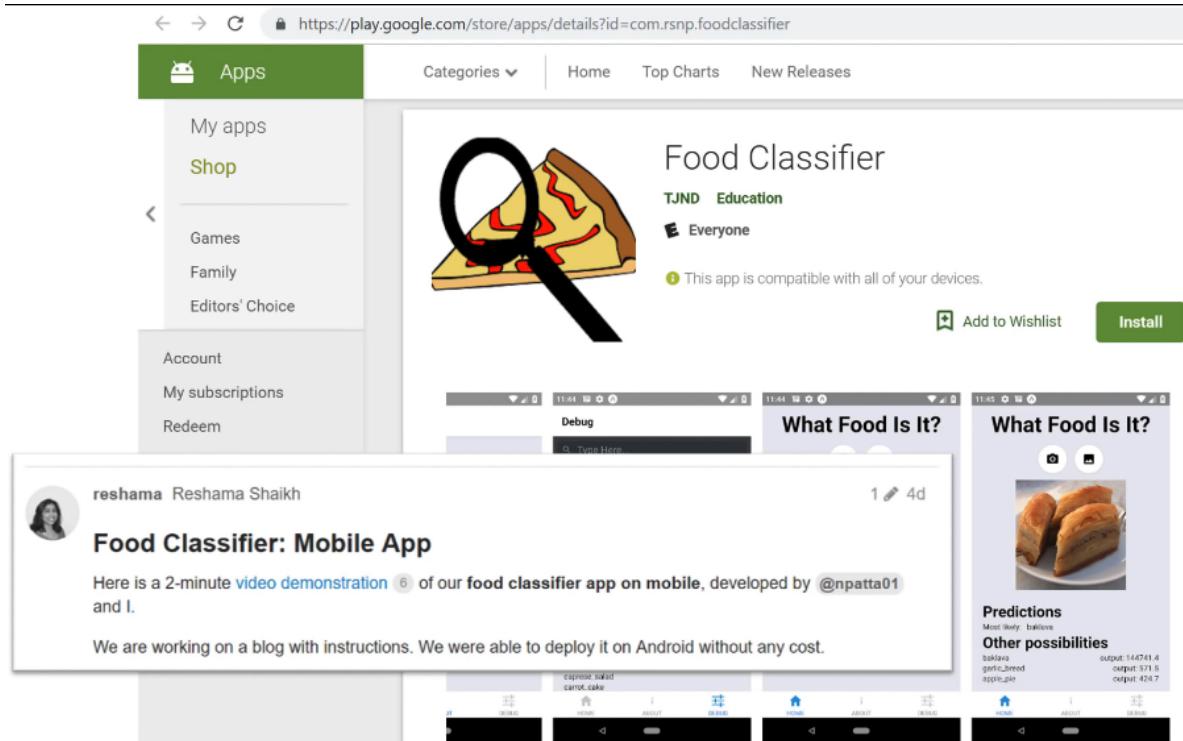


Lesson 7

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

欢迎来到第七课！这是课程第一部分的最后一课。这节课内容比较多。不要担心。这是因为，我想让你们在课程第二部分开始之前有足够的事情做。今天讲的一些东西，我不会讲太多细节。只会指出一些东西。我会说，我们今天不会讲它。然后，在课程第二部分，会学到这些内容的细节。今天会很快地学很多东西。你可能要多看几遍，做些实验，来完全理解。我们特意这样做，给你们一些东西，让你们在接下来一两个月里能过得充实有趣。



首先，我要展示一个很酷的东西，它是几个学生做的。Reshma和Nidhin开发了一个Android和iOS app，在这个[Reshma在论坛发的文章](#)

里可以看到，他们有一个怎样创建可以真正发布到Google Play Store和Apple App Stroe的Android和iOS app的demo，这很酷。这是我知的在App Store里的第一个使用fastai的东西。非常感谢Reshma，她为fast.ai社区和机器学习社区，和[女性机器学习社区](#)做出了贡献。她做了很多了不起的事情：提供了大量的文档和教程，组织社区等等。谢谢你，Reshma。祝贺这个app发布。

MNIST CNN [2:04]

可以看到，今天我们有很多notebook。第一个要学的notebook是[lesson7-resnet-mnist.ipynb](#)。我想看一些我们上周讲过的关于卷积和卷积神经网络的东西，利用它们从头做一些现代深度学习架构。我说的从头，不是重新实现所有东西，而是使用已有的PyTorch里的东西。我们使用MNIST数据集。URLS.MNIST有完整的MNIST数据集，我们经常只用它的一个子集。

```
1 %reload_ext autoreload  
2 %autoreload 2  
3 %matplotlib inline
```

```
1 from fastai.vision import *
```

```
1 | path = untar_data(URLs.MNIST)
```

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

```
1 | [PosixPath('/home/jhoward/.fastai/data/mnist_png/training'),
2 | PosixPath('/home/jhoward/.fastai/data/mnist_png/models'),
3 | PosixPath('/home/jhoward/.fastai/data/mnist_png/testing')]
```

这里有一个训练文件夹和一个测试文件夹。我把这个读进来后，我要演示一些data blocks API的细节，你看到过它们是怎样用的。通常，我们是这样 blah.blah.blah.blah.blah，在一个单元格里一起做，现在让我们在一个单元格里只做一个。

```
1 | il = ImageItemList.from_folder(path, convert_mode='L')
```

先说是什么类型的item list。这里是一个image的item list。然后是从哪里得到文件名list。这里，递归搜索一个文件夹。这是数据的来源。

你可以输入Pillow参数，Pillow是最终为我们打开这个的东西，这里它们是黑白的，不是RGB，所以你要用Pillow的 convert_mode='L'。在这个Python图形库文档里可以看到它的convert modes的更多细节。这一个参数代表是灰度的，MNIST就是这样的。

```
1 | il.items[0]
```

```
1 | PosixPath('/home/jhoward/.fastai/data/mnist_png/training/8/56315.png')
```

在item list里有一个 items 属性，items 属性里是你传入的数据。是用来创建条目要用的东西。这里，你传入的是一个文件名的list。这是它从这个目录里取到的。

```
1 | defaults.cmap='binary'
```

在你用show画图时，通常，它用RGB显示。这里，我们希望用binary color map。在fastai里，你可以设置默认的color map。参考matplotlib文档，可以查看更多关于cmp和color map的信息。

defaults.cmap='binary' 会设置fastai的默认color map。

```
1 | il
```

```
1 | ImageItemList (70000 items)
2 | [Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28),
3 | Image (1, 28, 28)]...
4 | Path: /home/jhoward/.fastai/data/mnist_png
```

我们的image item list包含70,000个条目，这是一堆1x28x28的图片。记住PyTorch把channel放在第一个维度。它们是一个通道的28x28。你可能会想，为什么不是只有28x28的矩阵，而是1x28x28的张量。因为这样更简单。所有 Conv2d 里的东西和其它的东西都是处理秩是3的张量的，所以你要用一个值是1的维度放在前面。当fastai读到只有一个通道的图片时，它会自动为你做这个。

```
1 | il[0].show()
```



这个`.items`属性存的是可以用来取图片的东西，这里它是文件名，如果你对item list直接做索引，你可以得到一个image对象。image对象有一个`show`方法，这里显示出图片。

```
1 | sd = il.split_by_folder(train='training', valid='testing')
```

你得到一个image item list后，你把它分成训练集和验证集。你通常需要验证集。如果你不需要，你可以用`.no_split`方法，创建一个空的验证集。你不能完全跳过这个方法。你需要告诉它怎么划分，其中一个操作是`no_split`。

记住，这是固定的顺序。首先创建item list，然后决定怎样划分。这个例子里，我们按照文件夹划分。MNIST的验证集文件夹是`testing`。在fastai里，我用和Kaggle一样的名称。训练集是你用来训练的东西，验证集（validation set）有标签，你用它来测试模型的效果，测试集（test set）没有标签，用它做预测，提交到竞赛，或者发给有标签的人让他们测试。所以，尽管你数据集里的文件夹叫`testing`，它不代表这是测试集（test set），它们有标签，所以这是验证集。

如果你想一次性预测很多数据，不是一次性预测一条数据，你要使用fastai里的`test=`，来表示这是没有标签的东西，我只用它们预测。

[6:54]

```
1 | sd
```

```
1 | ItemLists;
2 |
3 | Train: ImageItemList (60000 items)
4 | [Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28),
5 | Image (1, 28, 28)]...
6 |
7 | Valid: ImageItemList (10000 items)
8 | [Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28),
9 | Image (1, 28, 28)]...
10 |
11 | Test: None
```

可以看到，我划分后的数据是一个训练集和一个验证集。

```
1 | (path/'training').ls()
```

```
1 [PosixPath('/home/jhoward/.fastai/data/mnist_png/training/8'),
2  PosixPath('/home/jhoward/.fastai/data/mnist_png/training/5'),
3  PosixPath('/home/jhoward/.fastai/data/mnist_png/training/2'),
4  PosixPath('/home/jhoward/.fastai/data/mnist_png/training/3'),
5  PosixPath('/home/jhoward/.fastai/data/mnist_png/training/9'),
6  PosixPath('/home/jhoward/.fastai/data/mnist_png/training/6'),
7  PosixPath('/home/jhoward/.fastai/data/mnist_png/training/1'),
8  PosixPath('/home/jhoward/.fastai/data/mnist_png/training/4'),
9  PosixPath('/home/jhoward/.fastai/data/mnist_png/training/7'),
10 PosixPath('/home/jhoward/.fastai/data/mnist_png/training/0')]
```

在训练集里，每个类别都有一个文件夹。

```
1 | ll = sd.label_from_folder()
```

我们取这个划分的数据，执行 `label_from_folder`。

首先你创建一个item list，然后划分它，然后标记它。

```
1 | ll
```

```
1 LabelLists;
2
3 Train: LabelList
4 y: CategoryList (60000 items)
5 [Category 8, Category 8, Category 8, Category 8, Category 8]...
6 Path: /home/jhoward/.fastai/data/mnist_png
7 x: ImageItemList (60000 items)
8 [Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28),
9 Image (1, 28, 28)]...
10 Path: /home/jhoward/.fastai/data/mnist_png;
11
12 Valid: LabelList
13 y: CategoryList (10000 items)
14 [Category 8, Category 8, Category 8, Category 8, Category 8]...
15 Path: /home/jhoward/.fastai/data/mnist_png
16 x: ImageItemList (10000 items)
17 [Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28),
18 Image (1, 28, 28)]...
19 Path: /home/jhoward/.fastai/data/mnist_png;
20
21 Test: None
```

现在你可以看到我们有了 `x` 和 `y`，`y` 是类别对象。类别对象只是一个类。

```
1 | x,y = ll.train[0]
```

如果你对一个 `ll.train` 这样的label list做索引，你会得到一个自变量和因变量 (`x, y`)。这里，`x` 是可以用 `show` 画出来的图片对象，`y` 是可以打印出来的类别对象：

```
1 | x.show()
2 | print(y,x.shape)
```

```
1 | 8 torch.size([1, 28, 28])
```



这是数字8这个类别，图片是一个8。

[7:56]

```
1 | tfms = ([*rand_pad(padding=3, size=28, mode='zeros')], [])
```

下一个我们可以做的事情是添加变形 (transform)。这里，我们不会用普通的`get_transforms`函数，因为我们要做数据识别，对数字识别你不能做左右翻转，这会改变它的含义。你也不能旋转太多，这也会改变含义。因为这些图片很小，做放大会让它们变模糊，没法认出。通常，对这样的小的数字图片，你只能添加一些随机边框。所以我们使用random padding方法，它会返回两个transform，一个做边框，一个做随机裁剪。你需要用星号 (*) 来把这两个transform都放到list里。

```
1 | ll = ll.transform(tfms)
```

现在我们调用transform。这个空数组是存放验证集transform的。

```
In [16]: ┌─ tfms = ([*rand_pad(padding=3, size=28, mode='zeros')], [])
```

就是对验证集不做变形。

现在我们得到了一个变形过的有标签的list，我们设置一个批次大小，生成data bunch:

```
1 | bs = 128
```

```
1 | # not using imagenet_stats because not using pretrained model
2 | data = ll.databunch(bs=bs).normalize()
```

我们可以做标准化。这里，我们不用预训练模型，所以不需要用`imagenet_stats`。如果你这样调用`normalize`，不传入`stats`，它会随机取一批数据，用它来决定用什么标准化`stats`。如果你没有用预训练模型，这是一个好方法。

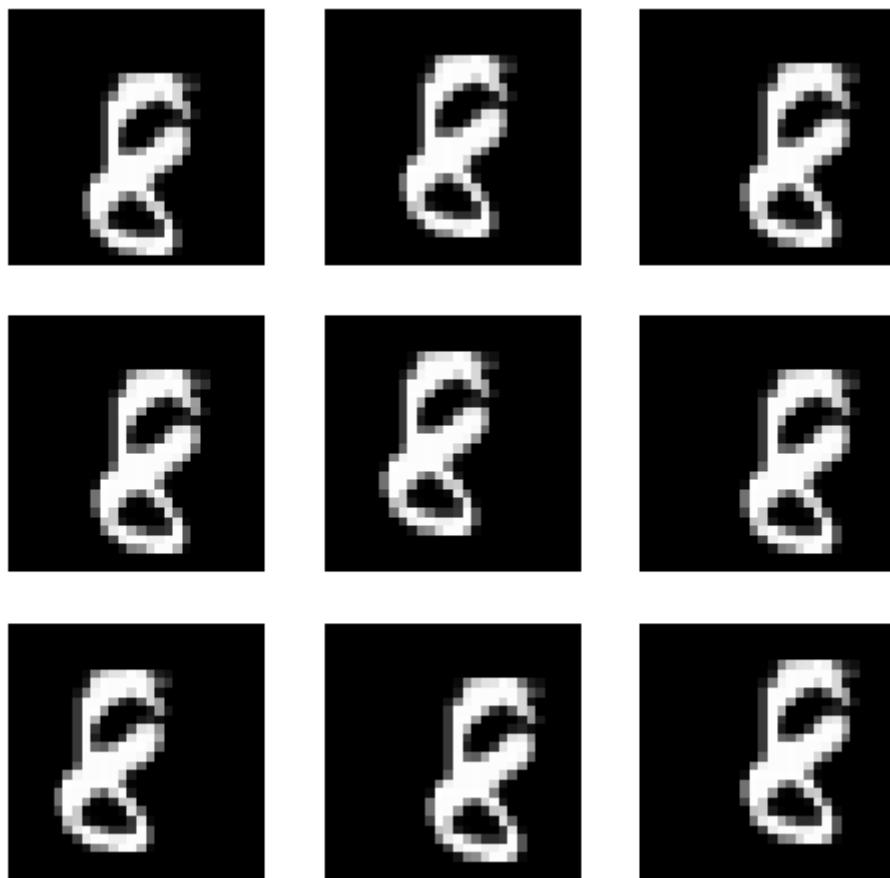
```
1 | x.show()
2 | print(y)
```

```
1 | 8
```



好了，我们得到了一个data bunch，在data bunch里有一个我们已经看过的data set。值得注意的是，现在这个训练集有了数据增强，因为我们做了变形。`plot_multi` 是一个fastai函数。它会画出为这个行和列里组成的网格里每一个元素调用这个函数的结果。这里，我的函数只是取出训练集的第一个图片，因为每次你从训练集里取出一些东西，它会从硬盘里加载它，实时对它做变形。人们有时问创建了多少图片变形的版本，答案是无限个。每次我们从数据集里取出一个东西，我们实时做随机变形，所以基本上每一个都是有点不同的。所以你可以看到，如果我们画很多次，8都在有点不同的位置，因为我们做了随机边框（random padding）。

```
1 | def _plot(i,j,ax): data.train_ds[0][0].show(ax, cmap='gray')
2 | plot_multi(_plot, 3, 3, figsize=(8,8))
```



[10:27]

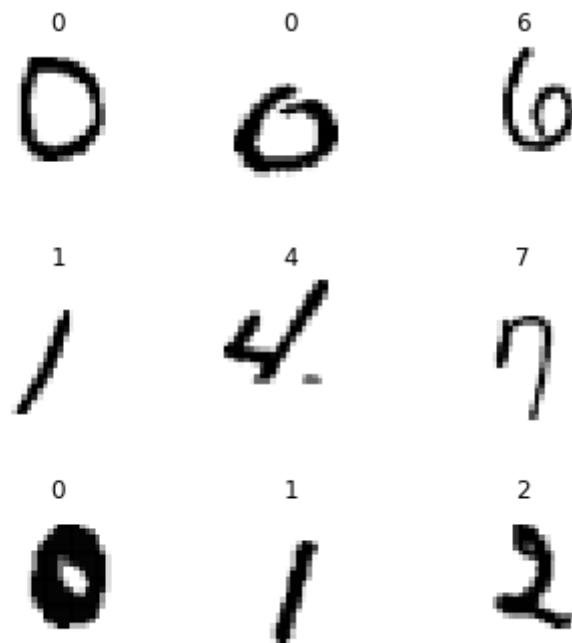
你可以从data bunch里取一批数据，记住，data bunch里有data loader，data loader可以一次取一批数据。你可以取一个X batch和一个Y batch，看看它们的形状，批次数量（batch size）x通道数x行x列：

```
1 | xb,yb = data.one_batch()  
2 | xb.shape,yb.shape
```

```
1 | (torch.size([128, 1, 28, 28]), torch.size([128]))
```

fastai里所有的data bunch都有show_batch方法，它会用合理的方式显示里面内容：

```
1 | data.show_batch(rows=3, figsize=(5,5))
```



我们快速过一遍用data block API取数据。

有batch norm 的基本CNN [11:01](#)

让我们开始创建一个简单的CNN。输出是 28×28 。创建架构时，我想定义一个函数，把我会做很多遍的东西放里面。我不会用相同的参数调用它，因为我会忘记或者出错。这里，所有的卷积的核的大小（kernel size）是3，步长（stride）是2，边框（padding）是1。让我们创建一个用这些参数做卷积的简单函数：

```
1 | def conv(ni,nf): return nn.Conv2d(ni, nf, kernel_size=3, stride=2, padding=1)
```

每次你做卷积时，它先跳过一个像素，每次跳两步。这意味着，每次完成一个卷积，它会把网格的尺寸减半。我在这里添加了一个注释，写出每次卷积后新网格的大小。

```
1 | model = nn.Sequential(  
2 |     conv(1, 8), # 14  
3 |     nn.BatchNorm2d(8),  
4 |     nn.ReLU(),  
5 |     conv(8, 16), # 7  
6 |     nn.BatchNorm2d(16),  
7 |     nn.ReLU(),  
8 |     conv(16, 32), # 4  
9 |     nn.BatchNorm2d(32),
```

```

10     nn.ReLU(),
11     conv(32, 16), # 2
12     nn.BatchNorm2d(16),
13     nn.ReLU(),
14     conv(16, 10), # 1
15     nn.BatchNorm2d(10),
16     Flatten()      # remove (1,1) grid
17 )

```

对第一次卷积，运行前我们有一个通道，因为它是只有一个通道的灰色图片，出来后会有多少个通道呢？你想要多少都可以。记住，你需要设定要创建多少个过滤器，不用管它是不是一个全连接层，如果是全连接层的话，它就是你要乘的矩阵的宽度，如果是2D卷积的话，它就是你要你需要的过滤器的数量。我选择8，它的步长是2，所以经过这一层后， 28×28 的图像现在是 14×14 的特征图，有8个通道。具体来说，它是一个 $8 \times 14 \times 14$ 的激活值张量。

然后我们做一个batch norm，然后我们做ReLU。下一个卷积层的输入的过滤器数量要和上一个卷积层的输出的卷积层的数量相等，我们可以不断增加通道数量，因为我们用的步长是2，它会不断减小网格的大小。注意，这里它从7变成4，因为如果你对大小是7的网格做一个步长是2的卷积，它会是 $7/2$ 的 `math.ceil`。

Batch norm，ReLU，conv。现在我们降到了 2×2 。Batch norm，ReLU，conv，现在我们降到了 1×1 。这个完成后，我们有了一个 $10 \times 1 \times 1$ 的特征图。这有意义吗？我们得到了一个大小是1的网格。它不是一个长度是10的向量，是一个秩是3的 $10 \times 1 \times 1$ 的张量。我们的损失函数通常希望接收一个向量，而不是一个秩是3的张量，所以你可以在最后调用 `flatten`，`flatten`就是删除所有只是1的维度。这会让它变成一个长度是10的向量，这是我们想要的。

这就是怎样创建一个CNN。然后我们可以用它创建一个learner，传入data、model和损失函数，和可选的度量（metrics）。像之前一样，我使用交叉熵做损失函数。然后，我们可以调用 `learn.summary()` 确认下。

```

1 | learn = Learner(data, model, loss_func = nn.CrossEntropyLoss(),
2 | metrics=accuracy)

```

```

1 | learn.summary()

```

```

1 =====
2 =====
3 Layer (type)          output Shape         Param #
4 =====
5 Conv2d                 [128, 8, 14, 14]      80
6 =====
7
8 BatchNorm2d            [128, 8, 14, 14]      16
9 =====
10
11 ReLU                  [128, 8, 14, 14]      0
12 =====
13 Conv2d                 [128, 16, 7, 7]      1168
14 =====
15
16 BatchNorm2d            [128, 16, 7, 7]      32
17 =====
18
19

```

```

14  ReLU           [128, 16, 7, 7]    0
15
16  Conv2d         [128, 32, 4, 4]   4640
17
18  BatchNorm2d   [128, 32, 4, 4]   64
19
20  ReLU           [128, 32, 4, 4]    0
21
22  Conv2d         [128, 16, 2, 2]   4624
23
24  BatchNorm2d   [128, 16, 2, 2]   32
25
26  ReLU           [128, 16, 2, 2]    0
27
28  Conv2d         [128, 10, 1, 1]   1450
29
30  BatchNorm2d   [128, 10, 1, 1]   20
31
32  Lambda          [128, 10]        0
33
34  Total params: 12126

```

在经过第一个卷积层后，我们降到了 14×14 ，在第二个卷积层后是 7×7 ，然后是 4×4 , 2×2 , 1×1 。`flatten`（在summary里是`Lambda`），可以看到它的输入是 1×1 ，现在它里面存的是长度是10的向量，批次里每个条目都有一条，所以整个mini batch是 128×10 的矩阵。

只是确认下有没有问题，我们可以取到我们之前创建的X的mini batch，把它放到GPU，直接调用model。对任何PyTorch模块，我们都可以说把它当作一个函数，它返回了一个 128×10 的结果，和我们预期的一样。

```
1 | xb = xb.cuda()
```

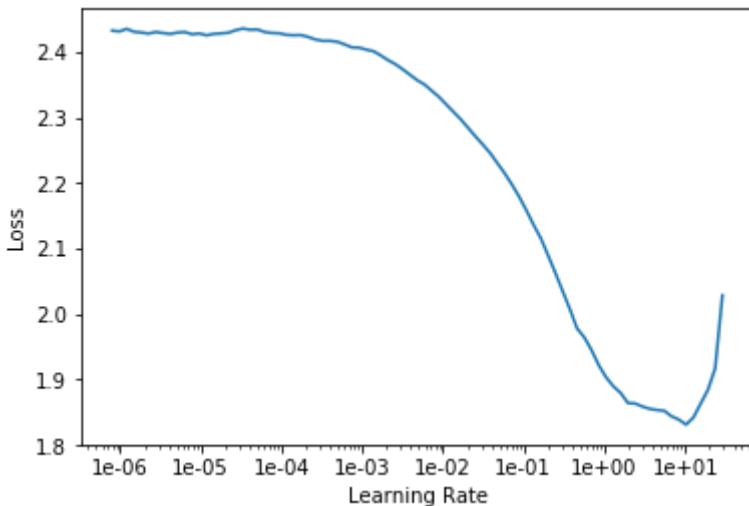
```
1 | model(xb).shape
```

```
1 | torch.size([128, 10])
```

这就是我们怎样直接得到预测值。`lr_find`, `fit_one_cycle`。好，我们得到了一个准确率98.6%的卷积网络。

```
1 | learn.lr_find(end_lr=100)
```

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



```
1 | learn.fit_one_cycle(3, max_lr=0.1)
```

Total time: 00:18

epoch	train_loss	valid_loss	accuracy
1	0.215413	0.169024	0.945300
2	0.129223	0.080600	0.974500
3	0.071847	0.042908	0.986400

这是从头训练的，当然，它不是预训练的。我们创建了我们自己的架构。这是你能想到的最简单架构。用了18秒训练。这就是怎样创建一个很准确的数字识别程序，很简单。

重构 15:42

我们把这个重构一下。不再总是写conv、batch norm、ReLU，fastai里已经有 `conv_layer`，可以让你创建conv、batch norm、ReLU的组合。它有很多其它的选项来做其它的任务，但最基本的版本就是我给你们看的这样。我们可以这样重构它：

```
1 | def conv2(ni,nf): return conv_layer(ni,nf,stride=2)
```

```
1 | model = nn.Sequential(
2 |     conv2(1, 8),    # 14
3 |     conv2(8, 16),   # 7
4 |     conv2(16, 32),  # 4
5 |     conv2(32, 16),  # 2
6 |     conv2(16, 10),  # 1
7 |     Flatten()       # remove (1,1) grid
8 | )
```

这是完全一样的神经网络。

```
1 | learn = Learner(data, model, loss_func = nn.CrossEntropyLoss(),
metrics=accuracy)
```

```
1 | learn.fit_one_cycle(10, max_lr=0.1)
```

Total time: 00:53

epoch	train_loss	valid_loss	accuracy
1	0.222127	0.147457	0.955700
2	0.189791	0.305912	0.895600
3	0.167649	0.098644	0.969200
4	0.134699	0.110108	0.961800
5	0.119567	0.139970	0.955700
6	0.104864	0.070549	0.978500
7	0.082227	0.064342	0.979300
8	0.060774	0.055740	0.983600
9	0.054005	0.029653	0.990900
10	0.050926	0.028379	0.991100

让我们再运行久一点，如果运行一分钟，它的准确率到了99.1%，很酷。

ResNet-ish 16:24

怎样能提升它呢？我们要创建一个更深的网络，要创建更深的网络一个很简单的方法是，在每个步长是2的conv后添加一个步长是1的conv。因为步长是1的conv不会改变特征图的大小，你可以随意加。但有一个问题，是这个论文指出的，这是一个非常非常有影响力的论文，叫[Deep Residual Learning for Image Recognition](#)，作者是微软研究院的Kaiming He和他的同事。

他们说，让我们来看看训练错误率。先忘记泛化，只看看在CIFAR-10上训练的网络的训练错误率，用一个20层的网络，每层是3x3的卷积层，就像我刚刚给你们看的网络一样，只是没有batch norm。他们在训练集上训练了一个20层的网络和一个56层的网络。

这个56层的网络有很多参数。中间有很多步长是1的卷积层。这个有更多参数的网络应该会过拟合，是吗？你会估计这个56层的网络的训练错误率很快下降到0，但这没有发生。它比这个规模小一点的网络更差。

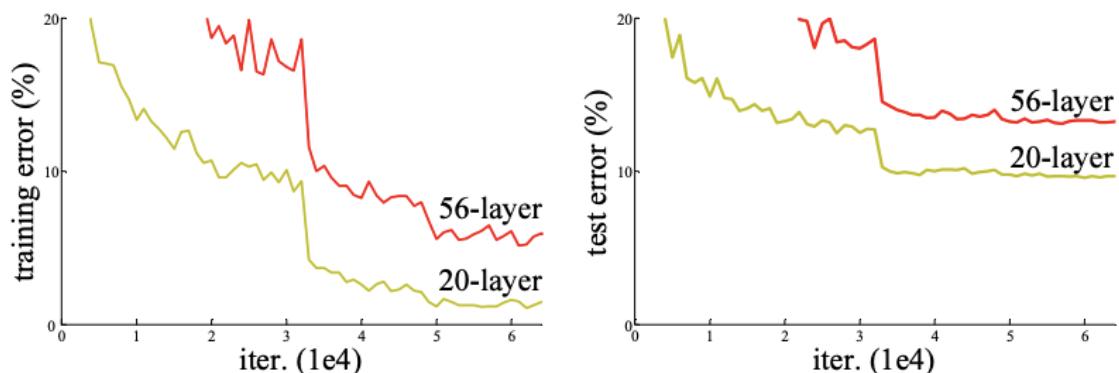


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

当看到奇怪的事情发生时，真正优秀的研究者不会说“噢，不，它没有效果”，他们会说“这很有意思”。Kaiming He说“这很有意思，发生了什么？”。他说“我不知道，但我知道，我可以做一个结构相同的新版本的56层的网络，让它效果至少和这个20层的一样好，就是这样：

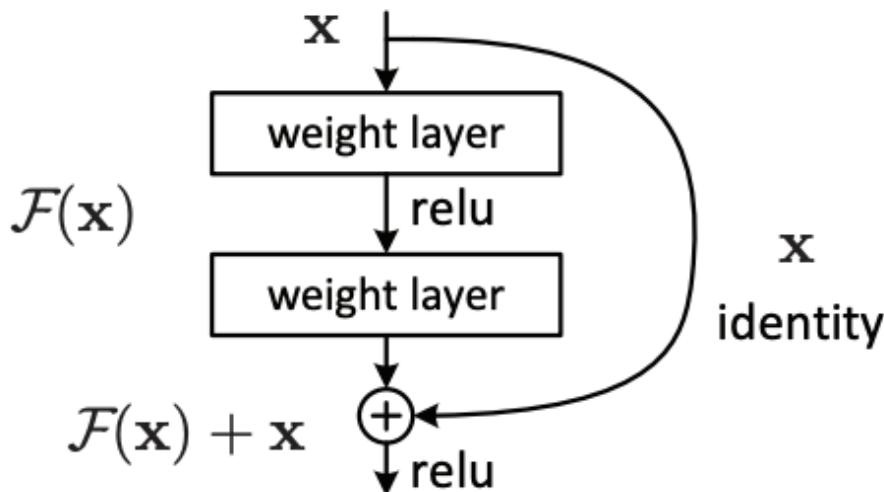


Figure 2. Residual learning: a building block.

每次卷积。我会把这两个卷积的输入和这两个卷积的结果加到一起。“换句话说，不再是用：

$$\text{Output} = \text{Conv2}(\text{Conv1}(x))$$

而是改成用：

$$\text{Output} = x + \text{Conv2}(\text{Conv1}(x))$$

他的观点是56层的卷积网络至少应该比20层的好，因为它可以把除了前20个层以外的，conv2和conv1的层的权重全部设成0，这样X（输入）可以直接穿过去。这个叫**identity connection**，它是identity函数，没有做任何事。它也被叫做**skip connection**。

这就是他的想法。这是这篇论文描述的这个做法背后的直观想法。我们创建的网络至少要比20层的好，因为它包含了这个20层的网络。你可以跳过所有后面这些卷积层。那结果怎样？

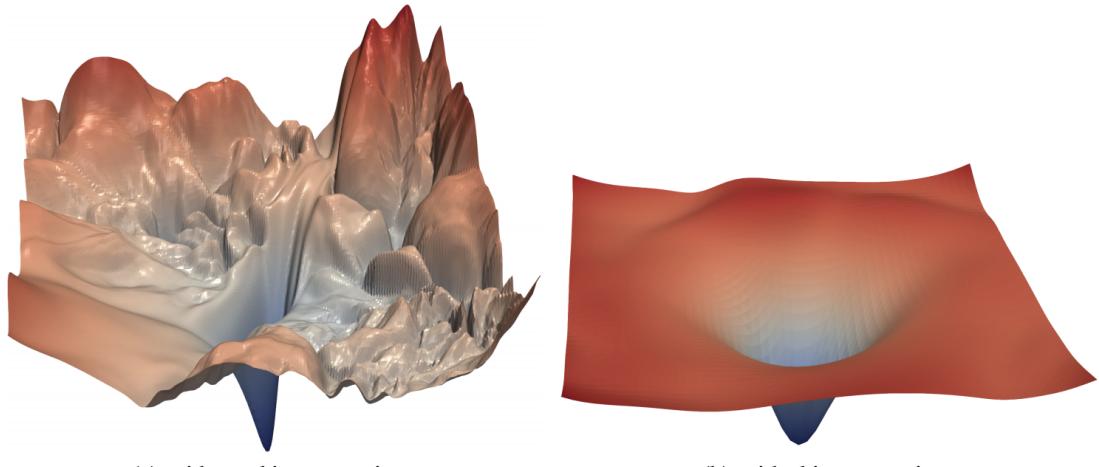
结果他赢了那年的ImageNet比赛。他轻松地赢了比赛。事实上，直到今天，我们还在使用它。我们去年在ImageNet训练速度上破了记录。ResNet是革命性的。

ResBlock Trick 20:36

如果你有兴趣做些研究的话，这是一个trick。每次你找到一个模型，不管它是做什么的，医学图像分割（medical image segmentation）也好，GAN也好，或者其它什么东西。它是在几年前写的，他们可能没有加入ResBlocks。Figure 2是我们叫做ResBlock的东西。他们可能没有把这个加进去。所以你可以把它们的卷积替换成一些ResBlock，你能训练得更快，得到更好的结果。这是一个好方法。

可视化神经网络的损失空间 [Visualizing the Loss Landscape of Neural Nets \[21:16\]](#)

Rachel、David、Sylvain和我刚刚参加NeurIPS会议回来，在那里，我们看到了一个新的展示，它们提出怎样可视化神经网络的损失曲面，这很酷。这是一个了不起的论文，现在看这个课的人，会理解这篇论文里最重要的概念。你可以现在阅读它。你不用全部理解它，但我敢肯定你会发现它很有意义。



(a) without skip connections

(b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

如果你画一个x和y的图，这是权重空间的两个投影，z是损失度。如果你穿过一个56层的没有skip connection的神经网络，它的权重空间是很曲折的。这就是为什么我们得到了这么低的结果，因为它被这些山峰山谷卡住了。有identity connection同样的网络的损失空间是这样的（右侧图）。这是2015年Kaiming He没有认识到的，这是一个肯定会解决它的方法，三年之后，人们才知道为什么这能解决它。这让我回忆起几周前我们讲的batch norm，人们有时会在事情发生过后一段时间，才意识到是什么起了作用。

```

1 | class ResBlock(nn.Module):
2 |     def __init__(self, nf):
3 |         super().__init__()
4 |         self.conv1 = conv_layer(nf, nf)
5 |         self.conv2 = conv_layer(nf, nf)
6 |
7 |     def forward(self, x): return x + self.conv2(self.conv1(x))

```

在代码里，按照刚刚讲过的，我们创建了一个ResBlock。我们创建了一个 `nn.Module`，`conv_layer`（记住，一个 `conv_layer` 是 `Conv2d`、`ReLU`、`batch norm` 的组合），我们创建了两个这样的东西，然后在 `forward` 里，我们运行 `conv1(x)`，然后对它运行 `conv2`，然后添加 `x`。

```
1 | help(res_block)
```

```

1 | Help on function res_block in module fastai.layers:
2 |
3 | res_block(nf, dense:bool=False, norm_type:Union[fastai.layers.NormType,
4 | NoneType]=<NormType.Batch: 1>, bottle:bool=False, **kwargs)
      Resnet block of `nf` features.

```

这是 `fastai` 里的 `res_block` 函数，你可以直接调用 `res_block`，你可以直接传入你需要多少过滤器。

```
1 model = nn.Sequential(
2     conv2(1, 8),
3     res_block(8),
4     conv2(8, 16),
5     res_block(16),
6     conv2(16, 32),
7     res_block(32),
8     conv2(32, 16),
9     res_block(16),
10    conv2(16, 10),
11    Flatten()
12 )
```

这是我在notebook里定义的ResBlock，用这个ResBlock，我重新写了之前的CNN，在除最后一个之外的所有conv2后面，添加了res_block，它有了更多的层，可以做更多的计算。但不会更难优化。

让我们再重构一次。因为我用了conv2 res_block很多次，我们来把它们提取到一个小的sequential模型里，这就把它重构成了这样：

```
1 def conv_and_res(ni, nf): return nn.Sequential(conv2(ni, nf), res_block(nf))
```

```
1 model = nn.Sequential(
2     conv_and_res(1, 8),
3     conv_and_res(8, 16),
4     conv_and_res(16, 32),
5     conv_and_res(32, 16),
6     conv2(16, 10),
7     Flatten()
8 )
```

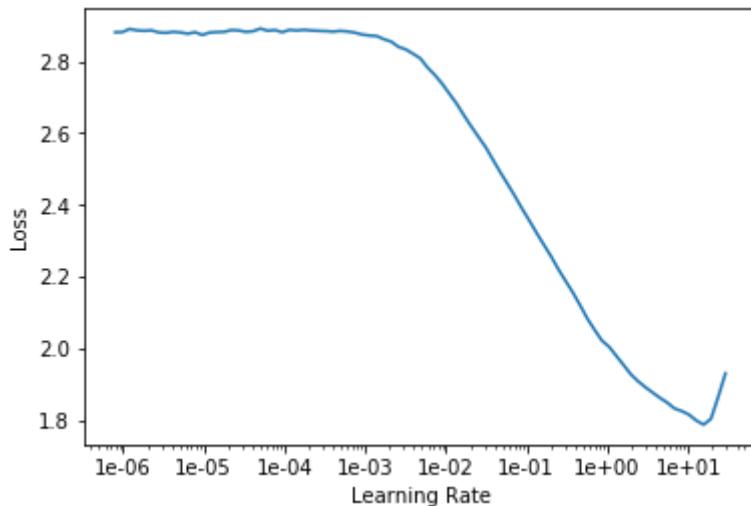
如果你在尝试新的架构，持续重构可以让你更少出错。很少人这样做。你看到的大多数研究代码都很笨重，这样经常会出现错误。不要这样做。你们都是coder，用你们的编程技能让事情保持简单。

[24:47]

好了，这是ResNet架构。像之前一样做lr_find，fit一会儿，得到了99.54%。

```
1 learn = Learner(data, model, loss_func = nn.CrossEntropyLoss(),
metrics=accuracy)
```

```
1 learn.lr_find(end_lr=100)
2 learn.recorder.plot()
```



```
1 | learn.fit_one_cycle(12, max_lr=0.05)
```

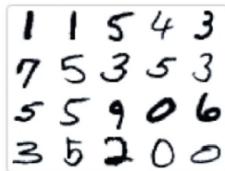
Total time: 01:48

epoch	train_loss	valid_loss	accuracy
1	0.179228	0.102691	0.971300
2	0.111155	0.089420	0.973400
3	0.099729	0.053458	0.982500
4	0.085445	0.160019	0.951700
5	0.074078	0.063749	0.980800
6	0.057730	0.090142	0.973800
7	0.054202	0.034091	0.989200
8	0.043408	0.042037	0.986000
9	0.033529	0.023126	0.992800
10	0.023253	0.017727	0.994400
11	0.019803	0.016165	0.994900
12	0.023228	0.015396	0.995400

这很有意义，因为我们是用一个我们从头开始构建的架构训练出来的，这不是从其它地方搬来的，是从头脑里写出来的。这是什么水平呢，0.45%的错误率基本是三四年前这个数据集的最佳成绩（state of the art）。

MNIST

who is the best in MNIST ?



MNIST 50 results collected

Units: error %

Classify handwritten digits. Some additional results are available on the [original dataset page](#).

0.40%	Hybrid Orthogonal Projection and Estimation (HOPE): A New Framework to Probe and Learn Neural Networks	arXiv 2015
0.42%	Multi-Loss Regularized Deep Neural Network	CSVT 2015
0.45%	Maxout Networks	ICML 2013
0.45%	Training Very Deep Networks	NIPS 2015

现在MNIST被认为是一个非常简单的数据集，我不能说“哇，我们打破了一些记录”这样的话。有人得到了更低的错误率。但我要说，ResNet的确是极其有用的网络。这是我们在快速训练ImageNet中使用的全部东西。它很流行，很多库花了很多时间优化它，它运行得很快。一些使用separable或group convolutions现代风格的架构通常在实践中不会运行得很快。

```
class ResBlock(nn.Module):
    def __init__(self, nf):
        super().__init__()
        self.conv1 = conv_layer(nf,nf)
        self.conv2 = conv_layer(nf,nf)

    def forward(self, x):
        return x + self.conv2(self.conv1(x))

class MergeLayer(nn.Module):
    def __init__(self, dense=False):
        super().__init__()
        self.dense=dense

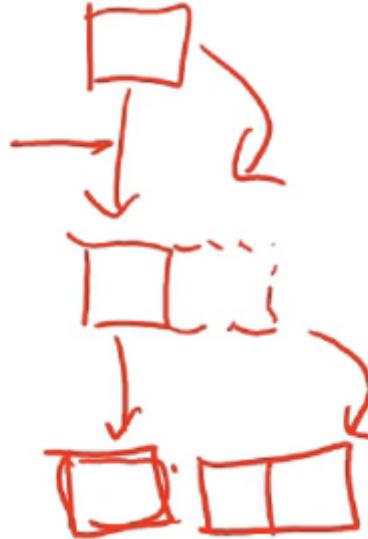
    def forward(self, x):
        return torch.cat([x,x.orig], dim=1) if self.dense else (x+x.orig)

def res_block(nf, dense=False, norm_type:Optional[NormType]=NormType.Batch,
             "Resnet block of `nf` features."
             norm2 = norm_type
             if not dense and (norm_type==NormType.Batch): norm2 = NormType.BatchZero
             nf_inner = nf//2 if bottle else nf
             return SequentialEx(conv_layer(nf, nf_inner, norm_type=norm_type, **kwargs),
                                 conv_layer(nf_inner, nf, norm_type=norm2, **kwargs),
                                 MergeLayer(dense))
```

如果你看看fastai里 `res_block` 的定义，你会看到它和这里的有点不同，这是我创建了一个叫 `MergeLayer` 的东西。在forward里，执行的是 `x+x.orig`。你可以看到一些ResNet形状的东西。什么是 `x.orig`？如果你创建一个叫 `SequentialEx` 的特别的sequential模型，它和fastai里的sequential拓展类似。它就像一个普通的sequential模型，但我们把输入保存在 `x.orig`。所以这个 `SequentialEx` 和 `conv_layer`, `conv_layer`, `MergeLayer` 做的事情和 `ResBlock` 一样。你可以用 `SequentialEx` 和 `MergeLayer` 很简单的创建你自己的ResNet block变体。

当你创建MergeLayer时，你可以选择设置 `dense=True`，这会发生什么？如果你这样做，它不会运行 `x+x.orig`，它会运行 `cat([x, x.orig])`。换句话说，不再用加法，它做了一个concatenate（连接）。你的输入进入了Res block，当你用concatenate替代相加时，它不再调用Res block，它调用的是一个dense block。它不再叫ResNet，它叫DenseNet。

DenseNet是在ResNet提出一年后提出的，如果你阅读DenseNet论文，会觉得非常复杂，和ResNet差别很大。但实际上，它们是相同的，只不过把加法换成了cat。你的输入进入dense block，有一些卷积在这里，然后你得到了输出，然后你做了identity connection，记住，它不是用加法，它用了concat，所以通道维度变大了一些。然后，我们做下一个dense block，最后，我们像以前一样，得到卷积的结果，只是这次变大了。



可以看到，使用dense block，它变得越来越大，有意思的是，原来的输入还在这里。实际上，不管走了多深，原始的输入像素还是保存在这，原始的第一层特征还在，原始的第二层特征还在。可以想象，DenseNet很耗内存。有方法可以处理这个。时不时的，你可以使用一个普通的卷积，它会把你的通道降下来，但它们会耗费内存。但是，它们有很少的参数。所以，处理小数据集时，你应该试下dense block和DenseNet。它们在小数据集上的效果会很好。

因为它能一直保存这些原始的输入像素，它在分割问题（segmentation）上效果很好。因为对分割来说，你需要能重新组织图片的原始图像，所以保存所有这些原始像素非常有帮助。

U-Net [30:16]

这就是ResNet。Resnet令人惊叹。这些skipped connection在其它地方也很有用。它们在为分割问题（segmentation）设计的架构上尤其有用。**在准备这节课时，我看了很多比较老的论文，想象如果作者有了我们现在的这些现代技术会怎样，我尝试用更现代的方式重新构造它们。我最近已经用更现代的方式重新构造U-Net，我们现在就要学习它。这篇语义分割论文，取到了CamVid的最佳成绩91.5。**

The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation



Simon Jégou¹ Michal Drozdzal^{2,3} David Vazquez^{1,4} Adriana Romero¹ Yoshua Bengio¹

¹Montreal Informatique et Recherche Opérationnelle, Université de Montréal

²Image & Vision Group, University of Waterloo

Model	Pretained	# parameters (M)	learn.fit_one_cycle(10, lrs)				Total time: 04:57			
			epoch	train_loss	valid_loss	acc_cavmid	Sidewalk	Cyclist	Mean IoU	Global accuracy
SegNet [1]	✓	29.5	1	0.163259	0.226014	0.939663	60.5	24.8	46.4	62.5
Bayesian SegNet [15]	✓	90.5	2	0.159221	0.223871	0.940497	63.1	86.9		

learn = unet_learner(data, models.resnet34, metrics=metrics, wd=wd, bottle=True)										
Model	Pretained	# parameters (M)	epoch	train_loss	valid_loss	acc_cavmid	Sidewalk	Cyclist	Mean IoU	Global accuracy
DeepLab-LFOV [5]	✓	37.3	5	0.150219	0.227719	0.941239	75.4	50.1	61.6	—
Dilation8 [37]	✓	140.8	6	0.155152	0.226728	0.941032	75.3	55.5	65.3	79.0
Dilation8 + FSO [17]	✓	140.8	7	0.150818	0.230083	0.940657	76.0	57.2	66.1	88.3
Classic Upsampling	✗	20	8	0.149479	0.229187	0.940948	69.6	25.1	55.2	86.8
FC-DenseNet56 (k=12)	✗	1.5	9	0.148236	0.229072	0.941316	79.9	31.1	58.9	88.9
FC-DenseNet67 (k=16)	✗	3.5	10	0.148074	0.234124	0.940629	81.9	52.1	65.8	90.8
FC-DenseNet103 (k=16)	✗	9.4					82.2	50.5	66.9	91.5

Table 3. Result

CN8 model

这周，使用我们即将学习的架构，我把它提升到了94.1。我们会持续提升它。实际上，就是添加所有的现代技巧，今天我会讲很多这些技巧，有一些会在课程第二部分讲。

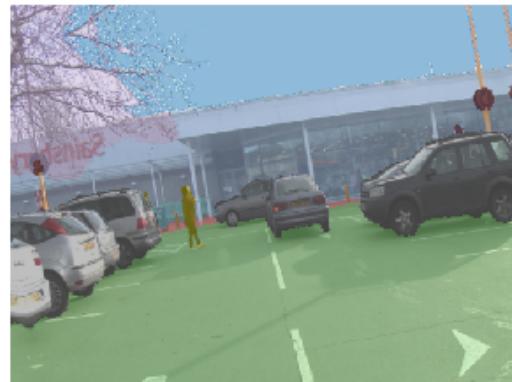
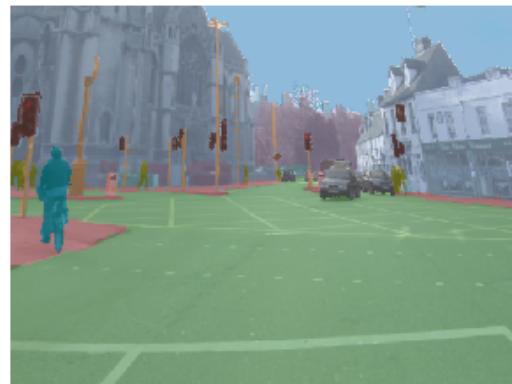
我们要使用U-Net来实现。我们已经用过U-Net了。当我们做CamVid分割时使用过它，但不清楚它做了什么。首先，我们要理解怎样做分割。如果你回头看下我们的[CamVid notebook](#)，在CamVid notebook里，你会记起，我们取了这些图片，为每个像素添加一个类别。

```
1 | bs, size = 8, src_size//2
```

```
1 | src = (SegmentationItemList.from_folder(path)
2 |         .split_by_folder(valid='val')
3 |         .label_from_func(get_y_fn, classes=classes))
```

```
1 | data = (src.transform(get_transforms(), tfm_y=True)
2 |          .databunch(bs=bs)
3 |          .normalize(imagenet_stats))
```

```
1 | data.show_batch(2, figsize=(10, 7))
```



当你对 `SegmentationItemList` 运行 `data.show_batch` 时，**它会自动显示这些用颜色标记过的像素。**

[32:35]

为了用颜色标记出这是一个行人，这是一个骑自行车的人，它需要知道这是什么。它需要真正知道行人是什么样的，它需要准确地知道行人在哪，知道这是行人的胳膊，不是他们的购物篮的一部分。要完成这个任务，它需要真正的理解图片的很多内容。**当看我们最好的模型的结果时，我用眼睛没办法看到单个的像素，我知道有一些错误，但看不出那些出错的像素。它很精确。它是怎么做到的？**

我们得到这个非常非常好的结果的方法**不是使用预训练模型**。

```
1 | name2id = {v:k for k,v in enumerate(codes)}
2 | void_code = name2id['void']
3 |
4 | def acc_camvid(input, target):
5 |     target = target.squeeze(1)
6 |     mask = target != void_code
7 |     return (input.argmax(dim=1)[mask]==target[mask]).float().mean()
```

```
1 | metrics=acc_camvid
2 | wd=1e-2
```

```
1 | learn = unet_learner(data, models.resnet34, metrics=metrics, wd=wd)
```

开始，我们使用了一个ResNet34，你可以在这里看到 `unet_learner(data, models.resnet34, ...)`，**如果你不写 `pretrained=False`，默认情况，用的是 `pretrained=True`。为什么呢？**

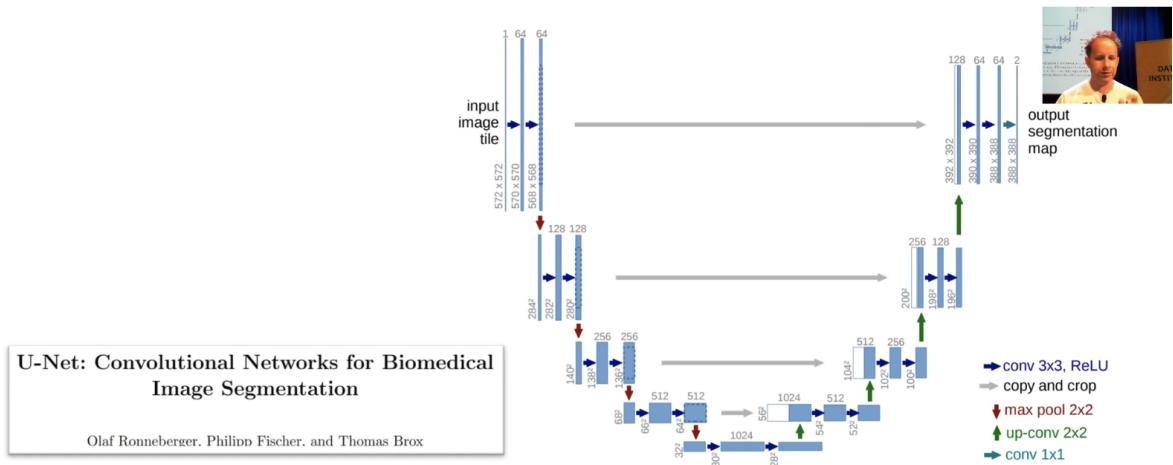


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

我们开始用了一个ResNet34，它处理一个大图片。图片来自于U-Net论文。它们的图片，是一通道572x572的。这是用来做医学图像分割。在经过步长是2的多个卷积后，通道数量增长到128，尺寸缩小到280x280。在这个原始的论文里，它们没有添加padding。所以它们每次做卷积时，会在每个边丢失一个像素，然后把大小缩小一半，再把大小缩小一半，直到降到28x28，有1024个通道。

这是U-Net的下采样 (downsampling) 过程 (左半边叫做下采样) 的样子。我们用了一个ResNet34，你可以用 `learn.summary()` 看到这是一个ResNet34。你可以看到它的大小持续减半，通道数量持续增加。

1 | `learn.summary()`

1	=====	=====	=====	=====
2	Layer (type)	Output Shape	Param #	Trainable
3	=====	=====	=====	=====
4	Conv2d	[8, 64, 180, 240]	9408	False
5				
6	BatchNorm2d	[8, 64, 180, 240]	128	True
7				
8	ReLU	[8, 64, 180, 240]	0	False
9				
10	MaxPool2d	[8, 64, 90, 120]	0	False
11				
12	Conv2d	[8, 64, 90, 120]	36864	False
13				
14	BatchNorm2d	[8, 64, 90, 120]	128	True
15				
16	ReLU	[8, 64, 90, 120]	0	False
17				
18	Conv2d	[8, 64, 90, 120]	36864	False
19				
20	BatchNorm2d	[8, 64, 90, 120]	128	True
21				
22	Conv2d	[8, 64, 90, 120]	36864	False
23				
24	BatchNorm2d	[8, 64, 90, 120]	128	True
25				
26	ReLU	[8, 64, 90, 120]	0	False
27				
28	Conv2d	[8, 64, 90, 120]	36864	False

29				
30	BatchNorm2d	[8, 64, 90, 120]	128	True
31				
32	Conv2d	[8, 64, 90, 120]	36864	False
33				
34	BatchNorm2d	[8, 64, 90, 120]	128	True
35				
36	ReLU	[8, 64, 90, 120]	0	False
37				
38	Conv2d	[8, 64, 90, 120]	36864	False
39				
40	BatchNorm2d	[8, 64, 90, 120]	128	True
41				
42	Conv2d	[8, 128, 45, 60]	73728	False
43				
44	BatchNorm2d	[8, 128, 45, 60]	256	True
45				
46	ReLU	[8, 128, 45, 60]	0	False
47				
48	Conv2d	[8, 128, 45, 60]	147456	False
49				
50	BatchNorm2d	[8, 128, 45, 60]	256	True
51				
52	Conv2d	[8, 128, 45, 60]	8192	False
53				
54	BatchNorm2d	[8, 128, 45, 60]	256	True
55				
56	Conv2d	[8, 128, 45, 60]	147456	False
57				
58	BatchNorm2d	[8, 128, 45, 60]	256	True
59				
60	ReLU	[8, 128, 45, 60]	0	False
61				
62	Conv2d	[8, 128, 45, 60]	147456	False
63				
64	BatchNorm2d	[8, 128, 45, 60]	256	True
65				
66	Conv2d	[8, 128, 45, 60]	147456	False
67				
68	BatchNorm2d	[8, 128, 45, 60]	256	True
69				
70	ReLU	[8, 128, 45, 60]	0	False
71				
72	Conv2d	[8, 128, 45, 60]	147456	False
73				
74	BatchNorm2d	[8, 128, 45, 60]	256	True
75				
76	Conv2d	[8, 128, 45, 60]	147456	False
77				
78	BatchNorm2d	[8, 128, 45, 60]	256	True
79				
80	ReLU	[8, 128, 45, 60]	0	False
81				
82	Conv2d	[8, 128, 45, 60]	147456	False
83				
84	BatchNorm2d	[8, 128, 45, 60]	256	True
85				
86	Conv2d	[8, 256, 23, 30]	294912	False

87				
88	BatchNorm2d	[8, 256, 23, 30]	512	True
89				
90	ReLU	[8, 256, 23, 30]	0	False
91				
92	Conv2d	[8, 256, 23, 30]	589824	False
93				
94	BatchNorm2d	[8, 256, 23, 30]	512	True
95				
96	Conv2d	[8, 256, 23, 30]	32768	False
97				
98	BatchNorm2d	[8, 256, 23, 30]	512	True
99				
100	Conv2d	[8, 256, 23, 30]	589824	False
101				
102	BatchNorm2d	[8, 256, 23, 30]	512	True
103				
104	ReLU	[8, 256, 23, 30]	0	False
105				
106	Conv2d	[8, 256, 23, 30]	589824	False
107				
108	BatchNorm2d	[8, 256, 23, 30]	512	True
109				
110	Conv2d	[8, 256, 23, 30]	589824	False
111				
112	BatchNorm2d	[8, 256, 23, 30]	512	True
113				
114	ReLU	[8, 256, 23, 30]	0	False
115				
116	Conv2d	[8, 256, 23, 30]	589824	False
117				
118	BatchNorm2d	[8, 256, 23, 30]	512	True
119				
120	Conv2d	[8, 256, 23, 30]	589824	False
121				
122	BatchNorm2d	[8, 256, 23, 30]	512	True
123				
124	ReLU	[8, 256, 23, 30]	0	False
125				
126	Conv2d	[8, 256, 23, 30]	589824	False
127				
128	BatchNorm2d	[8, 256, 23, 30]	512	True
129				
130	Conv2d	[8, 256, 23, 30]	589824	False
131				
132	BatchNorm2d	[8, 256, 23, 30]	512	True
133				
134	ReLU	[8, 256, 23, 30]	0	False
135				
136	Conv2d	[8, 256, 23, 30]	589824	False
137				
138	BatchNorm2d	[8, 256, 23, 30]	512	True
139				
140	Conv2d	[8, 256, 23, 30]	589824	False
141				
142	BatchNorm2d	[8, 256, 23, 30]	512	True
143				
144	ReLU	[8, 256, 23, 30]	0	False

145				
146	Conv2d	[8, 256, 23, 30]	589824	False
147				
148	BatchNorm2d	[8, 256, 23, 30]	512	True
149				
150	Conv2d	[8, 512, 12, 15]	1179648	False
151				
152	BatchNorm2d	[8, 512, 12, 15]	1024	True
153				
154	ReLU	[8, 512, 12, 15]	0	False
155				
156	Conv2d	[8, 512, 12, 15]	2359296	False
157				
158	BatchNorm2d	[8, 512, 12, 15]	1024	True
159				
160	Conv2d	[8, 512, 12, 15]	131072	False
161				
162	BatchNorm2d	[8, 512, 12, 15]	1024	True
163				
164	Conv2d	[8, 512, 12, 15]	2359296	False
165				
166	BatchNorm2d	[8, 512, 12, 15]	1024	True
167				
168	ReLU	[8, 512, 12, 15]	0	False
169				
170	Conv2d	[8, 512, 12, 15]	2359296	False
171				
172	BatchNorm2d	[8, 512, 12, 15]	1024	True
173				
174	Conv2d	[8, 512, 12, 15]	2359296	False
175				
176	BatchNorm2d	[8, 512, 12, 15]	1024	True
177				
178	ReLU	[8, 512, 12, 15]	0	False
179				
180	Conv2d	[8, 512, 12, 15]	2359296	False
181				
182	BatchNorm2d	[8, 512, 12, 15]	1024	True
183				
184	BatchNorm2d	[8, 512, 12, 15]	1024	True
185				
186	ReLU	[8, 512, 12, 15]	0	False
187				
188	Conv2d	[8, 1024, 12, 15]	4719616	True
189				
190	ReLU	[8, 1024, 12, 15]	0	False
191				
192	Conv2d	[8, 512, 12, 15]	4719104	True
193				
194	ReLU	[8, 512, 12, 15]	0	False
195				
196	Conv2d	[8, 1024, 12, 15]	525312	True
197				
198	Pixelshuffle	[8, 256, 24, 30]	0	False
199				
200	ReplicationPad2d	[8, 256, 25, 31]	0	False
201				
202	AvgPool2d	[8, 256, 24, 30]	0	False

203				
204	ReLU	[8, 1024, 12, 15]	0	False
205				
206	BatchNorm2d	[8, 256, 23, 30]	512	True
207				
208	Conv2d	[8, 512, 23, 30]	2359808	True
209				
210	ReLU	[8, 512, 23, 30]	0	False
211				
212	Conv2d	[8, 512, 23, 30]	2359808	True
213				
214	ReLU	[8, 512, 23, 30]	0	False
215				
216	ReLU	[8, 512, 23, 30]	0	False
217				
218	Conv2d	[8, 1024, 23, 30]	525312	True
219				
220	PixelShuffle	[8, 256, 46, 60]	0	False
221				
222	ReplicationPad2d	[8, 256, 47, 61]	0	False
223				
224	AvgPool2d	[8, 256, 46, 60]	0	False
225				
226	ReLU	[8, 1024, 23, 30]	0	False
227				
228	BatchNorm2d	[8, 128, 45, 60]	256	True
229				
230	Conv2d	[8, 384, 45, 60]	1327488	True
231				
232	ReLU	[8, 384, 45, 60]	0	False
233				
234	Conv2d	[8, 384, 45, 60]	1327488	True
235				
236	ReLU	[8, 384, 45, 60]	0	False
237				
238	ReLU	[8, 384, 45, 60]	0	False
239				
240	Conv2d	[8, 768, 45, 60]	295680	True
241				
242	PixelShuffle	[8, 192, 90, 120]	0	False
243				
244	ReplicationPad2d	[8, 192, 91, 121]	0	False
245				
246	AvgPool2d	[8, 192, 90, 120]	0	False
247				
248	ReLU	[8, 768, 45, 60]	0	False
249				
250	BatchNorm2d	[8, 64, 90, 120]	128	True
251				
252	Conv2d	[8, 256, 90, 120]	590080	True
253				
254	ReLU	[8, 256, 90, 120]	0	False
255				
256	Conv2d	[8, 256, 90, 120]	590080	True
257				
258	ReLU	[8, 256, 90, 120]	0	False
259				
260	ReLU	[8, 256, 90, 120]	0	False

```

261
262 Conv2d [8, 512, 90, 120] 131584 True
263
264 PixelShuffle [8, 128, 180, 240] 0 False
265
266 ReplicationPad2d [8, 128, 181, 241] 0 False
267
268 AvgPool2d [8, 128, 180, 240] 0 False
269
270 ReLU [8, 512, 90, 120] 0 False
271
272 BatchNorm2d [8, 64, 180, 240] 128 True
273
274 Conv2d [8, 96, 180, 240] 165984 True
275
276 ReLU [8, 96, 180, 240] 0 False
277
278 Conv2d [8, 96, 180, 240] 83040 True
279
280 ReLU [8, 96, 180, 240] 0 False
281
282 ReLU [8, 192, 180, 240] 0 False
283
284 Conv2d [8, 384, 180, 240] 37248 True
285
286 PixelShuffle [8, 96, 360, 480] 0 False
287
288 ReplicationPad2d [8, 96, 361, 481] 0 False
289
290 AvgPool2d [8, 96, 360, 480] 0 False
291
292 ReLU [8, 384, 180, 240] 0 False
293
294 MergeLayer [8, 99, 360, 480] 0 False
295
296 Conv2d [8, 49, 360, 480] 43708 True
297
298 ReLU [8, 49, 360, 480] 0 False
299
300 Conv2d [8, 99, 360, 480] 43758 True
301
302 ReLU [8, 99, 360, 480] 0 False
303
304 MergeLayer [8, 99, 360, 480] 0 False
305
306 Conv2d [8, 12, 360, 480] 1200 True
307
308
309 Total params: 41133018
310 Total trainable params: 19865370
311 Total non-trainable params: 21267648

```

最终，你下降到这个点，如果你使用U-Net架构，它是1024个通道的 28×28 张量。使用ResNet架构，输入224像素，会得到512通道的 7×7 张量。这是很小的网格大小。我们要得到和原始图片大小一样的东西。我们怎样做呢？怎样做增加网格尺寸的计算？在我们现在的工具箱里，没有做这个的方法。我们可以用一个步长是1的卷积做计算来保持网格大小，或者步长是2的卷积做计算来减半网格大小。

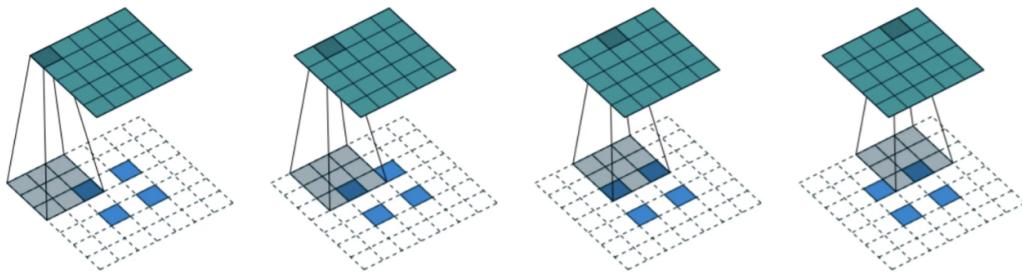
[35:58]

那怎样来加倍网格大小呢？我们做一个**stride half conv**（步长是一半的卷积），也就是**deconvolution**，也被叫做**transpose convolution**。



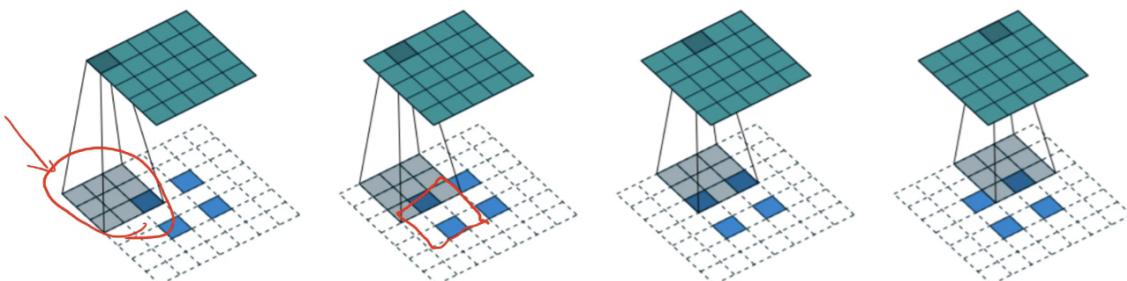
A guide to convolution arithmetic for deep learning

Vincent Dumoulin¹★ and Francesco Visin²★†



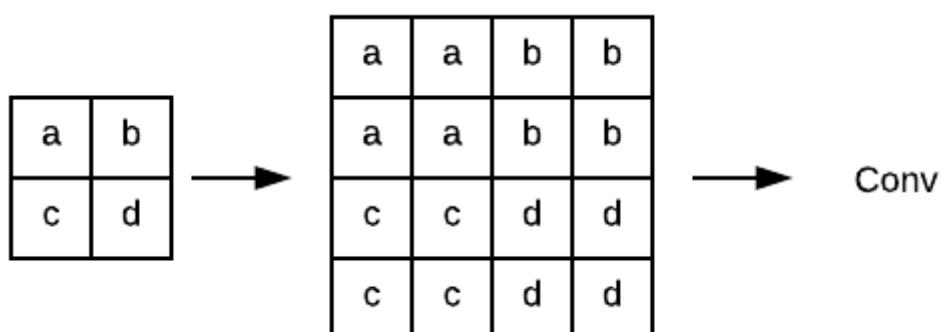
有一个极好的论文，叫[A guide to convolution arithmetic for deep learning](#)，里面有一个很棒的图片展示了一个3x3核、步长是一半的卷积层。如果你有一个2x2的输入，你不单单在周围添加两个像素的padding，还在每个像素之间添加一个像素的padding。现在，如果你把这个3x3的核放在这里，这里，和这里，你可以看到这个3x3的核怎样像之前一样遍历它，你最终能把一个2x2的输出变成5x5的输出。如果你只在周围添加一个像素的padding，你会得到一个4x4的输出。

这就是增加分辨率的方法。这是一两年前，人们使用的方法。有另外一个提升的方法。这其实是一个笨方法，很明显这是一个笨方法，有几个原因。一个原因是，看看左边这些阴影的区域，几乎所有这些像素都是白色的。它们基本都是0。这是多么浪费，多么浪费时间，多么浪费算力。这里什么都没有做。



还有，我们到这个3x3的区域时，9个像素里有2个不是白的，但是左边这个，9个像素里只有1个不是白的。**所以有不同数量的信息进入了卷积的不同部分。**这样传出信息，做这些不必要的计算，用卷积的不同部分处理不同数量的信息，这些没有意义。

现在，人们的做法很简单。如果你有个，比如说，2x2的输入，这些是像素的值 (a,b,c,d)，你想创建一个4x4，为什么不直接这样做？



我现在把 2×2 放大到 4×4 。我没有做什么有意义的运算，现在在它上面，我可以做一个步长是1的卷积，现在我做了一些运算。

这个upsample（上采样）被叫做 **nearest neighbor interpolation**（最近邻插值）。这超级快。你可以做一次最近邻插值，然后做一次步长是1的卷积，现在你用这些没有0的 4×4 矩阵做了一些计算，这样很好，因为得到了A和B的混合值，这是你需要的。

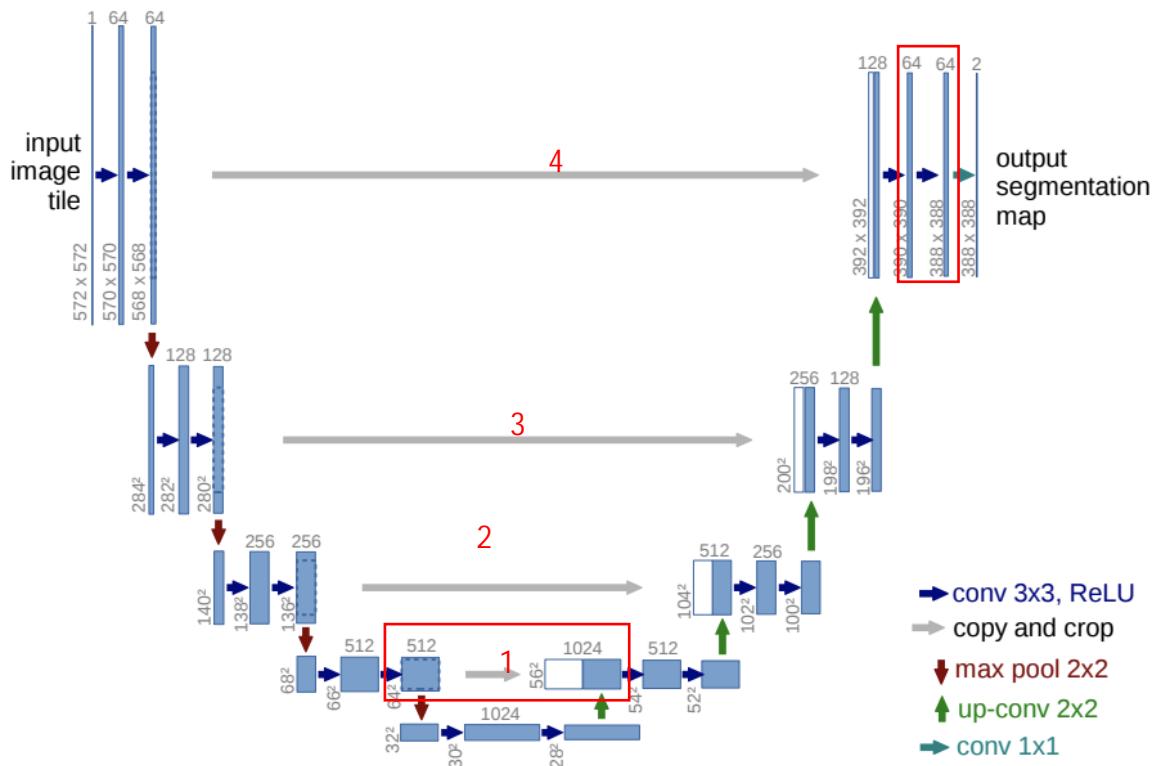
另外一个方法是，不再使用最近邻插值，你可以用双线性插值（bilinear interpolation），就是不再把A复制到所有不同的格子，而是取附近格子的加权平均。

a	a	b	b
a	a	b	b
c	c	d	d
c	c	d	d

比如，我们应该走到这里（红色格子），它周围有3个A，2个C，1个D和2个B，你可以做平均值，不是很准确，但是基本上就是一个加权平均。**双线性平均，你到处都可以看到它。它是一个很标准的技术。每次你在电脑屏幕上看到一个图片，改变它的大小，就是在做双线性插值。**所以你可以做一个双线性插值，然后做一个步长是1的卷积。这就是人们在用的，以后也会用的方法。这一部分就讲这么多，在课程第二部分，我们会学到，fastai库实际上做的是**pixel shuffle**，也叫**sub pixel convolutions**。它不是特别复杂，但今天没时间讲它。它们的思路都是一样的。所有这些东西都是让我们做一个卷积，来把尺寸加倍。

这就是我们的上采样（upsampling）。它让我们从 28×28 到 54×54 ，持续加倍，这很好。在U-Net出来之前人们用的这样的方法，但它效果不是很好，不奇怪，在这个 28×28 特征图里，怎么会有足够的信息来重新构建成一个 572×572 的输出空间？这是很难的任务。最后得到的东西会丢失很多细节。

[41:45]



Olaf Ronneberger et al. 做的是，他们说，嘿，**让我们添加一个skip connection**，一个identity connection，**很神奇，这是之前ResNet里的东西**。这就像一个巨大的跳跃，令人印象深刻。但不是添加一个跳过两个卷积层的skip connection，**他们添加了灰线这里的skip connection**。换句话说，他们添加了从下采样过程到上采样过程的相同尺寸部分的skip connection。**他们没有相加**，这就是为什么你可以看到这里白色和蓝色方块挨在一起，他们没有相加，而是做了连接(concatenate)。这就像dense block，但skip connection跳过越来越多的架构，所以在这里（顶部灰箭头），**你基本上把输入像素放到了最后几层里做计算**。这很简单地解决了分割任务里的细节问题，因为你基本上拿到了所有细节。缺点是，到这里（右上方）你没有做太多层的计算，**只有4层**。你最好能在这个阶段，做了所有必要的计算来识别出这是一个骑自行车的人还是一个行人，但是你可以在上面加上一些东西来判断这个像素是他们鼻子的末端还是树的一部分。**这个效果很好，这就是U-Net。**

[43:33]



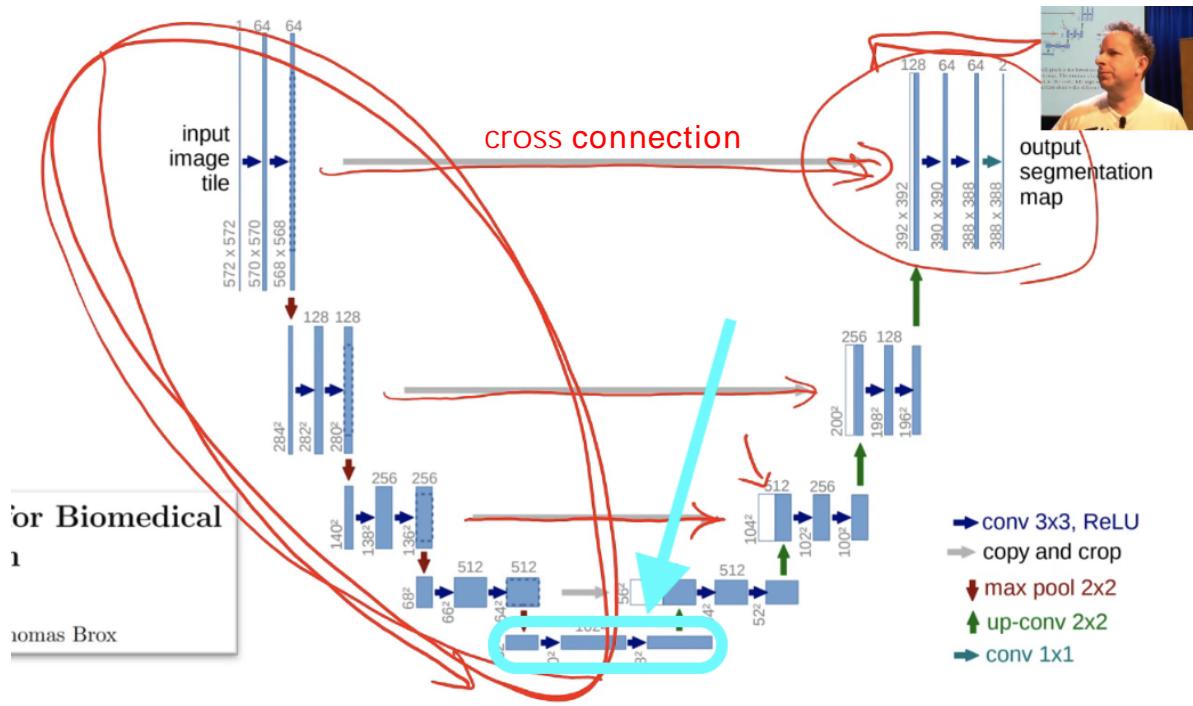
```
ni = sfs_szs[-1][1]
middle_conv = nn.Sequential(conv_layer(ni, ni*2, **kwargs),
                            conv_layer(ni*2, ni, **kwargs)).eval()
x = middle_conv(x)
layers = [encoder, batchnorm_2d(ni), nn.ReLU(), middle_conv]

for i, idx in enumerate(sfs_idxs):
    not_final = i != len(sfs_idxs) - 1
    up_in_c, x_in_c = int(x.shape[1]), int(sfs_szs[idx][1])
    do_blur = blur and (not_final or blur_final)
    sa = self_attention and (i == len(sfs_idxs) - 3)
    unet_block = UnetBlock(up_in_c, x_in_c, self.sfs[i], final_div=not_final,
                          **kwargs).eval()
    layers.append(unet_block)
    x = unet_block(x)

ni = x.shape[1]
if imsize != sfs_szs[0][-2:]:
    layers.append(Pixelshuffle_ICNR(ni, **kwargs))
if last_cross:
    layers.append(MergeLayer(dense=True))
    ni += 3
    layers.append(res_block(ni, bottle=bottle, **kwargs))
layers += [conv_layer(ni, n_classes, ks=1, use_activ=False, **kwargs)]
if y_range is not None:
    layers.append(SigmoidRange(*y_range))
super().__init__(*layers)
```

这是fastai里的代码，关键的东西是encoder。这个encoder代表U-Net的下采样部分，换句话说，在这里，就是ResNet34。在很多情况下，它们是更老的架构，**但是就像我说的，用ResNet替换更老的架构，会提升模型**。对我们来说，确实是这样。我们从encoder开始。

所以，我们的U-Net的 layers 是一个encoder，然后batch norm，然后ReLU，然后 middle_conv (它只是 conv_layer, conv_layer)。记住，在fastai里 conv_layer 只是一个 conv, ReLU, batch norm
。所以，**这个middle_conv是最底部的这两步。**



它做了一点计算。在可能的地方添加更多计算很好。所以是encoder, batch norm, ReLU, 然后两个卷积层。然后我们遍历这些索引 (`sfs_idxs`)。这些索引是什么? 我没有包含代码, 这些是每个步长是2的卷积出现的层数, 我们把它放进一个索引的数组里。然后我们可以遍历它, 我们可以对每个这样的点创建一个 `UnetBlock`, 告诉我们有多少个上采样通道, 有多少个cross connection。这些灰线叫cross connection, 至少我是这样叫的。

[45:16]

这就是 `UnetBlock` 里主要的东西。像我说的, 有很多技巧, 我们用了更好的encoder, 我们在整个上采样用了一些技巧, 用了pixel shuffle, 我们用了另外一个叫ICNR的技巧, 还有另外一个我上周做的方法, 不仅是取这些卷积的结果, 传递它, 我们取了输入像素, 做了另外两个cross connection。这就是这里的 `last_cross`。你可以看到我们添加了一个原始的输入到 `res_block` (你可以看我们的 `MergeLayer`)。

```
class UnetBlock(nn.Module):
    "A quasi-U-Net block, using `PixelShuffle_ICNR` upsampling."
    def __init__(self, up_in_c:int, x_in_c:int, hook:Hook, final_div:bool=True, blur:bool=False,
                 self_attention:bool=False, **kwargs):
        super().__init__()
        self.hook = hook
        self.shuf = Pixelshuffle_ICNR(up_in_c, up_in_c//2, blur=blur, leaky=leaky, **kwargs)
        self.bn = batchnorm_2d(x_in_c)
        ni = up_in_c//2 + x_in_c
        nf = ni if final_div else ni//2
        self.conv1 = conv_layer(ni, nf, leaky=leaky, **kwargs)
        self.conv2 = conv_layer(nf, nf, leaky=leaky, self_attention=self_attention, **kwargs)
        self.relu = relu(leaky=leaky)

    def forward(self, up_in:Tensor) -> Tensor:
        s = self.hook.stored
        up_out = self.shuf(up_in)
        ssh = s.shape[-2:]
        if ssh != up_out.shape[-2:]:
            up_out = F.interpolate(up_out, s.shape[-2:], mode='nearest')
        cat_x = self.relu(torch.cat([up_out, self.bn(s)], dim=1))
        return self.conv2(self.conv1(cat_x))
```

所有这些东西都在 `UnetBlock` 里, `UnetBlock` 要在每个下采样点存储激活值, 实现这个的方式, 就是我们在上节课学的, 使用hook。我们把hook放到ResNet34里, 每当有一个步长是2的卷积时存储激活值, 这里你可以看到, 我们取这个hook (`self.hook=hook`)。我们在这个hook里取出存储的值, 我们只是做 `torch.cat`, 所以, 我们连接上采样卷积和hook的结果, 这是我们做过batch norm和两次卷积的结果。

你在家可以尝试修改这里 (最后一行)。每次你看到这样的两个卷积, 这里有个明显的问题: 如果用ResNet block替代会怎样? 你可能会得到更好的结果, 这是我们要找的东西, 当你看到这样一个架构, 会想“噢, 两个卷积在一列里, 可能应该是一个ResNet block”。

好了，这就是U-Net，它先于ResNet，先于DenseNet，这令人惊奇。它甚至不是在机器学习会议发表的，它实际是在MICCAI发表的，一个专业的医学图像计算会议。很多年里，在医学图像圈子之外都没有多少人知道它。实际上，用U-Net的人轻松赢得了Kaggle的分割比赛，这是它第一次被医学图像圈子之外的人注意到。然后，逐渐地，一些机器学习学术圈的人开始关注，现在每个人都喜欢U-Net，我很高興，它棒极了。

identity connection，不管是相加的方式还是连接的方式，非常有用。它们可以让我们接近很多重要任务的最佳成绩（state of the art）。所以现在我想在另外一个任务里使用它。

图像修复 (Image restoration) [48:31]

下一个我想讲的任务是图像修复。图像修复是处理一张图片，现在我们不会创建一个分割遮罩，而是要创建一个更好的图片。有很多方面的更好，可以是不同的图片。我们用图片生成可以做的是：

- 把低分辨率图片变成高分辨率
- 把黑白图片变成彩色的
- 把被裁剪掉一部分的图片里被裁剪的那部分补全
- 把照片变成线条画
- 把照片变成莫奈的风格

这些都是图片生成任务的例子，在这节课结束后，你将会知道怎样做到这些。

在这里，我们要做图像修复，把一个低分辨率的、低画质的、写有字的JPEG，变成高分辨率的、高画质的、没有文字的图片。

提问：为什么在调用conv2(conv1(x))之前做concat，不是在调用之后？[49:50]

因为如果你在concat之前做卷积，没办法让两部分的通道相互影响。记住，2D conv，它其实是3D的。它作用在2维平面上，但每次都会和秩是3的张量的3个维度做点积。一般我们希望有尽可能多的相互影响。如果你把这个下采样的过程和这个上采样的过程结合起来，会发现有意义的东西。通常，在每次计算时你需要尽可能多的交互影响。

提问：在DenseNet里，当图片/特征图的尺寸在每层间是不同时，怎样把每层连接起来？[50:54]

这是一个很好的问题。如果你有一个步长是2的卷积，你不能用DenseNet。在DenseNet里，你会做dense block，增长，dense block，增长，dense block，增长，这样你得到了越来越多的通道。然后你做一个没有dense block的步长是2的卷积。然后你只做几个dense block。在实践中，dense block实际上不会一直保留所有的信息，到了这些步长是2的卷积这里，就会减少一部分信息。有很多方法处理这些瓶颈层，你可以直接重置。这也也可以让我们把内存控制住，在那时我们可以决定我们要多少通道。

回到图像修复 [52:01]

[lesson7-superres-gan.ipynb](#)

要创建可以把差图片变成好图片的模型，我们需要有包含好图片和差图片的数据集。最简单的方式，就是找到好图片，再把它们变差。

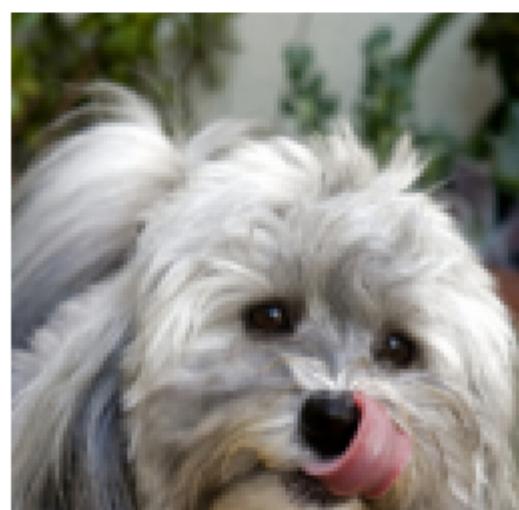
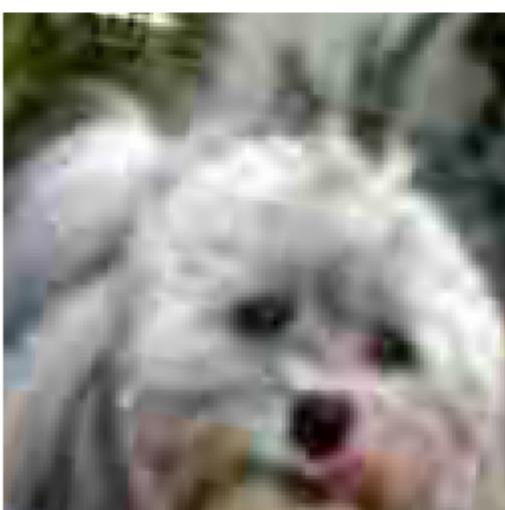
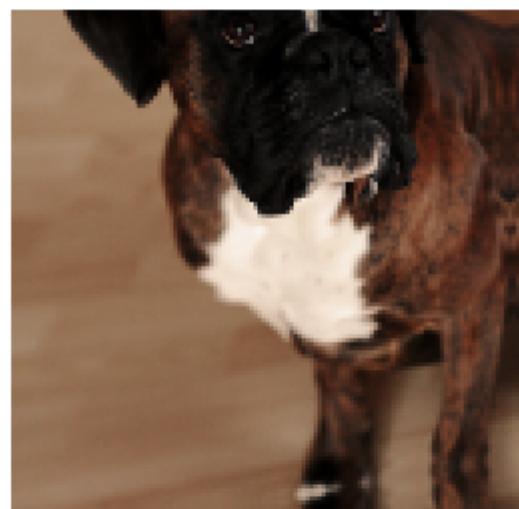
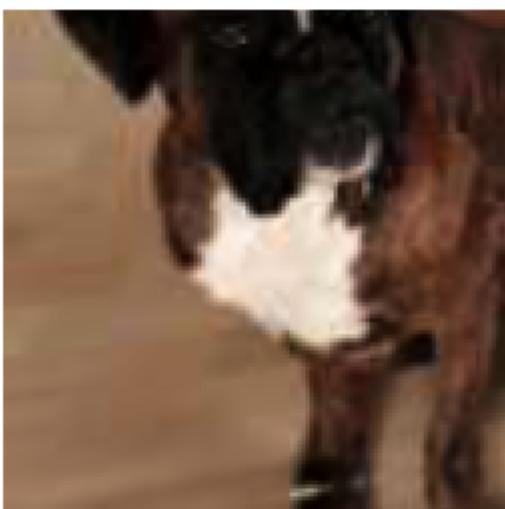
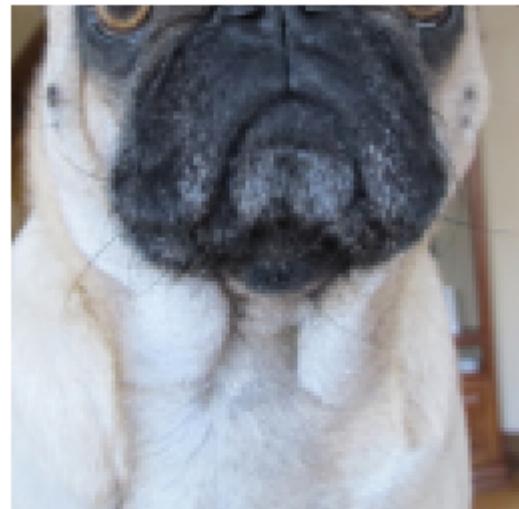
```
1 | from PIL import Image, ImageDraw, ImageFont
```

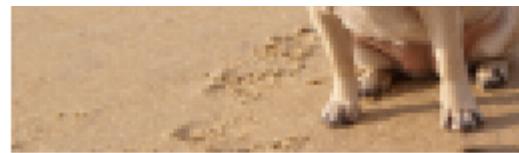
```
1 def crappify(fn,i):
2     dest = path_lr/fn.relative_to(path_hr)
3     dest.parent.mkdir(parents=True, exist_ok=True)
4     img = PIL.Image.open(fn)
5     targ_sz = resize_to(img, 96, use_min=True)
6     img = img.resize(targ_sz, resample=PIL.Image.BILINEAR).convert('RGB')
7     w,h = img.size
8     q = random.randint(10,70)
9
10    ImageDraw.Draw(img).text((random.randint(0,w//2),random.randint(0,h//2)),
11        str(q), fill=(255,255,255))
12    img.save(dest, quality=q)
```

把它们变差的方式是创建一个叫`crappify`的函数，里面包含了你把图片变差的逻辑。我的是这样的，你可以自己写一个：

- 打开好图片
- 调整大小，变成小的96x96分辨率，使用双线性插值
- 然后取一个10到70随机数
- 把这个数画到一些随机的位置
- 然后用这个随机数作为JPEG画质保存图片

如果JPEG画质是10，图片看起来就是垃圾，如果是70，就看起来还可以。生成的低质量的图片和高质量的图片看起来是这样的：





可以看下这个（最后一行），这里有个数字。它是翻转变形后的。你不会总是看到这样的数字，这是我们随机添加的，很多时候，它是没有的。

我们要演示怎样把这样有文字的、画质非常差的、故意制作的图片变成右边的这样。我使用的是剑桥的宠物数据集。我们在第一课用的那个。其它的画质都不如这些猫和狗的图片。

parallel 并行 [53:48]

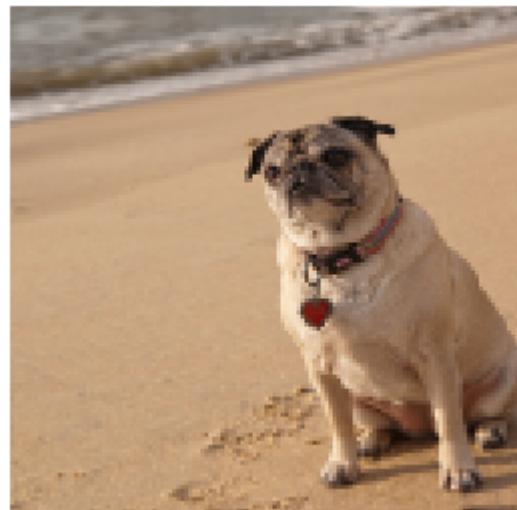
这个把图片变差的过程会花一段时间，但fastai有一个叫 `parallel` 的函数，如果你给 `parallel` 传入一个函数名和一个要处理的东西的list，它会并行地在这些东西上运行这个函数。这会运行得很快。

```
1 | il = ImageItemList.from_folder(path_hr)
2 | parallel(crappify, il.items)
```

写 `crappify` 函数的方式是这次作业里一件有意思的事。在里面做一些你想做的事，写一个有意思的 `crappify` 函数。如果你想为黑白图片着色，你要把它变成黑白的。如果你想补全被裁剪掉一块的图片，就添加一个大黑块。如果你想处理有折痕的旧照片的扫描件，就试着想想在图片上添加印刷墨点和折痕的办法。

你的模型学不会 `crappify` 里没有的东西。因为每次它看这些照片时，会想让输入和输出变成一样的，所以它不会修复那些没加进 `crappify` 里的东西。

[55:09]



现在我们想创建这样一个模型，可以输入左边这样的照片，输出右边这样的照片。显然，我们要用 U-Net，因为我们已经知道U-Net可以做这样的事。我们把数据传到U-Net里。

```
1 | arch = models.resnet34
2 | src = ImageItemList.from_folder(path_lr).random_split_by_pct(0.1, seed=42)
```

```
1 | def get_data(bs, size):
2 |     data = (src.label_from_func(lambda x: path_hr/x.name)
3 |             .transform(get_transforms(max_zoom=2.), size=size, tfm_y=True)
4 |             .databunch(bs=bs).normalize(imagenet_stats, do_y=True))
5 |
6 |     data.c = 3
7 |     return data
```

```
1 | data_gen = get_data(bs, size)
```

我们的data只是这两个文件夹里的文件名，做transform、data bunch、normalize。我们要用ImageNet stats，因为我们要用预训练模型。为什么我们用预训练模型？因为如果我们想掉这个46，你需要知道这可能是什么，你需要知道这是一个什么的图片。不然，怎样知道它应该是什么样子？所以我们要用预训练模型，它知道这些东西是什么。

```
1 | wd = 1e-3
```

```
1 | y_range = (-3., 3.)
```

```
1 | loss_gen = MSELossFlat()
```

```
1 | def create_gen_learner():
2 |     return unet_learner(data_gen, arch, wd=wd, blur=True,
3 |                          norm_type=NormType.Weight,
4 |                          self_attention=True, y_range=y_range,
5 |                          loss_func=loss_gen)
```

```
1 | learn_gen = create_gen_learner()
```

我们用这些数据创建U-Net，架构是ResNet34。这三个东西（blur, norm_type, self_attention）是重要的、有意义的、有用的，但我要把它们放在课程第二部分讲。现在，你每次使用U-Net处理这种问题时都应该包含它们。

这整个东西，我把它叫“generator”。它会generate，这是个生成模型（generative modeling）。它是这样一个东西：它的输出就像是一个真正的物品（在这里，是一个图片），而不仅仅是一个数字。这不是一个正式的定义。我们要创建一个generator learner，就是这个U-Net learner，然后拟合它。我们使用MSE loss，也就是实际像素值和我们的预测的像素值间的均方差。MSE 损失通常接收两个向量。这里，我们有两个图片，我们有一个MSE Loss Flat版本，它简单地把图片拉平成向量。没有理由不这样做，即使你有一个向量，它也可以用，如果你没有向量，它也会工作得很好。

```
1 | learn_gen.fit_one_cycle(2, pct_start=0.8)
```

Total time: 01:35

epoch	train_loss	valid_loss
1	0.061653	0.053493
2	0.051248	0.047272

我们在1分35秒后，把像素值均方差降到了0.05，这不错。像fastai里的所有东西一样，我们默认做迁移学习，它会冻结预训练部分。U-Net的预训练部分是这个下采样部分。这是ResNet在的地方。

```
1 | learn_gen.unfreeze()
```

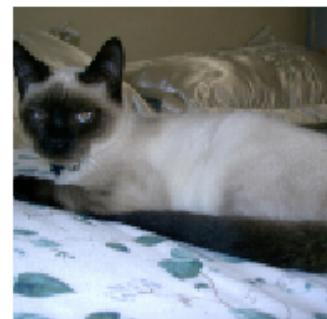
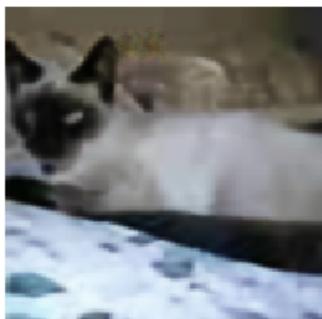
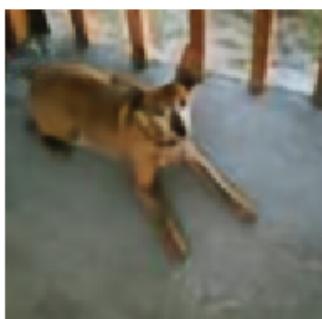
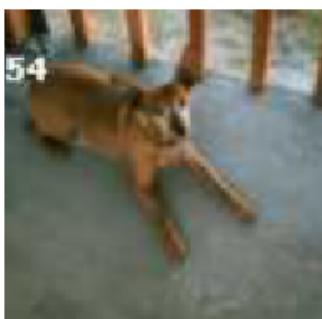
```
1 | learn_gen.fit_one_cycle(3, slice(1e-6,1e-3))
```

Total time: 02:24

epoch	train_loss	valid_loss
1	0.050429	0.046088
2	0.049056	0.043954
3	0.045437	0.043146

```
1 | learn_gen.show_results(rows=4)
```

Input / **Prediction** / Target

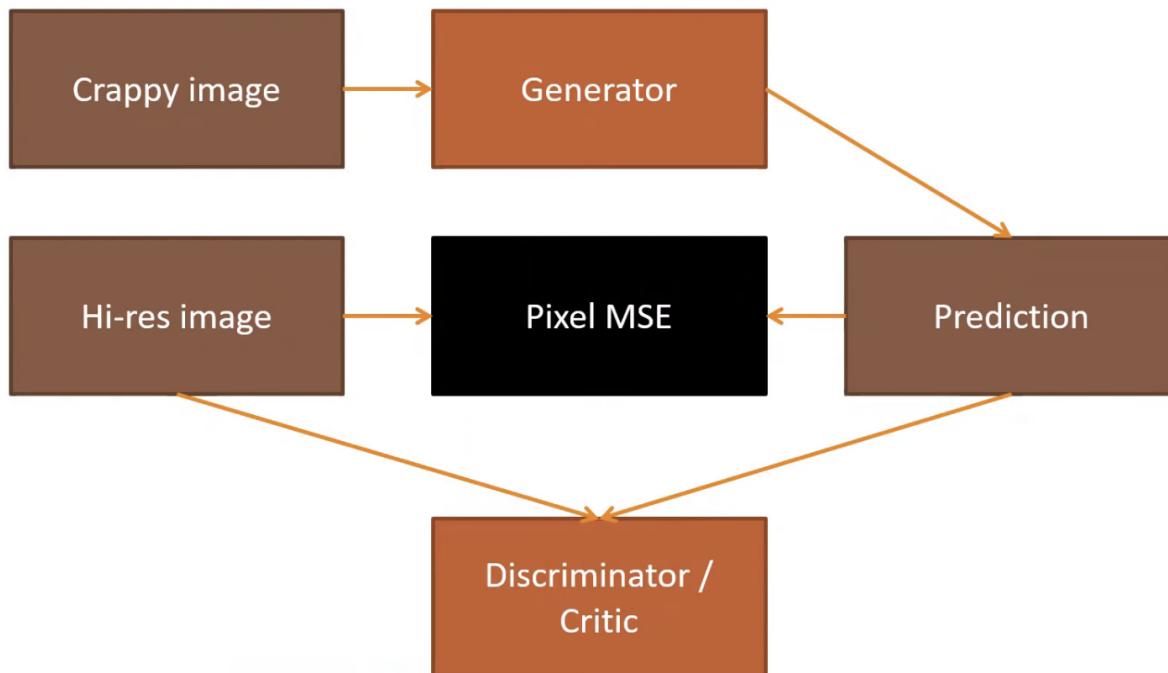


我们解冻它，再训练多些，看看结果！用4分钟的训练，我们得到了可以完美去除数字的模型。它的上采样做得不好，但它确实不错。有时在去除数字时，它可能会留下一点手动添加的东西。但是它确实做了很有用的东西。如果我们想做去水印，我们已经完成了目标。

我们想让中间的图更像右边的图。怎样做到呢？我们没有做到这个，是因为我们的损失函数没有体现我们想要的。因为实际上，左边图片和右边图片的像素均方差非常小。大多数像素和右边的颜色和接近。但我们丢失了枕头上的纹理、基本全部丢失了眼珠、丢失了身体上的纹理。像素均方差认为这是很好的图片了，我们需要更好的损失函数。

生成对抗网络 (Generative Adversarial Network) [59:23]

有一个合适的方法，就是用生成对抗网络 (Generative Adversarial Network, GAN)。GAN用一个损失函数来解决这个问题，这个函数实际上调用了另外一个模型。



我们拿到了差的图片，已经创建了一个generator。它不是特别棒，但也不怎么差，它做出了预测（中间图片）。我们有一个高分辨率的照片（右边图片），我们可以用像素均方差比较高分辨率图片和预测图片。

我们也可以训练另外一个模型，我们可以叫它discriminator（辨别者）或者critic（评价者），这都是相同的东西。我会叫它critic。我们构建一个二分类模型，给它这些生成的图片和原始的高分辨率图片，让它学习它们那个是生成的，哪个是高分辨率的。看一些图片，然后问“嘿，你怎么看，这是一个高分辨率的猫还是一个生成的猫？这个呢？是一个高分辨率的猫还是一个生成的猫？”这就是一个普通的标准二分类交叉熵分类器。我们已经知道怎样做它了。如果有这样一个东西，我们就可以调整generator，不再使用像素均方差做损失度，损失度可以是我们有多擅长瞒过这个critic？我们能不能创建让critic分辨不出的图片？

这是一个很好的方案，因为如果做到了这个，如果损失函数是“我能不能骗过critic”，它会学习生成critic分辨不出的图片。我们可以这样做一段时间，训练几个批次。但是这个critic不够好，因为要分辨它们不是很难。这些图片太差了，所以要分辨出来很简单。所以，在用这个critic做损失函数，训练了一段时间后，这个生成器变得很擅长骗过critic。现在我们要停止训练generator，我们要用这些新生成的图片再训练critic。现在这个generator更好了，对critic来说，要判断哪个是真的哪个是假的变得更难了。我们训练久一点。我们做完这个后，这个critic现在很擅长识别出生成的图片和原始图片的区别，我们要回过头来，再用这个更好的critic作为损失函数优化generator。

我们这样来回做。这就是一个GAN。这是我们的GAN版本。我不知道有没有人写过这个，我们做了一个新版本的GAN，它和原始的GAN很像，但是我们用了这个方法，我们用了预训练的generator和critic。GAN经常上新闻。是很时尚的工具，如果你见过它们，你可能听说过训练起来很难。但我们发现，最难的是在开始。如果你没有预训练的generator，没有预训练的critic，那基本是盲人骑瞎马。generator要生成能骗过critic的东西，但critic什么都不懂，所以没有什么事可做。然后critic试着判断是生成的图片是不是真的，这很明显，所以它直接就可以做到。所以它们都没有进步。然后它们运行的很快，越来越快。

生成模型用的损失度函数效果和像素均方差差不多，辨别模型就像是在第一次生成的数据上做判断，如果你可以找到一种方式避免这些，你可以有很大的进步。

创建一个critic/discriminator [1:04:04]

我们来创建critic。要创建一个完全标准的fastai分类模型，我们需要两个文件夹，一个放高分辨率的图片，一个放生成的图片。我们已经有了存高分辨率图片的文件夹，我们只需要保存生成的图片。

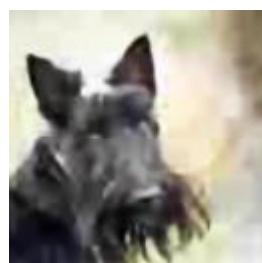
```
1 | name_gen = 'image_gen'  
2 | path_gen = path/name_gen  
  
1 | # shutil.rmtree(path_gen)  
  
1 | path_gen.mkdir(exist_ok=True)  
  
1 | def save_preds(dl):  
2 |     i=0  
3 |     names = dl.dataset.items  
4 |     for b in dl:  
5 |         preds = learn_gen.pred_batch(batch=b, reconstruct=True)  
6 |         for o in preds:  
7 |             o.save(path_gen/names[i].name)  
8 |             i += 1  
  
1 | save_preds(data_gen.fix_dl)
```

这是做这个的一小段代码。我们要创建一个叫 `image_gen`，把它放进一个叫 `path_gen` 变量里。我们写一个叫 `save_preds` 的小函数，它接收一个data loader。我们取出所有的文件名。对一个item list 来说，如果它是一个image item list，`.items` 里存的是文件名。这个data loader的dataset里是文件名。现在，我们遍历data loader里的每一个batch，我们取这个batch的预测batch (`preds`)，`reconstruct=True` 代表它会为batch里的每一个东西创建fastai图片对象。我们遍历每个预测值，保存它们。我们用和原始文件一样的名字，但是会把它放到新目录里。

就是这样。这样保存预测结果。可以看到，我不仅用fastai里现成的东西，也给你们看怎样写自己的东西。通常，这不需要多少代码。如果你学习课程第二部分，里面很多地方就像这样里的一样，会教你怎样用fastai库里的东西，当然，这里是是怎样写库里的代码。越来越多地，我们会学写自己的代码。

好了，保存了这些预测值。我们来在第一个上执行 `PIL.Image.open`，它显示在这里。这是我们生成的一个图片。

```
1 | PIL.Image.open(path_gen.ls()[0])
```



现在我可以像以前一样训练一个critic。**重启Jupyter notebook来释放内存很麻烦。如果你知道是什么占用了大量的GPU，你可以直接把它设成None**，这是一个简单的方法。我们对这个learner做了这样的处理，然后运行 `gc.collect`，这会让Python做内存垃圾回收，做完之后，内存就正常了。**你可以使用所有的GPU内存了。**

```
1 | learn_gen=None  
2 | gc.collect()
```

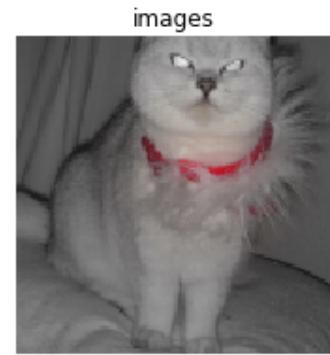
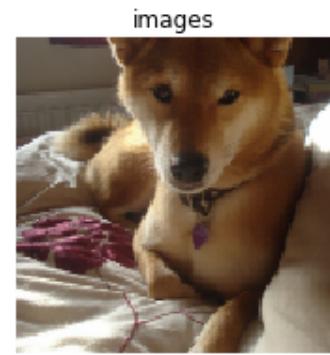
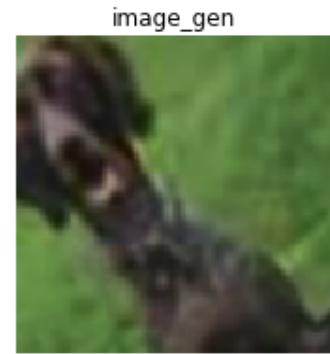
如果你用 nvidia-smi 查看内存，你看不出它被释放了，因为PyTorch还占用着它们做缓存，但是这些内存已经可以用了。这样，你不用重启notebook了。

```
1 | def get_crit_data(classes, bs, size):  
2 |     src = ImageItemList.from_folder(path,  
3 |         include=classes).random_split_by_pct(0.1, seed=42)  
4 |     ll = src.label_from_folder(classes=classes)  
5 |     data = (ll.transform(get_transforms(max_zoom=2.), size=size)  
6 |             .databunch(bs=bs).normalize(imagenet_stats))  
7 |     data.c = 3  
8 |     return data
```

我们要创建critic。和以前一样，这是一个从文件夹生成的image item list，这个classes是 image_gen 和 images。我们要做一个随机的分割得到一个验证集，因为我们想知道critic做得怎么样。我们像之前一样按文件夹标注，做一些变形，data bunch，normalize。这样我们得到了一个标准的分类器。这是里面的一些图片：

```
1 | data_crit = get_crit_data([name_gen, 'images'], bs=bs, size=size)
```

```
1 | data_crit.show_batch(rows=3, ds_type=DatasetType.Train, imgsize=3)
```



这是一个真的图片、真的图片、生成图片、生成图片等等，我们要区分它们的类别。

```
1 | loss_critic = AdaptiveLoss(nn.BCEwithLogitsLoss())
```

我们还是像以前一样要使用交叉熵。但是，我们不再使用ResNet。在课程第二部分，我们再详细讲原因。基本上，你在做GAN时，你要特别注意，generator和critic不能都用相同的方向，让权重增长到失去控制。我们需要用一种叫spectral normalization的东西让GAN正常运行，我们会在课程第二部分学习这个。

```
1 | def create_critic_learner(data, metrics):
2 |     return Learner(data, gan_critic(), metrics=metrics,
|         loss_func=loss_critic, wd=wd)
```

```
1 | learn_critic = create_critic_learner(data_crit, accuracy_thresh_expand)
```

不管怎样，如果你用 gan_critic，fastai会给你一个适合GAN的二元分类器。我怀疑我们可以在这里用一个ResNet，我们要创建一个带spectral norm的预训练ResNet。希望能快点验证下。我们会看到怎样做，但是现在，最好的方法是用 gan_critic。一个GAN critic在算损失度时会对图像的不同部分用稍微有点不同的方式做平均，所以现在在做GAN时，你要用 AdaptiveLoss 包裹住你的损失函数。还是一样，你会在课程第二部分看到细节。现在，只要知道这是你要做的，它很管用。

除了稍微改变损失函数，和稍微改变架构，其它的东西都是一样的。我们可以调用 `create_critic_learner` 来创建critic。因为我们用了不同的架构和损失函数，我们要用一个不同的 metric。这是critic的GAN版本的准确率。然后我们可以训练它，你可以看到它区分差的图片和好图片的准确率是98%。这些是生成的图片，generator已经知道怎样去掉上面的数字。

```
1 | learn_critic.fit_one_cycle(6, 1e-3)
```

Total time: 09:40

epoch	train_loss	valid_loss	accuracy_thresh_expand
1	0.678256	0.687312	0.531083
2	0.434768	0.366180	0.851823
3	0.186435	0.128874	0.955214
4	0.120681	0.072901	0.980228
5	0.099568	0.107304	0.962564
6	0.071958	0.078094	0.976239

完成 GAN [1:09:52]

```
1 | learn_crit=None  
2 | learn_gen=None  
3 | gc.collect()
```

```
1 | data_crit = get_crit_data(['crappy', 'images'], bs=bs, size=size)
```

```
1 | learn_crit = create_critic_learner(data_crit, metrics=None).load('critic-pre2')
```

```
1 | learn_gen = create_gen_learner().load('gen-pre2')
```

我们来完成这个GAN。现在我有了一个预训练的generator，和预训练的critic，我们要在这两个之间来回训练。在每边的训练的次数和学习率还不清楚。我们创建了一个 `GANLearner`，你可以直接传入你的generator和critic就行了，当你运行 `learn.fit` 时，它会找出要训练generator多少次、什么时候切换到训练discriminator/critic，自动为你切换。

```
1 | switcher = partial(AdaptiveGANSwitcher, critic_thresh=0.65)  
2 | learn = GANLearner.from_learners(learn_gen, learn_crit, weights_gen=(1., 50.),  
3 | show_img=False, switcher=switcher,  
3 | opt_func=partial(optim.Adam, betas=  
3 | (0., 0.99)), wd=wd)  
4 | learn.callback_fns.append(partial(GANDiscriminativeLR, mult_lr=5.))
```

[1:10:43]

关于这里的权重 (`weights_gen=(1., 50.)`)。我们不仅用critic做损失函数。如果我们只用critic做损失函数，GAN会很擅长创建看起来真实的图片，但它不会对原始图片做什么操作。我们把像素损失和critic损失加到一起，这两个损失有不同的比例，我们把像素损失乘上一个50到200的数，这样一个数通常是有效的。

另外，**GAN不用动量**。当你训练它们时，用动量训练没什么意义，因为你不停在generator和critic间切换，很难用上动量。可能有使用动量的方式，但没见过有人这样做过。所以，当你创建Adam优化器时，这个动量的参数 (`betas=(0., ...)`)，你要设成0。

如果你在做GAN，用这些超参数。

```
1 | GANLearner.from_learners(learn_gen, learn_crit, weights_gen=(1., 50.),
2 |     show_img=False,
3 |     switcher=switcher, opt_func=partial(optim.Adam,
|     betas=(0., 0.99)),
|     wd=wd)
```

它们是有效的。这是GANLearner做的，你可以执行fit，它会训练一会儿。

```
1 | lr = 1e-4
```

```
1 | learn.fit(40, lr)
```

Total time: 1:05:41

epoch	train_loss	gen_loss	disc_loss
1	2.071352	2.025429	4.047686
2	1.996251	1.850199	3.652173
3	2.001999	2.035176	3.612669
4	1.921844	1.931835	3.600355
5	1.987216	1.961323	3.606629
6	2.022372	2.102732	3.609494
7	1.900056	2.059208	3.581742
8	1.942305	1.965547	3.538015
9	1.954079	2.006257	3.593008
10	1.984677	1.771790	3.617556
11	2.040979	2.079904	3.575464
12	2.009052	1.739175	3.626755
13	2.014115	1.204614	3.582353
14	2.042148	1.747239	3.608723
15	2.113957	1.831483	3.684338
16	1.979398	1.923163	3.600483
17	1.996756	1.760739	3.635300
18	1.976695	1.982629	3.575843
19	2.088960	1.822936	3.617471
20	1.949941	1.996513	3.594223
21	2.079416	1.918284	3.588732
22	2.055047	1.869254	3.602390
23	1.860164	1.917518	3.557776
24	1.945440	2.033273	3.535242
25	2.026493	1.804196	3.558001
26	1.875208	1.797288	3.511697
27	1.972286	1.798044	3.570746
28	1.950635	1.951106	3.525849
29	2.013820	1.937439	3.592216
30	1.959477	1.959566	3.561970

epoch	train_loss	gen_loss	disc_loss
31	2.012466	2.110288	3.539897
32	1.982466	1.905378	3.559940
33	1.957023	2.207354	3.540873
34	2.049188	1.942845	3.638360
35	1.913136	1.891638	3.581291
36	2.037127	1.808180	3.572567
37	2.006383	2.048738	3.553226
38	2.000312	1.657985	3.594805
39	1.973937	1.891186	3.533843
40	2.002513	1.853988	3.554688

GAN里一个困难的地方是这些损失度没有意义。你不能期望随着generator变好这些值会下降，随着generator变好，对critic来说，任务越来越难，随着critic变好，对generator来说越来越难。这些值会保持不变。很难知道它们做得怎么样，这是训练GAN时一个困难的地方。要知道它们做得怎么样的唯一的方式是时不时地看看这些结果。如果我把 `show_img=True` 写在这里：

```

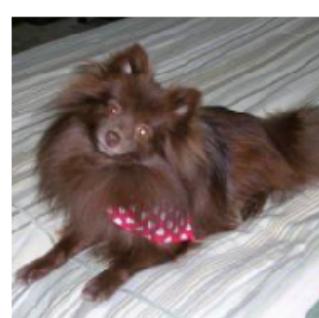
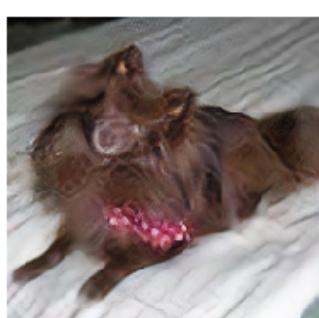
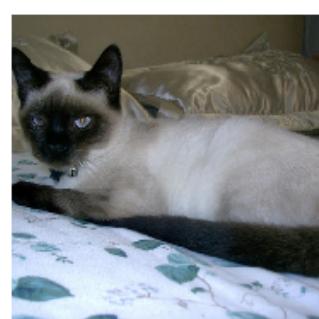
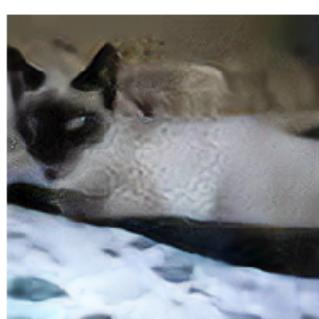
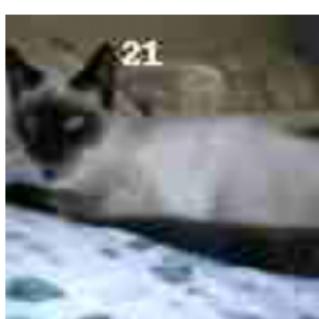
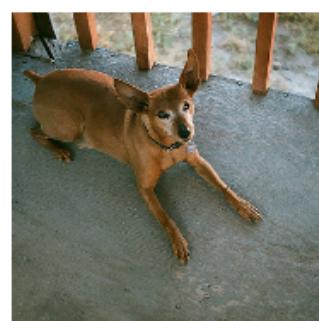
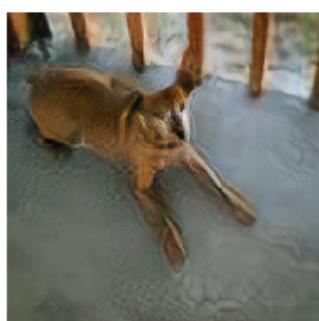
1 | GANLearner.from_learners(learn_gen, learn_crit, weights_gen=(1., 50.),
2 |                         show_img=False,
3 |                         switcher=switcher, opt_func=partial(optim.Adam,
4 |                         betas=(0., 0.99)),
5 |                         wd=wd)

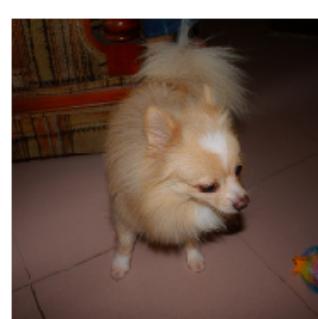
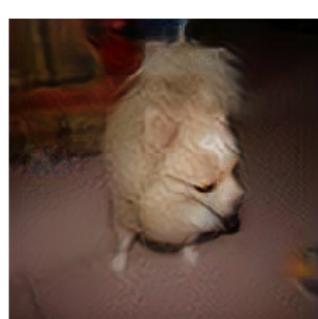
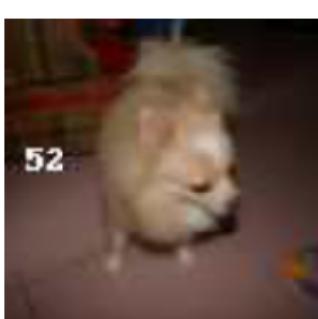
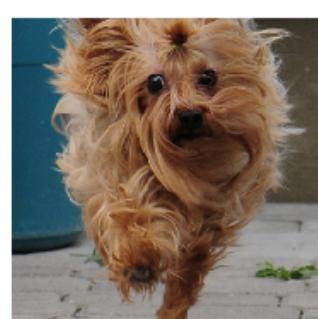
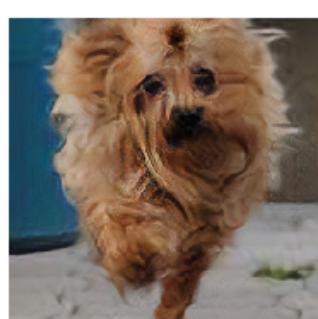
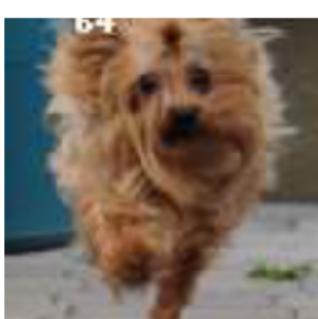
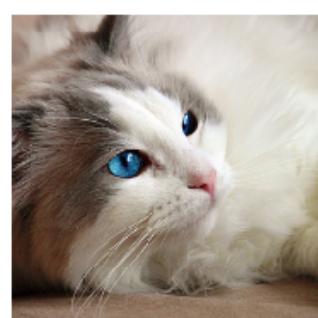
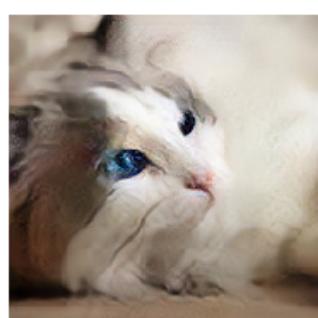
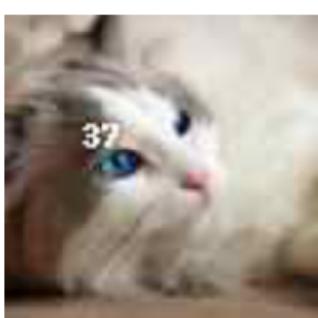
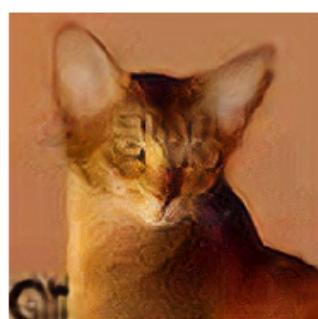
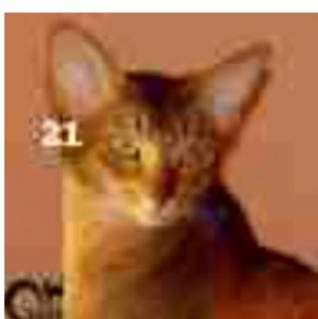
```

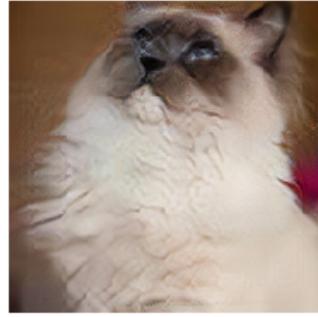
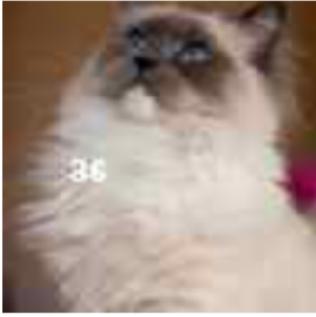
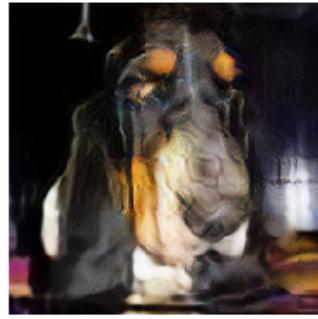
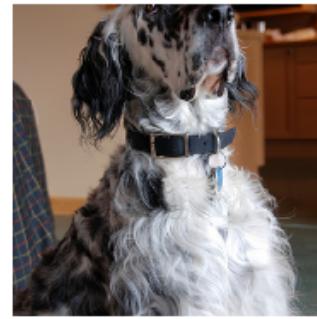
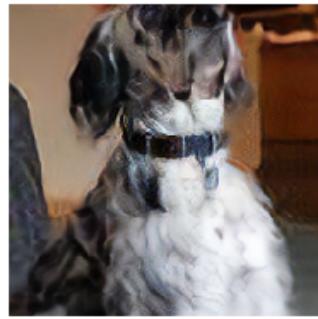
它会在每个epoch后打印出一个样本。我们没有把它放到notebook里，这对repo来说太大了，但是你可以试试。我把结果放到了最后，就是这里。

```
1 | learn.show_results(rows=16)
```

Input / Prediction / Target







很漂亮。我们已经知道怎样去掉这些数字，现在已经没有这些手动添加地东西，它确实把这只小猫画得很好。它不是总是很棒，这里有一些奇怪的噪声。确实比原来的糟糕的图片好多了。要把它变成这样是一个困难的任务。但有一些明显的问题。像这里（第三行），这里应该是眼球，但它们没有。为什么？我们的critic不知道眼球。即使它们知道，它也不知道眼球非常重要。我们关心眼睛。当我们看到一

个没有眼睛的猫，它一点也不可爱。它不知道这是一个重要的特征。**部分原因是因为，这个critic不是一个预训练模型。**我怀疑，如果我们用一个在ImageNet上预训练过的、适用于GAN的预训练模型替代它，它会做得更好。这肯定是一个快速的方法。课间休息后，我会演示怎样找到猫的眼球。

提问：对什么样的问题，你不使用U-Net？ [1:14:48]

U-Net用于你的输出的大小和输入的大小接近的时候。如果这样的分辨率对输出是不必要的、没有用处的，那就不需要用cross connection。是的，各种生成模型。分割也是一种生成模型。它生成了原始图片的标记遮罩图片。所以，大概所有你希望输出的分辨率和输入的分辨率一致的情况，都要用U-Net。显然，对分类器这样的东西，没什么用。在分类器里，你只需要下采样过程，因为你最后只需要一个数字，来表示它是一只狗、还是一只猫、还是什么种类的宠物。

Wasserstein GAN [1:15:59]

在结束GAN之前，讲下这里有一个你们可能有兴趣读的notebook [lesson7-wgan.ipynb](#)。几年前GAN刚出现时，人们一般用它凭空创建一些图片，我认为这没有什么用处。但是我猜这是一个好的研究练习。所以我们实现了这个[WGAN paper](#)，这是第一个能简单地完成足够的任务的模型。你可以看到怎样用fastai来做。

这很有意思，因为我们用的数据集是LSUN bedrooms，我们在URL里提供了这个数据集，里面只有卧室，很多很多卧室。这个任务里，我们用的方法是，只是说“我们能创建一个卧室吗？”。generator的输入不是一个我们清理过的图片。我们给generator输入了随机噪音。然后generator的任务是“能不能把随机噪音变成critic不能把它和真正的卧室分辨出来的东西？”。我们没有做预训练，也没有做其它能让它变快变好的优化。这是一个很传统的方法。但是你看，你还是只运行 `GANLearner`，这里有一个 `wgan` 版本，这是这个老式的方法。你只要像以前一样传入data、generator、critic，你就可以调用 `fit`。

你会看到（在这个例子里，我们使用了 `show_image`），在一个epoch后，它没有创建出很好的卧室，第二个、第三个也是一样。在早期的这种GAN里，你确实可以看到这种情况，它没有做出什么很好的东西。但最终，在几个小时的训练之后，它生成出了一些多多少少像卧室一样的东西。不管怎么说，这是一个你可以尝试下的notebook，有点有趣。

Feature Loss [1:18:37]

上周，我们把fastai做成这样时，我非常兴奋。我们让GAN用聪明的API运行了起来，这些API远比其它的更简练、更灵活。也有点失望，这花了很长时间训练，输出的结果还是一般，下一步是“**我们可以完全舍弃GAN吗？**”。显然，**我们想找出一个更好的损失函数。**我们需要一个可以很好地判断这不是一个高质量图片、并且没有GAN那么麻烦的损失函数。可能它不会只是说这是一个高质量的图片，但它是一个看起来就是我们想要的东西的图片。这篇一两年前的论文里有这样一个方法，[Perceptual Losses for Real-Time Style Transfer and Super-Resolution](#)，Justin Johnson et al. 创造了这个他们叫 **perceptual loss** 的东西。这是一篇好论文，但我不喜欢这个名字，因为没有什么有知觉的（perceptual）的东西。我会叫它“feature losses”，所以在fastai库里，你可以看到这用feature loss来表示。

Perceptual Losses for Real-Time Style Transfer and Super-Resolution

Justin Johnson, Alexandre Alahi, Li Fei-Fei

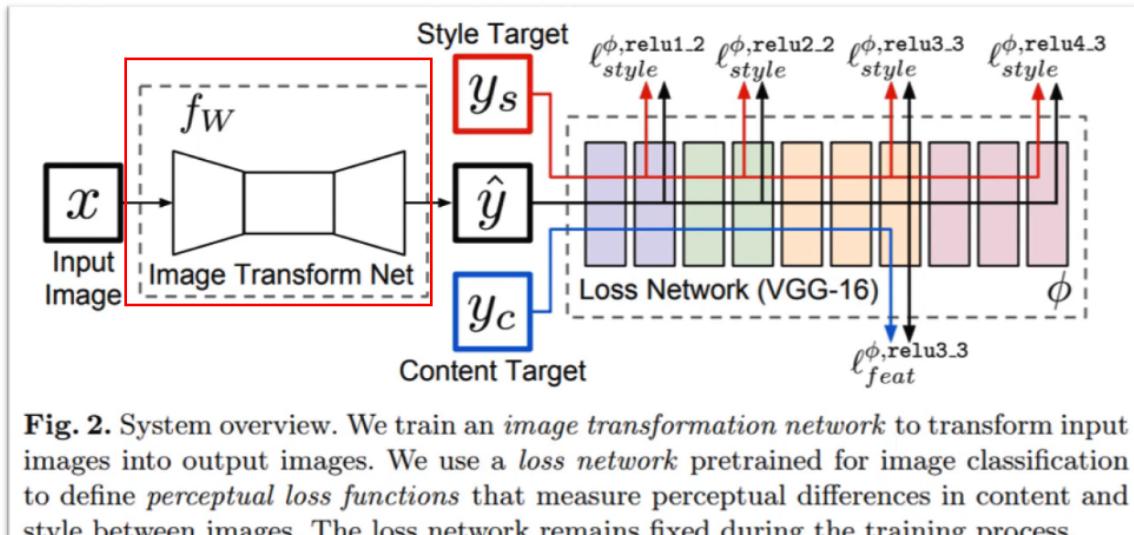


Fig. 2. System overview. We train an *image transformation network* to transform input images into output images. We use a *loss network* pretrained for image classification to define *perceptual loss functions* that measure perceptual differences in content and style between images. The loss network remains fixed during the training process.

它里面一些东西是和GAN一样的，在经过generator后（他们把generator叫“image transform net”），你可以看到它得到了这个形状像U-Net一样的东西。他们并没有用U-Net，因为在写这个论文时，研究机器学习的人里没人知道U-Net。当然，现在我们会用U-Net。不管怎样，这是一个和U-Net比较像的东西。

在这种架构里，你有一个下采样的过程，然后有一个上采样的过程，这个下采样的过程经常被叫做**encoder**，就像你在代码里见过的那样，上采样过程经常被叫做**decoder**。在生成模型里，一般包括文本生成模型、神经翻译和类似的东西，他们会被叫成**encoder**和**decoder**。

我们有了这个generator，我们需要一个损失函数来判断“它创建的这个东西和我想要的东西长得像吗？”。它们做出预测，记住我们用 \hat{y} 来表示模型的预测值。我们拿到这个预测值，把它放到一个预训练ImageNet网络里。在写这个论文时，他们用的预训练ImageNet网络是VGG。现在它过时了，但人们还是会用它，因为它对这个过程很有效。它们做出预测，把它放进VGG这个预训练ImageNet网络。用哪个预训练网络影响不大。

一般它的输出会告诉你“嘿，这个生成的东西是不是一条狗、一只猫、一架飞机、消防车还是什么东西？”。在得到最终结果的过程中，它经过很多不同的层。这里，他们为有相同尺寸和特征图（feature map）的层标记了相同的颜色。每次改变颜色时，尺寸就是不同的。这里有一个步长是2的卷积，在VGG里他们用的是一些maxpooling层，作用是一样的。

我们可以不取VGG模型的最终输出，而是取中间的东西。取中间一些层的激活值。这些激活值，可能是一个256通道的28x28的特征图。这些28x28的网格大概是用来判断“在这个28x28的网格里，有没有什么毛茸茸的东西？有没有什么发光的东西？有没有什么圆形的东西？有没有什么像眼球的东西？”

然后，我们取目标值（实际的 y 值），把它放进同一个预训练VGG网络里，我们取出相同的层的激活值，然后我们做一个均方差对比。它会说“在这个真正的图片里，这个28x28的特征图里的（1, 1）单元格是毛茸茸的、蓝色的、圆形的，在这个生成的图片里，它是毛茸茸的、蓝色的、不是圆形的。”这是一个好的匹配。

这应该能解决我们的眼球问题，因为这里，这个特征图会说“这里有眼球（目标图片），但这里没有（生成图片），所以继续努力，做出一个好点的眼球。”这就是它的思路。这就是feature loss，或者Johnson et al.叫的Perceptual loss。

要做这个，我们要用[lesson7-superres.ipynb](#)，这次我们要做的任务和之前的一样，但这个notebook是在做GAN notebook之前做的，当时还没有添加文字在上面，没有用随机的JPEG画质，画质一直是60，所以上面没有写文字，它是96x96的。当时还没有想到用“crappify”这个名字，所以它叫`resize_one`。

```
1 import fastai  
2 from fastai.vision import *  
3 from fastai.callbacks import *  
4  
5 from torchvision.models import vgg16_bn
```

```
1 path = untar_data(URLs.PETS)  
2 path_hr = path/'images'  
3 path_lr = path/'small-96'  
4 path_mr = path/'small-256'
```

```
1 il = ImageItemList.from_folder(path_hr)
```

```
1 def resize_one(fn,i):  
2     dest = path_lr/fn.relative_to(path_hr)  
3     dest.parent.mkdir(parents=True, exist_ok=True)  
4     img = PIL.Image.open(fn)  
5     targ_sz = resize_to(img, 96, use_min=True)  
6     img = img.resize(targ_sz, resample=PIL.Image.BILINEAR).convert('RGB')  
7     img.save(dest, quality=60)
```

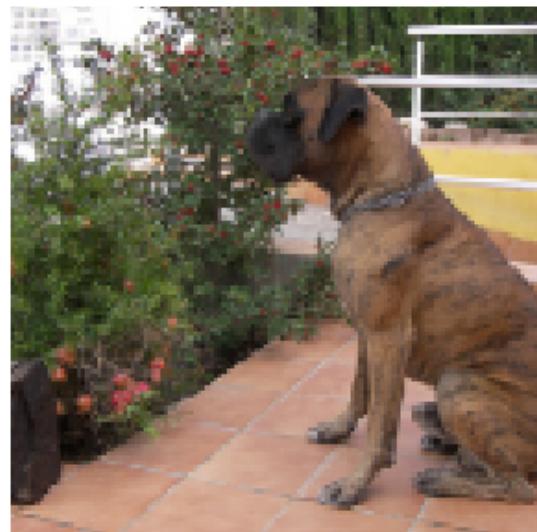
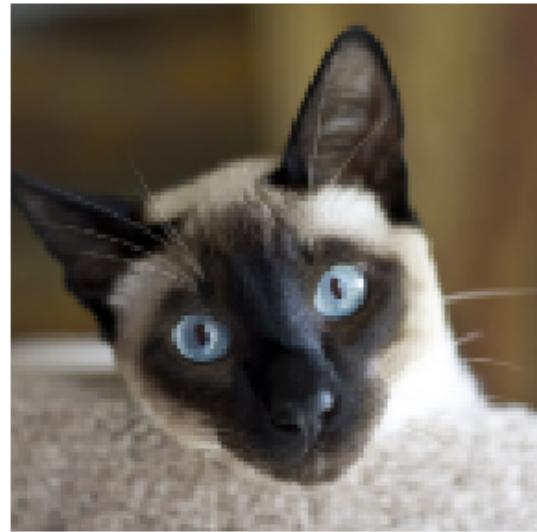
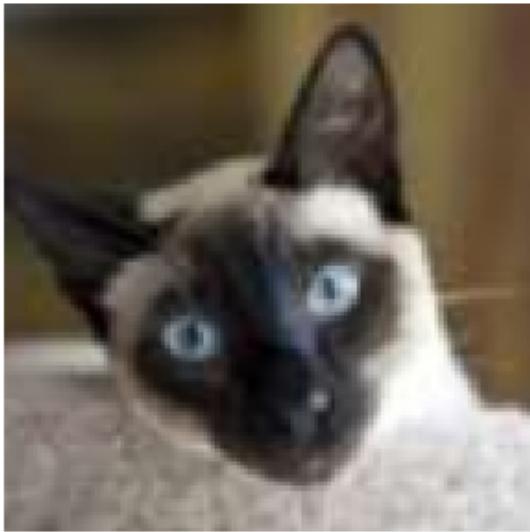
```
1 # to create smaller images, uncomment the next line when you run this the  
first time  
2 # parallel(resize_one, il.items)
```

```
1 bs, size=32, 128  
2 arch = models.resnet34  
3  
4 src = ImageImageList.from_folder(path_lr).random_split_by_pct(0.1, seed=42)
```

```
1 def get_data(bs, size):  
2     data = (src.label_from_func(lambda x: path_hr/x.name)  
3             .transform(get_transforms(max_zoom=2.), size=size, tfm_y=True)  
4             .databunch(bs=bs).normalize(imagenet_stats, do_y=True))  
5  
6     data.c = 3  
7     return data
```

```
1 data = get_data(bs, size)
```

```
1 data.show_batch(ds_type=DatasetType.valid, rows=2, figsize=(9,9))
```



这是我们的比较差的图片和原始的图片，这是一个和之前类似的任务。我创建一个损失函数来做这个 (feature loss)。首先定义一个基本的损失函数“我要怎样比较这些像素和这些特征？”，答案基本上是MSE或者L1。选哪个没什么关系。我更喜欢L1，所以我用L1。

```
1 | t = data.valid_ds[0][1].data  
2 | t = torch.stack([t,t])
```

```
1 | def gram_matrix(x):  
2 |     n,c,h,w = x.size()  
3 |     x = x.view(n, c, -1)  
4 |     return (x @ x.transpose(1,2))/(c*h*w)
```

```
1 | gram_matrix(t)
```

```
1 | tensor([[ [0.0759, 0.0711, 0.0643],  
2 |             [0.0711, 0.0672, 0.0614],  
3 |             [0.0643, 0.0614, 0.0573]],  
4 |  
5 |             [[0.0759, 0.0711, 0.0643],  
6 |             [0.0711, 0.0672, 0.0614],  
7 |             [0.0643, 0.0614, 0.0573]]])
```

```
1 | base_loss = F.l1_loss
```

所以每次你看到 `base_loss` 时，就是指L1 loss。你也可以用MSE。

```
1 | vgg_m = vgg16_bn(True).features.cuda().eval()  
2 | requires_grad(vgg_m, False)
```

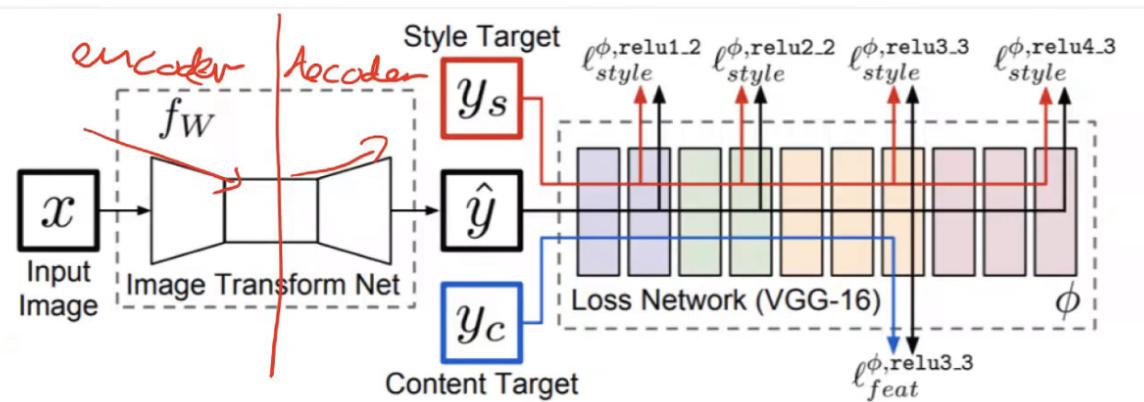
我们来用预训练模型创建一个VGG。在VGG里，有一个叫 `.feature` 的属性，它包含了模型的卷积部分。所以 `vgg16_bn(True).features` 是VGG模型的卷积部分。因为我们不需要前面的部分，我们只需要中间的激活值。

然后，我们会把它放到GPU，把它变成 `eval` 模式，因为我们不是在训练它。我们关掉 `requires_grad`，因为我们不想更新模型的权重。我们只想用它做推理（来得到损失度）。

```
1 | blocks = [i-1 for i,o in enumerate(children(vgg_m)) if  
2 |     isinstance(o,nn.MaxPool2d)]  
2 | blocks, [vgg_m[i] for i in blocks]
```

```
1 | ([5, 12, 22, 32, 42],  
2 | [ReLU(inplace), ReLU(inplace), ReLU(inplace), ReLU(inplace), ReLU(inplace)])
```

然后我们遍历模型的所有children，找出池化层，因为在VGG里这是改变网格尺寸的地方。就像可以在这个图片里看到的，我们需要找出每次网格尺寸变化时的特征：



我们取第 $i-1$ 层。这是改变前的层。这是池化层前的层的索引list (`[5, 12, 22, 32, 42]`)。所有的层都是ReLU。这是我们取特征的地方，我们把它放进 `blocks` 里，这是一个ID的list。

```
1 | class FeatureLoss(nn.Module):  
2 |     def __init__(self, m_feat, layer_ids, layer_wgts):  
3 |         super().__init__()  
4 |         self.m_feat = m_feat  
5 |         self.loss_features = [self.m_feat[i] for i in layer_ids]  
6 |         self.hooks = hook_outputs(self.loss_features, detach=False)  
7 |         self.wgts = layer_wgts  
8 |         self.metric_names = ['pixel',] + [f'feat_{i}' for i in  
range(len(layer_ids))  
9 |             ] + [f'gram_{i}' for i in range(len(layer_ids))]  
10 |  
11 |     def make_features(self, x, clone=False):  
12 |         self.m_feat(x)  
13 |         return [(o.clone() if clone else o) for o in self.hooks.stored]  
14 |
```

```

15     def forward(self, input, target):
16         out_feat = self.make_features(target, clone=True)
17         in_feat = self.make_features(input)
18         self.feat_losses = [base_loss(input, target)]
19         self.feat_losses += [base_loss(f_in, f_out)*w
20                             for f_in, f_out, w in zip(in_feat, out_feat,
21                                         self.wgts)]
21         self.feat_losses += [base_loss(gram_matrix(f_in),
22                                       gram_matrix(f_out))*w**2 * 5e3
23                             for f_in, f_out, w in zip(in_feat, out_feat,
24                                         self.wgts)]
24         self.metrics = dict(zip(self.metric_names, self.feat_losses))
25         return sum(self.feat_losses)
26
27     def __del__(self): self.hooks.remove()

```

这是我们的feature loss类，里面实现了这个方法（perceptual loss）。

```
1 | feat_loss = FeatureLoss(vgg_m, blocks[2:5], [5,15,2])
```

当我们调用feature loss类时，我们会传入一些预训练模型，模型的名字是`m_feat`。这是包含特征的模型，我们想用feature loss处理它。我们取这个网络的所有层，用里面的特征创建loss。

我们要hook所有这些输出，在PyTorch取中间层的方法就是hook它们。`self.hook`会存我们勾住的输出。

现在，在feature loss的`forward`里，我们要调用`make_features`，传入`traget`（实际的`y`），它会调用VGG模型，遍历所有存储的激活值，取出它们的值。我们对目标`out_feat`和输入（generator的输出，`in_feat`）都做同样的操作。现在，我们来计算像素的L1损失度。我们遍历所有层的特征，得到它们的L1损失度。我们遍历每个block的最后一层，取出激活值，算出L1。

最后放到叫`feat_losses`的list里，把它们加起来。我用list的原因是，我们有这个回调，如果你在损失度函数里把它们放进这个叫`.metrics`的东西里，它会打印所有层的损失，很方便。

就是这样，这就是我们的perceptual loss或者说feature loss类。

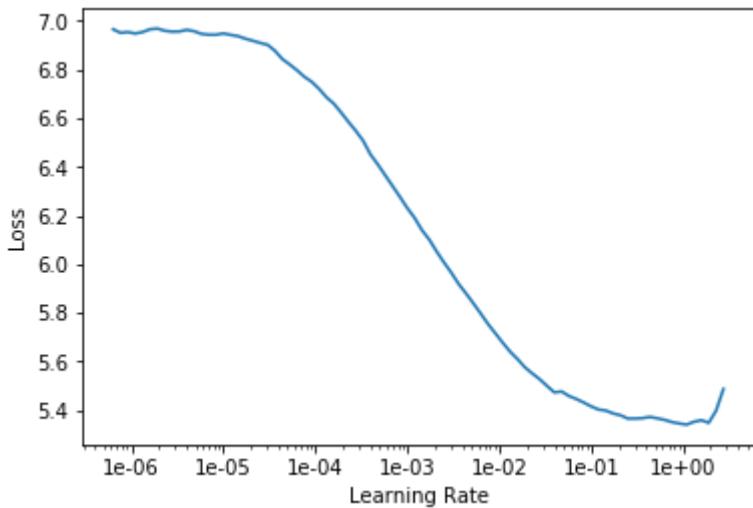
```

1 | wd = 1e-3
2 | learn = unet_learner(data, arch, wd=wd, loss_func=feat_loss,
3 |                         callback_fns=LossMetrics,
4 |                         blur=True, norm_type=NormType.Weight)
5 | gc.collect();

```

现在我们可以继续像往常一样训练一个U-Net，使用我们的数据，使用一个预训练ResNet34模型，传入损失函数，这个损失函数使用我们的预训练VGG模型。这个`callback_fns`是我提过的`LossMetrics`，它可以打印出所有层的损失。这（`blur`、`norm_type`）是两个我们在课程第二部分学习的东西，但现在我们要使用它。

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



```
1 | lr = 1e-3
```

```
1 | def do_fit(save_name, lrs=slice(lr), pct_start=0.9):
2 |     learn.fit_one_cycle(10, lrs, pct_start=pct_start)
3 |     learn.save(save_name)
4 |     learn.show_results(rows=1, imgsize=5)
```

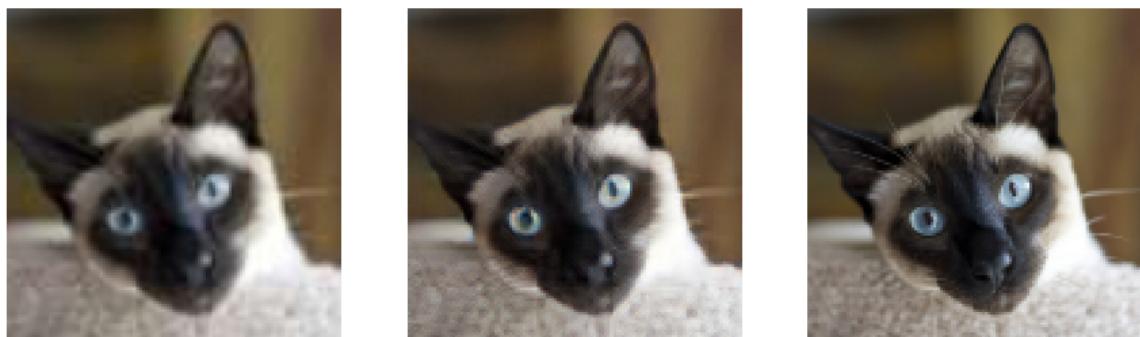
我创建了一个叫 `do_fit` 的函数，它执行 `fit_one_cycle`，然后保存模型，显示结果。

```
1 | do_fit('1a', slice(lr*10))
```

Total time: 11:16

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2	gram_0	gram_1	gram_2
1	3.873667	3.759143	0.144560	0.229806	0.314573	0.226204	0.552578	1.201812	1.089610
2	3.756051	3.650393	0.145068	0.228509	0.308807	0.218000	0.534508	1.164112	1.051389
3	3.688726	3.628370	0.157359	0.226753	0.304955	0.215417	0.522482	1.157941	1.043464
4	3.628276	3.524132	0.145285	0.225455	0.300169	0.211110	0.497361	1.124274	1.020478
5	3.586930	3.422895	0.145161	0.224946	0.294471	0.205117	0.472445	1.089540	0.991215
6	3.528042	3.394804	0.142262	0.220709	0.289961	0.201980	0.478097	1.083557	0.978238
7	3.522416	3.361185	0.139654	0.220379	0.288046	0.200114	0.471151	1.069787	0.972054
8	3.469142	3.338554	0.142112	0.219271	0.287442	0.199255	0.462878	1.059909	0.967688
9	3.418641	3.318710	0.146493	0.219915	0.284979	0.197340	0.455503	1.055662	0.958817
10	3.356641	3.187186	0.135588	0.215685	0.277398	0.189562	0.432491	1.018626	0.917836

Input / Prediction / Target



像之前一样，因为我们在U-Net里用了一个预训练网络，我们开始时用冻结的层做下采样，训练一阵。你可以看到，我们得到了损失度，还有像素损失度和每个特征层的损失度，还有Grom损失，这是我们要在课程第二部分学习的一个东西，我觉得目前没有人用过它来实现高分辨率。可以看到，结果很好。

用了八分钟，比GAN快得多，并且，你看这个输出，效果已经很好了。然后，我们解冻它，再训练一会儿。

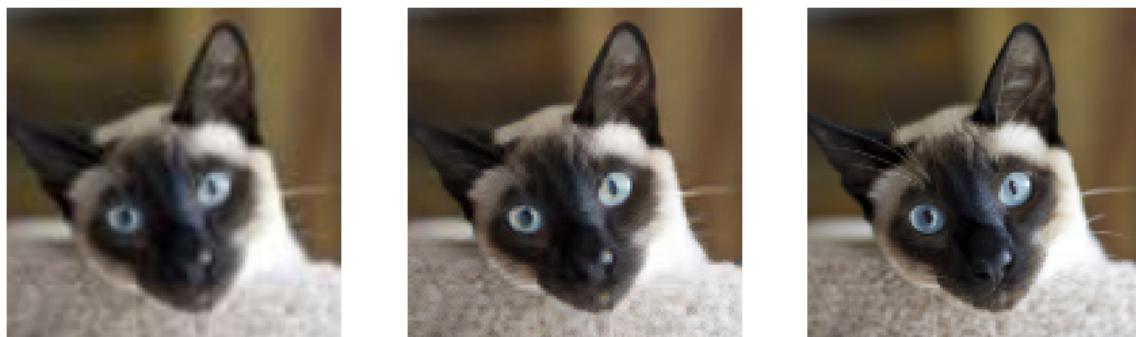
```
1 | learn.unfreeze()
```

```
1 | do_fit('1b', slice(1e-5, 1r))
```

Total time: 11:39

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2	gram_0	gram_1	gram_2
1	3.303951	3.179916	0.135630	0.216009	0.277359	0.189097	0.430012	1.016279	0.915531
2	3.308164	3.174482	0.135740	0.215970	0.277178	0.188737	0.428630	1.015094	0.913132
3	3.294504	3.169184	0.135216	0.215401	0.276744	0.188395	0.428544	1.013393	0.911491
4	3.282376	3.160698	0.134830	0.215049	0.275767	0.187716	0.427314	1.010877	0.909144
5	3.301212	3.168623	0.135134	0.215388	0.276196	0.188382	0.427277	1.013294	0.912951
6	3.299340	3.159537	0.135039	0.214692	0.275285	0.187554	0.427840	1.011199	0.907929
7	3.291041	3.159207	0.134602	0.214618	0.275053	0.187660	0.428083	1.011112	0.908080
8	3.285271	3.147745	0.134923	0.214514	0.274702	0.187147	0.423032	1.007289	0.906138
9	3.279353	3.138624	0.136035	0.213191	0.273899	0.186854	0.420070	1.002823	0.905753
10	3.261495	3.124737	0.135016	0.213681	0.273402	0.185922	0.416460	0.999504	0.900752

Input / Prediction / Target



这次又好了一点儿。然后，我们把尺寸加倍。我们要把批次的大小减半，避免GPU内存溢出，再解冻，训练一会儿。

```
1 | data = get_data(12, size*2)
```

```
1 | learn.data = data
2 | learn.freeze()
3 | gc.collect()
```

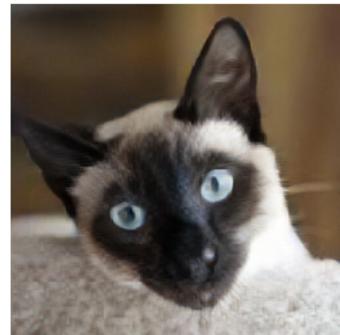
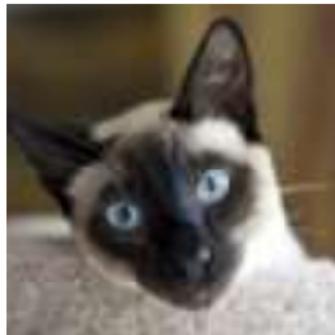
```
1 | learn.load('1b');
```

```
1 | do_fit('2a')
```

Total time: 43:44

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2	gram_0	gram_1	gram_2
1	2.249253	2.214517	0.164514	0.260366	0.294164	0.155227	0.385168	0.579109	0.375967
2	2.205854	2.194439	0.165290	0.260485	0.293195	0.154746	0.374004	0.573164	0.373555
3	2.184805	2.165699	0.165945	0.260999	0.291515	0.153438	0.361207	0.562997	0.369598
4	2.145655	2.159977	0.167295	0.260605	0.290226	0.152415	0.359476	0.563301	0.366659
5	2.141847	2.134954	0.168590	0.260219	0.288206	0.151237	0.348900	0.554701	0.363101
6	2.145108	2.128984	0.164906	0.259023	0.286386	0.150245	0.352594	0.555004	0.360826
7	2.115003	2.125632	0.169696	0.259949	0.286435	0.150898	0.344849	0.552517	0.361287
8	2.109859	2.111335	0.166503	0.258512	0.283750	0.148191	0.347635	0.549907	0.356835
9	2.092685	2.097898	0.169842	0.259169	0.284757	0.148156	0.333462	0.546337	0.356175
10	2.061421	2.080940	0.167636	0.257998	0.282682	0.147471	0.330893	0.540319	0.353941

Input / Prediction / Target



这次用了半个小时，结果更好了。然后解冻，再训练。

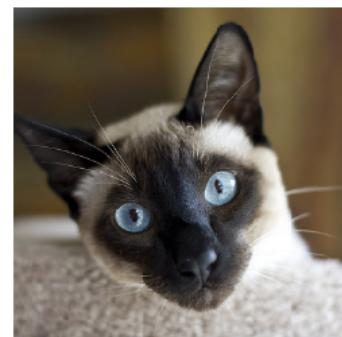
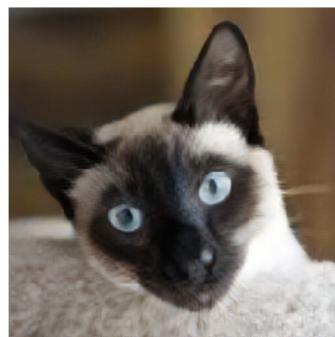
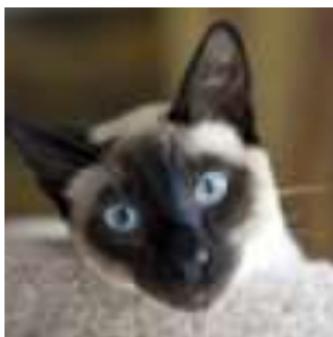
```
1 | learn.unfreeze()
```

```
1 | do_fit('2b', slice(1e-6, 1e-4), pct_start=0.3)
```

Total time: 45:19

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2	gram_0	gram_1	gram_2
1	2.061799	2.078714	0.167578	0.257674	0.282523	0.147208	0.330824	0.539797	0.353109
2	2.063589	2.077507	0.167022	0.257501	0.282275	0.146879	0.331494	0.539560	0.352776
3	2.057191	2.074605	0.167656	0.257041	0.282204	0.146925	0.330117	0.538417	0.352247
4	2.050781	2.073395	0.166610	0.256625	0.281680	0.146585	0.331580	0.538651	0.351665
5	2.054705	2.068747	0.167527	0.257295	0.281612	0.146392	0.327932	0.536814	0.351174
6	2.052745	2.067573	0.167166	0.256741	0.281354	0.146101	0.328510	0.537147	0.350554
7	2.051863	2.067076	0.167222	0.257276	0.281607	0.146188	0.327575	0.536701	0.350506
8	2.046788	2.064326	0.167110	0.257002	0.281313	0.146055	0.326947	0.535760	0.350139
9	2.054460	2.065581	0.167222	0.257077	0.281246	0.146016	0.327586	0.536377	0.350057
10	2.052605	2.064459	0.166879	0.256835	0.281252	0.146135	0.327505	0.535734	0.350118

Input / Prediction / Target



我们总共用了一个小时20分钟来训练，看看这个。它做到了。**它知道眼睛很重要，它把眼睛画出来了。它知道皮毛很重要，也把它画出来了。**开始时，在耳朵这里有些失真，眼睛这里只是浅蓝色的东西。它真的创建出了很多纹理，这只猫在一个盖着毛毯的架子上，画得很清楚，它真的认出了这个可能是一个毛毯之类的东西。效果很明显。

说很明显，我是指在之前没有用GAN时，我没有见过这样的输出，当我们能生成这个时，我很兴奋，我们训练得很快，只在一个GPU上用了一个半小时。如果你创建你自己的把图片变差的函数（crappification functions），训练这个模型，你可以构建出没有人做过的东西。我不知道有其它人这样做过。所以，我觉得有巨大的机会。所以，要尝试下。

中等分辨率 (Medium Resolution) [1:31:45]

这就是我们现在能做的。**前面用的是低分辨率的图片，我还保存了分辨率是256的图片，我们把它叫medium res，我们看下，如果我们用medium res会发生什么。**

```
1 | data_mr = (ImageImageList.from_folder(path_mr).random_split_by_pct(0.1,
2 |     seed=42)
3 |         .label_from_func(lambda x: path_hr/x.name)
4 |         .transform(get_transforms(), size=(1280,1600), tfm_y=True)
5 |         .databunch(bs=1).normalize(imagenet_stats, do_y=True))
6 | data_mr.c = 3
```

```
1 | learn.data = data_mr
```

```
1 | fn = data_mr.valid_ds.x.items[0]; fn
```

```
1 | PosixPath('/data1/jhoward/git/course-v3/nbs/dl1/data/oxford-iiit-pet/small-
2 | 256/Siamese_178.jpg')
```

```
1 | img = open_image(fn); img.shape
```

```
1 | torch.size([3, 256, 320])
```

```
1 | p,img_hr,b = learn.predict(img)
```

```
1 | show_image(img, figsize=(18,15), interpolation='nearest');
```



我们取中等分辨率的数据，这是我们中等分辨率的图片。你可以提升它吗？可以看到有很多提升的空间。睫毛这里，很像素化。这个应该是毛发的地方有点毛茸茸的。看看下面这个地方。

```
1 | Image(img_hr).show(figsize=(18,15))
```



看，它做完了。它把一个中等分辨率的图片变成了完全清晰的东西。这些毛又出现了。但看看眼球，我们回去看下。这原来的眼球是蓝色的东西，这里（生成的图片里）添加了所有正确的纹理。我认为这很让人兴奋。这是我用一个半小时创建的一个模型，用的都是你们学过的U-Net里的标准的东西，预训练模型，feature loss函数，我们得到了可以把中等分辨率图片变成高分辨率图片的模型。想想能用它做什么真让人兴奋。

有个来自Jason Antic的项目。Jason是去年这个课程的一个学生，[他离职每周花4天或6天来专注学习深度学习](#)，你们也应该这样，他创建了一个顶级的项目，他的项目是把GAN和feature loss结合在一起。[他把图片变差的方法是，把彩色图片变成黑白的。](#)他处理了整个ImageNet，创建了一个黑白版的ImageNet，然后训练了一个模型来重新着色。[他把这个提交到DeOldify](#)，现在他可以把真正的19世纪的老照片变成彩色的。



他做的东西让人难以置信。看这个。这个模型认为，“噢，这可能是一个铜壶，我要把它变成铜的颜色”，“噢，[这些图片在墙上，它们可能和墙的颜色不一样](#)”，“这看起来像一个镜子，可能它可以反射出外面的东西”，“这些可能是蔬菜，让我把它们变成红色”。他做得令人惊奇。[你们也可以做这个。你们可以用我们的feature loss和我们的GAN loss，把它们结合起来。](#)非常感谢Jason，[他帮我们设计了这节课，我们也能帮助他，因为他没有意识到他可以用这些预训练的东西](#)。希望在几周之内，能看到DeOldify变得更擅长还原颜色。也希望你们都能添加其它的还原方法。

如果可能的话，我喜欢每节课都展示一些新东西，因为每个学生都有机会构建从来没有的东西。这就是这样一种东西。除了这些明显提升的分割结果和这些明显简化、加速的还原结果，我想你们能构建出一些很酷的东西。

提问: 能不能把U-Net和GAN类似的方法用到NLP上？比如，如果我想标记句子里的动词名词，或者创建一个很好的莎士比亚生成程序？ [1:35:54]

是的，当然可以。我们还不完全清楚。这是一个新领域，但有很多机会。稍后我们会讲一些。



Chris Gorgolewski

@ChrisFiloG

Follow



Did you know that Dropout was originally introduced in a Master's thesis and was rejected from NIPS? Was disseminated via #arxiv! #OHB2018

Dropout

Srivastava, Nitish, et al. "Dropout: A simple way to prevent neural networks from overfitting." *The Journal of Machine Learning Research* 15.1 (2014): 1929-1958.

- Srivastava's Master's(!) thesis.
- Training scheme that randomly masks neurons at every step.
- Usually gives a small performance boost.
- Mysterious.

This paper was rejected from NIPS in 2012, and propagated solely as a preprint on arxiv.

5:30 PM - 20 Jun 2018

我其实在尝试实验这个。记得这个我上节课给你们看过的幻灯片里的图吗，这是一个看起来质量很差的图。我想，如果用这个模型处理它会怎样。它把这个图（左边）变成右边这样，我觉得这是一个很好的例子。你可以看到它没有把颜色去掉了。我没有修复它，因为在把图片变差的方法（crappification functions）里我没有去掉颜色。如果你想创建一个很好的图片修复程序，你需要做个很好的把图片变差的方法。

Srivastava, Nitish, et al. "Dropout: A simple way to prevent neural networks from overfitting." *The Journal of Machine Learning Research* 15.1 (2014): 1929-1958.



- Srivastava's Master's(!) thesis.
- Training scheme that randomly masks neurons at every step
- Usually gives a small performance boost.
- Mysterious.

Srivastava, Nitish, et al. "Dropout: A simple way to prevent neural networks from overfitting." *The Journal of Machine Learning Research* 15.1 (2014): 1929-1958.

- Srivastava's Master's(!) thesis.
- Training scheme that randomly masks neurons at every step
- Usually gives a small performance boost.
- Mysterious.

目前为止我们在这个课里学到了什么 [1:36:59]

Affine functions & non-linearities	Parameters & activations	Random init & transfer learning	SGD; Momentum; Adam
Convolutions	Batch-norm	Dropout	Data augmentation
Weight decay	Res/dense blocks	Image classification and regression	Embeddings
Continuous & Categorical Variables	Collaborative filtering	Language models; NLP classification	Segmentation; U-net; GANs

这是目前为止我们在这个课程里学到的主要的东西。我们学习了包含有影射函数（就是矩阵相乘，更通用的版本）、像ReLU这样的非线性运算的神经网络。我们学习了这些计算的结果叫激活值，参与计算的这些东西叫参数。参数是随机初始化的，或者是从预训练模型里复制过来的，然后我们用SGD或更快的东西训练它们。我们学了卷积是一种影射函数，它对自相关数据（auto correlated data），比如图片之类的效果很好。学了权重衰减作为正则化的方法。batch norm让训练更快。

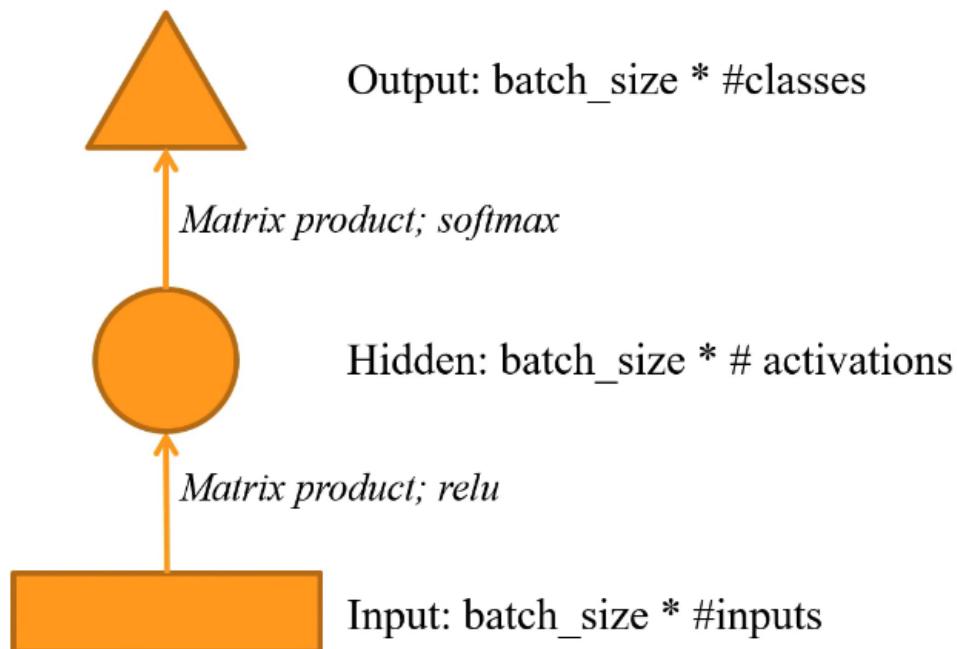
然后，今天，我们学了Res/Dense blocks。我们学了很多关于图片分类、回归、嵌入、类别和连续变量、协同过滤、语言模型、NLP分类的东西。还有分割、U-Net和GAN。

回顾这些东西，确保你对它们很熟悉。如果你只看了一遍这些视频，你肯定没有熟悉。人们一般要看三遍才能真正理解这些细节。

递归神经网络 Recurrent Neural Network (RNN) [1:38:31]

一个还没学的是RNN。这是最后一个要学的东西。我要用图示的方法来解释RNN。我先给你们看一个只有一个隐藏层的基本的神经网络。

Basic NN with single hidden layer



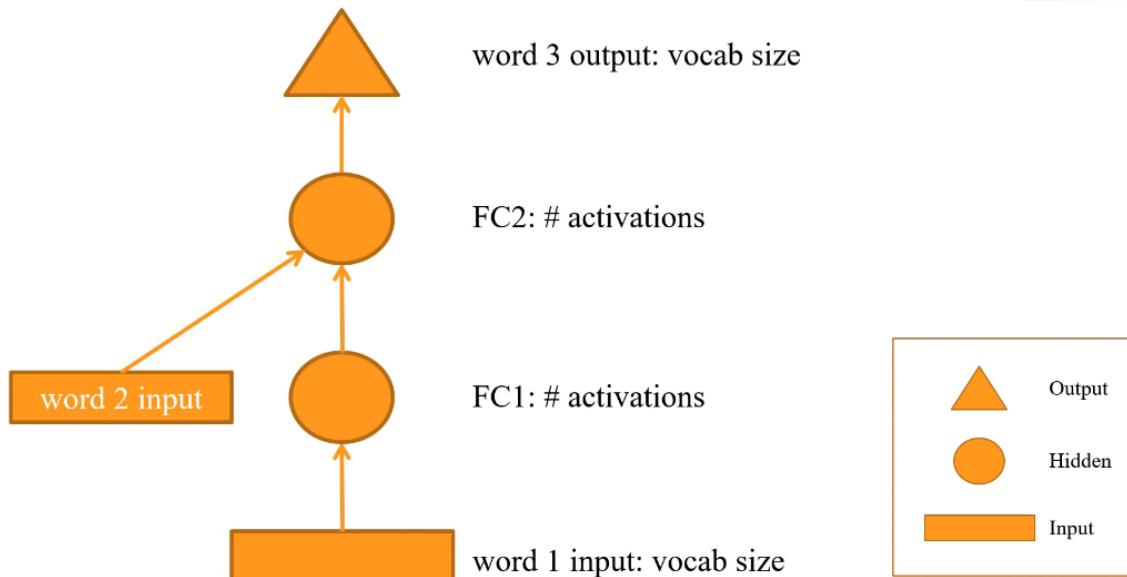
矩形代表一个输入，它的形状是批次数量 (batch_size) × 输入数量。箭头代表一个层，比如说一个矩阵乘积然后一个ReLU。一个圆圈是一个激活值。这里，我们有一组隐藏的激活值。这个（第一个箭头）是一个矩阵，尺寸是输入的数量×激活值的数量。输出的尺寸是批次数量×激活值数量。

知道怎样计算形状是很重要的。所以要多运行 `learn.summary()`，看看所有的形状。然后，这里有另一个箭头，这代表它是另外一个层，矩阵乘积然后做一个非线性计算。这里，下一层是输出层，所以我们用softmax。

三角形代表一个输出。这个矩阵形状是激活值数量×类别数量，所以我们的输出的尺寸是批次数量×类别数量。

Predicting word 3 using words 1 & 2

NB: layer operations
remember that arrows represent layers



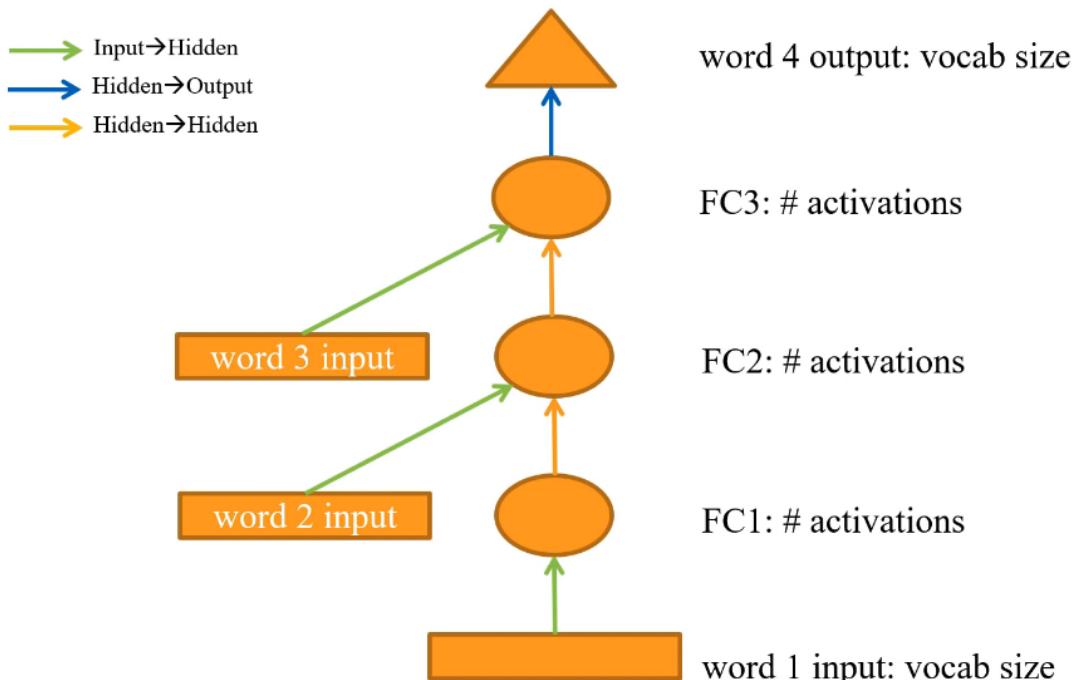
再用一次这些符号。三角形是输出，圆形是激活值（我们也叫它隐藏状态hidden state），矩形是输入。我们来想象一下，我们要把一个很大的文档拆分成3个单词的集合，对每个集合，用前两个单词，预测第三个单词。如果我们现在有这个数据集，我们可以：

1. 把第一个单词作为输入
2. 让它经过一个embedding，创建一些激活值
3. 让它经过一个矩阵乘法和非线性计算
4. 取第二个单词
5. 让它经过一个embedding
6. 然后我们可以把这两个东西加在一起，或者连接在一起（concatenate）。一般来说，你见到两组激活值在一个图里走到一起，你一般可以选择连接或者相加。这会创建第二组激活值。
7. 然后你可以让它经过下一个全连接层，和softmax来产生一个输出

这比一个完全标准的全连接的神经网络，多了一个小东西，就是在这里做了连接或者相加，我们要用这个网络来通过前两个单词预测第三个单词。

记住，箭头代表层操作，我在这里删除了细节。它们是一个仿射函数和一个非线性计算。

Predicting word 4 using words 1, 2 & 3



再多做些。如果我们要用前3个单词预测第4个单词，要怎样做呢？就是在上次的图片基础上，加一个额外的输入和一个额外的圆圈。但是我要说的一点是，每次我们从矩形走到圆形，我们在做相同的事情，我们在做embedding。就是一种特别的矩阵乘法，这里你有一个one-hot编码的输入。每次我们从圆形走到圆形，我们取到一个隐藏状态（激活值），通过加上另外一个单词，把它调整成另外一个激活值的集合。然后当我们从圆形走到三角形，我们是在把隐藏状态转成一个输出。我把这些箭头画成不同的颜色。每个箭头应该用相同的权重矩阵，因为它在做相同的事情。

Human numbers [1:43:11]

现在我们要进入human numbers，它在lesson7-human-numbers.ipynb里。这是一个我创建的数据集，里面包含了用英文写的从1到9,999的数字。

我们要创建一个可以预测这个文档里下一个单词的语言模型。这只是一个玩具示例。这里我们只有一个文档。文档里是数字的列表。所以我们可以用 `TextList` 来创建一个存文本的item list来训练。

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

```
1 | bs=64
```

```
1 | path = untar_data(URLs.HUMAN_NUMBERS)
2 | path.ls()
```

```
1 | [PosixPath('/data1/jhoward/git/course-
v3/nbs/dl1/data/human_numbers/valid.txt'),
2 | PosixPath('/data1/jhoward/git/course-
v3/nbs/dl1/data/human_numbers/train.txt'),
3 | PosixPath('/data1/jhoward/git/course-v3/nbs/dl1/data/human_numbers/models')]
```

```
1 | def readnums(d): return [', '.join(o.strip() for o in
open(path/d).readlines())]
```

```
1 | train_txt = readnums('train.txt'); train_txt[0][:80]
```

```
1 | 'one, two, three, four, five, six, seven, eight, nine, ten, eleven, twelve,
thirt'
```

```
1 | valid_txt = readnums('valid.txt'); valid_txt[0][-80:]
```

```
1 | ' nine thousand nine hundred ninety eight, nine thousand nine hundred ninety
nine'
```

```
1 | train = TextList(train_txt, path=path)
2 | valid = TextList(valid_txt, path=path)
3 |
4 | src = ItemLists(path=path, train=train, valid=valid).label_for_lm()
5 | data = src.databunch(bs=bs)
```

这里验证集是8,000之后的数字，训练集是1到8,000。我们可以把它们放在一起，转成一个data bunch。

```
1 | train[0].text[:80]
```

```
1 | 'xxbos one , two , three , four , five , six , seven , eight , nine , ten ,
eleve'
```

我们只有一个文档，`train[0]`里是文档，我们取它的`.text`，这是从一个text list里取内容的方法。这是它的前80个字符，它用一个特殊的token `xxbos` 开头。所有以 `xx` 开头的是特定的fastai token，`bos` 是stream token的起始 (beginning of stream token)。它基本上是说这是文档的起点，在NLP里，知道文档在哪开始很有用，这样你的模型可以学习识别它们。

```
1 | len(data.valid_ds[0][0].data)
```

```
1 | 13017
```

验证集有13,000个token。就是13,000个单词或者标点符号，所有空格间的东西都是一个token。

```
1 | data.bptt, len(data.valid_dl)
```

```
1 | (70, 3)
```

```
1 | 13017/70/bs
```

```
1 | 2.905580357142857
```

我们设的批次大小 (batch size) 是64，它用一个叫 `bptt` 的东西，默认值是70，`bptt`，我们简单提过，代表

“back prop through time”。这是序列的长度。对64个文档片段来说，我们把它分成有70个单词的list，我们每次都处理这样一个list。我们对验证集，取出整个的有13,000个token的字符串，然后我们把它分成64个长度相等的部分。人们经常把这个理解错。我不是说“它们的长度是64”，不是这样。它们是“64个长度基本相等的部分”。我们取这个文档的第一个1/64，然后第二个。

我们把它们中的每一个分成长度是70的片段。现在对这个13,000个token，有多少个批次？除以批次大小再除以70，会有3个批次。

```
1 | it = iter(data.valid_dl)
2 | x1,y1 = next(it)
3 | x2,y2 = next(it)
4 | x3,y3 = next(it)
5 | it.close()
```

```
1 | x1.numel() + x2.numel() + x3.numel()
```

```
1 | 12928
```

我们取一个data loader的迭代器，取前3个批次 (X和Y)，我们把元素数量加起来，我们得到的数据比13,017少一点，因为最后有一点不够组成一个批次了。你要尝试很多大小。

```
1 | x1.shape, y1.shape
```

```
1 | (torch.size([95, 64]), torch.size([95, 64]))
```

```
1 | x2.shape, y2.shape
```

```
1 | (torch.size([69, 64]), torch.size([69, 64]))
```

你可以看到，它是95x64的。我声明的是70x64。这是因为我们的语言模型data loader随机改变了 `bptt`，来让顺序随机改变 (shuffle)，有更多的随机性，它对模型有帮助。

```
1 | x1[:,0]
```

```
1 tensor([ 2, 18, 10, 11,  8, 18, 10, 12,  8, 18, 10, 13,  8, 18, 10, 14,  8,
2           18, 10, 15,  8, 18, 10, 16,  8, 18, 10, 17,  8, 18, 10, 18,  8, 18, 10,
3           19, 8, 18, 10, 28,  8, 18, 10, 29,  8, 18, 10, 30,  8, 18, 10, 31,  8,
4           18, 10, 32,  8, 18, 10, 33,  8, 18, 10, 34,  8, 18, 10, 35,  8, 18, 10,
5           36, 8, 18, 10, 37,  8, 18, 10, 20,  8, 18, 10, 20, 11,  8, 18, 10, 20,
6           12, 8, 18, 10, 20, 13], device='cuda:0')
```

```
1 | y1[:,0]
```

```
1 tensor([18, 10, 11,  8, 18, 10, 12,  8, 18, 10, 13,  8, 18, 10, 14,  8, 18,
2           10, 15,  8, 18, 10, 16,  8, 18, 10, 17,  8, 18, 10, 18,  8, 18, 10, 19,
3           8, 18, 10, 28,  8, 18, 10, 29,  8, 18, 10, 30,  8, 18, 10, 31,  8, 18,
4           10, 32,  8, 18, 10, 33,  8, 18, 10, 34,  8, 18, 10, 35,  8, 18, 10, 36,
5           8, 18, 10, 37,  8, 18, 10, 20,  8, 18, 10, 20, 11,  8, 18, 10, 20,
6           12, 18, 10, 20, 13,  8], device='cuda:0')
```

这里，你可以看到第一个批次的X（记住，我们已经把这些都变成数字了），这是第一个批次的Y。你可以看到这里 `x1` 是 `[2, 18, 10, 11, 8, ...]`，`y1` 是 `[18, 10, 11, 8, ...]`。`y1` 比 `x1` 偏移了一个位置。这是因为，我们要做一个语言模型，我们要预测下一个单词，所以2后面应该是18，18后面应该是10。

```
1 | v = data.valid_ds.vocab
```

```
1 | v.textify(x1[:,0])
```

```
1 'xxbos eight thousand one , eight thousand two , eight thousand three , eight
thousand four , eight thousand five , eight thousand six , eight thousand
seven , eight thousand eight , eight thousand nine , eight thousand ten ,
eight thousand eleven , eight thousand twelve , eight thousand thirteen ,
eight thousand fourteen , eight thousand fifteen , eight thousand sixteen ,
eight thousand seventeen , eight thousand eighteen , eight thousand nineteen ,
eight thousand twenty , eight thousand twenty one , eight thousand twenty
two , eight thousand twenty three'
```

```
1 | v.textify(y1[:,0])
```

```
1 | 'eight thousand one , eight thousand two , eight thousand three , eight  
thousand four , eight thousand five , eight thousand six , eight thousand  
seven , eight thousand eight , eight thousand nine , eight thousand ten ,  
eight thousand eleven , eight thousand twelve , eight thousand thirteen ,  
eight thousand fourteen , eight thousand fifteen , eight thousand sixteen ,  
eight thousand seventeen , eight thousand eighteen , eight thousand nineteen ,  
eight thousand twenty , eight thousand twenty one , eight thousand twenty  
two , eight thousand twenty three ,'
```

你可以取这个数据集的词汇表 (vocab) , 一个vocab有一个textify, 如果我们给textify输入这些参数, 它会在vocab里把单词找出来。你在这里可以看到xxbos eight thousand one, 但在y里, 没有xxbos, 只是eight thousand one。xxbos之后是eight, eight之后是thousand, thousand之后是one

```
1 | v.textify(x2[:, 0])
```

```
1 | ', eight thousand twenty four , eight thousand twenty five , eight thousand  
twenty six , eight thousand twenty seven , eight thousand twenty eight ,  
eight thousand twenty nine , eight thousand thirty , eight thousand thirty  
one , eight thousand thirty two , eight thousand thirty three , eight  
thousand thirty four , eight thousand thirty five , eight thousand thirty six  
, eight thousand thirty seven'
```

```
1 | v.textify(x3[:, 0])
```

```
1 | ', eight thousand thirty eight , eight thousand thirty nine , eight thousand  
forty , eight thousand forty one , eight thousand forty two , eight thousand  
forty three , eight thousand forty four , eight thousand forty '
```

在我们到了8023后, 就是x2。看看这个, 我们总是看第0列, 这是第一个批次 (第一个mini batch), 第一个值是8024, 然后是x3, 最后到了8040。

```
1 | v.textify(x1[:, 1])
```

```
1 | ', eight thousand forty six , eight thousand forty seven , eight thousand  
forty eight , eight thousand forty nine , eight thousand fifty , eight  
thousand fifty one , eight thousand fifty two , eight thousand fifty three ,  
eight thousand fifty four , eight thousand fifty five , eight thousand fifty  
six , eight thousand fifty seven , eight thousand fifty eight , eight  
thousand fifty nine , eight thousand sixty , eight thousand sixty one , eight  
thousand sixty two , eight thousand sixty three , eight thousand sixty four ,  
eight'
```

```
1 | v.textify(x2[:, 1])
```

```
1 | 'thousand sixty five , eight thousand sixty six , eight thousand sixty seven  
    , eight thousand sixty eight , eight thousand sixty nine , eight thousand  
    seventy , eight thousand seventy one , eight thousand seventy two , eight  
    thousand seventy three , eight thousand seventy four , eight thousand seventy  
    five , eight thousand seventy six , eight thousand seventy seven , eight  
    thousand seventy eight , eight'
```

```
1 | v.textify(x3[:,1])
```

```
1 | 'thousand seventy nine , eight thousand eighty , eight thousand eighty one ,  
    eight thousand eighty two , eight thousand eighty three , eight thousand  
    eighty four , eight thousand eighty five , eight thousand eighty six ,'
```

```
1 | v.textify(x3[:, -1])
```

```
1 | 'ninety , nine thousand nine hundred ninety one , nine thousand nine hundred  
    ninety two , nine thousand nine hundred ninety three , nine thousand nine  
    hundred ninety four , nine thousand nine hundred ninety five , nine'
```

然后我们回到开始的地方，看看索引1，就是第2个批次。现在我们继续。从8040跳到8046，这是因为最后的mini batch没有完全完成。每个mini batch和前一个mini batch连接在一起。你可以直接从x1[0]到x2[0]，它从8023继续到8024。如果你对`[:,1]`做同样的处理，你也会看到它们连在一起。所有的mini batch都连在一起。

```
1 | data.show_batch(ds_type=DatasetType.Valid)
```

idx	text
0	xxbos eight thousand one , eight thousand two , eight thousand three , eight thousand four , eight thousand five , eight thousand six , eight thousand seven , eight thousand eight , eight thousand nine , eight thousand ten , eight thousand eleven , eight thousand twelve , eight thousand thirteen , eight thousand fourteen , eight thousand fifteen , eight thousand sixteen , eight thousand
1	, eight thousand forty six , eight thousand forty seven , eight thousand forty eight , eight thousand forty nine , eight thousand fifty , eight thousand fifty one , eight thousand fifty two , eight thousand fifty three , eight thousand fifty four , eight thousand fifty five , eight thousand fifty six , eight thousand fifty seven , eight thousand fifty eight , eight thousand
2	thousand eighty seven , eight thousand eighty eight , eight thousand eighty nine , eight thousand ninety , eight thousand ninety one , eight thousand ninety two , eight thousand ninety three , eight thousand ninety four , eight thousand ninety five , eight thousand ninety six , eight thousand ninety seven , eight thousand ninety eight , eight thousand ninety nine , eight thousand one hundred
3	thousand one hundred twenty three , eight thousand one hundred twenty four , eight thousand one hundred twenty five , eight thousand one hundred twenty six , eight thousand one hundred twenty seven , eight thousand one hundred twenty eight , eight thousand one hundred twenty nine , eight thousand one hundred thirty , eight thousand one hundred thirty one , eight thousand one hundred thirty two
4	fifty two , eight thousand one hundred fifty three , eight thousand one hundred fifty four , eight thousand one hundred fifty five , eight thousand one hundred fifty six , eight thousand one hundred fifty seven , eight thousand one hundred fifty eight , eight thousand one hundred fifty nine , eight thousand one hundred sixty , eight thousand one hundred sixty one , eight thousand

这是数据，我们可以用show_batch来查看它。

```
1 | data = src.databunch(bs=bs, bptt=3, max_len=0, p_bptt=1.)
```

```
1 | x,y = data.one_batch()
2 | x.shape,y.shape
```

```
1 | (torch.size([3, 64]), torch.size([3, 64]))
```

```
1 | nv = len(v.itos); nv
```

```
1 | 38
```

```
1 | nh=64
```

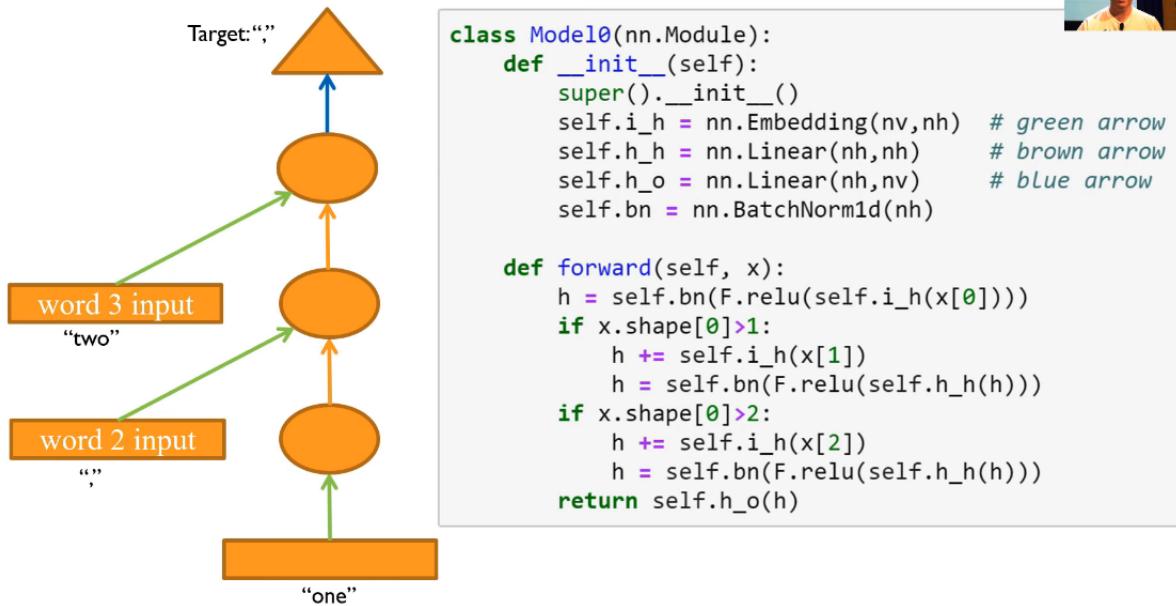
```
1 | def loss4(input,target): return F.cross_entropy(input, target[-1])
2 | def acc4 (input,target): return accuracy(input, target[-1])
```

这是我们的模型，它做了我们在图里看到的事情：

```
1 class Model0(nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.i_h = nn.Embedding(nv,nh) # green arrow
5         self.h_h = nn.Linear(nh,nh)    # brown arrow
6         self.h_o = nn.Linear(nh,nv)    # blue arrow
7         self.bn = nn.BatchNorm1d(nh)
8
9     def forward(self, x):
10        h = self.bn(F.relu(self.i_h(x[0])))
11        if x.shape[0]>1:
12            h += self.i_h(x[1])
13            h = self.bn(F.relu(self.h_h(h)))
14        if x.shape[0]>2:
15            h += self.i_h(x[2])
16            h = self.bn(F.relu(self.h_h(h)))
17        return self.h_o(h)
```

这是一段拷贝过来的代码：

Predicting word 4 using words 1, 2 & 3



它包含一个embedding（绿色箭头），一个隐藏状态到隐藏状态，棕色箭头的层，一个隐藏状态到输出。每个标颜色的箭头都有一个矩阵。在forward里，我们取第一个输入`x[0]`，让它通过输入到隐藏状态层（绿色箭头），创建我们第一个激活值集合，我们叫它`h`。假设有第二个单词，因为有时我们可能是在批次的最后，这就没有第二个单词。假设有第二个单词，我们把`h`加到`x[1]`的结果上，让它通过绿色箭头（`i_h`）。然后我们可以说，好了，我们的新`h`是这两个个相加的结果，放入隐藏状态到隐藏状态（棕色箭头），然后ReLU，然后batch norm。然后，对这第二个单词，做相同的事情。最后经过蓝色箭头，放入`h_o`。

这就是怎样把图转成代码。没有什么新东西。我们可以把它放进一个learner里，训练它，准确率46%。

```
1 | learn = Learner(data, Model0(), loss_func=loss4, metrics=acc4)
```

```
1 | learn.fit_one_cycle(6, 1e-4)
```

Total time: 00:05

epoch	train_loss	valid_loss	acc4
1	3.533459	3.489706	0.098855
2	3.011390	3.033105	0.450031
3	2.452748	2.552569	0.461247
4	2.154685	2.315783	0.461711
5	2.039904	2.236383	0.462020
6	2.016217	2.225322	0.462020

重构进循环 [1:50:48]

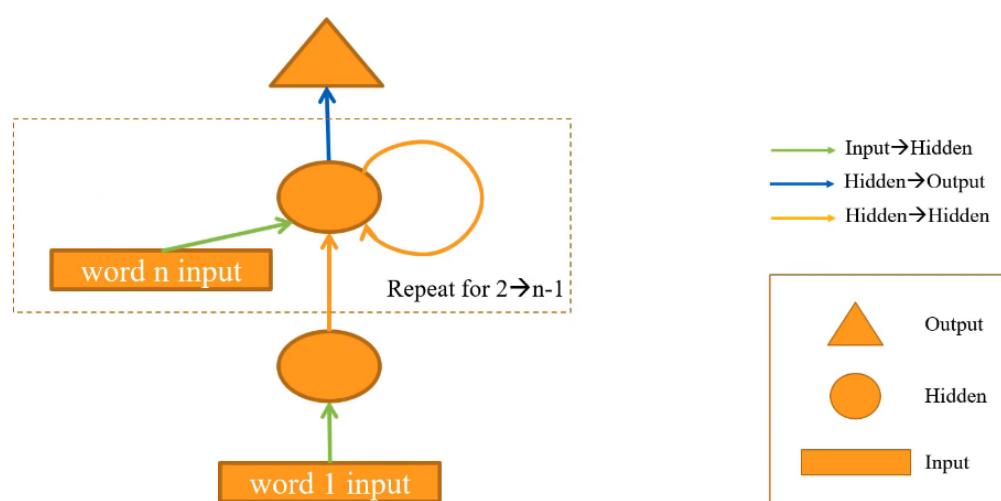
有很多重复代码，作为coder，当我们的看到重复代码时，应该怎样做？要重构。我们要把它重构进一个循环。

```
1 class Model1(nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.i_h = nn.Embedding(nv,nh) # green arrow
5         self.h_h = nn.Linear(nh,nh)    # brown arrow
6         self.h_o = nn.Linear(nh,nv)    # blue arrow
7         self.bn = nn.BatchNorm1d(nh)
8
9     def forward(self, x):
10        h = torch.zeros(x.shape[1], nh).to(device=x.device)
11        for xi in x:
12            h += self.i_h(xi)
13            h = self.bn(F.relu(self.h_h(h)))
14        return self.h_o(h)
```

好了，我们把它重构进了一个循环。我们要在循环里对每一个`x`里的`xi`，做这个。这就是RNN。RNN就是一个重构。没有什么新东西。现在，它是一个RNN了。我们来重构我们的图：

Predicting word n using words 1 to n-1

NB: no hidden/output layer



这是一个相同的图，但我用循环替换了它。它做了相同的事。它用了完全相同的 `__init__`，只是有一个在这里。在我开始之前，我需要保证我有一组0来加到上面。当然，训练出的结果是一样的。

```
1 | learn = Learner(data, Model1(), loss_func=loss4, metrics=acc4)
```

```
1 | learn.fit_one_cycle(6, 1e-4)
```

Total time: 00:07

epoch	train_loss	valid_loss	acc4
1	3.463261	3.436951	0.172881
2	2.937433	2.903948	0.385984
3	2.405134	2.457942	0.454827
4	2.100047	2.231621	0.459468
5	1.981868	2.155234	0.460860
6	1.957631	2.144365	0.461324

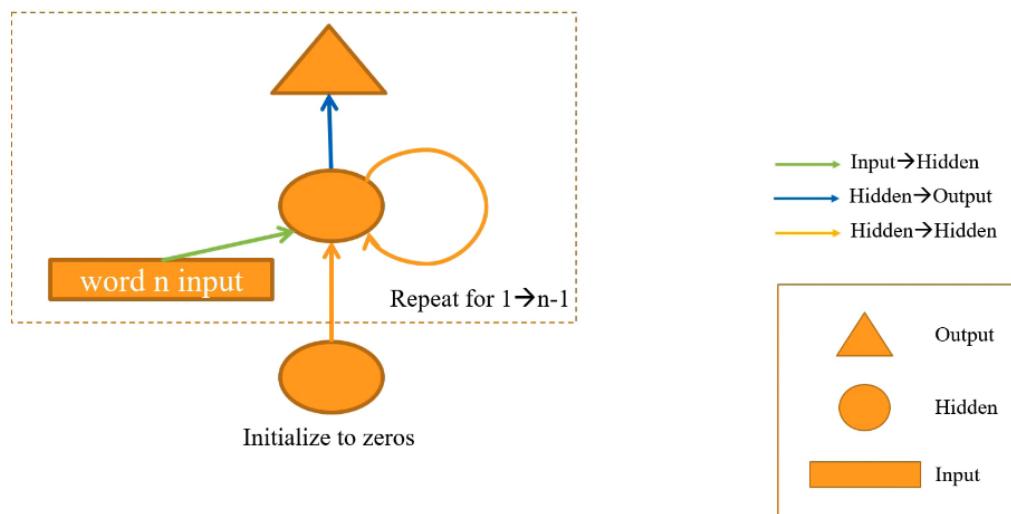
使用循环的一个好处是，如果我不是用前三个单词预测第四个，而是用前八个预测第九个。它也可以做到。可以用它来处理任意长的序列。

我们来把 `bptt` 变成20。现在，不再从前 $n - 1$ 个单词预测第 n 个单词，我们来用第一个单词预测第二个、用第二个单词预测第三个、用第三个单词预测第四个等等。看看我们的损失函数。

```
def loss4(input,target): return F.cross_entropy(input, target[-1])
def acc4 (input,target): return accuracy(input, target[-1])
```

之前我们把模型的结果和序列的最后一个单词做比较。这很浪费，因为序列里有很多单词。我们来把 x 的每个单词和 y 里的每个单词做比较。要做到这个，我们要改变这个图，不再是只在循环的最后有一个三角，而是三角进到训练里边：

Predicting words 2 to n using words 1 to n-1



换句话说，在每个循环后，预测，循环，预测，循环，预测。

```
1 | data = src.databunch(bs=bs, bptt=20)
```

```
1 | x,y = data.one_batch()
2 | x.shape,y.shape
```

```
1 | (torch.size([45, 64]), torch.size([45, 64]))
```

```
1 | class Model2(nn.Module):
2 |     def __init__(self):
3 |         super().__init__()
4 |         self.i_h = nn.Embedding(nv,nh)
5 |         self.h_h = nn.Linear(nh,nh)
6 |         self.h_o = nn.Linear(nh,nv)
7 |         self.bn = nn.BatchNorm1d(nh)
8 |
9 |     def forward(self, x):
10 |         h = torch.zeros(x.shape[1], nh).to(device=x.device)
11 |         res = []
12 |         for xi in x:
13 |             h += self.i_h(xi)
14 |             h = self.bn(F.relu(self.h_h(h)))
15 |             res.append(self.h_o(h))
16 |         return torch.stack(res)
```

这是代码。它和以前的一样，只是现在我创建了一个数组，每次在循环时，把 $h_o(h)$ 加入到这个数组里。现在对于 n 个输入，我创建了 n 个输出。每个单词后都做预测。

```
1 | learn = Learner(data, Model2(), metrics=accuracy)
```

```
1 | learn.fit_one_cycle(10, 1e-4, pct_start=0.1)
```

Total time: 00:06

epoch	train_loss	valid_loss	accuracy
1	3.704546	3.669295	0.023670
2	3.606465	3.551982	0.080213
3	3.485057	3.433933	0.156405
4	3.360244	3.323397	0.293704
5	3.245313	3.238923	0.350156
6	3.149909	3.181015	0.393054
7	3.075431	3.141364	0.404316
8	3.022162	3.121332	0.404548
9	2.989504	3.118630	0.408416
10	2.972034	3.114454	0.408029

之前的准确率是46%，现在是40%。为什么变差了？因为我在预测第二个单词时，只有一个单词的状态可以用。当我预测第三个单词时，只有两个单词的状态可以用。这是一个很难解决的问题。关键问题在这里：

```
class Model2(nn.Module):
    def __init__(self):
        super().__init__()
        self.i_h = nn.Embedding(nv,nh)
        self.h_h = nn.Linear(nh,nh)
        self.h_o = nn.Linear(nh,nv)
        self.bn = nn.BatchNorm1d(nh)

    def forward(self, x):
        h = torch.zeros(x.shape[1], nh).to(device=x.device)
        res = []
        for xi in x:
            h += self.i_h(xi)
            h = self.bn(F.relu(self.h_h(h)))
            res.append(self.h_o(h))
        return torch.stack(res)
```

我运行 `h = torch.zeros`。我每次开始做另外一个BPTT序列时，把状态重置成0。我们不要再这样做了。我们保持 `h`。我们可以这样做，因为每个批次和前面的批次连接着。我们重做一遍，这次把创建 `h` 的代码放到构造函数里。

```
1 class Model3(nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.i_h = nn.Embedding(nv,nh)
5         self.h_h = nn.Linear(nh,nh)
6         self.h_o = nn.Linear(nh,nv)
7         self.bn = nn.BatchNorm1d(nh)
8         self.h = torch.zeros(x.shape[1], nh).cuda()
9
10    def forward(self, x):
11        res = []
12        h = self.h
13        for xi in x:
14            h = h + self.i_h(xi)
15            h = F.relu(self.h_h(h))
16            res.append(h)
17            self.h = h.detach()
18        res = torch.stack(res)
19        res = self.h_o(self.bn(res))
20        return res
```

就是这样，现在它是 `self.h`。这还是相同的代码，但是最后，我们把新的 `h` 赋值给 `self.h`。它现在在做相同的事，但是不会丢掉这个状态。

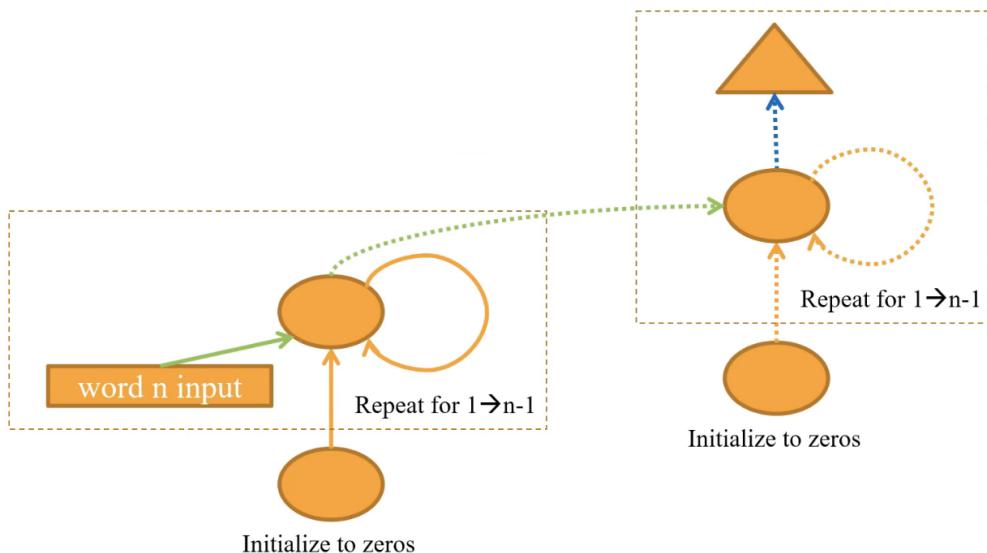
```
1 | learn = Learner(data, Model3(), metrics=accuracy)
```

```
1 | learn.fit_one_cycle(20, 3e-3)
```

Total time: 00:09

epoch	train_loss	valid_loss	accuracy
1	3.574752	3.487574	0.096380
2	3.218008	2.850531	0.269261
3	2.640497	2.155723	0.465269
4	2.107916	1.925786	0.372293
5	1.743533	1.977690	0.366027
6	1.461914	1.873596	0.417002
7	1.239240	1.885451	0.467923
8	1.069399	1.886692	0.476949
9	0.943912	1.961975	0.473159
10	0.827006	1.950261	0.510674
11	0.733765	2.038847	0.520471
12	0.658219	1.988615	0.524675
13	0.605873	1.973706	0.550201
14	0.551433	2.027293	0.540130
15	0.519542	2.041594	0.531250
16	0.497289	2.111806	0.537891
17	0.476476	2.104390	0.534837
18	0.458751	2.112886	0.534242
19	0.463085	2.067193	0.543007
20	0.452624	2.089713	0.542400

现在，我们的结果比上次好了。准确率上升到了54%。这就是RNN。你要一直保存状态。要记住，RNN没什么特别的，它就是一个普通的全连接网络，只是重构了一个循环。



在每个循环最后，你可以不单单输出一个结果，你可以把它输出到另外一个RNN里。你可以从一个RNN进到另外一个RNN。这很好，因为我们有了更多层来计算，它的效果会更好。

要做到这个，我们要再重构。我们复制 `Model13` 的代码，用PyTorch内置的代码替换它，可以这样写：

```

1 | class Model14(nn.Module):
2 |     def __init__(self):
3 |         super().__init__()
4 |         self.i_h = nn.Embedding(nv,nh)
5 |         self.rnn = nn.RNN(nh,nh)
6 |         self.h_o = nn.Linear(nh,nv)
7 |         self.bn = nn.BatchNorm1d(nh)
8 |         self.h = torch.zeros(1, x.shape[1], nh).cuda()
9 |
10 |    def forward(self, x):
11 |        res,h = self.rnn(self.i_h(x), self.h)
12 |        self.h = h.detach()
13 |        return self.h_o(self.bn(res))

```

`nn.RNN` 就是说为我做循环。我们还是会用相同的embedding、相同的输出、相同的batch norm、相同的 `h` 初始化，但是没有了循环。RNN的好处是你现在就可以说你想要多少层。当然，准确率是一样的：

```
1 | learn = Learner(data, Model14(), metrics=accuracy)
```

```
1 | learn.fit_one_cycle(20, 3e-3)
```

Total time: 00:04

epoch	train_loss	valid_loss	accuracy
1	3.502738	3.372026	0.252707
2	3.092665	2.714043	0.457998
3	2.538071	2.189881	0.467048
4	2.057624	1.946149	0.451719
5	1.697061	1.923625	0.466471
6	1.424962	1.916880	0.487856
7	1.221850	2.029671	0.507735
8	1.063150	1.911920	0.523128
9	0.926894	1.882562	0.541045
10	0.801033	1.920954	0.541228
11	0.719016	1.874411	0.553914
12	0.625660	1.983035	0.558014
13	0.574975	1.900878	0.560721
14	0.505169	1.893559	0.571627
15	0.468173	1.882392	0.576869
16	0.430182	1.848286	0.574489
17	0.400253	1.899022	0.580929
18	0.381917	1.907899	0.579285
19	0.365580	1.913658	0.578666
20	0.367523	1.918424	0.577197

现在，我要用两层：

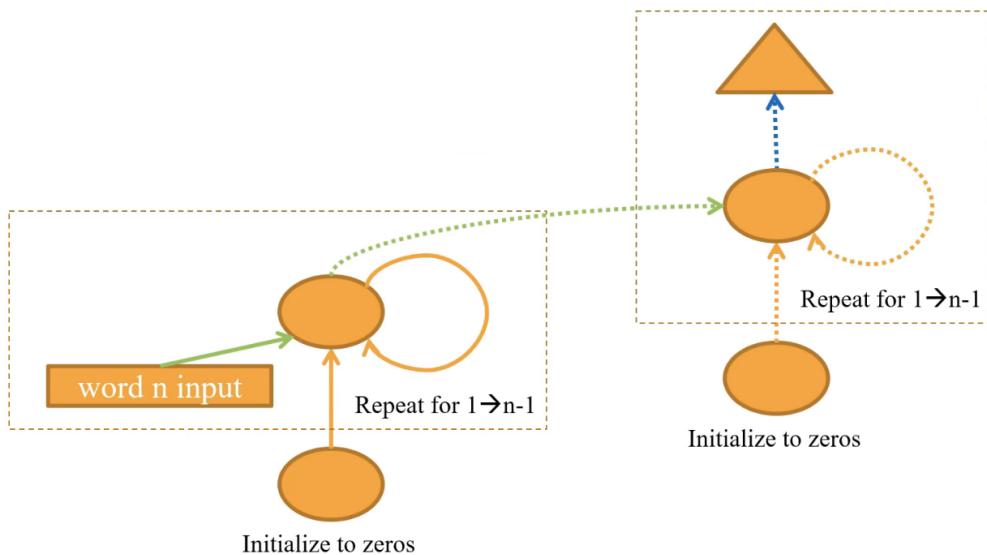
```

1  class Model5(nn.Module):
2      def __init__(self):
3          super().__init__()
4          self.i_h = nn.Embedding(nv,nh)
5          self.rnn = nn.GRU(nh,nh,2)
6          self.h_o = nn.Linear(nh,nv)
7          self.bn = nn.BatchNorm1d(nh)
8          self.h = torch.zeros(2, bs, nh).cuda()
9
10     def forward(self, x):
11         res,h = self.rnn(self.i_h(x), self.h)
12         self.h = h.detach()
13         return self.h_o(self.bn(res))

```

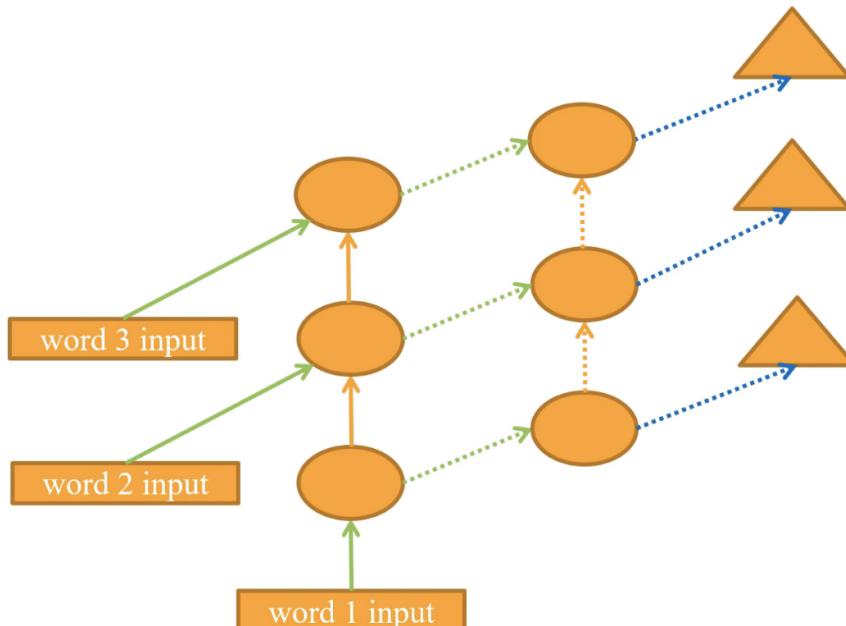
现在它是这样的：

Predicting words 2 to n using words 1 to n-1 using stacked RNNs



不用循环，它是这样的：

Unrolled stacked RNNs for sequences



我们用了20的BPTT，所以这里有20层。我们从可视化损失空间的论文里知道，深度网络有很曲折的损失平面。当我们创建很长的很多层的网络时，它很难训练。有一些可以使用的方法。一个是你可以在添加skip connection。人们经常不再单单把这些加在一起（绿色箭头和橙色箭头），它们用一个小的神经网络决定要保持多少绿色的箭头，多少橙色的箭头。当你这样做时，你得到了叫GRU或LSTM的东西，至于是哪个，这取决于这个小网络的细节。我们会在课程第二部分学习这些网络的细节。说实话，它们不是很重要。

现在，我们可以用一个GRU来替代。它和我们之前的很像，只是它可以在更深的网络里处理更长的序列。我们用两层。

```
1 | learn = Learner(data, Model5(), metrics=accuracy)
```

```
1 | learn.fit_one_cycle(10, 1e-2)
```

Total time: 00:02

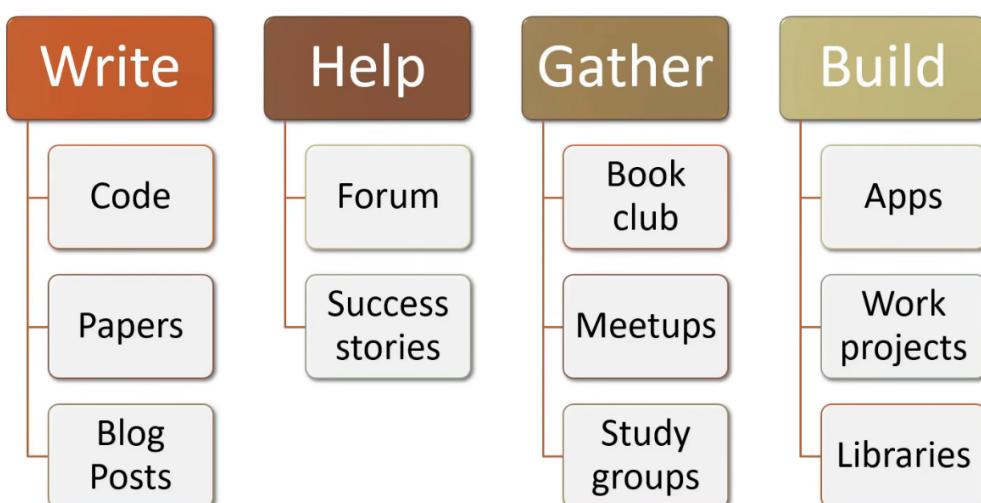
epoch	train_loss	valid_loss	accuracy
1	3.010502	2.602906	0.428063
2	2.087371	1.765773	0.532410
3	1.311803	1.571677	0.643796
4	0.675637	1.594766	0.730275
5	0.363373	1.432574	0.760873
6	0.188198	1.704319	0.762454
7	0.108004	1.716183	0.755837
8	0.064206	1.510942	0.763404
9	0.040955	1.633394	0.754179
10	0.034651	1.621733	0.755460

提升到了75%。这就是RNN。我讲它的主要原因，是要揭开最后的一个神秘的东西。这是深度学习里最后一个神秘的东西。它只是重构了一个全连接网络。不要让RNN吓倒你。这样，你有一个有 n 个输入的序列，一个有 n 个输出的序列，我们用它来做语言模型，你可以把它用在其它任务上。

比如，输出的序列可以用在每个单词上，来判断有没有什么东西很敏感，是不是要消除它。可以判断是不是私有的数据。它可以用来为单词做语音标注。可以来判断是不是要格式化单词。这些都被叫做**序列标注任务 (sequence labeling task)**，你可以用相同的方法来做所有的序列标注任务。你可以像我在前面课里做的一样，在做完语言模型后，去掉 `h_o`，放入一个标准的分类头，然后，你可以做NLP 分类了，这会给出一个非常好的结果，即使是很长的文档。这是超级有用的技术，不是遥远神秘的魔法。

接下来该做什么 [1:58:59]

So what now? Watch the videos again, and...



就是这样。这就是深度学习，至少在我看来实践的部分就是这些。只看一遍，你不会全部掌握。我不建议你看得很慢，在第一遍就全部掌握。你可以回过来再看一遍，慢慢来，有些地方你可能回觉得“噢，我现在知道他在讲什么了”。然后，你可以实现你以前做不到的东西，你可以比以前更深入。一定要再看一遍。并且，写代码，不只是自己写，还要把它提交到github上。无论你是不是觉得它们是好代码。你写代码并且分享它这件事本身就是有巨大意义的，如果你在论坛上告诉人们，“嘿，我写了这些代码。它不是很棒，但这是我的第一次努力。你们看有什么问题吗”
。人们会说“噢，这里做得很好。但是，这里，你可以用这个库，这会省些时间”。你会从同伴那里学到很多。

你们已经注意到了，我开始介绍越来越多的论文。课程第二部分会有很多论文，现在可以开始阅读一些已经介绍过的论文。对那些推导、定理、引理之类的东西，你可以跳过它们。我就是这样的。它们对你做深度学习实践来说没有什么影响。那些说为什么我们要解决这个问题、这些结果是什么之类的部分就非常有意义。然后试着写些文章，不是用来给Geoffrey Hinton和Yann LeCun看的文章，是给6个月之前的你看的文章。因为在读者里，像六个月之前的你一样的人比像Geoffrey Hinton和Yann LeCun这样的人多得多。这些人是你最了解的。你知道他们需要什么。

去寻求帮助，也帮助别人。告诉我们你成功的故事。可能最重要的是和其它人一起学。如果有社区交流，人们会学得更好。可以办一个读书聚会，参与meetup，创建学习小组，做些项目。还是一样，这些不需要做出特别令人惊奇的成绩。只需要做些你觉得能让世界变得更好些的东西，或者你觉得能让你两岁的孩子看到后能开心的东西，或者在你的兄弟来看你在做什么时，你想展示给他们的东西，等等。去做一些东西，把它做完，然后再试着让它更好些。



Other Tweets:

- “Humanity will also have an option to publish on its journey as an alien civilization. it will always like all human being.”
- “Mars is no longer possible. with the weather up to 60% it is now a place where most human water deaths are in winter.”
- “AI will definitely be the central intelligence agency. if it can contact all about the core but this is how we can improve the pace of fighting of humanity in the future”

比如我今天下午看到的Elon Musk的推文生成器。它读了很多老的推文，创建了一个Elon Musk的语言模型，然后产生这些新的推文，比如 "Humanity will also have an option to publish on its own journey as an alien civilization. it will always like all human being." "Mars is no longer possible," "AI will definitely be the central intelligence agency."



OCData_nerd Dave Smith



Given Elon's erratic Twitter behavior and trouble with the SEC, I thought it would be fun to build a Musk-like Tweet generator trained with over 6,000 of his tweets from 2010-2018:

<https://deapelon.com/>

It starts with the language model (WikiText-103) discussed in [lesson 4](#) and is fine-tuned with Elon's tweets. Thank you to [@alvisanovari](#) for the Zeit template as your Walt Whitman generator was a helpful starting point!

Head [here](#) to generate your own tweet or review the code (my first commits ever were today!). Thanks to [@rachel](#) and [@jeremy](#) for teaching a finance guy how to build an app in 8 weeks 😊

这很棒，我喜欢这个。我喜欢Dave Smith写的“这是我第一次提交。[感谢在八周里教会一个做金融的人怎样构建一个app](#)”。我觉得这很棒。我觉得这个项目会受到很多热心的关注。这会改变未来的社会发展方向吗？大概不会。但Elon Musk可能会看到这个，然后想“噢，可能我要重新考虑我说话的方式了”。我觉得这很棒。所以，去创建一些东西，提交到这里，花点时间去做它。

或者参与到fastai里来。这个fastai项目，有很多事情要做。[你可以帮助写文档、做测试，这些可能听起来有点无聊。但你会惊奇地发现这并不无聊。](#)拿一段没有文档的代码，研究它，理解它，在论坛上问Sylvain和我，[发生了什么？为什么这样写？](#)我们会发给你我们在实现的论文。[编写一个测试需要深刻地理解机器学习里的这部分内容来知道它应该怎样运行。这是很有意思的。](#)

Stats Bakman 创建了这个很好的[Dev Projects Index](#)，你可以到论坛上fastai开发项目板块，找到正在进行的，你想参加的项目。

创建一个学习小组。Deena已经在一月创建了一个旧金山学习小组，创建学习小组很简单，到论坛里，找到你所在时区的板块，添加一个文章，说“我们来成立一个学习小组吧”。但是要给人们一个Google sheet之类的东西来登记，真正做些事情。

[一个很好的例子是Pierre，他在巴西组织了上一期课程的学习小组，做得很好。他不断贴出人们一起学习深度学习、创建wiki、创建项目的照片，很棒的经历。](#)



Coming up: part 2! Cutting Edge Deep Learning

Deep dive into fastai codebase	Development and research process	Reading academic papers	Translating math->code
Attentional models	Speech recognition	Translation	Multi-modal models
CycleGAN	Object detection	Large and distributed training	...and more!

在课程第二部分，[我们会学习所有这些有趣的东西。在实践上，深入fastai代码，理解我们是怎样构建它的。我们会做一遍。我们构造它们，在每个阶段，都创建notebook来学习我们在做什么，我们会看到软件开发过程。我们会讲做研究的过程，怎样读学术论文，怎样把它从数学符号变成代码。然后是一堆各种类型的没有学过的模型。这会从深度学习实践进入到实际的前沿研究。](#)

提问 (Ask Jeremy Anything) [2:05:26]

我们在线上有AMA (ask me anything) 活动，我们有时间讲几个社区提问最多的问题。

提问：第一个是Jeremy的请求，尽管它不是被提问最多的问题。你的典型的一天是怎样的，你怎样在这么多事情上分配时间？

我总是听到这个问题，所以我觉得我需要回答下，有些人为这个投票了。来到我们学习小组的人总是会被我的没有条理和没有效率震惊，我总是听到人们说“哇，我以前觉得你是深度学习的模范，我想看看怎样能变得像你一样，现在，我不想成为你那样了”。对我来说，这只是花了很多时间，我没有做过很多计划。我只是坚持去完成我开始做的东西。如果你得不到乐趣，就很难继续下去，因为在深度学习里有很多挫折，这不像写一个web app，就是做授权、检查、后台服务检查、用户凭证、检查，你在做流程。另外一边，对GAN这样的东西，是这样的：它没有效果、它没有效果、它没有效果、它还是没有效果、它还是没有效果，直到“哦，天，这太神奇了。这是一只猫”。它是这样的东西。我们没有持续的反馈。所以，你需要对这有兴趣。另外，我不参加什么会议，我不打电话，我不喝咖啡，我不看电视，我不玩电脑游戏。我花很多时间陪家人，花很多时间锻炼，花很多时间阅读、写代码、做我喜欢的事。主要就是去把事情做完，彻底地完成它。你完成了80%，但还没有创建一个README，安装过程还有点麻烦，github上99%的项目都是这样。你会看到README里写着：“TODO：完成基线实验文档……”。不要做这样的人。彻底完成一些东西，可以和一些人一起做，把它做完。

提问：什么是最让你兴奋的、有前途的深度学习/机器学习的东西？你去年讲过你不是强化学习的粉丝。现在你还是这样觉得吗？

和三年前开始做这个课程时一样，我还是认为是迁移学习。它的价值没有被充分认识，还没有被充分研究。每次我们把迁移学习用到各种东西里，它就会变好很多。我们在NLP上用迁移学习的论文改变了NLP的方向，被《纽约时报》报道了，只是一个简单的、明显的小东西。所以我还是为它兴奋。我仍然不对把强化学习用到大部分东西上感到振奋。我没有看到它被普通人用在普遍的任务上。对那些可以被很简单很快速解决的问题，它是一个难以置信的低效的方法。它会有它擅长的领域，但是不会用到大多数人日常工作中。

提问：对于要参加2019年第二部分课程的人，在课程开始前，你建议做什么学习实践呢？

就是写代码。是的，就是一直写代码。我知道这完全可行，我听到有人学到这里还没有写任何代码。如果你是这样的，那没关系。你只要再学一遍，这次要写代码。看看输入的形状，看看输出，要知道怎样取一个mini batch，看看它的平均数和标准差，把它画出来。你们有很多资料。如果你能自己从头写出这些notebook，我说的从头写是说用fastai库，不是完全从头写，你会成为头部梯队的一员，因为你能自己做所有这些东西，这非常非常难得。这会让你对part2有充分的准备。

提问：你认为未来fastai库会怎样，比如说在五年内？

像我说过的，我不做什么计划，很随便，所以...我们唯一的计划是，fast.ai做为一个组织，来让深度学习成为一个普通人做普通工作时能使用的工具。只要我们还需要写代码，就做不到这个，因为世界上99.8%的人不会写代码。所以主要的目标会是，让人们不再用一个库，而是能用上一个不需要用户写代码的软件。并且，也不再需要一个像这个一样的非常长的、很困难的课程。所以，我希望能不需要这个课程、不需要写代码，我希望能做到这样，那你们就可以只做有用的东西，能快速地、简单地做。这会在五年之内做到吗？可能要更长。

好了。希望能在课程第二部分看到你们所有人。谢谢。