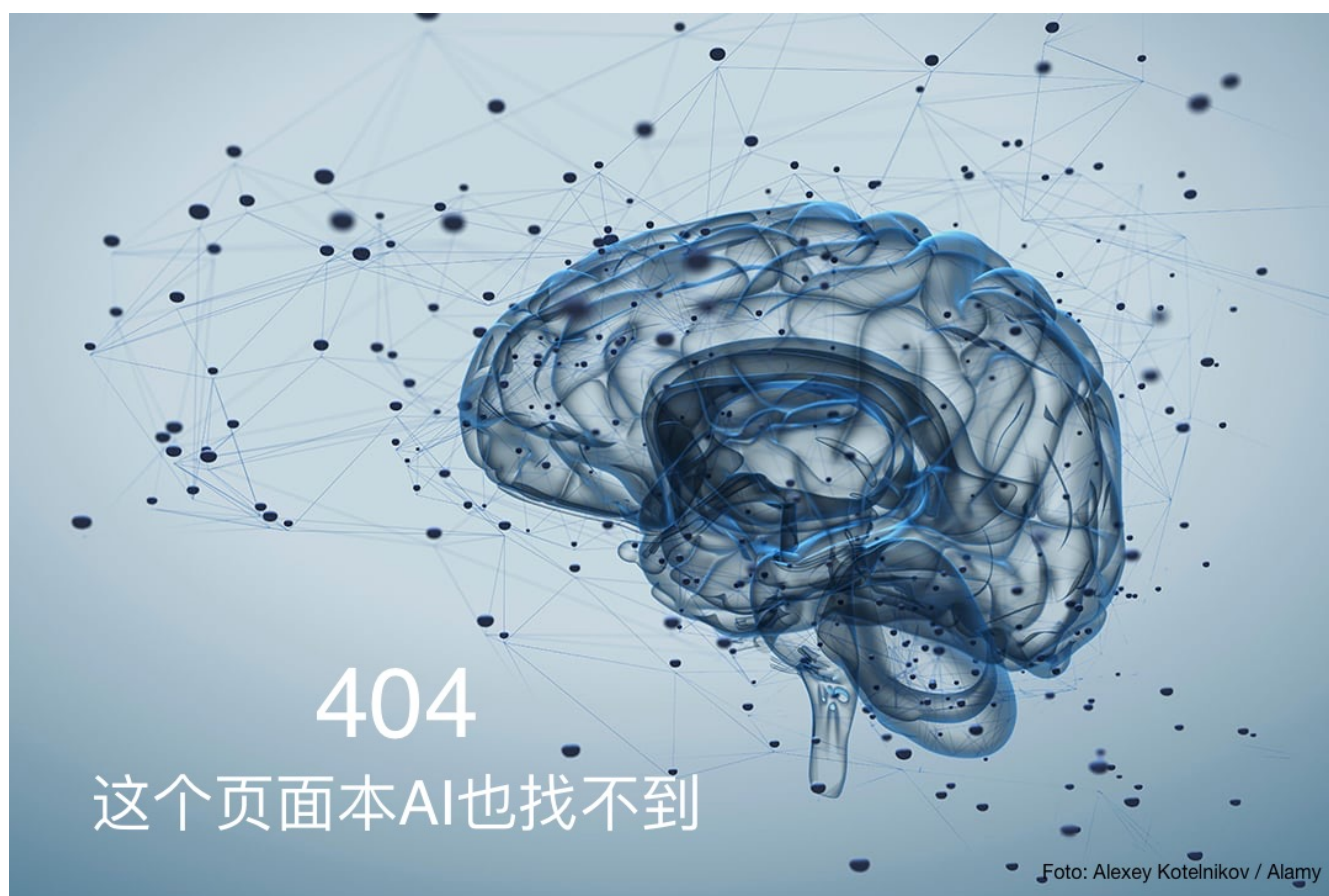


深度学习教材



作者：李沐团队

整理：gca

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。

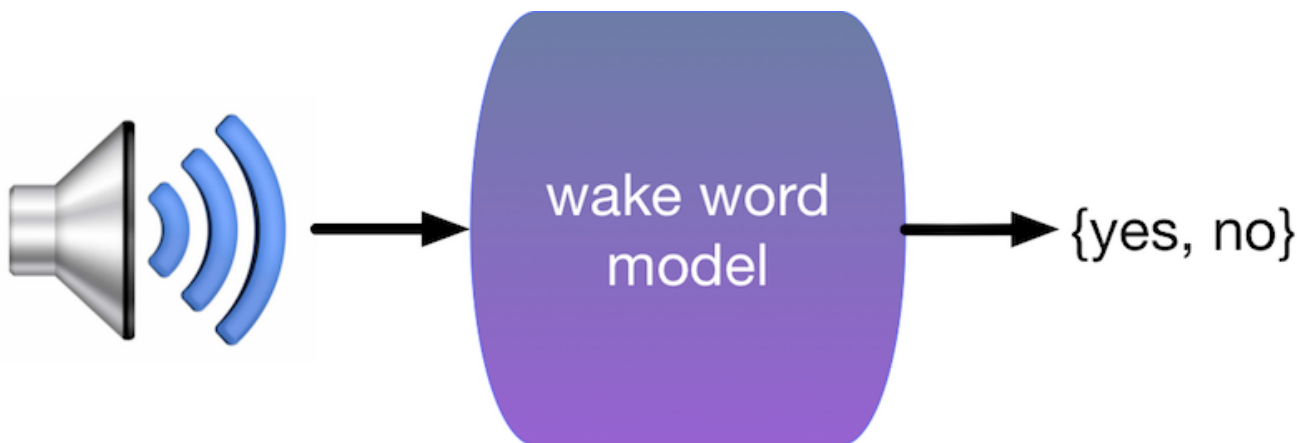
一个例子

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 == 'Okey, 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。

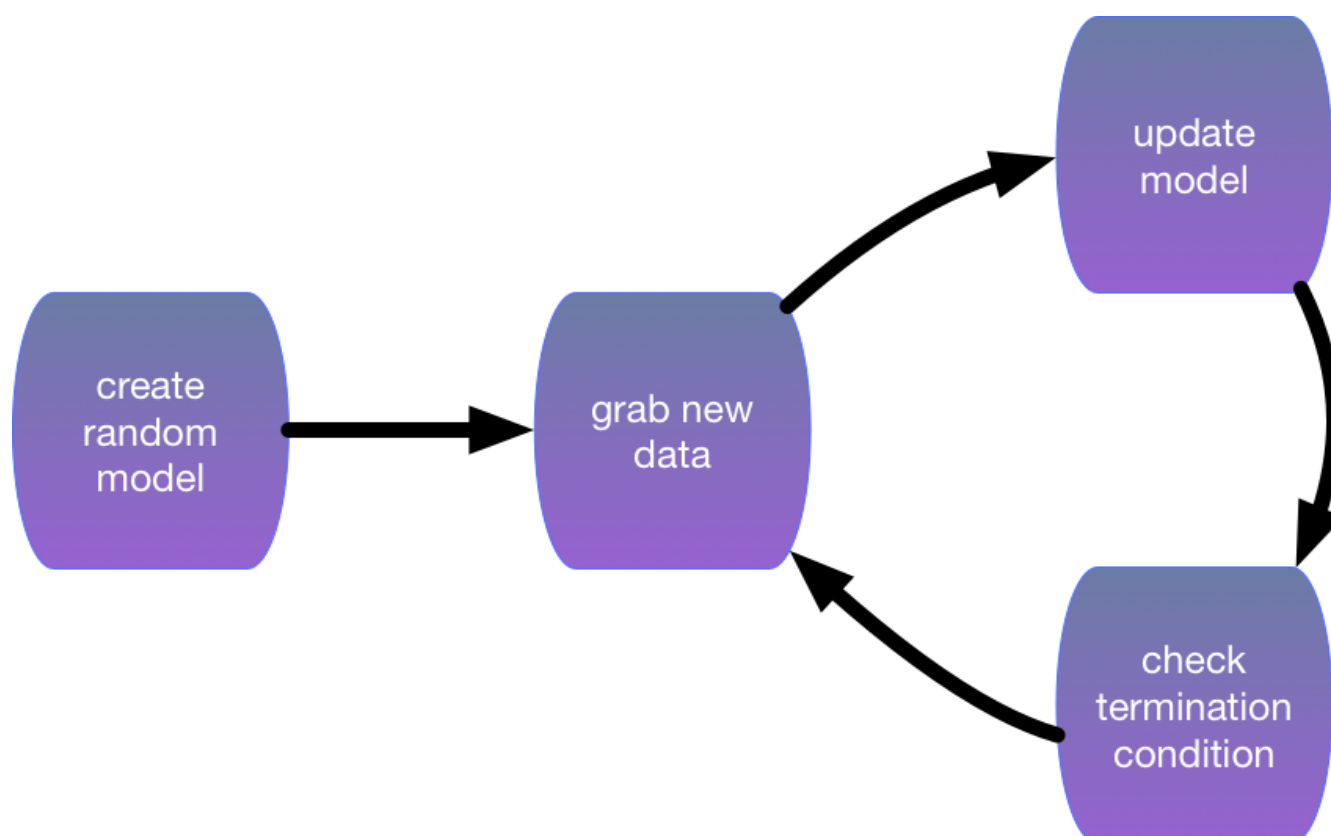


如果你被这个问题困住了，不用担心。这就是我们为什么要机器学习。

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。



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。

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：

```
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))
```

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

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

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

显然，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。

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，例如我们想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。

监督学习

监督学习描述的任务是，当给定输入 x ，如何通过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。

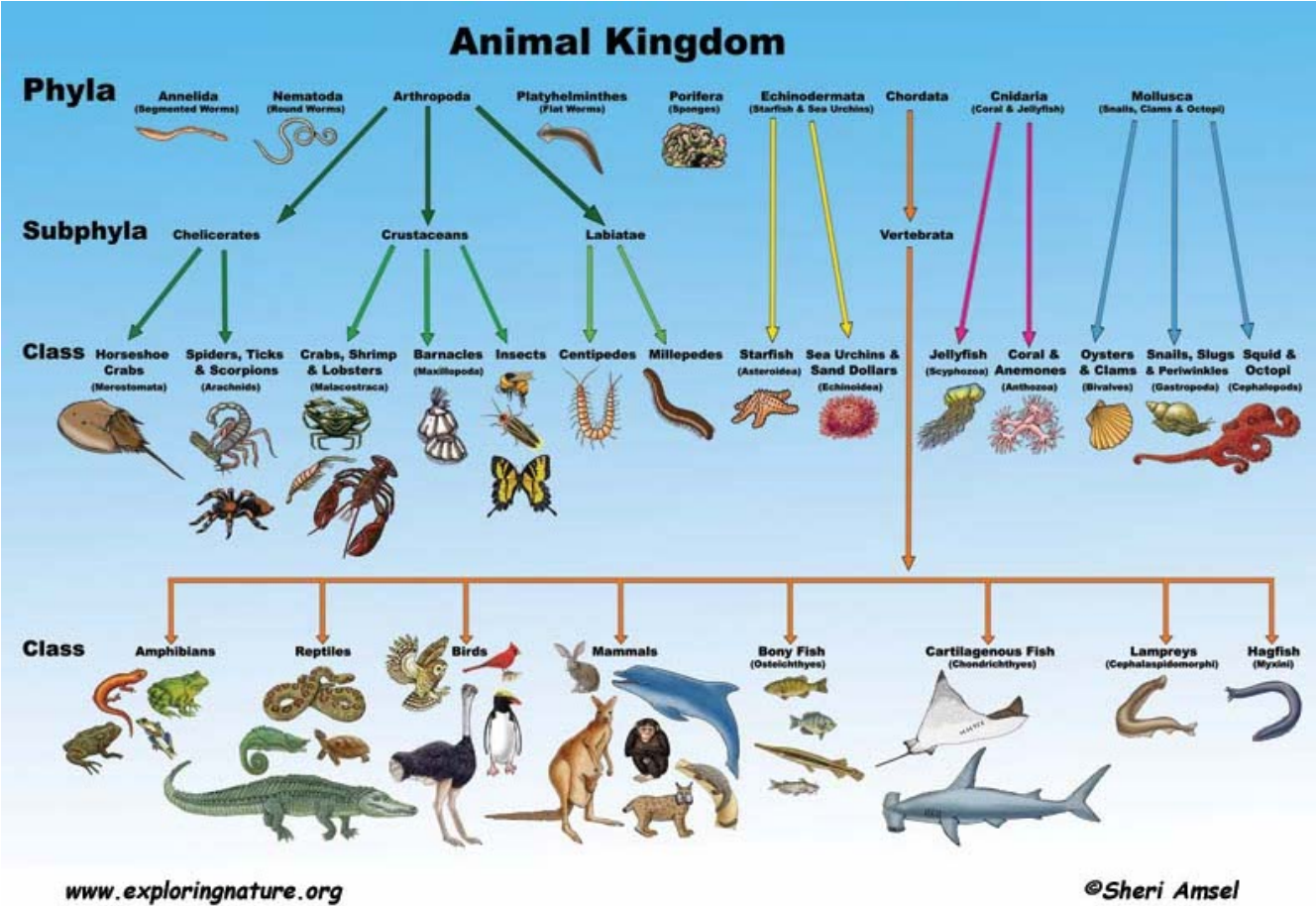
回归分析

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。



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。

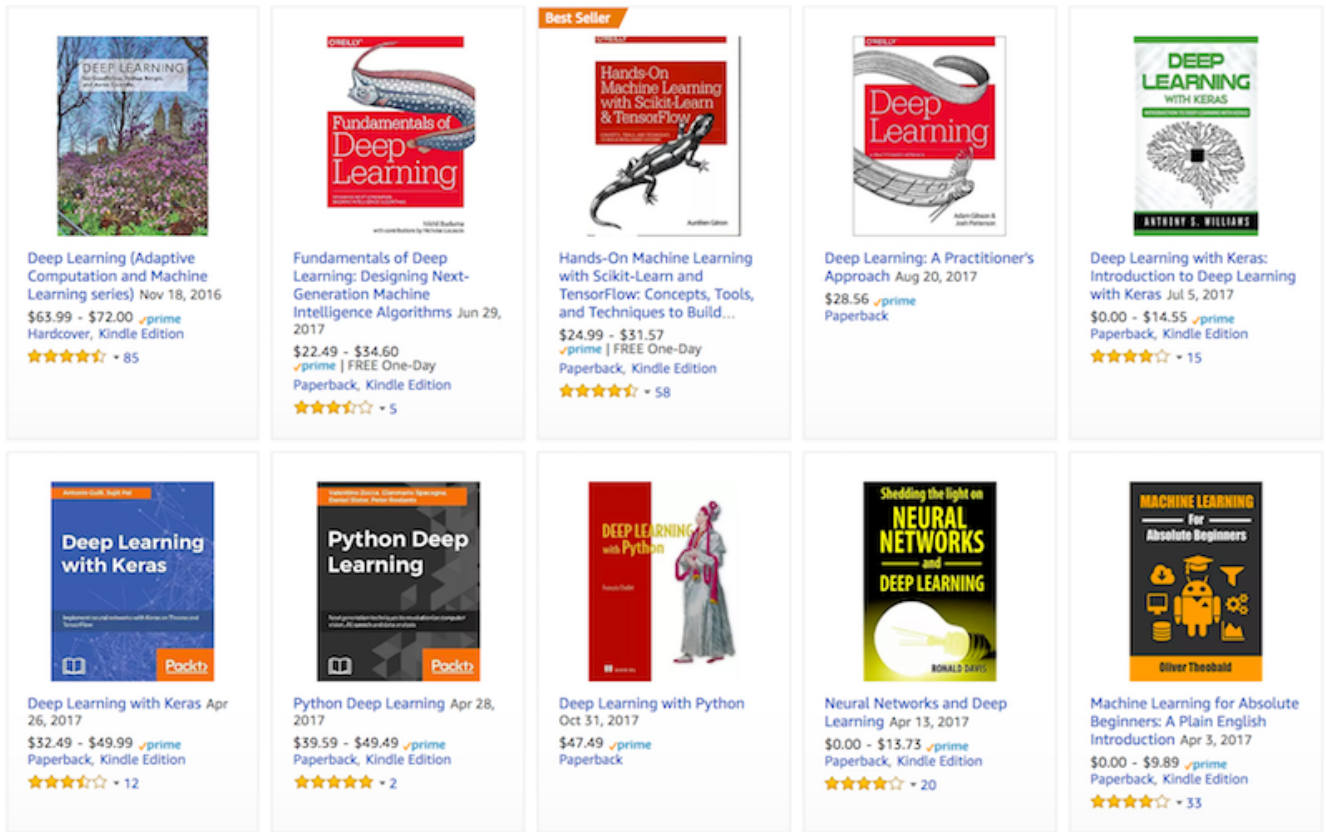
这类任务有时候也叫做多标签分类。想象一下，人们可能会把多个标签同时标注在自己的某篇技术类博客文章上，例如“机器学习”、“科技”、“编程语言”、“云计算”、“安全与隐私”和“AWS”。这里面的标签其实有时候相互关联，比如“云计算”和“安全与隐私”。当一篇文章可能被标注的数量很大时，人力标注就显得很吃力。这就需要使用机器学习了。

搜索与排序

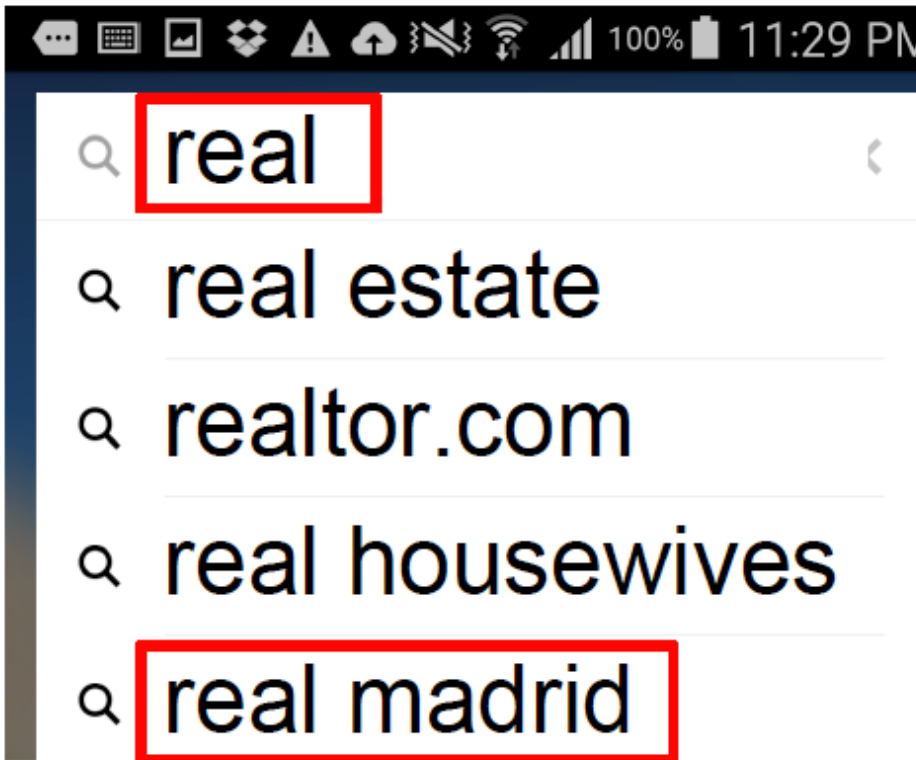
搜索与排序关注的问题更多的是如何把一堆对象排序。例如在信息检索领域，我们常常关注如何把一堆文档按照与检索条目的相关性排序。在互联网时代，由于搜索引擎的流行，我们更加关注如何对网页进行排序。互联网时代早期有一个著名的网页排序算法叫做 **PageRank**。该算法的排序结果并不取决于特定的用户检索条目。这些排序结果可以更好地为所包含检索条目的网页进行排序。

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中，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。

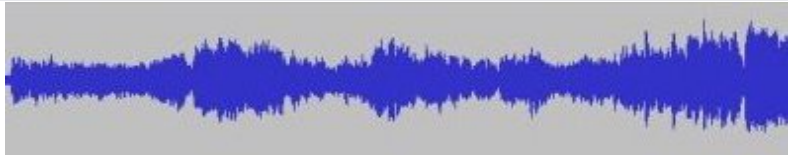
语法分析

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。

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

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。

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。

下面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。

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.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。

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

```
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

```
nd.zeros((3, 4))
```

类似的，我们可以创建数组每个元素被初始化成1。

```
x = nd.ones((3, 4))  
x
```

或者从python的数组直接构造

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

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。

```
y = nd.random_normal(0, 1, shape=(3, 4))  
y
```

跟 `NumPy` 一样，每个数组的形状可以通过 `.shape` 来获取

```
y.shape
```

它的大小，就是总元素个数，是形状的累乘。

```
y.size
```

操作符

NDArray支持大量的数学操作符，例如按元素加法：

```
x + y
```

乘法：

```
x * y
```

指数运算：

```
nd.exp(y)
```

也可以转秩一个矩阵然后计算矩阵乘法：

```
nd.dot(x, y.T)
```

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：

```
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)
```

跟NumPy的转换

ndarray可以很方便同numpy进行转换

```
import numpy as np
x = np.ones((2, 3))
y = nd.array(x) # numpy -> mxnet
z = y.asnumpy() # mxnet -> numpy
print([z, y])
```

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：

```
x = nd.ones((3, 4))
y = nd.ones((3, 4))

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

我们可以把结果通过 `[:]` 写到一个之前开好的数组里：

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

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：

```
nd.elemwise_add(x, y, out=z)
id(z) == before
```

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：

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

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](#)

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`。

```
import mxnet.ndarray as nd
import mxnet.autograd as ag
```

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。

```
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。

```
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记录我们需要求导的程序。

```
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()`。

```
z.backward()
```

现在我们来查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 $dz/dx = 4 * x$ 。

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

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）。

```
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。

```
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：

```
a.grad == c/a
```

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](#)。

```
with ag.record():
    y = x * 2
    z = y * x

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

吐槽和讨论欢迎点[这里](#)

4.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。

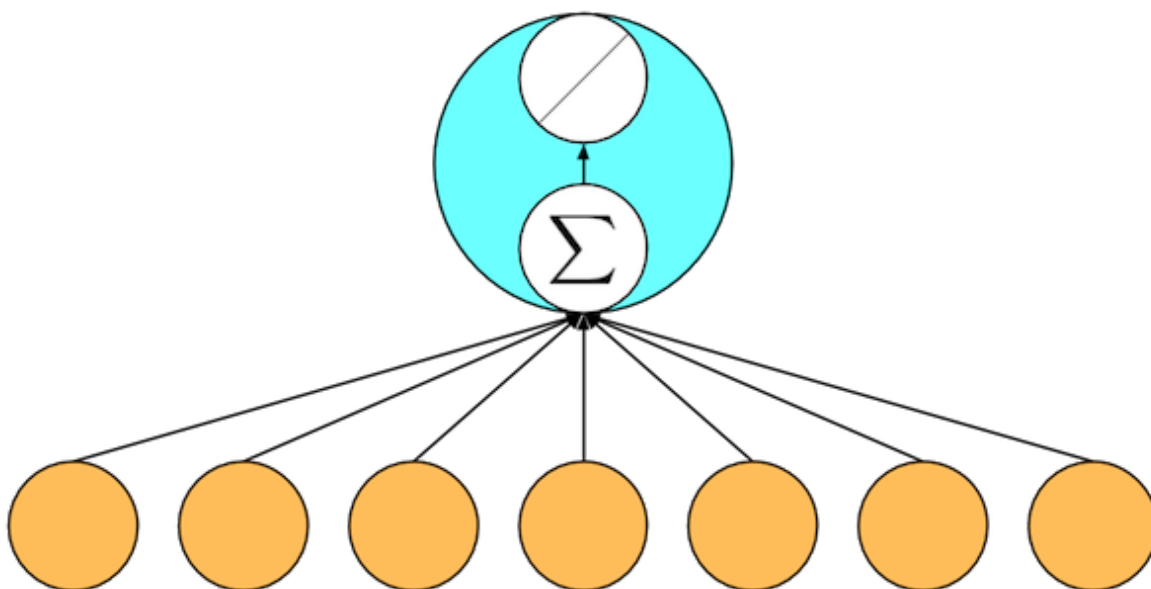
线性回归

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求和，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。



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。

```
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）。

```
print(X[0], y[0])
```

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。

```
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

```
for data, label in data_iter():
    print(data, label)
    break
```

初始化模型参数

下面我们随机初始化模型参数

```
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。

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

定义模型

线性模型就是将输入和模型做乘法再加上偏移：

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

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。

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

优化

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


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

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。

```
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))
```

训练完成后我们可以比较学到的参数和真实参数

```
true_w, w
```

```
true_b, b
```

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。

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](#)

5.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`。

创建数据集

我们生成同样的数据集

```
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)
```

数据读取

但这里使用 `data` 模块来读取数据。

```
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：

```
for data, label in data_iter:
    print(data, label)
    break
```

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：

```
net = gluon.nn.Sequential()
```

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

```
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。）

初始化模型参数

在使用前 `net` 我们必须初始化模型权重，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）。

```
net.initialize()
```

损失函数

`gluon` 提供了平方误差函数：

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

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。

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

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。

```
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))
```

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。

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

```
true_b, dense.bias.data()
```

结论

可以看到 `gluon` 可以帮助我们更快更干净地实现模型。

练习

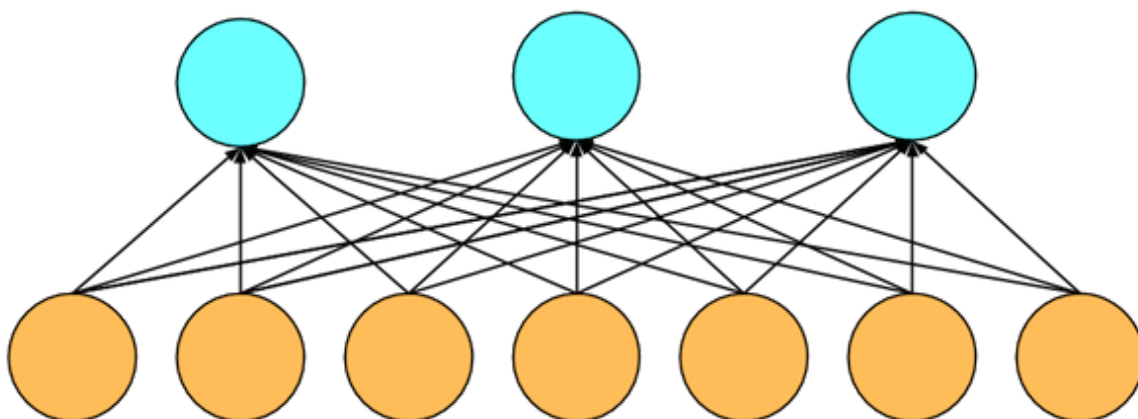
- 在训练的时候，为什么我们用了比前面要大10倍的学习率呢？（提示：可以尝试运行 `help(trainer.step)` 来寻找答案。）
- 如何拿到 `weight` 的梯度呢？（提示：尝试 `help(dense.weight)`）

吐槽和讨论欢迎[点这里](#)

6.多类逻辑回归 --- 从0开始

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

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



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。

```
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


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

我们画出前几个样本的内容，和对应的文本标号

```
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))
```

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.DataLoader` 函数，它每次 `yield` 一个批量。

```
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。

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：

```
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：

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

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。但在这里的分类里，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 $\hat{y} = \mathbf{W} \mathbf{x}$ ，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。

```
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。

```
X = nd.random_normal(shape=(2, 5))
X_prob = softmax(X)
print(X_prob)
print(X_prob.sum(axis=1))
```

现在我们可以定义模型了：

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

交叉熵损失函数

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

具体来说，我们先将真实标号表示成一个概率分布，例如如果 `y=1`，那么其对应的分布就是一个除了第二个元素为1其他全为0的长为10的向量，也就是 `yvec=[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]`。那么交叉熵就是 `yvec[0]*log(yhat[0])+...+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

```
def cross_entropy(yhat, y):
    return - nd.pick(nd.log(yhat), y)
```

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：

```
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`。）

```
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`。

```
evaluate_accuracy(test_data, net)
```

训练

训练代码跟前面的线性回归非常相似：

```
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))
```

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

```
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()))
```

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。

练习

尝试增大学习率，你会发现结果马上回变成很糟糕，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](#)

7.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。

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。

```
import sys
sys.path.append('.')
import utils

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

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会自动推导。

```
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()
```

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

```
softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()
```

优化

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

训练

```
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))
```

结论

Gluon提供的函数有时候比手工写的数值更稳定。

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](#)

8.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。

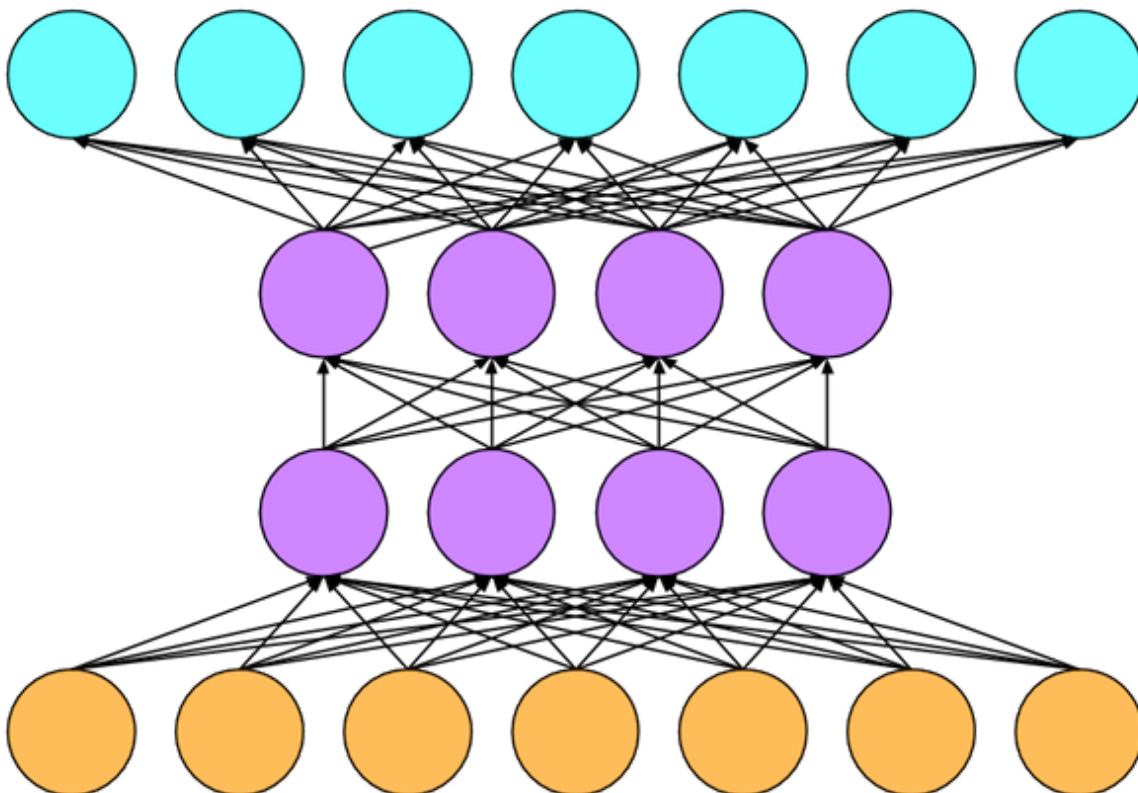
数据获取

我们继续使用FashionMNIST数据集。

```
import sys
sys.path.append('.')
import utils
batch_size = 256
train_data, test_data = utils.load_data_fashion_mnist(batch_size)
```

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。

```
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()

```

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

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

```

def relu(X):
    return nd.maximum(X, 0)

```

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：

```

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

```

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

```

from mxnet import gluon
softmax_cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()

```

训练

训练跟之前一样。

```
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))
```

总结

可以看到，加入一个隐含层后我们将精度提升了不少。

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_hidden` 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](#)

9.多层感知机 --- 使用Gluon

我们只需要稍微改动[多类Logistic回归](#)来实现多层感知机。

定义模型

唯一的区别在这里，我们加了一行进来。

```
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()
```

读取数据并训练

```
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))
```

结论

通过Gluon我们可以更方便地构造多层神经网络。

练习

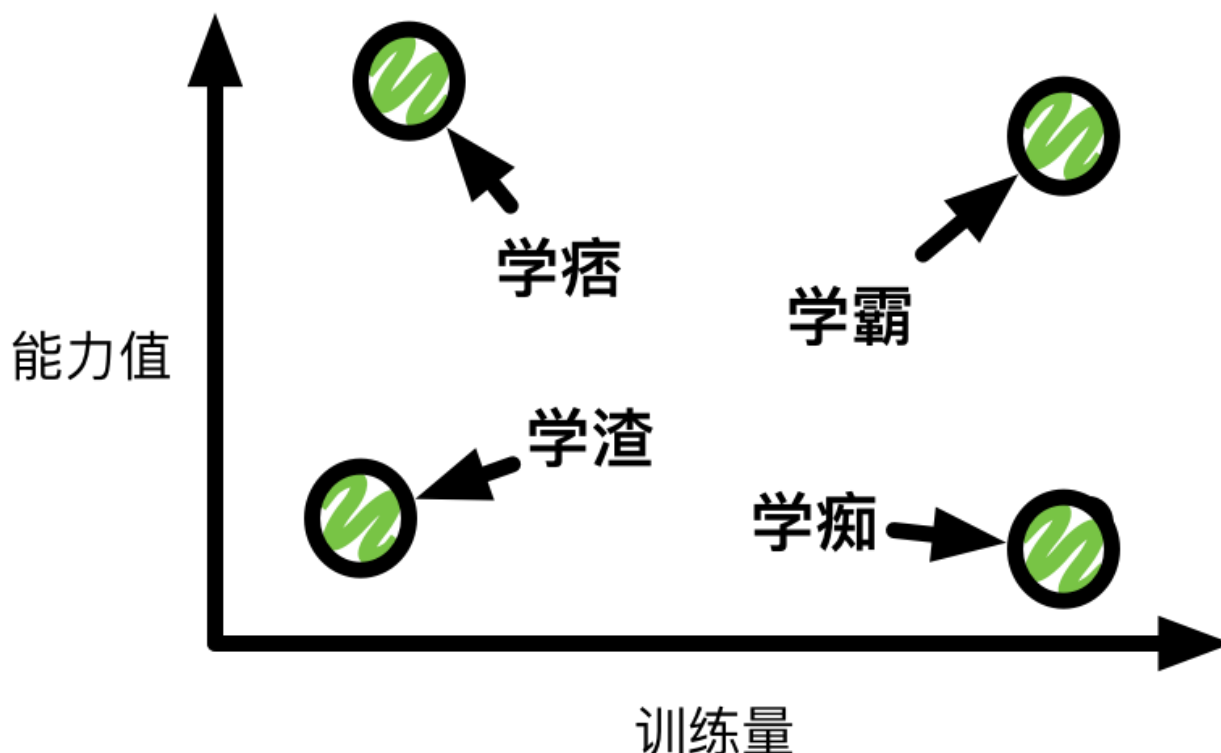
- 尝试多加入几个隐含层，对比从0开始的实现。
- 尝试使用一个另外的激活函数，可以使用 `help(nd.Activation)` 或者[线上文档](#)查看提供的选项。

吐槽和讨论欢迎[点这里](#)

10.欠拟合和过拟合

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

有人曾依据这种现象对学生群体简单粗暴地做了如下划分：



这里简要总结上图中四类学生的特点：

- 学渣：训练量较小，学习能力较差，容易考挂
- 学痞：训练量较小，学习能力较强，通常考不过学霸但比学渣好
- 学痴：训练量较大，学习能力较差，通常考不过学霸但比学渣好
- 学霸：训练量较大，学习能力较强，容易考好

（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。

训练误差和泛化误差

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。

欠拟合和过拟合

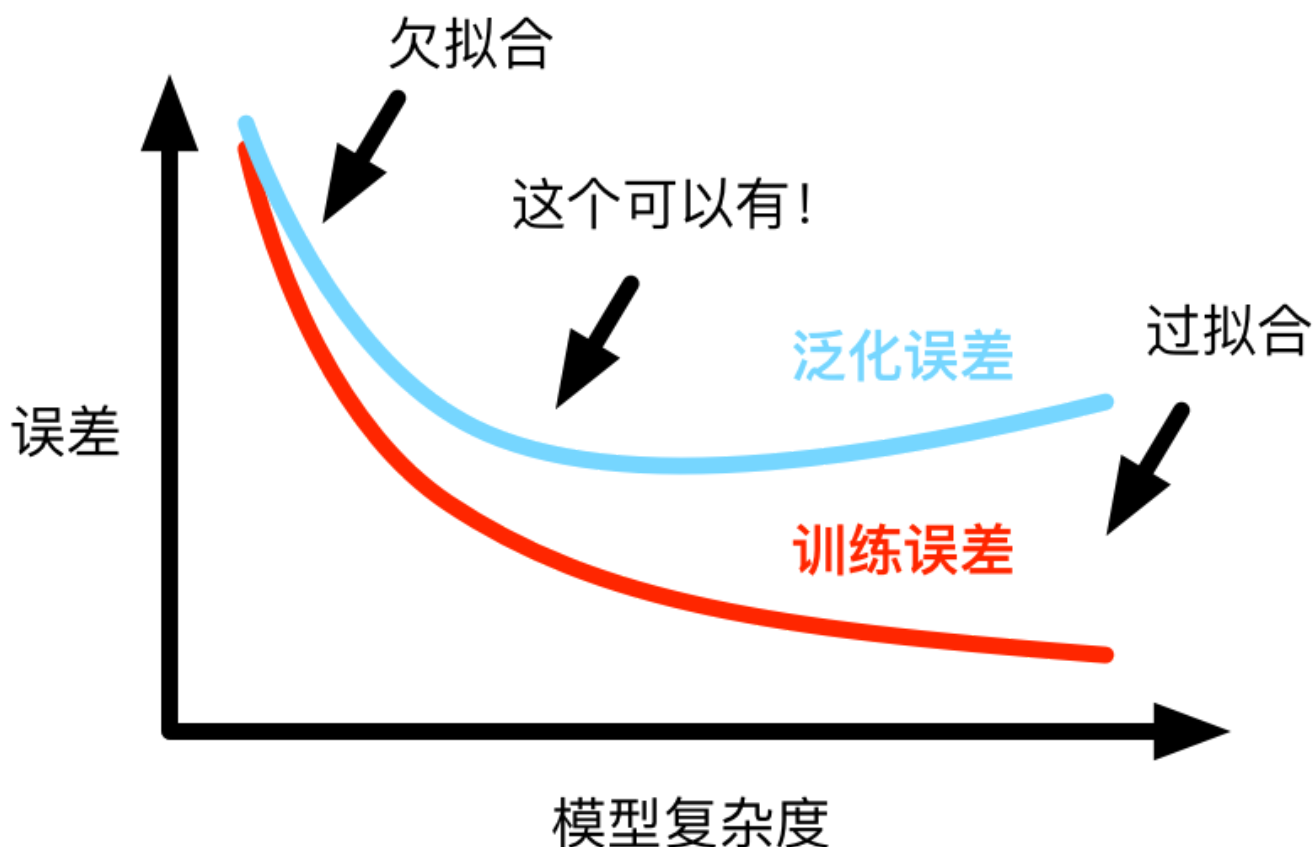
实践中，如果测试数据集是给定的，我们通常用机器学习模型在该测试数据集的误差来反映泛化误差。基于上述重要结论，以下两种拟合问题值得注意：

- **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。

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。

```
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。

```
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:]
```

我们把损失函数定义为平方误差。

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

下面定义模型和数据读取器。

```
def get_net_and_iter(X_train, y_train, batch_size):
    dataset_train = gluon.data.ArrayDataset(X_train, y_train)
    data_iter_train = gluon.data.DataLoader(dataset_train, batch_size, shuffle=True)
    net = gluon.nn.Sequential()
```

```
with net.name_scope():
    net.add(gluon.nn.Dense(1))
return net, data_iter_train
```

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。

```
def train(net, data_iter_train, lr, cur_loss, epochs, verbose_epoch, batch_size):
    net.collect_params().initialize(force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
    for epoch in range(epochs):
        total_loss = 0
        for data, label in data_iter_train:
            with autograd.record():
                output = net(data)
                loss = cur_loss(output, label)
            loss.backward()
            trainer.step(batch_size)
            total_loss += nd.sum(loss).asscalar()
        if epoch >= verbose_epoch:
            print("Epoch %d, train loss: %f" % (epoch, total_loss / y_train.shape[0]))
    return net
```

以下是测试步骤。

```
def test(X_test, y_test, net, cur_loss):
    loss_test = nd.sum(cur_loss(net(X_test), y_test)).asscalar() / \
        y_test.shape[0]
    print("Test loss: %f" % loss_test)
    print("True params: ", true_w, true_b)
    print("Learned params: ", net[0].weight.data(), net[0].bias.data())
```

机器学习全过程包含训练和测试步骤。

```
def learn(X_train, X_test, y_train, y_test, lr, cur_loss):
    epochs = 100
    verbose_epoch = 95
    batch_size = min(10, X_train.shape[0])
    net, data_iter_train = get_net_and_iter(X_train, y_train, batch_size)
    net_trained = train(net, data_iter_train, lr, cur_loss, epochs,
                        verbose_epoch, batch_size)
    test(X_test, y_test, net_trained, cur_loss)
```

三阶多项式拟合

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。

```
X_train_ord3, X_test_ord3 = X[:num_train, :], X[num_train:, :]

learning_rate = 0.025
learn(X_train_ord3, X_test_ord3, y_train, y_test, learning_rate, square_loss)
```

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。

```
x_train_ord1, x_test_ord1 = x[:num_train, :], x[num_train:, :]  
  
learning_rate = 0.025  
learn(x_train_ord1, x_test_ord1, y_train, y_test, learning_rate, square_loss)
```

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。

```
y_train, y_test = y[0:2], y[num_train:]  
X_train_ord3, X_test_ord3 = X[0:2, :], X[num_train:, :]  
  
learning_rate = 0.025  
learn(X_train_ord3, X_test_ord3, y_train, y_test, learning_rate, square_loss)
```

我们还将在今后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。

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。

练习

- 如果用一个三阶多项式模型来拟合一个线性模型生成的数据，可能会有什么问题？为什么？
- 在我们本节提到的三阶多项式拟合问题里，有没有可能把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](#)

11.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。

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 $p=200$ 。

```
from mxnet import ndarray as nd
from mxnet import autograd
from mxnet import gluon

num_train = 20
num_test = 1000
num_inputs = 200
```

生成数据集

这里定义模型真实参数。

```
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。

```
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。

```
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)
```

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。

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

\$L_2\$范数正则化

线性模型就是将输入和模型做乘法再加上偏移。

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。在训练模型时，如果 $\lambda = 0$ ，则未使用正则化。需要注意的是，在测试模型时， λ 必须为0。

```
def net(X, lambd, w, b):
    return nd.dot(X, w) + b + lambd * ((w**2).sum() + b**2)
```

损失函数和优化算法

我们使用平方误差和随机梯度下降。

```
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
```

定义训练和测试

我们定义一个训练函数，这样在跑不同的实验时不需要重复实现相同的步骤。

```
import matplotlib.pyplot as plt

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

def train(params, epochs, learning_rate, lambd):
    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)
            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 params
```

以下函数将调用模型的定义、训练和测试。


```
def learn(learning_rate, lambd):  
    epochs = 10  
    params = get_params()  
    (w, b) = train(params, epochs, learning_rate, lambd)  
    print('Learned w[:10]:', w[:10])  
    print('Learned b:', b)
```

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。

```
learning_rate = 0.0025  
lambd = 0  
learn(learning_rate, lambd)
```

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。

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。

```
lambd = 0.1  
learn(learning_rate, lambd)
```

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。

结论

- 我们可以使用正则化来应对过拟合问题。

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](#)

12.正则化 --- 使用Gluon

本章介绍如何使用 `Gluon` 的正则化来应对过拟合问题。

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。

```
from mxnet import ndarray as nd
from mxnet import autograd
from mxnet import gluon

num_train = 20
num_test = 1000
num_inputs = 200
```

这里定义模型真实参数。

```
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。

```
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:]
```

这里使用 `data` 模块来读取数据。

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

我们把损失函数定义为平方误差。

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

定义模型

我们将模型的定义放在一个函数里供多次调用。

```
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。

你也许发现了，`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。

```
import matplotlib.pyplot as plt

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

def train(net, epochs, learning_rate, weight_decay):
    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 net
```

以下函数将调用模型的定义、训练和测试。

```
def learn(learning_rate, weight_decay):
    epochs = 10
    net = get_net()
    net_trained = train(net, epochs, learning_rate, weight_decay)
```

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。

```
learn(learning_rate = 0.005, weight_decay = 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。

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。

```
learn(learning_rate = 0.005, weight_decay = 1)
```

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。

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。

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](#)

13. 创建神经网络

前面的教程我们教大家如何实现线性回归，多类Logistic回归和多层感知机。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）。

```
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)
```

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。

```
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。

```
net2 = MLP()
print(net2)
net2.initialize()
x = nd.random.uniform(shape=(4, 20))
```

```
y = net2(x)
y
```

```
nn.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。默认会自动会自动生成前缀，我们也可以在创建的时候手动指定。

```
print('default prefix:', net2.dense0.name)

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

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。

`nn.Block` 到底是什么东西？

在 `glueon` 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

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：

```
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：

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

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

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

```
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了权重 `weight`，并重复用了 `dense` 的层。测试一下：

```
fancy_mlp = FancyMLP()
fancy_mlp.initialize()
y = fancy_mlp(x)
print(y.shape)
```

`nn.Block` 和 `nn.Sequential` 的嵌套使用

现在我们知道了 `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。

```
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)
```

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。

练习

如果把 `RecMLP` 改成 `self.dense = [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](#)

14.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。

```
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。

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

正确的打开方式是这样

```
net.initialize()
net(x)
```

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：

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

然后我们可以通过 `data` 来访问参数，`grad` 来访问对应的梯度

```
print('weight:', w.data())
print('weight gradient', w.grad())
print('bias:', b.data())
print('bias gradient', b.grad())
```

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。

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

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

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

看得更加清楚点：

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

更多的方法参见[init的API](#)。下面我们自定义一个初始化方法。

```
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())
```

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。

```
net = get_net()
print(net.collect_params())
```

然后我们初始化

```
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。

```
net(x)
```

这时候我们看到shape里面的0被填上正确的值了。

```
print(net.collect_params())
```

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。

```
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())
```

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。

```
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

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

总结

我们可以很灵活地访问和修改模型参数。

练习

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](#)

15.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。

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。

```
from mxnet import nd

x = nd.ones(3)
y = nd.zeros(4)
filename = "../data/test1.params"
nd.save(filename, [x, y])
```

读回来

```
a, b = nd.load(filename)
print(a, b)
```

不仅可以读写单个NDArray，NDArray list，dict也是可以的：

```
mydict = {"x": x, "y": y}
filename = "../data/test2.params"
nd.save(filename, mydict)
```

```
c = nd.load(filename)
print(c)
```

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

```
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))
```

下面我们把模型参数存起来

```
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。

```
import mxnet as mx
net2 = get_net()
net2.load_params(filename, mx.cpu()) # FIXME, gluon will support default ctx later
print(net2(x))
```

总结

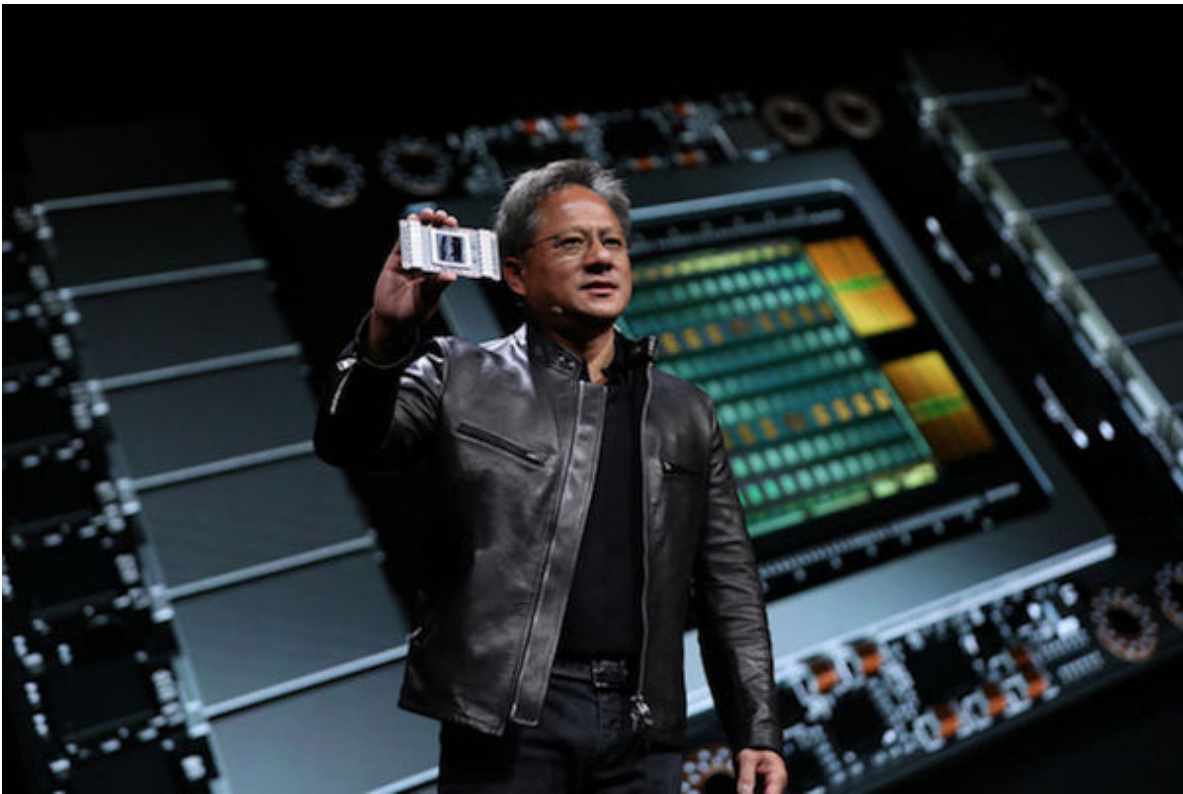
通过 `load_params` 和 `save_params` 可以很方便的读写模型参数。

16.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。



(黄教主和他的战术核武器)

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%`）。

```
!nvidia-smi
```

接下来要确认正确安装了的 `mxnet` 的GPU版本。具体来说卸载了 `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：

```
import pip
for pkg in ['mxnet', 'mxnet-cu75', 'mxnet-cu80']:
    pip.main(['show', pkg])
```

Context

MXNet使用Context来指定使用哪个设备来存储和计算。默认会将数据开在主内存，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。如果有多块GPU，我们用 `mx.gpu(i)` 来表示第*i*块GPU（*i*从0开始）。

```
import mxnet as mx
[mx.cpu(), mx.gpu(), mx.gpu(1)]
```

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。

```
from mxnet import nd
x = nd.array([1,2,3])
x.context
```

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）：

```
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)
```

尝试将内存开到另外一块GPU上。如果不存在会报错：

```
import sys

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

我么可以通过 `copyto` 和 `as_in_context` 来在设备直接传输数据。

```
y = x.copyto(mx.gpu())
z = x.as_in_context(mx.gpu())
(y, z)
```

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：

```
yy = y.as_in_context(mx.gpu())
zz = z.copyto(mx.gpu())
(yy is y, zz is z)
```

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：

```
nd.exp(z + 2) * y
```

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。

```
try:
    x + y
except mx.MXNetError as err:
    sys.stderr.write(str(err))
```

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。

```
print(y)
print(y.asnumpy())
print(y.sum().asscalar())
```

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：

```
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

```
data = nd.random.uniform(shape=[3, 2], ctx=mx.gpu())
net(data)
```

确认下权重：

```
net[0].weight.data()
```

总结

通过 `context` 我们可以很容易在不同的设备上计算。

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](#)

17.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。

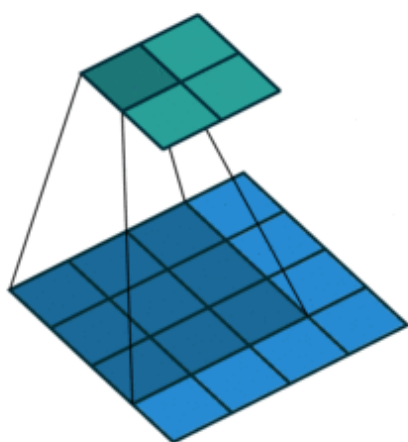
这一节我们介绍卷积神经网络，其有效了解决了上述两个问题。

卷积神经网络

卷积神经网络是指主要由卷积层构成的神经网络。

卷积层

卷积层跟前面的全连接层类似，但输入和权重不是做简单的矩阵乘法，而是使用每次作用在一个窗口上的卷积。下图演示了输入是一个 4×4 矩阵，使用一个 3×3 的权重，计算得到 2×2 结果的过程。每次我们采样一个跟权重一样大小的窗口，让它跟权重做按元素的乘法然后相加。通常我们也是用卷积的术语把这个权重叫kernel或者filter。



（图片版权属于vdumoulin@github）

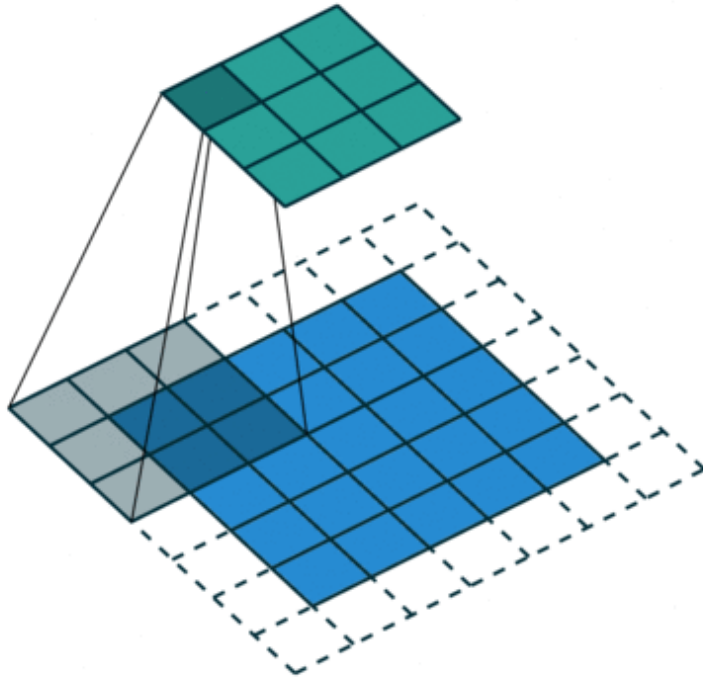
我们使用 `nd.Convolution` 来演示这个。

```
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都是1。
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)
```

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`。



```
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)
```

当输入数据有多个通道的时候，每个通道会有对应的权重，然后会对每个通道做卷积之后在通道之间求和

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

```
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)
```

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。

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

```
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)
```

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。

```
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)
```

下面我们可以开始使用这些层构建模型了。

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）

```
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)
```

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。

```
import mxnet as mx

try:
    ctx = mx.gpu()
    _ = nd.zeros((1, ), ctx=ctx)
except:
    ctx = mx.cpu()

ctx
```

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：

```
weight_scale = .01
num_outputs = 10
num_fc = 128

# 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。

```
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。

```
for data, _ in train_data:
    net(data, verbose=True)
    break
```

训练

跟前面没有什么不同的

```
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))

```

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给某大公司了。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。

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](#)

18.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](#)。

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。

```

from mxnet import gluon

net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.nn.Conv2D(channels=20, kernel_size=5, activation='relu'))
    net.add(gluon.nn.MaxPool2D(pool_size=2, strides=2))
    net.add(gluon.nn.Conv2D(channels=50, kernel_size=3, activation='relu'))
    net.add(gluon.nn.MaxPool2D(pool_size=2, strides=2))
    net.add(gluon.nn.Flatten())
    net.add(gluon.nn.Dense(128, activation="relu"))
    net.add(gluon.nn.Dense(10))

```

然后我们尝试将模型权重初始化在GPU上

```

import sys
sys.path.append('.')

```



```
import utils

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

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

获取数据然后训练

跟之前没什么两样。

```
from mxnet import autograd as autograd
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))
```

结论

使用Gluon来实现卷积网络轻松加随意。

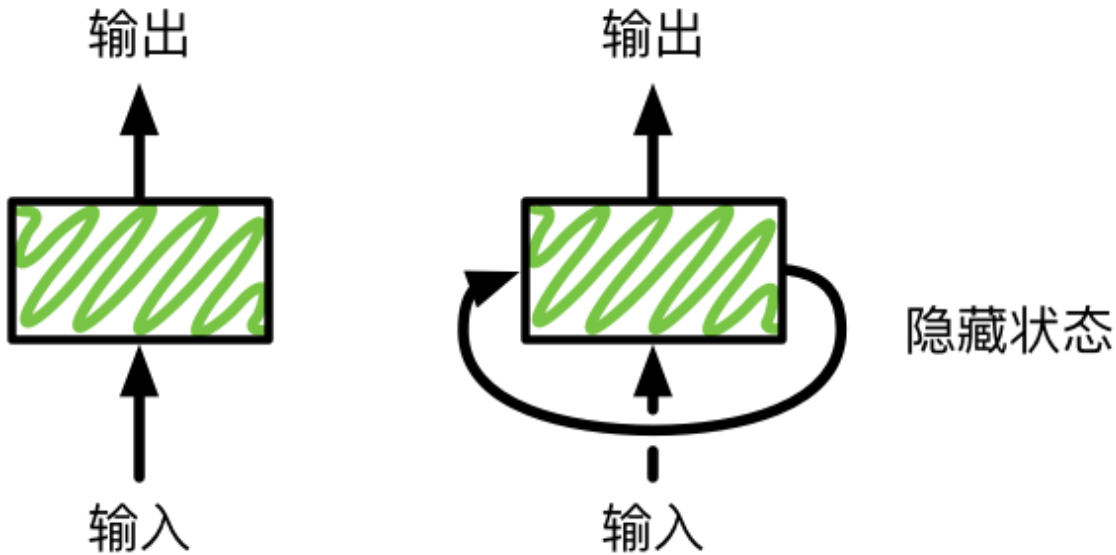
练习

再试试改改卷积层设定，是不是会比上一章容易很多？

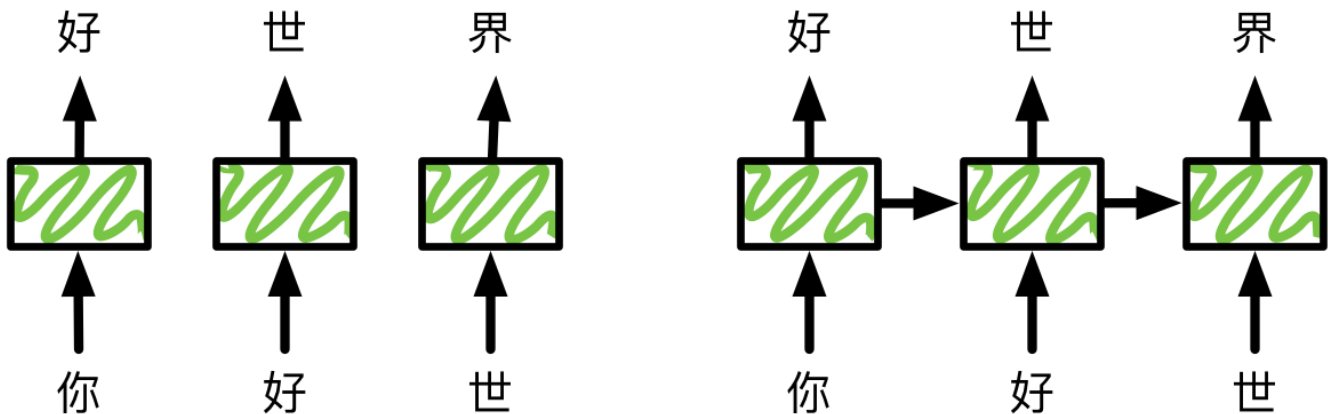
吐槽和讨论欢迎点[这里](#)

19.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。



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

循环神经网络使用一个隐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。

《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：

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

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。

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

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：

```
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)
```

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。

```
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)
```

数据读取

同前一样我们需要每次随机读取一些（`batch_size` 个）样本和其对应的标号。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`）设为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：

```
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：

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

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 ϕ ，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} ：

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

输出的计算跟前一致：

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

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。

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。

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

记得前面我们每次得到的数据是一个 `batch_size x 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 x vocab_size` 的矩阵

```
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)
```

初始化模型参数

模型的输入和输出维度都是 `vocab_size`。

```
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()
```

定义模型

我们将前面的模型公式定义直接写成代码。

```
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：

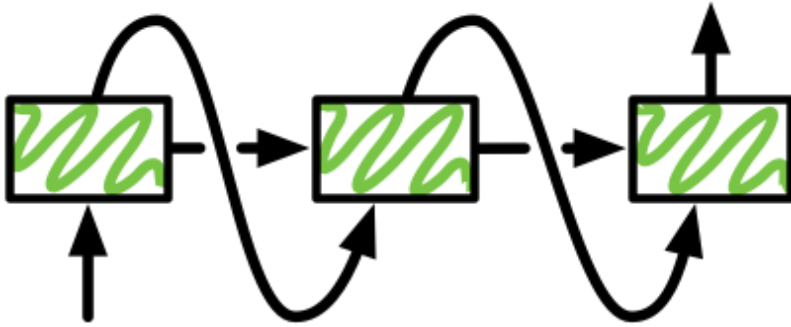
```
state = nd.zeros(shape=(data.shape[0], num_hidden), ctx=ctx)
outputs, state_new = rnn(get_inputs(data.as_in_context(ctx)), state)

print('output length: ', len(outputs))
```

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

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入作为下一个时间的输入。



```
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])
```

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 \mathbf{g} ，假设剪裁的阈值是 θ ，那么我们这样剪裁使得 $\|\mathbf{g}\|$ 不会超过 θ ：
$$\mathbf{g} = \min\left(\frac{\theta}{\|\mathbf{g}\|}, 1\right)\mathbf{g}$$

```
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
```

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

```
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, )
            # concatenate 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')
```

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。

结论

通过隐藏状态，循环神经网络很够更好的使用数据里的时序信息。

练习

调调参数（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区别。

吐槽和讨论欢迎[点这里](#)