



# 使用 MXNet/Gluon 来动手学深度学习

*Release 0.3*

**MXNet Community**

Sep 22, 2017



# CONTENTS

<b>1 前言</b>	<b>3</b>
1.1 为什么要做这个项目 . . . . .	3
1.2 前言 . . . . .	6
1.3 安裝和使用 . . . . .	8
1.4 GPU 购买指南 . . . . .	16
<b>2 预备知识</b>	<b>21</b>
2.1 机器学习简介 . . . . .	21
2.2 使用 NDArray 来处理数据 . . . . .	31
2.3 使用 autograd 来自动求导 . . . . .	35
<b>3 监督学习</b>	<b>39</b>
3.1 线性回归—从 0 开始 . . . . .	39
3.2 线性回归—使用 Gluon . . . . .	43
3.3 多类逻辑回归—从 0 开始 . . . . .	47
3.4 多类逻辑回归—使用 Gluon . . . . .	53
3.5 多层感知机—从 0 开始 . . . . .	55
3.6 多层感知机—使用 Gluon . . . . .	59
3.7 欠拟合和过拟合 . . . . .	60
3.8 正则化—从 0 开始 . . . . .	70
3.9 正则化—使用 Gluon . . . . .	75
3.10 丢弃法—从 0 开始 . . . . .	79
3.11 丢弃法—使用 Gluon . . . . .	84
3.12 实战 Kaggle 比赛——使用 Gluon 预测房价和 K 折交叉验证 . . . . .	86
<b>4 Gluon 基础</b>	<b>101</b>
4.1 创建神经网络 . . . . .	101
4.2 初始化模型参数 . . . . .	106

4.3 序列化—读写模型 . . . . .	112
4.4 设计自定义层 . . . . .	114
4.5 使用 GPU 来计算 . . . . .	118
<b>5 卷积神经网络</b>	<b>125</b>
5.1 卷积神经网络—从 0 开始 . . . . .	125
5.2 卷积神经网络—使用 Gluon . . . . .	134
5.3 批量归一化—从 0 开始 . . . . .	136
5.4 批量归一化—使用 Gluon . . . . .	143
5.5 深度卷积神经网络和 AlexNet . . . . .	145
5.6 VGG: 使用重复元素的非常深的网络 . . . . .	151
<b>6 循环神经网络</b>	<b>155</b>
6.1 循环神经网络—从 0 开始 . . . . .	155

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。

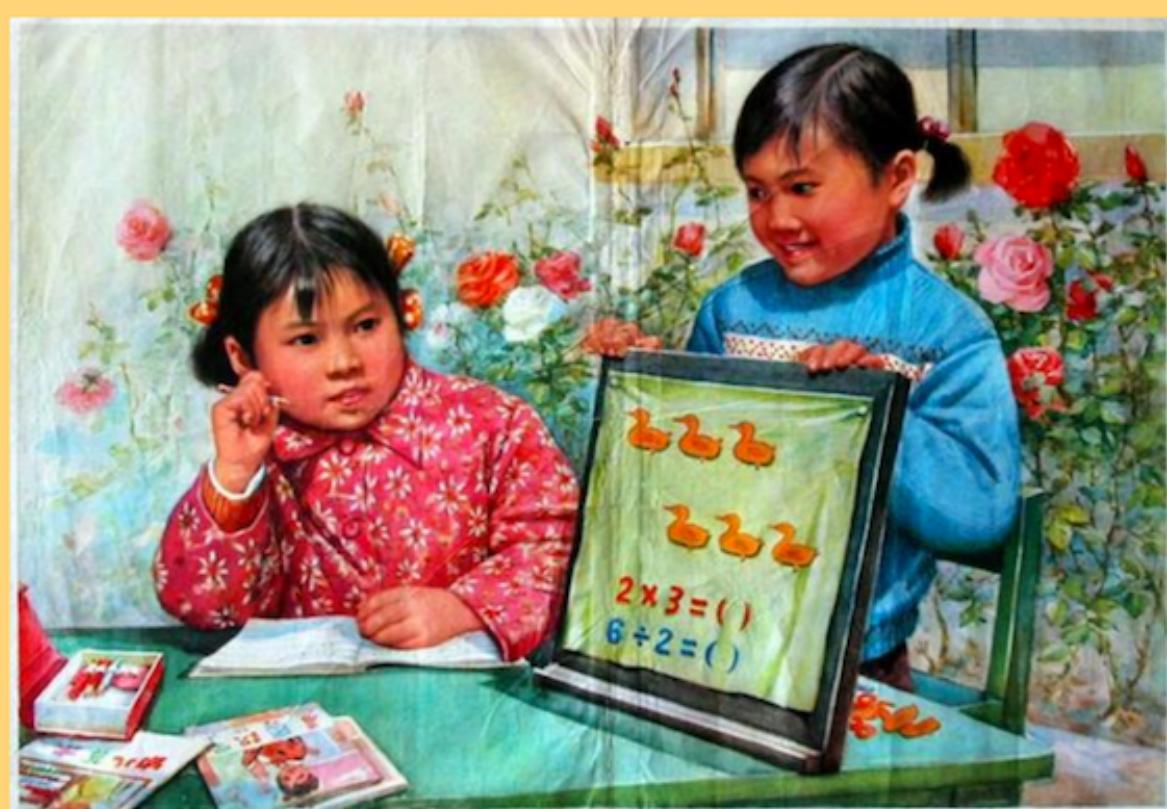
虽然在我认识的人里，好些人能够读一篇论文或者听一个报告后就能问出很好的问题，然后就基本  
弄懂了。但我在这个上笨很多。读过的论文就像喝过的水，第二天就不记得了。一定是需要静下心来，  
从头到尾实现一篇，跑上几个数据，调些参数，才能心安地觉得懂了。例如在港科大的两年读了  
很多论文，但现在反过来看，仍然记得可能就是那两个老老实实动手实现过写过论文的模型了。即  
使后来在机器学习这个方向又走了五年，学习任何新东西仍然是要靠动手。



## 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。



## 道理我都懂，但仍调不出这个参

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

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

为此我们做了（正在做）这五件事情：

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

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

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

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

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

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

@mli

## 1.2 前言

这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，但我们自认为我们理解这些模型是怎么运行的，以及为什么。而现在虽然机器学习系统非常强大，但留下一个巨大问题：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

我们坚信，学习深度学习的最好方式就是**动手实现深度学习模型**。

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

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

最后，英文中有句话叫做

“Get hands dirty.”

直译过来就是

“撸起袖子加油干。”

## 1.3 安装和使用

### 1.3.1 安装需求

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。

### 1.3.2 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 http://zh.gluon.ai/gluon_tutorials_zh.tar.gz -o tutorials.tar.gz
tar -xzvf tutorials.tar.gz && rm tutorials.tar.gz
```

Windows 用户可以用浏览器[下载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/
```

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
```

### 1.3.3 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。

### 1.3.4 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：这个只推荐给如果想上 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
```

## 1.3.5 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

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。

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

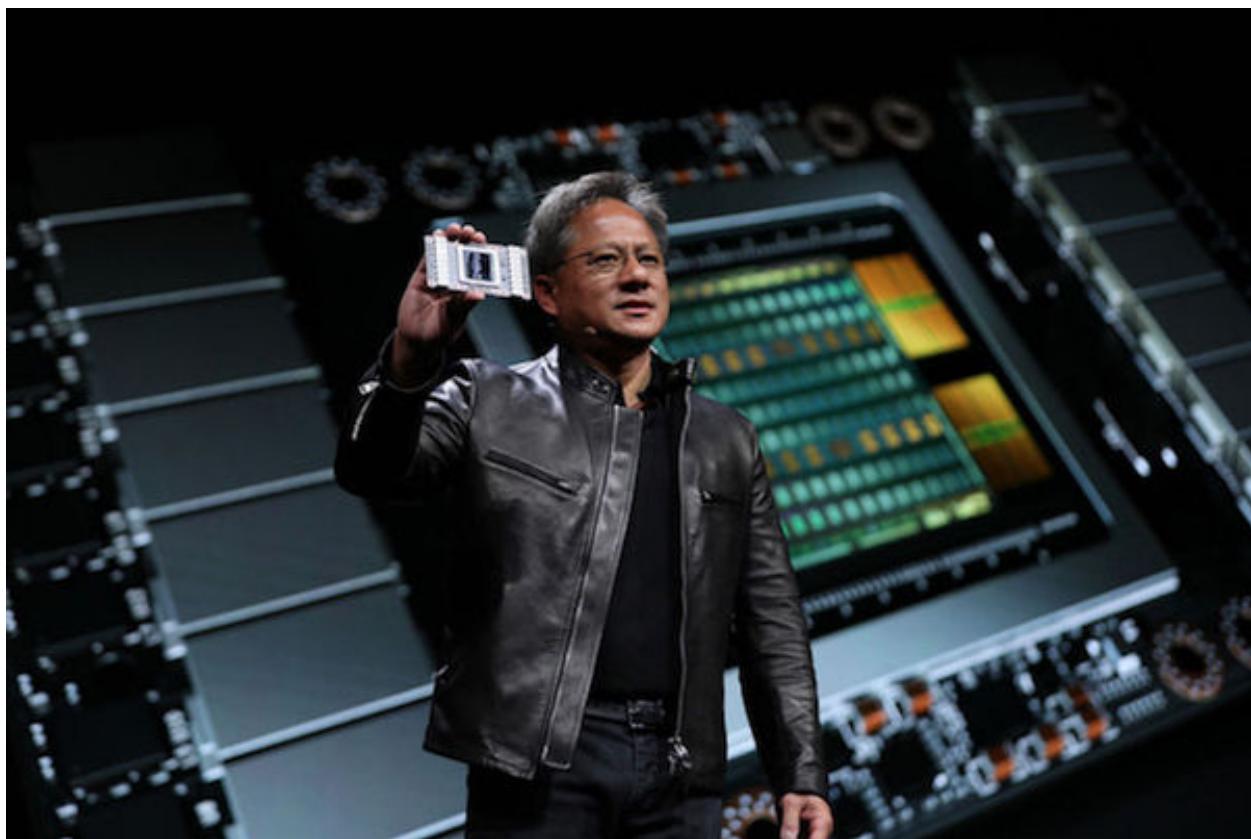
## 最后

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。

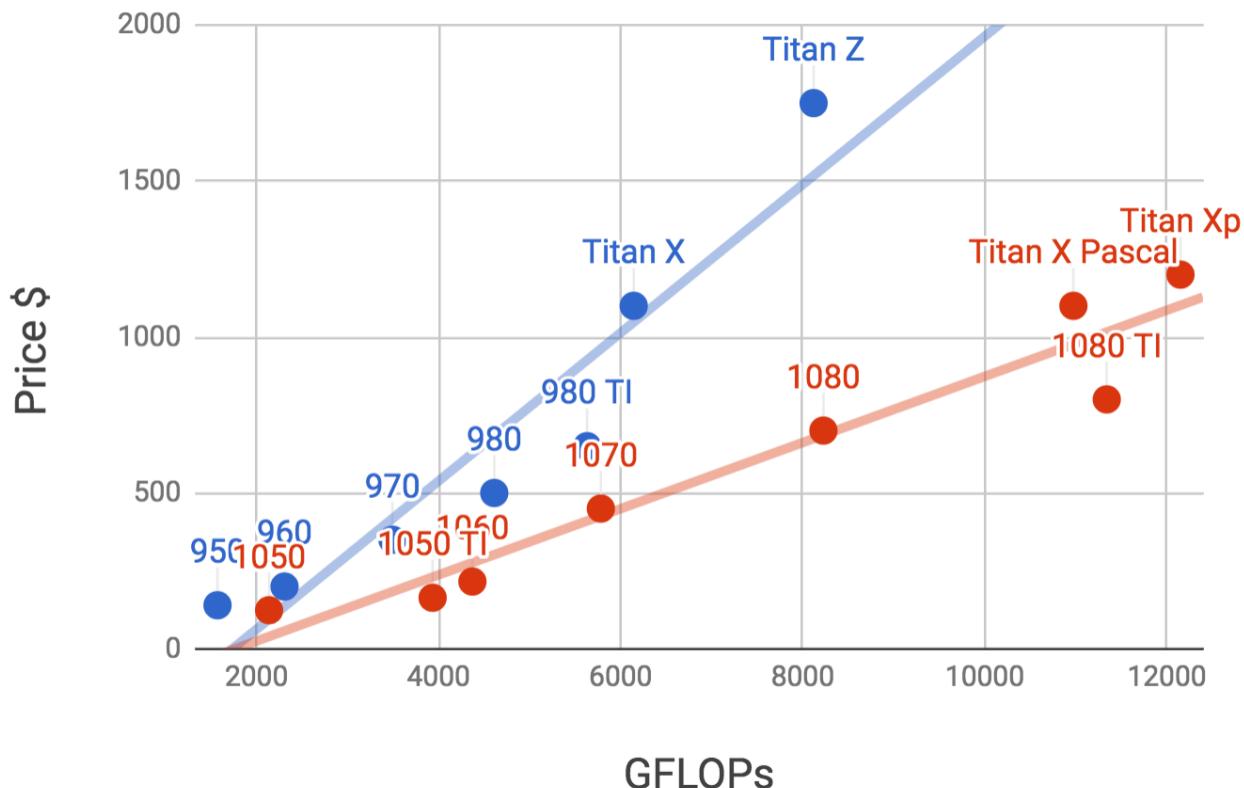
Nvidia 一般每一两年会更新一次大版本，例如目前最新的是 1000 系列。每个系列里面会有数个不同型号，对应不同的性能。

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](#)。

## 预备知识

### 2.1 机器学习简介

本书作者跟广大程序员一样，在开始写作前需要取来一杯咖啡。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”。手机这时马上显示出识别的命令，并且知道我们需要导航。接着它调出地图应用并给出数条路线方案，每条方案边上会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，这个不就是编程吗？或者，到底机器学习是什么？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。但不是每个程序都用了机器学习。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。

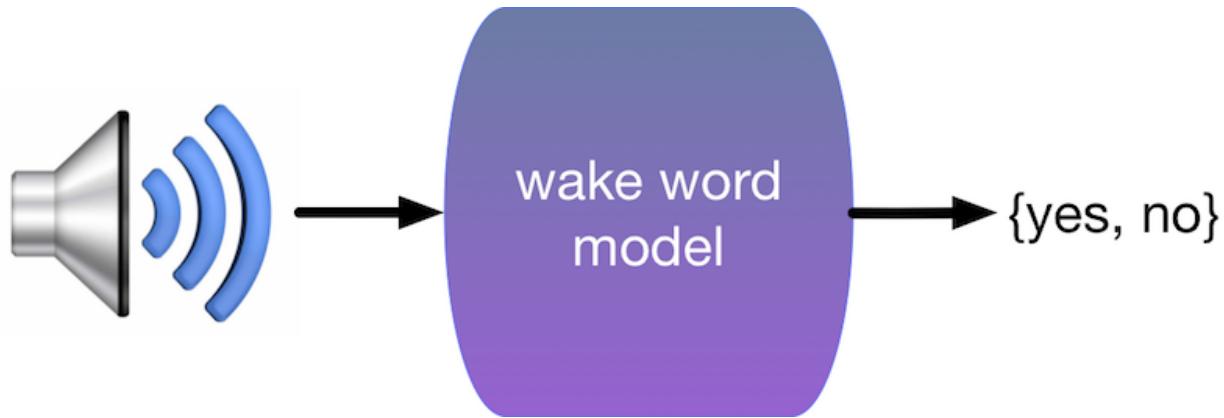
#### 2.1.1 一个例子

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来把一个商品 ID 和你的用户 ID 插入到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来用最基本的程序指令来实现这个功能。如果你发现你可以这么做，那么你就不应该使用机器学习。

对于机器学习科学家来说，幸运的是大部分应用没有那么容易。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”。如果你在一个只有你和代码编辑器的房间里写这个程序，你该怎么办？你可能会想像下面的程序

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

但实际上你能拿到的只是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。那么需要些什么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，判断这些信号里是不是就是说了唤醒词。



如果你被这个问题困住了，不用担心。这就是我们为什么要机器学习。

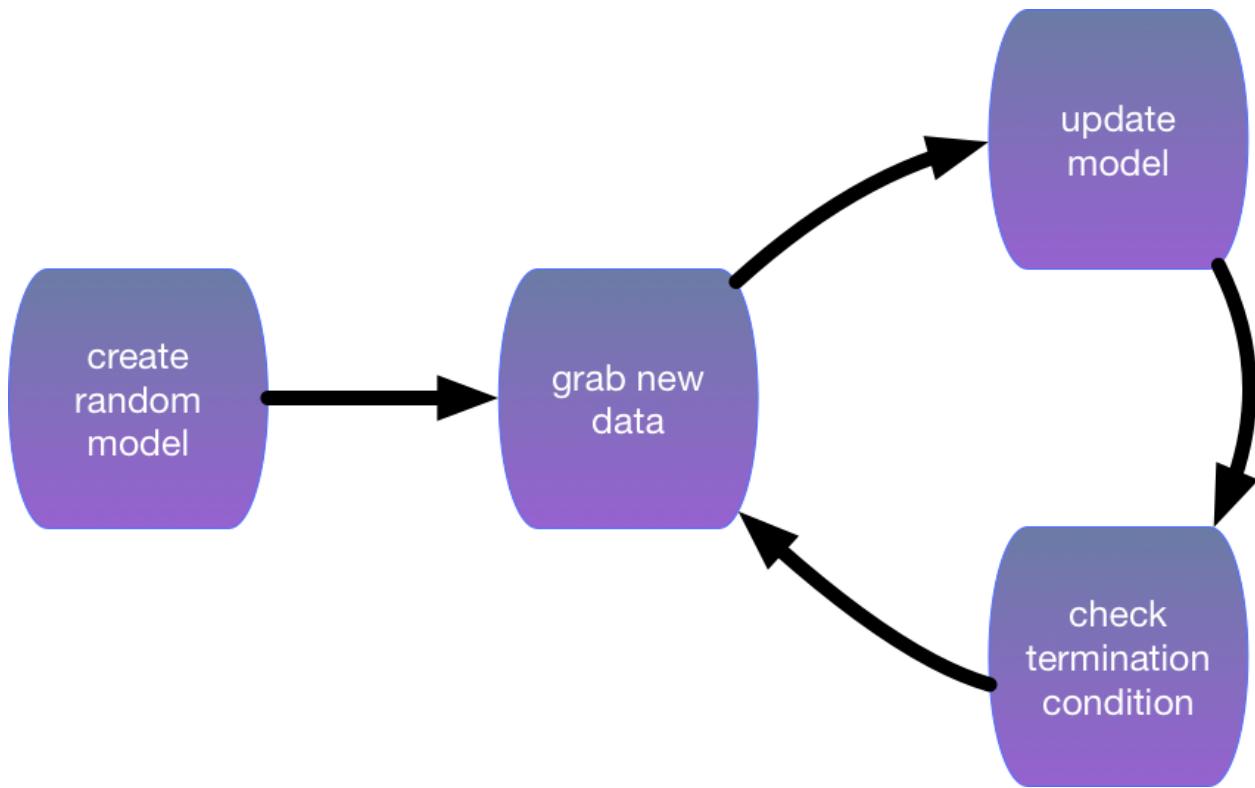
虽然我们不知道怎么告诉机器去把语音信号转成对应的字符串，但我们自己可以。我们可以收集一个巨大的**数据集**里包含了大量语音信号，以及每个语音型号是不是对应我们要的唤醒词。在机器学习里，我们不直接设计一个系统去辨别唤醒词，而是写一个灵活的程序，它的行为可以根据在读取数据集的时候改变。所以我们不是去直接写一个唤醒词辨别器，而是一个程序，当提供一个巨大的有标注的数据集的时候它能辨别唤醒词。你可以认为这种方式是**利用数据编程**。换言之，我们需要用数据训练机器学习模型，其过程通常如下：

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。

### 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，而不是有着固定行为的程序。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**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不能或者难以解决的问题，它被广泛应用在了搜索引擎、无人驾驶、机器翻译、医疗诊断、垃圾邮件过滤、玩游戏、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从而可以被机器学习模型解决。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  $X$   $[1, 2, 3, 4, \dots]$  和标注  $Y$   $['fizz', 'buzz', 'fizzbuzz', identity]$
- 训练数据，也就是系统输入输出的实例。例如  $[(2, 2), (6, fizz), (15, fizzbuzz), (23, 23), (40, buzz)]$
- 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，Joel 利用 TensorFlow 写了一个分类器。对于不按常理出牌的 Joel，面试官一脸黑线。而且这个分类器不是总是对的。

显然，用原子弹杀鸡了。为什么不直接写几行简单而且保证结果正确的 Python 代码呢？当然，这里有很多一个简单 Python 脚本不能分类的例子，即使简单的 3 岁小孩解决起来毫无压力。



幸运的是，这个正是机器学习的用武之地。我们通过提供大量的含有猫和狗的图片来编程一个猫狗检测器，一般来说它就是一个函数，它会输出一个大的正数如果图片里面是猫，一个大的负数如果是狗，如果不确信就输出一个 0 附近的。当然，这是机器学习能做的最简单例子。

### 2.1.4 机器学习最简要素

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。**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：例如你的手机图片，里面可能包含猫、狗、恐龙、高中同学聚会或者昨天的晚饭
- 文本：邮件、新闻和微信聊天记录
- 声音：有声书籍和电话记录
- 结构数据：Jupyter notebook（里面有文本、图片和代码）、网页、租车单和电费表
- **模型**。通常数据和我们最终想要的相差很远，例如我们知道照片中的人是不是在高兴，所以我们需要把一千万像素变成一个高兴度的概率值。通常我们需要在数据上应用数个非线性函数（例如神经网络）
- **损失函数**。我们需要对比模型的输出和真实值之间的误差。损失函数帮助我们决定 2017 年底亚马逊股票会不会价值 1500 美元。取决于我们想短线还是长线，这个函数可以很不一样。
- **训练**。通常一个模型里面有很多参数。我们通过最小化损失函数来学这些参数。不幸的是，即使我们做得很好也不能保证在新的没见过的数据上我们可以仍然做很好。
- **训练误差**。这是模型在评估用来训练模型的数据集上的误差。这个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**。这是模型在没见过的新数据上的误差，可能会跟训练误差不很一样（统计上叫过拟合）。这个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。）

下面我们详细讨论一些不同的机器学习应用。

### 2.1.5 监督学习

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  $P(y|x)$ 。在实际情景中，监督学习最为常用。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，预测出它的正确中文翻译；给定本月公司财报数据，预测下个月该公司股票价格。

#### 回归分析

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。

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

值得一提的是，回归分析所关注的预测往往可以解答输出为**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。

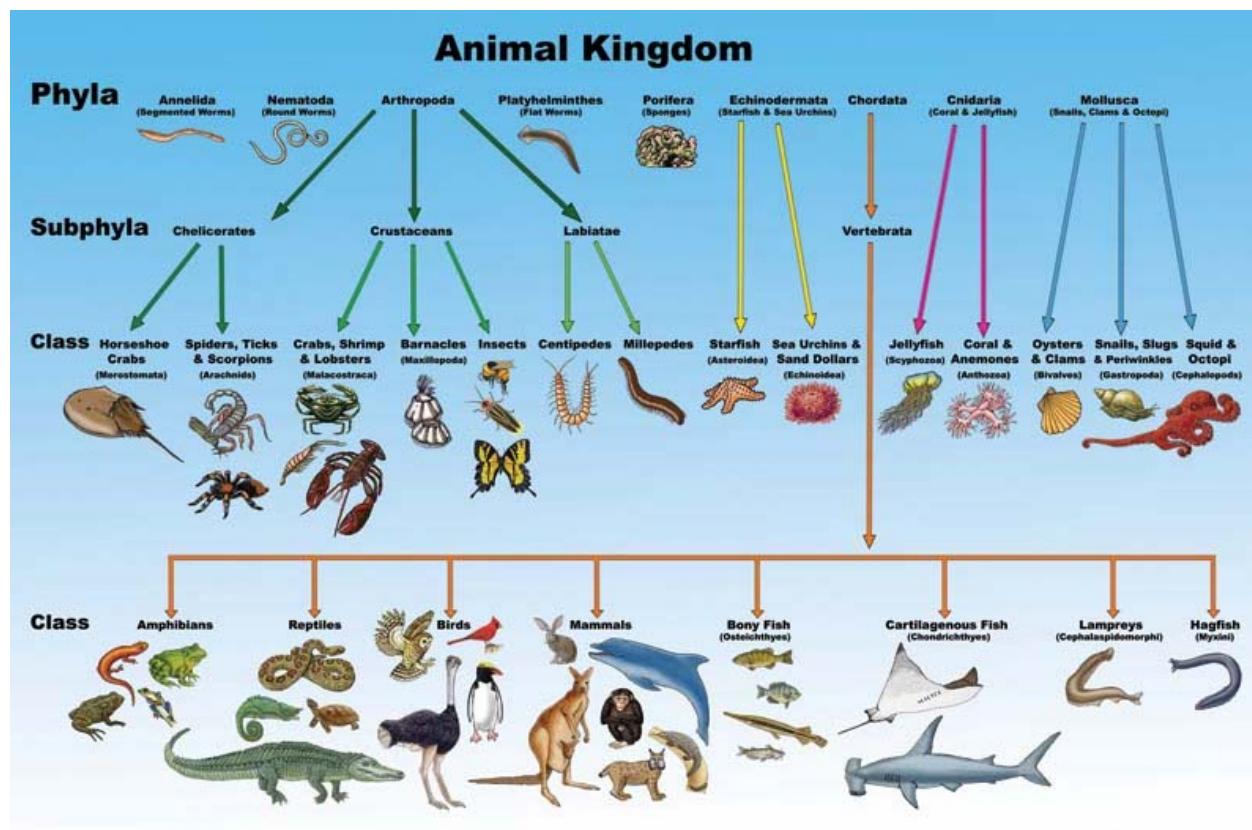


Fig. 2.1: 动物的分类

给定一个实例被抽取出的若干特征作为输入，我们的分类模型可以输出实例为各个类别的概率，并将概率最大的类别作为分类的结果。

## 标注

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。

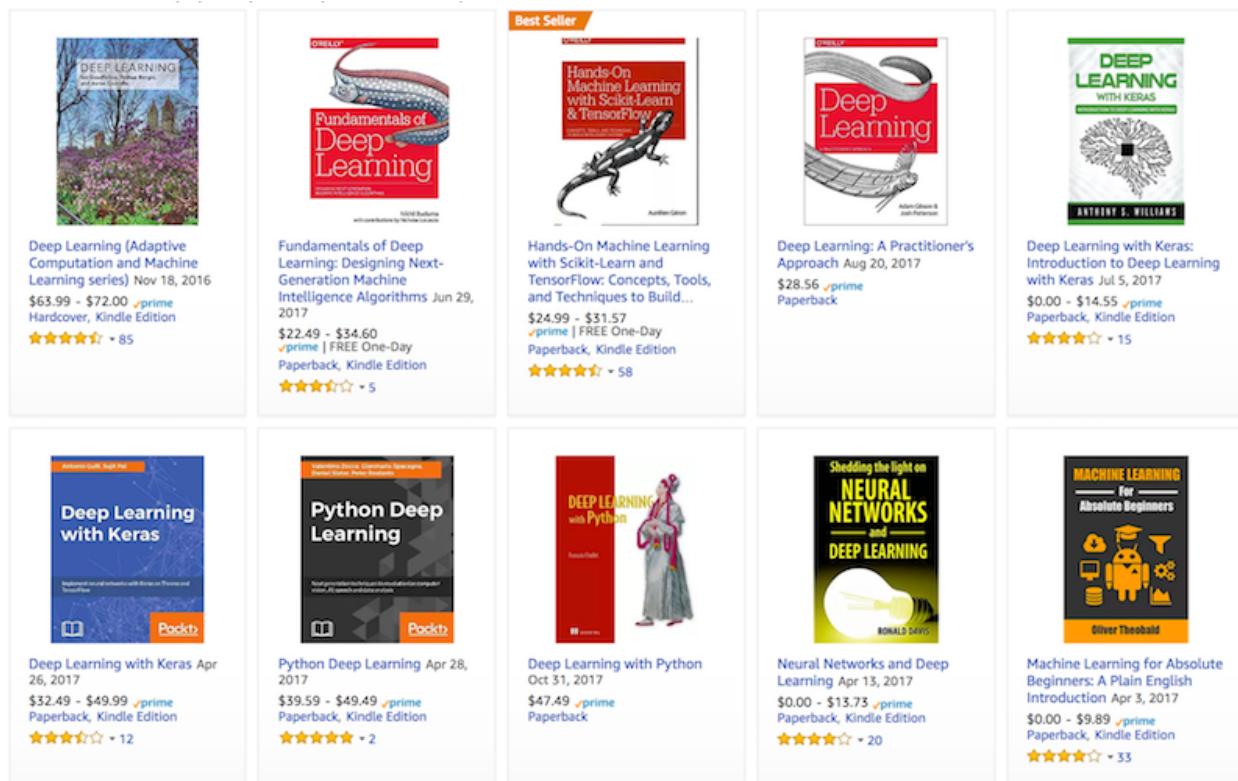
这类任务有时候也叫做多标签分类。想象一下，人们可能会把多个标签同时标注在自己的某篇技术类博客文章上，例如“机器学习”、“科技”、“编程语言”、“云计算”、“安全与隐私”和“AWS”。这里面的标签其实有时候相互关联，比如“云计算”和“安全与隐私”。当一篇文章可能被标注的数量很大时，人力标注就显得很吃力。这就需要使用机器学习了。

### 搜索与排序

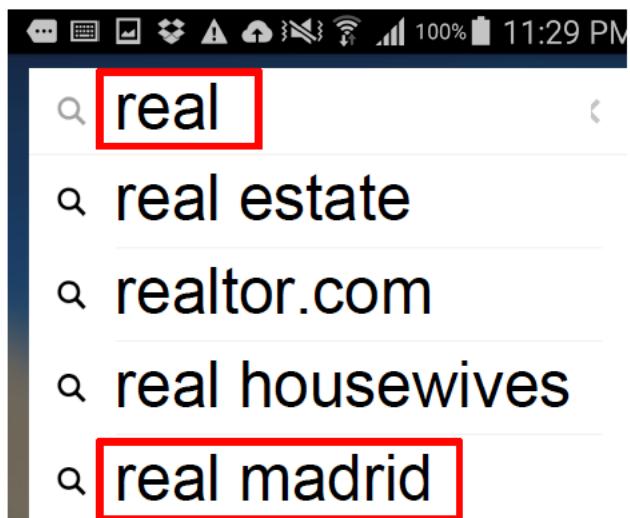
搜索与排序关注的问题更多的是如何把一堆对象排序。例如在信息检索领域，我们常常关注如何把一堆文档按照与检索条目的相关性排序。在互联网时代，由于搜索引擎的流行，我们更加关注如何对网页进行排序。互联网时代早期有一个著名的网页排序算法叫做 PageRank。该算法的排序结果并不取决于特定的用户检索条目。这些排序结果可以更好地为所包含检索条目的网页进行排序。

### 推荐系统

推荐系统与搜索排序关系紧密，并广泛应用于购物网站、搜索引擎、新闻门户网站等等。推荐系统的主要目标是把用户可能感兴趣的东西推荐给用户。推荐算法用到的信息多种多样，例如用户的自我描述、对过往推荐的反应、社交网络、喜好等等。下图展示了亚马逊网站对笔者之一有关深度学习类书籍的推荐结果。



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

序列学习也是一类近来备受关注的机器学习问题。在这类问题中，输入和输出不仅限于固定的数量。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。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。

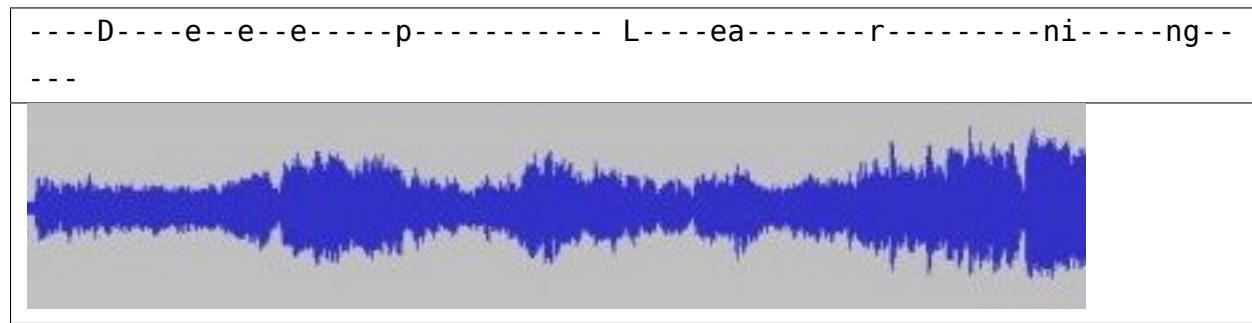
## 语法分析

一个常见语法分析的例子是，给定一个文本序列，如何找出其中的命名实体，例如人物姓名、城市名称等。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

在语音识别的问题里，输入序列通常都是麦克风的声音，而输出是对通过麦克风所说的话的文本转录。这类问题通常有一个难点，例如声音通常都在特定的采样率采样，因为声音和文本之间不存在一一对应。换言之，语音识别是一类序列转换问题。这里的输出往往比输入短很多。



## 文本转语音

这是语音识别问题的逆问题。这里的输入是一个文本序列，而输出才是声音序列。因此，这类问题的输出比输入长。

## 机器翻译

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。

### 2.1.6 非监督学习

上述的机器学习问题和应用场景都是基于监督学习的。与监督学习不同，非监督学习不需要训练数据被标识。以图片分析为例，对监督学习来说，训练数据里的图片需要被标识为狗、猫或者别的动物，如此一来，一个分类模型被训练后就能把一张新图片识别为某种动物。而对非监督学习而言，例如聚类学习，可以把一堆无标识的图片自动聚合成若干类，其中每类分别对应一种动物。

以下我们简要介绍一些常见的非监督学习任务。

- **聚类**问题通常研究如何把一堆数据点分成若干类，从而使得同类数据点相似而非同类数据点不似。根据实际问题，我们需要定义相似性。
- **子空间估计**问题通常研究如何将原始数据向量在更低维度下表示。理想情况下，子空间的表示要具有代表性从而才能与原始数据接近。一个常用方法叫做主成分分析。
- **表征学习**希望在欧几里得空间中找到原始对象的表示方式，从而能在欧几里得空间里表示出原始对象的符号性质。例如我们希望找到城市的向量表示，从而使得我们可以进行这样的向量运算：罗马 - 意大利 + 法国 = 巴黎。

- 生成对抗网络是最近一个很火的领域。这里描述数据的生成过程，并检查真实与生成的数据是否统计上相似。

## 2.1.7 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：数据读取和当数据已经在内存里时如何处理。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，或者正态分布，我们会在之后的章节分别详细介绍。

我们首先从 `mxnet` 导入 `ndarray` 这个包

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

然后我们创建一个有 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，就是说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。下面创建数组，它的元素服从均值 0 方差 1 的正态分布。

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

Out[5]:

```
[[ 2.21220636  1.16307867  0.7740038   0.48380461]
 [ 1.04344046  0.29956347  1.18392551  0.15302546]
 [ 1.89171135 -1.16881478 -1.23474145  1.55807114]]
<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]:

```
[[ 3.21220636  2.16307878  1.77400374  1.48380458]
 [ 2.04344034  1.29956341  2.18392563  1.15302551]]
```

```
[ 2.89171124 -0.16881478 -0.23474145  2.55807114]
<NDArray 3x4 @cpu(0)>
```

乘法:

In [9]: `x * y`

Out[9]:

```
[[ 2.21220636  1.16307867  0.7740038   0.48380461]
 [ 1.04344046  0.29956347  1.18392551  0.15302546]
 [ 1.89171135 -1.16881478 -1.23474145  1.55807114]]
<NDArray 3x4 @cpu(0)>
```

指数运算:

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

Out[10]:

```
[[ 9.13585091  3.19976926  2.16843081  1.6222347 ]
 [ 2.83896756  1.34926963  3.26717448  1.16535461]
 [ 6.63070631  0.31073502  0.29090998  4.74965096]]
<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]:

```
[[ 4.63309383  2.67995477  1.04622626]
 [ 4.63309383  2.67995477  1.04622626]
 [ 4.63309383  2.67995477  1.04622626]]
<NDArray 3x3 @cpu(0)>
```

## 2.2.3 广播

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.astype(np.float32) # 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

如果可以现有的数组之后不会再用, 我们也可以用复制操作符达到这个目的:

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

Out[17]: True

## 2.2.6 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  $f = 2 * (x ** 2)$  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  $dz/dx = 4 * x$ 。

```
In [6]: x.grad == 4*x
```

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  $df/da = x$ , 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  $dy(x)/dx$ 。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  $dz(y(x))/dx$  的结果。回想一下链式法则, 我们可以得到  $dz(y(x))/dx = [dz(y)/dy] * [dy(x)/dx]$ 。当  $y$  是一个更大的  $z$  函数的一部分, 并且我们希望求得  $dz/dx$  保存在  $x.grad$  中时, 我们可以传入头梯度  $dz/dy$  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](#)

## 3.1 线性回归—从 0 开始

虽然强大的深度学习框架可以减少很多重复性工作，但如果你过于依赖它提供的便利抽象，那么你可能不会很容易地理解到底深度学习是如何工作的。所以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  $y$ ，线性模型的目标是找一根线，其由向量  $w$  和位移  $b$  组成，来最好地近似每个样本  $X[i]$  和  $y[i]$ 。用数学符号来表示就是我们将学  $w$  和  $b$  来预测，

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

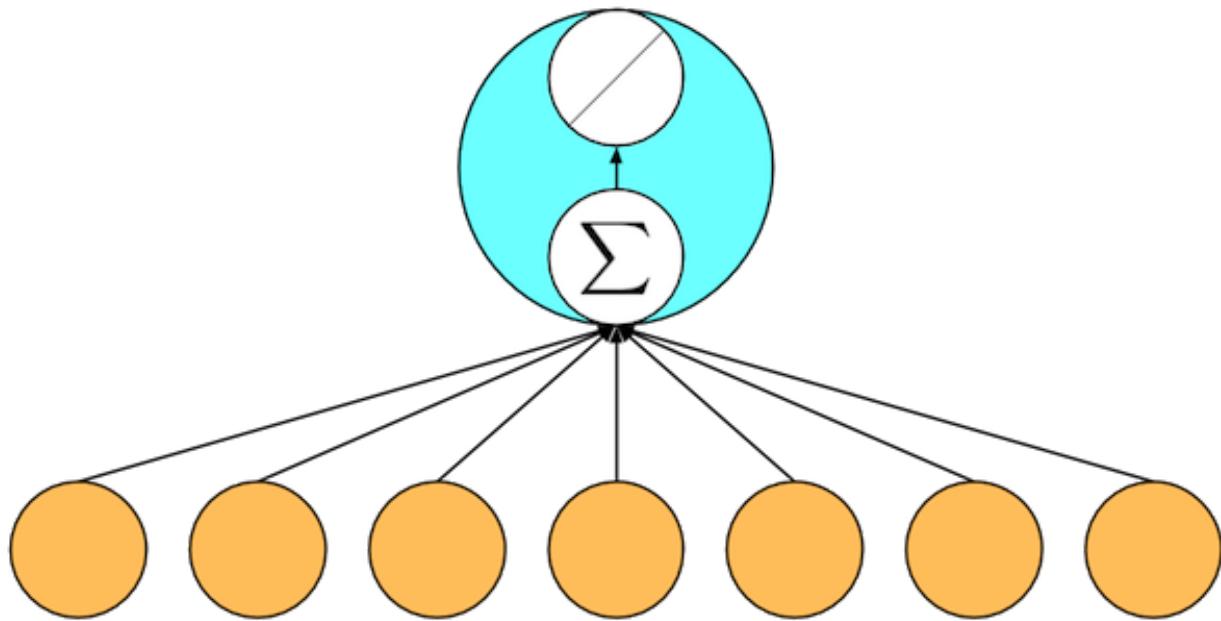
并最小化所有数据点上的平方误差

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

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很奇怪。实际上线性模型是最简单但也可能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，每一层使用下一层的节点作为输入，并输出给上面层使用。为了计算一个节点值，我们将输入节点值做加权和，然后再加上一个激活函数。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方法来生成数据



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

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])
```

```
[ 2.21220636  1.16307867]
<NDArray 2 @cpu(0)>
[ 4.6620779]
<NDArray 1 @cpu(0)>
```

### 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 [3]: 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 [4]: for data, label in data_iter():
    print(data, label)
    break
```

```
[[ 0.74177277  0.85860491]
 [ 0.16912138  0.86568999]
 [-0.36137256  0.98650014]
 [ 0.19090296  1.31934202]
 [ 1.23909569  2.17306304]
 [ 0.05408725  0.40822244]
 [ 0.59501076  0.37921664]
 [-1.81205451  1.8176862 ]
 [ 0.77932847  0.49383944]
 [ 0.54560065  0.10940859]]
<NDArray 10x2 @cpu(0)>
[ 2.77818441  1.60680676  0.11717813  0.09762684 -0.71278811  2.92113543
 4.0954175   -5.60291815  4.06573725  4.9092207 ]
<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 [5]: 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，所以我们需要创建它们的梯度。

```
In [6]: 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做乘法再加上偏移：

```
In [7]: 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 [8]: def square_loss(yhat, y):  
    # 注意这里我们把 y 变形成 yhat 的形状来避免自动广播  
    return (yhat - y.reshape(yhat.shape)) ** 2
```

### 3.1.7 优化

虽然线性回归有显试解，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。（我们会之后一直使用这个函数，我们将其保存在utils.py。）

```
In [9]: 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，一次迭代里，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 [10]: epochs = 5  
learning_rate = .001  
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()
print("Epoch %d, average loss: %f" % (e, total_loss/num_examples))

Epoch 0, average loss: 7.942103
Epoch 1, average loss: 0.099854
Epoch 2, average loss: 0.001388
Epoch 3, average loss: 0.000118
Epoch 4, average loss: 0.000103

```

训练完成后我们可以比较学到的参数和真实参数

```

In [11]: true_w, w

Out[11]: ([2, -3.4],
           [[ 1.99990547]
            [-3.40019464]]
           <NDArray 2x1 @cpu(0)>)

In [12]: true_b, b

Out[12]: (4.2,
           [ 4.19953823]
           <NDArray 1 @cpu(0)>

```

### 3.1.9 结论

我们现在看到仅仅使用 NDArray 和 autograd 我们可以很容易地实现一个模型。

### 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
```

```
[[ -2.14711547 -0.65737247]
 [-1.15005136  0.11681478]
 [ 0.58428466  0.35468408]
 [ 0.43498218 -0.52985734]
 [-0.22734004  0.64571917]
 [ 0.77373821  1.68393016]
 [ 1.56177032  0.9938466 ]
 [-1.70300341 -1.24758053]]
```

```
[-0.03598363 -0.68015915]
[-1.79335225  1.62773883]]
<NDArray 10x2 @cpu(0)>
[ 2.15768409  1.51200199  4.16360617  6.86338711  1.53583205  0.02224033
 3.92908359  5.03815031  6.43048239 -4.93066931]
<NDArray 10 @cpu(0)>
```

### 3.2.3 定义模型

当我们手写模型的时候，我们需要先声明模型参数，然后再使用它们来构建模型。但 `gluon` 提供大量提前定制好的层，使得我们只需要主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。

虽然我们之后会介绍如何构造任意结构的神经网络，构建模型最简单的办法是利用 `Sequential` 来所有层串起来。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，它唯一必须要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

这里的训练跟前面没有太多区别，唯一的就是我们不再是调用 `SGD`，而是 `trainer.step` 来更新模型。

```
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: 0.893984  
Epoch 1, average loss: 0.000052  
Epoch 2, average loss: 0.000052  
Epoch 3, average loss: 0.000052  
Epoch 4, average loss: 0.000052
```

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],  
[[ 1.99850559 -3.40052533]]  
<NDArray 1x2 @cpu(0)>)
```

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

```
Out[11]: (4.2,  
[ 4.20072603])
```

```
<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

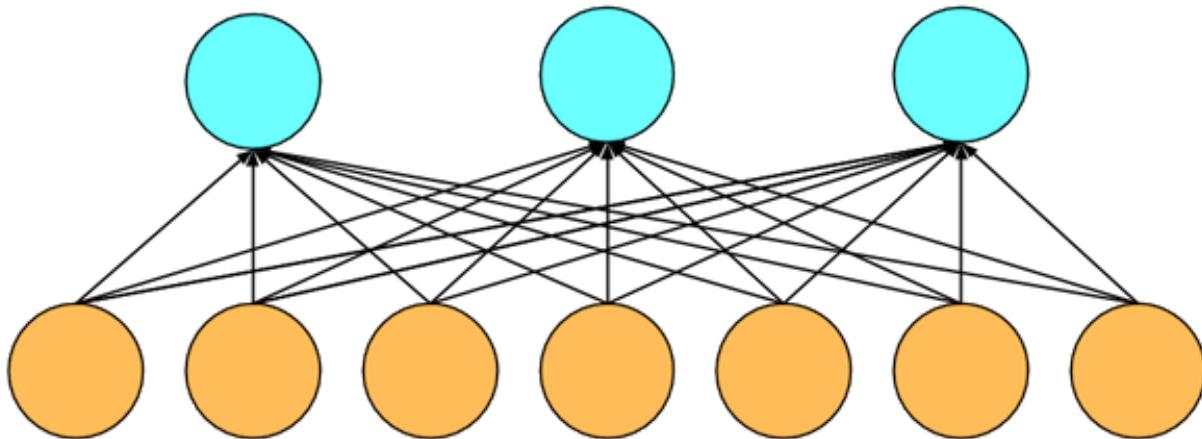
- 在训练的时候, 为什么我们用了比前面要大 10 倍的学习率呢? (提示: 可以尝试运行 `help(trainer.step)` 来寻找答案。)
- 如何拿到 `weight` 的梯度呢? (提示: 尝试 `help(dense.weight)`)

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。现在你知道如何读取和操作数据, 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。

```
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

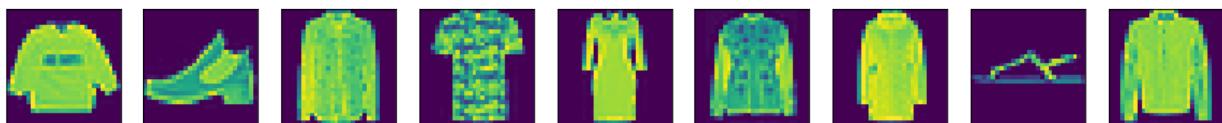
```
In [3]: import matplotlib.pyplot as plt

        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  $yhat$  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))

[[ 0.0444386   0.09409688   0.28758538   0.3520911   0.22178803]
 [ 0.02263156   0.65802747   0.1223551    0.13697426   0.06001162]]
<NDArray 2x5 @cpu(0)>

[ 1.  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  $yvec=[0, 1, 0, 0, 0, 0, 0, 0, 0]$ 。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/num_outputs = 0.1.`

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

```
Out[13]: 0.07958984375
```

### 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: 3.713226, Train acc 0.446193, Test acc 0.581738
Epoch 1. Loss: 1.952251, Train acc 0.615913, Test acc 0.651172
Epoch 2. Loss: 1.594117, Train acc 0.668401, Test acc 0.685352
Epoch 3. Loss: 1.405009, Train acc 0.696892, Test acc 0.703809
Epoch 4. Loss: 1.284098, Train acc 0.715531, Test acc 0.719434

```

### 3.3.8 预测

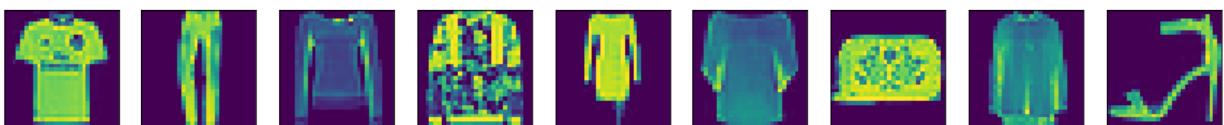
训练完成后, 现在我们可以演示对输入图片的标号的预测

```

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', 'shirt', 'bag', 'bag', '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，你会发现结果马上会变成很糟糕，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: 0.781384, Train acc 0.748055, Test acc 0.801270
Epoch 1. Loss: 0.574374, Train acc 0.810433, Test acc 0.817383
Epoch 2. Loss: 0.528390, Train acc 0.823232, Test acc 0.826855
Epoch 3. Loss: 0.505022, Train acc 0.829621, Test acc 0.832227
Epoch 4. Loss: 0.490404, Train acc 0.833655, Test acc 0.841797
```

### 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。

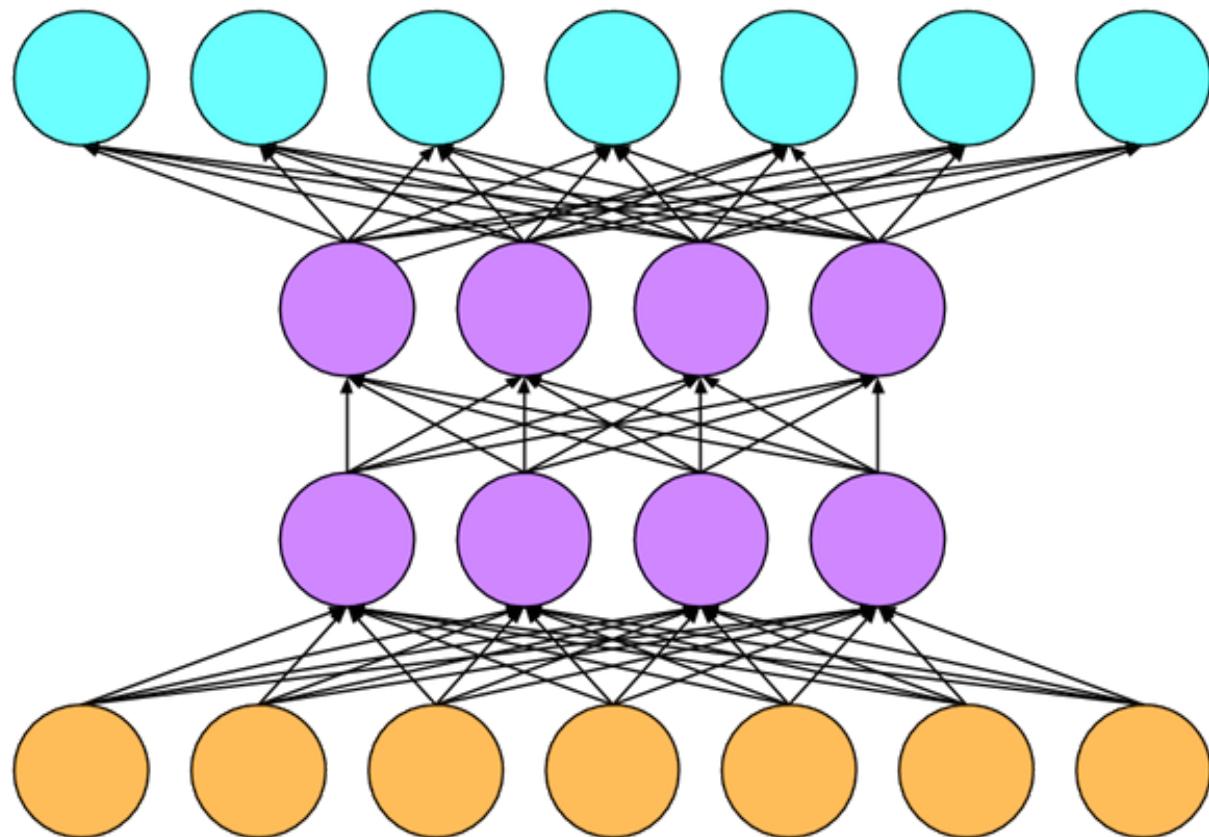
### 3.5.1 数据获取

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: 0.782815, Train acc 0.703330, Test acc 0.829590
Epoch 1. Loss: 0.491146, Train acc 0.816761, Test acc 0.853711
Epoch 2. Loss: 0.429556, Train acc 0.840514, Test acc 0.857422
Epoch 3. Loss: 0.396903, Train acc 0.853158, Test acc 0.859082
Epoch 4. Loss: 0.371960, Train acc 0.863187, Test acc 0.872949
```

### 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: 0.716610, Train acc 0.736181, Test acc 0.839941
Epoch 1. Loss: 0.465280, Train acc 0.827953, Test acc 0.853418
Epoch 2. Loss: 0.412255, Train acc 0.847900, Test acc 0.868164
Epoch 3. Loss: 0.377627, Train acc 0.860721, Test acc 0.877344
Epoch 4. Loss: 0.355243, Train acc 0.869958, Test acc 0.870020
```

### 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](#)

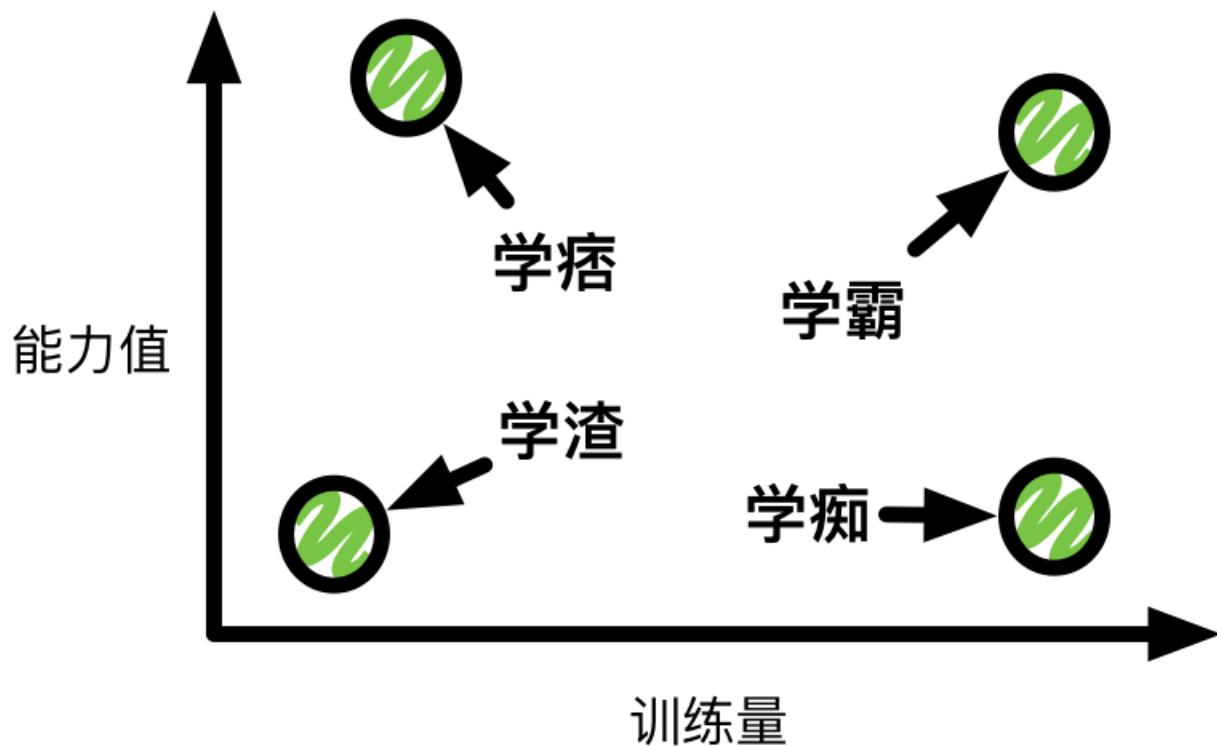
## 3.7 欠拟合和过拟合

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。

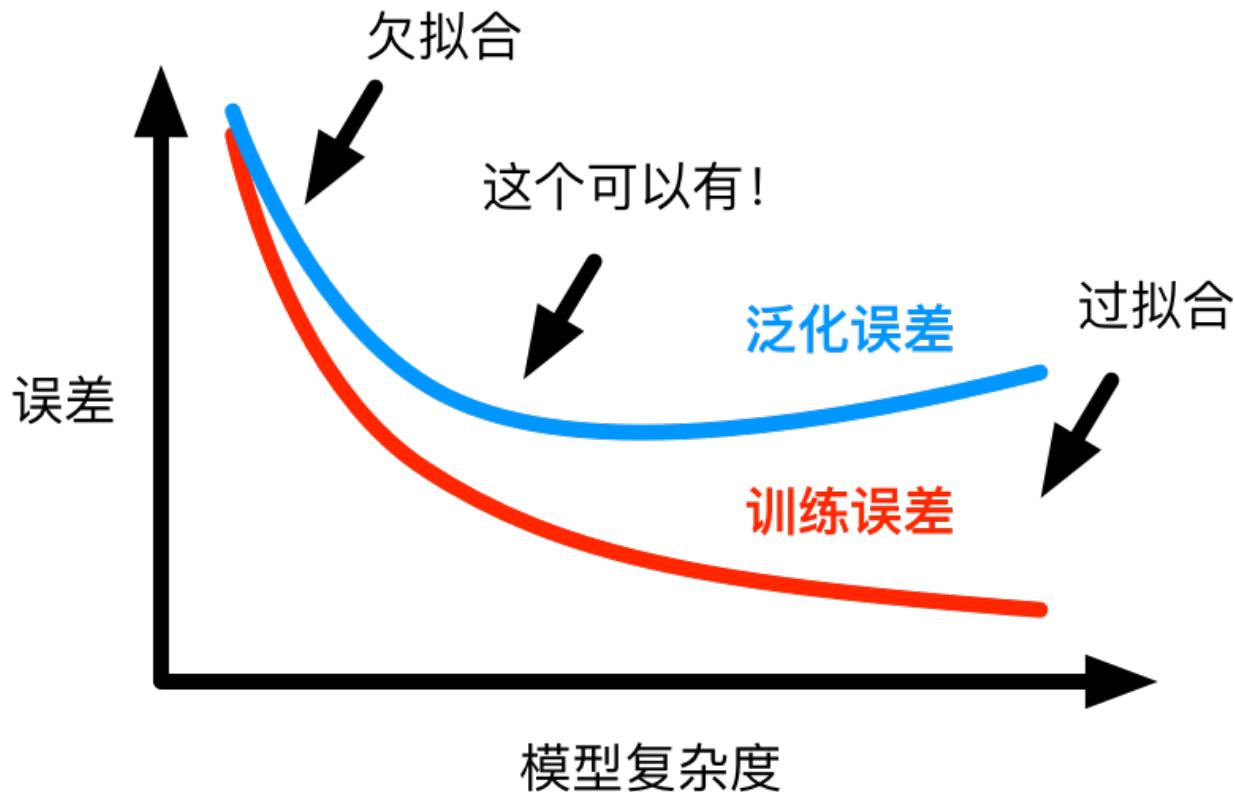
我们要尽可能同时避免欠拟合和过拟合的出现。虽然有很多因素可能导致这两种拟合问题，在这里我们重点讨论两个因素：模型的选择和训练数据集的大小。

#### 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

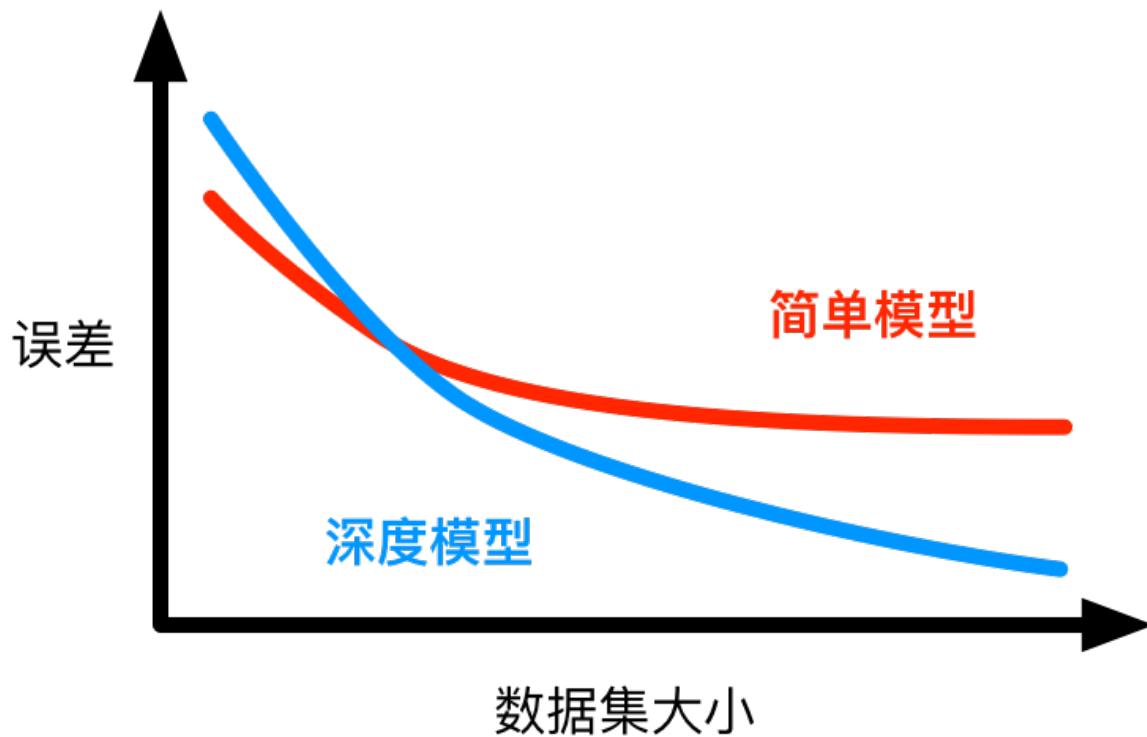
$$\hat{y} = b + \sum_{k=1}^K x^k w_k$$

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)
        y_train, y_test = y[:num_train], y[num_train:]

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

```
Out[2]: ('x:',  
         [[ 2.21220636]  
          [ 1.16307867]  
          [ 0.7740038 ]  
          [ 0.48380461]  
          [ 1.04344046]]  
        <NDArray 5x1 @cpu(0)>, 'X:',  
        [[ 2.21220636  4.893857 10.82622147]  
         [ 1.16307867  1.35275197 1.57335699]  
         [ 0.7740038   0.59908187 0.46369165]  
         [ 0.48380461   0.2340669  0.11324265]  
         [ 1.04344046   1.08876801 1.13606453]]  
        <NDArray 5x3 @cpu(0)>, 'y:',  
        [ 51.63998032 10.56862736  6.59616661  5.32011604  8.78539848]  
        <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。

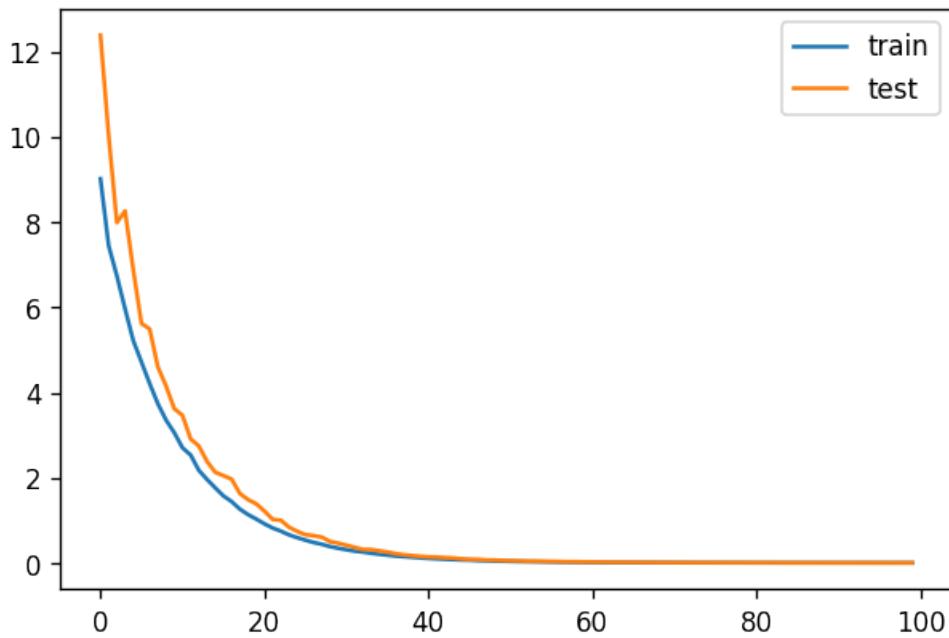
```
In [3]: %matplotlib inline  
import matplotlib as mpl  
mpl.rcParams['figure.dpi']= 120  
import matplotlib.pyplot as plt  
  
def test(net, X, y):  
    return square_loss(net(X), y).mean().asscalar()  
  
def train(X_train, X_test, y_train, y_test):  
    # 线性回归模型  
    net = gluon.nn.Sequential()  
    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:])`

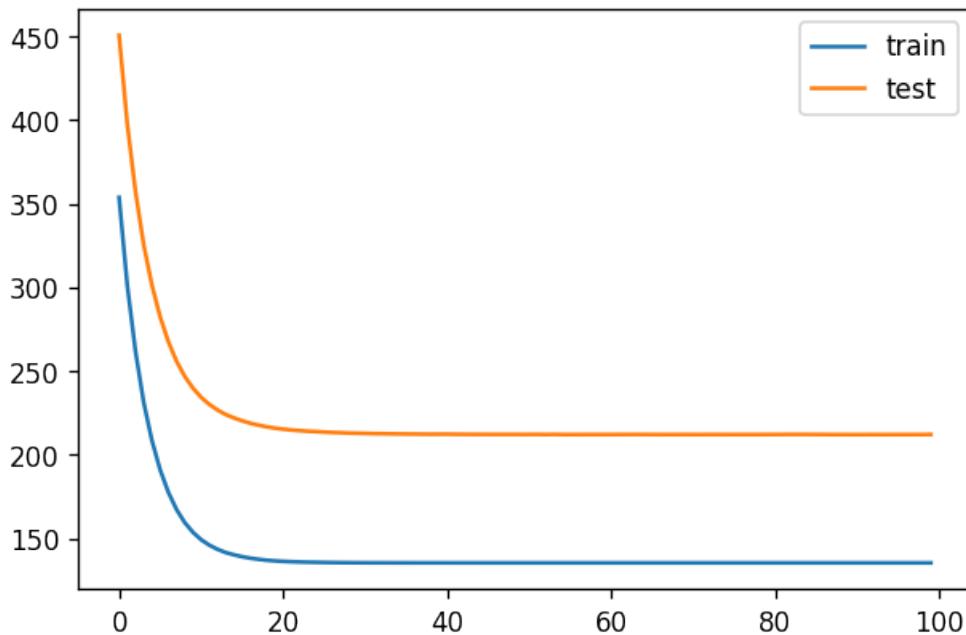


```
Out[4]: ('learned weight',
 [[ 1.19352305 -3.38453817  5.59780073]]
 <NDArray 1x3 @cpu(0)>, 'learned bias',
 [ 4.97252178]
 <NDArray 1 @cpu(0)>)
```

### 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:])
```

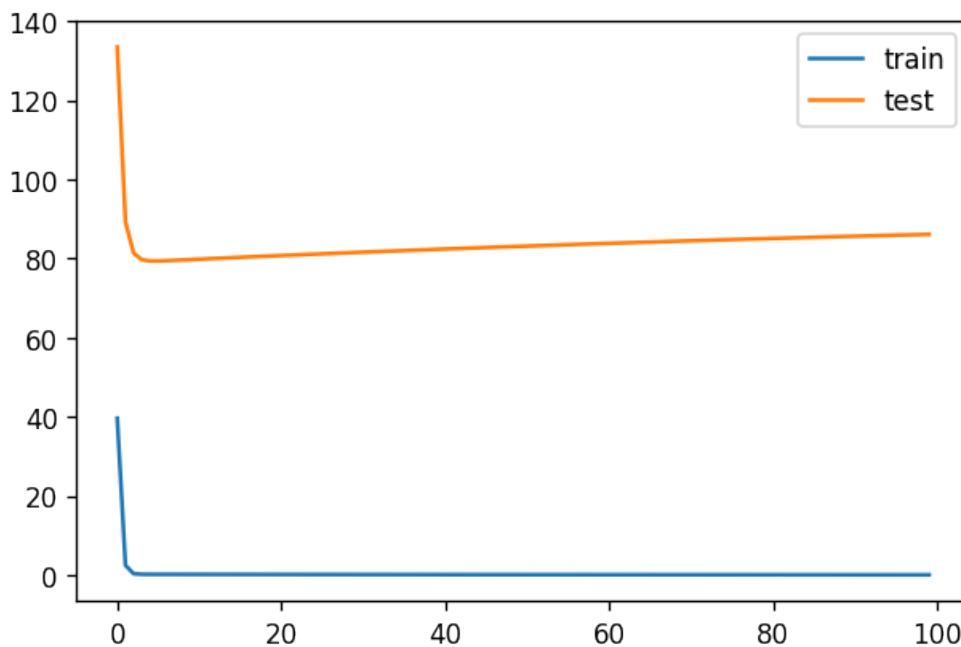


```
Out[5]: ('learned weight',
 [[ 20.59081841]],
 <NDArray 1x1 @cpu(0)>, 'learned bias',
 [-0.14694121]
 <NDArray 1 @cpu(0)>)
```

### 训练量不足（过拟合）

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

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



```
Out[6]: ('learned weight',
 [[ 1.11312687  1.81215715  3.67018008]],
 <NDArray 1x3 @cpu(0)>, 'learned bias',
 [ 0.64948529]
 <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

- 学渣、学痞、学痴、和学霸对应的模型复杂度、训练量、训练误差、泛化误差。
- 如果用一个三阶多项式模型来拟合一个线性模型生成的数据，可能会有什么问题？为什么？
- 在我们本节提到的三阶多项式拟合问题里，有没有可能把 1000 个样本的训练误差的期望降到 0，为什么？

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

        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 get_params():
    w = nd.random.normal(shape=(num_inputs, 1))*0.1
    b = nd.zeros((1,))
    for param in (w, b):
        param.attach_grad()
    return (w, b)
```

### 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
```

### 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

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

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

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

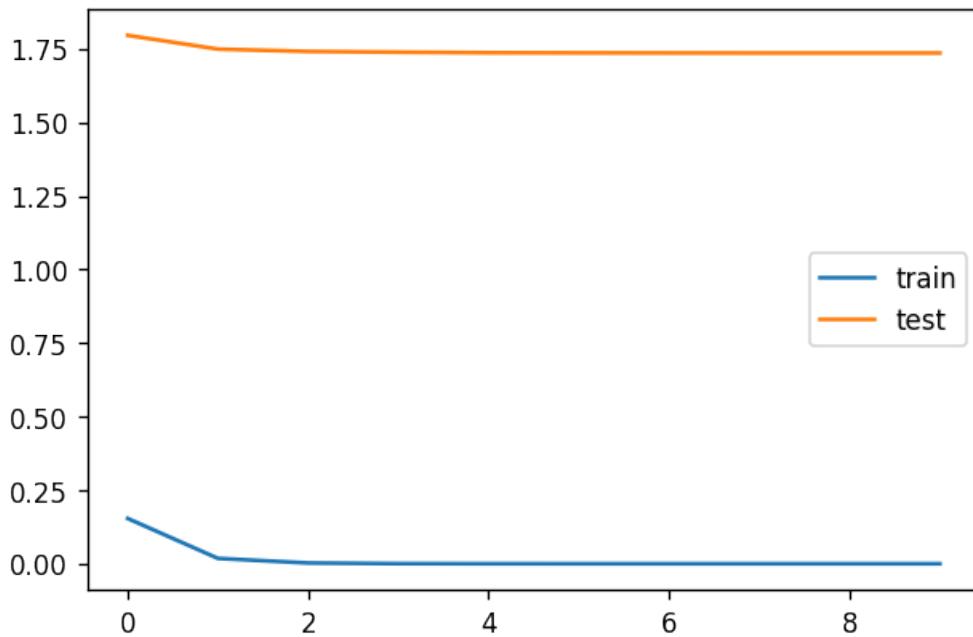
def test(params, X, y):
    return square_loss(net(X, 0, *params), y).mean().asscalar()

def train(lambd):
    epochs = 10
    learning_rate = 0.002
    params = get_params()
    train_loss = []
    test_loss = []
    for e in range(epochs):
        for data, label in data_iter(num_train):
            with autograd.record():
                output = net(data, lambd, *params)
                loss = square_loss(
                    output, label) + lambd * L2_penalty(*params)
            loss.backward()
            SGD(params, learning_rate)
        train_loss.append(test(params, X_train, y_train))
        test_loss.append(test(params, X_test, y_test))
    plt.plot(train_loss)
    plt.plot(test_loss)
    plt.legend(['train', 'test'])
    plt.show()
    return 'learned w[:10]:', params[0][:10], 'learend b:', params[1]
```

### 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]:',  
 [[ 0.17129801]  
 [-0.03934795]  
 [-0.06848308]  
 [ 0.00796483]  
 [-0.27748865]  
 [ 0.03912187]  
 [ 0.09117991]  
 [ 0.14972937]  
 [-0.15559918]  
 [ 0.177788 ]]  
<NDArray 10x1 @cpu(0)>, 'learend b:',  
 [-0.03464997]  
<NDArray 1 @cpu(0)>)

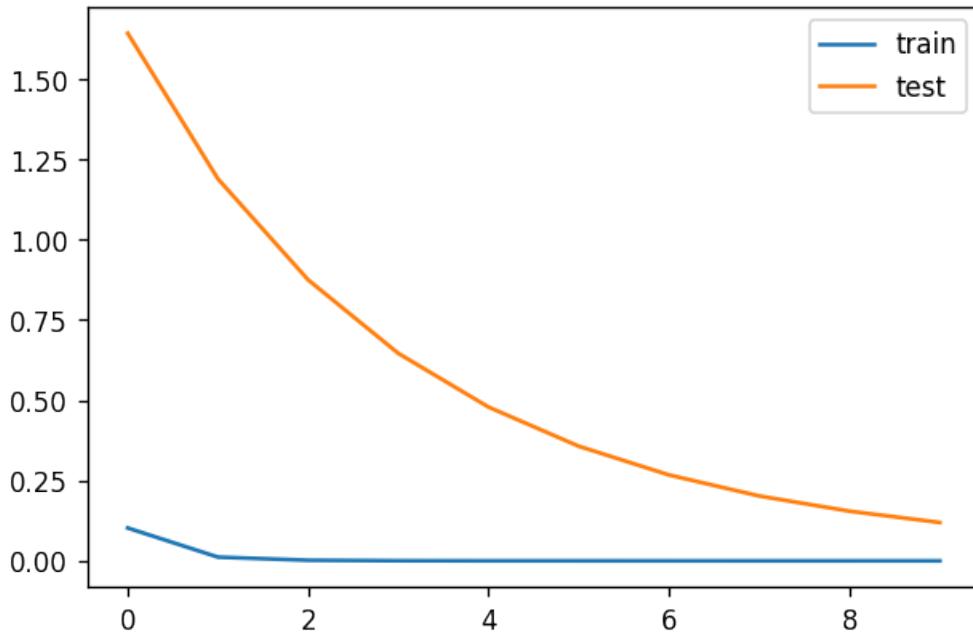
即便训练误差可以达到 0.000000，但是测试数据集上的误差很高。这是典型的过拟合现象。

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(2)`



```
Out[9]: ('learned w[:10]:',
 [-0.02098747]
 [ 0.01595206]
 [-0.00986921]
 [-0.02825547]
 [-0.02462067]
 [-0.01803801]
 [-0.02377258]
 [-0.00135702]
 [-0.04466932]
 [-0.0012121 ]
 <NDArray 10x1 @cpu(0)>, 'learend b:',
 [ 0.00050303]
 <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

        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

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):
    learning_rate = 0.005
    epochs = 10

    net = gluon.nn.Sequential()
    with net.name_scope():
        net.add(gluon.nn.Dense(1))
    net.initialize()

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

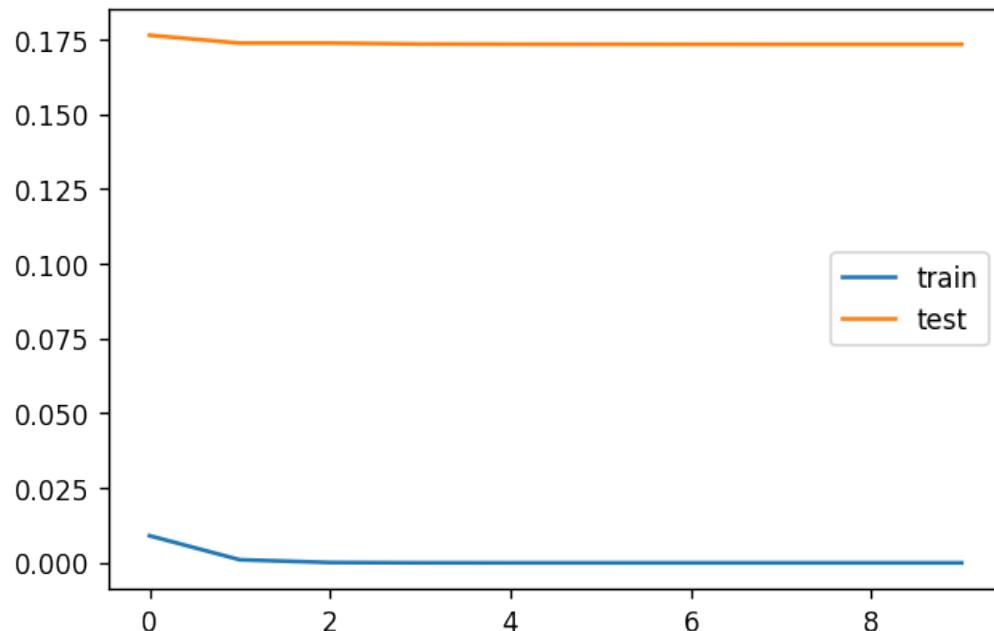
    train_loss = []
    test_loss = []
    for e in range(epochs):
        for data, label in data_iter_train:
            with autograd.record():
                output = net(data)
                loss = square_loss(output, label)
            loss.backward()
            trainer.step(batch_size)
       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

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

## 训练模型并观察过拟合

接下来我们训练并测试我们的高维线性回归模型。

In [3]: `train(0)`



Out[3]: ('learned w[:10]:',  
[[ -0.01392719 0.01498022 -0.00414371 -0.05509679 0.05354908 0.03420147  
 0.05406792 -0.04383744 -0.00049713 -0.00847002]]  
<NDArray 1x10 @cpu(0)>, 'learned b:',  
[-0.0033146]  
<NDArray 1 @cpu(0)>)

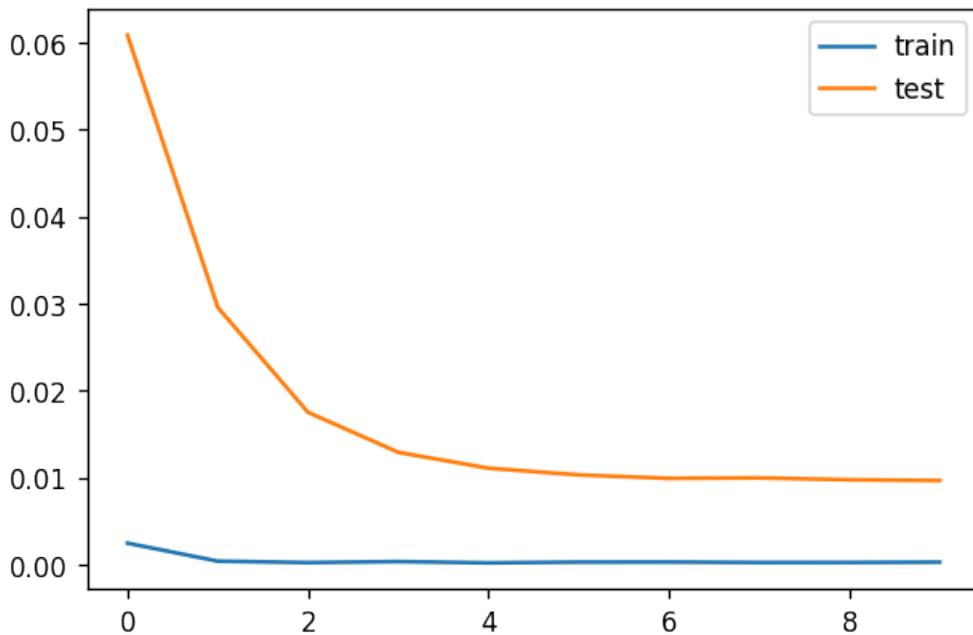
即便训练误差可以达到 0.000000，但是测试数据集上的误差很高。这是典型的过拟合现象。

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]:',
 [[ -5.31405443e-04 -1.74406427e-03  8.30002711e-04  2.16350309e-05
   -2.41776370e-03  1.32189784e-03 -2.02652346e-03  1.78635586e-04
   2.71238177e-03  2.31217383e-03]],
 <NDArray 1x10 @cpu(0)>, 'learned b:',
 [-0.00117249]
 <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—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.  0.  0.  6.]
 [ 0.  10. 0.  0.]
 [ 16. 18. 20. 0.]
 [ 24. 26. 0.  0.]
 [ 0.  34. 0.  0.]]
<NDArray 5x4 @cpu(0)>
```

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

Out[4]:

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

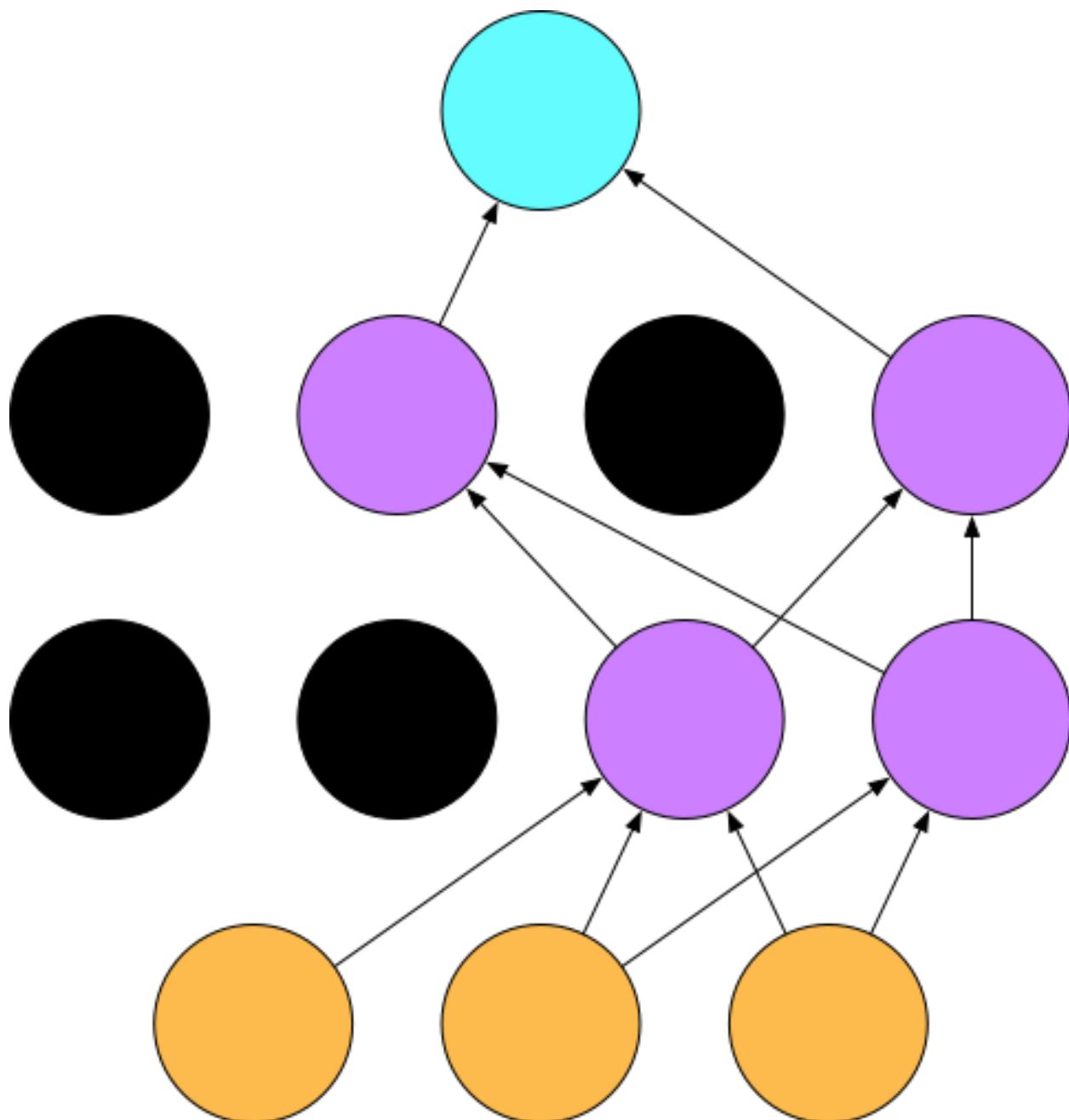
### 3.10.3 丢弃法的本质

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：学到的参数得使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。每个丢弃曾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: 1.197462, Train acc 0.537733, Test acc 0.770020
Epoch 1. Loss: 0.587562, Train acc 0.782746, Test acc 0.825684
Epoch 2. Loss: 0.504481, Train acc 0.816982, Test acc 0.841406
Epoch 3. Loss: 0.454021, Train acc 0.832031, Test acc 0.847266
Epoch 4. Loss: 0.422473, Train acc 0.846254, Test acc 0.856055
```

### 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？

## 3.11 丢弃法—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。

### 3.11.1 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})

        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: 0.824836, Train acc 0.696171, Test acc 0.819531
Epoch 1. Loss: 0.510905, Train acc 0.812938, Test acc 0.851465
Epoch 2. Loss: 0.452181, Train acc 0.836215, Test acc 0.860156
Epoch 3. Loss: 0.421572, Train acc 0.847684, Test acc 0.863477
Epoch 4. Loss: 0.399475, Train acc 0.852942, Test acc 0.872070
```

### 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。

## 3.12 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

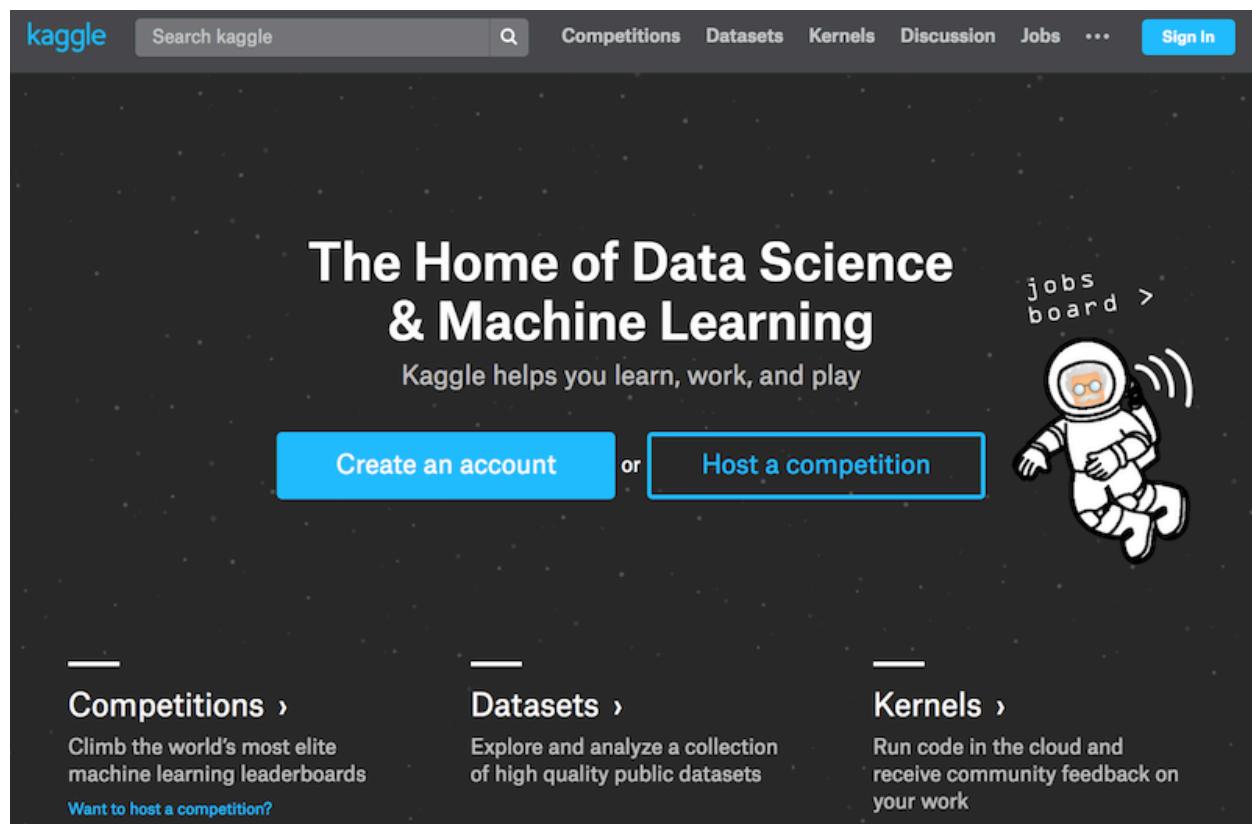
需要注意的是，本章仅提供一些基本实战流程供大家参考。对于数据的预处理、模型的设计和参数的选择等，我们特意只提供最基础的版本。希望大家一定要通过动手实战、仔细观察实验现象、认真分析实验结果并不断调整方法，从而得到令自己满意的结果。

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。

### 3.12.1 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。

### 3.12.2 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’。



### 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.

只有训练数据集包括了我们需要在测试数据集中预测的每个房子的价格。数据可以从[房价预测](#)问题中下载。

训练数据集下载地址 测试数据集下载地址

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

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

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

我们看看数据长什么样子。

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

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

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

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

[5 rows x 81 columns]
```

数据大小如下。

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

```
Out[3]: (1460, 81)
In [4]: test.shape
Out[4]: (1459, 80)
```

### 3.12.3 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()
```

### 3.12.4 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)
```

### 3.12.5 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
```

### 3.12.6 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。

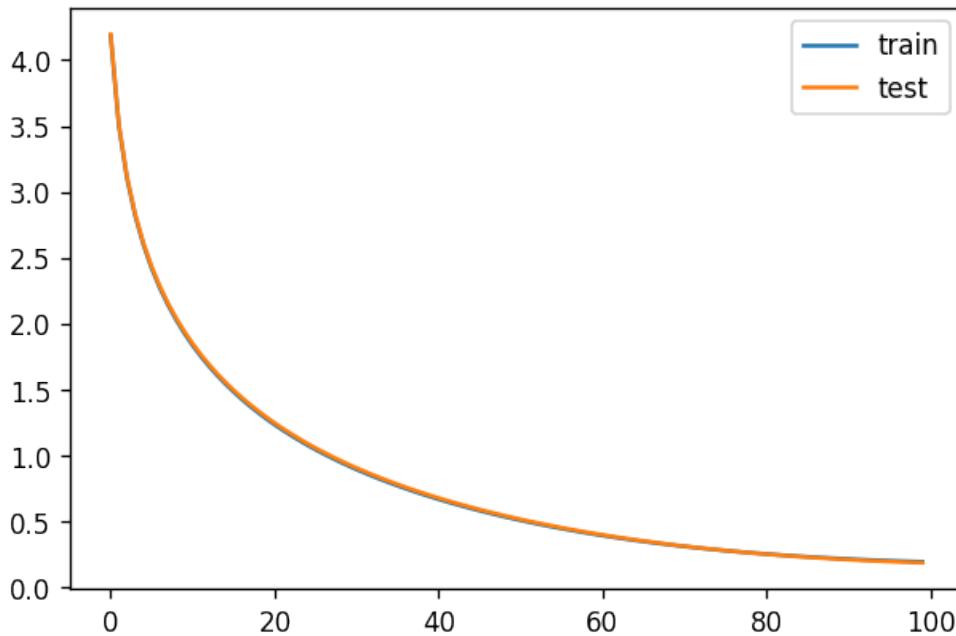
```
In [16]: train_loss, test_loss = k_fold_cross_valid(k, epochs, verbose_epoch, X_train,
                                                    y_train, learning_rate, weight_decay)
print("%d-fold validation: Avg train loss: %f, Avg test loss: %f" %
      (k, train_loss, test_loss))
```

Epoch 96, train loss: 0.201918

Epoch 97, train loss: 0.199799

Epoch 98, train loss: 0.197833

Epoch 99, train loss: 0.195934



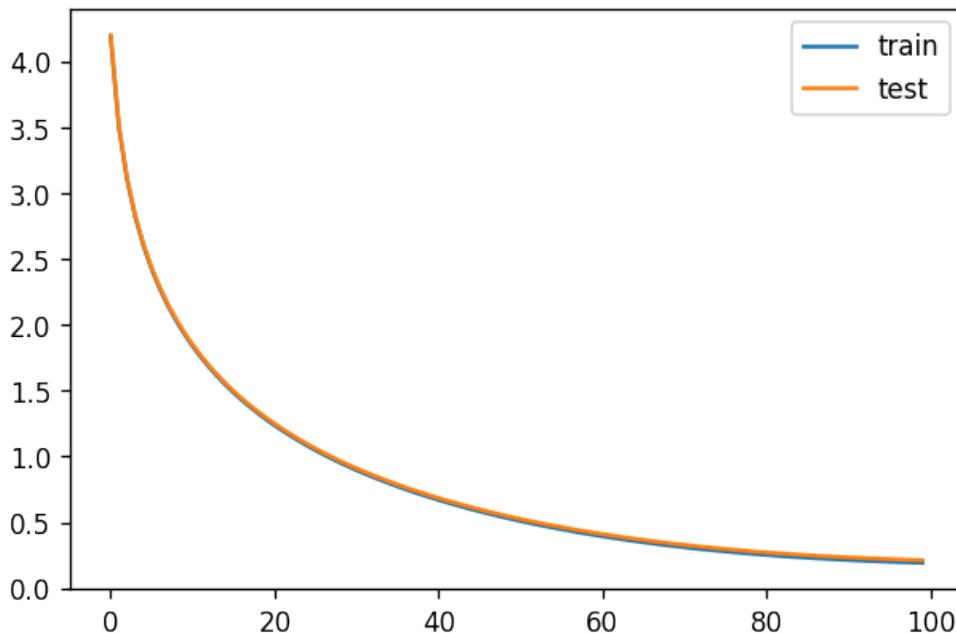
Test loss: 0.188764

Epoch 96, train loss: 0.197870

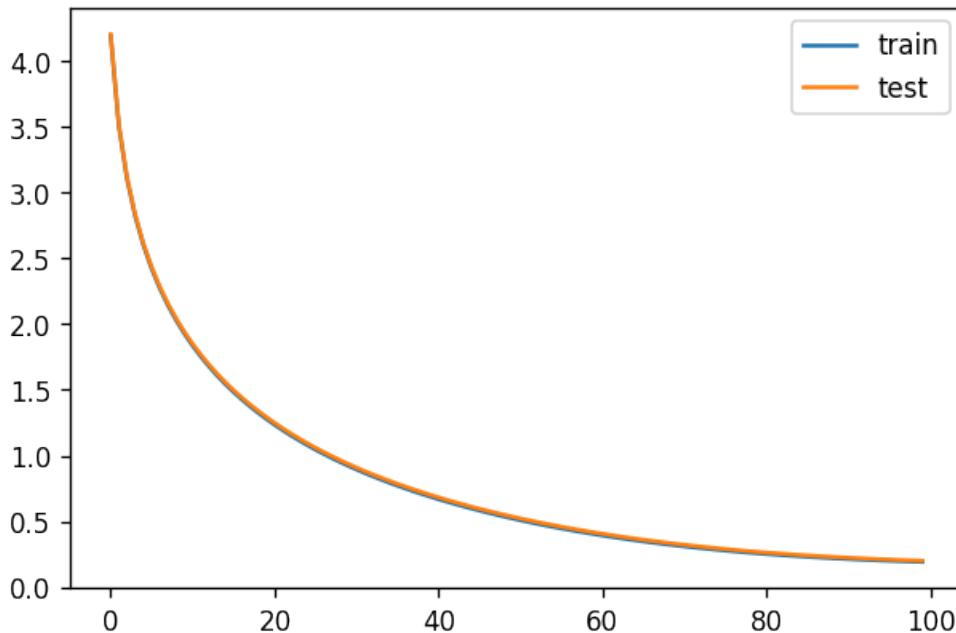
Epoch 97, train loss: 0.195711

Epoch 98, train loss: 0.193683

Epoch 99, train loss: 0.191721



```
Test loss: 0.211987  
Epoch 96, train loss: 0.198819  
Epoch 97, train loss: 0.196695  
Epoch 98, train loss: 0.194659  
Epoch 99, train loss: 0.192721
```

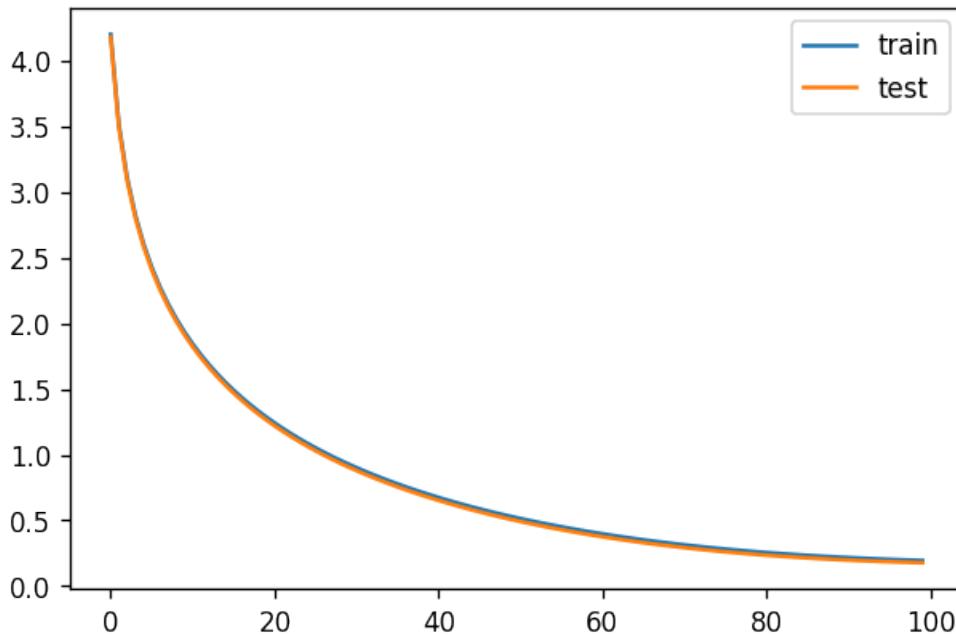


```
Test loss: 0.201222  
Epoch 96, train loss: 0.201556
```

Epoch 97, train loss: 0.199379

Epoch 98, train loss: 0.197326

Epoch 99, train loss: 0.195395



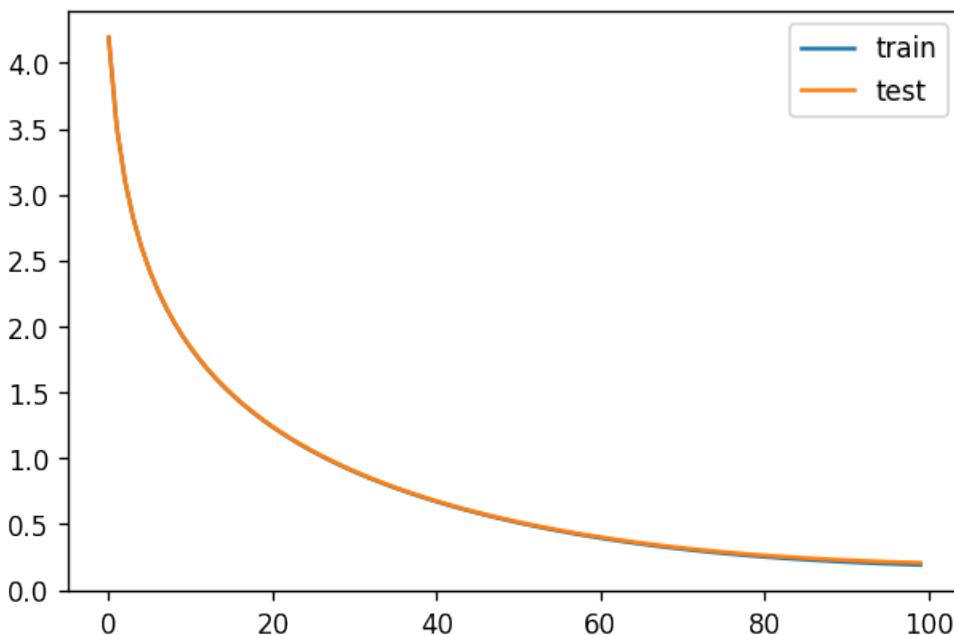
Test loss: 0.178347

Epoch 96, train loss: 0.196881

Epoch 97, train loss: 0.194632

Epoch 98, train loss: 0.192547

Epoch 99, train loss: 0.190558



Test loss: 0.206243

5-fold validation: Avg train loss: 0.193266, Avg test loss: 0.197313

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。

### 3.12.7 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。

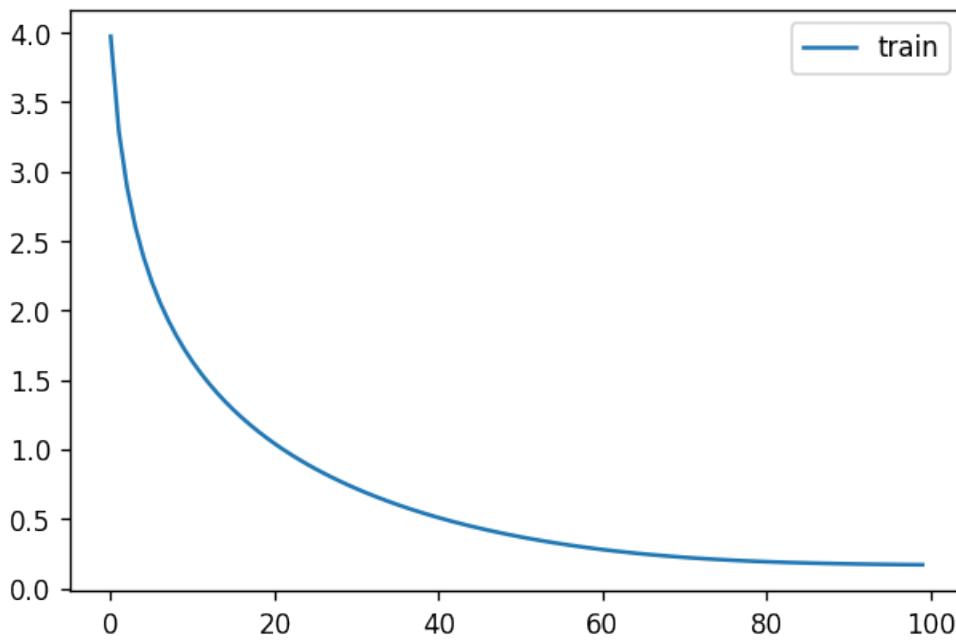
```
In [17]: def learn(epochs, verbose_epoch, X_train, y_train, test, learning_rate,
               weight_decay):
    net = get_net()
    train(net, X_train, y_train, None, None, epochs, verbose_epoch,
          learning_rate, weight_decay)
    preds = net(X_test).asnumpy()
    test['SalePrice'] = pd.Series(preds.reshape(1, -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: 0.171159
Epoch 97, train loss: 0.170576
Epoch 98, train loss: 0.170039
Epoch 99, train loss: 0.169553
```



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。

The screenshot shows the Kaggle competition interface for the 'House Prices: Advanced Regression Techniques' competition. At the top, there's a red house icon with a yellow 'SOLD' sign. To the right of the icon, the competition title 'House Prices: Advanced Regression Techniques' is displayed in bold black text. Below the title, a brief description reads 'Predict sales prices and practice feature engineering, RFs, and gradient boosting' and '1,698 teams · 2 years to go'. Below the description, there's a navigation bar with several tabs: 'Overview' (which is highlighted in blue), 'Data', 'Kernels', 'Discussion', 'Leaderboard', 'Rules', and 'Team'. To the right of the navigation bar, there are two buttons: 'My Submissions' and a large blue button labeled 'Submit Predictions'.

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。

The screenshot shows the Kaggle submission interface. At the top, it says "Step 1 Upload submission file". Below this is a large dashed box with an "Upload Submission File" button containing an upward arrow icon. To the left of the box is a "File Format" section stating: "Your submission should be in CSV format. You can upload this in a zip/gz/rar/7z archive, if you prefer." To the right is a "Number of Predictions" section stating: "We expect the solution file to have 1459 prediction rows. This file should have a header row. Please see sample submission file on the [data page](#)." Below this is "Step 2 Describe submission", which includes a rich text editor toolbar with buttons for bold, italic, etc., and a text area for describing the submission. A note says "Briefly describe your submission." At the bottom is a blue "Make Submission" button.

### 3.12.8 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(256, Activation(relu))
    (1): Dense(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(256, linear)
(dense1): Dense(10, linear)
)
```

Out[3]:

```
[[ 0.07253762 -0.06698283 -0.06698831 -0.05273689 -0.01113968 -0.00793779
  0.09705824 -0.03755333 -0.04266765  0.04197346]
 [ 0.0155654 -0.05711009 -0.0683545 -0.05742466  0.01055721  0.02320427
  0.08411142 -0.06533261 -0.03446646  0.03570375]
 [ 0.05067764 -0.05897554 -0.04308926 -0.08502439  0.00575621  0.03044649
  0.09501629 -0.03974968 -0.04040152  0.05879233]
 [ 0.06112296 -0.0441079 -0.03792209 -0.03608051  0.02816215  0.02037679
  0.08093711 -0.02402739 -0.05363904  0.06081349]]
<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。

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]:

```
[[ -0.06672797 -0.05538915  0.04318529  0.01911315  0.06418858 -0.04620389
   0.01699099 -0.0089059  -0.04179767  0.05090332]
 [-0.0731464  -0.03577812  0.05492381  0.00659474  0.03694215 -0.02706025
  0.05889042  0.02400289 -0.09183833  0.03147602]
 [-0.05731897 -0.04326176  0.05486837  0.00200205  0.04650036 -0.04220583
  0.02378599  0.00912581 -0.05819123  0.04085773]
 [-0.06538966 -0.05371606  0.04816957  0.04632074  0.00925024 -0.00943102
  0.03003146  0.0304444  -0.06500123  0.00145768]]
<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(256, Activation(relu))
(1): Dense(128, Activation(relu))
)
(dense): Dense(64, linear)
)
(1): Dense(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]:

```
[[ 0.00212593  0.00365805]
 [ 0.00161272  0.00441845]
 [ 0.00204872  0.00352518]]
<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:
[[-0.06206018  0.06491279 -0.03182812 -0.01631819 -0.00312688]
 [ 0.0408415   0.04370362  0.00404529 -0.0028032   0.00952624]
 [-0.01501013  0.05958354  0.04705103 -0.06005495 -0.02276454]
 [-0.0578019   0.02074406 -0.06716943 -0.01844618  0.04656678]]
<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)>

[[[-0.06206018  0.06491279 -0.03182812 -0.01631819 -0.00312688]
 [ 0.0408415   0.04370362  0.00404529 -0.0028032   0.00952624]
 [-0.01501013  0.05958354  0.04705103 -0.06005495 -0.02276454]
 [-0.0578019   0.02074406 -0.06716943 -0.01844618  0.04656678]]
<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)。它会对所有权重初始化成  $[-0.07, 0.07]$  之前均匀分布的随机数。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())

[[[-0.00359026  0.02804598  0.0302582   0.00220872 -0.01496244]
 [ 0.00701151  0.01725933  0.02721515 -0.02177767  0.00500832]
 [ 0.01344385  0.00112992  0.00272668  0.03227538 -0.00392631]
 [-0.01813176 -0.03435376 -0.00385197  0.01124353 -0.01286032]]
<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. 下面我们自定义一个初始化方法。

```
In [9]: 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)
    def _init_bias(self, _, arr):
        print('init bias', arr.shape)
        # 初始化偏移
        arr[:] = 2

        # FIXME: init_bias doesn't work
    params.initialize(init=MyInit(), force_reinit=True)
    print(net[0].weight.data(), net[0].bias.data())

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

[[ 6.11160707  8.08467007  6.93244457  9.71874046  9.51299286]
 [ 8.40910149  7.24975014  6.79753971  8.06531715  7.18515968]
 [ 9.51174355  8.48815536  5.49640179  5.30112743  9.84904448 ]
 [ 8.33383369  8.26570034  8.35318947  5.85454798  6.05191278]]
<NDArray 4x5 @cpu(0)>
[ 0.  0.  0.  0.]
<NDArray 4 @cpu(0)>
```

### 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 [10]: net = get_net()
        print(net.collect_params())

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 [11]: net.initialize(init=MyInit())
```

你会看到我们并没有看到 MyInit 打印的东西，这是因为我们仍然不知道形状。真正的初始化发生在我们看到数据时。

```
In [12]: net(x)
```

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

Out[12]:

```
[[ 714.38696289  674.64227295]
 [ 702.47253418  664.6595459 ]
 [ 563.41418457  533.46936035]]
<NDArray 3x2 @cpu(0)>
```

这时候我们看到 shape 里面的 0 被填上正确的值了。

```
In [13]: print(net.collect_params())
```

```
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'>)
)
```

#### 4.2.4 避免延后初始化

有时候我们不想要延后初始化，这时候可以在创建网络的时候指定输入大小。

```
In [14]: net = nn.Sequential()
    with net.name_scope():
        net.add(nn.Dense(4, in_units=5, activation="relu"))
        net.add(nn.Dense(2, in_units=4))

    net.initialize(MyInit())

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

#### 4.2.5 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 [15]: net = nn.Sequential()
    with net.name_scope():
        net.add(nn.Dense(4, in_units=4, activation="relu"))
        net.add(nn.Dense(4, in_units=4, activation="relu", params=net[-1].params))
        net.add(nn.Dense(2, in_units=4))
```

初始化然后打印

```
In [16]: net.initialize(MyInit())
    print(net[0].weight.data())
    print(net[1].weight.data())

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

[[ 7.40446758  5.59363842  8.44330597  6.58991575]
 [ 9.40237999  7.07131481  9.59117699  5.32073736]
 [ 6.08411074  8.46236038  7.82594442  7.83300734]
 [ 9.32551289  6.32694721  7.54484463  7.6162405 ]]
<NDArray 4x4 @cpu(0)>

[[ 7.40446758  5.59363842  8.44330597  6.58991575]
 [ 9.40237999  7.07131481  9.59117699  5.32073736]]
```

```
[ 6.08411074 8.46236038 7.82594442 7.83300734]
[ 9.32551289 6.32694721 7.54484463 7.6162405 ]
<NDArray 4x4 @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))
```

```
[[ -0.01796602  0.00558522]
 [-0.01267574  0.0035485 ]]
<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))
```

```
[[ -0.01796602  0.00558522]
 [-0.01267574  0.0035485 ]]
<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，各式各样的连接方式。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]:
[-4.65661294e-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  $3 \times 3$  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]: (  
[[ 0.0418165 -0.01914864  0.05928377]  
[-0.03545186 -0.02811849 -0.00089732]  
[-0.01562342 -0.04868037 -0.00192191]]  
<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.float64'>)
```

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]:
[[ 0.22405601  0.11224781  0.0761601   0.           0.01357741]
 [ 0.18777567  0.02327698  0.06112128  0.           0.03826571]]
<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]:
[[ 0.04900231  0.0135336 ]
 [ 0.02698925  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`

Fri Sep 22 07:50:22 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  | 0000:00:1E.0   Off  |                  0 |
| N/A   65C    P0    43W / 150W |      0MiB /  7612MiB |      93%     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
Version: 0.11.1b20170902
Summary: MXNet is an ultra-scalable deep learning framework. This version uses openblas.
Home-page: https://github.com/dmlc/mxnet
Author: UNKNOWN
Author-email: UNKNOWN
License: Apache 2.0
Location: /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages
Requires: graphviz, numpy
Name: mxnet-cu80
Version: 0.11.1b20170913
Summary: MXNet is an ultra-scalable deep learning framework. This version uses CUDA-8.0.
Home-page: https://github.com/dmlc/mxnet
Author: UNKNOWN
Author-email: UNKNOWN
License: Apache 2.0
Location: /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/site-packages
```

Requires: graphviz, numpy

### 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)>,
[[ 0.74021935  0.9209938   0.03902049]
 [ 0.96896291  0.92514056  0.4463501 ]]
<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：

In [6]: `import sys`

```
try:
    nd.array([1,2,3], ctx=mx.gpu(1))
except mx.MXNetError as err:
    sys.stderr.write(str(err))
```

[07:50:28] 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/_ctypes.
[bt] (6) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/lib-dynload/_ctypes.
[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/../lib/libpython3.6m.so.1.0(_P
```

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))
```

```
[07:50:28] src/c_api/c_api_ndarray.cc:129: Check failed: ndinputs[i].ctx().dev_mask() == c
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/_ctypes.
[bt] (6) /var/lib/jenkins/miniconda3/envs/gluon_zh_docs/lib/python3.6/lib-dynload/_ctypes.
[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/../lib/libpython3.6m.so.1.0(_P
```

### 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.asnumpy())
    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]:

```
[[ 0.0059893 ]
 [ 0.00988404]
 [ 0.00913574]]
<NDArray 3x1 @gpu(0)>
```

确认下权重:

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

Out[14]:

```
[[ 0.0068339   0.01299825]]
<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  $4 \times 4$  矩阵，使用一个  $3 \times 3$  的权重，计算得到  $2 \times 2$  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 vdumoulin@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 input_filter x output_filter x height x width, 这里 input_filter 和 output_filter 分别是 1 和 2
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=1` 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

$$\text{conv}(\text{data}, \text{w}, \text{b}) = \sum_i \text{conv}(\text{data}[:, i, :, :], \text{w}[0, i, :, :], \text{b})$$

```
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)>
```

当输入需要多通道时，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.]]

  [[[ 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。

```
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:
[[ 1.36640418e-04  1.00769263e-04  -8.63557943e-05 ... ,  -2.66836432e-05
  1.32240079e-04  7.94463267e-05]
 [ 7.32613553e-05  4.53796165e-05   4.78603124e-06 ... ,  -2.34715699e-05
  5.11510734e-06  5.56128507e-05]
 [ 1.17923890e-04  1.27743362e-04   1.12565976e-06 ... ,  -6.70055306e-05
  1.35457376e-04  7.63333519e-05]
 ...
 [ 8.82731329e-05  8.11657592e-05  -5.42089329e-05 ... ,  -2.48615252e-05
  1.09997469e-04  8.33676095e-05]
 [ 9.74532086e-05  1.14936956e-04  -6.81549591e-06 ... ,  -2.69450611e-05
  8.70642907e-05  9.34668642e-05]
 [ 1.39541895e-04  9.24538326e-05  -1.16575589e-04 ... ,  -4.22856174e-05
  1.58704846e-04  5.11915641e-05]]
<NDArray 256x10 @gpu(0)>
```

## 5.1.4 训练

跟前面没有什么不同的

```
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: 2.276526, Train acc 0.125643, Test acc 0.297949  
 Epoch 1. Loss: 1.109217, Train acc 0.581305, Test acc 0.733594  
 Epoch 2. Loss: 0.612865, Train acc 0.764705, Test acc 0.818848  
 Epoch 3. Loss: 0.505518, Train acc 0.810555, Test acc 0.842871  
 Epoch 4. Loss: 0.443159, Train acc 0.836359, Test acc 0.852637

## 5.1.5 结论

可以看到卷积神经网络比前面的多层感知的分类精度更好。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。不管你相信不详细，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'))
            net.add(nn.MaxPool2D(pool_size=2, strides=2))
            net.add(nn.Conv2D(channels=50, kernel_size=3, activation='relu'))
            net.add(nn.MaxPool2D(pool_size=2, strides=2))
            net.add(nn.Flatten())
            net.add(nn.Dense(128, activation="relu"))
            net.add(nn.Dense(10))
```

然后我们尝试将模型权重初始化在 GPU 上

```
In [2]: import sys
        sys.path.append('..')
        import utils

        ctx = utils.try_gpu()
        net.initialize(ctx=ctx)

        print('initialize weight on', ctx)
initialize weight on gpu(0)
```

## 5.2.2 获取数据然后训练

跟之前没什么两样。

```
In [3]: from mxnet import autograd
        from mxnet import gluon
        from mxnet import nd

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:
        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: 1.456226, Train acc 0.453696, Test acc 0.746484
Epoch 1. Loss: 0.603551, Train acc 0.766938, Test acc 0.817578
Epoch 2. Loss: 0.445212, Train acc 0.830502, Test acc 0.848730
Epoch 3. Loss: 0.391621, Train acc 0.851341, Test acc 0.855176
Epoch 4. Loss: 0.361640, Train acc 0.864971, Test acc 0.872559
```

## 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  $\eta$  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  $\gamma$  和偏移参数  $\beta$ 。

我们定义：

$$\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$$

$$\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$$

批量归一化层的输出是  $\{y_i = BN_{\gamma, \beta}(x_i)\}$ 。

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
```

```
# output channels = 20, kernel = (5,5)
c1 = 20
W1 = nd.random.normal(shape=(c1,1,5,5), scale=weight_scale, ctx=ctx)
b1 = nd.zeros(c1, ctx=ctx)

# batch norm 1
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)

# output channels = 50, kernel = (3,3)
c2 = 50
W2 = nd.random_normal(shape=(c2,c1,3,3), scale=weight_scale, ctx=ctx)
b2 = nd.zeros(c2, ctx=ctx)

# batch norm 2
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)

# output dim = 128
o3 = 128
W3 = nd.random.normal(shape=(1250, o3), scale=weight_scale, ctx=ctx)
b3 = nd.zeros(o3, 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)

# 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: 2.136276, Train acc 0.194332, Test acc 0.682617  
Epoch 1. Loss: 0.619482, Train acc 0.760284, Test acc 0.836328  
Epoch 2. Loss: 0.423679, Train acc 0.842797, Test acc 0.862598  
Epoch 3. Loss: 0.358198, Train acc 0.867974, Test acc 0.886621  
Epoch 4. Loss: 0.320372, Train acc 0.882491, Test acc 0.890137

### 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
# from mxnet import init

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: 0.637443, Train acc 0.762688, Test acc 0.851660
Epoch 1. Loss: 0.378895, Train acc 0.858627, Test acc 0.871484
Epoch 2. Loss: 0.317261, Train acc 0.881028, Test acc 0.884180
Epoch 3. Loss: 0.285683, Train acc 0.894531, Test acc 0.896582
Epoch 4. Loss: 0.259694, Train acc 0.904294, Test acc 0.897949
```

### 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。

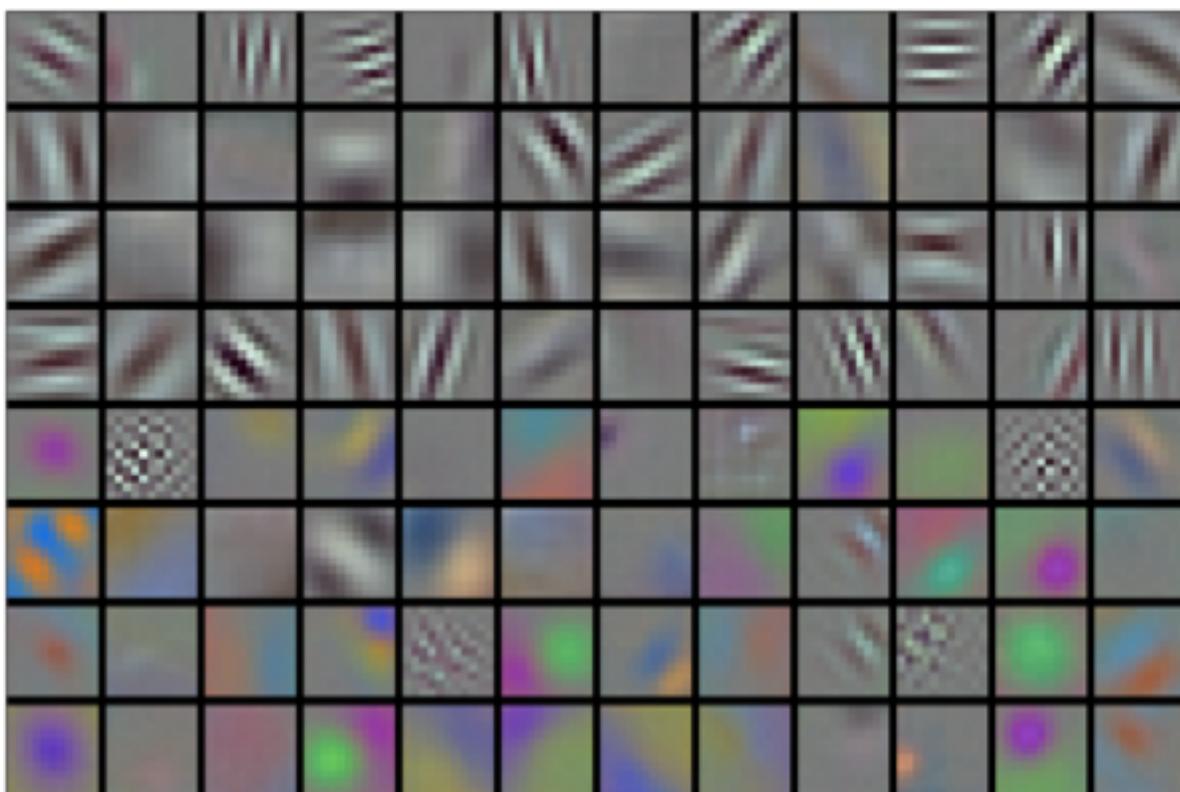
取而代之，研究者们通过勤劳，智慧和黑魔法生成了许多手工特征。通常的模式是 1. 找个数据集 2. 用一堆已有的特征提取函数生成特征 3. 把这些特征表示放进一个简单的线性模型（当时认为的机器学习部分仅限这一步）

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囊中羞涩的 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  $4 \times 4$  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还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。

```

net = nn.Sequential()
with net.name_scope():
    # 第一阶段
    net.add(nn.Conv2D(
        channels=96, kernel_size=11, strides=4, activation='relu'))
    net.add(nn.MaxPool2D(pool_size=3, strides=2))
    # 第二阶段
    net.add(nn.Conv2D(
        channels=256, kernel_size=5, padding=2, activation='relu'))
    net.add(nn.MaxPool2D(pool_size=3, strides=2))
    # 第三阶段
    net.add(nn.Conv2D(
        channels=384, kernel_size=3, padding=1, activation='relu'))
    net.add(nn.Conv2D(
        channels=384, kernel_size=3, padding=1, activation='relu'))
    net.add(nn.Conv2D(
        channels=256, kernel_size=3, padding=1, activation='relu'))
    net.add(nn.MaxPool2D(pool_size=3, strides=2))
    # 第四阶段
    net.add(nn.Flatten())
    net.add(nn.Dense(4096, activation="relu"))
    net.add(nn.Dropout(.5))
    # 第五阶段
    net.add(nn.Dense(4096, activation="relu"))
    net.add(nn.Dropout(.5))
    # 第六阶段
    net.add(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  $224 \times 224$ 。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  $224 \times 224$ 。

```

In [2]: import sys
        sys.path.append('..')
        import utils
        from mxnet import image

        def transform(data, label):
            # resize from 28 x 28 to 224 x 224

```

```
    data = image.imresize(data, 224, 224)
    return utils.transform_mnist(data, label)

batch_size = 64
train_data, test_data = utils.load_data_fashion_mnist(
    batch_size, transform)
```

## 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 autograd
        from mxnet import gluon
        from mxnet import nd
        from mxnet import init
        ctx = utils.try_gpu()
        net.initialize(ctx=ctx, init=init.Xavier())

        softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
        trainer = gluon.Trainer(
            net.collect_params(), 'sgd', {'learning_rate': 0.01})

        for epoch in range(1):
            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: 0.998259, Train acc 0.627732, Test acc 0.797870

### 5.5.7 结论

从 LeNet 到 Alexnet, 虽然学术界花了 20 多年, 但实现起来也就多了几行而已。

### 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 ( $3 \times 3$ ) 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()`  
    `with net.name_scope():`  
        `net.add(vgg_stack(architecture))`  
        `net.add(nn.Flatten())`  
        `net.add(nn.Dense(4096, activation="relu"))`  
        `net.add(nn.Dropout(.5))`  
        `net.add(nn.Dense(4096, activation="relu"))`  
        `net.add(nn.Dropout(.5))`  
        `net.add(nn.Dense(num_outputs))`

### 5.6.2 模型训练

这里跟 Alexnet 的训练代码一样：

```
In [5]: import sys
        sys.path.append('..')
        import utils
        from mxnet import image

        def transform(data, label):
            # resize from 28 x 28 to 224 x 224
            data = image.imresize(data, 224, 224)
            return utils.transform_mnist(data, label)

        batch_size = 64
        train_data, test_data = utils.load_data_fashion_mnist(
            batch_size, transform)

        from mxnet import autograd
        from mxnet import gluon
        from mxnet import nd
        from mxnet import init
        ctx = utils.try_gpu()
        net.initialize(ctx=ctx, init=init.Xavier())

        softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
        trainer = gluon.Trainer(
            net.collect_params(), 'sgd', {'learning_rate': 0.05})

        for epoch in range(1):
            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: 1.047353, Train acc 0.614389, Test acc 0.863654
```

### 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。

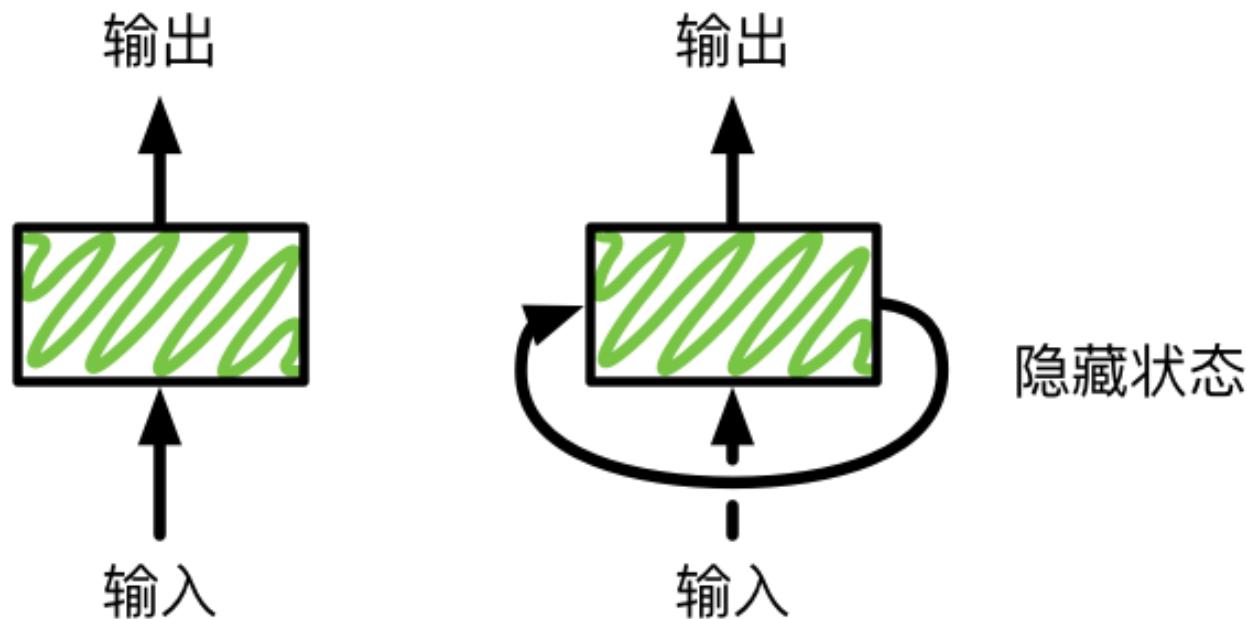
### 5.6.4 练习

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。)

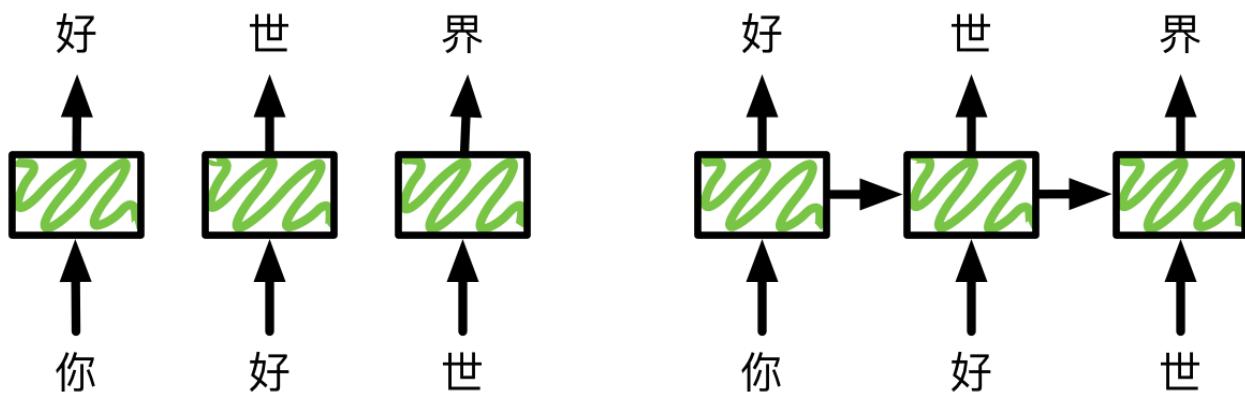
## 循环神经网络

### 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**。为什么叫前馈是整个网络是一条链（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。

在更加正式的介绍这个模型前，我们先去弄一个比“你好世界”稍微复杂点的数据。

### 6.1.1 《时间机器》数据集

我们用《时间机器》这本书做数据集主要是因为古登堡计划计划使得可以免费下载，而且我们看了太多用莎士比亚作为例子的教程。下面我们读取这个数据并看看前面 500 个字符（char）是什么样的：

```
In [1]: with open("data/timemachine.txt") as f:
    time_machine = f.read()
    print(time_machine[0:500])
```

The Time Machine, by H. G. Wells [1898]

I

The Time Traveller (for so it will be convenient to speak of him) was expounding a recondite matter to us. His grey eyes shone and twinkled, and his usually pale face was flushed and animated. The fire burned brightly, and the soft radiance of the incandescent lights in the lilies of silver caught the bubbles that flashed and passed in our glasses. Our chairs, being his patents, embraced and caressed us rather than submitted to be sat upon, and the

接着我们稍微处理下数据集。包括全部改为小写，去除换行符，然后截去后面一段使得接下来的训练会快一点。

```
In [2]: time_machine = time_machine.lower().replace('\n', '').replace('\r', '')
time_machine = time_machine[0:10000]
```

### 6.1.2 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 [3]: character_list = list(set(time_machine))
character_dict = dict([(char,i) for i,char in enumerate(character_list)])
vocab_size = len(character_dict)

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

vocab size: 43

'm': 0, "'": 1, 'w': 2, 'p': 3, 'k': 4, 't': 5, '8': 6, 'j': 7, 'o': 8, ')': 9, '9': 10, '

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

```
In [4]: time_numerical = [character_dict[char] for char in time_machine]

sample = time_numerical[:40]

print('chars: \n', ''.join([character_list[idx] for idx in sample]))
print('\nindices: \n', sample)
```

chars:

the time machine, by h. g. wells [1898]i

indices:

```
[5, 42, 18, 33, 5, 22, 0, 18, 33, 0, 20, 32, 42, 22, 24, 18, 12, 33, 40, 13, 33, 42, 39,
```

### 6.1.3 数据读取

同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 10，那么一个可能的样本是 `The Time T`。其对应的标号仍然是长为 10 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 `he Time Tr`。

下面代码每次从数据里随机采样一个批量：

```
In [5]: import random
        from mxnet import nd

        def data_iter(batch_size, seq_len, ctx=None):
            num_examples = (len(time_numerical)-1) // seq_len
            num_batches = num_examples // batch_size
            # 随机化样本
            idx = list(range(num_examples))
            random.shuffle(idx)
            # 返回 seq_len 个数据
            def _data(pos):
                return time_numerical[pos:pos+seq_len]
            for i in range(num_batches):
                # 每次读取 batch_size 个随机样本
                examples = idx[i:i+batch_size]
                data = nd.array(
                    [_data(j*seq_len) for j in examples], ctx=ctx)
                label = nd.array(
                    [_data(j*seq_len+1) for j in examples], ctx=ctx)
                yield data, label
```

看下读出来长什么样：

```
In [6]: for data, label in data_iter(batch_size=3, seq_len=8):
        print('data: ', data, '\n\nlabel:', label)
        break

data:
[[ 33.  2.  22.  16.  16.  24.  18.  36.]
```

```
[ 20.  32.   4.  33.   5.   8.  33.   5.]  
[ 18.  21.   3.  16.  22.  32.  22.   5.]]  
<NDArray 3x8 @cpu(0)>
```

```
label:  
[[ 2.  22.  16.  16.  24.  18.  36.  18.]  
 [ 32.   4.  33.   5.   8.  33.   5.  42.]  
 [ 21.   3.  16.  22.  32.  22.   5.  12.]]  
<NDArray 3x8 @cpu(0)>
```

### 6.1.4 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。

首先回忆下单隐层的前馈神经网络的定义，假设隐层的激活函数是  $\phi$ ，那么这个隐层的输出就是

$$H = \phi(XW_{wh} + b_h)$$

最终的输出是

$$\hat{Y} = \text{softmax}(HW_{hy} + b_y)$$

(跟多层感知机相比，这里我们把下标从  $W_1$  和  $W_2$  改成了意义更加明确的  $W_{wh}$  和  $W_{hy}$ )

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  $X_t$  是序列中的第  $t$  个输入，对应的隐层输出和最终输出是  $H_t$  和  $\hat{Y}_t$ 。循环神经网络只需要在计算隐层的输出的时候加上跟前一时间输入的加权和，为此我们引入一个新的可学习的权重  $W_{hh}$ ：

$$H_t = \phi(X_t W_{xh} + H_{t-1} W_{hh} + b_h)$$

输出的计算跟前一致：

$$\hat{Y}_t = \text{softmax}(H_t W_{hy} + b_y)$$

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。我们的输出是完全只基于这个状态。最开始的状态， $H_{-1}$ ，通常会被初始为 0.

### 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。就是说，如果值是  $i$ ，那么我们创建一个全 0 的长为 `vocab_size` 的向量，并将其第  $i$  位表示成 1.

```
In [7]: nd.one_hot(nd.array([0,4]), vocab_size)
```

Out[7]:

```
[[ 1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]  
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]  
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]  
[ 0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]  
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]  
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]  
<NDArray 2x43 @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

```
In [8]: 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:  8  
input[0] shape:  (3, 43)
```

## 6.1.6 初始化模型参数

模型的输入和输出维度都是 `vocab_size`。

```
In [9]: import mxnet as mx  
  
# 尝试使用 GPU  
import sys  
sys.path.append('..')  
import utils  
ctx = utils.try_gpu()  
print('Will use ', ctx)  
  
num_hidden = 256  
weight_scale = .01  
  
# 隐含层  
Wxh = nd.random_normal(shape=(vocab_size,num_hidden), ctx=ctx) * weight_scale  
Whh = nd.random_normal(shape=(num_hidden,num_hidden), ctx=ctx) * weight_scale
```

```

bh = nd.zeros(num_hidden, ctx=ctx)
# 输出层
Why = nd.random_normal(shape=(num_hidden,vocab_size), ctx=ctx) * weight_scale
by = nd.zeros(vocab_size, ctx=ctx)

params = [Wxh, Whh, bh, Why, by]
for param in params:
    param.attach_grad()

Will use gpu(0)

```

### 6.1.7 定义模型

我们将前面的模型公式定义直接写成代码。

```

In [10]: def rnn(inputs, H):
    # inputs: seq_len 个 batch_size x vocab_size 矩阵
    # H: batch_size x num_hidden 矩阵
    # outputs: seq_len 个 batch_size x vocab_size 矩阵
    outputs = []
    for X in inputs:
        H = nd.tanh(nd.dot(X, Wxh) + nd.dot(H, Whh) + bh)
        Y = nd.dot(H, Why) + by
        outputs.append(Y)
    return (outputs, H)

```

做个简单的测试:

```

In [11]: state = nd.zeros(shape=(data.shape[0], num_hidden), ctx=ctx)
outputs, state_new = rnn(get_inputs(data.as_in_context(ctx)), state)

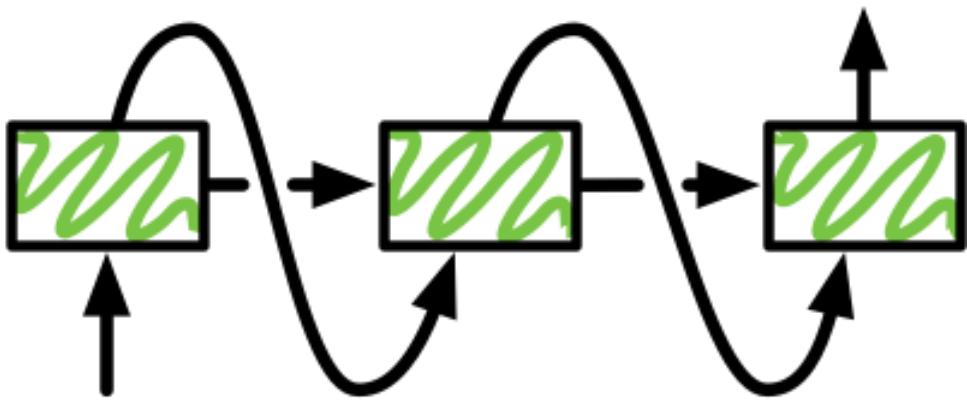
print('output length: ', len(outputs))
print('output[0] shape: ', outputs[0].shape)
print('state shape: ', state_new.shape)

output length: 8
output[0] shape: (3, 43)
state shape: (3, 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 [12]: def predict(prefix, num_chars):
    # 预测以 prefix 开始的接下来的 num_chars 个字符
    prefix = prefix.lower()
    state = nd.zeros(shape=(1, num_hidden), ctx=ctx)
    output = [character_dict[prefix[0]]]
    for i in range(num_chars+len(prefix)):
        X = nd.array([output[-1]], ctx=ctx)
        Y, state = rnn(get_inputs(X), state)
        #print(Y)
        if i < len(prefix)-1:
            next_input = character_dict[prefix[i+1]]
        else:
            next_input = int(Y[0].argmax(axis=1).asscalar())
        output.append(next_input)
    return ''.join([character_list[i] for i in output])
```

### 6.1.9 梯度剪裁

在求梯度时，循环神经网络因为需要反复做  $O(\text{seq\_len})$  次乘法，有可能会有数值稳定性问题。（想想  $2^{40}$  和  $0.5^{40}$ ）。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  $\theta$ ，那么我们这样剪裁使得  $\|g\|$  不会超过  $\theta$ ：

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

```
In [13]: 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，这里只有两个不同。

1. 通常我们使用 Perplexity(PPL) 这个指标。可以简单的认为就是对交叉熵做 exp 运算使得数值更好读。
2. 在更新前我们对梯度做剪裁

```

In [14]: 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()

for e in range(epochs+1):
    train_loss, num_examples = 0, 0
    state = nd.zeros(shape=(batch_size, num_hidden), ctx=ctx)
    for data, label in data_iter(batch_size, seq_len, ctx):
        with autograd.record():
            outputs, state = rnn(get_inputs(data), state)
            # reshape label to (batch_size*seq_len, )
            # concat outputs to (batch_size*seq_len, vocab_size)
            label = label.T.reshape((-1,))
            outputs = nd.concat(*outputs, dim=0)
            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. PPL %f" % (e, exp(train_loss/num_examples)))  
    print(' - ', predict('The Time Ma', 100))  
    print(' - ', predict("The Medical Man rose, came to the lamp,", 100), '\n')
```

Epoch 0. PPL 30.224488

- the time maa
  - the medical man rose, came to the lamp, eee

Epoch 20. PPL 10.761675



Epoch 40. PPL 8.413974

- the time mand for and ther and for the time time time time time time time time time ti
  - the medical man rose, came to the lamp, the time time time time time time time time ti

Epoch 60. PPL 7.498084

- the time may ine that is ore the time tion the the greatly anothe the time tion the th
  - the medical man rose, came to the lamp, ane that is ore the time tion the the greatly

Epoch 80. PPL 6.337197

- the time mall and the pare the gree dime spould mand ally a durthe time sion the pare
  - the medical man rose, came to the lamp,' so epsilon so and dis soply he the thing in an

Epoch 100. PPL 4.741698

- the time mather all yor can and that is all the time trave time at ure bote the prome
  - the medical man rose, came to the lamp, the trove than the threand what ingat is all t

Epoch 120. PPL 4.515792

- the time massin wery it and the list, and the list, and the list, and the list, and the
  - the medical man rose, came to the lamp, and the gith, and to ion anowhere las in thit

Epoch 140. PPL 3.779370

- the time matingations of is wead dinensions, and sone sour simensions, bet ofreat and
  - the medical man rose, came to the lamp,' in and the time traveller said the time trave

Epoch 160. PPL 2.991604

- the time mather mithed the thin the time traveller. thes so both thit mowe there then
  - the medical man rose, came to the lamp,' shacing the tree trace than a the three dimen

Epoch 180. PPL 2.624719

- the time mad in ane'breghthe very ald are arofthe bely ce thing is a moners, shiok is
- the medical man rose, came to the lamp, andithe this the mery ll andithe thing the eri

Epoch 200. PPL 2.705925

- the time mand fof that the time traveller.'it wo a fillly ho grove. helention.''ss anc
- the medical man rose, came to the lamp, 'no dion, and there wrong to eratll time trave

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（数据集大小，模型复杂度，学习率），看看对 Perplexity 和预测的结果造成的影响。

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）