随机梯度下降算法。由于梯度下降是优化算法的核心部分，深刻理解梯度的意义十分重要。为了帮助大家深刻理解梯度，我们将从数学上阐释梯度下降的意义。

7.2.1 一维梯度下降

我们先以简单的一维梯度下降为例，解释梯度下降算法可以降低目标函数值的原因。一维梯度是一个标量，也称导数。

假设函数 $f : \mathbb{R} \rightarrow \mathbb{R}$ 的输入和输出都是标量。根据泰勒展开公式，我们得到

$$f(x + \epsilon) \approx f(x) + f'(x)\epsilon$$

假设 η 是一个常数，将 ϵ 替换为 $-\eta f'(x)$ 后，我们有

$$f(x - \eta f'(x)) \approx f(x) - \eta f'(x)^2$$

如果 η 是一个很小的正数，那么

$$f(x - \eta f'(x)) \leq f(x)$$

也就是说，如果当前导数 $f'(x) \neq 0$ ，按照 $x := x - \eta f'(x)$ 更新 x 可能降低 $f(x)$ 的值。

由于导数 $f'(x)$ 是梯度在一维空间的特殊情况，上述更新 x 的方法也即一维空间的梯度下降。一维空间的梯度下降如下图所示，参数 x 沿着梯度方向不断更新。

7.2.2 学习率

上述梯度下降算法中的 η （取正数）叫做学习率或步长。需要注意的是，学习率过大可能会造成 x 迈过（overshoot）最优解，甚至不断发散而无法收敛，如下图所示。

然而，如果学习率过小，优化算法收敛速度会过慢。实际中，一个合适的学习率通常是需要通过实验调出来的。

7.2.3 多维梯度下降

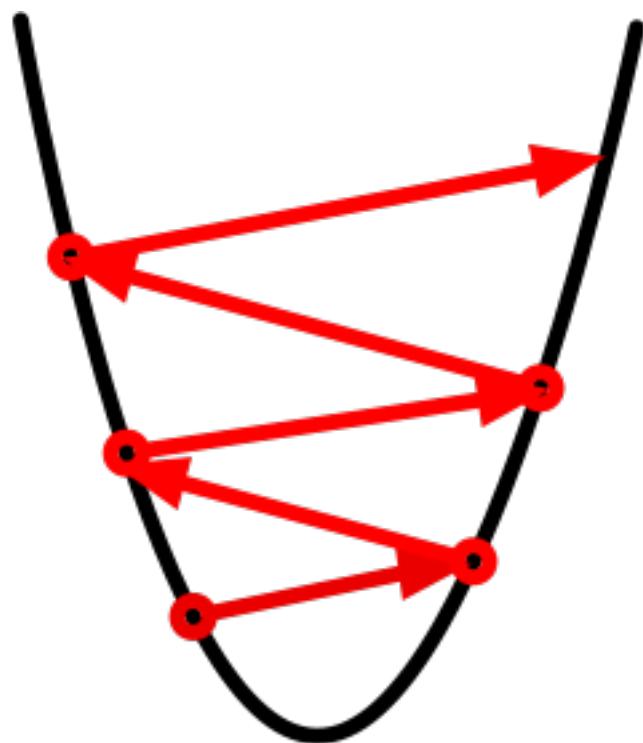
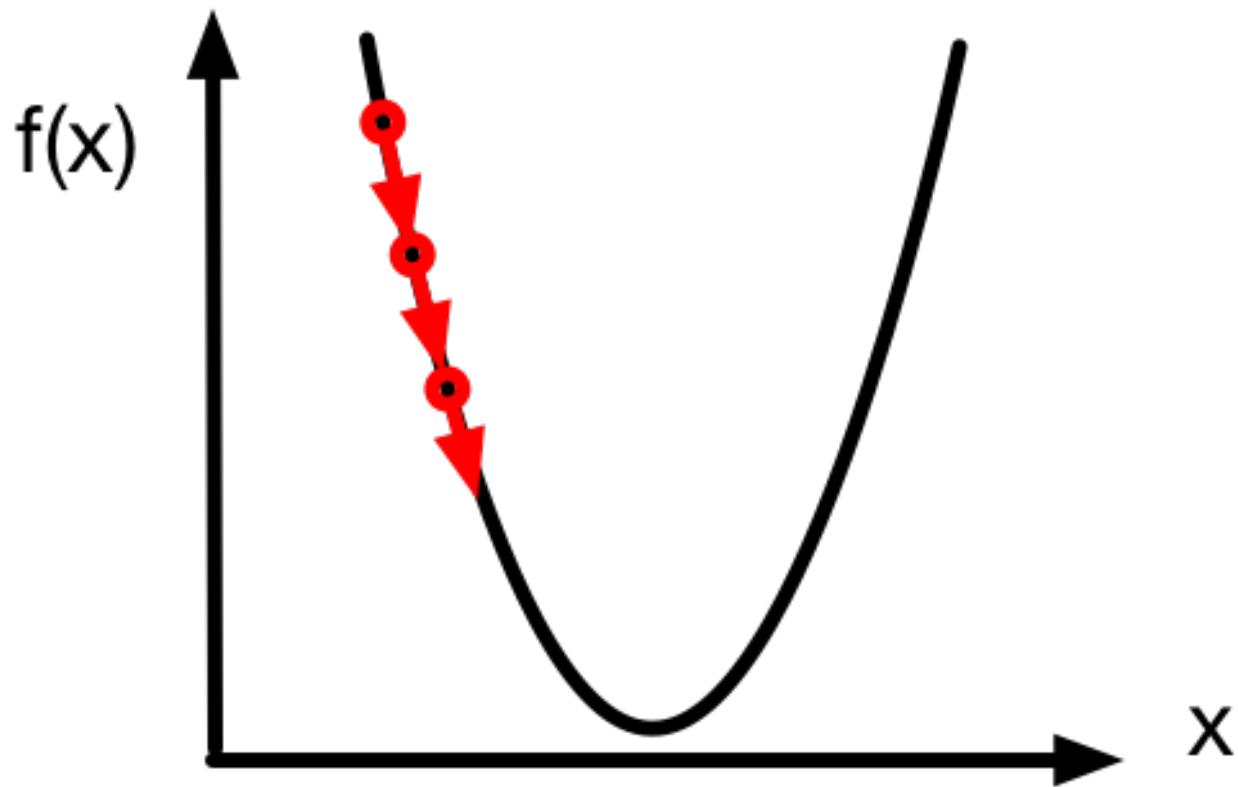
现在我们考虑一个更广义的情况：目标函数的输入为向量，输出为标量。

假设目标函数 $f : \mathbb{R}^d \rightarrow \mathbb{R}$ 的输入是一个多维向量 $\mathbf{x} = [x_1, x_2, \dots, x_d]^\top$ 。目标函数 $f(\mathbf{x})$ 有关 \mathbf{x} 的梯度是一个由偏导数组成的向量：

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_d} \right]^\top.$$

为表示简洁，我们有时用 $\nabla f(\mathbf{x})$ 代替 $\nabla_{\mathbf{x}} f(\mathbf{x})$ 。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

$$D_{\mathbf{u}} f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{u}) - f(\mathbf{x})}{h}$$



由链式法则，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 \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 $\nabla f(\mathbf{x})$ 和 \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：

$$\mathbf{x} := \mathbf{x} - \eta \nabla f(\mathbf{x})$$

相同地，其中 η （取正数）称作学习率或步长。

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 的训练数据点的损失函数。需要强调的是，梯度下降每次迭代的计算开销随着 n 线性增长。因此，当 n 很大时，每次迭代的计算开销很高。

这时我们需要**随机梯度下降算法**。在每次迭代时，该算法随机均匀采样 i 并计算 $\nabla f_i(\mathbf{x})$ 。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，每次迭代可以随机均匀采样一个由训练数据点索引所组成的小批量 \mathcal{B} 。类似地，我们可以使用

$$\nabla f_{\mathcal{B}}(\mathbf{x}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla f_i(\mathbf{x})$$

来更新 \mathbf{x} ：

$$\mathbf{x} := \mathbf{x} - \eta \nabla f_{\mathcal{B}}(\mathbf{x}),$$

其中 $|\mathcal{B}|$ 代表批量中索引数量, η (取正数) 称作学习率或步长。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 $\mathcal{O}(|\mathcal{B}|)$ 。因此, 当批量较小时, 每次迭代的计算开销也较小。

7.2.6 算法实现和实验

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

实验中, 我们以线性回归为例。其中真实参数 w 为 $[2, -3.4]$, b 为 4.2。

```
In [2]: import mxnet as mx
from mxnet import autograd
from mxnet import ndarray as nd
from mxnet import gluon
import random

mx.random.seed(1)
random.seed(1)

# 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
X = nd.random_normal(scale=1, shape=(num_examples, num_inputs))
y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
y += .01 * nd.random_normal(scale=1, shape=y.shape)
dataset = gluon.data.ArrayDataset(X, y)

# 构造迭代器。
import random
def data_iter(batch_size):
    idx = list(range(num_examples))
```

```
random.shuffle(idx)
for batch_i, i in enumerate(range(0, num_examples, batch_size)):
    j = nd.array(idx[i: min(i + batch_size, num_examples)])
    yield batch_i, X.take(j), y.take(j)

# 初始化模型参数。
def init_params():
    w = nd.random_normal(scale=1, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    for param in params:
        param.attach_grad()
    return params

# 线性回归模型。
def net(X, w, b):
    return nd.dot(X, w) + b

# 损失函数。
def square_loss(yhat, y):
    return (yhat - y.reshape(yhat.shape)) ** 2 / 2
```

接下来定义训练函数。当 epoch 大于 2 时 (epoch 从 1 开始计数)，学习率以自乘 0.1 的方式自我衰减。训练函数的 period 参数说明，每次采样过该数目的数据点后，记录当前目标函数值用于作图。例如，当 period 和 batch_size 都为 10 时，每次迭代后均会记录目标函数值。

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

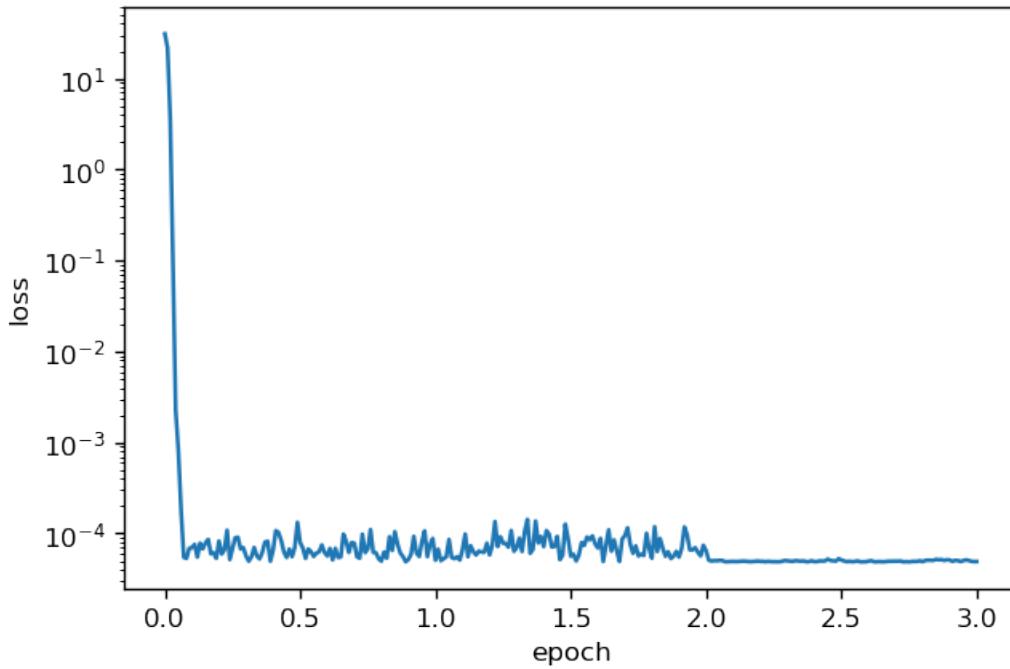
def train(batch_size, lr, epochs, period):
    assert period >= batch_size and period % batch_size == 0
    w, b = init_params()
    total_loss = [np.mean(square_loss(net(X, w, b), y).asnumpy())]
    # 注意 epoch 从 1 开始计数。
    for epoch in range(1, epochs + 1):
        # 学习率自我衰减。
        if epoch > 2:
            lr *= 0.1
        for batch_i, data, label in data_iter(batch_size):
```

```
with autograd.record():
    output = net(data, w, b)
    loss = square_loss(output, label)
    loss.backward()
    sgd([w, b], lr, batch_size)
if batch_i * batch_size % period == 0:
    total_loss.append(
        np.mean(square_loss(net(X, w, b), y).asnumpy()))
print("Batch size %d, Learning rate %f, Epoch %d, loss %.4e" %
      (batch_size, lr, epoch, total_loss[-1]))
print('w:', np.reshape(w.asnumpy(), (1, -1)),
      'b:', b.asnumpy()[0], '\n')
x_axis = np.linspace(0, epochs, len(total_loss), endpoint=True)
plt.semilogy(x_axis, total_loss)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

当批量大小为 1 时，训练使用的是随机梯度下降。在当前学习率下，目标函数值在早期快速下降后略有波动。当 epoch 大于 2，学习率自我衰减后，目标函数值下降后较平稳。最终学到的参数值与真实值较接近。

In [4]: train(batch_size=1, lr=0.2, epochs=3, period=10)

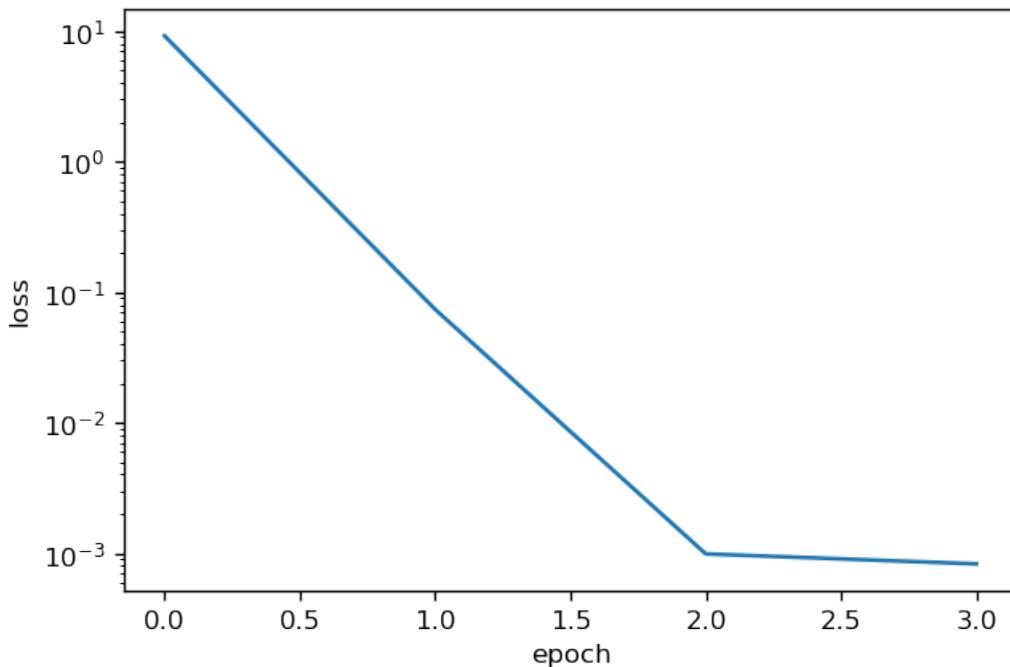
```
Batch size 1, Learning rate 0.200000, Epoch 1, loss 5.0910e-05
Batch size 1, Learning rate 0.200000, Epoch 2, loss 6.4832e-05
Batch size 1, Learning rate 0.020000, Epoch 3, loss 4.9259e-05
w: [[ 2.00118256 -3.4002037 ]] b: 4.19923
```



当批量大小为 1000 时, 由于训练数据集含 1000 个样本, 此时训练使用的是梯度下降。在当前学习率下, 目标函数值在前两个 epoch 下降较快。当 epoch 大于 2, 学习率自我衰减后, 目标函数值下降较慢。最终学到的参数值与真实值较接近。

```
In [5]: train(batch_size=1000, lr=0.999, epochs=3, period=1000)
```

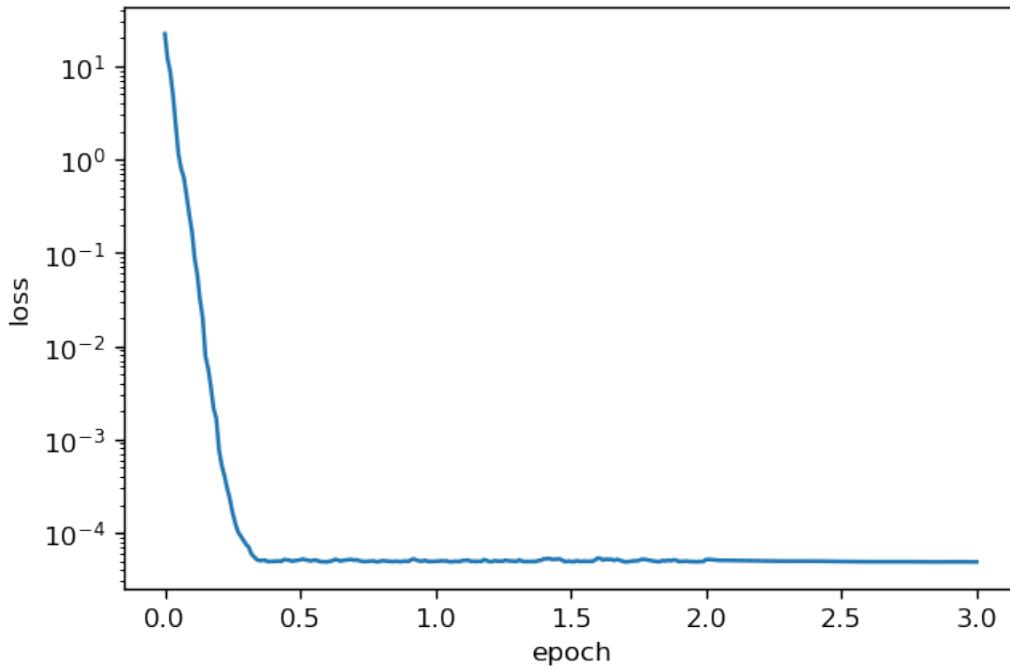
```
Batch size 1000, Learning rate 0.999000, Epoch 1, loss 7.3842e-02
Batch size 1000, Learning rate 0.999000, Epoch 2, loss 9.8761e-04
Batch size 1000, Learning rate 0.099900, Epoch 3, loss 8.2789e-04
w: [[ 2.01470947 -3.37011194]] b: 4.17426
```



当批量大小为 10 时, 由于训练数据集含 1000 个样本, 此时训练使用的是小批量随机梯度下降。最终学到的参数值与真实值较接近。

```
In [6]: train(batch_size=10, lr=0.2, epochs=3, period=10)
```

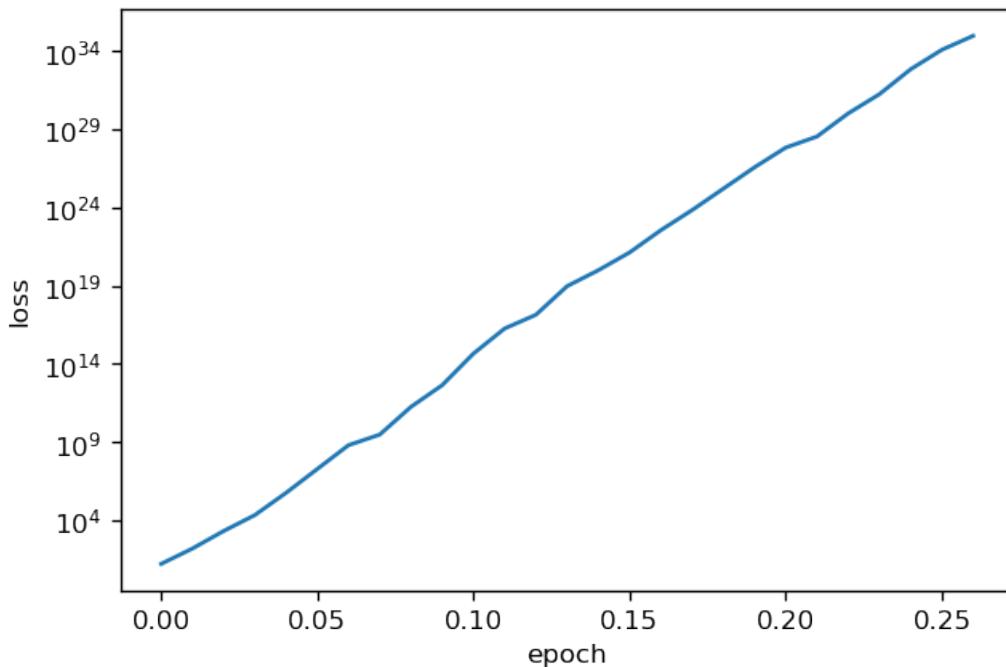
```
Batch size 10, Learning rate 0.200000, Epoch 1, loss 4.9434e-05
Batch size 10, Learning rate 0.200000, Epoch 2, loss 5.1733e-05
Batch size 10, Learning rate 0.020000, Epoch 3, loss 4.8792e-05
w: [[ 2.00055027 -3.40005398]] b: 4.20035
```



同样是批量大小为 10，我们把学习率改大。这时我们观察到目标函数值不断增大。这时典型的 overshooting 问题。

In [7]: `train(batch_size=10, lr=5, epochs=3, period=10)`

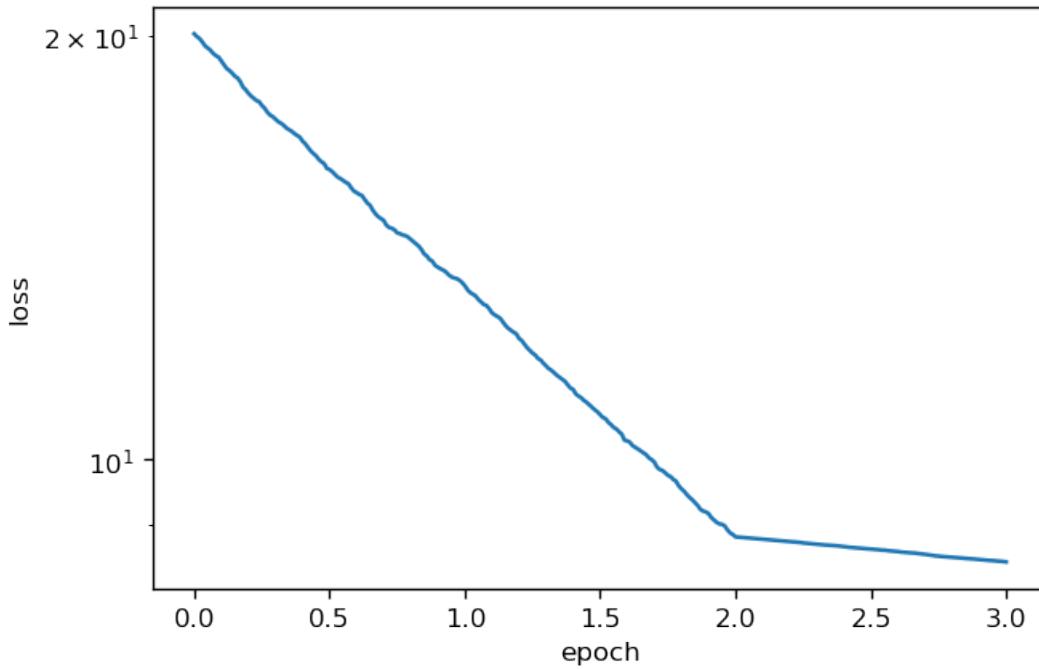
```
Batch size 10, Learning rate 5.000000, Epoch 1, loss nan
Batch size 10, Learning rate 5.000000, Epoch 2, loss nan
Batch size 10, Learning rate 0.500000, Epoch 3, loss nan
w: [[ nan  nan]] b: nan
```



同样是批量大小为 10，我们把学习率改小。这时我们观察到目标函数值下降较慢，直到 3 个 epoch 也没能得到接近真实值的解。

```
In [8]: train(batch_size=10, lr=0.002, epochs=3, period=10)
```

```
Batch size 10, Learning rate 0.002000, Epoch 1, loss 1.3283e+01
Batch size 10, Learning rate 0.002000, Epoch 2, loss 8.8150e+00
Batch size 10, Learning rate 0.000200, Epoch 3, loss 8.4618e+00
w: [[ 0.30856001 -0.91306931]] b: 1.43488
```



7.2.7 结论

- 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。合适的学习率要靠实验来调。

7.2.8 练习

- 为什么实验中随机梯度下降的学习率是自我衰减的？
- 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？

吐槽和讨论欢迎点[这里](#)

7.3 梯度下降和随机梯度下降—使用 Gluon

在 Gluon 里，使用小批量随机梯度下降很容易。我们无需重新实现该算法。特别地，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]: import mxnet as mx
        from mxnet import autograd
        from mxnet import gluon
        from mxnet import ndarray as nd
```

```

import numpy as np
import random

mx.random.seed(1)
random.seed(1)

# 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
X = nd.random_normal(scale=1, shape=(num_examples, num_inputs))
y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
y += .01 * nd.random_normal(scale=1, shape=y.shape)
dataset = gluon.data.ArrayDataset(X, y)

net = gluon.nn.Sequential()
net.add(gluon.nn.Dense(1))
square_loss = gluon.loss.L2Loss()

```

为了使学习率在两个 epoch 后自我衰减，我们需要访问 `gluon.Trainer` 的 `learning_rate` 属性和 `set_learning_rate` 函数。

```

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

def train(batch_size, lr, epochs, period):
    assert period >= batch_size and period % batch_size == 0
    net.collect_params().initialize(mx.init.Normal(sigma=1), force_reinit=True)
    # 随机梯度下降。
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': lr})
    data_iter = gluon.data.DataLoader(dataset, batch_size, shuffle=True)
    total_loss = [np.mean(square_loss(net(X), y).asnumpy())]
    for epoch in range(1, epochs + 1):
        # 学习率自我衰减。
        if epoch > 2:
            trainer.set_learning_rate(trainer.learning_rate * 0.1)
        for batch_i, (data, label) in enumerate(data_iter):
            with autograd.record():

```

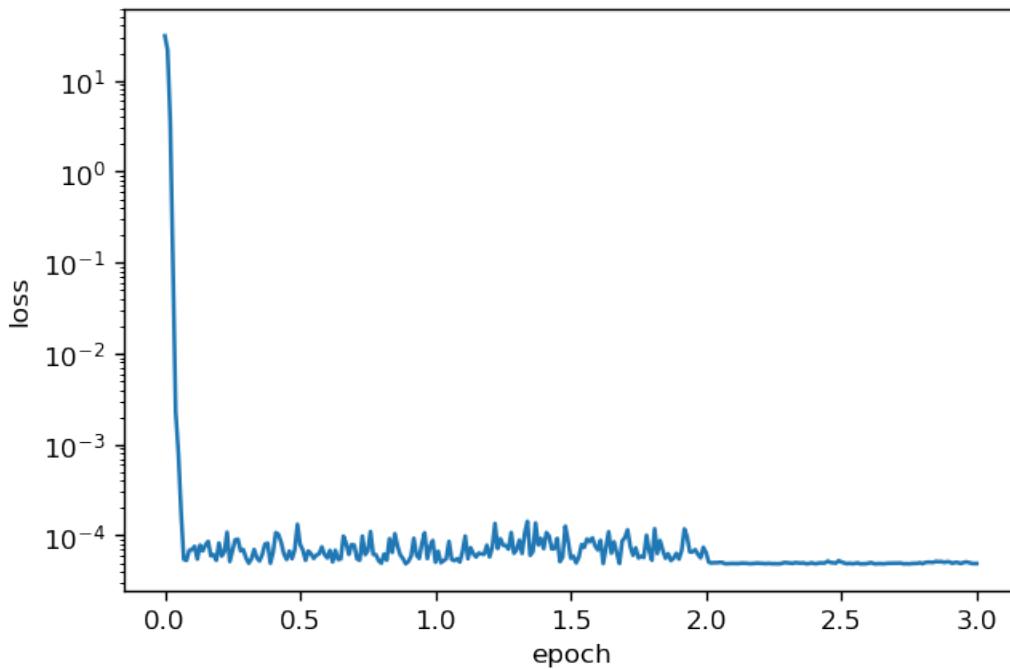
```
        output = net(data)
        loss = square_loss(output, label)
        loss.backward()
        trainer.step(batch_size)
        if batch_i * batch_size % period == 0:
            total_loss.append(np.mean(square_loss(net(X), y).asnumpy()))
        print("Batch size %d, Learning rate %f, Epoch %d, loss %.4e" %
              (batch_size, trainer.learning_rate, epoch, total_loss[-1]))

print('w:', np.reshape(net[0].weight.data().asnumpy(), (1, -1)),
      'b:', net[0].bias.data().asnumpy()[0], '\n')
x_axis = np.linspace(0, epochs, len(total_loss), endpoint=True)
plt.semilogy(x_axis, total_loss)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

当批量大小为 1 时，训练使用的是随机梯度下降。在当前学习率下，目标函数值在早期快速下降后略有波动。当 epoch 大于 2，学习率自我衰减后，目标函数值下降后较平稳。最终学到的参数值与真实值较接近。

In [3]: train(batch_size=1, lr=0.2, epochs=3, period=10)

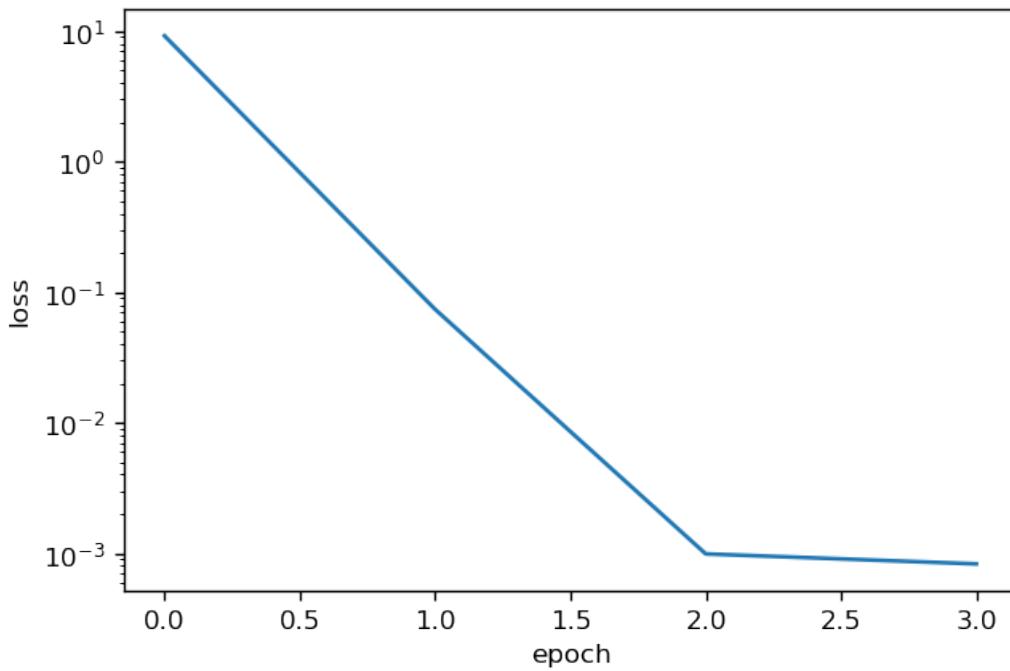
```
Batch size 1, Learning rate 0.200000, Epoch 1, loss 5.0910e-05
Batch size 1, Learning rate 0.200000, Epoch 2, loss 6.4832e-05
Batch size 1, Learning rate 0.020000, Epoch 3, loss 4.9259e-05
w: [[ 2.00118256 -3.4002037 ]] b: 4.19923
```



当批量大小为 1000 时，由于训练数据集含 1000 个样本，此时训练使用的是梯度下降。在当前学习率下，目标函数值在前两个 epoch 下降较快。当 epoch 大于 2，学习率自我衰减后，目标函数值下降较慢。最终学到的参数值与真实值较接近。

```
In [4]: train(batch_size=1000, lr=0.999, epochs=3, period=1000)
```

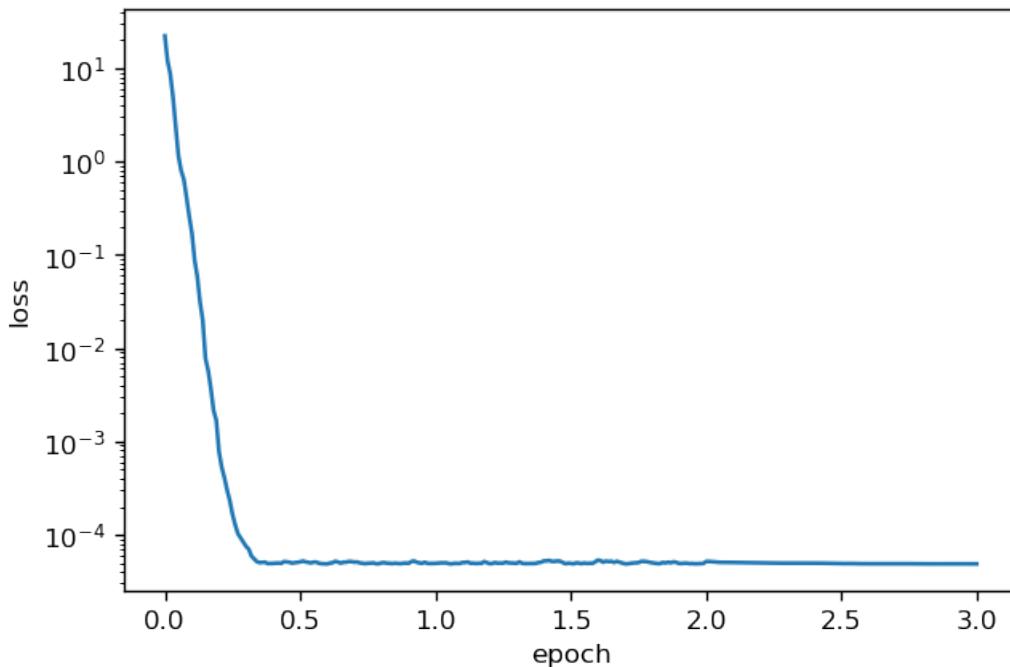
```
Batch size 1000, Learning rate 0.999000, Epoch 1, loss 7.3842e-02
Batch size 1000, Learning rate 0.999000, Epoch 2, loss 9.8761e-04
Batch size 1000, Learning rate 0.099900, Epoch 3, loss 8.2789e-04
w: [[ 2.01470947 -3.37011194]] b: 4.17426
```



当批量大小为 10 时, 由于训练数据集含 1000 个样本, 此时训练使用的是(小批量)随机梯度下降。最终学到的参数值与真实值较接近。

In [5]: `train(batch_size=10, lr=0.2, epochs=3, period=10)`

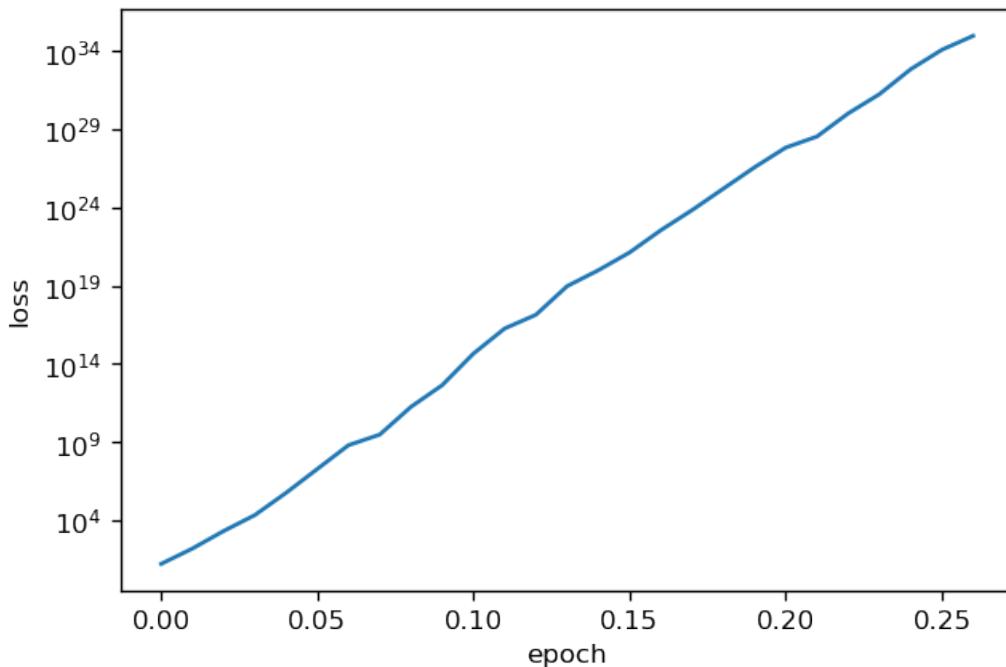
```
Batch size 10, Learning rate 0.200000, Epoch 1, loss 4.9434e-05
Batch size 10, Learning rate 0.200000, Epoch 2, loss 5.1733e-05
Batch size 10, Learning rate 0.020000, Epoch 3, loss 4.8792e-05
w: [[ 2.00055027 -3.40005398]] b: 4.20035
```



同样是批量大小为 10，我们把学习率改大。这时我们观察到目标函数值不断增大。这时典型的 overshooting 问题。

```
In [6]: train(batch_size=10, lr=5, epochs=3, period=10)
```

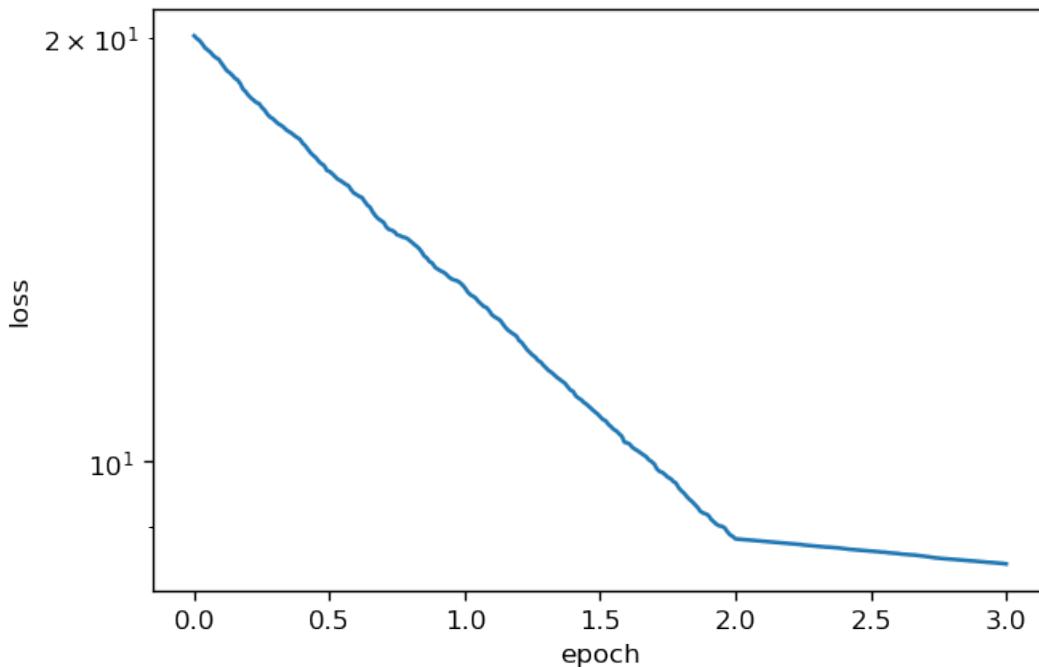
```
Batch size 10, Learning rate 5.000000, Epoch 1, loss nan
Batch size 10, Learning rate 5.000000, Epoch 2, loss nan
Batch size 10, Learning rate 0.500000, Epoch 3, loss nan
w: [[ nan  nan]] b: nan
```



同样是批量大小为 10，我们把学习率改小。这时我们观察到目标函数值下降较慢，直到 3 个 epoch 也没能得到接近真实值的解。

```
In [7]: train(batch_size=10, lr=0.002, epochs=3, period=10)
```

```
Batch size 10, Learning rate 0.002000, Epoch 1, loss 1.3283e+01
Batch size 10, Learning rate 0.002000, Epoch 2, loss 8.8150e+00
Batch size 10, Learning rate 0.000200, Epoch 3, loss 8.4618e+00
w: [[ 0.30856001 -0.91306931]] b: 1.43488
```



7.3.1 结论

- 使用 Gluon 的 Trainer 可以使模型训练变得更容易。
- 使用 gluon.Trainer 的 learning_rate 属性和 set_learning_rate 函数可以随意调整学习率。

7.3.2 练习

- 你还能想到哪些学习率自我衰减的方法?

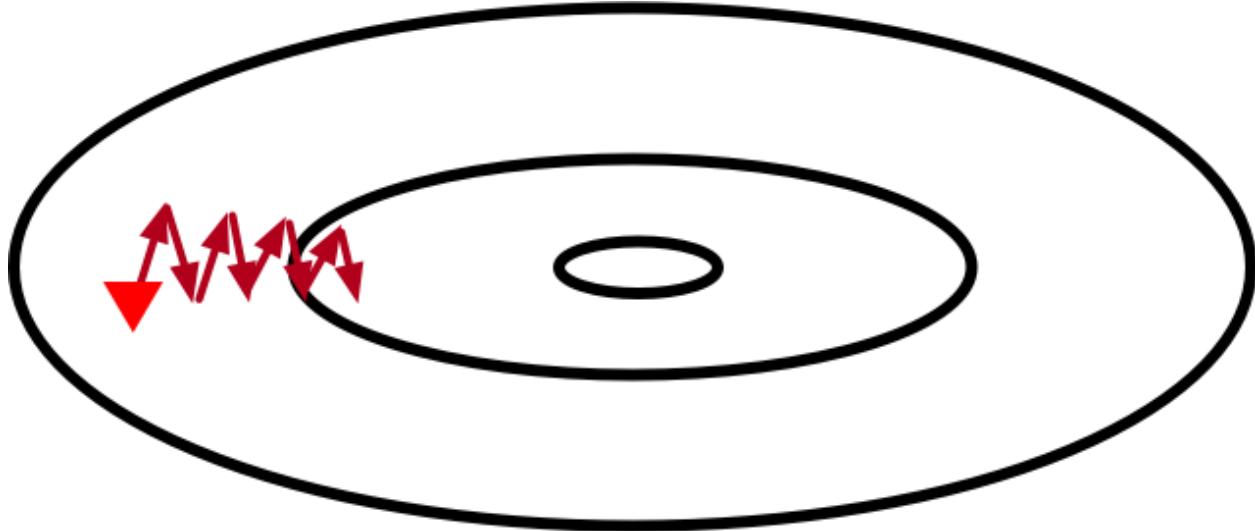
吐槽和讨论欢迎点[这里](#)

7.4 动量法—从 0 开始

在梯度下降和随机梯度下降的章节中，我们介绍了梯度下降算法：每次迭代时，该算法沿着目标函数下降最快的方向更新参数。因此，梯度下降有时也叫做最陡下降 (steepest descent)。在梯度下降中，每次更新参数的方向仅仅取决当前位置，这可能会带来一些问题。

7.4.1 梯度下降的问题

考虑一个输入为二维向量 $\mathbf{x} = [x_1, x_2]^\top$, 输出为标量的目标函数 $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ 。下面为该函数的等高线示意图（每条等高线表示相同函数值的点：越靠近中间函数值越小）。



上图中，红色三角形代表参数 \mathbf{x} 的初始值。带箭头的线段表示每次迭代时参数的更新。由于目标函数在竖直方向 (x_2 轴方向) 上比在水平方向 (x_1 轴方向) 弯曲得更厉害，梯度下降迭代参数时会使参数在竖直方向比在水平方向移动更猛烈。因此，我们需要一个较小的学习率从而避免参数在竖直方向上 overshoot。这就造成了上图中参数向最优解移动速度的缓慢。

7.4.2 动量法

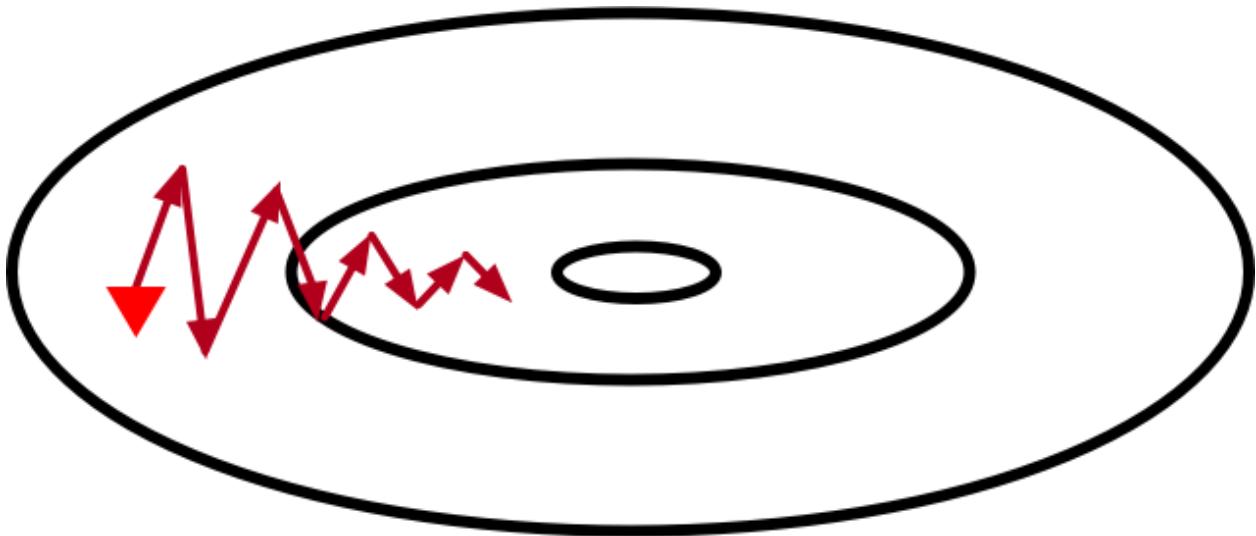
动量法的提出是为了应对梯度下降的上述问题。广义上，以小批量随机梯度下降为例（当批量大小等于训练集大小时，该算法即为梯度下降；批量大小为 1 即为随机梯度下降），我们对梯度下降和随机梯度下降的章节中的小批量随机梯度算法做如下修改：

$$\mathbf{v} := \gamma \mathbf{v} + \eta \nabla f_B(\mathbf{x}),$$

$$\mathbf{x} := \mathbf{x} - \mathbf{v},$$

其中 \mathbf{v} 是当前速度， γ 是动量参数。其余符号如学习率 η 、有关小批量 B 的随机梯度 $\nabla f_B(\mathbf{x})$ 都已在梯度下降和随机梯度下降的章节中定义，此处不再赘述。

当前速度 \mathbf{v} 的更新可以理解为对 $[\eta/(1 - \gamma)]\nabla f_B(\mathbf{x})$ 做指数加权移动平均。因此，动量法的每次迭代中，参数在各个方向上移动幅度不仅取决当前梯度，还取决过去各个梯度在各个方向上是否一致。当过去的所有梯度都在同一方向，例如都是水平向右，那么参数在水平向右的移动幅度最大。如果过去的梯度中在竖直方向上时上时下，那么参数在竖直方向的移动幅度将变小。这样，我们就可以使用较大的学习率，从而如下图收敛更快。



7.4.3 动量参数

为了有助于理解动量参数 γ , 让我们考虑一个简单的问题: 每次迭代的小批量随机梯度 $\nabla f_B(\mathbf{x})$ 都等于 \mathbf{g} 。由于所有小批量随机梯度都在同一方向, 动量法在该方向使参数移动加速:

$$\begin{aligned}\mathbf{v}_1 &:= \eta \mathbf{g}, \\ \mathbf{v}_2 &:= \gamma \mathbf{v}_1 + \eta \mathbf{g} = \eta \mathbf{g}(\gamma + 1), \\ \mathbf{v}_3 &:= \gamma \mathbf{v}_2 + \eta \mathbf{g} = \eta \mathbf{g}(\gamma^2 + \gamma + 1), \\ &\dots \\ \mathbf{v}_{\text{inf}} &:= \frac{\eta \mathbf{g}}{1 - \gamma}.\end{aligned}$$

例如, 当 $\gamma = 0.99$, 最终的速度将是学习率乘以相应小批量随机梯度 $\eta \mathbf{g}$ 的 100 倍大。

7.4.4 算法实现和实验

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项。

```
In [1]: # 动量法。
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
```

实验中, 我们以线性回归为例。其中真实参数 w 为 $[2, -3.4]$, b 为 4.2。我们把速度项初始化为和参数形状相同的零张量。

```
In [2]: import mxnet as mx
from mxnet import autograd
```

```
from mxnet import ndarray as nd
from mxnet import gluon
import random

mx.random.seed(1)
random.seed(1)

# 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
X = nd.random_normal(scale=1, shape=(num_examples, num_inputs))
y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
y += .01 * nd.random_normal(scale=1, shape=y.shape)
dataset = gluon.data.ArrayDataset(X, y)

# 构造迭代器。
import random
def data_iter(batch_size):
    idx = list(range(num_examples))
    random.shuffle(idx)
    for batch_i, i in enumerate(range(0, num_examples, batch_size)):
        j = nd.array(idx[i: min(i + batch_size, num_examples)])
        yield batch_i, X.take(j), y.take(j)

# 初始化模型参数。
def init_params():
    w = nd.random_normal(scale=1, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    vs = []
    for param in params:
        param.attach_grad()
        # 把速度项初始化为和参数形状相同的零张量。
        vs.append(param.zeros_like())
    return params, vs

# 线性回归模型。
def net(X, w, b):
    return nd.dot(X, w) + b
```

```
# 损失函数。
def square_loss(yhat, y):
    return (yhat - y.reshape(yhat.shape)) ** 2 / 2
```

接下来定义训练函数。当 epoch 大于 2 时 (epoch 从 1 开始计数), 学习率以自乘 0.1 的方式自我衰减。训练函数的 period 参数说明, 每次采样过该数目的数据点后, 记录当前目标函数值用于作图。例如, 当 period 和 batch_size 都为 10 时, 每次迭代后均会记录目标函数值。

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

def train(batch_size, lr, mom, epochs, period):
    assert period >= batch_size and period % batch_size == 0
    [w, b], vs = init_params()
    total_loss = [np.mean(square_loss(net(X, w, b), y).asnumpy())]

    # 注意 epoch 从 1 开始计数。
    for epoch in range(1, epochs + 1):
        # 重设学习率。
        if epoch > 2:
            lr *= 0.1
        for batch_i, data, label in data_iter(batch_size):
            with autograd.record():
                output = net(data, w, b)
                loss = square_loss(output, label)
            loss.backward()
            sgd_momentum([w, b], vs, lr, mom, batch_size)
            if batch_i * batch_size % period == 0:
                total_loss.append(np.mean(square_loss(net(X, w, b), y).asnumpy()))
        print("Batch size %d, Learning rate %f, Epoch %d, loss %.4e" %
              (batch_size, lr, epoch, total_loss[-1]))
    print('w:', np.reshape(w.asnumpy(), (1, -1)),
          'b:', b.asnumpy()[0], '\n')
    x_axis = np.linspace(0, epochs, len(total_loss), endpoint=True)
    plt.semilogy(x_axis, total_loss)
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.show()
```

使用动量法，最终学到的参数值与真实值较接近。

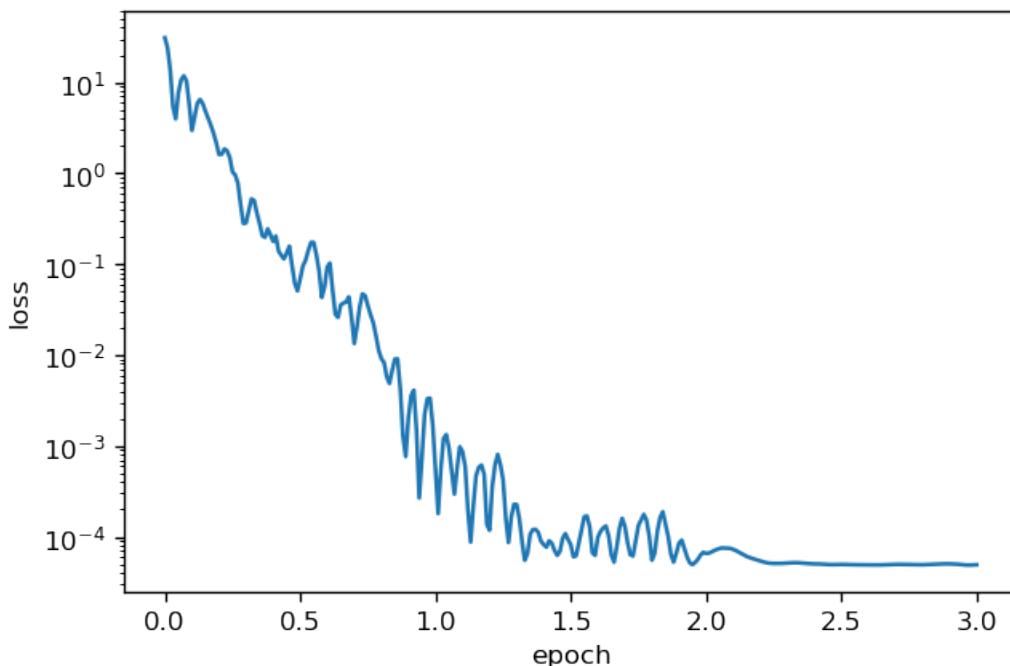
```
In [4]: train(batch_size=10, lr=0.2, mom=0.9, epochs=3, period=10)
```

```
Batch size 10, Learning rate 0.200000, Epoch 1, loss 5.5845e-04
```

```
Batch size 10, Learning rate 0.200000, Epoch 2, loss 6.5758e-05
```

```
Batch size 10, Learning rate 0.020000, Epoch 3, loss 4.9267e-05
```

```
w: [[ 2.00156975 -3.39996457]] b: 4.19986
```



7.4.5 结论

- 动量法可以提升随机梯度下降，例如对于某些问题可以选用较大学习率从而加快收敛。

7.4.6 练习

- 试着使用较小的动量参数，观察实验结果。

吐槽和讨论欢迎点[这里](#)

7.5 动量法—使用 Gluon

在 Gluon 里，使用动量法很容易。我们无需重新实现它。例如，在随机梯度下降中，我们可以定义 momentum 参数。

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

        mx.random.seed(1)
        random.seed(1)

        # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        X = nd.random_normal(scale=1, shape=(num_examples, num_inputs))
        y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
        y += .01 * nd.random_normal(scale=1, shape=y.shape)
        dataset = gluon.data.ArrayDataset(X, y)

        net = gluon.nn.Sequential()
        net.add(gluon.nn.Dense(1))
        square_loss = gluon.loss.L2Loss()
```

为了使学习率在两个 epoch 后自我衰减，我们需要访问 `gluon.Trainer` 的 `learning_rate` 属性和 `set_learning_rate` 函数。

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

        def train(batch_size, lr, mom, epochs, period):
            assert period >= batch_size and period % batch_size == 0
            net.collect_params().initialize(mx.init.Normal(sigma=1), force_reinit=True)
            # 动量法。
            trainer = gluon.Trainer(net.collect_params(), 'sgd',
```

```
        {'learning_rate': lr, 'momentum': mom})
data_iter = gluon.data.DataLoader(dataset, batch_size, shuffle=True)
total_loss = [np.mean(square_loss(net(X), y).asnumpy())]

for epoch in range(1, epochs + 1):
    # 重设学习率。
    if epoch > 2:
        trainer.set_learning_rate(trainer.learning_rate * 0.1)
    for batch_i, (data, label) in enumerate(data_iter):
        with autograd.record():
            output = net(data)
            loss = square_loss(output, label)
            loss.backward()
            trainer.step(batch_size)

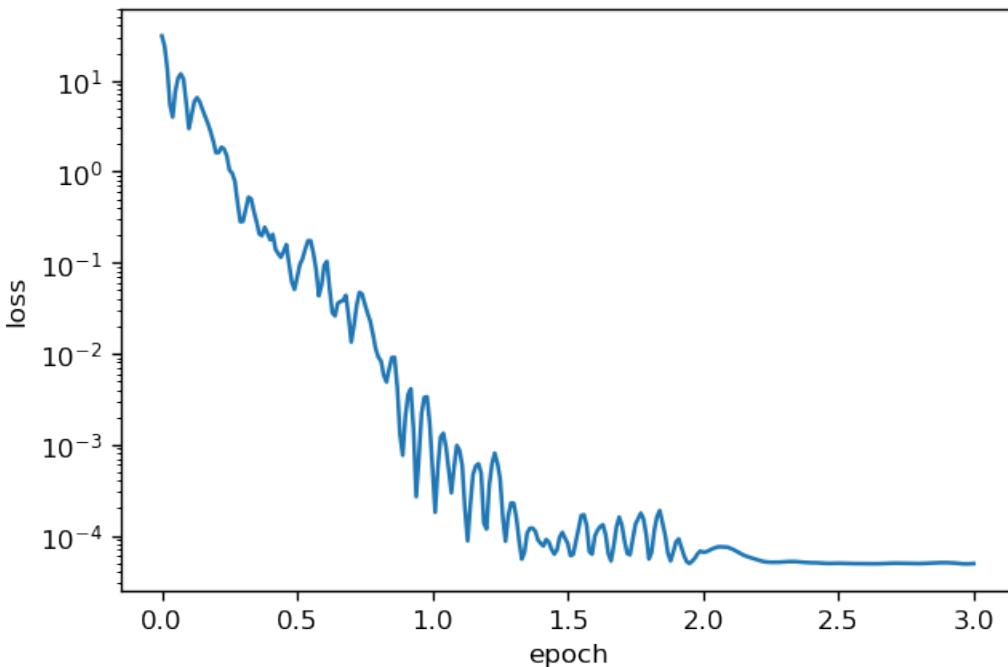
        if batch_i * batch_size % period == 0:
            total_loss.append(np.mean(square_loss(net(X), y).asnumpy()))
    print("Batch size %d, Learning rate %f, Epoch %d, loss %.4e" %
          (batch_size, trainer.learning_rate, epoch, total_loss[-1]))

print('w:', np.reshape(net[0].weight.data().asnumpy(), (1, -1)),
      'b:', net[0].bias.data().asnumpy()[0], '\n')
x_axis = np.linspace(0, epochs, len(total_loss), endpoint=True)
plt.semilogy(x_axis, total_loss)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

使用动量法，最终学到的参数值与真实值较接近。

```
In [3]: train(batch_size=10, lr=0.2, mom=0.9, epochs=3, period=10)

Batch size 10, Learning rate 0.200000, Epoch 1, loss 5.5845e-04
Batch size 10, Learning rate 0.200000, Epoch 2, loss 6.5758e-05
Batch size 10, Learning rate 0.020000, Epoch 3, loss 4.9267e-05
w: [[ 2.00156975 -3.39996433]] b: 4.19986
```



7.5.1 结论

- 使用 Gluon 的 Trainer 可以轻松使用动量法。

7.5.2 练习

- 如果想用以上代码重现随机梯度下降，应该把动量参数改为多少？

吐槽和讨论欢迎点[这里](#)

7.6 Adagrad —从 0 开始

在我们之前的优化算法中，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，模型参数中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，当一个模型的损失函数为 L ，参数为一个多维向量 $[x_1, x_2]^\top$ 时，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：

$$\begin{aligned}x_1 &:= x_1 - \eta \frac{\partial L}{\partial x_1} \\x_2 &:= x_2 - \eta \frac{\partial L}{\partial x_2}\end{aligned}$$

其中元素 x_1 和 x_2 都使用相同的学习率 η 来自我迭代。如果让 x_1 和 x_2 使用不同的学习率自我迭代呢？

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。

7.6.1 Adagrad 算法

由于小批量随机梯度下降包含了梯度下降和随机梯度下降这两种特殊形式，我们在之后的优化章节里提到的梯度都指的是小批量随机梯度。由于我们会经常用到按元素操作，这里稍作介绍。假设 $\mathbf{x} = [4, 9]^\top$ ，以下是一些按元素操作的例子：

- 按元素相加： $\mathbf{x} + 1 = [5, 10]^\top$
- 按元素相乘： $\mathbf{x} \odot \mathbf{x} = [16, 81]^\top$
- 按元素相除： $72 / \mathbf{x} = [18, 8]^\top$
- 按元素开方： $\sqrt{\mathbf{x}} = [2, 3]^\top$

Adagrad 的算法会使用一个梯度按元素平方的累加变量 \mathbf{s} ，并将其中每个元素初始化为 0。在每次迭代中，首先计算小批量梯度 \mathbf{g} ，然后将该梯度按元素平方后累加到变量 \mathbf{s} ：

$$\mathbf{s} := \mathbf{s} + \mathbf{g} \odot \mathbf{g}$$

然后我们将模型参数中每个元素的学习率通过按元素操作重新调整一下：

$$\mathbf{g}' := \frac{\eta}{\sqrt{\mathbf{s} + \epsilon}} \odot \mathbf{g}$$

其中 η 是初始学习率， ϵ 是为了维持数值稳定性而添加的常数，例如 10^{-7} 。请注意其中按元素开方、除法和乘法的操作。这些按元素操作使得模型参数中每个元素都分别拥有自己的学习率。

需要强调的是，由于梯度按元素平方的累加变量 \mathbf{s} 出现在分母，Adagrad 的核心思想是：如果模型损失函数有关一个参数元素的偏导数一直都较大，那么就让它的学习率下降快一点；反之，如果模型损失函数有关一个参数元素的偏导数一直都较小，那么就让它的学习率下降慢一点。然而，由于 \mathbf{s} 一直在累加按元素平方的梯度，每个元素的学习率在迭代过程中一直在降低或不变。所以在有些问题下，当学习率在迭代早期降得较快时且当前解依然不理想时，Adagrad 在迭代后期可能较难找到一个有用的解。

最后的参数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：

$$\mathbf{x} := \mathbf{x} - \mathbf{g}'$$

7.6.2 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]: # Adagrad 算法
def adagrad(params, sqrs, lr, batch_size):
    eps_stable = 1e-7
    for param, sqr in zip(params, sqrs):
        g = param.grad / batch_size
        sqr[:] += nd.square(g)
        div = lr * g / nd.sqrt(sqr + eps_stable)
        param[:] -= div
```

7.6.3 实验

实验中，我们以线性回归为例。其中真实参数 w 为 $[2, -3.4]$ ， b 为 4.2。我们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。

```
In [2]: from mxnet import ndarray as nd
import mxnet as mx
from mxnet import autograd
from mxnet import gluon
import random

mx.random.seed(1)
random.seed(1)

# 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
X = nd.random_normal(scale=1, shape=(num_examples, num_inputs))
y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
y += .01 * nd.random_normal(scale=1, shape=y.shape)
dataset = gluon.data.ArrayDataset(X, y)

# 构造迭代器。
import random
def data_iter(batch_size):
    idx = list(range(num_examples))
    random.shuffle(idx)
```

```

for batch_i, i in enumerate(range(0, num_examples, batch_size)):
    j = nd.array(idx[i: min(i + batch_size, num_examples)])
    yield batch_i, X.take(j), y.take(j)

# 初始化模型参数。
def init_params():
    w = nd.random_normal(scale=1, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    sqrs = []
    for param in params:
        param.attach_grad()
        # 把梯度按元素平方的累加变量初始化为和参数形状相同的零张量。
        sqrs.append(param.zeros_like())
    return params, sqrs

# 线性回归模型。
def net(X, w, b):
    return nd.dot(X, w) + b

# 损失函数。
def square_loss(yhat, y):
    return (yhat - y.reshape(yhat.shape)) ** 2 / 2

```

接下来定义训练函数。训练函数的 period 参数说明，每次采样过该数目的数据点后，记录当前目标函数值用于作图。例如，当 period 和 batch_size 都为 10 时，每次迭代后均会记录目标函数值。

另外，与随机梯度下降算法不同，这里的初始学习率 lr 没有自我衰减。

```

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

def train(batch_size, lr, epochs, period):
    assert period >= batch_size and period % batch_size == 0
    [w, b], sqrs = init_params()
    total_loss = [np.mean(square_loss(net(X, w, b), y).asnumpy())]

    # 注意 epoch 从 1 开始计数。
    for epoch in range(1, epochs + 1):

```

```
for batch_i, data, label in data_iter(batch_size):
    with autograd.record():
        output = net(data, w, b)
        loss = square_loss(output, label)
    loss.backward()
    adagrad([w, b], sqrs, lr, batch_size)
    if batch_i * batch_size % period == 0:
        total_loss.append(np.mean(square_loss(net(X, w, b), y).asnumpy()))
print("Batch size %d, Learning rate %f, Epoch %d, loss %.4e" %
      (batch_size, lr, epoch, total_loss[-1]))
print('w:', np.reshape(w.asnumpy(), (1, -1)),
      'b:', b.asnumpy()[0], '\n')
x_axis = np.linspace(0, epochs, len(total_loss), endpoint=True)
plt.semilogy(x_axis, total_loss)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

使用 Adagrad, 最终学到的参数值与真实值较接近。

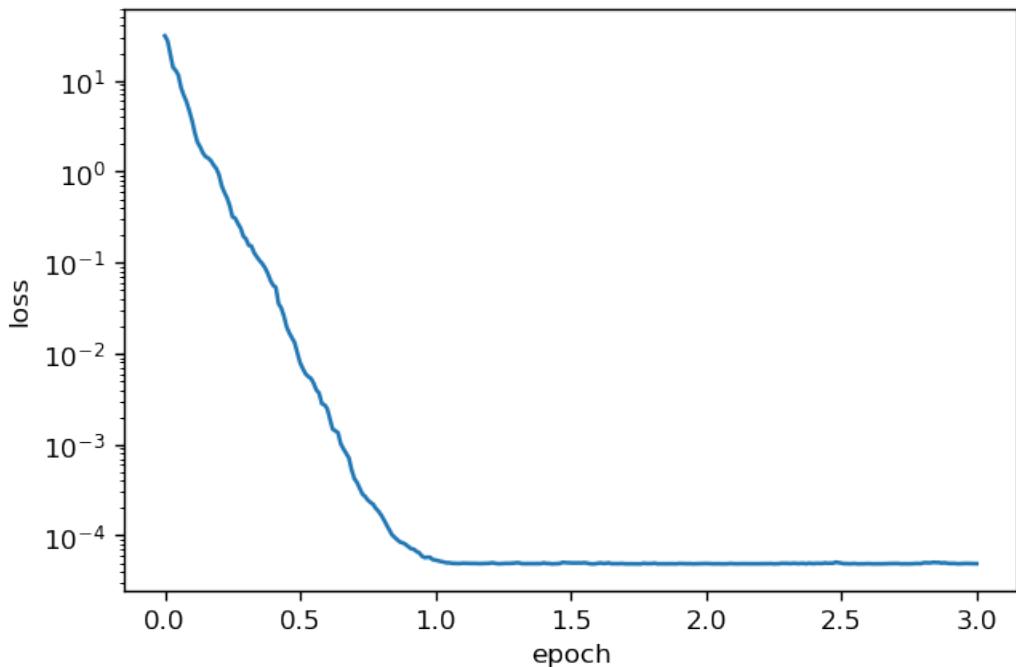
In [4]: train(batch_size=10, lr=0.9, epochs=3, period=10)

Batch size 10, Learning rate 0.900000, Epoch 1, loss 5.3753e-05

Batch size 10, Learning rate 0.900000, Epoch 2, loss 4.9244e-05

Batch size 10, Learning rate 0.900000, Epoch 3, loss 4.9044e-05

w: [[2.00095844 -3.3999505]] b: 4.19947



7.6.4 结论

- Adagrad 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。

7.6.5 练习

- 我们提到了 Adagrad 可能的问题在于按元素平方的梯度累加变量。你能想到什么办法来应对这个问题吗？

吐槽和讨论欢迎点[这里](#)

7.7 Adagrad – 使用 Gluon

在 Gluon 里，使用 Adagrad 很容易。我们无需重新实现它。

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

```

mx.random.seed(1)
random.seed(1)

# 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
X = nd.random_normal(scale=1, shape=(num_examples, num_inputs))
y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
y += .01 * nd.random_normal(scale=1, shape=y.shape)
dataset = gluon.data.ArrayDataset(X, y)

net = gluon.nn.Sequential()
net.add(gluon.nn.Dense(1))
square_loss = gluon.loss.L2Loss()

```

我们需要在 `gluon.Trainer` 中指定优化算法名称 `adagrad` 并设置参数。例如设置初始学习率 `learning_rate`。

```

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

def train(batch_size, lr, epochs, period):
    assert period >= batch_size and period % batch_size == 0
    net.collect_params().initialize(mx.init.Normal(sigma=1), force_reinit=True)
    # Adagrad.
    trainer = gluon.Trainer(net.collect_params(), 'adagrad',
                           {'learning_rate': lr})
    data_iter = gluon.data.DataLoader(dataset, batch_size, shuffle=True)
    total_loss = [np.mean(square_loss(net(X), y).asnumpy())]

    for epoch in range(1, epochs + 1):
        for batch_i, (data, label) in enumerate(data_iter):
            with autograd.record():
                output = net(data)
                loss = square_loss(output, label)
            loss.backward()
            trainer.step(batch_size)

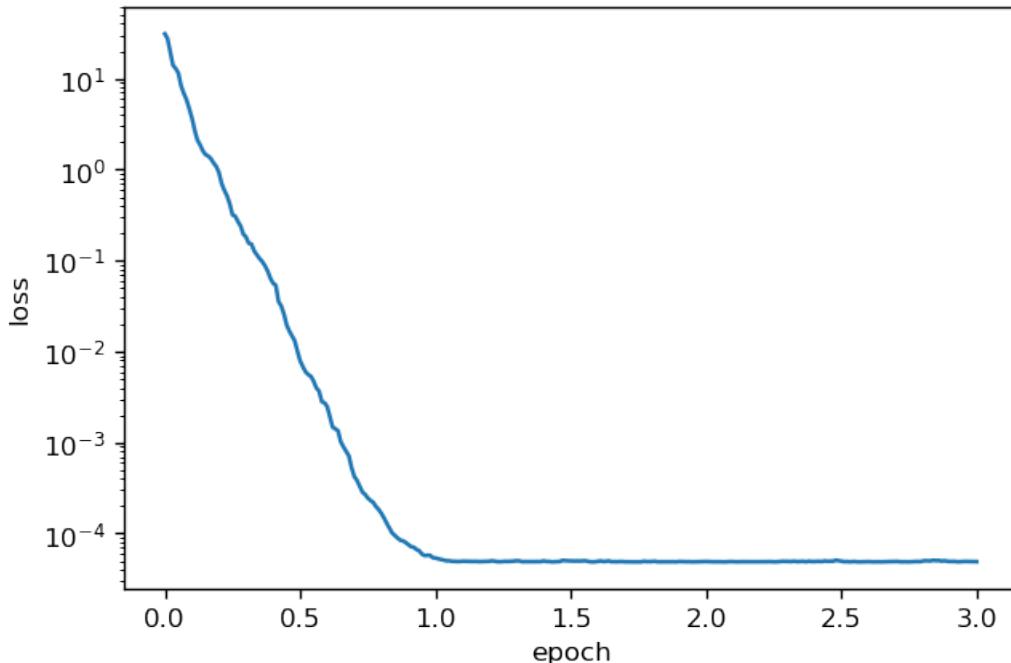
```

```
if batch_i * batch_size % period == 0:  
    total_loss.append(np.mean(square_loss(net(X), y).asnumpy()))  
print("Batch size %d, Learning rate %f, Epoch %d, loss %.4e" %  
    (batch_size, trainer.learning_rate, epoch, total_loss[-1]))  
  
print('w:', np.reshape(net[0].weight.data().asnumpy(), (1, -1)),  
      'b:', net[0].bias.data().asnumpy()[0], '\n')  
x_axis = np.linspace(0, epochs, len(total_loss), endpoint=True)  
plt.semilogy(x_axis, total_loss)  
plt.xlabel('epoch')  
plt.ylabel('loss')  
plt.show()
```

使用 Adagrad, 最终学到的参数值与真实值较接近。

In [3]: train(batch_size=10, lr=0.9, epochs=3, period=10)

```
Batch size 10, Learning rate 0.900000, Epoch 1, loss 5.3753e-05  
Batch size 10, Learning rate 0.900000, Epoch 2, loss 4.9244e-05  
Batch size 10, Learning rate 0.900000, Epoch 3, loss 4.9044e-05  
w: [[ 2.00095844 -3.3999505 ]] b: 4.19947
```



7.7.1 结论

- 使用 Gluon 的 Trainer 可以轻松使用 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？

吐槽和讨论欢迎点[这里](#)

7.8 RMSProp —从 0 开始

我们在*Adagrad*里提到，由于学习率分母上的变量 \mathbf{s} 一直在累加按元素平方的梯度，每个元素的学习率在迭代过程中一直在降低或不变。所以在有些问题下，当学习率在迭代早期降得较快时且当前解依然不理想时，Adagrad 在迭代后期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。我们先给出 RMSProp 算法。

7.8.1 RMSProp 算法

RMSProp 算法会使用一个梯度按元素平方的指数加权移动平均变量 \mathbf{s} ，并将其中每个元素初始化为 0。在每次迭代中，首先计算小批量梯度 \mathbf{g} ，然后对该梯度按元素平方后做指数加权移动平均并计算 \mathbf{s} :

$$\mathbf{s} := \gamma \mathbf{s} + (1 - \gamma) \mathbf{g} \odot \mathbf{g}$$

然后我们将模型参数中每个元素的学习率通过按元素操作重新调整一下：

$$\mathbf{g}' := \frac{\eta}{\sqrt{\mathbf{s} + \epsilon}} \odot \mathbf{g}$$

其中 η 是初始学习率， 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，最后的参数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：

$$\mathbf{x} := \mathbf{x} - \mathbf{g}'$$

需要强调的是，RMSProp 只在 Adagrad 的基础上修改了变量 \mathbf{s} 的更新方法：把累加改成了指数加权移动平均。因此，每个元素的学习率在迭代过程中既可能降低又可能升高。

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]: # RMSProp
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) * nd.square(g)
        div = lr * g / nd.sqrt(sqr + eps_stable)
        param[:] -= div
```

7.8.3 实验

实验中，我们以线性回归为例。其中真实参数 w 为 $[2, -3.4]$ ， b 为 4.2。我们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。

```
In [2]: from mxnet import ndarray as nd
import mxnet as mx
from mxnet import autograd
from mxnet import gluon
import random

mx.random.seed(1)
random.seed(1)

# 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
X = nd.random_normal(scale=1, shape=(num_examples, num_inputs))
y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
y += .01 * nd.random_normal(scale=1, shape=y.shape)
dataset = gluon.data.ArrayDataset(X, y)

# 构造迭代器。
import random
def data_iter(batch_size):
    idx = list(range(num_examples))
    random.shuffle(idx)
```

```

for batch_i, i in enumerate(range(0, num_examples, batch_size)):
    j = nd.array(idx[i: min(i + batch_size, num_examples)])
    yield batch_i, X.take(j), y.take(j)

# 初始化模型参数。
def init_params():
    w = nd.random_normal(scale=1, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    sqrs = []
    for param in params:
        param.attach_grad()
        # 把梯度按元素平方的指数加权移动平均变量初始化为和参数形状相同的零张量。
        sqrs.append(param.zeros_like())
    return params, sqrs

# 线性回归模型。
def net(X, w, b):
    return nd.dot(X, w) + b

# 损失函数。
def square_loss(yhat, y):
    return (yhat - y.reshape(yhat.shape)) ** 2 / 2

```

接下来定义训练函数。训练函数的 period 参数说明，每次采样过该数目的数据点后，记录当前目标函数值用于作图。例如，当 period 和 batch_size 都为 10 时，每次迭代后均会记录目标函数值。

```

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

def train(batch_size, lr, gamma, epochs, period):
    assert period >= batch_size and period % batch_size == 0
    [w, b], sqrs = init_params()
    total_loss = [np.mean(square_loss(net(X, w, b), y).asnumpy())]

    # 注意 epoch 从 1 开始计数。
    for epoch in range(1, epochs + 1):
        for batch_i, data, label in data_iter(batch_size):

```

```

with autograd.record():
    output = net(data, w, b)
    loss = square_loss(output, label)
    loss.backward()
    rmsprop([w, b], sqrs, lr, gamma, batch_size)
if batch_i * batch_size % period == 0:
    total_loss.append(np.mean(square_loss(net(X, w, b), y).asnumpy()))
print("Batch size %d, Learning rate %f, Epoch %d, loss %.4e" %
      (batch_size, lr, epoch, total_loss[-1]))
print('w:', np.reshape(w.asnumpy(), (1, -1)),
      'b:', b.asnumpy()[0], '\n')
x_axis = np.linspace(0, epochs, len(total_loss), endpoint=True)
plt.semilogy(x_axis, total_loss)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()

```

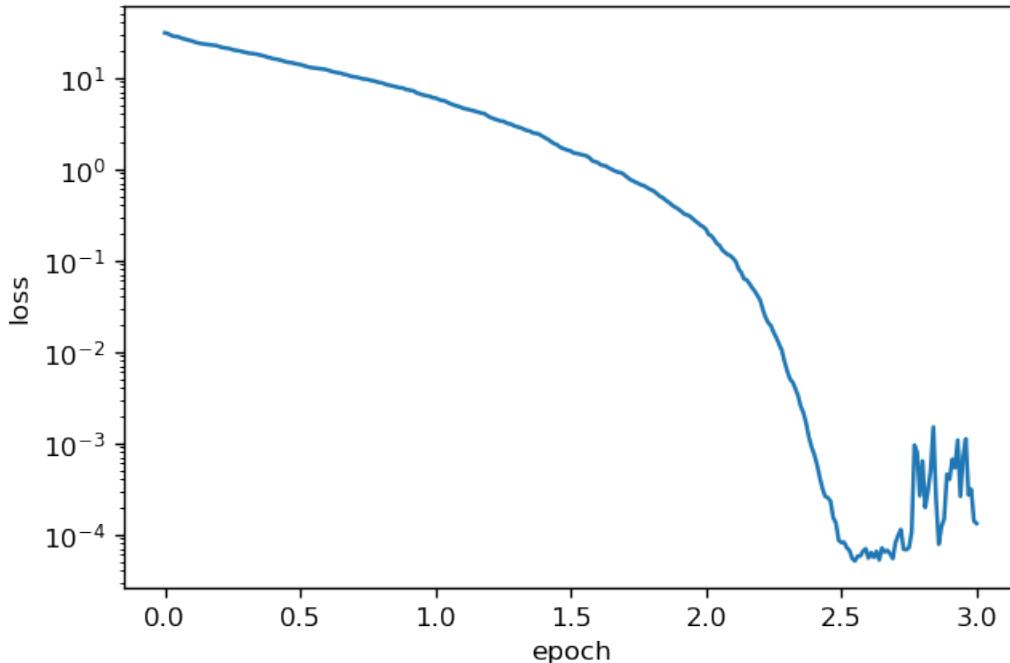
我们将初始学习率设为 0.03，并将 gamma 设为 0.9。损失函数在迭代后期较震荡。

In [4]: train(batch_size=10, lr=0.03, gamma=0.9, epochs=3, period=10)

```

Batch size 10, Learning rate 0.030000, Epoch 1, loss 6.0151e+00
Batch size 10, Learning rate 0.030000, Epoch 2, loss 2.1849e-01
Batch size 10, Learning rate 0.030000, Epoch 3, loss 1.3210e-04
w: [[ 1.99253047 -3.40613008]] b: 4.19266

```



我们将 gamma 调大一点，例如 0.999。这时损失函数在迭代后期较平滑。

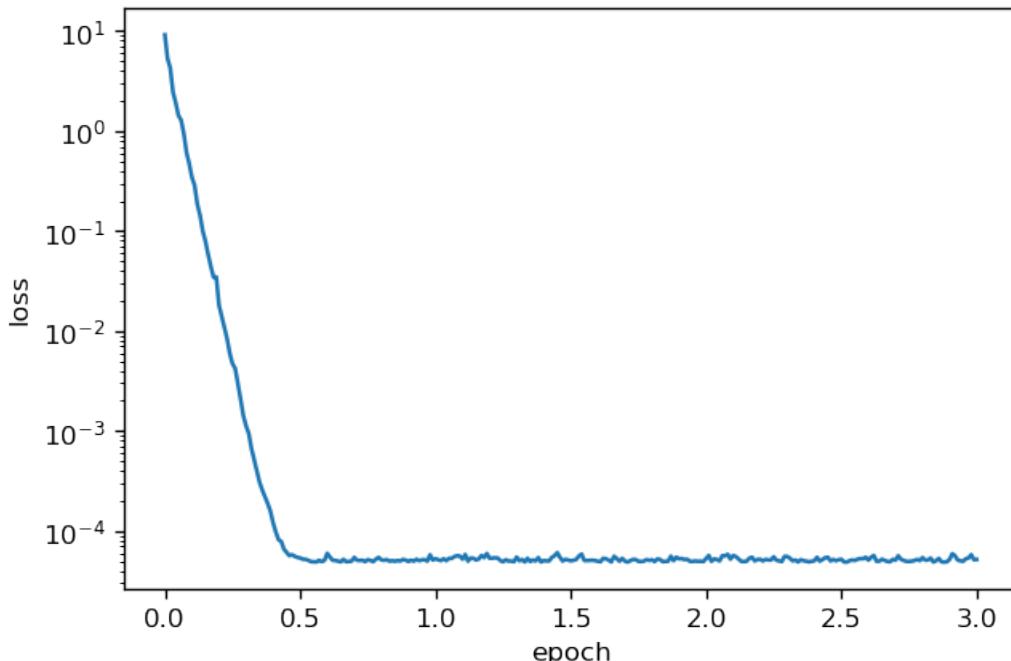
```
In [5]: train(batch_size=10, lr=0.03, gamma=0.999, epochs=3, period=10)
```

```
Batch size 10, Learning rate 0.030000, Epoch 1, loss 5.2718e-05
```

```
Batch size 10, Learning rate 0.030000, Epoch 2, loss 5.2795e-05
```

```
Batch size 10, Learning rate 0.030000, Epoch 3, loss 5.1766e-05
```

```
w: [[ 1.99898291 -3.40138555]] b: 4.1992
```



7.8.4 结论

- RMSProp 和 Adagrad 的不同在于，RMSProp 使用了梯度按元素平方的指数加权移动平均变量来调整学习率。
- 通过调整指数加权移动平均中 gamma 参数的值可以控制学习率的变化。

7.8.5 练习

- 通过查阅网上资料，你对指数加权移动平均是怎样理解的？
- 为什么 gamma 调大后，损失函数在迭代后期较平滑？

吐槽和讨论欢迎点[这里](#)

7.9 RMSProp – 使用 Gluon

在 Gluon 里, 使用 RMSProp 很容易。我们无需重新实现它。

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

        mx.random.seed(1)
        random.seed(1)

        # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        X = nd.random_normal(scale=1, shape=(num_examples, num_inputs))
        y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
        y += .01 * nd.random_normal(scale=1, shape=y.shape)
        dataset = gluon.data.ArrayDataset(X, y)

        net = gluon.nn.Sequential()
        net.add(gluon.nn.Dense(1))
        square_loss = gluon.loss.L2Loss()
```

我们需要在 `gluon.Trainer` 中指定优化算法名称 `rmsprop` 并设置参数。例如设置初始学习率 `learning_rate` 和指数加权移动平均中 `gamma1` 参数。

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

        def train(batch_size, lr, gamma, epochs, period):
            assert period >= batch_size and period % batch_size == 0
            net.collect_params().initialize(mx.init.Normal(sigma=1), force_reinit=True)
            # RMSProp。
            trainer = gluon.Trainer(net.collect_params(), 'rmsprop',
                                    {'learning_rate': lr, 'gamma1': gamma})
```

```

data_iter = gluon.data.DataLoader(dataset, batch_size, shuffle=True)
total_loss = [np.mean(square_loss(net(X), y).asnumpy())]

for epoch in range(1, epochs + 1):
    for batch_i, (data, label) in enumerate(data_iter):
        with autograd.record():
            output = net(data)
            loss = square_loss(output, label)
            loss.backward()
            trainer.step(batch_size)
        if batch_i * batch_size % period == 0:
            total_loss.append(np.mean(square_loss(net(X), y).asnumpy()))
    print("Batch size %d, Learning rate %f, Epoch %d, loss %.4e" %
          (batch_size, trainer.learning_rate, epoch, total_loss[-1]))

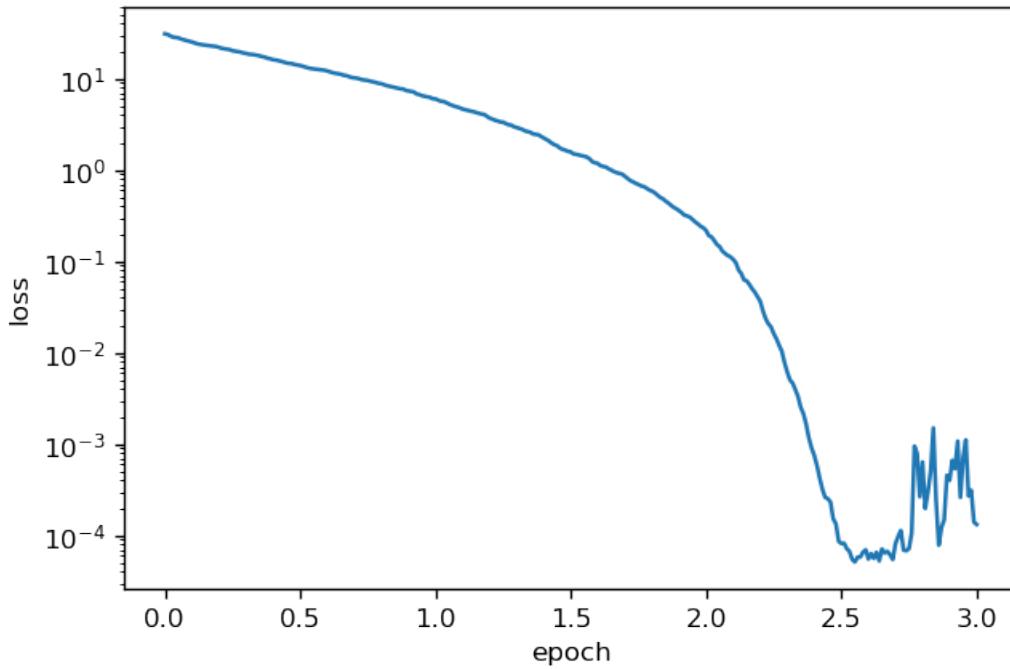
print('w:', np.reshape(net[0].weight.data().asnumpy(), (1, -1)),
      'b:', net[0].bias.data().asnumpy()[0], '\n')
x_axis = np.linspace(0, epochs, len(total_loss), endpoint=True)
plt.semilogy(x_axis, total_loss)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()

```

我们将初始学习率设为 0.03，并将 gamma 设为 0.9。损失函数在迭代后期较震荡。

```
In [3]: train(batch_size=10, lr=0.03, gamma=0.9, epochs=3, period=10)

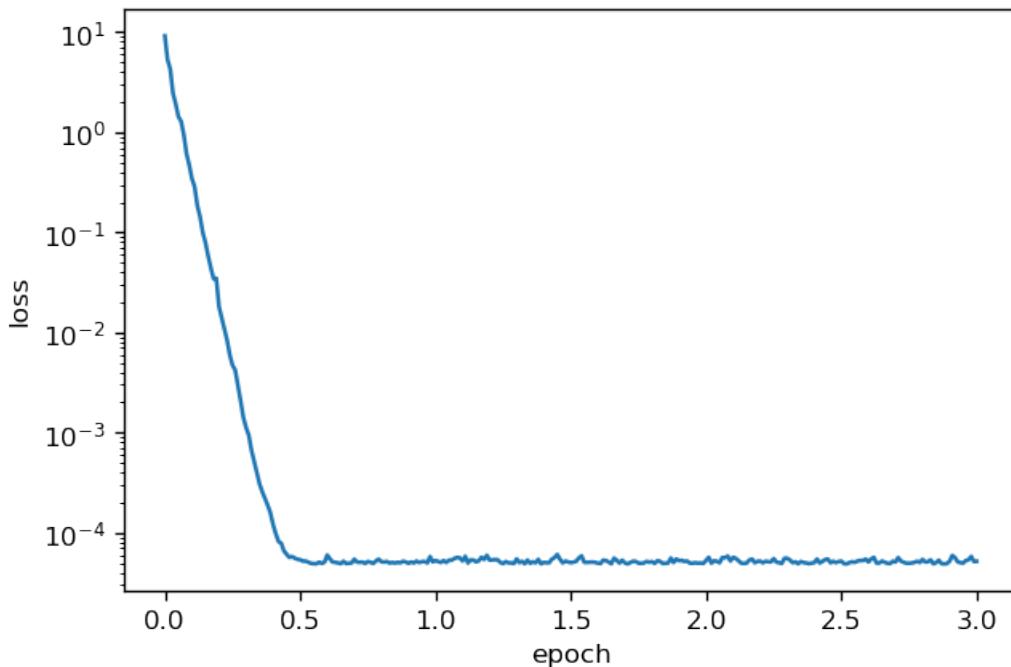
Batch size 10, Learning rate 0.030000, Epoch 1, loss 6.0151e+00
Batch size 10, Learning rate 0.030000, Epoch 2, loss 2.1849e-01
Batch size 10, Learning rate 0.030000, Epoch 3, loss 1.3210e-04
w: [[ 1.99253058 -3.40612984]] b: 4.19266
```



我们将 gamma 调大一点，例如 0.999。这时损失函数在迭代后期较平滑。

```
In [4]: train(batch_size=10, lr=0.03, gamma=0.999, epochs=3, period=10)
```

```
Batch size 10, Learning rate 0.030000, Epoch 1, loss 5.2718e-05
Batch size 10, Learning rate 0.030000, Epoch 2, loss 5.2795e-05
Batch size 10, Learning rate 0.030000, Epoch 3, loss 5.1766e-05
w: [[ 1.99898291 -3.40138555]] b: 4.1992
```



7.9.1 结论

- 使用 Gluon 的 Trainer 可以轻松使用 RMSProp。

7.9.2 练习

- 试着使用其他的初始学习率和 gamma 参数的组合，观察实验结果。

吐槽和讨论欢迎点[这里](#)

7.10 Adadelta —从 0 开始

我们在[Adagrad](#)里提到，由于学习率分母上的变量 s 一直在累加按元素平方的梯度，每个元素的学习率在迭代过程中一直在降低或不变。所以在有些问题下，当学习率在迭代早期降得较快时且当前解依然不理想时，Adagrad 在迭代后期可能较难找到一个有用的解。我们在[RMSProp](#)介绍了应对这一问题的一种方法：对梯度按元素平方使用指数加权移动平均而不是累加。

事实上，Adadelta 也是一种应对这个问题的方法。有意思的是，它没有学习率参数。

7.10.1 Adadelta 算法

Adadelta 算法也像 RMSProp 一样，使用了一个梯度按元素平方的指数加权移动平均变量 \mathbf{s} ，并将其中每个元素初始化为 0。在每次迭代中，首先计算小批量梯度 \mathbf{g} ，然后对该梯度按元素平方后做指数加权移动平均并计算 \mathbf{s} ：

$$\mathbf{s} := \rho \mathbf{s} + (1 - \rho) \mathbf{g} \odot \mathbf{g}$$

然后我们计算当前需要更新的参数的变化量：

$$\mathbf{g}' = \frac{\sqrt{\Delta \mathbf{x} + \epsilon}}{\sqrt{\mathbf{s} + \epsilon}} \odot \mathbf{g}$$

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 一样，模型参数中每个元素都分别拥有自己的学习率。其中 $\Delta \mathbf{x}$ 初始化为零张量，并做如下 \mathbf{g}' 按元素平方的指数加权移动平均：

$$\Delta \mathbf{x} := \rho \Delta \mathbf{x} + (1 - \rho) \mathbf{g}' \odot \mathbf{g}'$$

同样地，最后的参数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：

$$\mathbf{x} := \mathbf{x} - \mathbf{g}'$$

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]: # Adadelta
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) * nd.square(g)
        cur_delta = nd.sqrt(delta + eps_stable) / nd.sqrt(sqr + eps_stable) * g
        delta[:] = rho * delta + (1. - rho) * cur_delta * cur_delta
        param[:] -= cur_delta
```

7.10.3 实验

实验中，我们以线性回归为例。其中真实参数 w 为 $[2, -3.4]$ ， b 为 4.2。我们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。

```
In [2]: from mxnet import ndarray as nd
import mxnet as mx
from mxnet import autograd
from mxnet import gluon
import random

mx.random.seed(1)
random.seed(1)

# 生成数据集。
num_inputs = 2
num_examples = 1000
true_w = [2, -3.4]
true_b = 4.2
X = nd.random_normal(scale=1, shape=(num_examples, num_inputs))
y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
y += .01 * nd.random_normal(scale=1, shape=y.shape)
dataset = gluon.data.ArrayDataset(X, y)

# 构造迭代器。
import random
def data_iter(batch_size):
    idx = list(range(num_examples))
    random.shuffle(idx)
    for batch_i, i in enumerate(range(0, num_examples, batch_size)):
        j = nd.array(idx[i: min(i + batch_size, num_examples)])
        yield batch_i, X.take(j), y.take(j)

# 初始化模型参数。
def init_params():
    w = nd.random_normal(scale=1, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    sqrs = []
    deltas = []
    for param in params:
        param.attach_grad()
        # 把算法中基于指数加权移动平均的变量初始化为和参数形状相同的零张量。
        sqrs.append(param.zeros_like())
        deltas.append(param.zeros_like())
    return params, sqrs, deltas
```

```
# 线性回归模型。
def net(X, w, b):
    return nd.dot(X, w) + b

# 损失函数。
def square_loss(yhat, y):
    return (yhat - y.reshape(yhat.shape)) ** 2 / 2
```

接下来定义训练函数。当 epoch 大于 2 时 (epoch 从 1 开始计数), 学习率以自乘 0.1 的方式自我衰减。训练函数的 period 参数说明, 每次采样过该数目的数据点后, 记录当前目标函数值用于作图。例如, 当 period 和 batch_size 都为 10 时, 每次迭代后均会记录目标函数值。

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

def train(batch_size, rho, epochs, period):
    assert period >= batch_size and period % batch_size == 0
    [w, b], sqrs, deltas = init_params()
    total_loss = [np.mean(square_loss(net(X, w, b), y).asnumpy())]

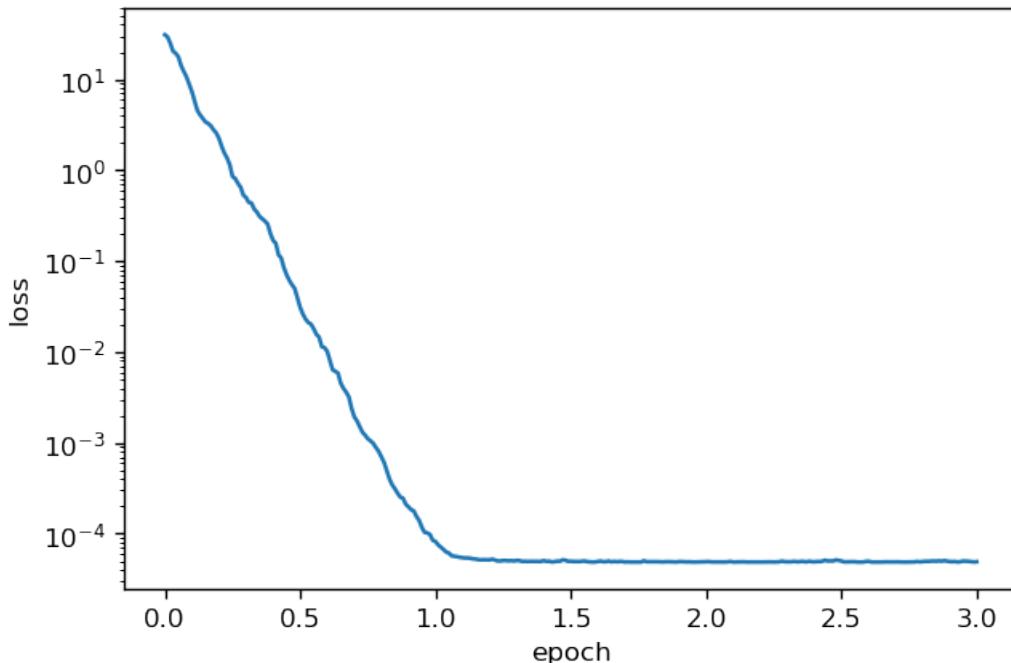
    # 注意 epoch 从 1 开始计数。
    for epoch in range(1, epochs + 1):
        for batch_i, data, label in data_iter(batch_size):
            with autograd.record():
                output = net(data, w, b)
                loss = square_loss(output, label)
                loss.backward()
                adadelta([w, b], sqrs, deltas, rho, batch_size)
            if batch_i * batch_size % period == 0:
                total_loss.append(np.mean(square_loss(net(X, w, b), y).asnumpy()))
        print("Batch size %d, Epoch %d, loss %.4e" %
              (batch_size, epoch, total_loss[-1]))
    print('w:', np.reshape(w.asnumpy(), (1, -1)),
          'b:', b.asnumpy()[0], '\n')
    x_axis = np.linspace(0, epochs, len(total_loss), endpoint=True)
    plt.semilogy(x_axis, total_loss)
    plt.xlabel('epoch')
    plt.ylabel('loss')
```

```
plt.show()
```

使用 Adadelta，最终学到的参数值与真实值较接近。

```
In [4]: train(batch_size=10, rho=0.9999, epochs=3, period=10)
```

```
Batch size 10, Epoch 1, loss 8.2180e-05
Batch size 10, Epoch 2, loss 4.9327e-05
Batch size 10, Epoch 3, loss 4.9216e-05
w: [[ 2.00076938 -3.39997005]] b: 4.19921
```



7.10.4 结论

- 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参数？它被什么代替了？

吐槽和讨论欢迎点[这里](#)

7.11 Adadelta — 使用 Gluon

在 Gluon 里，使用 Adadelta 很容易。我们无需重新实现它。

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

        mx.random.seed(1)
        random.seed(1)

        # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        X = nd.random_normal(scale=1, shape=(num_examples, num_inputs))
        y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
        y += .01 * nd.random_normal(scale=1, shape=y.shape)
        dataset = gluon.data.ArrayDataset(X, y)

        net = gluon.nn.Sequential()
        net.add(gluon.nn.Dense(1))
        square_loss = gluon.loss.L2Loss()
```

我们需要在 `gluon.Trainer` 中指定优化算法名称 `adadelta` 并设置 `rho` 参数。

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

        def train(batch_size, rho, epochs, period):
            assert period >= batch_size and period % batch_size == 0
            net.collect_params().initialize(mx.init.Normal(sigma=1), force_reinit=True)
            # Adadelta。
            trainer = gluon.Trainer(net.collect_params(), 'adadelta',
                                    {'rho': rho})
            data_iter = gluon.data.DataLoader(dataset, batch_size, shuffle=True)
```

```
total_loss = [np.mean(square_loss(net(X), y).asnumpy())]

for epoch in range(1, epochs + 1):
    for batch_i, (data, label) in enumerate(data_iter):
        with autograd.record():
            output = net(data)
            loss = square_loss(output, label)
            loss.backward()
            trainer.step(batch_size)

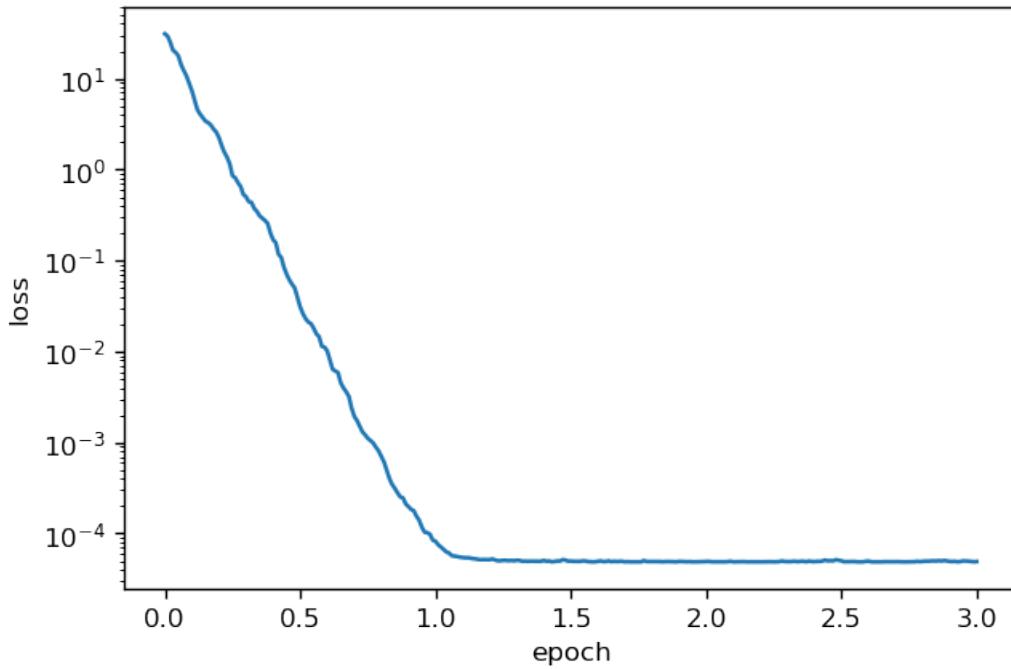
        if batch_i * batch_size % period == 0:
            total_loss.append(np.mean(square_loss(net(X), y).asnumpy()))
    print("Batch size %d, Epoch %d, loss %.4e" %
          (batch_size, epoch, total_loss[-1]))

print('w:', np.reshape(net[0].weight.data().asnumpy(), (1, -1)),
      'b:', net[0].bias.data().asnumpy()[0], '\n')
x_axis = np.linspace(0, epochs, len(total_loss), endpoint=True)
plt.semilogy(x_axis, total_loss)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

使用 Adadelta，最终学到的参数值与真实值较接近。

```
In [3]: train(batch_size=10, rho=0.9999, epochs=3, period=10)

Batch size 10, Epoch 1, loss 8.2180e-05
Batch size 10, Epoch 2, loss 4.9327e-05
Batch size 10, Epoch 3, loss 4.9216e-05
w: [[ 2.00076938 -3.39997005]] b: 4.19921
```



7.11.1 结论

- 使用 Gluon 的 Trainer 可以轻松使用 Adadelta。

7.11.2 练习

- 如果把试验中的参数 rho 改小会怎样，例如 0.9？观察实验结果。

吐槽和讨论欢迎点[这里](#)

7.12 Adam —从 0 开始

Adam 是一个组合了动量法和 RMSProp 的优化算法。

7.12.1 Adam 算法

Adam 算法会使用一个动量变量 \mathbf{v} 和一个 RMSProp 中梯度按元素平方的指数加权移动平均变量 \mathbf{s} ，并将它们中每个元素初始化为 0。在每次迭代中，首先计算小批量梯度 \mathbf{g} ，并递增迭代次数

$$t := t + 1$$

然后对梯度做指数加权移动平均并计算动量变量 \mathbf{v} :

$$\mathbf{v} := \beta_1 \mathbf{v} + (1 - \beta_1) \mathbf{g}$$

该梯度按元素平方后做指数加权移动平均并计算 \mathbf{s} :

$$\mathbf{s} := \beta_2 \mathbf{s} + (1 - \beta_2) \mathbf{g} \odot \mathbf{g}$$

在 Adam 算法里, 为了减轻 \mathbf{v} 和 \mathbf{s} 被初始化为 0 在迭代初期对计算指数加权移动平均的影响, 我们做下面的偏差修正:

$$\hat{\mathbf{v}} := \frac{\mathbf{v}}{1 - \beta_1^t}$$

和

$$\hat{\mathbf{s}} := \frac{\mathbf{s}}{1 - \beta_2^t}$$

可以看到, 当 $0 \leq \beta_1, \beta_2 < 1$ 时 (算法作者建议分别设为 0.9 和 0.999), 当迭代后期 t 较大时, 偏差修正几乎就不再有影响。我们使用以上偏差修正后的动量变量和 RMSProp 中梯度按元素平方的指数加权移动平均变量, 将模型参数中每个元素的学习率通过按元素操作重新调整一下:

$$\mathbf{g}' := \frac{\eta \hat{\mathbf{v}}}{\sqrt{\hat{\mathbf{s}}} + \epsilon}$$

其中 η 是初始学习率, 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, 最后的参数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:

$$\mathbf{x} := \mathbf{x} - \mathbf{g}'$$

7.12.2 Adam 的实现

Adam 的实现很简单。我们只需要把上面的数学公式翻译成代码。

```
In [1]: # Adam.
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) * nd.square(g)
v_bias_corr = v / (1. - beta1 ** t)
sqr_bias_corr = sqr / (1. - beta2 ** t)
div = lr * v_bias_corr / (nd.sqrt(sqr_bias_corr) + eps_stable)
param[:] = param - div
```

7.12.3 实验

实验中，我们以线性回归为例。其中真实参数 w 为 $[2, -3.4]$ ， b 为 4.2。我们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。

```
In [2]: import mxnet as mx
        from mxnet import autograd
        from mxnet import ndarray as nd
        from mxnet import gluon
        import random

        mx.random.seed(1)
        random.seed(1)

        # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        X = nd.random_normal(scale=1, shape=(num_examples, num_inputs))
        y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
        y += .01 * nd.random_normal(scale=1, shape=y.shape)
        dataset = gluon.data.ArrayDataset(X, y)

        # 构造迭代器。
        import random
        def data_iter(batch_size):
            idx = list(range(num_examples))
            random.shuffle(idx)
            for batch_i, i in enumerate(range(0, num_examples, batch_size)):
                j = nd.array(idx[i: min(i + batch_size, num_examples)])
                yield batch_i, X.take(j), y.take(j)

        # 初始化模型参数。
```

```

def init_params():
    w = nd.random_normal(scale=1, shape=(num_inputs, 1))
    b = nd.zeros(shape=(1,))
    params = [w, b]
    vs = []
    sqrs = []
    for param in params:
        param.attach_grad()
        # 把算法中基于指数加权移动平均的变量初始化为和参数形状相同的零张量。
        vs.append(param.zeros_like())
        sqrs.append(param.zeros_like())
    return params, vs, sqrs

# 线性回归模型。
def net(X, w, b):
    return nd.dot(X, w) + b

# 损失函数。
def square_loss(yhat, y):
    return (yhat - y.reshape(yhat.shape)) ** 2 / 2

```

接下来定义训练函数。当 epoch 大于 2 时 (epoch 从 1 开始计数), 学习率以自乘 0.1 的方式自我衰减。训练函数的 period 参数说明, 每次采样过该数目的数据点后, 记录当前目标函数值用于作图。例如, 当 period 和 batch_size 都为 10 时, 每次迭代后均会记录目标函数值。

```

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

def train(batch_size, lr, epochs, period):
    assert period >= batch_size and period % batch_size == 0
    [w, b], vs, sqrs = init_params()
    total_loss = [np.mean(square_loss(net(X, w, b), y).asnumpy())]

    # 注意 epoch 从 1 开始计数。
    t = 0
    for epoch in range(1, epochs + 1):
        for batch_i, data, label in data_iter(batch_size):
            with autograd.record():
                output = net(data, w, b)

```

```

        loss = square_loss(output, label)
        loss.backward()
        # 必须在调用 Adam 前。
        t += 1
        adam([w, b], vs, sqrs, lr, batch_size, t)
        if batch_i * batch_size % period == 0:
            total_loss.append(np.mean(square_loss(net(X, w, b), y).asnumpy()))
        print("Batch size %d, Learning rate %f, Epoch %d, loss %.4e" %
              (batch_size, lr, epoch, total_loss[-1]))
        print('w:', np.reshape(w.asnumpy(), (1, -1)),
              'b:', b.asnumpy()[0], '\n')
    x_axis = np.linspace(0, epochs, len(total_loss), endpoint=True)
    plt.semilogy(x_axis, total_loss)
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.show()

```

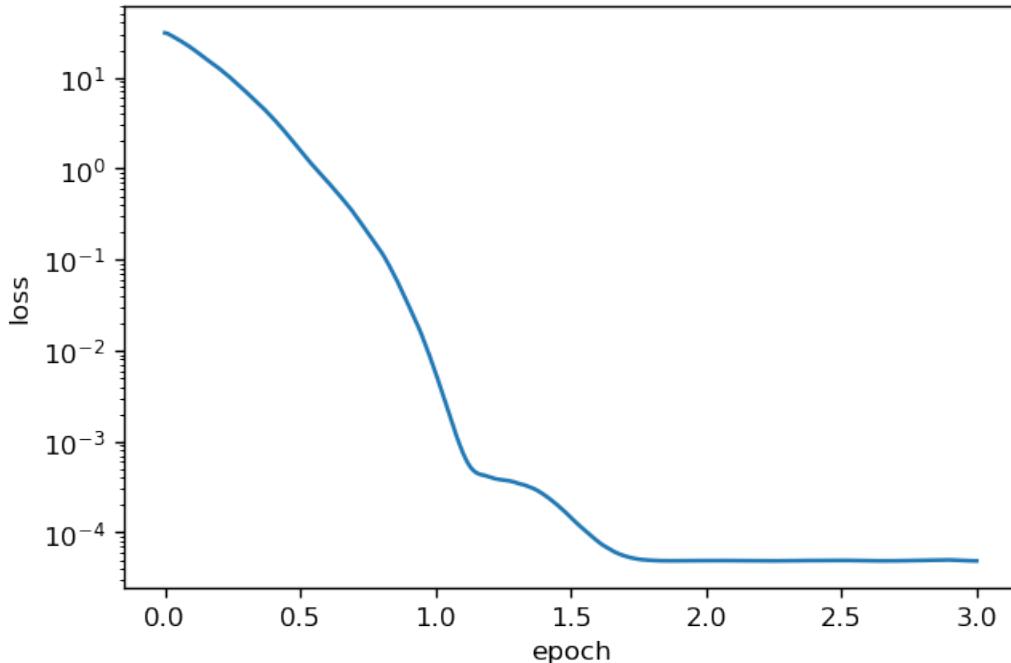
使用 Adam, 最终学到的参数值与真实值较接近。

In [4]: train(batch_size=10, lr=0.1, epochs=3, period=10)

```

Batch size 10, Learning rate 0.100000, Epoch 1, loss 5.7474e-03
Batch size 10, Learning rate 0.100000, Epoch 2, loss 4.9194e-05
Batch size 10, Learning rate 0.100000, Epoch 3, loss 4.8772e-05
w: [[ 2.00043869 -3.39973044]] b: 4.20008

```



7.12.4 结论

- Adam 组合了动量法和 RMSProp。

7.12.5 练习

- 你是怎样理解 Adam 算法中的偏差修正项的?

吐槽和讨论欢迎点[这里](#)

7.13 Adam — 使用 Gluon

在 Gluon 里，使用 Adam 很容易。我们无需重新实现它。

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

        mx.random.seed(1)
        random.seed(1)

        # 生成数据集。
        num_inputs = 2
        num_examples = 1000
        true_w = [2, -3.4]
        true_b = 4.2
        X = nd.random_normal(scale=1, shape=(num_examples, num_inputs))
        y = true_w[0] * X[:, 0] + true_w[1] * X[:, 1] + true_b
        y += .01 * nd.random_normal(scale=1, shape=y.shape)
        dataset = gluon.data.ArrayDataset(X, y)

        net = gluon.nn.Sequential()
        net.add(gluon.nn.Dense(1))
        square_loss = gluon.loss.L2Loss()
```

我们需要在 `gluon.Trainer` 中指定优化算法名称 `adam` 并设置学习率。

```
In [2]: %matplotlib inline
import matplotlib as mpl
```

```
mpl.rcParams['figure.dpi']= 120
import matplotlib.pyplot as plt

def train(batch_size, lr, epochs, period):
    assert period >= batch_size and period % batch_size == 0
    net.collect_params().initialize(mx.init.Normal(sigma=1), force_reinit=True)
    # Adam.
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    data_iter = gluon.data.DataLoader(dataset, batch_size, shuffle=True)
    total_loss = [np.mean(square_loss(net(X), y).asnumpy())]

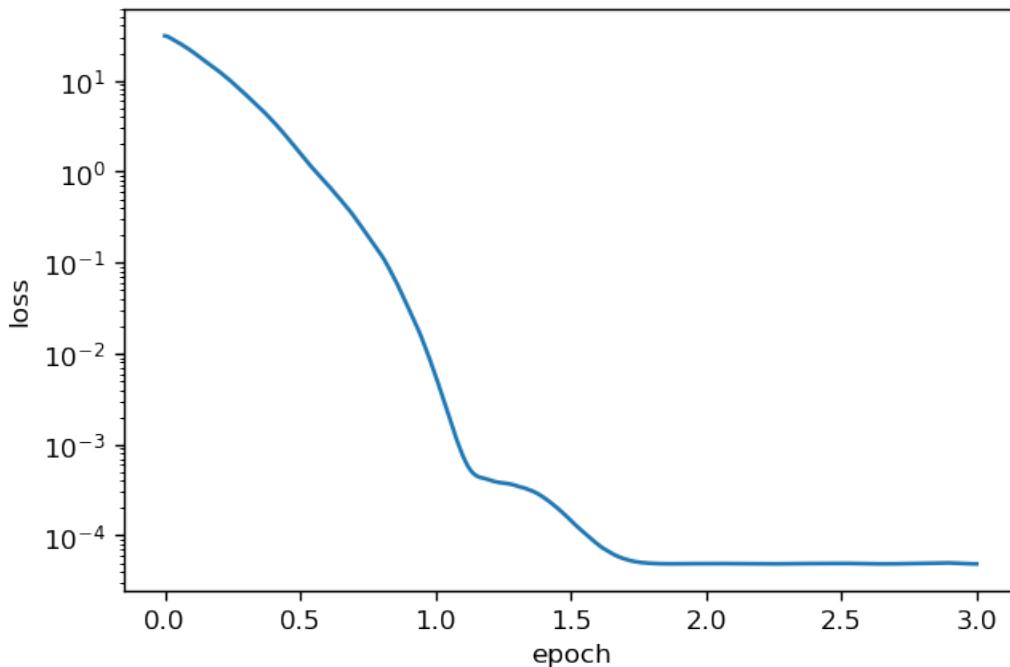
    for epoch in range(1, epochs + 1):
        for batch_i, (data, label) in enumerate(data_iter):
            with autograd.record():
                output = net(data)
                loss = square_loss(output, label)
            loss.backward()
            trainer.step(batch_size)
            if batch_i * batch_size % period == 0:
                total_loss.append(np.mean(square_loss(net(X), y).asnumpy()))
        print("Batch size %d, Learning rate %f, Epoch %d, loss %.4e" %
              (batch_size, trainer.learning_rate, epoch, total_loss[-1]))

    print('w:', np.reshape(net[0].weight.data().asnumpy(), (1, -1)),
          'b:', net[0].bias.data().asnumpy()[0], '\n')
    x_axis = np.linspace(0, epochs, len(total_loss), endpoint=True)
    plt.semilogy(x_axis, total_loss)
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.show()
```

使用 Adam, 最终学到的参数值与真实值较接近。

In [3]: `train(batch_size=10, lr=0.1, epochs=3, period=10)`

```
Batch size 10, Learning rate 0.100000, Epoch 1, loss 5.7464e-03
Batch size 10, Learning rate 0.100000, Epoch 2, loss 4.9194e-05
Batch size 10, Learning rate 0.100000, Epoch 3, loss 4.8772e-05
w: [[ 2.00043869 -3.39973044]] b: 4.20008
```



7.13.1 结论

- 使用 Gluon 的 Trainer 可以轻松使用 Adam。

7.13.2 练习

- 试着使用其他 Adam 初始学习率，观察实验结果。

7.13.3 赠诗一首以总结优化章节

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，优化还需已调参。

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: 这个参数被换成别的了
- 权: 指数加权移动平均

吐槽和讨论欢迎点[这里](#)

GLUON 高级

8.1 Hybridize：更快和更好移植

到目前为止我们看到的教程都使用了命令式的编程。你可能之前都从来没有听说这个词。不过，一直以来我们都是用这种方式写 Python 代码。

考虑下面这段代码：

```
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

正如大家希望的那样，在运行 `E = add(A, B)` 的时候，我们实际上会做加法运算并返回结果。之后的指令 `F =` 和 `G =` 会跟在后面顺序执行。

这个编程方式的主要优点是很自然，大部分用户可能都不会意识还有别的其他的编程方式。但它的不足是可能会慢。这是因为我们不断跟（可能很慢的）Python 的运行环境打交道。即使我们重复调用了 `add` 三次，我们还是会跟 Python 打三次交道。另外一点是，我们需要保存 `E` 和 `F` 的结果直到 `fancy_func` 结束。因为之前我们不知道是不是还会有谁用这些结果。

事实上这里不同的打开方式。其中一个叫**符号式**编程，大部分的深度学习框架包括 Theano 和 TensorFlow 用了这个方式。通常这个方式的程序需要下面三个步骤：

1. 定义计算流程

2. 编译成可执行的程序
3. 给定输入调用编译好的程序

我们重新实现上面的程序：

```
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()
y = compile(prog, '', 'exec')
exec(y)
```

10

可以看到我们定义的三个函数都只是返回计算流程。之后我们编译再执行。在编译的时候系统能够看到整个程序，因此有更多的优化空间。例如编译的时候可以将程序改写成 `print((1+2)+(3+4))`，甚至直接 `print(10)`。这里我们不仅减少了函数调用，同时节省了内存。

总结一下

- **命令式编程更方便。** 当我们在 Python 里用一个命令式编程库时，我们在写 Python 代码，绝大部分代码很符合直觉。同样很容易逮 BUG，因为我们可以拿到所有中间变量值，我们可以简单打印它们，或者使用 Python 的 debug 工具。
- **符号式编程更加高效而且更容易移植。** 之前我们提到在编译的时候系统可以容易的做更多的优化。另外一个好处是可以将程序变成一个与 Python 无关的格式，从而我们可以在非

Python 环境下运行。

8.1.1 使用 hybridize 来拿到两者的好处

大部分的深度学习框架通常在命令式和符号式之间二选一。例如 Theano 和它启发的后来者，例如 TensorFlow，使用了符号式。Chainer 和它的追随者 PyTorch 使用了命令式。在设计 Gluon 的时候我们问了这个问题：可能拿到命令式的全部好处，但仍然享受符号式的优势吗？从另一方面来说，用户应该用纯命令式的方法来使用 Gluon 进行开发和调试。但当需要产品级别的性能和部署的时候，我们可以将代码，至少大部分，转换成符号式来运行。

事实这一点可以做到。我们可以通过使用 `HybridBlock` 或者 `HybridSequential` 来构建神经网络。默认他们跟 `Block` 和 `Sequential` 一样使用命令式执行。当我们调用 `.hybridize()` 后，系统会转换成符号式来执行。事实上，所有 Gluon 里定义的层全是 `HybridBlock`，这个意味着大部分的神经网络都可以享受符号式执行的优势。

8.1.2 HybridSequential

我们之前学习了如何使用 `Sequential` 来串联多个层。如果你想要它跑得飞快，那你应该考虑替换成 `HybridSequential`。

```
In [3]: from mxnet.gluon import nn
        from mxnet import nd

        def get_net():
            net = nn.HybridSequential()
            with net.name_scope():
                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[3]:
[[0.13706432 0.2319649]]
<NDArray 1x2 @cpu(0)>

我们可以通过 `hybridize` 来编译和优化 `HybridSequential`。

```
In [4]: net.hybridize()  
      net(x)  
  
Out[4]:  
[[ 0.13706432  0.2319649 ]]  
<NDArray 1x2 @cpu(0)>
```

注意到只有继承自 `HybridBlock` 的层才会被优化。`HybridSequential` 和 Gluon 提供的层都是它的子类。如果一个层只是继承自 `Block`, 那么我们将跳过优化。我们会接下会讨论如何使用 `HybridBlock`。

8.1.3 性能

我们比较 `hybridize` 前和后的计算时间来展示符号式执行的性能提升。这里我们计时 1000 次 forward:

```
In [5]: from time import time  
  
def bench(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'%(bench(net, x)))  
net.hybridize()  
print('After hybridizing: %.4f sec'%(bench(net, x)))  
  
Before hybridizing: 0.3383 sec  
After hybridizing: 0.1975 sec
```

可以看到 `hybridize` 提供近似两倍的加速。

8.1.4 获取符号式的程序

之前我们给 `net` 输入 `NDArray` 类型的 `x`, 然后 `net(x)` 会直接返回结果。对于调用过 `hybridize()` 后的网络, 我们可以给它输入一个 `Symbol` 类型的变量, 其会返回同样是 `Symbol` 类型的程序。

In [6]: `from mxnet import sym`

```
x = sym.var('data')
y = net(x)
y
```

Out[6]: <Symbol hybridsequential1_dense2_fwd>

我们可以通过 `export()` 来保存这个程序到硬盘。它可以之后不仅被 Python, 同时也可以其他支持的前端语言, 例如 C++, Scala, R…, 读取。

TODO(mli) export 需要 `mxnet>=0.11.1b20171015`, 样例之后放进来。

8.1.5 通过 HybridBlock 深入理解 hybridize 工作机制

前面我们展示了通过 `hybridize` 我们可以获得更好的性能和更高的移植性。现在我们来解释这个是如何影响灵活性的。记得我们提过 gluon 里面的 `Sequential` 是 `Block` 的一个便利形式, 同理, 可以 `HybridSequential` 是 `HybridBlock` 的子类。跟 `Block` 需要实现 `forward` 方法不一样, 对于 `HybridBlock` 我们需要实现 `hybrid_forward` 方法。

In [7]: `class HybridNet(nn.HybridBlock):`

```
def __init__(self, **kwargs):
    super(HybridNet, self).__init__(**kwargs)
    with self.name_scope():
        self.fc1 = nn.Dense(10)
        self.fc2 = nn.Dense(2)

    def hybrid_forward(self, F, x):
        print(F)
        print(x)
        x = F.relu(self.fc1(x))
        print(x)
        return self.fc2(x)
```

`hybrid_forward` 方法加入了额外的输入 `F`, 它使用了 MXNet 的一个独特的特征。MXNet 有一个符号式的 API (`symbol`) 和命令式的 API (`ndarray`)。这两个接口里面的函数基本是一致的。系统会根据输入来决定 `F` 是使用 `symbol` 还是 `ndarray`。

我们实例化一个样例, 然后可以看到默认 `F` 是使用 `ndarray`。而且我们打印出了输入和第一层 `relu` 的输出。

In [8]: `net = HybridNet()`
`net.initialize()`

```
x = nd.random.normal(shape=(1, 4))
y = net(x)
```

```
<module 'mxnet.ndarray' from '/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/m
[[ 0.07395727 -0.39969039 -0.59775633 -0.36118516]
<NDArray 1x4 @cpu(0)>

[[ 0.          0.05385821  0.          0.01470048  0.          0.02936197
  0.01792259  0.02284899  0.01588759  0.04298554]]
<NDArray 1x10 @cpu(0)>
```

再运行一次会得到同样的结果。

```
In [9]: y = net(x)
```

```
<module 'mxnet.ndarray' from '/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/m
[[ 0.07395727 -0.39969039 -0.59775633 -0.36118516]
<NDArray 1x4 @cpu(0)>

[[ 0.          0.05385821  0.          0.01470048  0.          0.02936197
  0.01792259  0.02284899  0.01588759  0.04298554]]
<NDArray 1x10 @cpu(0)>
```

接下来看看 `hybridize` 后会发生什么。

```
In [10]: net.hybridize()
y = net(x)
```

```
<module 'mxnet.symbol' from '/var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/m
<Symbol data>
<Symbol hybridnet0_relu0>
```

可以看到：

1. F 变成了 symbol.
2. 即使输入数据还是 NDArray 的类型，但 `hybrid_forward` 里不论是输入还是中间输出，全部变成了 Symbol

再运行一次看看

```
In [11]: y = net(x)
```

可以看到什么都没有输出。这是因为第一次 `net(x)` 的时候，会先将输入替换成 Symbol 来构建符号式的程序，之后运行的时候系统将不再访问 Python 的代码，而是直接在 C++ 后端执行这个

符号式程序。这是为什么 `hybridize` 后会变快的一个原因。

但它可能的问题是我们损失写程序的灵活性。因为 Python 的代码只执行一次，而且是符号式的执行，那么使用 `print` 来调试，或者使用 `if` 和 `for` 来做复杂的控制都不可能了。

8.1.6 结论

通过 `HybridSequential` 和 `HybridBlock`, 我们可以简单的用 `hybridize` 来将命令式的程序转成符号式程序。我们推荐大家尽可能的使用这个来获得最好的性能加速。

吐槽和讨论欢迎点[这里](#)

8.2 延迟执行

MXNet 使用**延迟执行**来提升系统性能。绝大部分情况下我们不用知道它的存在，因为它不会对正常使用带来影响。但理解它的工作原理有助于开发更高效的程序。

延迟执行是指命令可以等到之后它的结果真正的需要的时候再执行。我们先来看一个例子：

```
In [1]: a = 1 + 1
       # some other things
       print(a)
```

2

第一句对 `a` 赋值，再执行一些其他指令后打印 `a` 的结果。因为这里我们可能很久以后才用 `a` 的值，所以我们可以把它的执行延迟到后面。这样的主要好处是在执行之前系统可以看到后面指令，从而有更多机会来对程序进行优化。例如如果 `a` 在被使用前被重新赋值了，那么我们可以不需要真正执行第一条语句。

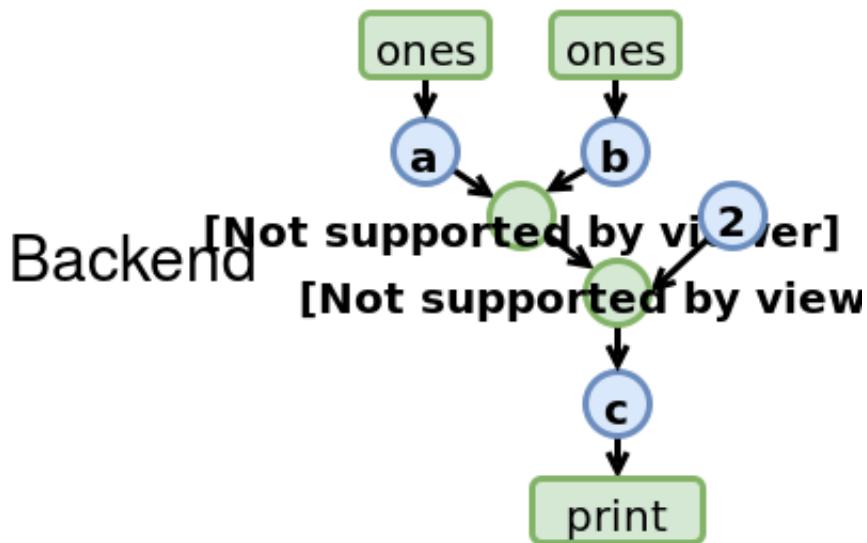
在 MXNet 里，我们把用户打交道的部分叫做前端。例如这个教程里我们一直在使用 Python 前端写代码。除了 Python 外，MXNet 还支持其他例如 Scala, R, C++ 的前端。不管使用什么前端，MXNet 的程序执行主要都在 C++ 后端。前端只是把程序传给后端。后端有自己的线程来不断的收集任务，构造计算图，优化，并执行。本章我们介绍后端优化之一：延迟执行。

考虑下图的样例，我们在前端调用四条语句，它们被后端的线程分析依赖并构建成计算图。

在延迟执行中，前端执行前三个语句的时候，它仅仅是把任务放进后端的队列里就返回了。当在需要打印结果时，前端会等待后端线程把 `c` 的结果计算完。

这个设计的一个好处是前端，就是 Python 线程，不需要做实际计算工作，从而不管 Python 的性能如何，它对整个程序的影响会很小。只需要 C++ 后端足够高效，那么不管前端语言性能如何，都可以提供一致的性能。

Frontend [Not supported by viewer]



下面的例子通过计时来展示了延后执行的效果。可以看到，当 `y=...` 返回的时候并没有等待它真的被计算完。

```
In [2]: from mxnet import nd
        from time import time

        start = time()
        x = nd.random_uniform(shape=(2000,2000))
        y = nd.dot(x, x)
        print('workloads are queued:\t%f sec' % (time() - start))
        print(y)
        print('workloads are finished:\t%f sec' % (time() - start))

workloads are queued: 0.000618 sec
```

[[501.98400879 507.21395874 485.15576172 ..., 491.92022705
 497.25241089 483.55505371]
[491.52926636 503.83093262 480.88876343 ..., 495.77688599
 500.18539429 492.50854492]
[518.2310791 521.22814941 499.03594971 ..., 510.80603027
 508.33139038 505.09320068]

```
...,
[ 503.28393555  506.78427124  490.05950928 ... ,  501.06356812
 500.14096069  493.59692383],
[ 512.7734375   513.4161377   497.60931396 ... ,  499.51367188
 502.92895508  494.03161621],
[ 514.21496582  517.6932373   496.9541626 ... ,  508.26495361
 503.22174072  498.45629883]]]
<NDArray 2000x2000 @cpu(0)>
workloads are finished: 0.195543 sec
```

延迟执行大部分情况是对用户透明的。因为除非我们需要打印或者保存结果外，我们基本不需要关心目前是不是结果在内存里面已经计算好了。

事实上，只要数据是保存在 NDArray 里，而且使用 MXNet 提供的运算子，后端将默认使用延迟执行来获取最大的性能。

8.2.1 立即获取结果

除了前面介绍的 `print` 外，我们还有别的方法可以让前端线程等待直到结果完成。我们可以使用 `nd.NDArray.wait_to_read()` 等待直到特定结果完成，或者 `nd.waitall()` 等待所有前面结果完成。后者是测试性能常用方法。

```
In [3]: start = time()
y = nd.dot(x, x)
y.wait_to_read()
time() - start

Out[3]: 0.12824106216430664
```

```
In [4]: start = time()
y = nd.dot(x, x)
z = nd.dot(x, x)
nd.waitall()
time() - start
```

```
Out[4]: 0.24973082542419434
```

任何方法将内容从 NDArray 搬运到其他不支持延迟执行的数据结构里都会触发等待，例如 `asnumpy()`, `asscalar()`

```
In [5]: start = time()
y = nd.dot(x, x)
y.asnumpy()
time() - start
```

Out[5]: 0.1330885887145996

```
In [6]: start = time()
y = nd.dot(x, x)
y.norm().asscalar()
time() - start
```

Out[6]: 0.12353062629699707

8.2.2 延迟执行带来的便利

下面例子中，我们不断的对 y 进行赋值。如果每次我们需要等到 y 的值，那么我们必须要计算它。而在延迟执行里，系统有可能省略掉一些执行。

```
In [7]: start = time()
```

```
for i in range(1000):
    y = x + 1
    y.wait_to_read()

print('No lazy evaluation: %f sec' % (time()-start))

start = time()
for i in range(1000):
    y = x + 1
nd.waitall()
print('With evaluation: %f sec' % (time()-start))
```

No lazy evaluation: 1.067476 sec

With evaluation: 0.734601 sec

8.2.3 延迟执行带来的影响

在延迟执行里，只要最终结果是一致的，系统可能使用跟代码不一样的顺序来执行，例如假设我们写

```
In [8]: a = 1
b = 2
a + b
```

Out[8]: 3

第一句和第二句之间没有依赖，所以把 $b=2$ 提前到 $a=1$ 前执行也是可以的。但这样可能会导致内存使用的变化。

下面我们列举几个在训练和预测中常见的现象。一般每个批量我们都会评测一下，例如计算损失或者精度，其中会用到 `asscalar` 或者 `asnumpy`。这样我们会每次仅仅将一个批量的任务放进后端系统执行。但如果我们去掉这些同步函数，会导致我们将大量的批量任务同时放进系统，从而可能导致系统占用过多资源。

为了演示这种情况，我们定义一个数据获取函数，它会打印什么数据是什么时候被请求的。

```
In [9]: def get_data():
    start = time()
    batch_size = 1024
    for i in range(60):
        if i % 10 == 0:
            print('batch %d, time %f sec' %(i, time()-start))
        x = nd.ones((batch_size, 1024))
        y = nd.ones((batch_size,))
        yield x, y
```

使用两层网络和 L2 损失函数作为样例

```
In [10]: from mxnet import gluon
         from mxnet.gluon import nn

         net = nn.Sequential()
         with net.name_scope():
             net.add(
                 nn.Dense(1024, activation='relu'),
                 nn.Dense(1024, activation='relu'),
                 nn.Dense(1),
             )
         net.initialize()
         trainer = gluon.Trainer(net.collect_params(), 'sgd', {})
         loss = gluon.loss.L2Loss()
```

我们定义辅助函数来监测内存的使用（只能在 Linux 运行）

```
In [11]: import os
         import subprocess

         def get_mem():
             """get memory usage in MB"""
             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（回忆[延后初始化](#)）。

```
In [12]: for x, y in get_data():
    break
    loss(y, net(x)).wait_to_read()

batch 0, time 0.000002 sec
```

如果我们用 `net` 来做预测, 正常情况下对每个批量的结果我们把它复制出 NDArray, 例如打印或者保存在磁盘上。这里我们简单使用 `wait_to_read` 来模拟。

```
In [13]: mem = get_mem()

for x, y in get_data():
    loss(y, net(x)).wait_to_read()
    nd.waitall()

print('Increased memory %f MB' % (get_mem() - mem))

batch 0, time 0.000002 sec
batch 10, time 0.366813 sec
batch 20, time 0.731830 sec
batch 30, time 1.096895 sec
batch 40, time 1.461910 sec
batch 50, time 1.826849 sec
Increased memory 3.116000 MB
```

假设我们不使用 `wait_to_read()`, 那么前端会将所有批量的计算一次性的添加进后端。可以看到每个批量的数据都会在很短的时间内生成, 同时在接下来的数秒钟内, 我们看到了内存的增长(包括了在内存中保存所有 `x` 和 `y`)。

```
In [14]: mem = get_mem()

for x, y in get_data():
    loss(y, net(x))

    nd.waitall()
    print('Increased memory %f MB' % (get_mem() - mem))

batch 0, time 0.000003 sec
batch 10, time 0.006039 sec
batch 20, time 0.011333 sec
batch 30, time 0.016585 sec
batch 40, time 0.021897 sec
batch 50, time 0.027219 sec
Increased memory 77.740000 MB
```

同样对于训练，如果我们每次计算损失，那么就加入了同步

```
In [15]: from mxnet import autograd

mem = get_mem()

total_loss = 0
for x, y in get_data():
    with autograd.record():
        L = loss(y, net(x))
    total_loss += L.sum().asscalar()
    L.backward()
    trainer.step(x.shape[0])

nd.waitall()
print('Increased memory %f MB' % (get_mem() - mem))

batch 0, time 0.000004 sec
batch 10, time 0.965655 sec
batch 20, time 1.990963 sec
batch 30, time 3.018715 sec
batch 40, time 4.043917 sec
batch 50, time 5.069017 sec
Increased memory -60.672000 MB
```

但如果不去掉同步，同样会首先把数据全部生成好，导致占用大量内存。

```
In [16]: from mxnet import autograd

mem = get_mem()

total_loss = 0
for x, y in get_data():
    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 0.000003 sec
batch 10, time 0.013743 sec
batch 20, time 0.026652 sec
```

```
batch 30, time 0.040668 sec
batch 40, time 0.053465 sec
batch 50, time 0.066189 sec
Increased memory 241.740000 MB
```

8.2.4 总结

延后执行使得系统有更多空间来做性能优化。但我们推荐每个批量里至少有一个同步函数，例如对损失函数进行评估，来避免将过多任务同时丢进后端系统。

8.2.5 练习

为什么同步版本的训练中，我们看到了内存使用的大量下降？

吐槽和讨论欢迎点[这里](#)

8.3 自动并行

在延后执行里我们提到后端系统会自动构建计算图。通过计算图系统可以知道所有计算的依赖关系，有了它系统可以选择将没有依赖关系任务同时执行来获得性能的提升。

仍然考虑下面这个例子，这里 `a = ...` 和 `b = ...` 之间没有数据依赖关系，从而系统可以选择并行执行他们。

通常一个运算符，例如 `+` 或者 `dot`，会用掉一个计算设备上所有计算资源。`dot` 同样用到所有 CPU 的核（即使是有多个 CPU）和单 GPU 上所有线程。因此在单设备上并行运行多个运算符可能效果并不明显。自动并行主要的用途是多设备的计算并行，和计算与通讯的并行。

【注意】本章需要至少一个 GPU 才能运行。

8.3.1 多设备的并行计算

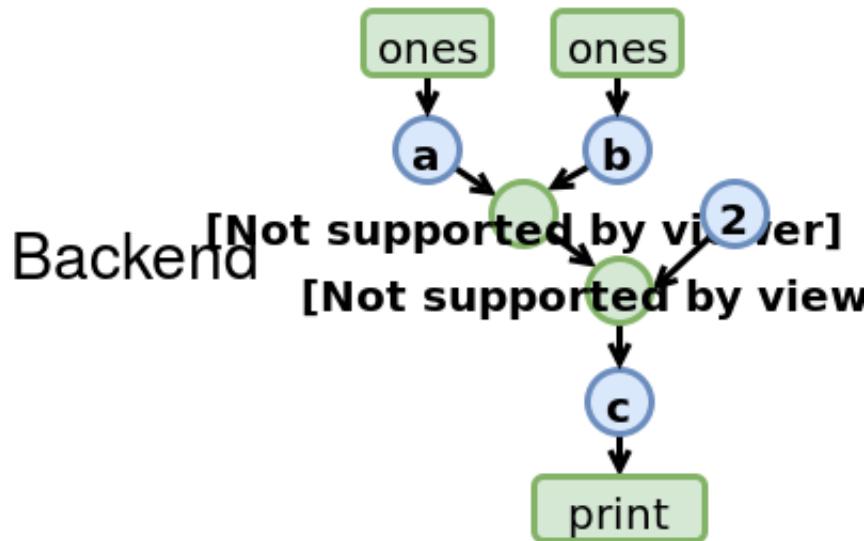
我们首先定义一个函数，它做 10 次矩阵乘法。

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

def run(x):
    """push 10 matrix-matrix multiplications"""
    return [nd.dot(x,x) for i in range(10)]
```

我们分别计算在 CPU 和 GPU 上运行时间

Frontend [Not supported by viewer]



```

In [2]: from mxnet import gpu
        from time import time

        x_cpu = nd.random.uniform(shape=(2000,2000))
        x_gpu = nd.random.uniform(shape=(6000,6000), ctx=gpu(0))
        nd.waitall()

        # warm up
        run(x_cpu)
        run(x_gpu)
        nd.waitall()

        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: 1.223033 sec  
Run on GPU: 1.185213 sec
```

我们去掉两次 `run` 之间的 `waitall`, 希望系统能自动并行这两个任务:

```
In [3]: 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: 1.223693 sec
```

可以看到两个一起执行时, 总时间不是分开执行的总和。这个表示后端系统能有效并行执行它们。

8.3.2 计算和通讯的并行

在多设备计算中, 我们经常需要在设备之间复制数据。例如下面我们在 GPU 上计算, 然后将结果复制回 CPU。

```
In [4]: from mxnet import cpu  
  
def copy_to_cpu(x):  
    """copy data to a device"""  
    return [y.copyto(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: 1.209245 sec  
Copy to CPU: 0.521475 sec
```

同样我们去掉运行和复制之间的 `waitall`:

```
In [5]: 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: 1.253460 sec

可以看到总时间小于前面两者之和。这个任务稍微不同于上面，因为运行和复制之间有依赖关系。就是 $y[i]$ 必须先计算好才能复制到 CPU。但在计算 $y[i]$ 的时候系统可以复制 $y[i-1]$ ，从而获得总运行时间的减少。

8.3.3 总结

MXNet 能够自动并行执行没有数据依赖关系的任务从而提升系统性能。

8.3.4 练习

- `run` 里面计算了 10 次运算，他们也没有依赖关系。看看系统有没有自动并行执行他们
- 试试有更加复杂数据依赖的任务，看看系统能不能得到正确的结果，而且性能有提升吗？

吐槽和讨论欢迎点[这里](#)

8.4 多 GPU 来训练—从 0 开始

本教程我们将展示如何使用多个 GPU 来加速训练。正如你期望的那样，这个教程需要至少两块 GPU 来运行。事实上，一台机器上安装多块 GPU 非常常见，因为通常主板上会有多个 PCIe 插槽。下图是一台服务器上安装了 8 块 Titan X。

如果正确安装了 NVIDIA 驱动，我们可以通过 `nvidia-smi` 来查看当前系统有多少个 GPU。

In [1]: `!nvidia-smi`

Tue Nov 14 06:36:25 2017

NVIDIA-SMI 375.26			Driver Version: 375.26		
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile Uncorr. ECC
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util Compute M.
=	=	=	=	=	=
0	Tesla M60	Off	0000:00:1D.0	Off	0



```

| N/A   39C   P0    38W / 150W |      0MiB / 7612MiB |      0%     Default |
+-----+-----+-----+
|   1   Tesla M60          Off | 0000:00:1E.0     Off |           0 |
| N/A   42C   P0    39W / 150W |      0MiB / 7612MiB |     98%     Default |
+-----+-----+-----+
+
+-----+
| Processes:                                     GPU Memory |
| GPU       PID  Type  Process name             Usage   |
+=====+
| No running processes found                     |
+-----+

```

在自动并行里我们提到虽然大部分的运算可以要么全部使用所有的 CPU 计算资源，或者单 GPU 的资源。但对于多 GPU 的情况，我们仍然需要来实现对应的算法。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面使用最广泛的用来将任务划分到多设备的办法。它是这样工作的，假设这里有 k 个 GPU，每个 GPU 将维护一个模型参数的复制。然后每次我们将一个批量里面的样本划分成 k 块并分给每个 GPU 一块。每个 GPU 使用分到的数据计算梯度。然后我们将所有 GPU

上梯度相加得到这个批量上的完整梯度。之后每个 GPU 使用这个完整梯度对自己维护的模型做更新。

8.4.2 定义模型

我们使用卷积神经网络—从 0 开始里介绍的 LeNet 来作为本章的样例任务。

```
In [2]: from mxnet import nd
        from mxnet import gluon

        # initialize parameters
        scale = .01
        W1 = nd.random.normal(shape=(20,1,3,3))*scale
        b1 = nd.zeros(shape=20)
        W2 = nd.random.normal(shape=(50,20,5,5))*scale
        b2 = nd.zeros(shape=50)
        W3 = nd.random.normal(shape=(800,128))*scale
        b3 = nd.zeros(shape=128)
        W4 = nd.random.normal(shape=(128,10))*scale
        b4 = nd.zeros(shape=10)
        params = [W1, b1, W2, b2, W3, b3, W4, b4]

        # network and loss
        def lenet(X, params):
            # first conv
            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))
            # second conv
            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)
            # first dense
            h3_linear = nd.dot(h2, params[4]) + params[5]
            h3 = nd.relu(h3_linear)
            # second dense
```

```
yhat = nd.dot(h3, params[6]) + params[7]
return yhat

loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

然后我们先实现几个在 GPU 同步数据的辅助函数。

8.4.3 在多 GPU 之间同步数据

下面函数将模型参数复制到某个特定设备并初始化梯度。

In [3]: `from mxnet import gpu`

```
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

# copy param to GPU(0)
new_params = get_params(params, 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.]
<NDArray 20 @gpu(0)>
b1 grad =
[ 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 之间数据，我们定义一个函数它将这些数据加起来，然后再广播到所有 GPU 上。

In [4]: `def allreduce(data):`

```
# sum on data[0].context, and then broadcast
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])
```

```

data = [nd.ones((1,2), ctx=gpu(i))*(i+1) for i in range(2)]
print('Before:', data)
allreduce(data)
print('After:', data)

Before:
[[ 1.  1.]]
<NDArray 1x2 @gpu(0)>,
[[ 2.  2.]]
<NDArray 1x2 @gpu(1)>
After:
[[ 3.  3.]]
<NDArray 1x2 @gpu(0)>,
[[ 3.  3.]]
<NDArray 1x2 @gpu(1)>

```

最后给定一个批量，我们划分它并复制到各个 GPU 上。

```

In [5]: 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)]

batch = nd.arange(16).reshape((4,4))
ctx = [gpu(0), 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.]]
<NDArray 4x4 @cpu(0)>
Load into [gpu(0), gpu(1)]
Output:
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]]
<NDArray 2x4 @gpu(0)>,
[[ 8.  9. 10. 11.]]

```

```
[ 12.  13.  14.  15.]]  
<NDArray 2x4 @gpu(1)>]
```

8.4.4 训练一个批量

现在我们可以实现如何使用数据并行在多个 GPU 上训练一个批量了。

```
In [6]: from mxnet import autograd  
import sys  
sys.path.append('..')  
import utils  
  
def train_batch(data, label, params, ctx, lr):  
    # split the data batch and load them on GPUs  
    data_list = split_and_load(data, ctx)  
    label_list = split_and_load(label, ctx)  
    # run forward on each GPU  
    with autograd.record():  
        losses = [loss(lenet(X, W), Y)  
                  for X, Y, W in zip(data_list, label_list, params)]  
    # run backward on each gpu  
    for l in losses:  
        l.backward()  
    # aggregate gradient over GPUs  
    for i in range(len(params[0])):  
        allreduce([params[c][i].grad for c in range(len(ctx))])  
    # update parameters with SGD on each GPU  
    for p in params:  
        utils.SGD(p, lr/data.shape[0])
```

8.4.5 开始训练

现在我们可以定义完整的训练函数。这个跟前面教程里没有什么区别。

```
In [7]: from time import time  
  
def train(num_gpus, batch_size, lr):  
    train_data, test_data = utils.load_data_fashion_mnist(batch_size)  
  
    ctx = [gpu(i) for i in range(num_gpus)]  
    print('Running on', ctx)
```

```
# copy parameters to all GPUs
dev_params = [get_params(params, c) for c in ctx]

for epoch in range(5):
    # train
    start = time()
    for data, label in train_data:
        train_batch(data, label, dev_params, ctx, lr)
    nd.waitall()
    print('Epoch %d, training time = %.1f sec'%(epoch, time()-start))

    # validating on GPU 0
    net = lambda data: lenet(data, dev_params[0])
    test_acc = utils.evaluate_accuracy(test_data, net, ctx[0])
    print('validation accuracy = %.4f'%(test_acc))
```

首先我们使用一个 GPU 来训练。

In [8]: `train(1, 256, 0.3)`

```
Running on [gpu(0)]
Epoch 0, training time = 2.6 sec
    validation accuracy = 0.1001
Epoch 1, training time = 2.1 sec
    validation accuracy = 0.6712
Epoch 2, training time = 2.1 sec
    validation accuracy = 0.7687
Epoch 3, training time = 2.1 sec
    validation accuracy = 0.8116
Epoch 4, training time = 2.2 sec
    validation accuracy = 0.8194
```

使用多个 GPU 但不改变其他参数会得到跟单 GPU 一致的结果（但数据是随机顺序，所以会有细微区别）

In [9]: `train(2, 256, 0.3)`

```
Running on [gpu(0), gpu(1)]
Epoch 0, training time = 1.9 sec
    validation accuracy = 0.2565
Epoch 1, training time = 1.8 sec
    validation accuracy = 0.7471
```

```
Epoch 2, training time = 1.8 sec
    validation accuracy = 0.8086
Epoch 3, training time = 1.8 sec
    validation accuracy = 0.8218
Epoch 4, training time = 1.8 sec
    validation accuracy = 0.8308
```

但在多 GPU 时，通常我们需要增加批量大小使得每个 GPU 能得到足够多的任务来保证性能。但一个大的批量大小可能使得收敛变慢。这时候的一个常用做法是将学习率增大些。

```
In [10]: train(2, 512, 0.6)
```

```
Running on [gpu(0), gpu(1)]
Epoch 0, training time = 1.4 sec
    validation accuracy = 0.0995
Epoch 1, training time = 1.4 sec
    validation accuracy = 0.1573
Epoch 2, training time = 1.4 sec
    validation accuracy = 0.6169
Epoch 3, training time = 1.4 sec
    validation accuracy = 0.7015
Epoch 4, training time = 1.4 sec
    validation accuracy = 0.7949
```

可以看到使用两个 GPU 能有效的减少训练时间。

8.4.6 结论

数据并行可以有效的在多 GPU 上提升训练性能。

8.4.7 练习

- 试试不同的批量大小和学习率
- 将预测也改成多 GPU 版本
- 注意到我们使用 GPU 0 来做梯度求和，会有带来什么问题吗？

吐槽和讨论欢迎点[这里](#)

8.5 多 GPU 来训练—使用 Gluon

在 Gluon 里可以很容易的使用数据并行。在多 GPU 来训练—从 0 开始里我们手动实现了几个数据同步函数来使用数据并行, Gluon 里实现了同样的功能。

8.5.1 多设备上的初始化

之前我们介绍了如果使用 `initialize()` 里的 `ctx` 在 CPU 或者特定 GPU 上初始化模型。事实上, `ctx` 可以接受一系列的设备, 它会将初始好的参数复制所有的设备上。

这里我们使用之前介绍 Resnet18 来作为演示。

```
In [1]: import sys
        sys.path.append('..')
        import utils
        from mxnet import gpu
        from mxnet import cpu

        net = utils.resnet18(10)
        ctx = [gpu(0), gpu(1)]
        net.initialize(ctx=ctx)
```

记得前面提到的[延迟初始化](#), 这里参数还没有被初始化。我们需要先给定数据跑一次。

Gluon 提供了之前我们实现的 `split_and_load` 函数, 它将数据分割并返回各个设备上的复制。然后根据输入的设备, 计算也会在相应的数据上执行。

```
In [2]: from mxnet import nd
        from mxnet import gluon

        x = nd.random.uniform(shape=(4, 1, 28, 28))
        x_list = gluon.utils.split_and_load(x, ctx)
        print(net(x_list[0]))
        print(net(x_list[1]))

[[ 0.02322223  0.03840514 -0.08426391 -0.09523742  0.07289453 -0.00830653
 -0.05956023 -0.04624154 -0.07814114 -0.0534247 ]
 [ 0.0084       0.03061475 -0.09439502 -0.10653993  0.09124557 -0.0092835
 -0.08189345 -0.0349006  -0.08704413 -0.05281062]]
<NDArray 2x10 @gpu(0)>

[[ 0.01711464  0.04199681 -0.09543805 -0.09148098  0.07008949 -0.00863865
```

```
-0.07488217 -0.04885159 -0.08255464 -0.05474427]
[ 0.0287668  0.0228651 -0.09766636 -0.09784378  0.07257111 -0.00666697
 -0.07330478 -0.04908057 -0.0876241  -0.05890433]
<NDArray 2x10 @gpu(1)>
```

这时候我们可以来看初始的过程发生了什么了。记得我们可以通过 `data` 来访问参数值, 它默认会返回 CPU 上值。但这里我们只在两个 GPU 上初始化了, 在访问的对应设备的值的时候, 我们需要指定设备。

```
In [3]: weight = net[1].params.get('weight')
print(weight.data(ctx[0])[0])
print(weight.data(ctx[1])[0])
try:
    weight.data(cpu())
except:
    print('Not initialized on', cpu())
```

```
[[[ 0.01847461 -0.03004881 -0.02461551]
 [-0.01465906 -0.05932271 -0.0595007 ]
 [ 0.0434817   0.04195441  0.05774786]]]
<NDArray 1x3x3 @gpu(0)>
```

```
[[[ 0.01847461 -0.03004881 -0.02461551]
 [-0.01465906 -0.05932271 -0.0595007 ]
 [ 0.0434817   0.04195441  0.05774786]]]
<NDArray 1x3x3 @gpu(1)>
Not initialized on cpu(0)
```

上一章我们提到过如何在多 GPU 之间复制梯度求和并广播, 这个在 `gluon.Trainer` 里面会被默认执行。这样我们可以实现完整的训练函数了。

8.5.2 训练

```
In [4]: from mxnet import gluon
from mxnet import autograd
from time import time
from mxnet import init

def train(num_gpus, batch_size, lr):
    train_data, test_data = utils.load_data_fashion_mnist(batch_size)

    ctx = [gpu(i) for i in range(num_gpus)]
```

```
print('Running on', ctx)

net = utils.resnet18(10)
net.initialize(init=init.Xavier(), ctx=ctx)
loss = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(
    net.collect_params(),'sgd', {'learning_rate': lr})

for epoch in range(5):
    start = time()
    total_loss = 0
    for data, label in train_data:
        data_list = gluon.utils.split_and_load(data, ctx)
        label_list = gluon.utils.split_and_load(label, ctx)
        with autograd.record():
            losses = [loss(net(X), y) for X, y in zip(
                data_list, label_list)]
            for l in losses:
                l.backward()
        total_loss += sum([l.sum().asscalar() for l in losses])
        trainer.step(batch_size)

    nd.waitall()
    print('Epoch %d, training time = %.1f sec'%(epoch, time()-start))

    test_acc = utils.evaluate_accuracy(test_data, net, ctx[0])
    print('validation accuracy = %.4f'%(test_acc))
```

尝试在单 GPU 上执行。

In [5]: train(1, 256, .1)

```
Running on [gpu(0)]
Epoch 0, training time = 15.0 sec
    validation accuracy = 0.8796
Epoch 1, training time = 14.3 sec
    validation accuracy = 0.8994
Epoch 2, training time = 14.3 sec
    validation accuracy = 0.9006
Epoch 3, training time = 14.6 sec
    validation accuracy = 0.8887
```

```
Epoch 4, training time = 14.4 sec
    validation accuracy = 0.9140
```

同样的参数，但使用两个 GPU。

```
In [6]: train(2, 256, .1)
```

```
Running on [gpu(0), gpu(1)]
Epoch 0, training time = 11.0 sec
    validation accuracy = 0.8806
Epoch 1, training time = 10.4 sec
    validation accuracy = 0.9022
Epoch 2, training time = 10.3 sec
    validation accuracy = 0.8954
Epoch 3, training time = 10.2 sec
    validation accuracy = 0.9060
Epoch 4, training time = 10.4 sec
    validation accuracy = 0.9137
```

增大批量值和学习率

```
In [7]: train(2, 512, .2)
```

```
Running on [gpu(0), gpu(1)]
Epoch 0, training time = 8.6 sec
    validation accuracy = 0.8175
Epoch 1, training time = 8.4 sec
    validation accuracy = 0.8930
Epoch 2, training time = 8.4 sec
    validation accuracy = 0.8969
Epoch 3, training time = 8.5 sec
    validation accuracy = 0.8631
Epoch 4, training time = 8.3 sec
    validation accuracy = 0.8932
```

8.5.3 结论

Gluon 的参数初始化和 Trainer 都支持多设备，从单设备到多设备非常容易。

8.5.4 练习

- 跟多 GPU 来训练—从 0 开始不一样，这里我们使用了更现代些的 ResNet。看看不同的批量大小和学习率对不同 GPU 个数上的不一样。

- 有时候各个设备计算能力不一样，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？

吐槽和讨论欢迎点[这里](#)

计算机视觉

9.1 图片增广

AlexNet 当年能取得巨大的成功，其中图片增广功不可没。图片增广通过一系列的随机变化生成大量“新”的样本，从而减低过拟合的可能。现在在深度卷积神经网络训练中，图片增广是必不可少的一部分。

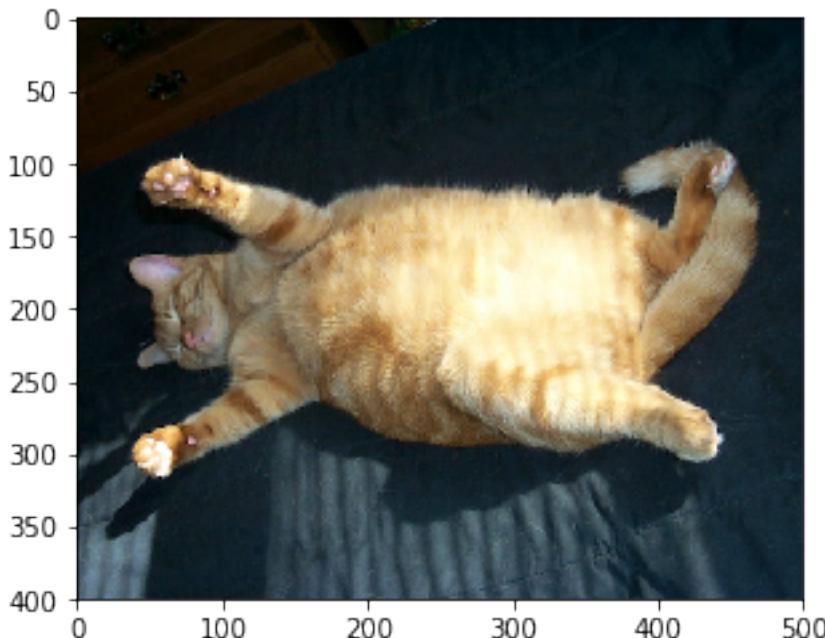
9.1.1 常用增广方法

我们首先读取一张 400×500 的图片作为样例

```
In [1]: %matplotlib inline
import matplotlib.pyplot as plt
from mxnet import image

img = image.imread(open('../img/cat1.jpg', 'rb').read())
plt.imshow(img.asnumpy())
```

Out[1]: <matplotlib.image.AxesImage at 0x7f99a4177710>



接下来我们定义一个辅助函数，给定输入图片 `img` 的增广方法 `aug`，它会运行多次并画出结果。

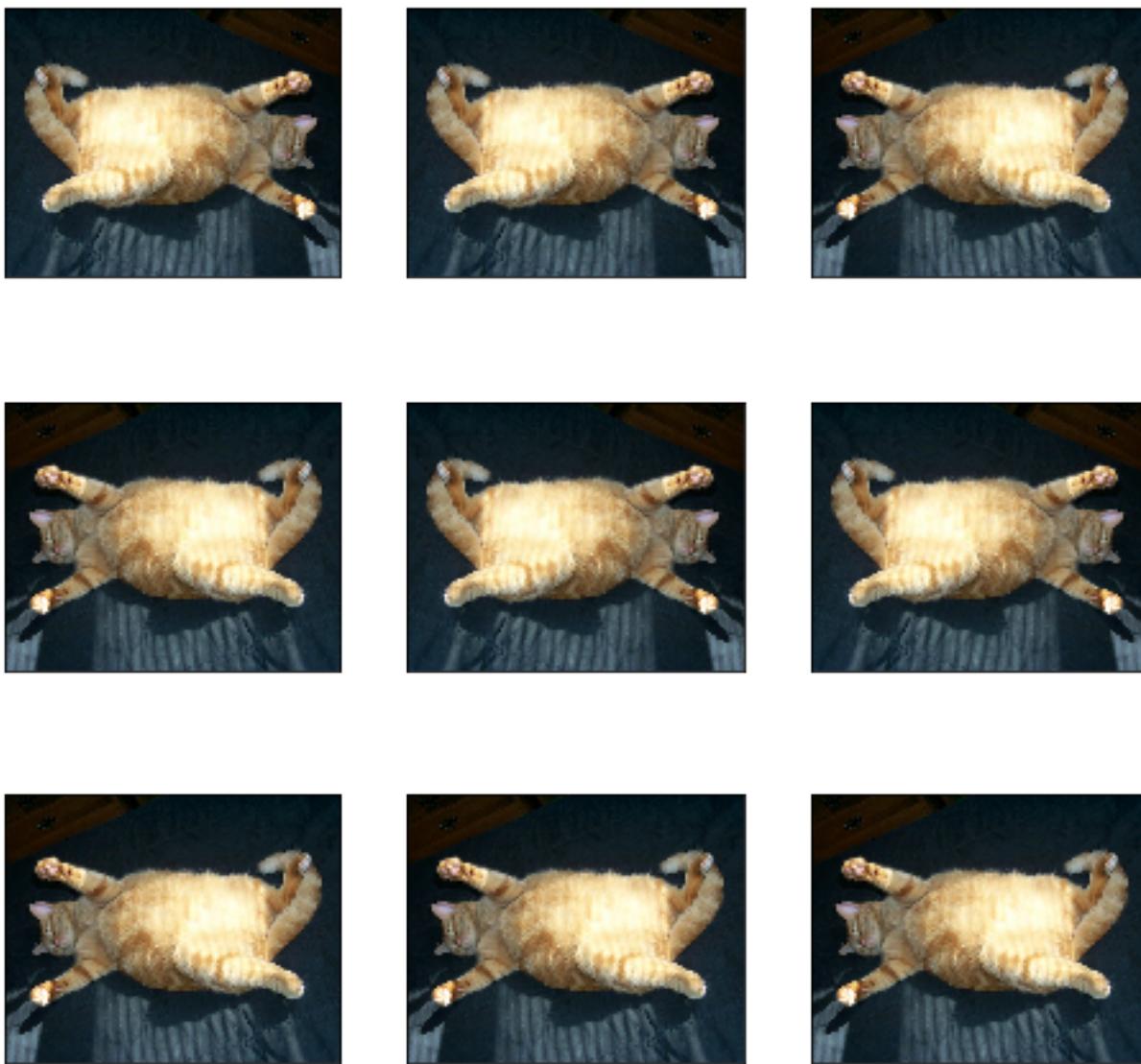
```
In [2]: from mxnet import nd
import sys
sys.path.append('..')
import utils

def apply(img, aug, n=3):
    # 转成 float, 一是因为 aug 需要 float 类型数据来方便做变化。
    # 二是这里会有一次 copy 操作, 因为有些 aug 直接通过改写输入
    # (而不是新建输出) 获取性能的提升
    X = [aug(img.astype('float32')) for _ in range(n*n)]
    # 有些 aug 不保证输入是合法值, 所以做一次 clip
    # 显示浮点图片时 imshow 要求输入在 [0,1] 之间
    Y = nd.stack(*X).clip(0,255)/255
    utils.show_images(Y, n, n, figsize=(8,8))
```

变形

水平方向翻转图片是最早也是最广泛使用的一种增广。

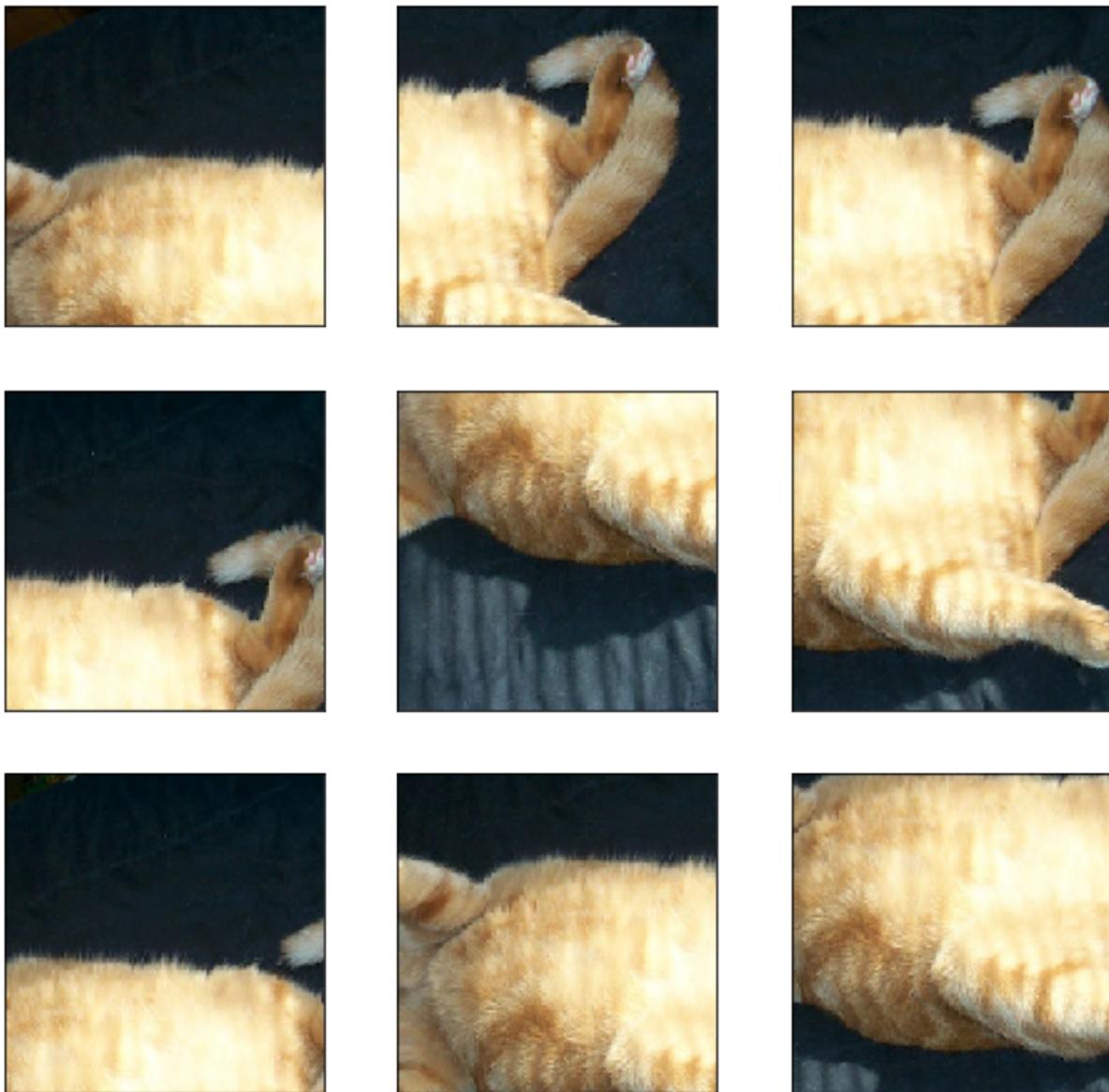
```
In [3]: # 以.5 的概率做翻转
aug = image.HorizontalFlipAug(.5)
apply(img, aug)
```



样例图片里我们关心的猫在图片正中间，但一般情况下可能不是这样。前面我们提到池化层能弱化卷积层对目标位置的敏感度，但也不能完全解决这个问题。一个常用增广方法是随机的截取其中的一块。

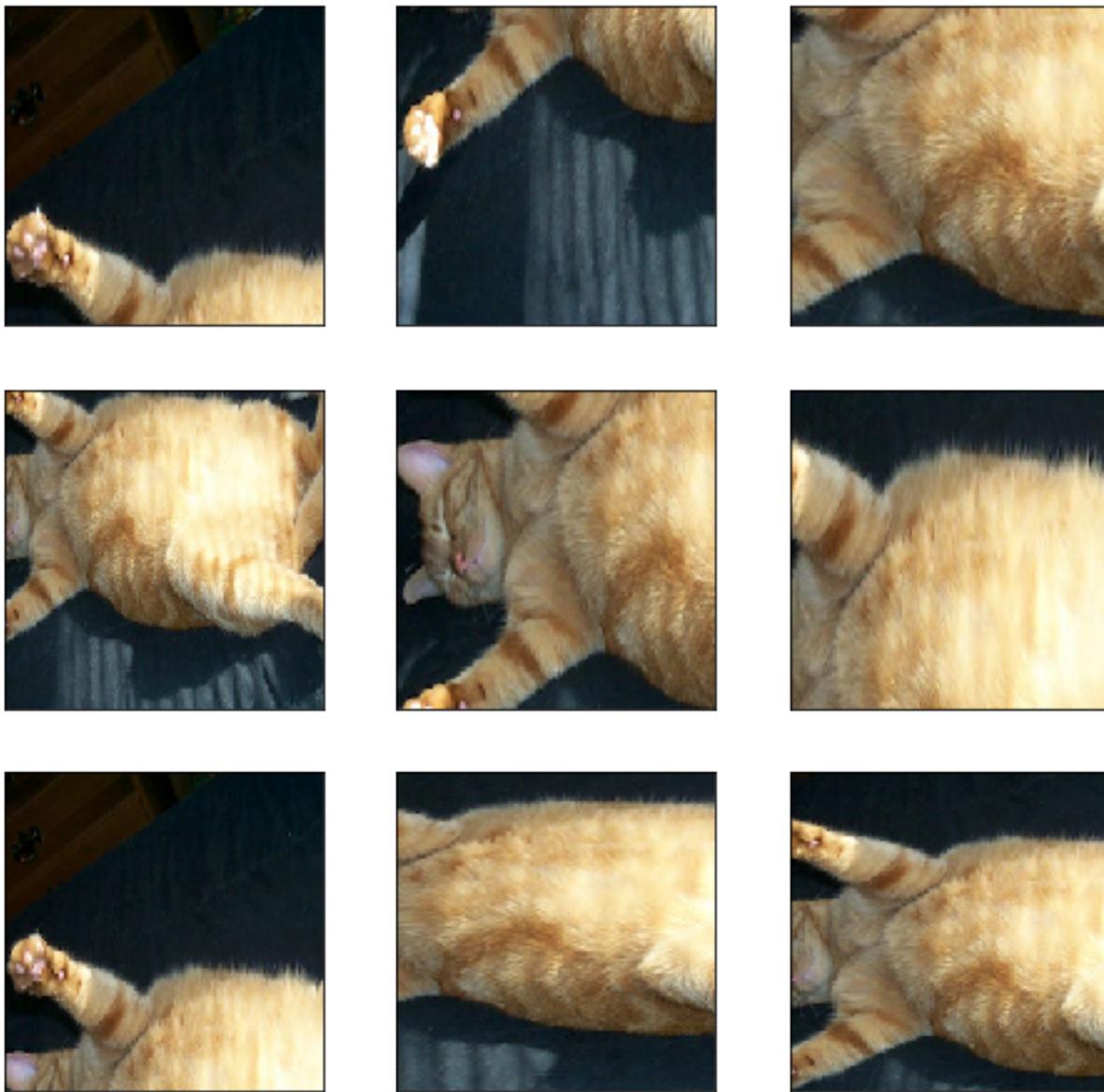
注意到随机截取一般会缩小输入的形状。如果原始输入图片过小，导致没有太多空间进行随机裁剪，通常做法是先将其放大的足够大的尺寸。所以如果你的原始图片足够大，建议不要事先将它们裁到网络需要的大小。

```
In [4]: # 随机裁剪一个块 200 x 200 的区域
aug = image.RandomCropAug([200,200])
apply(img, aug)
```



我们也可以随机裁剪一块随机大小的区域

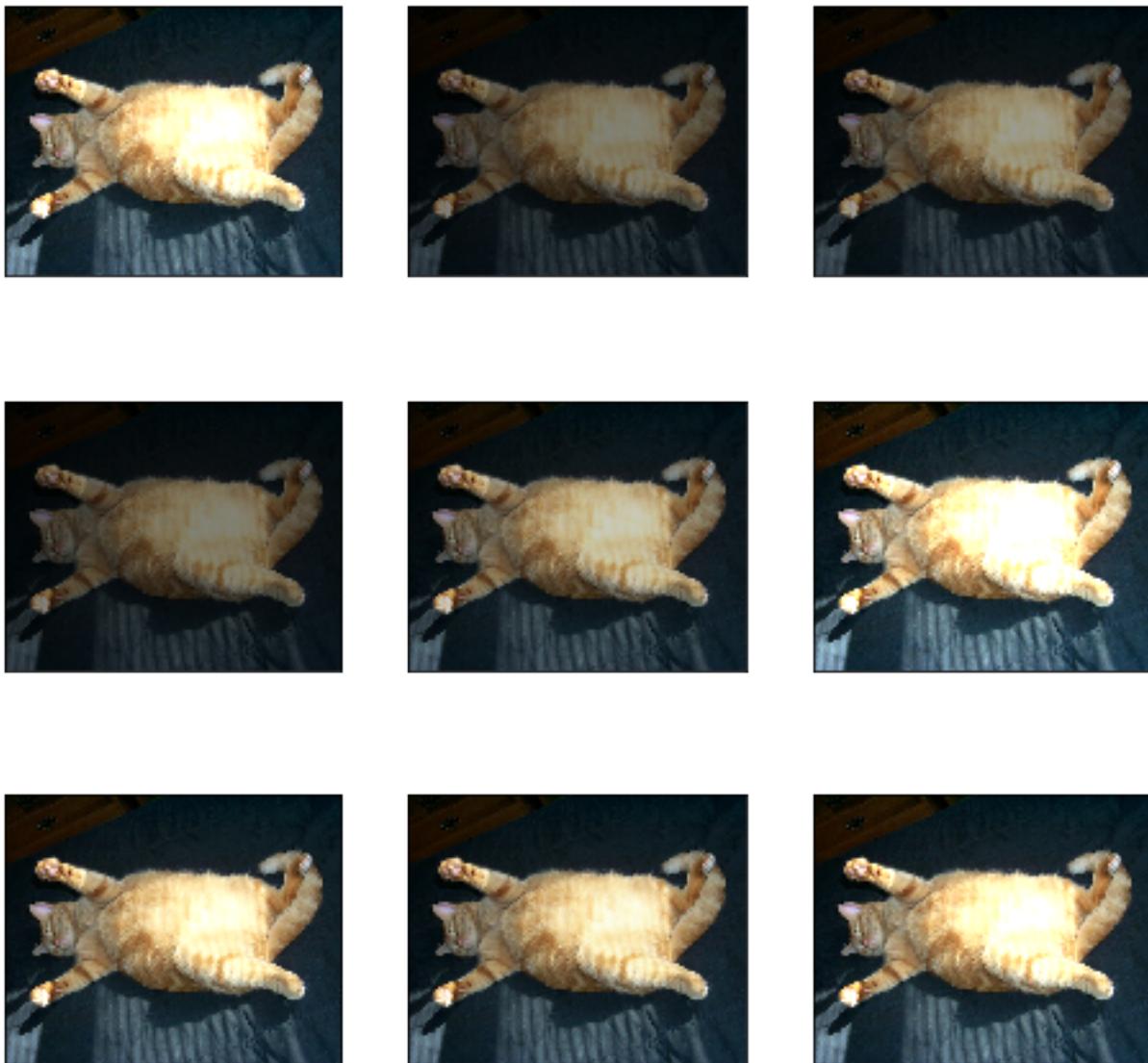
```
In [5]: # 随机裁剪, 要求保留至少 .1 的区域, 随机长宽比在 .5 和 2 之间。  
# 最后将结果 resize 到 200x200  
aug = image.RandomSizedCropAug((200,200), .1, (.5,2))  
apply(img, aug)
```



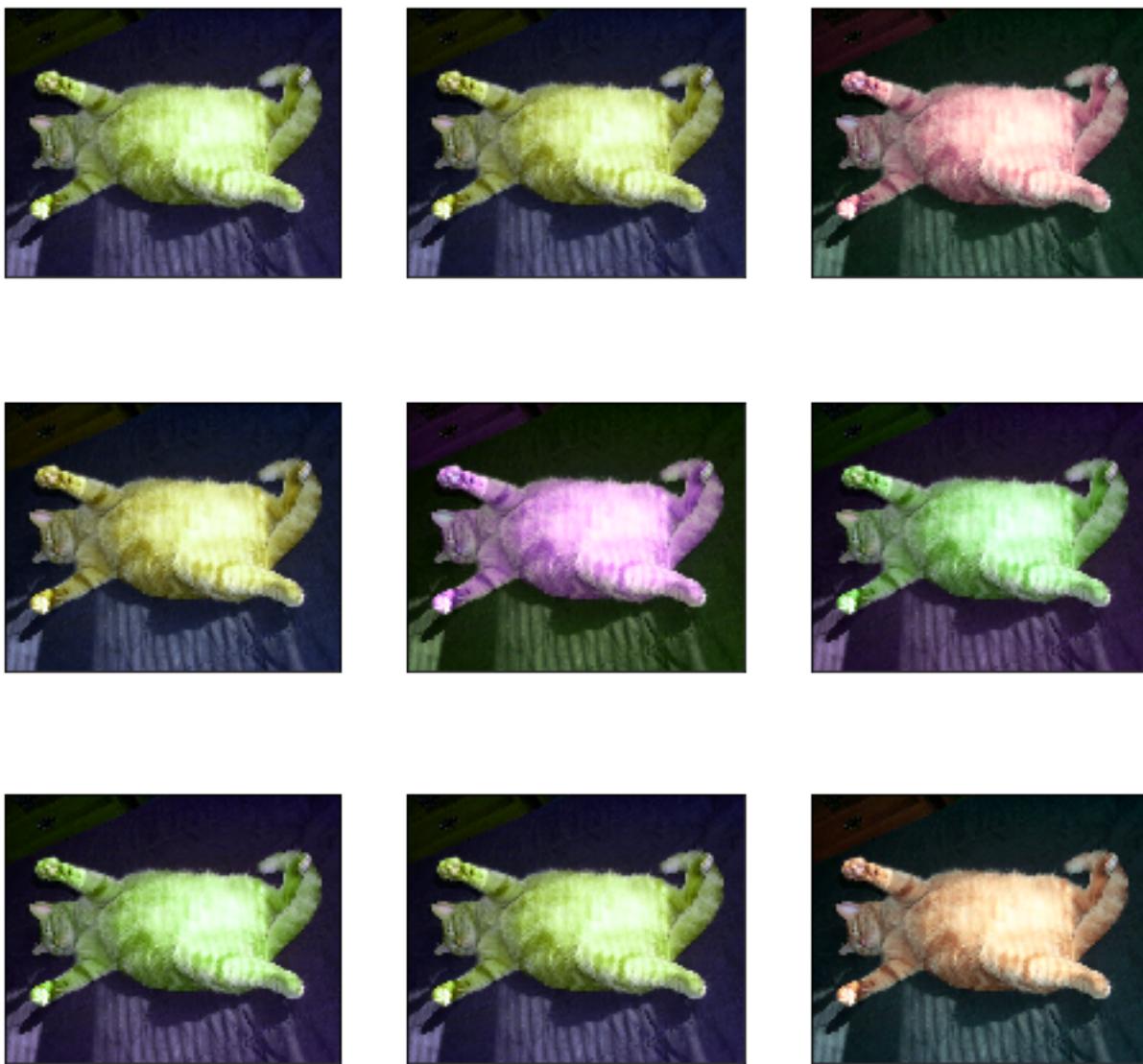
颜色变化

形状变化外的一个另一大类是变化颜色。

```
In [6]: # 随机将亮度增加或者减小在 0-50% 间的一个量  
aug = image.BrightnessJitterAug(.5)  
apply(img, aug)
```



```
In [7]: # 随机色调变化  
aug = image.HueJitterAug(.5)  
apply(img, aug)
```



9.1.2 如何使用

通常使用时我们会将数个增广方法一起使用。注意到图片增广通常只是针对训练数据，对于测试数据则用得较小。后者常用的是做 5 次随机剪裁，然后讲 5 张图片的预测结果做均值。

下面我们使用 CIFAR10 来演示图片增广对训练的影响。我们这里不使用前面一直用的 Fashion-MNIST，这是因为这个数据的图片基本已经对齐好了，而且是黑白图片，所以不管是变形还是变色增广效果都不会明显。

数据读取

我们首先定义一个辅助函数可以对图片按顺序应用数个增广：

```
In [8]: def apply_aug_list(img, augs):
    for f in augs:
        img = f(img)
    return img
```

对于训练图片我们随机水平翻转和剪裁。对于测试图片仅仅就是中心剪裁。CIFAR10 图片尺寸是 $32 \times 32 \times 3$, 我们剪裁成 $28 \times 28 \times 3$.

```
In [9]: train_augs = [
    image.HorizontalFlipAug(.5),
    image.RandomCropAug((28,28))
]

test_augs = [
    image.CenterCropAug((28,28))
]
```

然后定义数据读取, 这里跟前面的 FashionMNIST 类似, 但在 `transform` 中加入了图片增广:

```
In [10]: from mxnet import gluon
from mxnet import nd
import sys
sys.path.append('..')
import utils

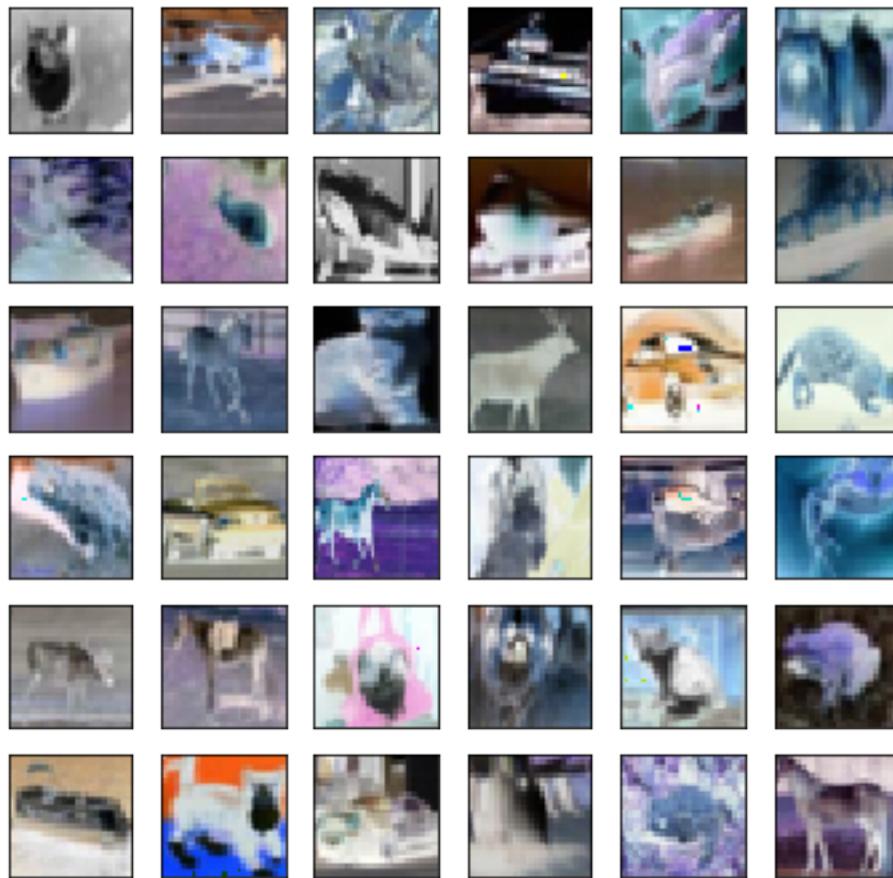
def get_transform(augs):
    def transform(data, label):
        # data: sample x height x width x channel
        # label: sample
        data = data.astype('float32')
        if augs is not None:
            # apply to each sample one-by-one and then stack
            data = nd.stack([
                apply_aug_list(d, augs) for d in data])
        data = nd.transpose(data, (0,3,1,2))
        return data, label.astype('float32')
    return transform

def get_data(batch_size, train_augs, test_augs=None):
    cifar10_train = gluon.data.vision.CIFAR10(
```

```
    train=True, transform=get_transform(train_augs))
cifar10_test = gluon.data.vision.CIFAR10(
    train=False, transform=get_transform(test_augs))
train_data = utils.DataLoader(
    cifar10_train, batch_size, shuffle=True)
test_data = utils.DataLoader(
    cifar10_test, batch_size, shuffle=False)
return (train_data, test_data)
```

画出前几张看看

```
In [11]: train_data, _ = get_data(36, train_augs)
for imgs, _ in train_data:
    break
utils.show_images(imgs.transpose((0,2,3,1)), 6, 6)
```



```
In [12]: imgs.shape
```

```
Out[12]: (36, 3, 28, 28)
```

9.1.3 训练

我们使用*ResNet 18*训练。并且训练代码整理成一个函数使得可以重读调用：

In [13]: `from mxnet import init`

```
def train(train_augs, test_augs, learning_rate=.1):
    batch_size = 128
    num_epochs = 10
    ctx = utils.try_all_gpus()
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    train_data, test_data = get_data(
        batch_size, train_augs, test_augs)
    net = utils.resnet18(10)
    net.initialize(ctx=ctx, init=init.Xavier())
    net.hybridize()
    trainer = gluon.Trainer(net.collect_params(),
                           'sgd', {'learning_rate': learning_rate})
    utils.train(
        train_data, test_data, net, loss, trainer, ctx, num_epochs)
```

使用增广：

In [14]: `train(train_augs, test_augs)`

```
Start training on [gpu(0), gpu(1)]
Epoch 0. Loss: 1.489, Train acc 0.47, Test acc 0.56, Time 18.4 sec
Epoch 1. Loss: 1.080, Train acc 0.62, Test acc 0.63, Time 17.9 sec
Epoch 2. Loss: 0.905, Train acc 0.68, Test acc 0.69, Time 18.0 sec
Epoch 3. Loss: 0.775, Train acc 0.73, Test acc 0.74, Time 17.7 sec
Epoch 4. Loss: 0.702, Train acc 0.76, Test acc 0.77, Time 18.0 sec
Epoch 5. Loss: 0.641, Train acc 0.78, Test acc 0.75, Time 18.2 sec
Epoch 6. Loss: 0.594, Train acc 0.79, Test acc 0.77, Time 18.0 sec
Epoch 7. Loss: 0.555, Train acc 0.81, Test acc 0.79, Time 18.0 sec
Epoch 8. Loss: 0.519, Train acc 0.82, Test acc 0.80, Time 17.9 sec
Epoch 9. Loss: 0.488, Train acc 0.83, Test acc 0.80, Time 17.7 sec
```

不使用增广：

In [15]: `train(test_augs, test_augs)`

```
Start training on [gpu(0), gpu(1)]
Epoch 0. Loss: 1.451, Train acc 0.48, Test acc 0.56, Time 16.4 sec
Epoch 1. Loss: 0.971, Train acc 0.66, Test acc 0.65, Time 16.2 sec
Epoch 2. Loss: 0.752, Train acc 0.74, Test acc 0.67, Time 16.1 sec
```

```
Epoch 3. Loss: 0.588, Train acc 0.80, Test acc 0.73, Time 16.1 sec
Epoch 4. Loss: 0.455, Train acc 0.84, Test acc 0.73, Time 16.1 sec
Epoch 5. Loss: 0.349, Train acc 0.88, Test acc 0.73, Time 16.0 sec
Epoch 6. Loss: 0.255, Train acc 0.91, Test acc 0.74, Time 16.5 sec
Epoch 7. Loss: 0.189, Train acc 0.94, Test acc 0.74, Time 16.3 sec
Epoch 8. Loss: 0.134, Train acc 0.95, Test acc 0.71, Time 16.2 sec
Epoch 9. Loss: 0.101, Train acc 0.97, Test acc 0.75, Time 16.0 sec
```

可以看到使用增广后，训练精度提升更慢，但测试精度比不使用更好。

9.1.4 总结

图片增广可以有效避免过拟合。

9.1.5 练习

尝试换不同的增广方法试试。

吐槽和讨论欢迎点[这里](#)

9.2 Fine-tuning: 通过微调来迁移学习

在前面的章节里我们展示了如何训练神经网络来识别小图片里的问题。我们也介绍了 ImageNet 这个学术界默认的数据集，它有超过一百万的图片和一千类的物体。这个数据集很大的改变计算机视觉这个领域，展示了很多事情虽然在小的数据集上做不到，但在数 GB 的大数据上是可能的。事实上，我们目前还不知道有什么技术可以在类似的但小图片数据集上，例如一万张图片，训练出一个同样强大的模型。

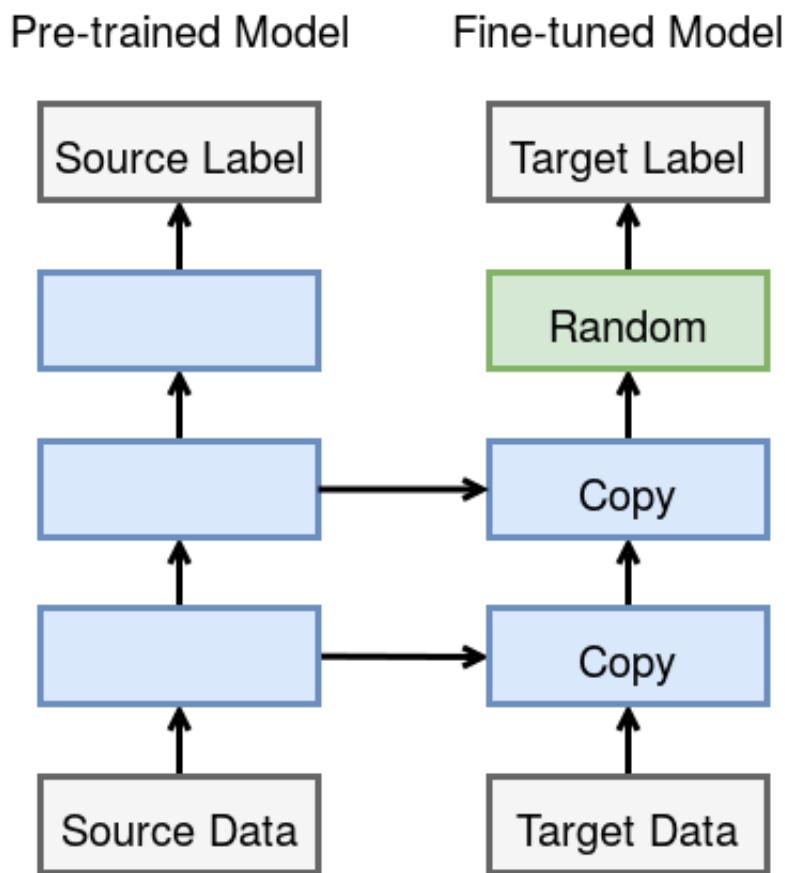
所以这是一个问题。尽管深度卷积神经网络在 ImageNet 上有了很惊讶的结果，但大部分人不关心 Imagenet 这个数据集本身。他们关心他们自己的问题。例如通过图片里面的人脸识别身份，或者识别图片里面的 10 种不同的珊瑚。通常大部分在非 BAT 类似大机构里的人在解决计算机视觉问题的时候，能获得的只是相对来说中等规模的数据。几百张图片很正常，找到几千张图片也有可能，但很难同 Imagenet 一样获得上百万张图片。

于是我们会有一个很自然的问题，如何使用在百万张图片上训练出来的强大的模型来帮助提升在小数据集上的精度呢？这种在源数据上训练，然后将学到的知识应用到目标数据集上的技术通常被叫做**迁移学习**。幸运的是，我们有一些有效的技术来解决这个问题。

对于深度神经网络来说，最为流行的一个方法叫做**微调**（fine-tuning）。它的想法很简单但有效：

- 在源数据 S 上训练一个神经网络。
- 砍掉它的头，将它的输出层改成适合目标数据 S 的大小
- 将输出层的权重初始化成随机值，但其它层保持跟原先训练好的权重一致
- 然后开始在目标数据集开始训练

下图图示了这个算法：



9.2.1 热狗识别

这一章我们将通过[ResNet](#)来演示如何进行微调。因为通常不会每次从 0 开始在 ImageNet 上训练模型，我们直接从 Gluon 的模型园下载已经训练好的。然后将其迁移到一个我们感兴趣的问题上：识别热狗。

热狗识别是一个二分类问题。我们这里使用的热狗数据集是从网上抓取的，它有 1400 张正类和同样多的负类，负类主要是食品相关图片。我们将各类的 1000 张作为训练集合，其余的作为测试集合。

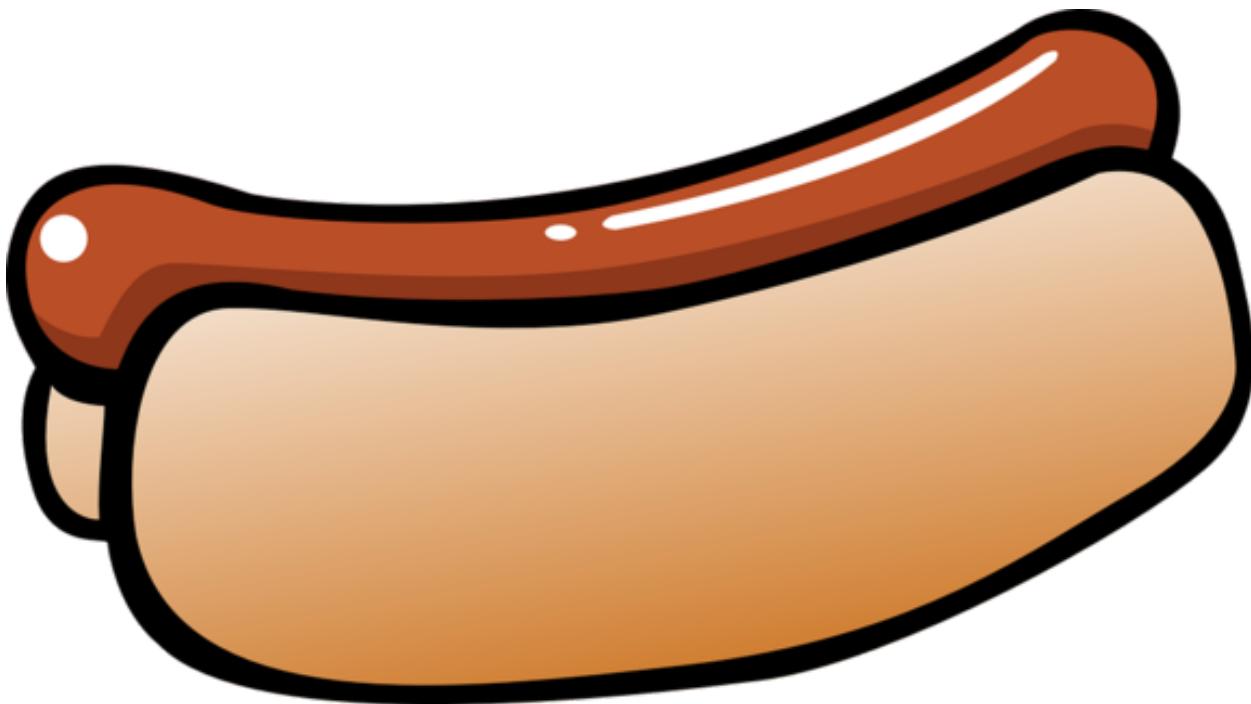


Fig. 9.1: hot dog

获取数据

我们首先从网上下载数据并解压到`../data/hotdog`。每个文件夹下会有对应的 png 文件。

```
In [1]: from mxnet import gluon  
import zipfile  
  
data_dir = '../data'  
fname = gluon.utils.download(  
    'https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/dataset/hotdog.zip',  
    path=data_dir, sha1_hash='fba480ffa8aa7e0febbb511d181409f899b9baa5')
```

```
with zipfile.ZipFile(fname, 'r') as f:  
    f.extractall(data_dir)
```

```
Downloading ../data/hotdog.zip from https://apache-mxnet.s3-accelerate.amazonaws.com/gluon
```

我们使用图片增强里类似的方法来处理图片。

```
In [2]: from mxnet import nd  
from mxnet import image  
from mxnet import gluon  
  
train_augs = [
```

```
        image.HorizontalFlipAug(.5),
        image.RandomCropAug((224,224))
    ]

test_augs = [
    image.CenterCropAug((224,224))
]

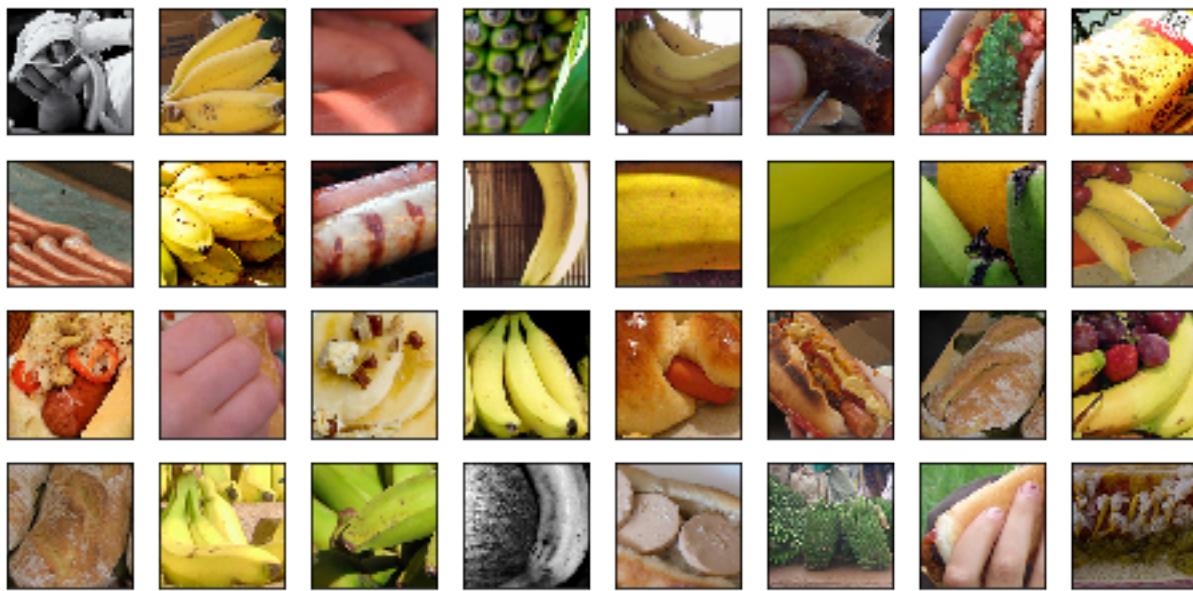
def transform(data, label, augs):
    data = data.astype('float32')
    for aug in augs:
        data = aug(data)
    data = nd.transpose(data, (2,0,1))
    return data, nd.array([label]).asscalar().astype('float32')
```

读取文件夹下的图片，并且画出一些图片

```
In [3]: %matplotlib inline
import sys
sys.path.append('..')
import utils

train_imgs = gluon.data.vision.ImageFolderDataset(
    data_dir+'/hotdog/train',
    transform=lambda X, y: transform(X, y, train_augs))
test_imgs = gluon.data.vision.ImageFolderDataset(
    data_dir+'/hotdog/test',
    transform=lambda X, y: transform(X, y, test_augs))

data = gluon.data.DataLoader(train_imgs, 32, shuffle=True)
for X, _ in data:
    X = X.transpose((0,2,3,1)).clip(0,255)/255
    utils.show_images(X, 4, 8)
    break
```



模型和训练

这里我们将使用 Gluon 提供的 ResNet18 来训练。我们先从模型园里获取改良过 ResNet。使用 `pretrained=True` 将会自动下载并加载从 ImageNet 数据集上训练而来的权重。

```
In [4]: from mxnet.gluon.model_zoo import vision as models
```

```
pretrained_net = models.resnet18_v2(pretrained=True)
```

通常预训练好的模型由两块构成，一是 `features`，二是 `output`。后者主要包括最后一层全连接层，前者包含从输入开始的大部分层。这样的划分的一个主要目的是为了更方便做微调。我们先看下 `output` 的内容：

```
In [5]: pretrained_net.output
```

```
Out[5]: Dense(512 -> 1000, linear)
```

我们可以看一下第一个卷积层的部分权重。

```
In [6]: pretrained_net.features[1].weight.data()[0][0]
```

```
Out[6]:
```

```
[[-0.04693392  0.11487006 -0.13209556  0.16124195 -0.21484604  0.18044543
 -0.05956454]
 [-0.00242769 -0.03129578  0.01799692  0.15277492 -0.41541672  0.38176033
 -0.13370997]
 [ 0.10314132 -0.30472746  0.59482247 -0.52606624  0.0621427   0.25646785
 -0.12772678]]
```

```
[ 0.01783164 -0.21222414  0.58199424 -0.84664404  0.57027811 -0.20741715
  0.02784866]
[ 0.01255781 -0.02931368  0.1608634  -0.33185521  0.31180814 -0.16463067
  0.05555796]
[-0.0167121   0.03173966  0.00400858 -0.02572511 -0.02412852  0.08885808
 -0.04472235]
[-0.05655501  0.08309566 -0.08147315  0.02597015 -0.03567177  0.0657132
 -0.03488606]]
<NDArray 7x7 @cpu(0)>
```

在微调里，我们一般新建一个网络，它的定义跟之前训练好的网络一样，除了最后的输出数等于当前数据的类别数。新网络的 `features` 被初始化前面训练好网络的权重，而 `output` 则是从头开始训练。

In [7]: `from mxnet import init`

```
finetune_net = models.resnet18_v2(classes=2)
finetune_net.features = pretrained_net.features
finetune_net.output.initialize(init.Xavier())
```

我们先定义一个可以重复使用的训练函数。

In [8]:

```
def train(net, ctx, batch_size=64, epochs=10, learning_rate=0.01, wd=0.001):
    train_data = gluon.data.DataLoader(train_imgs, batch_size, shuffle=True)
    test_data = gluon.data.DataLoader(test_imgs, batch_size)

    # 确保 net 的初始化在 ctx 上
    net.collect_params().reset_ctx(ctx)
    net.hybridize()
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    # 训练
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {
        'learning_rate': learning_rate, 'wd': wd})
    utils.train(train_data, test_data, net, loss, trainer, ctx, epochs)
```

现在我们可以训练了。

In [9]: `ctx = utils.try_all_gpus()`
`train(finetune_net, ctx)`

```
Start training on [gpu(0), gpu(1)]
Epoch 0. Loss: 0.316, Train acc 0.87, Test acc 0.93, Time 26.1 sec
Epoch 1. Loss: 0.177, Train acc 0.93, Test acc 0.94, Time 23.2 sec
Epoch 2. Loss: 0.142, Train acc 0.94, Test acc 0.95, Time 22.8 sec
```

```
Epoch 3. Loss: 0.107, Train acc 0.96, Test acc 0.95, Time 22.6 sec
Epoch 4. Loss: 0.086, Train acc 0.97, Test acc 0.95, Time 23.1 sec
Epoch 5. Loss: 0.079, Train acc 0.97, Test acc 0.95, Time 23.0 sec
Epoch 6. Loss: 0.069, Train acc 0.98, Test acc 0.95, Time 22.7 sec
Epoch 7. Loss: 0.061, Train acc 0.98, Test acc 0.96, Time 22.9 sec
Epoch 8. Loss: 0.044, Train acc 0.99, Test acc 0.95, Time 23.2 sec
Epoch 9. Loss: 0.052, Train acc 0.98, Test acc 0.95, Time 22.6 sec
```

对比起见我们尝试从随机初始值开始训练一个网络。

```
In [10]: scratch_net = models.resnet18_v2(classes=2)
scratch_net.initialize(init=init.Xavier())
train(scratch_net, ctx)

Start training on [gpu(0), gpu(1)]
Epoch 0. Loss: 0.473, Train acc 0.78, Test acc 0.79, Time 23.3 sec
Epoch 1. Loss: 0.359, Train acc 0.85, Test acc 0.79, Time 22.8 sec
Epoch 2. Loss: 0.354, Train acc 0.85, Test acc 0.86, Time 23.3 sec
Epoch 3. Loss: 0.329, Train acc 0.86, Test acc 0.82, Time 23.3 sec
Epoch 4. Loss: 0.335, Train acc 0.85, Test acc 0.82, Time 22.9 sec
Epoch 5. Loss: 0.313, Train acc 0.87, Test acc 0.85, Time 23.1 sec
Epoch 6. Loss: 0.301, Train acc 0.87, Test acc 0.81, Time 23.3 sec
Epoch 7. Loss: 0.306, Train acc 0.86, Test acc 0.83, Time 23.0 sec
Epoch 8. Loss: 0.288, Train acc 0.87, Test acc 0.85, Time 23.5 sec
Epoch 9. Loss: 0.296, Train acc 0.87, Test acc 0.82, Time 22.8 sec
```

可以看到，微调版本收敛比从随机值开始的要快很多。

图片预测

```
In [11]: import matplotlib.pyplot as plt

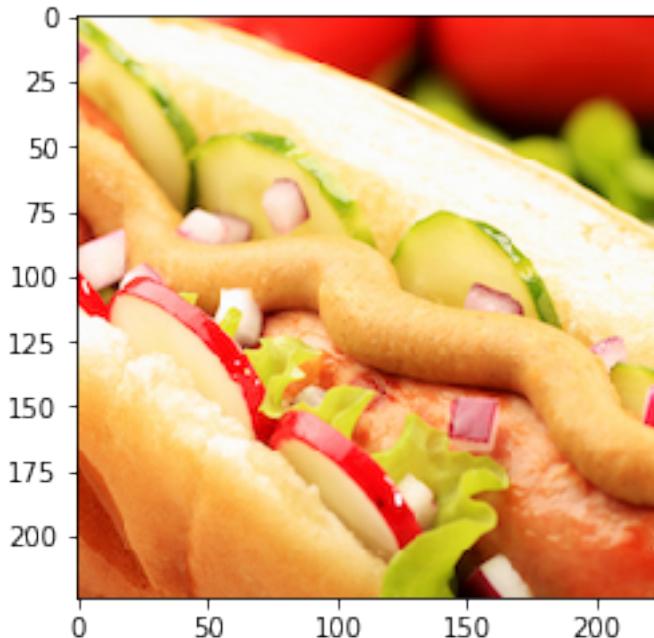
def classify_hotdog(net, fname):
    with open(fname, 'rb') as f:
        img = image.imread(f.read())
    data, _ = transform(img, -1, test_augs)
    plt.imshow(data.transpose((1, 2, 0)).asnumpy() / 255)
    data = data.expand_dims(axis=0)
    out = net(data.as_in_context(ctx[0]))
    out = nd.SoftmaxActivation(out)
    pred = int(nd.argmax(out, axis=1).asscalar())
    prob = out[0][pred].asscalar()
    label = train_imgs.synsets
```

```
return 'With prob=%f, %s'%(prob, label[pred])
```

接下来我们用训练好的模型来预测几张图片：

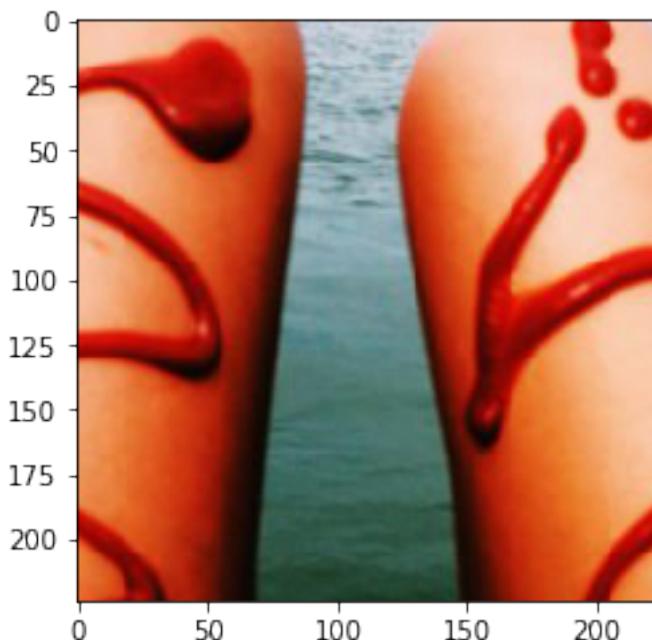
```
In [12]: classify_hotdog(finetune_net, '../img/real_hotdog.jpg')
```

```
Out[12]: 'With prob=0.994203, hotdog'
```



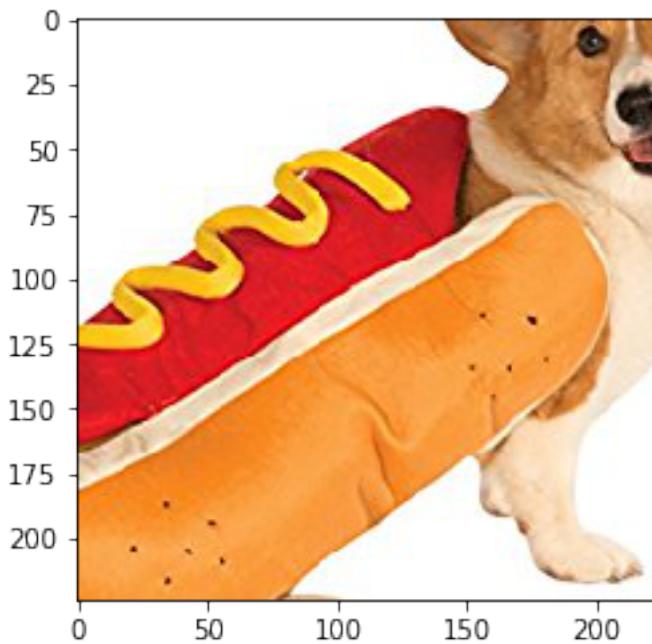
```
In [13]: classify_hotdog(finetune_net, '../img/leg_hotdog.jpg')
```

```
Out[13]: 'With prob=0.898811, hotdog'
```



```
In [14]: classify_hotdog(finetune_net, '../img/dog_hotdog.jpg')
```

```
Out[14]: 'With prob=0.826779, not-hotdog'
```



9.2.2 结论

我们看到，通过一个预先训练好的模型，我们可以在即使较小的数据集上训练得到很好的分类器。这是因为这两个任务里面的数据表示有很多共通性，例如都需要如何识别纹理、形状、边等等。而

这些通常被在靠近数据的层有效的处理。因此，如果你有一个相对较小的数据在手，而且担心它可能不够训练出很好的模型，你可以寻找跟你数据类似的大数据集来先训练你的模型，然后再在你手上的数据集上微调。

9.2.3 练习

- 多跑几个 epochs 直到收敛（你可以也需要调调参数），看看 scratch_net 和 fine-tune_net 最后的精度是不是有区别
- 这里 finetune_net 重用了 pretrained_net 除最后全连接外的所有权重，试试少重用些权重，有会有什么区别
- 事实上 ImageNet 里也有 hotdog 这个类，它的 index 是 713。例如它对应的 weight 可以这样拿到。试试如何重用这个权重

```
In [15]: weight = pretrained_net.output.weight  
hotdog_w = nd.split(weight.data(), 1000, axis=0)[713]  
hotdog_w.shape
```

Out[15]: (1, 512)

- 试试不让 finetune_net 里重用的权重参与训练，就是不更新权重
- 如果图片预测这一章里我们训练的模型没有分对所有的图片，如何改进？

吐槽和讨论欢迎点[这里](#)

9.3 使用卷积神经网络的物体检测

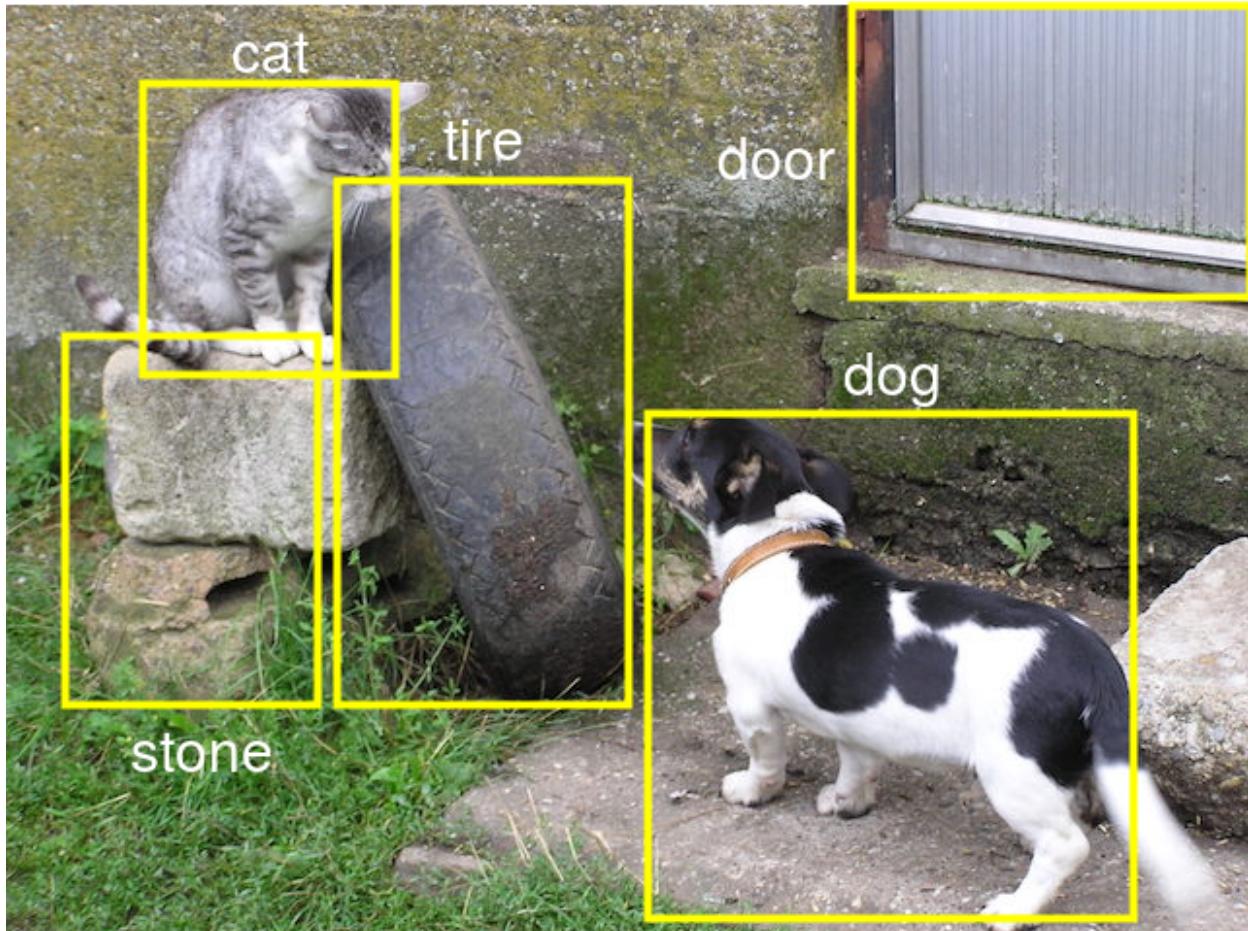
当我们讨论对图片进行预测时，到目前为止我们都是谈论分类。我们问过这个数字是 0 到 9 之间的哪一个，这个图片是鞋子还是衬衫，或者下面这张图片里面是猫还是狗。

但现实图片会更加复杂，它们可能不仅仅包含一个主体物体。物体检测就是针对这类问题提出，它不仅是要分析图片里有什么，而且需要识别它在什么位置。我们使用在[机器学习简介](#)那章讨论过的图片作为样例，并对它标上主要物体和位置。

可以看出物体检测跟图片分类有几个不同点：

1. 图片分类器通常只需要输出对图片中的主物体的分类。但物体检测必须能够识别多个物体，即使有些物体可能在图片中不是占主要版面。严格上来说，这个任务一般叫**多类物体检测**，但绝大部分研究都是针对多类的设置，所以我们这里为了简单去掉了“多类”





2. 图片分类器只需要输出将图片物体识别成某类的概率，但物体检测不仅需要输出识别概率，还需要识别物体在图片中的位置。这个通常是一个括住这个物体的方框，通常也称之为**边界框**（bounding box）。

但也看到物体检测跟图片分类有类似之处，都是对一块图片区域判断其包含的主要物体。因此可以想象我们在前面介绍的基于卷积神经网络的图片分类可以被应用到这里。

这一章我们将介绍数个基于卷积神经网络的物体检测算法的思想：

- R-CNN: <https://arxiv.org/abs/1311.2524>
- Fast R-CNN: <https://arxiv.org/abs/1504.08083>
- Faster R-CNN: <https://arxiv.org/abs/1506.01497>
- Mask R-CNN: <https://arxiv.org/abs/1703.06870>
- SSD: <https://arxiv.org/abs/1512.02325>
- YOLO: <https://arxiv.org/abs/1506.02640>
- YOLOv2: <https://arxiv.org/abs/1612.08242>

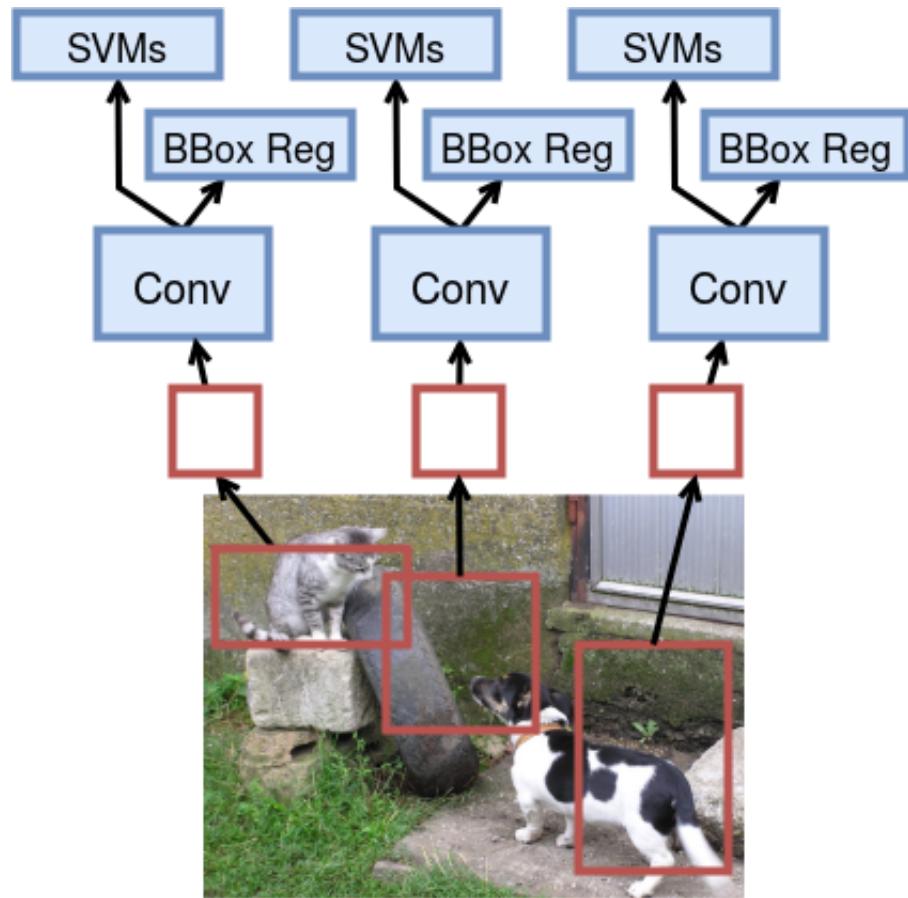
9.3.1 R-CNN：区域卷积神经网络

这是基于卷积神经网络的物体检测的奠基之作。其核心思想是在对每张图片选取多个区域，然后每个区域作为一个样本进入一个卷积神经网络来抽取特征，最后使用分类器来对齐分类，和一个回归器来得到准确的边框。

具体来说，这个算法有如下几个步骤：

1. 对每张输入图片使用一个基于规则的“选择性搜索”算法来选取多个提议区域
2. 跟微调迁移学习里那样，选取一个预先训练好的卷积神经网络并去掉最后一个输入层。每个区域被调整成这个网络要求的输入大小并计算输出。这个输出将作为这个区域的特征。
3. 使用这些区域特征来训练多个 SVM 来做物体识别，每个 SVM 预测一个区域是不是包含某个物体
4. 使用这些区域特征来训练线性回归器将提议区域

直观上 R-CNN 很好理解，但问题是它可能特别慢。一张图片我们可能选出上千个区域，导致一张图片需要做上千次预测。虽然跟微调不一样，这里训练可以不用更新用来抽特征的卷积神经网络，从而我们可以事先算好每个区域的特征并保存。但对于预测，我们无法避免这个。从而导致 R-CNN 很难实际中被使用。



Selective Search

Fig. 9.2: R-CNN

9.3.2 Fast R-CNN：快速的区域卷积神经网络

Fast R-CNN 对 R-CNN 主要做了两点改进来提升性能。

1. 考虑到 R-CNN 里面的大量区域可能是相互覆盖，每次重新抽取特征过于浪费。因此 Fast R-CNN 先对输入图片抽取特征，然后再选取区域
2. 代替 R-CNN 使用多个 SVM 来做分类，Fast R-CNN 使用单个多类逻辑回归，这也是前面教程里默认使用的。

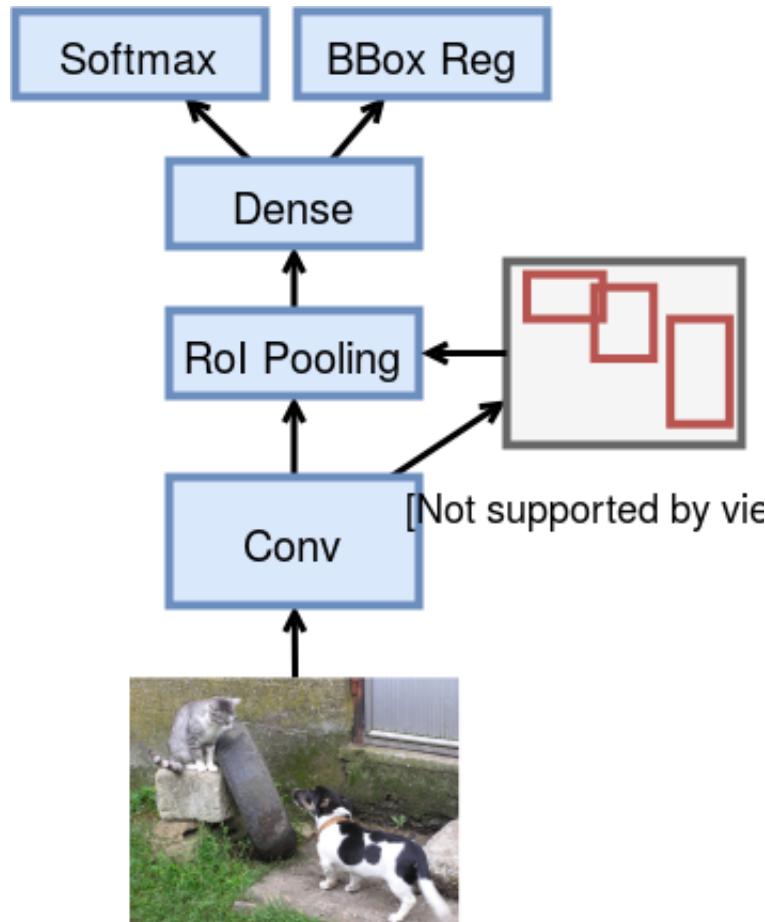


Fig. 9.3: Fast R-CNN

从示意图可以看到，使用选择性搜索选取的区域是作用在卷积神经网络提取的特征上。这样我们只需要对原始的输入图片做一次特征提取即可，如此节省了大量重复计算。

Fast R-CNN 提出兴趣区域池化层（Region of Interest (RoI) pooling），它的输入为特征和一系列的区域，对每个区域它将其均匀划分成 $n \times m$ 的小区域，并对每个小区域做最大池化，从而得到一个 $n \times m$ 的输出。因此不管输入区域的大小，RoI 池化层都将其池化成固定大小输出。

下面我们仔细看一下 RoI 池化层是如何工作的，假设对于一张图片我们提出了一个 4×4 的特征，

并且通道数为 1.

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

x = nd.arange(16).reshape((1,1,4,4))
x

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

然后我们创建两个区域，每个区域由一个长为 5 的向量表示。第一个元素是其对应的物体的标号，之后分别是 `x_min`, `y_min`, `x_max`, 和 `y_max`。这里我们生成了 3×3 和 4×3 大小的两个区域。

ROI 池化层的输出大小是 `num_regions` \times `num_channels` \times `n` \times `m`。它可以当做一个样本个数是 `num_regions` 的普通批量进入到其他层进行训练。

```
In [2]: rois = nd.array([[0,0,0,2,2], [0,0,1,3,3]])
nd.ROIPooling(x, rois, pooled_size=(2,2), spatial_scale=1)

Out[2]:
[[[[ 5.   6.]
   [ 9.  10.]]

   [[ 9.  11.]
    [ 13.  15.]]]]
<NDArray 2x1x2x2 @cpu(0)>
```

9.3.3 Faster R-CNN：更快速的区域卷积神经网络

Fast R-CNN 沿用了 R-CNN 的选择性搜索方法来选择区域。这个通常很慢。Faster R-CNN 做的主要改进是提出了**区域提议网络** (region proposal network, RPN) 来替代选择性搜索。它是这么工作的：

1. 在输入特征上放置一个 $1 \times 1 \times 256 \times 3 \times 3$ 卷积。这样每个像素，连同它的周围 8 个像素，都没映射成一个长为 256 的向量。
2. 以对每个像素为中心生成数个大小和长宽比预先设计好的 k 个默认边框，通常也叫**锚框**。
3. 对每个边框，使用其中心像素对应的 256 维向量作为特征，RPN 训练一个 2 类分类器来判断这个区域是不是含有任何感兴趣的物体还是只是背景，和一个 4 维输出的回归器来预测一

个更准确的边框。

- 对于所有的锚框，个数为 nmk 如果输入大小是 $n \times m$ ，选出被判断成还有物体的，然后前他们对应的回归器预测的边框作为输入放进接下来的 ROI 池化层

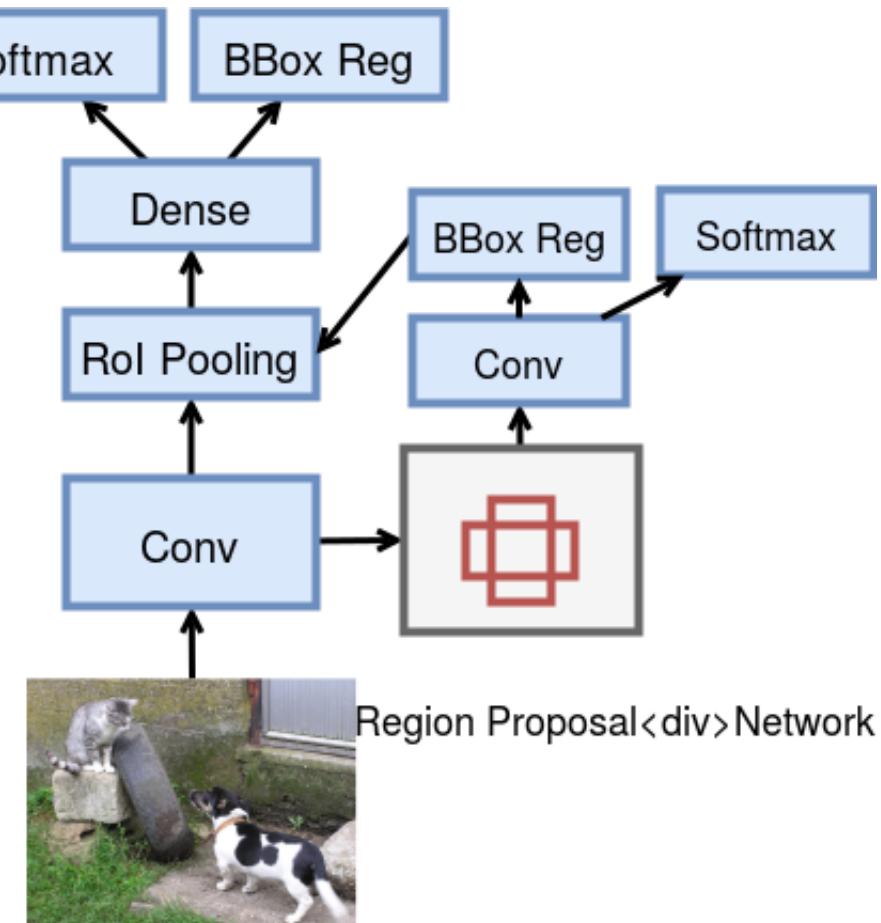


Fig. 9.4: Faster R-CNN

虽然看上有些复杂，但 RPN 思想非常直观。首先提议预先配置好的一些区域，然后通过神经网络来判断这些区域是不是感兴趣的，如果是，那么再预测一个更加准确的边框。这样我们能有效降低搜索任何形状的边框的代价。

9.3.4 Mask R-CNN

Mask R-CNN 在 Faster R-CNN 上加入了一个新的像素级别的预测层，它不仅对一个锚框预测它对应的类和真实的边框，而且它会判断这个锚框类每个像素对应的哪个物体还是只是背景。后者是语义分割要解决的问题。Mask R-CNN 使用了之后我们将介绍的全连接卷积网络 (FCN) 来完成这个预测。当然这也意味这训练数据必须有像素级别的标注，而不是简单的边框。

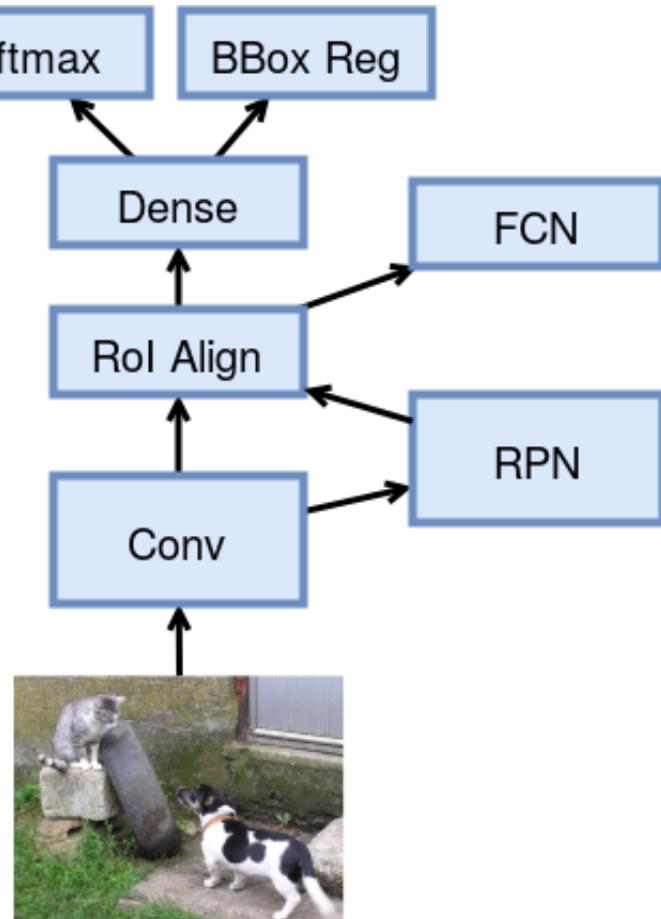


Fig. 9.5: Mask R-CNN

因为 FCN 会精确预测每个像素的类别，就是输入图片中的每个像素都会在标注中对应一个类别。对于输入图片中的一个锚框，我们可以精确的匹配到像素标注中对应的区域。但是 PoI 池化是作用在卷积之后的特征上，其默认是将锚框做了定点化。例如假设选择的锚框是 (x, y, w, h) ，且特征抽取将图片变小了 16 倍，就是如果原始图片是 256×256 ，那么特征大小就是 16×16 。这时候在特征上对应的锚框就是变成了 $(\lfloor x/16 \rfloor, \lfloor y/16 \rfloor, \lfloor w/16 \rfloor, \lfloor h/16 \rfloor)$ 。如果 x, y, w, h 中有任何一个不被 16 整除，那么就可能发生错位。同样道理，在上面的样例中我们看到，如果锚框的长宽不被池化大小整除，那么同样会定点化，从而带来错位。

通常这样的错位只是在几个像素之间，对于分类和边框预测影响不大。但对于像素级别的预测，这样的错位可能会带来大问题。Mask R-CNN 提出一个 RoI Align 层，它类似于 RoI 池化层，但是去除了定点化步骤，就是移除了所有 $\lfloor \cdot \rfloor$ 。如果计算得到的表框不是刚好在像素之间，那么我们就用四周的像素来线性插值得到这个点上的值。

对于一维情况，假设我们要计算 x 点的值 $f(x)$ ，那么我们可以用 x 左右的整点的值来插值：

$$f(x) = (\lfloor x \rfloor + 1 - x)f(\lfloor x \rfloor) + (x - \lfloor x \rfloor)f(\lfloor x \rfloor + 1)$$

我们实际要使用的是二维差值来估计 $f(x, y)$ ，我们首先在 x 轴上差值得到 $f(x, \lfloor y \rfloor)$ 和 $f(x, \lfloor y \rfloor + 1)$ ，然后根据这两个值来差值得到 $f(x, y)$ 。

9.3.5 SSD: 单发多框检测器

在 R-CNN 系列模型里。区域提议和分类是分作两块来进行的。SSD 则将其统一成一个步骤来使得模型更加简单并且速度更快，这也是为什么它被称之为单发的原因。

它跟 Faster R-CNN 主要有两点不一样

1. 对于锚框，我们不再首先判断它是不是含有感兴趣物体，再将正类锚框放入真正物体分类。SSD 里我们直接使用一个 `num_class+1` 类分类器来判断它对应的是哪类物体，还是只是背景。我们也不再有额外的回归器对边框再进一步预测，而是直接使用单个回归器来预测真实边框。
2. SSD 不只是对卷积神经网络输出的特征做预测，它会进一步将特征通过卷积和池化层变小来做预测。这样达到多尺度预测的效果。

SSD 的具体实现将在下一章详细阐述。

9.3.6 YOLO：只需要看一遍

不管是 Faster R-CNN 还是 SSD，它们生成的锚框仍然有大量是相互重叠的，从而导致仍然有大量的区域被重复计算了。YOLO 试图来解决这个问题。它将图片特征均匀的切成 $S \times S$ 块，每一块当做一个锚框。每个锚框预测 B 个边框，以及这个锚框主要包含哪个物体。

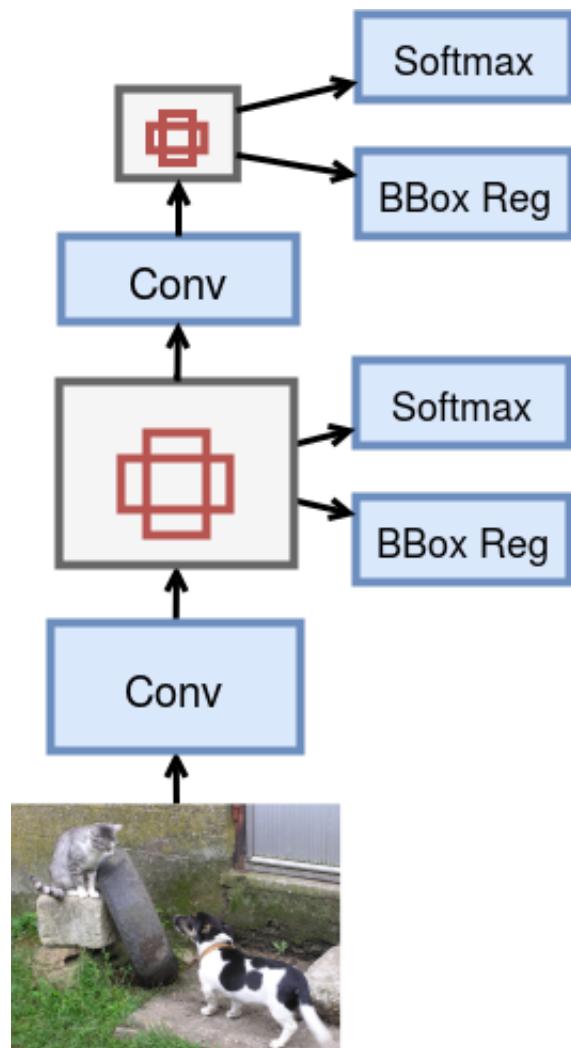


Fig. 9.6: SSD

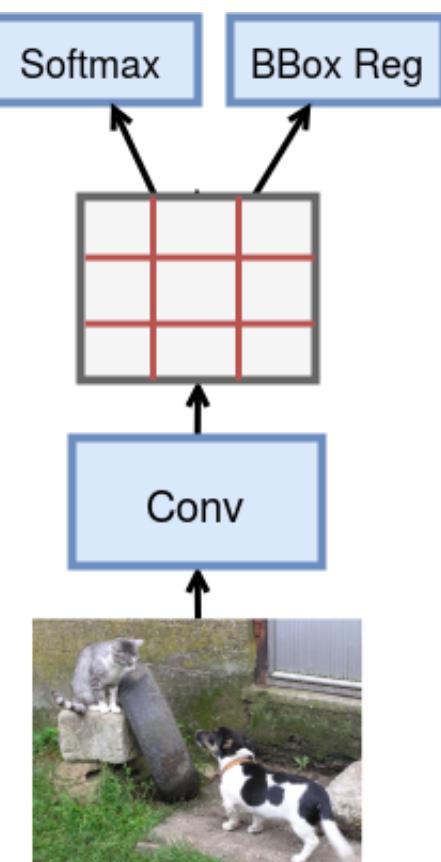


Fig. 9.7: Yolo

9.3.7 YOLO v2：更好，更块，更强

YOLO v2 对 YOLO 进行一些地方的改进，其主要包括：

1. 使用更好的卷积神经网络来做特征提取，使用更大输入图片 448×448 使得特征输出大小增大到 13×13
2. 不再使用均匀切来的锚框，而是对训练数据里的真实锚框做聚类，然后使用聚类中心作为锚框。相对于 SSD 和 Faster R-CNN 来说可以大幅降低锚框的个数。
3. 不再使用 YOLO 的全连接层来预测，而是同 SSD 一样使用卷积。例如假设使用 5 个锚框（聚类为 5 类），那么物体分类使用通道数是 $5*(1+num_classes)$ 的 1×1 卷积，边框回归使用通道数 $4*5$.

9.3.8 总结

我们描述了基于卷积神经网络的几个物体检测算法。他们之间的共同点在于首先提出锚框，使用卷积神经网络抽取特征后来预测其包含的主要物体和更准确的边框。但他们在锚框的选择和预测上各有不同，导致他们在计算实际和精度上也各有权衡。

吐槽和讨论欢迎点[这里](#)

9.4 SSD — 使用 Gluon

这一章下面我们将实现上一章介绍的 SSD 来检测野生皮卡丘。

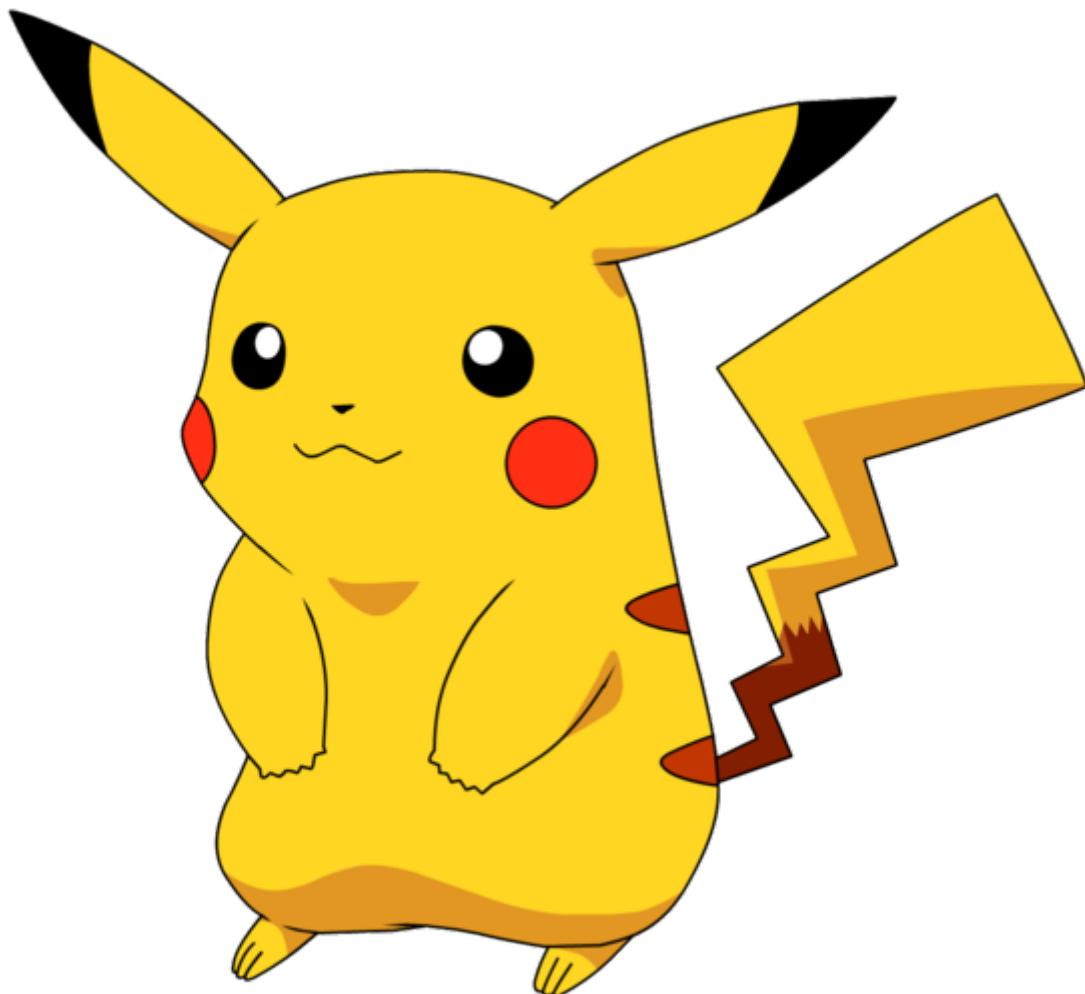
9.4.1 数据集

为此我们合成了一个人工数据集。我们首先使用一个开源的皮卡丘 3D 模型，用其生成 1000 张不同角度和大小的照片。然后将其随机的放在背景图片里。我们将图片打包成 `rec` 文件，这是一个 MXNet 常用的二进制数据格式。我们可以使用 MXNet 下的[tools/im2rec.py](#) 来将图片打包。（TODO(@mli) 加一个教程关于如何使用 im2rec）。

下载数据

打包好的数据可以直接在网上下载：

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



```
root_url = ('https://apache-mxnet.s3-accelerate.amazonaws.com/'  
           'gluon/dataset/pikachu/')  
data_dir = '../data/pikachu/'  
dataset = {'train.rec': '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)  
  
Downloading ../data/pikachu/train.rec from https://apache-mxnet.s3-accelerate.amazonaws.com/  
Downloading ../data/pikachu/train.idx from https://apache-mxnet.s3-accelerate.amazonaws.com/  
Downloading ../data/pikachu/val.rec from https://apache-mxnet.s3-accelerate.amazonaws.com/
```

读取数据集

我们使用 `image.ImageDetIter` 来读取数据。这是针对物体检测的迭代器, (`Det` 表示 `Detection`)。它跟 `image.ImageIter` 使用很类似。主要不同是它返回的标号不是单个图片标号, 而是每个图片里所有物体的标号, 以及其对用的边框。

```
In [2]: from mxnet import image  
        from mxnet import nd  
  
        data_shape = 256  
        batch_size = 32  
        rgb_mean = nd.array([123, 117, 104])  
  
        def get_iterators(data_shape, batch_size):  
            class_names = ['pikachu']  
            num_class = len(class_names)  
            train_iter = image.ImageDetIter(  
                batch_size=batch_size,  
                data_shape=(3, data_shape, data_shape),  
                path_imgrec=data_dir+'train.rec',  
                path_imgidx=data_dir+'train.idx',  
                shuffle=True,  
                mean=True,  
                rand_crop=1,  
                min_object_covered=0.95,  
                max_attempts=200)  
            val_iter = image.ImageIter(  
                batch_size=batch_size,  
                data_shape=(3, data_shape, data_shape),
```

```

    path_imgrec=data_dir+'val.rec',
    shuffle=False,
    mean=True)
return train_iter, val_iter, class_names, num_class

train_data, test_data, class_names, num_class = get_iterators(
    data_shape, batch_size)

```

我们读取一个批量。可以看到标号的形状是 `batch_size x num_object_per_image x 5`。这里数据里每个图片里面只有一个标号。每个标号由长为 5 的数组表示，第一个元素是其对用物体的标号，其中 -1 表示非法物体，仅做填充使用。后面 4 个元素表示边框。

```
In [3]: batch = train_data.next()
print(batch)

DataBatch: data shapes: [(32, 3, 256, 256)] label shapes: [(32, 1, 5)]
```

图示数据

我们画出几张图片和其对应的标号。可以看到比卡丘的角度大小位置在每张图图片都不一样。不过也注意到这个数据集是直接将二次元动漫皮卡丘跟三次元背景相结合。可能通过简单判断区域的色彩直方图就可以有效的区别是不是有我们要的物体。我们用这个简单数据集来演示 SSD 是如何工作的。实际中遇到的数据集通常会复杂很多。

```

In [4]: %matplotlib inline
import matplotlib as mpl
mpl.rcParams['figure.dpi']= 120
import matplotlib.pyplot as plt

def box_to_rect(box, color, linewidth=3):
    """convert an anchor box to a matplotlib rectangle"""
    box = box.asnumpy()
    return plt.Rectangle(
        (box[0], box[1]), box[2]-box[0], box[3]-box[1],
        fill=False, edgecolor=color, linewidth=linewidth)

_, figs = plt.subplots(3, 3, figsize=(6,6))
for i in range(3):
    for j in range(3):
        img, labels = batch.data[0][3*i+j], batch.label[0][3*i+j]
        # (3L, 256L, 256L) => (256L, 256L, 3L)
        img = img.transpose((1, 2, 0)) + rgb_mean

```

```
img = img.clip(0,255).asnumpy()/255
fig = figs[i][j]
fig.imshow(img)
for label in labels:
    rect = box_to_rect(label[1:5]*data_shape, 'red', 2)
    fig.add_patch(rect)
fig.axes.get_xaxis().set_visible(False)
fig.axes.get_yaxis().set_visible(False)
plt.show()
```



9.4.2 SSD 模型

锚框：默认的边界框

因为边框可以出现在图片中的任何位置，并且可以有任意大小。为了简化计算，SSD 跟 Faster R-CNN 一样使用一些默认的边界框，或者称之为锚框（anchor box），做为搜索起点。具体来说，对输入的每个像素，以其为中心采样数个有不同形状和不同比例的边界框。假设输入大小是 $w \times h$,

- 给定大小 $s \in (0, 1]$, 那么生成的边界框形状是 $ws \times hs$
- 给定比例 $r > 0$, 那么生成的边界框形状是 $w\sqrt{r} \times \frac{h}{\sqrt{r}}$

在采样的时候我们提供 n 个大小 (sizes) 和 m 个比例 (ratios)。为了计算简单这里不生成 nm 个锚框, 而是 $n + m - 1$ 个。其中第 i 个锚框使用

- sizes[i] 和 ratios[0] 如果 $i \leq n$
- sizes[0] 和 ratios[i-n] 如果 $i > n$

我们可以使用 contrib.ndarray 里的 MultiBoxPrior 来采样锚框。这里锚框通过左下角和右上角两个点来确定, 而且被标准化成了区间 $[0, 1]$ 的实数。

```
In [5]: from mxnet import nd
        from mxnet.contrib.ndarray import MultiBoxPrior

        # shape: batch x channel x height x weight
        n = 40
        x = nd.random.uniform(shape=(1, 3, n, n))

        y = MultiBoxPrior(x, sizes=[.5,.25,.1], ratios=[1,2,.5])

        boxes = y.reshape((n, n, -1, 4))
        print(boxes.shape)
        # The first anchor box centered on (20, 20)
        # its format is (x_min, y_min, x_max, y_max)
        boxes[20, 20, 0, :]

(40, 40, 5, 4)
```

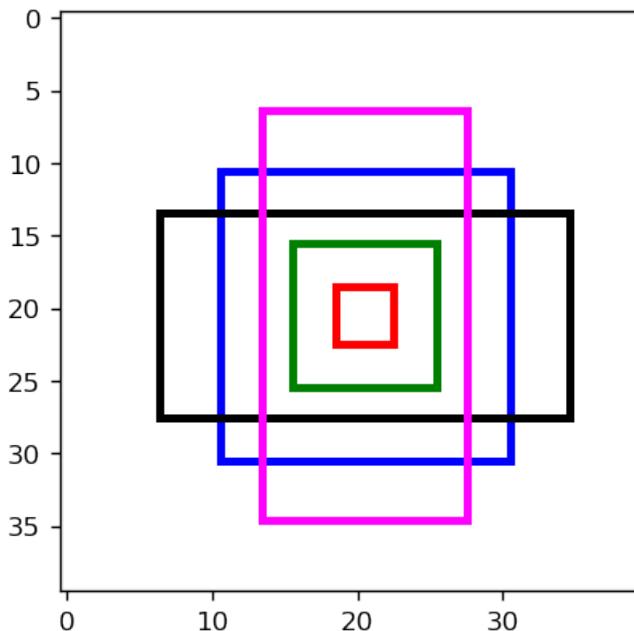
Out[5]:

```
[ 0.26249999  0.26249999  0.76249999  0.76249999]
<NDArray 4 @cpu(0)>
```

我们可以画出以 $(20, 20)$ 为中心的所有锚框:

```
In [6]: colors = ['blue', 'green', 'red', 'black', 'magenta']

        plt.imshow(nd.ones((n, n, 3)).asnumpy())
        anchors = boxes[20, 20, :, :]
        for i in range(anchors.shape[0]):
            plt.gca().add_patch(box_to_rect(anchors[i,:]*n, colors[i]))
        plt.show()
```



预测物体类别

对每一个锚框我们需要预测它是不是包含了我们感兴趣的物体，还是只是背景。这里我们使用一个 3×3 的卷积层来做预测，加上 `pad=1` 使用它的输出和输入一样。同时输出的通道数是 `num_anchors*(num_classes+1)`，每个通道对应一个锚框对某个类的置信度。假设输出是 `Y`，那么对应输入中第 n 个样本的第 (i, j) 像素的置信值是在 `Y[n, :, i, j]` 里。具体来说，对于以 (i, j) 为中心的第 a 个锚框，

- 通道 $a * (\text{num_class} + 1)$ 是其只包含背景的分数
- 通道 $a * (\text{num_class} + 1) + 1 + b$ 是其包含第 b 个物体的分数

我们定义一个这样的类别分类器函数：

```
In [7]: from mxnet.gluon import nn
def class_predictor(num_anchors, num_classes):
    """return a layer to predict classes"""
    return nn.Conv2D(num_anchors * (num_classes + 1), 3, padding=1)

cls_pred = class_predictor(5, 10)
cls_pred.initialize()
x = nd.zeros((2, 3, 20, 20))
y = cls_pred(x)
y.shape
```

Out[7]: (2, 55, 20, 20)

预测边界框

因为真实的边界框可以是任意形状，我们需要预测如何从一个锚框变换为真正的边界框。这个变换可以由一个长为 4 的向量来描述。同上一样，我们用一个有 `num_anchors * 4` 通道的卷积。假设输出是 Y，那么对应输入中第 n 个样本的第 (i, j) 像素为中心的锚框的转换在 `Y[n, :, i, j]` 里。具体来说，对于第 a 个锚框，它的变换在 $a*4$ 到 $a*4+3$ 通道里。

```
In [8]: def box_predictor(num_anchors):
    """return a layer to predict delta locations"""
    return nn.Conv2D(num_anchors * 4, 3, padding=1)

    box_pred = box_predictor(10)
    box_pred.initialize()
    x = nd.zeros((2, 3, 20, 20))
    y = box_pred(x)
    y.shape

Out[8]: (2, 40, 20, 20)
```

减半模块

我们定义一个卷积块，它将输入特征的长宽减半，以此来获取多尺度的预测。它由两个 Conv-BatchNorm-Relu 组成，我们使用填充为 1 的 3×3 卷积使得输入和输入有同样的长宽，然后再通过跨度为 2 的最大池化层将长宽减半。

```
In [9]: def down_sample(num_filters):
    """stack two Conv-BatchNorm-Relu blocks and then a pooling layer
    to halve the feature size"""
    out = nn.HybridSequential()
    for _ in range(2):
        out.add(nn.Conv2D(num_filters, 3, strides=1, padding=1))
        out.add(nn.BatchNorm(in_channels=num_filters))
        out.add(nn.Activation('relu'))
    out.add(nn.MaxPool2D(2))
    return out

    blk = down_sample(10)
    blk.initialize()
    x = nd.zeros((2, 3, 20, 20))
    y = blk(x)
    y.shape
```

```
Out[9]: (2, 10, 10, 10)
```

合并来自不同层的预测输出

前面我们提到过 SSD 的一个重要性质是它会在多个层同时做预测。每个层由于长宽和锚框选择不一样，导致输出的数据形状会不一样。这里我们用物体类别预测作为样例，边框预测是类似的。

我们首先创建一个特定大小的输入，然后对它输出类别预测。然后对输入减半，再输出类别预测。

```
In [10]: x = nd.zeros((2, 8, 20, 20))
          print('x:', x.shape)

          cls_pred1 = class_predictor(5, 10)
          cls_pred1.initialize()
          y1 = cls_pred1(x)
          print('Class prediction 1:', y1.shape)

          ds = down_sample(16)
          ds.initialize()
          x = ds(x)
          print('x:', x.shape)

          cls_pred2 = class_predictor(3, 10)
          cls_pred2.initialize()
          y2 = cls_pred2(x)
          print('Class prediction 2:', y2.shape)

x: (2, 8, 20, 20)
Class prediction 1: (2, 55, 20, 20)
x: (2, 16, 10, 10)
Class prediction 2: (2, 33, 10, 10)
```

可以看到 `y1` 和 `y2` 形状不同。为了之后处理简单，我们将不同层的输入合并成一个输出。首先我们将通道移到最后的维度，然后将其展成 2D 数组。因为第一个维度是样本个数，所以不同输出之间是不变，我们可以将所有输出在第二个维度上拼接起来。

```
In [11]: def flatten_prediction(pred):
          return pred.transpose(axes=(0,2,3,1)).flatten()

          def concat_predictions(preds):
              return nd.concat(*preds, dim=1)

flat_y1 = flatten_prediction(y1)
```

```

print('Flatten class prediction 1', flat_y1.shape)
flat_y2 = flatten_prediction(y2)
print('Flatten class prediction 2', flat_y2.shape)
y = concat_predictions([flat_y1, flat_y2])
print('Concat class predictions', y.shape)

Flatten class prediction 1 (2, 22000)
Flatten class prediction 2 (2, 3300)
Concat class predictions (2, 25300)

```

主体网络

主体网络用来从原始像素抽取特征。通常前面介绍的用来图片分类的卷积神经网络，例如 ResNet，都可以用来作为主体网络。这里为了示范，我们简单叠加几个减半模块作为主体网络。

```

In [12]: def body():
    out = nn.HybridSequential()
    for nfilters in [16, 32, 64]:
        out.add(down_sample(nfilters))
    return out

bnet = body()
bnet.initialize()
x = nd.random.uniform(shape=(2, 3, 256, 256))
y = bnet(x)
y.shape

Out[12]: (2, 64, 32, 32)

```

创建一个玩具 SSD 模型

现在我们可以创建一个玩具 SSD 模型了。我们称之为玩具是因为这个网络不管是层数还是锚框个数都比较小，仅仅适合之后我们之后使用的一个小数据集。但这个模型不会影响我们介绍 SSD。

这个网络包含四块。主体网络，三个减半模块，以及五个物体类别和边框预测模块。其中预测分别应用在在主体网络输出，减半模块输出，和最后的全局池化层上。

```

In [13]: def toy_ssd_model(num_anchors, num_classes):
    downsamplers = nn.Sequential()
    for _ in range(3):
        downsamplers.add(down_sample(128))

    class_predictors = nn.Sequential()

```

```
box_predictors = nn.Sequential()
for _ in range(5):
    class_predictors.add(class_predictor(num_anchors, num_classes))
    box_predictors.add(box_predictor(num_anchors))

model = nn.Sequential()
model.add(body(), downsamplers, class_predictors, box_predictors)
return model
```

计算预测

给定模型和每层预测输出使用的锚框大小和形状，我们可以定义前向函数。

```
In [14]: def toy_ssd_forward(x, model, sizes, ratios, verbose=False):
    body, downsamplers, class_predictors, box_predictors = model
    anchors, class_preds, box_preds = [], [], []
    # feature extraction
    x = body(x)
    for i in range(5):
        # predict
        anchors.append(MultiBoxPrior(
            x, sizes=sizes[i], ratios=ratios[i]))
        class_preds.append(
            flatten_prediction(class_predictors[i](x)))
        box_preds.append(
            flatten_prediction(box_predictors[i](x)))
    if verbose:
        print('Predict scale', i, x.shape, 'with',
              anchors[-1].shape[1], 'anchors')
    # down sample
    if i < 3:
        x = downsamplers[i](x)
    elif i == 3:
        x = nd.Pooling(
            x, global_pool=True, pool_type='max',
            kernel=(x.shape[2], x.shape[3]))
    # concat date
    return (concat_predictions(anchors),
            concat_predictions(class_preds),
            concat_predictions(box_preds))
```

完整的模型

```
In [15]: from mxnet import gluon
class ToySSD(gluon.Block):
    def __init__(self, num_classes, verbose=False, **kwargs):
        super(ToySSD, self).__init__(**kwargs)
        # anchor box sizes and ratios for 5 feature scales
        self.sizes = [[.2,.272], [.37,.447], [.54,.619],
                      [.71,.79], [.88,.961]]
        self.ratios = [[1,2,.5]]*5
        self.num_classes = num_classes
        self.verbose = verbose
        num_anchors = len(self.sizes[0]) + len(self.ratios[0]) - 1
        # use name_scope to guard the names
        with self.name_scope():
            self.model = toy_ssd_model(num_anchors, num_classes)

    def forward(self, x):
        anchors, class_preds, box_preds = toy_ssd_forward(
            x, self.model, self.sizes, self.ratios,
            verbose=self.verbose)
        # it is better to have class predictions reshaped for softmax computation
        class_preds = class_preds.reshape(shape=(0, -1, self.num_classes+1))
        return anchors, class_preds, box_preds
```

我们看看一下输入图片的形状是如何改变的，已经输出的形状。

```
In [16]: net = ToySSD(num_classes=2, verbose=True)
net.initialize()
x = batch.data[0][0:1]
print('Input:', x.shape)
anchors, class_preds, box_preds = net(x)
print('Output achors:', anchors.shape)
print('Output class predictions:', class_preds.shape)
print('Output box predictions:', box_preds.shape)
```

```
Input: (1, 3, 256, 256)
Predict scale 0 (1, 64, 32, 32) with 4096 anchors
Predict scale 1 (1, 128, 16, 16) with 1024 anchors
Predict scale 2 (1, 128, 8, 8) with 256 anchors
Predict scale 3 (1, 128, 4, 4) with 64 anchors
Predict scale 4 (1, 128, 1, 1) with 4 anchors
Output achors: (1, 5444, 4)
```

```
Output class predictions: (1, 5444, 3)
Output box predictions: (1, 21776)
```

9.4.3 训练

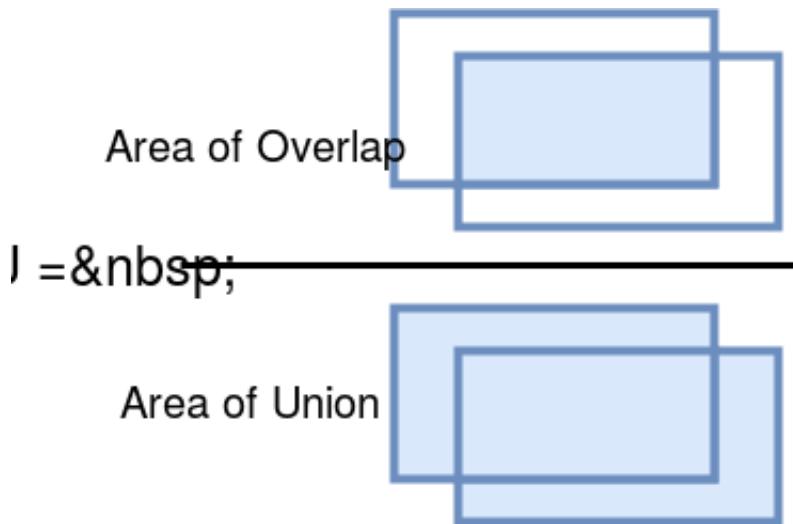
之前的教程我们主要是关注分类。对于分类的预测结果和真实的标号，我们通过交叉熵来计算他们的差异。但物体检测里我们需要预测边框。这里我们先引入一个概率来描述两个边框的距离。

IoU：交集除并集

我们知道判断两个集合的相似度最常用的衡量叫做 Jaccard 距离，给定集合 A 和 B ，它的定义是

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

边框可以看成是像素的集合，我们可以类似的定义它。这个标准通常被称之为 Intersection over Union (IoU)。



大的值表示两个边框很相似，而小的值则表示不相似。

损失函数

虽然每张图片里面通常只有几个标注的边框，但 SSD 会生成大量的锚框。可以想象很多锚框都不会框住感兴趣的物体，就是说跟任何对应感兴趣物体的表框的 IoU 都小于某个阈值。这样就会产生大量的负类锚框，或者说对应标号为 0 的锚框。对于这类锚框有两点要考虑的：

1. 边框预测的损失函数不应该包括负类锚框，因为它们并没有对应的真实边框

2. 因为负类锚框数目可能远多于其他，我们可以只保留其中的一些。而且是保留那些目前预测最不确信它是负类的，就是对类 0 预测值排序，选取数值最小的哪一些困难的负类锚框。

我们可以使用 `MultiBoxTarget` 来完成上面这两个操作。

```
In [17]: from mxnet.contrib.ndarray import MultiBoxTarget
def training_targets(anchors, class_preds, labels):
    class_preds = class_preds.transpose(axes=(0,2,1))
    return MultiBoxTarget(anchors, labels, class_preds)

out = training_targets(anchors, class_preds, batch.label[0][0:1])
out
```

Out[17]: [
[[0. 0. 0. ..., 0. 0. 0.]]
<NDArray 1x21776 @cpu(0)>,
[[0. 0. 0. ..., 0. 0. 0.]]
<NDArray 1x21776 @cpu(0)>,
[[0. 0. 0. ..., 0. 0. 0.]]
<NDArray 1x5444 @cpu(0)>]

它返回三个 `NDArray`, 分别是

1. 预测的边框跟真实边框的偏移, 大小是 `batch_size x (num_anchors*4)`
2. 用来遮掩不需要的负类锚框的掩码, 大小跟上面一致
3. 锚框的真实的标号, 大小是 `batch_size x num_anchors`

我们可以计算这次只选中了多少个锚框进入损失函数:

```
In [18]: out[1].sum()/4
```

Out[18]:

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

然后我们可以定义需要的损失函数了。

对于分类问题, 最常用的损失函数是之前一直使用的交叉熵。这里我们定义一个类似于交叉熵的损失, 不同于交叉熵的定义 $\log(p_j)$, 这里 j 是真实的类别, 且 p_j 是对于的预测概率。我们使用一个被称之为关注损失的函数, 给定正的 γ 和 α , 它的定义是

$$-\alpha(1 - p_j)^\gamma \log(p_j)$$

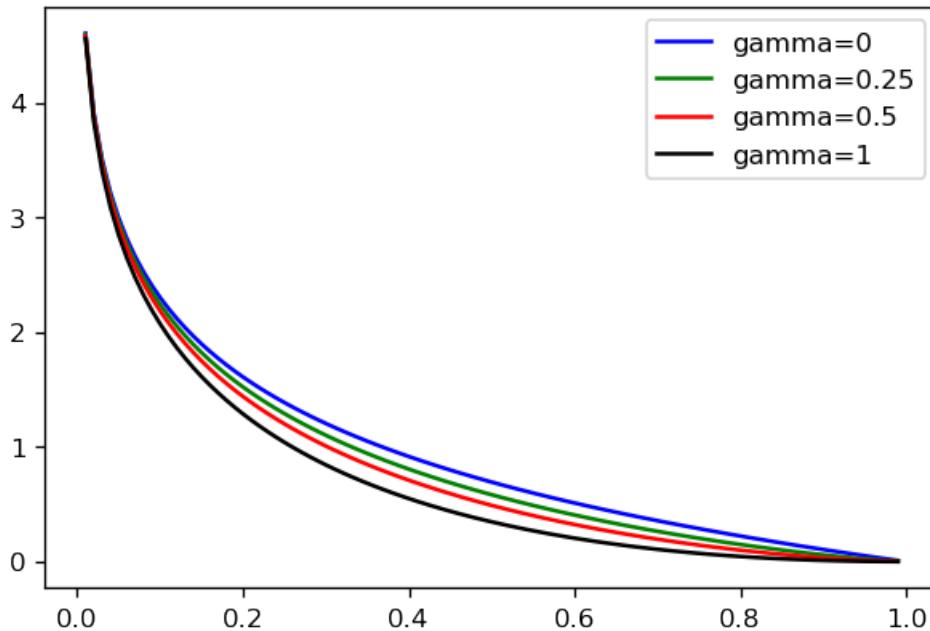
下图我们演示不同 γ 导致的变化。可以看到, 增加 γ 可以使得对正类预测值比较大时损失变小。

In [19]: `import numpy as np`

```
def focal_loss(gamma, x):
    return - (1-x)**gamma*np.log(x)

x = np.arange(0.01, 1, .01)
gammas = [0,.25,.5,1]
for i,g in enumerate(gammas):
    plt.plot(x, focal_loss(g,x), colors[i])

plt.legend(['gamma=' + str(g) for g in gammas])
plt.show()
```



这个自定义的损失函数可以简单通过继承 `gluon.loss.Loss` 来实现。

In [20]: `class FocalLoss(gluon.loss.Loss):`

```
def __init__(self, axis=-1, alpha=0.25, gamma=2, batch_axis=0, **kwargs):
    super(FocalLoss, self).__init__(None, batch_axis, **kwargs)
    self._axis = axis
    self._alpha = alpha
    self._gamma = gamma

def hybrid_forward(self, F, output, label):
    output = F.softmax(output)
    pj = output.pick(label, axis=self._axis, keepdims=True)
    loss = - self._alpha * ((1 - pj) ** self._gamma) * pj.log()
```

```

    return loss.mean(axis=self._batch_axis, exclude=True)

cls_loss = FocalLoss()
cls_loss

Out[20]: FocalLoss(batch_axis=0, w=None)

```

对于边框的预测是一个回归问题。通常可以选择平方损失函数（L2 损失） $f(x) = x^2$ 。但这个损失对于比较大的误差的惩罚很高。我们可以采用稍微缓和一点绝对损失函数（L1 损失） $f(x) = |x|$ ，它是随着误差线性增长，而不是平方增长。但这个函数在 0 点处导数不唯一，因此可能会影响收敛。一个通常的解决办法是在 0 点附近使用平方函数使得它更加平滑。它被称之为平滑 L1 损失函数。它通过一个参数 σ 来控制平滑的区域：

$$f(x) = \begin{cases} (\sigma x)^2/2, & \text{if } x < 1/\sigma^2 \\ |x| - 0.5/\sigma^2, & \text{otherwise} \end{cases}$$

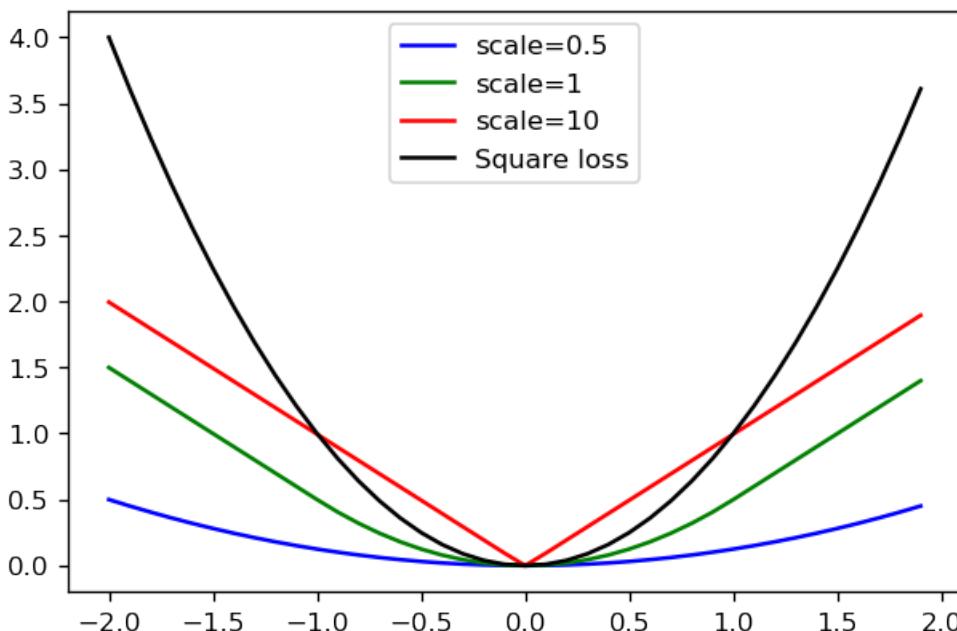
我们图示不同的 σ 的平滑 L1 损失和 L2 损失的区别。

```

In [21]: scales = [.5, 1, 10]
          x = nd.arange(-2, 2, 0.1)

          for i,s in enumerate(scales):
              y = nd.smooth_l1(x, scalar=s)
              plt.plot(x.asnumpy(), y.asnumpy(), color=colors[i])
          plt.plot(x.asnumpy(), (x**2).asnumpy(), color=colors[len(scales)])
          plt.legend(['scale=' + str(s) for s in scales] + ['Square loss'])
          plt.show()

```



我们同样通过继承 `Loss` 来定义这个损失。同时它接受一个额外参数 `mask`, 这是用来屏蔽掉不需要被惩罚的负例样本。

```
In [22]: class SmoothL1Loss(gluon.loss.Loss):
    def __init__(self, batch_axis=0, **kwargs):
        super(SmoothL1Loss, self).__init__(None, batch_axis, **kwargs)

    def hybrid_forward(self, F, output, label, mask):
        loss = F.smooth_l1((output - label) * mask, scalar=1.0)
        return loss.mean(self._batch_axis, exclude=True)

box_loss = SmoothL1Loss()
box_loss

Out[22]: SmoothL1Loss(batch_axis=0, w=None)
```

评估测量

对于分类好坏我们可以沿用之前的分类精度。评估边框预测的好坏的一个常用是是平均绝对误差。记得[在线性回归](#)我们使用了平均平方误差。但跟上面对损失函数的讨论一样，平方误差对于大的误差给予过大的值，从而数值上过于敏感。平均绝对误差就是将二次项替换成绝对值，具体来说就是预测的边框和真实边框在 4 个维度上的差值的绝对值。

```
In [23]: from mxnet import metric

cls_metric = metric.Accuracy()
box_metric = metric.MAE()
```

初始化模型和训练器

```
In [24]: from mxnet import init
from mxnet import gpu

ctx = gpu(0)
# the CUDA implementation requires each image has at least 3 labels.
# Pad two -1 labels for each instance
train_data.reshape(label_shape=(3, 5))
train_data = test_data.sync_label_shape(train_data)

net = ToySSD(num_class)
net.initialize(init.Xavier(magnitude=2), ctx=ctx)
```

```
trainer = gluon.Trainer(net.collect_params(),
                        'sgd', {'learning_rate': 0.1, 'wd': 5e-4})
```

训练模型

训练函数跟前面的不一样在于网络会有多个输出，而且有两个损失函数。

```
In [25]: import time
        from mxnet import autograd
        for epoch in range(30):
            # reset data iterators and metrics
            train_data.reset()
            cls_metric.reset()
            box_metric.reset()
            tic = time.time()
            for i, batch in enumerate(train_data):
                x = batch.data[0].as_in_context(ctx)
                y = batch.label[0].as_in_context(ctx)
                with autograd.record():
                    anchors, class_preds, box_preds = net(x)
                    box_target, box_mask, cls_target = training_targets(
                        anchors, class_preds, y)
                    # losses
                    loss1 = cls_loss(class_preds, cls_target)
                    loss2 = box_loss(box_preds, box_target, box_mask)
                    loss = loss1 + loss2
                loss.backward()
                trainer.step(batch_size)
                # update metrics
                cls_metric.update([cls_target], [class_preds.transpose((0,2,1))])
                box_metric.update([box_target], [box_preds * box_mask])

            print('Epoch %2d, train %s %.2f, %s %.5f, time %.1f sec' %
                  (epoch, *cls_metric.get(), *box_metric.get(), time.time()-tic
                  ))
```

Epoch 0, train accuracy 0.94, mae 0.00504, time 18.5 sec
 Epoch 1, train accuracy 0.99, mae 0.00370, time 11.8 sec
 Epoch 2, train accuracy 0.99, mae 0.00315, time 11.9 sec
 Epoch 3, train accuracy 0.99, mae 0.00327, time 11.9 sec
 Epoch 4, train accuracy 0.99, mae 0.00321, time 11.8 sec
 Epoch 5, train accuracy 1.00, mae 0.00305, time 11.7 sec

```
Epoch 6, train accuracy 1.00, mae 0.00286, time 11.8 sec
Epoch 7, train accuracy 1.00, mae 0.00297, time 11.8 sec
Epoch 8, train accuracy 1.00, mae 0.00293, time 11.9 sec
Epoch 9, train accuracy 1.00, mae 0.00291, time 11.8 sec
Epoch 10, train accuracy 1.00, mae 0.00285, time 11.9 sec
Epoch 11, train accuracy 1.00, mae 0.00264, time 11.8 sec
Epoch 12, train accuracy 1.00, mae 0.00268, time 11.7 sec
Epoch 13, train accuracy 1.00, mae 0.00279, time 11.8 sec
Epoch 14, train accuracy 1.00, mae 0.00277, time 11.9 sec
Epoch 15, train accuracy 1.00, mae 0.00280, time 11.9 sec
Epoch 16, train accuracy 1.00, mae 0.00283, time 11.7 sec
Epoch 17, train accuracy 1.00, mae 0.00262, time 11.8 sec
Epoch 18, train accuracy 1.00, mae 0.00260, time 11.8 sec
Epoch 19, train accuracy 1.00, mae 0.00273, time 11.7 sec
Epoch 20, train accuracy 1.00, mae 0.00272, time 11.9 sec
Epoch 21, train accuracy 1.00, mae 0.00261, time 11.8 sec
Epoch 22, train accuracy 1.00, mae 0.00270, time 11.8 sec
Epoch 23, train accuracy 1.00, mae 0.00251, time 11.8 sec
Epoch 24, train accuracy 1.00, mae 0.00253, time 11.8 sec
Epoch 25, train accuracy 1.00, mae 0.00256, time 11.9 sec
Epoch 26, train accuracy 1.00, mae 0.00259, time 11.6 sec
Epoch 27, train accuracy 1.00, mae 0.00246, time 11.8 sec
Epoch 28, train accuracy 1.00, mae 0.00250, time 11.8 sec
Epoch 29, train accuracy 1.00, mae 0.00240, time 11.8 sec
```

9.4.4 预测

在预测阶段，我们希望能把图片里面所有感兴趣的物体找出来。

我们先定一个数据读取和预处理函数。

```
In [26]: def process_image(fname):
    with open(fname, 'rb') as f:
        im = image.imread(f.read())
    # resize to data_shape
    data = image.imresize(im, data_shape, data_shape)
    # minus rgb mean
    data = data.astype('float32') - rgb_mean
    # convert to batch x channel x height x width
    return data.transpose((2,0,1)).expand_dims(axis=0), im
```

然后我们跟训练那样预测表框和其对应的物体。但注意到因为我们对每个像素都会生成数个锚框，这样我们可能会预测出大量相似的表框，从而导致结果非常嘈杂。一个办法是对于 IoU 比较高的两个表框，我们只保留预测执行度比较高的那个。这个算法（称之为 non maximum suppression）在 MultiBoxDetection 里实现了。下面我们实现预测函数：

```
In [27]: from mxnet.contrib.ndarray import MultiBoxDetection

def predict(x):
    anchors, cls_preds, box_preds = net(x.as_in_context(ctx))
    cls_probs = nd.SoftmaxActivation(
        cls_preds.transpose((0,2,1)), mode='channel')

    return MultiBoxDetection(cls_probs, box_preds, anchors,
                           force_suppress=True, clip=False)
```

预测函数会输出所有边框，每个边框由 [class_id, confidence, xmin, ymin, xmax, ymax] 表示。其中 class_id=-1 表示要么这个边框被预测只含有背景，或者被去重掉了。

```
In [28]: x, im = process_image('../img/pikachu.jpg')
out = predict(x)
out.shape
```

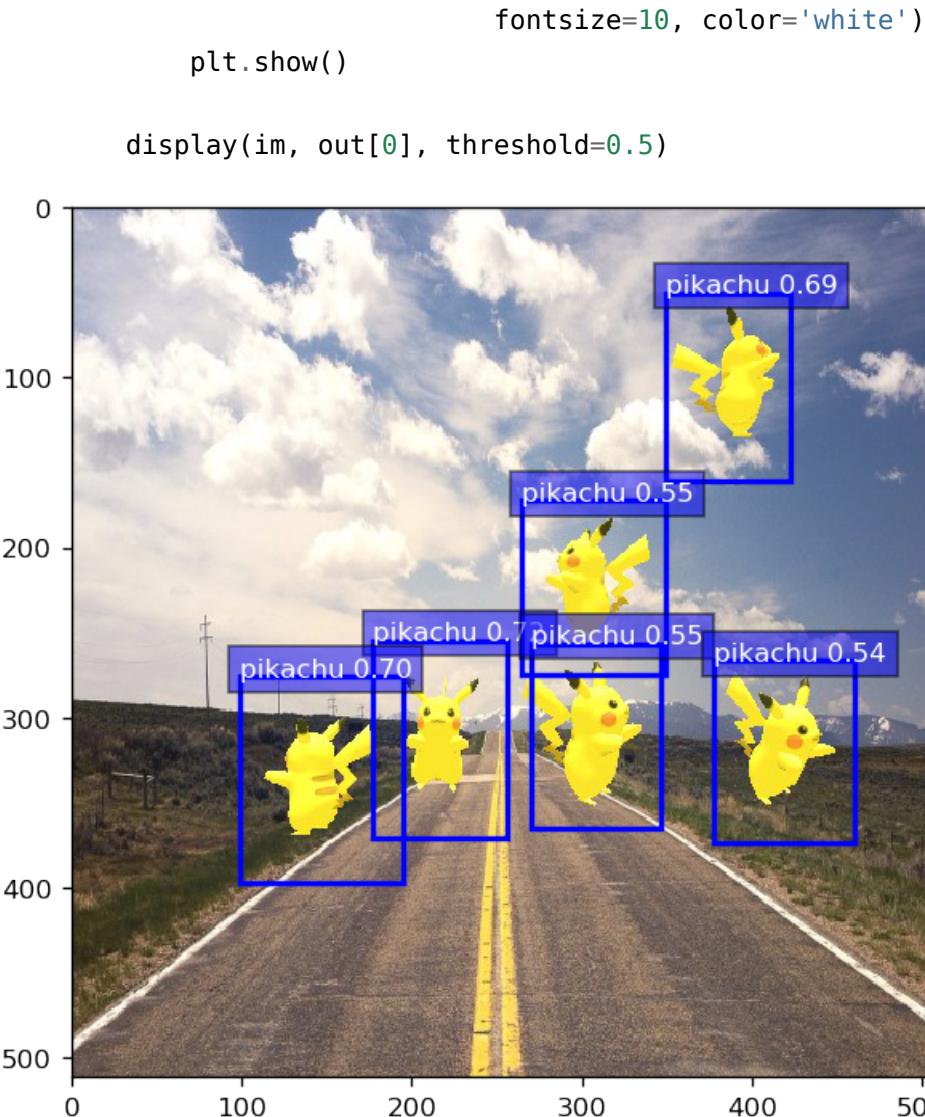
Out[28]: (1, 5444, 6)

最后我们将预测出置信度超过某个阈值的边框画出来：

```
In [29]: mpl.rcParams['figure.figsize'] = (6,6)

def display(im, out, threshold=0.5):
    plt.imshow(im.asnumpy())
    for row in out:
        row = row.asnumpy()
        class_id, score = int(row[0]), row[1]
        if class_id < 0 or score < threshold:
            continue
        color = colors[class_id%len(colors)]
        box = row[2:6] * np.array([im.shape[0],im.shape[1]]*2)
        rect = box_to_rect(nd.array(box), color, 2)
        plt.gca().add_patch(rect)

        text = class_names[class_id]
        plt.gca().text(box[0], box[1],
                      '{:s} {:.2f}'.format(text, score),
                      bbox=dict(facecolor=color, alpha=0.5),
```



9.4.5 结论

物体检测比分类要困难很多。因为我们不仅要预测物体类别，还要找到它们的位置。这一章我们展示我们还是可以在合理篇幅里实现 SSD 算法。

9.4.6 练习

我们有很多细节并没有展开讨论。例如

1. 锚框的大小和长宽比是如何选取的
2. MultiBoxTarget 里我们没有采样负例

3. 分类和回归损失我们直接加起来了，并没有给予权重

4. 在展示的时候如何选取阈值 `threshold`

吐槽和讨论欢迎点[这里](#)

9.5 语义分割：FCN

我们已经学习了如何识别图片里面的主要物体，和找出里面物体的边框。语义分割则在之上更进一步，它对每个像素预测它是否只是背景，还是属于哪个我们感兴趣的物体。



Fig. 9.8: Semantic Segmentation

可以看到，跟物体检测相比，语义分割预测的边框更加精细。

也许大家还听到过计算机视觉里的一个常用任务：图片分割。它跟语义分割类似，将每个像素划分的某个类。但它跟语义分割不同的时候，图片分割不需要预测每个类具体对应哪个物体。因此图片分割经常只需要利用像素之间的相似度即可，而语义分割则需要详细的类别标号。这也是为什么称其为语义的原因。

本章我们将介绍利用卷积神经网络解决语义分割的一个开创性工作之一：全链接卷积网络。在此之前我们先了解用来做语义分割的数据。

9.5.1 数据集

VOC2012是一个常用的语义分割数据集。输入图片跟之前的数据集类似，但标注也是保存称相应大小的图片来方便查看。下面代码下载这个数据集并解压。注意到压缩包大小是 2GB，可以预先下好放置在 `data_root` 下。

```
In [1]: import os
        import tarfile
        from mxnet import gluon

        data_root = '../data'
        voc_root = data_root + '/VOCdevkit/VOC2012'
        url = ('http://host.robots.ox.ac.uk/pascal/VOC/voc2012'
               '/VOCtrainval_11-May-2012.tar')
        sha1 = '4e443f8a2eca6b1dac8a6c57641b67dd40621a49'

        fname = gluon.utils.download(url, data_root, sha1_hash=sha1)

        if not os.path.isfile(voc_root+'/ImageSets/Segmentation/train.txt'):
            with tarfile.open(fname, 'r') as f:
                f.extractall(data_root)
```

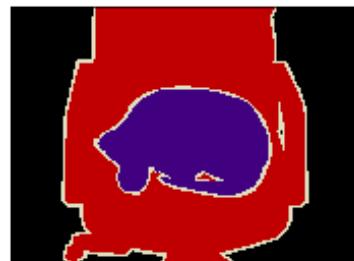
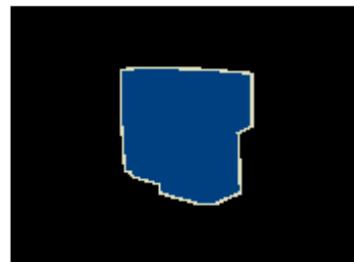
下面定义函数将训练图片和标注按序读进内存。

```
In [2]: from mxnet import image

def read_images(root=voc_root, train=True):
    txt_fname = root + '/ImageSets/Segmentation/' + (
        'train.txt' if train else 'val.txt')
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    n = len(images)
    data, label = [None] * n, [None] * n
    for i, fname in enumerate(images):
        data[i] = image.imread('%s/JPEGImages/%s.jpg' % (
            root, fname))
        label[i] = image.imread('%s/SegmentationClass/%s.png' % (
            root, fname))
    return data, label
```

我们画出前面三张图片和它们对应的标号。在标号中，白色代表边框黑色代表背景，其他不同的颜色对应不同物体。

```
In [3]: import sys  
sys.path.append('..')  
import utils  
  
train_images, train_labels = read_images()  
  
imgs = []  
for i in range(3):  
    imgs += [train_images[i], train_labels[i]]  
  
utils.show_images(imgs, nrows=3, ncols=2, figsize=(12,8))  
[im.shape for im in imgs]
```



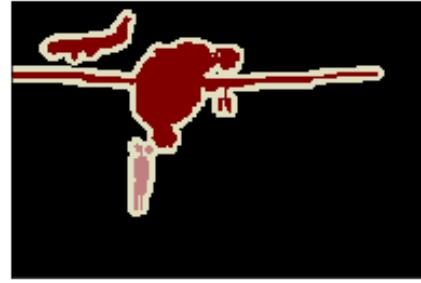
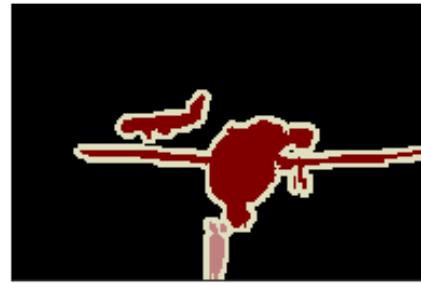
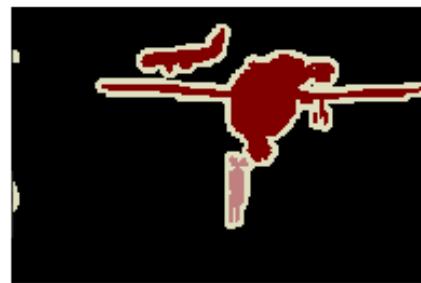
```
Out[3]: [(281, 500, 3),  
         (281, 500, 3),  
         (375, 500, 3),
```

```
(375, 500, 3),  
(375, 500, 3),  
(375, 500, 3)]
```

同时注意到图片的宽度基本是 500，但高度各不一样。为了能将多张图片合并成一个批量来加速计算，我们需要输入图片都是同样的大小。之前我们通过 `imresize` 来将他们调整成同样的大小。但在语义分割里，我们需要对标注做同样的变化来达到像素级别的匹配。但调整大小将改变像素颜色，使得再将它们映射到物体类别变得困难。

这里我们仅仅使用剪切来解决这个问题。就是说对于输入图片，我们随机剪切出一个固定大小的区域，然后对标号图片做同样位置的剪切。

```
In [4]: def rand_crop(data, label, height, width):  
    data, rect = image.random_crop(data, (width, height))  
    label = image.fixed_crop(label, *rect)  
    return data, label  
  
imgs = []  
for _ in range(3):  
    imgs += rand_crop(train_images[0], train_labels[0],  
                      200, 300)  
  
utils.show_images(imgs, nrows=3, ncols=2, figsize=(12,8))
```



接下来我们列出每个物体和背景对应的 RGB 值

```
In [5]: classes = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
                 'bottle', 'bus', 'car', 'cat', 'chair', 'cow', 'diningtable',
                 'dog', 'horse', 'motorbike', 'person', 'potted plant',
                 'sheep', 'sofa', 'train', 'tv/monitor']

# RGB color for each class
colormap = [[0,0,0],[128,0,0],[0,128,0], [128,128,0], [0,0,128],
            [128,0,128],[0,128,128],[128,128,128],[64,0,0],[192,0,0],
            [64,128,0],[192,128,0],[64,0,128],[192,0,128],
            [64,128,128],[192,128,128],[0,64,0],[128,64,0],
            [0,192,0],[128,192,0],[0,64,128]]
```

```
len(classes), len(colormap)
```

Out[5]: (21, 21)

这样给定一个标号图片，我们就可以将每个像素对应的物体标号找出来。

```
In [6]: import numpy as np
        from mxnet import nd

        cm2lbl = np.zeros(256**3)
        for i,cm in enumerate(colormap):
            cm2lbl[(cm[0]*256+cm[1])*256+cm[2]] = i

        def image2label(im):
            data = im.astype('int32').asnumpy()
            idx = (data[:, :, 0]*256+data[:, :, 1])*256+data[:, :, 2]
            return nd.array(cm2lbl[idx])
```

可以看到第一张训练图片的标号里面属于飞机的像素被标记成了 1.

```
In [7]: y = image2label(train_labels[0])
y[105:115, 130:140]
```

Out[7]:

```
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  0.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  1.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  1.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  1.  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.  0.  0.  1.  1.  1.]]
<NDArray 10x10 @cpu(0)>
```

现在我们可以定义数据读取了。每一次我们将图片和标注随机剪切到要求的形状，并将标注里每个像素转成对应的标号。简单起见我们将小于要求大小的图片全部过滤掉了。

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

        rgb_mean = nd.array([0.485, 0.456, 0.406])
        rgb_std = nd.array([0.229, 0.224, 0.225])

        def normalize_image(data):
            return (data.astype('float32') / 255 - rgb_mean) / rgb_std

        class VOCSegDataset(gluon.data.Dataset):
```

```

def _filter(self, images):
    return [im for im in images if (
        im.shape[0] >= self.crop_size[0] and
        im.shape[1] >= self.crop_size[1])]

def __init__(self, train, crop_size):
    self.crop_size = crop_size
    data, label = read_images(train=train)
    data = self._filter(data)
    self.data = [normalize_image(im) for im in data]
    self.label = self._filter(label)
    print('Read '+str(len(self.data))+' examples')

def __getitem__(self, idx):
    data, label = rand_crop(
        self.data[idx], self.label[idx],
        *self.crop_size)
    data = data.transpose((2, 0, 1))
    label = image2label(label)
    return data, label

def __len__(self):
    return len(self.data)

```

我们采用 320×480 的大小用来训练，注意到这个比前面我们使用的 224×224 要大上很多。但是同样我们将长宽都定义成了 32 的整数倍。

```

In [9]: # height x width
        input_shape = (320, 480)
        voc_train = VOCSegDataset(True, input_shape)
        voc_test = VOCSegDataset(False, input_shape)

Read 1114 examples
Read 1078 examples

```

最后定义批量读取。可以看到跟之前的不同是批量标号不再是一个向量，而是一个三维数组。

```

In [10]: batch_size = 64
        train_data = gluon.data.DataLoader(
            voc_train, batch_size, shuffle=True, last_batch='discard')
        test_data = gluon.data.DataLoader(
            voc_test, batch_size, last_batch='discard')

```

```
for data, label in train_data:  
    print(data.shape)  
    print(label.shape)  
    break  
  
(64, 3, 320, 480)  
(64, 320, 480)
```

9.5.2 全连接卷积网络

在数据的处理过程我们看到语义分割跟前面介绍的应用的主要区别在于，预测的标号不再是一个或者几个数字，而是每个像素都需要有标号。在卷积神经网络里，我们通过卷积层和池化层逐渐减少数据长宽但同时增加通道数。例如 ResNet18 里，我们先将输入长宽减少 32 倍，由 $3 \times 224 \times 224$ 的图片转成 $512 \times 7 \times 7$ 的输出，应该全局池化层变成 512 长向量，然后最后通过全链接层转成一个长度为 n 的输出向量，这里 n 是类数，既 `num_classes`。但在这里，对于输出为 $3 \times 320 \times 480$ 的图片，我们需要输出是 $n \times 320 \times 480$ ，就是每个输入像素都需要预测一个长度为 n 的向量。

全连接卷积网络 (FCN) 的提出是基于这样一个观察。假设 f 是一个卷积层，而且 $y = f(x)$ 。那么在反传求导时， $\partial f(y)$ 会返回一个跟 x 一样形状的输出。卷积是一个对偶函数，就是 $\partial^2 f = f$ 。那么如果我们想得到跟输入一样的输入，那么定义 $g = \partial f$ ，这样 $g(f(x))$ 就能达到我们想要的。

具体来说，我们定义一个卷积转置层 (transposed convolutional, 也经常被错误的叫做 deconvolutions)，它就是将卷积层的 `forward` 和 `backward` 函数兑换。

下面例子里我们看到使用同样的参数，除了替换输入和输出通道数外，`Conv2DTranspose` 可以将 `nn.Conv2D` 的输出还原其输入大小。

```
In [11]: from mxnet.gluon import nn  
  
conv = nn.Conv2D(10, kernel_size=4, padding=1, strides=2)  
conv_trans = nn.Conv2DTranspose(3, kernel_size=4, padding=1, strides=2)  
  
conv.initialize()  
conv_trans.initialize()  
  
x = nd.random.uniform(shape=(1, 3, 64, 64))  
y = conv(x)  
print('Input:', x.shape)  
print('After conv:', y.shape)  
print('After transposed conv', conv_trans(y).shape)  
  
Input: (1, 3, 64, 64)  
After conv: (1, 10, 32, 32)
```

After transposed conv (1, 3, 64, 64)

另外一点要注意的是，在最后的卷积层我们同样使用平化层（`nn.Flatten`）或者（全局）池化层来使得方便使用之后的全连接层作为输出。但是这样会损害空间信息，而这个对语义分割很重要。一个解决办法是去掉不需要的池化层，并将全连接层替换成 1×1 卷基层。

所以给定一个卷积网络，FCN 主要做下面的改动

- 替换全连接层成 1×1 卷基
- 去掉过于损失空间信息的池化层，例如全局池化
- 最后接上卷积转置层来得到需要大小的输出
- 为了训练更快，通常权重会初始化为预先训练好的权重

下面我们基于 Resnet18 来创建 FCN。首先我们下载一个预先训练好的模型。

```
In [12]: from mxnet.gluon.model_zoo import vision as models
pretrained_net = models.resnet18_v2(pretrained=True)

(pretrained_net.features[-4:], pretrained_net.output)

Out[12]: ([BatchNorm(axis=1, eps=1e-05, momentum=0.9, fix_gamma=False, in_channels=512),
           Activation(relu),
           GlobalAvgPool2D(size=(1, 1), stride=(1, 1), padding=(0, 0), ceil_mode=True),
           Flatten],
           Dense(512 -> 1000, linear))
```

我们看到 `feature` 模块最后两层是 `GlobalAvgPool2D` 和 `Flatten`，都是我们不需要的。所以我们定义一个新的网络，它复制除了最后两层的 `features` 模块的权重。

```
In [13]: net = nn.HybridSequential()
for layer in pretrained_net.features[:-2]:
    net.add(layer)

x = nd.random.uniform(shape=(1,3,*input_shape))
print('Input:', x.shape)
print('Output:', net(x).shape)
```

Input: (1, 3, 320, 480)

Output: (1, 512, 10, 15)

然后接上一个通道数等于类数的 1×1 卷积层。注意到 `net` 已经将输入长宽减少了 32 倍。那么我们需要接入一个 `strides=32` 的卷积转置层。我们使用一个比 `strides` 大两倍的 `kernel`，然后补上适当的填充。

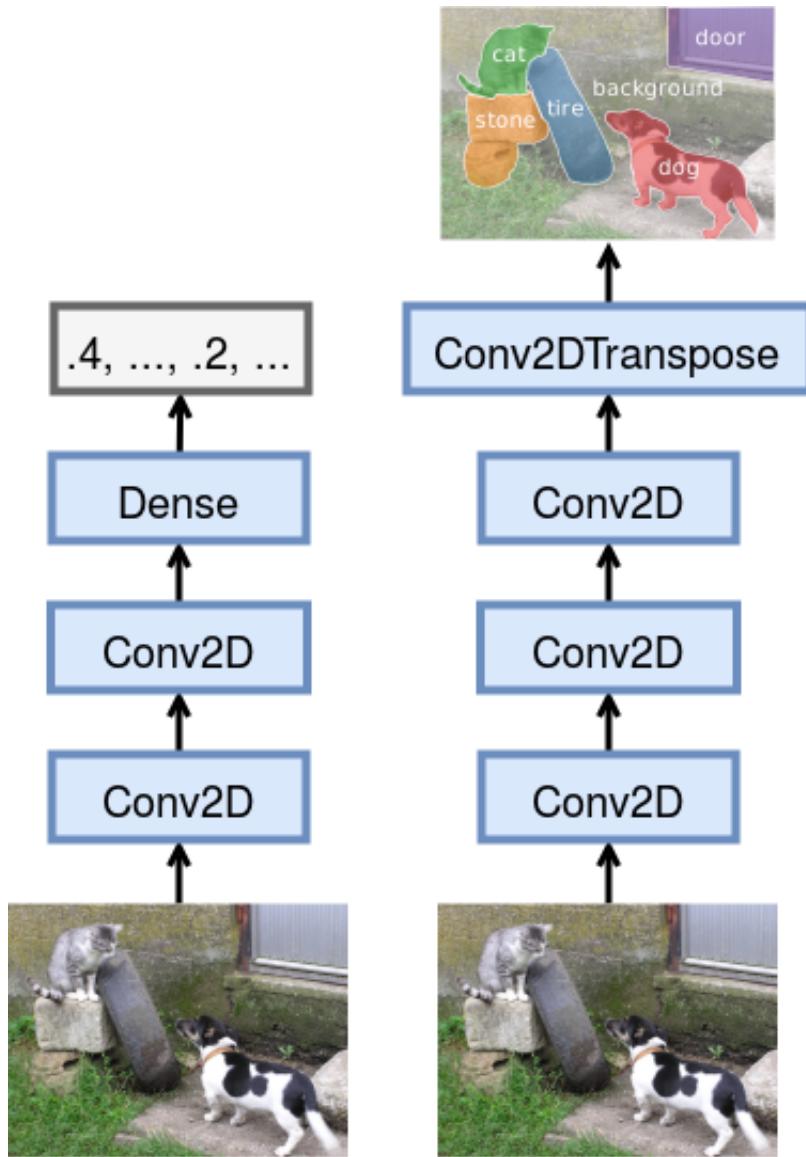


Fig. 9.9: FCN

```
In [14]: num_classes = len(classes)
```

```
with net.name_scope():
    net.add(
        nn.Conv2D(num_classes, kernel_size=1),
        nn.Conv2DTranspose(num_classes, kernel_size=64, padding=16, strides=32)
    )
```

9.5.3 训练

训练的时候我们需要初始化新添加的两层。我们可以随机初始化，但实际上发现将卷积转置层初始化成双线性差值函数可以使得训练更容易。

```
In [15]: def bilinear_kernel(in_channels, out_channels, kernel_size):
    factor = (kernel_size + 1) // 2
    if kernel_size % 2 == 1:
        center = factor - 1
    else:
        center = factor - 0.5
    og = np.ogrid[:kernel_size, :kernel_size]
    filt = (1 - abs(og[0] - center) / factor) * \
           (1 - abs(og[1] - center) / factor)
    weight = np.zeros(
        (in_channels, out_channels, kernel_size, kernel_size),
        dtype='float32')
    weight[range(in_channels), range(out_channels), :, :] = filt
    return nd.array(weight)
```

下面代码演示这样的初始化等价于对图片进行双线性差值放大。

```
In [16]: from matplotlib import pyplot as plt

x = train_images[0]
print('Input', x.shape)
x = x.astype('float32').transpose((2, 0, 1)).expand_dims(axis=0)/255

conv_trans = nn.Conv2DTranspose(
    3, in_channels=3, kernel_size=8, padding=2, strides=4)
conv_trans.initialize()
conv_trans(x)
conv_trans.weight.set_data(bilinear_kernel(3, 3, 8))
```

```
y = conv_trans(x)
y = y[0].clip(0,1).transpose((1,2,0))
print('Output', y.shape)

plt.imshow(y.asnumpy())
plt.show()

Input (281, 500, 3)
Output (1124, 2000, 3)
```



所以网络的初始化包括了三部分。主体卷积网络从训练好的 ResNet18 复制得来，替代 ResNet18 最后全连接的卷积层使用随机初始化。

最后的卷积转置层则使用双线性差值。对于卷积转置层，我们可以自定义一个初始化类。简单起见，这里我们直接通过权重的 `set_data` 函数改写权重。记得我们介绍过 Gluon 使用延后初始化来减少构造网络时需要制定输入大小。所以我们先随意初始化它，计算一次 `forward`，然后再改写权重。

```
In [17]: from mxnet import init

conv_trans = net[-1]
conv_trans.initialize(init=init.Zero())
net[-2].initialize(init=init.Xavier())

x = nd.zeros((batch_size, 3, *input_shape))
net(x)
```

```
shape = conv_trans.weight.data().shape
conv_trans.weight.set_data(bilinear_kernel(*shape[0:3]))
```

这时候我们可以真正开始训练了。值得一提的是我们使用卷积转置层的通道来预测像素的类别。所以在做 softmax 和预测的时候我们需要使用通道这个维度，既维度 1。所以在 SoftmaxCrossEntropyLoss 里加入了额外的 axis=1 选项。其他的部分跟之前的训练一致。

```
In [18]: import sys
        sys.path.append('..')
        import utils

        loss = gluon.loss.SoftmaxCrossEntropyLoss(axis=1)

        ctx = utils.try_all_gpus()
        net.collect_params().reset_ctx(ctx)

        trainer = gluon.Trainer(net.collect_params(),
                               'sgd', {'learning_rate': .1, 'wd': 1e-3})

        utils.train(train_data, test_data, net, loss,
                   trainer, ctx, num_epochs=10)

Start training on [gpu(0), gpu(1)]
Epoch 0. Loss: 1.623, Train acc 0.71, Test acc 0.76, Time 63.3 sec
Epoch 1. Loss: 0.631, Train acc 0.82, Test acc 0.83, Time 29.4 sec
Epoch 2. Loss: 0.494, Train acc 0.85, Test acc 0.84, Time 29.4 sec
Epoch 3. Loss: 0.406, Train acc 0.87, Test acc 0.84, Time 29.4 sec
Epoch 4. Loss: 0.364, Train acc 0.88, Test acc 0.85, Time 29.4 sec
Epoch 5. Loss: 0.336, Train acc 0.89, Test acc 0.84, Time 29.5 sec
Epoch 6. Loss: 0.299, Train acc 0.90, Test acc 0.85, Time 29.5 sec
Epoch 7. Loss: 0.282, Train acc 0.91, Test acc 0.85, Time 29.4 sec
Epoch 8. Loss: 0.261, Train acc 0.91, Test acc 0.85, Time 29.4 sec
Epoch 9. Loss: 0.244, Train acc 0.92, Test acc 0.85, Time 29.4 sec
```

9.5.4 预测

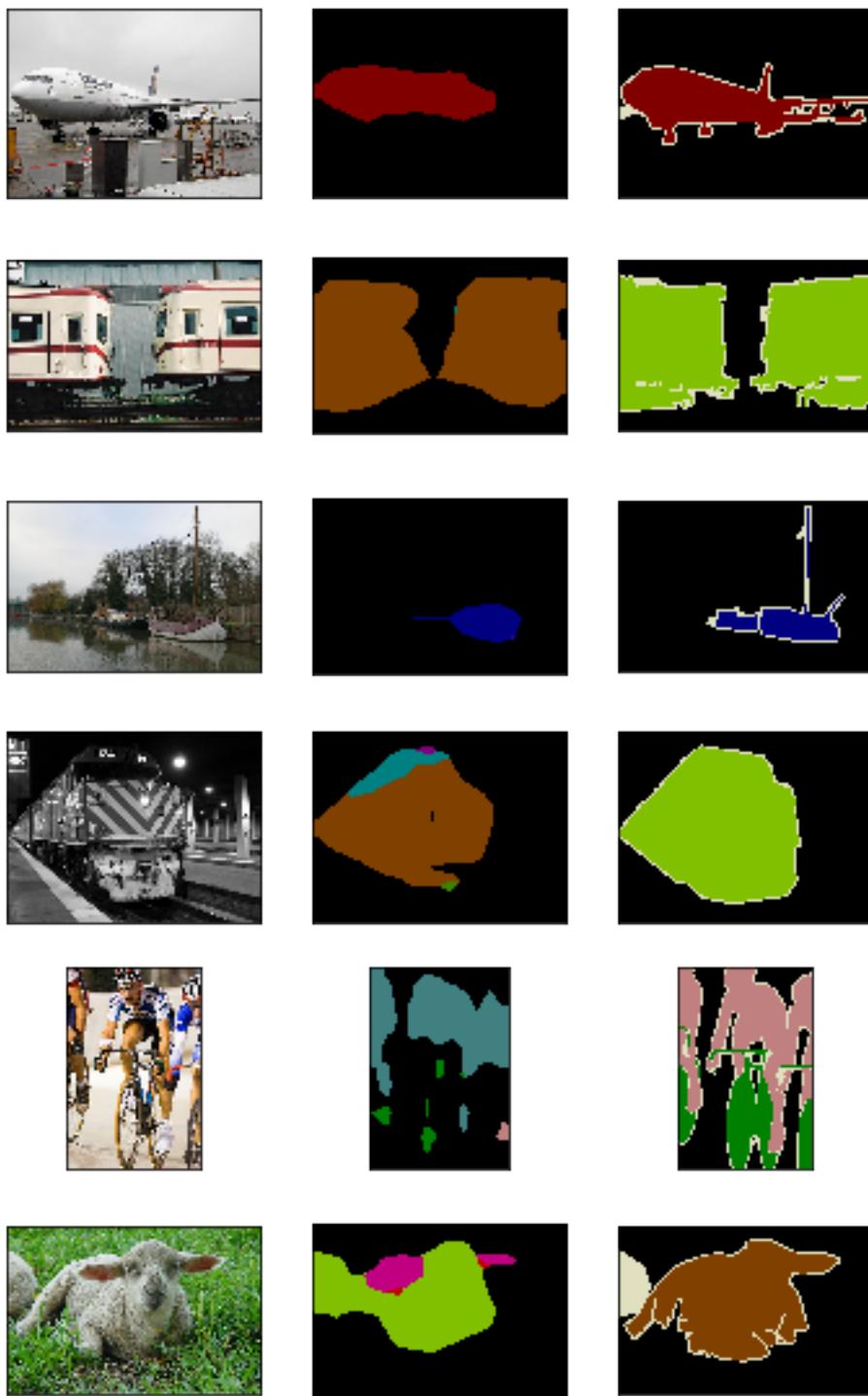
预测函数跟之前的图片分类预测类似，但跟上面一样，主要不同在于我们需要在 axis=1 上做 argmax。同时我们定义 image2label 的反函数，它将预测值转成图片。

```
In [19]: def predict(im):
        data = normalize_image(im)
```

```
data = data.transpose((2,0,1)).expand_dims(axis=0)
yhat = net(data.as_in_context(ctx[0]))
pred = nd.argmax(yhat, axis=1)
return pred.reshape((pred.shape[1], pred.shape[2]))\n\ndef label2image(pred):
    x = pred.astype('int32').asnumpy()
    cm = np.array(colormap).astype('uint8')
    return nd.array(cm[x,:])
```

我们读取前几张测试图片并对其进行预测。

```
In [20]: test_images, test_labels = read_images(train=False)\n\nn = 6
imgs = []
for i in range(n):
    x = test_images[i]
    pred = label2image(predict(x))
    imgs += [x, pred, test_labels[i]]\n\nutils.show_images(imgs, nrows=n, ncols=3, figsize=(6,10))
```



9.5.5 总结

通过使用卷积转置层，我们可以得到更大分辨率的输出。

9.5.6 练习

1. 试着改改最后的卷积转置层的参数设定
2. 看看双线性差值初始化是不是必要的
3. 试着改改训练参数来使得收敛更好些
4. FCN 论文中提到了不只是使用主体卷积网络输出，还可以将前面层的输出也加进来。试着实现。

吐槽和讨论欢迎点[这里](#)

9.6 样式迁移

喜欢拍照的同学可能都接触过滤镜，它们能改变照片的颜色风格，使得风景照更加锐利，或者人像更加美白。但一个滤镜通常只能改变照片的某个方面，要达到想要的风格，经常需要我们大量组合尝试多个滤镜。这个过程被通常称之为“PS一下”。对于简单的调整，例如拉一拉颜色曲线变成日系小清新风或者加点德味，通常都不难做到。但对于复杂的要求，例如将图片调成梵高风，则需要大量的专业技巧。例如 17 年上映的《挚爱梵高》由 115 名专业画师画了 65,000 张梵高风格的油画而得。

一个自然的想法是，我们能不能通过神经网络来自动化这个过程。具体来说，我们希望将一张指定的图片的风格，例如梵高的某张油画，应用到另外一张内容图片上。

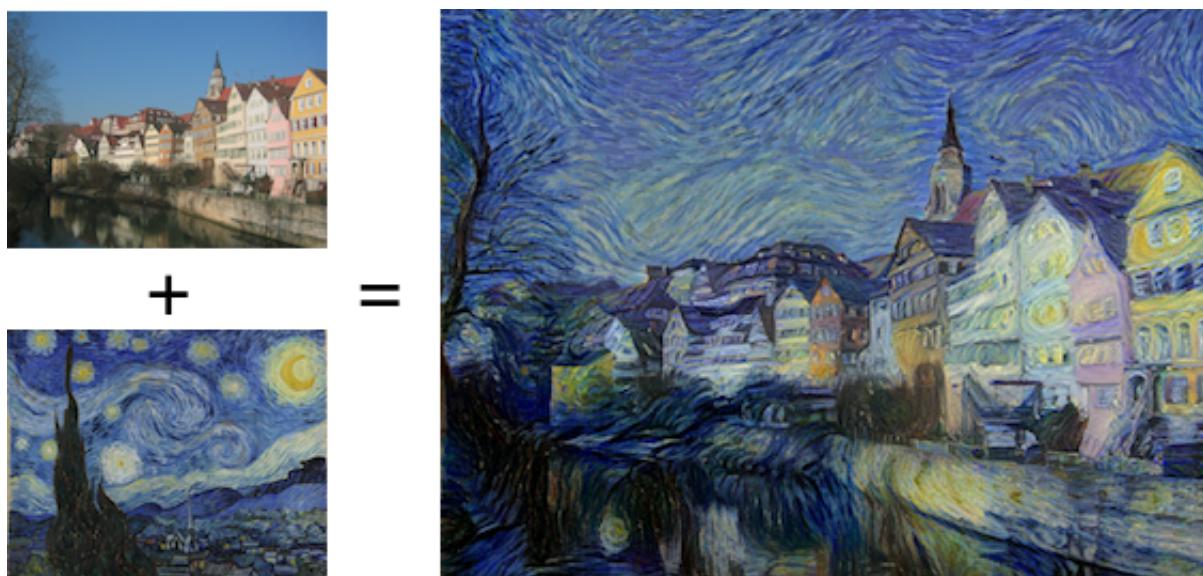


Fig. 9.10: Neural Style

Gatys 等人开创性的通过匹配卷积神经网络的中间层输出来训练出合成图片。它的流程如下所示：

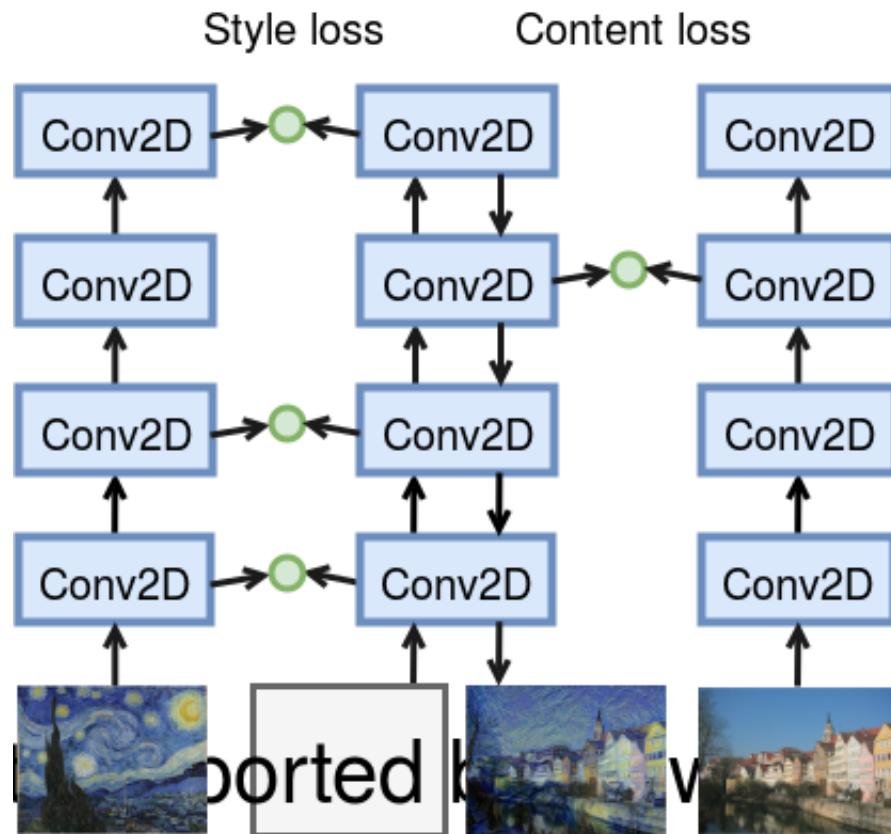


Fig. 9.11: Neural Style Training

1. 我们首先挑选一个卷积神经网络来提取特征。我们选择它的特定层来匹配样式，特定层来匹配内容。示意图中我们选择层 1,2,4 作为样式层，层 3 作为内容层。
2. 输入样式图片并保存样式层输出，记第 i 层输出为 s_i
3. 输入内容图片并保存内容层输出，记第 i 层输出为 c_i
4. 初始化合成图片 x 为随机值或者其他更好的初始值。然后进行迭代使得用 x 抽取的特征能够匹配上 s_i 和 c_i 。具体来说，我们如下迭代直到收敛。
5. 输入 x 计算样式层和内容层输出，记第 i 层输出为 y_i
6. 使用样式损失函数来计算 y_i 和 s_i 的差异
7. 使用内容损失函数来计算 y_i 和 c_i 的差异
8. 对损失求和并对输入 x 求导，记导数为 g
9. 更新 x ，例如 $x = x - \eta g$

内容损失函数使用通常回归用的均方误差。对于样式，我们可以将它看成是像素点在每个通道的统计分布。例如要匹配两张图片的颜色，我们的一个做法是匹配这两张图片在 RGB 这三个通道上的直方图。更一般的，假设卷积层的输出格式是 $c \times h \times w$ ，既 `channels × height × width`。那么我们可以把它变形成 $c \times hw$ 的 2D 数组，并将它看成是一个维度为 c 的随机变量采样到的 hw 个点。所谓的样式匹配就是使得两个 c 维随机变量统计分布一致。

匹配统计分布常用的做法是冲量匹配，就是说使得他们有一样的均值，协方差，和其他高维的冲量。为了计算简单起见，我们这里假设卷积输出已经是均值为 0 了，而且我们只匹配协方差。也就是说，样式损失函数就是对 s_i 和 y_i 计算 Gram 矩阵然后应用均方误差

$$\text{styleloss}(s_i, y_i) = \frac{1}{c^2 hw} \|s_i s_i^T - y_i y_i^T\|_F$$

这里假设我们已经将 s_i 和 y_i 变形成了 $c \times hw$ 的 2D 矩阵了。

下面我们将实现这个算法来深入理解各个参数，例如样式层和内容层的选取，对实际结果的影响。

9.6.1 数据

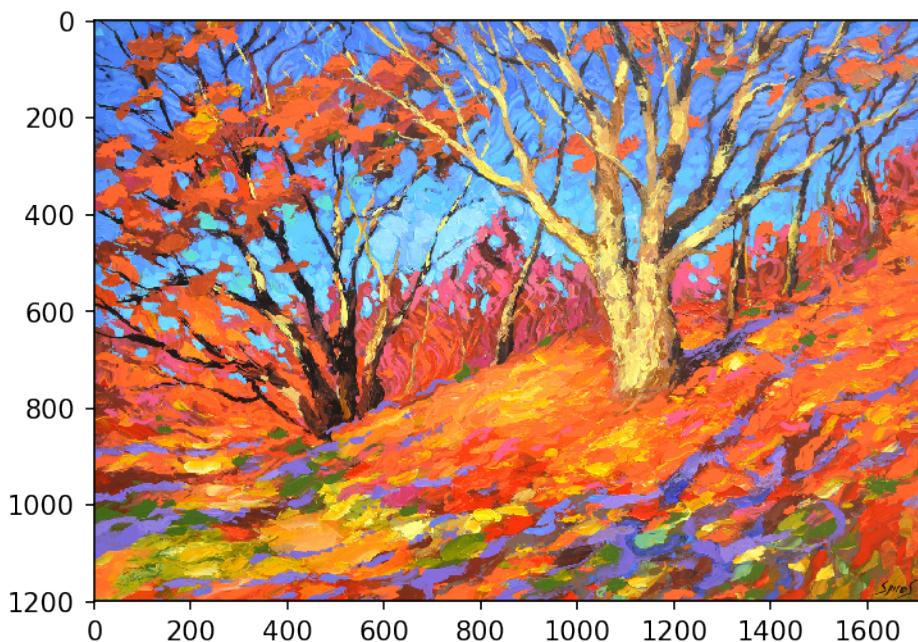
我们将尝试将下面的水粉橡树的样式应用到实拍的松树上。

```
In [1]: %matplotlib inline
import matplotlib as mpl
mpl.rcParams['figure.dpi']= 150
import matplotlib.pyplot as plt

from mxnet import image

style_img = image.imread('../img/autumn_oak.jpg')
content_img = image.imread('../img/pine-tree.jpg')

plt.imshow(style_img.asnumpy())
plt.show()
plt.imshow(content_img.asnumpy())
plt.show()
```



跟前面教程一样我们定义预处理和后处理函数，它们将原始图片进行归一化并转换成卷积网络接受的输入格式，和还原成能展示的图片格式。

```
In [2]: from mxnet import nd

rgb_mean = nd.array([0.485, 0.456, 0.406])
rgb_std = nd.array([0.229, 0.224, 0.225])

def preprocess(img, image_shape):
```

```
    img = image.imresize(img, *image_shape)
    img = (img.astype('float32')/255 - rgb_mean) / rgb_std
    return img.transpose((2,0,1)).expand_dims(axis=0)

def postprocess(img):
    img = img[0].as_in_context(rgb_std.context)
    return (img.transpose((1,2,0))*rgb_std + rgb_mean).clip(0,1)
```

9.6.2 模型

我们使用原论文使用的 VGG 19 模型。并下载在 Imagenet 上训练好的权重。

In [3]: `from mxnet.gluon.model_zoo import vision as models`

```
pretrained_net = models.vgg19(pretrained=True)
pretrained_net
```

Out[3]: VGG(
 (features): HybridSequential(
 (0): Conv2D(3 -> 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
 (1): Activation(relu)
 (2): Conv2D(64 -> 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
 (3): Activation(relu)
 (4): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
 (5): Conv2D(64 -> 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
 (6): Activation(relu)
 (7): Conv2D(128 -> 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
 (8): Activation(relu)
 (9): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
 (10): Conv2D(128 -> 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
 (11): Activation(relu)
 (12): Conv2D(256 -> 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
 (13): Activation(relu)
 (14): Conv2D(256 -> 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
 (15): Activation(relu)
 (16): Conv2D(256 -> 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
 (17): Activation(relu)
 (18): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
 (19): Conv2D(256 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
 (20): Activation(relu)
 (21): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
 (22): Activation(relu)

```

(23): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(24): Activation(relu)
(25): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(26): Activation(relu)
(27): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
(28): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(29): Activation(relu)
(30): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(31): Activation(relu)
(32): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(33): Activation(relu)
(34): Conv2D(512 -> 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(35): Activation(relu)
(36): MaxPool2D(size=(2, 2), stride=(2, 2), padding=(0, 0), ceil_mode=False)
(37): Dense(25088 -> 4096, Activation(relu))
(38): Dropout(p = 0.5)
(39): Dense(4096 -> 4096, Activation(relu))
(40): Dropout(p = 0.5)
)
(output): Dense(4096 -> 1000, linear)
)

```

回忆[VGG](#)这一章里，我们使用五个卷积块 `vgg_block` 来构建网络。快之间使用 `nn.MaxPool2D` 来做间隔。我们有很多种选择来使用某些层作为样式和内容的匹配层。通常越靠近输入层越容易匹配内容和样式的细节信息，越靠近输出则越倾向于语义的内容和全局的样式。这里我们按照原论文使用每个卷积块的第一个卷基层输出来匹配样式，和第四个块中的最后一个卷积层来匹配内容。根据 `pretrained_net` 的输出我们记录下这些层对应的位置。

```
In [4]: style_layers = [0,5,10,19,28]
content_layers = [25]
```

因为只需要使用中间层的输出，我们构建一个新的网络，它只保留我们需要的层。

```
In [5]: from mxnet.gluon import nn
```

```

def get_net(pretrained_net, content_layers, style_layers):
    net = nn.Sequential()
    for i in range(max(content_layers+style_layers)+1):
        net.add(pretrained_net.features[i])
    return net

net = get_net(pretrained_net, content_layers, style_layers)
```

给定输入 x , 简单使用 $\text{net}(x)$ 只能拿到最后的输出, 而这里我们还需要 net 的中间层输出。因此我们逐层计算, 并保留需要的输出。

```
In [6]: def extract_features(x, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        x = net[i](x)
        if i in style_layers:
            styles.append(x)
        if i in content_layers:
            contents.append(x)
    return contents, styles
```

9.6.3 损失函数

内容匹配是一个典型的回归问题, 我们将来使用均方误差来比较内容层的输出。

```
In [7]: def content_loss(yhat, y):
    return (yhat - y).square().mean()
```

样式匹配则是通过拟合 Gram 矩阵。我们先定义它的计算:

```
In [8]: def gram(x):
    c = x.shape[1]
    n = x.size / x.shape[1]
    y = x.reshape((c, int(n)))
    return nd.dot(y, y.T) / n

gram(nd.ones((1, 3, 4, 4)))
```

Out[8]:

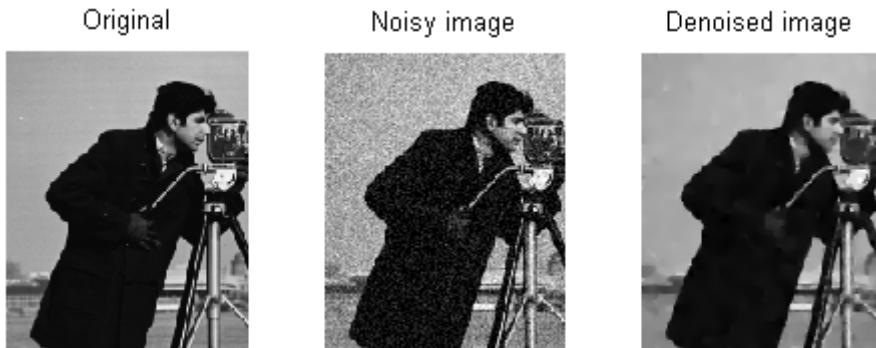
```
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
<NDArray 3x3 @cpu(0)>
```

和对应的损失函数。对于要匹配的样式图片它的样式输出在训练中不会改变, 我们将提前计算好它的 Gram 矩阵来作为输入使得计算加速。

```
In [9]: def style_loss(yhat, gram_y):
    return (gram(yhat) - gram_y).square().mean()
```

当使用靠近输出层的高层输出来拟合时, 经常可以观察到学到的图片里面有大量高频噪音。这个有点类似老式天线电视机经常遇到的白噪音。有多种方法来降噪, 例如可以加入模糊滤镜, 或者使

用总变差降噪 (Total Variation Denoising)。



假设 $x_{i,j}$ 表示像素 (i,j) , 那么我们加入下面的损失函数, 它使得邻近的像素值相似:

$$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}|$$

```
In [10]: def tv_loss(yhat):
    return 0.5*((yhat[:, :, 1:, :] - yhat[:, :, :-1, :]).abs().mean() +
                (yhat[:, :, :, 1:] - yhat[:, :, :, :-1]).abs().mean())
```

总损失函数是上述三个损失函数的加权和。通过调整权重值我们可以控制学到的图片是否保留更多样式, 更多内容, 还是更加干净。注意到样式匹配中我们使用了 5 个层的输出, 我们对靠近输入的层给予比较大的权重。

```
In [11]: channels = [net[l].weight.shape[0] for l in style_layers]
style_weights = [1e4/n**2 for n in channels]
content_weights = [1]
tv_weight = 10
```

我们可以使用 `nd.add_n` 来将多个损失函数的输出按权重加起来。

```
In [12]: def sum_loss(loss, preds, truths, weights):
    return nd.add_n(*[w*loss(yhat, y) for w, yhat, y in zip(
        weights, preds, truths)])
```

9.6.4 训练

首先我们定义两个函数, 他们分别对源内容图片和源样式图片提取特征。

```
In [13]: def get_contents(image_shape):
    content_x = preprocess(content_img, image_shape).copyto(ctx)
    content_y, _ = extract_features(content_x, content_layers, style_layers)
    return content_x, content_y
```

```
def get_styles(image_shape):
    style_x = preprocess(style_img, image_shape).copyto(ctx)
    _, style_y = extract_features(style_x, content_layers, style_layers)
    style_y = [gram(y) for y in style_y]
    return style_x, style_y
```

训练过程跟之前的主要的主要不同在于

1. 这里我们的损失函数更加复杂。
2. 我们只对输入进行更新，这个意味着我们需要对输入 x 预先分配了梯度。
3. 我们可能会替换匹配内容和样式的层，和调整他们之间的权重，来得到不同风格的输出。这里我们对梯度做了一般化，使得不同参数下的学习率不需要太大变化。
4. 仍然使用简单的梯度下降，但每 n 次迭代我们会减小一次学习率

```
In [14]: from time import time
        from mxnet import autograd

        def train(x, max_epochs, lr, lr_decay_epoch=200):
            tic = time()
            for i in range(max_epochs):
                with autograd.record():
                    content_py, style_py = extract_features(
                        x, content_layers, style_layers)
                    content_L = sum_loss(
                        content_loss, content_py, content_y, content_weights)
                    style_L = sum_loss(
                        style_loss, style_py, style_y, style_weights)
                    tv_L = tv_weight * tv_loss(x)
                    loss = style_L + content_L + tv_L

                    loss.backward()
                    x.grad[:] /= x.grad.abs().mean() + 1e-8
                    x[:] -= lr * x.grad
                    # add sync to avoid large mem usage
                    nd.waitall()

                if i and i % 20 == 0:
                    print('batch %d, content %.2f, style %.2f, '
                          'TV %.2f, time %.1f sec' % (
                          i, content_L.asscalar(), style_L.asscalar(),
                          tv_L.asscalar(), time()-tic))
```

```

tic = time()

if i and i % lr_decay_epoch == 0:
    lr *= 0.1
    print('change lr to ', lr)

plt.imshow(postprocess(x).asnumpy())
plt.show()
return x

```

现在所有函数都定义好了，我们可以真正开始训练了。从性能上考虑，首先我们在一个大小为 300×200 的输入上进行训练。因为我们不会更新源图片和模型参数，所以我们可以提前抽取好他们的特征。我们把要合成图片的初始值设成内容图片来加速收敛。

```

In [15]: import sys
        sys.path.append('..')
        import utils

        image_shape = (300, 200)

        ctx = utils.try_gpu()
        net.collect_params().reset_ctx(ctx)

        content_x, content_y = get_contents(image_shape)
        style_x, style_y = get_styles(image_shape)

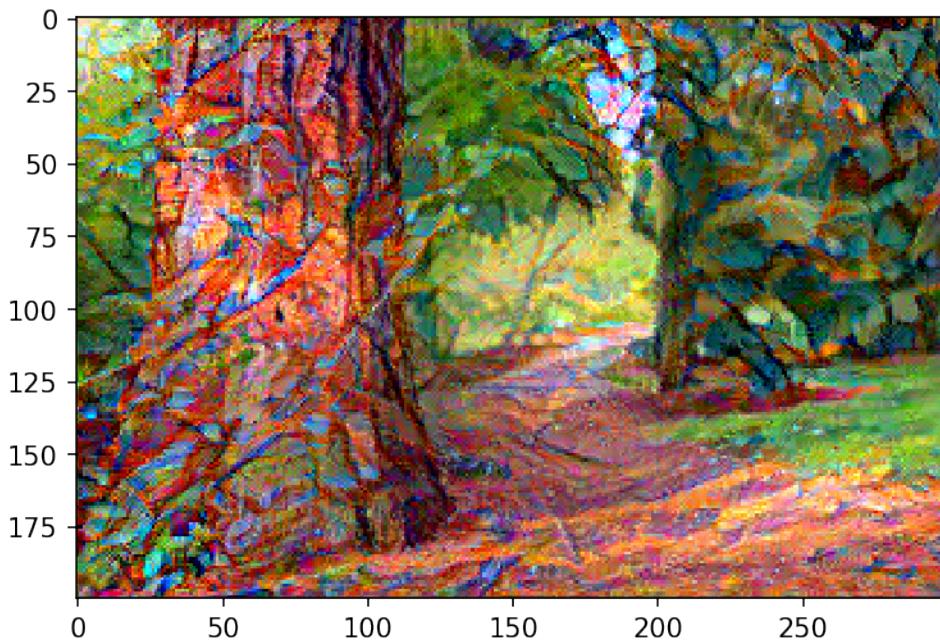
        x = content_x.copyto(ctx)
        x.attach_grad()

        y = train(x, 500, 0.1)

batch 20, content 29.82, style 594.68, TV 4.60, time 1.4 sec
batch 40, content 30.75, style 321.59, TV 4.93, time 1.3 sec
batch 60, content 28.44, style 367.96, TV 5.06, time 1.3 sec
batch 80, content 30.43, style 279.22, TV 5.35, time 1.3 sec
batch 100, content 30.22, style 183.02, TV 5.47, time 1.3 sec
batch 120, content 28.69, style 222.55, TV 5.50, time 1.3 sec
batch 140, content 28.89, style 213.33, TV 5.69, time 1.3 sec
batch 160, content 29.00, style 161.57, TV 5.78, time 1.3 sec
batch 180, content 28.33, style 182.41, TV 5.77, time 1.3 sec
batch 200, content 28.42, style 199.64, TV 5.94, time 1.3 sec
change lr to 0.01000000000000002
batch 220, content 24.38, style 31.79, TV 5.74, time 1.3 sec

```

```
batch 240, content 22.06, style 27.11, TV 5.68, time 1.3 sec
batch 260, content 20.85, style 24.52, TV 5.64, time 1.3 sec
batch 280, content 19.89, style 23.38, TV 5.61, time 1.3 sec
batch 300, content 19.47, style 21.25, TV 5.60, time 1.3 sec
batch 320, content 18.74, style 20.69, TV 5.55, time 1.3 sec
batch 340, content 18.56, style 19.32, TV 5.55, time 1.3 sec
batch 360, content 18.15, style 18.25, TV 5.52, time 1.3 sec
batch 380, content 17.75, style 18.42, TV 5.49, time 1.3 sec
batch 400, content 17.48, style 18.87, TV 5.48, time 1.3 sec
change lr to 0.001000000000000002
batch 420, content 16.96, style 12.41, TV 5.46, time 1.3 sec
batch 440, content 16.61, style 12.00, TV 5.44, time 1.3 sec
batch 460, content 16.34, style 11.69, TV 5.43, time 1.3 sec
batch 480, content 16.11, style 11.43, TV 5.42, time 1.3 sec
```



观察损失值的变化。因为我们使用了内容图片作为初始化，所以一开始看到样式损失比较大。但随着迭代的进行，它减少的非常迅速，尤其是在每次调整学习率后。噪音在训练中有略微的增加，但在后期还是控制在合理的范围。

最后的结果里可以看到明显的我们将样式图片里面的大的色块应用到了内容图片上。但是由于输入图片大小比较小，所以看到细节上比较模糊。

下面我们在更大的 1200×800 的尺寸上训练，希望能得到更加清晰的合成图片。同样为了加速收敛，我们将前面得到的合成图片放大成我们要的尺寸做为初始值。

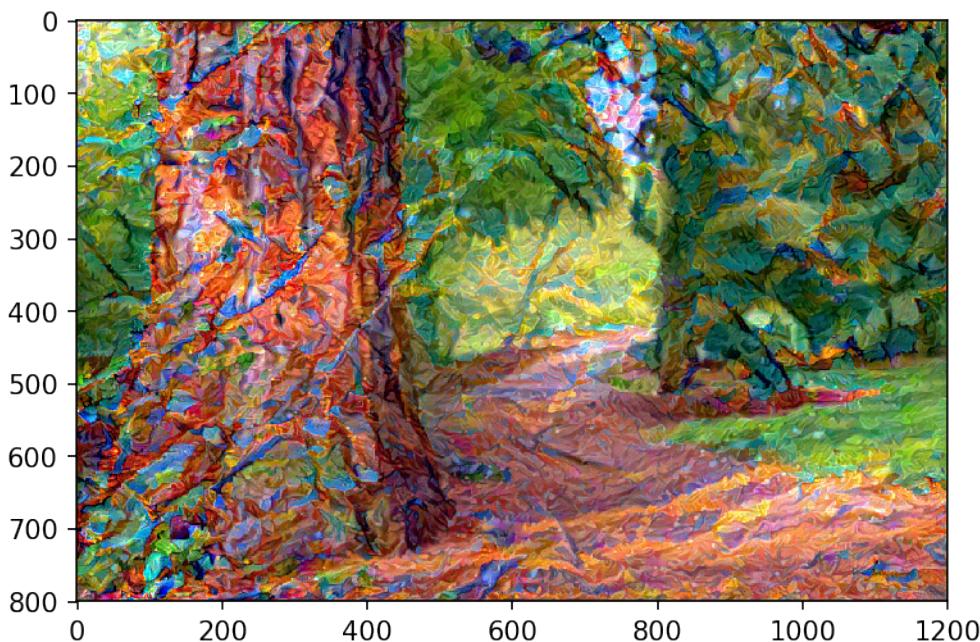
In [16]: `image_shape = (1200,800)`

```
content_x, content_y = get_contents(image_shape)
style_x, style_y = get_styles(image_shape)

x = preprocess(postprocess(y)*255, image_shape).copyto(ctx)
x.attach_grad()

z = train(x, 300, 0.1, 100)

batch 20, content 48.22, style 773.74, TV 3.29, time 18.7 sec
batch 40, content 46.03, style 715.05, TV 3.59, time 17.5 sec
batch 60, content 46.00, style 722.61, TV 3.82, time 17.5 sec
batch 80, content 46.64, style 755.22, TV 4.01, time 17.5 sec
batch 100, content 46.83, style 787.54, TV 4.18, time 17.5 sec
change lr to 0.01000000000000002
batch 120, content 29.22, style 62.62, TV 3.68, time 17.6 sec
batch 140, content 25.32, style 30.62, TV 3.47, time 17.6 sec
batch 160, content 23.03, style 45.08, TV 3.38, time 17.6 sec
batch 180, content 23.18, style 26.50, TV 3.32, time 17.6 sec
batch 200, content 21.39, style 28.34, TV 3.27, time 17.6 sec
change lr to 0.00100000000000002
batch 220, content 20.30, style 13.06, TV 3.23, time 17.6 sec
batch 240, content 19.50, style 11.98, TV 3.20, time 17.6 sec
batch 260, content 18.91, style 11.30, TV 3.18, time 17.5 sec
batch 280, content 18.44, style 10.79, TV 3.17, time 17.6 sec
```



可以看到由于初始值更加好，这次的收敛更加迅速，虽然每次迭代花时间更长。由于是图片更大，我们可以更清楚的看到细节。里面不仅有大块的色彩，色彩块里面也有细微的纹理。这是由于我们在匹配样式的时候使用了多层的输出。

最后我们可以把合成的图片保存下来。

```
In [17]: plt.imsave('result.png', postprocess(z).asnumpy())
```

9.6.5 总结

通过匹配神经网络的中间层输出，我们可以有效的融合不同图片的内容和样式。

9.6.6 练习

1. 改变内容和样式层
2. 使用不同的权重
3. 换几张样式和内容图片

吐槽和讨论欢迎点[这里](#)

9.7 实战 Kaggle 比赛——使用 Gluon 对原始图像文件分类（CIFAR-10）

我们在监督学习中的一章里，以房价预测问题为例，介绍了如何使用 Gluon 来实战Kaggle 比赛。

我们在本章中选择了 Kaggle 中著名的CIFAR-10 原始图像分类问题。我们以该问题为例，为大家提供使用 Gluon 对原始图像文件进行分类的示例代码。

计算机视觉一直是深度学习的主战场，请

Get your hands dirty。

9.7.1 Kaggle 中的 CIFAR-10 原始图像分类问题

Kaggle是一个著名的供机器学习爱好者交流的平台。为了便于提交结果，请大家注册Kaggle账号。然后请大家先点击CIFAR-10 原始图像分类问题了解有关本次比赛的信息。

CIFAR-10 - Object Recognition in Images

Identify the subject of 60,000 labeled images
231 teams · 3 years ago

Overview Data Discussion Leaderboard Rules Team My Submissions **Late Submission**

Overview

Description	CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the 80 million tiny images dataset and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.
Evaluation	Kaggle is hosting a CIFAR-10 leaderboard for the machine learning community to use for fun and practice. You can see how your approach compares to the latest research methods on Rodrigo Benenson's classification results page .

9.7.2 整理原始数据集

比赛数据分为训练数据集和测试数据集。训练集包含 5 万张图片。测试集包含 30 万张图片：其中有 1 万张图片用来计分，但为了防止人工标注测试集，里面另加了 29 万张不计分的图片。

两个数据集都是 png 彩色图片，大小为 $32 \times 32 \times 3$ 。训练集一共有 10 类图片，分别为飞机、汽车、鸟、猫、鹿、狗、青蛙、马、船和卡车。

(那么问题来了，你觉得你用肉眼能把下面 100 个图片正确分类吗?)

下载数据集

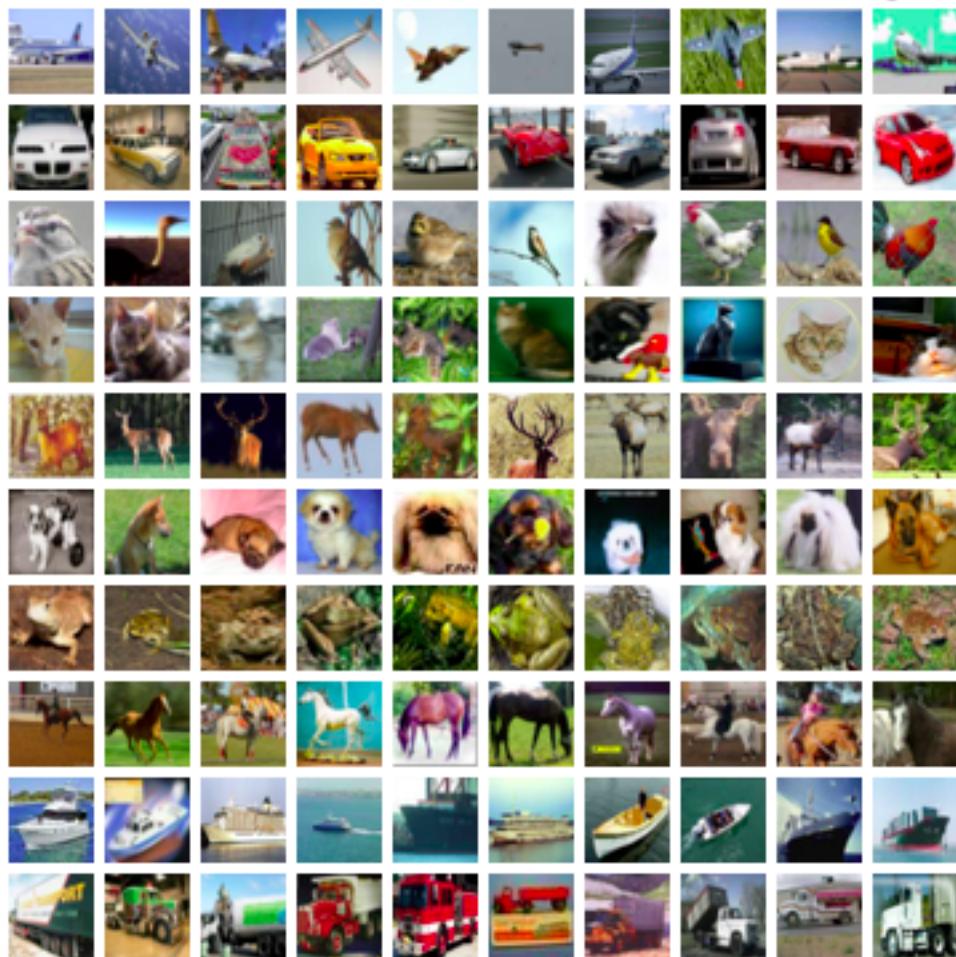
登录 Kaggle 后，数据可以从[CIFAR-10 原始图像分类问题](#)中下载。

- [训练数据集 train.7z 下载地址](#)
- [测试数据集 test.7z 下载地址](#)
- [训练数据标签 trainLabels.csv 下载地址](#)

解压数据集

训练数据集 `train.7z` 和测试数据集 `test.7z` 都是压缩格式，下载后请解压缩。解压缩后原始数据集的路径可以如下：

- `../data/kaggle_cifar10/train/[1-50000].png`
- `../data/kaggle_cifar10/test/[1-300000].png`
- `../data/kaggle_cifar10/trainLabels.csv`



为了使网页编译快一点,我们在 git repo 里仅仅存放 100 个训练样本('train_tiny.zip')和 1 个测试样本('test_tiny.zip')。执行以下代码会从 git repo 里解压生成小样本训练和测试数据,文件夹名称分别为' train_tiny' 和' test_tiny'。训练数据标签的压缩文件将被解压成 trainLabels.csv。

```
In [1]: # 如果训练下载的 Kaggle 的完整数据集, 把下面改 False
demo = True
if demo:
    import zipfile
    for fin in ['train_tiny.zip', 'test_tiny.zip', 'trainLabels.csv.zip']:
        with zipfile.ZipFile('../data/kaggle_cifar10/' + fin, 'r') as zin:
            zin.extractall('../data/kaggle_cifar10/')
```

整理数据集

我们定义下面的 reorg_cifar10_data 函数来整理数据集。整理后,同一类图片将出现在在同一个文件夹下,便于 Gluon 稍后读取。

函数中的参数如 data_dir、train_dir 和 test_dir 对应上述数据存放路径及训练和测试的图片集文件夹名称。参数 label_file 为训练数据标签的文件名称。参数 input_dir 是整理后数据集文件夹名称。参数 valid_ratio 是验证集占原始训练集的比重。以 valid_ratio=0.1 为例,由于原始训练数据有 5 万张图片,调参时将有 4 万 5 千张图片用于训练(整理后存放在 input_dir/train)而另外 5 千张图片为验证集(整理后存放在 input_dir/valid)。

```
In [2]: import os
import shutil

def reorg_cifar10_data(data_dir, label_file, train_dir, test_dir, input_dir, valid_ratio):
    # 读取训练数据标签。
    with open(os.path.join(data_dir, label_file), 'r') as f:
        # 跳过文件头行(栏名称)。
        lines = f.readlines()[1:]
        tokens = [l.rstrip().split(',') for l in lines]
        idx_label = dict((int(idx), label) for idx, label in tokens)
    labels = set(idx_label.values())

    num_train = len(os.listdir(os.path.join(data_dir, train_dir)))
    num_train_tuning = int(num_train * (1 - valid_ratio))
    assert 0 < num_train_tuning < num_train
    num_train_tuning_per_label = num_train_tuning // len(labels)
    label_count = dict()

    def mkdir_if_not_exist(path):
```

```
if not os.path.exists(os.path.join(*path)):
    os.makedirs(os.path.join(*path))

# 整理训练和验证集。
for train_file in os.listdir(os.path.join(data_dir, train_dir)):
    idx = int(train_file.split('.')[0])
    label = idx_label[idx]
    mkdir_if_not_exist([data_dir, input_dir, 'train_valid', label])
    shutil.copy(os.path.join(data_dir, train_dir, train_file),
                os.path.join(data_dir, input_dir, 'train_valid', label))
    if label not in label_count or label_count[label] < num_train_tuning_per_label:
        mkdir_if_not_exist([data_dir, input_dir, 'train', label])
        shutil.copy(os.path.join(data_dir, train_dir, train_file),
                    os.path.join(data_dir, input_dir, 'train', label))
        label_count[label] = label_count.get(label, 0) + 1
    else:
        mkdir_if_not_exist([data_dir, input_dir, 'valid', label])
        shutil.copy(os.path.join(data_dir, train_dir, train_file),
                    os.path.join(data_dir, input_dir, 'valid', label))

# 整理测试集。
mkdir_if_not_exist([data_dir, input_dir, 'test', 'unknown'])
for test_file in os.listdir(os.path.join(data_dir, test_dir)):
    shutil.copy(os.path.join(data_dir, test_dir, test_file),
                os.path.join(data_dir, input_dir, 'test', 'unknown'))
```

再次强调，为了使网页编译快一点，我们在这里仅仅使用 100 个训练样本和 1 个测试样本。训练和测试数据的文件夹名称分别为' train_tiny' 和' test_tiny'。相应地，我们仅将批量大小设为 1。实际训练和测试时应使用 Kaggle 的完整数据集。由于数据集较大，批量大小 batch_size 大小可设为一个较大的整数，例如 128。

我们将 10% 的训练样本作为调参时的验证集。

In [3]: if demo:

```
# 注意：此处使用小训练集为便于网页编译。Kaggle 的完整数据集应包括 5 万训练样本。
train_dir = 'train_tiny'
# 注意：此处使用小测试集为便于网页编译。Kaggle 的完整数据集应包括 30 万测试样本。
test_dir = 'test_tiny'
# 注意：此处相应使用小批量。对 Kaggle 的完整数据集可设较大的整数，例如 128。
batch_size = 1
else:
    train_dir = 'train'
    test_dir = 'test'
```

```

batch_size = 128

data_dir = '../data/kaggle_cifar10'
label_file = 'trainLabels.csv'
input_dir = 'train_valid_test'
valid_ratio = 0.1
reorg_cifar10_data(data_dir, label_file, train_dir, test_dir, input_dir, valid_ratio)

```

9.7.3 使用 Gluon 读取整理后的数据集

为避免过拟合，我们在这里使用 `image.CreateAugmenter` 来增广数据集。例如我们设 `rand_mirror=True` 即可随机对每张图片做镜面反转。我们也通过 `mean` 和 `std` 对彩色图像 RGB 三个通道分别做标准化。以下我们列举了该函数里的所有参数，这些参数都是可以调的。

```

In [4]: from mxnet import autograd
        from mxnet import gluon
        from mxnet import image
        from mxnet import init
        from mxnet import nd
        from mxnet.gluon.data import vision
        import numpy as np

def transform_train(data, label):
    im = data.astype('float32') / 255
    auglist = image.CreateAugmenter(data_shape=(3, 32, 32), resize=0,
                                    rand_crop=False, rand_resize=False, rand_mirror=True,
                                    mean=np.array([0.4914, 0.4822, 0.4465]),
                                    std=np.array([0.2023, 0.1994, 0.2010]),
                                    brightness=0, contrast=0,
                                    saturation=0, hue=0,
                                    pca_noise=0, rand_gray=0, inter_method=2)
    for aug in auglist:
        im = aug(im)
    # 将数据格式从"高 * 宽 * 通道" 改为"通道 * 高 * 宽"。
    im = nd.transpose(im, (2, 0, 1))
    return (im, nd.array([label]).asscalar().astype('float32'))

# 测试时，无需对图像做标准化以外的增强数据处理。
def transform_test(data, label):
    im = data.astype('float32') / 255
    auglist = image.CreateAugmenter(data_shape=(3, 32, 32),

```

```
        mean=np.array([0.4914, 0.4822, 0.4465]),
        std=np.array([0.2023, 0.1994, 0.2010]))
    for aug in auglist:
        im = aug(im)
    im = nd.transpose(im, (2,0,1))
    return (im, nd.array([label]).asscalar().astype('float32'))
```

接下来，我们可以使用 Gluon 中的 `ImageFolderDataset` 类来读取整理后的数据集。

```
In [5]: input_str = data_dir + '/' + input_dir + '/'

# 读取原始图像文件。flag=1 说明输入图像有三个通道（彩色）。
train_ds = vision.ImageFolderDataset(input_str + 'train', flag=1,
                                      transform=transform_train)
valid_ds = vision.ImageFolderDataset(input_str + 'valid', flag=1,
                                      transform=transform_test)
train_valid_ds = vision.ImageFolderDataset(input_str + 'train_valid',
                                           flag=1, transform=transform_train)
test_ds = vision.ImageFolderDataset(input_str + 'test', flag=1,
                                     transform=transform_test)

loader = gluon.data.DataLoader
train_data = loader(train_ds, batch_size, shuffle=True, last_batch='keep')
valid_data = loader(valid_ds, batch_size, shuffle=True, last_batch='keep')
train_valid_data = loader(train_valid_ds, batch_size, shuffle=True, last_batch='ke
test_data = loader(test_ds, batch_size, shuffle=False, last_batch='keep')

# 交叉熵损失函数。
softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

9.7.4 设计模型

我们这里使用了ResNet-18模型。我们使用*hybridizing*来提升执行效率。

请注意：模型可以重新设计，参数也可以重新调整。

```
In [6]: from mxnet.gluon import nn
from mxnet import nd

class Residual(nn.HybridBlock):
    def __init__(self, channels, same_shape=True, **kwargs):
        super(Residual, self).__init__(**kwargs)
        self.same_shape = same_shape
```

```

    with self.name_scope():
        strides = 1 if same_shape else 2
        self.conv1 = nn.Conv2D(channels, kernel_size=3, padding=1,
                             strides=strides)
        self.bn1 = nn.BatchNorm()
        self.conv2 = nn.Conv2D(channels, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm()
        if not same_shape:
            self.conv3 = nn.Conv2D(channels, kernel_size=1,
                             strides=strides)

    def hybrid_forward(self, F, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        if not self.same_shape:
            x = self.conv3(x)
        return F.relu(out + x)

class ResNet(nn.HybridBlock):
    def __init__(self, num_classes, verbose=False, **kwargs):
        super(ResNet, self).__init__(**kwargs)
        self.verbose = verbose
        with self.name_scope():
            net = self.net = nn.HybridSequential()
            # 模块 1
            net.add(nn.Conv2D(channels=32, kernel_size=3, strides=1, padding=1))
            net.add(nn.BatchNorm())
            net.add(nn.Activation(activation='relu'))
            # 模块 2
            for _ in range(3):
                net.add(Residual(channels=32))
            # 模块 3
            net.add(Residual(channels=64, same_shape=False))
            for _ in range(2):
                net.add(Residual(channels=64))
            # 模块 4
            net.add(Residual(channels=128, same_shape=False))
            for _ in range(2):
                net.add(Residual(channels=128))
            # 模块 5

```

```
net.add(nn.AvgPool2D(pool_size=8))
net.add(nn.Flatten())
net.add(nn.Dense(num_classes))

def hybrid_forward(self, F, x):
    out = x
    for i, b in enumerate(self.net):
        out = b(out)
        if self.verbose:
            print('Block %d output: %s'%(i+1, out.shape))
    return out

def get_net(ctx):
    num_outputs = 10
    net = ResNet(num_outputs)
    net.initialize(ctx=ctx, init=init.Xavier())
    return net
```

9.7.5 训练模型并调参

在过拟合中我们讲过，过度依赖训练数据集的误差来推断测试数据集的误差容易导致过拟合。由于图像分类训练时间可能较长，为了方便，我们这里不再使用 K 折交叉验证，而是依赖验证集的结果来调参。

我们定义模型训练函数。这里我们记录每个 epoch 的训练时间。这有助于我们比较不同模型设计的时间成本。

```
In [7]: import datetime
import sys
sys.path.append('..')
import utils

def train(net, train_data, valid_data, num_epochs, lr, wd, ctx, lr_period, lr_decay):
    trainer = gluon.Trainer(
        net.collect_params(), 'sgd', {'learning_rate': lr, 'momentum': 0.9, 'wd': wd})

    prev_time = datetime.datetime.now()
    for epoch in range(num_epochs):
        train_loss = 0.0
        train_acc = 0.0
        if epoch > 0 and epoch % lr_period == 0:
```

```

    trainer.set_learning_rate(trainer.learning_rate * lr_decay)
    for data, label in train_data:
        label = label.as_in_context(ctx)
        with autograd.record():
            output = net(data.as_in_context(ctx))
            loss = softmax_cross_entropy(output, label)
            loss.backward()
        trainer.step(batch_size)
        train_loss += nd.mean(loss).asscalar()
        train_acc += utils.accuracy(output, label)
    cur_time = datetime.datetime.now()
    h, remainder = divmod((cur_time - prev_time).seconds, 3600)
    m, s = divmod(remainder, 60)
    time_str = "Time %02d:%02d:%02d" % (h, m, s)
    if valid_data is not None:
        valid_acc = utils.evaluate_accuracy(valid_data, net, ctx)
        epoch_str = ("Epoch %d. Loss: %f, Train acc %f, Valid acc %f, "
                     % (epoch, train_loss / len(train_data),
                        train_acc / len(train_data), valid_acc))
    else:
        epoch_str = ("Epoch %d. Loss: %f, Train acc %f, "
                     % (epoch, train_loss / len(train_data),
                        train_acc / len(train_data)))
    prev_time = cur_time
    print(epoch_str + time_str + ', lr ' + str(trainer.learning_rate))

```

以下定义训练参数并训练模型。这些参数均可调。为了使网页编译快一点，我们这里将 epoch 数量有意设为 1。事实上，epoch 一般可以调大些，例如 100。

我们将依据验证集的结果不断优化模型设计和调整参数。依据下面的参数设置，优化算法的学习率将在每 80 个 epoch 自乘 0.1。

```
In [8]: ctx = utils.try_gpu()
num_epochs = 1
learning_rate = 0.1
weight_decay = 5e-4
lr_period = 80
lr_decay = 0.1

net = get_net(ctx)
net.hybridize()
train(net, train_data, valid_data, num_epochs, learning_rate,
      weight_decay, ctx, lr_period, lr_decay)
```

Epoch 0. Loss: 3.498740, Train acc 0.100000, Valid acc 0.000000, Time 00:00:01, lr 0.1

9.7.6 对测试集分类

当得到一组满意的模型设计和参数后，我们使用全部训练数据集（含验证集）重新训练模型，并对测试集分类。

```
In [9]: import numpy as np
        import pandas as pd

        net = get_net(ctx)
        net.hybridize()
        train(net, train_valid_data, None, num_epochs, learning_rate,
              weight_decay, ctx, lr_period, lr_decay)

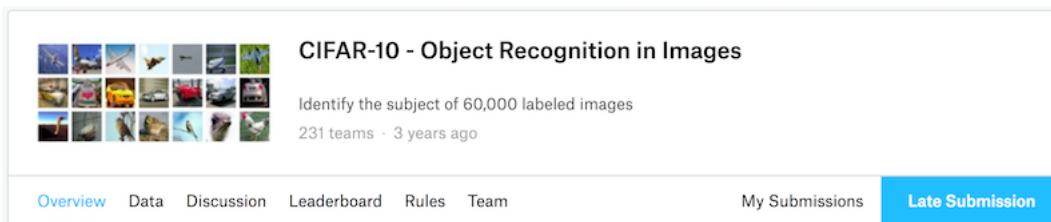
        preds = []
        for data, label in test_data:
            output = net(data.as_in_context(ctx))
            preds.extend(output.argmax(axis=1).astype(int).asnumpy())

        sorted_ids = list(range(1, len(test_ds) + 1))
        sorted_ids.sort(key = lambda x:str(x))

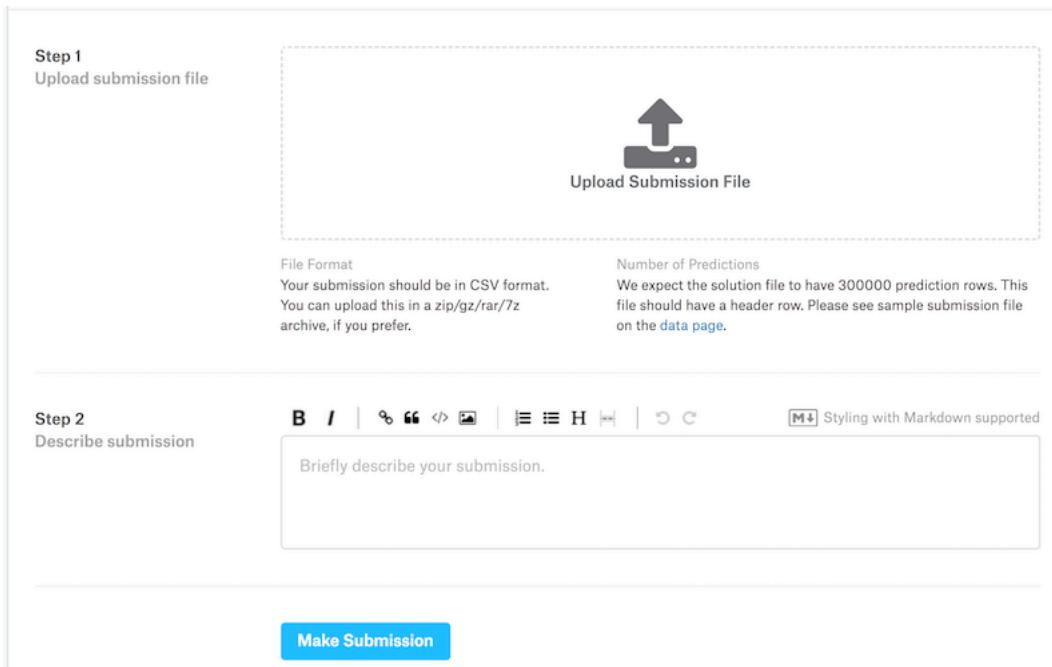
        df = pd.DataFrame({'id': sorted_ids, 'label': preds})
        df['label'] = df['label'].apply(lambda x: train_valid_ds.synsets[x])
        df.to_csv('submission.csv', index=False)
```

Epoch 0. Loss: 3.485362, Train acc 0.060000, Time 00:00:01, lr 0.1

上述代码执行完会生成一个 `submission.csv` 的文件用于在 Kaggle 上提交。这是 Kaggle 要求的提交格式。这时我们可以在 Kaggle 上把对测试集分类的结果提交并查看分类准确率。你需要登录 Kaggle 网站，打开CIFAR-10 原始图像分类问题，并点击下方右侧 Late Submission 按钮。



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



9.7.7 作业（汇报作业和查看其他小伙伴作业）：

- 使用 Kaggle 完整 CIFAR-10 数据集，把 batch_size 和 num_epochs 分别改为 128 和 100，可以在 Kaggle 上拿到什么样的准确率和名次？
- 如果不使用增强数据的方法能拿到什么样的准确率？
- 你还有什么其他办法可以继续改进模型和参数？小伙伴们都期待你的分享。

吐槽和讨论欢迎点[这里](#)

9.8 实战 Kaggle 比赛——使用 Gluon 识别 120 种狗 (ImageNet Dogs)

我们在本章中选择了 Kaggle 中的120 种狗类识别问题。这是著名的 ImageNet 的子集数据集。与之前的CIFAR-10 原始图像分类问题不同，本问题中的图片文件大小更接近真实照片大小，且大小不一。本问题的输出也变的更加通用：我们将输出每张图片对应 120 种狗的分别概率。

9.8.1 Kaggle 中的 CIFAR-10 原始图像分类问题

Kaggle是一个著名的供机器学习爱好者交流的平台。为了便于提交结果，请大家注册Kaggle账号。然后请大家先点击[120 种狗类识别问题](#)了解有关本次比赛的信息。

Playground Prediction Competition

Dog Breed Identification

Determine the breed of a dog in an image

Kaggle · 176 teams · 4 months to go

Overview Data Kernels Discussion Leaderboard Rules Team My Submissions Submit Predictions

Overview

Description Who's a good dog? Who likes ear scratches? Well, it seems those fancy deep neural networks don't have *all* the answers. However, maybe they can answer that ubiquitous question we all ask when meeting a four-legged stranger: what kind of good pup is that?

Evaluation In this playground competition, you are provided a strictly canine subset of [ImageNet](#) in order to practice fine-grained image categorization. How well you can tell your Norfolk Terriers from your Norwich Terriers? With 120 breeds of dogs and a limited number training images per class, you might find the problem more, err, ruff than you anticipated.



9.8.2 整理原始数据集

比赛数据分为训练数据集和测试数据集。训练集包含 10,222 张图片。测试集包含 10,357 张图片。

两个数据集都是 jpg 彩色图片，大小接近真实照片大小，且大小不一。训练集一共有 120 类狗的图片。

下载数据集

登录 Kaggle 后，数据可以从 120 种狗类识别问题中下载。

- 训练数据集 train.zip 下载地址
- 测试数据集 test.zip 下载地址
- 训练数据标签 label.csv.zip 下载地址

解压数据集

训练数据集 train.zip 和测试数据集 test.zip 都是压缩格式，下载后它们的路径可以如下：

-/data/kaggle_dog/train.zip
-/data/kaggle_dog/test.zip
-/data/kaggle_dog/labels.csv.zip

为了使网页编译快一点，我们在 git repo 里仅仅存放小数据样本 ('train_valid_test_tiny.zip')。执行以下代码会从 git repo 里解压生成小数据样本。

```
In [1]: # 如果训练下载的 Kaggle 的完整数据集，把 demo 改为 False。
demo = True
data_dir = '../data/kaggle_dog'

if demo:
    zipfiles= ['train_valid_test_tiny.zip']
else:
    zipfiles= ['train.zip', 'test.zip', 'labels.csv.zip']

import zipfile
for fin in zipfiles:
    with zipfile.ZipFile(data_dir + '/' + fin, 'r') as zin:
        zin.extractall(data_dir)
```

整理数据集

对于 Kaggle 的完整数据集，我们需要定义下面的 reorg_dog_data 函数来整理一下。整理后，同一类狗的图片将出现在在同一个文件夹下，便于 Gluon 稍后读取。

函数中的参数如 data_dir、train_dir 和 test_dir 对应上述数据存放路径及原始训练和测试的图片集文件夹名称。参数 label_file 为训练数据标签的文件名称。参数 input_dir 是整理后数据集文件夹名称。参数 valid_ratio 是验证集中每类狗的数量占原始训练集中数量最少一类的狗的数量(66) 的比重。

```
In [2]: import math
        import os
        import shutil
        from collections import Counter

        def reorg_dog_data(data_dir, label_file, train_dir, test_dir, input_dir,
                           valid_ratio):
            # 读取训练数据标签。
            with open(os.path.join(data_dir, label_file), 'r') as f:
                # 跳过文件头行（栏名称）。
                lines = f.readlines()[1:]
                tokens = [l.rstrip().split(',') for l in lines]
                idx_label = dict(((idx, label) for idx, label in tokens))
            labels = set(idx_label.values())

            num_train = len(os.listdir(os.path.join(data_dir, train_dir)))
            # 训练集中数量最少一类的狗的数量。
            min_num_train_per_label = (
                Counter(idx_label.values()).most_common()[:-2:-1][0][1])
            # 验证集中每类狗的数量。
            num_valid_per_label = math.floor(min_num_train_per_label * valid_ratio)
            label_count = dict()

            def mkdir_if_not_exist(path):
                if not os.path.exists(os.path.join(*path)):
                    os.makedirs(os.path.join(*path))

            # 整理训练和验证集。
            for train_file in os.listdir(os.path.join(data_dir, train_dir)):
                idx = train_file.split('.')[0]
                label = idx_label[idx]
                mkdir_if_not_exist([data_dir, input_dir, 'train_valid', label])
```

```

shutil.copy(os.path.join(data_dir, train_dir, train_file),
            os.path.join(data_dir, input_dir, 'train_valid', label))
if label not in label_count or label_count[label] < num_valid_per_label:
    mkdir_if_not_exist([data_dir, input_dir, 'valid', label])
    shutil.copy(os.path.join(data_dir, train_dir, train_file),
                os.path.join(data_dir, input_dir, 'valid', label))
    label_count[label] = label_count.get(label, 0) + 1
else:
    mkdir_if_not_exist([data_dir, input_dir, 'train', label])
    shutil.copy(os.path.join(data_dir, train_dir, train_file),
                os.path.join(data_dir, input_dir, 'train', label))

# 整理测试集。
mkdir_if_not_exist([data_dir, input_dir, 'test', 'unknown'])
for test_file in os.listdir(os.path.join(data_dir, test_dir)):
    shutil.copy(os.path.join(data_dir, test_dir, test_file),
                os.path.join(data_dir, input_dir, 'test', 'unknown'))

```

再次强调，为了使网页编译快一点，我们在这里仅仅使用小数据样本。相应地，我们仅将批量大小设为 2。实际训练和测试时应使用 Kaggle 的完整数据集并调用 reorg_dog_data 函数整理便于 Gluon 读取的格式。由于数据集较大，批量大小 batch_size 大小可设为一个较大的整数，例如 128。

In [3]: if demo:

```

# 注意：此处使用小数据集为便于网页编译。
input_dir = 'train_valid_test_tiny'
# 注意：此处相应使用小批量。对 Kaggle 的完整数据集可设较大的整数，例如 128。
batch_size = 2
else:
    label_file = 'labels.csv'
    train_dir = 'train'
    test_dir = 'test'
    input_dir = 'train_valid_test'
    batch_size = 128
    valid_ratio = 0.1
    reorg_dog_data(data_dir, label_file, train_dir, test_dir, input_dir,
                   valid_ratio)

```

9.8.3 使用 Gluon 读取整理后的数据集

为避免过拟合，我们在这里使用 `image.CreateAugmenter` 来增广数据集。例如我们设 `rand_mirror=True` 即可随机对每张图片做镜面反转。以下我们列举了该函数里的所有参数，这些参数都是可以调的。

```
In [4]: from mxnet import autograd
        from mxnet import gluon
        from mxnet import image
        from mxnet import init
        from mxnet import nd
        from mxnet.gluon.data import vision
        import numpy as np

def transform_train(data, label):
    im = image.imresize(data.astype('float32') / 255, 96, 96)
    auglist = image.CreateAugmenter(data_shape=(3, 96, 96), resize=0,
                                    rand_crop=False, rand_resize=False, rand_mirror=True,
                                    mean=None, std=None,
                                    brightness=0, contrast=0,
                                    saturation=0, hue=0,
                                    pca_noise=0, rand_gray=0, inter_method=2)
    for aug in auglist:
        im = aug(im)
    # 将数据格式从"高 * 宽 * 通道" 改为"通道 * 高 * 宽"。
    im = nd.transpose(im, (2, 0, 1))
    return (im, nd.array([label]).asscalar().astype('float32'))

def transform_test(data, label):
    im = image.imresize(data.astype('float32') / 255, 96, 96)
    im = nd.transpose(im, (2, 0, 1))
    return (im, nd.array([label]).asscalar().astype('float32'))
```

接下来，我们可以使用 Gluon 中的 `ImageFolderDataset` 类来读取整理后的数据集。

```
In [5]: input_str = data_dir + '/' + input_dir + '/'

# 读取原始图像文件。flag=1 说明输入图像有三个通道（彩色）。
train_ds = vision.ImageFolderDataset(input_str + 'train', flag=1,
                                      transform=transform_train)
valid_ds = vision.ImageFolderDataset(input_str + 'valid', flag=1,
                                      transform=transform_test)
train_valid_ds = vision.ImageFolderDataset(input_str + 'train_valid',
```

```

        flag=1, transform=transform_train)
test_ds = vision.ImageFolderDataset(input_str + 'test', flag=1,
                                    transform=transform_test)

loader = gluon.data.DataLoader
train_data = loader(train_ds, batch_size, shuffle=True, last_batch='keep')
valid_data = loader(valid_ds, batch_size, shuffle=True, last_batch='keep')
train_valid_data = loader(train_valid_ds, batch_size, shuffle=True,
                           last_batch='keep')
test_data = loader(test_ds, batch_size, shuffle=False, last_batch='keep')

# 交叉熵损失函数。
softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()

```

9.8.4 设计模型

我们这里使用了ResNet-18模型。我们使用*hybridizing*来提升执行效率。

请注意：模型可以重新设计，参数也可以重新调整。

```

In [6]: from mxnet.gluon import nn
         from mxnet import nd

class Residual(nn.HybridBlock):
    def __init__(self, channels, same_shape=True, **kwargs):
        super(Residual, self).__init__(**kwargs)
        self.same_shape = same_shape
        with self.name_scope():
            strides = 1 if same_shape else 2
            self.conv1 = nn.Conv2D(channels, kernel_size=3, padding=1,
                                 strides=strides)
            self.bn1 = nn.BatchNorm()
            self.conv2 = nn.Conv2D(channels, kernel_size=3, padding=1)
            self.bn2 = nn.BatchNorm()
            if not same_shape:
                self.conv3 = nn.Conv2D(channels, kernel_size=1,
                                 strides=strides)

    def hybrid_forward(self, F, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        if not self.same_shape:

```

```
x = self.conv3(x)
return F.relu(out + x)

class ResNet(nn.HybridBlock):
    def __init__(self, num_classes, verbose=False, **kwargs):
        super(ResNet, self).__init__(**kwargs)
        self.verbose = verbose
        with self.name_scope():
            net = self.net = nn.HybridSequential()
            # 模块 1
            net.add(nn.Conv2D(channels=32, kernel_size=3, strides=1,
                             padding=1))
            net.add(nn.BatchNorm())
            net.add(nn.Activation(activation='relu'))
            # 模块 2
            for _ in range(3):
                net.add(Residual(channels=32))
            # 模块 3
            net.add(Residual(channels=64, same_shape=False))
            for _ in range(2):
                net.add(Residual(channels=64))
            # 模块 4
            net.add(Residual(channels=128, same_shape=False))
            for _ in range(2):
                net.add(Residual(channels=128))
            # 模块 5
            net.add(nn.GlobalAvgPool2D())
            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 = 120
net = ResNet(num_outputs)
net.initialize(ctx=ctx, init=init.Xavier())
return net

```

9.8.5 训练模型并调参

在过拟合中我们讲过，过度依赖训练数据集的误差来推断测试数据集的误差容易导致过拟合。由于图像分类训练时间可能较长，为了方便，我们这里不再使用 K 折交叉验证，而是依赖验证集的结果来调参。

我们定义损失函数以便于计算验证集上的损失函数值。我们也定义了模型训练函数，其中的优化算法和参数都是可以调的。

```

In [7]: import datetime
        import sys
        sys.path.append('..')
        import utils

        def get_loss(data, net, ctx):
            loss = 0.0
            for feas, label in data:
                label = label.as_in_context(ctx)
                output = net(feas.as_in_context(ctx))
                cross_entropy = softmax_cross_entropy(output, label)
                loss += nd.mean(cross_entropy).asscalar()
            return loss / len(data)

        def train(net, train_data, valid_data, num_epochs, lr, wd, ctx, lr_period,
                  lr_decay):
            trainer = gluon.Trainer(
                net.collect_params(), 'sgd', {'learning_rate': lr, 'momentum': 0.9,
                                              'wd': wd})
            prev_time = datetime.datetime.now()
            for epoch in range(num_epochs):
                train_loss = 0.0
                if epoch > 0 and epoch % lr_period == 0:
                    trainer.set_learning_rate(trainer.learning_rate * lr_decay)
                for data, label in train_data:
                    label = label.as_in_context(ctx)
                    with autograd.record():
                        output = net(data.as_in_context(ctx))

```

```
        loss = softmax_cross_entropy(output, label)
        loss.backward()
        trainer.step(batch_size)
        train_loss += nd.mean(loss).asscalar()
        cur_time = datetime.datetime.now()
        h, remainder = divmod((cur_time - prev_time).seconds, 3600)
        m, s = divmod(remainder, 60)
        time_str = "Time %02d:%02d:%02d" % (h, m, s)
        if valid_data is not None:
            valid_loss = get_loss(valid_data, net, ctx)
            epoch_str = ("Epoch %d. Train loss: %f, Valid loss %f, "
                         % (epoch, train_loss / len(train_data), valid_loss))
        else:
            epoch_str = ("Epoch %d. Train loss: %f, "
                         % (epoch, train_loss / len(train_data)))
        prev_time = cur_time
        print(epoch_str + time_str + ', lr ' + str(trainer.learning_rate))
```

以下定义训练参数并训练模型。这些参数均可调。为了使网页编译快一点，我们这里将 epoch 数量有意设为 1。事实上，epoch 一般可以调大些。

我们将依据验证集的结果不断优化模型设计和调整参数。依据下面的参数设置，优化算法的学习率将在每 80 个 epoch 自乘 0.1。

```
In [8]: ctx = utils.try_gpu()
num_epochs = 1
learning_rate = 0.01
weight_decay = 5e-4
lr_period = 80
lr_decay = 0.1

net = get_net(ctx)
net.hybridize()
train(net, train_data, valid_data, num_epochs, learning_rate,
      weight_decay, ctx, lr_period, lr_decay)

Epoch 0. Train loss: 5.521051, Valid loss 6.690127, Time 00:00:02, lr 0.01
```

9.8.6 对测试集分类

当得到一组满意的模型设计和参数后，我们使用全部训练数据集（含验证集）重新训练模型，并对测试集分类。

In [9]: `import numpy as np`

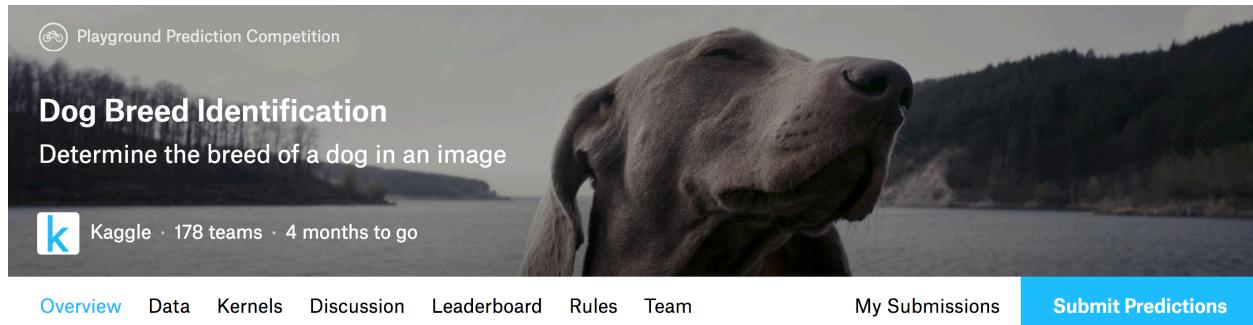
```
net = get_net(ctx)
net.hybridize()
train(net, train_valid_data, None, num_epochs, learning_rate, weight_decay,
      ctx, lr_period, lr_decay)

outputs = []
for data, label in test_data:
    output = nd.softmax(net(data.as_in_context(ctx)))
    outputs.extend(output.asnumpy())
ids = sorted(os.listdir(os.path.join(data_dir, input_dir, 'test/unknown')))

with open('submission.csv', 'w') as f:
    f.write('id,' + ','.join(train_valid_ds.synsets) + '\n')
    for i, output in zip(ids, outputs):
        f.write(i.split('.')[0] + ',' + ','.join(
            [str(num) for num in output]) + '\n')
```

Epoch 0. Train loss: 5.203509, Time 00:00:04, lr 0.01

上述代码执行完会生成一个 `submission.csv` 的文件用于在 Kaggle 上提交。这是 Kaggle 要求的提交格式。这时我们可以在 Kaggle 上把对测试集分类的结果提交并查看分类准确率。你需要登录 Kaggle 网站，打开120 种狗类识别问题，并点击下方右侧 `Submit Predictions` 按钮。



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

温馨提醒，目前 **Kaggle** 仅限每个账号一天以内 5 次提交结果的机会。所以提交结果前务必三思。

9.8.7 作业（汇报作业和查看其他小伙伴作业）：

- 使用 Kaggle 完整数据集，把 `batch_size` 和 `num_epochs` 分别调大些，可以在 Kaggle 上拿到什么样的准确率和名次？

Step 1

Upload submission file



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 10357 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  |  C
 Styling with Markdown supported

Briefly describe your submission.

Make Submission

- 你还有什么其他办法可以继续改进模型和参数？小伙伴们都期待你的分享。

吐槽和讨论欢迎点[这里](#)

我们将持续的加入新的内容。如果想提前了解，可以参见 [英文版本](#)（注意：中文版本根据社区的反馈做了比较大的更改，我们还在努力的将改动同步到英文版）