

# Lesson 5

[Video\(YouTube\)](#) / [Video\(bilibili\)](#) / [Lesson Forum](#)

欢迎来到第5课。我们已经完成了前半部分课程，上节课已经开始了下半部分，现在继续。

我们是从计算机视觉开始的，因为它是最成熟的开箱即用的深度学习应用。在计算机视觉上，如果你不用深度学习，你没办法得到好的结果。上过这个课的人和没有上过这个课的人相比，有了新的技能。在这个课程里可以学到很多训练的技能和有效的神经网络。

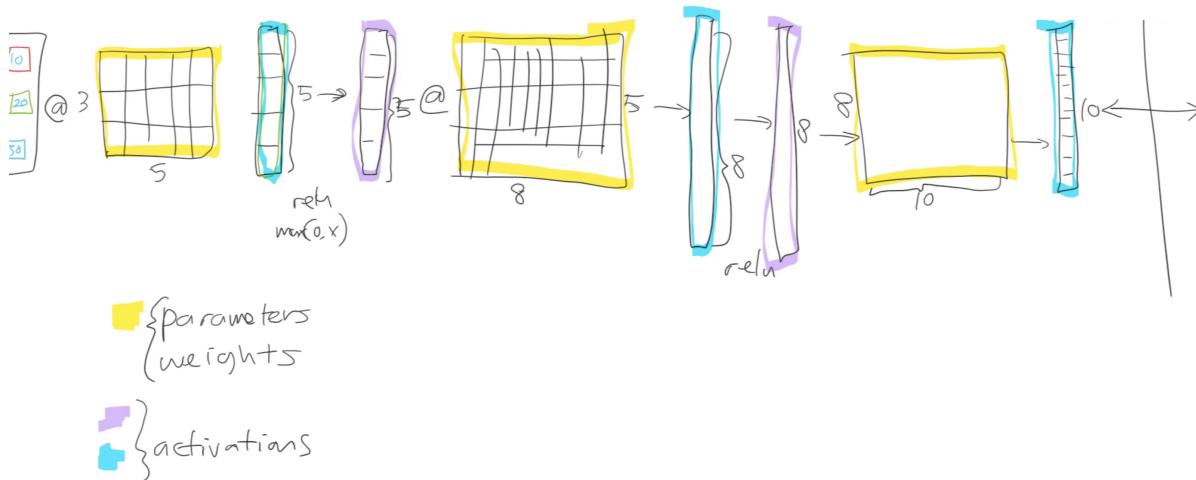
然后我们学了NLP，因为一般来说文本处理是另一个不使用深度学习就没办法做得很好的领域。现在，这个领域可以做得很好了。事实上，昨天纽约时报刚刚发表了文章，报道了深度学习在文本处理的最新进展，里面讲了很多我们和OpenAI、Google、Allen AI研究所合作取得的成果。

然后我们讲了表格数据和协同过滤，把它们放在了最后，一部分原因是，没有神经网络，也可以把它们做得很好。所以这不是很大的进展。这不是以前无法做到的、全新的事情。另一方面，我们要做到理解实现这些功能的每一行代码，表格和协同过滤的实现比视觉和NLP简单很多。所以，在我们回过头来讲解我们究竟做了些什么来实现它们、它们是怎样工作的时侯，我们从协同过滤和表格开始，它们正是刚刚讲过的。今天的课上，我们会看到每一行代码做了什么。这是我们的目标。

这节课上，不要期待能学到新的应用。但你能更好地理解，我们究竟是怎样实现我们学过的那些应用的。具体来说，我们会理解正则化 (regularization)，这是用来处理过拟合的。你可以使用这节课讲的工具，来改进你以前的项目，提升一些性能，或者处理一些你以前觉得没有足够数据、可能过拟合的模型，等等。这也会为我们后面两节课深入学习卷积神经网络和递归神经网络打下基础。在学习这些时我们还会学两个新的应用，两个新的视觉和NLP应用。

## 上周内容回顾 [3:32]

接着上周的内容讲。还记得这张图吗？



上周我们看了深度神经网络是怎样的，我们有不同的层。我们指出的第一件事是，这里有并且只有两种层：存放参数的层和存放激活值的层。参数是你的神经网络要去学习的东西。它们是你用梯度下降来做 `parameters -= learning_rate * parameters.grad` 这个运算的东西。这就是我们做的。这些参数是用来乘以输入的激活值 (input activations) 做矩阵乘法的。

这些黄色的是我们的权重矩阵 (weight matrices)，或者更准确地说，是权重张量 (weight tensors)，叫权重矩阵也没什么问题。我们传入一些输入激活值 (input activations)，或者激活层激活值 (layer activations)，用权重矩阵乘以它，得到一组激活值 (activations)。激活值就是数字，是计算出的数字。在学习小组里，我总是遇到这样的问题：这些数字是哪里来的？我总是这样回答：“你告诉我，这是一个参数还是一个激活值。因为它总会是其中一个”。这就是这些数字是哪里来的这个问题的

答案。我认为输入是一种特别的激活值。它们不是计算出来的，它本来就存在。所以这是一种特殊的激活值。所以你看到的数字可能是一个输入、可能是一个参数、或者是一个激活值。

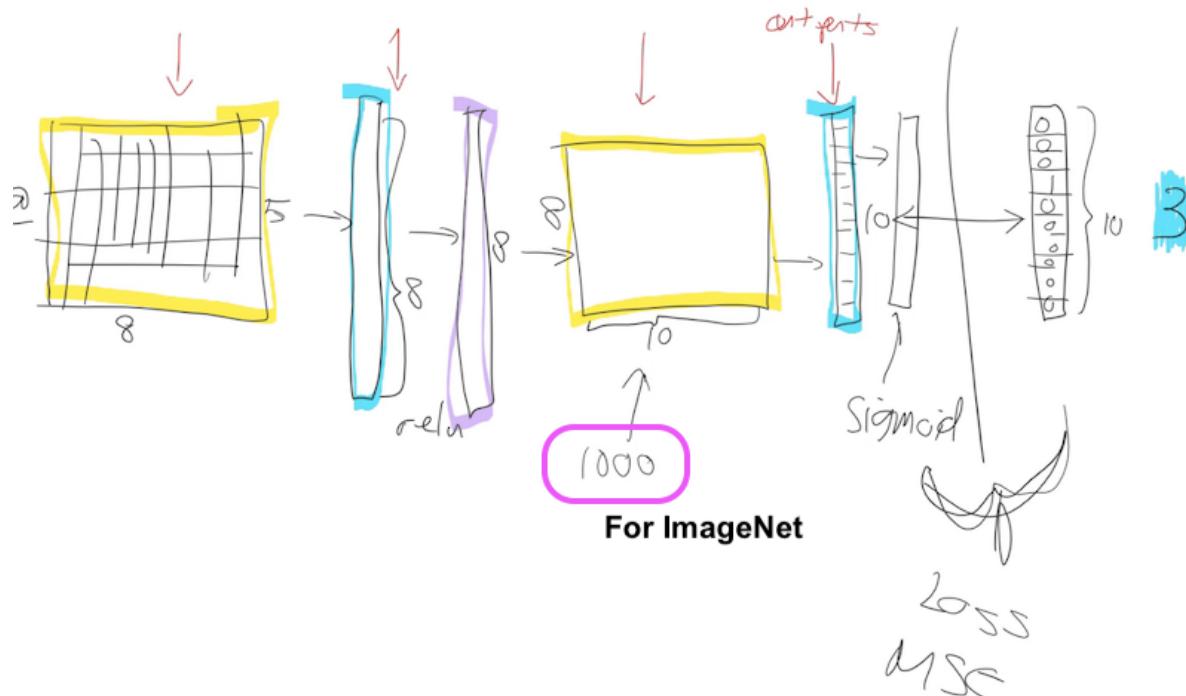
激活值不仅来自于矩阵乘法，也会来自于激活函数。对激活函数，要记住的最重要的东西是它是元素方式 (element-wise) 的函数。它是依次作用在输入的每个元素上的函数，激活函数为输入的每个元素返回一个激活值。所以，如果它被传入了一个长度是20的向量，它就会返回一个长度是20的向量。依次遍历里面每个元素，做些操作，然后输出结果。这就是元素方式的函数。ReLU是我们用过的最主要的激活函数。说实话，选用哪个激活函数不太重要。我们不会花太多时间讲激活函数，因为直接使用ReLU，基本上就能在所有情况下得到很好的结果。

然后我们学了，这种矩阵乘法和ReLU的组合有这个叫做通用逼近定理 (universal approximation theorem) 的奇妙的数学性质。如果你有足够大的权重矩阵，它可以解出任何复杂的数学函数，可以获得任意高的准确度（假设你可以训练这些参数，有足够的文化和数据，等等）。我发现尤其是一些高级的计算机科学家会对这点感到困惑。他们经常问，然后呢？有什么诀窍？它是怎样工作的？但是，这就是全部。你只需要做这些，你传回梯度，然后用学习率更新权重，就是这样。

这里，我们得到真实目标和最后一层的输出（最后的激活值）之间的损失函数，我们计算对于所有这些黄色部分的梯度，然后我们通过减去学习率和梯度的乘积来更新这些黄色的部分。这个计算梯度和做减法的过程叫做 **反向传播 (back propagation)**。当你听到反向传播时，这只是神经网络里常用的一个术语，它听起来很特别，但是你可以在脑袋里把它替换成 `weights -= weight.grad * learning rate`，weights (权重) 也可以叫parameters(参数)，这稍微常用一点儿。这就是我们上周讲的。上周提到，我们会讲更多东西，今天晚些时候我们会继续讲交叉熵 (cross-entropy) 和softmax。

## 微调 (Fine tuning) [8:45]

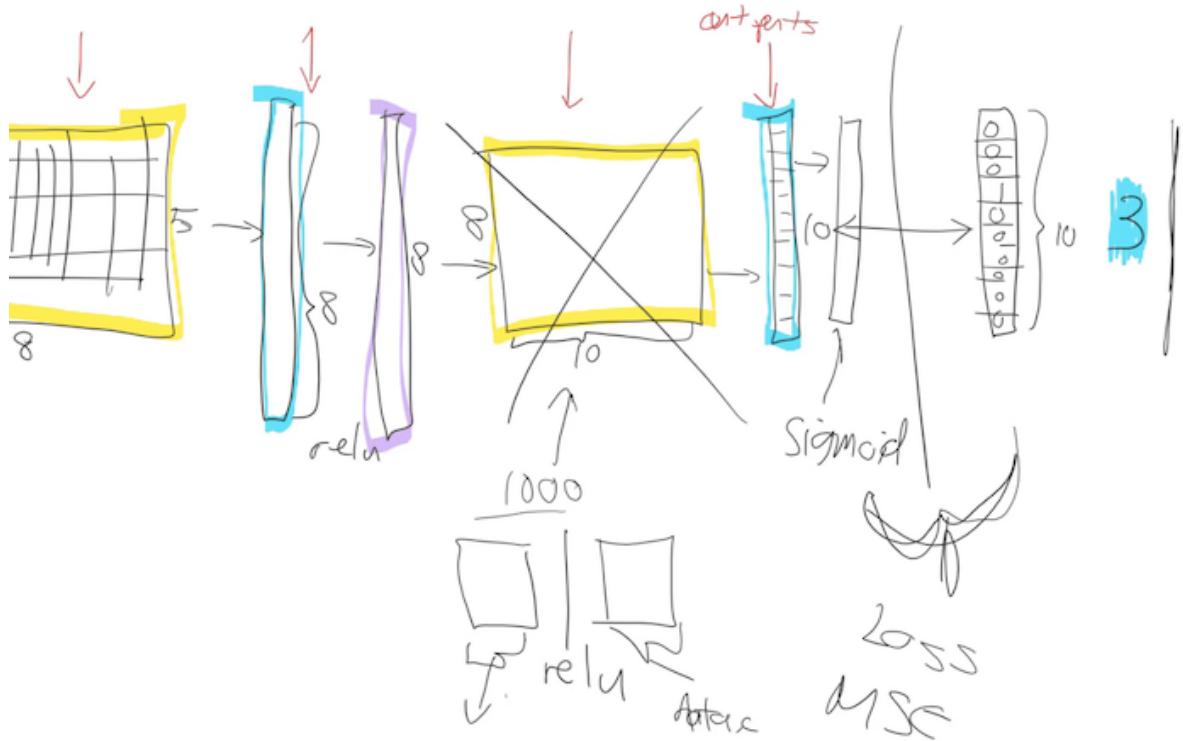
我们讲下微调。我们使用ResNet34和使用迁移学习时会发生什么？是怎样运行的？首先要注意的是，我们从ImageNet得到的ResNet34最终是一个特别的权重矩阵。它是一个有1000列的权重矩阵：



为什么？因为在ImageNet竞赛里他们要你解决的问题的是指出某个图片是1000个分类里的哪一个。这就是为什么我们需要1000列，因为在ImageNet里，目标向量的长度需要是1000。你需要得到1000个代表概率的数字。

在做迁移学习时，这个权重矩阵不合适，有两个原因。第一个原因是，你大概没有一千个分类。我们只是要区分泰迪熊、黑熊和灰熊。所以我不想要1000个分类。第二个原因是，即使我确实刚好有一千个分类，它们也和ImageNet里的不一样。所以，用整个的权重矩阵对我来说是浪费时间。那我们怎样做呢？我们丢掉它。当你在fastai里用create\_cnn时，它会删除这个。然后它会怎样做呢？作为替代，它会

放入两个新的权重矩阵，在它们中间，有一个ReLU。



第一个矩阵的大小，有一些默认值，第二个矩阵的大小取决于你的需要。从你传入leaner的数据bunch里，我们可以知道你需要多少激活值。如果你在做分类任务，它就是你的类别的数量。如果你在做回归，它就是你要预测多少数字。所以，如果你的数据bunch叫 `data`，它会调用 `data.c`。我们会添加一个和 `data.c` 一样大小的矩阵。

[11:08]

好了，现在我们要训练这些矩阵，因为开始时，它们里面都是随机的数字。新的权重矩阵里面的数字是随机的。这些矩阵是新的。我们刚刚创建了它们，把它们放在这里，所以我们需要训练它们。但是其它的层不是新的。其它的层已经擅长做一些事情了。它们擅长做什么？回忆下[Zeiler 和 Fergus 的论文](#)。这是权重矩阵里一些过滤器可视化后的例子，和这些过滤器找到的东西的例子。

---

# Visualizing and Understanding Convolutional Networks

---

Matthew D. Zeiler

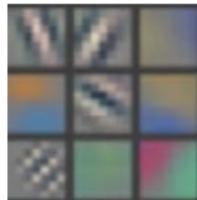
Dept. of Computer Science, Courant Institute, New York University

ZEILER@CS.NYU.EDU

Rob Fergus

Dept. of Computer Science, Courant Institute, New York University

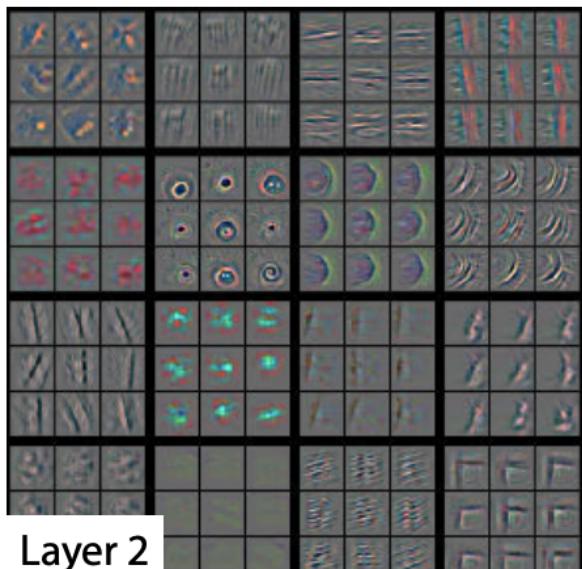
FERGUS@CS.NYU.EDU



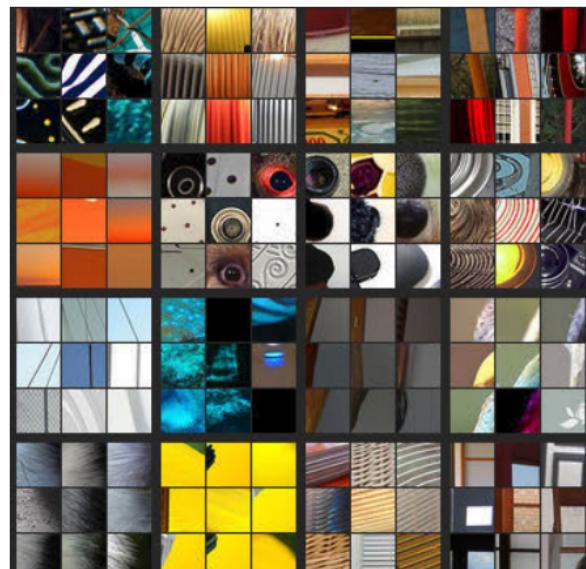
## Layer 1



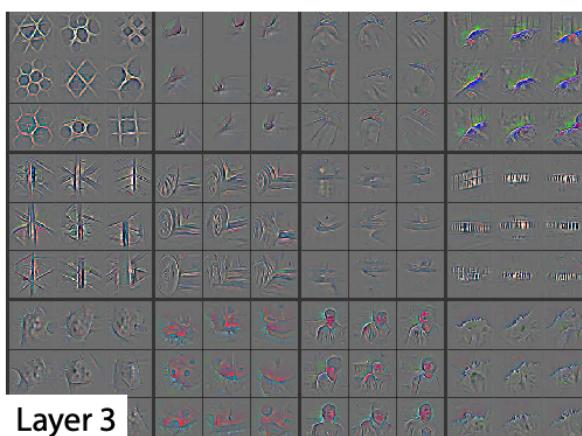
第一层有些矩阵擅长找出这种方向的对角线。



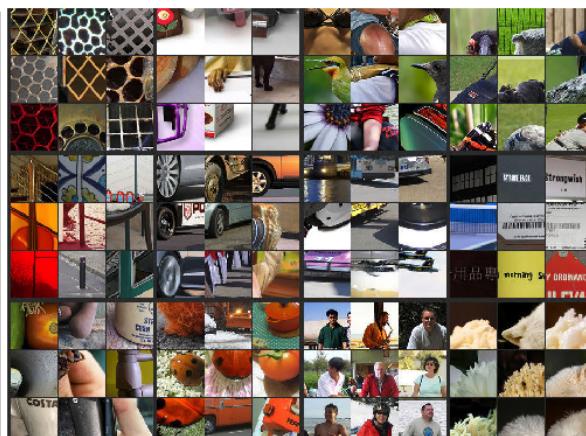
## Layer 2



然后，第二层的一个过滤器擅长找出左上角的角。



## Layer 3

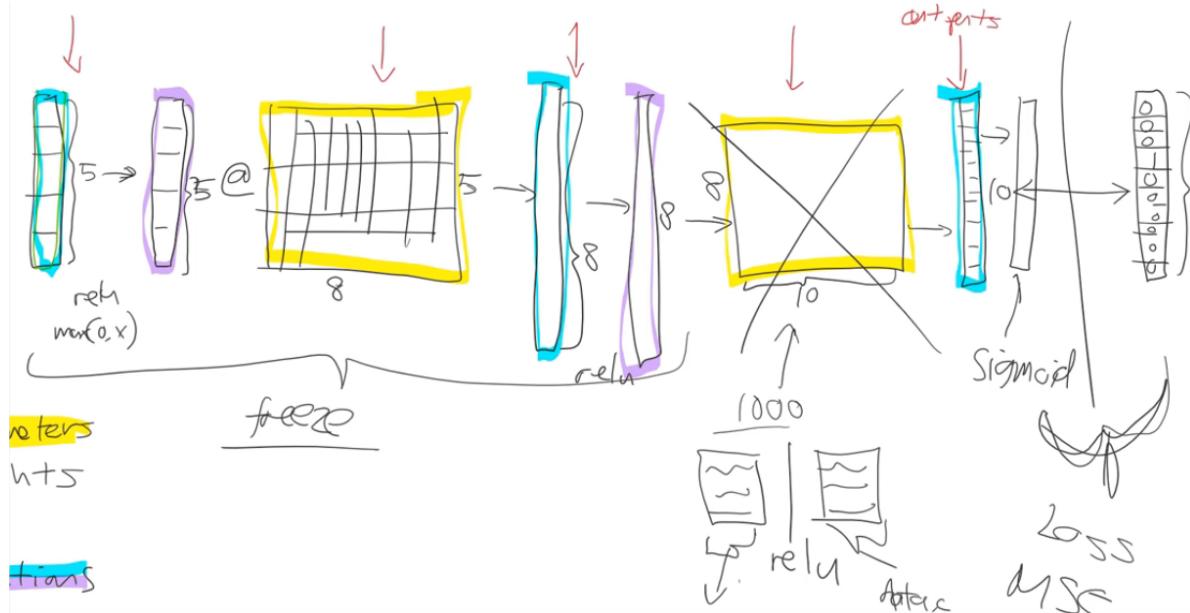


第三层里的一个过滤器擅长找出重复的模式，另一个擅长找出圆形的橙色的东西，有的擅长找出毛茸茸的东西或者文字。

越往后，它们就越复杂，也越具体。我猜，在第4层，可以找出眼球之类的东西。如果你做迁移，让它学习一些医学影像，里面可能没有眼球，对不对？所以后面的这些层对你不合适。层越靠前，你越有可能需要它。

## 冻结 (Freezing layers) [13:00]

首先，我们显然要训练这些新添加的层的权重，因为它们是随机的。开始时，我们不用训练其它层的权重。我们把这叫做冻结。



我们冻结其它所有的层。这是什么意思？这代表我们要求fastai和PyTorch在我们训练时，也就时在调用fit时，不要把梯度反向传播到这些层。换句话说，当你执行 `parameters=parameters - learning rate * gradient` 时，只对新加的层执行这个，不要对其他层执行它。这就是冻结：就是不要更新那些参数。

这样也会快一些，因为要做得计算少了。它也会使用更少的内存，因为要存储的梯度少了。最重要的是，它不会改变那些参数，这些参数已经不错了，至少比随机数要好。

这就是我们执行冻结时发生的事。它不会冻结所有的东西。它会冻结除了新加入的随机的层之外的那些层。

## 解冻，差别学习率 (Unfreezing and Using Discriminative Learning Rates)

接下来会发生什么？过一会儿后，我们会说，“好了，这看起来很不错。我们大概需要训练网络的其他部分了”。所以我们做解冻。现在我们会把所有的层连起来，但我们仍然清楚，我们在最后加的这些新的层需要更多的训练，这些在前面的层（比如识别对角线的层）不需要太多训练。所以我们把模型分成几部分，我们给不同的部分设置不同的学习率。比如，模型前面的部分，我们可能会用 `1e-5` 做学习率，后面的部分，我们可能会用 `1e-3` 做学习率。

接下来，我们可以训练整个网络了。因为前面几层的学习率比较小，参数的变化也比较小，因为我们认为它们已经相当不错，接近最优值了，如果使用更大的学习率，它会偏离最优值，这会让网络的效果变差，我们不想出现这样的情况。这个过程被叫做**差别学习率 (discriminative learning rates)**。你在网上找不到这个，我想我们大概是第一个使用它来做这个的（或者至少是第一个广泛宣传它的。可能其他人也使用过它，但没有写出来）。关于这个，你能找到的绝大部分都是fastai的学生。但是它开始慢慢变得广为人知了。这是一个非常非常重要的概念。对于迁移学习来说，不使用它的话，你得不到很好的结果。

在fastai里我们是怎样用差别学习率的呢？在所有你可以传入学习率的地方都可以用它，比如 `fit` 函数。你传入的第一个参数是迭代的数量（number of epochs），第二个参数就是学习率（如果你使用 `fit_one_cycle` 也是一样的）。对学习率，你可以传入这样几种东西：

- 你可以传入一个数字（比如 `1e-3`）：每一层使用相同的学习率。你不会使用差别学习率。
- 你可以写一个切片（slice）。你可以写一个接收一个参数的slice（比如 `slice(1e-3)`）：最后一层的学习率是你输入的那个数，其他所有层使用这个数的1/3作为学习率。这样，所有其他层的学习率是 `1e-3/3`，最后一层的学习率是 `1e-3`。
- 你可以写两个参数的slice（比如 `slice(1e-5, 1e-3)`）。最后几层（随机添加的层）的学习率会是 `1e-3`。第一层会是 `1e-5`，其他层使用平均分它们的那些数作为学习率。如果有三层，学习率就是 `1e-5, 1e-4, 1e-3`，每次都是相同的间隔。

有一个小细节，为了更方便地管理这些，我们没有真的给每一层一个不同的学习率。我们为每个分组（layer group）分配一个学习率。具体讲，我做的是，把随机添加的额外的层放在一个分组。这是默认的，你也可以修改它。其他的层，平均分成两组。

默认情况下（至少对CNN），你会有三个分组。如果你使用 `slice(1e-5, 1e-3)`，第一层的学习率就是 `1e-5`，第二层是 `1e-4`，第三层是 `1e-3`。现在，如果你回过头来看我们训练的方法，你会看到这非常有意义。

这被分成3份，这有一点奇怪，我们会在课程第二部分讲解。这是一个关于批次标准化（batch normalization）的内容，它看起来有点奇怪。如果有人感兴趣，我们可以在进阶主题中讨论它。

这就是微调。希望讲完后，大家会不再觉得它有多么神秘了。

[19:49]

上节课我们讲了协同过滤。

That's all we need to create and train a model:

```
In [ ]: data = CollabDataBunch.from_df(ratings, seed=42)

In [ ]: y_range = [0, 5.5]

In [ ]: learn = collab_learner(data, n_factors=50, y_range=y_range)

In [ ]: learn.fit_one_cycle(3, 5e-3)

Total time: 00:04
epoch  train_loss  valid_loss
1      1.600185   0.962681   (00:01)
2      0.851333   0.678732   (00:01)
3      0.660136   0.666290   (00:01)
```

在协同过滤的例子中，我们调用了 `fit_one_cycle` 方法，只传入了一个数字。这是合理的，因为在协同过滤里，我们只有一层。它里面有几部分。但是没有这样的东西：做一个矩阵相乘，传入激活函数，再做矩阵相乘。

## 仿射函数（Affine Function） [20:24]

再介绍一个术语。它和矩阵乘法并不完全相同，但非常相似。它也是把数值加在一起的线性函数，但比矩阵乘法用得更广泛，它叫**Affine Function**（仿射函数）。如果你听到我说仿射函数这个词，你可以在大脑里用矩阵乘法替代它。但是，就像我们会在卷积操作里看到的那样，卷积操作是矩阵乘法，但是它里面的一些参数被限制了。所以把这叫做仿射函数会更准确。每节课我都会介绍一些术语，这样你在读书、读论文或者学习课程、阅读文档时，会知道这些术语是什么。当你看到仿射函数时，它只是一个线性函数。它和矩阵乘法很像。矩阵乘法是最常见的仿射函数，至少在深度学习里是这样。

在协同过滤里，我们用的模型是这个：

NB: These are initialized to random numbers	0.71	0.92	0.68	0.83	0.60	0.18	0.26	0.91	0.99
Then we use Solver to optimize them	0.81	0.55	0.28	0.88	0.50	0.31	0.08	0.47	0.94
with gradient descent	0.74	0.86	0.53	0.33	0.81	0.68	0.92	0.61	0.46
	0.04	0.44	0.16	0.41	0.73	0.39	0.29	0.94	0.12
	movield	0.04	0.80	0.94	0.24	0.53	0.09	0.74	0.13
	userid	27	49	57	72	79	89	92	99
0.19	0.63	0.31	0.44	0.51	14	0.91	1.40	1.02	1.12
0.25	0.83	0.71	0.96	0.59	29	1.44	2.20	1.49	1.71
0.30	0.44	0.19	0.00	0.72	72	0.73	1.26	1.10	0.87
0.02	0.72	0.69	0.35	0.25	211	1.12	1.36	0.86	1.08
0.60	0.87	0.76	0.30	0.04	212	1.71	0.00	1.14	1.65
0.73	0.70	0.44	0.47	0.29	293	1.44	0.00	1.27	1.63
0.23	0.81	0.36	0.47	0.12	310	1.10	1.27	0.76	1.24
0.68	0.90	0.20	0.92	0.74	379	1.43	2.30	1.67	1.98
0.81	0.41	0.81	0.15	0.17	451	1.52	1.88	1.28	1.41
0.70	0.61	0.90	0.89	0.24	467	1.70	2.35	1.49	1.84
0.50	0.27	0.73	0.44	0.83	483	1.16	2.10	1.65	1.28

在这里有一组数据（左侧），在这里有一组数据（上方），我们取它们的点积。这是一行，这是一列，这和矩阵乘积是一样的。在Excel里 MMULT 是矩阵乘法，这里 (H25到W39) 是这两组数据的矩阵乘积。

上周我用Excel里的求解器做了训练，我们还没有看它是怎样工作的。现在我们看下。均方差下降到了0.39。我们要预测0.5到5之间的值。我们出错的平均值大概是0.4。这个效果很好。看下这里目标值是3、5、1，预测值是3.23、5.1、0.98，这很接近，效果不错。现在，你们对这个已经有了大致的概念。

## 嵌入 (Embedding) [22:51]

下面我讲下嵌入矩阵 (embedding matrices) 这个概念。要理解它，我们先看下[另外一个 worksheet](#) (“movielens\_1hot” tab)。

这是另外一个worksheet。我把前一个worksheet里的这两个权重矩阵拷了过来。这个是用户的矩阵（O3到AC280），这个是电影的矩阵（AK3到AY280）。我把电影矩阵做了转置，现在它的维度和用户矩阵是一样的了。

这是两个权重矩阵（橙色）。初始时，它们是随机的，我们可以用梯度下降训练它们。在原始数据里，用户ID和电影ID是这样的数字，为了更方便，我们把这些数字转换成1到15（`user_idx` 和 `movie_idx`）。在这些列里，对每一个rating（评分），我都有一个转换后的用户ID和电影ID，它们从1开始。

现在我把1这个用户ID用这个向量代替：向量里第一个数字是1，其他的都是0。然后对第二个用户，用这个向量代替：向量里第一个数字是0，第二个是1，其他的都是0。对于14这个电影ID也是一样，用一个这样的向量代替：前面13个数字是0，然后是一个1，和一个0。这叫做one-hot编码。这不是神经网络的一部分，这是对输入的预处理，我把它作为新的输入：

这是新的电影的输入数据，这是新的用户的输入数据。它们是神经网络的输入。

现在我要做的是，取这个矩阵，用它和权重矩阵做乘法。它们可以相乘，因为权重矩阵有15行，这个one-hot编码向量有15列。它们是对应的，我们可以把它们乘到一起。在Excel里你可以用 `MMULT` 函数做矩阵乘法。在使用Excel时注意下，这是一个返回很多数字的函数，输入完成时你不能只按enter键，你要按 `ctrl + shift + enter`。`ctrl + shift + enter` 代表这是一个数组函数，它会返回多个数值。

这（User activations）是输入矩阵和权重矩阵的矩阵乘积。这是一个普通的神经网络层。这是一个常规的矩阵乘法。我们可以对电影数据做相同的事情，这是电影数据的矩阵乘法。

就是这样。这个输入，是用户ID的one-hot编码，这些激活值是ID是1的用户的激活值。为什么呢？想想看，one-hot编码的向量和一个矩阵的乘积，当one-hot里的1是在第n个位置时，实际上是要找出矩阵的第N行。我们这里做的是，做一个计算出激活值的向量乘法。它是用一种很有趣的方式做的：它在输入矩阵中找出一行。

做完这个之后，我们把这两组数乘在一起（做点积），然后我们可以得到损失值，做平方，然后可以得到平均值。

看，这个值0.39和前面sheet的求解器算出的值是一样的，因为它们做的是相同的事情。

这里 (“dotprod”里) 是找出特定的用户embedding，这里 (“movielens\_1hot”) 是只做一个矩阵乘法，这样我们知道它们在数学上是等价的。

## 再次Embedding [27:57]

我们再做一次。这是我们最后的版本（这部分建议观看视频）：

The screenshot shows an Excel spreadsheet titled "collab\_filter.xlsx". It contains several tables and formulas. The first table, "original data", has columns for user ID, movie ID, rating, and user index. The second table, "user embedding", maps user indices to 5-dimensional vectors. The third table, "movie embedding", maps movie indices to 5-dimensional vectors. The bottom section, "Movies", lists movie IDs and their embeddings. The formula bar shows "dotprod", "movielens\_1hot", and "movielens\_emb" being used.

这还是相同的权重矩阵，我拷贝了相同的过来。这是用户ID和电影ID。这次，我们用普通的表格形式存它们。这次，我还是得到里相同的激活值，和“movielens\_1hot”里的一样。但这次，我用的是Excel里的“OFFSET”函数来计算这些激活值，它做的是数组查找。这个版本（“movielens\_emb”）和“movielens\_1hot”是等价的，但显然它用了更少的内存，运行得更快。因为我没有创建一个one-hot编码矩阵，没有做矩阵相乘。对大部分数字是0的矩阵做乘法，就是浪费时间。

换句话说，乘以一个one-hot编码的矩阵和做数组查找（array lookup）是等价的。所以**我们应该用数组查找的版本**，我们有这样一种方式，来用one-hot编码矩阵做矩阵乘法，但并不真正的创建one-hot矩阵。我只是传入一组数字，假装它们是one-hot编码的。这就叫**embedding**。

你可能在很多地方听说过“embedding”这个词，好像它是一个很高深的数学术语，实际上只是代表在数组里找出一些东西。不过，要知道在数组里找出一些东西和用one-hot做向量乘法是相同的。所以，在神经网络标准模型里，embedding很合适。

就像时我们有了另外一种层。在这种层里，我们从数组里查出数据。但实际上没有做任何其他事情。我们只是简化了计算，这只是一个快速的内存高效的做one-hot矩阵乘法的方法。

这很重要。当你听到人们说embedding时，你需要在大脑里把它替换乘“一个数组查找”，它和用one-hot矩阵做乘法在数学上是等价的。

就是这样，它有有意思含义。当你用one-hot矩阵乘以别的矩阵时，你可以看到这样的特性，权重矩阵里，只有用户ID输入里值是1的对应的那行才会出来。换句话说，你可以得到和你的输入对应的一个权重矩阵。因为我们计算输出激活值的唯一方式是做两个输入向量的点积。这意味着它们需要相互对应。需要有这样一种方式，这些值对某个用户比较高，这些值对某个电影比较高，这样这个用户就喜欢这个电影。它的含义可能是这样的，**这些数字代表用户兴趣特征和对应的电影特征**。比如，这个电影里有John Travolta，并且这个用户喜欢John Travolta，那么这个用户就喜欢这个电影。

我们没有让这些行代表所有东西。我们没有做什么事情来让这些行代表所有东西。梯度下降可以得出好的结果的唯一方法是，它能不能找出电影偏好是什么以及对应的电影的特征是什么。这里说的潜在的特征被叫做**latent factors**或者**latent features**（潜在特征）。就是这些本来隐藏的，一训练它们马上就出现的东西。

## 偏置 (Bias) [33:08]

这里有个问题。没人会喜欢《Battlefield Earth》。尽管里面有John Travolta，它也不是一个好电影。那我们要怎样处理它。这里有一个叫做“我喜欢John Travolta”的特征，这有个叫做“这部电影里有John Travolta”的特征，看起来你会喜欢这个电影。但是我们需要有些方法实现“除非它是《Battlefield Earth》”或者“你是山达基人 (Scientologist)”，这两个都可以。我们要怎样做呢？我们需要加入**bias**。

还是相同的东西，相同的结构，所有的形状都是相同的。但这次我们要加入一个额外的行。现在它不仅是这两个的矩阵乘积，我还加上了这个数字（A列）和这个数字（19行）。这代表，现在所有电影都有了一个总分代表“这是一个很棒的电影”或者“这不是一个很棒的电影”，每个用户有这样一个总分代表“这个用户给电影打分高”或者“这个用户给电影打分低”，这就叫bias。这看起来很熟悉。这是常见的线性模型里的概念，也是有矩阵乘积和偏置的神经网络线性层里的概念。

记得吗，从第二课SGD的notebook开始，你不会真的使用一个bias，你可以只是添加一列全部是1的数组到输入数据里，然后会自动得出bias，但是这很没有效率。在实践中，所有的神经网络类库都会有一个显式的bias的概念。我们不需要添加一列值是1的数组。

它的效果怎么样呢？就在今天我进来之前，我运行了求解器，我们我们可以检查RMSE。均方差根是0.32，没有bias的版本的值是0.39。可以看到这个更好的模型给出了更好的结果。说它更好，是因为它提供了更大的灵活性，它在语义上也更有意义，我喜欢这个电影不仅取决于它的演员是谁、它是不是很多对话、有多少动作等等，也取决于它是不是一个好电影和我是不是喜欢给电影评高分的人。

这就是协同过滤模型的所有内容。

**提问:** 当加载预训练模型时, 能不能浏览激活值看看它们擅长识别什么? [36:11]

可以的。下节课你可以学到怎样做。

**提问：**能不能解释下，`fit_one_cycle`里第一个参数代表什么？和epoch是相同的吗？

是的，`fit_one_cycle` 和 `fit` 的第一个参数是 epoch 的数量。换句话说，一次 epoch 是把所有输入看一遍。如果你做 10 次 epoch，你就是把所有数据看了 10 遍。如果你有很多参数和一个比较高的学习率，你可能会过拟合。如果只做一个 epoch，不会过拟合。这就是为什么要清楚你做了多少 epoch。

**提问：**什么是affine function (仿射函数) ?

仿射函数是一个线性函数。我不清楚，我们是否还需要更多的细节。如果你把一些东西乘起来，把它们加在一起，这就是一个仿射函数。我不关心它的准确的数学定义，一方面是我没那么擅长数学，另一方面是这没那么重要。你只是把东西乘在一起，然后把它们加在一起，这就是最重要的东西。它是线性的。因此，如果你在一个仿射函数外面又用了一个仿射函数，它还是一个仿射函数。你不会得到什么新的东西。这就是在浪费时间。你需要在外面用一个非线性函数，比如把负值变成0 (ReLU)。如果你做了affine, ReLU, affine, ReLU, affine, ReLU, 你就得到了一个深度神经网络。

[38:25]

继续看[collaborative filtering notebook](#)。这次我们用完整的Movielens的100k的数据集。还有一个两千万的数据集。这是一个叫GroupLens的团队做的项目，它很棒。他们更新了Movielens数据集，好心地提供了原始数据。我们会用原始的数据，这样我们可以和基线做对比。因为每个人都会说，如果你想和基线对比，就使用相同的数据集。这就是我们要用的。不幸的是，这样我们只能用1998之前的电影。当你下载后，使用它时，如果想要用最新的电影数据，你可以用 m1-latest 替代它：

```
1 | path=Path('data/m1-100k/')
```

比较新的Movielens数据集是在CSV文件里的，这超级方便。原始的数据集有点麻烦。首先，它们没有用逗号分隔，它们用了tabs。在Pandas里，你可以指明用的分隔符是什么。然后加载它。第二点是它们没有表头行，你需要告诉Pandas这个文件没有表头。因为没有表头，你需要告诉Pandas这四列的名字是什么。这是我们要做的。

```
1 | ratings = pd.read_csv(path/'u.data', delimiter='\t', header=None,
2 |                               names=[user, item, 'rating', 'timestamp'])
3 | ratings.head()
```

	userId	movieId	rating	timestamp
0	196	242	3	881250949
1	186	302	3	891717742
2	22	377	1	878887116
3	244	51	2	880606923
4	166	346	1	886397596

然后我们可以用 head 看下最前面的几行，这里有rating、user、movie。

让我们做得更有趣些。我们看下这些电影是什么。

```
1 | movies = pd.read_csv(path/'u.item', delimiter='|', encoding='latin-1',
2 |                           header=None,
3 |                           names=[item, 'title', 'date', 'N', 'url', *[f'g{i}' for i in
range(19)]])
3 | movies.head()
```

	movieId	title	date	N	url	g0	g1	g2	g3	g4	...	g9	g10	g11	g12	g13	g14	g15	g16
0	1	Toy Story (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?Toy%20Story%20...	0	0	0	1	1	...	0	0	0	0	0	0	0	
1	2	GoldenEye (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?GoldenEye%20(...	0	1	1	0	0	...	0	0	0	0	0	0	1	
2	3	Four Rooms (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?Four%20Rooms%...	0	0	0	0	0	...	0	0	0	0	0	0	1	
3	4	Get Shorty (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?Get%20Shorty%...	0	1	0	0	0	...	0	0	0	0	0	0	0	
4	5	Copycat (1995)	01-Jan-1995	NaN	http://us.imdb.com/M/title-exact?Copycat%20(1995)	0	0	0	0	0	...	0	0	0	0	0	0	1	

5 rows x 24 columns

我要讲下，这里有个叫 `encoding=` 的参数，去掉它的话，我会得到这个错误：

```
1 UnicodeDecodeError                                     Traceback (most recent call last)
2 pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._convert_tokens()
3 pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._convert_with_dtype()
4 pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._string_convert()
5 pandas/_libs/parsers.pyx in pandas._libs.parsers._string_box_utf8()
6
7
8 UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 3:
9     invalid continuation byte
10
11 During handling of the above exception, another exception occurred:
12 UnicodeDecodeError                                     Traceback (most recent call last)
13 <ipython-input-15-d6ba3ac593ed> in <module>
14     1 movies = pd.read_csv(path['u.item'], delimiter='|', header=None,
15     ----> 2                                         names=[item, 'title', 'date', 'N', 'url', *
16     [f'g{i}' for i in range(19)]])
17     3 movies.head()
18
19 ~/src/miniconda3/envs/fastai/lib/python3.6/site-
20 packages/pandas/io/parsers.py in parser_f(filepath_or_buffer, sep,
21 delimiter, header, names, index_col, usecols, squeeze, prefix,
22 mangle_dupe_cols, dtype, engine, converters, true_values, false_values,
23 skipinitialspace, skiprows, nrows, na_values, keep_default_na, na_filter,
24 verbose, skip_blank_lines, parse_dates, infer_datetime_format,
25 keep_date_col, date_parser, dayfirst, iterator, chunksize, compression,
26 thousands, decimal, lineterminator, quotechar, quoting, escapechar, comment,
27 encoding, dialect, tupleize_cols, error_bad_lines, warn_bad_lines,
28 skipfooter, doublequote, delim_whitespace, low_memory, memory_map,
29 float_precision)
30     676                                         skip_blank_lines=skip_blank_lines)
31     677
32 --> 678         return _read(filepath_or_buffer, kwds)
33     679
34     680     parser_f.__name__ = name
35
36 ~/src/miniconda3/envs/fastai/lib/python3.6/site-
37 packages/pandas/io/parsers.py in _read(filepath_or_buffer, kwds)
38     444
39     445     try:
40 --> 446         data = parser.read(nrows)
41     447     finally:
42     448         parser.close()
43
44 ~/src/miniconda3/envs/fastai/lib/python3.6/site-
45 packages/pandas/io/parsers.py in read(self, nrows)
46     1034         raise ValueError('skipfooter not supported for
47 iteration')
48     1035
49 --> 1036         ret = self._engine.read(nrows)
50     1037
51     1038         # May alter columns / col_dict
```

```
38 ~/src/miniconda3/envs/fastai/lib/python3.6/site-
  packages/pandas/io/parsers.py in read(self, nrows)
39     1846     def read(self, nrows=None):
40         1847         try:
41             -> 1848             data = self._reader.read(nrows)
42         1849         except StopIteration:
43             1850             if self._first_chunk:
44
45 pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader.read()
46 pandas/_libs/parsers.pyx in
pandas._libs.parsers.TextReader._read_low_memory()
47 pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._read_rows()
48 pandas/_libs/parsers.pyx in
pandas._libs.parsers.TextReader._convert_column_data()
49 pandas/_libs/parsers.pyx in
pandas._libs.parsers.TextReader._convert_tokens()
50 pandas/_libs/parsers.pyx in
pandas._libs.parsers.TextReader._convert_with_dtype()
51 pandas/_libs/parsers.pyx in
pandas._libs.parsers.TextReader._string_convert()
52 pandas/_libs/parsers.pyx in pandas._libs.parsers._string_box_utf8()
53
54 UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 3:
  invalid continuation byte
```

我指出这个是因为你们都会在某个时候遇到这个报错 `codec can't decode .....`。这代表这不是一个Unicode文件。当你使用一个比较早的数据集时，这很常见。西方国家里使用英语的人意识到，有很多使用其他语言的人。现在我们在处理不同语言上做得更好了。我们使用Unicode这个标准编码来处理。Python默认使用Unicode。如果你要加载一个不是Unicode编码的老文件，你需要猜猜它用的是什么编码。如果它是西方欧美国家的人创建的，很有可能用的是Latin-1。如果你用Python的 `open` 或者 Pandas的 `open` 或者其他类似的方法，只要你传入 `encoding='latin-1'`，就能解决这个问题。

还是一样，它们没有列名，我们需要列出这些列名。这需要讲下。每个类别都有单独一列，总共有19个类别。可以看到，这有点像ont-hot编码，实际上，这是N hot编码。也就是说，一个电影可以属于多个类别。我们今天不关心类别，但要讲下，有时人们会用这样的方式表示类别这样的东西。最近的版本里，它们直接列出类别，这就方便多了。

```
1 | len(ratings)
```

```
1 | 100000
```

[42:07]

有十万个评分。我发现建模时你不遵循数据表设计规范（denormalize data）会做起来更容易。我希望电影名字显示在rating表里，Pandas有一个merge函数可以用来做这个。这是带电影名字的rating表。

```
1 | rating_movie = ratings.merge(movies[[item, title]])
2 | rating_movie.head()
```

	userId	movieId	rating	timestamp	title
0	196	242	3	881250949	Kolya (1996)
1	63	242	3	875747190	Kolya (1996)
2	226	242	5	883888671	Kolya (1996)
3	154	242	3	879138235	Kolya (1996)
4	306	242	5	876503793	Kolya (1996)

像往常一样，我们可以为应用创建一个data bunch，对协同过滤应用，有CollabDataBunch。输入是什么？是一个data frame。这里有data frame。先不管验证集。如果你要和基准（benchmark）比较，我们确实需要用验证集交叉验证。稍后再做比较。

```
1 | data = CollabDataBunch.from_df(rating_movie, seed=42, pct_val=0.1,
item_name=title)
```

```
1 | data.show_batch()
```

userId	title	target
588	Twister (1996)	3.0
664	Grifters, The (1990)	4.0
758	Wings of the Dove, The (1997)	4.0
711	Empire Strikes Back, The (1980)	5.0
610	People vs. Larry Flynt, The (1996)	3.0
407	Star Trek: The Wrath of Khan (1982)	4.0
649	Independence Day (ID4) (1996)	2.0
798	Sabrina (1954)	4.0

默认情况下，CollabDataBunch假设第一列是用户，第二列是条目（item），第三列是rating。但是现在，我们要使用title列做条目，所以我们需要告诉它item列的名字是什么（item\_name=title）。我们所有的data bunch都支持show\_batch，你可以检查里面的数据，它们是这样的。

## 一些技巧 [43:18]

我会尽可能做出好的结果，所以我会尝试各种技巧。一个方法是使用Y range。记住，Y range是把最后的激活函数变成sigmoid。上周，我们用了一个从0到5的sigmoid，这样可以保证让神经网络预测出在这个范围（range）里的东西。

我没有在Excel的版本里做这个，所以你可以看到结果里有一些负值，还有一些大于5的数。如果你想在Excel里得到超过我的成绩，你可以添加这个sigmoid到excel里来训练，你会得到一个好一些的结果。

现在的问题是，sigmoid只会接近最大值（这里是5），这意味着你没办法预测出5。但是很多电影的评分是5，这是一个问题。所以最好让Y range的范围稍微小于最小值，稍微大于最大值。数据的最小值是0.5，最大值是5，这样的话sigmoid的范围就要更大些。这是一个可以得到高一点准确率的小技巧。

```
1 | y_range = [0, 5.5]
```

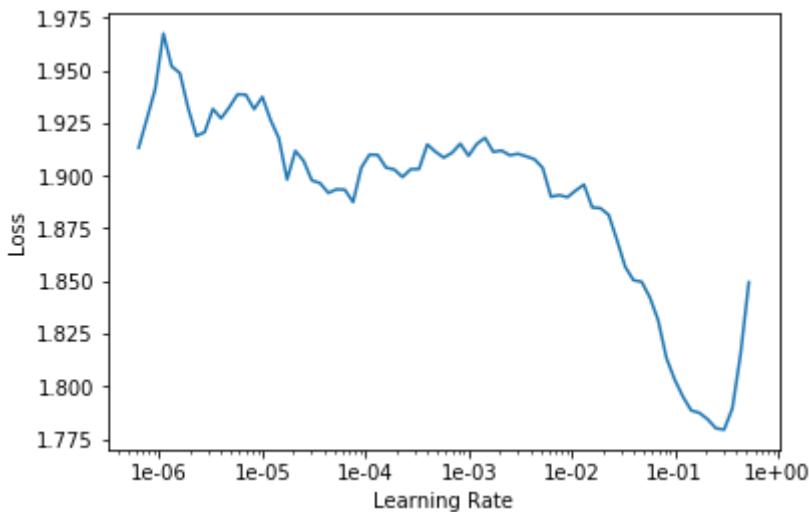
另外一个技巧是添加weight decay（权重衰减），后面我们会学习它。这部分完成后，我们会学习权重衰减。

```
1 | learn = collab_learner(data, n_factors=40, y_range=y_range, wd=1e-1)
```

你想用多少个factor，什么是factor？factor的数量是embedding矩阵的宽度。为什么我们不说embedding的大小呢？或许我们应该这样说，但是在协同过滤问题里，他们不用这个单词。他们用“factor”这个词，这是因为“latent factors”（潜在因素）这个概念，另外，也有这个原因：做协同过滤的标准方法里用了**matrix factorization**（矩阵分解）。事实上，我们刚刚看到的就是一种做矩阵分解的方式。所以，我们今天其实顺便学习了怎样做矩阵分解。它是这个领域里特有的一个术语。但你可以把它当作embedding矩阵的宽度。

为什么值是40？这是一个对网络结构的决定，你需要来回试试，看哪个值管用。我试了10、20、40和80，发现40看起来效果比较好。它训练得很快，你可以把它放到一个for循环里，多试一些值，看看哪个最好。

```
1 | learn.lr_find()
2 | learn.recorder.plot(skip_end=15)
```



然后是学习率，这是learning rate finder，和以前一样。`5e-3`看来看去效果很好。记住，这只是一个经验法则。`5e-3`比用Sylvain和我的规则得到的值都要小，Sylvain的规则是找到底部，然后减小到 $1/10$ ，所以按他这个规则大概是`2e-2`。我的规则是找出最陡的部分，大概是这里，这和Sylvain的规则得出的值是吻合的，也是`2e-2`。我试了下，我会放大10倍，缩小10倍来检查下。我发现小一点更好。如果遇到“我应该这样做吗”这个问题，答案是“试试看”。这样你才成为一个好的实践者。

```
1 | learn.fit_one_cycle(5, 5e-3)
```

```

1 | Total time: 00:33
2 | epoch  train_loss  valid_loss
3 | 1      0.938132   0.928146   (00:06)
4 | 2      0.862458   0.885790   (00:06)
5 | 3      0.753191   0.831451   (00:06)
6 | 4      0.667046   0.814966   (00:07)
7 | 5      0.546363   0.813588   (00:06)

```

```
1 | learn.save('dotprod')
```

它给出了0.813的结果。像往常一样，你可以把它保存下。这样下次你就不用再重新做了，可以节省你33秒的时间。

有一个叫做LibRec的库，他们发布了一些MovieLens 100k的基准，里面有一个均方差根（root mean squared error）的部分。大概是0.91。这大概是他们能得到最好的结果。0.91是均方差根。我们用的是均方差（mean squared error），没有求根。0.91<sup>2</sup>是0.83，而我们得到了0.81。这很棒，使用这个非常简单的模型，我们得到了更好的结果，实际上好了很多。尽管，就像我说过的，不要太把这个结果当回事儿，因为我们对数据的切分方式不一样，交叉验证方法也不一样。但至少我们相对他们的方法是很有竞争力的。

稍后，我们会看下实现这个的Python代码，现在，我们看看这个Excel是怎么实现协同过滤的。

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
1																							
2		Users	Embeddings																				
3	idx	Row Lab																					
4	1	14	0.21	1.61	2.89	-1.26	0.82																
5	2	29	1.55	0.75	0.22	1.62	1.26																
6	3	72	1.50	1.17	0.22	1.08	1.49																
7	4	211	0.47	0.89	1.32	1.13	0.77																
8	5	212	0.31	2.10	1.47	-0.29	-0.15																
9	6	293	1.00	1.45	0.37	0.83	0.67																
10	7	310	1.16	1.16	0.19	2.16	-0.03																
11	8	379	0.79	1.07	1.30	1.29	0.70																
12	9	451	1.52	0.54	0.64	1.36	0.94																
13	10	467	1.00	0.69	0.41	0.75	1.02																
14	11	508	0.86	1.29	0.80	0.19	1.79																
15	12	546	0.61	-0.09	2.40	1.57	-0.18																
16	13	563	1.45	0.59	1.40	1.29	-0.13																
17	14	579	0.68	0.95	1.53	0.84	0.64																
18	15	623	1.70	1.00	0.20	-0.25	2.05																
19																							
20		Movies																					
21	idx	Row Lab																					
22	1	27	-1.69	1.01	0.82	1.89	2.39																
23	2	49	1.49	0.12	1.48	0.50	1.13																

把数据放到数组里，把它们乘起来，相加，做均方差损失函数。输入这些，我们可以看到，要做的 是找到这些“latent factors”（潜在因素）。这让它找出的东西有了依据。在找出这些latent factors时， 我们也找出了每个用户和电影的bias。

现在你可以说出每个电影的平均评分。但还有一点问题。有些东西你经常看，比如说动漫。有些人热爱动漫，他们看了很多动漫，他们给所有动漫的评分都很高。所以，你经常会在一些电影的榜单中， 看到很多动漫排在前面。尤其是，如果有一个100多集的系列动漫，你可以看到这个系列里的每一个都在榜单的前1000名里。

## 解释bias（偏置） [49:29]

怎样处理这种情况？好的一点是，如果我们观察电影bias时，同时考虑用户bias（一个动漫爱好者 的这个值很可能很高，因为他们给很多电影高分），也计算这种电影的影响，剩下的就是这个电影本身 的影响。观察电影bias是一种判断什么是好电影、什么是人们真正喜欢的电影的方式，尽管有的人不给 电影评高分，或者那个电影没有他可能评高分的那种特征。这很有用。这很好。我们可以这样说，使用 bias，我们得到了没有偏见的（unbiased）电影评分。

我们应该怎样做呢？因为这个数据集只包含了1998之前的电影，为了容易理解，我们只看大多数人看过的电影。我们用Pandas取 rating\_movie 表，按title分组，统计评分的数量。不是要看评分多高，只是看看有多少条评分。

```
1 | learn.load('dotprod');
```

```
1 | learn.model
```

```
1 | EmbeddingDotBias(  
2 |     (u_weight): Embedding(944, 40)  
3 |     (i_weight): Embedding(1654, 40)  
4 |     (u_bias): Embedding(944, 1)  
5 |     (i_bias): Embedding(1654, 1)  
6 | )
```

```
1 | g = rating_movie.groupby(title)['rating'].count()  
2 | top_movies = g.sort_values(ascending=False).index.values[:1000]  
3 | top_movies[:10]
```

```
1 | array(['Star Wars (1977)', 'Contact (1997)', 'Fargo (1996)', 'Return of the  
2 | Jedi (1983)', 'Liar Liar (1997)',  
3 |         'English Patient, The (1996)', 'Scream (1996)', 'Toy Story (1995)',  
        'Air Force One (1997)',  
        'Independence Day (ID4) (1996)'], dtype=object)
```

最前面的1000部电影是被评价最多的，也很有可能是我们看过的。这就是我做这个的唯一原因。我把它叫做 top\_movies，就是说它们不一定是好电影，只是我们可能看过的电影。

毫不奇怪，《星球大战》（Star Wars）是一个，那时评价它的人最多。《独立日》（Independence Day），是这样的。我们可以取我们训练好的learner，让它给出这些电影的bias。

```
1 | movie_bias = learn.bias(top_movies, is_item=True)  
2 | movie_bias.shape
```

```
1 | torch.size([1000])
```

`is_item=True` 这里，你要传入 `True` 来表明我想要电影（item）的数据，如果传入 `False`，表明想要用户（user）的数据。这是协同过滤里常用的命名方法，这些ID（用户ID）会被叫作用户（user），这些ID（电影ID）会被叫作条目（item），即使你的问题和用户和条目一点关心也没有。我们使用这些名称，纯粹是因为方便。这只是一个名称。在这个地方，我们输入的是item。`top_movies` 是我们想要的条目（item）的列表，我们想要它的bias。这是协同过滤特有的。

这会给我们返回1000个数字，因为这个输入的列表里有1000个电影。为了对比，我们也按照平均评分分组。然后可以zip每个电影和bias，取对应的评分、bias和电影。然后我们可以按的0列，也就是 bias，来排序。

```
1 | mean_ratings = rating_movie.groupby(title)['rating'].mean()  
2 | movie_ratings = [(b, i, mean_ratings.loc[i]) for i,b in  
        zip(top_movies,movie_bias)]
```

```
1 | item0 = lambda o:o[0]
```

这是最低的数字：

```
1 | sorted(movie_ratings, key=item0)[:15]
```

```
1 | [(tensor(-0.3264),  
2 |   'Children of the Corn: The Gathering (1996)',  
3 |   1.3157894736842106),  
4 |   (tensor(-0.3241),  
5 |     'Lawnmower Man 2: Beyond Cyberspace (1996)',  
6 |     1.7142857142857142),  
7 |     (tensor(-0.2799), 'Island of Dr. Moreau, The (1996)', 2.1578947368421053),  
8 |     (tensor(-0.2761), 'Mortal Kombat: Annihilation (1997)',  
9 |       1.9534883720930232),  
10 |      (tensor(-0.2703), 'Cable Guy, The (1996)', 2.339622641509434),  
11 |      (tensor(-0.2484), 'Leave It to Beaver (1997)', 1.8409090909090908),  
12 |      (tensor(-0.2413), 'Crow: City of Angels, The (1996)', 1.9487179487179487),  
13 |      (tensor(-0.2395), 'Striptease (1996)', 2.2388059701492535),  
14 |      (tensor(-0.2389), 'Free Willy 3: The Rescue (1997)', 1.7407407407407407),  
15 |      (tensor(-0.2346), 'Barb Wire (1996)', 1.9333333333333333),  
16 |      (tensor(-0.2325), 'Grease 2 (1982)', 2.0),  
17 |      (tensor(-0.2294), 'Beverly Hills Ninja (1997)', 2.3125),  
18 |      (tensor(-0.2223), "Joe's Apartment (1996)", 2.2444444444444445),  
19 |      (tensor(-0.2218), 'Bio-Dome (1996)', 1.903225806451613),  
20 |      (tensor(-0.2117), "Stephen King's The Langoliers (1995)",  
21 |        2.413793103448276)]
```

你知道《Mortal Kombat Annihilation》不是一部好电影。《Lawnmower Man 2》不是一部好电影。我没看过《Children of the Corn》，但我们今天在旧金山学习小组讨论了很久，看过这个电影的人认为这不是一部好电影。你可以看到，这里面有些电影评分很不错。这个《Island of Dr. Moreau, The (1996)》的评分比下一个高很多。这大概是因为里面有某个演员，或者是因为电影，或者是因为观众，导致它的评分比较高。

这是倒着排序：

```
1 | sorted(movie_ratings, key=lambda o: o[0], reverse=True)[:15]
```

```
1 | [(tensor(0.6105), "Schindler's List (1993)", 4.466442953020135),  
2 |   (tensor(0.5817), 'Titanic (1997)', 4.2457142857142856),  
3 |   (tensor(0.5685), 'Shawshank Redemption, The (1994)', 4.445229681978798),  
4 |   (tensor(0.5451), 'L.A. Confidential (1997)', 4.1616161616162),  
5 |   (tensor(0.5350), 'Rear Window (1954)', 4.3875598086124405),  
6 |   (tensor(0.5341), 'Silence of the Lambs, The (1991)', 4.28974358974359),  
7 |   (tensor(0.5330), 'Star Wars (1977)', 4.3584905660377355),  
8 |   (tensor(0.5227), 'Good Will Hunting (1997)', 4.262626262626263),  
9 |   (tensor(0.5114), 'As Good As It Gets (1997)', 4.196428571428571),  
10 |   (tensor(0.4800), 'Casablanca (1942)', 4.45679012345679),  
11 |   (tensor(0.4698), 'Boot, Das (1981)', 4.203980099502488),  
12 |   (tensor(0.4589), 'Close Shave, A (1995)', 4.491071428571429),  
13 |   (tensor(0.4567), 'Apt Pupil (1998)', 4.1),  
14 |   (tensor(0.4566), 'Vertigo (1958)', 4.251396648044692),  
15 |   (tensor(0.4542), 'Godfather, The (1972)', 4.283292978208232)]
```

《Schindler's List (辛德勒的名单)》，《Titanic (泰坦尼克)》，《Shawshank Redemption (肖申克的救赎)》，看起来是合理的。你也可以看到一些评分不高，但也出现在这里的电影。看起来至少在1998，人们没那么喜欢Leonardo DiCaprio (莱昂纳多·迪卡普里奥)，没有那么喜欢对话驱动的电影 (dialogue-driven movies)，没有那么喜欢浪漫故事，等等。但大家还是比你认为的更加喜欢这些。用这种方式解释我们的模型很有趣。

## 解释Weights (权重) [54:27]

更进一步，我们不只取出bias，再取出weight。

```
1 | movie_w = learn.weight(top_movies, is_item=True)
2 | movie_w.shape
```

```
1 | torch.size([1000, 40])
```

```
1 | movie_pca = movie_w.pca(3)
2 | movie_pca.shape
```

```
1 | torch.size([1000, 3])
```

我们再取出top movie里条目的权重。它是1000x40的，因为我们设置了40个factor，所以width不是5，是40。

说实话，通常在概念上没有40个潜在因素 (latent factors) 影响观众的偏好 (taste)，所以要看出这40项没有那么直观。我们要把这40个压缩到3个。有个叫PCA的东西，全称是Principal Components Analysis (主成分分析)，我们不会深入讲它。这个 `movie_w` 是一个torch张量，fastai给torch张量添加了PCA方法。PCA是一个简单的线性变换，它接收一个矩阵，尝试找到更少的列，能覆盖原始矩阵的大部分空间。如果这听起来很有意思，你应该学习我们的计算线性代数课程，是Rachel讲的，这个课程里，我们会演示怎样从头计算PCA、为什么要这样做，很多人都喜欢这个课。它不是我们这个课里任何内容的先修内容，但确实值得学习，使用神经网络，并且用PCA处理通常是一个好办法。你经常会遇到层里的激活值比你想要的多，出于各种原因你需要处理它。比如，坐在我旁边的Francisco今天在处理关于图片相似性 (image similarity) 的东西。对于图片相似性，一个好的方法是比较模型的激活值，但通常这些激活值规模会很大，这样处理时会很慢、很笨重。所以对像图片相似性这样的东西，人们经常需要用PCA处理，这很酷。这个例子里，我们这样做就可以把40个成分变成3个，但愿这样就能方便我们做解释。

```
1 | fac0, fac1, fac2 = movie_pca.t()
2 | movie_comp = [(f, i) for f, i in zip(fac0, top_movies)]
```

我们取出这三个因素，把它叫做 `fac0`, `fac1`, `fac2`。取出电影的成分，然后排序。现在，我们不知道这代表什么。但是，我们很知道，它们是某些方面偏好和电影特征。如果我们打印出最高和最低的，我们可以看到在这个特征上排名最高的东西，你可以把这叫做类似“鉴赏电影 (connoisseur movies)”。

```
1 | sorted(movie_comp, key=itemgetter(0), reverse=True)[:10]
```

```
1 [(tensor(1.0834), 'Chinatown (1974)'),
2  (tensor(1.0517), 'Wrong Trousers, The (1993)'),
3  (tensor(1.0271), 'Casablanca (1942)'),
4  (tensor(1.0193), 'Close Shave, A (1995)'),
5  (tensor(1.0093), 'Secrets & Lies (1996)'),
6  (tensor(0.9771), 'Lawrence of Arabia (1962)'),
7  (tensor(0.9724), '12 Angry Men (1957)'),
8  (tensor(0.9660), 'Some Folks Call It a Sling Blade (1993)'),
9  (tensor(0.9517), 'Ran (1985)'),
10 (tensor(0.9460), 'Third Man, The (1949)']
```

```
1 | sorted(movie_comp, key=itemgetter(0))[:10]
```

```
1 [(tensor(-1.2521), 'Jungle2Jungle (1997)'),
2  (tensor(-1.1917), 'Children of the Corn: The Gathering (1996)'),
3  (tensor(-1.1746), 'Home Alone 3 (1997)'),
4  (tensor(-1.1325), "McHale's Navy (1997)"),
5  (tensor(-1.1266), 'Bio-Dome (1996)'),
6  (tensor(-1.1115), 'D3: The Mighty Ducks (1996)'),
7  (tensor(-1.1062), 'Leave It to Beaver (1997)'),
8  (tensor(-1.1051), 'Congo (1995)'),
9  (tensor(-1.0934), 'Batman & Robin (1997)'),
10 (tensor(-1.0904), 'Flipper (1996)']
```

《Chinatown》，确实是经典的Jack Nicholson电影。每个人都知道《Casablanca》。像《Wrong Trousers》基本是经典的粘土动画电影，等等。这确实在鉴赏角度上水平比较高的电影。另一方面，大概《Home Alone 3》不是鉴赏家非常喜爱的电影。不是说没有人不喜欢它，但大概和喜欢《Secrets & Lies》的不是同一类人。你可以看到，它找出来了一些电影特征，这些特征和人们的喜好是对应的。

我们看下另外一个特征。

```
1 | movie_comp = [(f, i) for f,i in zip(fac1, top_movies)]
```

```
1 | sorted(movie_comp, key=itemgetter(0), reverse=True)[:10]
```

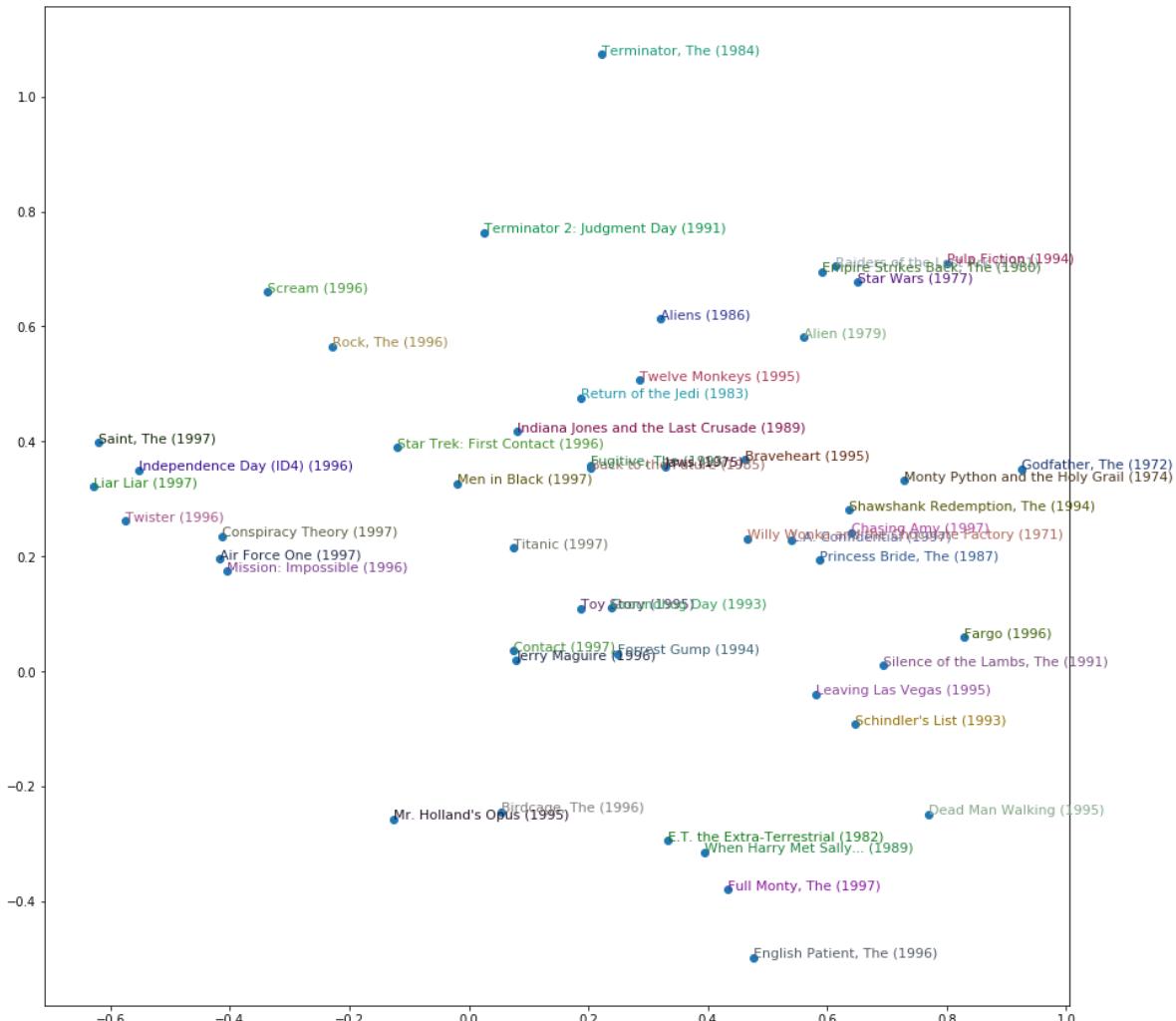
```
1 [(tensor(0.8120), 'Ready to Wear (Pret-A-Porter) (1994)'),
2  (tensor(0.7939), 'Keys to Tulsa (1997)'),
3  (tensor(0.7862), 'Nosferatu (Nosferatu, eine Symphonie des Grauens) (1922)'),
4  (tensor(0.7634), 'Trainspotting (1996)'),
5  (tensor(0.7494), 'Brazil (1985)'),
6  (tensor(0.7492), 'Heavenly Creatures (1994)'),
7  (tensor(0.7446), 'Clockwork Orange, A (1971)'),
8  (tensor(0.7420), 'Beavis and Butt-head Do America (1996)'),
9  (tensor(0.7271), 'Rosencrantz and Guildenstern Are Dead (1990)'),
10 (tensor(0.7249), 'Jude (1996)']
```

```
1 | sorted(movie_comp, key=itemgetter(0))[:10]
```

```
1 [(tensor(-1.1900), 'Braveheart (1995)'),
 2  (tensor(-1.0113), 'Raiders of the Lost Ark (1981)'),
 3  (tensor(-0.9670), 'Titanic (1997)'),
 4  (tensor(-0.9409), 'Forrest Gump (1994)'),
 5  (tensor(-0.9151), "It's a Wonderful Life (1946"),
 6  (tensor(-0.8721), 'American President, The (1995)'),
 7  (tensor(-0.8211), 'Top Gun (1986)'),
 8  (tensor(-0.8207), 'Hunt for Red October, The (1990)'),
 9  (tensor(-0.8177), 'Sleepless in Seattle (1993)'),
10  (tensor(-0.8114), 'Pretty Woman (1990)')]
```

这是编号1的factor。它看起来找到了...好吧，它们只是你可以和全家人一起看的大片。实际上不是这样，《Trainspotting (猜火车)》是很现实的电影。它大概是找到了有吸引力 (interesting) 这个特征。我们可以把它们画出来。

```
1 idxs = np.random.choice(len(top_movies), 50, replace=False)
2 idxs = list(range(50))
3 X = fac0[idxs]
4 Y = fac2[idxs]
5 plt.figure(figsize=(15,15))
6 plt.scatter(X, Y)
7 for i, x, y in zip(top_movies[idxs], X, Y):
8     plt.text(x,y,i, color=np.random.rand(3)*0.7, fontsize=11)
9 plt.show()
```



我只随机选择一部分，来方便查看。这只是50个受欢迎的电影，根据它们被评分的次数选出的。对这个factor，《The Terminators》很高。《The English Patient》和《Schindler's List》在另一端。《The Godfather》和《Monty Python》在最右边，《Independence Day》和《Liar Liar》在最左边。这样你就明白了。这个factor是有趣（fun）。如果你能在工作中或者其他数据集里找出一些特征，做些分析，会是很有意思的。

**提问：**为什么我有时训练时会得到负的损失度？ [\[59:49\]](#)

不应该这样。你有些地方做错了。因为大家会上传练习，我猜其他人也遇到过，你可以把它发到论坛上。我们今天课间休息后，会学习交叉熵（cross entropy）和负对数似然（negative log likelihood）。它们是对输入有特定要求的损失函数。如果你的输入不满足条件，它们会给出很奇怪的结果。你大概做错了什么，要修改一下。

## collab\_learner [1:00:43]

```
def collab_learner(data, n_factors:int=None, use_nn:bool=False, metrics=None,
                    emb_szs:Dict[str,int]=None, wd:float=0.01, **kwargs) -> Learner:
    "Create a Learner for collaborative filtering on `data`."
    emb_szs = data.get_emb_szs(ifnone(emb_szs, {}))
    u,m = data.classes.values()
    if use_nn: model = EmbeddingNN(emb_szs=emb_szs, **kwargs)
    else:      model = EmbeddingDotBias(n_factors, len(u), len(m), **kwargs)
    return CollabLearner(data, model, metrics=metrics, wd=wd)
```

这是 `collab_learner` 函数。像其他的一样，这个learner函数也接收一个 `data bunch`。通常 learner 对象也要传入你要指定的网络架构信息。这个learner里，只有一个和这个有关的参数 `use_nn`，表示你想用一个多层神经网络还是经典的协同过滤。今天我们只讲经典的协同过滤，可能也会简单地讲下神经网络，看情况。

这里究竟做了什么？我们基本上是创建了一个 `EmbeddingDotBias` 模型，然后返回了一个learner，这个learner包含了我们的数据和这个模型。显然，重要的东西是在 `EmbeddingDotBias` 里实现的，我们看下它。

```
class EmbeddingDotBias(nn.Module):
    "Base dot model for collaborative filtering."
    def __init__(self, n_factors:int, n_users:int, n_items:int, y_range:Tuple[float,float]=None):
        super().__init__()
        self.y_range = y_range
        (self.u_weight, self.i_weight, self.u_bias, self.i_bias) = [embedding(*o) for o in [
            (n_users, n_factors), (n_items, n_factors), (n_users, 1), (n_items, 1)
        ]]

    def forward(self, users:LongTensor, items:LongTensor) -> Tensor:
        dot = self.u_weight(users)* self.i_weight(items)
        res = dot.sum(1) + self.u_bias(users).squeeze() + self.i_bias(items).squeeze()
        if self.y_range is None: return res
        return torch.sigmoid(res) * (self.y_range[1]-self.y_range[0]) + self.y_range[0]
```

这是 `EmbeddingDotBias`。它是一个 `nn.Module`。提醒下，在PyTorch里，所有的PyTorch层和模型都是 `nn.Moudle`。它们是这样的东西：在创建时，像一个函数一样，你可以用圆括号来调用，可以传入参数。但它们不是函数，它们没有 `__call__`。通常在Python里，要让一个东西看起来像函数，你需要给它一个叫duner call（`call`）的函数。这个类里并没有。原因是，PyTorch希望你有叫 `forward` 的东西，当你像调用函数一样调用它时，PyTorch会为你调用 `forward`。

当训练模型时，要得到预测值，它会为我们调用 `forward`。预测值是在 `forward` 里计算的。可以看到，这里传入...为什么是 `users` 而不是 `user`，这是因为每次数据都是一个mini-batch。当我在 PyTorch module 里看到 `forward` 时，我一般会在大脑里忽略这里有一个 mini batch。我会当作只有一个数据。因为 PyTorch 会对所有数据做同样的处理，它自动处理这些。我们当作这里只有一个 user 就

好。这个 `self.u_weight` 是什么？它是一个embedding。我们创建了user x factor、item x factor、user x 1、item x 1的embedding。这是有意义的。user x 1的是user的bias。user x factor的是特征embedding。user x factor的是第一个tuple，它会被放入 `u_weight`，`(n_users, 1)` 是第三个，它会被放入 `u_bias`。

记住，当PyTorch创建 `nn.Module` 时，它会调用dunder init。这是我们创建权重矩阵的地方。我们没有创建真正的权重矩阵。我们一般使用PyTorch的便利函数（convenience functions），课间休息后，我们会学习一下这个。现在，只要知道这个函数会为我们创建一个embedding矩阵。它也是一个PyTorch `nn.Module`，所以，要真的传入东西到embedding矩阵并得到激活值，你要把它当成一个函数：用圆括号调用它。如果你查看PyTorch的源代码，找到 `nn.Embedding`，你可以找到里面有个叫 `.forward`，它会为我们做数组查找。

[1:05:29]

`self.u_weight(users)` 是我们传入user的地方，`self.i_weight(items)` 是我们传入item的地方。现在我们有了所有的embedding。现在，我们把它们相乘，加在一起，再加上user bias和item bias。如果我们设了 `y_range`，我们会做sigmoid操作。好了，现在你了解了整个的模型。这不是一个普通的模型。这是一个我们最近才发现的模型，它很有竞争力，甚至可能比专门做这个的团队发表的成绩还要好一些。

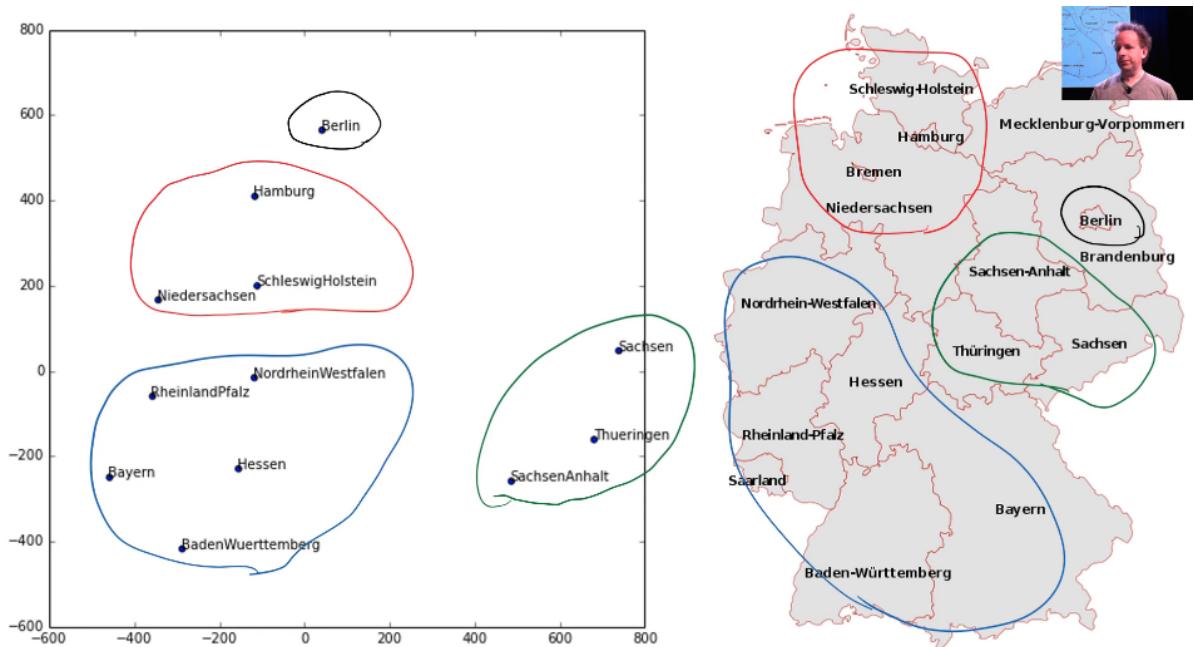
## Embeddings令人惊奇[1:07:03]

解释embedding很有用。就像这节课里我们稍后会看到的，我们在表格数据里为类别变量创建的也是embedding，这用得更广泛。再说一遍，它只是一个普通的矩阵乘以一个one-hot编码的输入，我们跳过计算和内存消耗，使用一个更有效率的方式实现它，恰好有这种有意思的效果。[有一篇有意思的论文](#)，它的作者在一个叫做Rossman的Kaggle比赛中得到了第二名。我们大概会在课程第二部分学习Rossman的更多细节。我想在课程第一部分没有时间了。它基本上是很标准的表格数据问题。有意思的地方在于预处理。他们得了第二名，得第一名的人和其他在榜单前列的非常多的人都用了很多专门的特征工程。文章作者用的特征工程比其他人都要少。他们用了神经网络，那是在2016年，那时没有人这样做。没人用神经网络处理表格数据。

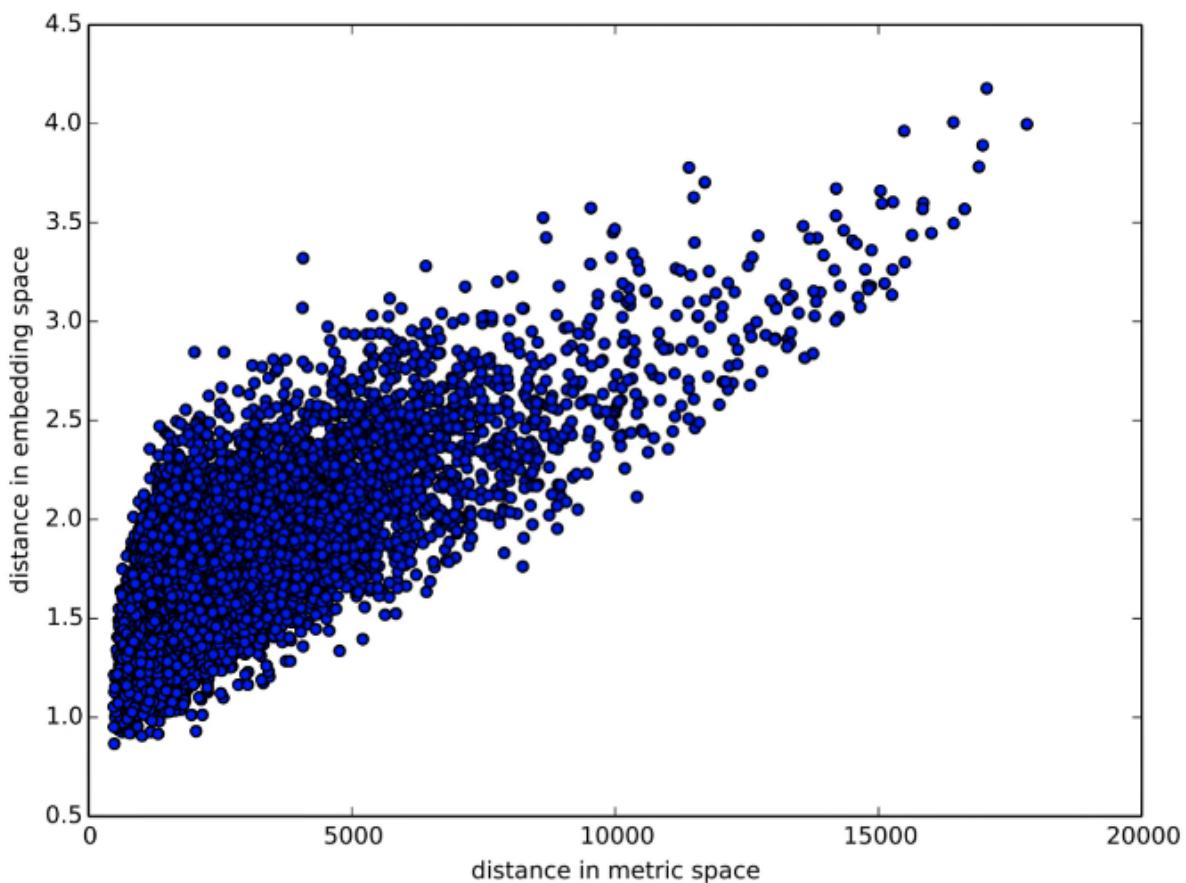
我们讲的这些东西是那时开始出现的，至少是那时才开始流行的。当我说流行时，我指的是只有一点点流行，大多数人还不知道这个方法。但这很酷，因为在他们的论文里，他们比较了多种方法的主要平均百分比误差（main average percentage error）：K近邻、随机森林、gradient boosted trees

method	MAPE	MAPE (with EE)
KNN	0.290	0.116
random forest	0.158	0.108
gradient boosted trees	0.152	0.115
neural network	0.101	0.093

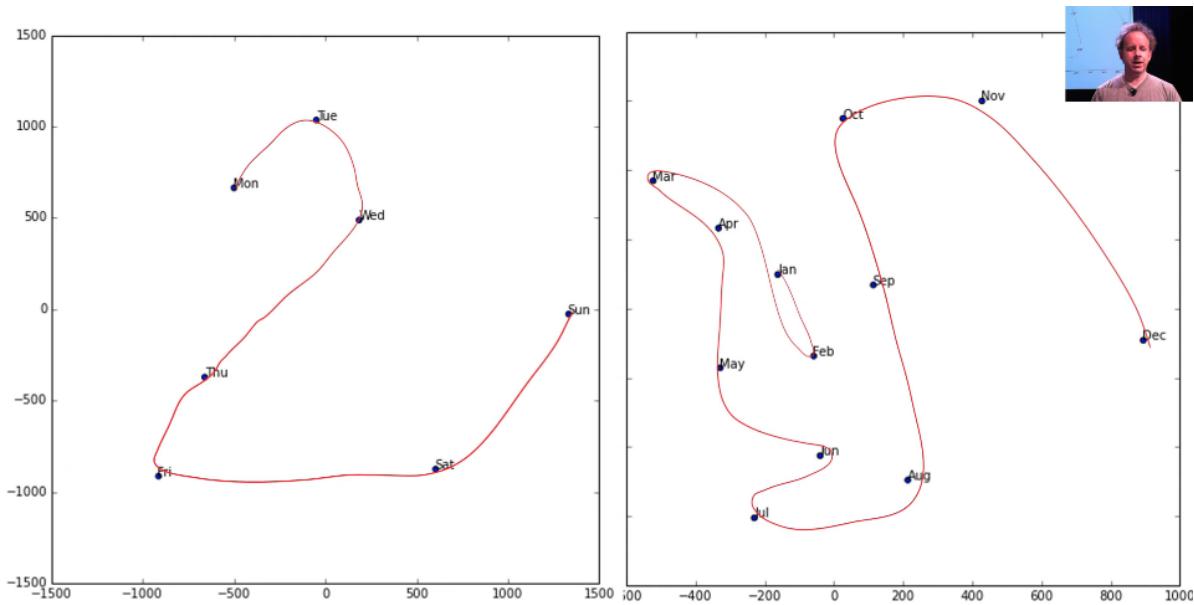
首先，可以看到，神经网络做得好很多，然后使用entity embedding后效果更好了，他们在训练完成后，给所有的任务都添加了entity embedding。添加了entity embedding的神经网络的效果是最好的，添加了空的embedding的随机森林差得远。这很好，你可以在生产、仓储、基因序列或其它各种领域的应用中训练这些实体矩阵（entity matrices），然后用在很多不同的模型中，可以用在随机森林这样比较快的方法中，也能得到很多提升。



有一点值得注意。他们为德国省份的embedding矩阵做了一个二维的投影，因为这是一个德国的连锁超市，用的方法和我们用过的一样，我记不太清他们用的是PCA还是有些细微区别的其它东西。然后出现了有趣的事情。在这个embedding空间里，我圈出了一些东西，我用相同的颜色在这个地图上圈出了这些东西，结果就像是“天呀，embedding投射 (embedding projection) 学会了地理”。实际上，它们并没有学会这个，但它们找到了在超市采购模式上相互接近的一些东西，因为这是关于预测这里会有多少销量的。里面有一些地理因素。



事实上，这是一张两个embedding向量间距离的图。你可以选一个embedding向量，看一下它的平方和与其他的向量相差多少。这是欧几里得距离（在embedding空间里的距离），然后画出它和商店在现实中的距离的对应关系，你可以看到这样明显正相关的结果。



这是一周各天的embedding空间，可以看到，它们之间有一个清晰的路径。这是一年各月的embedding空间，同样的，也有一个清晰的路径。

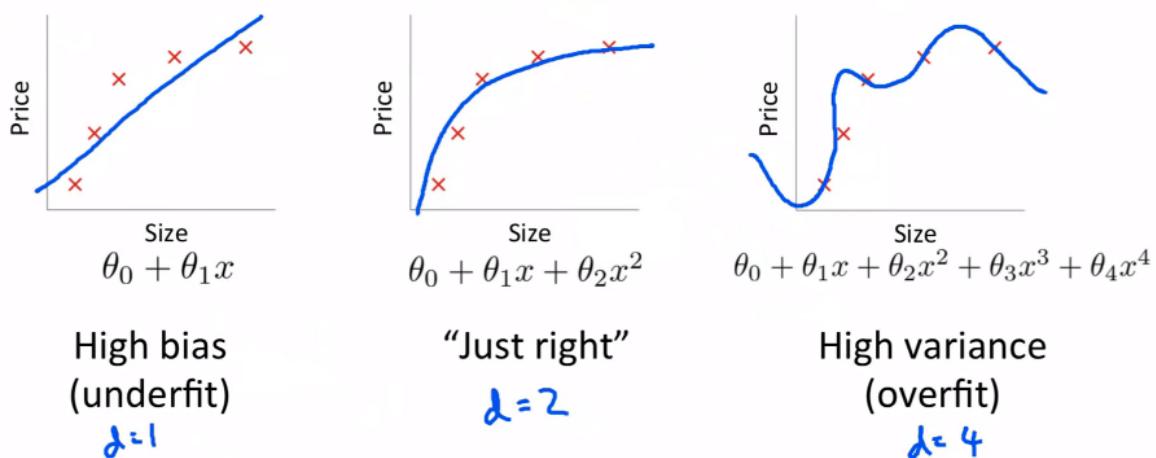
Embedding是令人惊奇的，我觉得没有人像我这样做过解释。如果你在做基因序列、或者植物物种、或者商店商品预测或者其他什么东西，可以训练一些模型，然后尝试微调一些embedding，然后用这种方式查看下相关性，把它们聚类，投影到2d空间之类的，这是很有用的。

## 正则化 (Regularization) : 权重衰减 (Weight Decay)

[1:12:09]

我们要努力确保你们能理解在这个我们构建的这个优异的collab learner模型里的每行代码在做什么。刚才没有讲这个`wd`，`wd`代表weight decay（权重衰减）。什么是权重衰减？它是一种正则化(regularization)。什么是正则化？

### Bias/variance



Andrew Ng

让我们先回来看这张简单的图，它很棒，是来自吴恩达的机器学习课程里的，上面画了一些相同的数据，和经过这些数据的不同的线。因为吴恩达在斯坦福，只能用希腊字母。如果有需要的话，我们可以用 $a + bx$ 表示 $\theta_0 + \theta_1 x$ ，它是一个直线。用希腊字母表示的话也是一个直线。这是一个二阶的多项式，有一点弯曲。这是一个高阶的多项式，非常弯曲。

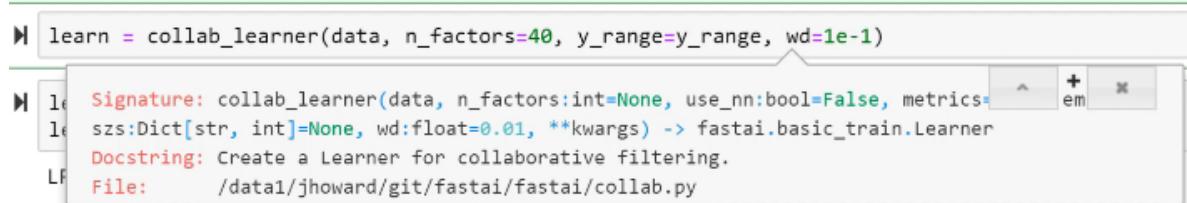
有更多参数的模型更容易长成这样。在传统的统计学中，我们会用较少的参数，因为我们不希望它长成这样。因为如果是这样的话，预测值会偏左偏右很多，这不是我们想要的。它泛化能力很差。我们过拟合了。我们通过使用较少的参数来防止过拟合。如果你们当中有人不幸被统计学、心理学、经济学或其它类似的课程洗脑过，你需要忘记这个“需要较少参数”的观点。你要知道，你被告诉这个错误的观点是因为它是一个简化的假设，实际上是因为你不希望你的函数太复杂。使用较少函数是一个让它变简单的方法。但是如果你有1000个参数，但其中999个参数都是 $1e-9$ 会怎样呢？如果它们是0会怎样呢？如果它们是0，这些参数实际上就是不存在的。为什么不用很多参数，其中很多都是很小的？这样是可行的。用参数数量来限制复杂性，这个方法的效果是非常有限的，这是一个很有问题的假设，如果你用有多少个参数来衡量复杂性，那就错了，要用特性来衡量。

为什么我们在意这个问题？为什么我想用较多的参数？因为更多的参数意味着更加非线性、更多的相互作用、更多的弯曲。现实世界里有很多弯曲的地方。但我们不希望有不必要的弯曲和不必要的相互作用。所以，我们用很多参数，然后惩罚复杂性。惩罚复杂性的一个方法（就像之前建议过的）是把参数的值加起来。现在这个方法不是很有效，因为一些参数是正的，一些是负的。把这些参数的平方加起来怎么样？这是一个很好的方法。

我们来创建一个模型，在损失函数里，我们会加上这些参数的平方和。这样会有一个问题。可能这个数字很大，以至于最好的损失度相对于这个值接近0。这样不行。我们要避免发生这样的情况，所以我们不简单地把参数的平方和加上，而是乘以一个指定的数字。在fastai里，我们选择的这个数字叫`wd`。这就是我们要做的，我们取损失函数，把参数平方和与`wd`这个数的乘积加在上面。

这个值应该是多少呢？一般，它应该是0.1。有光鲜的机器学习博士学位的人极其怀疑鄙视任何学习率大多数时候可以用 $3e-3$ 、权重衰减（weight decay）可以大多数时候用0.1的说法。但事实就是这样，我们在很多数据集上做了很多实验，很难找出0.1的权重衰减效果不好的情况。然而，我们没有把它设成默认值。实际上，我们把默认值设成了0.01。为什么，因为在少数情况，你的权重衰减太高的话，无论训练多久，都不会拟合得很好。一开始就会过拟合，你需要早点处理。

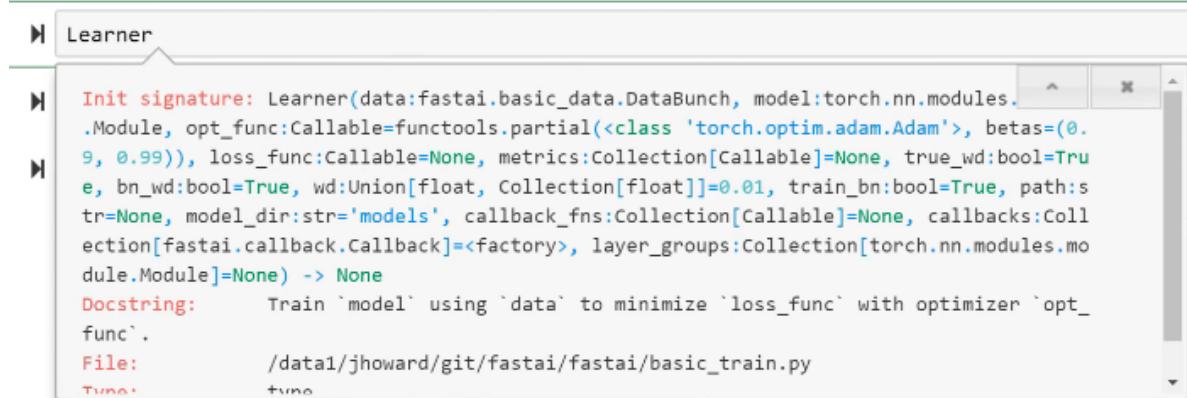
所以我们让我们的默认值保守些，但我的建议值还是这个0.1。现在你知道每个learner都有一个`wd`参数，我需要提醒你，它不会总是出现在这个列表里：



```
learn = collab_learner(data, n_factors=40, y_range=y_range, wd=1e-1)
```

Signature: `collab_learner(data, n_factors:int=None, use_nn:bool=False, metrics=None, szs:Dict[str, int]=None, wd:float=0.01, **kwargs) -> fastai.basic_train.Learner`  
Docstring: Create a Learner for collaborative filtering.  
File: /data1/jhoward/git/fastai/fastai/collab.py

因为Python里有一个`**kwargs`的概念，它是传到下一个调用的方法里的参数。所以基本上，所有的learner最终会调用这个构造函数：



```
Learner
```

```
Init signature: Learner(data:fastai.basic_data.DataBunch, model:torch.nn.modules._Module, opt_func:Callable=functools.partial(<class 'torch.optim.adam.Adam'>, betas=(0.9, 0.99)), loss_func:Callable=None, metrics:Collection[Callable]=None, true_wd:bool=True, bn_wd:bool=True, wd:Union[float, Collection[float]]=0.01, train_bn:bool=True, path:str=None, model_dir:str='models', callback_fns:Collection[Callable]=None, callbacks:Collection[fastai.callback.Callback]=<factory>, layer_groups:Collection[torch.nn.modules.module.Module]=None) -> None  
Docstring: Train `model` using `data` to minimize `loss_func` with optimizer `opt_func`.  
File: /data1/jhoward/git/fastai/fastai/basic_train.py
```

这个构造函数有一个`wd`。这只是众多这种参数中的一个，你可以在这个文档里看到它们。每次你在fastai里各种函数中构造learner时，你可以传入`wd`。用0.1替代默认的0.01通常是有用的。

## 回到 Lesson2 SGD notebook [1:19:16]

这里究竟做了什么？回头看下[lesson 2 SGD](#)会有帮助，因为今天剩下的时间里我们在这里做的所有事情都是基于这个的。

我们创建了一些数据，添加了一个损失函数MSE，然后我们创建了一个叫做 `update` 的函数，它计算了我们的预测值。这是我们的权重矩阵和输入相乘：

```
def update():
    y_hat = x@a
    loss = mse(y, y_hat)
    if t % 10 == 0: print(loss)
    loss.backward()
    with torch.no_grad():
        a.sub_(lr * a.grad)
        a.grad.zero_()
```

这只有一层，没有ReLU。我们用均方差计算损失度。我们用 `loss.backward` 计算梯度。然后我们原地减去学习率和梯度的乘积。如果你没有看过lesson2-sgd notebook，请去看一遍，因为这是我们的起点。如果你不了解它，我们现在讲的这些都没有意义。如果你在看这个视频，可以暂停下来，回去看下第二课里的这部分，确保你们学会了它。

记住 `a.sub_` 基本和 `a -=` 是一样的，因为在PyTorch里 `a.sub` 是做减法，如果你在后面加一个下划线，就代表在原地减。所以，这样更新了 `a` 这个参数，它本来是 `[-1., 1.]`，我随便选择了这些数字，这个程序会逐渐让它们变得更好。

让我们把这个写下来，在第  $t$  个 epoch 或者第  $t$  次计算这些参数（我会把它们叫做权重，这更常用），它等于上一个 epoch 的权重减去学习率乘以损失函数对第  $t - 1$  次权重的导数。

$$w_t = w_{t-1} - lr \times \frac{dL}{dw_{t-1}}$$

这就是这段代码做的事情：

```
def update():
    y_hat = x@a
    loss = mse(y, y_hat)
    if t % 10 == 0: print(loss)
    loss.backward()
    with torch.no_grad():
        a.sub_(lr * a.grad)
        a.grad.zero_()
```

我们不需要计算导数，这很麻烦，电脑可以很快地为我们计算，然后把它存储在 `grad` 里，我们可以继续了。要确保你对这个等式或者这段代码没什么疑问，它们是相同的东西。

我们的损失度是什么？损失度函数是一个关于自变量  $X$  和权重的一个函数 ( $L(x, w)$ )。在这个例子里，我们用的是的均方差，预测值和实际值之间的均方差。

$$L(x, w) = mse(\hat{y}, y)$$

$X$  和  $w$  是怎样来的？我们的预测值是模型（我们叫它  $m$ ）输出的，模型里包含了一些权重。所以我们的损失函数可以写成这样：

$$L(x, w) = mse(m(x, w), y)$$

我们今天会学习更多其他的类型的损失函数。最后创建一个 `a.grad`。

我们还要再做些事情。我们要加上权重衰减。这里它是 0.1 乘以权重的平方和。

$$L(x, w) = mse(m(x, w), y) + wd \cdot \sum w^2$$

## MNIST SGD [1:23:59]

[lesson5-sgd-mnist.ipynb](#)

我们现在不再用生成的数据，改用真实的数据，这会更有意义。我们用MNIST，手写数字数据集。我们用一个标准全连接网络来做，不用卷积网络，我们还没有学习怎样从头创建一个这样的网络的细节。这里，我们用的是[deeplearning.net](#)提供的MNIST Python pickle文件，换句话说，这是一个只有Python可以打开的文件，用它可以直接得到numpy数组。它们是扁平的numpy数组，我们不需要对它们做任何处理。取到文件，它是一个gzip文件，你可以直接 `gzip.open` 它，然后可以直接 `pickle.load` 它，要用 `encoding='latin-1'`。

```
1 | path = Path('data/mnist')
```

```
1 | path.ls()
```

```
1 | [PosixPath('data/mnist/mnist.pkl.gz')]
```

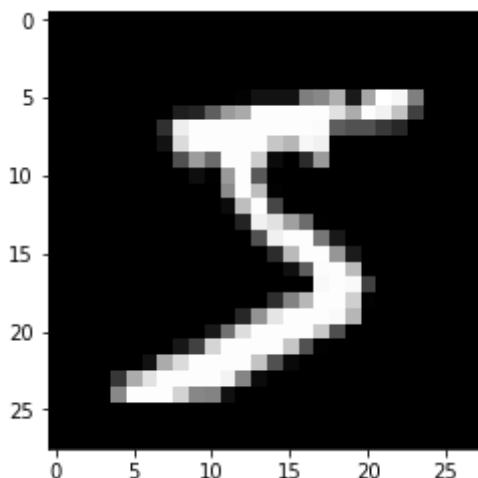
```
1 | with gzip.open(path/'mnist.pkl.gz', 'rb') as f:
2 |     ((x_train, y_train), (x_valid, y_valid), _) = pickle.load(f,
3 |         encoding='latin-1')
```

我们得到了训练集、验证集、测试集。我们不关心测试集，在Python里，如果你不关心一些东西，你可以使用这个命名成下划线（`_`）的特殊变量。你不是必须这样做。只是，如果这样的话，人们就会知道你不关心它。好的，这是我们的训练集`x`、`y`，和验证集`x`、`y`。

它（`x_train`）的形状是50,000行、784列，这784列实际上是 $28 \times 28$ 的图片。如果我把其中一个reshape成 $28 \times 28$ 的图片，把它画出来，你可以看到数字5。

```
1 | plt.imshow(x_train[0].reshape((28,28)), cmap="gray")
2 | x_train.shape
```

```
1 | (50000, 784)
```



这就是我们的数据。我们看过了reshape之前的版本，这是扁平的版本。我们会用这个扁平的版本。

现在它们是numpy array。我们需要它们变成tensor。我可以把它们都map成 `torch.tensor`，现在它们是tensor了。

```
1 | x_train,y_train,x_valid,y_valid = map(torch.tensor,
2 | (x_train,y_train,x_valid,y_valid))
3 | n,c = x_train.shape
```

```
1 | (torch.size([50000, 784]), tensor(0), tensor(9))
```

我创建了一个变量存放数据的数量，我们把它叫 `n`。我们用 `c` 来代表列数（抱歉，这不是一个好名字）。毫不意外，`y` 的最小值是0，最大值是9，这是我们要预测的实际数字。

[1:26:38]

在lesson 2 SGD里，我们创建了一个`x`，里面的数字都是1，没有用bias：

```
1 | x = torch.ones(n,2)
2 | def mse(y_hat, y): return ((y_hat-y)**2).mean()
3 | y_hat = x@a
```

我们现在不用再这样做了。我们用PyTorch自动地为我们做这个。我们以前要写自己的MSE函数，现在不用再这样做了。我们以前要写自己的矩阵乘法方法，现在不用再这样做了。我们会用PyTorch为我们做所有这些东西。

另外，重要的是，我们要用mini batch，因为这是一个相当大的数据集，我们不想一次性处理它。如果你想用mini batch，我们在这里不用太多fastai里的东西，PyTorch里有个叫 `TensorDataset` 的东西，它取两个Tensor创建一个数据集，记住，数据集是这样的东西，如果你提供一个索引，你会得到一个x值和一个y值。它看起来很像xy tuple的list。

```
1 | bs=64
2 | train_ds = TensorDataset(x_train, y_train)
3 | valid_ds = TensorDataset(x_valid, y_valid)
4 | data = DataBunch.create(train_ds, valid_ds, bs=bs)
```

当有了一个dataset后，你可以调用 `DataBunch.create`，它会为你创建data loader。data loader是这样的东西，你不用说我要第一个数据或者第五个数据，你只需要说我要“next”数据，它会给你一个mini batch，尺寸由你指定。具体来说，它会给你mini batch的X和y。我对迭代器做next（这是标准的Python）。这是 `DataBunch.create` 为你创建的训练data loader（`data.train_dl`）。你可以检查下它，你期望X的尺寸是64 x 784，因为扁平化后有784个像素，mini batch里有64个数据，Y只有64个数字，这是我们要预测的东西。

```
1 | x,y = next(iter(data.train_dl))
2 | x.shape,y.shape
```

如果你看一下 `DataBunch.create` 的代码，你可以看到里面内容不多，不用害怕。我们只是确保训练集被随机打乱顺序。我们保证数据被加载到GPU上。就是做了这两个事情，这个方法让我们写起代码来更方便。不要觉得这是什么魔法，如果不明白就看下它的源代码来确保你知道里面发生了什么。

不再用这个 `y_hat = x@a`，我们要创建一个 `nn.Module`。如果你想创建一个 `nn.Module` 来做一些新的任务，你需要继承它。继承在PyTorch里是非常普遍的。如果你对Python里的继承不熟悉，就去读一些教程，把它搞明白。主要的是，你要override构造函数dunder init（`__init__`），并且确保调用超类（super class）的构造函数（`super().__init__()`），因为 `nn.Module` 超类构造函数会做所

有让它成为一个 `nn.Module` 的工作。如果你在创建一个 PyTorch 子类，但它不能正常运行，很有可能是你遗漏了这行代码。

```
1 | class Mnist_Logistic(nn.Module):
2 |     def __init__(self):
3 |         super().__init__()
4 |         self.lin = nn.Linear(784, 10, bias=True)
5 |
6 |     def forward(self, xb): return self.lin(xb)
```

[1:30:04]

我们要增加的唯一的东西就是在类里创建一个属性，它存放了一个 `nn.Linear` 模块。什么是 `nn.Linear` 模块？它是做  $x@a$  的东西。实际上，它做的不仅仅是这个，而是  $x@a + b$ 。换句话说，我们不用再添加一列 1。这就是它做的事情。如果你想多尝试下，为什么不创建你自己的 `nn.Linear` 类呢？你可以创建叫 `MyLinear` 的类，它会花费你一两个小时（取决于你的 PyTorch 水平）。我们不想这里的任何东西成为魔法，你们已经知道了创建这个类的所有必要的知识。这是你们这周要做的作业。这不是新的应用，尝试从头开始写些这样的东西，让它们运行起来。学习怎样调试它们、看看它的输入输出，等等。

我们可以直接用 `nn.Linear`，它里面会有一个 `def forward`，里面会做  $a@x + b$ 。在 `forward` 里，怎样计算这个的结果呢？记住，每个 `nn.Module` 都像一个函数，我们传入  $x$  mini batch 到 `self.lin`，我用 `xb` 表示  $X$  的一个 batch，这会返回对这个 mini batch  $a@x + b$  的结果。

这是一个逻辑回归模型。逻辑回归模型也是一个没有隐藏层的神经网络，这是一个单层神经网络，没有非线性层。

因为我们自己写了一些东西，我们需要手动把权重矩阵放到 GPU 里。输入 `.cuda()` 就可以做到。

```
1 | model = Mnist_Logistic().cuda()
```

```
1 | model
```

```
1 | Mnist_Logistic(
2 |     (lin): Linear(in_features=784, out_features=10, bias=True)
3 | )
```

这就是我们的模型。可以看到 `nn.Module` 自动给出了它的信息。它自动存储了这个 `.lin`，告诉我们里面有什么。

```
1 | model.lin
```

```
1 | Linear(in_features=784, out_features=10, bias=True)
```

```
1 | model(x).shape
```

```
1 | torch.size([64, 10])
```

```
1 | [p.shape for p in model.parameters()]
```

```
1 | [torch.size([10, 784]), torch.size([10])]
```

PyTorch提供了很多简短的便捷方法。现在如果你查看 `model.lin`，你可以看到它是这样的。

最值得指出的是，我们的模型自动得到了一批方法和属性。最值得注意的是叫做 `parameters` 的东西，它存放了我们图片里所有黄色方块。它存放了我们的参数。它存放了权重矩阵和bias矩阵。如果我们看下 `p.shape for p in model.parameters()`，里面有  $10 \times 784$  的 tensor，和一个长度是 10 的 tensor。它们是什么？ $10 \times 784$  的这个，它会接收 784 维的输入，输出 10 维的数组。这很简单，因为我们的输入是 784 维的，我们需要得到对 10 个数字的概率。然后，我们得到 10 个激活值，然后把 bias 加到上面。这是一个长度是 10 的向量。可以看到，我们创建的这个模型里有做  $a @ x + b$  需要的东西。

[1:33:40]

```
1 | lr=2e-2
```

```
1 | loss_func = nn.CrossEntropyLoss()
```

我们设置一个学习率。稍后我们会回来讲这个损失函数，对这个问题，我们不能用 MSE，因为我们不是要看“你们有多近”。你预测了 3，实际上是 4，啊，这很近。错！在你预测别人画的数字是什么时，3 和 4 差很多，像 0 和 4 之间的差别一样大。我们不再用 MSE，我们要用交叉熵（cross-entropy）损失度，晚些我们会学习它。

```
1 | def update(x,y,lr):
2 |     wd = 1e-5
3 |     y_hat = model(x)
4 |     # weight decay
5 |     w2 = 0.
6 |     for p in model.parameters(): w2 += (p**2).sum()
7 |     # add to regular loss
8 |     loss = loss_func(y_hat, y) + w2*wd
9 |     loss.backward()
10 |    with torch.no_grad():
11 |        for p in model.parameters():
12 |            p.sub_(lr * p.grad)
13 |            p.grad.zero_()
14 |    return loss.item()
```

这是我们的 `update` 函数。我从 lesson 2 SGD 里把它拷过来，但现在我们使用我们的 `model`，不再使用 `a @ x`。我们像使用函数一样调用 `model`，来得到 `y_hat`。我们调用 `loss_func` 来得到损失度，不再用 MSE。其余的还是和以前一样的，除了遍历每个参数，执行 `parameter.sub_(learning_rate*gradient)`，我们对每个参数循环。因为 PyTorch 为我们在 dunder init 里创建的东西自动创建了一个参数列表。

看，我还添加了一些其他的东西。我创建了这个 `w2`，为遍历 `model.parameters()` 里的每个 `p`，把平方和加到上面。`w2` 现在存放了权重的平方和。然后用一个数乘以它，这里设的是 `1e-5`。这样，我就实现了权重衰减。当人们讲权重衰减时，它不是一个神奇的魔法，不是什么包含数千行 CUDA C++ 代码的复杂的东西。它只是 Python 里这两行代码：

```
1 |     w2 = 0.
2 |     for p in model.parameters(): w2 += (p**2).sum()
```

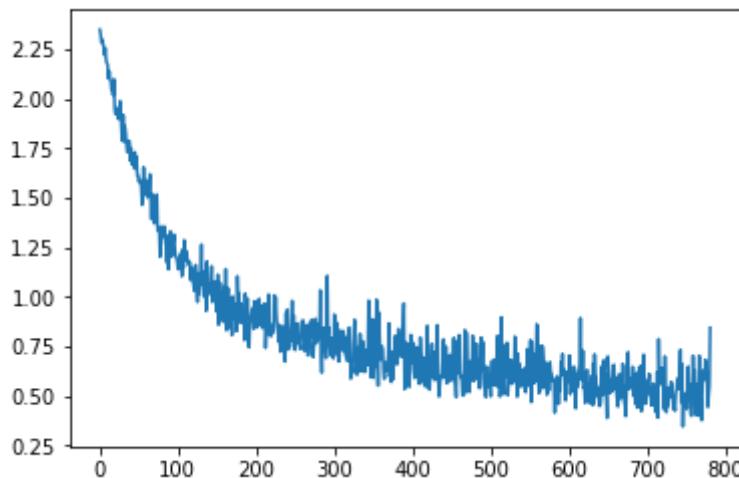
这就是权重衰减。这不是一个现在够用就好的简化版本，这就是 权重衰减。这是全部的东西。

就是这样。有两种理解权重衰减的方式。一种就是我们在添加权重平方和，这看起来是一个合理的事情，确实如此。好了，我们继续，运行这个。

```
1 | losses = [update(x,y,lr) for x,y in data.train_dl]
```

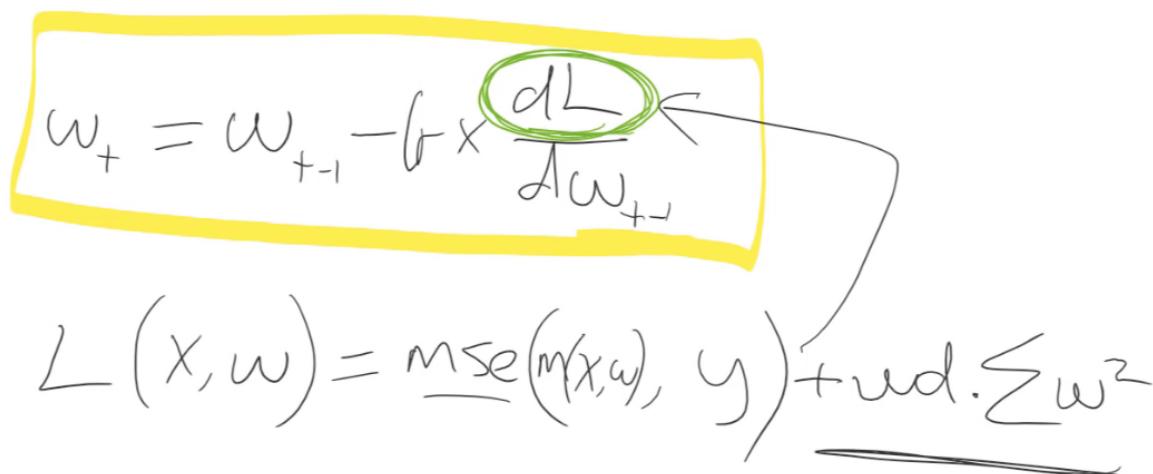
这里，我得到了一个列表，它遍历data loader。data loader每次返回一个mini batch，给出X和Y，我会对每一个都调用update。每一次都返回一个损失度。现在PyTorch张量是存放在GPU上，因为我在GPU运行的。它用所有这些东西来计算梯度，它会用大量的内存。如果你调用标量张量 (scalar tensor) 的.item()，会把它转成一个普通的Python数字。这代表我要返回普通Python数字。

```
1 | plt.plot(losses);
```



然后我可以画出它们，就是这样。我的损失度在下降。使用这个来检查它在按照你的预期运行，这很好。当我们离最终答案越来越近，它的波动越来越大，因为我们离我们应该在的位置很近了。在权重空间里，它可能变得更扁平了，所以我们跳得更远了。所以你可以看到为什么我们想减小学习率（学习率退火learning rate annealing）。

现在是这样的。


$$w_+ = w_{+-} - lr \times \frac{dL}{dw_{+-}}$$
$$L(x, w) = \text{mse}(m(x, w), y) + wd \cdot \sum w^2$$

这个 $wd \cdot \sum w^2$ 只对训练神经网络有意义，因为它出现在这个 $dL$ 这里。我们取它的梯度。这是真正更新权重的东西。对 $wd \cdot \sum w^2$ 来说，真正有意义的东西是它的梯度。这里我们不用太多的数学，我想我们可以处理它。如果你记得你的高中数学，这整个东西的梯度等于每部分的梯度的和。我们只求 $wd \cdot \sum w^2$ 的梯度，因为 $\text{mse}(m(x, w), y)$ 的梯度和之前的是一样的。那么 $wd \cdot \sum w^2$ 的梯度是什么呢？

让我们去掉求和，假装这里只有一个参数。这不会改变它的性质。那么 $wd \cdot w^2$ 对于 $w$ 的梯度是什么呢？

$$\frac{d}{dw} wd \cdot w^2 = 2wd \cdot w$$

它只是 $2wd \cdot w$ 。记住， $wd$ 是一个常量，在这段循环代码里，它的值是 $1e-5$ 。 $w$ 是我们的权重。我们可以用 $2wd$ 来替代 $wd$ ，这不会改变问题的性质，所以我们可以不管这个2。换句话说，权重衰减做的所有事情就是在每次处理一个batch时，减去常量乘以权重。这就是为什么它被叫做权重衰减的原因。

- 当它是这种形式 $wd \cdot w^2$ ，添加平方到损失函数时，被叫做**L2 正则化 (L2 regularization)**
- 当它是这种形式 $wd \cdot w$ ，在梯度上减去 $wd$ 和权重的乘积时，被叫做**weight decay (权重衰减)**

这在数学上是相同的。对我们目前看到的所有东西，在数学上其实都是相同的。稍后，我们会看到一个例外，这会比较有意思。这是一个很重要的工具。你可以做巨大的神经网络，通过添加更多的权重衰减来避免过拟合。你可以把适中的大尺寸的模型用在小型的数据集上，通过权重衰减避免过拟合。这不是魔法。你可能还会遇到这样的情况：添加很多权重衰减后，不会再出现过拟合，而是会发现数据不够，不能训练得很好。会出现这样的情况。至少，这是现在你可以运用的东西了。

## MNIST 神经网络 [1:40:33]

现在我们有了这个`update`函数，我们可以从头构建一个神经网络，用MNIST神经网络替代这个`Mnist_Logistic`。

```
1 | class Mnist_NN(nn.Module):  
2 |     def __init__(self):  
3 |         super().__init__()  
4 |         self.lin1 = nn.Linear(784, 50, bias=True)  
5 |         self.lin2 = nn.Linear(50, 10, bias=True)  
6 |  
7 |     def forward(self, xb):  
8 |         x = self.lin1(xb)  
9 |         x = F.relu(x)  
10 |        return self.lin2(x)
```

现在我们只需要两个线性层。在第一个里面，我们可以用一个大小是50的矩阵。我们要保证第二个线性层的输入是50，这样才能对应上。最后一层的输出大小应该是10，因为我们要预测的类别的数量是10。所以我们的`forward`做了这些：

- 运行一个线性层
- 计算ReLU
- 运行第二个线性层

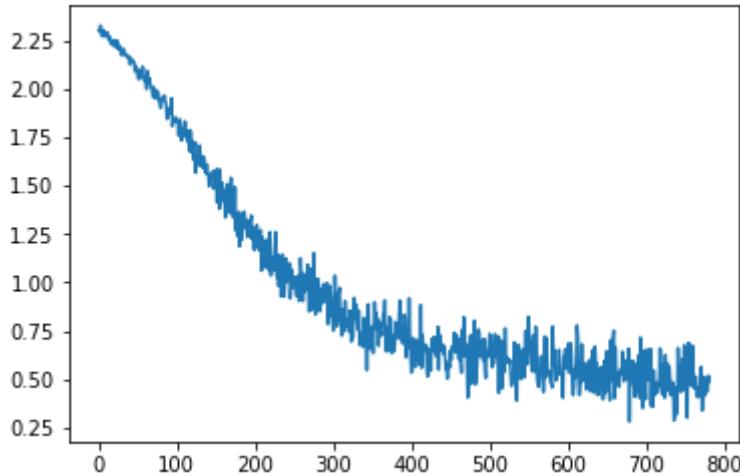
现在我们实际上从头创建了一个神经网络。我们没有自己写`nn.Linear`，但是你可以自己写一个，或者可以直接用矩阵。

然后，我们可以执行`model`的`.cuda()`方法，再对同样的`update`函数计算损失度，就是这样。

```
1 | model = Mnist_NN().cuda()
```

```
1 | losses = [update(x,y,lr) for x,y in data.train_dl]
```

```
1 | plt.plot(losses);
```



这里就可以看到为什么说神经网络的概念很简单。只要你有可以做梯度下降的东西，你就可以尝试不同的模型。你可以添加更多PyTorch里的东西。为什么不直接用`opt = optim.something`，不再自己做所有的事情（`update`函数）？目前，我们用到得“something”是SGD。

```
▶ def update(x,y,lr):
    opt = optim.SGD(model.parameters(), lr)
    y_hat = model(x)
    loss = loss_func(y_hat, y)
    loss.backward()
    opt.step()
    opt.zero_grad()
    return loss.item()
```

现在你告诉PyTorch，让它用SGD优化这些参数。并且，不再用`for p in parameters: p -= lr * p.grad`，而是直接用`opt.step()`。这是相同的东西。只是代码少了，做的事情还是一样的。值得注意的是，现在你可以用`Adam`替代`SGD`，你甚至可以添加权重衰减，PyTorch里有更多现成的东西。这就是我们为什么想用`optim.各种东西`。这实际上就是我们在fastai里做的。

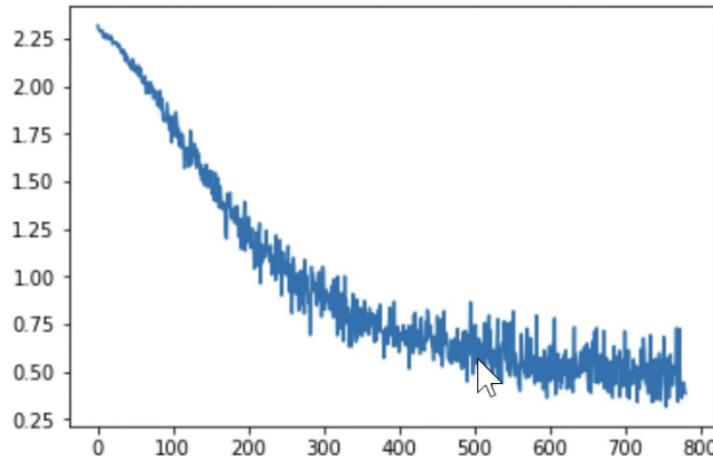
[1:42:54]

如果你使用`optim.SGD`，图形是之前这样的：

```
In [30]: def update(x,y,lr):
    opt = optim.SGD(model.parameters(), lr)
    y_hat = model(x)
    loss = loss_func(y_hat, y)
    loss.backward()
    opt.step()
    opt.zero_grad()
    return loss.item()
```

```
In [31]: losses = [update(x,y,lr) for x,y in data.train_dl]
```

```
In [32]: plt.plot(losses);
```

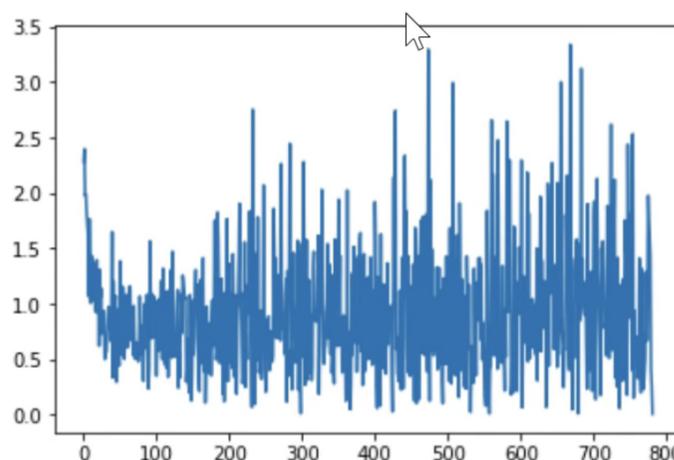


如果我们换成另外一个优化器 (optimizer) , 看看会发生什么:

```
In [37]: def update(x,y,lr):
    opt = optim.Adam(model.parameters(), lr)
    y_hat = model(x)
    loss = loss_func(y_hat, y)
    loss.backward()
    opt.step()
    opt.zero_grad()
    return loss.item()
```

```
In [38]: losses = [update(x,y,lr) for x,y in data.train_dl]
```

```
In [39]: plt.plot(losses);
```

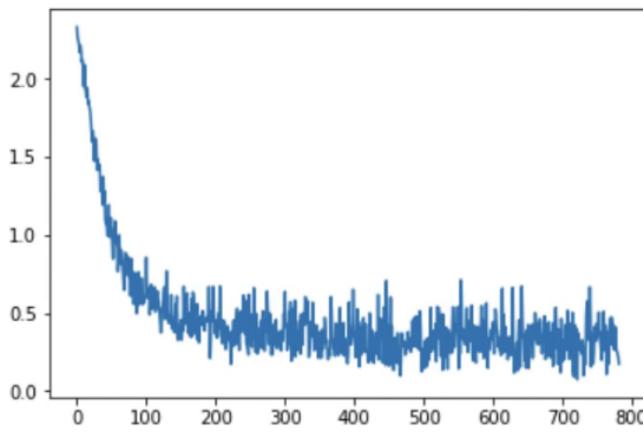


它偏离 (diverged) 了。我们看到过一个学生做的很棒的图片，展示了偏离是什么样的。这是在你训练时，它的样子。因为我们用了不同的优化器，我们需要用不同的学习率。在它偏离时，你不能继续训练，因为权重会很大很大或者很小很小，它们不会再回到正确的值。所以我们重新开始。

```
In [41]: def update(x,y,lr):
    opt = optim.Adam(model.parameters(), lr)
    y_hat = model(x)
    loss = loss_func(y_hat, y)
    loss.backward()
    opt.step()
    opt.zero_grad()
    return loss.item()

In [42]: losses = [update(x,y,1e-3) for x,y in data.train_dl]

In [43]: plt.plot(losses);
```



好，这是一个比较好的学习率。看这里，我们在第200个epoch到达了0.5以下。之前，SGD的版本里，它都没能达到这个水平。那么发生了什么？什么是Adam。让我来告诉你们。

## Adam [1:43:56]

[graddesc.xlsx](#)

我们要在Excel里做梯度下降。这是一些随机生成的数据：

	A	B	C	D
1	b	const		30
2	a	slope		2
3				
4		x	y	
5			18	66
6			28	86
7			85	200
8			38	106
9			28	86
10			40	110
11			77	184
12			9	48
13			30	90
14			61	152
15			8	46
16			40	110
17			55	140
18			90	210
19			18	66
20			32	94
21			13	56
22			55	140
23			56	142
24			93	216
25			89	208
26			47	124
27			75	180
28				

这是随机生成的X，这些Y都是用  $ax+b$  计算出来的，a是2，b是30。这就是我们要尝试拟合的数据。这是SGD：

graddesc.xlsx - Excel												
	A	B	C	D	E	F	G	H	I	J	K	L
1	Reset	intercept	1	learn	0.0001				de/db=2(ax+b-y)			
2	Run	slope	1					e=(ax+b-y)^2 de/da=x^2(ax+b-y)				
3	x	y	intercept	slope	y_pred	err^2	errb1	est de/db	err1	est de/da	de/db	de/da
4	14	58	1	1	15	1,849.00	1,848.14	-85.99	1,836.98	-1,202.04	-86.00	-1,204.00
5	86	202	1.0086	1.1204	97.363	10,948.90	10,946.81	-209.26	10,769.67	-17,923.60	-209.27	-17,997.56
6	28	86	1.029527	2.9202	82.7939	10.28	10.22	-6.40	8.56	-171.70	-6.41	-179.54
7	51	132	1.030169	2.9381	150.874	356.22	356.60	37.76	375.73	1,951.14	37.75	1,925.13
8	28	86	1.026394	2.7456	77.9031	65.56	65.40	-16.18	61.10	-445.58	-16.19	-453.42
9	29	88	1.028013	2.7909	81.9653	36.42	36.30	-12.06	33.00	-341.60	-12.07	-350.01
10	72	174	1.02922	2.8259	204.497	930.07	930.68	61.00	974.50	4,443.41	60.99	4,391.57
11	62	154	1.023121	2.3868	149.004	24.96	24.86	-9.98	19.15	-581.09	-9.99	-619.53
12	84	198	1.02412	2.4487	206.718	76.01	76.18	17.45	91.36	1,535.20	17.44	1,464.64
13	15	60	1.022376	2.3023	35.5565	597.49	597.00	-48.88	590.17	-731.06	-48.89	-733.31
14	42	114	1.027265	2.3756	100.803	174.17	173.91	-26.38	163.26	-1,090.94	-26.39	-1,108.58
15	62	154	1.029905	2.4865	155.191	1.42	1.44	2.39	3.28	186.07	2.38	147.63
16	47	124	1.029666	2.4717	117.2	46.25	46.11	-13.59	40.07	-617.15	-13.60	-639.24
17	35	100	1.031027	2.5356	89.7779	104.49	104.29	-20.43	97.46	-703.30	-20.44	-715.55
18	9	48	1.033071	2.6072	24.4977	552.36	551.89	-46.99	548.14	-422.23	-47.00	-423.04
19	38	106	1.037771	2.6495	101.718	18.33	18.25	-8.55	15.22	-310.98	-8.56	-325.42
20	44	118	1.038628	2.682	119.048	1.10	1.12	2.11	2.21	111.56	2.10	92.20
21	99	228	1.038418	2.6728	265.646	1,417.23	1,417.98	75.30	1,492.75	7,551.94	75.29	7,453.93

我们需要用SGD来做。在我们lesson 2 SGD notebook里，我们把整个数据集做一个batch来处理。在刚刚的notebook里，我们做了minibatch。在这个Excel里，我们要做在线梯度下降（online gradient descent），就是每一行作为一个batch。batch的大小是1。

像往常一样，我们随机选择一个斜率（slope）和截距（intercept），我选择用1，这没什么影响。我把数据拷到这里。这是x, y, 截距（C1），斜率（C2），它们的值是1。我会引用这个单元格（C1）。

用这个截距和斜率得到的预测值是14乘以1，再加1，结果是15（E4），这是方差（F4）：

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1		intercept	1	learn	0.0001			de/db=2(ax+b-y)						
2		slope	1				e=(ax+b-y)^2 de/da=x^2(ax+b-y)							
3	x	y	intercept	slope	y_pred	err^2	errb1	est de/db	erra1	est de/da	de/db	de/da	new a	new b
4	14	58	1	1	15	1,849.00	1,848.14	-85.99	1,836.98	-1,202.04	-86.00	-1,204.00	1.12	1.01

现在，我需要计算梯度，这样才能更新。有两种计算梯度的方式。一个是数学分析，你可以直接在Wolfram Alpha或其他工具中查出梯度，你可以手工计算，或者用工具查，这是得到的梯度（I1, I2）  
de/db=2(ax+b-y)。

或者你可以用有限差分（finite differencing），因为梯度就是当变量做微小的变动时，结果移动的距离除以变量移动的距离。让我们做一个很小的变动。

	A	B	C	D	E	F	G	H	I	J
1	Reset	intercept	1	learn	0.0001			de/db=2(ax+b-y)		
2	Run	slope	1				e=(ax+b-y)^2 de/da=x^2(ax+b-y)			
3	x	y	intercept	slope	y_pred	err^2	errb1	est de/db	erra1	est de/d
4	14	58	1	1	15	1,849.00	=((C4+0.01)+A4*D4)-B4)^2			-1,202.
5	86	202	1.0086	1.1204	97.363	10,948.90	10,946.81	-209.26	10,769.67	-17,923.
6	28	86	1.029527	2.9202	82.7939	10.28	10.22	-6.40	8.56	-171.
7	51	132	1.030169	2.9381	150.874	356.22	356.60	37.76	375.73	1,951.
8	28	86	1.026394	2.7456	77.9031	65.56	65.40	-16.18	61.10	-445.
9	29	88	1.028013	2.7909	81.9653	36.42	36.30	-12.06	33.00	-341.

这里，我们在截距上加了0.01（G4），然后计算损失度，你可以看到我们的损失度下降了一点，那我们的导数就是这个差值除以0.01（H4）：

	A	B	C	D	E	F	G	H	I	J
1	Reset	intercept	1	learn	0.0001			de/db=2(ax+b-y)		
2	Run	slope	1				e=(ax+b-y)^2 de/da=x^2(ax+b-y)			
3	x	y	intercept	slope	y_pred	err^2	errb1	est de/db	erra1	est de/d
4	14	58	1	1	15	1,849.00	1,848.14	=(G4-F4)/0.01		-1,202.
5	86	202	1.0086	1.1204	97.363	10,948.90	10,946.81	-209.26	10,769.67	-17,923.
6	28	86	1.029527	2.9202	82.7939	10.28	10.22	-6.40	8.56	-171.
7	51	132	1.030169	2.9381	150.874	356.22	356.60	37.76	375.73	1,951.
8	28	86	1.026394	2.7456	77.9031	65.56	65.40	-16.18	61.10	-445.
9	29	88	1.028013	2.7909	81.9653	36.42	36.30	-12.06	33.00	-341.

这叫做有限差分。你可以用有限差分来求导。它很慢。在实践中我们不使用它，但它对单单做检查很有用。我们可以对a（斜率）做相同的事，把它加0.01，用差值除以0.01。

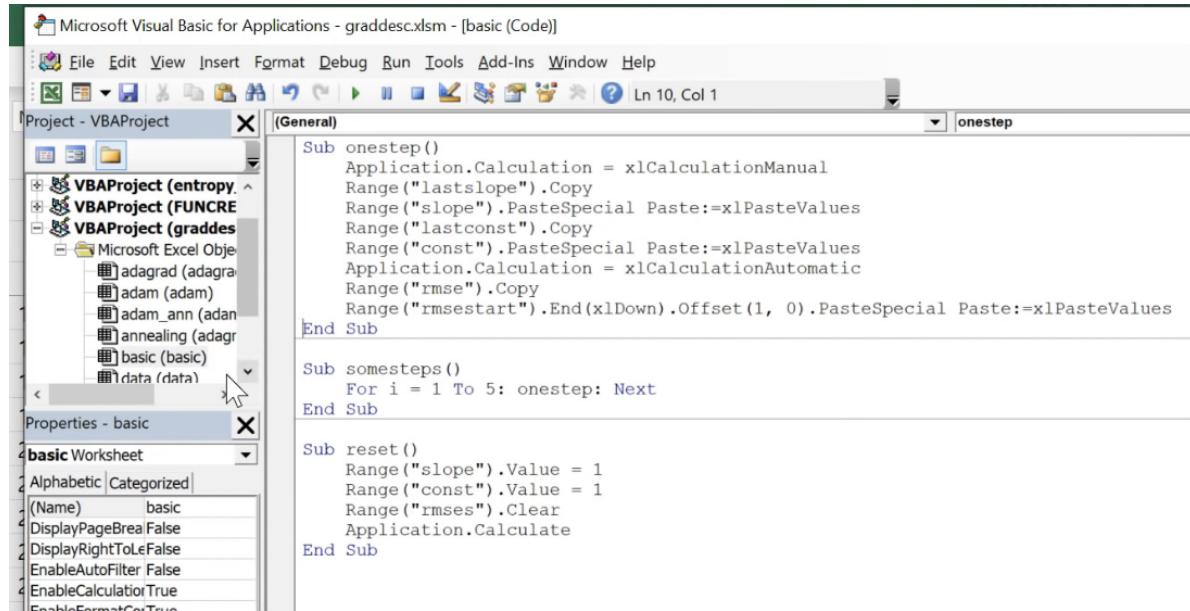
	intercept	1	learn	0.0001			de/db=2(ax+b-y)						
	slope	1					e=(ax+b-y)^2 de/da=x^2(ax+b-y)						
x	y	intercept	slope	y_pred	err^2	errb1	est de/db	erra1	est de/da	de/db	de/da	new a	new b
14	58	1	1	15	1,849.00	1,848.14	-85.99	1,836.98	-1,202.04	-86.00	-1,204.00	1.12	1.01
86	202	1.0086	1.1204	97.363	10,948.90	10,946.81	-209.26	10,769.67	-17,923.60	-209.27	-17,997.56	2.92	1.03
28	86	1.029527	2.9202	82.7939	10.28	10.22	-6.40	8.56	-171.70	-6.41	-179.54	2.94	1.03

我说过，我们可以直接用导数分析计算导数，你可以看到 `est de/db` 和 `de/db` 非常接近，和我们期望的一样（对 `est de/da` 和 `de/da` 来说也是一样）。

按照梯度下降，我们取权重(`slope`, D列)的当前值，减去学习率和导数的乘积，得到这个结果 (`new a`, M列。 `new b`, N列)。现在，我们可以把这个截距和斜率拷贝的下一行，再做一遍。做很多次，最后我们完成了一个epoch。

epoch的最后，我们可以说，“太棒了，这是我们的斜率，我们把它拷贝到存放斜率的位置 (C2)，这是我们的截距，我们把它拷贝到存放截距的位置 (C1)，现在用它们做下一个epoch”。

复制粘贴有点麻烦，我创建了一个复杂的宏来帮你们做复制粘贴（我只是录制了它）。我创建了一个复杂的循环来做5次：

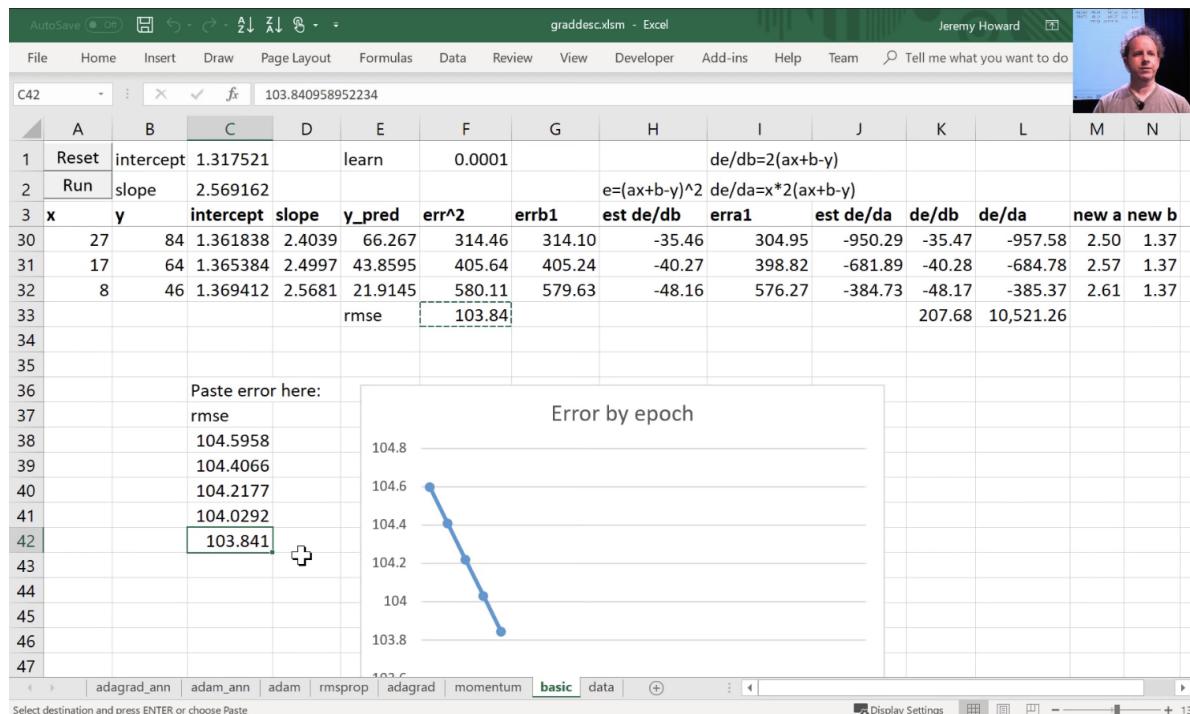


```
Sub onestep()
    Application.Calculation = xlCalculationManual
    Range("lastslope").Copy
    Range("slope").PasteSpecial Paste:=xlPasteValues
    Range("lastconst").Copy
    Range("const").PasteSpecial Paste:=xlPasteValues
    Application.Calculation = xlCalculationAutomatic
    Range("rmse").Copy
    Range("rmsestart").End(xlDown).Offset(1, 0).PasteSpecial Paste:=xlPasteValues
End Sub

Sub somesteps()
    For i = 1 To 5: onestep: Next
End Sub

Sub reset()
    Range("slope").Value = 1
    Range("const").Value = 1
    Range("rmse").Clear
    Application.Calculate
End Sub
```

我把它添加到Run按钮，这样，如果我按run，它会执行，计算5遍，记录下每次的偏差。



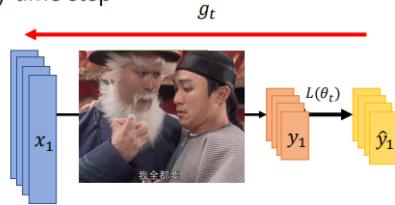
这就是SGD。可以看到，它慢得令人发指，截距被设成了30，我们只更新到了1.57，它运行的太慢了。我们来加速它。

## 动量 (Momentum) [1:48:40]

- 见李宏毅机器学习部分《Optimization》PDF，百度网盘有。

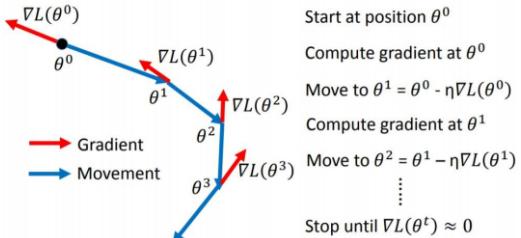
### On-line vs Off-line

- Off-line : pour all  $(x_t, \hat{y}_t)$  into the model at every time step



- The rest of this lecture will focus on the off-line cases

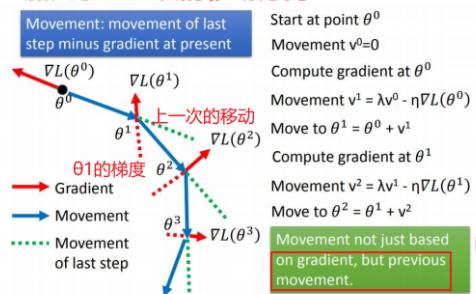
### SGD



Credit to 李宏毅老師上課投影片

### SGD with Momentum(SGDM)

#### 加入了上一次的移动方向



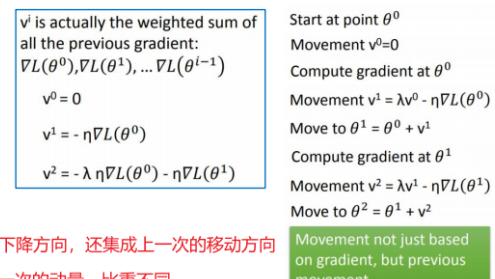
Movement: movement of last step minus gradient at present  
Start at point  $\theta^0$   
Movement  $v^0=0$   
Compute gradient at  $\theta^0$   
Movement  $v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$   
Move to  $\theta^1 = \theta^0 + v^1$   
Compute gradient at  $\theta^1$   
Movement  $v^2 = \lambda v^1 - \eta \nabla L(\theta^1)$   
Move to  $\theta^2 = \theta^1 + v^2$

Movement not just based on gradient, but previous movement.

除了当前位置的下降方向, 还集成上一次的移动方向  
梯度和上一次的动量, 比重不同

Credit to 李宏毅老師上課投影片

### SGD with Momentum(SGDM)

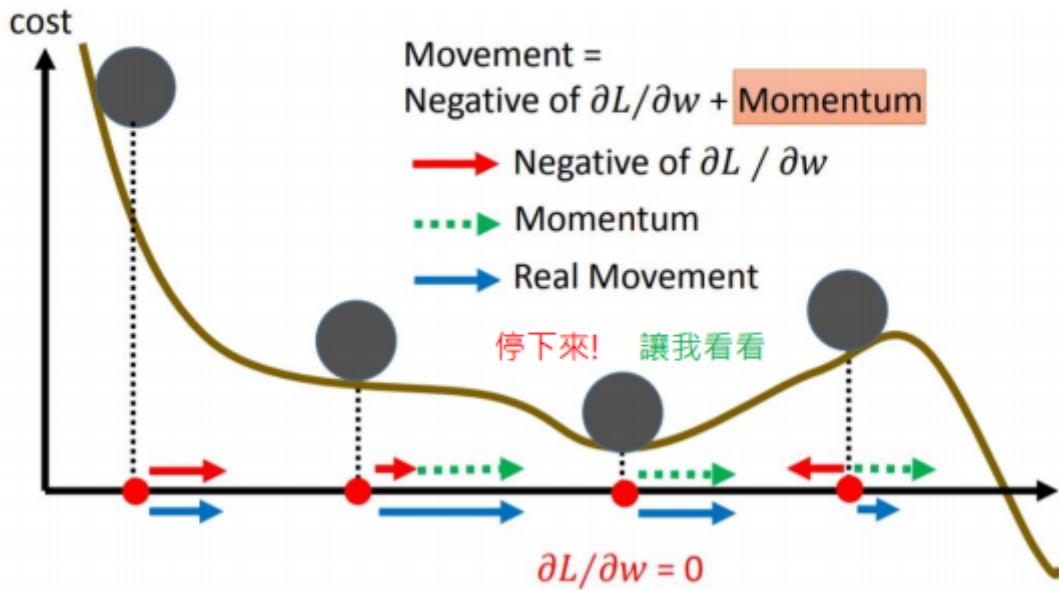


Start at point  $\theta^0$   
Movement  $v^0=0$   
Compute gradient at  $\theta^0$   
Movement  $v^1 = \lambda v^0 - \eta \nabla L(\theta^0)$   
Move to  $\theta^1 = \theta^0 + v^1$   
Compute gradient at  $\theta^1$   
Movement  $v^2 = \lambda v^1 - \eta \nabla L(\theta^1)$   
Move to  $\theta^2 = \theta^1 + v^2$   
Movement not just based on gradient, but previous movement.

Credit to 李宏毅老師上課投影片

## Why momentum?

为什么要加入动量? 跳出一些局部最小值, 继续往前看看



Credit to 李宏毅老師上課投影片

我们可以用来加速它的第一个东西是这个叫动量的东西。还是原来那个电子表格文件。我去掉了有限差分的部分，它们用处不是很大，只留下数学分析的部分。`de/db` 是存放导数的，要用导数来更新。

	<i>fx</i>	$=J8*\$J\$1+\$K\$1*F9$												
1		A	B	C	D	E	F	G	H	I	J	K	L	
2		b		1.0000	learn					beta	0.9	0.1		
3	x	y	b	a	pred	de/db	de/da	new b	new a	-18.33	98.25	err^2		
4	14	58	1.0000	1.0000	15.0000	-86.00	-1204.00	1.00	1.00	-25.1	-31.98	1849		
5	86	202	1.0025	1.0032	87.2775	-229.44	-19732.27	1.01	1.20	-45.53	-2002	13161.25		
6	28	86	1.0071	1.2034	34.7022	-102.60	-2872.68	1.01	1.41	-51.24	-2089	2631.462		
7	51	132	1.0122	1.4123	73.0398	-117.92	-6013.94	1.02	1.66	-57.91	-2482	3476.306		
8	28	86	1.0180	1.6605	47.5109	-76.98	-2155.39	1.02	1.91	-59.82	-2449	1481.409		
9	29	88	1.0240	1.9054	56.2793	-63.44	-1839.80	=J8*\\$J\\$1+\\$K\\$1*F9		-2388	1006.203			
10	72	174	1.0300	2.1442	155.4095	-37.18	-2677.04	1.04	2.39	-57.88	-2417	345.6083		
11	62	154	1.0358	2.3859	148.9586	-10.08	-625.13	1.04	2.61	-53.1	-2238	25.41554		
12	84	198	1.0411	2.6096	220.2498	44.50	3737.96	1.05	2.77	-43.34	-1640	495.0529		
13	15	60	1.0454	2.7736	42.6501	-34.70	-520.50	1.05	2.93	-42.48	-1528	301.0194		
14	42	114	1.0497	2.9265	123.9612	19.92	836.74	1.05	3.06	-36.24	-1292	99.22647		
15	62	154	1.0533	3.0556	190.5028	73.01	4526.35	1.06	3.13	-25.31	-709.9	1332.457		
16	47	124	1.0558	3.1266	148.0074	48.01	2256.69	1.06	3.17	-17.98	-413.2	576.3537		
17	35	100	1.0576	3.1680	111.9360	23.87	835.52	1.06	3.20	-13.79	-288.4	142.4678		
18	9	48	1.0590	3.1968	29.8301	-36.34	-327.06	1.06	3.23	-16.05	-292.2	330.145		
19	38	106	1.0606	3.2260	123.6492	35.30	1341.34	1.06	3.24	-10.91	-128.9	311.4928		
20	44	118	1.0617	3.2389	143.5734	51.15	2250.46	1.06	3.23	-4.708	109.1	654		
21	99	228	1.0622	3.2280	320.6340	185.27	18341.52	1.06	3.03	14.29	1932	8581.05		
22	13	56	1.0607	3.0348	40.5127	-30.97	-402.67	1.06	2.86	9.763	1699	239.8561		
23	21	72	1.0598	2.8649	61.2224	-21.56	-452.66	1.06	2.72	6.632	1484	116.1567		
24	28	86	1.0591	2.7165	77.1217	-17.76	-497.18	1.06	2.59	4.193	1286	78.824		
25	20	70	1.0587	2.5880	52.8180	-34.36	-687.28	1.06	2.48	0.337	1088	295.2221		
26	8	46	1.0586	2.4791	20.8917	-50.22	-401.73	1.06	2.39	-4.718	939.3	630.4254		
27	64	158	1.0591	2.3852	153.7124	-8.58	-548.81	1.06	2.31	-5.104	790.5	18.38337		
28	99	228	1.0596	2.3062	229.3695	2.74	271.16	1.06	2.23	-4.32	738.5	1.875537		
29	70	170	1.0600	2.2323	157.3215	-25.36	-1774.99	1.06	2.18	-6.423	487.2	160.745		
30	27	84	1.0607	2.1836	60.0175	-47.96	-1295.05	1.06	2.15	-10.58	309	575.1583		
31	17	64	1.0617	2.1527	37.6575	-52.69	-895.65	1.06	2.13	-14.79	188.5	693.928		
32	8	46	1.0632	2.1338	18.1339	-55.73	-445.86	1.07	2.12	-18.88	125.1	776.5169		
33										-18.33	98.25	200.9652		

但我做的是用导数乘以0.1，用上一轮的更新结果乘以0.9，把这两个加起来。换句话说，我做的更新不是基于导数，而是导数的1/10和上一次方向的90%。这叫做动量。想想在寻找最小值时发生了什么。

你在这里，你的学习率太小，你还是保持原来的步子（step）。如果你保持原来的步子，如果你还是在上次的结果上添加，你的步子会越来越大，最终，它们会走得太远。现在，你的梯度指向动量指着的另外一个方向。这样你可以在这里用很小的步子，然后用小的步子，大一点的步子，更大一点的步子，小步子，更大的步子，就像这样。这就是动量做的。

如果你像这样走得太远，再慢下来，，你最后几步的平均值是这两个之间的地方，不是吗？所以这是一个很普遍的想法：你在第 $T$ 次的这一步等于某个数（人们经常用alpha）乘以我要做的东西（这里是梯度）加上一减去alpha乘以你上一次的值( $S_{t-1}$ )：

$$S_t = \alpha \cdot g + (1 - \alpha) S_{t-1}$$

这叫做指数加权移动平均数（exponentially weighted moving average）。为什么这样叫，你可以想下，这个 $(1 - \alpha)$ 要乘上去。如果是 $S_{t-2}$ ，它会有 $(1 - \alpha)^2$ ，如果是 $S_{t-3}$ ，它会有 $(1 - \alpha)^3$ 。

换句话说， $S_t$ 最终是想让 $(\alpha \cdot g)$ 加上最近几次的加权平均值，最近的值权重更高。这样不断上升。这就是动量。它就是基于目前的梯度加上最近几步的指数加权移动平均值，这很有用。这叫做包含了动量的SGD，我们通过改变这个来实现它：

```
1 | opt = optim.Adam(model.parameters(), lr)
```

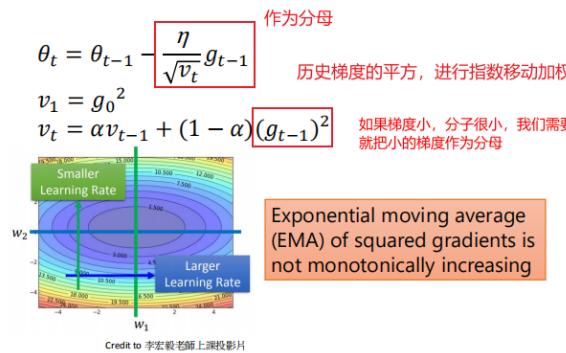
到

```
1 | opt = optim.SGD(model.parameters(), lr, momentum=0.9)
```

动量值用0.9很普遍。对基本的问题来说，它大概都是用0.9。这就是包含了动量的SGD。还是一样，这并不是简化的版本，是全部内容。这是SGD。还是一样，你可以自己写。试着写出来。在lesson 2 SGD上添加动量是一个很好的作业。或者在这个做MNIST的notebook里不再用`optim.`，自己写一个包含动量的update函数。

## RMSProp [1:53:30]

### RMSProp



### Adam

• SGDM

$$\theta_t = \theta_{t-1} - \eta m_t$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_{t-1}$$

• RMSProp

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{v_t + \epsilon}} \hat{m}_t$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\beta_1 = 0.9$$

$$\beta_2 = 0.999$$

$$\epsilon = 10^{-8}$$

Adam的公式

de-biasing

### What you have known before?

- SGD [Cauchy, 1847]
- SGD with momentum [Rumelhart, et al., Nature'86]
  - Adaptive learning rate
- Adagrad [Duchi, et al., JMLR'11]
- RMSProp [Hinton, et al., Lecture slides, 2013]
- Adam [Kingma, et al., ICLR'15]

### Optimizers: Real Application



有一个很酷的东西叫RMSProp。关于RMSProp，最酷的事情是Geoffrey Hinton提出了它，所有人都使用它。它特别流行，特别普遍。RMSProp的准确出处是Coursera在线MOOC，在那里他第一次提出了RMSProp，我喜欢这种在MOOC上出现而不是在论文里出现的新方法。

										J	K
1	b	3.1891	learn						0.9	0.1	
2	a	2.4036	0.02								
3	x	y	b	a	pred	de/db	de/da	new b	new a	1134.42	1679911.369
4	14	58	3.1891	2.4036	36.8398	-42.32	-592.49	3.21	2.41	1200.08	1547024.308
5	86	202	3.2142	2.4128	210.7118	17.42	1498.43	3.20	2.39	1110.43	1616851.221
6	28	86	3.2041	2.3887	70.0869	-31.83	-891.14	3.22	2.40	1100.68	1534578.48
7	51	132	3.2232	2.4027	125.7602	-12.48	-636.46	3.23	2.41	1006.19	1421629.139
8	28	86	3.2308	2.4130	70.7937	-30.41	-851.56	=J7*\$J\$1+\$K\$1*F8^2			1351980.855
9	29	88	3.2499	2.4272	73.6400	-28.72	-832.88	3.27	2.44	980.74	1286151.404
10	72	174	3.2681	2.4416	179.0612	10.12	728.81	3.26	2.43	892.912	1210653.026
11	62	154	3.2617	2.4287	153.8422	-0.32	-19.57	3.26	2.43	803.631	1089626.035
12	84	198	3.2619	2.4291	207.3040	18.61	1563.08	3.25	2.40	757.894	1224984.744
13	15	60	3.2487	2.3991	39.2356	-41.53	-622.93	3.28	2.41	854.568	1141290.635
14	42	114	3.2789	2.4104	104.5149	-18.97	-796.75	3.29	2.43	805.098	1090641.833
15	62	154	3.2919	2.4253	153.6603	-0.68	-42.12	3.29	2.43	724.634	981755.038
16	47	124	3.2924	2.4261	117.3193	-13.36	-627.99	3.30	2.44	670.024	923016.5515
17	35	100	3.3023	2.4388	88.6596	-22.68	-793.83	3.32	2.46	654.463	893731.181
18	9	48	3.3198	2.4553	25.4176	-45.16	-406.48	3.36	2.46	793.003	820880.9685
19	38	106	3.3551	2.4639	96.9835	-18.03	-685.25	3.37	2.48	746.222	785749.9911
20	44	118	3.3679	2.4790	112.4453	-11.11	-488.81	3.38	2.49	683.941	731068.6545
21	99	228	3.3761	2.4901	249.8920	43.78	4334.63	3.34	2.39	807.252	2536859.803
22	13	56	3.3426	2.3887	34.3953	-43.21	-561.72	3.37	2.40	913.232	2314727.072
23	21	72	3.3730	2.3957	53.6832	-36.63	-769.31	3.40	2.41	956.112	2142437.695
24	28	86	3.3972	2.4058	70.7606	-30.48	-853.40	3.42	2.42	953.396	2001023.888
25	20	70	3.4170	2.4175	51.7669	-36.47	-729.32	3.44	2.43	991.035	1854112.966
26	8	46	3.4406	2.4278	22.8630	-46.27	-370.19	3.47	2.43	1106.06	1682405.833
27	64	158	3.4700	2.4332	159.1977	2.40	153.30	3.47	2.43	996.027	1516515.337
28	99	228	3.4685	2.4309	244.1258	32.25	3192.90	3.45	2.38	1000.44	2384327.076
29	70	170	3.4481	2.3790	169.9799	-0.04	-2.81	3.45	2.38	900.397	2145895.159
30	27	84	3.4481	2.3791	67.6828	-32.63	-881.13	3.47	2.39	916.857	2008944.353
31	17	64	3.4699	2.3911	44.1184	-39.76	-675.97	3.50	2.40	983.282	1853743.844
32	8	46	3.4961	2.4006	22.7012	-46.60	-372.78	3.53	2.41	1102.09	1682266.029
33										1134.42	1679911.369

RMSProp和动量很像，但是这次，我们的指数加权移动平均数不是关于梯度更新值的，而是关于梯度的平方的，它是梯度的平方。梯度平方乘以0.1加上前一个值乘以0.9。这是梯度平方的指数加权移动平均值。这个值代表什么呢？如果我的梯度很小，并且一直很小，它会是一个很小的数。如果我的梯度变化很大，它会是一个很大的数。如果梯度一直很多，它也是一个很大的数。

这有什么意义呢？因为当我们做更新时，我们让权重减去学习率乘以梯度除以梯度的平方。

$$weight - \frac{lr \cdot g}{x^2}$$

换句话说，如果梯度一直很小，变化不大，那就让步幅大些。这是我们想要的，是吗？当我们观察到截距移动地这么慢时，很明显，你需要试着加快些。

const											
1	Reset	b	3.1891	learn						0.9	0.1
2	Run	a	2.4036	0.02							
3	x	y	b	a	pred	de/db	de/da	new b	new a	1134.424	1679911.369
23	21	72	3.3730	2.3957	53.6832	-36.63	-769.31	3.40	2.41	956.1117	2142437.695
24	28	86	3.3972	2.4058	70.7606	-30.48	-853.40	3.42	2.42	953.3959	2001023.888
25	20	70	3.4170	2.4175	51.7669	-36.47	-729.32	3.44	2.43	991.035	1854112.966
26	8	46	3.4406	2.4278	22.8630	-46.27	-370.19	3.47	2.43	1106.059	1682405.833
27	64	158	3.4700	2.4332	159.1977	2.40	153.30	3.47	2.43	996.0269	1516515.337
28	99	228	3.4685	2.4309	244.1258	32.25	3192.90	3.45	2.38	1000.44	2384327.076
29	70	170	3.4481	2.3790	169.9799	-0.04	-2.81	3.45	2.38	900.3966	2145895.159
30	27	84	3.4481	2.3791	67.6828	-32.63	-881.13	3.47	2.39	916.8572	2008944.353
31	17	64	3.4699	2.3911	44.1184	-39.76	-675.97	3.50	2.40	983.282	1853743.844
32	8	46	3.4961	2.4006	22.7012	-46.60	-372.78	3.53	2.41	1102.088	1682266.029
33										1134.42	1679911.369

如果你现在运行这个，在5个epoch后，它已经达到3了。用基本的版本，在经过5的epoch时，它还是1.27。记住，我们需要达到30。

## Adam [1:55:44]

显然，要同时做这两件事。这就叫**Adam**。Adam就是简单地追踪梯度平方的指数加权移动平均值（RMSProp），也追踪每步的指数加权移动平均值（动量）。这两个都除以一个变量的平方的指数加权移动平均值，都用上一次相同方向的一步的0.9。这个变量是动量和RMSProp，它们被叫做Adam。看这个，5步，我们得到了25。

人们把这样的优化叫做动态学习率。很多人误解了它，说并没有设置学习率。你当然设置了。就像你需要移动地快一些，或者持续按同一个方向移动来尝试找到参数。这不代表你不需要学习率。我们还需要学习率。事实上，如果你再运行一遍这个，它会更好，但最终会在某个地方周围停下。你可以看到学习率太高了。我们可以把它降下来，多运行一下。现在很接近了，是吗？

你可以看到，即使有了Adam，你还是需要学习率退火。尝试一个这个电子表格很有趣。我有一个basic SGD的[Google sheets version](#)，它可以运行，宏也可以运行。Google sheet太糟糕了，我被这个折磨疯了，所以我决定不再处理其他的表格了。我会分享一个Google sheet版本的链接。它确实有一个宏语言，但它很荒唐。无论如何，如果有人想挑战它，让所有这些表格能运行，是可以做到的。只是会让人很痛苦。可能有人可以让这些在Google sheet运行起来。

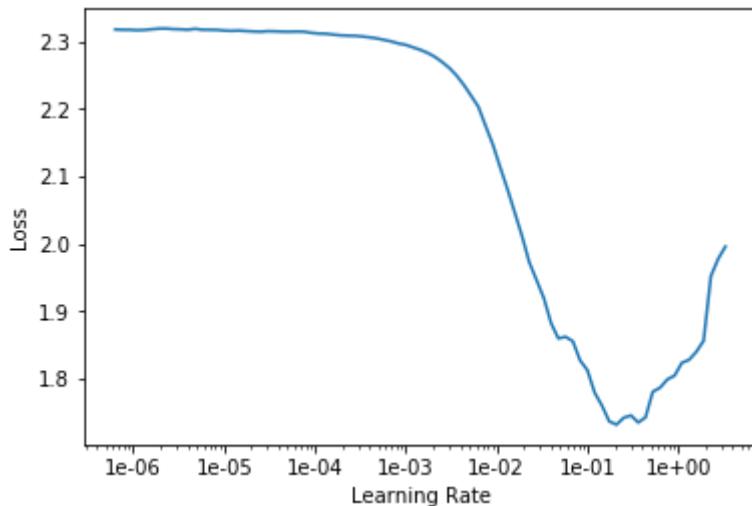
[1:58:37]

这是权重衰减和Adam。Adam令人惊奇地快。

```
1 | learn = Learner(data, Mnist_NN(), loss_func=loss_func, metrics=accuracy)
```

我们不想使用`optim.`，会自己创建optimizer和所有这些东西。因为我们要使用learner，learner就是做这些东西的。还是一样，没有魔法。你创建一个learner，这是data bunch，这是PyTorch `nn.Module` 实例，这是损失函数，这是度量（metrics）。记住，度量只是用来打印的。就是这样。然后你运行`learn.lr_find`，它记录下这个：

```
1 | learn.lr_find()
2 | learn.recorder.plot()
```



你可以用 `fit_one_cycle` 替代 `fit`。这很有帮助。

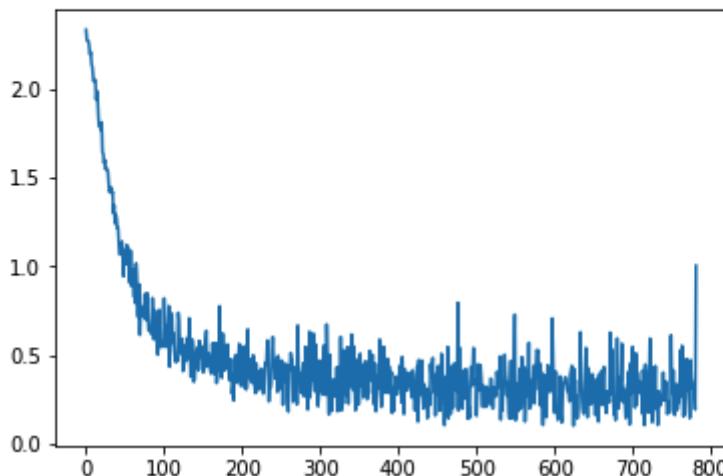
```
1 | learn.fit_one_cycle(1, 1e-2)
```

```
1 | Total time: 00:03
2 | epoch  train_loss  valid_loss  accuracy
3 | 1      0.148536    0.135789    0.960800  (00:03)
```

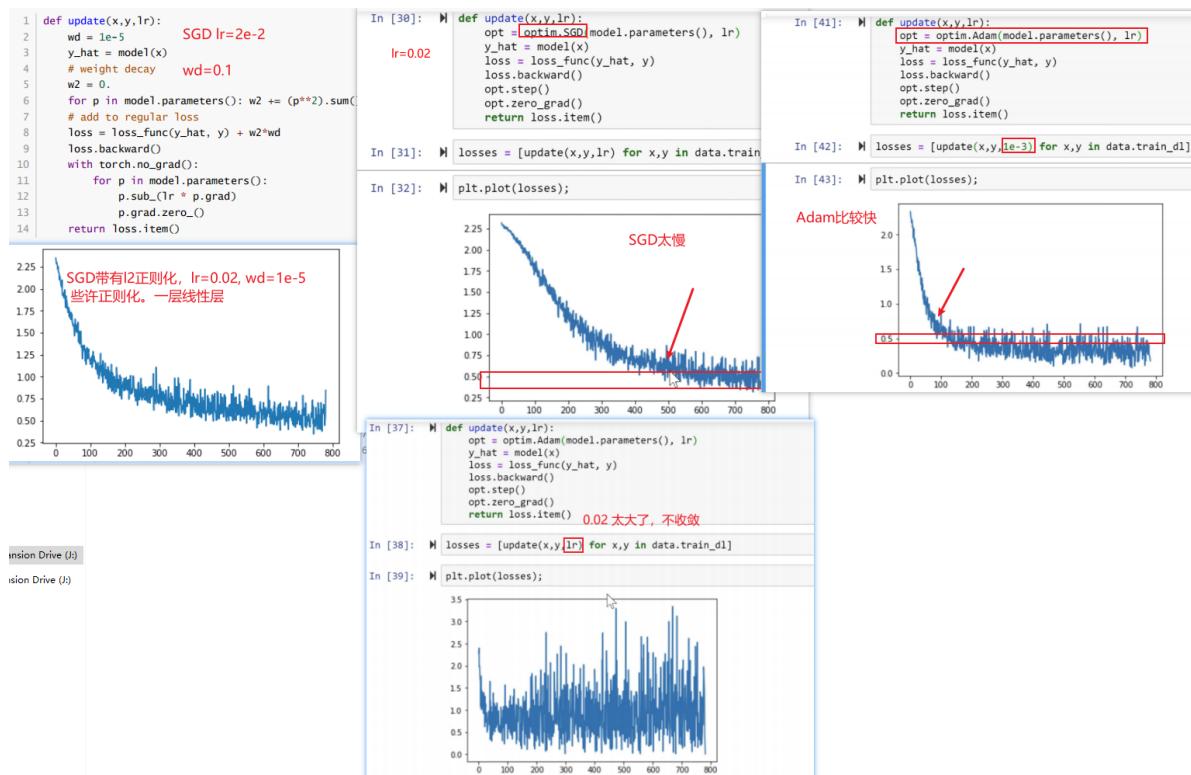
通过使用学习率查找器，我找到了一个好的学习率。看这里，我的损失度是0.13。这里没有比0.5低很多：

```
losses = [update(x,y,1e-3) for x,y in data.train_dl]
```

```
plt.plot(losses);
```



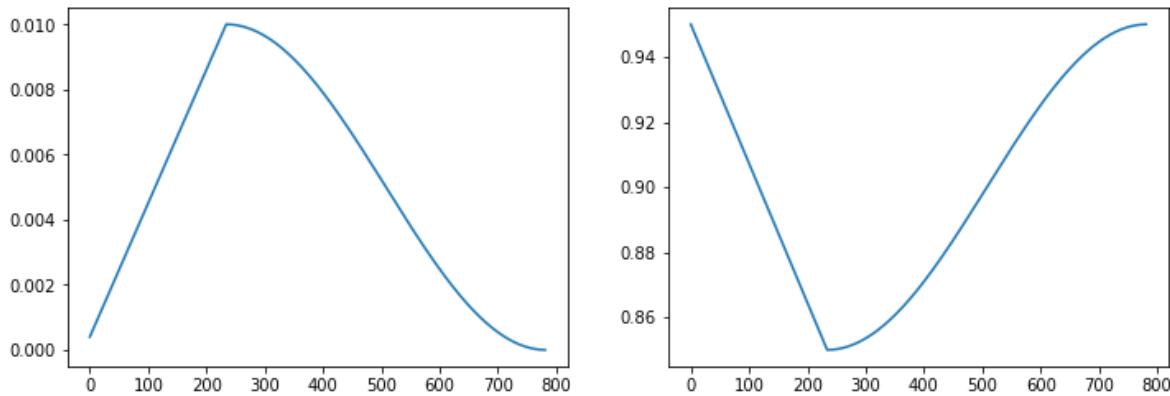
这些技巧产生了很多的差别，不是微小的差别。这只是一个epoch。



## Fit one cycle [2:00:02]

这个fit\_one\_cycle是做什么的呢？它究竟做了什么？这是它做的：

```
1 | learn.recorder.plot_lr(show_moms=True)
```

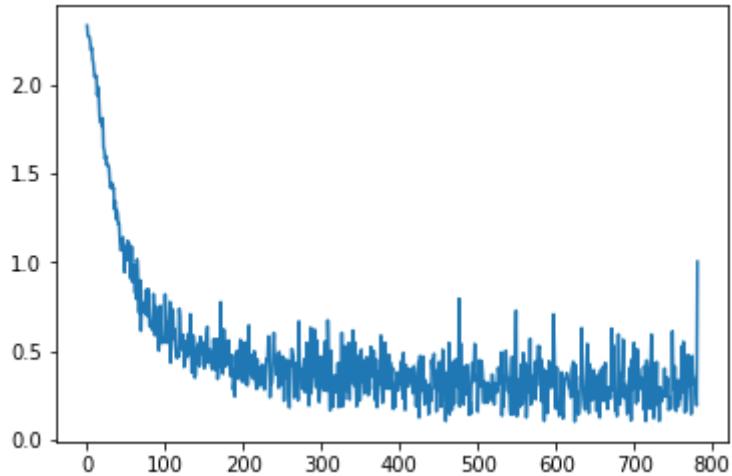


我们之前看过左边这个图。提醒下，这是在画每个batch的学习率。记住，Adam有学习率，我们默认使用Adam（或者变种，我们可能会讲）。学习率开始时很低，它在前半部分增长，然后在后半部分下降。因为在最开始时，我们不知道我们在哪。我们在函数空间的某个地方，它很曲折。如果你开始时跳得远，这些凸起有很大的梯度，这会把你抛到空间里疯狂的位置。所以开始时慢一些，你可以逐步移动到权重空间的合理位置。因为你到了合理的位置，你可以加大学习率，因为梯度在你想要的方向上。就像我们讲过几次的，当你接近最终答案时，你需要给学习率退火，来打磨它。

这是有趣的事情，在右边是动量的图形。当学习率比较小时，动量比较高。为什么？因为我有一个小学习率，但你还在沿着相同的方向走，你可能走得更快。但是，如果你跳很远，不要跳很远，因为那会把你抛出去。当你到了最后，你在微调，但实际上如果你持续在相同的方向走，会更快。所以这个组合叫one cycle，它很简单，但效果令人惊讶。这会让训练快上10倍，非常令人难以置信。

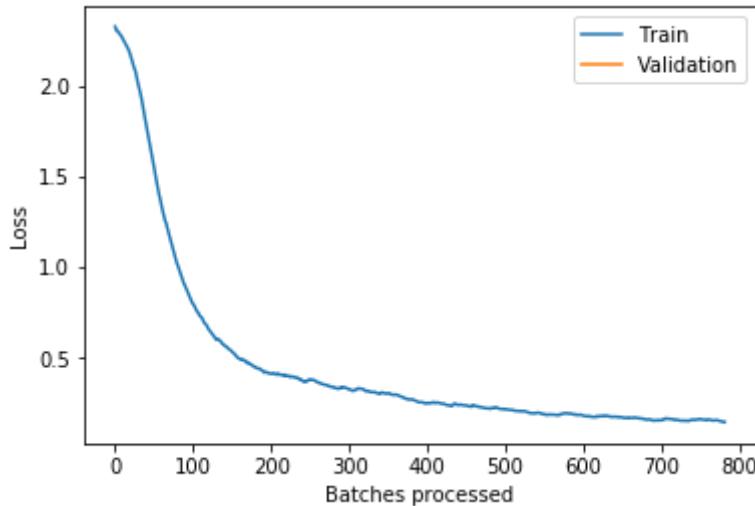
这是去年才发的论文。你们当中的一些人可能看过我上周对Leslie Smith的采访。他是一个令人惊奇的人，难以置信的谦虚，做了开创性研究，已经60多岁了，这些都令人振奋。

我还会演示一些其他的事情。当你用fastai画损失度时，它看起来不是这样的：



它是这样：

```
1 | learn.recorder.plot_losses()
```



为什么？因为fastai为你计算了损失度的指数加权移动平均数。这个指数加权很方便，我经常使用它。它可以让图形更容易查看。这意味着用fastai画出的图可能会比正常的延后一两个batch。当使用指数加权平均时，有一点不好的地方，就是你会受到一点历史的影响。但它可以让你更容易看到程序是怎么运行的。

## Back to Tabular [2:03:15]

[Notebook](#)

现在我们就要完成协同过滤和表格数据的部分了。我们要理解表格数据模型里的所有代码。这个表格数据模型使用的是adult这个数据集，它要预测谁会挣更多钱。这是一个分类问题，我们有一些类别变量和一些连续变量。

```
1 | from fastai.tabular import *
```

```
1 | path = untar_data(URLs.ADULT_SAMPLE)
2 | df = pd.read_csv(path/'adult.csv')
```

```

1 dep_var = '>=50k'
2 cat_names = ['workclass', 'education', 'marital-status', 'occupation',
   'relationship', 'race']
3 cont_names = ['age', 'fnlwgt', 'education-num']
4 procs = [FillMissing, Categorify, Normalize]

```

我们还不知道怎样预测一个类别变量。目前为止，我们只知道我们是用了 `nn.CrossEntropyLoss` 这个损失函数。这是什么？让我们找出答案。我们看看这个 [Microsoft Excel](#) 来找出答案。

Cross-entropy (交叉熵) 损失函数只是另一个损失函数。你们已经知道了一个损失函数，均方差 (mean squared error)  $(\hat{y} - y)^2$ 。对于MNIST，这不是一个好的损失函数，我们有10个可能的数字，和取到这10个数字的概率。所以我们需要这样的东西：如果预测正确，并且很有信心，那么损失度就很低；如果预测错了，又很有信心，那么损失度就比较高。

这是一个例子：

B	C	D	E	F	G
Cat	Dog	Pred(Cat)	Pred(Dog)	X-Entropy	
1	0	0.5	0.5	0.30	0.30
1	0	0.98	0.02	0.01	0.01
0	1	0.9	0.1	1.00	1.00
0	1	0.5	0.5	0.30	0.30
1	0	0.9	0.1	0.05	0.05
				1.66	

前两列是猫和狗的one hot编码。这两列是我的模型的给出的激活值，是猫的概率和是狗的概率。第一行对结果不确定。第二行很确信这是一只猫，这个答案是对的。第三行很确信这是一只猫，但是错了。我们想给第一行一个中等的损失，因为它没有自信地做出预测不是我们想要的。损失度是0.3。第二行自信地预测了正确的答案，所以损失度是0.01。第三行自信地预测了错误的答案，所以损失度是1.0。  
。

我们是怎样做的？这是交叉熵损失：

= -B2*LOG(D2)-C2*LOG(E2)					
B	C	D	E	F	G
Cat	Dog	Pred(Cat)	Pred(Dog)	X-Entropy	
1	0	0.5	0.5	= -B2*LOG(D2)-C2*LOG(E2)	
1	0	0.98	0.02	0.01	0.01
0	1	0.9	0.1	1.00	1.00
0	1	0.5	0.5	0.30	0.30
1	0	0.9	0.1	0.05	0.05
				1.66	

它等于：是否是猫 (B2) 乘以猫的激活值 (D2) 的log值 (底数默认是10)，取负数，减去是否是狗 (C2) 乘以狗的激活值 (E2) 的log值。就是这样。换句话说，它是所有one-hot编码变量乘以激活值的和。

B	C	D	E	F	G	H	I
Cat	Dog	Pred(Cat)	Pred(Dog)	X-Entropy			
1	0	0.5	0.5	0.30	=-IF(B2=1,LOG(D2),LOG(1-D2))		
1	0	0.98	0.02	0.01	0.01		
0	1	0.9	0.1	1.00	1.00		
0	1	0.5	0.5	0.30	0.30		
1	0	0.9	0.1	0.05	0.05		
				1.66			

有趣的是，这一列（G列）的数，和F列的数完全一样，但是我输入的公式是不同的。我用了一个if函数，因为如果是0的话，不用加任何值。所以实际上，它是指，如果这是一只猫，就取猫激活值（有多少信心这是猫）的log，如果是一只狗，就取1减去猫的激活值（也就是狗的激活值）的log。所以，one-hot编码乘以激活值的和相当于一个if函数。这只是一个矩阵乘法，它和索引查找是一样的（就像我们在embedding里讲过的）。所以，要做交叉熵，你也可以直接找预测值的激活值的log。

这只在这些值加起来等于1时有效。这是你得到荒唐的交叉熵值的一个原因：如果它们加起来不等于1，就会出问题（这就是为什么我说你按错了按钮）。那怎样保证它们加起来等于1呢？你要通过在你的最后一层使用正确的激活函数来保证它们加起来等于1。这里要用的正确的激活函数是softmax。softmax是这样一个激活函数：

- 所有激活值加起来等于1
- 所有激活值都大于0
- 所有激活值都小于1

这正是我们想要的。这就是我们需要的。怎样做到呢？比如说我们在五个物品之间做预测：猫、狗、飞机、鱼、建筑。这些是我们的神经网络给出的一组预测值（output）。

我们做以 $e$ 为底的乘方，右边一列（C列）是乘方结果（幂）。因为 $e$ 的任何次方都是大于0的，所以得到的这一组数总是大于0的。这是这些幂的和（12.14）。这（D2）是这个幂（C2）除以这些幂（C列）的和（C7）：

SUM	A	B	C	D
1		output	exp	softmax
2	cat	-2.17	0.11	=C2/\$C\$7
3	dog	-1.63	0.20	0.02
4	plane	-2.75	0.06	0.01
5	fish	2.39	10.91	0.90
6	building	-0.16	0.86	0.07
7			12.14	1.00

现在这个数总是会小于1，因为所有数都是正的，所以不会有任何一个数大于它们的和。所有这些数加起来等于1，因为每个数就是占所有数的百分比。就是这样。softmax等于以 $e$ 做底，以激活值做指数的幂值除以所有幂值的商。这被叫做softmax。

我们在做单标签的多类别分类时，你一般会想用softmax做激活函数，用交叉熵做损失函数。这两个很密切，PyTorch为你实现了它们。你可能注意到了在MNIST的例子里，我没有添加softmax：

```

class Mnist_Logistic(nn.Module):
    def __init__(self):
        super().__init__()
        self.lin = nn.Linear(784, 10, bias=True)

    def forward(self, xb): return self.lin(xb)

```

这是因为如果你调用交叉熵损失 (`nn.CrossEntropyLoss`)，它会在损失函数里做softmax。所以它不仅是交叉熵损失，它实际上是先做softmax，再做交叉熵损失。

你可能已经注意到这个了。但有时你的模型的预测值看起来会是这样：

	A	B	C	D
1		output	exp	softmax
2	cat	-2.17	0.11	0.01
3	dog	-1.63	0.20	0.02
4	plane	-2.75	0.06	0.01
5	fish	2.39	10.91	0.90
6	building	+ -0.16	0.86	0.07
7			12.14	1.00

数值很大，还有负数。并不是0到1之间的数，和等于1。出现这种情况的原因可能是，它是一个没有softmax的PyTorch模型，因为我们用了交叉熵，你需要手动做softmax。

Fastai越来越擅长判断什么时候会出现这种情况。一般，当你做预测时，如果我们发现你使用了损失函数，我们会尝试为你添加softmax。但是，如果你使用了在一些处理之后调用 `nn.CrossEntropyLoss` 的自定义损失函数，你可能会遇到这种情况。

我们还有3分钟，我还要讲一个问题。下周，我们会在最初10分钟，结束表格数据部分，这个是表格模块里的 `forward`：

```

def forward(self, x_cat:Tensor, x_cont:Tensor) -> Tensor:
    if self.n_emb != 0:
        x = [e(x_cat[:,i]) for i,e in enumerate(self embeds)]
        x = torch.cat(x, 1)
        x = self.emb_drop(x)
    if self.n_cont != 0:
        x_cont = self.bn_cont(x_cont)
        x = torch.cat([x, x_cont], 1) if self.n_emb != 0 else x_cont
    x = self.layers(x)
    if self.y_range is not None:
        x = (self.y_range[1]-self.y_range[0]) * torch.sigmoid(x) + self.y_range[0]
    return x

```

它遍历了一组embedding。它会调用每一个embedding `e`，你可以像用函数一样使用它。它会把每个类别变量传入到embedding里，把它们连在一起，放进一个矩阵。然后它会调用一组层，它们是线性层。然后会做sigmoid。这里只有两个要学的新东西。一个是dropout，另一个是batch norm (`bn_cont`)。这是两个额外的正则化策略 (regularization strategies)。BatchNorm做的不只是正则化。做正则化的基本方式是权重衰减、batch norm和dropout。你也可以用数据增强 (data augmentation) 来避免过拟合。下周课程开始的时候，我们会接触batch norm和dropout。我们也会学数据增强。然后，我们会学习什么是卷积。我们要学习一些新的计算机视觉架构和一些新的计算机视觉应用。但基本上，我们差不多了。你们已经知道了整个 `collab.py` (`fastai.collab`) 是怎么运行的。你们知道了为什么会有它、它是怎样的，也快要知道整个表格模型做的事了。表格模型，如果你把它用在Rossmann上，你会得到和我给你看的论文里一样的结果。你可以在论文第二部分看到结果。事

实上，我们的结果会更好一点。下周，我会演示怎样做一些额外的实验，用一些小技巧，做出比论文更好一点的结果。非常感谢。