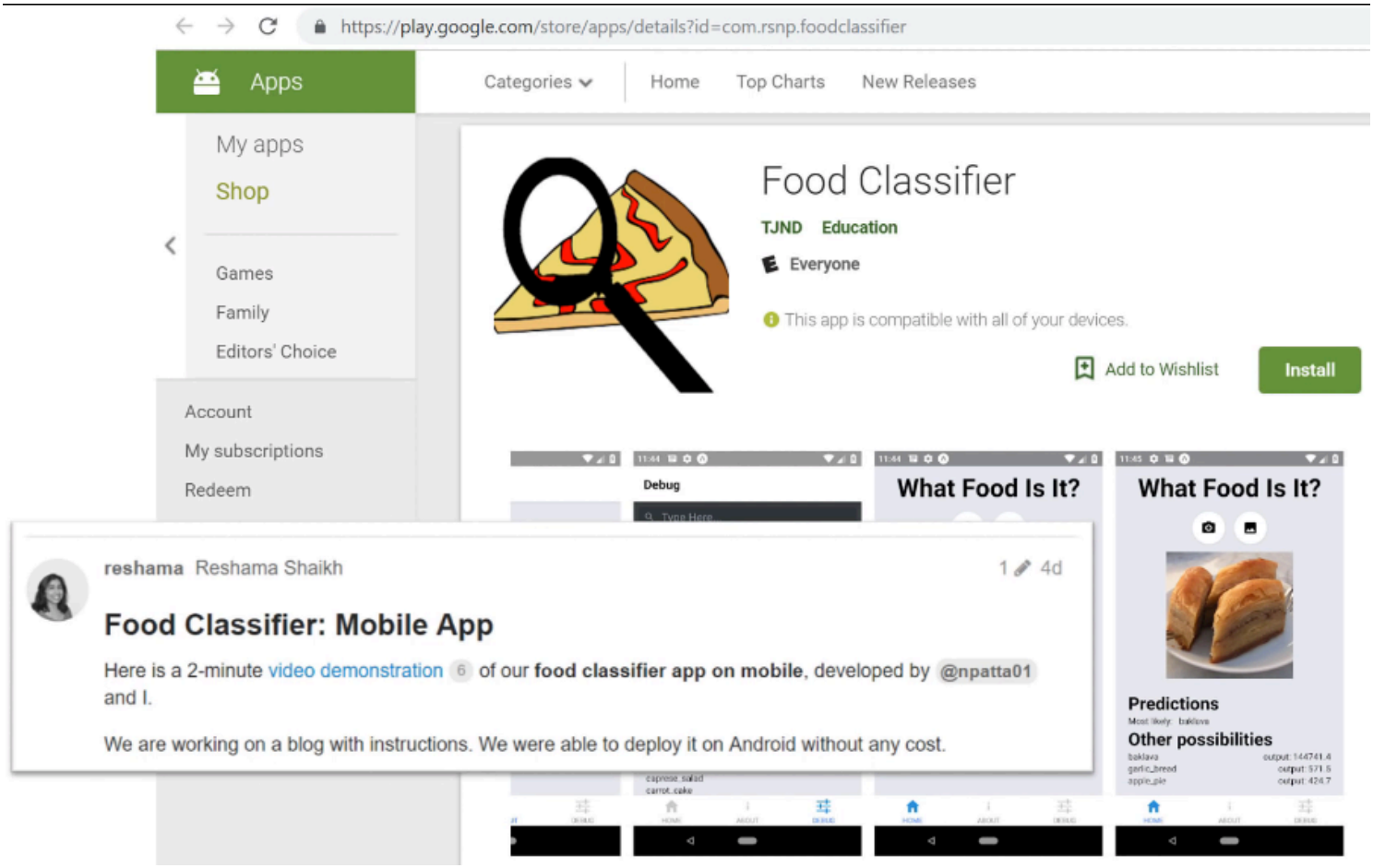
Lesson 7

Video / Course Forum

Welcome to lesson 7! The last lesson of part 1. This will be a pretty intense lesson. Don't let that bother you because partly what I want to do is to give you enough things to think about to keep you busy until part 2. In fact, some of the things we cover today, I'm not going to tell you about some of the details. I'll just point out a few things. I'll say like okay that we're not talking about yet, that we're not talking about yet. Then come back in part 2 to get the details on some of these extra pieces. So today will be a lot of material pretty quickly. You might require a few viewings to fully understand at all or a few experiments and so forth. That's kind of intentional. I'm trying to give you stuff to to keep you amused for a couple of months.



I wanted to start by showing some cool work done by a couple of students; Reshama and Nidhin who have developed an Android and an iOS app, so check out Reshma's post on the forum about that because they have a demonstration of how to create both Android and iOS apps that are actually on the Play Store and on the Apple App Store, so that's pretty cool. First ones I know of that are on the App Store's that are using fast.ai. Let me also say a huge thank you to Reshama for all of the work she does both for the fast.ai community and the machine learning community more generally, and also the Women in Machine Learning community in particular. She does a lot of fantastic work including providing lots of fantastic documentation and tutorials and community organizing and so many other things. So thank you, Reshama and congrats on getting this app out there.

MNIST CNN [2:04]

We have lots of lesson 7 notebooks today, as you see. The first notebook we're going to look at is lesson7-resnet-

mnist.ipynb. What I want to do is look at some of the stuff we started talking about last week around convolutions and convolutional neural networks, and start building on top of them to create a fairly modern deep learning architecture largely from scratch. When I say from scratch, I'm not going to re-implement things we already know how to implement, but use the pre-existing PyTorch bits of those. So we're going to use the MNIST dataset. `URLs.MNIST` has the whole MNIST dataset, often we've done stuff with a subset of it.

```
%reload_ext autoreload
%autoreload 2
%matplotlib inline

from fastai.vision import *

path = untar_data(URLs.MNIST)

path.ls()

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

In there, there's a training folder and a testing folder. As I read this in, I'm going to show some more details about pieces of the data blocks API, so that you see what's going on. Normally with the date blocks API, we've kind of said `blah.blah.blah.blah.blah` and done it all in one cell, but let's do it in one cell at a time.

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

First thing you say is what kind of item list do you have. So in this case it's an item list of images. Then where are you getting the list of file names from. In this case, by looking in a folder recursively. That's where it's coming from.

You can pass in arguments that end up going to Pillow because Pillow (a.k.a. PIL) is the thing that actually opens that for us, and in this case these are black and white rather than RGB, so you have to use Pillow's `convert_mode='L'`. For more details refer to the python imaging library documentation to see what their convert modes are. But this one is going to be a grayscale which is what MNIST is.

```
il.items[0]

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

So inside an item list is an `items` attribute, and the `items` attribute is kind of thing that you gave it. It's the thing that it's going to use to create your items. So in this case, the thing you gave it really is a list of file names. That's what it got from the folder.

```
defaults.cmap='binary'
```

When you show images, normally it shows them in RGB. In this case, we want to use a binary color map. In fast.ai, you can set a default color map. For more information about cmap and color maps, refer to the matplotlib documentation. And `defaults.cmap='binary'` world set the default color map for fast.ai.

```
il
```

```
ImageItemList (70000 items)
[Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Imag
Path: /home/jhoward/.fastai/data/mnist_png
```

Our image item list contains 70,000 items, and it's a bunch of images that are 1 by 28 by 28. Remember that PyTorch puts channel first, so they are one channel 28x28. You might think why aren't there just 28 by 28 matrices rather than a 1 by 28 by 28 rank 3 tensor. It's just easier that way. All the `Conv2d` stuff and so forth works on rank 3 tensors, so you want to include that unit axis at the start, so fast.ai will do that for you even when it's reading one channel images.

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

The `.items` attribute contains the things that's read to build the image which in this case is the file name, but if you just index into an item list directly, you'll get the actual image object. The actual image object has a `show` method, and so there's the image.

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

Once you've got an image item list, you then split it into training versus validation. You nearly always want validation. If you don't, you can actually use the `.no_split` method to create an empty validation set. You can't skip it entirely. You have to say how to split, and one of the options is `no_split`.

So remember, that's always the order. First create your item list, then decide how to split. In this case, we're going to do it based on folders. The validation folder for MNIST is called `testing`. In fast.ai parlance, we use the same kind of parlance that Kaggle does which is the training set is what you train on, the validation set has labels and you do it for testing that your models working. The test set doesn't have labels and you use it for doing inference, submitting to a competition, or sending it off to somebody who's held out those labels for vendor testing or whatever. So just because a folder in your data set is called `testing`, doesn't mean it's a test set. This one has labels, so it's a validation set.

If you want to do inference on lots of things at a time rather than one thing at a time, you want to use the `test=` in fast.ai to say this is stuff which has no labels and I'm just using for inference.

[6:54]

```
sd
```

```
ItemLists;
```

```
Train: ImageItemList (60000 items)
[Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Imag
Path: /home/jhoward/.fastai/data/mnist_png;

Valid: ImageItemList (10000 items)
[Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Imag
Path: /home/jhoward/.fastai/data/mnist_png;

Test: None
```

So my split data is a training set and a validation set, as you can see.

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

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

Inside the training set, there's a folder for each class.

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

Now we can take that split data and say `label_from_folder`.

So first you create the item list, then you split it, then you label it.

```
ll
```

```
LabelLists;

Train: LabelList
y: CategoryList (60000 items)
[Category 8, Category 8, Category 8, Category 8, Category 8]...
Path: /home/jhoward/.fastai/data/mnist_png
x: ImageItemList (60000 items)
[Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Imag
Path: /home/jhoward/.fastai/data/mnist_png;

Valid: LabelList
y: CategoryList (10000 items)
[Category 8, Category 8, Category 8, Category 8, Category 8]...
Path: /home/jhoward/.fastai/data/mnist_png
x: ImageItemList (10000 items)
[Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Imag
```

```
Path: /home/jhoward/.fastai/data/mnist_png;

Test: None
```

You can see now we have an `x` and the `y`, and the `y` are category objects. Category object is just a class basically.

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

If you index into a label list such as `ll.train` as a label list, you will get back an independent variable and independent variable (i.e. x and y). In this case, the `x` will be an image object which I can show, and the `y` will be a category object which I can print:

```
x.show()
print(y,x.shape)
```

```
8 torch.Size([1, 28, 28])
```

That's the number 8 category, and there's the 8.

[7:56]

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

Next thing we can do is to add transforms. In this case, we're not going to use the normal `get_transforms` function because we're doing digit recognition and digit recognition, you wouldn't want to flip it left right. That would change the meaning of it. You wouldn't want to rotate it too much, that would change the meaning of it. Also because these images are so small, doing zooms and stuff is going to make them so fuzzy as to be unreadable. So normally, for small images of digits like this, you just add a bit of random padding. So I'll use the random padding function which actually returns two transforms; the bit that does the padding and the bit that does the random crop. So you have to use star(`*`) to say put both these transforms in this list.

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

Now we call transform. This empty array here is referring to the validation set transforms:

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

So no transforms with the validation set.

Now we've got a transformed labeled list, we can pick a batch size and choose data bunch:

```
bs = 128

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

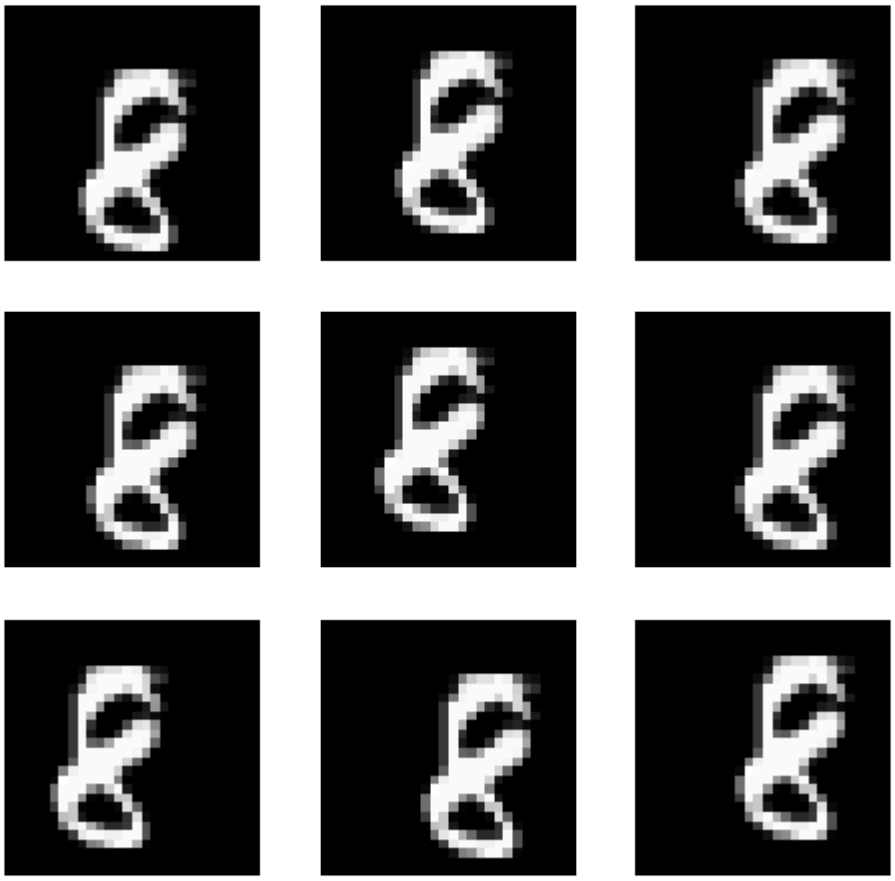
We can choose normalize. In this case, we're not using a pre-trained model, so there's no reason to use ImageNet stats here. So if you call normalize like this without passing in stats, it will grab a batch of data at random and use that to decide what normalization stats to use. That's a good idea if you're not using a pre-trained model.

```
x.show()
print(y)
```

```
8
```

Okay, so we've got a data bunch and in that data bunch is a data set which we've seen already. But what is interesting is that the training data set now has data augmentation because we've got transforms. `plot_multi` is a fast.ai function that will plot the result of calling some function for each of this row by column grid. So in this case, my function is just grab the first image from the training set and because each time you grab something from the training set, it's going to load it from disk and it's going to transform it on the fly. People sometimes ask how many transformed versions of the image do you create and the answer is infinite. Each time we grab one thing from the data set, we do a random transform on the fly, so potentially every one will look a little bit different. So you can see here, if we plot the result of that lots of times, we get 8's in slightly different positions because we did random padding.

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



[10:27]

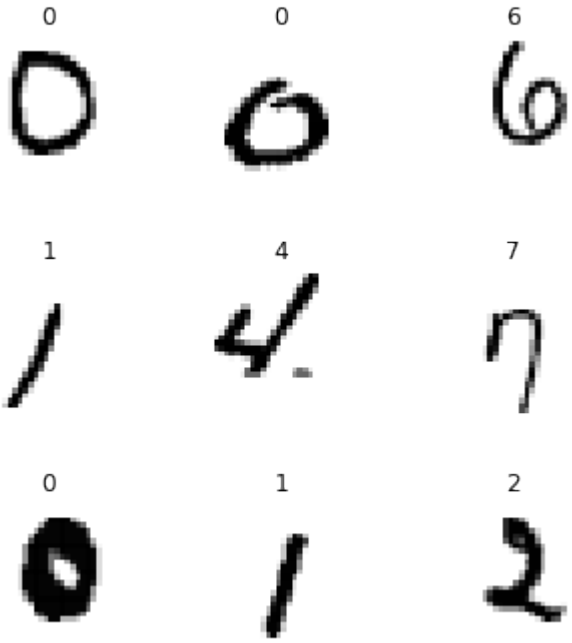
You can always grab a batch of data then from the data bunch, because remember, data bunch has data loaders, and data loaders are things you grab a batch at a time. So you can then grab a X batch and a Y batch, look at their shape - batch size by channel by row by column:

```
xb,yb = data.one_batch()
xb.shape,yb.shape
```

```
(torch.Size([128, 1, 28, 28]), torch.Size([128]))
```

All fast.ai data bunches have a show_batch which will show you what's in it in some sensible way:

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



That was a quick walk through with a data block API stuff to grab our data.

Basic CNN with batch norm 11:01

Let's start out creating a simple CNN. The input is 28 by 28. I like to define when I'm creating architectures a function which kind of does the things that I do again and again and again. I don't want to call it with the same arguments because I'll forget or I make a mistake. In this case, all of my convolution is going to be kernel size 3 stride 2 padding 1. So let's just create a simple function to do a conv with those parameters:

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

Each time you have a convolution, it's skipping over one pixel so it's jumping two steps each time. That means that each time we have a convolution, it's going to halve the grid size. I've put a comment here showing what the new grid size is after each one.

```
model = nn.Sequential(
    conv(1, 8), # 14
    nn.BatchNorm2d(8),
    nn.ReLU(),
    conv(8, 16), # 7
    nn.BatchNorm2d(16),
    nn.ReLU(),
    conv(16, 32), # 4
    nn.BatchNorm2d(32),
    nn.ReLU(),
    conv(32, 16), # 2
    nn.BatchNorm2d(16),
    nn.ReLU(),
    conv(16, 10), # 1
    nn.BatchNorm2d(10),
    Flatten()     # remove (1,1) grid
```

```
)
```

After the first convolution, we have one channel coming in because it's a grayscale image with one channel, and then how many channels coming out? Whatever you like. So remember, you always get to pick how many filters you create regardless of whether it's a fully connected layer in which case it's just the width of the matrix you're multiplying by, or in this case with the 2D conv, it's just how many filters do you want. So I picked 8 and so after this, it's stride 2 to so the 28 by 28 image is now a 14 by 14 feature map with 8 channels. Specifically therefore, it's an 8 by 14 by 14 tensor of activations.

Then we'll do a batch norm, then we'll do ReLU. The number of input filters to the next conv has to equal the number of output filters from the previous conv, and we can just keep increasing the number of channels because we're doing stride 2, it's got to keep decreasing the grid size. Notice here, it goes from 7 to 4 because if you're doing a stride 2 conv over 7, it's going to be `math.ceiling` of 7/2.

Batch norm, ReLU, conv. We are now down to 2 by 2. Batch norm, ReLU, conv, we're now down to 1 by 1. After this, we have a feature map of 10 by 1 by 1. Does that make sense? We've got a grid size of one now. It's not a vector of length 10, it's a rank 3 tensor of 10 by 1 by 1. Our loss functions expect (generally) a vector not a rank 3 tensor, so you can chuck `flatten` at the end, and flatten just means remove any unit axes. So that will make it now just a vector of length 10 which is what we always expect.

That's how we can create a CNN. Then we can return that into a learner by passing in the data and the model and the loss function and optionally some metrics. We're going to use cross-entropy as usual. We can then call `learn.summary()` and confirm.

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

learn.summary()
```

```
==========================================================================
Layer (type)                 Output Shape              Param #
==========================================================================
Conv2d                       [128, 8, 14, 14]          80
__________________________________________________________________________
BatchNorm2d                  [128, 8, 14, 14]          16
__________________________________________________________________________
ReLU                         [128, 8, 14, 14]          0
__________________________________________________________________________
Conv2d                       [128, 16, 7, 7]           1168
__________________________________________________________________________
BatchNorm2d                  [128, 16, 7, 7]           32
__________________________________________________________________________
ReLU                         [128, 16, 7, 7]           0
__________________________________________________________________________
Conv2d                       [128, 32, 4, 4]           4640
__________________________________________________________________________
BatchNorm2d                  [128, 32, 4, 4]           64
__________________________________________________________________________
ReLU                         [128, 32, 4, 4]           0
__________________________________________________________________________
Conv2d                       [128, 16, 2, 2]           4624
__________________________________________________________________________
```

```
BatchNorm2d                 [128, 16, 2, 2]       32
____________________________________________________________________________
ReLU                        [128, 16, 2, 2]       0
____________________________________________________________________________
Conv2d                      [128, 10, 1, 1]       1450
____________________________________________________________________________
BatchNorm2d                 [128, 10, 1, 1]       20
____________________________________________________________________________
Lambda                      [128, 10]             0
____________________________________________________________________________
Total params:  12126
```

After that first conv, we're down to 14 by 14 and after the second conv 7 by 7, 4 by 4, 2 by 2, 1 by 1. The `flatten` comes out (calling it a `Lambda`), that as you can see it gets rid of the 1 by 1 and it's now just a length 10 vector for each item in the batch so 128 by 10 matrix in the whole mini batch.

Just to confirm that this is working okay, we can grab that mini batch of X that we created earlier (there's a mini batch of X), pop it onto the GPU, and call the model directly. Any PyTorch module, we can pretend it's a function and that gives us back as we hoped a 128 by 10 result.

```
xb = xb.cuda()
```

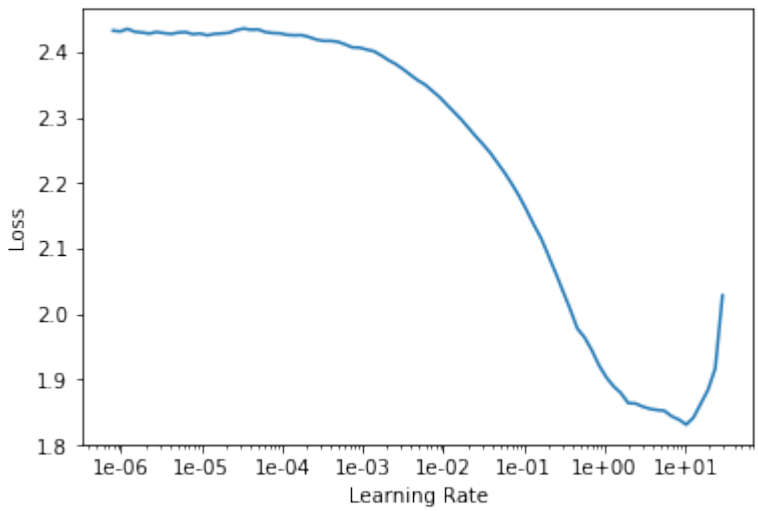
```
model(xb).shape
```

```
torch.Size([128, 10])
```

That's how you can directly get some predictions out. LR find, fit one cycle, and bang. We already have a 98.6% accurate conv net.

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

```
learn.recorder.plot()
```




```
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

This is trained from scratch, of course, it's not pre-trained. We've literally created our own architecture. It's about the simplest possible architecture you can imagine. 18 seconds to train, so that's how easy it is to create a pretty accurate digit detector.

Refactor 15:42

Let's refactor that a little. Rather than saying conv, batch norm, ReLU all the time, fast.ai already has something called `conv_layer` which lets you create conv, batch norm, ReLU combinations. It has various other options to do other tweaks to it, but the basic version is just exactly what I just showed you. So we can refactor that like so:

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

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

That's exactly the same neural net.

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

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

Let's just try a little bit longer and it's actually 99.1% accurate if we train it for all of a minute, so that's cool.

ResNet-ish 16:24

How can we improve this? What we really want to do is create a deeper network, and so a very easy way to create a deeper network would be after every stride 2 conv, add a stride 1 conv. Because the stride 1 conv doesn't change the feature map size at all, so you can add as many as you like. But there's a problem. The problem was pointed out in this paper, very very very influential paper, called Deep Residual Learning for Image Recognition by Kaiming He and colleagues at (then) Microsoft Research.

They did something interesting. They said let's look at the training error. So forget generalization even, let's just look at the training error of a network trained on CIFAR-10 and let's try one network of 20 layers just basic 3x3 convs - basically the same network I just showed you, but without batch norm. They trained a 20 layer one and a 56 layer one on the training set.

The 56 layer one has a lot more parameters. It's got a lot more of these stride 1 convs in the middle. So the one with more parameters should seriously over fit, right? So you would expect the 56 layer one to zip down to zero-ish training error pretty quickly and that is not what happens. It is worse than the shallower network.

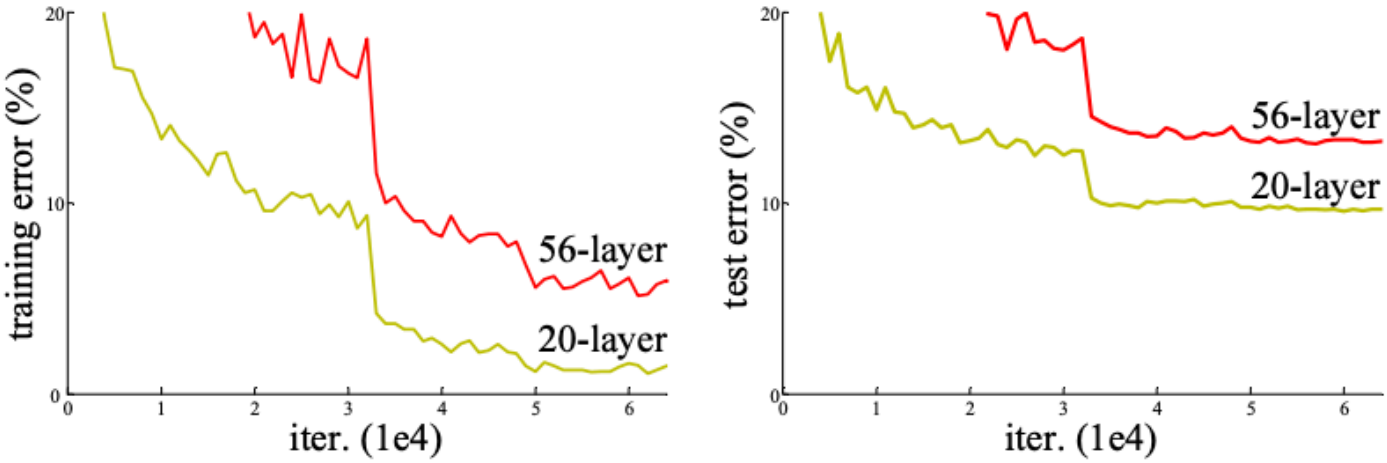


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

When you see something weird happen, really good researchers don't go "oh no, it's not working" they go "that's interesting." So Kaiming He said "that's interesting. What's going on?" and he said "I don't know, but what I do know is this - I could take this 56 layer network and make a new version of it which is identical but has to be at least as good as the 20 layer network and here's how:

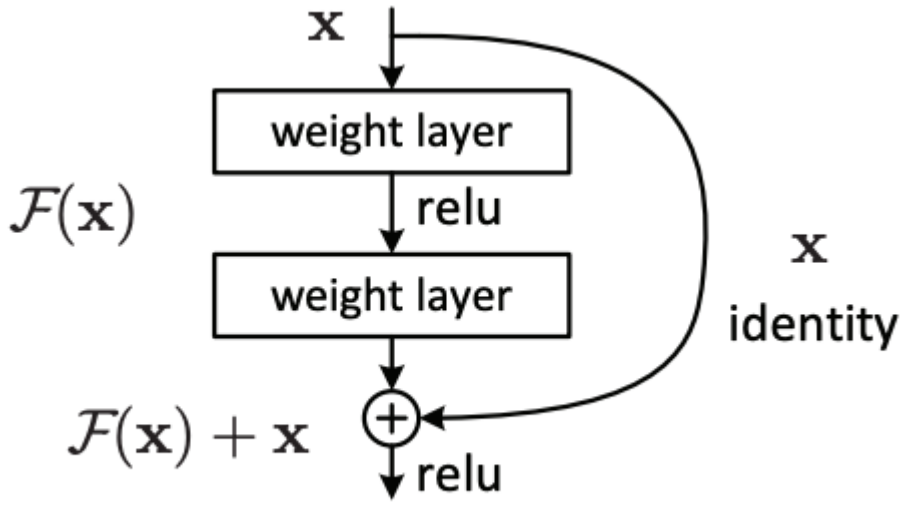


Figure 2. Residual learning: a building block.

Every to convolutions, I'm going to add together the input to those two convolutions with the result of those two convolutions." In other words, he's saying instead of saying:

$$Output = Conv2(Conv1(x))$$

Instead, he's saying:

$$Output = x + Conv2(Conv1(x))$$

His theory was 56 layers worth of convolutions in that has to be at least good as the 20 layer version because it could always just set conv2 and conv1 to a bunch of 0 weights for everything except for the first 20 layers because the X (i.e. the input) could just go straight through. So this thing here is (as you see) called an **identity connection**. It's the identity function - nothing happens at all. It's also known as a **skip connection**.

So that was the theory. That's what the paper describes as the intuition behind this is what would happen if we created something which has to train at least as well as a 20 layer neural network because it kind of contains that 20 layer neural network. There's literally a path you can just skip over all the convolutions. So what happens?

What happened was he won ImageNet that year. He easily won ImageNet that year. In fact, even today, we had that record-breaking result on ImageNet speed training ourselves in the last year, we used this too. ResNet has been revolutionary.

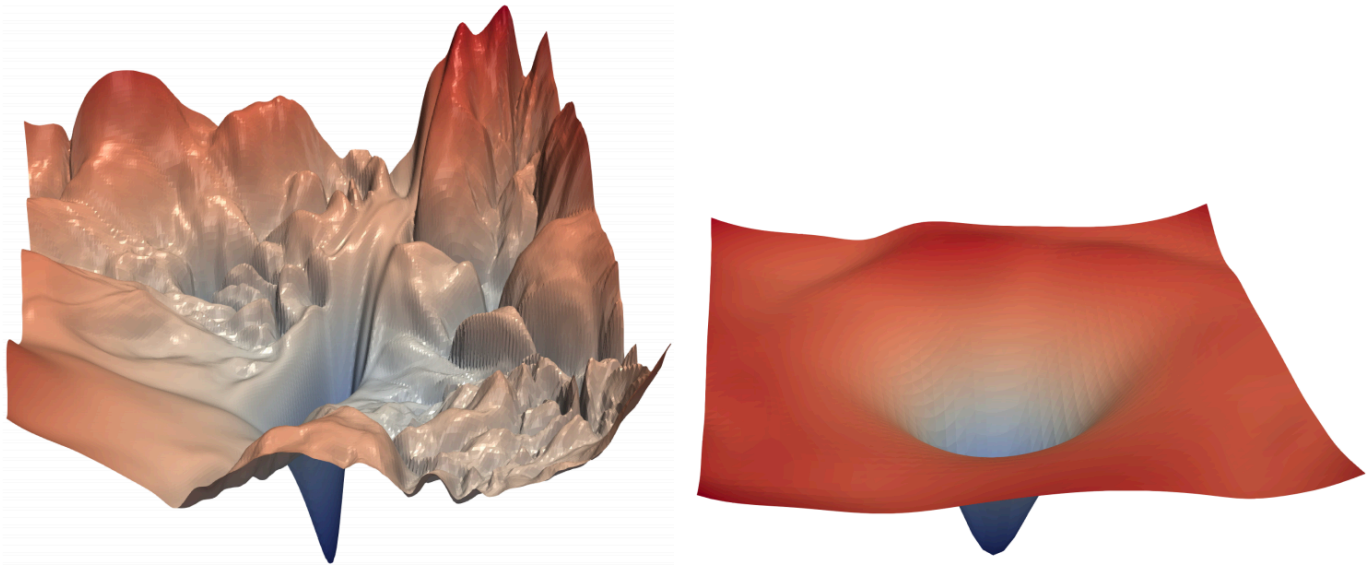
ResBlock Trick 20:36

Here's a trick if you're interested in doing some research. Anytime you find some model for anything whether it's medical image segmentation or some kind of GAN or whatever and it was written a couple of years ago, they might have forgotten to put ResBlocks in. Figure 2 is what we normally call a ResBlock. They might have forgotten to put ResBlocks in. So replace their convolutional path with a bunch of ResBlocks and you will almost always get better results faster. It's a good trick.

Visualizing the Loss Landscape of Neural Nets [21:16]

At NeurIPS, which Rachel, I, David, and Sylvain all just came back from, we saw a new presentation where they

actually figured out how to visualize the loss surface of a neural net which is really cool. This is a fantastic paper and anybody who's watching this lesson 7 is at a point where they will understand the most of the important concepts in this paper. You can read this now. You won't necessarily get all of it, but I'm sure you'll get it enough to find it interesting.



(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.

The big picture was this one. Here's what happens if you if you draw a picture where x and y here are two projections of the weight space, and z is the loss. As you move through the weight space, a 56 layer neural network without skip connections is very very bumpy. That's why this got nowhere because it just got stuck in all these hills and valleys. The exact same network with identity connections (i.e. with skip connections) has this loss landscape (on the right). So it's kind of interesting how Kaiming He recognized back in 2015 this shouldn't happen, here's a way that must fix it and it took three years before people were able to say oh this is kind of why it fixed it. It kind of reminds me of the batch norm discussion we had a couple of weeks ago that people realizing a little bit after the fact sometimes what's going on and why it helps.

```
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))
```

In our code, we can create a ResBlock in just the way I described. We create a `nn.Module`, we create two conv layers (remember, a `conv_layer` is Conv2d, ReLU, batch norm), so create two of those and then in forward we go `conv1(x)`, `conv2` of that and then add `x`.

```
help(res_block)
```

```
Help on function res_block in module fastai.layers:
```

```
res_block(nf, dense:bool=False, norm_type:Union[fastai.layers.NormType, NoneType]
    Resnet block of `nf` features.
```

There's a `res_block` function already in fast.ai so you can just call `res_block` instead, and you just pass in something saying how many filters you want.

```
model = nn.Sequential(
    conv2(1, 8),
    res_block(8),
    conv2(8, 16),
    res_block(16),
    conv2(16, 32),
    res_block(32),
    conv2(32, 16),
    res_block(16),
    conv2(16, 10),
    Flatten()
)
```

There's the ResBlock that I defined in our notebook, and so with that ResBlock, I've just copied the previous CNN and after every conv2 except the last one, I added a res_block so this has now got three times as many layers, so it should be able to do more compute. But it shouldn't be any harder to optimize.

Let's just refactor it one more time. Since I go `conv2 res_block` so many times, let's just pop that into a little mini sequential model here and so I can refactor that like so:

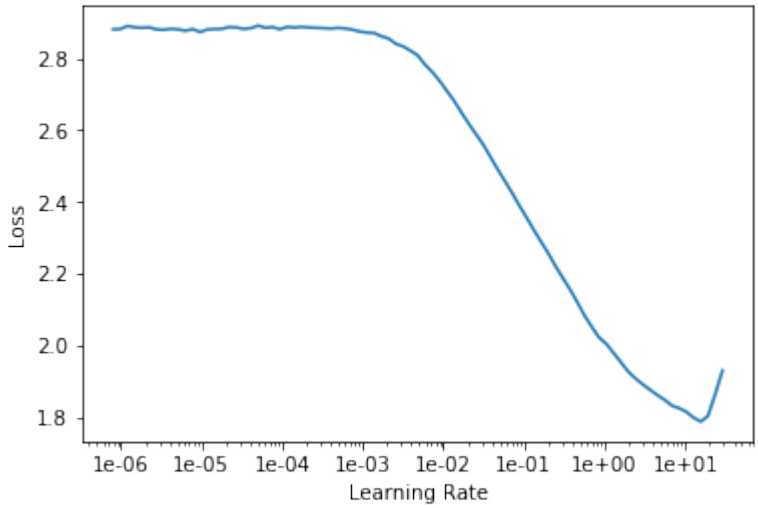
```
def conv_and_res(ni,nf): return nn.Sequential(conv2(ni, nf), res_block(nf))

model = nn.Sequential(
    conv_and_res(1, 8),
    conv_and_res(8, 16),
    conv_and_res(16, 32),
    conv_and_res(32, 16),
    conv2(16, 10),
    Flatten()
)
```

Keep refactoring your architectures if you're trying novel architectures because you'll make less mistakes. Very few people do this. Most research code you look at is clunky as all heck and people often make mistakes in that way, so don't do that. You're all coders, so use your coding skills to make life easier.

[24:47]

Okay, so there's my ResNet-ish architecture. `lr_find` as usual, `fit` for a while, and I get 99.54%.

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

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




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

Total time: 01:48

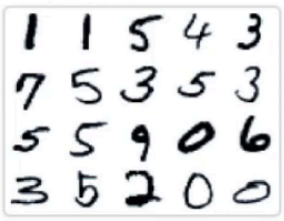
epoch	train_loss	valid_loss	accuracy
1	0.179228	0.102691	0.971300
2	0.111155	0.089420	0.973400
3	0.099729	0.053458	0.982500
4	0.085445	0.160019	0.951700
5	0.074078	0.063749	0.980800
6	0.057730	0.090142	0.973800
7	0.054202	0.034091	0.989200
8	0.043408	0.042037	0.986000
9	0.033529	0.023126	0.992800
10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

That’s interesting because we’ve trained this literally from scratch with an architecture we built from scratch, I didn’t look out this architecture anywhere. It’s just the first thing that came to mind. But in terms of where that puts us, 0.45% error is around about the state of the art for this data set as of three or four years ago.

Not secure | rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#4d4e495354

MNIST

who is the best in MNIST ?



MNIST 50 results collected

Units: error %

> Classify handwriten 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

Today MNIST considered a trivially easy dataset, so I'm not saying like wow, we've broken some records here. People have got beyond 0.45% error, but what I'm saying is this kind of ResNet is a genuinely extremely useful network still today. This is really all we use in our fast ImageNet training still. And one of the reasons as well is that it's so popular so the vendors of the library spend a lot of time optimizing it, so things tend to work fast. Where else, some more modern style architectures using things like separable or group convolutions tend not to actually train very quickly in practice.

```
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))
```

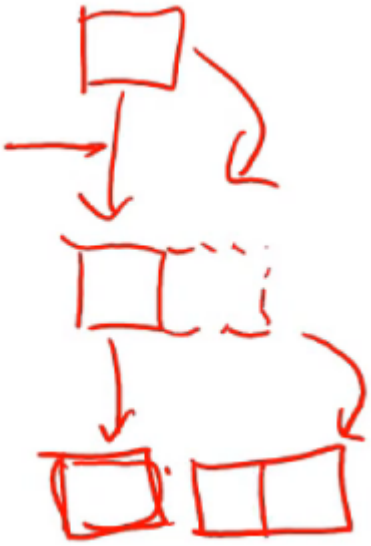
```
class MergeLayer(nn.Module):
    def __init__(self, dense:bool=False):
        super().__init__()
        self.dense=dense

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

```
def res_block(nf, dense:bool=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))
```

If you look at the definition of `res_block` in the fast.ai code, you'll see it looks a little bit different to this, and that's because I've created something called a `MergeLayer`. A `MergeLayer` is something which in the forward (just skip dense for a moment), the forward says `x+x.orig`. So you can see there's something ResNet-ish going on here. What is `x.orig`? Well, if you create a special kind of sequential model called a `SequentialEx` so this is like fast.ai's sequential extended. It's just like a normal sequential model, but we store the input in `x.orig`. So this `SequentialEx`, `conv_layer`, `conv_layer`, `MergeLayer`, will do exactly the same as `ResBlock`. So you can create your own variations of ResNet blocks very easily with this `SequentialEx` and `MergeLayer`.

There's something else here which is when you create your MergeLayer, you can optionally set `dense=True`, and what happens if you do? Well, if you do, it doesn't go `x+x.orig`, it goes `cat([x,x.orig])`. In other words, rather than putting a plus in this connection, it does a concatenate. That's pretty interesting because what happens is that you have your input coming in to your Res block, and once you use concatenate instead of plus, it's not called a Res block anymore, it's called a dense block. And it's not called a ResNet anymore, it's called a DenseNet.

The DenseNet was invented about a year after the ResNet, and if you read the DenseNet paper, it can sound incredibly complex and different, but actually it's literally identical but plus here is placed with cat. So you have your input coming into your dense block, and you've got a few convolutions in here, and then you've got some output coming out, and then you've got your identity connection, and remember it doesn't plus, it concats so the channel axis gets a little bit bigger. Then we do another dense block, and at the end of that, we have the result of the convolution as per usual, but this time the identity block is that big.

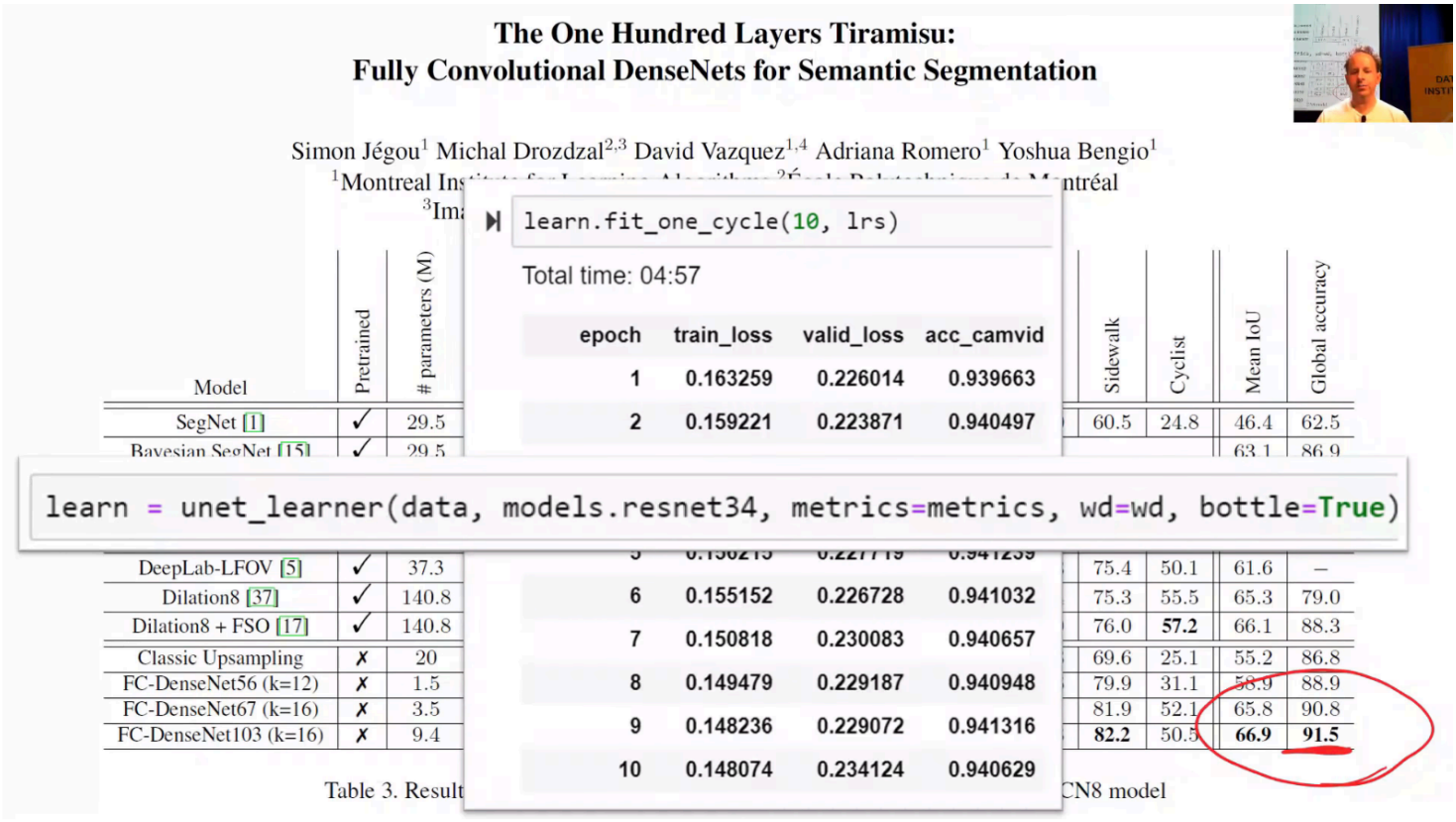


So you can see that what happens is that with dense blocks it's getting bigger and bigger and bigger, and kind of interestingly the exact input is still here. So actually, no matter how deep you get the original input pixels are still there, and the original layer 1 features are still there, and the original layer 2 features are still there. So as you can imagine, DenseNets are very memory intensive. There are ways to manage this. From time to time, you can have a regular convolution and it squishes your channels back down, but they are memory intensive. But, they have very few parameters. So for dealing with small datasets, you should definitely experiment with dense blocks and DenseNets. They tend to work really well on small datasets.

Also, because it's possible to keep those original input pixels all the way down the path, they work really well for segmentation. Because for segmentation, you want to be able to reconstruct the original resolution of your picture, so having all of those original pixels still there is a super helpful.

U-Net [30:16]

That's ResNets. One of the main reasons other than the fact that ResNets are awesome to tell you about them is that these skipped connections are useful in other places as well. They are particularly useful in other places in other ways of designing architectures for segmentation. So in building this lesson, I keep trying to take old papers and imagining like what would that person have done if they had access to all the modern techniques we have now, and I try to rebuild them in a more modern style. So I've been really rebuilding this next architecture we're going to look at called U-Net in a more modern style recently, and got to the point now I keep showing you this semantic segmentation paper with the state of the art for CamVid which was 91.5.



This week, I got it up to 94.1 using the architecture I'm about to show you. So we keep pushing this further and further and further. And it's really it was all about adding all of the modern tricks - many of which I'll show you today, some of which we will see in part 2.

What we're going to do to get there is we're going to use this U-Net. We've used a U-Net before. We used it when we did the CamVid segmentation but we didn't understand what it was doing. So we're now in a position where we can understand what it was doing. The first thing we need to do is to understand the basic idea of how you can do segmentation. If we go back to our CamVid notebook, in our CamVid notebook you'll remember that basically what we were doing is we were taking these photos and adding a class to every single pixel.

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

src = (SegmentationItemList.from_folder(path)
       .split_by_folder(valid='val')
       .label_from_func(get_y_fn, classes=codes))

data = (src.transform(get_transforms(), tfm_y=True)
        .databunch(bs=bs)
        .normalize(imagenet_stats))

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

So when you go `data.show_batch` for something which is a `SegmentationItemList`, it will automatically show you these color-coded pixels.

[32:35]

Here's the thing. In order to color code this as a pedestrian, but this as a bicyclist, it needs to know what it is. It needs to actually know that's what a pedestrian looks like, and it needs to know that's exactly where the pedestrian is, and this is the arm of the pedestrian and not part of their shopping basket. It needs to really understand a lot about this picture to do this task, and it really does do this task. When you looked at the results of our top model, I can't see a single pixel by looking at it by eye, I know there's a few wrong, but I can't see the ones that are wrong. It's that accurate. So how does it do that?

The way that we're doing it to get these really really good results is not surprisingly using pre-training.

```
name2id = {v:k for k,v in enumerate(codes)}
void_code = name2id['Void']

def acc_camvid(input, target):
    target = target.squeeze(1)
    mask = target != void_code
    return (input.argmax(dim=1)[mask]==target[mask]).float().mean()

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

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

So we start with a ResNet 34 and you can see that here `unet_learner(data, models.resnet34,...)`. If you don't say `pretrained=False` , by default, you get `pretrained=True` because … why not?

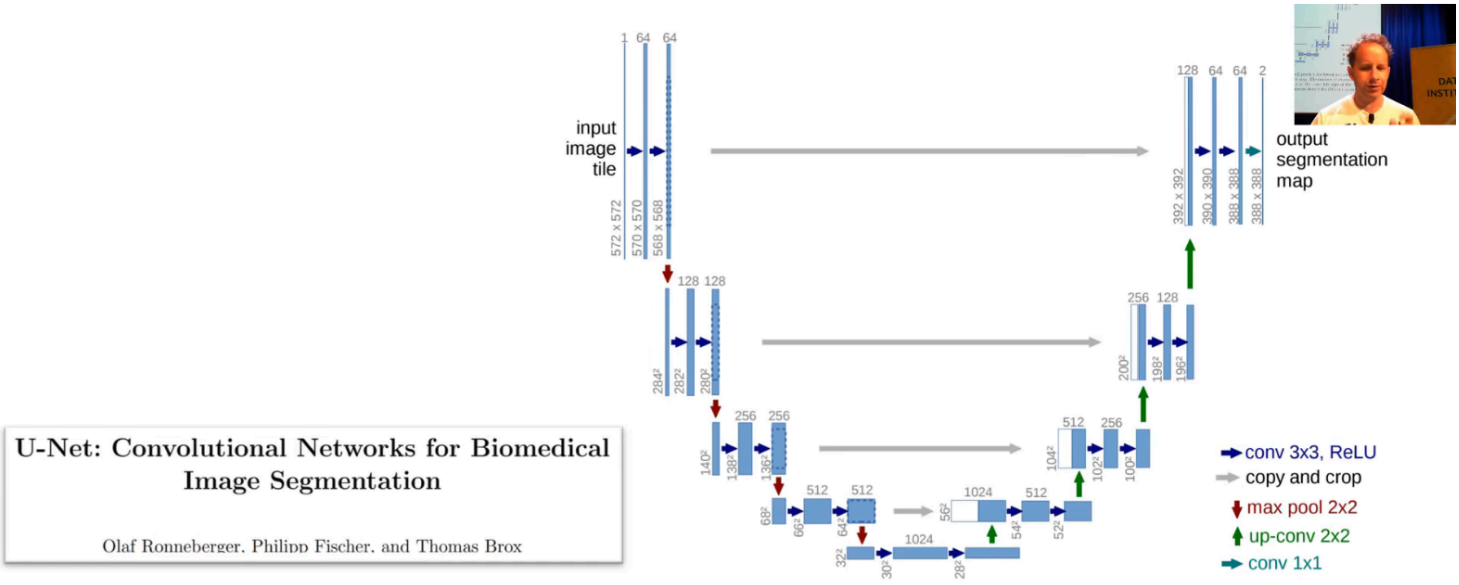


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows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

We start with a ResNet 34 which starts with a big image. In this case, this is from the U-Net paper. Their images, they started with one channel by 572 by 572. This is for medical imaging segmentation. After your stride 2 conv, they're doubling the number of channels to 128, and they're halving the size so they're now down to 280 by 280. In this original unit paper, they didn't add any padding. So they lost a pixel on each side each time they did a conv. That's why you are losing these two. But basically half the size, and then half the size, and then half the size, and then half the size, until they're down to 28 by 28 with 1024 channels.

So that's what the U-Net's downsampling path (the left half is called the downsampling path) look like. Ours is just a ResNet 34. So you can see it here `learn.summary()`, this is literally a ResNet 34. So you can see that the size keeps halving, channels keep going up and so forth.

```
learn.summary()
```

```
======================================================================
Layer (type)          Output Shape          Param #     Trainable
======================================================================
Conv2d                [8, 64, 180, 240]     9408        False
______________________________________________________________________
BatchNorm2d           [8, 64, 180, 240]     128         True
______________________________________________________________________
ReLU                  [8, 64, 180, 240]     0           False
______________________________________________________________________
MaxPool2d             [8, 64, 90, 120]      0           False
______________________________________________________________________
Conv2d                [8, 64, 90, 120]      36864       False
```

BatchNorm2d	[8, 64, 90, 120]	128	True
ReLU	[8, 64, 90, 120]	0	False
Conv2d	[8, 64, 90, 120]	36864	False
BatchNorm2d	[8, 64, 90, 120]	128	True
Conv2d	[8, 64, 90, 120]	36864	False
BatchNorm2d	[8, 64, 90, 120]	128	True
ReLU	[8, 64, 90, 120]	0	False
Conv2d	[8, 64, 90, 120]	36864	False
BatchNorm2d	[8, 64, 90, 120]	128	True
Conv2d	[8, 64, 90, 120]	36864	False
BatchNorm2d	[8, 64, 90, 120]	128	True
ReLU	[8, 64, 90, 120]	0	False
Conv2d	[8, 64, 90, 120]	36864	False
BatchNorm2d	[8, 64, 90, 120]	128	True
Conv2d	[8, 128, 45, 60]	73728	False
BatchNorm2d	[8, 128, 45, 60]	256	True
ReLU	[8, 128, 45, 60]	0	False
Conv2d	[8, 128, 45, 60]	147456	False
BatchNorm2d	[8, 128, 45, 60]	256	True
Conv2d	[8, 128, 45, 60]	8192	False
BatchNorm2d	[8, 128, 45, 60]	256	True
Conv2d	[8, 128, 45, 60]	147456	False
BatchNorm2d	[8, 128, 45, 60]	256	True
ReLU	[8, 128, 45, 60]	0	False
Conv2d	[8, 128, 45, 60]	147456	False

BatchNorm2d	[8, 128, 45, 60]	256	True
Conv2d	[8, 128, 45, 60]	147456	False
BatchNorm2d	[8, 128, 45, 60]	256	True
ReLU	[8, 128, 45, 60]	0	False
Conv2d	[8, 128, 45, 60]	147456	False
BatchNorm2d	[8, 128, 45, 60]	256	True
Conv2d	[8, 128, 45, 60]	147456	False
BatchNorm2d	[8, 128, 45, 60]	256	True
ReLU	[8, 128, 45, 60]	0	False
Conv2d	[8, 128, 45, 60]	147456	False
BatchNorm2d	[8, 128, 45, 60]	256	True
Conv2d	[8, 256, 23, 30]	294912	False
BatchNorm2d	[8, 256, 23, 30]	512	True
ReLU	[8, 256, 23, 30]	0	False
Conv2d	[8, 256, 23, 30]	589824	False
BatchNorm2d	[8, 256, 23, 30]	512	True
Conv2d	[8, 256, 23, 30]	32768	False
BatchNorm2d	[8, 256, 23, 30]	512	True
Conv2d	[8, 256, 23, 30]	589824	False
BatchNorm2d	[8, 256, 23, 30]	512	True
ReLU	[8, 256, 23, 30]	0	False
Conv2d	[8, 256, 23, 30]	589824	False
BatchNorm2d	[8, 256, 23, 30]	512	True
Conv2d	[8, 256, 23, 30]	589824	False
BatchNorm2d	[8, 256, 23, 30]	512	True
ReLU	[8, 256, 23, 30]	0	False

Conv2d	[8, 256, 23, 30]	589824	False
BatchNorm2d	[8, 256, 23, 30]	512	True
Conv2d	[8, 256, 23, 30]	589824	False
BatchNorm2d	[8, 256, 23, 30]	512	True
ReLU	[8, 256, 23, 30]	0	False
Conv2d	[8, 256, 23, 30]	589824	False
BatchNorm2d	[8, 256, 23, 30]	512	True
Conv2d	[8, 256, 23, 30]	589824	False
BatchNorm2d	[8, 256, 23, 30]	512	True
ReLU	[8, 256, 23, 30]	0	False
Conv2d	[8, 256, 23, 30]	589824	False
BatchNorm2d	[8, 256, 23, 30]	512	True
Conv2d	[8, 256, 23, 30]	589824	False
BatchNorm2d	[8, 256, 23, 30]	512	True
ReLU	[8, 256, 23, 30]	0	False
Conv2d	[8, 256, 23, 30]	589824	False
BatchNorm2d	[8, 256, 23, 30]	512	True
Conv2d	[8, 512, 12, 15]	1179648	False
BatchNorm2d	[8, 512, 12, 15]	1024	True
ReLU	[8, 512, 12, 15]	0	False
Conv2d	[8, 512, 12, 15]	2359296	False
BatchNorm2d	[8, 512, 12, 15]	1024	True
Conv2d	[8, 512, 12, 15]	131072	False
BatchNorm2d	[8, 512, 12, 15]	1024	True
Conv2d	[8, 512, 12, 15]	2359296	False

BatchNorm2d	[8, 512, 12, 15]	1024	True
ReLU	[8, 512, 12, 15]	0	False
Conv2d	[8, 512, 12, 15]	2359296	False
BatchNorm2d	[8, 512, 12, 15]	1024	True
Conv2d	[8, 512, 12, 15]	2359296	False
BatchNorm2d	[8, 512, 12, 15]	1024	True
ReLU	[8, 512, 12, 15]	0	False
Conv2d	[8, 512, 12, 15]	2359296	False
BatchNorm2d	[8, 512, 12, 15]	1024	True
BatchNorm2d	[8, 512, 12, 15]	1024	True
ReLU	[8, 512, 12, 15]	0	False
Conv2d	[8, 1024, 12, 15]	4719616	True
ReLU	[8, 1024, 12, 15]	0	False
Conv2d	[8, 512, 12, 15]	4719104	True
ReLU	[8, 512, 12, 15]	0	False
Conv2d	[8, 1024, 12, 15]	525312	True
PixelShuffle	[8, 256, 24, 30]	0	False
ReplicationPad2d	[8, 256, 25, 31]	0	False
AvgPool2d	[8, 256, 24, 30]	0	False
ReLU	[8, 1024, 12, 15]	0	False
BatchNorm2d	[8, 256, 23, 30]	512	True
Conv2d	[8, 512, 23, 30]	2359808	True
ReLU	[8, 512, 23, 30]	0	False
Conv2d	[8, 512, 23, 30]	2359808	True
ReLU	[8, 512, 23, 30]	0	False
ReLU	[8, 512, 23, 30]	0	False

Conv2d	[8, 1024, 23, 30]	525312	True
PixelShuffle	[8, 256, 46, 60]	0	False
ReplicationPad2d	[8, 256, 47, 61]	0	False
AvgPool2d	[8, 256, 46, 60]	0	False
ReLU	[8, 1024, 23, 30]	0	False
BatchNorm2d	[8, 128, 45, 60]	256	True
Conv2d	[8, 384, 45, 60]	1327488	True
ReLU	[8, 384, 45, 60]	0	False
Conv2d	[8, 384, 45, 60]	1327488	True
ReLU	[8, 384, 45, 60]	0	False
ReLU	[8, 384, 45, 60]	0	False
Conv2d	[8, 768, 45, 60]	295680	True
PixelShuffle	[8, 192, 90, 120]	0	False
ReplicationPad2d	[8, 192, 91, 121]	0	False
AvgPool2d	[8, 192, 90, 120]	0	False
ReLU	[8, 768, 45, 60]	0	False
BatchNorm2d	[8, 64, 90, 120]	128	True
Conv2d	[8, 256, 90, 120]	590080	True
ReLU	[8, 256, 90, 120]	0	False
Conv2d	[8, 256, 90, 120]	590080	True
ReLU	[8, 256, 90, 120]	0	False
ReLU	[8, 256, 90, 120]	0	False
Conv2d	[8, 512, 90, 120]	131584	True
PixelShuffle	[8, 128, 180, 240]	0	False
ReplicationPad2d	[8, 128, 181, 241]	0	False

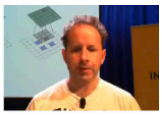
AvgPool2d	[8, 128, 180, 240]	0	False
ReLU	[8, 512, 90, 120]	0	False
BatchNorm2d	[8, 64, 180, 240]	128	True
Conv2d	[8, 96, 180, 240]	165984	True
ReLU	[8, 96, 180, 240]	0	False
Conv2d	[8, 96, 180, 240]	83040	True
ReLU	[8, 96, 180, 240]	0	False
ReLU	[8, 192, 180, 240]	0	False
Conv2d	[8, 384, 180, 240]	37248	True
PixelShuffle	[8, 96, 360, 480]	0	False
ReplicationPad2d	[8, 96, 361, 481]	0	False
AvgPool2d	[8, 96, 360, 480]	0	False
ReLU	[8, 384, 180, 240]	0	False
MergeLayer	[8, 99, 360, 480]	0	False
Conv2d	[8, 49, 360, 480]	43708	True
ReLU	[8, 49, 360, 480]	0	False
Conv2d	[8, 99, 360, 480]	43758	True
ReLU	[8, 99, 360, 480]	0	False
MergeLayer	[8, 99, 360, 480]	0	False
Conv2d	[8, 12, 360, 480]	1200	True

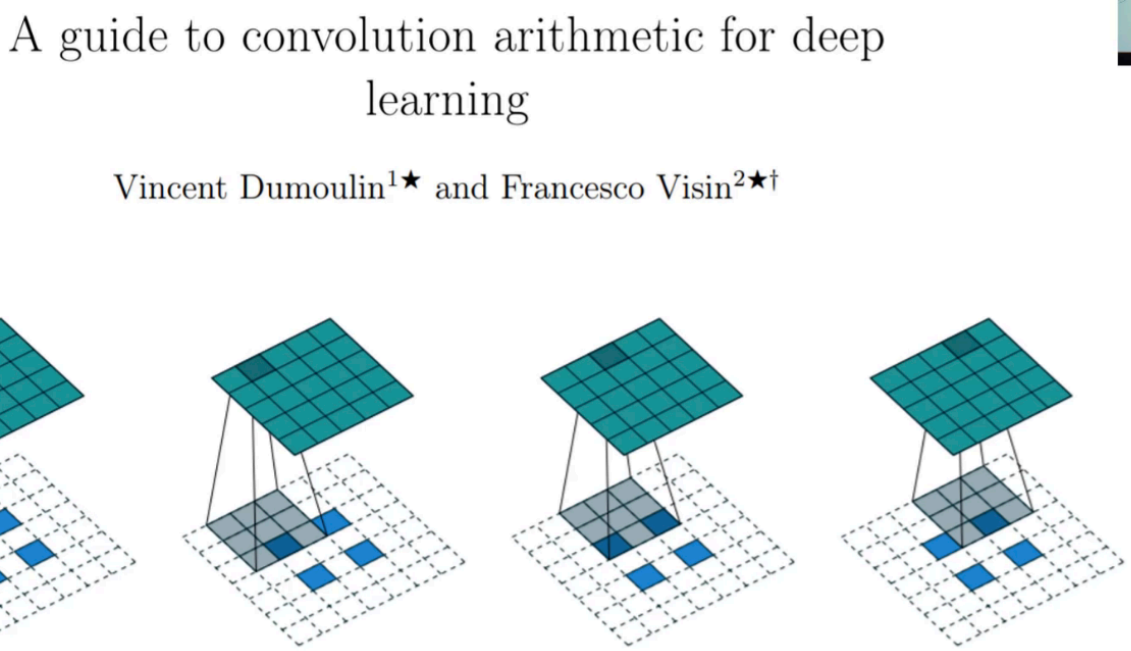
```
Total params:  41133018
Total trainable params:  19865370
Total non-trainable params:  21267648
```

Eventually, you've got down to a point where, if you use U-Net architecture, it's 28 by 28 with 1,024 channels. With the ResNet architecture with a 224 pixel input, it would be 512 channels by 7 by 7. So it's a pretty small grid size on this feature map. Somehow, we've got to end up with something which is the same size as our original picture. So how do we do that? How do you do computation which increases the grid size? Well, we don't have a way to do that in our current bag of tricks. We can use a stride one conv to do computation and keeps grid size or a stride 2 conv to do computation and halve the grid size.

[35:58]

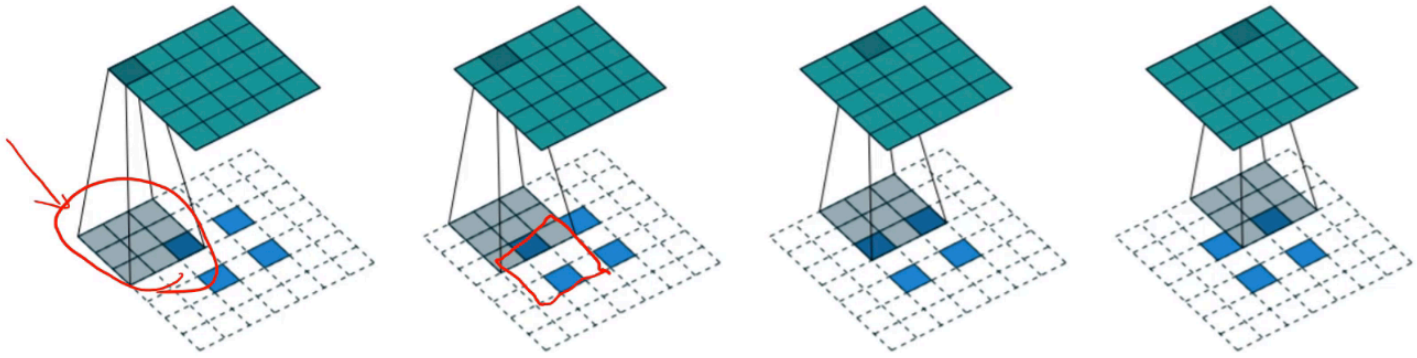
So how do we double the grid size? We do a **stride half conv**, also known as a **deconvolution**, also known as a **transpose convolution**.





There is a fantastic paper called A guide to convolution arithmetic for deep learning that shows a great picture of exactly what does a 3x3 kernel stride half conv look like. And it's literally this. If you have a 2x2 input, so the blue squares are the 2x2 input, you add not only 2 pixels of padding all around the outside, but you also add a pixel of padding between every pixel. So now if we put this 3x3 kernel here, and then here, and then here, you see how the 3x3 kernels just moving across it in the usual way, you will end up going from a 2x2 output to a 5x5 output. If you only added one pixel of padding around the outside, you would end up with a 4x4 output.

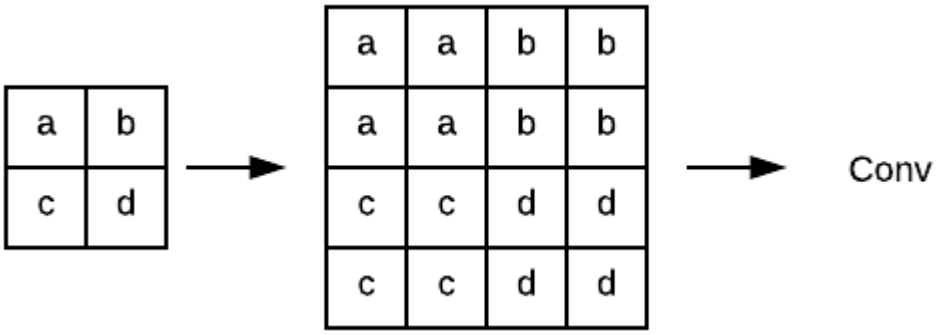
This is how you can increase the resolution. This was the way people did it until maybe a year or two ago. There's another trick for improving things you find online. Because this is actually a dumb way to do it. And it's kind of obvious it's a dumb way to do it for a couple of reasons. One is that, have a look at the shaded area on the left, nearly all of those pixels are white. They're nearly all zeros. What a waste. What a waste of time, what a waste of computation. There's just nothing going on there.



Also, this one when you get down to that 3x3 area, 2 out of the 9 pixels are non-white, but this left one, 1 out of the

9 are non-white. So there's different amounts of information going into different parts of your convolution. So it just doesn't make any sense to throw away information like this and to do all this unnecessary computation and have different parts of the convolution having access to different amounts of information.

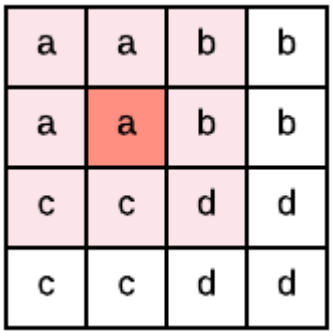
What people generally do nowadays is something really simple. If you have a, let's say, 2x2 input with these are your pixel values (a, b, c, d) and you want to create a 4x4, why not just do this?



So I've now up scaled from 2 by 2 to 4 by 4. I haven't done any interesting computation, but now on top of that, I could just do a stride 1 convolution, and now I have done some computation.

An upsample, this is called **nearest neighbor interpolation**. That's super fast which is nice. So you can do a nearest neighbor interpolation, and then a stride 1 conv, and now you've got some computation which is actually using there's no zeros in upper left 4x4, this (one pixel to the right) is kind of nice because it gets a mixture of A's and B's which is kind of what you would want and so forth.

Another approach is instead of using nearest neighbor interpolation, you can use bilinear interpolation which basically means instead of copying A to all those different cells you take a weighted average of the cells around it.

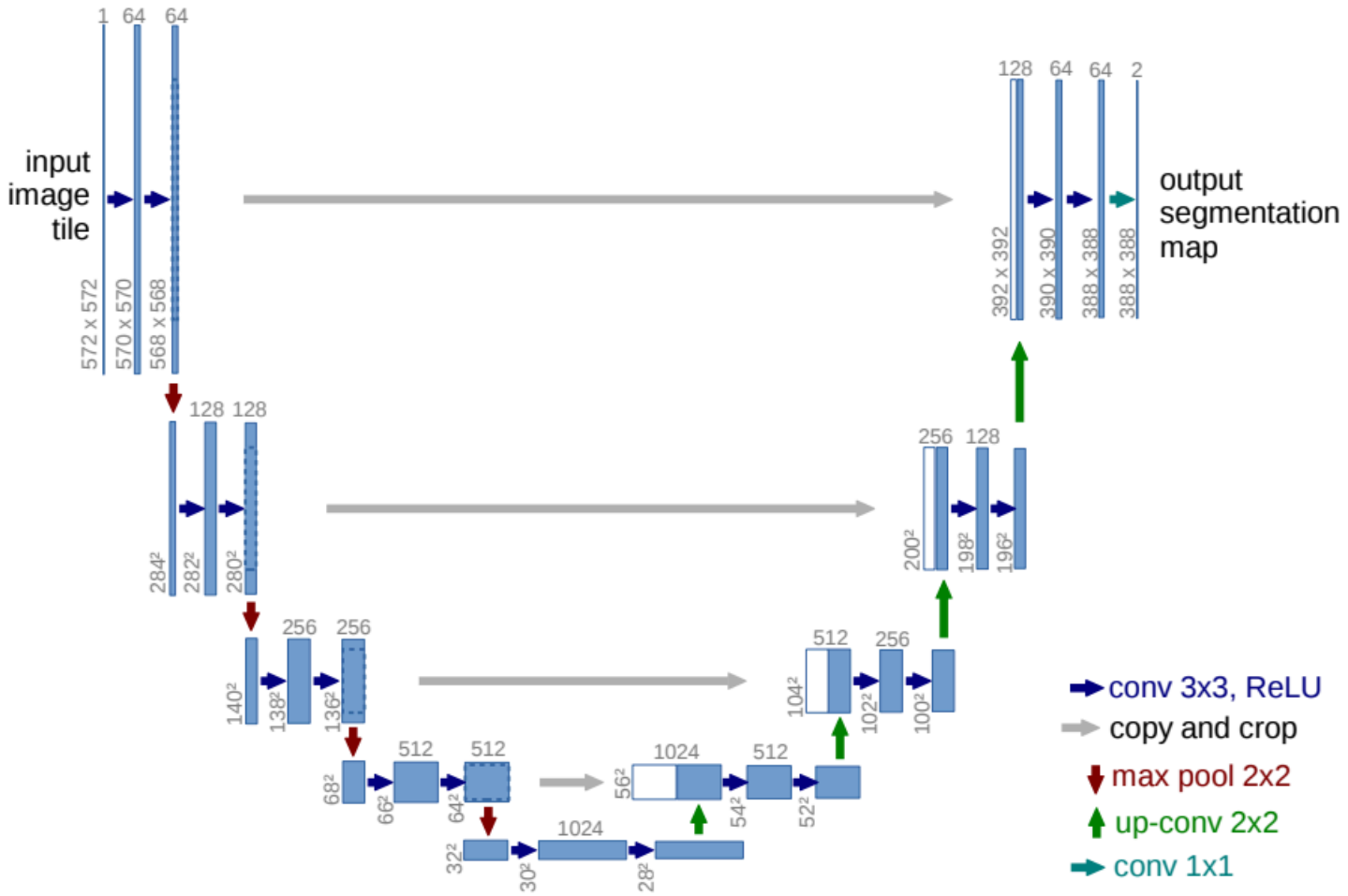


For example if you were looking at what should go here (red), you would kind of go, oh it's about 3 A's, 2 C's, 1 D, and 2 B's, and you take the average, not exactly, but roughly just a weighted average. Bilinear interpolation, you'll find all over the place - it's pretty standard technique. Anytime you look at a picture on your computer screen and change its size, it's doing bilinear interpolation. So you can do that and then a stride 1 conv. So that was what people were using, well, what people still tend to use. That's as much as I going to teach you this part. In part 2, we will actually learn what the fast.ai library is actually doing behind the scenes which is something called a **pixel shuffle** also known as **sub pixel convolutions**. It's not dramatically more complex but complex enough that I won't cover it today. They're the same basic idea. All of these things is something which is basically letting us do a convolution that ends up with something that's twice the size.

That gives us our upsampling path. That lets us go from 28 by 28 to 54 by 54 and keep on doubling the size, so that's good. And that was it until U-Net came along. That's what people did and it didn't work real well which is not surprising because like in this 28 by 28 feature map, how the heck is it going to have enough information to

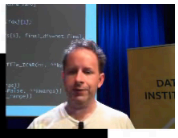
reconstruct a 572 by 572 output space? That's a really tough ask. So you tended to end up with these things that lack fine detail.

[41:45]



So what Olaf Ronneberger et al. did was they said hey let's add a skip connection, an identity connection, and amazingly enough, this was before ResNets existed. So this was like a really big leap, really impressive. But rather than adding a skip connection that skipped every two convolutions, they added skip connections where these gray lines are. In other words, they added a skip connection from the same part of the downsampling path to the same-sized bit in the upsampling path. And they didn't add, that's why you can see the white and the blue next to each other, they didn't add they concatenated. So basically, these are like dense blocks, but the skip connections are skipping over larger and larger amounts of the architecture so that over here (top gray arrow), you've nearly got the input pixels themselves coming into the computation of these last couple of layers. That's going to make it super handy for resolving the fine details in these segmentation tasks because you've literally got all of the fine details. On the downside, you don't have very many layers of computation going on here (top right), just four. So you better hope that by that stage, you've done all the computation necessary to figure out is this a bicyclist or is this a pedestrian, but you can then add on top of that something saying is this exact pixel where their nose finishes or is at the start of the tree. So that works out really well and that's U-Net.

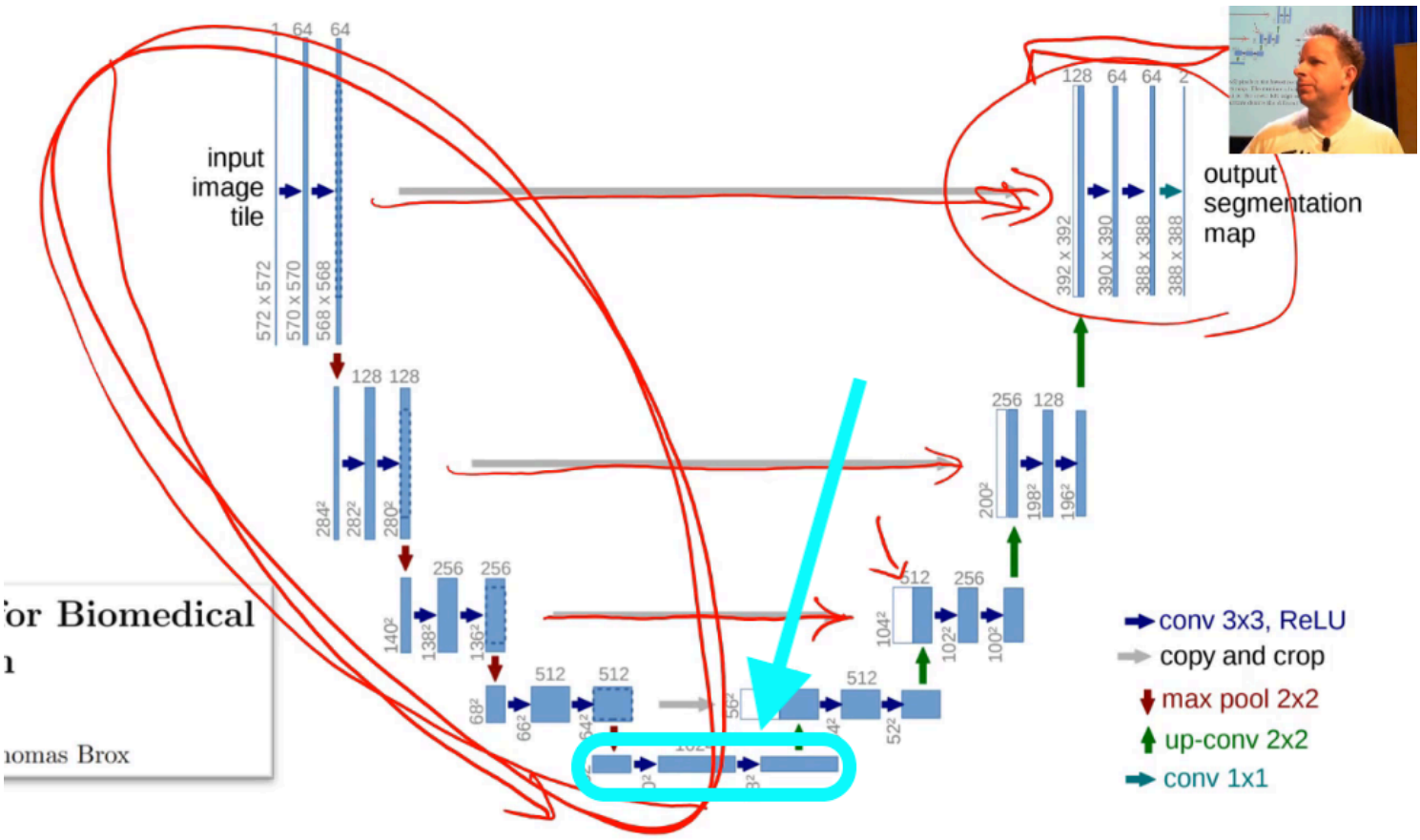
[43:33]



```
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)
```

This is the unit code from fast.ai, and the key thing that comes in is the encoder. The encoder refers to the downsampling part of U-Net, in other words, in our case a ResNet 34. In most cases they have this specific older style architecture, but like I said, replace any older style architecture bits with ResNet bits and life improves particularly if they're pre-trained. So that certainly happened for us. So we start with our encoder.

So our `layers` of our U-Net is an encoder, then batch norm, then ReLU, and then `middle_conv` which is just (`conv_layer`, `conv_layer`). Remember, `conv_layer` is a conv, ReLU, batch norm in fast.ai. So that middle con is these two extra steps here at the bottom:



It's doing a little bit of computation. It's kind of nice to add more layers of computation where you can. So encoder, batch norm, ReLU, and then two convolutions. Then we enumerate through these indexes (`sfs_idxs`). What are these indexes? I haven't included the code but these are basically we figure out what is the layer number where each of these stride 2 convs occurs and we just store it in an array of indexes. Then we can loop through that and we can basically say for each one of those points create a `UnetBlock` telling us how many upsampling channels that are and how many cross connection. These gray arrows are called cross connections - at least that's what I call them.

[45:16]

That's really the main works going on in the in the `UnetBlock`. As I said, there's quite a few tweaks we do as well as the fact we use a much better encoder, we also use some tweaks in all of our upsampling using this pixel shuffle, we use another tweak called ICNR, and then another tweak which I just did in the last week is to not just take the result of the convolutions and pass it across, but we actually grab the input pixels and make them another cross connection. That's what this `last_cross` is here. You can see we're literally appending a `res_block` with the original inputs (so you can see our `MergeLayer`).

```
class UnetBlock(nn.Module):
    "A quasi-UNet 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))
```

So really all the work is going on in a `UnetBlock` and `UnetBlock` has to store the the activations at each of these downsampling points, and the way to do that, as we learn in the last lesson, is with hooks. So we put hooks into the ResNet 34 to store the activations each time there's a stride 2 conv, and so you can see here, we grab the hook (`self.hook =hook`). And we grab the result of the stored value in that hook, and we literally just go `torch.cat` so we concatenate the upsampled convolution with the result of the hook which we chuck through batch norm, and then we do two convolutions to it.

Actually, something you could play with at home is pretty obvious here (the very last line). Anytime you see two convolutions like this, there's an obvious question is what if we used a ResNet block instead? So you could try replacing those two convs with a ResNet block, you might find you get even better results. They're the kind of things I look for when I look at an architecture is like "oh, two convs in a row, probably should be a ResNet block.

Okay, so that's U-Net and it's amazing to think it preceded ResNet, preceded DenseNet. It wasn't even published in a major machine learning venue. It was actually published in MICCAI which is a specialized medical image computing conference. For years, it was largely unknown outside of the medical imaging community. Actually, what happened was Kaggle competitions for segmentation kept on being easily won by people using U-Nets and that was the first time I saw it getting noticed outside the medical imaging community. Then gradually, a few people in the academic machine learning community started noticing, and now everybody loves U-Net, which I'm glad because it's just awesome.

So identity connections, regardless of whether they're a plus style or a concat style, are incredibly useful. They can basically get us close to the state of the art on lots of important tasks. So I want to use them on another task now.

Image restoration [48:31]

The next task I want to look at is image restoration. Image restoration refers to starting with an image and this time we're not going to create a segmentation mask but we're going to try and create a better image. There's lots of kind of versions of better - there could be different image. The kind of things we can do with this kind of image generation would be:

- take a low res image make it high res
- take a black-and-white image make a color
- take an image where something's being cut out of it and trying to replace the cutout thing
- take a photo and try and turn it into what looks like a line drawing

- take a photo and try and talk like it look like a Monet painting

These are all examples of image to image generation tasks which you'll know how to do after this part class.

So in our case, we're going to try to do image restoration which is going to start with low resolution, poor quality JPEGs, with writing written over the top of them, and get them to replace them with high resolution, good quality pictures in which the text has been removed.

Question: Why do you concat before calling conv2(conv1(x)), not after? [49:50]

Because if you did your convs before you concat, then there's no way for the channels of the two parts to interact with each other. So remember in a 2D conv, it's really 3D. It's moving across 2 dimensions but in each case it's doing a dot product of all 3 dimensions of a rank 3 tensor (row by column by channel). So generally speaking, we want as much interaction as possible. We want to say this part of the downsampling path and this part of the upsampling path, if you look at the combination of them, you find these interesting things. So generally you want to have as many interactions going on as possible in each computation that you do.

Question: How does concatenating every layer together in a DenseNet work when the size of the image/feature maps is changing through the layers? [50:54]

That's a great question. If you have a stride 2 conv, you can't keep DenseNet-ing. That's what actually happens in a DenseNet is you kind of go like dense block, growing, dense block, growing, dense block, growing, so you are getting more and more channels. Then you do a stride 2 conv without a dense block, and so now it's kind of gone. Then you just do a few more dense blocks and then it's gone. So in practice, a dense block doesn't actually keep all the information all the way through, but just up until every one of these stride 2 convs. There's various ways of doing these bottlenecking layers where you're basically saying hey let's reset. It also helps us keep memory under control because at that point we can decide how many channels we actually want.

Back to image restoration [52:01]

lesson7-superres-gan.ipynb

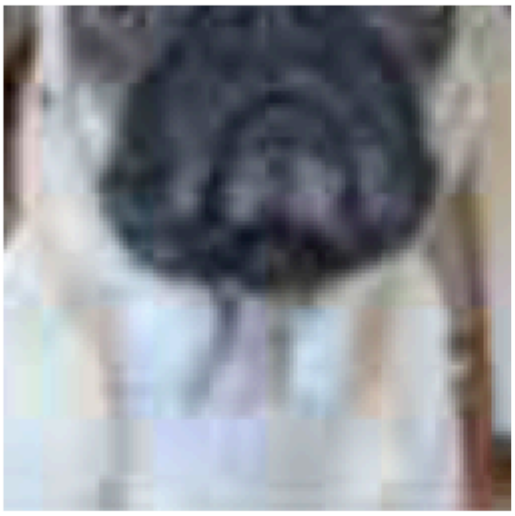
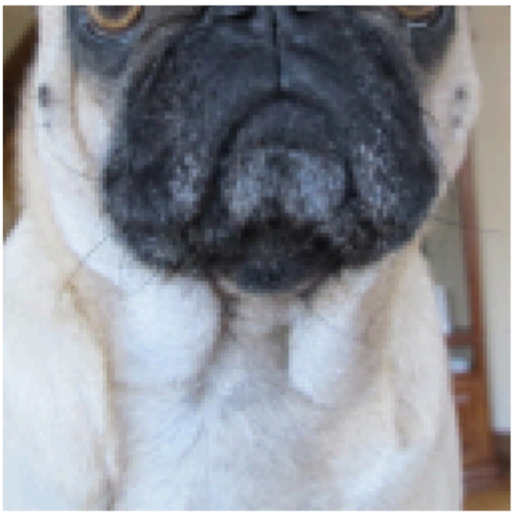
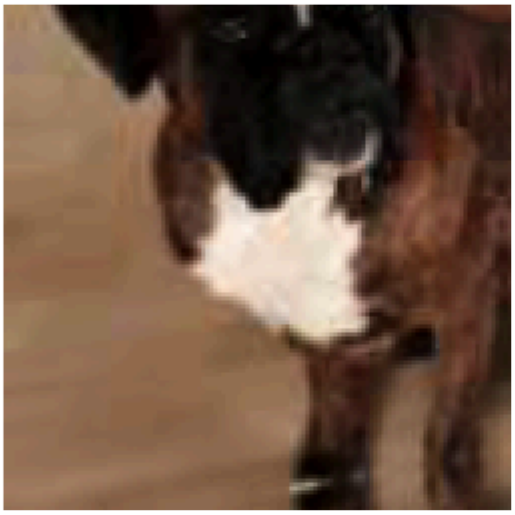
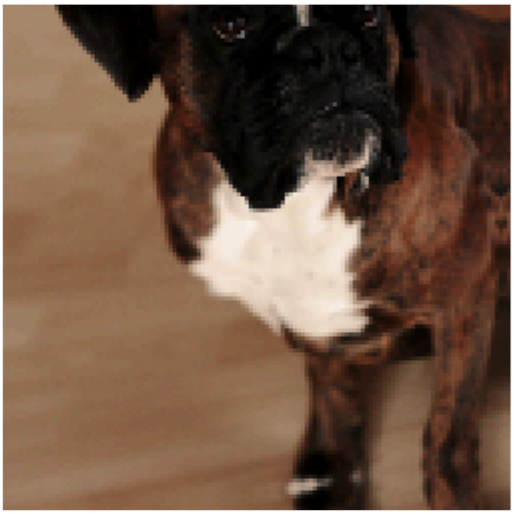
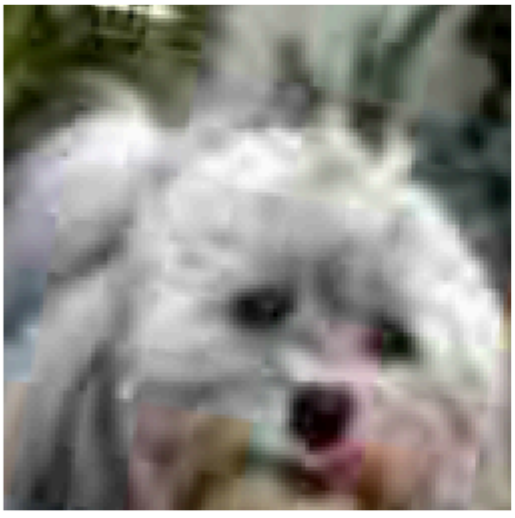
In order to create something which can turn crappy images into nice images, we needed dataset containing nice versions of images and crappy versions of the same images. The easiest way to do that is to start with some nice images and "crappify" them.

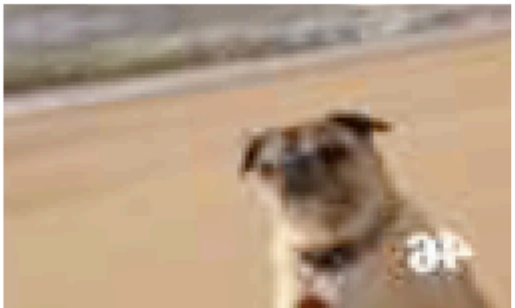
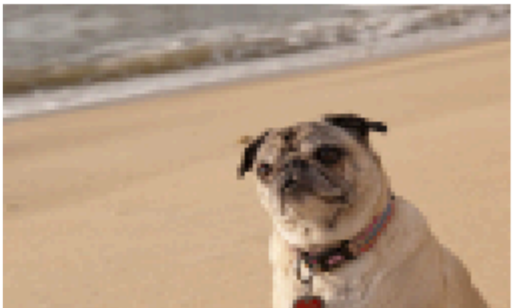
```
from PIL import Image, ImageDraw, ImageFont

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

The way to crappify them is to create a function called crappify which contains your crappification logic. My crappification logic, you can pick your own, is that:

- I open up my nice image
- I resize it to be really small 96 by 96 pixels with bilinear interpolation
- I then pick a random number between 10 and 70
- I draw that number into my image at some random location
- Then I save that image with a JPEG quality of that random number.

A JPEG quality of 10 is like absolute rubbish, a JPEG a quality of 70 is not bad at all. So I end up with high quality images and low quality images that look something like these:

You can see this one (bottom row) there's the number, and this is after transformation so that's why it's been flipped and you won't always see the number because we're zooming into them, so a lot of the time, the number is cropped out.

It's trying to figure out how to take this incredibly JPEG artifactory thing with text written over the top, and turn it into this (image on the right). I'm using the Oxford pets data set again. The same one we used in lesson one. So there's nothing more high quality than pictures of dogs and cats, I think we can all agree with that.

`parallel` [53:48]

The crappification process can take a while, but fast.ai has a function called `parallel`. If you pass `parallel` a function name and a list of things to run that function on, it will run that function on them all in parallel. So this actually can run pretty quickly.

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

The way you write this `crappify` function is where you get to do all the interesting stuff in this assignment. Try and think of an interesting crappification which does something that you want to do. So if you want to colorize black-and-white images, you would replace it with black-and-white. If you want something which can take large cutout blocks of image and replace them with kind of hallucinatin image, add a big black box to these. If you want something which can take old families photos scans that have been like folded up and have crinkles in, try and find a way of adding dust prints and crinkles and so forth.

Anything that you don't include in `crappify`, your model won't learn to fix. Because every time it sees that in your photos, the input and output will be the same. So it won't consider that to be something worthy of fixing.

[55:09]


We now want to create a model which can take an input photo that looks like that (left) and output something that looks like that (right). So obviously, what we want to do is use U-Net because we already know that U-Net can do exactly that kind of thing, and we just need to pass the U-Net that data.

```
arch = models.resnet34
src = ImageImageList.from_folder(path_lr).random_split_by_pct(0.1, seed=42)
```

```
def get_data(bs,size):
    data = (src.label_from_func(lambda x: path_hr/x.name)
           .transform(get_transforms(max_zoom=2.), size=size, tfm_y=True)
           .databunch(bs=bs).normalize(imagenet_stats, do_y=True))

    data.c = 3
    return data

data_gen = get_data(bs,size)
```

Our data is just literally the file names from each of those two folders, do some transforms, data bunch, normalize. We'll use ImageNet stats because we're going to use a pre-trained model. Why are we using a pre-trained model? Because if you're going to get rid of this 46, you need to know what probably was there, and to know what probably was there you need to know what this is a picture of. Otherwise, how can you possibly know what it ought to look like. So let's use a pre-trained model that knows about these kinds of things.

```
wd = 1e-3

y_range = (-3.,3.)

loss_gen = MSELossFlat()

def create_gen_learner():
    return unet_learner(data_gen, arch, wd=wd, blur=True, norm_type=NormType.Weig
                        self_attention=True, y_range=y_range, loss_func=loss_gen

learn_gen = create_gen_learner()
```

So we created our U-Net with that data, the architecture is ResNet 34. These three things (`blur`, `norm_type`, `self_attention`) are important and interesting and useful, but I'm going to leave them to part 2. For now, you should always include them when you use U-Net for this kind of problem.

This whole thing, I'm calling a "generator". It's going to generate. This is generative modeling. There's not a really formal definition, but it's basically something where the thing we're outputting is like a real object, in this case an image - it's not just a number. So we're going to create a generator learner which is this U-Net learner, and then we can fit. We're using MSE loss, so in other words what's the mean squared error between the actual pixel value that it should be in the pixel value that we predicted. MSE loss normally expects two vectors. In our case, we have two images so we have a version called MSE loss flat which simply flattens out those images into a big long vector. There's never any reason not to use this, even if you do have a vector, it works fine, if you don't have a vector, it'll also work fine.

```
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

We're already down to 0.05 mean squared error on the pixel values which is not bad after 1 minute 35. Like all things in fast.ai pretty much, because we're doing transfer learning by default when you create this, it'll freeze the

the pre-trained part. And the pre-trained part of a U-Net is the downsampling part. That's where the ResNet is.

```
learn_gen.unfreeze()
```

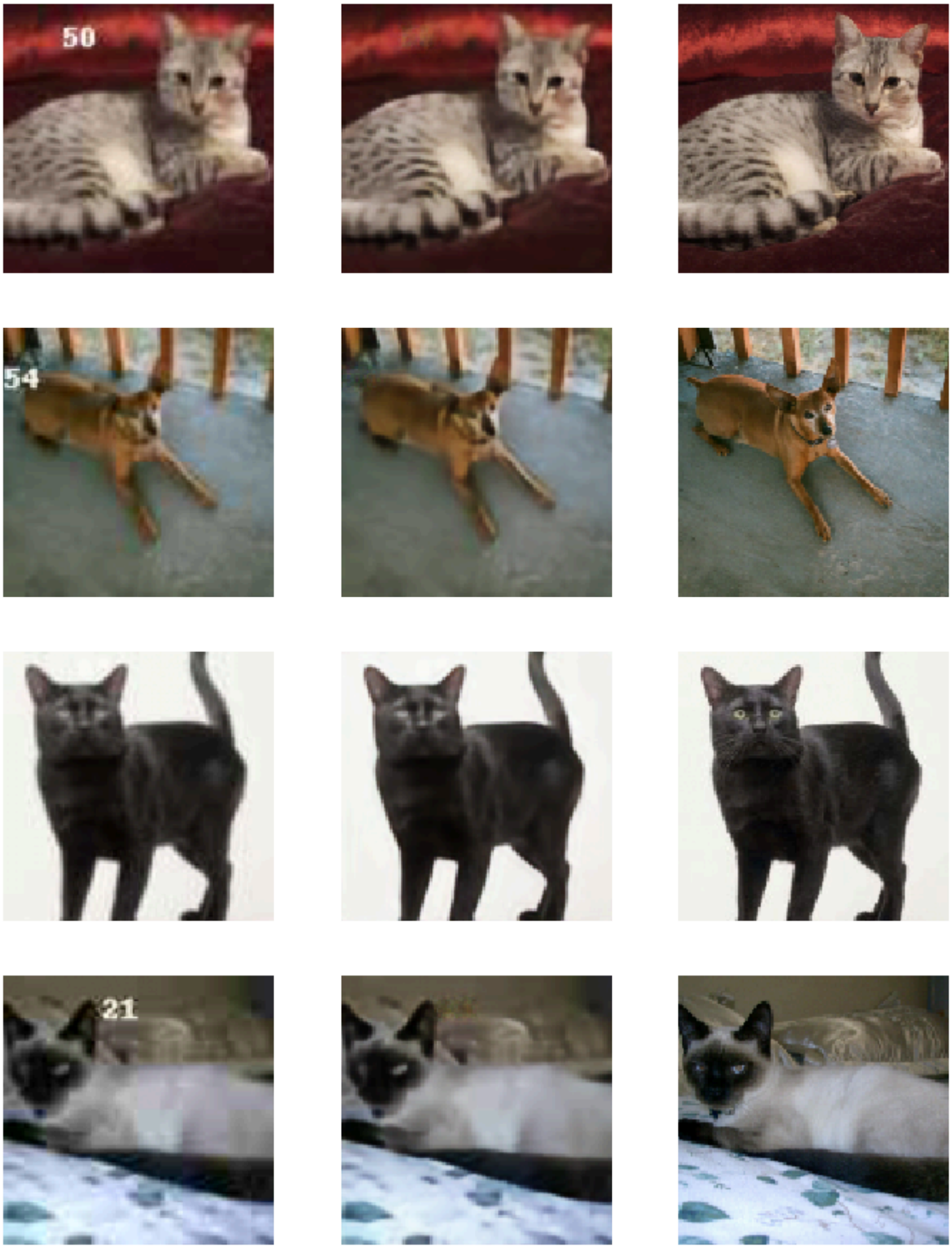
```
learn_gen.fit_one_cycle(3, slice(1e-6,1e-3))
```

Total time: 02:24

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

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

Input / Prediction / Target

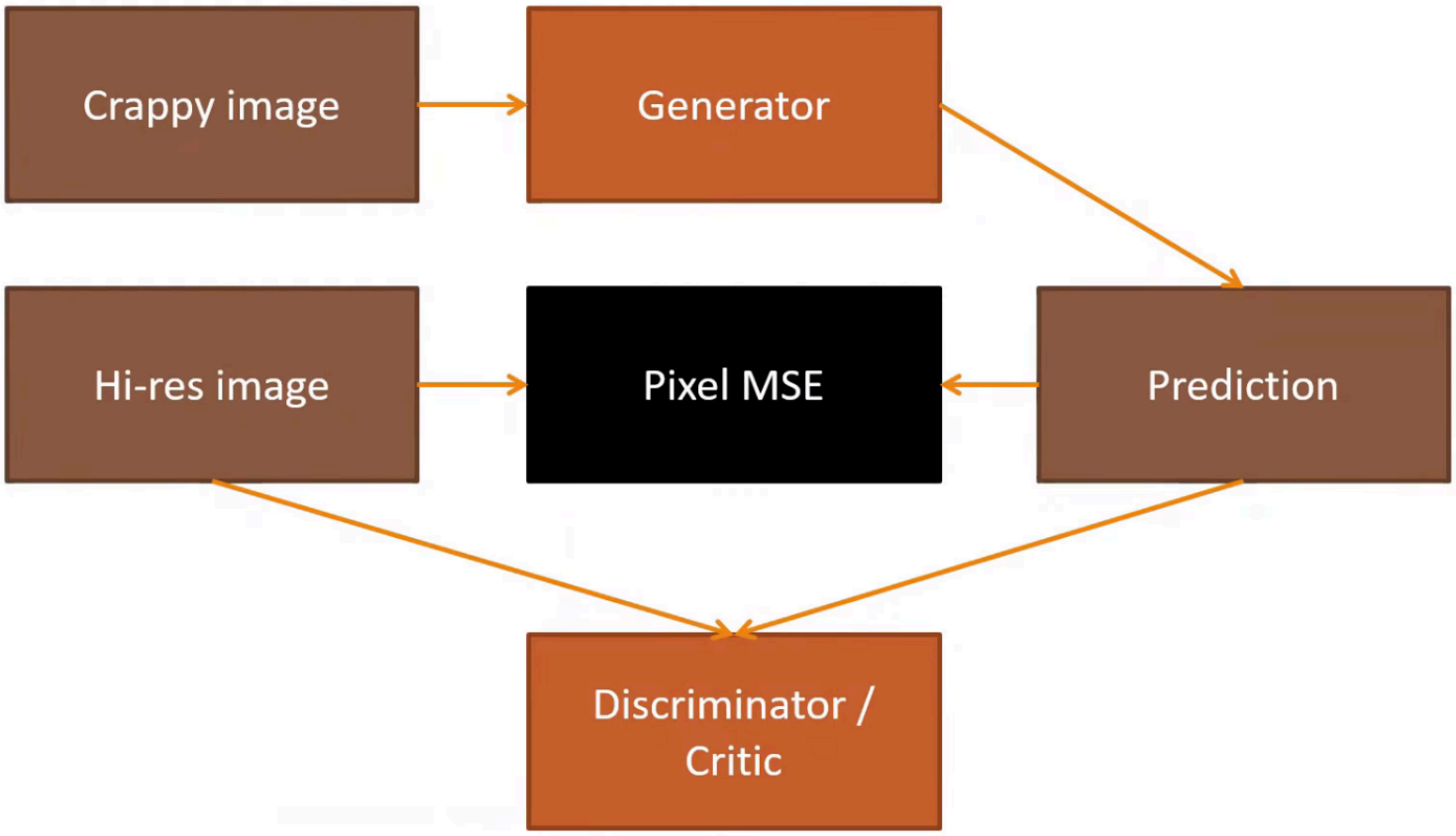


Let's unfreeze that and train a little more and look at that! With four minutes of training, we've got something which is basically doing a perfect job of removing numbers. It's certainly not doing a good job of upsampling, but it's definitely doing a nice job. Sometimes when it removes a number, it maybe leaves a little bit of JPEG artifact. But it's certainly doing something pretty useful. So if all we wanted to do was kind of watermark removal, we would be finished.

We're not finished because we actually want this thing (middle) to look more like this thing (right). So how are we going to do that? The reason that we're not making as much progress with that as we'd like is that our loss function doesn't really describe what we want. Because actually, the mean squared error between the pixels of this (middle) and this (right) is actually very small. If you actually think about it, most of the pixels are very nearly the right color. But we're missing the texture of the pillow, and we're missing the eyeballs entirely pretty much. We're missing the texture of the fur. So we want some lost function that does a better job than pixel mean squared error loss of saying like is this a good quality picture of this thing.

Generative Adversarial Network [59:23]

There's a fairly general way of answering that question, and it's something called a Generative adversarial network or GAN. A GAN tries to solve this problem by using a loss function which actually calls another model. Let me describe it to you.



We've got our crappy image, and we've already created a generator. It's not a great one, but it's not terrible and that's creating predictions (like the middle picture). We have a high-res image (like the right picture) and we can compare the high-res image to the prediction with pixel MSE.

We could also train another model which we would variously call either the discriminator or the critic - they both mean the same thing. I'll call it a critic. We could try and build a binary classification model that takes all the pairs of the generated image and the real high-res image, and learn to classify which is which. So look at some picture

and say “hey, what do you think? Is that a high-res cat or is that a generated cat? How about this one? Is that a high-res cat or a generated cat?” So just a regular standard binary cross-entropy classifier. We know how to do that already. If we had one of those, we could fine tune the generator and rather than using pixel MSE is the loss, the loss could be how good are we at fooling the critic? Can we create generated images that the critic thinks are real?

That would be a very good plan, because if it can do that, if the loss function is “am I fooling the critic?” then it’s going to learn to create images which the critic can’t tell whether they’re real or fake. So we could do that for a while, train a few batches. But the critic isn’t that great. The reason the critic isn’t that great is because it wasn’t that hard. These images are really crappy, so it’s really easy to tell the difference. So after we train the generator a little bit more using that critic as the loss function, the generators going to get really good at falling the critic. So now we’re going to stop training the generator, and we’ll train the critic some more on these newly generated images. Now that the generator is better, it’s now a tougher task for the critic to decide which is real and which is fake. So we’ll train that a little bit more. Then once we’ve done that and the critic is now pretty good at recognizing the difference between the better generated images and the originals, we’ll go back and we’ll fine tune the generator some more using the better discriminator (i.e. the better critic) as the loss function.

So we’ll just go ping pong ping pong, backwards and forwards. That’s a GAN. That’s our version of GAN. I don’t know if anybody’s written this before, we’ve created a new version of GAN which is kind of a lot like the original GANs but we have this neat trick where we pre-train the generator and we pre-train the critic. GANs have been kind of in the news a lot. They’re pretty fashionable tool, and if you’ve seen them, you may have heard that they’re a real pain to train. But it turns out we realized that really most of the pain of training them was at the start. If you don’t have a pre-trained generator and you don’t have a pre-trained critic, then it’s basically the blind leading the blind. The generator is trying to generate something which fools a critic, but the critic doesn’t know anything at all, so it’s basically got nothing to do. Then the critic is trying to decide whether the generated images are real or not, and that’s really obvious so that just does it. So they don’t go anywhere for ages. Then once they finally start picking up steam, they go along pretty quickly,

If you can find a way to generate things without using a GAN like mean squared pixel loss, and discriminate things without using a GAN like predict on that first generator, you can make a lot of progress.

Creating a critic/discriminator [1:04:04]

Let’s create the critic. To create just a totally standard fast.ai binary classification model, we need two folders; one folder containing high-res images, one folder containing generated images. We already have the folder with high-res images, so we just have to save our generated images.

```
name_gen = 'image_gen'
path_gen = path/name_gen

# shutil.rmtree(path_gen)

path_gen.mkdir(exist_ok=True)

def save_preds(dl):
    i=0
    names = dl.dataset.items
    for b in dl:
        preds = learn_gen.pred_batch(batch=b, reconstruct=True)
        for o in preds:
            o.save(path_gen/names[i].name)
            i += 1
```

```
save_preds(data_gen.fix_dl)
```

Here's a teeny tiny bit of code that does that. We're going to create a directory called `image_gen`, pop it into a variable called `path_gen`. We got a little function called `save_preds` which takes a data loader. We're going to grab all of the file names. Because remember that in an item list, `.items` contains the file names if it's an image item list. So here's the file names in that data loader's dataset. Now let's go through each batch of the data loader, and let's grab a batch of predictions for that batch, and then `reconstruct=True` means it's actually going to create fast.ai image objects for each thing in the batch. Then we'll go through each of those predictions and save them. The name we'll save it with is the name of the original file, but we're going to pop it into our new directory.

That's it. That's how you save predictions. So you can see, I'm increasingly not just using stuff that's already in the fast.ai library, but try to show you how to write stuff yourself. And generally it doesn't require heaps of code to do that. So if you come back for part 2, lots of part 2 are like here's how you use things inside the library, and of course, here's how we wrote the library. So increasingly, writing our own code.

Okay, so save those predictions and let's just do a `PIL.Image.open` on the first one, and yep there it is. So there's an example of the generated image.

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

Now I can train a critic in the usual way. It's really annoying to have to restart Jupyter notebook to reclaim GPU memory. One easy way to handle this is if you just set something that you knew was using a lot of GPU to `None` like this learner, and then just go `gc.collect`, that tells Python to do memory garbage collection, and after that you'll generally be fine. You'll be able to use all of your GPU memory again.

```
learn_gen=None
gc.collect()
```

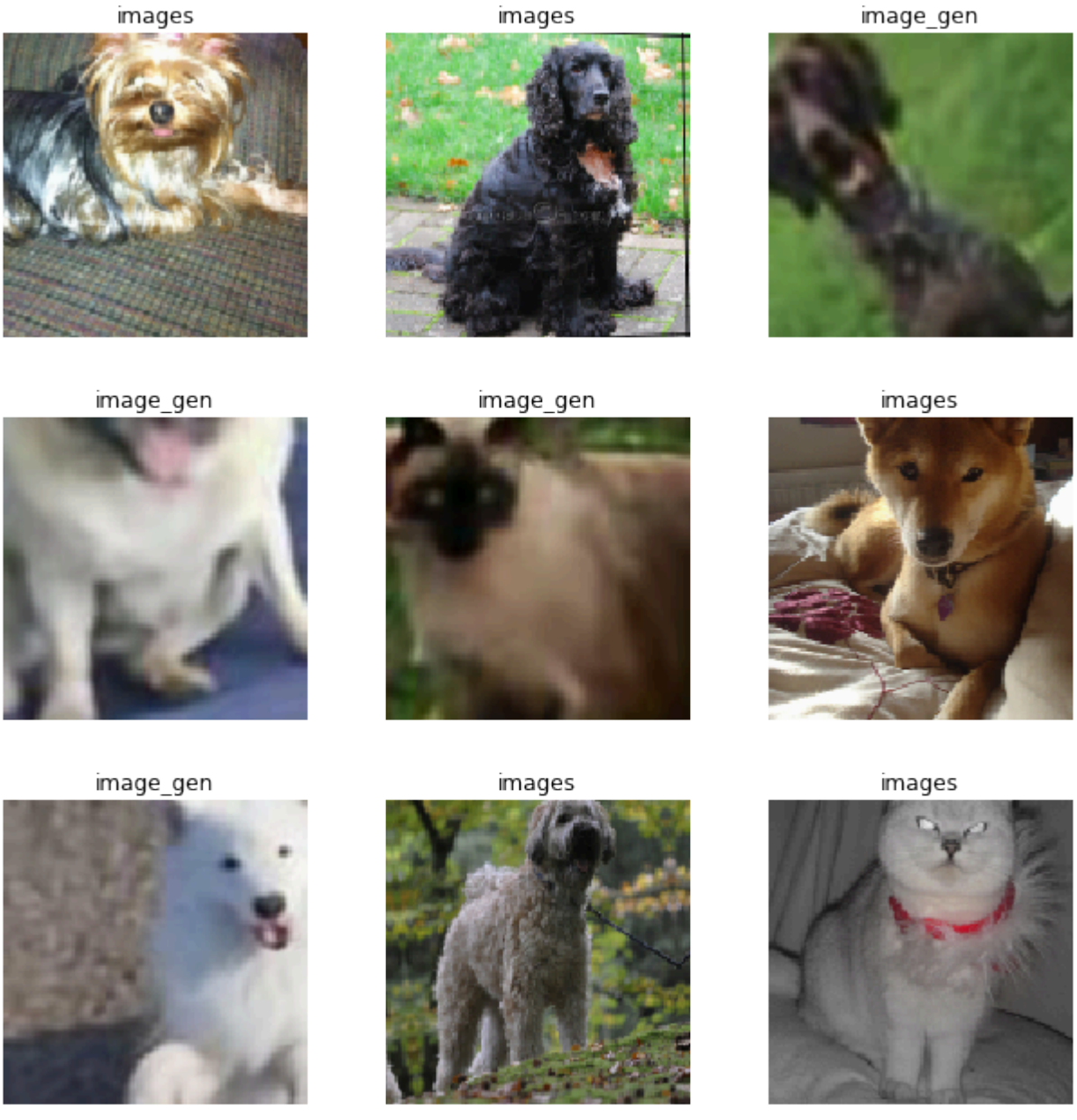
If you're using `nvidia-smi` to actually look at your GPU memory, you won't see it clear because PyTorch still has a kind of allocated cache, but it makes it available. So you should find this is how you can avoid restarting your notebook.

```
def get_crit_data(classes, bs, size):
    src = ImageItemList.from_folder(path, include=classes).random_split_by_pct(0.
    ll = src.label_from_folder(classes=classes)
    data = (ll.transform(get_transforms(max_zoom=2.), size=size)
           .databunch(bs=bs).normalize(imagenet_stats))
    data.c = 3
    return data
```

We're going to create our critic. It's just an image item list from folder in the totally usual way, and the classes will be the `image_gen` and `images`. We will do a random split because we want to know how well we're doing with

the critic to have a validation set. We just label it from folder in the usual way, add some transforms, data bunch, normalized. So we've got a totally standard classifier. Here's what some of it looks like:

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

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



Here's one from the real images, real images, generated images, generated images, … So it's got to try and figure out which class is which.

```
loss_critic = AdaptiveLoss(nn.BCEWithLogitsLoss())
```

We're going to use binary cross entropy as usual. However, we're not going to use a ResNet here. The reason, we'll get into in more detail in part 2, but basically when you're doing a GAN, you need to be particularly careful that the generator and the critic can't both push in the same direction and increase the weights out of control. So we have to use something called spectral normalization to make GANs work nowadays. We'll learned about that in

part 2.

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

learn_critic = create_critic_learner(data_crit, accuracy_thresh_expand)
```

Anyway, if you say `gan_critic`, fast.ai will give you a binary classifier suitable for GANs. I strongly suspect we probably can use a ResNet here. We just have to create a pre trained ResNet with spectral norm. Hope to do that pretty soon. We'll see how we go, but as of now, this is kind of the best approach is this thing called `gan_critic`. A GAN critic uses a slightly different way of averaging the different parts of the image when it does the loss, so anytime you're doing a GAN at the moment, you have to wrap your loss function with `AdaptiveLoss`. Again, we'll look at the details in part 2. For now, just know this is what you have to do and it'll work.

Other than that slightly odd loss function and that slightly odd architecture, everything else is the same. We can call that (`create_critic_learner` function) to create our critic. Because we have this slightly different architecture and slightly different loss function, we did a slightly different metric. This is the equivalent GAN version of accuracy for critics. Then we can train it and you can see it's 98% accurate at recognizing that kind of crappy thing from that kind of nice thing. But of course we don't see the numbers here anymore. Because these are the generated images, the generator already knows how to get rid of those numbers that are written on top.

```
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

Finishing up GAN [1:09:52]

```
learn_crit=None
learn_gen=None
gc.collect()

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

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

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

Let's finish up this GAN. Now that we have pre-trained the generator and pre-trained the critic, we now need to get it to kind of ping pong between training a little bit of each. The amount of time you spend on each of those things and the learning rates you use is still a little bit on the fuzzy side, so we've created a `GANLearner` for you which you just pass in your generator and your critic (which we've just simply loaded here from the ones we just trained)

and it will go ahead and when you go `learn.fit`, it will do that for you - it'll figure out how much time to train generator and then when to switch to training the discriminator/critic and it'll go backward and forward.

```
switcher = partial(AdaptiveGANSwitcher, critic_thresh=0.65)
learn = GANLearner.from_learners(learn_gen, learn_crit, weights_gen=(1.,50.), sho
                                 opt_func=partial(optim.Adam, betas=(0.,0.99)), w
learn.callback_fns.append(partial(GANDiscriminativeLR, mult_lr=5.))
```

[1:10:43]

These weights here (`weights_gen=(1.,50.)`) is that, what we actually do is we don't only use the critic as the loss function. If we only use the critic as the loss function, the GAN could get very good at creating pictures that look like real pictures, but they actually have nothing to do with the original photo at all. So we actually add together the pixel loss and the critic loss. Those two losses on different scales, so we multiplied the pixel loss by something between about 50 and about 200 - something in that range generally works pretty well.

Something else with GANs. **GANs hate momentum** when you're training them. It kind of doesn't make sense to train them with momentum because you keep switching between generator and critic, so it's kind of tough. Maybe there are ways to use momentum, but I'm not sure anybody's figured it out. So this number here (`betas=(0.,...)`) when you create an Adam optimizer is where the momentum goes, so you should set that to zero.

Anyways, if you're doing GANs, use these hyper parameters:

```
GANLearner.from_learners(learn_gen, learn_crit, weights_gen=(1.,50.), show_img=Fa
                         switcher=switcher, opt_func=partial(optim.Adam, betas=(0
                         wd=wd)
```

It should work. That's what GAN learner does. Then you can go fit, and it trains for a while.

```
lr = 1e-4

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

epoch	train_loss	gen_loss	disc_loss
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
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

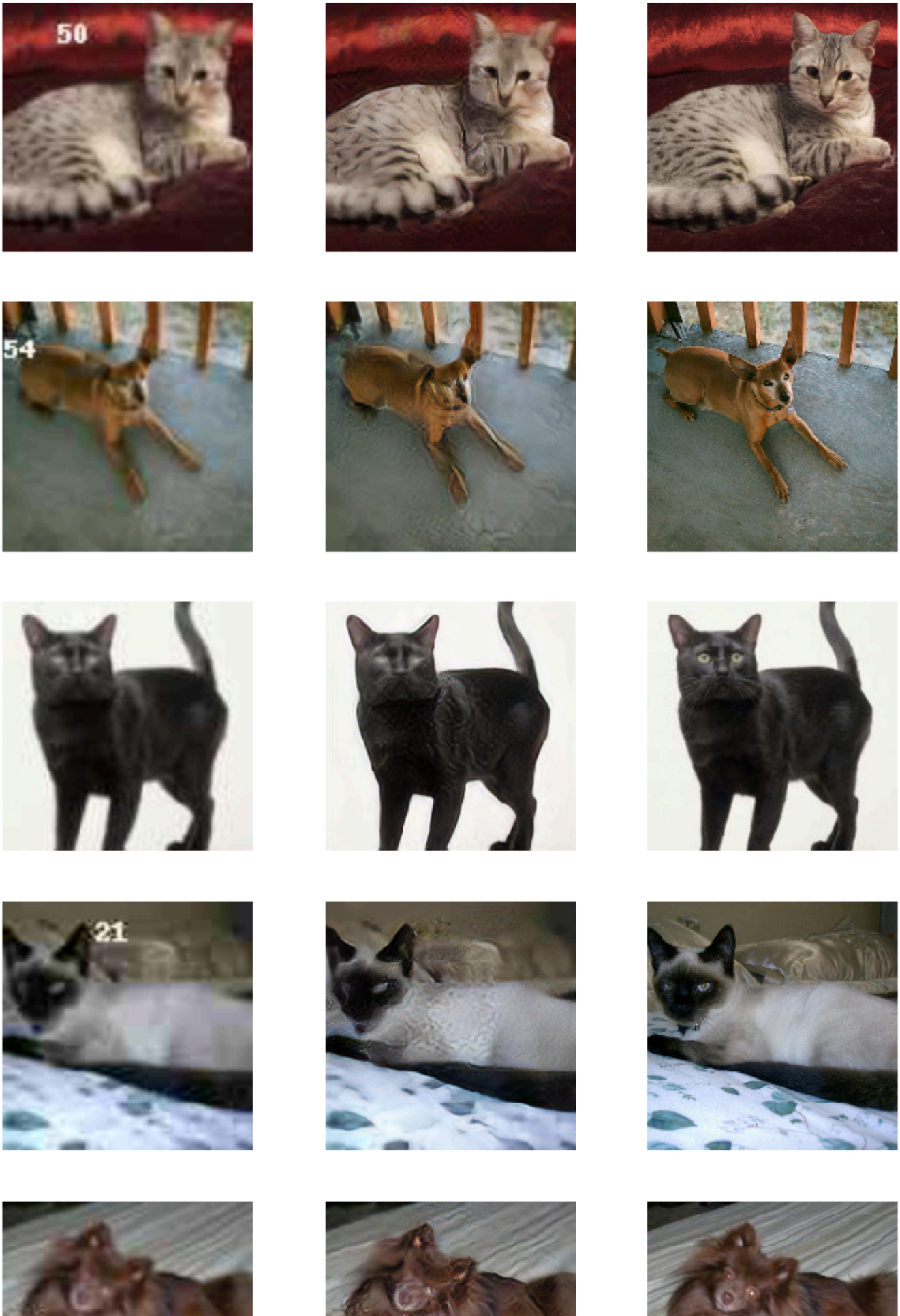
One of the tough things about GANs is that these loss numbers, they're meaningless. You can't expect them to go down because as the generator gets better, it gets harder for the discriminator (i.e. the critic) and then as the critic gets better, it's harder for the generator. So the numbers should stay about the same.So that's one of the tough things about training GANs is it's hard to know how are they doing. The only way to know how are they doing is to actually take a look at the results from time to time. If you put `show_img=True` here:

```
GANLearner.from_learners(learn_gen, learn_crit, weights_gen=(1.,50.), show_img=Fa
                         switcher=switcher, opt_func=partial(optim.Adam, betas=(0
                         wd=wd)
```

It'll actually print out a sample after every epoch. I haven't put that in the notebook because it makes it too big for the repo, but you can try that. So I've just put the results at the bottom, and here it is.

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

Input / Prediction / Target



Pretty beautiful, I would say. We already knew how to get rid of the numbers, but we now don't really have that kind of artifact of where it used to be, and it's definitely sharpening up this little kitty cat quite nicely. It's not great always. There's some weird kind of noise going on here. Certainly a lot better than the horrible original. This is a tough job to turn that into that. But there are some really obvious problems. Like here (the third row), these things ought to be eyeballs and they're not. So why aren't they? Well, our critic doesn't know anything about eyeballs. Even if it did, it wouldn't know that eyeballs are particularly important. We care about eyes. When we see a cat without eyes, it's a lot less cute. I mean I'm more of a dog person, but you know. It just doesn't know that this is a feature that matters. Particularly because the critic, remember, is not a pre-trained network. So I kind of suspect that if we replace the critic with a pre-trained network that's been pre-trained on ImageNet but is also compatible with GANs, it might do a better job here. But it's definitely a shortcoming of this approach. After the break I will show you how to find the cat's eye balls.

Question: For what kind of problems, do you not want to use U-Net? [1:14:48]

U-Nets are for when the the size of your output is similar to the size of your input and kind of aligned with it. There's no point having cross connections if that level of spatial resolution in the output isn't necessary or useful. So yeah, any kind of generative modeling and segmentation is kind of generative modeling. It's generating a picture which is a mask of the original objects. So probably anything where you want that kind of resolution of the output to be at the same kind of fidelity as a resolution of the input. Obviously something like a classifier makes no sense. In a classifier, you just want the downsampling path because at the end you just want a single number which is like is it a dog, or a cat, or what kind of pet is it or whatever.

Wasserstein GAN [1:15:59]

Just before we leave GANs, I just mention there's another notebook you might be interested in looking at which is lesson7-wgan.ipynb. When GANs started a few years ago, people generally use them to create images out of thin air which I personally don't think is a particularly useful or interesting thing to do. But it's a good research exercise, I guess. So we implemented this WGAN paper which was kind of really the first one to do a somewhat adequate job somewhat easily. You can see how to do that with the fast.ai library.

It's kind of interesting because the the dataset that we use is this LSUN bedrooms dataset which we've provided in our URLs which just has bedrooms, lots and lots and lots of bedrooms. The approach we use in this case is to just say "can we create a bedroom?" So what we actually do is that the input to the generator isn't an image that we clean up. We actually feed to the generator random noise. Then the generator's task is "can you turn random noise into something which the critic can't tell the difference between that output and a real bedroom?" We're not doing any pre-training here or any of the stuff that makes this fast and easy, so this is a very traditional approach. But you can see, you still just go `GANLearner` and there's actually a `wgan` version which is this older style approach. But you just pass in the data, the generator, and the critic in the usual way and you call `fit`.

You'll see (in this case, we have a `show_image` on) after a epoch one, it's not creating great bedrooms or two or three. And you can really see that in the early days of these kinds of GANs, it doesn't do a great job of anything. But eventually, after a couple of hours of training, it's producing somewhat like bedroom-ish things. Anyway, it's a notebook you can have a play with, and it's a bit of fun.

Feature Loss [1:18:37]

I was very excited when we got fast.ai to the point in the last week or so that we had GAN's working in a way where API wise, they're far more concise and more flexible than any other library that exists. But also kind of disappointed with them. They take a long time to train and the outputs are still like so-so, and so the next step was "can we get rid of GANs entirely?" Obviously, the thing we really want to do is come up with a better loss

function. We want a loss function that does a good job of saying this is a high-quality image without having to go all the GAN trouble, and preferably it also doesn't just say it's a high-quality image but it's an image which actually looks like the thing is meant to. So the real trick here comes back to this paper from a couple of years ago, Perceptual Losses for Real-Time Style Transfer and Super-Resolution - Justin Johnson et al. created this thing they call perceptual losses. It's a nice paper, but I hate this term because they're nothing particularly perceptual about them. I would call them "feature losses", so in the fast.ai library, you'll see this referred to as feature losses.

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

Justin Johnson, Alexandre Alahi, Li Fei-Fei

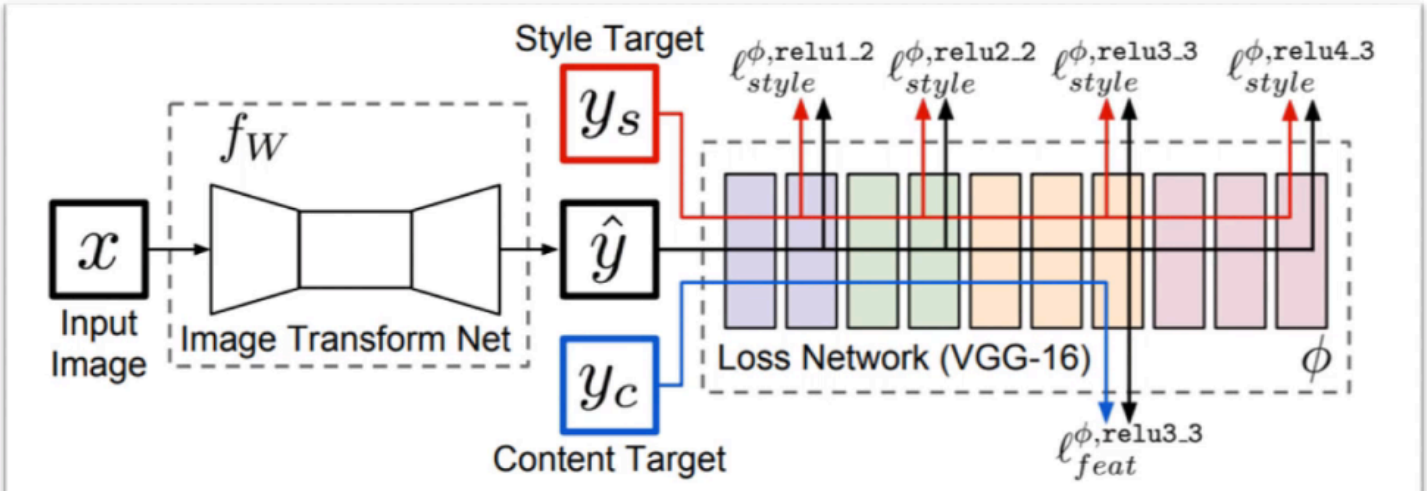


Fig. 2. System overview. We train an *image transformation network* to transform input images into output images. We use a *loss network* pretrained for image classification to define *perceptual loss functions* that measure perceptual differences in content and style between images. The loss network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

It shares something with GANs which is that after we go through our generator which they call the "image transform net" and you can see it's got this kind of U-Net shaped thing. They didn't actually use U-Nets because at the time this came out, nobody in the machine learning world knew about U-Nets. Nowadays, of course, we use U-Nets. But anyway, something U-Net-ish.

In these kind of architectures where you have a downsampling path followed by an upsampling path, the downsampling path is very often called the **encoder** as you saw in our code. And the upsampling path is very often called the **decoder**. In generative models, generally including generative text models, neural translation, stuff like that, they tend to be called the encoder and the decoder - two pieces.

Anyway, so we have this generator and we want a loss function that says "is the thing that it's created like the thing that we want?" and so the way they do that is they take the prediction, remember, $\hat{y}$ is what we normally use for a prediction from a model. We take the prediction and we put it through a pre-trained ImageNet network. At the time that this came out, the pre-trained ImageNet network they were using was VGG. It's kind of old now, but people still tend to use it because it works fine for this process. So they take the prediction, and they put it through VGG - the pre-trained ImageNet network. It doesn't matter too much which one it is.

So normally, the output of that would tell you “hey, is this generated thing a dog, a cat, an airplane, or a fire engine or whatever?” But in the process of getting to that final classification, it goes through lots of different layers. In this case, they’ve color-coded all the layers with the same grid size and the feature map with the same color. So every time we switch colors, we’re switching grid size. So there’s a stride 2 conv or in VGG’s case they still used to use some maxpooling layers which is a similar idea.

What we could do is say let’s not take the final output of the VGG model on this generated image, but let’s take something in the middle. Let’s take the activations of some layer in the middle. Those activations, it might be a feature map of like 256 channels by 28 by 28. So those kind of 28 by 28 grid cells will kind of roughly semantically say things like “in this part of that 28 by 28 grid, is there something that looks kind of furry? Or is there something that looks kind of shiny? Or is there something that was kind of circular? Is there something that kind of looks like an eyeball?”

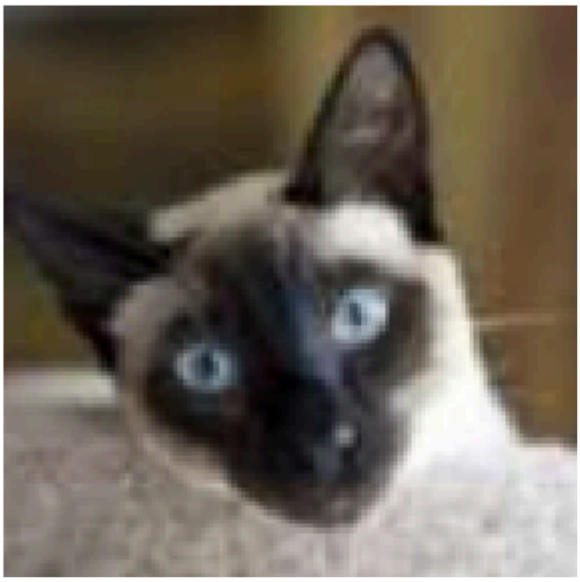
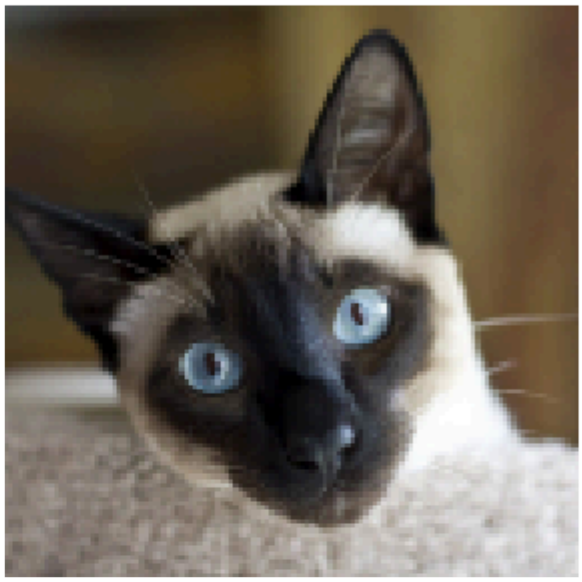
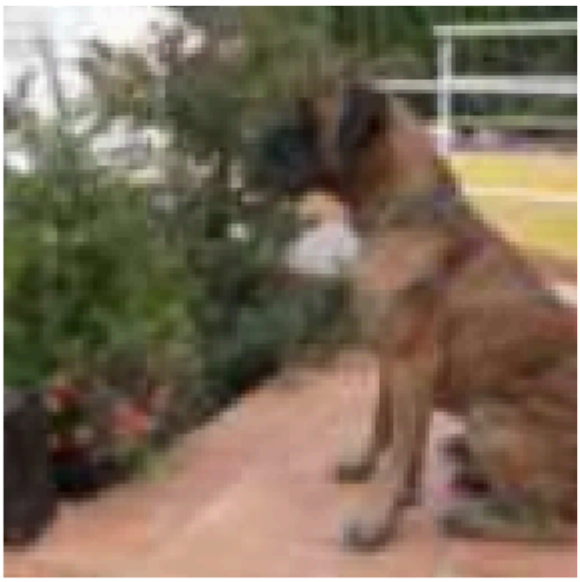
So what we do is that we then take the target (i.e. the actual y value) and we put it through the same pre-trained VGG network, and we pull out the activations of the same layer. Then we do a mean square error comparison. So it’ll say “in the real image, grid cell (1, 1) of that 28 by 28 feature map is furry and blue and round shaped. And in the generated image, it’s furry and blue and not round shape.” So it’s an okay match.

That ought to go a long way towards fixing our eyeball problem, because in this case, the feature map is going to say “there’s eyeballs here (in the target), but there isn’t here (in the generated version), so do a better job of that please. Make better eyeballs.” So that’s the idea. That’s what we call feature losses or Johnson et al. called perceptual losses.

To do that, we’re going to use the lesson7-superres.ipynb, and this time the task we’re going to do is kind of the same as the previous task, but I wrote this notebook a little bit before the GAN notebook - before I came up with the idea of like putting text on it and having a random JPEG quality, so the JPEG quality is always 60, there’s no text written on top, and it’s 96 by 96. And before I realized what a great word “crappify” is, so it’s called `resize_one`.

```
import fastai
from fastai.vision import *
from fastai.callbacks import *

from torchvision.models import vgg16_bn

path = untar_data(URLs.PETS)
path_hr = path/'images'
path_lr = path/'small-96'
path_mr = path/'small-256'

il = ImageItemList.from_folder(path_hr)

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

# to create smaller images, uncomment the next line when you run this the first t
```

```
# parallel(resize_one, il.items)

bs,size=32,128
arch = models.resnet34

src = ImageImageList.from_folder(path_lr).random_split_by_pct(0.1, seed=42)

def get_data(bs,size):
    data = (src.label_from_func(lambda x: path_hr/x.name)
           .transform(get_transforms(max_zoom=2.), size=size, tfm_y=True)
           .databunch(bs=bs).normalize(imagenet_stats, do_y=True))

    data.c = 3
    return data

data = get_data(bs,size)

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


Here's our crappy images and our original images - kind of a similar task to what we had before. I'm going to try and create a loss function which does this (perceptual loss). The first thing I do is I define a base loss function which is basically like "how am I going to compare the pixels and the features?" And the choice is mainly like MSE or L1. It doesn't matter too much which you choose. I tend to like L1 better than MSE actually, so I picked L1.

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

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

gram_matrix(t)
```

```
tensor([[[0.0759, 0.0711, 0.0643],
         [0.0711, 0.0672, 0.0614],
         [0.0643, 0.0614, 0.0573]],

        [[0.0759, 0.0711, 0.0643],
         [0.0711, 0.0672, 0.0614],
         [0.0643, 0.0614, 0.0573]]])

base_loss = F.l1_loss
```

So anytime you see `base_loss`, we mean L1 loss. You could use MSE loss as well.

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

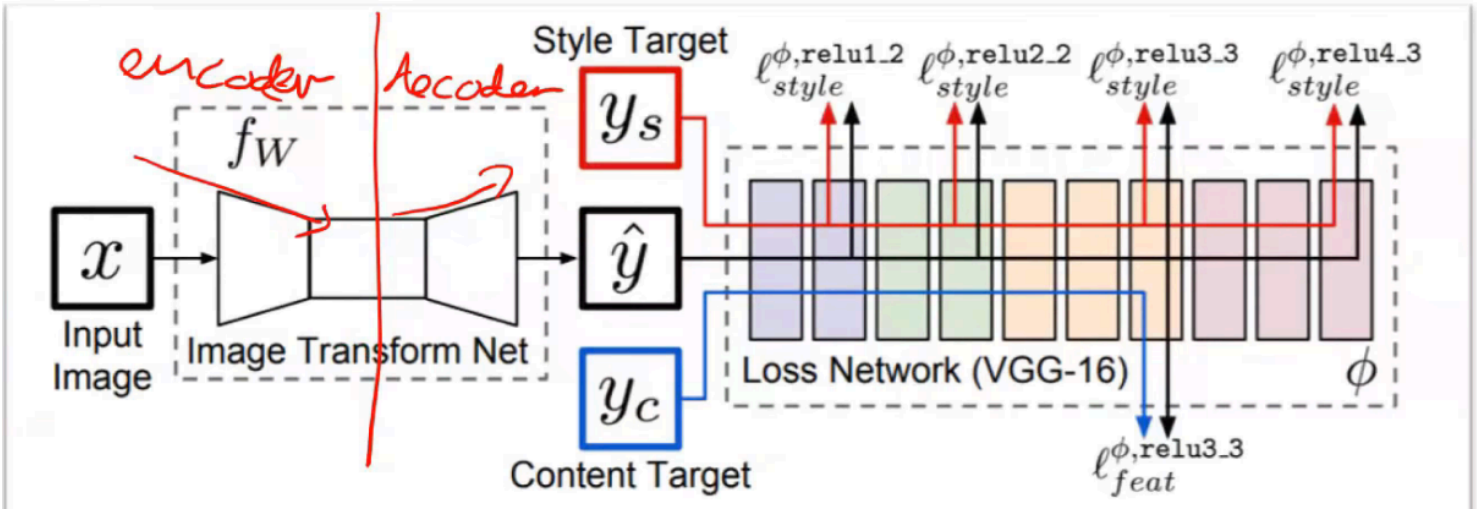
Let's create a VGG model - just using the pre-trained model. In VGG, there's a attribute called `.features` which contains the convolutional part of the model. So `vgg16_bn(True).features` is the convolutional part of the VGG model. Because we don't need the head. We only want the intermediate activations.

Then we'll check that on the GPU, we'll put it into `eval` mode because we're not training it. And we'll turn off `requires_grad` because we don't want to update the weights of this model. We're just using it for inference (i.e. for the loss).

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

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

Then let's enumerate through all the children of that model and find all of the max pooling layers, because in the VGG model that's where the grid size changes. And as you could see from this picture, we want to grab features from every time just before the grid size changes:



So we grab layer `i-1`. That's the layer before it changes. So there's our list of layer numbers just before the max pooling layers (`[5, 12, 22, 32, 42]`). All of those are ReLU's, not surprisingly. Those are where we want to grab some features from, and so we put that in `blocks` - it's just a list of ID's.

```
class FeatureLoss(nn.Module):
    def __init__(self, m_feat, layer_ids, layer_wgts):
        super().__init__()
        self.m_feat = m_feat
        self.loss_features = [self.m_feat[i] for i in layer_ids]
        self.hooks = hook_outputs(self.loss_features, detach=False)
        self.wgts = layer_wgts
        self.metric_names = ['pixel',] + [f'feat_{i}' for i in range(len(layer_id
              ] + [f'gram_{i}' for i in range(len(layer_ids))]

    def make_features(self, x, clone=False):
        self.m_feat(x)
        return [(o.clone() if clone else o) for o in self.hooks.stored]

    def forward(self, input, target):
        out_feat = self.make_features(target, clone=True)
        in_feat = self.make_features(input)
        self.feat_losses = [base_loss(input,target)]
        self.feat_losses += [base_loss(f_in, f_out)*w
                             for f_in, f_out, w in zip(in_feat, out_feat, self.wg
        self.feat_losses += [base_loss(gram_matrix(f_in), gram_matrix(f_out))*w**
                             for f_in, f_out, w in zip(in_feat, out_feat, self.wg
        self.metrics = dict(zip(self.metric_names, self.feat_losses))
        return sum(self.feat_losses)

    def __del__(self): self.hooks.remove()
```

Here's our feature loss class which is going to implement this idea (perceptual loss).

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

Basically, when we call the feature loss class, we're going to pass it some pre-trained model which is going to be called `m_feat`. That's the model which contains the features which we want our feature loss on. So we can go ahead and grab all of the layers from that network that we want the features for to create the losses.

We're going to need to hook all of those outputs because that's how we grab intermediate layers in PyTorch is by hooking them. So `self.hook` is going to contain our our hooked outputs.

Now in the `forward` of feature loss, we're going to call `make_features` passing in the `target` (i.e. our actual y) which is just going to call that VGG model and go through all of the stored activations and just grab a copy of them. We're going to do that both for the target (`out_feat`) and for the input - so that's the output of the generator (`in_feat`). Now let's calculate the L1 loss between the pixels, because we still want the pixel loss a little bit. Then let's also go through all of those layers' features and get the L1 loss on them. So we're basically going through every one of these end of each block and grabbing the activations and getting the L1 on each one.

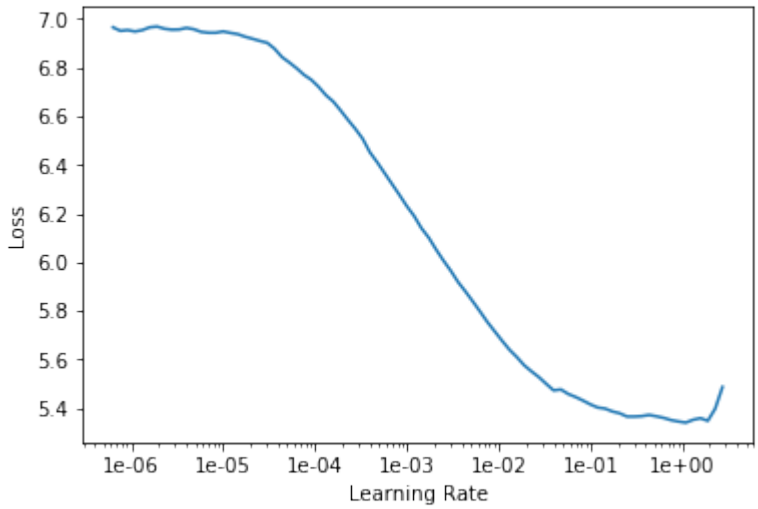
That's going to end up in this list called `feat_losses`, then sum them all up. By the way, the reason I do it as a list is because we've got this nice little callback that if you put them into thing called `.metrics` in your loss function, it'll print out all of the separate layer loss amounts for you which is super handy.

So that's it. That's our perceptual loss or feature loss class.

```
wd = 1e-3
learn = unet_learner(data, arch, wd=wd, loss_func=feat_loss, callback_fns=LossMet
                     blur=True, norm_type=NormType.Weight)
gc.collect();
```

Now we can just go ahead and train a U-Net in the usual way with our data and pre-trained architecture which is a ResNet 34, passing in our loss function which is using our pre trained VGG model. This (`callback_fns`) is that callback I mentioned LossMetrics which is going to print out all the different layers losses for us. These are two things (`blur` and `norm_type`) that we'll learn about in part 2 of the course, but you should use them.

```
learn.lr_find()
learn.recorder.plot()
```




```
lr = 1e-3

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

I just created a little function called `do_fit` that does fit one cycle, and then saves the model, and then shows the results.

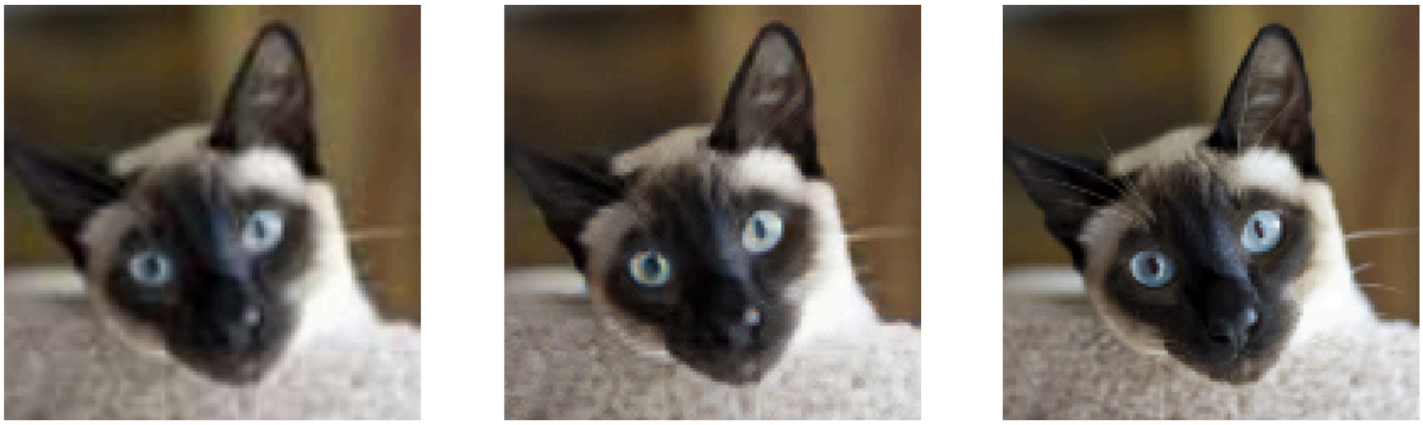
```
do_fit('1a', slice(lr*10))
```

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

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2	gram_0	gram_1	gram_2
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



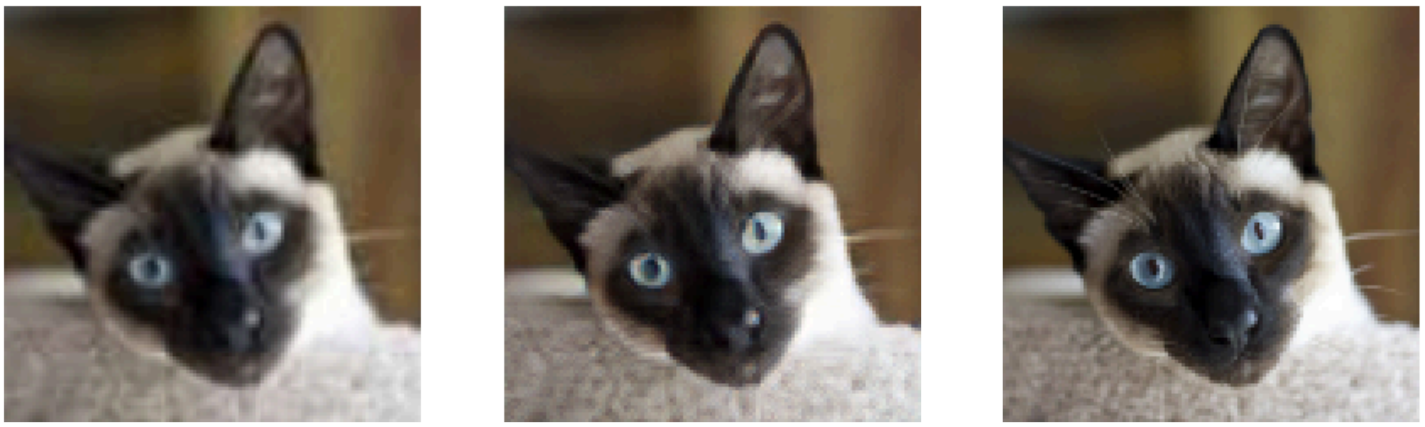
As per usual, because we're using a pre-trained network in our U-Net, we start with frozen layers for the downsampling path, train for a while. As you can see, we get not only the loss, but also the pixel loss and the loss at each of our feature layers, and then also something we'll learn about in part 2 called Gram loss which I don't think anybody's used for super resolution before as far as I know. But as you'll see, it turns out great.

That's eight minutes, so much faster than a GAN, and already, as you can see this is our model's output, pretty good. Then we unfreeze and train some more.

```
learn.unfreeze()

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

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



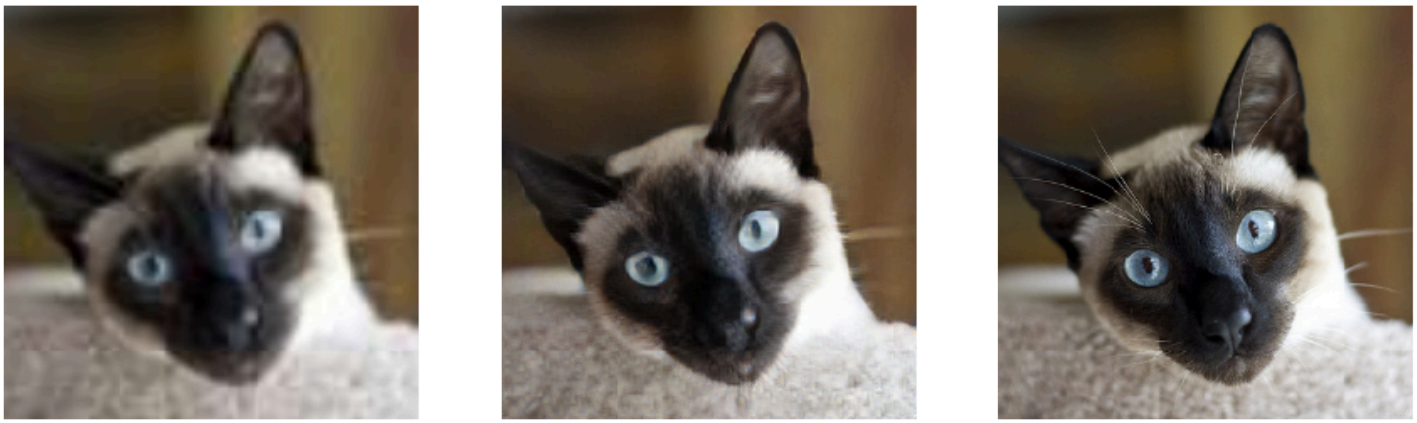
And it's a little bit better. Then let's switch up to double the size. So we need to also halve the batch size to avoid running out of GPU memory, and freeze again, and train some more.

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

learn.data = data
learn.freeze()
gc.collect()

learn.load('1b');

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



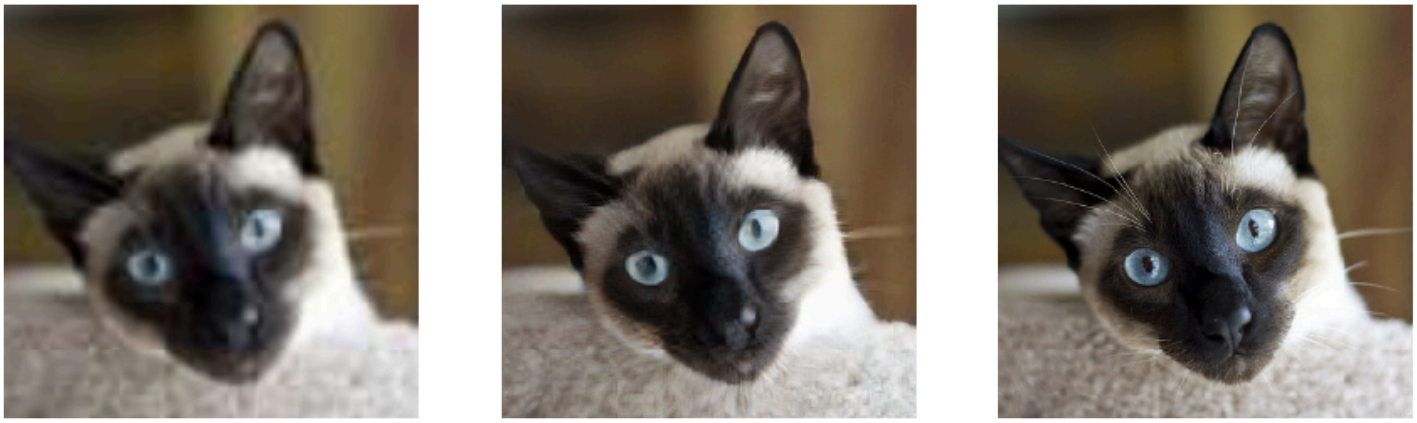
It's now taking half an hour, (the result is) even better. Then unfreeze and train some more.

```
learn.unfreeze()

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



All in all, we've done about an hour and twenty minutes of training and look at that! It's done it. It knows that eyes are important so it's really made an effort. It knows that fur is important so it's really made an effort. It started with something with JPEG artifacts around the ears and all this mess and eyes that are just kind of vague light blue things, and it really created a lot of texture. This cat is clearly looking over the top of one of those little clawing frames covered in fuzz, so it actually recognized that this thing is probably a carpeting materials. It's created a carpeting material for us. So I mean, that's just remarkable.

Talking for markable, I've mean I've never seen outputs like this before without a GAN so I was just so excited when we were able to generate this and so quickly - one GPU, an hour and a half. If you create your own crappification functions and train this model, you'll' build stuff that nobody's built before. Because nobody else that I know of is doing it this way. So there are huge opportunities, I think. So check this out.

Medium Resolution [1:31:45]

What we can now do is we can now, instead of starting with our low res, I actually stored another set at size 256 which are called medium res, so let's see what happens if we up size a medium res

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

learn.data = data_mr

fn = data_mr.valid_ds.x.items[0]; fn

PosixPath('/data1/jhoward/git/course-v3/nbs/dl1/data/oxford-iiit-pet/small-256/Si

img = open_image(fn); img.shape

torch.Size([3, 256, 320])

p,img_hr,b = learn.predict(img)

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

We’re going to grab our medium res data, and here is our medium res stored photo. Can we improve this? You can see, there’s still a lot of room for improvement. The lashes here are very pixelated. The place where there should be hair here is just kind of fuzzy. So watch this area as I hit down on my keyboard:

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

Look at that. It's done it. It's taken a medium res image and it's made a totally clear thing here. The furs reappeared. But look at the eyeball. Let's go back. The eyeball here (the before) is just kind of a general blue thing, here (the after) it's added all the right texture. So I just think this is super exciting. Here's a model I trained in an hour and a half using standard stuff that you've all learnt about a U-Net, a pre trained model, feature loss function and we've got something which can turn that medium res into that high res, or this absolute mess into this. It's really exciting to think what could you do with that.

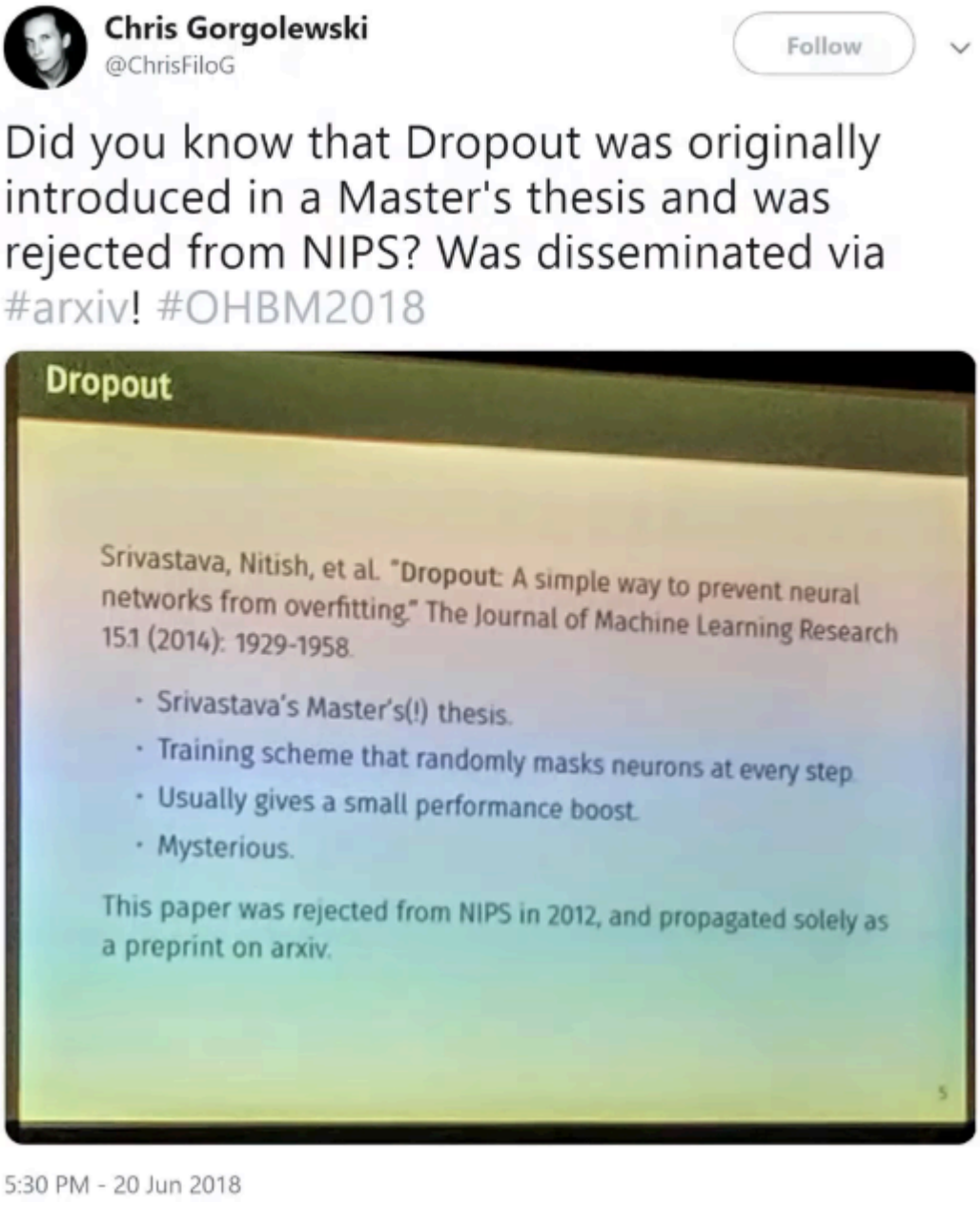
One of the inspirations here has been Jason Antic. Jason was a student in the course last year, and what he did very sensibly was decided to focus basically nearly quit his job and work four days a week or really six days a week on studying deep learning, and as you should do, he created a kind of capstone project. His project was to combine GANs and feature losses together. And his crappification approach was to take color pictures and make them black and white. So he took the whole of ImageNet, created a black and white ImageNet, and then trained a model to re-colorize it. He's put this up as DeOldify and now he's got these actual old photos from the 19th century that he's turning into color.

What this is doing is incredible. Look at this. The model thought “oh that’s probably some kind of copper kettle, so I’ll make it copper colored” and “oh these pictures are on the wall, they’re probably like different colors to the wall” and “maybe that looks a bit like a mirror, maybe it would be reflecting stuff outside.” “These things might be vegetables, vegetables are often red. Let’s make them red.” It’s extraordinary what it’s done. And you could totally do this too. You can take our feature loss and our GAN loss and combine them. So I’m very grateful to Jason because he’s helped us build this lesson, and it has been really nice because we’ve been able to help him too because he hadn’t realized that he can use all this pre-training and stuff. So hopefully you’ll see DeOldify in a couple of weeks be even better at the deoldification. But hopefully you all can now add other kinds of decrappification methods as well.

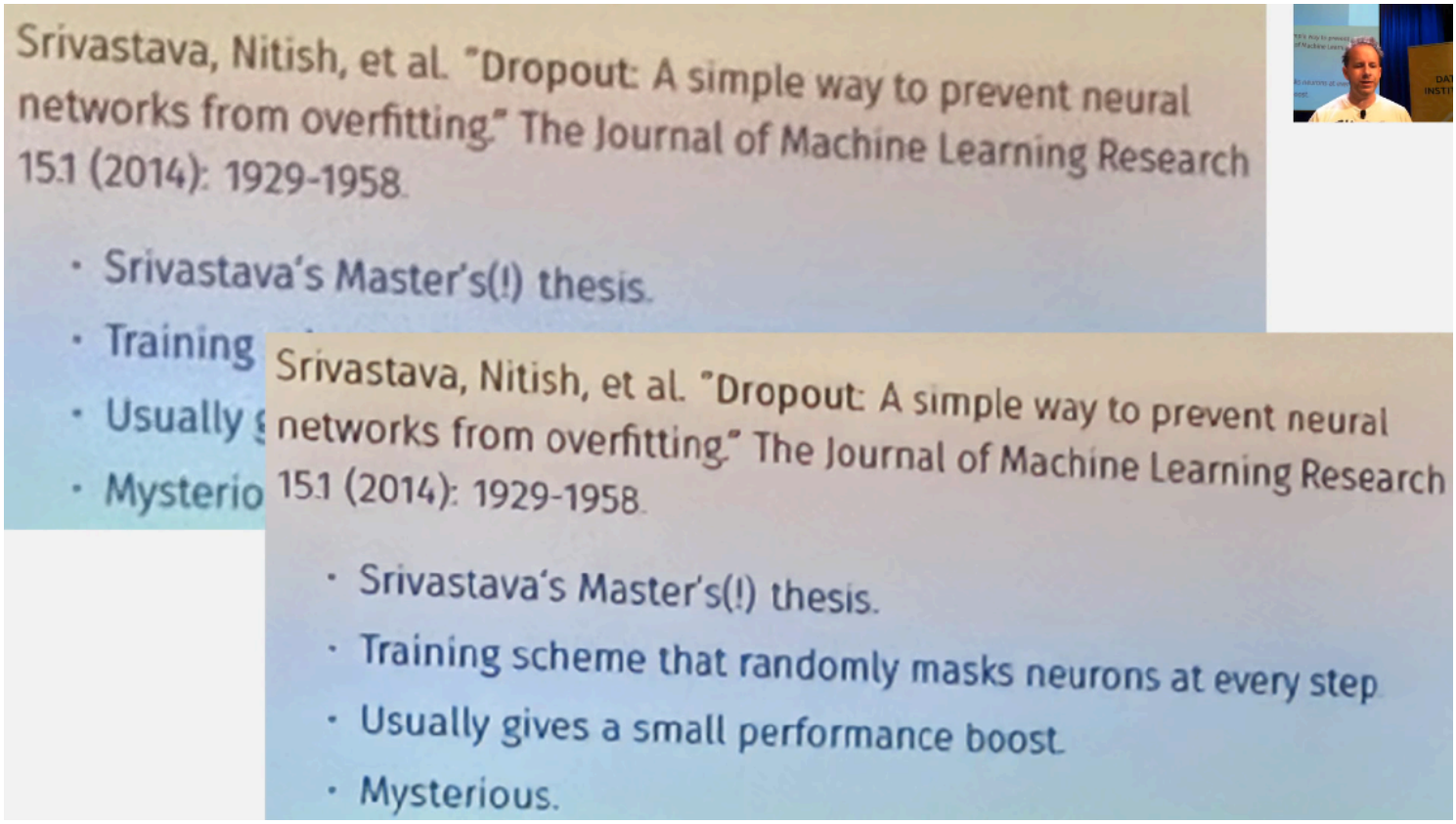
I like every course if possible to show something totally new, because then every student has the chance to basically build things that have never been built before. So this is kind of that thing. But between the much better segmentation results and these much simpler and faster decrappification results, i think you can build some really cool stuff.

Question: Is it possible to use similar ideas to U-Net and GANs for NLP? For example if I want to tag the verbs and nouns in a sentence or create a really good Shakespeare generator [1:35:54]

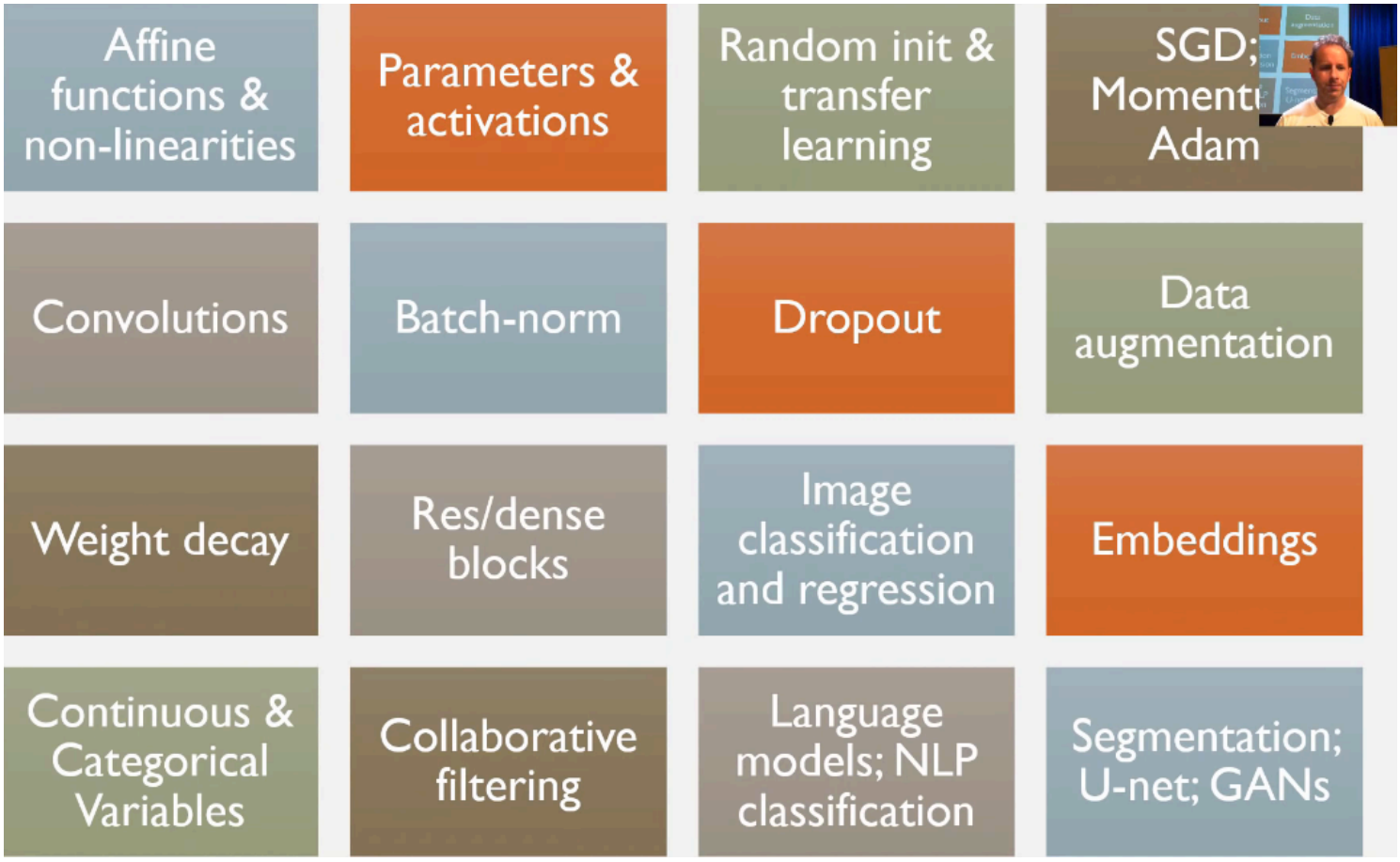
Yeah, pretty much. We don’t fully know yet. It’s a pretty new area, but there’s a lot of opportunities there. And we’ll be looking at some in a moment actually.



I actually tried testing this on this. Remember this picture I showed you of a slide last lesson, and it's a really rubbishy looking picture. And I thought, what would happen if we tried running this just through the exact same model. And it changed it from that (left) to that (right) so I thought that was a really good example. You can see something it didn't do which is this weird discoloration. It didn't fix it because I didn't crappify things with weird discoloration. So if you want to create really good image restoration like I say you need really good crappification.



What we learned in the class so far [1:36:59]



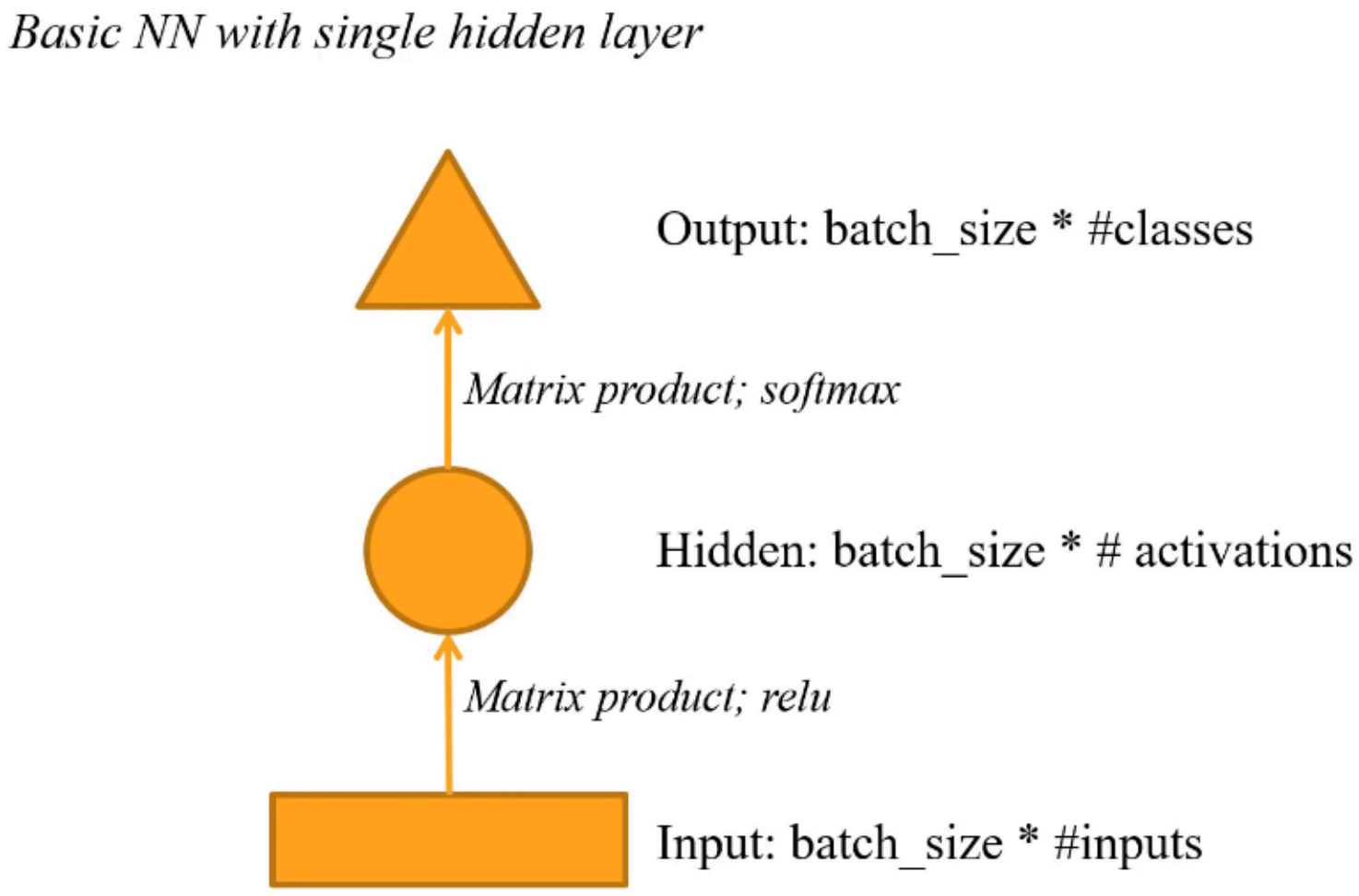
Here's some of the main things we've learned so far in the course. We've learned that neural nets consist of sandwich layers of affine functions (which are basically matrix multiplications, slightly more general version) and nonlinearities like ReLU. And we learnt that the results of those calculations are called activations, and the things that go into those calculations we learned are called parameters. The parameters are initially randomly initialized or we copy them over from a pre-trained model, and then we train them with SGD or faster versions. We learnt that convolutions are a particular affine function that work great for auto correlated data, so things like images and stuff. We learnt about batch norm, dropout, data augmentation, and weight decay as ways of regularizing models. Also batch norm helps train models more quickly.

Then today, we've learned about Res/Dense blocks. We ovbiously learnt a lot about image classification and regression, embeddings, categorical and continuous variables, collaborative filtering, language models, and NLP classification. Then segmentation, U-Net and GANs.

So go over these things and make sure that you feel comfortable with each of them. If you've only watched this series once, you definitely won't. People normally watch it three times or so to really understand the detail.

Recurrent Neural Network (RNN) [1:38:31]

One thing that doesn't get here is RNNs. So that's the last thing we're going to do. RNNs; I'm going to introduce a little diagrammatic method here to explain to RNNs, and the diagrammatic method, I'll start by showing your basic neural net with a single hidden layer.

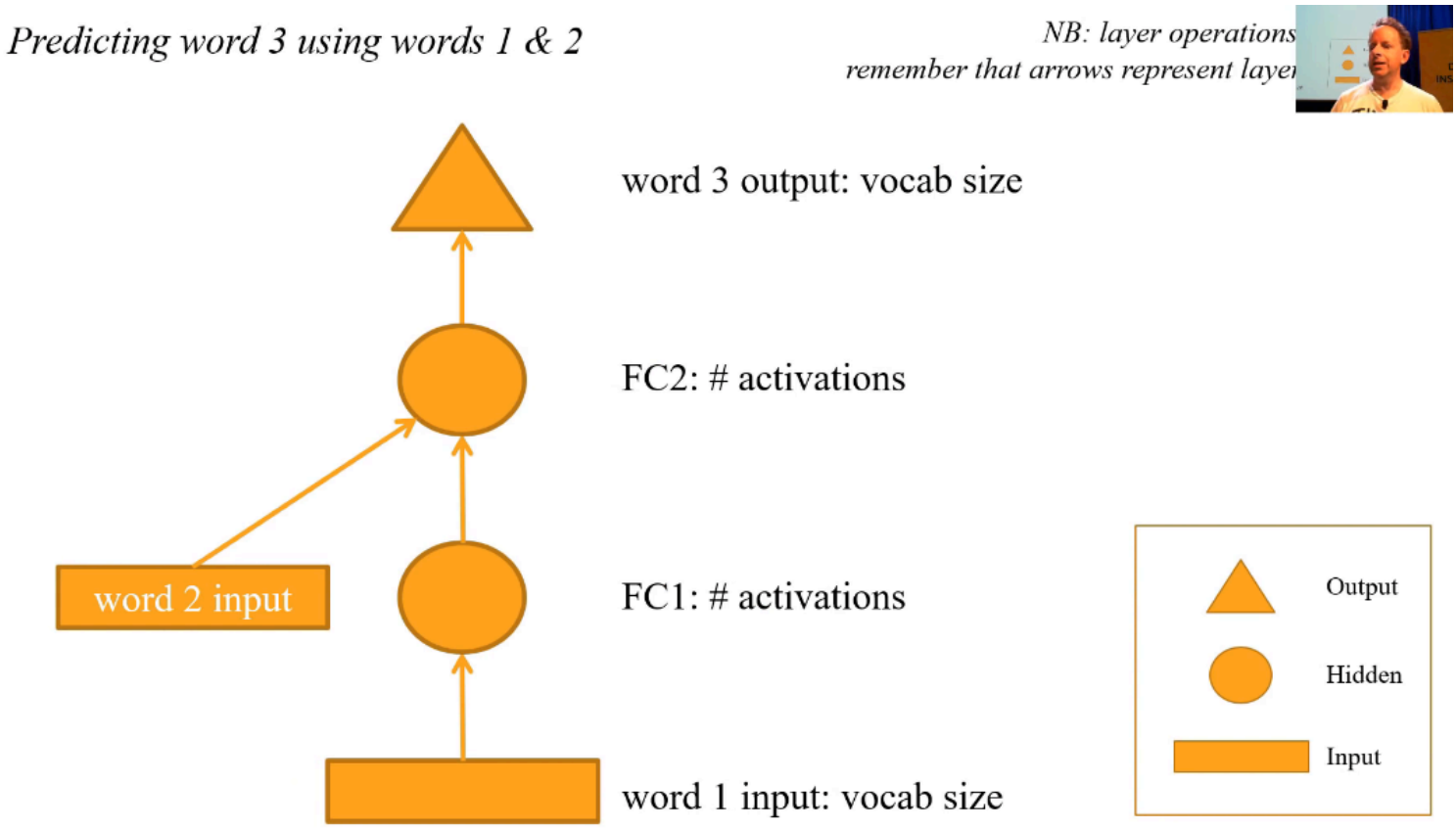


Rectangle means an input. That'll be batch size by number of inputs. An **arrow** means a layer (broadly defined) such as matrix product followed by ReLU. A **circle** is activations. So this case, we have one set of hidden

activations and so given that the input was number of inputs, this here (the first arrow) is a matrix of number of inputs by number of activations. So the output will be batch size by a number of activations.

It's really important you know how to calculate these shapes. So go `learn.summary()` lots to see all the shapes. Then here's another arrow, so that means it's another layer; matrix product followed by non-linearity. In this case, we go into the output, so we use softmax.

Then **triangle** means an output. This matrix product will be number of activations by a number of classes, so our output is batch size by number classes.

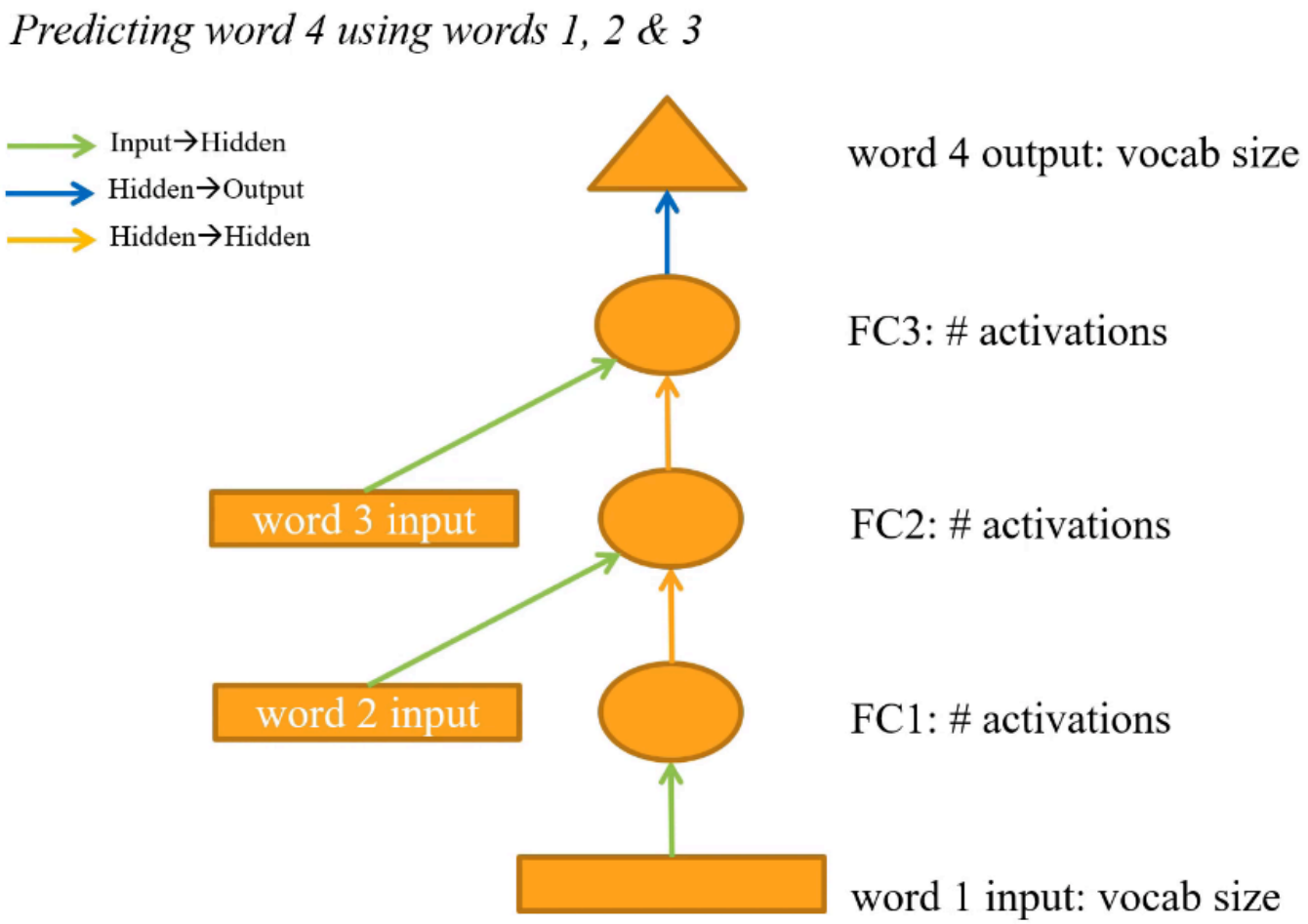


Let's reuse the that key; triangle is output, circle is activations (hidden state - we also call that) and rectangle is input. Let's now imagine that we wanted to get a big document, split it into sets of three words at a time, and grab each set of three words and then try to predict the third word using the first two words. If we had the dataset in place, we could:

1. Grab word 1 as an input.
2. Chuck it through an embedding, create some activations.
3. Pass that through a matrix product and nonlinearity.
4. Grab the second word.
5. Put it through an embedding.
6. Then we could either add those two things together or concatenate them. Generally speaking, when you see two sets of activations coming together in a diagram, you normally have a choice of concatenate or or add. And that's going to create the second bunch of activations.
7. Then you can put it through one more fully connected layer and softmax to create an output.

So that would be a totally standard, fully connected neural net with one very minor tweak which is concatenating or adding at this point, which we could use to try to predict the third word from pairs of two words.

Remember, arrows represent layer operations and I removed in this one the specifics of what they are because they're always an affine function followed by a non-linearity.



Let's go further. What if we wanted to predict word 4 using words 1 and 2 and 3? It's basically the same picture as last time except with one extra input and one extra circle. But I want to point something out which is each time we go from rectangle to circle, we're doing the same thing - we're doing an embedding. Which is just a particular kind of matrix multiply where you have a one hot encoded input. Each time we go from circle to circle, we're basically taking one piece of hidden state (a.k.a activations) and turning it into another set of activations by saying we're now at the next word. Then when we go from circle to triangle, we're doing something else again which is we're saying let's convert the hidden state (i.e. these activations) into an output. So I've colored each of those arrows differently. Each of those arrows should probably use the same weight matrix because it's doing the same thing. So why would you have a different set of embeddings for each word or a different matrix to multiply by to go from this hidden state to this hidden state versus this one? So this is what we're going to build.

Human numbers [1:43:11]

We're now going to jump into human numbers which is lesson7-human-numbers.ipynb. This is a dataset that I created which literally just contains all the numbers from 1 to 9,999 written out in English.

We're going to try to create a language model that can predict the next word in this document. It's just a toy example for this purpose. In this case, we only have one document. That one document is the list of numbers. So we can use a `TextList` to create an item list with text in for the training of the validation.

```
from fastai.text import *
```

```
bs=64

path = untar_data(URLs.HUMAN_NUMBERS)
path.ls()

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

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

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

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

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

' nine thousand nine hundred ninety eight, nine thousand nine hundred ninety nine

train = TextList(train_txt, path=path)
valid = TextList(valid_txt, path=path)

src = ItemLists(path=path, train=train, valid=valid).label_for_lm()
data = src.databunch(bs=bs)
```

In this case, the validation set is the numbers from 8,000 onwards, and the training set is 1 to 8,000. We can combine them together, turn that into a data bunch.

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

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

We only have one document, so `train[0]` is the document grab its `.text` that's how you grab the contents of a text list, and here are the first 80 characters. It starts with a special token `xxbos`. Anything starting with `xx` is a special fast.ai token, `bos` is the beginning of stream token. It basically says this is the start of a document, and it's very helpful in NLP to know when documents start so that your models can learn to recognize them.

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

13017
```

The validation set contains 13,000 tokens. So 13,000 words or punctuation marks because everything between spaces is a separate token.

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

(70, 3)

13017/70/bs

2.905580357142857
```

The batch size that we asked for was 64, and then by default it uses something called `bptt` of 70. `bptt`, as we briefly mentioned, stands for "back prop through time". That's the sequence length. For each of our 64 document segments, we split it up into lists of 70 words that we look at at one time. So what we do for the validation set is we grab this entire string of 13,000 tokens, and then we split it into 64 roughly equal sized sections. People very very very often think I'm saying something different. I did not say "they are of length 64" - they're not. They're **64 roughly equally sized segments**. So we take the first 1/64 of the document - piece 1. The second 1/64 - piece 2.

Then for each of those 1/64 of the document, we then split those into pieces of length 70. So let's now say for those 13,000 tokens, how many batches are there? Well, divide by batch size and divide by 70, so there's going to be 3 batches.

```
it = iter(data.valid_dl)
x1,y1 = next(it)
x2,y2 = next(it)
x3,y3 = next(it)
it.close()

x1.numel()+x2.numel()+x3.numel()

12928
```

Let's grab an iterator for a data loader, grab 1 2 3 batches (the X and the Y), and let's add up the number of elements, and we get back slightly less than 13,017 because there's a little bit left over at the end that doesn't quite make up a full batch. This is the kind of stuff you should play around with a lot - lots of shapes and sizes and stuff and iterators.

```
x1.shape,y1.shape

(torch.Size([95, 64]), torch.Size([95, 64]))

x2.shape,y2.shape

(torch.Size([69, 64]), torch.Size([69, 64]))
```

As you can see, it's 95 by 64. I claimed it was going to be 70 by 64. That's because our data loader for language models slightly randomizes `bptt` just to give you a bit more shuffling - get bit more randomization - it helps the model.

```
x1[:,0]

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

y1[:,0]

tensor([18, 10, 11,  8, 18, 10, 12,  8, 18, 10, 13,  8, 18, 10, 14,  8, 18, 10,
        15,  8, 18, 10, 16,  8, 18, 10, 17,  8, 18, 10, 18,  8, 18, 10, 19,  8,
```

```
        18, 10, 28,  8, 18, 10, 29,  8, 18, 10, 30,  8, 18, 10, 31,  8, 18, 10,
        32,  8, 18, 10, 33,  8, 18, 10, 34,  8, 18, 10, 35,  8, 18, 10, 36,  8,
        18, 10, 37,  8, 18, 10, 20,  8, 18, 10, 20, 11,  8, 18, 10, 20, 12,  8,
        18, 10, 20, 13,  8], device='cuda:0')
```

So here, you can see the first batch of X (remember, we've numeric alized all these) and here's the first batch of Y. And you'll see here `x1` is `[2, 18, 10, 11, 8, ...]`, `y1` is `[18, 10, 11, 8, ...]`. So `y1` is offset by 1 from `x1`. Because that's what you want to do with a language model. We want to predict the next word. So after 2, should come 18, and after 18, should come 10.

```
v = data.valid_ds.vocab
```

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

```
'xxbos eight thousand one , eight thousand two , eight thousand three , eight tho
```

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

```
'eight thousand one , eight thousand two , eight thousand three , eight thousand
```

You can grab the vocab for this dataset, and a vocab has a `textify` so if we look at exactly the same thing but with `textify`, that will just look it up in the vocab. So here you can see `xxbos eight thousand one` where else in the `y`, there's no `xxbos`, it's just `eight thousand one`. So after `xxbos` is `eight`, after `eight` is `thousand`, after `thousand` is `one`.

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

```
', eight thousand twenty four , eight thousand twenty five , eight thousand twent
```

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

```
', eight thousand thirty eight , eight thousand thirty nine , eight thousand fort
```

Then after we get 8023, comes `x2`, and look at this, we're always looking at column 0, so this is the first batch (the first mini batch) comes 8024 and then `x3`, all the way up to 8,040 .

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

```
', eight thousand forty six , eight thousand forty seven , eight thousand forty e
```

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

```
'thousand sixty five , eight thousand sixty six , eight thousand sixty seven , ei
```

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

```
'thousand seventy nine , eight thousand eighty , eight thousand eighty one , eigh
```

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

```
'ninety , nine thousand nine hundred ninety one , nine thousand nine hundred nine
```

Then we can go right back to the start, but look at batch index 1 which is batch number 2. Now we can continue. A slight skip from 8,040 to 8,046, that's because the last mini batch wasn't quite complete. What this means is that every mini batch joins up with a previous mini batch. So you can go straight from x1[0] to x2[0] - it continues 8,023, 8,024. If you took the same thing for `:,1`, you'll also see they join up. So all the mini batches join up.

```
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

That's the data. We can do show_batch to see it.

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

x,y = data.one_batch()
x.shape,y.shape
```

```
(torch.Size([3, 64]), torch.Size([3, 64]))
```

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

```
38
```

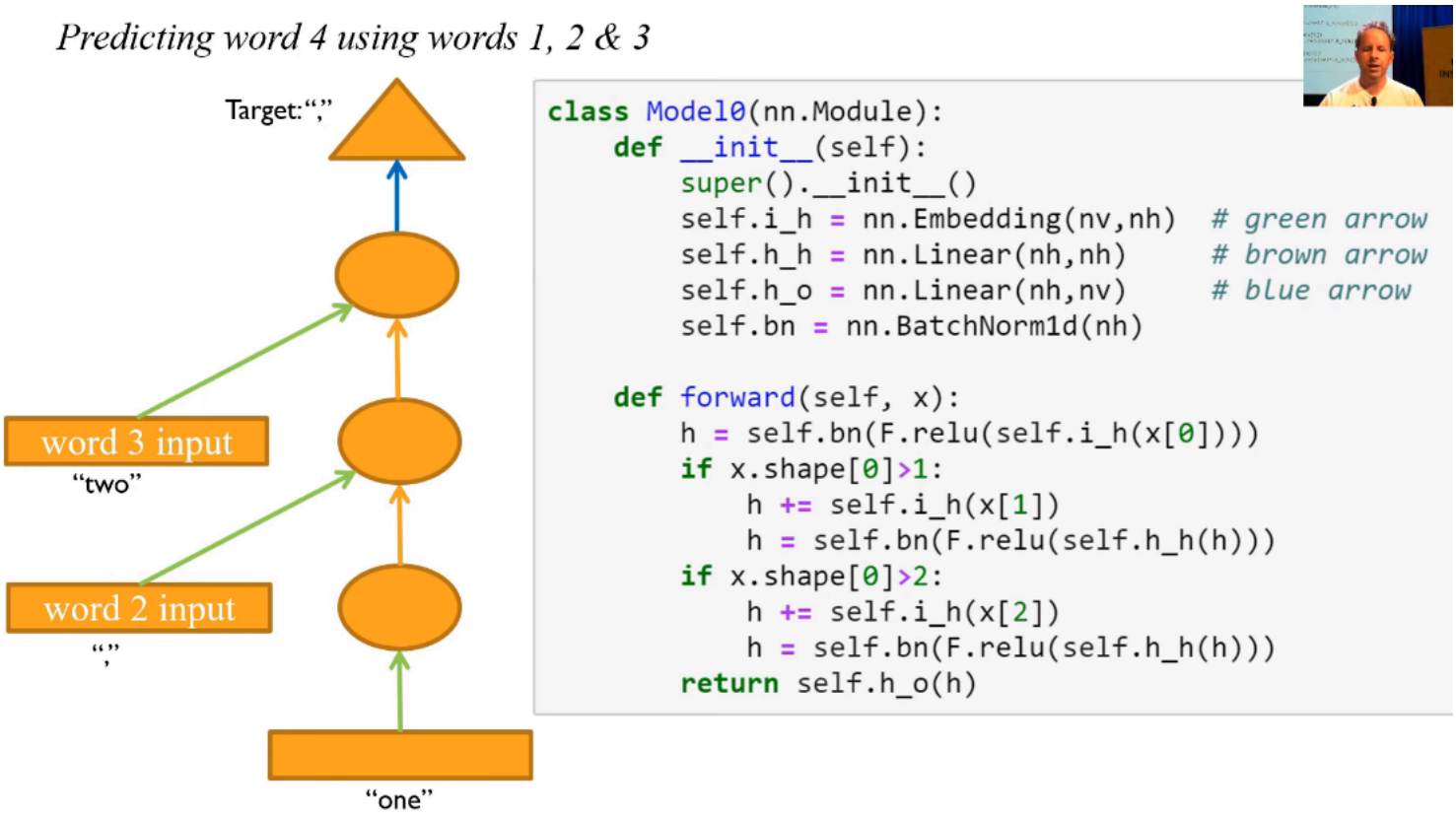
```
nh=64

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

Here is our model which is doing what we saw in the diagram:

```
class Model0(nn.Module):
    def __init__(self):
```

```
        super().__init__()
        self.i_h = nn.Embedding(nv,nh)  # green arrow
        self.h_h = nn.Linear(nh,nh)     # brown arrow
        self.h_o = nn.Linear(nh,nv)     # blue arrow
        self.bn = nn.BatchNorm1d(nh)

    def forward(self, x):
        h = self.bn(F.relu(self.i_h(x[0])))
        if x.shape[0]>1:
            h += self.i_h(x[1])
            h = self.bn(F.relu(self.h_h(h)))
        if x.shape[0]>2:
            h += self.i_h(x[2])
            h = self.bn(F.relu(self.h_h(h)))
        return self.h_o(h)
```

This is just a code copied over:



It content contains 1 embedding (i.e. the green arrow), one hidden to hidden - brown arrow layer, and one hidden to output. So each colored arrow has a single matrix. Then in the forward pass, we take our first input `x[0]` and put it through input to hidden (the green arrow), create our first set of activations which we call `h`. Assuming that there is a second word, because sometimes we might be at the end of a batch where there isn't a second word. Assume there is a second word then we would add to `h` the result of `x[1]` put through the green arrow (that's `i_h`). Then we would say, okay our new `h` is the result of those two added together, put through our hidden to hidden (orange arrow), and then ReLU then batch norm. Then for the second word, do exactly the same thing. Then finally blue arrow - put it through `h_o`.

So that's how we convert our diagram to code. Nothing new here at all. We can chuck that in a learner, and we can

train it - 46%.

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

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

Total time: 00:05

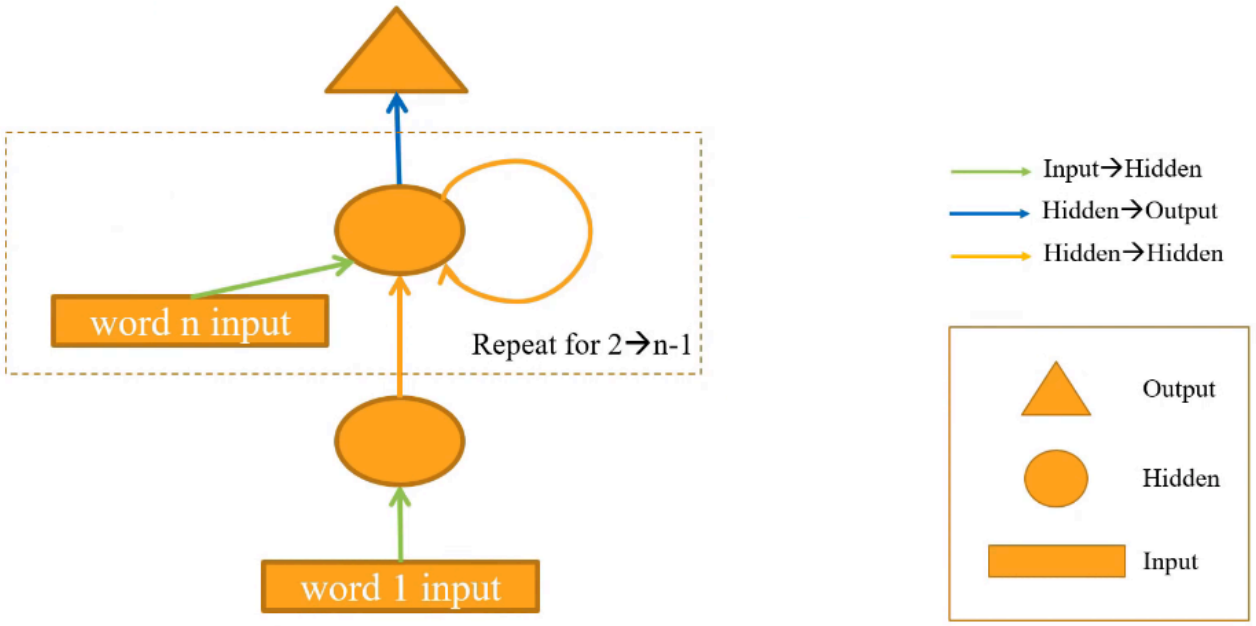
epoch	train_loss	valid_loss	acc4
1	3.533459	3.489706	0.098855
2	3.011390	3.033105	0.450031
3	2.452748	2.552569	0.461247
4	2.154685	2.315783	0.461711
5	2.039904	2.236383	0.462020
6	2.016217	2.225322	0.462020

Same thing with a loop [1:50:48]

Let's take this code and recognize it's pretty awful. There's a lot of duplicate code, and as coders, when we see duplicate code, what do we do? We refactor. So we should refactor this into a loop.

```
class Model1(nn.Module):
    def __init__(self):
        super().__init__()
        self.i_h = nn.Embedding(nv,nh)  # green arrow
        self.h_h = nn.Linear(nh,nh)     # brown arrow
        self.h_o = nn.Linear(nh,nv)     # blue arrow
        self.bn = nn.BatchNorm1d(nh)

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

Here we are. We've refactored it into a loop. So now we're going for each `xi` in `x`, and doing it in the loop. Guess what? That's an RNN. An RNN is just a refactoring. It's not anything new. This is now an RNN. And let's refactor our diagram:



This is the same diagram, but I've just replaced it with my loop. It does the same thing, so here it is. It's got exactly the same `__init__`, literally exactly the same, just popped a loop here. Before I start, I just have to make sure that I've got a bunch of zeros to add to. And of course, I get exactly the same result when I train it.

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

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

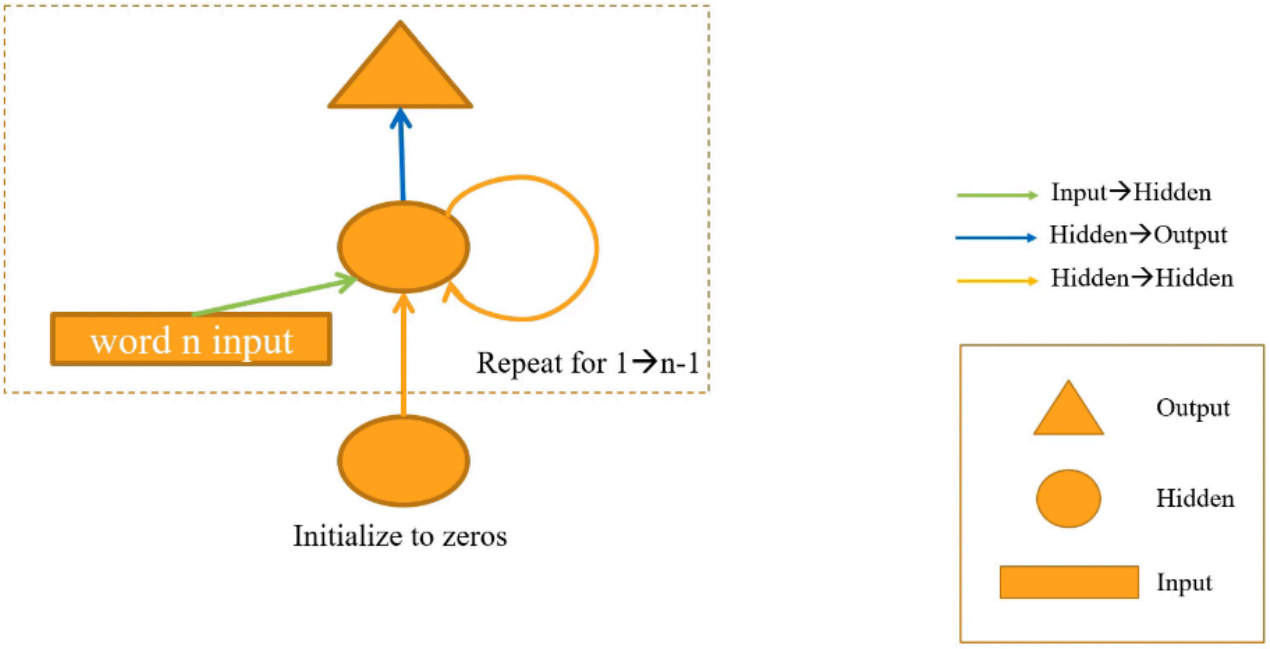
One nice thing about the loop though, is now this will work even if I'm not predicting the fourth word from the previous three, but the ninth word from the previous eight. It'll work for any arbitrarily length long sequence which is nice.

So let's up the `bptt` to 20 since we can now. And let's now say, okay, instead of just predicting the nth word from the previous $n-1$, let's try to predict the second word from the first, the third from the second, and the fourth from the third, and so forth. Look at our loss function.

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

Previously we were comparing the result of our model to just the last word of the sequence. It is very wasteful, because there's a lot of words in the sequence. So let's compare every word in `x` to every word and `y`. To do that, we need to change the diagram so it's not just one triangle at the end of the loop, but the triangle is inside the loop:

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



In other words, after every loop, predict, loop, predict, loop, predict.

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

x,y = data.one_batch()
x.shape,y.shape

(torch.Size([45, 64]), torch.Size([45, 64]))

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)
```

Here's this code. It's the same as the previous code, but now I've created an array, and every time I go through the loop, I append `h_o(h)` to the array. Now, for n inputs, I create n outputs. So I'm predicting after every word.

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

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

Previously I had 46%, now I have 40%. Why is it worse? It's worse because now when I'm trying to predict the second word, I only have one word of state to use. When I'm looking at the third word, I only have two words of state to use. So it's a much harder problem for it to solve. The key problem is here:

```
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)
```

I go `h = torch.zeros`. I reset my state to zero every time I start another BPTT sequence. Let's not do that. Let's keep `h`. And we can, because remember, each batch connects to the previous batch. It's not shuffled like happens in image classification. So let's take this exact model and replicate it again, but let's move the creation of `h` into the constructor.

```
class Model3(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)
        self.h = torch.zeros(x.shape[1], nh).cuda()

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

There it is. So it's now `self.h`. So this is now exactly the same code, but at the end, let's put the new `h` back into `self.h`. It's now doing the same thing, but it's not throwing away that state.

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

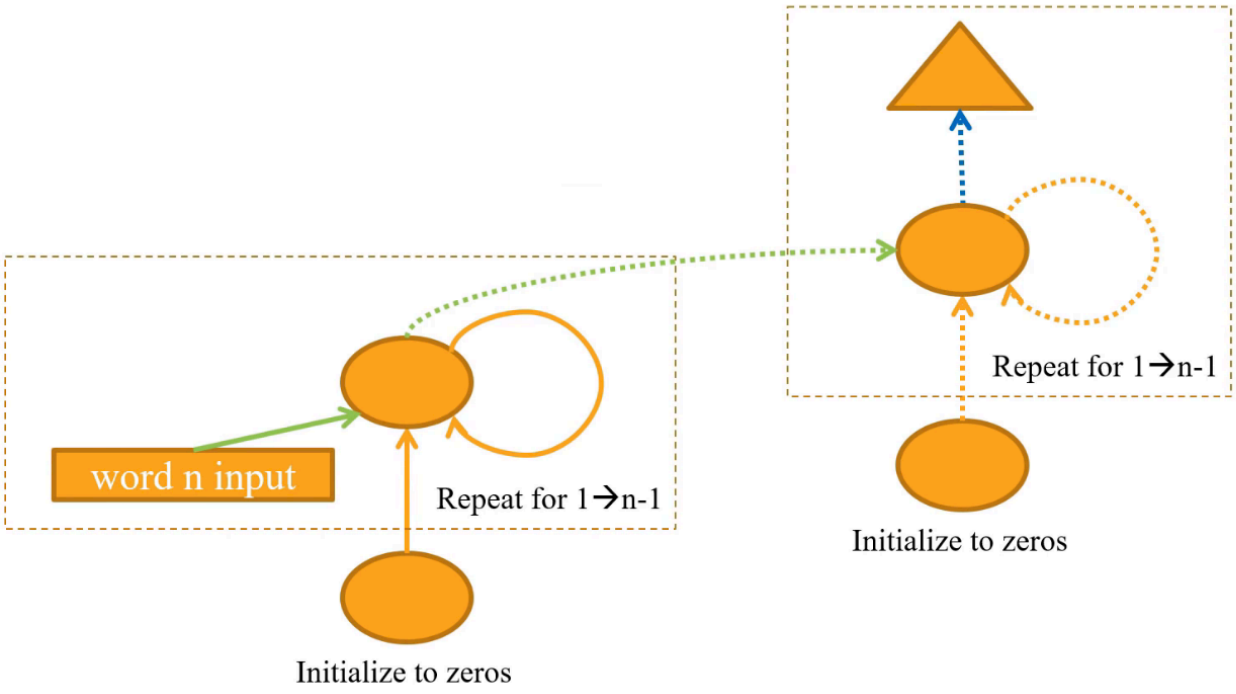
```
learn.fit_one_cycle(20, 3e-3)
```

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

Therefore, now we actually get above the original. We get all the way up to 54% accuracy. So this is what a real RNN looks like. You always want to keep that state. But just keep remembering, there's nothing different about an RNN, and it's a totally normal fully connected neural net. It's just that you've got a loop you refactored.

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



What you could do though is at the end of your every loop, you could not just spit out an output, but you could spit it out into another RNN. So you have an RNN going into an RNN. That's nice because we now got more layers of computation, you would expect that to work better.

To get there, let's do some more refactoring. Let's take this code (`Model3`) and replace it with the equivalent built in PyTorch code which is you just say that:

```
class Model4(nn.Module):
    def __init__(self):
        super().__init__()
        self.i_h = nn.Embedding(nv,nh)
        self.rnn = nn.RNN(nh,nh)
        self.h_o = nn.Linear(nh,nv)
        self.bn = nn.BatchNorm1d(nh)
        self.h = torch.zeros(1, x.shape[1], nh).cuda()

    def forward(self, x):
        res,h = self.rnn(self.i_h(x), self.h)
        self.h = h.detach()
        return self.h_o(self.bn(res))
```

So `nn.RNN` basically says do the loop for me. We've still got the same embedding, we've still got the same output, still got the same batch norm, we still got the same initialization of `h`, but we just got rid of the loop. One of the nice things about RNN is that you can now say how many layers you want. This is the same accuracy of course:

```
learn = Learner(data, Model4(), metrics=accuracy)
```

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

Total time: 00:04

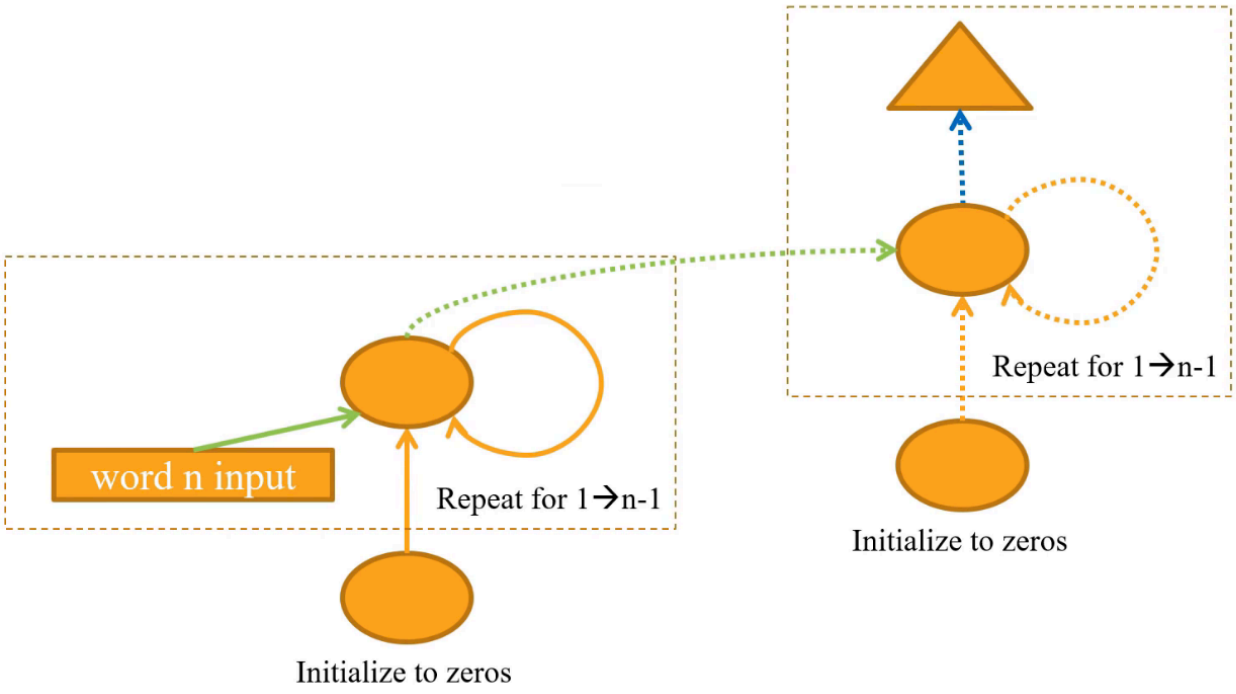
epoch	train_loss	valid_loss	accuracy
1	3.502738	3.372026	0.252707
2	3.092665	2.714043	0.457998
3	2.538071	2.189881	0.467048
4	2.057624	1.946149	0.451719
5	1.697061	1.923625	0.466471
6	1.424962	1.916880	0.487856
7	1.221850	2.029671	0.507735
8	1.063150	1.911920	0.523128
9	0.926894	1.882562	0.541045
10	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

So here, I'm going to do it with two layers:

```
class Model5(nn.Module):
    def __init__(self):
        super().__init__()
        self.i_h = nn.Embedding(nv,nh)
        self.rnn = nn.GRU(nh,nh,2)
        self.h_o = nn.Linear(nh,nv)
        self.bn = nn.BatchNorm1d(nh)
        self.h = torch.zeros(2, bs, nh).cuda()

    def forward(self, x):
        res,h = self.rnn(self.i_h(x), self.h)
        self.h = h.detach()
        return self.h_o(self.bn(res))
```

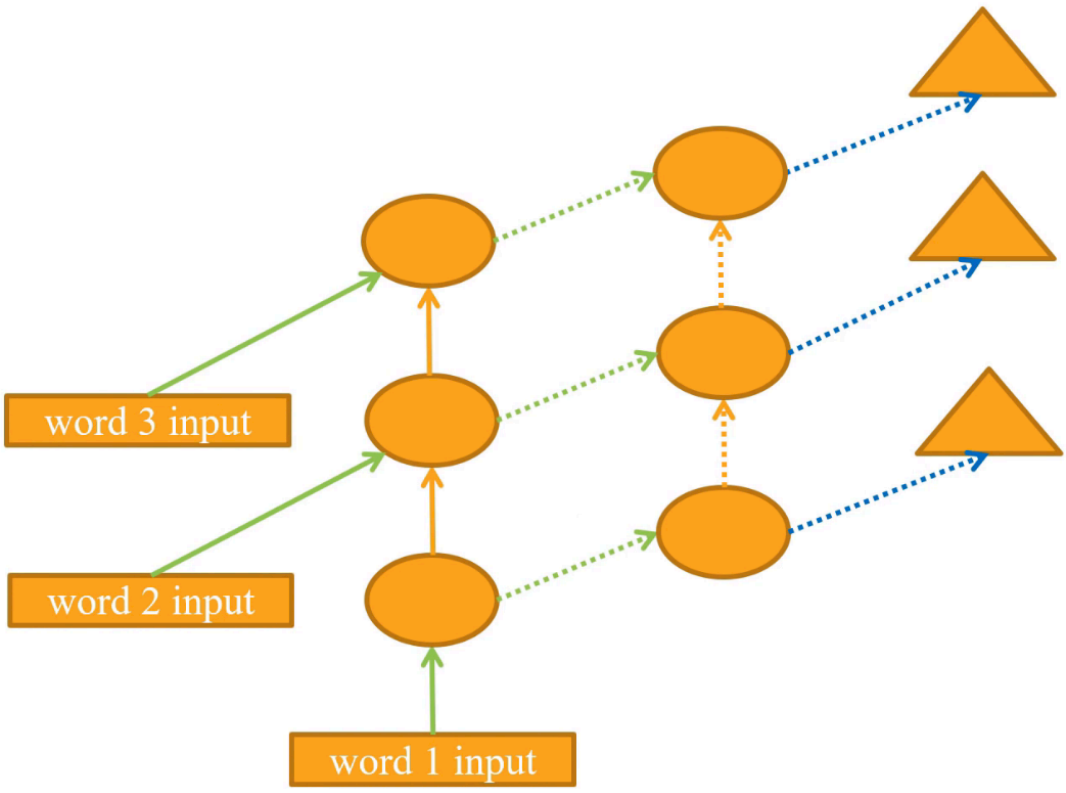
But here's the thing. When you think about this:

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



Think about it without the loop. It looks like this:

Unrolled stacked RNNs for sequences



It keeps on going, and we've got a BPTT of 20, so there's 20 layers of this. And we know from that visualizing the loss landscapes paper, that deep networks have awful bumpy loss surfaces. So when you start creating long timescales and multiple layers, these things get impossible to train. There's a few tricks you can do. One thing is

you can add skip connections, of course. But what people normally do is, instead of just adding these together(green and orange arrows), they actually use a little mini neural net to decide how much of the green arrow to keep and how much of the orange arrow to keep. When you do that, you get something that's either called GRU or LSTM depending on the details of that little neural net. And we'll learn about the details of those neural nets in part 2. They really don't matter though, frankly.

So we can now say let's create a GRU instead. It's just like what we had before, but it'll handle longer sequences in deeper networks. Let's use two layers.

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

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

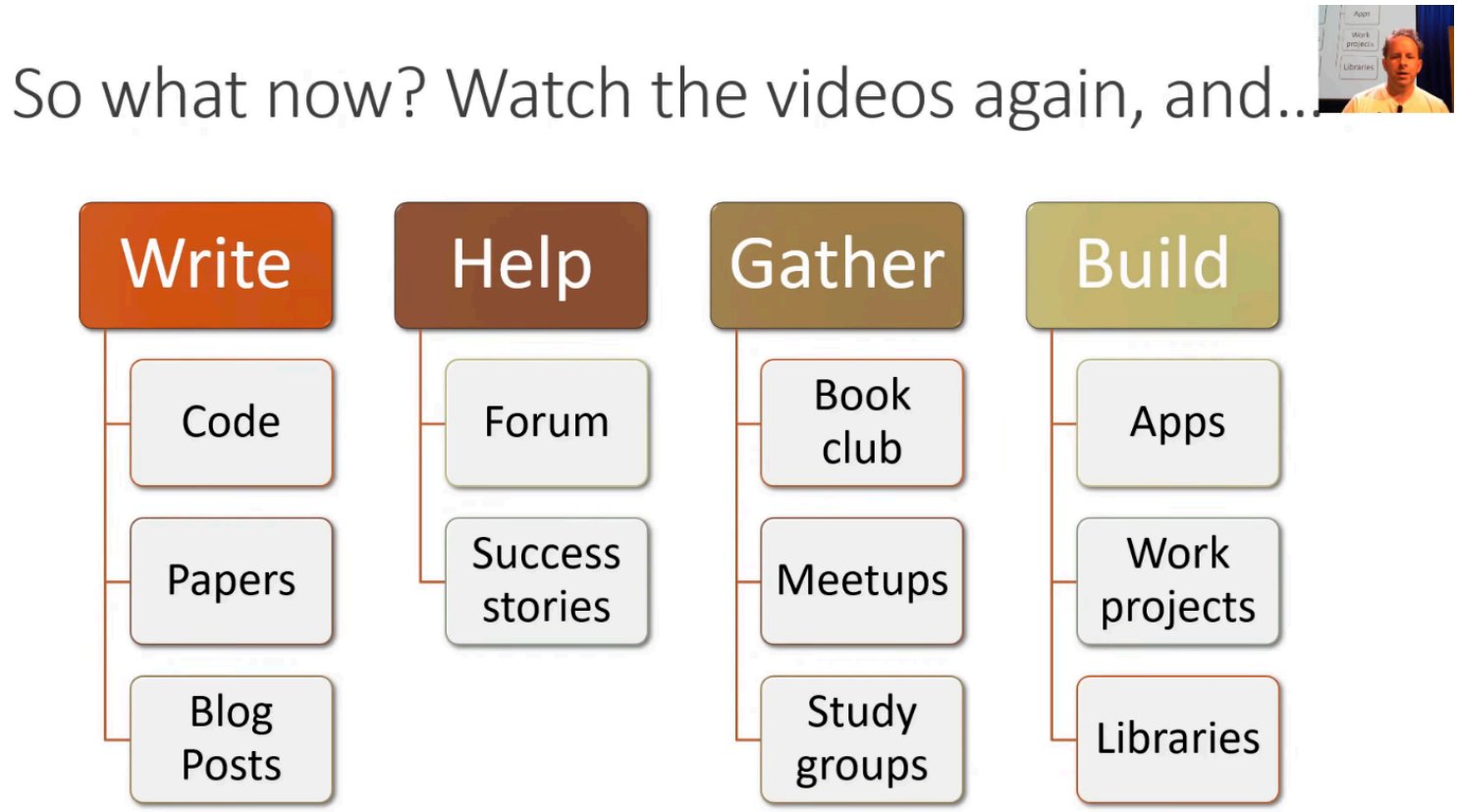
Total time: 00:02

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

And we're up to 75%. That's RNNs and the main reason I wanted to show it to you was to remove the last remaining piece of magic, and this is one of the least magical things we have in deep learning. It's just a refactored fully connected network. So don't let RNNs ever put you off. With this approach where you basically have a sequence of n inputs and a sequence of n outputs we've been using for language modeling, you can use that for other tasks.

For example, the sequence of outputs could be for every word there could be something saying is there something that is sensitive and I want to anonymize or not. So it says private data or not. Or it could be a part of speech tag for that word, or it could be something saying how should that word be formatted, or whatever. These are called **sequence labeling tasks** and so you can use this same approach for pretty much any sequence labeling task. Or you can do what I did in the earlier lesson which is once you finish building your language model, you can throw away the `h_o` bit, and instead pop there a standard classification head, and then you can now do NLP classification which as you saw earlier will give you a state of the art results even on long documents. So this is a super valuable technique, and not remotely magical.

What now? [1:58:59]

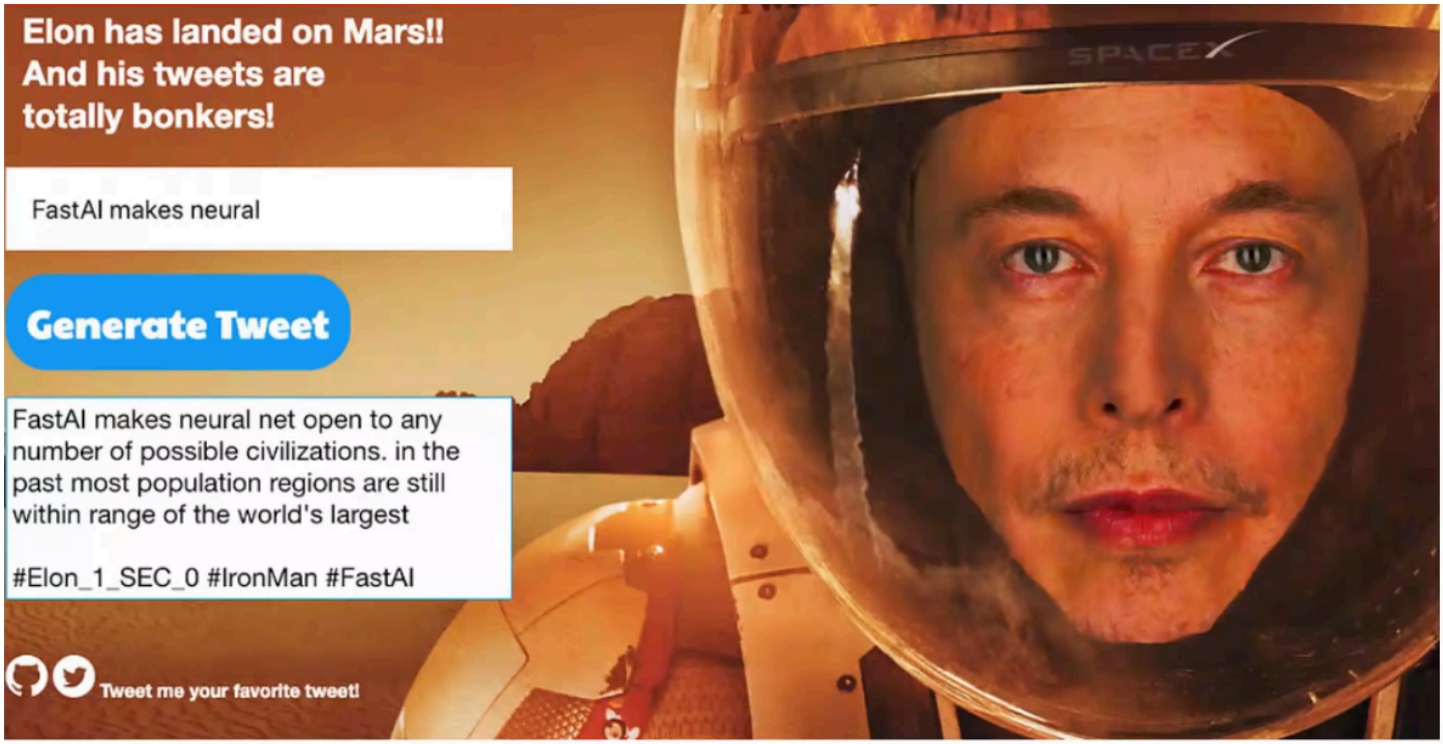


That's it. That's deep learning, or at least the practical pieces from my point of view. Having watched this one time, you won't get it all. And I don't recommend that you do watch this so slowly that you get it all the first time, but you go back and look at it again, take your time, and there'll be bits that you go like "oh, now I see what he's saying" and then you'll be able to implement things you couldn't before or you'll be able to dig in more than before. So definitely go back and do it again. And as you do, write code, not just for yourself but put it on github. It doesn't matter if you think it's great code or not. The fact that you're writing code and sharing it is impressive and the feedback you'll get if you tell people on the forum "hey, I wrote this code. It's not great but it's my first effort. Anything you see jump out at you?" People will say like "oh, that bit was done well. But you know, for this bit, you could have used this library and saved you sometime." You'll learn a lot by interacting with your peers.

As you've noticed, I've started introducing more and more papers. Part 2 will be a lot of papers, so it's a good time to start reading some of the papers that have been introduced in this section. All the bits that say like derivation and theorems and lemmas, you can skip them. I do. They add almost nothing to your understanding of your practical deep learning. But the bits that say why are we solving this problem, and what are the results, and so forth, are really interesting. Then try and write English prose. Not English prose that you want to be read by Geoffrey Hinton and Yann LeCun, but English prose you want to be read by you as of six months ago. Because there's a lot more people in the audience of you as of six months ago than there is of Geoffrey Hinton and Yann LeCun. That's the person you best understand. You know what they need.

Go and get help, and help others. Tell us about your success stories. But perhaps the most important one is get together with others. People's learning works much better if you've got that social experience. So start a book club, get involved in meetups, create study groups, and build things. Again, it doesn't have to be amazing. Just build something that you think the world would be a little bit better if that existed. Or you think it would be kind of slightly delightful to your two-year-old to see that thing. Or you just want to show it to your brother the next time they come around to see what you're doing, whatever. Just finish something. Finish something and then try make it

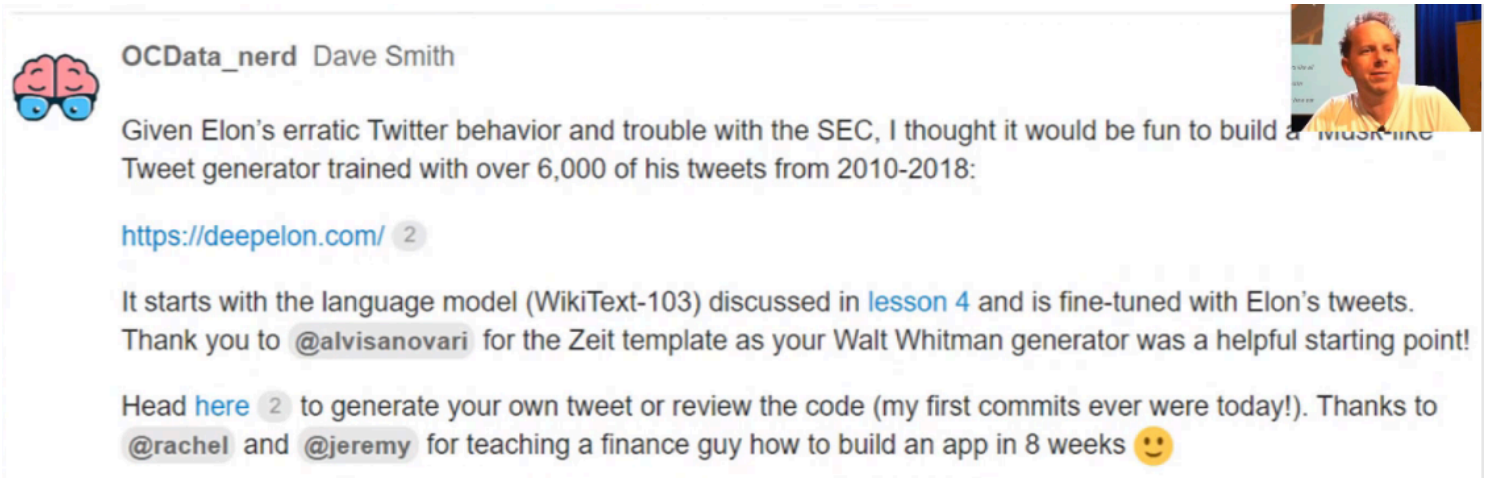
a bit better.



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"*

For example something I just saw this afternoon is the Elon Musk tweet generator. Looking at lots of older tweets, creating a language model from Elon Musk, and then creating new tweets such as "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."



So this is great. I love this. And I love that Dave Smith wrote and said "these are my first-ever commits. Thanks for

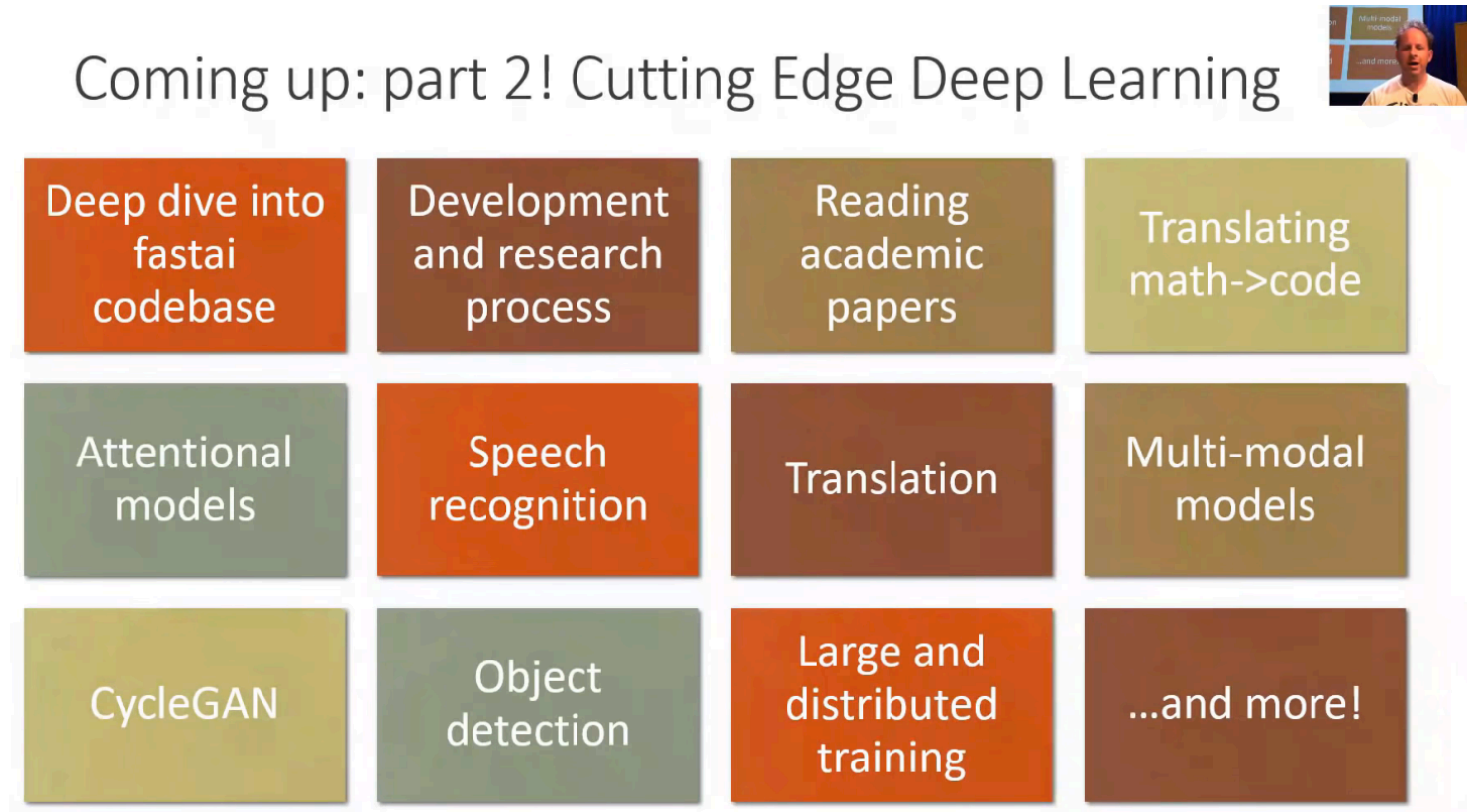
teaching a finance guy how to build an app in eight weeks". I think this is awesome and I think clearly a lot of care and passion is being put into this project. Will it systematically change the future direction of society as a whole? Maybe not. But maybe Elon will look at this and think "oh, maybe I need to rethink my method of prose," I don't know. I think it's great. So yeah, create something, put it out there, put a bit of yourself into it.

Or get involved in the fast.ai. The fast.ai project, there's a lot going on. You can help with documentation and tests which might sound boring but you'd be surprised how incredibly not boring it is to take a piece of code that hasn't been properly documented, and research it, and understand it, and ask Sylvain and I on the forum; what's going on? Why did you write it this way? We'll send you off to the papers that we were implementing. Writing a test requires deeply understanding that part of the machine learning world to really understand how it's meant to work. So that's always interesting.

Stats Bakman has created this nice Dev Projects Index which you can go onto the forum in the fast.ai dev projects section and find like here's some stuff going on that you might want to get involved in. Maybe there's stuff you want to exist you can add your own.

Create a study group you know Deena's already created a study group for San Francisco starting in January. This is how easy it is to create a study group. Go on the forum, find your little timezone subcategory, and add a post saying let's create a study group. But make sure you give people like a Google sheet to sign up, some way to actually do something.

A great example is Pierre who's been doing a fantastic job in Brazil of running study groups for the last couple of parts of the course. And he keeps posting these pictures of people having a good time and learning deep learning together, creating wiki's together, creating projects together - great experience.



Then come back for part 2 where we'll be looking at all of this interesting stuff. In particular, going deep into the fast.ai code base to understand how did we build it exactly. We will actually go through. As we were building it, we created notebooks of here's where we were at each stage, so we're actually going to see the software development process itself. We'll talk about the process of doing research; how to read academic papers, how to

turn math into code. Then a whole bunch of additional types of models that we haven't seen yet. So it'll be kind of like going beyond practical deep learning into actually cutting-edge research.

Ask Jeremy Anything [2:05:26]

We had an AMA going on online and so we're going to have time for a couple of the highest ranked AMA questions from the community

Question: The first one is by Jeremy's request, although it's not the highest ranked. What's your typical day like? How do you manage your time across so many things that you do?

I hear that all the time, so I thought I should answer it, and I think it got a few votes. People who come to our study group are always shocked at how disorganized and incompetent I am, and so I often hear people saying like "oh wow, I thought you were like this deep learning role model and I'd get to see how to be like you and now I'm not sure I want to be like you at all." For me, it's all about just having a good time with it. I never really have many plans. I just try to finish what I start. If you're not having fun with it, it's really really hard to continue because there's a lot of frustration in deep learning. Because it's not like writing a web app where it's like authentication, check, backend service watchdog, check, user credentials, check - you're making progress. Where else, for stuff like this GAN stuff that we've been doing the last couple of weeks, it's just like; it's not working, it's not working, it's not working, no it also didn't work, and it also didn't work, until like "OH MY GOD IT'S AMAZING. It's a cat." That's kind of what it is. So you don't get that regular feedback. So yeah, you gotta have fun with it. The other thing, I don't do any meetings. I don't do phone calls. I don't do coffees. I don't watch TV. I don't play computer games. I spend a lot of time with my family. A lot of time exercising, and a lot time reading, and coding, and doing things I like. So the main thing is just finish something. Like properly finish it. So when you get to that point where you think 80% of the way through, but you haven't quite created a README yet, and the install process is still a bit clunky - this is what 99% of github projects look like. You'll see the README says "TODO: complete baseline experiments document blah blah blah." Don't be that person. Just do something properly and finish it and maybe get some other people around you to work with you so that you're all doing it together and get it done.

Question: What are the up-and-coming deep learning/machine learning things that you're most excited about? Also you've mentioned last year that you are not a believer in reinforcement learning. Do you still feel the same way?

I still feel exactly the same way as I did three years ago when we started this which is it's all about transfer learning. It's under-appreciated, it's under-researched. Every time we put transfer learning into anything, we make it much better. Our academic paper on transfer learning for NLP has helped be one piece of changing the direction of NLP this year, made it all the way to the New York Times - just a stupid, obvious little thing that we threw together. So I remain excited about that. I remain unexcited about reinforcement learning for most things. I don't see it used by normal people for normal things for nearly anything. It's an incredibly inefficient way to solve problems which are often solved more simply and more quickly in other ways. It probably has (maybe?) a role in the world but a limited one and not in most people's day-to-day work.

Question: For someone planning to take part 2 in 2019, what would you recommend doing learning practicing until the part 2 of course starts?

Just code. Yeah, just code all the time. I know it's perfectly possible, I hear from people who get to this point of the course and they haven't actually written any code yet. And if that's you, it's okay. You just go through and do it again, and this time do code. And look at the shapes of your inputs, look at your outputs, make sure you know how to grab a mini batch, look at its mean and standard deviation, and plot it. There's so much material that we've covered. If you can get to a point where you can rebuild those notebooks from scratch without too much cheating,

when I say from scratch, I mean using the fast.ai library, not from scratch from scratch, you'll be in the top echelon of practitioners because you'll be able to do all of these things yourself and that's really really rare. And that'll put you in a great position to part 2.

Question: Where do you see the fast.ai library going in the future, say in five years?

Well, like I said, I don't make plans I just piss around so… Our only plan for fast.ai as an organization is to make deep learning accessible as a tool for normal people to use for normal stuff. So as long as we need to code, we failed at that because 99.8% of the world can't code. So the main goal would be to get to a point where it's not a library but it's a piece of software that doesn't require code. It certainly shouldn't require a goddamn lengthy hard working course like this one. So I want to get rid of the course, I want to get rid of the code, I want to make it so you can just do useful stuff quickly and easily. So that's maybe five years? Yeah, maybe longer.

All right. I hope to see you all back here for part 2. Thank you.