



vishal-burman Update Lesson1.md (#25)

f64234c on Mar 18

7 contributors



1003 lines (565 sloc) 75 KB

Raw

Blame

History



Lesson 1

[Webpage](#) / [Video](#) / [Lesson Forum](#) / [General Forum](#)

Welcome!

Make sure your GPU environment is set up and you can run Jupyter Notebook

[00_notebook_tutorial.ipynb](#)

Four shortcuts:

- `Shift + Enter`: Runs the code or markdown on a cell
- `Up Arrow + Down Arrow`: Toggle across cells
- `b`: Create new cell
- `ø + ø`: Restart Kernel

[2:45]

Jupyter Notebook is a really interesting device for data scientists because it lets you run interactive experiments and give you not just a static piece of information but something you can interactively experiment with.

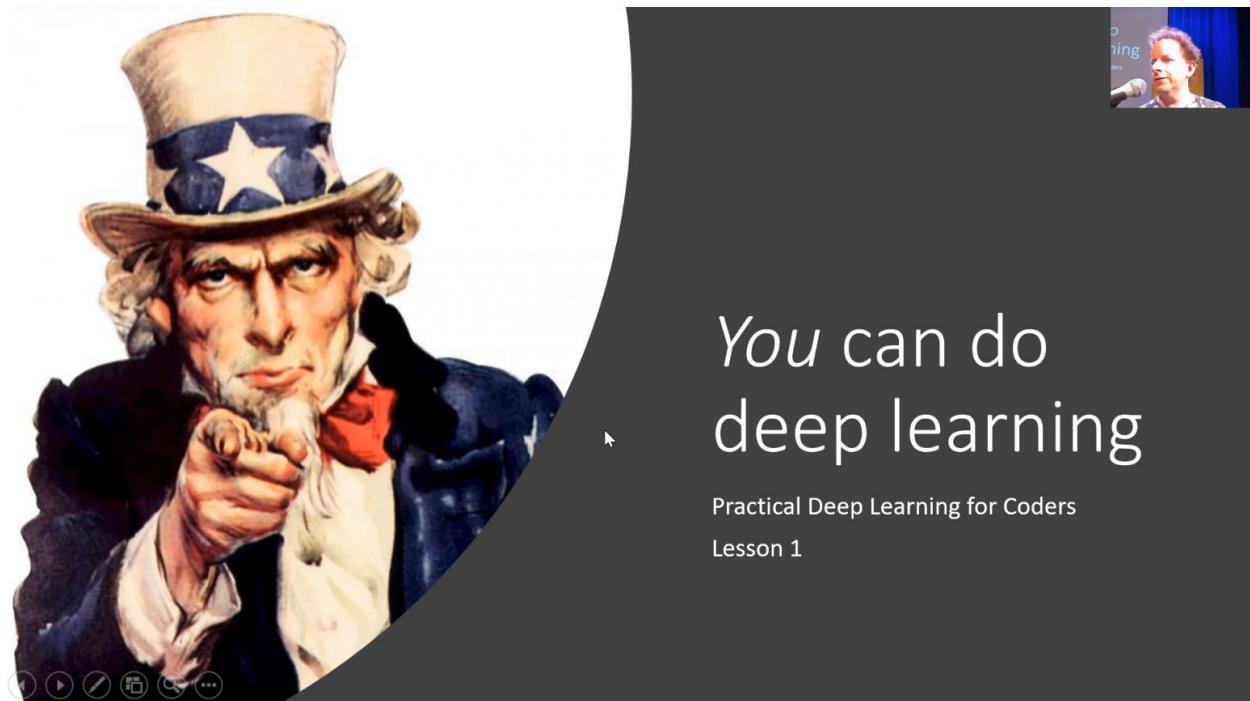
How to use notebooks and the materials well based on the last three years of experience:

1. Just watch a lesson end to end.
 - Don't try to follow along because it's not really designed to go the speed where you can follow along. It's designed to be something where you just take in the information, you get a general sense of all the pieces, how it all fits together.
 - Then you can go back and go through it more slowly pausing the video, trying

things out, making sure that you can do the things that I'm doing and you can try and extend them to do things in your own way.

- Don't try and stop and understand everything the first time.

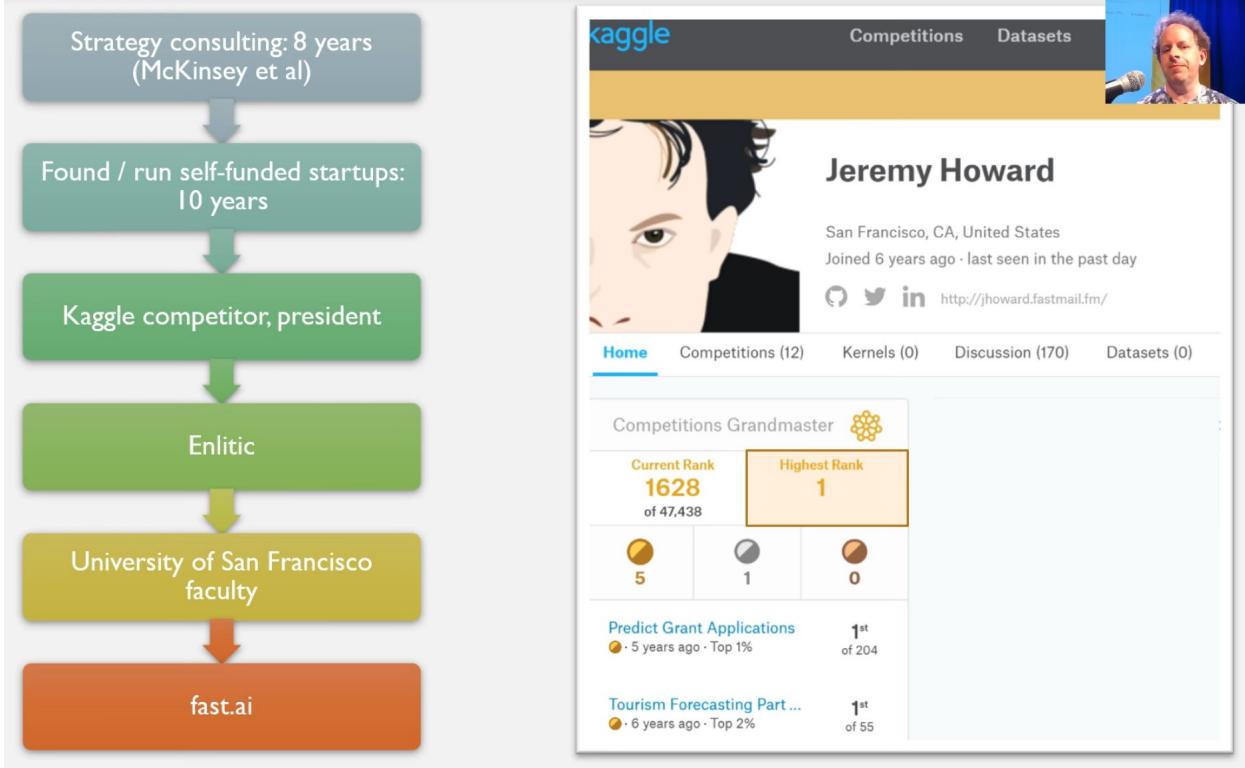
You can do world-class practitioner level deep learning [4:31]



Main places to be looking for things are:

- <http://course-v3.fast.ai/>
- <https://forums.fast.ai/>

A little bit about why we should listen to Jeremy [5:27]



Jeremy Howard

San Francisco, CA, United States
Joined 6 years ago · last seen in the past day

Home Competitions (12) Kernels (0) Discussion (170) Datasets (0)

Competitions Grandmaster

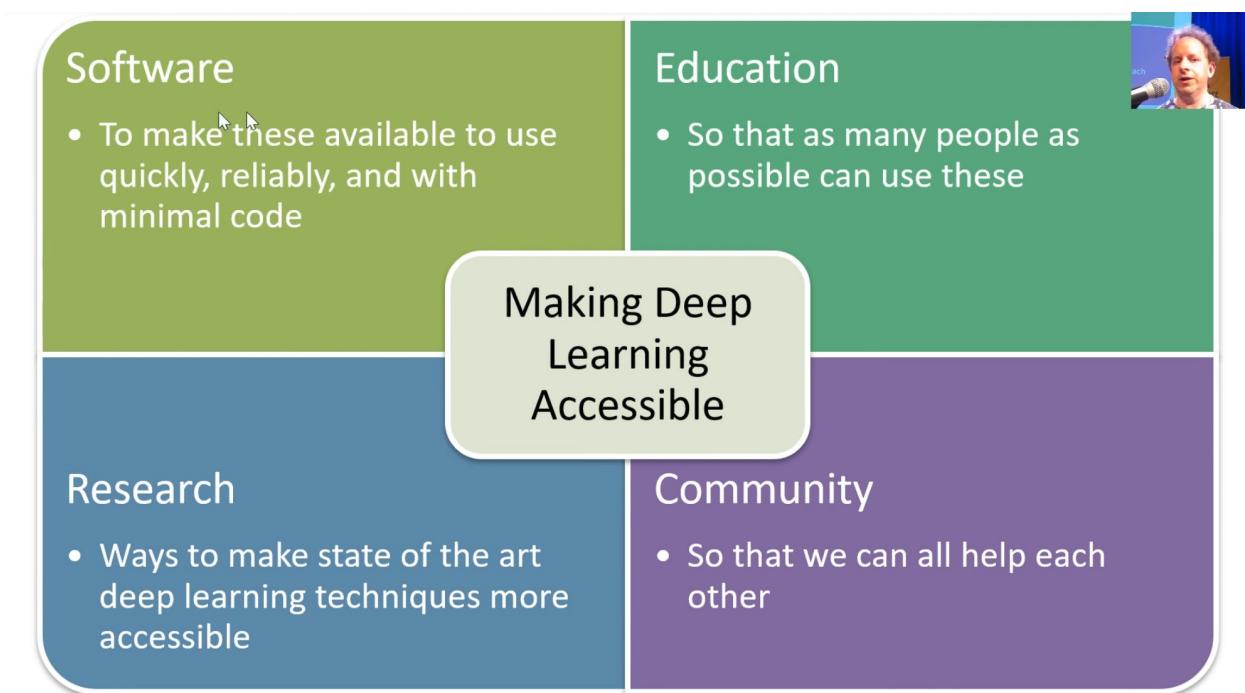
Current Rank **1628** of 47,438 Highest Rank **1**

5	1	0
---	---	---

Predict Grant Applications 5 years ago · Top 1% 1st of 204

Tourism Forecasting Part ... 6 years ago · Top 2% 1st of 55

Using machine learning to do useful things [6:48]



[7:26]

What can I do after 7 lessons?

Create a world class model to...



If you follow along with 10 hours a week or so approach for the 7 weeks, by the end, you will be able to:

1. Build an image classification model on pictures that you choose that will work at a world class level
2. Classify text using whatever datasets you're interested in
3. Make predictions of commercial applications like sales
4. Build recommendation systems such as the one used by Netflix

Not toy examples of any of these but actually things that can come top 10 in Kaggle competitions, that can beat everything that's in the academic community.

The prerequisite is one year of coding and high school math.

What people say about deep learning which are either pointless or untrue [9:05]



Black box

- Interpretable ML
- Visualize gradients and activations

Needs too much data

- Transfer learning
- Share pre-trained nets

Needs ML PhD

- No longer true
- fastai & keras libs, MOOCs, etc

Only for vision

- No longer true
- SoTA for speech, structured data, time series...

Needs lots of GPUs

- Was never true
- ...except for some research projects

“Not really AI”

- Who cares?
- Do you really want to build a brain?

- It's not a black box. It's really great for interpreting what's going on.
- It does not need much data for most practical applications.
- You don't need a PhD. Rachel has one so it doesn't actually stop you from doing deep learning if you have a PhD.
- It can be used very widely for lots of different applications, not just for vision.
- You don't need lots of hardware. 36 cents an hour server is more than enough to get world-class results for most problems.
- It is true that maybe this is not going to help you build a sentient brain, but that's not our focus. We are focused on solving interesting real-world problems.

[10:24]

Cricket vs. Baseball

with just 30 training images

```
In [25]: # 1. A few correct Labels at random
plot_val_with_title(rand_by_correct(True), "Correctly classified")
```

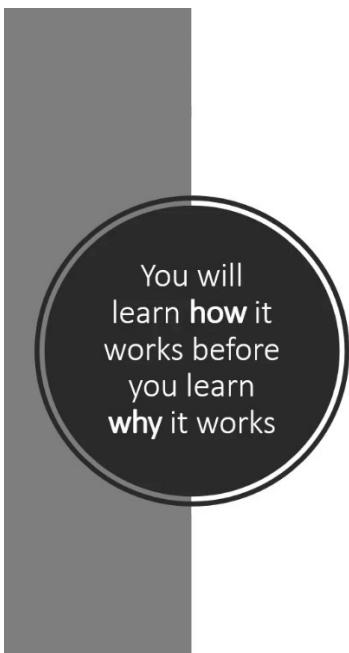
Correctly classified



Nikhil Balaji

Baseball vs. Cricket - An example by Nikhil of what you are going to be able to do by the end of lesson 1:

Topdown approach [11:02]



```
File Edit View Insert Cell Kernel Navigate Widgets Help
```

```
In [8]: learn = ConvLearner(data, tvm.resnet34, metrics=accuracy)
```

```
In [8]: learn.fit_one_cycle(5)
```

epoch	train loss	valid loss	accuracy
0	1.260069	0.331716	0.989774 (00:17)
1	0.533219	0.257838	0.917977 (00:16)
2	0.362374	0.233951	0.926863 (00:16)
3	0.245795	0.203109	0.937115 (00:16)
4	0.203346	0.204246	0.939166 (00:16)

```
In [9]: learn.save('stage-1')
```

```
In [32]: interp = ClassificationInterpretation.from_learner(learn)
interp.most_confused(min_val=3)
```

```
Out[32]: [('Ragdoll', 'Birman', 5),
           ('american_pit_bull_terrier', 'staffordshire_bull_terrier', 5),
           ('Egyptian_Mau', 'Bengal', 4),
           ('staffordshire_bull_terrier', 'american_pit_bull_terrier', 4)]
```

We are going to start by looking at code which is different to many of academic courses. We are going to learn to build a useful thing today. That means that at the end of today, you won't know all the theory. There will be lots of aspects of what we do that you don't know why or how it works. That's okay! You will learn why and how it works over the next 7 weeks. But for now, we've found that what works really well is to actually get your hands dirty coding - not focusing on theory.

What's your pet [12:26]

[lesson1-pets.ipynb](#)

`Shift + Enter` to run a cell

These three lines is what we start every notebook with:

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

These things starting `%` are special directives to Jupyter Notebook itself, they are not Python code. They are called "magics."

- If somebody changes underlying library code while I'm running this, please reload it automatically
- If somebody asks to plot something, then please plot it here in this Jupyter Notebook

The next two lines load up the fastai library:

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

What is fastai library? <http://docs.fast.ai/>

Everything we are going to do is going to be using either fastai or PyTorch which fastai sits on top of. PyTorch is fast growing extremely popular library. We use it because we used to use TensorFlow a couple years ago and we found we can do a lot more, a lot more quickly with PyTorch.

Currently fastai supports four applications:

1. Computer vision
2. Natural language text
3. Tabular data
4. Collaborative filtering

[15:45]

`import * - something you've all been told to never ever do.`

There are very good reasons to not use `import *` in standard production code with most libraries. But things like MATLAB is the opposite. Everything is there for you all the time. You don't even have to import things a lot of the time. It's kind of funny - we've got these two extremes of how do I code. The scientific programming community has one way, and then software engineering community has the other. Both have really good reasons for doing things.

With the fastai library, we actually support both approaches. In Jupyter Notebook where you want to be able to quickly interactively try stuff out, you don't want to constantly going back up to the top and importing more stuff. You want to be able to use lots of tab complete and be very experimental, so `import *` is great. When you are building stuff in production, you can do the normal PEP8 style proper software engineering practices. This is a different style of coding. It's not that there are no rules in data science programming, the rules are different. When you're training models, the most important thing is to be able to interactively experiment quickly. So you will see we use a lot of different processes, styles, and stuff to what you are used to. But they are there for a reason and you'll learn about them over time.

The other thing to mention is that the fastai library is designed in a very interesting modular way and when you do use `import *`, there's far less clobbering of things than you might expect. It's all explicitly designed to allow you to pull in things and use them quickly without having problems.

Looking at the data [17:56]

Two main places that we will be tending to get data from for the course:

1. Academic datasets

- Academic datasets are really important. They are really interesting. They are things where academics spend a lot of time curating and gathering a dataset so that they can show how well different kinds of approaches work with that data. The idea is they try to design datasets that are challenging in some way and require some kind of breakthrough to do them well.
- We are going to start with an academic dataset called the pet dataset.

2. Kaggle competition datasets

Both types of datasets are interesting for us particularly because they provide strong baseline. That is to say you want to know if you are doing a good job. So with Kaggle datasets that come from a competition, you can actually submit your results to Kaggle and see how well you would have gone in that competition. If you can get in about the top 10%, then I'd say you are doing pretty well.

Academic datasets, academics write down in papers what the state of the art is so how well did they go with using models on that dataset. So this is what we are going to do. We are going to try to create models that get right up towards the top of Kaggle competitions, preferably in the top 10, not just top 10% or that meet or exceed academic state-of-the-art published results. So when you use an academic dataset, it's important to cite it. You don't need to read that paper right now, but if you are interested in learning more about it and why it was created and how it was created, all the details are there.

Pet dataset is going to ask us to distinguish between 37 different categories of dog breed and cat breed. So that's really hard. In fact, every course until this one, we've used a different dataset which is one where you just have to decide if something is a dog or a cat. So you've got a 50-50 chance right away and dogs and cats look really different. Or else lots of dog breeds and cat breeds look pretty much the same.

So why have we changed the dataset? We've got to the point now where deep learning is so fast and so easy that the dogs versus cats problem which a few years ago was considered extremely difficult ~80% accuracy was the state of the art, it's now too easy. Our models were basically getting everything right all the time without any tuning and so there weren't really a lot of opportunities for me to show you how to do more sophisticated stuff. So we've picked a harder problem this year.

[20:51]

This kind of thing where you have to distinguish between similar categories is called fine grained classification in the academic context.

untar_data

The first thing we have to do is download and extract the data that we want. We're going to be using this function called `untar_data` which will download it automatically and untar it. AWS has been kind enough to give us lots of space and bandwidth for these datasets so they'll download super quickly for you.

```
path = untar_data(URLs.PETS); path
```

help

The first question then would be how do I know what `untar_data` does. You could just type `help` and you will find out what module it came from (since we did `import *` you don't necessarily know that), what it does, and something you might not have seen before even if you are an experienced programmer is what exactly you pass to it. You're probably used to seeing the names: `url`, `fname`, `dest`, but you might not be used to seeing `Union[pathlib.Path, str]`. These bits are types and if you're used to typed programming language, you would be used to seeing them, but Python programmers are less used to it. But if you think about it, you don't actually know how to use a function unless you know what type each thing is that you're providing it. So we make sure that we give you that type information directly here in the help.

In this case, `url` is a string, `fname` is either path or a string and defaults to nothing (`Union` means "either"). `dest` is either a string or a path and defaults to nothing.

```
help(untar_data)
```

```
Help on function untar_data in module fastai.datasets:
```

```
untar_data(url:str, fname:Union[pathlib.Path, str]=None,
dest:Union[pathlib.Path, str]=None)
    Download `url` if doesn't exist to `fname` and un-tgz to folder `dest`
```

We'll learn more shortly about how to get more documentation about the details of this, but for now, we can see we don't have to pass in a file name `fname` or a destination `dest`, it'll figure them out for us from the URL.

For all the datasets we'll be using in the course, we already have constants defined for all of them. So in this `URLs` class, you can see where it's going to grab it from.

`untar_data` will download that to some convenient path and untar it for us and it will then return the value of path.

```
path = untar_data(URLs.PETS); path  
  
PosixPath('/data1/jhoward/git/course-v3/nbs/dl1/data/oxford-iiit-pet')
```

In Jupyter Notebook, you can just write a variable on its own (semicolon is just an end of statement in Python) and it prints it. You can also say `print(path)` but again, we are trying to do everything fast and interactively, so just write it and here is the path where it's given us our data.

Next time you run this, since you've already downloaded it, it won't download it again. Since you've already untared it, it won't untar it again. So everything is designed to be pretty automatic and easy.

[23:50]

There are some things in Python that are less convenient for interactive use than they should be. For example, when you do have a path object, seeing what's in it actually takes a lot more typing than I would like. So sometimes we add functionality into existing Python stuff. One of the things we do is add a `ls()` method to path.

```
path.ls()
```

```
['annotations', 'images']
```

These are what's inside this path, so that's what we just downloaded.

Python 3 pathlib [24:25]

```
path_anno = path/'annotations'  
path_img = path/'images'
```

If you are an experienced Python programmer, you may not be familiar with this approach of using a slash like this. This is a really convenient function that's part of Python 3. It's functionality from [pathlib](#). Path object is much better to use than strings. They let you use basically create sub paths like this. It doesn't matter if you're on Windows, Linux, or Mac. It is always going to work exactly the same way. `path_img` is the path to the images in that dataset.

[24:57]

So if you are starting with a brand new dataset trying to do some deep learning on it. What do you do? Well, the first thing you would want to do is probably see what's in there. So we found that `annotations` and `images` are the directories in there, so what's in this `images`?

get_image_files [25:15]

`get_image_files` will just grab an array of all of the image files based on extension in a path.

```
fnames = get_image_files(path_img)
fnames[:5]
```

```
[PosixPath('/data1/jhoward/git/course-v3/nbs/dl1/data/oxford-iiit-pet/images
/american_bulldog_146.jpg'),
 PosixPath('/data1/jhoward/git/course-v3/nbs/dl1/data/oxford-iiit-pet/images
/german_shorthaired_137.jpg'),
 PosixPath('/data1/jhoward/git/course-v3/nbs/dl1/data/oxford-iiit-pet/images
/japanese_chin_139.jpg'),
 PosixPath('/data1/jhoward/git/course-v3/nbs/dl1/data/oxford-iiit-pet/images
/great_pyrenees_121.jpg'),
 PosixPath('/data1/jhoward/git/course-v3/nbs/dl1/data/oxford-iiit-pet/images
/Bombay_151.jpg')]
```

This is a pretty common way for computer vision datasets to get passed around - just one folder with a whole bunch of files in it. So the interesting bit then is how do we get the labels. In machine learning, the labels refer to the thing we are trying to predict. If we just eyeball this, we could immediately see that the labels are actually part of the file names. It's kind of like `path/label_number.extension`. We need to somehow get a list of `label` bits of each file name, and that will give us our labels. Because that's all you need to build a deep learning model:

- Pictures (files containing the images)
- Labels

In fastai, this is made really easy. There is an object called `ImageDataBunch`. An `ImageDataBunch` represents all of the data you need to build a model and there's some factory method which try to make it really easy for you to create that data bunch - a training set, a validation set with images and labels.

In this case, we need to extract the labels from the names. We are going to use `from_name_re`. `re` is the module in Python that does regular expressions - things that's really useful for extracting text.

Here is the regular expression that extract the label for this dataset:

```
np.random.seed(2)
pat = r'/(.+)\d+.jpg$'
```

With this factory method, we can basically say:

- path_img: a path containing images
- fnames: a list of file names
- pat: a regular expression (i.e. pattern) to be used to extract the label from the file name
- ds_tfm: we'll talk about transforms later
- size: what size images do you want to work with.

This might seem weird because images have size. This is a shortcoming of current deep learning technology which is that a GPU has to apply the exact same instruction to a whole bunch of things at the same time in order to be fast. If the images are different shapes and sizes, you can't do that. So we actually have to make all of the images the same shape and size. In part 1 of the course, we are always going to be making images square shapes. Part 2, we will learn how to use rectangles as well. It turns out to be surprisingly nuanced. But pretty much everybody in pretty much all computer vision modeling nearly all of it uses this approach of square. 224 by 224, for reasons we'll learn about, is an extremely common size that most models tend to use so if you just use size=224, you're probably going to get pretty good results most of the time. This is kind of the little bits of artisanship that I want to teach you which is what generally just works. So if you just use size 224, that'll generally just work for most things most of the time.

```
data = ImageDataBunch.from_name_re(path_img, fnames, pat, ds_tfms=get_transforms(
    data.normalize(imagenet_stats))
```

[29:16]

`ImageDataBunch.from_name_re` is going to return a DataBunch object. In fastai, everything you model with is going to be a DataBunch object. Basically DataBunch object contains 2 or 3 datasets - it contains your training data, validation data, and optionally test data. For each of those, it contains your images and your labels, your texts and your labels, or your tabular data and your labels, or so forth. And that all sits there in this one place(i.e. `data`).

Something we will learn more about in a little bit is normalization. But generally in nearly all machine learning tasks, you have to make all of your data about the same "size" - they are specifically about the same mean and standard deviation. So there is a normalize function that we can use to normalize our data bunch in that way.

[30:25]

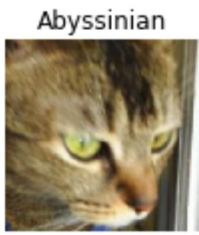
Question: What does the function do if the image size is not 224?

This is what we are going to learn about shortly. Basically this thing called transforms is used to do a number of the things and one of the things it does is to make something size 224.

data.show_batch

Let's take a look at a few pictures. Here are a few pictures of things from my data bunch. So you can see data.show_batch can be used to show me some of the contents in my data bunch. So you can see roughly what's happened is that they all seem to have been zoomed and cropped in a reasonably nice way. So basically what it'll do is something called by default center cropping which means it'll grab the middle bit and it'll also resize it. We'll talk more about the detail of this because it turns out to actually be quite important, but basically a combination of cropping and resizing is used.

```
data.show_batch(rows=3, figsize=(7,6))
```



Something else we are going to learn about is we also use this to do something called data augmentation. So there's actually some randomization in how much and where it crops and stuff like that.

Basic the basic idea is some cropping, resizing, and padding. So there's all kinds of different ways it depends on data augmentation which we are going to learn about shortly.

[31:51]

Question: What does it mean to normalize the images?

Normalizing the images, we're going to be learning more about later in the course, but in short, it means that the pixel values start out from naught to 255. And some channels might tend to be really bright, some might tend to be really not bright at all, some might vary a lot, and some might not very much at all. It really helps train a deep learning model if each one of those red green and blue channels has a mean of zero and a standard deviation of one.

If your data is not normalized, it can be quite difficult for your model to train well. So if you have trouble training a model, one thing to check is that you've normalized it.

[33:00] **Question:** As GPU mem will be in power of 2, doesn't size 256 sound more practical considering GPU utilization compared to 224?

The brief answer is that the models are designed so that the final layer is of size 7 by 7, so we actually want something where if you go 7 times 2 a bunch of times ($224 = 7 \times 2^5$), then you end up with something that's a good size.

[33:27]

We will get to all these details but the key thing is I wanted to get to training a model as quickly as possible.

It is important to look at the data

One of the most important things to be a really good practitioner is to be able to look at your data. So it's really important to remember to go to `data.show_batch` and take a look. It's surprising how often when you actually look at the dataset you've been given that you realize it's got weird black borders on it, some of the things have text covering up some of it, or some of it is rotated in odd ways. So make sure you take a look.

The other thing we want to do is to look at the labels. All of the possible label names are called your classes. With DataBunch, you can print out your `data.classes`.

```
print(data.classes)
len(data.classes), data.c
```

```
['american_bulldog', 'german_shorthaired', 'japanese_chin', 'great_pyrenees',
'Bombay', 'Bengal', 'keeshond', 'shiba_inu', 'Sphynx', 'boxer',
'english_cocker_spaniel', 'american_pit_bull_terrier', 'Birman',
'basset_hound', 'British_Shorthair', 'leonberger', 'Abyssinian',
'wheaten_terrier', 'scottish_terrier', 'Maine_Coon', 'saint_bernard',
'newfoundland', 'yorkshire_terrier', 'Persian', 'havanese', 'pug',
```

```
'miniature_pinscher', 'Russian_Blue', 'staffordshire_bull_terrifier', 'beagle',
'Siamese', 'samoyed', 'chihuahua', 'Egyptian_Mau', 'Ragdoll', 'pomeranian',
'english_setter']
```

(37, 37)

That's all of the possible labels that we found by using that regular expression on the file names. We learnt earlier on at the top that there are 37 possible categories, so just checking `len(data.classes)`, it is indeed 37. DataBunch will always have a property called `c`. We will get to the technical detail later, but for now, you can kind of think of it as being the number of classes. For things like regression problems and multi-label classification, that's not exactly accurate, but it'll do for now. It is important to know that `data.c` is a really important piece of information that is something like, or at least for classification problems it is, the number of classes.

Training [35:07]

Believe it or not, we are now ready to train a model. A model is trained in fastai using something called a "learner".

- **DataBunch:** A general fastai concept for your data, and from there, there are subclasses for particular applications like ImageDataBunch
- **Learner:** A general concept for things that can learn to fit a model. From that, there are various subclasses to make things easier in particular, there is a convnet learner (something that will create a convolutional neural network for you).

```
learn = create_cnn(data, models.resnet34, metrics=error_rate)
```

For now, just know that to create a learner for a convolutional neural network, you just have to tell it two things: `data` : What's your data. Not surprisingly, it takes a data bunch. `arch` : What's your architecture. There are lots of different ways of constructing a convolutional neural network.

For now, the most important thing for you to know is that there's a particular kind of model called ResNet which works extremely well nearly all the time. For a while, at least, you really only need to be doing choosing between two things which is what size ResNet do you want. There are ResNet34 and ResNet50. When we are getting started with something, I'll pick a smaller one because it'll train faster. That's as much as you need to know to be a pretty good practitioner about architecture for now which is that there are two variants of one architecture that work pretty well: ResNet34 and ResNet50. Start with a smaller one and see if it's good enough.

That is all the information we need to create a convolutional neural network learner.

There is one other thing I'm going to give it though which is a list of metrics. Metrics are literally just things that gets printed out as it's training. So I'm saying I would like you to print out error rate.

[37:25]

Training: resnet34

Now we will start training our model. We will use a [convolutional neural network](#) backbone and a fully connected head with a single hidden layer as a classifier. Don't know what these things mean? Not to worry, we will dive deeper in the coming lessons. For the moment you need to know that we are building a model which will take images as input and will output the predicted probability for each of the categories (in this case, it will have 37 outputs).

We will train for 5 epochs (5 cycles through all our data).

```
learn = ConvLearner(data, models.resnet34, metrics=error_rate)

Downloading: "https://download.pytorch.org/models/resnet34-333f7ec4.pth" to /home/hiromi.suenaga/.torch/models/resnet34-333f7ec4.pth
100%|██████████| 87306240/87306240 [00:09<00:00, 9590563.05it/s]

learn.fit_one_cycle(4)
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

The first time I run this on a newly installed box, it downloads the ResNet34 pre-trained weights. What that means is that this particular model has actually already been trained for a particular task. And that particular task is that it was trained on looking at about one and a half million pictures of all kinds of different things, a thousand categories of things, using an image dataset called ImageNet. So we can download those pre-trained weights so that we don't start with a model that knows nothing about anything, but we actually start with a model that knows how to recognize a thousand categories of things in ImageNet. I don't think all of these 37 categories of pet are in ImageNet but there were certainly some kinds of dog and some kinds of cat. So this pre-trained model knows quite a little bit about what pets look like, and it certainly knows quite a lot about what animals look like and what photos look like. So the idea is that we don't start with a model that knows nothing at all, but we start by downloading a model that knows something about recognizing images already. So it downloads for us automatically, the first time we use it, a pre-trained model and then from now on, it won't need to download it again - it'll just use the one we've got.

Transfer learning [38:54]

This is really important. We are going to learn a lot about this. It's kind of the focus of the whole course which is how to do this thing called "transfer learning." How to take a model that already knows how to do something pretty well and make it so that it can do your thing really well. We will take a pre-trained model, and then we fit it so that instead of predicting a thousand categories of ImageNet with ImageNet data, it predicts the 37 categories of pets using your pet data. By doing this, you can train models in 1/100 or less of the time of regular model training with 1/100 or less of the data of regular model training. Potentially, many thousands of times less. Remember I showed you the slide of Nikhil's lesson 1 project from last year? He used 30 images. There are not cricket and baseball images in ImageNet but it turns out that ImageNet is already so good at recognizing things in the world that just 30 examples of people playing baseball and cricket was enough to build a nearly perfect classifier.

Overfitting [40:05]

Wait a minute, how do you know it can actually recognize pictures of people playing cricket versus baseball in general? Maybe it just learnt to recognize those 30. Maybe it's just cheating. That's called "overfitting". We'll be talking a lot about that during this course. But overfitting is where you don't learn to recognize pictures of say cricket versus baseball, but just these particular cricketers in these particular photos and these particular baseball players in these particular photos. We have to make sure that we don't overfit. The way to do that is using something called a validation set. A validation set is a set of images that your model does not get to look at. So these metrics (e.g. `error_rate`) get printed out automatically using the validation set - a set of images that our model never got to see. When we created our data bunch, it automatically created a validation set for us. We'll learn lots of ways of creating and using validation sets, but because we're trying to bake in all of the best practices, we actually make it nearly impossible for you not to use a validation set. Because if you're not using a validation set, you don't know if you're overfitting. So we always print out the metrics on a validation, we've always hold it out, we always make sure that the model doesn't touch it. That's all done for you, and all built into this data bunch object.

Fitting your model [41:40]

So now we have a ConvLearner, we can fit it. You can just use a method called `fit` but in practice, you should nearly always use a method called `fit_one_cycle`. In short, one cycle learning is a [paper](#) that was released in April and turned out to be dramatically better both more accurate and faster than any previous approach. Again, I don't want to teach you how to do 2017 deep learning. In 2018, the best way to fit models is to use something called one cycle.

For now, just know that this number, 4, basically decides how many times do we go through the entire dataset, how many times do we show the dataset to the model so that it can learn from it. Each time it sees a picture, it's going to get a little bit better. But it's going to take time and it means it could overfit. If it sees the same picture too many times, it will just learn to recognize that picture, not pets in general. We'll learn all about how to tune this number during the next couple of lessons but starting out with 4 is a pretty good start just to see how it goes and you can actually see after four epochs or four cycles, we got an error rate of 6%. And it took 1 minute and 56 seconds.

```
learn.fit_one_cycle(4)
```

```
Total time: 01:10
epoch  train loss  valid loss  error_rate
1      1.175709   0.318438   0.099800   (00:18)
2      0.492309   0.229078   0.075183   (00:17)
3      0.336315   0.211106   0.067199   (00:17)
4      0.233666   0.191813   0.057219   (00:17)
```

So 94% of the time, we correctly picked the exact right one of those 37 dog and cat breeds which feels pretty good to me. But to get a sense of how good it is, maybe we should go back and look at the paper. Remember, I said the nice thing about using academic papers or Kaggle dataset is we can compare our solution to whatever the best people in Kaggle did or in the academics did. This particular dataset of pet breeds is from 2012 and if I scroll through the paper, you'll generally find in any academic paper there'll be a section called experiments about 2/3 of the way through. If you find a section on experiments, then you can find a section on accuracy and they've got lots of different models and their models. The models as you'll read about in the paper, it's really pet specific. They learn something about how pet heads look and how pet bodies look, and pet image in general look. And they combine them all together and once they use all of this complex code and math, they got an accuracy of 59%. So in 2012, this highly pet specific analysis got an accuracy of 59%. These were the top researchers from Oxford University. Today in 2018, with basically about three lines of code, we got 94% (i.e. 6% error). So that gives you a sense of how far we've come with deep learning, and particularly with PyTorch and fastai, how easy things are.

[46:43] We just trained a model. We don't know exactly what that involved or how it happened but we do know that with 3 or 4 lines of code, we've built something which smashed the accuracy of the state-of-the-art of 2012. 6% error certainly sounds like pretty impressive for something that can recognize different dog breeds and cat breeds, but we don't really know why it work, but we will. That's okay.

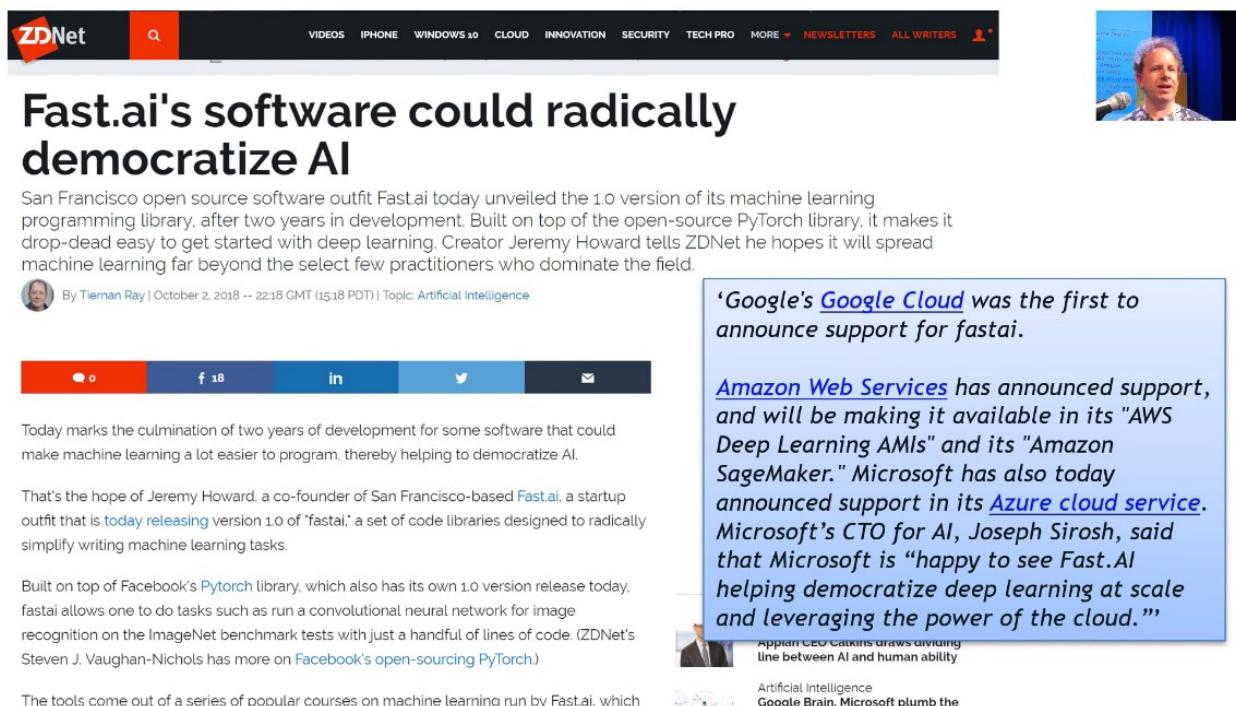
The number one regret of past students:

"I personally fell into the habit of watching the lectures too much and googling definitions / concepts / etc too much, without running the code. At first I thought that I should read the code quickly and then spend time researching the theory behind it..."

"In retrospect, I should have spent the majority of my time on the actual code in the notebooks instead, in terms of running it and seeing that goes into it and what comes out of it"

So please run the code. Really run the code. [47:54]

Your most important skills to practice are learning and understanding what goes in and what comes out.



The screenshot shows a ZDNet article titled "Fast.ai's software could radically democratize AI". The article is by Tieman Ray and was published on October 2, 2018. It discusses the release of version 1.0 of the Fast.ai machine learning library, which is built on PyTorch. The library is designed to make deep learning easier to program. The article includes a quote from Jeremy Howard, co-founder of Fast.ai, and mentions support from Google, Amazon, Microsoft, and others. There are social sharing buttons and a sidebar with related news items.

Fast.ai's software could radically democratize AI

San Francisco open source software outfit Fast.ai today unveiled the 1.0 version of its machine learning programming library, after two years in development. Built on top of the open-source PyTorch library, it makes it drop-dead easy to get started with deep learning. Creator Jeremy Howard tells ZDNet he hopes it will spread machine learning far beyond the select few practitioners who dominate the field.

By Tieman Ray | October 2, 2018 -- 22:18 GMT (15:18 PDT) | Topic: Artificial Intelligence

Today marks the culmination of two years of development for some software that could make machine learning a lot easier to program, thereby helping to democratize AI.

That's the hope of Jeremy Howard, a co-founder of San Francisco-based [Fast.ai](#), a startup outfit that is [today releasing](#) version 1.0 of "fastai," a set of code libraries designed to radically simplify writing machine learning tasks.

Built on top of Facebook's [Pytorch](#) library, which also has its own 1.0 version release today, fastai allows one to do tasks such as run a convolutional neural network for image recognition on the ImageNet benchmark tests with just a handful of lines of code. (ZDNet's Steven J. Vaughan-Nichols has more on [Facebook's open-sourcing PyTorch](#).)

The tools come out of a series of popular courses on machine learning run by Fast.ai, which

'Google's Google Cloud was the first to announce support for fastai.'

Amazon Web Services has announced support, and will be making it available in its "AWS Deep Learning AMIs" and its "Amazon SageMaker." Microsoft has also today announced support in its [Azure cloud service](#). Microsoft's CTO for AI, Joseph Sirosh, said that Microsoft is "happy to see Fast.AI helping democratize deep learning at scale and leveraging the power of the cloud."

Fastai library is pretty new, but it's getting an extraordinary amount of traction. It's making a lot of things a lot easier, but it's also making new things possible. So really understanding the fastai software is something which is going to take you a long way. And the best way to really understand the fastai software well is by using the [fastai documentation](#).

Keras[49:25]



Dogs vs. Cats

fastai v1 for PyTorch: Fast, accurate, easier deep learning
Written: 02 Oct 2018 by Jeremy Howard



	fastai resnet34*	fastai resnet50	Keras
Lines of code (excluding imports)	5	5	31
Stage 1 error	0.70%	0.65%	2.05%
Stage 2 error	0.50%	0.50%	0.80%
Test time augmentation (TTA) error	0.30%	0.40%	N/A*
Stage 1 time	4:56	9:30	8:30
Stage 2 time	6:44	12:48	17:38

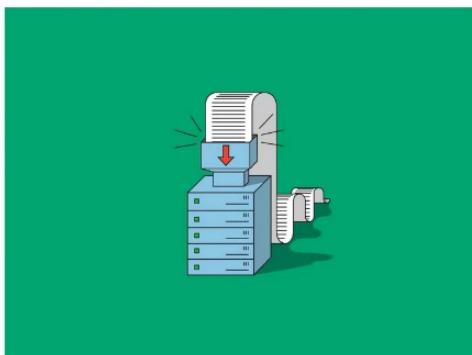
* Keras does not provide resnet 34 or TTA

So how does it compare? There's only one major other piece of software like fastai that tries to make deep learning easy to use and that's Keras. Keras is a really terrific piece of software, we actually used it for the previous courses until we switch to fastai. It runs on top of Tensorflow. It was the gold standard for making deep learning easy to use before. But life is much easier with fastai. So if you look at the last year's course exercise which is getting dogs vs. cats, fastai lets you get much more accurate (less than half the error on a validation set), training time is less than half the time, lines of code is about 1/6. The lines of code are more important than you might realize because those 31 lines of Keras code involved you making a lot of decisions, setting lots of parameters, doing lots of configuration. So that's all stuff where you have to know how to set those things to get best practice results. Or else, those 5 lines of code, any time we know what to do for you, we do it for you. Anytime we can pick a good default, we pick it for you. So hopefully you will find this a really useful library, not just for learning deep learning but for taking it a very long way.

[50:53]

GREGORY BARBER BUSINESS 08.07.18 01:59 PM

AI CAN RECOGNIZE IMAGES. BUT CAN IT UNDERSTAND THIS HEADLINE?



© CASEY CHIN

SHARE

IN 2012, ARTIFICIAL intelligence researchers revealed a big improvement in computers' ability to recognize images by feeding a neural network millions of labeled images from a database called ImageNet. It ushered in an exciting phase for computer vision, as it became clear that a model trained using ImageNet could help tackle all sorts of image-recognition



'...recent research from [fast.ai](#), [OpenAI](#), and the [Allen Institute for AI](#) suggests a potential breakthrough, with more robust language models that can help researchers tackle a range of unsolved problems. Sebastian Ruder, a researcher behind one of the new models, calls it his field's "ImageNet moment."'

How far can you take it? All of the research that we do at fastai uses the library and an example of the research we did which was recently featured in Wired describes a new breakthrough in a natural language processing which people are calling the ImageNet moment which is basically we broke a new state-of-the-art result in text classification which OpenAI then built on top of our paper with more computing, more data to do different tasks to take it even further. This is an example of something we've done in the last 6 months in conjunction with my colleague Sebastian Ruder - an example of something that's being built in the fastai library and you are going to learn how to use this brand new model in three lessons time. You're actually going to get this exact result from this exact paper yourself.

[51:50]



GitHub Engineering

Towards Natural Language Semantic Code Search

hamelsmu hohsiangwu September 18, 2018

"The semantic code search demo is only the tip of the iceberg, as folks in sales, marketing, fraud are currently leveraging the power of fastai to bring transformative change to their business areas."

-- Github Senior ML Scientist Hamel Husain

Another example, one of our alumni, Hamel Husain built a new system for natural language semantic code search, you can find it on Github where you can actually type in English sentences and find snippets of code that do the thing you asked for. Again, it's being built with the fastai library using the techniques you'll learn in the next seven weeks.

[52:27]

The best place to learn about these things and get involved in these things is on the forums where as well as categories for each part of the course and there is also a general category for deep learning where people talk about research papers applications.

Even though today, we are focusing on a small number of lines of code to a particular thing which is image classification and we are not learning much math or theory, over these seven weeks and then part two, we are going to go deeper and deeper.

Where can that take you? [53:05]



This is Sarah Hooker. She did our first course a couple of years ago. She started learning to code two years before she took our course. She started a nonprofit called Delta Analytics, they helped build this amazing system where they attached old mobile phones to trees in Kanyan rain forests and used it to listen for chainsaw noises, and then they used deep learning to figure out when there was a chainsaw being used and then they had a system setup to alert rangers to go out and stop illegal deforestation in the rainforests. That was something she was doing while she was in the course as part of her class projects.



Evaluating Feature Importance Estimates

Sara Hooker^{1,2} Dumitru Erhan¹ Pieter-Jan Kindermans^{1,2} Been Kim¹

Abstract

ating the influence of a given feature to a prediction is challenging. We introduce **R, RemOve And Retrain**, a benchmark to test the accuracy of interpretability methods estimate input feature importance in deep networks. We remove a fraction of input features deemed to be most important according to the estimator and measure the change to the accuracy upon retraining. The most accurate estimator will identify inputs whose removal causes the most damage to model performance relative to all other estimators. This notion produces thought-provoking results – that several estimators are less accurate than random assignment of feature importance, even averaging a set of squared noisy estimators (a variant of a technique pronounced by

(DNN) is particularly challenging because there is typically a high number of input features and we are unable to concisely determine the representation learnt by the model. For example, the model input for a computer vision task is often a digital image and an input feature can be defined as a pixel or a group of connected pixels. Thus, a single image in the dataset can easily be associated with a quarter of a million input features. The aim of feature importance estimators is to quantify the importance of each of these input features to the model prediction for that image. The set of estimates for all pixels in the image is often treated as a sorted ranking of importance, or they are visualized as a natural image “heatmap” to assist humans in understanding parts of the image that the model pays attention to.

Despite the challenges involved, a substantial body of research on input importance estimation and numerous estimators have been proposed (Bach et al., 2010; Bach et al., 2015; Zintereit et al., 2017; Selvaraju et al., 2017;

Sara Hooker

Algorithms and Theory Machine Intelligence
Machine Perception

ent at Google Brain. Her research interests include compression and security in deep neural networks. She built an open source machine learning for social good. In 2014, Sara founded Delta Analytics, a non-profit technical capacity to help communities across the world.

She is now a Google Brain researcher, publishing some papers, and now she is going to Africa to set up a Google Brain's first deep learning research center in Africa. She worked her arse off. She really really invested in this course. Not just doing all of the assignments but also going out and reading Ian Goodfellow's book, and doing lots of other things. It really shows where somebody who has no computer science or math background at all can be now one of the world's top deep learning researchers and doing very valuable work.

[54:49]

Deep Learning Explorations

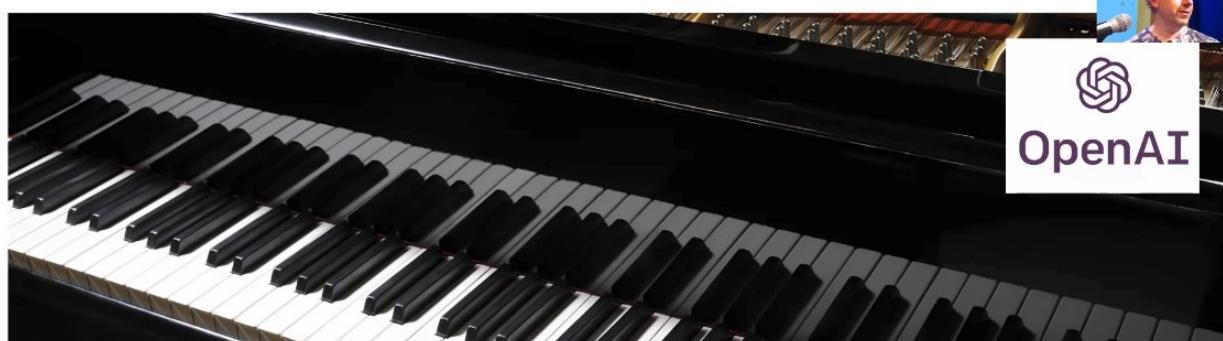
BLOG

LEARNING ABOUT DEEP LEARNING

NEURAL GALLERY

EDUCATIONAL APPS

CHAR



CLARA: A NEURAL NET MUSIC GENERATOR

By mcleavey Posted August 29, 2018 In Highlights, Music

3

Christine McLeavey Payne



Another example from our most recent course, Christine Payne. She is now at OpenAI and you can find [her post](#) and actually listen to her music samples of something she built to automatically create chamber music compositions.



Vive la Musique Francaise!

BY JANOS GERESEN, May 27, 2014

French classical music is not an endangered genre in the Bay Area: San Francisco Symphony alone has just concluded subscription programs featuring Debussy's *Images* and coming up this week, Charles Dutoit conducts Poulenc's *Gloria* and Fauré's *Requiem*.

Still, you can never have enough of a good thing, so consider the June 20-21 **First Festival of French Classical Music**, presented by Alliance Française of Silicon Valley.

To be held in the Mountain View Community School of Music and Art's Tateuchi Recital Hall, the concerts present 28 Bay Area musicians in performance of music by Maurice Ravel, Ernest Chausson, Gabriel Fauré, André Jolivet, Darius Milhaud, François Poulenc, and Charles-Valentin Alkan; also songs made famous by Edith Piaf.



"Une soirée parisienne" performers:
Roman Fukshansky, clarinet; Moni Simeonov, violin; Christine McLeavey Payne, piano; and Jonah Kim, cello

DANCE JAZZ OPERA

Enter Email
GET THE SFCV WEEKLY

FIND EVENTS

All Dates
All Region
All Types

Add your event here



She is a classical pianist. Now I will say she is not your average classical pianist. She's a classical pianist who also has a master's in medical research in Stanford, and studied neuroscience, and was a high-performance computing expert at DE Shaw, Co-Valedictorian at Princeton. Anyway. Very annoying person, good at everything she does. But I think it's really cool to see how a domain expert of playing piano can go through the fastai course and come out the other end as OpenAI fellow.

Interestingly, one of our other alumni of the course recently interviewed her for a blog post series he is doing on top AI researchers and she said one of the most important pieces of advice she got was from me and she said the advice was:

Pick one project. Do it really well. Make it fantastic. [56:20](#)

We're going to be talking a lot about you doing projects and making them fantastic during this course.

[[56:36](#)] Having said that, I don't really want you to go to AI or Google Brain. What I really want you to do is to go back to your workplace or your passion project and apply these skills there.

Josh Gordon @random_forests · Feb 6
MIT has shared an Intro to Deep Learning course, see: introtodeeplearning.com.
Labs include @TensorFlow code (haven't had a chance to go through them yet, but look pretty cool! I'm a big fan of medical imaging). Videos are uploading now - goo.gl/7FzMBn.

Alexandre Cadrian @alexandrecadrian

Replies to [@random_forests](#) @TensorFlow

Pneumothorax activation heatmaps overlapping both diaphragms on an **upright** chest film ?! Very unlikely. The activations should be apical in the lungs. If there is no citation error, this is almost a proof of model overfitting.

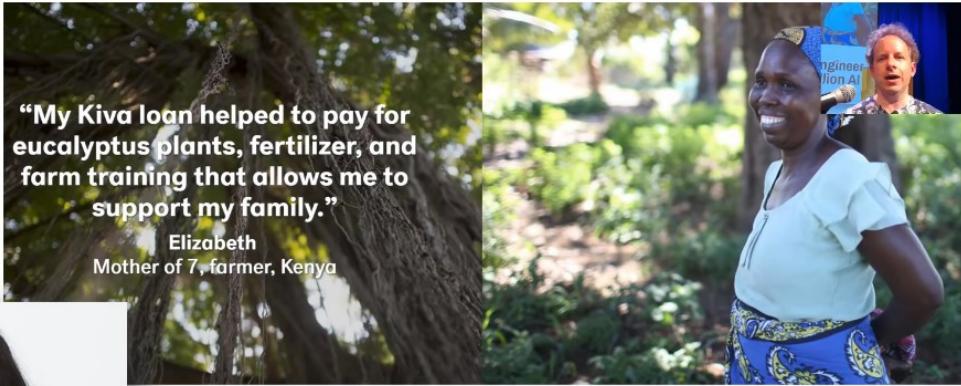
7:45 PM - 8 Feb 2018

3 Retweets 27 Likes

MIT released a deep learning course and they highlighted in their announcement this medical imaging example. One of our students Alex who is a radiologist said you guys just showed a model overfitting. I can tell because I am a radiologist and this is not what this would look like on a chest film. This is what it should look like and as a deep learning practitioner, this is how I know this is what happened in your model. So Alex is combining his knowledge of radiology and his knowledge of deep learning to assess MIT's model from just two images very accurately. So this is actually what I want most of you to be doing is to take your domain expertise and combine it with the deep learning practical aspects you'll learn in this course and bring them together like Alex is doing here. So a lot of radiologists have actually gone through this course now and have built journal clubs and American Council of Radiology practice groups. There's a data science institute at the ACR now and Alex is one of the people who is providing a lot of leadership in this area. And I would love you to do the same kind of thing that Alex is doing which is to really bring deep learning leadership into your industry and to your social impact project, whatever it is that you are trying to do.

[58:22]

Melissa
Fabros



CrowdFlower Names Kiva Engineer First Round Winner of \$1 Million AI For Everyone Challenge

Another great example. This is Melissa Fabros who is a English literature PhD who studied gendered language in English literature or something and actually Rachel at the previous job taught her to code. Then she came to the fastai course. She helped Kiva, a micro lending a social impact organization, to build a system that can recognize faces. Why is that necessary? We're going to be talking a lot about this but because most AI researchers are white men, most computer vision software can only recognize white male faces effectively. In fact, I think it was IBM system was like 99.8% accurate on common white face men versus 65% accurate on dark skinned women. So it's like 30 or 40 times worse for black women versus white men. This is really important because for Kiva, black women perhaps are the most common user base for their micro lending platform. So Melissa after taking our course, again working her arse off, and being super intense in her study and her work won this \$1,000,000 AI challenge for her work for Kiva.

[59:53]



empowering visually impaired to live more independently.

Envision is a tool that uses artificial intelligence to make visual information accessible to visually impaired. With Envision, visually impaired users can shop in supermarkets, use public transport, read menu cards in restaurants, recognise their friends, find their belongings and so much more, all on their own.

[Download for iOS](#)[Sign up for Android](#)

Karthik did our course and realized that the thing he wanted to do wasn't at his company. It was something else which is to help blind people to understand the world around them. So he started a new startup called envision. You can download the app and point your phone to things and it will tell you what it sees. I actually talked to a blind lady about these kinds of apps the other day and she confirmed to me this is a super useful thing for visually disabled users.

[1:00:24]

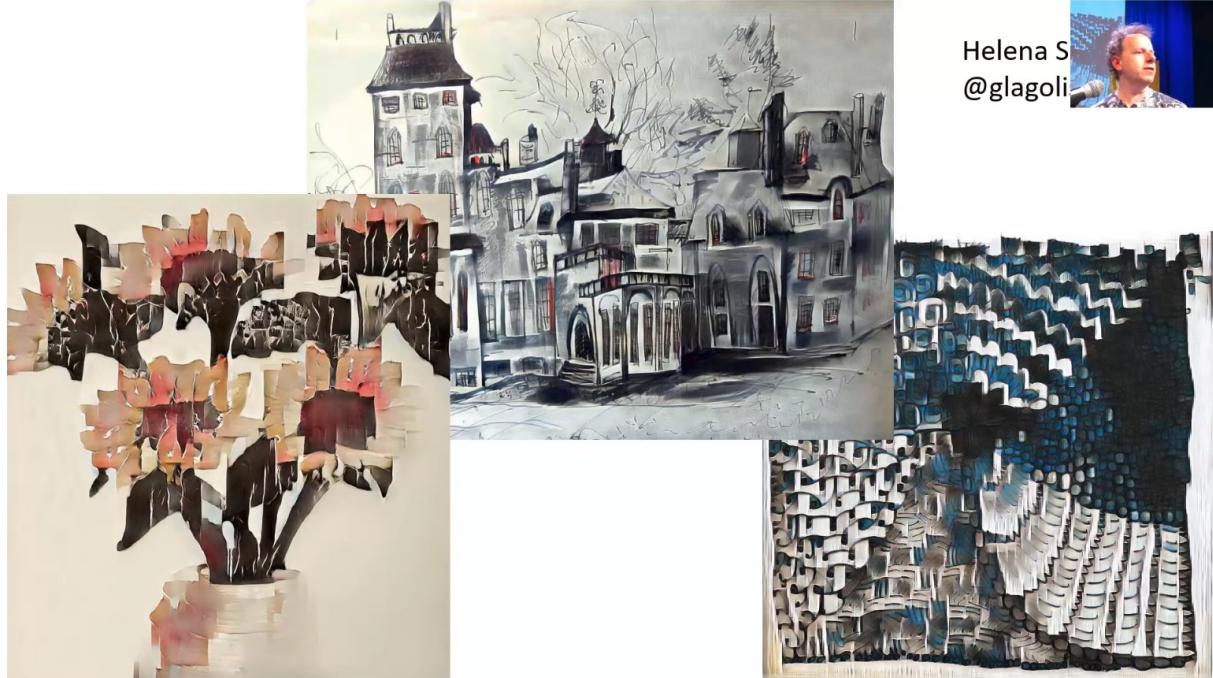
A screenshot of the MIT Technology Review website. The top navigation bar includes links for 'Log in / Create an account', 'Search', and 'Subscribe'. Below the header, there's a large image of a dense collection of small, colorful images, described as a 'snapshot of the ImageNet database' by Andrew Ng. The main article title is 'A small team of student AI coders beats Google's machine-learning code' by Will Knight, dated August 10, 2018. The text discusses how students from Fast.ai beat Google's AI code.

“The fast.ai algorithm was trained on the ImageNet database in 18 minutes using 16 Amazon Web Service instances, at a total compute cost of around \$40.”



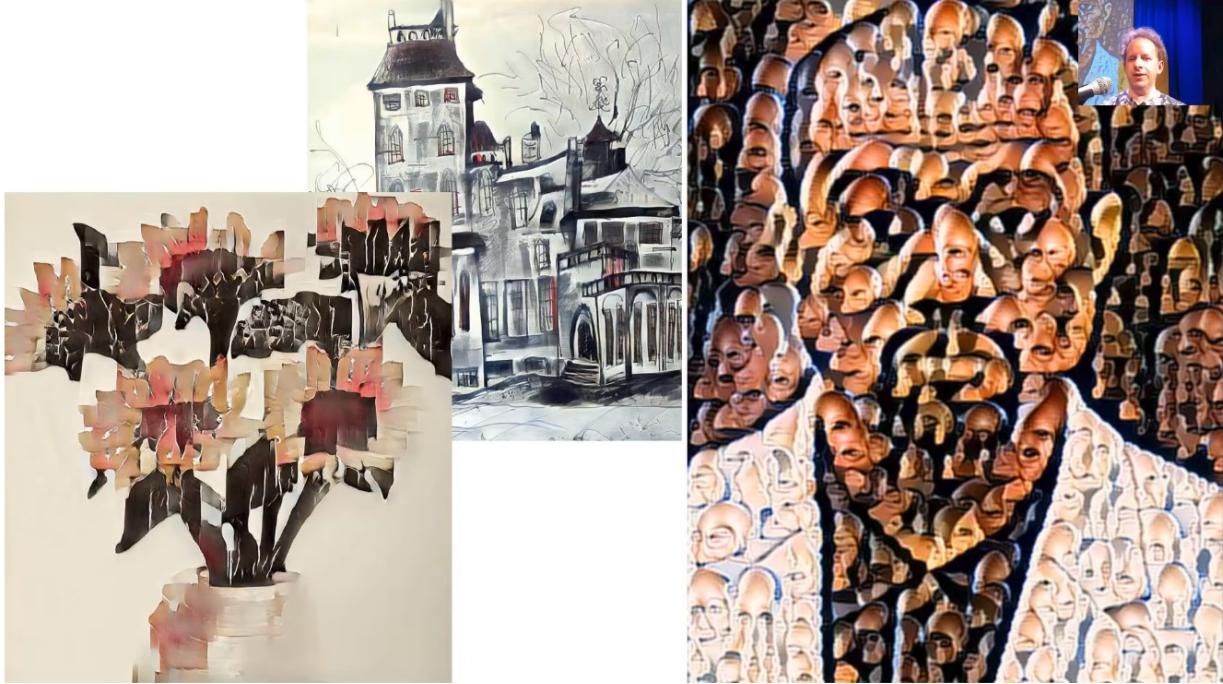
The level that you can get to, with the content that you're going to get over these seven weeks and with this software can get you right to the cutting edge in areas you might find surprising. I helped a team of some of our students and some collaborators on actually breaking the world record for how quickly you can train ImageNet. We used standard AWS cloud infrastructure, cost of \$40 of compute to train this model using fastai library, the technique you learn in this course. So it can really take you a long way. So don't be put off by this what might seem pretty simple at first. We are going deeper and deeper.

[1:01:17]



Helena S
@glagolista

You can also use it for other kinds of passion project. Helena Sarin - you should definitely check out her Twitter account [@glagolista](#). This art is basically a new style of art that she's developed which combines her painting and drawing with generative adversarial models to create these extraordinary results. I think this is super cool. She is not a professional artist, she is a professional software developer but she keeps on producing these beautiful results. When she started, her art had not really been shown or discussed anywhere, now there's recently been some quite high profile article describing how she is creating a new form of art.



Equally important, Brad Kenstler who figured out how to make a picture of Kanye out of pictures of Patrick Stewart's head. Also something you will learn to do if you wish to. This particular type of what's called "style transfer" - it's a really interesting tweak that allowed him to do something that hadn't quite been done before. This particular picture helped him to get a job as a deep learning specialist at AWS.

[1:02:41]

Another alumni actually worked at Splunk as a software engineer and he designed an algorithm which basically turned Splunk to be fantastically good at identifying fraud and we'll talk more about it shortly.

The image consists of two side-by-side screenshots. On the left is a presentation slide from Splunk. The title reads "splunk> SECURITY" and "Splunk and Tensorflow for Security: Catching the Fraudster with Behavior Biometrics". On the right is a promotional image for a hotdog. It features a green header with the text "Hotdog!" and a small video player showing a man speaking. Below the header is a circular image of a hotdog in a bun. The main image shows a single hotdog in a white paper tray on a wooden surface. At the bottom is a blue "Share" button and a "No Thanks" link.

If you've seen Silicon Valley, the HBO series, the hotdog Not Hotdog app - that's actually a real app you can download and it was built by Tim Anglade as a fastai student project. So there's a lot of cool stuff that you can do. It was Emmy nominated. We only have one Emmy nominated fastai alumni at this stage, so please help change that.

The image shows two side-by-side screenshots of forum threads from the fast.ai community. The left thread, titled 'Language Model Zoo', has a post by 'lesscomfortable' (Francisco Ingham) discussing the ULMFiT implementations in different languages. The right thread, titled 'ULMFiT - German', has a post by 'piotr.czaplak' showing results for German language models. Both threads include tables comparing various experiments across metrics like LM Perplexity and Micro F1 on GermEval2017 task 1 timestamp 1 and 2.

[1:03:30]

The other thing, the forum thread can turn into these really cool things. So Francisco was a really boring McKinsey consultant like me. So Francisco and I both have this shameful past that we were McKinsey consultants, but we left and we're okay now. He started this thread saying like this stuff we've just been learning about building NLP in different languages, let's try and do lots of different languages, and he started this thing called the language model zoo and out of that, there's now been an academic competition won in Polish that led to an academic paper, Thai state of the art, German state of the art, basically as students have been coming up with new state of the art results across lots of different languages and this all is entirely done by students working together through the forum.

So please get on the forum. But don't be intimidated because everybody you see on the forum, the vast majority of posting post all the darn time. They've been doing this a lot and they do it a lot of the time. So at first, it can feel intimidating because it can feel like you're the only new person there. But you're not. You're all new people, so when you just get out there and say like "okay all you people getting these state of the art results in German language modeling, I can't start my server, I try to click the notebook and I get an error, what do I do?" People will help you. Just make sure you provide all the information ([how to ask for help](#)).

Or if you've got something to add! If people are talking about crop yield analysis and you're a farmer and you think oh I've got something to add, please mention it even if you are not sure it's exactly relevant. It's fine. Just get involved. Because remember, everybody else in the forum started out also intimidated. We all start out not knowing things. So just get out there and try it!

[1:05:59] Question: Why are we using ResNet as opposed to Inception?

There are lots of architectures to choose from and it would be fair to say there isn't one best one but if you look at things like the Stanford DAWN Bench benchmark of image classification, you'll see in first place, second place, third place, and fourth place all use ResNet. ResNet is good enough, so it's fine.

DAWN Bench About Submit Stanford DAWN

Image Classification on ImageNet

Training Time ⏱

All Submissions

Objective: Time taken to train an image classification model to a top-5 validation accuracy of 93% or greater on ImageNet.

Rank	Time to 93% Accuracy	Model	Hardware	Framework
1 Sep 2018	0:18:06	ResNet-50 <i>fast.ai/DIUx (Yaroslav Bulatov, Andrew Shaw, Jeremy Howard)</i> source	16 p3.16xlarge (AWS)	PyTorch 0.4.1
2 Sep 2018	0:18:53	Resnet 50 <i>Andrew Shaw, Yaroslav Bulatov, Jeremy Howard</i> source	64 * V100 (8 machines - AWS p3.16xlarge)	ncluster / Pytorch 0.5.0a0+0e8088d
3 Sep 2018	0:29:43	Resnet 50 <i>Andrew Shaw, Yaroslav Bulatov, Jeremy Howard</i> source	32 * V100 (4 machines - AWS p3.16xlarge)	ncluster / Pytorch 0.5.0a0+0e8088d
4 Apr 2018	0:30:43	ResNet50 <i>Google</i> source	Half of a TPUv2 Pod	TensorFlow 1.8.0-rc1
5 Apr 2018	1:06:32	AmoebaNet-D N6F256 <i>Google</i> source	1/4 of a TPUv2 Pod	TensorFlow 1.8.0-rc1

The main reason you might want a different architecture is if you want to do edge computing, so if you want to create a model that's going to sit on somebody's mobile phone. Having said that, even there, most of the time, I reckon the best way to get a model onto somebody's mobile phone is to run it on your server and then have your mobile phone app talk to it. It really makes life a lot easier and you get a lot more flexibility. But if you really do need to run something on a low powered device, then there are special architectures for that. So the particular question was about Inception. That's a particular another architecture which tends to be pretty memory intensive but it's okay. It's not terribly resilient. One of the things we try to show you is stuff which just tends to always work even if you don't quite tune everything perfectly. So ResNet tends to work pretty well across a wide range of different kind of details around choices that you might make. So I think it's pretty good.

[1:07:58]

We've got this trained model and what's actually happened as we'll learn is it's basically creating a set of weights. If you've ever done anything like a linear regression or logistic regression, you'll be familiar with coefficients. We basically found some coefficients and parameters that work pretty well and it took us a minute and 56 seconds. So if we want to start doing some more playing around and come back later, we probably should save those weights. You can just go `learn.save` and give it a name. It's going to put it in a model subdirectory in the same place the data came from, so if you save different models or different data bunches from different datasets, they'll all be kept separate. So don't worry about it.

```
learn.save('stage-1')
```

Results [1:08:54]

To see what comes out, we could use this class for class interpretation. We are going to use this factory method from learner, so we pass in a learn object. Remember a learn object knows two things:

1. What's your data
2. What is your model. Now it's not just an architecture, it's actually a trained model

That's all the information we need to interpret that model.

```
interp = ClassificationInterpretation.from_learner(learn)
```

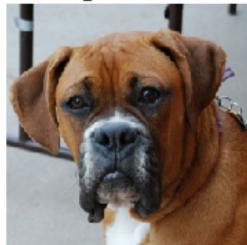
One of the things, perhaps the most useful things to do is called `plot_top_losses`. We are going to be learning a lot about this idea of loss functions shortly but in short, a loss function is something that tells you how good was your prediction. Specifically that means if you predicted one class of cat with great confidence, but actually you were wrong, then that's going to have a high loss because you were very confident about the wrong answer. So that's what it basically means to have high loss. By plotting the top losses, we are going to find out what were the things that we were the most wrong on, or the most confident about what we got wrong.

```
interp.plot_top_losses(9, figsize=(15,11))
```

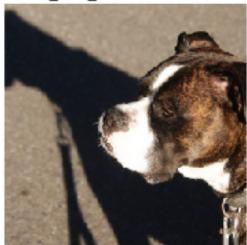
chihuahua/saint bernard / 9.48 / 0.23



boxer/saint bernard / 8.63 / 0.92



staffordshire bull terrier/boxer / 5.82 / 0.99



american_pit_bull_terrier/boxer / 5.63 / 0.91



Ragdoll/Maine_Coon / 5.50 / 0.31



Ragdoll/Maine_Coon / 4.79 / 0.92

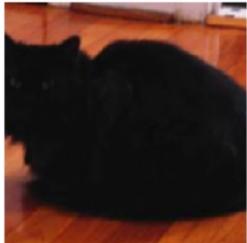


`american_bulldog/staffordshire_bull_terrier / 4.77 staffordshire_bull_terrier/american_bulldog / 4.50 / 0.99`



A medium-sized dog with a brown and white coat is standing on a bed of brown mulch. The dog is facing the camera, with its front paws on the ground and its back legs slightly bent. It has a white patch on its chest and white markings on its paws. The background shows a grassy area and some trees.

Bombay/Persian / 4.36 / 0.81



It prints out four things. What do they mean? Perhaps we should look at the document.

We have already seen `help`, and `help` just prints out a quick little summary. But if you want to really see how to do something use `doc`.

```
In [19]: doc(interp.plot_top_losses)

In [20]: interp.plot_confusion_matrix(figsize=(12,12), dpi=60)

Confusion matrix
[[{"label": "beagle", "pred": "beagle", "count": 45}, {"label": "beagle", "pred": "Ragdoll", "count": 27}, {"label": "beagle", "pred": "english_setter", "count": 40}, {"label": "beagle", "pred": "wheaten_terrrier", "count": 40}, {"label": "beagle", "pred": "Russian_Blue", "count": 37}, {"label": "beagle", "pred": "Frontier_Mini", "count": 38}, {"label": "Ragdoll", "pred": "beagle", "count": 27}, {"label": "Ragdoll", "pred": "Ragdoll", "count": 40}, {"label": "Ragdoll", "pred": "english_setter", "count": 5}, {"label": "Ragdoll", "pred": "wheaten_terrrier", "count": 0}, {"label": "Ragdoll", "pred": "Russian_Blue", "count": 0}, {"label": "Ragdoll", "pred": "Frontier_Mini", "count": 0}, {"label": "english_setter", "pred": "beagle", "count": 40}, {"label": "english_setter", "pred": "Ragdoll", "count": 0}, {"label": "english_setter", "pred": "english_setter", "count": 5}, {"label": "english_setter", "pred": "wheaten_terrrier", "count": 0}, {"label": "english_setter", "pred": "Russian_Blue", "count": 0}, {"label": "english_setter", "pred": "Frontier_Mini", "count": 0}, {"label": "wheaten_terrrier", "pred": "beagle", "count": 40}, {"label": "wheaten_terrrier", "pred": "Ragdoll", "count": 0}, {"label": "wheaten_terrrier", "pred": "english_setter", "count": 0}, {"label": "wheaten_terrrier", "pred": "wheaten_terrrier", "count": 0}, {"label": "wheaten_terrrier", "pred": "Russian_Blue", "count": 0}, {"label": "wheaten_terrrier", "pred": "Frontier_Mini", "count": 0}, {"label": "Russian_Blue", "pred": "beagle", "count": 37}, {"label": "Russian_Blue", "pred": "Ragdoll", "count": 0}, {"label": "Russian_Blue", "pred": "english_setter", "count": 0}, {"label": "Russian_Blue", "pred": "wheaten_terrrier", "count": 0}, {"label": "Russian_Blue", "pred": "Russian_Blue", "count": 0}, {"label": "Russian_Blue", "pred": "Frontier_Mini", "count": 2}, {"label": "Frontier_Mini", "pred": "beagle", "count": 38}, {"label": "Frontier_Mini", "pred": "Ragdoll", "count": 0}, {"label": "Frontier_Mini", "pred": "english_setter", "count": 0}, {"label": "Frontier_Mini", "pred": "wheaten_terrrier", "count": 0}, {"label": "Frontier_Mini", "pred": "Russian_Blue", "count": 0}, {"label": "Frontier_Mini", "pred": "Frontier_Mini", "count": 1}, {"label": "Frontier_Mini", "pred": "Unknown", "count": 2}]]
```

`doc` tells you the same information as `help` but it has this very important thing which is `Show in docs`. When you click on it, it pops up the documentation for that method or class or function or whatever:

vision.learner | fastai

https://docs.fast.ai/vision.learner#ClassificationInterpretation.plot_top_losses

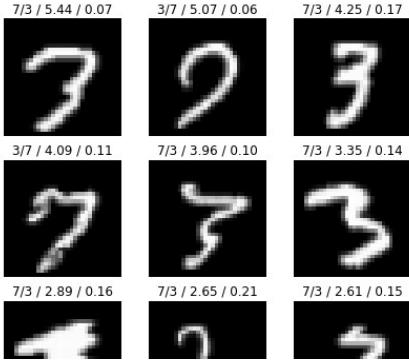
plot_top_losses

```
plot_top_losses(k, largest=True, figsize=(12, 12))

Show images in top_losses along with their loss, label, and prediction.

The k items are arranged as a square, so it will look best if k is a square number (4, 9, 16, etc). The title of each image shows: prediction, actual, loss, probability of actual class.

interp.plot_top_losses(9, figsize=(7,7))
```



It starts out by showing us the same information about what are the parameters it takes a long with the doc string. But then tells you more information:

The title of each image shows: prediction, actual, loss, probability of actual class.

The documentation always has working code. This is your friend when you're trying to figure out how to use these things. The other thing I'll mention is if you're somewhat experienced Python programmer, you'll find the source code of fastai really easy to read. We are trying to write everything in just a small number (much less than half a screen) of code. If you click on [source] you can jump straight to the source code.

```
82     def plot_top_losses(self, k, largest=True, figsize=(12,12)):
83         "Show images in 'top_losses' along with their prediction, actual, loss, and probability of actual class."
84         tl_val,tl_idx = self.top_losses(k,largest)
85         classes = self.data.classes
86         rows = math.ceil(math.sqrt(k))
87         fig,axes = plt.subplots(rows,rows,figsize=figsize)
88         for i,idx in enumerate(tl_idx):
89             t=self.data.valid_ds[idx]
90             t[0].show(ax=axes.flat[i], title=
91                         f'{classes[self.pred_class[idx]]}/{classes[t[1]]} / {self.losses[idx]:.2f} / {self.probs[idx][t[1]]:.2f}'")
```

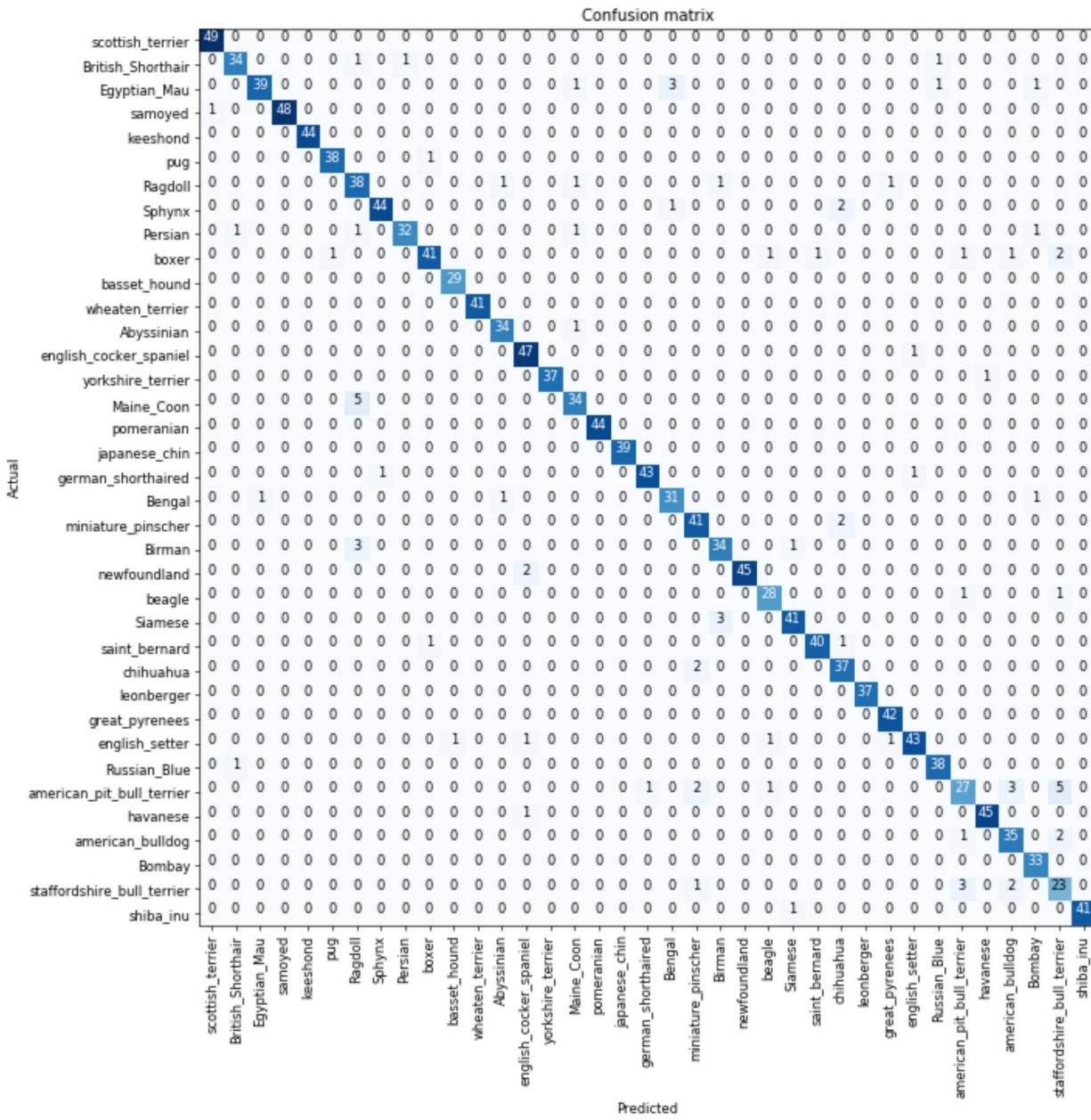
Here is `plot_top_loss`, and this is also a great way to find out how to use the fastai library. Because nearly every line of code here, is calling stuff in the fastai library. So don't be afraid to look at the source code.

[1:12:48]

So that's how we can look at top losses and these are perhaps the most important image classification interpretation tools that we have because it lets us see what we are getting wrong. In this case, if you are a dog and cat expert, you'll realize that the things that's getting wrong are breeds that are actually very difficult to tell apart and you'd be able to look at these and say "oh I can see why they've got this one wrong". So this is a really useful tool.

Confusion matrix 1:13:21

Another useful tool, kind of, is to use something called a confusion matrix which basically shows you for every actual type of dog or cat, how many times was it predicted to be that dog or cat. But unfortunately, in this case, because it's so accurate, this diagonal basically says how it's pretty much right all the time.



And you can see there are slightly darker ones like a five here, it's really hard to read exactly what their combination is. So what I suggest you use instead of, if you've got lots of classes, don't use confusion matrix, but this is my favorite named function in fastai and I'm very proud of this - you can call "most confused".

Most confused [1:13:52]

```
interp.most_confused(min_val=2)
```

```
[('american_pit_bull_terrier', 'staffordshire_bull_terrier', 5),
```

```
('Birman', 'Ragdoll', 5),
('english_setter', 'english_cocker_spaniel', 4),
('staffordshire_bull_terrier', 'american_pit_bull_terrier', 4),
('boxer', 'american_bulldog', 4),
('Ragdoll', 'Birman', 3),
('miniature_pinscher', 'chihuahua', 3),
('Siamese', 'Birman', 3)]
```

`most_confused` will simply grab out of the confusion matrix the particular combinations of predicted and actual that got wrong the most often. So this case, (`'american_pit_bull_terrier', 'staffordshire_bull_terrier'`, 7) :

- Actual `'american_pit_bull_terrier'`
- Prediction `'staffordshire_bull_terrier'`
- This particular combination happened 7 times.

So this is a very useful thing because you can look and say "with my domain expertise, does it make sense?"

Unfreezing, fine-tuning, and learning rates [1:14:38]

Let's make our model better. How? We can make it better by using fine-tuning. So far we fitted 4 epochs and it ran pretty quickly. The reason it ran pretty quickly is that there was a little trick we used. These convolutional networks, they have many layers. We'll learn a lot about exactly what layers are, but for now, just know it goes through a lot of computations. What we did was we added a few extra layers to the end and we only trained those. We basically left most of the model exactly as it was, so that's really fast. If we are trying to build a model at something that's similar to the original pre-trained model (in this case, similar to the ImageNet data), that works pretty well.

But what we really want to do is to go back and train the whole model. This is why we pretty much always use this two stage process. By default, when we call `fit` or `fit_one_cycle` on a ConvLearner, it'll just fine-tune these few extra layers added to the end and it will run very fast. It will basically never overfit but to really get it good, you have to call `unfreeze`. `unfreeze` is the thing that says please train the whole model. Then I can call `fit_one_cycle` again.

```
learn.unfreeze()
learn.fit_one_cycle(1)
```

```
Total time: 00:20
epoch  train_loss  valid_loss  error_rate
1      1.045145    0.505527    0.159681    (00:20)
```

Uh-oh. The error got much worse. Why? In order to understand why, we are actually going to have to learn more about exactly what's going on behind the scenes. So let's start out by trying to get an intuitive understanding of what's going on behind the scenes. We are going to do it by looking at pictures.

[1:16:28]

Visualizing and Understanding Convolutional Networks

Matthew D. Zeiler

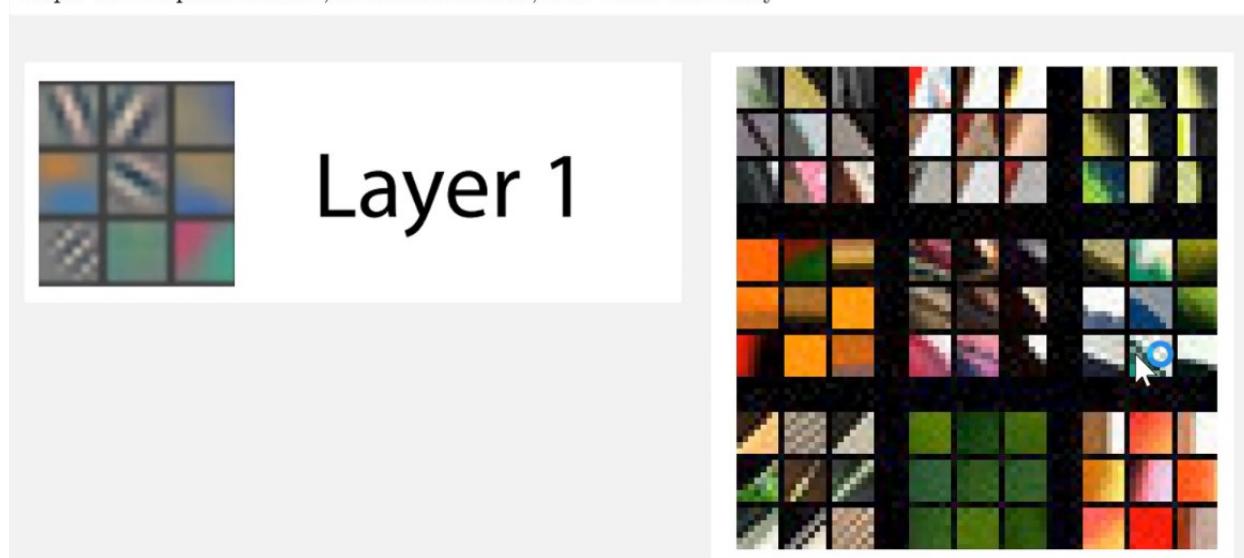
ZEILER@CS.NYU.EDU

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

Rob Fergus

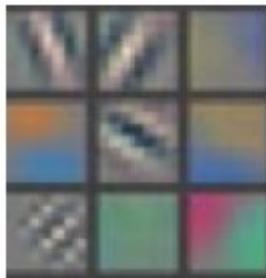
FERGUS@CS.NYU.EDU

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

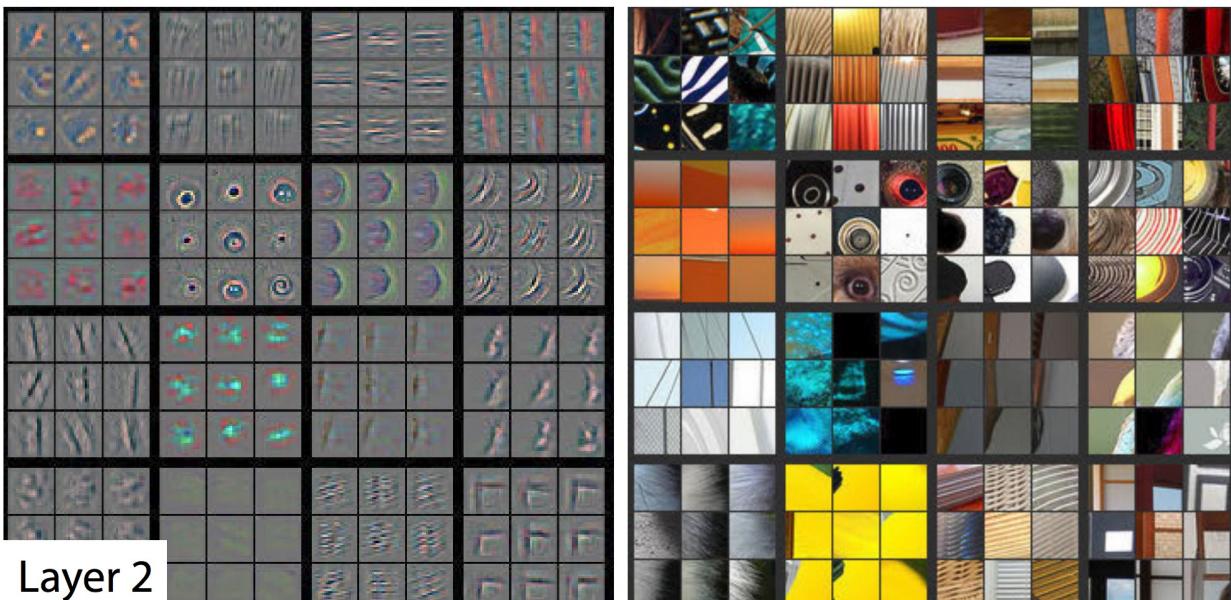


These pictures come from [a fantastic paper](#) by Matt Zeiler who nowadays is a CEO of Clarify which is a very successful computer vision startup and his supervisor for his PhD Rob Fergus. They wrote a paper showing how you can visualize the layers of a convolutional neural network. A convolutional neural network, which we will learn mathematically about what the layers are shortly, but the basic idea is that your red, green, and blue pixel values that are numbers from nought to 255 go into the simple computation (i.e. the first layer) and something comes out of that, and then the result of that goes into a second layer, and the result of that goes into the third layer and so forth. There can be up to a thousand layers of neural network. ResNet34 has 34 layers, and ResNet50 has 50 layers, but let's look at layer one. There's this very simple computation which is a convolution if you know what they are. What comes out of this first layer? Well, we can actually visualize these specific coefficients, the specific parameters by drawing them as a picture. There's actually a few dozen of them in the first layer, so we don't draw all of them. Let's just look at 9 at random.

[1:17:45]

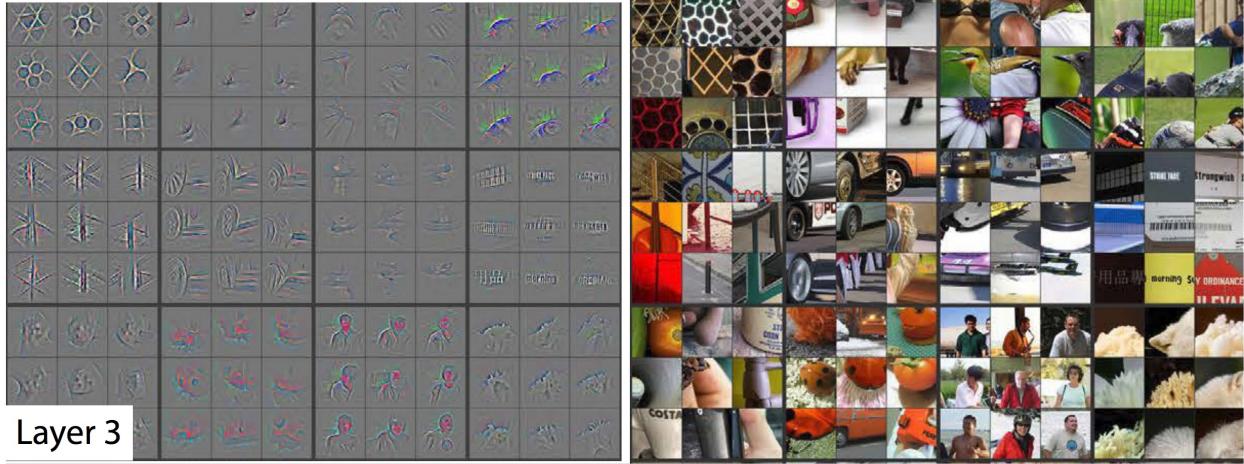


Here are nine examples of the actual coefficients from the first layer. So these operate on groups of pixels that are next to each other. So this first one basically finds groups of pixels that have a little diagonal line, the second one finds diagonal line in the other direction, the third one finds gradients that go from yellow to blue, and so forth. They are very simple little filters. That's layer one of ImageNet pre-trained convolutional neural net.

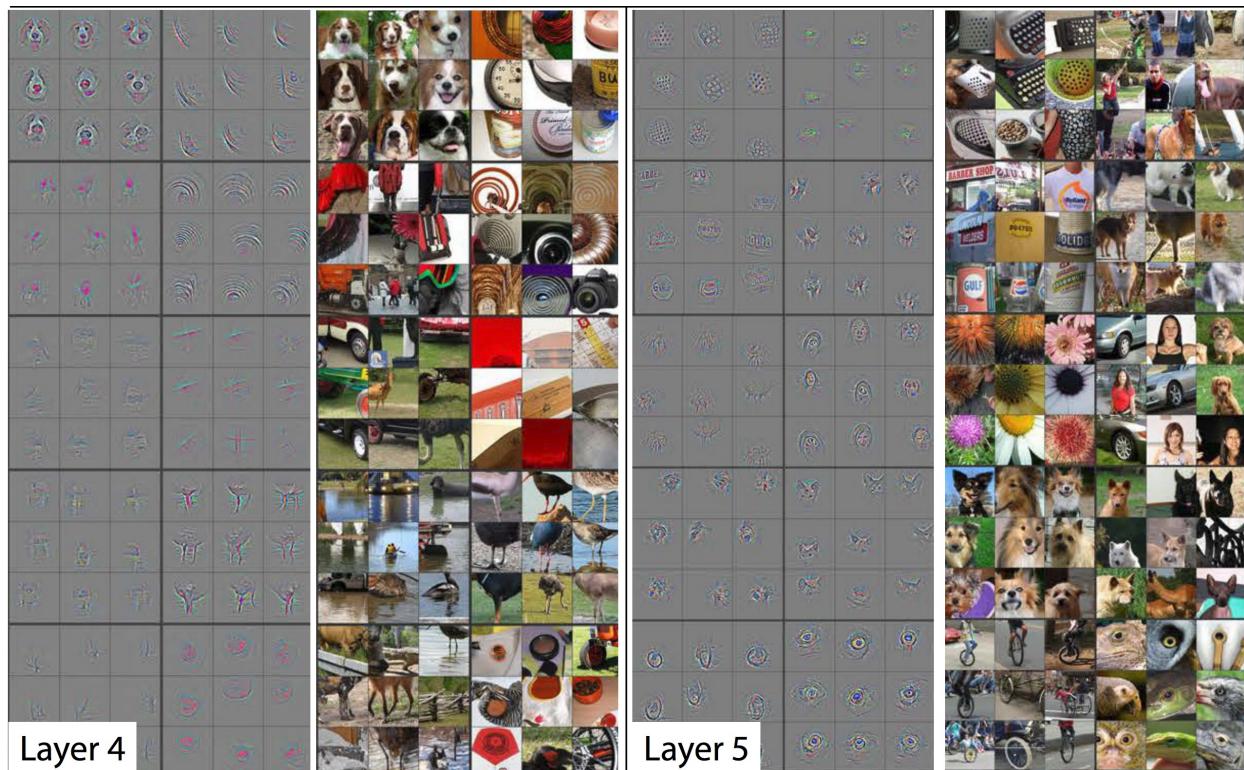


Layer 2 takes the results of those filters and does a second layer of computation. The bottom right are nine examples of a way of visualizing one of the second layer features. As you can see, it basically learned to create something that looks for top left corners. There are ones that learned to find right-hand curves, and little circles, etc. In layer one, we have things that can find just one line, and in layer 2, we can find things that have two lines joined up or one line repeated. If you then look over to the right, these nine show you nine examples of actual bits of the actual photos that activated this filter a lot. So in other words, the filter on the bottom right was good at finding these window corners etc.

So this is the kind of stuff you've got to get a really good intuitive understanding for. The start of my neural net is going to find very simple gradients and lines, the second layer can find very simple shapes, the third layer can find combination of those.



Now we can find repeating pattern of two dimensional objects or we can find things that joins together, or bits of text (although sometimes windows) - so it seems to find repeated horizontal patterns. There are also ones that seem to find edges of fluffy or flowery things or geometric patterns. So layer 3 was able to take all the stuff from layer 2 and combine them together.



Layer 4 can take all the stuff from layer 3 and combine them together. By layer 4, we got something that can find dog faces or bird legs.

By layer 5, we've got something that can find the eyeballs of bird and lizards, or faces of particular breeds of dogs and so forth. So you can see how by the time you get to layer 34, you can find specific dog breeds and cat breeds. This is kind of how it works.

So when we first trained (i.e. fine-tuned) the pre-trained model, we kept all of these layers that you've seen so far and we just trained a few more layers on top of all of those sophisticated features that are already being created. So now we are going back and saying "let's change all of these". We will start with where they are, but let's see if we can make them better.

Now, it seems very unlikely that we can make layer 1 features better. It's very unlikely that the definition of a diagonal line is going to be different when we look at dog and cat breeds versus the ImageNet data that this was originally trained on. So we don't really want to change the layer 1 very much if at all. Or else, the last layers, like types of dog face seems very likely that we do want to change that. So you want this intuition, this understanding that the different layers of a neural network represents different level of semantic complexity.

[1:22:06]

This is why our attempt to fine-tune this model didn't work because by default, it trains all the layers at the same speed which is to say it will update those things representing diagonal lines and gradients just as much as it tries to update the things that represent the exact specifics of what an eyeball looks like, so we have to change that.

To change it, we first of all need to go back to where we were before. We just broke this model, much worse than it started out. So if we just go:

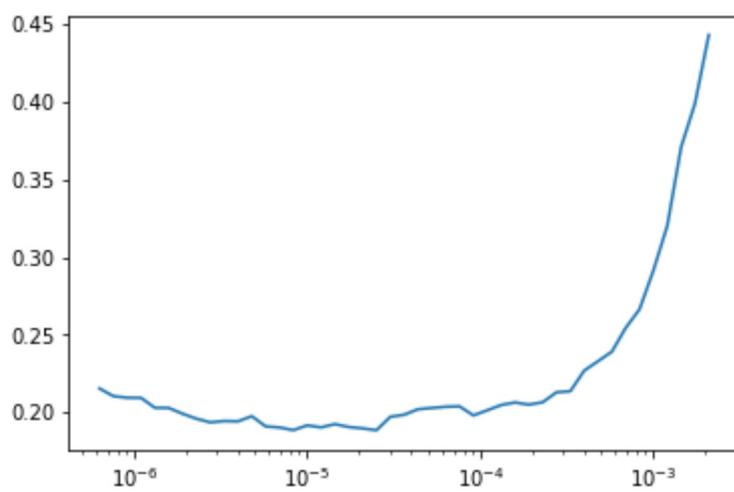
```
learn.load('stage-1')
```

This brings back the model that we saved earlier. So let's load that back up and now our models back to where it was before we killed it.

Learning rate finder [1:22:58]

Let's run learning rate finder. We are learning about what that is next week, but for now, just know this is the thing that figures out what is the fastest I can train this neural network at without making it zip off the rails and get blown apart.

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



This will plot the result of our LR finder and what this basically shows you is this key parameter called a learning rate. The learning rate basically says how quickly am I updating the parameters in my model. The x-axis one here shows me what happens as I increase the learning rate. The y axis show what the loss is. So you can see, once the learning rate gets passed 10^{-4} , my loss gets worse. It actually so happens, in fact I can check this if I press `shift + tab` here, my learning defaults to 0.003. So you can see why our loss got worse. Because we are trying to fine-tune things now, we can't use such a high learning rate. So based on the learning rate finder, I tried to pick something well before it started getting worse. So I decided to pick `1e-6`. But there's no point training all the layers at that rate, because we know that the later layers worked just fine before when we were training much more quickly. So what we can actually do is we can pass a range of learning rates to `learn.fit_one_cycle`. And we do it like this:

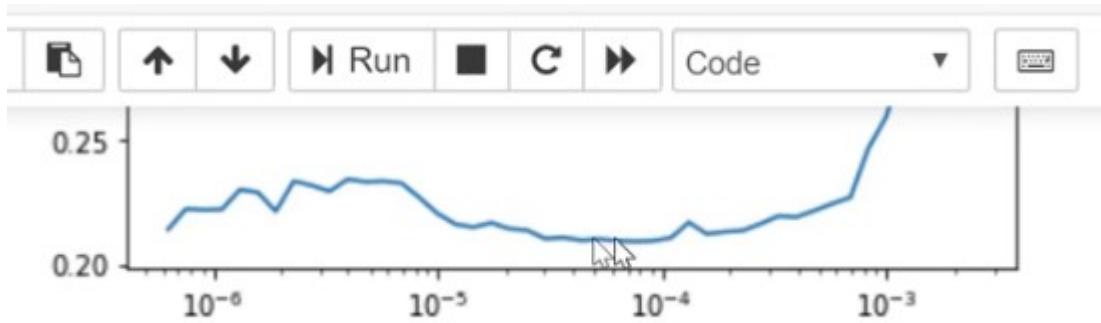
```
learn.unfreeze()
learn.fit_one_cycle(2, max_lr=slice(1e-6,1e-4))
```

```
Total time: 00:41
epoch  train_loss  valid_loss  error_rate
1      0.226494    0.173675    0.057219    (00:20)
2      0.197376    0.170252    0.053227    (00:20)
```

You use this keyword in Python called `slice` and that can take a start value and a stop value and basically what this says is train the very first layers at a learning rate of $1e-6$, and the very last layers at a rate of $1e-4$, and distribute all the other layers across that (i.e. between those two values equally).

How to pick learning rates after unfreezing [1:25:23]

A good rule of thumb is after you unfreeze (i.e. train the whole thing), pass a max learning rate parameter, pass it a slice, make the second part of that slice about 10 times smaller than your first stage. Our first stage defaulted to about 1e-3 so it's about 1e-4. And the first part of the slice should be a value from your learning rate finder which is well before things started getting worse. So you can see things are starting to get worse maybe about here:



So I picked something that's at least 10 times smaller than that.

If I do that, then the error rate gets a bit better. So I would perhaps say for most people most of the time, these two stages are enough to get pretty much a world-class model. You won't win a Kaggle competition, particularly because now a lot of fastai alumni are competing on Kaggle and this is the first thing that they do. But in practice, you'll get something that's about as good in practice as the vast majority of practitioners can do.

ResNet50 [1:26:55]

We can improve it by using more layers and we will do this next week but by basically doing a ResNet50 instead of ResNet34. And you can try running this during the week if you want to. You'll see it's exactly the same as before, but I'm using ResNet50.

```
data = ImageDataBunch.from_name_re(path_img, fnames, pat, ds_tfms=get_transforms()
data.normalize(imagenet_stats)

learn = ConvLearner(data, models.resnet50, metrics=error_rate)
```

What you'll find is it's very likely if you try to do this, you will get an error and the error will be your GPU has ran out of memory. The reason for that is that ResNet50 is bigger than ResNet34, and therefore, it has more parameters and use more of your graphics card memory, just totally separate to your normal computer RAM, this is GPU RAM. If you're using the default Salamander, AWS, then you'll be having a 16G of GPU memory. The card I use most of the time has 11G GPU memory, the cheaper ones have 8G. That's kind of the main range you tend to get. If yours have less than 8G of GPU memory, it's going to be frustrating for you.

It's very likely that if you try to run this, you'll get an out of memory error and that's because it's just trying to do too much - too many parameter updates for the amount of RAM you have. That's easily fixed. `ImageDataBunch` constructor has a parameter at the end `bs` - a batch size. This basically says how many images do you train at one time. If you run out of memory, just make it smaller.

It's fine to use a smaller batch size. It might take a little bit longer. That's all. So that's just one number you'll need to try during the week.

```
learn.fit_one_cycle(8, max_lr=slice(1e-3))
```

```
Total time: 07:08
epoch  train_loss  valid_loss  error_rate
1      0.926640   0.320040   0.076555   (00:52)
2      0.394781   0.205191   0.063568   (00:52)
3      0.307754   0.203281   0.069036   (00:53)
4      0.244182   0.160488   0.054682   (00:53)
5      0.185785   0.153520   0.049214   (00:53)
6      0.157732   0.149660   0.047163   (00:53)
7      0.107212   0.136898   0.043062   (00:53)
8      0.097324   0.136638   0.042379   (00:54)
```

Again, we fit it for a while and we get down to 4.2% error rate. So this is pretty extraordinary. I was pretty surprised because when we first did in the first course, this cats vs. dogs, we were getting somewhere around 3% error for something where you've got a 50% chance of being right and the two things look totally different. So the fact that we can get 4.2% error for such a fine grain thing, it's quite extraordinary.

Interpreting the results again 1:29:41

```
interp = ClassificationInterpretation.from_learner(learn)
interp.most_confused(min_val=2)

[('Ragdoll', 'Birman', 7),
 ('american_pit_bull_terrier', 'staffordshire_bull_terrier', 6),
 ('Egyptian_Mau', 'Bengal', 6),
 ('Maine_Coon', 'Bengal', 3),
 ('staffordshire_bull_terrier', 'american_pit_bull_terrier', 3)]
```

You can call the most_confused here and you can see the kinds of things that it's getting wrong. Depending on when you run it, you're going to get slightly different numbers, but you'll get roughly the same kind of things. So quite often, I find the Ragdoll and Birman are things that it gets confused. I actually have never heard of either of those things, so I actually looked them up and found a page on the cat site called "Is this a Birman or Ragdoll kitten?" and there was a long thread of cat experts arguing intensely about which it is. So I feel fine that my computer had problems.

thecatsite HOME FORUMS ARTICLES REVIEWS GALLERY MEMBERS Q SEARCH

Is this a Birman or Ragdoll kitten?
Discussion in 'Describing Cats - What Does My Cat Look Like?' started by esteevius, Jan 14, 2015.

Jan 14, 2015 #1

 esteevius Thread Starter TCS Member Kitten 8 1 Jan 14, 2015 New Mexico

I adopted my boy, Etson at the animal shelter on November 9th, 2014. He eats more than my other kitten Gando and I sometimes have to stop him. At first I thought he was Siamese because I have never really researched cat types.. but someone was visiting and said he was definitely not Siamese, but looked like a Ragdoll. I decided to look into cat types.

The two types I believe he resembles the most are the Ragdoll and the Birman. The animal shelter only had him listed as a Domestic shorthair, but his fur is quite longer and really soft. He also has white back legs, which Birman tend to not have. The nose however looks more Birman to me, but I'm really having a hard time deciding. He's grown quite a bit since I got him, considering he's only almost 3 months old. I'm just wondering if I should start looking for a bigger litter box early.. haha.

(The shadowing on his youngest and first picture looks like he has coloring on his rear legs, but he does not.)



I found something similar, I think it was this pitbull versus staffordshire bull terrier, apparently the main difference is the particular kennel club guidelines as to how they are assessed. But some people think that one of them might have a slightly redder nose. So this is the kind of stuff where actually even if you're not a domain expert, it helps you become one. Because I now know more about which kinds of pet breeds are hard to identify than I used to. So model interpretation works both ways.

Homework [1:30:58]

So what I want you to do this week is to run this notebook, make sure you can get through it, but then I really want you to do is to get your own image dataset and actually Francisco is putting together a guide that will show you how to download data from Google Images so you can create your own dataset to play with. But before I go, I want to show you how to create labels in lots of different ways because your dataset where you get it from won't necessarily be that kind of regex based approach. It could be in lots of different formats. So to show you how to do this, I'm going to use the MNIST sample. MNIST is a picture of hand drawn numbers - just because I want to show you different ways of creating these datasets.

```
path = untar_data(URLs.MNIST_SAMPLE); path
```

```
path.ls()
```

```
['train', 'valid', 'labels.csv', 'models']
```

You see there are a training set and the validation set already. So basically the people that put together this dataset have already decided what they want you to use as a validation set.

Scenario 1: Labels are folder names

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

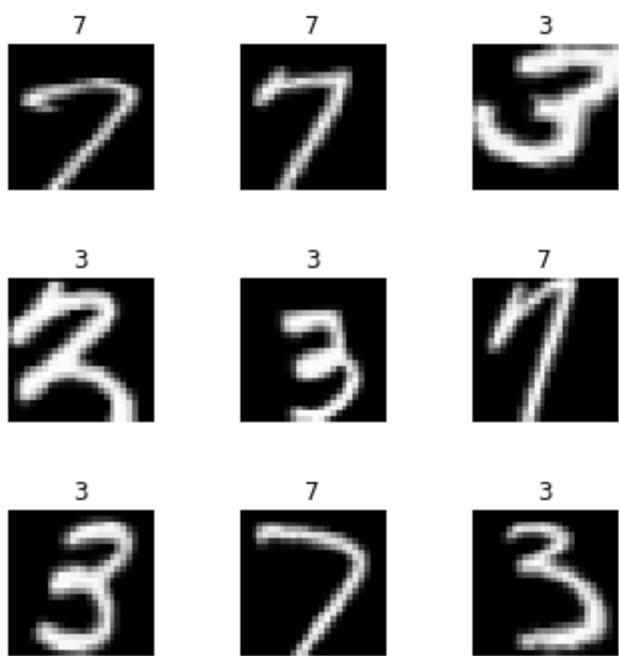
```
['3', '7']
```

There are a folder called 3 and a folder called 7. Now this is really common way to give people labels. Basically it says everything that's a three, I put in a folder called three. Everything that's a seven, I'll put in a folder called seven. This is often called an "ImageNet style dataset" because this is how ImageNet is distributed. So if you have something in this format where the labels are just whatever the folders are called, you can say `from_folder`.

```
tfms = get_transforms(do_flip=False)
data = ImageDataBunch.from_folder(path, ds_tfms=tfms, size=26)
```

This will create an `ImageDataBunch` for you and as you can see it created the labels:

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



Scenario 2: CSV file [1:33:17]

Another possibility, and for this MNIST sample, I've got both, it might come with a CSV file that would look something like this.

```
df = pd.read_csv(path/'labels.csv')
df.head()
```

	name	label
0	train/3/7463.png	0
1	train/3/21102.png	0
2	train/3/31559.png	0
3	train/3/46882.png	0
4	train/3/26209.png	0

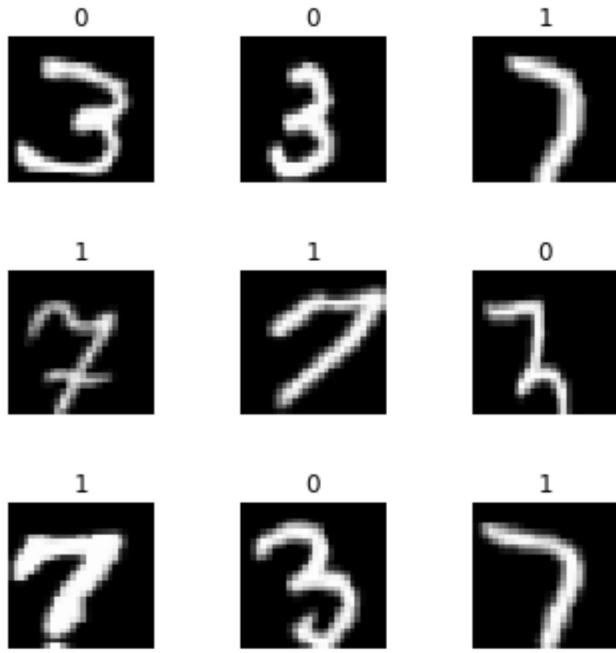
For each file name, what's its label. In this case, labels are not three or seven, they are 0 or 1 which basically is it a 7 or not. So that's another possibility. If this is how your labels are, you can use `from_csv`:

```
data = ImageDataBunch.from_csv(path, ds_tfms=tfms, size=28)
```

And if it is called `labels.csv`, you don't even have to pass in a file name. If it's called something else, then you can pass in the `csv_labels`

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

```
[0, 1]
```



Scenario 3: Using regular expression

```
fn_paths = [path/name for name in df['name']]; fn_paths[:2]
```

```
[PosixPath('/home/jhoward/.fastai/data/mnist_sample/train/3/7463.png'),
 PosixPath('/home/jhoward/.fastai/data/mnist_sample/train/3/21102.png')]
```

This is the same thing, these are the folders. But I could actually grab the label by using a regular expression. We've already seen this approach:

```
pat = r"/(\d)/\d+\.\png$"
data = ImageDataBunch.from_name_re(path, fn_paths, pat=pat, ds_tfms=tfms, size=24
data.classes
```

```
['3', '7']
```

Scenario 4: Something more complex [1:34:21]

You can create an arbitrary function that extracts a label from the file name or path. In that case, you would say `from_name_func` :

```
data = ImageDataBunch.from_name_func(path, fn_paths, ds_tfms=tfms, size=24,
                                     label_func = lambda x: '3' if '/3/' in str(x) else '7')
data.classes
```

Scenario 5: You need something even more flexible

If you need something even more flexible than that, you're going to write some code to create an array of labels. So in that case, you can just use `from_lists` and pass in the array.

```
labels = [('3' if '/3/' in str(x) else '7') for x in fn_paths]
labels[:5]
```

```
data = ImageDataBunch.from_lists(path, fn_paths, labels=labels, ds_tfms=tfms, size=24)
data.classes
```

So you can see there's lots of different ways of creating labels. So during the week, try this out.

Now you might be wondering how would you know to do all these things? Where am I going to find this kind of information? So I'll show you something incredibly cool. You know how to get documentation:

```
doc(ImageDataBunch.from_name_re)
```

[\[Show in docs\]](#)

from_name_re

[source]

```
from_name_re( path : PathOrStr , fnames : FilePathList , pat : str ,
    valid_pct : int = 0.2 , test : str = None , kwargs )
```

Creates an `ImageDataBunch` from `fnames`, calling a regular expression (containing one *re group*) on the file names to get the labels, putting aside `valid_pct` for the validation. In the same way as `ImageDataBunch.from_csv`, an optional `test` folder contains unlabelled data.

Our previously created dataframe contains the labels in the filenames so we can leverage it to test this new method. `ImageDataBunch.from_name_re` needs the exact path of each file so we will append the data path to each filename before creating our `ImageDataBunch` object.

```
fn_paths = [path / name for name in df['name']] ; fn_paths[:2]

[PosixPath('/home/ubuntu/.fastai/data/mnist_sample/train/3/7463.png'),
 PosixPath('/home/ubuntu/.fastai/data/mnist_sample/train/3/21102.png')]

pat = r'/(\\d)\\d+.png$'
data = ImageDataBunch.from_name_re(path, fn_paths, pat=pat, ds_tfms=tfms, size=24)

data.classes

['3', '7']
```

from_name_func

[source]

Every single line of code I just showed you, I took it this morning and I copied and pasted it from the documentation. So you can see here the exact code that I just used. So the documentation for fastai doesn't just tell you what to do, but step to step how to do it. And here is perhaps the coolest bit. If you go to [fastai/fastai_docs](#) and click on [docs/src](#).

All of our documentation is actually just Jupyter Notebooks. You can git clone this repo and if you run it, you can actually run every single line of the documentation yourself.

This is the kind of the ultimate example to me of experimenting. Anything that you read about in the documentation, nearly everything in the documentation has actual working examples in it with actual datasets that are already sitting in there in the repo for you. So you can actually try every single function in your browser, try seeing what goes in and try seeing what comes out.

[1:37:27]

Question: Will the library use multi GPUs in parallel by default?

The library will use multiple CPUs by default but just one GPU by default. We probably won't be looking at multi GPU until part 2. It's easy to do and you'll find it on the forum, but most people won't be needing to use that now.

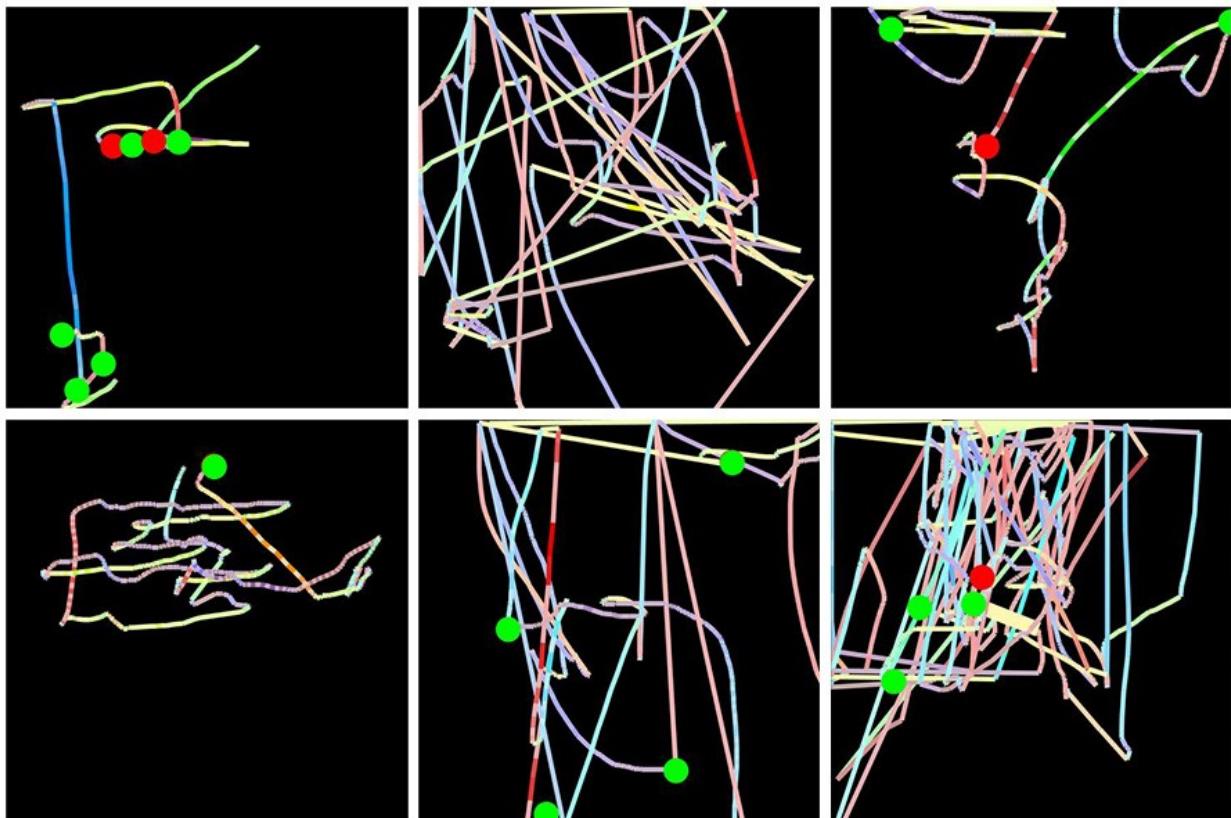
Question: Can the library use 3D data such as MRI or CAT scan?

Yes, it can. AND there is actually a forum thread about that already. Although that's not as developed as 2D yet but maybe by the time the MOOC is out, it will be.

blog

Before I wrap up, I'll just show you an example of the kind of interesting stuff that you can do by doing this kind of exercise.

Remember earlier I mentioned that one of our alumni who works at Splunk which is a NASDAQ listed big successful company created this new anti-fraud software. This is actually how he created it as part of a fastai part 1 class project:



He took the telemetry of users who had Splunk analytics installed and watched their mouse movements and he created pictures of the mouse movements. He converted speed into color and right and left clicks into splotches. He then took the exact code that we saw with an earlier version of the software and trained a CNN in exactly the same way we saw and used that to train his fraud model. So he took something which is not obviously a picture and he turned it into a picture and got these fantastically good results for a piece of fraud analysis software.

So it pays to think creatively. So if you are wanting to study sounds, a lot of people that study sounds do it by actually creating a spectrogram image and then sticking that into a ConvNet. So there's a lot of cool stuff you can do with this.

So during the week, get your GPU going, try and use your first notebook, make sure that you can use lesson 1 and work through it. Then see if you can repeat the process on your own dataset. Get on the forum and tell us any little success you had. Any constraints you hit, try it for an hour or two but if you get stuck, please ask. If you are able to successfully build a model with a new dataset, let us know! I will see you next week.



marcostoscano95 Fixed typo lesson2 (#35)

c4afa76 5 days ago

6 contributors



1309 lines (759 sloc) 105 KB

Raw

Blame

History



⌚ Lesson 2

[Video](#) / Lesson Forum / General Forum

⌚ Deeper Dive into Computer Vision

Taking a deeper dive into computer vision applications, taking some of the amazing stuff you've all been doing during the week, and going even further.

⌚ Forum tips and tricks [0:17]

Two important forum topics:

- [FAQ, resources, and official course updates](#)
- [Lesson 2 official resources and updates](#)

⌚ "Summarize This Topic" [2:32]

After just one week, the most popular thread has 1.1k replies which is intimidatingly large number. You shouldn't need to read all of it. What you should do is click "Summarize This Topic" and it will only show the most liked ones.



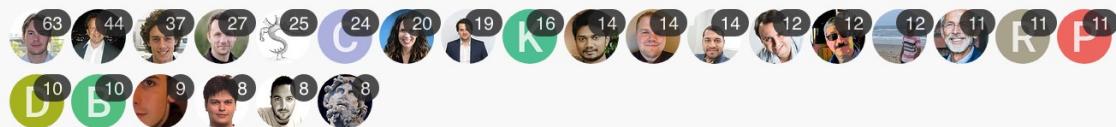
Lesson 1 chat

Part 1 v3



created 9d last reply 10h 1.1k replies 5.8k views 283 users 1.5k likes 50 links

Frequent Posters



Popular Links

- 104 Practical Deep Learning for Coders, v3 | fast.ai course v3 fast.ai
- 91 Another data science student's blog – The 1cycle policy sgugger.github.io
- 70 YouTube youtube.com
- 55 Deep Learning ver3 Lesson 1 – Vikas Jha – Medium medium.com
- 48 python - Google Colaboratory: misleading information about its GPU (only 5% RAM avail... stackoverflow.com



There are **1066** replies with an estimated read time of **87 minutes**.

[Summarize This Topic](#)

Back

⌚ Returning to work [3:19]

<https://course-v3.fast.ai/> now has a "Returning to work" section which will show you (for each specific platform you use):

- How to make sure you have the latest notebooks
- How to make sure you have the latest fastai library

If things aren't working for you, if you get into some kind of messy situation, which we all do, just delete your instance and start again unless you've got mission-critical stuff there — it's the easiest way just to get out of a sticky situation.

⌚ What people have been doing this week [4:19]

[Share your work here](#)

Share your work here ✅

Part 1 v3

created

6d

last reply

20m

167

replies

1.6k

views

84

users

48

likes



jeremy 🎤 Jeremy Howard (Admin)

Show us what you've created with what you learned in fast.ai! 😊



astronomy88 Harold Nguyen

2 1d

I was interested in doing voice recognition detection. I used Audacity (<https://www.audacityteam.org>) to trim the audio from the following clips:

1. Ben Affleck's speech in The Boiler Room (<https://www.youtube.com/watch?v=JfIKzReNDF4&t=62s> 1)
2. Joe Rogan and Elon Musk Podcast (<https://www.youtube.com/watch?v=Ra3fv8gl6NE>)

And used 3 min 30 seconds of audio voice from each of Ben Affleck, Joe Rogan, and Elon Musk.



chans.best chandan

2d

I always have to manually clean up my Whatsapp downloaded images folder because Memes and other images sit in same folder along with camera pics shared by my contacts. Hence i trained 34 model with 2000 images without unfreezing, 1000 manually classified images from my own Whatsapp and another 1000 sourced from google search.

- Figuring out who is talking — is it Ben Affleck or Joe Rogan
- Cleaning up Watsapp downloaded images folder to get rid of memes

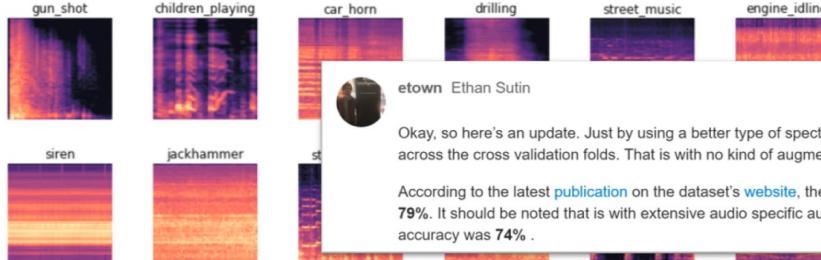
IEEE SIGNAL PROCESSING LETTERS, ACCEPTED NOVEMBER 2016

Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification

Justin Salamon and Juan Pablo Bello

```
data = ImageDataBunch.from_folder(data_directory/'1', ds_tfms=[], size=224)
data.normalize(imagenet_stats)
```

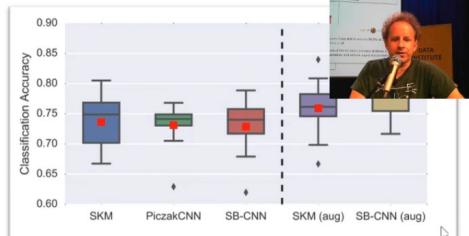
```
data.show_batch(rows=6, figsize=(12,12))
```



etown Ethan Sutin



Fig. 1. Left of the dashed line: classification accuracy without augmentation – dictionary learning (SKM [7]), Piczak's CNN (PiczakCNN [11]) and the proposed model (SB-CNN). Right of the dashed line: classification accuracy for SKM and SB-CNN with augmentation.



Okay, so here's an update. Just by using a better type of spectrogram, I was able to achieve 80.5% accuracy across the cross validation folds. That is with no kind of augmentation at all.

According to the latest [publication](#) on the dataset's [website](#), the state-of-the-art mean accuracy achieved was 79%. It should be noted that is with extensive audio specific augmentation, and without augmentation their top accuracy was 74% .

Forum post

One of the really interesting projects was looking at the sound data that was used in [this paper](#). In this paper, they were trying to figure out what kind of sound things were. They got a state of the art of nearly 80% accuracy. Ethan Sutin then tried using the lesson 1 techniques and got 80.5% accuracy, so I think this is pretty awesome. Best as we know, it's a new state of the art for this problem. Maybe somebody since has published something we haven't found it yet. So take all of these with a slight grain of salt, but I've mentioned them on Twitter and lots of people on Twitter follow me, so if anybody knew that there was a much better approach, I'm sure somebody would have said so.

[6:01]

State of the art on DHCD (देवनागरी)



Suvash Thapaliya @suvash · Oct 27
After lots of intentional tuneups, I can now announce that SoTA for accuracy on #DHCD #देवनागरी dataset is 99.02% (previously 98.50%). All thanks to @fastdotai library & courses, and the dataset creators @pk_gyawali amongst others.
#resnet34 #devanagari #deeplearning

Model fitting (transfer learning) on

_loss	valid_loss	accuracy
342	0.037567	0.989493
461	0.035617	0.990072
426	0.034761	0.990290

Training cycle (Accuracy = 99.02%)

3 12 63



Forum post

Suvash has a new state of the art accuracy for Devanagari text recognition. I think he's got it even higher than this now. This is actually confirmed by the person on Twitter who created the dataset. I don't think he had any idea, he just posted here's a nice thing I did and this guy on Twitter said: "Oh, I made that dataset. Congratulations, you've got a new record." So that was pretty cool.

[6:28]

Alena Harley Follow · Oct 29 · 6 min read

The Mystery of the Origin

Chapter

Approximately where they through the Determining problems in dependent on

Cancer class because the mutations in

EGFR inhibitors
Cyclin-dependent kinase inhibitors
Immune activating anti-CTLA4 mAb
Telomerase Inhibitors
Selective anti-inflammatory drugs
Inhibitors of VEGF signaling
Inhibitors of HGF/c-Met
Sustaining proliferative signaling
Evading growth suppressors
Avoiding immune destruction
Enabling replicative immortality
Tumor-promoting inflammation
Inducing angiogenesis
Activating invasion & metastasis
Regulating cellular energetics
Genome stability & mutation

point mutation data using pathways, but how? What in Gene2Vec encoding using information from gene membership in pathways.

The Mystery of the Origin

I really like this post from Alena Harley. She describes in quite a bit of detail about the issue of metastasizing cancers and the use of point mutations and why that's a challenging important problem. She's got some nice pictures describing what she wants to do with this and how she can go about turning this into pictures. This is the cool trick — it's the same with turning sounds into pictures and then using the lesson 1 approach. Here is turning point mutations into pictures and then using the lesson 1 approach. And it seems that she's got a new state of the art result by more than 30% beating the previous best. Somebody on Twitter who is a VP at a genomics analysis company looked at this as well and thought it looked to be a state of the art in this particular point mutation one as well. So that's pretty exciting.

When we talked about last week this idea that this simple process is something which can take you a long way, it really can. I will mention that something like this one in particular is using a lot of domain expertise, like figuring out that picture to create. I wouldn't know how to do that because I don't really know what a point mutation is, let alone how to create something that visually is meaningful that a CNN could recognize. But the actual deep learning side is actually straight forward.

[8:07]

Won at Science Hack Day

→ C https://simonwillison.net/2018/Oct/29/transfer-learning/

Simon Willison's Weblog

Automatically playing science communication games with transfer learning and fastai

This weekend was the 9th annual [Science Hack Day San Francisco](#), which was also the 100th Science Hack Day held worldwide.

Natalie and I decided to combine our interests and build something fun.

I'm currently enrolled in Jeremy Howard's Deep Learning course so I figured this was a great opportunity to try out some computer vision.

Natalie runs the [SciComm Games calendar](#) and accompanying [@SciCommGames](#) bot to promote and catalogue science communication hashtag games on Twitter.



Another cool result from Simon Willison and Natalie Downe, they created a cougar or not web application over the weekend and won the Science Hack Day award in San Francisco. So I think that's pretty fantastic. So lots of examples of people doing really interesting work. Hopefully this will be inspiring to you to think well to think wow, this is cool that I can do this with what I've learned. It can also be intimidating to think like wow, these people are doing amazing things. But it's important to realize that as thousands of people are doing this course, I'm just picking out a few of really amazing ones. And in fact Simon is one of these very annoying people like Christine Payne who we talked about last week who seems to be good at everything he does. He created Django which is the world's most popular web frameworks, he founded a very successful startup, etc. One of those annoying people who tends to keep being good at things, now turns out he's good at deep learning as well. So that's fine. Simon can go on and win a hackathon on his first week of playing with deep learning. Maybe it'll take you two weeks to win your first hackathon. That's okay.

[9:22]

James Dellinger [Follow](#)
Oct 29 · 13 min read

If I Can You Can (and you should!)

My machine learning and deep learning journey almost ended because I began back in early January of this year, when I saw [the following page](#) on scikit-learn.org. I momentarily and incorrectly assumed I wasn't "technical" enough to grasp machine learning because I couldn't immediately understand the mathematical notation that was pasted into articles all over the website.

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable y and dependent feature vector x_1 through x_n :

$$P(y | x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots, x_n | y)}{P(x_1, \dots, x_n)}$$

Because of the naive assumption that

$$P(x_i | y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i | y),$$

we have

$$P(y | x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^n P(x_i | y)}{P(x_1, \dots, x_n)}$$

Given the input, we can use the following classification rule:

$$P(y | x_1, \dots, x_n) \propto P(y) \prod_{i=1}^n P(x_i | y)$$
$$\hat{y} = \arg \max_y P(y) \prod_{i=1}^n P(x_i | y),$$

I think it's important to mention this because there was this really inspiring blog post this week from James Dellinger who talked about how he created a bird classifier using techniques from lesson 1. But what I really found interesting was at the end, he said he nearly didn't start on deep learning at all because he went through the scikit-learn website which is one of the most important libraries of Python and he saw this. And he described in this post how he was just like that's not something I can do. That's not something I understand. Then this kind of realization of like oh, I can do useful things without reading the Greek, so I thought that was really cool message.

[10:01]

Daniel.R.Armstrong Daniel Armstrong

Quote

5d



What is the best way to contribute to the fast.ai library? When I was reading the fast.ai developer docs, I found the whole process a little overwhelming. I am wondering if anyone has any best practices other than using hub as Jeremy described above.



Daniel.R.Armstrong Daniel Armstrong

Quote

4d

Unfortunately I can't tell you the specific parts of the developer docs that caused my confusion. I think my problem is I don't have any real world experience with all the nuances of git, and version controls. Frankly I had no idea how much there was so much to it. I think it just caught me off guard. I thought it was just push, pull, and clone. Reading the dev docs is kind of like reading a different language, my brain just shut down. 🐶. I will learn it all soon, but I will have to put in some time to learn it. Maybe then I can give you a better answer.

One of the reasons I love the fast.ai way, is it forces me to learn so much more than just the fast.ai library. I have learned about bash, terminal, linux, notebooks, curl, AWS, computer hardware, vim, tmux, debugger, pytorch, and so much more.



Daniel.R.Armstrong Daniel Armstrong



jeremy

3d

I am happy to say that I submitted my first PR this morning, and it was accepted/merged this morning! I was so excited!



I really wanted to highlight Daniel Armstrong on the forum. I think really shows he's a great role model here. He was saying I want to contribute to the library and I looked at the docs and I just found it overwhelming. The next message, one day later, was I don't know what any of this is, I didn't know how much there is to it, caught me off guard, my brain shut down but I love the way it forces me to learn so much. And a day later, I just submitted my first pull request. So I think that's awesome. It's okay to feel intimidated. There's a lot. But just pick one piece and dig into it. Try and push a piece of code or a documentation update, or create a classifier or whatever.

[10:49]

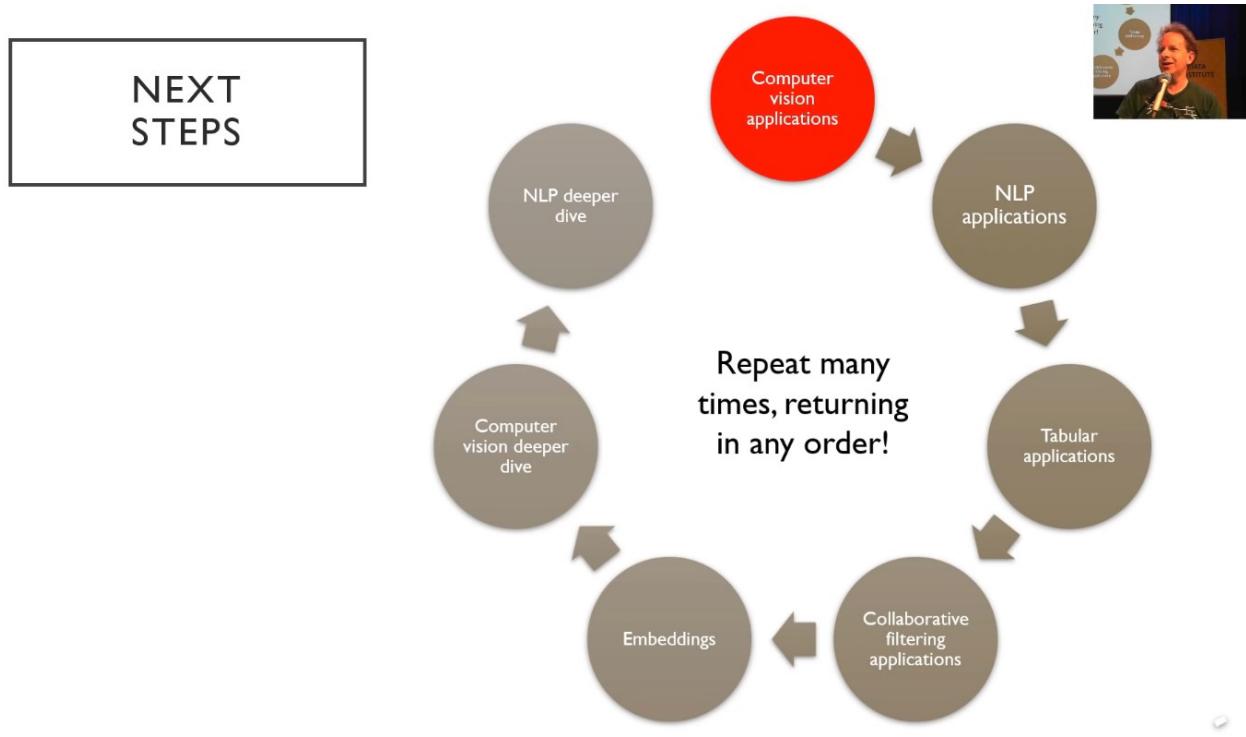
So here's lots of cool classifiers people have built. It's been really inspiring.

- Trinidad and Tobago islanders versus masquerader classifier
- A zucchini versus cucumber classifier
- Dog and cat breed classifier from last week and actually doing some exploratory work to see what the main features were, and discovered that one was most hairy dog and naked cats. So there are interesting things you can do with interpretation.
- Somebody else in the forum took that and did the same thing for anime to find that they had accidentally discovered an anime hair color classifier.
- We can now detect the new versus the old Panamanian buses.
- Henri Palacci discovered that he can recognize with 85% accuracy which of 110 countries a satellite image is of which is definitely got to be beyond human performance of just about anybody.
- Batik cloth classification with a hundred percent accuracy
- Dave Luo did this interesting one. He actually went a little bit further using some techniques we'll be discussing in the next couple of courses to build something that can recognize complete/incomplete/foundation buildings and actually plot them

on aerial satellite view.

So lots and lots of fascinating projects. So don't worry. It's only been one week. It doesn't mean everybody has to have had a project out yet. A lot of the folks who already have a project out have done a previous course, so they've got a bit of a head start. But we will see today how you can definitely create your own classifier this week.

[12:56]



So from today, after we did a bit deeper into really how to make these computer vision classifiers and particular work well, we're then going to look at the same thing for text. We're then going to look at the same thing for tabular data. They are more like spreadsheets and databases. Then we're going to look at collaborative filtering (i.e. recommendation systems). That's going to take us into a topic called embeddings which is a key underlying platform behind these applications. That will take us back into more computer vision and then back into more NLP. So the idea here is that it turns out that it's much better for learning if you see things multiple times so rather than being like okay, that's computer vision, you won't see it again for the rest of the course, we're actually going to come back to the two key applications NLP and computer vision a few weeks apart. That's going to force your brain to realize oh, I have to remember this. It's not something I can throw away.

[14:06]

IF YOU'RE STUCK, KEEP GOING!

Code first

Focus on learning from experiments

The whole game

It's like learning soccer as a kid (Perkins)

Concepts, not details

We'll gradually dig in to all the details

Do lesson 2

...even if you don't understand all of lesson 1

For people who have more of a hard sciences background in particular, a lot of folks find this hey, here's some code, type it in, start running it approach rather than here's lots of theory approach confusing and surprising and odd at first. So for those of you, I just wanted to remind you this basic tip which is keep going. You're not expected to remember everything yet. You're not expected to understand everything yet. You're not expected to know why everything works yet. You just want to be in a situation where you can enter the code and you can run it and you can get something happening and then you can start to experiment and you get a feel for what's going on. Then push on. Most of the people who have done the course and have gone on to be really successful watch the videos at least three times. So they kind of go through the whole lot and then go through it slowly the second time, then they go through it really slowly the third time. I consistently hear them say I get a lot more out of it each time I go through. So don't pause at lesson 1 and stop until you can continue.

This approach is based on a lot of academic research into learning theory. One guy in particular David Perkins from Harvard has this really great analogy. He is a researcher into learning theory. He describes this approach of whole game which is basically if you're teaching a kid to play soccer, you don't first of all teach them about how the friction between a ball and grass works and then teach them how to saw a soccer ball with their bare hands, and then teach them the mathematics of parabolas when you kick something in the air. No. You say, here's a ball. Let's watch some people playing soccer. Okay, now we'll play soccer and then gradually over the following years, learn more and more so that you can get better and better at it. So this is kind of what we're trying to get you to do is to play soccer which in our case is to type code and look at the inputs and look at the outputs.

⌚ Teddy bear detector using Google Images [16:21]

Let's dig into our first notebook which is called [lesson2-download.ipynb](#). What we are going to do is we are going to see how to create your own classifier with your own images. It's going to be a lot like last week's pet detector but it will detect whatever you like. So to be like some of those examples we just saw. How would you create your own Panama bus detector from scratch. This approach is inspired by Adrian Rosebrock who has a terrific website called [pyimagesearch](#) and he has this nice explanation of [how to create a deep learning dataset using Google Images](#). So that was definitely an inspiration for some of the techniques we use here, so thank you to Adrian and you should definitely check out his site. It's full of lots of good resources.

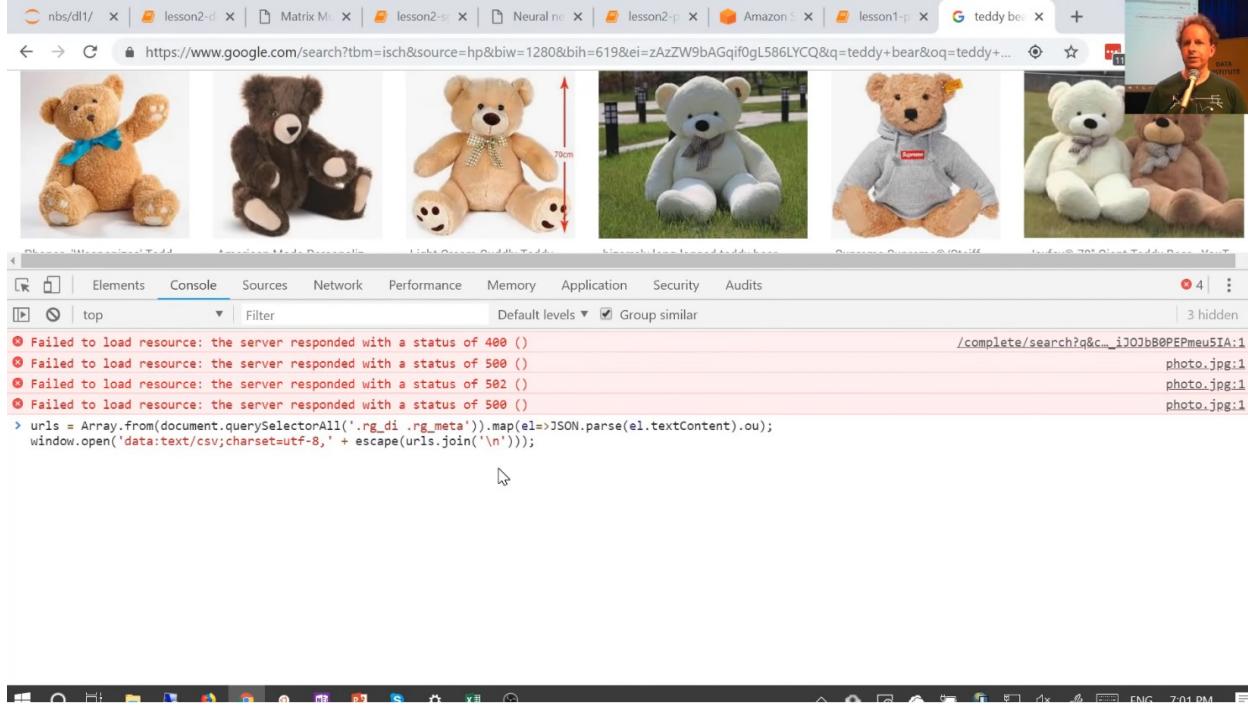
We are going to try to create a teddy bear detector. And we're going to separate teddy bears from black bears, from grizzly bears. This is very important. I have a three year old daughter and she needs to know what she's dealing with. In our house, you would be surprised at the number of monsters, lions, and other terrifying threats that are around particularly around Halloween. So we always need to be on the lookout to make sure that the things we're about to cuddle is in fact a genuine teddy bear. So let's deal with that situation as best as we can.

⌚ Step 1: Gather URLs of each class of images

Our starting point is to find some pictures of teddy bears so we can learn what they look like. So I go to <https://images.google.com/> and I type in Teddy bear and I just scroll through until I find a goodly bunch of them. Okay, that looks like plenty of teddy bears to me.

Then I go back to [the notebook](#) and you can see it says "go to Google Images and search and scroll." The next thing we need to do is to get a list of all the URLs there. To do that, back in your google images, you hit **Ctrl Shift J** in Windows/Linux and **Cmd Opt J** in Mac, and you paste the following into the window that appears:

```
urls = Array.from(document.querySelectorAll('.rg_di .rg_meta')).map(el=>JSON.parse(window.open('data:text/csv;charset=utf-8,' + escape(urls.join('\n'))));
```



This is a Javascript console for those of you who haven't done any Javascript before. I hit enter and it downloads my file for me. So I would call this `teddies.txt` and press "Save". Okay, now I have a file containing URLs of teddies. Then I would repeat that process for black bears and for grizzly bears, and I put each one in a file with an appropriate name.

⌚ Step 2: Download images [19:39]

So step 2 is we now need to download those URLs to our server. Because remember when we're using Jupyter Notebook, it's not running on our computer. It's running on SageMaker or Crestle, or Google cloud, etc. So to do that, we start running some Jupyter cells. Let's grab the `fastai` library:

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

And let's start with black bears. So I click on this cell for black bears and I'll run it. So here, I've got three different cells doing the same thing but different information. This is one way I like to work with Jupyter notebook. It's something that a lot of people with more strict scientific background are horrified by. This is not reproducible research. I click on the black bear cell, and run it to create a folder called `black` and a file called `urls_black.txt` for my black bears. I skip the next two cells.

```
folder = 'black'
file = 'urls_black.txt'
```

```
folder = 'teddys'
file = 'urls_teddys.txt'
```

```
folder = 'grizzly'  
file = 'urls_grizzly.txt'
```

Then I run this cell to create that folder.

```
path = Path('data/bears')  
dest = path/folder  
dest.mkdir(parents=True, exist_ok=True)
```

Then I go down to the next section and I run the next cell which is download images for black bears. So that's just going to download my black bears to that folder.

```
classes = ['teddys', 'grizzly', 'black']  
  
download_images(path/file, dest, max_pics=200)
```

Now I go back and I click on 'teddys'. And I scroll back down and repeat the same thing. That way, I'm just going backwards and forwards to download each of the classes that I want. Very manual but for me, I'm very iterative and very experimental, that works well for me. If you are better at planning ahead than I am, you can write a proper loop or whatever and do it that way. But when you see my notebooks and see things that are kind of like configuration cells (i.e. doing the same thing in different places), this is a strong sign that I didn't run this in order. I clicked one place, went to another, ran that. For me, I'm experimentalist. I really like to experiment in my notebook, I treat it like a lab journal, I try things out and I see what happens. So this is how my notebooks end up looking.

It's a really controversial topic. For a lot of people, they feel this is "wrong" that you should only ever run things top to bottom. Everything you do should be reproducible. For me, I don't think that's the best way of using human creativity. I think human creativity is best inspired by trying things out and seeing what happens and fiddling around. You can see how you go. See what works for you.

So that will download the images to your server. It's going to use multiple processes to do so. One problem there is if something goes wrong, it's a bit hard to see what went wrong. So you can see in the next section, there's a commented out section that says `max_workers=0`. That will do it without spinning up a bunch of processes and will tell you the errors better. So if things aren't downloading, try using the second version.

```
# If you have problems download, try with `max_workers=0` to see exceptions:  
# download_images(path/file, dest, max_pics=20, max_workers=0)
```

⌚ Step 3: Create ImageDataBunch [22:50]

The next thing that I found I needed to do was to remove the images that aren't actually images at all. This happens all the time. There's always a few images in every batch that are corrupted for whatever reason. Google image told us this URL had an image but it doesn't anymore. So we got this thing in the library called `verify_images` which will check all of the images in a path and will tell you if there's a problem. If you say `delete=True`, it will actually delete it for you. So that's a really nice easy way to end up with a clean dataset.

```
for c in classes:  
    print(c)  
    verify_images(path/c, delete=True, max_workers=8)
```

teddys

 100.00% [272/272 00:06]

grizzly

 100.00% [168/168 00:05]

```
cannot identify image file '/data0/datasets/part1v3/bears/grizzly/00000011.jpg'  
cannot identify image file '/data0/datasets/part1v3/bears/grizzly/00000014.jpg'  
black
```

 100.00% [176/176 00:05]

So at this point, I now have a bears folder containing a grizzly folder, teddys folder, and black folder. In other words, I have the basic structure we need to create an ImageDataBunch to start doing some deep learning. So let's go ahead and do that.

Now, very often, when you download a dataset from like Kaggle or from some academic dataset, there will often be folders called train, valid, and test containing the different datasets. In this case, we don't have a separate validation set because we just grabbed these images from Google search. But you still need a validation set, otherwise you don't know how well your model is going and we'll talk more about this in a moment.

Whenever you create a data bunch, if you don't have a separate training and validation set, then you can just say the training set is in the current folder (i.e. . because by default, it looks in a folder called train) and I want you to set aside 20% of the data, please. So this is going to create a validation set for you automatically and randomly. You'll see that whenever I create a validation set randomly, I always set my random seed to something fixed beforehand. This means that every time I run this code, I'll get the same validation set. In general, I'm not a fan of making my machine learning experiments reproducible (i.e. ensuring I get exactly the same results every time). The randomness is to me a really important part of finding out your solution stable and it is going to work each time you run it. But what is important is that you always have the same validation set. Otherwise when you are trying to decide has this hyper parameter change improved my model but you've got a different set of data you are testing it on, then you don't know maybe that set of data just happens to be a bit easier. So that's why I always set the random seed here.

```
np.random.seed(42)
data = ImageDataBunch.from_folder(path, train=".", valid_pct=0.2,
    ds_tfms=get_transforms(), size=224, num_workers=4).normalize(imagenet_stats)
```

[25:37]

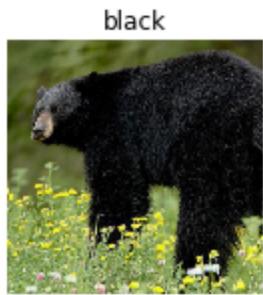
We've now got a data bunch, so you can look inside at the data.classes and you'll see these are the folders that we created. So it knows that the classes (by classes, we mean all the possible labels) are black bear, grizzly bear, or teddy bear.

```
data.classes
```

```
['black', 'grizzly', 'teddys']
```

We can run show_batch and take a little look. And it tells us straight away that some of these are going to be a little bit tricky. Some of them are not photo, for instance. Some of them are cropped funny, if you ended up with a black bear standing on top of a grizzly bear, that might be tough.

```
data.show_batch(rows=3, figsize=(7,8))
```



You can kind of double check here. Remember, `data.c` is the attribute which the classifiers tell us how many possible labels there are. We'll learn about some other more specific meanings of `c` later. We can see how many things are now training set, how many things are in validation set. So we've got 473 training set, 140 validation set.

```
data.classes, data.c, len(data.train_ds), len(data.valid_ds)
```

```
(['black', 'grizzly', 'teddys'], 3, 473, 140)
```

⌚ Step 4: Training a model [26:49]

So at that point, we can go ahead and create our convolutional neural network using that data. I tend to default to using a resnet34, and let's print out the error rate each time.

```
learn = create_cnn(data, models.resnet34, metrics=error_rate)
```

Then run `fit_one_cycle` 4 times and see how we go. And we have a 2% error rate. So that's pretty good. Sometimes it's easy for me to recognize a black bear from a grizzly bear, but sometimes it's a bit tricky. This one seems to be doing pretty well.

```
learn.fit_one_cycle(4)
```

```
Total time: 00:54
epoch  train_loss  valid_loss  error_rate
1      0.710584   0.087024   0.021277   (00:14)
2      0.414239   0.045413   0.014184   (00:13)
3      0.306174   0.035602   0.014184   (00:13)
4      0.239355   0.035230   0.021277   (00:13)
```

After I make some progress with my model and things are looking good, I always like to save where I am up to to save me the 54 seconds of going back and doing it again.

```
learn.save('stage-1')
```

As per usual, we unfreeze the rest of our model. We are going to be learning more about what that means during the course.

```
learn.unfreeze()
```

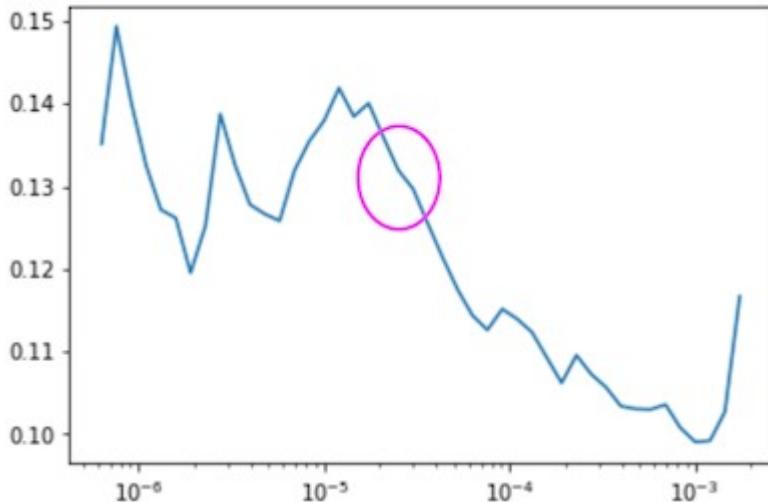
Then we run the learning rate finder and plot it (it tells you exactly what to type). And we take a look.

```
learn.lr_find()
```

```
LR Finder complete, type {learner_name}.recorder.plot() to see the graph.
```

We are going to be learning about learning rates today, but for now, here's what you need to know. On the learning rate finder, what you are looking for is the strongest downward slope that's kind of sticking around for quite a while. It's something you are going to have to practice with and get a feel for - which bit works. So if you are not sure which, try both learning rates and see which one works better. I've been doing this for a while and I'm pretty sure this (between 10^{-5} and 10^{-3}) looks like where it's really learning properly, so I will probably pick something back here for my learning rate [28:28].

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



So you can see, I picked `3e-5` for my bottom learning rate. For my top learning rate, I normally pick `1e-4` or `3e-4`, it's kind of like I don't really think about it too much. That's a rule of thumb - it always works pretty well. One of the things you'll realize is that most of these parameters don't actually matter that much in detail. If you just copy the numbers that I use each time, the vast majority of the time, it'll just work fine. And we'll see places where it doesn't today.

```
learn.fit_one_cycle(2, max_lr=slice(3e-5,3e-4))
```

```
Total time: 00:28
epoch  train_loss  valid_loss  error_rate
1      0.107059   0.056375   0.028369   (00:14)
2      0.070725   0.041957   0.014184   (00:13)
```

So we've got 1.4% error rate after doing another couple of epochs, so that's looking great. So we've downloaded some images from Google image search, created a classifier, and we've got 1.4% error rate, let's save it.

```
learn.save('stage-2')
```

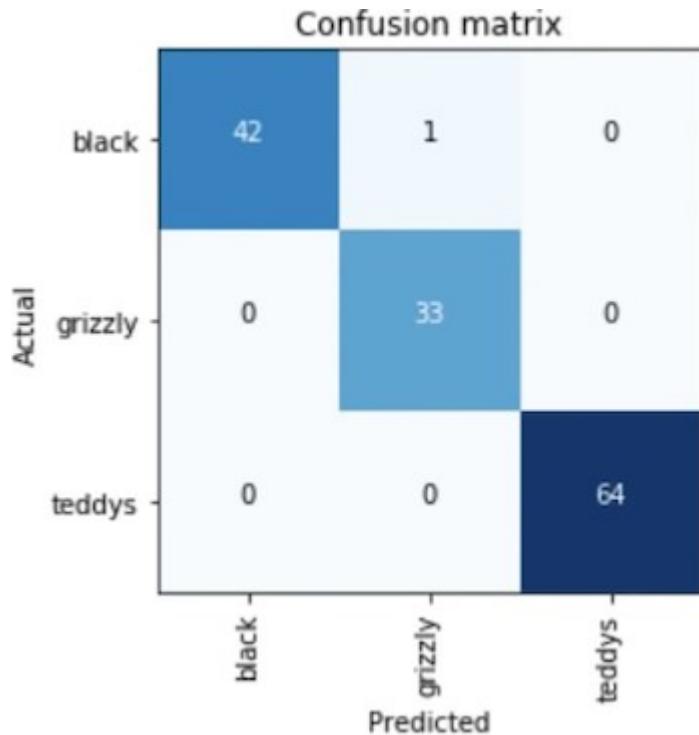
⌚ Interpretation [29:38]

As per usual, we can use the `ClassificationInterpretation` class to have a look at what's going on.

```
learn.load('stage-2')
```

```
interp = ClassificationInterpretation.from_learner(learn)
```

```
interp.plot_confusion_matrix()
```



In this case, we made one mistake. There was one black bear classified as grizzly bear. So that's a really good step. We've come a long way. But possibly you could do even better if your dataset was less noisy. Maybe Google image search didn't give you exactly the right images all the time. So how do we fix that? We want to clean it up. So combining human expert with a computer learner is a really good idea. Very very few people publish on this or teach this, but to me, it's the most useful skill, particularly for you. Most of the people watching this are domain experts, not computer science experts, so this is where you can use your knowledge of point mutations in genomics or Panamanian buses or whatever. So let's see how that would work. What I'm going to do is, do you remember the plot top losses from last time where we saw the images which it was either the most wrong about or the least confident about. We are going to look at those and decide which of those are noisy. If you think about it, it's very unlikely that if there is a mislabeled data that it's going to be predicted correctly and with high confidence. That's really unlikely to happen. So we're going to focus on the ones which the model is saying either it's not confident of or it was confident of and it was wrong about. They are the things which might be mislabeled.

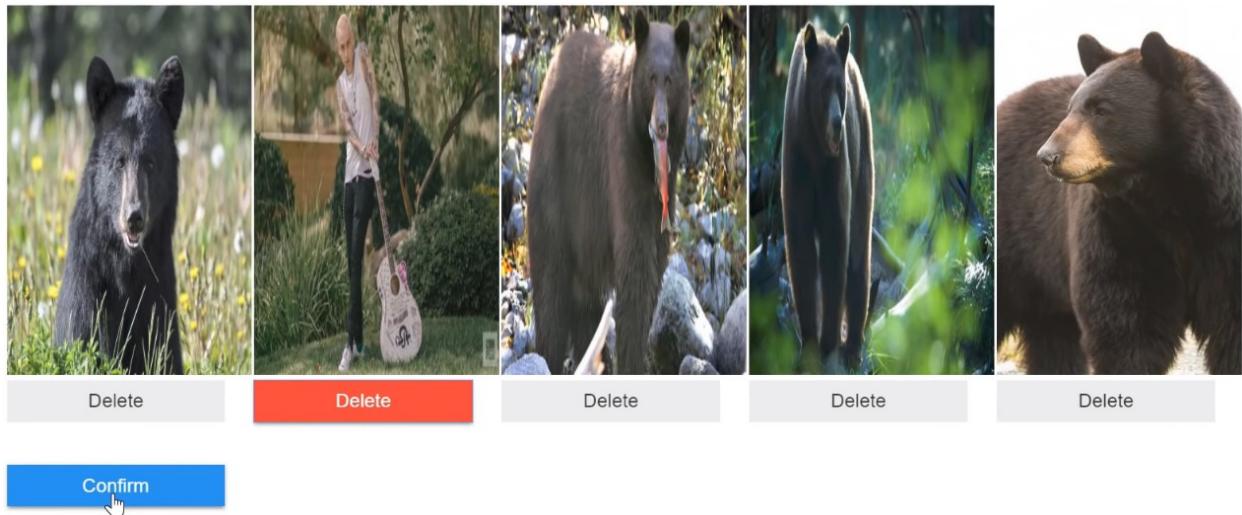
A big shout-out to the San Francisco fastai study group who created this new widget this week called the FileDeleter. Zach, Jason, and Francisco built this thing where we basically can take the top losses from that interpretation object we just created. There is not just `plot_top-losses` but there's also `top_losses` and `top_losses` returns two things: the losses of the things that were the worst and the indexes into the dataset of the things that were the worst. If you don't pass anything at all, it's going to actually return the entire dataset, but sorted so the first things will be the highest losses. Every dataset in fastai has `x` and `y` and the `x` contains the things that are used to, in this case, get the images. So this is the image file names and the `y`'s will be the labels. So if we grab the indexes and pass them into the dataset's `x`, this is going to give us the file names of the dataset ordered by which ones had the highest loss (i.e. which ones it was either confident and wrong about or not confident about). So we can pass that to this new widget that they've created.

Just to clarify, this `top_loss_paths` contains all of the file names in our dataset. When I say "out dataset", this particular one is our validation dataset. So what this is going to do is it's going to clean up mislabeled images or images that shouldn't be there and we're going to remove them from a validation set so that our metrics will be more correct. You then need to rerun these two steps replacing `valid_ds` with `train_ds` to clean up your training set to get the noise out of that as well. So it's a good practice to do both. We'll talk about test sets later as well, if you also have a test set, you would then repeat the same thing.

```
from fastai.widgets import *

losses,idxs = interp.top_losses()
top_loss_paths = data.valid_ds.x[idxs]

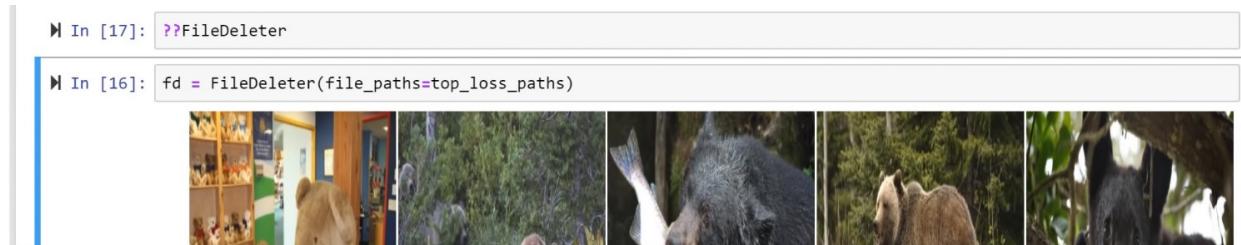
fd = FileDeleter(file_paths=top_loss_paths)
```



So we run FileDeleter passing in that sorted list of paths and so what pops up is basically the same thing as `plot_top_losses`. In other words, these are the ones which is either wrong about or least confident about. So not surprisingly, this one here (the second from left) does not appear to be a teddy bear, black bear, or grizzly bear. So this shouldn't be in our dataset. So what I do is I wack on the delete button, all the rest do look indeed like bears, so I can click confirm and it'll bring up another five.

What I tend to do when I do this is I'll keep going confirm until I get to a coupe of screen full of the things that all look okay and that suggests to me that I've got past the worst bits of the data. So that's it so now you can go back for the training set as well and retrain your model.

I'll just note here that what our San Francisco study group did here was that they actually built a little app inside Jupyter notebook which you might not have realized as possible. But not only is it possible, it's actually surprisingly straightforward. Just like everything else, you can hit double question mark to find out their secrets. So here is the source code.



```
In [17]: ??FileDeleter
In [16]: fd = FileDeleter(file_paths=top_loss_paths)
```

```
Init signature: FileDeleter(file_paths:Collection[Union[pathlib.Path, str]], batch_size:int=5)
Source:
class FileDeleter():
    "Flag images in `file_paths` for deletion and confirm to delete images, showing `batch_size` at a time."
    def __init__(self, file_paths:Collection[PathOrStr], batch_size:int=5):
        self.all_images, self.batch = [], []
        self.batch_size = batch_size
        for fp in [o for o in map(Path,file_paths)] if o.is_file():
            img = self.make_img(fp)
            delete_btn = self.make_button('Delete', file_path=fp, handler=self.on_delete)
            self.all_images.append((img, delete_btn, fp))
        self.render()
```

Really, if you've done any GUI programming before, it'll look incredibly normal. There's basically call backs for what happens when you click on a button where you just do standard Python things and to actually render it, you just use widgets and you can lay it out using standard boxes. So this idea of creating applications inside notebooks is really underused but it's super neat because it lets you create tools for your fellow practitioners or experimenters. And you could definitely envisage taking this a lot further. In fact, by the time you're watching this on the MOOC, you will probably find that there's a whole a lot more buttons here because we've already got a long list of to-do that we're going to add to this particular thing.

I'd love for you to have a think about, now that you know it's possible to write applications in your notebook, what are you going to write and if you google for "[ipywidgets](#)", you can learn about the little GUI framework to find out what kind of widgets you can create, what they look like, and how they work, and so forth. You'll find it's actually a pretty complete GUI programming environment you can play with. And this will all work nice with your models. It's not a great way to productionize an application because it is sitting inside a notebook. This is really for things which are going to help other practitioners or experimentalists. For productionizing things, you need to actually build a production web app which we will look at next.

➲ Putting your model in production [37:36]

After you have cleaned up your noisy images, you can then retrain your model and hopefully you'll find it's a little bit more accurate. One thing you might be interested to discover when you do this is it actually doesn't matter most of the time very much. On the whole, these models are pretty good at dealing with moderate amounts of noisy data. The problem would occur is if your data was not randomly noisy but biased noisy. So I guess the main thing I'm saying is if you go through this process of cleaning up your data and then rerun your model and find it's .001% better, that's normal. It's fine. But it's still a good idea just to make sure that you don't have too much noise in your data in case it is biased.

At this point, we're ready to put our model in production and this is where I hear a lot of people ask me about which mega Google Facebook highly distributed serving system they should use and how do they use a thousand GPUs at the same time. For the vast majority of things you all do, you will want to actually run in production on a CPU, not a GPU. Why is that? Because GPU is good at doing lots of things at the same time, but unless you have a very busy website, it's pretty unlikely that you're going to have 64 images to classify at the same time to put into a batch into a GPU. And if you did, you've got to deal with all that queuing and running it all together, all of your users have to wait until that batch has got filled up and run - it's whole a lot of hassle. Then if you want to scale that, there's another whole lot of hassle. It's much easier if you just wrap one thing, throw it at a CPU to get it done, and comes back again. Yes, it's going to take maybe 10 or 20 times longer so maybe it'll take 0.2 seconds rather than 0.01 seconds. That's about the kind of times we are talking about. But it's so easy to scale. You can chuck it on any standard serving infrastructure. It's going to be cheap, and you can horizontally scale it really easily. So most people I know who are running apps that aren't at Google scale, based on deep learning are using CPUs. And the term we use is "inference". When you are not training a model but you've got a trained model and you're getting it to predict things, we call that inference. That's why we say here:

| You probably want to use CPU for inference

At inference time, you've got your pre-trained model, you saved those weights, and how are you going to use them to create something like Simon Willison's cougar detector?

The first thing you're going to need to know is what were the classes that you trained with. You need to know not just what are they but what were the order. So you will actually need to serialize that or just type them in, or in some way make sure you've got exactly the same classes that you trained with.

```
data.classes
```

```
['black', 'grizzly', 'teddys']
```

If you don't have a GPU on your server, it will use the CPU automatically. If you have a GPU machine and you want to test using a CPU, you can just uncomment this line and that tells fastai that you want to use CPU by passing it back to PyTorch.

```
# fastai.defaults.device = torch.device('cpu')
```

[41:14]

So here is an example. We don't have a cougar detector, we have a teddy bear detector. And my daughter Claire is about to decide whether to cuddle this friend. What she does is she takes daddy's deep learning model and she gets a picture of this and here is a picture that she's uploaded to the web app and here is a picture of the potentially cuddlesome object. We are going to store that in a variable called `img`, and `open_image` is how you open an image in fastai, funnily enough.

```
img = open_image(path/'black'/'00000021.jpg')
img
```



Here is that list of classes that we saved earlier. And as per usual, we created a data bunch, but this time, we're not going to create a data bunch from a folder full of images, we're going to create a special kind of data bunch which is one that's going to grab one single image at a time. So we're not actually passing it any data. The only reason we pass it a path is so that it knows where to load our model from. That's just the path that's the folder that the model is going to be in.

But what we need to do is that we need to pass it the same information that we trained with. So the same transforms, the same size, the same normalization. This is all stuff we'll learn more about. But just make sure it's the same stuff that you used before.

Now you've got a data bunch that actually doesn't have any data in it at all. It's just something that knows how to transform a new image in the same way that you trained with so that you can now do inference.

You can now `create_cnn` with this kind of fake data bunch and again, you would use exactly the same model that you trained with. You can now load in those saved weights. So this is the stuff that you only do once - just once when your web app is starting up. And it takes 0.1 of a second to run this code.

```
classes = ['black', 'grizzly', 'teddys']
data2 = ImageDataBunch.single_from_classes(path, classes, tfms=get_transforms(),
learn = create_cnn(data2, models.resnet34)
learn.load('stage-2')
```

Then you just go `learn.predict(img)` and it's lucky we did that because it's not a teddy bear. This is actually a black bear. So thankfully due to this excellent deep learning model, my daughter will avoid having a very embarrassing black bear cuddle incident.

```
pred_class,pred_idx,outputs = learn.predict(img)
pred_class

'black'
```

So what does this look like in production? I took [Simon Willison's code](#), shamelessly stole it, made it probably a little bit worse, but basically it's going to look something like this. Simon used a really cool web app toolkit called [Starlette](#). If you've ever used Flask, this will look extremely similar but it's kind of a more modern approach - by modern what I really mean is that you can use `await` which is basically means that you can wait for something that takes a while, such as grabbing some data, without using up a process. So for things like I want to get a prediction or I want to load up some data, it's really great to be able to use this modern Python 3 asynchronous stuff. So Starlette could come highly recommended for creating your web app.

You just create a route as per usual, in that you say this is `async` to ensure it doesn't steal the process while it's waiting for things.

You open your image you call `learner.predict` and you return that response. Then you can use Javascript client or whatever to show it. That's it. That's basically the main contents of your web app.

```
@app.route("/classify-url", methods=["GET"])
async def classify_url(request):
    bytes = await get_bytes(request.query_params["url"])
    img = open_image(BytesIO(bytes))
    _, losses = learner.predict(img)
    return JSONResponse({
        "predictions": sorted(
            zip(cat_learner.data.classes, map(float, losses)),
            key=lambda p: p[1],
            reverse=True
        )
    })
```

So give it a go this week. Even if you've never created a web application before, there's a lot of nice little tutorials online and kind of starter code, if in doubt, why don't you try Starlette. There's a free hosting that you can use, there's one called [PythonAnywhere](#), for example. The one Simon has used, [Zeit Now](#), it's something you can basically package it up as a docker thing and shoot it off and it'll serve it up for you. So it doesn't even need to cost you any money and all these classifiers that you're creating, you can turn them into web application. I'll be really interested to see what you're able to make of that. That'll be really fun.

https://course-v3.fast.ai/deployment_zeit.html

⌚ Things that can go wrong [46:06]

I mentioned that most of the time, the kind of rules of thumb I've shown you will probably work. And if you look at the share your work thread, you'll find most of the time, people are posting things saying I downloaded these images, I tried this thing, they worked much better than I expected, well that's cool. Then like 1 out of 20 says I had a problem. So let's have a talk about what happens when you have a problem. This is where we start getting into a little bit of theory because in order to understand why we have these problems and how we fix them, it really helps to know a little bit about what's going on.

First of all, let's look at examples of some problems. The problems basically will be either:

- Your learning rate is too high or low

- Your number of epochs is too high or low

So we are going to learn about what those mean and why they matter. But first of all, because we are experimentalists, let's try them.

⌚ Learning rate (LR) too high

So let's grow with our teddy bear detector and let's make our learning rate really high. The default learning rate is 0.003 that works most of the time. So what if we try a learning rate of 0.5. That's huge. What happens? Our validation loss gets pretty darn high. Remember, this is something that's normally something underneath 1. So if you see your validation loss do that, before we even learn what validation loss is, just know this, if it does that, your learning rate is too high. That's all you need to know. Make it lower. Doesn't matter how many epochs you do. If this happens, there's no way to undo this. You have to go back and create your neural net again and fit from scratch with a lower learning rate.

```
learn = create_cnn(data, models.resnet34, metrics=error_rate)
```

```
learn.fit_one_cycle(1, max_lr=0.5)
```

```
Total time: 00:13
epoch  train_loss  valid_loss  error_rate
1      12.220007  1144188288.000000  0.765957    (00:13)
```

⌚ Learning rate (LR) too low [48:02]

What if we used a learning rate not of 0.003 but 1e-5 (0.00001)?

```
learn = create_cnn(data, models.resnet34, metrics=error_rate)
```

This is just copied and pasted what happened when we trained before with a default learning rate:

```
Total time: 00:57
epoch  train_loss  valid_loss  error_rate
1      1.030236   0.179226   0.028369   (00:14)
2      0.561508   0.055464   0.014184   (00:13)
3      0.396103   0.053801   0.014184   (00:13)
4      0.316883   0.050197   0.021277   (00:15)
```

And within one epoch, we were down to a 2 or 3% error rate.

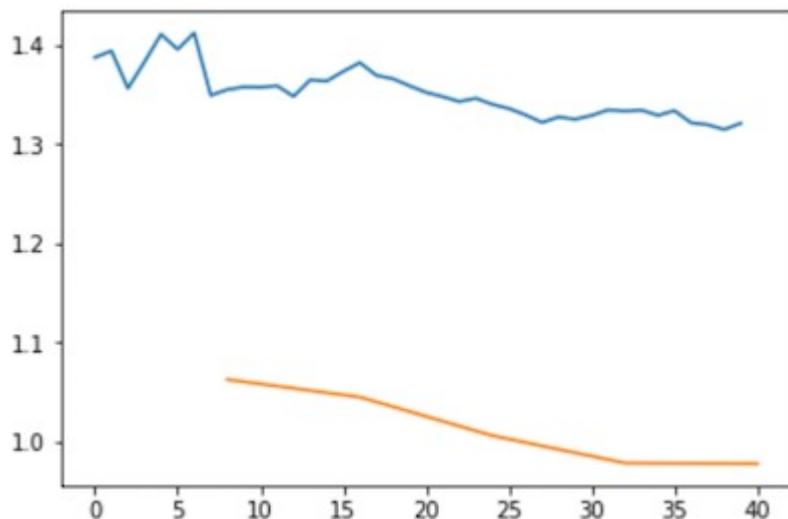
With this really low learning rate, our error rate does get better but very very slowly.

```
learn.fit_one_cycle(5, max_lr=1e-5)
```

```
Total time: 01:07
epoch  train_loss  valid_loss  error_rate
1      1.349151   1.062807   0.609929   (00:13)
2      1.373262   1.045115   0.546099   (00:13)
3      1.346169   1.006288   0.468085   (00:13)
4      1.334486   0.978713   0.453901   (00:13)
5      1.320978   0.978108   0.446809   (00:13)
```

And you can plot it. So `learn.recorder` is an object which is going to keep track of lots of things happening while you train. You can call `plot_losses` to plot out the validation and training loss. And you can just see them gradually going down so slow. If you see that happening, then you have a learning rate which is too small. So bump it by 10 or bump it up by 100 and try again. The other thing you see if your learning rate is too small is that your training loss will be higher than your validation loss. You never want a model where your training loss is higher than your validation loss. That always means you haven't fitted enough which means either your learning rate is too low or your number of epochs is too low. So if you have a model like that, train it some more or train it with a higher learning rate.

```
learn.recorder.plot_losses()
```



As well as taking a really long time, it's getting too many looks at each image, so may overfit.

⌚ Too few epochs [49:42]

What if we train for just one epoch? Our error rate is certainly better than random, 5%. But look at this, the difference between training loss and validation loss - a training loss is much higher than the validation loss. So too few epochs and too lower learning rate look very similar. So you can just try running more epochs and if it's taking forever, you can try a higher learning rate. If you try a higher learning rate and the loss goes off to 100,000 million, then put it back to where it was and try a few more epochs. That's the balance. That's all you care about 99% of the time. And this is only the 1 in 20 times that the defaults don't work for you.

```
learn = create_cnn(data, models.resnet34, metrics=error_rate, pretrained=False)
```

```
learn.fit_one_cycle(1)
```

```
Total time: 00:14
epoch  train_loss  valid_loss  error_rate
1      0.602823   0.119616   0.049645   (00:14)
```

⌚ Too many epochs [50:30]

Too many epochs create something called "overfitting". If you train for too long as we're going to learn about it, it will learn to recognize your particular teddy bears but not teddy bears in general. Here is the thing. Despite what you may have heard, it's very hard to overfit with deep learning. So we were trying today to show you an example of overfitting and I turned off everything. I turned off all the data augmentation, dropout, and weight decay. I tried to make it overfit as much as I can. I trained it on a small-ish learning rate, I trained it for a really long time. And maybe I started to get it to overfit. Maybe.

So the only thing that tells you that you're overfitting is that the error rate improves for a while and then starts getting worse again. You will see a lot of people, even people that claim to understand machine learning, tell you that if your training loss is lower than your validation loss, then you are overfitting. As you will learn today in more detail and during the rest of course, that is **absolutely not true**.

Any model that is trained correctly will always have a lower training loss than validation loss.

That is not a sign of overfitting. That is not a sign you've done something wrong. That is a sign you have done something right. The sign that you're overfitting is that your error starts getting worse, because that's what you care about. You want your model to have a low error. So as long as you're training and your model error is improving, you're not overfitting. How could you be?

```

np.random.seed(42)
data = ImageDataBunch.from_folder(path, train=".", valid_pct=0.9, bs=32,
                                  ds_tfms=get_transforms(do_flip=False, max_rotate=0, max_zoom=1, max_light
                                                        ), size=224, num_workers=4).normalize(imagenet_stats

learn = create_cnn(data, models.resnet50, metrics=error_rate, ps=0, wd=0)
learn.unfreeze()

learn.fit_one_cycle(40, slice(1e-6,1e-4))

```

Total time: 06:39

epoch	train_loss	valid_loss	error_rate
1	1.513021	1.041628	0.507326 (00:13)
2	1.290093	0.994758	0.443223 (00:09)
3	1.185764	0.936145	0.410256 (00:09)
4	1.117229	0.838402	0.322344 (00:09)
5	1.022635	0.734872	0.252747 (00:09)
6	0.951374	0.627288	0.192308 (00:10)
7	0.916111	0.558621	0.184982 (00:09)
8	0.839068	0.503755	0.177656 (00:09)
9	0.749610	0.433475	0.144689 (00:09)
10	0.678583	0.367560	0.124542 (00:09)
11	0.615280	0.327029	0.100733 (00:10)
12	0.558776	0.298989	0.095238 (00:09)
13	0.518109	0.266998	0.084249 (00:09)
14	0.476290	0.257858	0.084249 (00:09)
15	0.436865	0.227299	0.067766 (00:09)
16	0.457189	0.236593	0.078755 (00:10)
17	0.420905	0.240185	0.080586 (00:10)
18	0.395686	0.255465	0.082418 (00:09)
19	0.373232	0.263469	0.080586 (00:09)
20	0.348988	0.258300	0.080586 (00:10)
21	0.324616	0.261346	0.080586 (00:09)
22	0.311310	0.236431	0.071429 (00:09)
23	0.328342	0.245841	0.069597 (00:10)
24	0.306411	0.235111	0.064103 (00:10)
25	0.289134	0.227465	0.069597 (00:09)
26	0.284814	0.226022	0.064103 (00:09)
27	0.268398	0.222791	0.067766 (00:09)
28	0.255431	0.227751	0.073260 (00:10)
29	0.240742	0.235949	0.071429 (00:09)
30	0.227140	0.225221	0.075092 (00:09)
31	0.213877	0.214789	0.069597 (00:09)
32	0.201631	0.209382	0.062271 (00:10)
33	0.189988	0.210684	0.065934 (00:09)
34	0.181293	0.214666	0.073260 (00:09)
35	0.184095	0.222575	0.073260 (00:09)

36	0.194615	0.229198	0.076923	(00:10)
37	0.186165	0.218206	0.075092	(00:09)
38	0.176623	0.207198	0.062271	(00:10)
39	0.166854	0.207256	0.065934	(00:10)
40	0.162692	0.206044	0.062271	(00:09)

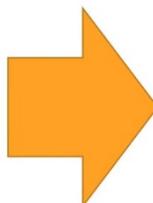
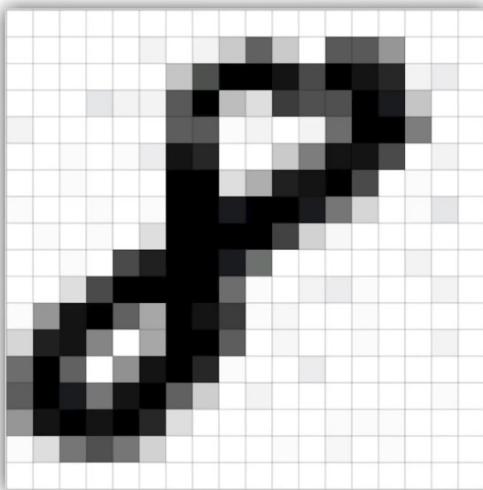
[52:23]

So they are the main four things that can go wrong. There are some other details that we will learn about during the rest of this course but honestly if you stopped listening now (please don't, that would be embarrassing) and you're just like okay I'm going to go and download images, I'm going to create CNNs with resnet32 or resnet50, I'm going to make sure that my learning rate and number of epochs is okay and then I'm going to chuck them up in a Starlette web API, most of the time you are done. At least for computer vision. Hopefully you will stick around because you want to learn about NLP, collaborative filtering, tabular data, and segmentation, etc as well.

[53:10]

Let's now understand what's actually going on. What does "loss" mean? What does "epoch" mean? What does "learning rate" mean? Because for you to really understand these ideas, you need to know what's going on. So we are going to go all the way to the other side. Rather than creating a state of the art cougar detector, we're going to go back and create the simplest possible linear model. So we're going to actually see a little bit of math. But don't be turned off. It's okay. We're going to do a little bit of math but it's going to be totally fine. Even if math is not your thing. Because the first thing we're going to realize is that when we see a picture like this number eight:

PREDICTORS ARE FUNCTIONS OF PIXEL VALUES



Number	Prob
0	0.01
1	0.03
2	0.03
3	0.04
4	0.07
5	0.08
6	0.10
7	0.02
<code>np.argmax</code>	8
9	0.01

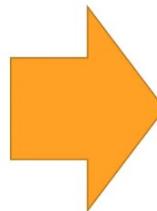


Adam Geitgey [Follow](#)

Interested in computers and machine learning. Likes to write about it.
Jun 13, 2016 · 15 min read

It's actually just a bunch of numbers. For this grayscale one, it's a matrix of numbers. If it was a color image, it would have a third dimension. So when you add an extra dimension, we call it a tensor rather than a matrix. It would be a 3D tensor of numbers - red, green, and blue.

PREDICTORS ARE FUNCTIONS OF PIXEL VALUES



Number	Prob
0	0.01
1	0.03
2	0.03
3	0.04
4	0.07
5	0.08
6	0.10
7	0.02
<i>np.argmax</i> 8	0.59
9	0.01



Adam Geitgey [Follow](#)

Interested in computers and machine learning. Likes to write about it.
Jun 13, 2016 · 15 min read

So when we created that teddy bear detector, what we actually did was we created a mathematical function that took the numbers from the images of the teddy bears and a mathematical function converted those numbers into, in our case, three numbers: a number for the probability that it's a teddy, a probability that it's a grizzly, and the probability that it's a black bear. In this case, there's some hypothetical function that's taking the pixel representing a handwritten digit and returning ten numbers: the probability for each possible outcome (i.e. the numbers from zero to nine).

So what you'll often see in our code and other deep learning code is that you'll find this bunch of probabilities and then you'll find a function called max or argmax attached to it. What that function is doing is, it's saying find the highest number (i.e. probability) and tell me what the index is. So `np.argmax` or `torch.argmax` of the above array would return the index 8.

In fact, let's try it. We know that the function to predict something is called `learn.predict`. So we can chuck two question marks before or after it to get the source code.

In [1]: learn.predict??

And here it is. `pred_max = res.argmax()`. Then what is the class? We just pass that into the classes array. So you should find that the source code in the fastai library can both strengthen your understanding of the concepts and make sure that you know what's going on and really help you here.

```
Signature: learn.predict(img:fastai.vision.image.Image)
Source:
def predict(self, img:Image):
    "Return prect class, label and probabilities for `img`."
    ds = self.data.valid_ds
    ds.set_item(img)
    res = self.pred_batch()
    ds.clear_item()
    pred_max = res.argmax()
    return self.data.classes[pred_max],pred_max,res
File:      /datalab/jhoward/git/fastai/fastai/vision/learner.py
Type:      method
```

Question: Can we have a definition of the error rate being discussed and how it is calculated? I assume it's cross validation error [56:38].

Sure. So one way to answer the question of how is error rate calculated would be to type `error_rate??` and look at the source code, and it's $1 - \text{accuracy}$.

```
In [10]: error_rate??
```

Signature: error_rate(input:torch.Tensor, targs:torch.Tensor) -> <function NewType.<locals>.new_type at 0x7fdf944d11e0>
Source:

```
def error_rate(input:Tensor, targs:Tensor)->Rank0Tensor:  
    "1 - `accuracy`"  
    return 1-accuracy(input, targs)
```

File: /data1/jhoward/git/fastai/fastai/metrics.py
Type: function

So then a question might be what is accuracy:

```
Signature: accuracy(input:torch.Tensor, targs:torch.Tensor) -> <function NewType.<locals>.new_type at 0x7fdf944d11e0>
Source:
def accuracy(input:Tensor, targs:Tensor)->Rank0Tensor:
    "Compute accuracy with `targs` when `input` is bs * n_classes."
    n = targs.shape[0]
    input = input.argmax(dim=1).view(n,-1)
    targs = targs.view(n,-1)
    return (input==targs).float().mean()
File:      /data1/jhoward/git/fastai/fastai/metrics.py
Type:     function
```

It is argmax. So we now know that means find out which particular thing it is, and then look at how often that equals the target (i.e. the actual value) and take the mean. So that's basically what it is. So then the question is, okay, what does that being applied to and always in fastai, metrics (i.e. the things that we pass in) are always going to be applied to the validation set. Any time you put a metric here, it'll be applied to the validation set because that's your best practice:

```
| In [ ]: learn = create_cnn(data, models.resnet50, metrics=error_rate, ps=0, wd=0)
          learn.unfreeze()

| In [ ]: learn.fit_one_cycle(40, slice(1e-6,1e-4))

Total time: 06:39
epoch  train_loss  valid_loss  error_rate
  1      1.513021    1.041628    0.507326  (00:13)
  2      1.290093    0.994758    0.443223  (00:09)
  3      1.185764    0.936145    0.410256  (00:09)
  4      1.087500    0.886116    0.386116  (00:09)
```

That's what you always want to do is make sure that you're checking your performance on data that your model hasn't seen, and we'll be learning more about the validation set shortly.

Remember, you can also type `doc` if the source code is not what you want which might well not be, you actually want the documentation, that will both give you a summary of the types in and out of the function and a link to the full documentation where you can find out all about how metrics work and what other metrics there are and so forth. Generally speaking, you'll also find links to more information where, for example, you will find complete run through and sample code showing you how to use all these things. So don't forget that the `doc` function is your friend. Also both in the `doc` function and in documentation, you'll see a source link. This is like `??` but what the source link does is it takes you into the exact line of code in Github. So you can see exactly how that's implemented and what else is around it. So lots of good stuff there.

Question: Why were you using `3e` for your learning rates earlier? With `3e-5` and `3e-4` [59:11]?

We found that `3e-3` is just a really good default learning rate. It works most of the time for your initial fine-tuning before you unfreeze. And then, I tend to kind of just multiply from there. So then the next stage, I will pick 10 times lower than that for the second part of the slice, and whatever the LR finder found for the first part of the slice. The second part of the slice doesn't come from the LR finder. It's just a rule of thumb which is 10 times less than your first part which defaults to `3e-3`, and then the first part of the slice is what comes out of the LR finder. We'll be learning a lot more about these learning rate details both today and in the coming lessons. But for now, all you need to remember is that your basic approach looks like this:

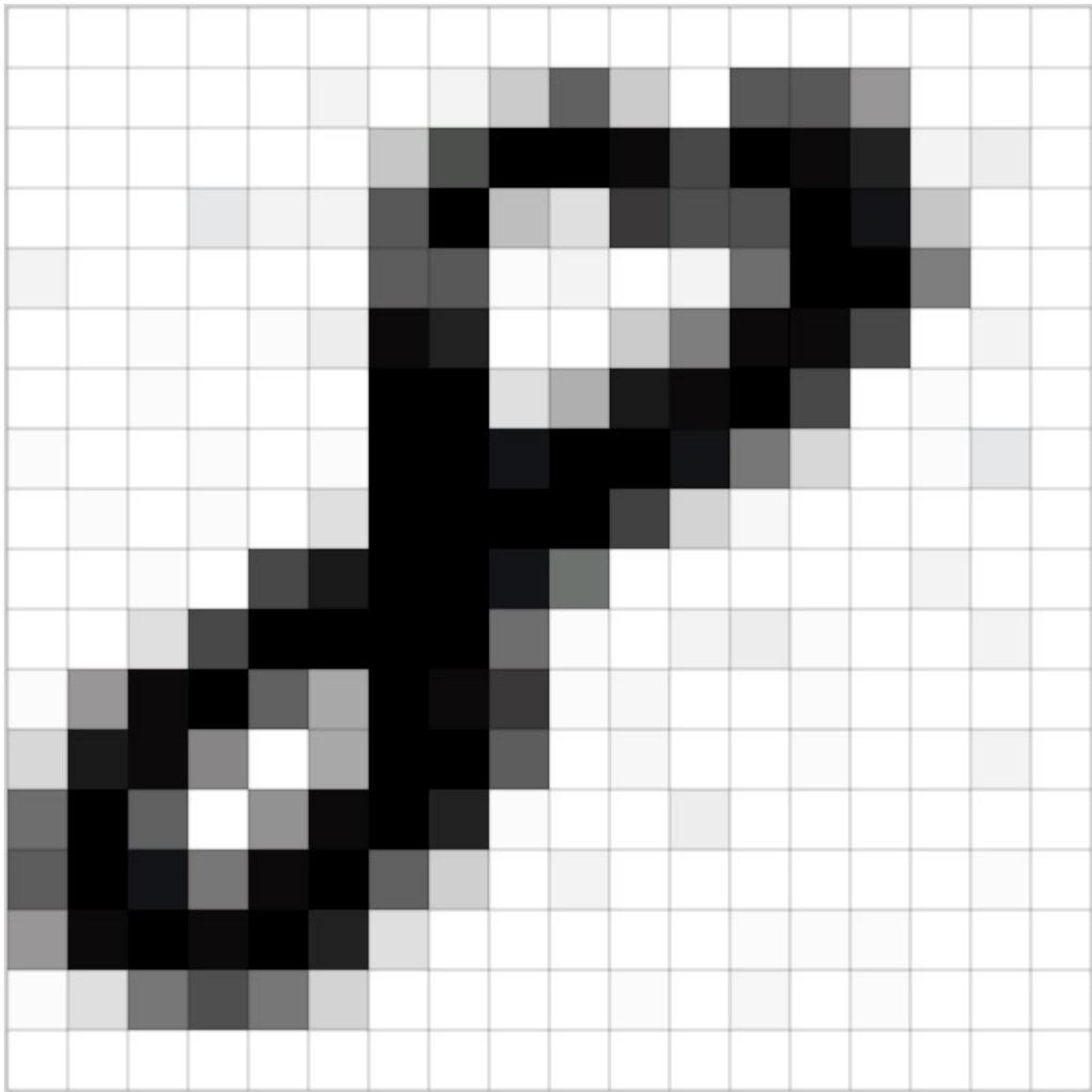
- `learn.fit_one_cycle`
 - Some number of epochs, I often pick 4
 - Some learning rate which defaults to `3e-3`. I'll just type it up fully so you can see.
- Then we do that for a bit and then we unfreeze it.
- Then we learn some more and so this is a bit where I just take whatever I did last time and divide it by 10. Then I also write like that (`slice`) then I have to put one more number in here and that's the number I get from the learning rate finder - a bit where it's got the strongest slope.

```
learn.fit_one_cycle(4, 3e-3)
learn.unfreeze()
learn.fit_one_cycle(4, slice(xxx, 3e-4))
```

So that's kind of don't have to think about it, don't really have to know what's going on rule of thumb that works most of the time. But let's now dig in and actually understand it more completely.

⌚ Digging in and looking at the math [1:01:17]

We're going to create this mathematical function that takes the numbers that represent the pixels and spits out probabilities for each possible class.



By the way, a lot of the stuff that we're using here, we are stealing from other people who are awesome and so we're putting their details here. So please check out their work because they've got great work that we are highlighting in our course. I really like this idea of this little animated gif of the numbers, so thank you to [Adam Geitgey](#) for creating that.

[1:02:05]

Let's look and see how we create one of these functions, and let's start with the simplest functions I know:

$$y = ax + b$$

That's a line where

- a: gradient of the line
- b: the intercept of the line

Hopefully when we said that you need to know high school math to do this course, these are the things we are assuming that you remember. If we do mention some math thing which I am assuming you remember and you don't remember it, don't freak out. It happens to all of us. [Khan Academy](#) is actually terrific. It's not just for school kids. Go to Khan Academy, find the concept you need a refresher on, and he explains things really well. So strongly recommend checking that out. Remember, I'm just a philosophy student, so all the time I'm trying to either remind myself about something or I never learnt something. So we have the whole internet to teach us these things.

I'm going to re-write this slightly:

$$y = a_1x + a_2$$

So let's just replace b with a a_2 , just give it a different name. So there's another way of saying the same thing. Then another way of saying that would be if I could multiply a_2 by the number 1.

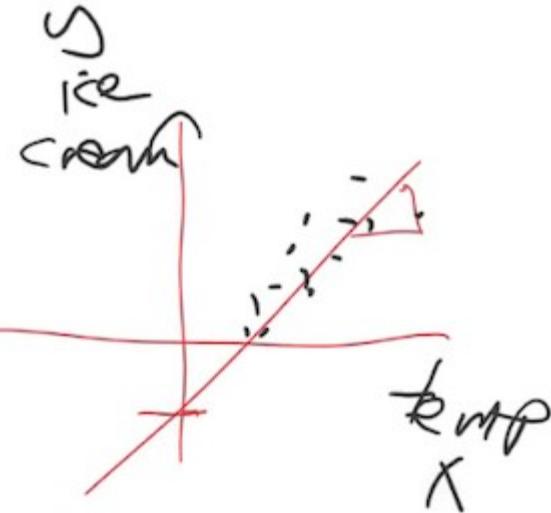
$$y = a_1x + a_2 \cdot 1$$

This still is the same thing. Now at this point, I'm actually going to say let's not put the number 1 there, but put an x_1 here and x_2 here:

$$y = a_1x_1 + a_2x_2$$

$$x_2 = 1$$

So far, this is pretty early high school math. This is multiplying by 1 which I think we can handle. So this and $y = ax + b$ are equivalent with a bit of renaming. Now in machine learning, we don't just have one equation, we've got lots. So if we've got some data that represents the temperature versus the number of ice creams sold, then we have lots of dots.



So each one of those dots, we might hypothesize is based on this formula ($y = a_1x_1 + a_2x_2$). And basically there's lots of values of y and lots of values of x so we can stick little i here:

$$y_i = ax_i + b$$

The way we do that is a lot like numpy indexing, but rather than things in square brackets, we put them down here in the subscript in our equation:

$$y_i = a_1x_{i,1} + a_2x_{i,2}$$

So this is now saying there's actually lots of these different y_i 's based on lots of different $x_{i,1}$ and $x_{i,2}$ but notice there is still one of each of these (a_1, a_2). They called the coefficients or the parameters. So this is our linear equation and we are still going to say that every $x_{i,2}$ is equal to 1. Why did I do it that way? Because I want to do linear algebra? Why do I want to do linear algebra? One reason is because [Rachel teaches the world's best linear algebra course](#), so if you're interested, check it out. So it's a good opportunity for me to throw in a pitch for this which we make no money but never mind. But more to the point right now, it's going to make life much easier. Because I hate writing loops, I hate writing code, I just want the computer to do everything for me. And anytime you see this little i subscripts, that sounds like you're going to have to do loops and all kind of stuff. But what you might remember from school is that when you've got two things being multiplied together, two things being multiplied together, then they get added up, that's called a "dot product". If you do that for lots and lots of different numbers i , then that's called a matrix product. So in fact, this whole thing can be written like this:

$$\vec{y} = X\vec{a}$$

Rather than lots of different y_i 's, we can say there's one vector called \vec{y} which is equal to one matrix called X times one vector called \vec{a} . At this point, I know a lot of you don't remember that. That's fine. We have a picture to show you.

Andre Stoltz created this fantastic called <http://matrixmultiplication.xyz/> and here we have a matrix by a vector, and we are going to do a matrix vector product.

Matrix Multiplication

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 0 \\ 2 & 3 & 4 \end{bmatrix} \times \begin{bmatrix} 2 \\ 6 \\ 1 \end{bmatrix}$$

- + - +

 Multiply

That is what matrix vector multiplication does. In other words, it's just $y_i = a_1x_{i,1} + a_2x_{i,2}$ except his version is much less messy.

Question: When generating new image dataset, how do you know how many images are enough? What are ways to measure "enough"? [1:08:35]

Great question. Another possible problem you have is you don't have enough data. How do you know if you don't have enough data? Because you found a good learning rate (i.e. if you make it higher than it goes off into massive losses; if you make it lower, it goes really slowly) and then you train for such a long time that your error starts getting worse. So you know that you trained for long enough. And you're still not happy with the accuracy - it's not good enough for the teddy bear cuddling level of safety you want. So if that happens, there's a number of things you can do and we'll learn pretty much all of them during this course but one of the easiest one is get more data. If you get more data, then you can train for longer, get a higher accuracy, lower error rate, without overfitting.

Unfortunately there is no shortcut. I wish there was. I wish there's some way to know ahead of time how much data you need. But I will say this - most of the time, you need less data than you think. So organizations very commonly spend too much time gathering data, getting more data than it turned out they actually needed. So get a small amount first and see how you go.

Question: What do you do if you have unbalanced classes such as 200 grizzly and 50 teddy? [1:10:00]

Nothing. Try it. It works. A lot of people ask this question about how do I deal with unbalanced data. I've done lots of analysis with unbalanced data over the last couple of years and I just can't make it not work. It always works. There's actually a paper that said if you want to get it slightly better then the best thing to do is to take that uncommon class and just make a few copies of it. That's called "oversampling" but I haven't found a situation in practice where I needed to do that. I've found it always just works fine, for me.

Question: Once you unfreeze and retrain with one cycle again, if your training loss is still higher than your validation loss (likely underfitting), do you retrain it unfrozen again (which will technically be more than one cycle) or you redo everything with longer epoch per the cycle? [1:10:47]

You guys asked me that last week. My answer is still the same. I don't know. Either is fine. If you do another cycle, then it'll maybe generalize a little bit better. If you start again, do twice as long, it's kind of annoying, depends how patient you are. It won't make much difference. For me personally, I normally just train a few more cycles. But it doesn't make much difference most of the time.

Question: Question about this code example:

```
classes = ['black', 'grizzly', 'teddys']
data2 = ImageDataBunch.single_from_classes(path, classes,
tfms=get_transforms(), size=224).normalize(imagenet_stats)
learn = create_cnn(data2, models.resnet34)
learn.load('stage-2')
```

This requires `models.resnet34` which I find surprising: I had assumed that the model created by `.save(...)` (which is about 85MB on disk) would be able to run without also needing a copy of `resnet34`. [1:11:37]

We're going to be learning all about this shortly. There is no "copy of ResNet34", ResNet34 is what we call "architecture" - it's a functional form. Just like $y = ax + b$ is a linear functional form. It doesn't take up any room, it doesn't contain anything, it's just a function. ResNet34 is just a function. I think the confusion here is that we often use a pre-trained neural net that's been learnt on ImageNet. In this case, we don't need to use a pre-trained neural net. Actually, to avoid that even getting created, you can actually pass `pretrained=False`:

```
learn = create_cnn(data, models.resnet34, metrics=error_rate, pretrained=False)
```

That'll ensure that nothing even gets loaded which will save you another 0.2 seconds, I guess. But we'll be learning a lot more about this. So don't worry if this is a bit unclear. The basic idea is `models.resnet34` above is basically the equivalent of saying is it a line or is it a quadratic or is it a reciprocal - this is just a function. This is a ResNet34 function. It's a mathematical function. It doesn't take any storage, it doesn't have any numbers, it doesn't have to be loaded as opposed to a pre-trained model. When we did it at the inference time, the thing that took space is this bit:



```
In [ ]: classes = ['black', 'grizzly', 'teddys']
data2 = ImageDataBunch.single_from_classes(path, classes, tfms=get_transforms(), size=224).normalize(imagenet_stats)
learn = create_cnn(data2, models.resnet34)
learn.load('stage-2')
```

Which is where we load our parameters. It is basically saying, as we are about find out, what are the values of a and b - we have to store these numbers. But for ResNet 34, you don't just store 2 numbers, you store a few million or few tens of millions of numbers.

[1:14:13]

So why did we all this? It's because I wanted to be able to write it out like this: $\vec{y} = X\vec{a}$ and the reason I wanted to be able to like this is that we can now do that in PyTorch with no loops, single line of code, and it's also going to run faster.

PyTorch really doesn't like loops

It really wants you to send it a whole equation to do all at once. Which means, you really want to try and specify things in these kind of linear algebra ways. So let's go and take a look because what we're going to try and do then is we're going to try and take this $\vec{y} = X\vec{a}$ (we're going to call this an architecture). It's the world's tiniest neural network. It's got two parameters a_1 and a_2 . We are going to try and fit this architecture to some data.

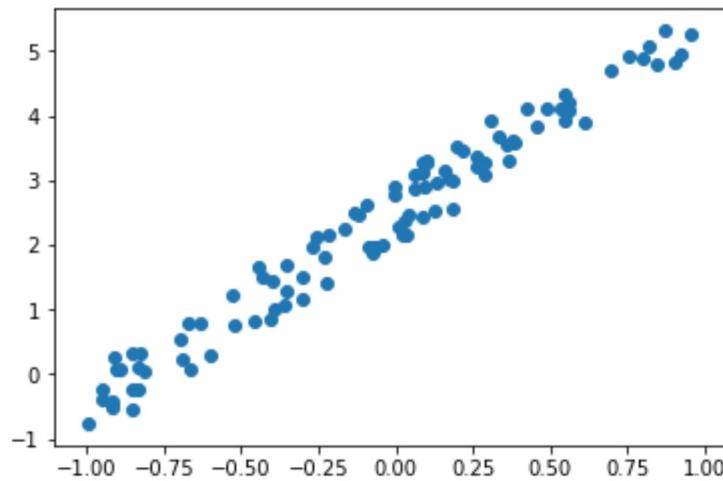
SGD [1:15:06]

So let's jump into a notebook and generate some dots, and see if we can get it to fit a line somehow. And the "somehow" is going to be using something called SGD. What is SGD? Well, there's two types of SGD. The first one is where I said in lesson 1 "hey you should all try building these models and try and come up with something cool" and you guys all experimented and found really good stuff. So that's where the S would be Student. That would be Student Gradient Descent. So that's version one of SGD.

Version two of SGD which is what I'm going to talk about today is where we are going to have a computer try lots of things and try and come up with a really good function and that would be called Stochastic Gradient Descent. The other one that you hear a lot on Twitter is Stochastic Gradient Descent.

↪ Linear Regression problem [1:16:08]

We are going to jump into [lesson2-sgd.ipynb](#). We are going to go bottom-up rather than top-down. We are going to create the simplest possible model we can which is going to be a linear model. And the first thing we need is we need some data. So we are going to generate some data. The data we're going to generate looks like this:



So x-axis might represent temperature, y-axis might represent number of ice creams we sell, or something like that. But we're just going to create some synthetic data that we know is following a line. As we build this, we're actually going to learn a little bit about PyTorch as well.

```
%matplotlib inline
from fastai import *

n=100

x = torch.ones(n,2)
x[:,0].uniform_(-1.,1)
x[:5]

tensor([[-0.1338,  1.0000],
       [-0.4062,  1.0000],
       [-0.3621,  1.0000],
       [ 0.4551,  1.0000],
       [-0.8161,  1.0000]])
```

Basically the way we're going to generate this data is by creating some coefficients. a_1 will be 3 and a_2 will be 2. We are going to create a column of numbers for our x 's and a whole bunch of 1's.

```
a = tensor(3., 2); a
```

```
tensor([3., 2.])
```

And then we're going to do this $x@a$. What is $x@a$? $x@a$ in Python means a matrix product between x and a . And it actually is even more general than that. It can be a vector vector product, a matrix vector product, a vector matrix product, or a matrix matrix product. Then actually in PyTorch, specifically, it can mean even more general things where we get into higher rank tensors which we will learn all about very soon. But this is basically the key thing that's going to go on in all of our deep learning. The vast majority of the time, our computers are going to be basically doing this - multiplying numbers together and adding them up which is the surprisingly useful thing to do.

```
y = x@a + torch.rand(n)
```

[1:17:57]

So we basically are going to generate some data by creating a line and then we're going to add some random numbers to it. But let's go back and see how we created x and a . I mentioned that we've basically got these two coefficients 3 and 2. And you'll see that we've wrapped it in this function called `tensor`. You might have heard this word "tensor" before. It's one of these words that sounds scary and apparently if you're a physicist, it actually is scary. But in the world of deep learning, it's actually not scary at all. "Tensor" means array, but specifically it's an array of a regular shape. So it's not an array where row 1 has two things, row 3 has three things, and row 4 has one thing, what you call a "jagged array". That's not a tensor. A tensor is any array which has a rectangular or cube or whatever - a shape where every row is the same length and every column is the same length. The following are all tensors:

- 4 by 3 matrix
- A vector of length 4
- A 3D array of length 3 by 4 by 6

That's all tensor is. We have these all the time. For example, an image is a 3 dimensional tensor. It's got number of rows by number of columns by number of channels (normally red, green, blue). So for example, VGA picture could be 640 by 480 by 3 or actually we do things backwards so when people talk about images, they normally go width by height, but when we talk mathematically, we always go a number of rows by number of columns, so it would actually be 480 by 640 by 3 that will catch you out. We don't say dimensions, though, with tensors. We use one of two words, we either say rank or axis. Rank specifically means how many axes are there, how many dimensions are there. So an image is generally a rank 3 tensor. What we created here is a rank 1 tensor (also known as a vector). But in math, people come up with very different words for slightly different concepts. Why is a one dimensional array a vector and a two dimensional array is a matrix, and a three dimensional array doesn't have a name. It doesn't make any sense. With computers, we try to have some simple consistent naming conventions. They are all called tensors - rank 1 tensor, rank 2 tensor, rank 3 tensor. You can certainly have a rank 4 tensor. If you've got 64 images, then that would be a rank 4 tensor of 64 by 480 by 640 by 3. So tensors are very simple. They just mean arrays.

In PyTorch, you say `tensor` and you pass in some numbers, and you get back, which in this case just a list, a vector. This then represents our coefficients: the slope and the intercept of our line.

```
a = tensor(3.,2); a  
tensor([3., 2.])
```

Because we are not actually going to have a special case of $ax + b$, instead, we are going to say there's always this second x value which is always 1

$$y_i = a_1 x_{i,1} + a_2 x_{i,2}$$

You can see it here, always 1 which allows us just to do a simple matrix vector product:

```
x = torch.ones(n,2)  
x[:,0].uniform_(-1.,1)  
x[:5]  
  
tensor([[-0.1338,  1.0000],  
       [-0.4062,  1.0000],  
       [-0.3621,  1.0000],  
       [ 0.4551,  1.0000],  
       [-0.8161,  1.0000]])
```

So that's `a`. Then we wanted to generate this `x` array of data. We're going to put random numbers in the first column and a whole bunch of 1's in the second column. To do that, we say to PyTorch that we want to create a rank 2 tensor of `n` by 2. Since we passed in a total of 2 things, we get a rank 2 tensor. The number of rows will be `n` and the number of columns will be 2. In there, every single thing in it will be a 1 - that's what `torch.ones` means.

[1:22:45]

Then this is really important. You can index into that just like you can index into a list in Python. But you can put a colon anywhere and a colon means every single value on that axis/dimension. This here `x[:,0]` means every single row of column 0. So `x[:,0].uniform_(-1.,1)` is every row of column 0, I want you to grab a uniform random numbers.

Here is another very important concept in PyTorch. Anytime you've got a function that ends with an underscore, it means don't return to me that uniform random number, but replace whatever this is being called on with the result of this function. So this `x[:,0].uniform_(-1.,1)` takes column 0 and replaces it with a uniform random number between -1 and 1. So there's a lot to unpack there.

```
x = torch.ones(n,2)
x[:,0].uniform_(-1.,1)
x[:5]

tensor([[-0.1338,  1.0000],
       [-0.4062,  1.0000],
       [-0.3621,  1.0000],
       [ 0.4551,  1.0000],
       [-0.8161,  1.0000]])
```

But the good news is these two lines of code and `x@a` which we are coming to cover 95% of what you need to know about PyTorch.

1. How to create an array
2. How to change things in an array
3. How to do matrix operations on an array

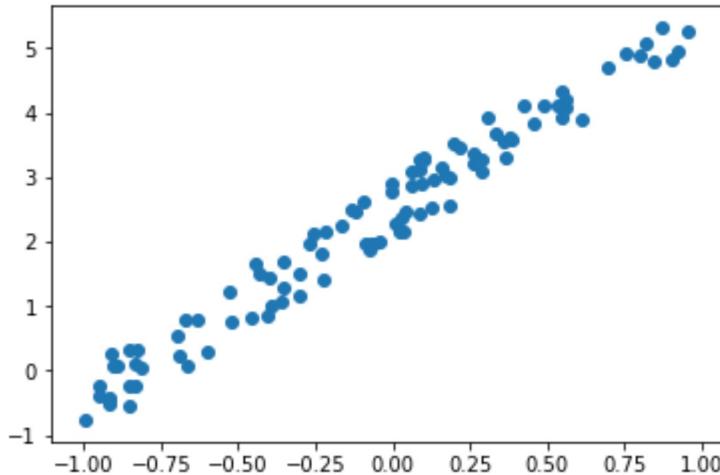
So there's a lot to unpack, but these small number of concepts are incredibly powerful. So I can now print out the first five rows. `[:5]` is a standard Python slicing syntax to say the first 5 rows. So here are the first 5 rows, 2 columns looking like - my random numbers and my 1's.

Now I can do a matrix product of that `x` by my `a`, add in some random numbers to add a bit of noise.

```
y = x@a + torch.rand(n)
```

Then I can do a scatter plot. I'm not really interested in my scatter plot in this column of ones. They are just there to make my linear function more convenient. So I'm just going to plot my zero index column against my `y`'s.

```
plt.scatter(x[:,0], y);
```



`plt` is what we universally use to refer to the plotting library, `matplotlib`. That's what most people use for most of their plotting in scientific Python. It's certainly a library you'll want to get familiar with because being able to plot things is really important. There are lots of other plotting packages. Lots of the other packages are better at certain things than `matplotlib`, but `matplotlib` can do everything reasonably well. Sometimes it's a little awkward, but for me, I do pretty much everything in `matplotlib` because there is really nothing it can't do even though some libraries can do other things a little bit better or prettier. But it's really powerful so once you know `matplotlib`, you can do everything. So here, I'm asking `matplotlib` to give me a scatterplot with my `x`'s against my `y`'s. So this is my dummy data representing temperature and ice cream sales.

[1:26:18]

Now what we're going to do is, we are going to pretend we were given this data and we don't know that the values of our coefficients are 3 and 2. So we're going to pretend that we never knew that and we have to figure them out. How would we figure them out? How would we draw a line to fit this data and why would that even be interesting? Well, we're going to look at more about why it's interesting in just a moment. But the basic idea is:

If we can find a way to find those two parameters to fit that line to those 100 points, we can also fit these arbitrary functions that convert from pixel values to probabilities.

It will turn out that these techniques that we're going to learn to find these two numbers works equally well for the 50 million numbers in ResNet34. So we're actually going to use an almost identical approach. This is the bit that I found in previous classes people have the most trouble digesting. I often find, even after week 4 or week 5, people will come up to me and say:

Student: I don't get it. How do we actually train these models?

Jeremy: It's SGD. It's that thing we saw in the notebook with the 2 numbers.

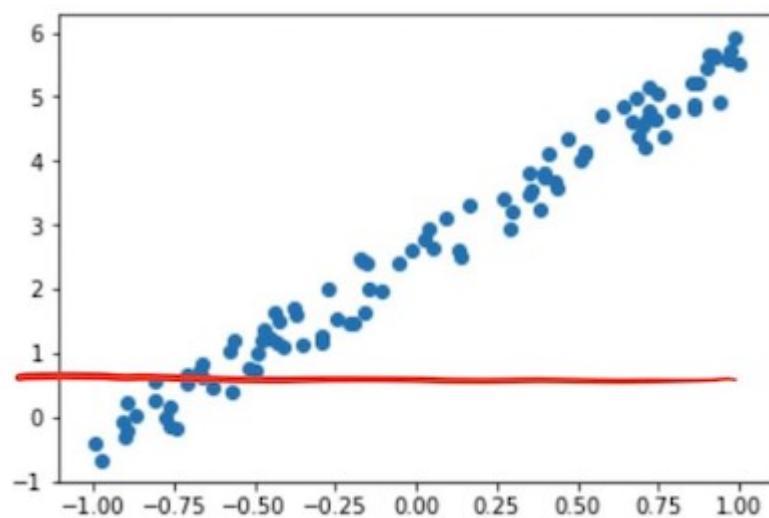
Student: yeah, but... but we are fitting a neural network.

Jeremy: I know and we can't print the 50 million numbers anymore, but it's literally identically doing the same thing.

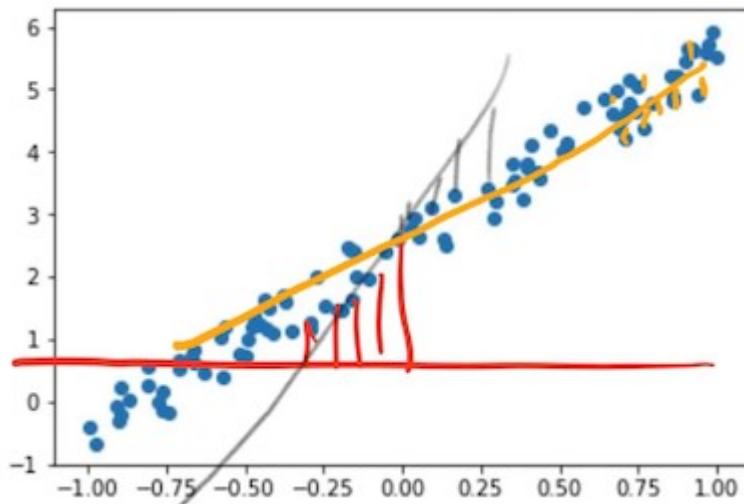
The reason this is hard to digest is that the human brain has a lot of trouble conceptualizing of what an equation with 50 million numbers looks like and can do. So for now, you'll have to take my word for it. It can do things like recognize teddy bears. All these functions turn out to be very powerful. We're going to learn about how to make them extra powerful. But for now, this thing we're going to learn to fit these two numbers is the same thing that we've just been using to fit 50 million numbers.

⌚ Loss function [1:28:36]

We want to find what PyTorch calls **parameters**, or in statistics, you'll often hear it called coefficient (i.e. these values of a_1 and a_2). We want to find these parameters such that the line that they create minimizes the error between that line and the points. In other words, if the a_1 and a_2 we came up with resulted in this line:



Then we'd look and we'd see how far away is that line from each point. That's quite a long way. So maybe there was some other a_1 and a_2 which resulted in the gray line. And they would say how far away is each of those points. And then eventually we come up with the yellow line. In this case, each of those is actually very close.



So you can see how in each case we can say how far away is the line at each spot away from its point, and then we can take the average of all those. That's called the **loss**. That is the value of our loss. So you need a mathematical function that can basically say how far away is this line from those points.

For this kind of problem which is called a regression problem (a problem where your dependent variable is continuous, so rather than being grizzlies or teddies, it's some number between -1 and 6), the most common loss function is called mean squared error which pretty much everybody calls MSE. You may also see RMSE which is root mean squared error. The mean squared error is a loss which is the difference between some predictions that you made which is like the value of the line and the actual number of ice cream sales. In the mathematics of this, people normally refer to the actual as y and the prediction, they normally call it \hat{y} (y hat).

When writing something like mean squared error equation, there is no point writing "ice cream" and "temperature" because we want it to apply to anything. So we tend to use these mathematical placeholders.

So the value of mean squared error is simply the difference between those two ($y_{\text{hat}} - y$) squared. Then we can take the mean because both y_{hat} and y are rank 1 tensors, so we subtract one vector from another vector, it does something called "element-wise arithmetic" in other words, it subtracts each one from each other, so we end up with a vector of differences. Then if we take the square of that, it squares everything in that vector. So then we can take the mean of that to find the average square of the differences between the actuals and the predictions.

```
def mse(y_hat, y): return ((y_hat-y)**2).mean()
```

If you're more comfortable with mathematical notation, what we just wrote was:

$$\frac{\sum(\hat{y} - y)^2}{n}$$

One of the things I'll note here is, I don't think $((y_{\hat{}} - y)^2).mean()$ is more complicated or unwieldy than $\frac{\sum(\hat{y} - y)^2}{n}$ but the benefit of the code is you can experiment with it. Once you've defined it, you can use it, you can send things into it, get stuff out of it, and see how it works. So for me, most of the time, I prefer to explain things with code rather than with math. Because they are the same, just different notations. But one of the notations is executable. It's something you can experiment with. And the other is abstract. That's why I'm generally going to show code.

So the good news is, if you're a coder with not much of a math background, actually you do have a math background. Because code is math. If you've got more of a math background and less of a code background, then actually a lot of the stuff that you learned from math is going to translate directly into code and now you can start to experiment with your math.

[1:34:03]

`mse` is a loss function. This is something that tells us how good our line is. Now we have to come up with what is the line that fits through here. Remember, we are going to pretend we don't know. So what you actually have to do is you have to guess. You actually have to come up with a guess what are the values of a_1 and a_2 . So let's say we guess that a_1 and a_2 are -1 and 1.

```
a = tensor(-1., 1)
```

Here is how you create that tensor and I wanted to write it this way because you'll see this all the time. Written out fully, it would be `tensor(-1.0, 1.0)`. We can't write it without the point because `tensor(-1, 1)` is now an int, not a floating point. So that's going to spit the dummy (Australian for "behave in a bad-tempered or petulant way") if you try to do calculations with that in neural nets.

I'm far too lazy to type `.0` every time. Python knows perfectly well that if you added `.` next to any of these numbers, then the whole thing is now floats. So that's why you'll often see it written this way, particularly by lazy people like me.

So `a` is a tensor. You can see it's floating-point. You see, even PyTorch is lazy. They just put a dot. They don't bother with a zero.

```
a
```

```
tensor([-1., 1.])
```

But if you want to actually see exactly what it is, you can write `.type()` and you can see it's a `FloatTensor`:

```
a.type()  
'torch.FloatTensor'
```

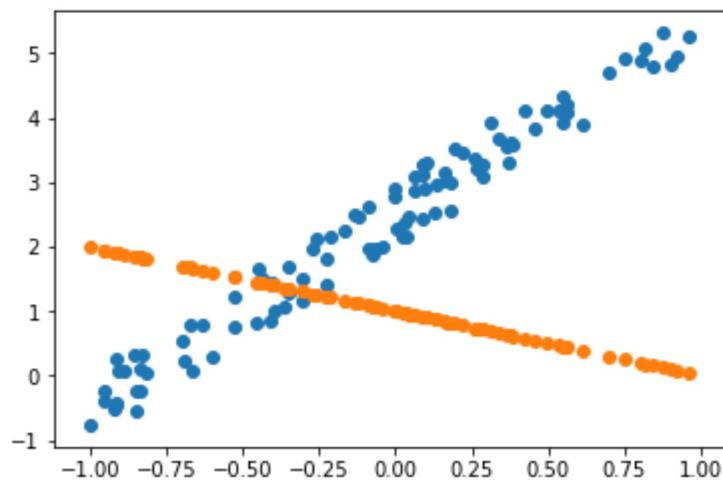
So now we can calculate our predictions with this random guess. $x@a$ a matrix product of x and a . And we can now calculate the mean squared error of our predictions and their actuals, and that's our loss. So for this regression, our loss is 0.9.

```
y_hat = x@a  
mse(y_hat, y)
```

```
tensor(8.8945)
```

So we can now plot a scatter plot of x against y and we can plot the scatter plot of x against y_{hat} . And there they are.

```
plt.scatter(x[:,0],y)  
plt.scatter(x[:,0],y_hat);
```

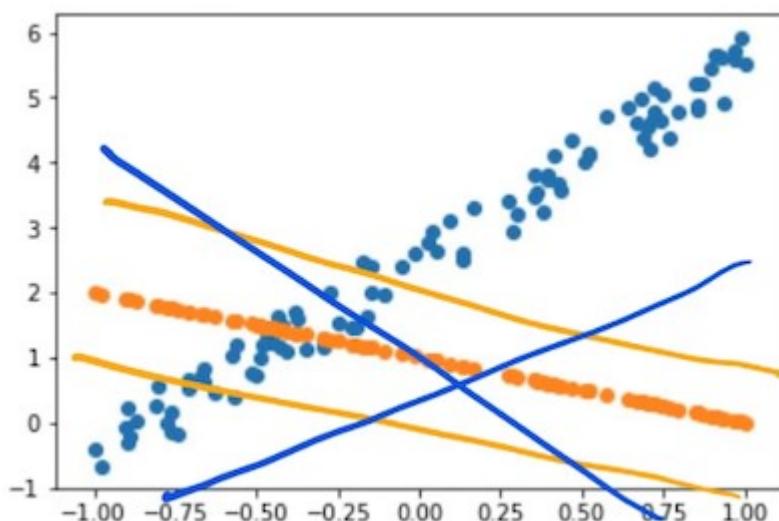


So that is not great - not surprising. It's just a guess. So SGD or gradient descent more generally and anybody who's done engineering or probably computer science at school would have done plenty of this like Newton's method, etc at university. If you didn't, don't worry. We're going to learn it now.

It's basically about taking this guess and trying to make it a little bit better. How do we make it a little better? Well, there are only two numbers and the two numbers are the two numbers are the intercept of the orange line and the gradient of the orange line. So what we are going to do with gradient descent is we're going to simply say:

- What if we changed those two numbers a little bit?
 - What if we made the intercept a little bit higher or a little bit lower?
 - What if we made the gradient a little bit more positive or a little bit more

negative?



There are 4 possibilities and then we can calculate the loss for each of those 4 possibilities and see what works. Did lifting it up or down make it better? Did tilting it more positive or more negative make it better? And then all we do is we say, okay, whichever one of those made it better, that's what we're going to do. That's it.

But here is the cool thing for those of you that remember calculus. You don't actually have to move it up and down, and round about. You can actually calculate the derivative. The derivative is the thing that tells you would moving it up or down make it better, or would rotating it this way or that way make it better. The good news is if you didn't do calculus or you don't remember calculus, I just told you everything you need to know about it. It tells you how changing one thing changes the function. That's what the derivative is, kind of, not quite strictly speaking, but close enough, also called the gradient. The gradient or the derivative tells you how changing a_1 up or down would change our MSE, how changing a_2 up or down would change our MSE, and this does it more quickly than actually moving it up and down.

In school, unfortunately, they forced us to sit there and calculate these derivatives by hand. We have computers. Computers can do that for us. We are not going to calculate them by hand.

```
a = nn.Parameter(a); a
```

```
Parameter containing:tensor([-1.,  1.], requires_grad=True)
```

[1:39:12]

Instead, we're doing to call `.backward()`. On our computer, that will calculate the gradient for us.

```

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

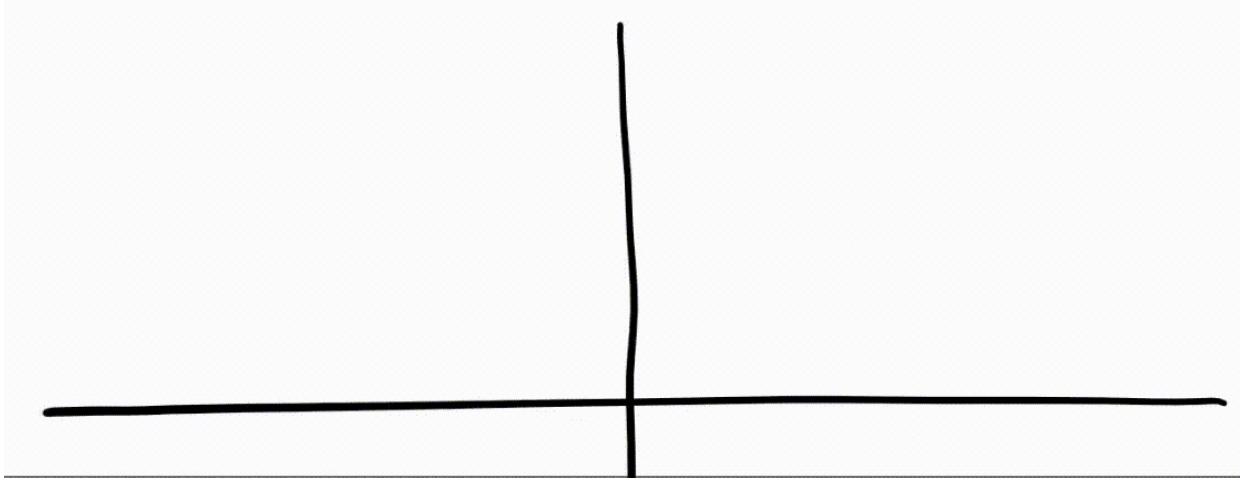
```

So here is what we're going to do. We are going to create a loop. We're going to loop through 100 times, and we're going to call a function called `update`. That function is going to:

- Calculate `y_hat` (i.e. our prediction)
- Calculate loss (i.e. our mean squared error)
- From time to time, it will print that out so we can see how we're going
- Calculate the gradient. In PyTorch, calculating the gradient is done by using a method called `backward`. Mean squared error was just a simple standard mathematical function. PyTorch keeps track of how it was calculated and lets us calculate the derivative. So if you do a mathematical operation on a tensor in PyTorch, you can call `backward` to calculate the derivative and the derivative gets stuck inside an attribute called `.grad`.
- Take my coefficients and I'm going to subtract from them my gradient (`sub_`). There is an underscore there because that's going to do it in-place. It's going to actually update those coefficients `a` to subtract the gradients from them. Why do we subtract? Because the gradient tells us if I move the whole thing downwards, the loss goes up. If I move the whole thing upwards, the loss goes down. So I want to do the opposite of the thing that makes it go up. We want our loss to be small. That's why we subtract.
- `lr` is our learning rate. All it is is the thing that we multiply by the gradient. Why is there any `lr` at all? Let me show you why.

⌚ Why is there any LR at all? [1:41:31]

Let's take a really simple example, a quadratic. And let's say your algorithm's job was to find where that quadratic was at its lowest point. How could it do this? Just like what we're doing now, the starting point would be just to pick some x value at random. Then find out what the value of y is. That's the starting point. Then it can calculate the gradient and the gradient is simply the slope, but it tells you moving in which direction is making you go down. So the gradient tells you, you have to go this way.



- If the gradient was really big, you might jump left a very long way, so you might jump all the way over to here. If you jumped over there, then that's actually not going to be very helpful because it's worse. We jumped too far so we don't want to jump too far.
- Maybe we should just jump a little bit. That is actually a little bit closer. So then we'll just do another little jump. See what the gradient is and do another little jump, and repeat.
- In other words, we find our gradient to tell us what direction to go and if we have to go a long way or not too far. But then we multiply it by some number less than 1 so we don't jump too far.

Hopefully at this point, this might be reminding you of something which is what happened when our learning rate was too high.

Learning rate (LR) too high

```
learn = create_cnn(data, models.resnet34, metrics=error_rate)
```

```
learn.fit_one_cycle(1, max_lr=0.5)
```

```
Total time: 00:13
epoch  train_loss  valid_loss  error_rate
1      12.220007  1144188288.000000  0.765957    (00:13)
```

Do you see why that happened now? Our learning rate was too high meant that we jumped all the way past the right answer further than we started with, and it got worse, and worse, and worse. So that's what a learning rate too high does.

On the other hand, if our learning rate is too low, then you just take tiny little steps and so eventually you're going to get there, but you are doing lots and lots of calculations along the way. So you really want to find something where it's either big enough steps like stairs or a little bit of back and forth. You want something that gets in there quickly but not so quickly it jumps out and diverges, not so slowly that it takes lots of steps. That's why we need a good learning rate and that's all it does.

So if you look inside the source code of any deep learning library, you'll find this:

```
a.sub_(lr * a.grad)
```

You will find something that says "coefficients - learning rate times gradient". And we will learn about some easy but important optimization we can do to make this go faster.

That's about it. There's a couple of other little minor issues that we don't need to talk about now: one involving zeroing out the gradient and other involving making sure that you turn gradient calculation off when you do the SGD update. If you are interested, we can discuss them on the forum or you can do our introduction to machine learning course which covers all the mechanics of this in more detail.

⌚ Training loop [1:45:43]

If we run `update` 100 times printing out the loss from time to time, you can see it starts at 8.9, and it goes down.

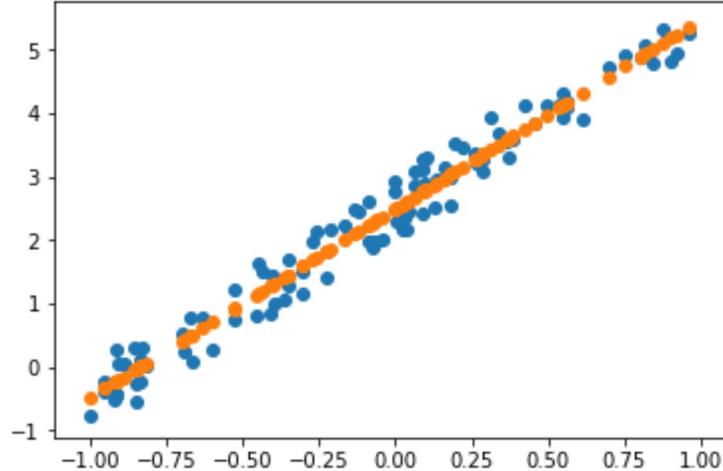
```
lr = 1e-1
for t in range(100): update()

tensor(8.8945, grad_fn=<MeanBackward1>
tensor(1.6115, grad_fn=<MeanBackward1>
tensor(0.5759, grad_fn=<MeanBackward1>
tensor(0.2435, grad_fn=<MeanBackward1>
tensor(0.1356, grad_fn=<MeanBackward1>
tensor(0.1006, grad_fn=<MeanBackward1>
tensor(0.0892, grad_fn=<MeanBackward1>
tensor(0.0855, grad_fn=<MeanBackward1>
tensor(0.0843, grad_fn=<MeanBackward1>
tensor(0.0839, grad_fn=<MeanBackward1>
```

So you can then print out scatterplots and there it is.

```
plt.scatter(x[:,0],y)
```

```
plt.scatter(x[:,0],x@a);
```



That's it! Believe it or not, that's gradient descent. So we just need to start with a function that's a bit more complex than `x@a` but as long as we have a function that can represent things like if this is a teddy bear, we now have a way to fit it.

⌚ Animate it! [1:46:20]

Let's now take a look at this as an animation. This is one of the nice things that you can do with matplotlib. You can take any plot and turn it into an animation. So you can now actually see it updating each step.

```
from matplotlib import animation, rc
rc('animation', html='html5')
```

You may need to uncomment the following to install the necessary plugin the first time you run this: (after you run following commands, make sure to restart the kernal for this notebook) If you are running in colab, the installs are not needed; just change the cell above to be ... `html='jshtml'` instead of ... `html='html5'`

```
#! sudo add-apt-repository -y ppa:mc3man/trusty-media
#! sudo apt-get update -y
#! sudo apt-get install -y ffmpeg
#! sudo apt-get install -y frei0r-plugins
```

Let's see what we did here. We simply said, as before, create a scatter plot, but then rather than having a loop, we used matplotlib's `FuncAnimation` to call 100 times this `animate` function. And this function just calls that `update` we created earlier then update the `y` data in our line. Repeat that 100 times, waiting 20 milliseconds after each one.

```
a = nn.Parameter(tensor(-1.,1))
```

```

fig = plt.figure()
plt.scatter(x[:,0], y, c='orange')
line, = plt.plot(x[:,0], x@a)
plt.close()

def animate(i):
    update()
    line.set_ydata(x@a)
    return line,

animation.FuncAnimation(fig, animate, np.arange(0, 100), interval=20)

```

You might think visualizing your algorithms with animations is something amazing and complex thing to do, but actually now you know it's 11 lines of code. So I think it's pretty darn cool.

That is SGD visualized and we can't visualize as conveniently what updating 50 million parameters in a ResNet 34 looks like but basically doing the same thing. So studying these simple version is actually a great way to get an intuition. So you should try running this notebook with a really big learning rate, with a really small learning rate, and see what this animation looks like, and try to get a feel for it. Maybe you can even try a 3D plot. I haven't tried that yet, but I'm sure it would work fine.

⌚ Mini-batches [1:48:08]

The only difference between stochastic gradient descent and this is something called *mini-batches*. You'll see, what we did here was we calculated the value of the loss on the whole dataset on every iteration. But if your dataset is 1.5 million images in ImageNet, that's going to be really slow. Just to do a single update of your parameters, you've got to calculate the loss on 1.5 million images. You wouldn't want to do that. So what we do is we grab 64 images or so at a time at random, and we calculate the loss on those 64 images, and we update our weights. Then we have another 64 random images, and we update our weights. In other words, the loop basically looks exactly the same but add some random indexes on our `x` and `y` to do a mini-batch at a time, and that would be the basic difference.

```

1 def update():
2     y_hat = x[rand_idx]@a
3     loss = mse(y[rand_idx], y_hat)
4     if t % 10 == 0: print(loss)
5     loss.backward()
6     with torch.no_grad():
7         a.sub_(lr * a.grad)
8         a.grad.zero_()

```

Once you add those grab a random few points each time, those random few points are called your mini-batch, and that approach is called SGD for Stochastic Gradient Descent.

⌚ Vocabulary [1:49:40]

There's quite a bit of vocab we've just covered, so let's remind ourselves.

- **Learning rate:** A thing we multiply our gradient by to decide how much to update the weights by.
- **Epoch:** One complete run through all of our data points (e.g. all of our images). So for non-stochastic gradient descent we just did, every single loop, we did the entire dataset. But if you've got a dataset with a thousand images and our mini-batch size is 100, then it would take you 10 iterations to see every image once. So that would be one epoch. Epochs are important because if you do lots of epochs, then you are looking at your images lots of times, so every time you see an image, there's a bigger chance of overfitting. So we generally don't want to do too many epochs.
- **Mini-batch:** A random bunch of points that you use to update your weights.
- **SGD:** Stochastic gradient descent using mini-batches.
- **Model / Architecture:** They kind of mean the same thing. In this case, our architecture is $\vec{y} = X\vec{a}$ - the architecture is the mathematical function that you're fitting the parameters to. And we're going to learn later today or next week what the mathematical function of things like ResNet34 actually is. But it's basically pretty much what you've just seen. It's a bunch of matrix products.
- **Parameters / Coefficients / Weights:** Numbers that you are updating.
- **Loss function:** The thing that's telling you how far away or how close you are to the correct answer. For classification problems, we use *cross entropy loss*, also known as *negative log likelihood loss*. This penalizes incorrect confident predictions, and correct unconfident predictions.

These models / predictors / teddy bear classifiers are functions that take pixel values and return probabilities. They start with some functional form like $\vec{y} = X\vec{a}$ and they fit the parameter a using SGD to try and do the best to calculate your predictions. So far, we've learned how to do regression which is a single number. Next we'll learn how to do the same thing for classification where we have multiple numbers, but basically the same.

In the process, we had to do some math. We had to do some linear algebra and calculus and a lot of people get a bit scared at that point and tell us "I am not a math person". If that's you, that's totally okay. But you are wrong. You are a math person. In fact, it turns out that in the actual academic research around this, there are not "math people" and "non-math people". It turns out to be entirely a result of culture and expectations. So you should check out Rachel's talk:

[There is no such thing as "not a math person"](#)

If you think you're not good at math...

Proceedings of the National Academy of Sciences of the United States of America

CURRENT ISSUE // ARCHIVE // NEWS & MULTIMEDIA // AUTHORS // ABOUT // COLLECTED ARTICLES // BROWSE BY TOPIC

Home > Current Issue > vol. 107 no. 5 > Sian L. Beilock, 1860–1863

Check for updates

Female teachers' math anxiety affects girls' math achievement

Sian L. Beilock¹, Elizabeth A. Gunderson, Gerardo Ramirez, and Susan C. Levine

omoju

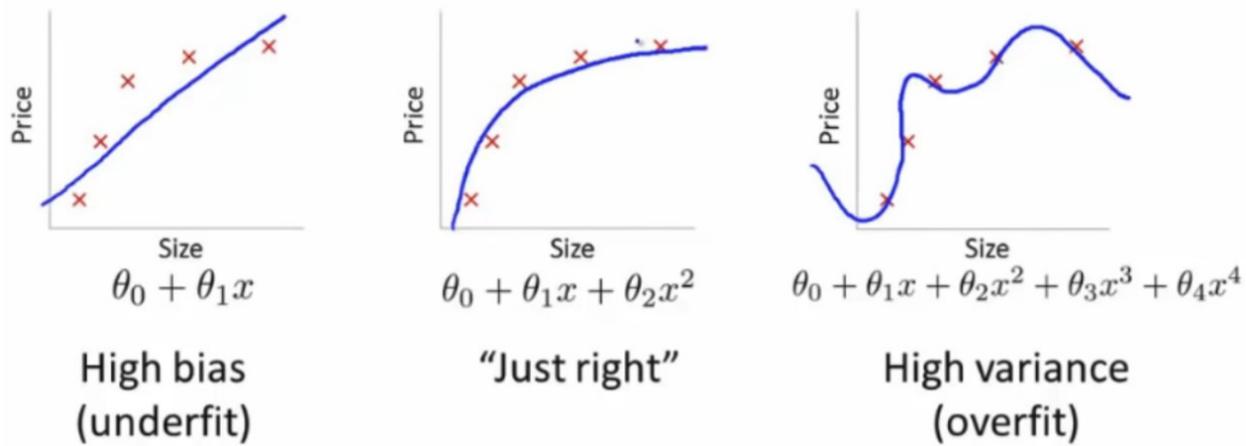
The Myth of Innate Ability in Tech

"A Mathematician's Lament" --Lockhart

She will introduce you to some of that academic research. If you think of yourself as not a math person, you should watch this so that you learn that you're wrong that your thoughts are actually there because somebody has told you you're not a math person. But there's actually no academic research to suggest that there is such a thing. In fact, there are some cultures like Romania and China where the "not a math person" concept never even appeared. It's almost unheard of in some cultures for somebody to say I'm not a math person because that just never entered that cultural identity.

So don't freak out if words like derivative, gradient, and matrix product are things that you're kind of scared of. It's something you can learn. Something you'll be okay with.

The last thing I want to close with is the idea of underfitting and overfitting. We just fit a line to our data. But imagine that our data wasn't actually line shaped. So if we try to fit which was something like constant + constant times X (i.e. a line) to it, it's never going to fit very well. No matter how much we change these two coefficients, it's never going to get really close.



On the other hand, we could fit some much bigger equation, so in this case it's a higher degree polynomial with lots of wiggly bits. But if we did that, it's very unlikely we go and look at some other place to find out the temperature and how much ice cream they are selling and we will get a good result. Because the wiggles are far too wiggly. So this is called overfitting.

We are looking for some mathematical function that fits just right to stay with a teddy bear analogies. You might think if you have a statistics background, the way to make things it just right is to have exactly the same number of parameters (i.e. to use a mathematical function that doesn't have too many parameters in it). It turns out that's actually completely not the right way to think about it.

⌚ Regularization and Validation Set [1:56:07]

There are other ways to make sure that we don't overfit. In general, this is called regularization. Regularization or all the techniques to make sure when we train our model that it's going to work not only well on the data it's seen but on the data it hasn't seen yet. The most important thing to know when you've trained a model is actually how well does it work on data that it hasn't been trained with. As we're going to learn a lot about next week, that's why we have this thing called a validation set.

What happens with the validation set is that we do our mini-batch SGD training loop with one set of data with one set of teddy bears, grizzlies, black bears. Then when we're done, we check the loss function and the accuracy to see how good is it on a bunch of images which were not included in the training. So if we do that, then if we have something which is too wiggly, it will tell us. "Oh, your loss function and your error is really bad because on the bears that it hasn't been trained with, the wiggly bits are in the wrong spot." Or else if it was underfitting, it would also tell us that your validation set is really bad.

Even for people that don't go through this course and don't learn about the details of deep learning, if you've got managers or colleagues at work who are wanting to learn about AI, the only thing that you really need to be teaching them is about the idea of a validation set. Because that's the thing they can then use to figure out if somebody's telling them snake oil or not. They hold back some data and they get told "oh, here's a model that we're going to roll out" and then you say "okay, fine. I'm just going to check it on this held out data to see whether it generalizes." There's a lot of details to get right when you design your validation set. We will talk about them briefly next week, but a more full version would be in Rachel's piece on the fast.ai blog called [How \(and why\) to create a good validation set](#). And this is also one of the things we go into a lot of detail in the intro to machine learning course. So we're going to try and give you enough to get by for this course, but it is certainly something that's worth deeper study as well.

Thanks everybody! I hope you have a great time building your web applications. See you next week.

 shivu1998 updated lesson3 (#31)

f50eda8 5 days ago

6 contributors 

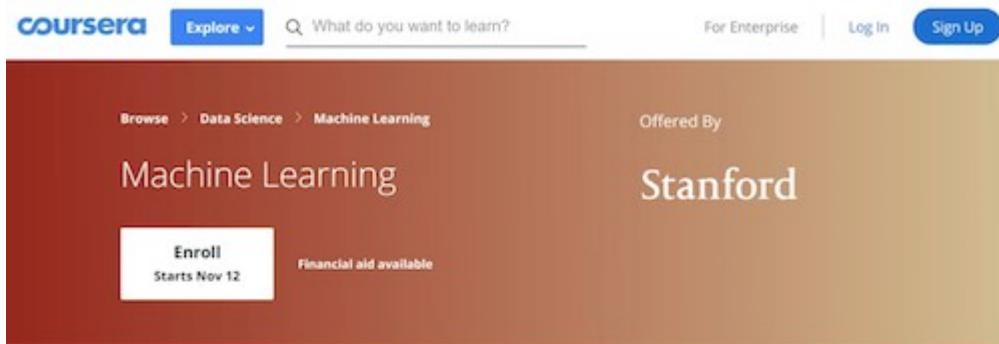
1797 lines (1184 sloc) 118 KB

[Raw](#)[Blame](#)[History](#)

⌚ Lesson 3

[Video / Lesson Forum](#)

A quick correction on citation. This chart originally came from Andrew Ng's excellent machine learning course on Coursera. Apologies for the incorrect citation.



The screenshot shows the Coursera homepage with the search bar "What do you want to learn?". Below it, the "Machine Learning" course by "Stanford" is displayed. The course card includes the title "Machine Learning", the provider "Offered By Stanford", an "Enroll" button, and a note that it starts on Nov 12. It also mentions "Financial aid available". At the bottom of the course card, there are links for "About", "Syllabus", "Reviews", "Instructors", "Enrollment Options", and "FAQ".

About this Course

★★★★★ 4.9 85,364 ratings • 21,950 reviews

Machine learning is the science of getting computers to act without being explicitly programmed. In the past decade, machine learning has given us self-driving cars, practical speech recognition, effective web search, and a vastly improved understanding of the human genome. Machine learning is so pervasive today that you probably use it dozens of times a day without knowing it. Many SHOW ALL

Andrew Ng's machine learning course on Coursera is great. In some ways, it's a little dated but a lot of the content is as appropriate as ever and taught in a bottom-up style. So it can be quite nice to combine it with our top down style and meet somewhere in the middle.

Also, if you are interested in machine learning foundations, you should check out our [machine learning course](#) as well. It is about twice as long as this deep learning course and takes you much more gradually through some of the foundational stuff around validation sets, model interpretation, how PyTorch tensor works, etc. I think all these courses together, if you really dig deeply into the material, do all of them. I know a lot of people who have and end up saying "oh, I got more out of each one by doing a whole lot". Or you can backwards and forwards to see which one works for you.

We started talking about deploying your web app last week. One thing that's going to make life a lot easier for you is that <https://course-v3.fast.ai/> has a production section where right now we have one platform but more will be added showing you how to deploy your web app really easily. When I say easily, for example, here is [how to deploy on Zeit guide](#) created by Navjot.

The screenshot shows a web page from the fast.ai course v3. At the top, there is a navigation bar with a logo and the text 'fast.ai course v3'. On the left, a sidebar menu is open, showing categories like 'Getting started', 'Server setup', 'Returning to work', 'Production' (which is currently selected), 'Zeit', and 'fastai v1'. The main content area is titled 'Deploying on Zeit'. Below the title is a 'Table of Contents' section with several items: 'One-time setup' (with sub-items 'Install Now's CLI' and 'Grab starter pack for model deployment'), 'Per-project setup' (with sub-items 'Upload your trained model file', 'Customize the app for your model', 'Deploy', 'Scaling', and 'Test the URL of your working app'), and 'Local testing'. At the bottom of the content area, there is a small image showing two screenshots: one of a browser window titled 'Now – Global Serverless Deployments' and one of a terminal window showing a directory structure for a PyTorch app.

As you can see, it's just a page. There's almost nothing to and it's free. It's not going to serve 10,000 simultaneous requests but it'll certainly get you started and I found it works really well. It's fast. Deploying a model doesn't have to be slow or complicated anymore. And the nice thing is, you can use this for a Minimum Viable Product (MVP). If you do find it's starting to get a thousand simultaneous requests, then you know that things are working out and you can start to upgrade your instance types or add to a more traditional big engineering approach. If you actually use this starter kit, it will create my teddy bear finder for you. So the idea is, this template is as simple as possible. So you can fill in your own style sheets, your own custom logic, and so forth. This is designed to be a minimal thing, so you can see exactly what's going on. The backend is a simple REST style interface that sends back JSON and the frontend is a super simple little Javascript thing. It should be a good way to get a sense of how to build a web app which talks to a PyTorch model.

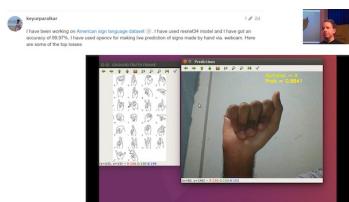
Edward Ross built the what Australian car is that? app

The screenshot shows the homepage of the **WHATCAR** website. At the top, it says "WHAT CAR IS THAT?" and "Upload a photo by clicking below and we'll tell you WhatCar it is". Below this is a large image of a dark-colored Honda CR-V parked on a grassy area. A yellow banner at the bottom of the image reads "HONDA CR-V". To the right of the car, there is a quote in a white box: "*Building an app is a great experience; being able to test my model live with photos from my phone is amazing and helped me understand my model a lot more*". Above the quote is a small image of Edward Ross. On the right side of the page, there is a grid of many small car images. Below the grid, text states: "It currently classifies around 400 models of the most popular cars in Australia". Further down, it says "It has 88% accuracy on a balanced validation set (although there are some duplicates in my dataset; if any leaked into the validation set this is optimistic). This is pretty impressive since some of the models are really similar". There are also two images of Mercedes-Benz sedans labeled "Mercedes Benz C200 Kompressor" and "Mercedes Benz C180 Kompressor".

I thought it was interesting that Edward said on the forum that building this app was actually a great experience in terms of understanding how the model works himself better. It's interesting that he's describing trying it out on his phone. A lot of people think "Oh, if I want something on my phone, I have to create some kind of mobile TensorFlow, ONNX, whatever tricky mobile app" - you really don't. You can run it all in the cloud and make it just a web app or use some kind of simple little GUI frontend that talks to a rest backend. It's not that often that you'll need to actually run stuff on the phone. So this is a good example of that.

The image contains six separate screenshots of machine learning projects:

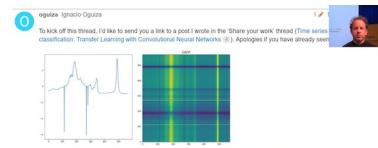
- Guitar Classifier by Christian Werner:** Shows a guitar and a prediction of "Gibson Les Paul".
- Healthy or Not! by Nikhil Utane:** Shows a salad and a prediction of "That's probably healthy!"
- Hummingbird Classifier by Nissan Dookeran:** Shows a hummingbird and a list of bird species.
- Edible Mushroom? by Ramon:** Shows a mushroom and a prediction of "edible".
- Cousin Recognizer by Charlie Harrington:** Shows a person's face and a prediction of "That's probably Charlie!".
- Emotion Classifier by Ethan Sutin and Team 26:** Shows a person's face with a smiley face icon and a caption about exploring high-performance use cases.



American Sign Language by Keyur Paralkar



Your City from Space by Henri Palacci



It shows one way in which a univariate TS dataset (Oncology) from the UCR time series datasets (—) can be transformed into images (using Gramian Angular Field), and then modelled following the general transfer learning paradigm.

The results surprised me (very close to state of the art) considering:

1. How small the train sample is (20 samples only)
2. These GAFD images are very different from those in ImageNet
3. I was just applying the standard fastai method, only tuning epochs and lr. I'm sure there is room for improvement

Univariate TS as images using Gramian Angular Field by Ignacio Oguiza



Face Expression Recognition by Pierre Guillou



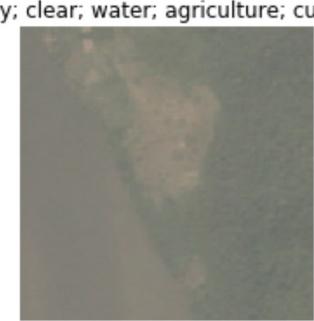
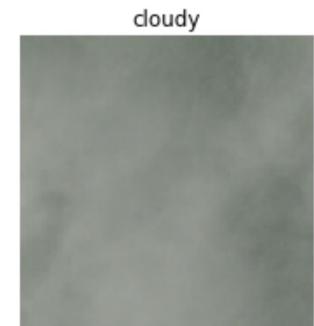
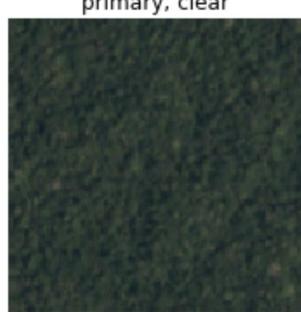
Tumor-normal sequencing by Alena Harley

Nice to see what people have been building in terms of both web apps and just classifiers. What we are going to do today is look at a whole lot more different types of model that you can build and we're going to zip through them pretty quickly and then we are going to go back and see how all these things work and what the common denominator is. All of these things, you can create web apps from these as well but you'll have to think about how to slightly change that template to make it work with these different applications. I think that'll be a really good exercise in making sure you understand the material.

Multi-label classification with Planet Amazon dataset 9:51

[lesson3-planet.ipynb](#)

The first one we're going to look at is a dataset of satellite images. Satellite imaging is a really fertile area for deep learning. Certainly a lot of people are already using deep learning in satellite imaging but only scratching the surface. The dataset we are going to look at looks like this:



It has satellite tiles and for each one, as you can see, there's a number of different labels for each tile. One of the labels always represents the weather (e.g. cloudy, partly_cloudy). And all of the other labels tell you any interesting features that are seen there. So primary means primary rainforest, agriculture means there's some farming, road means road, and so forth. As I am sure you can tell, this is a little different to all the classifiers we've seen so far because there's not just one label, there's potentially multiple labels. So, multi-label classification can be done in a very similar way but the first thing we are going to need to do is to download the data.

⌚ Downloading the data 11:02

This data comes from Kaggle. Kaggle is mainly known for being a competitions website and it's really great to download data from Kaggle when you're learning because you can see how would I have done in that competition. And it's a good way to see whether you know what you are doing. I tend to think the goal is to try and get in the top 10%. In my experience, all the people in the top 10% of a competition really know what they're doing. So if you can get in the top 10%, then that's a really good sign.

Pretty much every Kaggle dataset is not available for download outside of Kaggle (at least competition datasets) so you have to download it through Kaggle. The good news is that Kaggle provides a python-based downloader tool which you can use, so we've got a quick description here of how to download stuff from Kaggle.

You first have to install the Kaggle download tool via `pip`.

```
#! pip install kaggle --upgrade
```

What we tend to do when there's a one-off thing to do is we show you the commented out version in the notebook and you can just remove the comment. If you select a few lines and then hit `ctrl + /`, it uncomment them all. Then when you are done, select them again, `ctrl + /` again and re-comments them all. So this line will install `kaggle` for you. Depending on your platform, you may need `sudo` or `/something/pip`, you may need `source activate` so have a look on the setup instructions or the returning to work instructions on the course website to see when we do `conda install`, you have to do the same basic steps for your `pip install`.

Once you've got that module installed, you can then go ahead and download the data. Basically it's as simple as saying `kaggle competitions download -c competition_name -f file_name`. The only other steps you do is that you have to authenticate yourself and there is a little bit of information here on exactly how you can go about downloading from Kaggle the file containing your API authentication information. I wouldn't bother going through it here, but just follow these steps.

Then you need to upload your credentials from Kaggle on your instance. Login to `kaggle` and click on your profile picture on the top left corner, then 'My account'. Scroll down until you find a button named 'Create New API Token' and click on it. This will trigger the download of a file named 'kaggle.json'.

Upload this file to the directory this notebook is running in, by clicking "Upload" on your main Jupyter page, then uncomment and execute the next two commands (or run them in a terminal).

```
#! mkdir -p ~/.kaggle/  
#! mv kaggle.json ~/.kaggle/
```

You're all set to download the data from [planet competition](#). You **first need to go to its main page and accept its rules**, and run the two cells below (uncomment the shell commands to download and unzip the data). If you get a `403 forbidden` error it means you haven't accepted the competition rules yet (you have to go to the competition page, click on *Rules* tab, and then scroll to the bottom to find the *Accept* button).

```
path = Config.data_path()/'planet'
path.mkdir(exist_ok=True)
path

PosixPath('/home/jhoward/.fastai/data/planet')

# ! kaggle competitions download -c planet-understanding-the-amazon-from-space -f
# ! kaggle competitions download -c planet-understanding-the-amazon-from-space -f
# ! unzip -q -n {path}/train_v2.csv.zip -d {path}
```

Sometimes stuff on Kaggle is not just zipped or tarred but it's compressed with a program called 7zip which will have a .7z extension. If that's the case, you'll need to either `apt install p7zip` or here is something really nice. Some kind person has created a `conda` installation of 7zip that works on every platform. So you can always just run this `conda install` - doesn't even require a `sudo` or anything like that. This is actually a good example of where `conda` is super handy. You can actually install binaries and libraries and stuff like that and it's nicely cross-platform. So if you don't have 7zip installed, that's a good way to get it.

To extract the content of this file, we'll need 7zip, so uncomment the following line if you need to install it (or run `sudo apt install p7zip` in your terminal).

```
# ! conda install -y -c haasad eidl7zip
```

This is how you unzip a 7zip file. In this case, it's tarred and 7zipped, so you can do this all in one step. `7za` is the name of the 7zip archival program you would run.

That's all basic stuff which if you are not familiar with the command line and stuff, it might take you a little bit of experimenting to get it working. Feel free to ask on the forum, make sure you search the forum first to get started.

```
# ! 7za -bd -y -so x {path}/train-jpg.tar.7z | tar xf - -C {path}
```

⌚ Multiclassification 14:49

Once you've got the data downloaded and unzipped, you can take a look at it. In this case, because we have multiple labels for each tile, we clearly can't have a different folder for each image telling us what the label is. We need some different way to label it. The way Kaggle did it was they provided a CSV file that had each file name along with a list of all the labels. So in order to just take a look at that CSV file, we can read it using the Pandas library. If you haven't used pandas before, it's kind of the standard way of dealing with tabular data in Python. It pretty much always appears in the `pd` namespace. In this case we're not really doing anything with it other than just showing you the contents of this file. So we can read it, take a look at the first few lines, and there it is:

```
df = pd.read_csv(path/'train_v2.csv')
df.head()
```

	image_name	tags
0	train_0	haze primary
1	train_1	agriculture clear primary water
2	train_2	clear primary
3	train_3	clear primary
4	train_4	agriculture clear habitation primary road

We want to turn this into something we can use for modeling. So the kind of object that we use for modeling is an object of the DataBunch class. We have to somehow create a data bunch out of this. Once we have a data bunch, we'll be able to go `.show_batch()` to take a look at it. And then we'll be able to go `create_cnn` with it, and we would be able to start training.

So really the trickiest step previously in deep learning has often been getting your data into a form that you can get it into a model. So far we've been showing you how to do that using various "factory methods" which are methods where you say, "I want to create this kind of data from this kind of source with these kinds of options." That works fine, sometimes, and we showed you a few ways of doing it over the last couple of weeks. But sometimes you want more flexibility, because there's so many choices that you have to make about:

- Where do the files live
- What's the structure they're in
- How do the labels appear
- How do you spit out the validation set
- How do you transform it

So we've got this unique API that I'm really proud of called the [data block API](#). The data block API makes each one of those decisions a separate decision that you make. There are separate methods with their own parameters for every choice that you make around how to create / set up my data.

```
tfms = get_transforms(flip_vert=True, max_lighting=0.1, max_zoom=1.05, max_warp=0

np.random.seed(42)
src = (ImageFileList.from_folder(path)
        .label_from_csv('train_v2.csv', sep=' ', folder='train-jpg', suffix='.jpg')
        .random_split_by_pct(0.2))

data = (src.datasets()
        .transform(tfms, size=128)
        .databunch().normalize(imagenet_stats))
```

For example, to grab the planet data we would say:

- We've got a list of image files that are in a folder
- They're labeled based on a CSV with this name (`train_v2.csv`)
 - They have this separator () - remember I showed you back here that there's a space between them. By passing in separator, it's going to create multiple labels.
 - The images are in this folder (`train-jpg`)
 - They have this suffix (`.jpg`)
- They're going to randomly spit out a validation set with 20% of the data
- We're going to create datasets from that, which we are then going to transform with these transformations (`tfms`)
- Then we're going to create a data bunch out of that, which we will then normalize using these statistics (`imagenet_stats`)

So there's all these different steps. To give you a sense of what that looks like, the first thing I'm going to do is go back and explain what are all of the PyTorch and fastai classes you need to know about that are going to appear in this process. Because you're going to see them all the time in the fastai docs and PyTorch docs.

⌚ Dataset (PyTorch) 18:30

The first one you need to know about is a class called `Dataset`. The `Dataset` class is part of PyTorch and this is the source code for the `Dataset` class:

```

class Dataset(object):
    """An abstract class representing a Dataset.

    All other datasets should subclass it. All subclasses should override
    ``__len__``, that provides the size of the dataset, and ``__getitem__``,
    supporting integer indexing in range from 0 to len(self) exclusive.
    """

    def __getitem__(self, index):
        raise NotImplementedError

    def __len__(self):
        raise NotImplementedError

```

As you can see, it actually does nothing at all. The Dataset class in PyTorch defines two things: `__getitem__` and `__len__`. In Python these special things that are "underscore underscore something underscore underscore" - Pythonists call them "dunder" something. So these would be "dunder get items" and "dunder len". They're basically special magical methods with some special behavior. This particular method means that your object, if you had an object called `o`, it can be indexed with square brackets (e.g. `o[3]`). So that would call `__getitem__` with 3 as the index.

Then this one called `__len__` means that you can go `len(o)` and it will call that method. In this case, they're both not implemented. That is to say, although PyTorch says "in order to tell PyTorch about your data, you have to create a dataset", it doesn't really do anything to help you create the dataset. It just defines what the dataset needs to do. In other words, the starting point for your data is something where you can say:

- What is the third item of data in my dataset (that's what `__getitem__` does)
- How big is my dataset (that's what `__len__` does)

Fastai has lots of Dataset subclasses that do that for all different kinds of stuff. So far, you've been seeing image classification datasets. They are datasets where `__getitem__` will return an image and a single label of what is that image. So that's what a dataset is.

⌚ DataLoader (PyTorch) 20:37

Now a dataset is not enough to train a model. The first thing we know we have to do, if you think back to the gradient descent tutorial last week is we have to have a few images/items at a time so that our GPU can work in parallel. Remember we do this thing called a "mini-batch"? Mini-batch is a few items that we present to the model at a time that it can train from in parallel. To create a mini-batch, we use another PyTorch class called a DataLoader.

A DataLoader takes a dataset in its constructor, so it's now saying "Oh this is something I can get the third item and the fifth item and the ninth item." It's going to:

- Grab items at random

- Create a batch of whatever size you asked for
- Pop it on the GPU
- Send it off to your model for you

So a DataLoader is something that grabs individual items, combines them into a mini-batch, pops them on the GPU for modeling. So that's called a DataLoader and that comes from a Dataset.

You can see, already there are choices you have to make: what kind of dataset am I creating, what is the data for it, where it's going to come from. Then when I create my DataLoader: what batch size do I want to use.

⌚ DataBunch (fastai) 21:59

It still isn't enough to train a model, because we've got no way to validate the model. If all we have is a training set, then we have no way to know how we're doing because we need a separate set of held out data, a validation set, to see how we're getting along.

For that we use a fastai class called a DataBunch. A DataBunch is something which binds together a training data loader (`train_dl`) and a valid data loader (`valid_dl`). When you look at the fastai docs when you see these mono spaced font things, they're always referring to some symbol you can look up elsewhere. In this case you can see `train_dl` is the first argument of DataBunch. There's no point knowing that there's an argument with a certain name unless you know what that argument is, so you should always look after the `:` to find out that is a DataLoader. So when you create a DataBunch, you're basically giving it a training set data loader and a validation set data loader. And that's now an object that you can send off to a learner and start fitting,

They're the basic pieces. Coming back to here, these are all the stuff which is creating the dataset:

```

1 np.random.seed(42)
2 src = (ImageFileList.from_folder(path)
3         .label_from_csv('train_v2.csv', sep=' ', folder='train-jpg', suffix='.jpg')
4         .random_split_by_pct(0.2))
5
6 data = (src.datasets()
7         .transform(tfms, size=128)
8         .databunch().normalize(imagenet_stats))

```

With the dataset, the indexer returns two things: the image and the labels (assuming it's an image dataset).

- Where do the images come from?
- Where do the labels come from?
- Then I'm going to create two separate data sets the training and the validation

- `.datasets()` actually turns them into PyTorch datasets
- `.transform()` is the thing that transforms them
- `.databunch()` is actually going to create the the DataLoader and the DataBunch in one go

⌚ Data block API examples 23:56

Let's look at some examples of this data block API because once you understand the data block API, you'll never be lost for how to convert your dataset into something you can start modeling with.

[data_block.ipynb](#)

⌚ MNIST

Here are some examples of using the data block API. For example, if you're looking at MNIST (the pictures and classes of handwritten numerals), you can do something like this:

```
path = untar_data(URLs.MNIST_TINY)
tfms = get_transforms(do_flip=False)
path.ls()
```

```
[PosixPath('/home/jhoward/.fastai/data/mnist_tiny/valid'),
 PosixPath('/home/jhoward/.fastai/data/mnist_tiny/models'),
 PosixPath('/home/jhoward/.fastai/data/mnist_tiny/train'),
 PosixPath('/home/jhoward/.fastai/data/mnist_tiny/test'),
 PosixPath('/home/jhoward/.fastai/data/mnist_tiny/labels.csv')]
```

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

```
[PosixPath('/home/jhoward/.fastai/data/mnist_tiny/train/3'),
 PosixPath('/home/jhoward/.fastai/data/mnist_tiny/train/7')]
```

```
data = (ImageFileList.from_folder(path) #Where to find the data? -> in path and
        .label_from_folder()           #How to label? -> depending on the folder
        .split_by_folder()             #How to split in train/valid? -> use the
        .add_test_folder()             #Optionally add a test set
        .datasets()                   #How to convert to datasets?
        .transform(tfms, size=224)     #Data augmentation? -> use tfms with a size
        .databunch())                 #Finally? -> use the defaults for conversion
```

- What kind of data set is this going to be?

- It's going to come from a list of image files which are in some folder.
- They're labeled according to the folder name that they're in.
- We're going to split it into train and validation according to the folder that they're in (`train` and `valid`).
- You can optionally add a test set. We're going to be talking more about test sets later in the course.
- We'll convert those into PyTorch datasets now that that's all set up.
- We will then transform them using this set of transforms (`tfms`), and we're going to transform into something of this size (`224`).
- Then we're going to convert them into a data bunch.

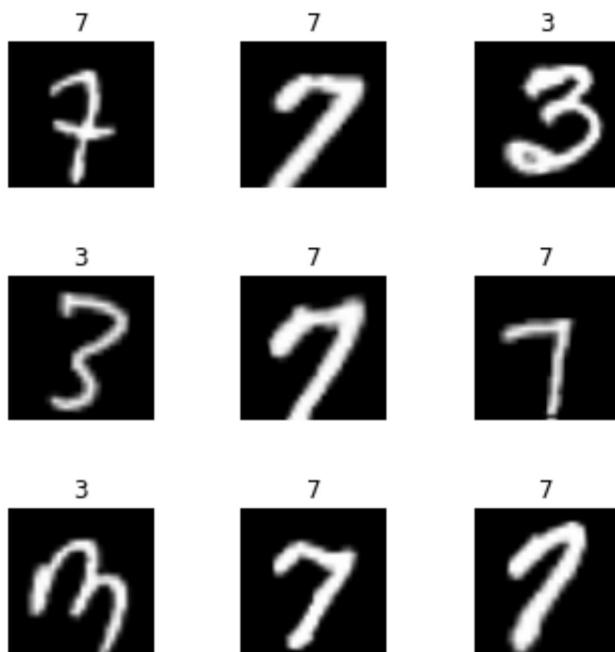
So each of those stages inside these parentheses are various parameters you can pass to customize how that all works. But in the case of something like this MNIST dataset, all the defaults pretty much work, so this is all fine.

```
data.train_ds[0]
```

```
(Image (3, 224, 224), 0)
```

Here it is. `data.train_ds` is the dataset (not the data loader) so I can actually index into it with a particular number. So here is the zero indexed item in the training data set: it's got an image and a label.

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



We can show batch to see an example of the pictures of it. And we could then start training.

```
data.valid_ds.classes
```

```
['3', '7']
```

Here are the classes that are in that dataset. This little cut-down sample of MNIST has 3's and 7's.

⌚ Planet 26:01

Here's an example of using planet dataset. This is actually again a little subset of planet we use to make it easy to try things out.

```
planet = untar_data(URLs.PLANET_TINY)
planet_tfms = get_transforms(flip_vert=True, max_lighting=0.1, max_zoom=1.05, max_
data = ImageDataBunch.from_csv(planet, folder='train', size=128, suffix='.jpg', s
```

With the data block API we can rewrite this like that:

```
data = (ImageFileList.from_folder(planet)
    #Where to find the data? -> in planet and its subfolders
    .label_from_csv('labels.csv', sep=' ', folder='train', suffix='.jpg')
    #How to label? -> use the csv file labels.csv in path,
    #add .jpg to the names and take them in the folder train
    .random_split_by_pct()
    #How to split in train/valid? -> randomly with the default 20% in valid
    .datasets()
    #How to convert to datasets? -> use ImageMultiDataset
    .transform(planet_tfms, size=128)
    #Data augmentation? -> use tfms with a size of 128
    .databunch())
    #Finally? -> use the defaults for conversion to databunch
```

In this case:

- Again, it's an ImageFileList
- We are grabbing it from a folder
- This time we're labeling it based on a CSV file
- We're randomly splitting it (by default it's 20%)
- Creating data sets
- Transforming it using these transforms (`planet_tfms`), we're going to use a smaller size (128).

- Then create a data bunch

```
data.show_batch(rows=3, figsize=(10,8))
```

partly_cloudy; primary



primary; agriculture; clear; road; habitation



primary; agriculture; clear



primary; clear



partly_cloudy; primary; water



primary; clear



primary; clear



partly_cloudy; primary



primary; clear



There it is. Data bunches know how to draw themselves amongst other things.

⌚ CAMVID 26:38

Here's some more examples we're going to be seeing later today.

```
camvid = untar_data(URLs.CAMVID_TINY)
path_lbl = camvid/'labels'
path_img = camvid/'images'
```

```
codes = np.loadtxt(camvid/'codes.txt', dtype=str); codes
```

```
array(['Animal', 'Archway', 'Bicyclist', 'Bridge', 'Building', 'Car',
'CartLuggagePram', 'Child', 'Column_Pole',
'Fence', 'LaneMkgsDriv', 'LaneMkgsNonDriv', 'Misc_Text',
'MotorcycleScooter', 'OtherMoving', 'ParkingBlock',
```

```
'Pedestrian', 'Road', 'RoadShoulder', 'Sidewalk', 'SignSymbol', 'Sky',  
'SUVPickupTruck', 'TrafficCone',  
'TrafficLight', 'Train', 'Tree', 'Truck_Bus', 'Tunnel',  
'VegetationMisc', 'Void', 'Wall'], dtype='<U17')
```

```
get_y_fn = lambda x: path_lbl/f'{x.stem}_P{x.suffix}'
```

```
data = (ImageFileList.from_folder(path_img)  
    .label_from_func(get_y_fn)  
    .random_split_by_pct()  
    .datasets(SegmentationDataset, classes=codes)  
    .transform(get_transforms(), size=96, tfm_y=True)  
    .databunch(bs=64))  
  
data.show_batch(rows=2, figsize=(5,5))
```

#Where are the input files
#How to label? -> use a function
#How to split between training and validation
#How to create a dataset
#Data aug -> Use standard transforms
#Lastly convert in a data bunch

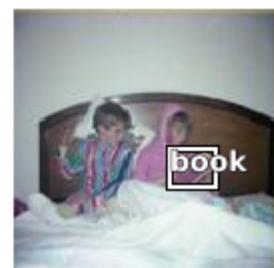
What if we look at this data set called CAMVID? CAMVID looks like this. It contains pictures and every pixel in the picture is color coded. So in this case :

- We have a list of files in a folder.
- We're going to label them using a function. So this function (`get_y_fn`) is basically the thing which tells it whereabouts of the color coding for each pixel. It's in a different place.
- Randomly split it in some way
- Create some datasets in some way. We can tell it for our particular list of classes, how do we know what pixel you know value 1 versus pixel value 2 is. That was something that we can read in.
- Some transforms.
- Create a data bunch. You can optionally pass in things like what batch size do you want.

Again, it knows how to draw itself and you can start learning with that.

↪ COCO 27:41

One more example. What if we wanted to create something like this:



This is call an object detection dataset. Again, we've got a little minimal COCO dataset. COCO is the most famous academic dataset for object detection.

```
coco = untar_data(URLs.COCO_TINY)
images, lbl_bbox = get_annotations(coco/'train.json')
img2bbox = {img:bb for img, bb in zip(images, lbl_bbox)}
get_y_func = lambda o:img2bbox[o.name]

data = (ImageFileList.from_folder(coco)
        #Where are the images? -> in coco
        .label_from_func(get_y_func)
        #How to find the labels? -> use get_y_func
        .random_split_by_pct()
        #How to split in train/valid? -> randomly with the default 20% in valid
        .datasets(ObjectDetectDataset)
        #How to create datasets? -> with ObjectDetectDataset
        #Data augmentation? -> Standard transforms with tfm_y=True
        .databunch(bs=16, collate_fn=bb_pad_collate))
        #Finally we convert to a DataBunch and we use bb_pad_collate
```

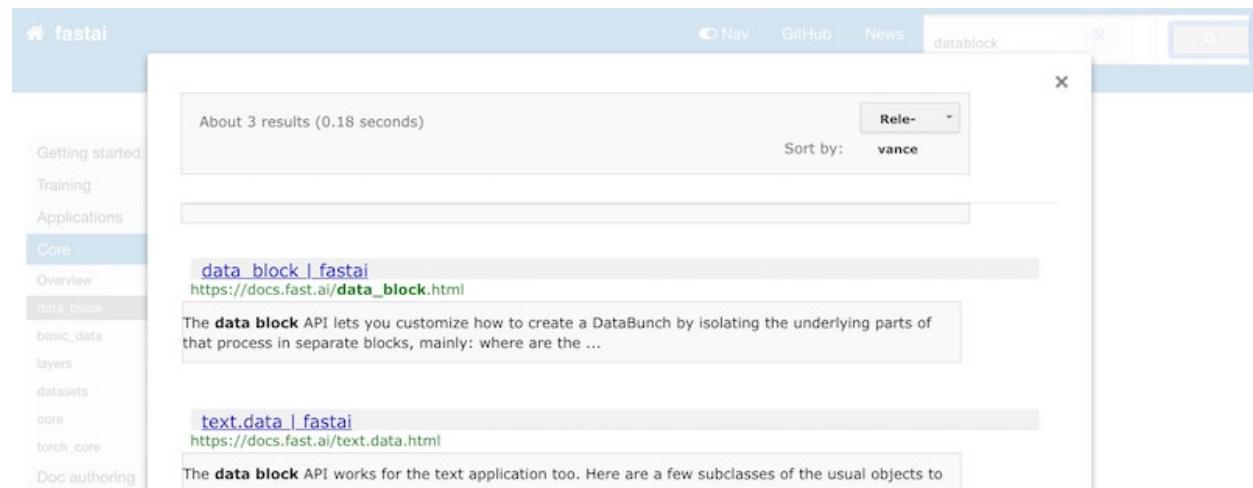
We can create it using the same process:

- Grab a list of files from a folder.

- Label them according to this little function (`get_y_func`).
- Randomly split them.
- Create an object detection dataset.
- Create a data bunch. In this case you have to use generally smaller batch sizes or you'll run out of memory. And you have to use something called a "collation function".

Once that's all done we can again show it and here is our object detection data set. So you get the idea. So here's a really convenient notebook. Where will you find this? Ah, this notebook is the documentation. Remember how I told you that all of the documentation comes from notebooks? You'll find them in [fastai repo in docs_src](#). This which you can play with and experiment with inputs and outputs, and try all the different parameters, you will find the [data block API examples of use](#), if you go to the documentation here it is - the [data block API examples of use](#).

Everything that you want to use in fastai, you can look it up in the documentation. There is also search functionality available:



So once you find some documentation that you actually want to try playing with yourself, just look up the name (e.g. `data_block.html`) and then you can open up a notebook with the same name (e.g. `data_block.ipynb`) in the fastai repo and play with it yourself.

⌚ Creating satellite image data bunch 29:35

That was a quick overview of this really nice data block API, and there's lots of documentation for all of the different ways you can label inputs, split data, and create datasets. So that's what we're using for planet.

In the documentation, these two steps were all joined up together:

```
np.random.seed(42)
src = (ImageFileList.from_folder(path)
```

```
.label_from_csv('train_v2.csv', sep=' ', folder='train-jpg', suffix='.jpg'  
.random_split_by_pct(0.2))
```

```
data = (src.datasets()  
.transform(tfms, size=128)  
.databunch().normalize(imagenet_stats))
```

We can certainly do that here too, but you'll learn in a moment why it is that we're actually splitting these up into two separate steps which is also fine as well.

A few interesting points about this.

- **Transforms:** Transforms by default will flip randomly each image, but they'll actually randomly only flip them horizontally. If you're trying to tell if something is a cat or a dog, it doesn't matter whether it's pointing left or right. But you wouldn't expect it to be upside down. On the other hand for satellite imagery whether something's cloudy or hazy or whether there's a road there or not, could absolutely be flipped upside down. There's no such thing as a right way up from space. So `flip_vert` which defaults to `False`, we're going to flip over to `True` to say you should actually do that. And it doesn't just flip it vertically, it actually tries each possible 90-degree rotation (i.e. there are 8 possible symmetries that it tries out).
- **Warp:** Perspective warping is something which very few libraries provide, and those that do provide it it tends to be really slow. I think fastai is the first one to provide really fast perspective warping. Basically, the reason this is interesting is if I look at you from below versus above, your shape changes. So when you're taking a photo of a cat or a dog, sometimes you'll be higher, sometimes you'll be lower, then that kind of change of shape is certainly something that you would want to include as you're creating your training batches. You want to modify it a little bit each time. Not true for satellite images. A satellite always points straight down at the planet. So if you added perspective warping, you would be making changes that aren't going to be there in real life. So I turn that off.

This is all something called data augmentation. We'll be talking a lot more about it later in the course. But you can start to get a feel for the kind of things that you can do to augment your data. In general, maybe the most important one is if you're looking at astronomical data, pathology digital slide data, or satellite data where there isn't really an up or down, turning on `flip vertical` true is generally going to make your models generalize better.

⌚ Creating multi-label classifier 35:59

Now to create a multi-label classifier that's going to figure out for each satellite tile what's the weather and what else can I see in it, there's basically nothing else to learn. Everything else that you've already learned is going to be exactly nearly the same.

```
arch = models.resnet50

acc_02 = partial(accuracy_thresh, thresh=0.2)
f_score = partial(fbeta, thresh=0.2)
learn = create_cnn(data, arch, metrics=[acc_02, f_score])
```

When I first built this notebook, I used `resnet34` as per usual. Then I tried `resnet50` as I always like to do. I found `resnet50` helped a little bit and I had some time to run it, so in this case I was using `resnet50`.

There's one more change I make which is metrics. To remind you, a metric has got nothing to do with how the model trains. Changing your metrics will not change your resulting model at all. The only thing that we use metrics for is we print them out during training.

```
lr = 0.01
```

```
learn.fit_one_cycle(5, slice(lr))
```

```
Total time: 04:17
epoch  train_loss  valid_loss  accuracy_thresh  fbeta
1      0.115247   0.103319   0.950703       0.910291 (00:52)
2      0.108289   0.099074   0.953239       0.911656 (00:50)
3      0.102342   0.092710   0.953348       0.917987 (00:51)
4      0.095571   0.085736   0.957258       0.926540 (00:51)
5      0.091275   0.085441   0.958006       0.926234 (00:51)
```

Here it's printing out accuracy and this other metric called `fbeta`. If you're trying to figure out how to do a better job with your model, changing the metrics will never be something that you need to do. They're just to show you how you're doing.

You can have one metric, no metrics, or a list of multiple metrics to be printed out as your models training. In this case, I want to know two things:

1. The accuracy.
2. How would I do on Kaggle.

Kaggle told me that I'm going to be judged on a particular metric called the F score. I'm not going to bother telling you about the F score - it's not really interesting enough to be worth spending your time on. But it's basically this. When you have a classifier, you're going to have some false positives and some false negatives. How do you weigh up those two things to create a single number? There's lots of different ways of doing that and something called the F score is a nice way of combining that into a single number. And there are various kinds of F scores: F1, F2 and so forth. And Kaggle said in the competition rules, we're going to use a metric called F2.

In [23]: arch = models.resnet50

In []: fbeta()

In [24]: Signature: fbeta(y_pred:torch.Tensor, y_true:torch.Tensor, thresh:float=0.2, beta:float=2.0, eps:float=1e-09, sigmoid:bool=True) -> <function NewType.<locals>.new_type at 0x7f6c9d6c9b70>

Docstring: Computes the f_beta between preds and targets

We use the LR Finder to pick a good learning rate.

We have a metric called `fbeta`. In other words, it's F with 1, 2, or whatever depending on the value of beta. We can have a look at its signature and it has a threshold and a beta. The beta is 2 by default, and Kaggle said that they're going to use F 2 so I don't have to change that. But there's one other thing that I need to set which is a threshold.

What does that mean? Here's the thing. Do you remember we had a little look the other day at the source code for the accuracy metric? And we found that it used this thing called `argmax`. The reason for that was we had this input image that came in, it went through our model, and at the end it came out with a table of ten numbers. This is if we're doing MNIST digit recognition and the ten numbers were the probability of each of the possible digits. Then we had to look through all of those and find out which one was the biggest. So the function in Numpy, PyTorch, or just math notation that finds the biggest in returns its index is called `argmax`.

To get the accuracy for our pet detector, we use this accuracy function called `argmax` to find out which class ID pet was the one that we're looking at. Then it compared that to the actual, and then took the average. That was the accuracy.

37:23

We can't do that for satellite recognition because there isn't one label we're looking for - there's lots. A data bunch has a special attribute called `c` and `c` is going to be how many outputs do we want our model to create. For any kind of classifier, we want one probability for each possible class. In other words, `data.c` for classifiers is always going to be equal to the length of `data.classes`.

```
1 | data.c
```

17

```
1 | len(data.classes)
```

17

```
1 | data.classes
```

```
['road',
 'slash_burn',
 'cultivation',
 'water',
 'selective_logging',
 'bare_ground',
 'artisinal_mine',
 'blow_down',
 'partly_cloudy',
 'clear',
 'conventional_mine',
 'cloudy',
 'blooming',
 'agriculture',
 'haze',
 'habitation',
 'primary']
```

They are the 17 possibilities. So we're going to have one probability for each of those. But then we're not just going to pick out one of those 17, we're going to pick out n of those 17. So what we do is, we compare each probability to some threshold. Then we say anything that's higher than that threshold, we're going to assume that the models saying it does have that feature. So we can pick that threshold.

```
1 | arch = models.resnet50
```

```
1 | acc_02 = partial(accuracy_thresh, thresh=0.2)
2 | f_score = partial(fbeta, thresh=0.2)
3 | learn = create_cnn(data, arch, metrics=[acc_02, f_score])
```

I found that for this particular dataset, a threshold of 0.2 seems to generally work pretty well. This is the kind of thing you can easily just experiment to find a good threshold. So I decided I want to print out the accuracy at a threshold of 0.2.

The normal accuracy function doesn't work that way. It doesn't `argmax`. We have to use a different accuracy function called `accuracy_thresh`. That's the one that's going to compare every probability to a threshold and return all the things higher than that threshold and compare accuracy that way.

accuracy_thresh()

```
Signature: accuracy_thresh(y_pred:torch.Tensor, y_true:torch.Tensor, thresh:float, gmoid:bool=True) -> <function NewType.<locals>.new_type at 0x7f6c9d6c9b70>
Docstring: Compute accuracy when `y_pred` and `y_true` are the same size.
File:      /data1/jhoward/git/fastai/fastai/metrics.py
```

⌚ Python3 partial [39:17]

One of the things we had passed in is `thresh`. Now of course our metric is going to be calling our function for us, so we don't get to tell it every time it calls back what threshold do we want, so we really want to create a special version of this function that always uses a threshold of 0.2. One way to do that would be defining a function `acc_02` as below:

```
def acc_02(inp, targ): return accuracy_thresh(inp, targ, thresh=0.2)
```

We could do it that way. But it's so common that computer science has a term for that called a "partial" / "partial function application" (i.e. create a new function that's just like that other function but we are always going to call it with a particular parameter).

Python3 has something called `partial` that takes some function and some list of keywords and values, and creates a new function that is exactly the same as this function (`accuracy_thresh`) but is always going to call it with that keyword argument (`thresh=0.2`).

```
acc_02 = partial(accuracy_thresh, thresh=0.2)
```

This is a really common thing to do particularly with the fastai library because there's lots of places where you have to pass in functions and you very often want to pass in a slightly customized version of a function so here's how you do it.

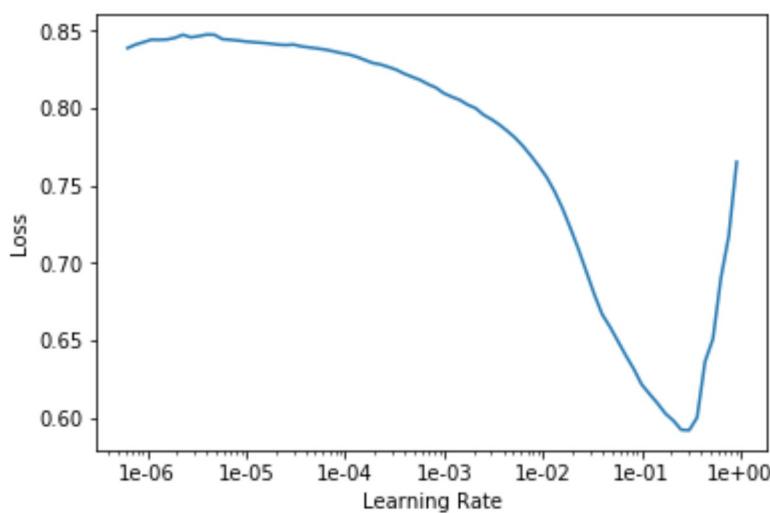
Similarly, `fbeta` with `thresh=0.2`:

```
acc_02 = partial(accuracy_thresh, thresh=0.2)
f_score = partial(fbeta, thresh=0.2)
learn = create_cnn(data, arch, metrics=[acc_02, f_score])
```

I can pass them both in as metrics and I can then go ahead and do all the normal stuff.

```
learn.lr_find()
```

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



Find the thing with the steepest slope - so somewhere around $1e-2$, make that our learning rate.

```
lr = 0.01
```

Then fit for awhile with `5, slice(lr)` and see how we go.

```
learn.fit_one_cycle(5, slice(lr))
```

Total time: 04:17				
epoch	train_loss	valid_loss	accuracy_thresh	fbeta
1	0.115247	0.103319	0.950703	0.910291 (00:52)
2	0.108289	0.099074	0.953239	0.911656 (00:50)
3	0.102342	0.092710	0.953348	0.917987 (00:51)
4	0.095571	0.085736	0.957258	0.926540 (00:51)
5	0.091275	0.085441	0.958006	0.926234 (00:51)

So we've got an accuracy of about 96% and F beta of about 0.926 and so you could then go and have a look at [Planet private leaderboard](#). The top 50th is about 0.93 so we kind of say like oh we're on the right track. So as you can see, once you get to a point that the data is there, there's very little extra to do most of the time.

Question: When your model makes an incorrect prediction in a deployed app, is there a good way to "record" that error and use that learning to improve the model in a more targeted way? [42:01]

That's a great question. The first bit - is there a way to record that? Of course there is. You record it. That's up to you. Maybe some of you can try it this week. You need to have your user tell you that you were wrong. This Australian car you said it was a Holden and actually it's a Falcon. So first of all, you'll need to collect that feedback and the only way to do that is to ask the user to tell you when it's wrong. So you now need to record in some log somewhere - something saying you know this was the file, I've stored it here, this was the prediction I made, this was the actual that they told me. Then at the end of the day or at the end of the week, you could set up a little job to run something or you can manually run something. What are you going to do? You're going to do some fine-tuning. What does fine-tuning look like? Good segue Rachel! It looks like this.

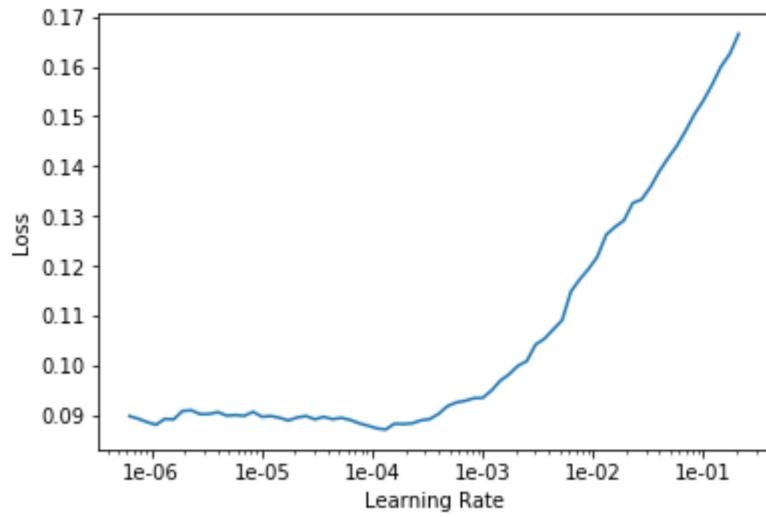
So let's pretend here's your saved model:

```
learn.save('stage-1-rn50')
```

Then we unfreeze:

```
learn.unfreeze()
```

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



Then we fit a little bit more. Now in this case, I'm fitting with my original dataset. But you could create a new data bunch with just the misclassified instances and go ahead and fit. The misclassified ones are likely to be particularly interesting. So you might want to fit at a slightly higher learning rate to make them really mean more or you might want to run them through a few more epochs. But it's exactly the same thing. You just call fit with your misclassified examples and passing in the correct classification. That should really help your model quite a lot.

There are various other tweaks you can do to this but that's the basic idea.

```
learn.fit_one_cycle(5, slice(1e-5, lr/5))
```

```
Total time: 05:48
epoch  train_loss  valid_loss  accuracy_thresh  fbeta
1      0.096917   0.089857   0.964909        0.923028 (01:09)
2      0.095722   0.087677   0.966341        0.924712 (01:09)
3      0.088859   0.085950   0.966813        0.926390 (01:09)
4      0.085320   0.083416   0.967663        0.927521 (01:09)
5      0.081530   0.082129   0.968121        0.928895 (01:09)
```

```
learn.save('stage-2-rn50')
```

Question: Could someone talk a bit more about the data block ideology? I'm not quite sure how the blocks are meant to be used. Do they have to be in a certain order? Is there any other library that uses this type of programming that I could look at? [44:01]

Yes, they do have to be in a certain order and it's basically the order that you see in [the example of use](#).

```
data = (ImageItemList.from_folder(path) #Where to find the data? -> in path and i
       .split_by_folder()           #How to split in train/valid? -> use the
       .label_from_folder()         #How to label? -> depending on the folder
       .add_test_folder()          #Optionally add a test set (here default
       .transform(tfms, size=64)    #Data augmentation? -> use tfms with a si
       .databunch())               #Finally? -> use the defaults for convers
```

- What kind of data do you have?
- Where does it come from?
- How do you split it?
- How do you label it?
- What kind of datasets do you want?
- Optionally, how do I transform it?
- How do I create a data bunch from?

They're the steps. We invented this API. I don't know if other people have independently invented it. The basic idea of a pipeline of things that dot into each other is pretty common in a number of places - not so much in Python, but you see it more in JavaScript. Although this kind of approach of each stage produces something slightly different, you tend to see it more in like ETL software (extraction transformation and loading software) where this particular stages in a pipeline. It's been inspired by a bunch of things. But all you need to know is to use this example to guide you, and then look up the documentation to see which particular kind of thing you want. In this case, the `ImageItemList`, you're actually not going to find the documentation of `ImageItemList` in datablocks documentation because this is specific to the vision application. So to then go and actually find out how to do something for your particular application, you would then go to look at text, vision, and so forth. That's where you can find out what are the datablock API pieces available for that application.

Of course, you can then look at the source code if you've got some totally new application. You could create your own "part" of any of these stages. Pretty much all of these functions are very few lines of code. Maybe we could look an example of one. Let's try.

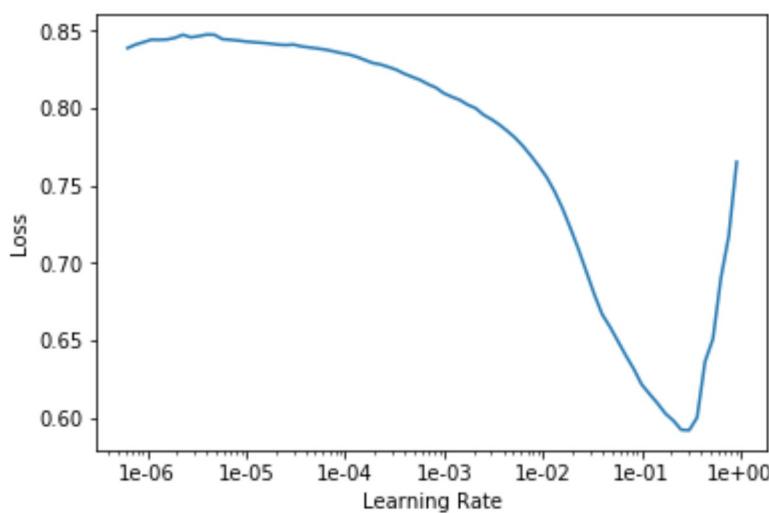
You can look at the documentation to see exactly what that does. As you can see, most fastai functions are no more than a few lines of code. They're normally pretty straightforward to see what are all the pieces there and how can you use them. It's probably one of these things that, as you play around with it, you'll get a good sense of how it all gets put together. But if during the week there are particular things where you're thinking I don't understand how to do this please let us know and we'll try to help you.

Question: What resources do you recommend for getting started with video? For example, being able to pull frames and submit them to your model. [47:39]

The answer is it depends. If you're using the web which I guess probably most of you will be then there's web API's that basically do that for you. So you can grab the frames with the web API and then they're just images which you can pass along. If you're doing a client side, I guess most people would tend to use OpenCV for that. But maybe during the week, people who are doing these video apps can tell us what have you used and found useful, and we can start to prepare something in the lesson wiki with a list of video resources since it sounds like some people are interested.

⌚ How to choose good learning rates [48:50]

One thing to notice here is that before we unfreeze you'll tend to get this shape pretty much all the time:



If you do your learning rate finder before you unfreeze. It's pretty easy - find the steepest slope, **not the bottom**. Remember, we're trying to find the bit where we can like slide down it quickly. So if you start at the bottom it's just gonna send you straight off to the end here.

Then we can call it again after you unfreeze, and you generally get a very different shape.

[49:24](#)

This is a little bit harder to say what to look for because it tends to be this kind of shape where you get a little bit of upward and then it kind of very gradual downward and then up here. So I tend to kind of look for just before it shoots up and go back about 10x as a kind of a rule of thumb. So 1e-5. That is what I do for the first half of my slice. And then for the second half of my slice, I normally do whatever learning rate are used for the the frozen part. So lr which was 0.01 kind of divided by five or ten. Somewhere around that. That's my rule of thumb:

- Look for the bit kind of at the bottom, find about 10x smaller, that's the number that I put as the first half of my slice.
- $lr/5$ or $lr/10$ is kind of what I put as the second half of my slice.

This is called discriminative learning rates as the course continues.

⌚ Making the model better [50:30](#)

How am I going to get this better? We want to get into the top 10% which is going to be about 0.929-ish. So we're not quite there (0.9288).

So here's the trick [51:01]. When I created my dataset, I put `size=128` and actually the images that Kaggle gave us are 256. I used the size of 128 partially because I wanted to experiment quickly. It's much quicker and easier to use small images to experiment. But there's a second reason. I now have a model that's pretty good at recognizing the contents of 128 by 128 satellite images. So what am I going to do if I now want to create a model that's pretty good at 256 by 256 satellite images? Why don't I use transfer learning? Why don't I start with the model that's good at 128 by 128 images and fine-tune that? So don't start again. That's actually going to be really interesting because if I trained quite a lot and I'm on the verge of overfitting then I'm basically creating a whole new dataset effectively - one where my images are twice the size on each axis right so four times bigger. So it's really a totally different data set as far as my convolutional neural networks concerned. So I got to lose all that overfitting. I get to start again. Let's keep our same learner but use a new data bunch where the data bunch is 256 by 256. That's why I actually stopped here before I created my data sets:

```
1 np.random.seed(42)
2 src = (ImageItemList.from_csv(path, 'train_v2.csv', folder='train-jpg'
3     .random_split_by_pct(0.2)
4     .label_from_df(sep=' '))

1 data = (src.transform(tfms, size=128)
2     .databunch().normalize(imagenet_stats))
```

Because I'm going to now take this this data source (`src`) and I'm going to create a new data bunch with 256 instead. So let's have a look at how we do that.

```
data = (src.transform(tfms, size=256)
    .databunch().normalize(imagenet_stats))
```

So here it is. Take that source, transform it with the same transforms as before but this time use size 256. That should be better anyway because this is going to be higher resolution images. But also I'm going to start with this kind of pre-trained model (I haven't got rid of my learner it's the same learner I had before).

I'm going to replace the data inside my learner with this new data bunch.

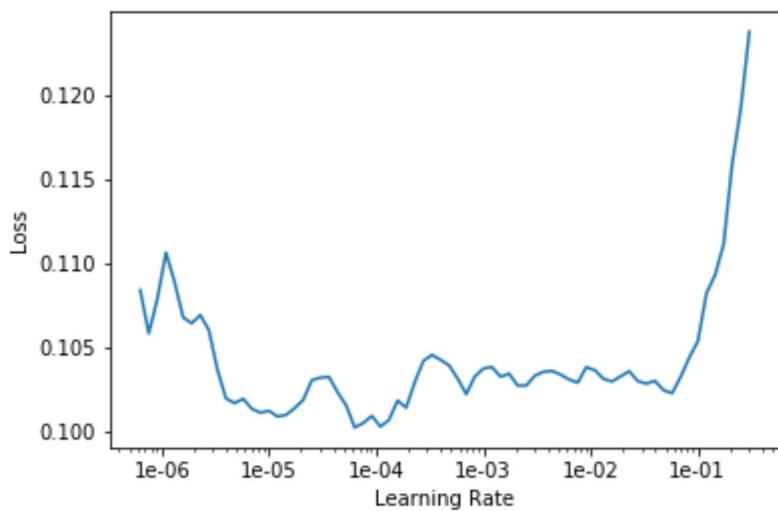
```
learn.data = data
data.train_ds[0][0].shape
```

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

```
learn.freeze()
```

Then I will freeze again (i.e. I'm going back to just training the last few layers) and I will do a new `lr_find()`.

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



Because I actually now have a pretty good model (it's pretty good for 128 by 128 so it's probably gonna be like at least okay for 256 by 256), I don't get that same sharp shape that I did before. But I can certainly see where it's way too high. So I'm gonna pick something well before where it's way too high. Again maybe 10x smaller. So here I'm gonna go `1e-2/2` - that seems well before it shoots up.

```
lr=1e-2/2
```

So let's fit a little bit more.

```
learn.fit_one_cycle(5, slice(lr))
```

```
Total time: 14:21  
epoch  train_loss  valid_loss  accuracy_thresh  fbeta  
1      0.088628   0.085883   0.966523       0.924035 (02:53)  
2      0.089855   0.085019   0.967126       0.926822 (02:51)  
3      0.083646   0.083374   0.967583       0.927510 (02:51)  
4      0.084014   0.081384   0.968405       0.931110 (02:51)  
5      0.083445   0.081085   0.968659       0.930647 (02:52)
```

We are frozen again so we're just training the last few layers and fit a little bit more. As you can see, I very quickly remember 0.928 was where we got to before after quite a few epochs. We're straight up there and suddenly we've passed 0.93. So we're now already into the top 10%. So we've hit our first goal. We're, at the very least, pretty confident at the problem of recognizing satellite imagery.

```
learn.save('stage-1-256-rn50')
```

But of course now, we can do the same thing as before. We can unfreeze and train a little more.

```
learn.unfreeze()
```

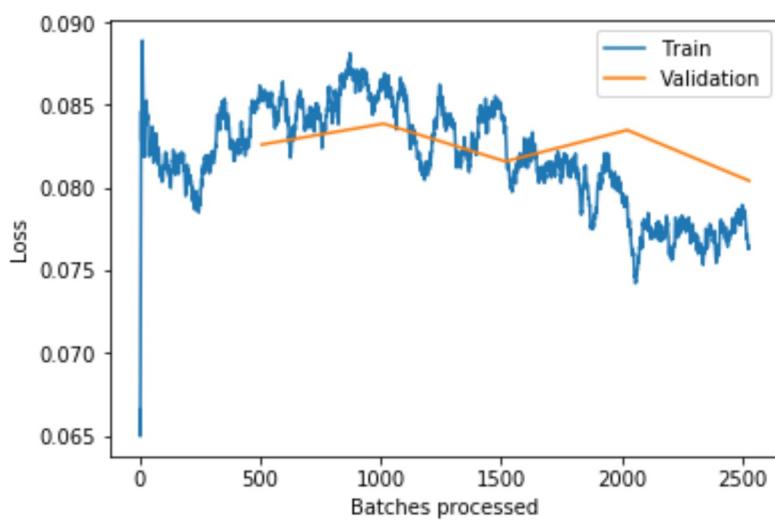
Again using the same kind of approach I described before, lr/5 on the right and even smaller one on the left.

```
learn.fit_one_cycle(5, slice(1e-5, lr/5))
```

```
Total time: 18:23
epoch  train_loss  valid_loss  accuracy_thresh  fbeta
1      0.083591   0.082895   0.968310       0.928210 (03:41)
2      0.088286   0.083184   0.967424       0.928812 (03:40)
3      0.083495   0.083084   0.967998       0.929224 (03:40)
4      0.080143   0.081338   0.968564       0.931363 (03:40)
5      0.074927   0.080691   0.968819       0.931414 (03:41)
```

Train a little bit more. 0.9314 so that's actually pretty good - somewhere around top 25ish. Actually when my friend Brendan and I entered this competition we came 22nd with 0.9315 and we spent (this was a year or two ago) months trying to get here. So using pretty much defaults with the minor tweaks and one trick which is the resizing tweak you can get right up into the top of the leaderboard of this very challenging competition. Now I should say we don't really know where we'd be - we would actually have to check it on the test set that Kaggle gave us and actually submit to the competition which you can do. You can do a late submission. So later on in the course, we'll learn how to do that. But we certainly know we're doing very well so that's great news.

```
learn.recorder.plot_losses()
```



```
learn.save('stage-2-256-rn50')
```

You can see as I kind of go along I tend to save things. You can name your models whatever you like but I just want to basically know is it before or after the unfreeze (stage 1 or 2), what size was I training on, what architecture was I training on. That way I could have always go back and experiment pretty easily. So that's planet. Multi label classification.

↪ Segmentation example: CamVid [56:31]

[Notebook](#)

The next example we're going to look at is this dataset called CamVid. It's going to be doing something called segmentation. We're going to start with a picture like the left:

and we're going to try and create a color-coded picture like the right where all of the bicycle pixels are the same color, all of the road line pixels are the same color, all of the tree pixels are the same color, all of the building pixels are the same color, the sky the same color, and so forth.

Now we're not actually going to make them colors, we're actually going to do it where each of those pixels has a unique number. In this case the top left is building, so I guess building is number 4, the top right is tree, so tree is 26, and so forth.

In other words, this single top left pixel, we're going to do a classification problem just like the pet's classification for the very top left pixel. We're going to say "What is that top left pixel? Is it bicycle, road lines, sidewalk, building?". Then, "What is the next pixel along?". So we're going to do a little classification problem for every single pixel in every single image. That's called segmentation.

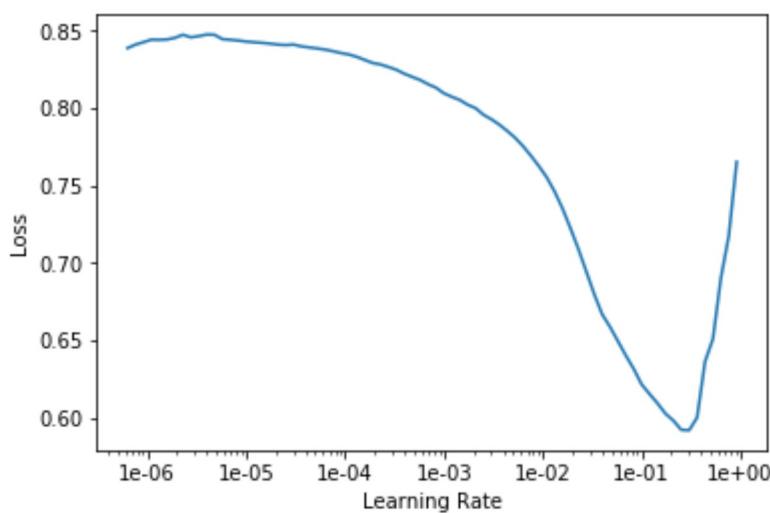
In order to build a segmentation model, you actually need to download or create a dataset where someone has actually labeled every pixel. As you can imagine, that's a lot of work, so you're probably not going to create your own segmentation datasets but you're probably going to download or find them from somewhere else.

This is very common in medicine and life sciences. If you're looking through slides at nuclei, it's very likely you already have a whole bunch of segmented cells and segmented nuclei. If you're in radiology, you probably already have lots of examples of segmented lesions and so forth. So there's a lot of different domain areas where there are domain-specific tools for creating these segmented images. As you could guess from this example, it's also very common in self-driving cars and stuff like that where you need to see what objects are around and where are they.

In this case, there's a nice dataset called CamVid which we can download and they have already got a whole bunch of images and segment masks prepared for us. Remember, pretty much all of the datasets that we have provided inbuilt URLs for, you can see their details at <https://course.fast.ai/datasets> and nearly all of them are academic datasets where some very kind people have gone to all of this trouble for us so that we can use this dataset and made it available for us to use. So if you do use one of these datasets for any kind of project, it would be very very nice if you were to go and find the citation and say "Thanks to these people for this dataset". Because they've provided it and all they're asking in return is for us to give them that credit. So here is the CamVid dataset and the citation (on our data sets page, that will link to the academic paper where it came from).

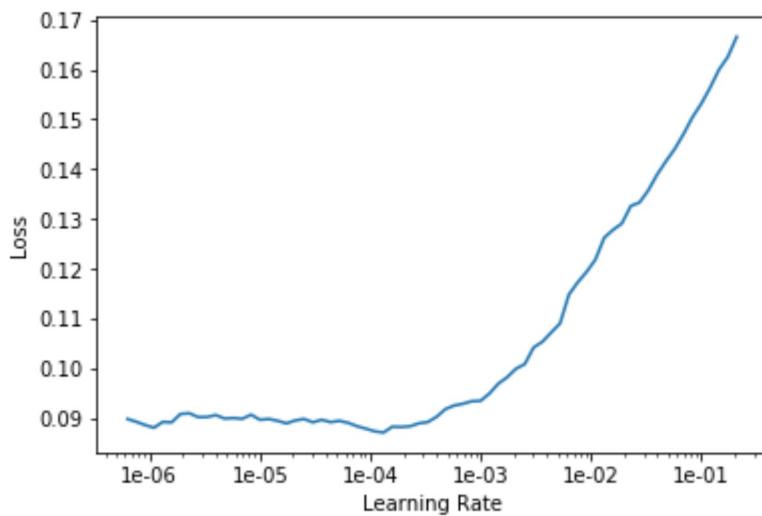
Question: Is there a way to use `learn.lr_find()` and have it return a suggested number directly rather than having to plot it as a graph and then pick a learning rate by visually inspecting that graph? (And there are a few other questions around more guidance on reading the learning rate finder graph) [1:00:26]

The short answer is no and the reason the answer is no is because this is still a bit more artisanal than I would like. As you can see, I've been saying how I read this learning rate graph depends a bit on what stage I'm at and what the shape of it is. I guess when you're just training the head (so before you unfreeze), it pretty much always looks like this:



And you could certainly create something that creates a smooth version of this, finds the sharpest negative slope and picked that. You would probably be fine nearly all the time.

But then for you know these kinds of ones, it requires a certain amount of experimentation:



But the good news is you can experiment. Obviously if the lines going up, you don't want it. Almost certainly at the very bottom point, you don't want it right there because you needed to be going downwards. But if you kind of start with somewhere around 10x smaller than that, and then also you could try another 10x smaller than that. Try a few numbers and find out which ones work best.

And within a small number of weeks, you will find that you're picking the best learning rate most of the time. So at this stage, it still requires a bit of playing around to get a sense of the different kinds of shapes that you see and how to respond to them. Maybe by the time this video comes out, someone will have a pretty reliable auto learning rate finder. We're not there yet. It's probably not a massively difficult job to do. It would be an interesting project - collect a whole bunch of different datasets, maybe grab all the datasets from our datasets page, try and come up with some simple heuristic, compare it to all the different lessons I've shown. It would be a really fun project to do. But at the moment, we don't have that. I'm sure it's possible but we haven't got them.

⌚ Image Segmentation [1:03:05]

So how do we do image segmentation? The same way we do everything else. Basically we're going to start with some path which has got some information in it of some sort.

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

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

So I always start by un-tarring my data, do an `ls`, see what I was given. In this case there's a label folder called `labels` and a folder called `images`, so I'll create paths for each of those.

```
path = untar_data(URLs.CAMVID)
path.ls()

[PosixPath('/home/ubuntu/course-v3/nbs/dl1/data/camvid/images'),
 PosixPath('/home/ubuntu/course-v3/nbs/dl1/data/camvid/codes.txt'),
 PosixPath('/home/ubuntu/course-v3/nbs/dl1/data/camvid/valid.txt'),
 PosixPath('/home/ubuntu/course-v3/nbs/dl1/data/camvid/labels')]
```

```
path_lbl = path/'labels'
path_img = path/'images'
```

We'll take a look inside each of those.

```
fnames = get_image_files(path_img)
fnames[:3]
```

```
[PosixPath('/home/ubuntu/course-v3/nbs/dl1/data/camvid/images/0016E5_08370.png'),  
 PosixPath('/home/ubuntu/course-v3/nbs/dl1/data/camvid/images/Seq05VD_f04110.png'),  
 PosixPath('/home/ubuntu/course-v3/nbs/dl1/data/camvid/images/0001TP_010170.png')]
```

```
lbl_names = get_image_files(path_lbl)  
lbl_names[:3]
```

```
[PosixPath('/home/ubuntu/course-v3/nbs/dl1/data/camvid/labels/0016E5_01890_P.png'),  
 PosixPath('/home/ubuntu/course-v3/nbs/dl1/data/camvid/labels/Seq05VD_f00330_P.png'),  
 PosixPath('/home/ubuntu/course-v3/nbs/dl1/data/camvid/labels/Seq05VD_f01140_P.png')]
```

You can see there's some kind of coded file names for the images and some kind of coded file names for the segment masks. Then you kind of have to figure out how to map from one to the other. Normally, these kind of datasets will come with a README you can look at or you can look at their website.

```
img_f = fnames[0]  
img = open_image(img_f)  
img.show(figsize=(5,5))
```

```
get_y_fn = lambda x: path_lbl/f'{x.stem}_P{x.suffix}'
```

Often it's obvious. In this case I just guessed. I thought it's probably the same thing + _P , so I created a little function that basically took the filename and added the _P and put it in the different place (path_lbl) and I tried opening it and I noticed it worked.

```
mask = open_mask(get_y_fn(img_f))  
mask.show(figsize=(5,5), alpha=1)
```

So I've created this little function that converts from the image file names to the equivalent label file names. I opened up that to make sure it works. Normally, we use `open_image` to open a file and then you can go `.show` to take a look at it, but as we described, this is not a usual image file that contains integers. So you have to use `open_masks` rather than `open_image` because we want to return integers not floats. fastai knows how to deal with masks, so if you go `mask.show`, it will automatically color code it for you in some appropriate way. That's why we said `open_masks`.

```
src_size = np.array(mask.shape[1:])
src_size,mask.data

(array([720, 960]), tensor([[[30, 30, 30, ..., 4, 4, 4],
   [30, 30, 30, ..., 4, 4, 4],
   [30, 30, 30, ..., 4, 4, 4],
   ...,
   [17, 17, 17, ..., 17, 17, 17],
   [17, 17, 17, ..., 17, 17, 17],
   [17, 17, 17, ..., 17, 17, 17]]]))
```

We can kind of have a look inside, look at the data, see what the size is. So there's 720 by 960. We can take a look at the data inside, and so forth. The other thing you might have noticed is that they gave us a file called `codes.txt` and a file called `valid.txt`.

```
codes = np.loadtxt(path/'codes.txt', dtype=str); codes

array(['Animal', 'Archway', 'Bicyclist', 'Bridge', 'Building', 'Car',
'CartLuggagePram', 'Child', 'Column_Pole',
'Fence', 'LaneMkgsDriv', 'LaneMkgsNonDriv', 'Misc_Text',
'MotorcycleScooter', 'OtherMoving', 'ParkingBlock',
'Pedestrian', 'Road', 'RoadShoulder', 'Sidewalk', 'SignSymbol', 'Sky',
'SUVPickupTruck', 'TrafficCone',
'TrafficLight', 'Train', 'Tree', 'Truck_Bus', 'Tunnel',
'VegetationMisc', 'Void', 'Wall'], dtype='<U17')
```

`code.txt` contains a list telling us that, for example, number 4 is `building`. Just like we had grizzlies, black bears, and teddies, here we've got the coding for what each one of these pixels means.

⌚ Creating a data bunch [1:05:53]

To create a data bunch, we can go through the data block API and say:

- We've got a list of image files that are in a folder.
- We then need to split into training and validation. In this case I don't do it randomly

because the pictures they've given us are frames from videos. If I did them randomly I would be having two frames next to each other: one in the validation set, one in the training set. That would be far too easy and treating. So the people that created this dataset actually gave us a list of file names (`valid.txt`) that are meant to be in your validation set and they are non-contiguous parts of the video. So here's how you can split your validation and training using a file name file.

- We need to create labels which we can use that `get_y_fn` (get Y file name function) we just created .

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

src = (SegmentationItemList.from_folder(path_img)
       .split_by_fname_file('../valid.txt')
       .label_from_func(get_y_fn, classes=classes))
```

From that, I can create my datasets.

So I actually have a list of class names. Often with stuff like the planet dataset or the pets dataset, we actually have a string saying this is a pug, this is a ragdoll, or this is a birman, or this is cloudy or whatever. In this case, you don't have every single pixel labeled with an entire string (that would be incredibly inefficient). They're each labeled with just a number and then there's a separate file telling you what those numbers mean. So here's where we get to tell the data block API this is the list of what the numbers mean. So these are the kind of parameters that the data block API gives you.

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

Here's our transformations. Here's an interesting point. Remember I told you that, for example, sometimes we randomly flip an image? What if we randomly flip the independent variable image but we don't also randomly flip the target mask? Now I'm not matching anymore. So we need to tell fastai that I want to transform the Y (X is our independent variable, Y is our dependent) - I want to transform the Y as well. So whatever you do to the X, I also want you to do to the Y (`tfm_y=True`). There's all these little parameters that we can play with.

I can create our data bunch. I'm using a smaller batch size (`bs=8`) because, as you can imagine, I'm creating a classifier for every pixel, that's going to take a lot more GPU right. I found a batch size of 8 is all I could handle. Then normalize in the usual way.

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

This is quite nice. Because fastai knows that you've given it a segmentation problem, when you call show batch, it actually combines the two pieces for you and it will color code the photo. Isn't that nice? So this is what the ground truth data looks.

⌚ Training [1:09:00]

Once we've got that, we can go ahead and

- Create a learner. I'll show you some more details in a moment.
- Call `lr_find`, find the sharpest bit which looks about 1e-2.
- Call `fit` passing in `slice(lr)` and see the accuracy.
- Save the model.
- Unfreeze and train a little bit more.

That's the basic idea.

```
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
# metrics=accuracy

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

lr_find(learn)
learn.recorder.plot()

lr=1e-2
```

```
learn.fit_one_cycle(10, slice(lr))
```

```
Total time: 02:46
epoch  train_loss  valid_loss  acc_camvid
1      1.537235   0.785360   0.832015   (00:20)
2      0.905632   0.677888   0.842743   (00:15)
3      0.755041   0.759045   0.844444   (00:16)
4      0.673628   0.522713   0.854023   (00:16)
5      0.603915   0.495224   0.864088   (00:16)
6      0.557424   0.433317   0.879087   (00:16)
7      0.504053   0.419078   0.878530   (00:16)
8      0.457378   0.371296   0.889752   (00:16)
9      0.428532   0.347722   0.898966   (00:16)
10     0.409673   0.341935   0.901897   (00:16)
```

```
learn.save('stage-1')
```

```
learn.load('stage-1');
```

```
learn.unfreeze()
```

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

```
lrs = slice(1e-5, lr/5)
```

```
learn.fit_one_cycle(12, lrs)
```

```
Total time: 03:36
epoch  train_loss  valid_loss  acc_camvid
1      0.399582   0.338697   0.901930   (00:18)
2      0.406091   0.351272   0.897183   (00:18)
3      0.415589   0.357046   0.894615   (00:17)
4      0.407372   0.337691   0.904101   (00:18)
5      0.402764   0.340527   0.900326   (00:17)
6      0.381159   0.317680   0.910552   (00:18)
7      0.368179   0.312087   0.910121   (00:18)
8      0.358906   0.310293   0.911405   (00:18)
```

9	0.343944	0.299595	0.912654	(00:18)
10	0.332852	0.305770	0.911666	(00:18)
11	0.325537	0.294337	0.916766	(00:18)
12	0.320488	0.295004	0.916064	(00:18)

Question: Could you use unsupervised learning here (pixel classification with the bike example) to avoid needing a human to label a heap of images [1:10:03]

Not exactly unsupervised learning, but you can certainly get a sense of where things are without needing these kind of labels. Time permitting, we'll try and see some examples of how to do that. You're certainly not going to get as such a quality and such a specific output as what you see here though. If you want to get this level of segmentation mask, you need a pretty good segmentation mask ground truth to work with.

Question: Is there a reason we shouldn't deliberately make a lot of smaller datasets to step up from in tuning? let's say 64x64, 128x128, 256x256, etc... [1:10:51]

Yes, you should totally do that. It works great. This idea, it's something that I first came up with in the course a couple of years ago and I thought it seemed obvious and just presented it as a good idea, then I later discovered that nobody had really published this before. And then we started experimenting with it. And it was basically the main tricks that we use to win the DAWN Bench ImageNet training competition.

Not only was this not standard, but nobody had heard of it before. There's been now a few papers that use this trick for various specific purposes but it's still largely unknown. It means that you can train much faster, it generalizes better. There's still a lot of unknowns about exactly how small, how big, and how much at each level and so forth. We call it "progressive resizing". I found that going much under 64 by 64 tends not to help very much. But yeah, it's a great technique and I definitely try a few different sizes.

Question: [1:12:35] What does accuracy mean for pixel wise segmentation? Is it

#correctly classified pixels / #total number of pixels ?

Yep, that's it. So if you imagined each pixel was a separate object you're classifying, it's exactly the same accuracy. So you actually can just pass in `accuracy` as your metric, but in this case, we actually don't. We've created a new metric called `acc_cavmid` and the reason for that is that when they labeled the images, sometimes they labeled a pixel as `Void`. I'm not quite sure why but some of the pixels are `Void`. And in the CamVid paper, they say when you're reporting accuracy, you should remove the void pixels. So we've created accuracy CamVid. So all metrics take the actual output of the neural net (i.e. that's the `input` to the metric) and the target (i.e. the labels we are trying to predict).

We then basically create a mask (we look for the places where the target is not equal to `Void`) and then we just take the input, do the `argmax` as per usual, but then we just grab those that are not equal to the void code. We do the same for the target and we take the mean, so it's just a standard accuracy.

It's almost exactly the same as the accuracy source code we saw before with the addition of this mask. This quite often happens. The particular Kaggle competition metric you're using or the particular way your organization scores things, there's often little tweaks you have to do. And this is how easy it is. As you'll see, to do this stuff, the main thing you need to know pretty well is how to do basic mathematical operations in PyTorch so that's just something you kind of need to practice.

Question: I've noticed that most of the examples and most of my models result in a training loss greater than the validation loss. What are the best ways to correct that? I should add that this still happens after trying many variations on number of epochs and learning rate. [1:15:03]

Remember from last week, if your training loss is higher than your validation loss then you're **underfitting**. It definitely means that you're underfitting. You want your training loss to be lower than your validation loss. If you're underfitting, you can:

- Train for longer.
- Train the last bit at a lower learning rate.

But if you're still under fitting, then you're going to have to decrease regularization. We haven't talked about that yet. In the second half of this part of the course, we're going to be talking quite a lot about regularization and specifically how to avoid overfitting or underfitting by using regularization. If you want to skip ahead, we're going to be learning about:

- weight decay
- dropout
- data augmentation

They will be the key things that are we talking about.

⌚ U-Net [1:16:24]

For segmentation, we don't just create (use) a convolutional neural network. We can, but actually an architecture called U-Net turns out to be better.

This is what a U-Net looks like. This is from the [University website](#) where they talk about the U-Net. So we'll be learning about this both in this part of the course and in part two if you do it. But basically this bit down on the left hand side is what a normal convolutional neural network looks like. It's something which starts with a big image and gradually makes it smaller and smaller until eventually you just have one prediction. What a U-Net does is it then takes that and makes it bigger and bigger and bigger again, and then it takes every stage of the downward path and copies it across, and it creates this U shape.

It's was originally actually created/published as a biomedical image segmentation method. But it turns out to be useful for far more than just biomedical image segmentation. It was presented at MICCAI which is the main medical imaging conference, and as of just yesterday, it actually just became the most cited paper of all time from that conference. So it's been incredibly useful - over 3,000 citations.

You don't really need to know any details at this stage. All you need to know is if you want to create a segmentation model, you want to be saying `Learner.create_unet` rather than `create_cnn`. But you pass it the normal stuff: their data bunch, architecture, and some metrics.

Having done that, everything else works the same.

⌚ A little more about `learn.recorder` [1:18:54]

Here's something interesting. `learn.recorder` is where we keep track of what's going on during training. It's got a number nice methods, one of which is `plot_losses`.

```
learn.recorder.plot_losses()
```

```
learn.recorder.plot_lr()
```

This plots your training loss and your validation loss. Quite often, they actually go up a bit before they go down. Why is that? That's because (you can also plot your learning rate over time and you'll see that) the learning rate goes up and then it goes down. Why is that? Because we said `fit_one_cycle`. That's what fit one cycle does. It actually makes the learning rate start low, go up, and then go down again.

Why is that a good idea? To find out why that's a good idea, let's first of all look at a really cool project done by José Fernández Portal during the week. He took our gradient descent demo notebook and actually plotted the weights over time, not just the ground truth and model over time. He did it for a few different learning rates.

Remember we had two weights we were doing basically $y = ax + b$ or in his nomenclature $y = w_0x + w_1$.

We can actually look and see what happens to those weights over time. And we know this is the correct answer (marked with red X). A learning rate of 0.1, they're kind of like slides on in here and you can see that it takes a little bit of time to get to the right point. You can see the loss improving.

At a higher learning rate of 0.7, you can see that the model jumps to the ground truth really quickly. And you can see that the weights jump straight to the right place really quickly.

What if we have a learning rate that's really too high? You can see it takes a very very long time to get to the right point.

Or if it's really too high, it diverges.

So you can see why getting the right learning rate is important. When you get the right learning rate, it zooms into the best spot very quickly.

Now as you get closer to the final spot, something interesting happens which is that you really want your learning rate to decrease because you're getting close to the right spot.

So what actually happens is (I can only draw 2d sorry), you don't generally have some kind of loss function surface that looks like that (remember there's lots of dimensions), but it actually tends to look bumpy like that. So you want a learning rate that's like high enough to jump over the bumps, but once you get close to the best answer, you don't want to be just jumping backwards and forwards between bumps. You want your learning rate to go down so that as you get closer, you take smaller and smaller steps. That's why we want our learning rate to go down at the end.

This idea of decreasing the learning rate during training has been around forever. It's just called **learning rate annealing**. But the idea of gradually increasing it at the start is much more recent and it mainly comes from a guy called Leslie Smith ([meetup with Leslie Smith](#)).

Loss function surfaces tend to have flat areas and bumpy areas. If you end up in the bottom of a bumpy area, that solution will tend not to generalize very well because you've found a solution that's good in that one place but it's not very good in other places. Where else if you found one in the flat area, it probably will generalize well because it's not only good in that one spot but it's good to kind of around it as well.

If you have a really small learning rate, it'll tend to kind of plod down and stick in these places. But if you gradually increase the learning rate, then it'll kind of like jump down and as the learning rate goes up, it's going to start going up again like this. Then the learning rate is now going to be up here, it's going to be bumping backwards and forwards. Eventually the learning rate starts to come down again, and it'll tend to find its way to these flat areas.

So it turns out that gradually increasing the learning rate is a really good way of helping the model to explore the whole function surface, and try and find areas where both the loss is low and also it's not bumpy. Because if it was bumpy, it would get kicked out again. This allows us to train at really high learning rates, so it tends to mean that we solve our problem much more quickly, and we tend to end up with much more generalizable solutions.

⌚ What you are looking for in `plot_losses` [1:25:01]

If you call `plot_losses` and find that it's just getting a little bit worse and then it gets a lot better you've found a really good maximum learning rate.

So when you actually call `fit` one cycle, you're not actually passing in a learning rate. You're actually passing in a maximum learning rate. If it's kind of always going down, particularly after you unfreeze, that suggests you could probably bump your learning rates up a little bit - because you really want to see this kind of shape. It's going to train faster and generalize better. You'll tend to particularly see it in the validation set (the orange is the validation set). Again, the difference between kind of knowing this theory and being able to do it, is looking at lots of these pictures. So after you train stuff, type `learn.recorder.` and hit tab, and see what's in there - particularly the things that start with "plot" and start getting a sense of what are these pictures looking like when you're getting good results. Then try making the learning rate much higher, try making it much lower, more epochs, less epochs, and get a sense for what these look like.

In this case, we used the size (in our transforms) of the `original image size/2`. These two slashes in Python means integer divide because obviously we can't have half pixel amounts in our sizes. We use the batch size of 8. Now I found that fits on my GPU, it might not fit on yours. If it doesn't, you can just decrease the batch size down to 4.

This isn't really solving the problem because the problem is to segment all of the pixels - not half of the pixels. So I'm going to use the same trick that I did last time which is I'm now going to put the size up to the full size of the source images which means I now have to halve my batch size otherwise I'll run out of GPU memory.

```
size = src_size
bs=4

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

learn = Learner.create_unet(data, models.resnet34, metrics=metrics)

learn.load('stage-2');
```

I can either say `learn.data = data` but I actually found it had a lot of trouble with GPU memory, so I generally restarted my kernel, came back here, created a new learner, and loaded up the weights that I saved last time.

The key thing is that this learner now has the same weights that I had before, but the data is now the full image size.

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

lr=1e-3

learn.fit_one_cycle(10, slice(lr))
```

```
Total time: 08:44
epoch  train_loss  valid_loss  acc_camvid
1      0.454597   0.349557   0.900428   (01:02)
2      0.418897   0.351502   0.897495   (00:51)
3      0.402104   0.330255   0.906775   (00:50)
4      0.385497   0.313330   0.911832   (00:51)
5      0.359252   0.297264   0.916108   (00:52)
6      0.335910   0.297875   0.917553   (00:50)
7      0.336133   0.305602   0.913439   (00:51)
8      0.321016   0.305374   0.914063   (00:51)
9      0.311554   0.299226   0.915997   (00:51)
10     0.308389   0.301060   0.915253   (00:51)
```

```
learn.save('stage-1-big')
```

```
learn.load('stage-1-big');
```

```
learn.unfreeze()
```

```
lrs = slice(1e-6,lr)
```

```
learn.fit_one_cycle(10, lrs, wd=1e-3)
```

```
Total time: 09:30
epoch  train_loss  valid_loss  acc_camvid
1      0.323283   0.300749   0.915948   (00:56)
2      0.329482   0.290447   0.918337   (00:56)
3      0.324378   0.298494   0.920271   (00:57)
4      0.316414   0.296469   0.918053   (00:56)
5      0.305226   0.284694   0.920893   (00:57)
6      0.301774   0.306676   0.914202   (00:57)
7      0.279722   0.285487   0.919991   (00:57)
8      0.269306   0.285219   0.920963   (00:57)
9      0.260325   0.284758   0.922026   (00:57)
10     0.251017   0.285375   0.921562   (00:57)
```

```
learn.save('stage-2-big')
```

```
learn.load('stage-2-big')
```

```
learn.show_results()
```

You can go `learn.show_results()` to see how your predictions compare to the ground truth, and they really look pretty good.

How good is pretty good? An accuracy of 92.15%, the best paper I know of for segmentation was a paper called [The One Hundred Layers Tiramisu](#) which developed a convolutional dense net came out about two years ago. After I trained this today, I went back and looked at the paper to find their state-of-the-art accuracy and their best was 91.5% and we got 92.1%. I don't know if better results have come out since this paper, but I remember when this paper came out and it was a really big deal. I said "Wow, this is an exceptionally good segmentation result." When you compare it to the previous bests that they compared it to, it was a big step up.

In last year's course, we spent a lot of time re-implementing the hundred layers tiramisu. Now with our totally default fastai class, and it's easily beating 91.5%. I also remember I had to train for hours and hours. Where else, today's version, I trained in minutes. So this is a super strong architecture for segmentation.

I'm not going to promise that this is the definite state-of-the-art today, because I haven't done a complete literature search to see what's happened in the last two years. But it's certainly beating the world's best approach the last time I looked into this which was in last year's course basically. So these are all the little tricks we've picked up along the way in terms of how to train things well: things like using the pre-trained model and the one cycle convergence. All these little tricks they work extraordinarily well.

We actually haven't published the paper on the exact details of how this variation of the U-Net works - there's a few little tweaks we do, but if you come back for part 2, we'll be going into all of the details about how we make this work so well. But for you, all you have to know at this stage is that you can say `learner.create_unet` and you should get great results also.

⌚ Another trick: Mixed precision training [1:30:59]

There's another trick you can use if you're running out of memory a lot. You can actually do something called mixed precision training. Mixed precision training means that (for those of you that have done a little bit of computer science) instead of using single precision floating point numbers, you can do most of the calculations in your model with half precision floating point numbers - so 16 bits instead of 32 bits. The very idea of this has only been around for the last couple of years - in terms of like hardware that actually does this reasonably quickly. Then fastai library, I think, is the first and probably still the only one that makes it actually easy to use this.

If you add `to_fp16()` on the end of any learner call, you're actually going to get a model that trains in 16-bit precision. Because it's so new, you'll need to have the most recent CUDA drivers and all that stuff for this even to work. When I tried it this morning on some of the platforms, it just killed the kernel, so you need to make sure you've got the most recent drivers. If you've got a really recent GPU like 2080Ti, not only will it work, but it'll work about twice as fast as otherwise. The reason I'm mentioning it is that it's going to use less GPU RAM, so even if you don't have a 2080Ti, you'll probably find that things that didn't fit into your GPU without this, do fit in.

I actually have never seen people use mixed precision floating point for segmentation before, just for a bit of a laugh I tried it and actually discovered that I got even better result. I only found this this morning so I don't have anything more to add here rather than quite often when you make things a little bit less precise in deep learning, it generalizes a little bit better. I've never seen a 92.5% accuracy on CamVid before, so not only will this be faster, you'll be able to use bigger batch sizes, but you might even find like I did that you get an even better result. So that's a cool little trick.

You just need to make sure that every time you create a learner you add this `to_fp16()`. If your kernel dies, it probably means you have slightly out of date CUDA drivers or maybe even a too old graphics card. I'm not sure exactly which cards support FP16.

⌚ Regression with BIWI head pose dataset [1:34:03]

[lesson3-head-pose.ipynb](#)

Two more before we kind of rewind. The first one I'm going to show you is an interesting data set called the [BIWI head pose dataset](#). Gabriele Fanelli was kind enough to give us permission to use this in the class. His team created this cool dataset.

Here's what the data set looks like. It's actually got a few things in it. We're just going to do a simplified version, and one of the things they do is they have a dot saying this is the center of the face. So we're going to try and create a model that can find this dot on the face.

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

from fastai import *
```

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

For this dataset, there's a few dataset specific things we have to do which I don't really even understand but I just know from the readme that you have to. They use some kind of depth sensing camera, I think they actually use Xbox Kinect.

```
path = untar_data(URLs.BIWI_HEAD_POSE)
```

There's some kind of calibration numbers that they provide in a little file which I had to read in:

```
cal = np.genfromtxt(path/'01'/'rgb.cal', skip_footer=6); cal
```

```
array([[517.679,    0.    , 320.    ],
       [    0.    , 517.679, 240.5  ],
       [    0.    ,    0.    ,    1.    ]])
```

```
fname = '09/frame_00667_rgb.jpg'
```

```
def img2txt_name(f): return path/f'{str(f)[:-7]}pose.txt'
```

```
img = open_image(path/fname)
img.show()
```

```
ctr = np.genfromtxt(img2txt_name(fname), skip_header=3); ctr
```

```
array([187.332 , 40.3892, 893.135 ])
```

Then they provided a little function that you have to use to take their coordinates to change it from this depth sensor calibration thing to end up with actual coordinates.

```
def convert_biwi(coords):
    c1 = coords[0] * cal[0][0]/coords[2] + cal[0][2]
    c2 = coords[1] * cal[1][1]/coords[2] + cal[1][2]
    return tensor([c2,c1])
```

```

def get_ctr(f):
    ctr = np.genfromtxt(img2txt_name(f), skip_header=3)
    return convert_biwi(ctr)

def get_ip(img, pts): return ImagePoints(FlowField(img.size, pts), scale=True)

```

So when you open this and you see these conversion routines, I'm just doing what they told us to do basically. It's got nothing particularly to do with deep learning to end up with this red dot.

```

get_ctr(fname)

tensor([263.9104, 428.5814])

ctr = get_ctr(fname)
img.show(y=get_ip(img, ctr), figsize=(6, 6))

```

The interesting bit really is where we create something which is not an image or an image segment but an image points. We'll mainly learn about this later in the course, but basically image points use this idea of coordinates. They're not pixel values, they're XY coordinates (just two numbers).

Here's an example for a particular image file name (`09/frame_00667_rgb.jpg`). The coordinates of the centre of the face are `[263.9104, 428.5814]` . So there's just two numbers which represent whereabouts on this picture is the center of the face. So if we're going to create a model that can find the center of a face, we need a neural network that spits out two numbers. But note, this is not a classification model. These are not two numbers that you look up in a list to find out that they're road or building or ragdoll cat or whatever. They're actual locations.

So far, everything we've done has been a classification model - something that created labels or classes. This, for the first time, is what we call a regression model. A lot of people think regression means linear regression, it doesn't. Regression just means any kind of model where your output is some continuous number or set of numbers. So we need to create an image regression model (i.e. something that can predict these two numbers). How do you do that? Same way as always.

```

data = (ImageItemList.from_folder(path)
        .split_by_valid_func(lambda o: o.parent.name=='13')
        .label_from_func(get_ctr, label_cls=PointsItemList)
        .transform(get_transforms(), tfm_y=True, size=(120, 160)))

```

```
.databunch().normalize(imagenet_stats)
)
```

We can actually just say:

- I've got a list of image files.
- It's in a folder.
- I'm going to split it according to some function. So in this case, the files they gave us are from videos. So I picked just one folder (13) to be my validation set (i.e. a different person). So again, I was trying to think about how do I validate this fairly, so I said the the fair validation would be to make sure that it works well on a person that it's never seen before. So my validation set is all going to be a particular person.
- I want to label them using this function that we wrote that basically does the stuff that the readme says to grab the coordinates out of their text files. So that's going to give me the two numbers for every one.
- Create a dataset. This data set, I just tell it what kind of data set it is - they're going to be a set of points of specific coordinates.
- Do some transforms. Again, I have to say `tfm_y=True` because that red dot needs to move if I flip or rotate or warp.
- Pick some size. I just picked a size that's going to work pretty quickly.
- Create a data bunch.
- Normalize it.

```
data.show_batch(3, figsize=(9,6))
```

I noticed that their red dots don't always seem to be quite in the middle of the face. I don't know exactly what their internal algorithm for putting dots on. It sometimes looks like it's meant to be the nose, but sometimes it's not quite the nose. Anyway it's somewhere around the center of the face or the nose.

⌚ Create a regression model [1:38:59]

So how do we create a model? We create a CNN. We're going to be learning a lot about loss functions in the next few lessons, but basically the loss function is that number that says how good is the model. For classification, we use this loss function called cross-entropy loss which says basically "Did you predict the correct class and were you confident of that prediction?" We can't use that for regression, so instead we use something called mean squared error. If you remember from last lesson, we actually implemented mean squared error from scratch. It's just the difference between the two, squared, and added up together.

```
learn = create_cnn(data, models.resnet34)
learn.loss_func = MSELossFlat()
```

So we need to tell it this is not classification so we have to use mean squared error.

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

```
lr = 2e-2
```

```
learn.fit_one_cycle(5, slice(lr))
```

```
Total time: 07:28
epoch  train_loss  valid_loss
1      0.043327   0.010848   (01:34)
2      0.015479   0.001792   (01:27)
3      0.006021   0.001171   (01:28)
4      0.003105   0.000521   (01:27)
5      0.002425   0.000381   (01:29)
```

Once we've created the learner, we've told it what loss function to use, we can go ahead and do `lr_find`, then `fit` and you can see here within a minute and a half our mean squared error is 0.0004.

The nice thing is about mean squared error, that's very easy to interpret. We're trying to predict something which is somewhere around a few hundred, and we're getting a squared error on average of 0.0004. So we can feel pretty confident that this is a really good model. Then we can look at the results:

```
learn.show_results()
```

It's doing nearly perfect job. That's how you can do image regression models. Anytime you've got something you're trying to predict which is some continuous value, you use an approach that's something like this.

↪ IMDB [1:41:07]

Last example before we look at more foundational theory stuff, NLP. Next week, we're going to be looking at a lot more NLP, but let's now do the same thing but rather than creating a classification of pictures, let's try and classify documents. We're going to go through this in a lot more detail next week, but let's do the quick version.

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

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

Rather than importing from `fastai.vision`, I now import, for the first time, from `fastai.text`. That's where you'll find all the application specific stuff for analyzing text documents.

In this case, we're going to use a dataset called IMDB. IMDB has lots of movie reviews. They're generally about a couple of thousand words, and each movie review has been classified as either negative or positive.

```
path = untar_data(URLs.IMDB_SAMPLE)  
path.ls()  
  
[PosixPath('/home/jhoward/.fastai/data/imdb_sample/texts.csv'),  
 PosixPath('/home/jhoward/.fastai/data/imdb_sample/models')]
```

It's just in a CSV file, so we can use pandas to read it and we can take a little look.

```
df = pd.read_csv(path/'texts.csv')  
df.head()
```

	label	text	is_valid
0	negative	Un-bleeping-believable! Meg Ryan doesn't even ...	False
1	positive	This is a extremely well-made film. The acting...	False
2	negative	Every once in a long while a movie will come a...	False
3	positive	Name just says it all. I watched this movie wi...	False
4	negative	This movie succeeds at being one of the most u...	False

```
df['text'][1]
```

'This is a extremely well-made film. The acting, script and camera-work are all first-rate. The music is good, too, though it is mostly early in the film, when things are still relatively cheery. There are no really superstars in the cast, though several faces will be familiar. The entire cast does an excellent job with the script.

But it is hard to watch, because there is no good end to a situation like the one presented. It is now fashionable to blame the British for setting Hindus and Muslims against each other, and then cruelly separating them into two countries. There is some merit in this view, but it's also true that no one forced Hindus and Muslims in the region to mistreat each other as they did around the time of partition. It seems more likely that the British simply saw the tensions between the religions and were clever enough to exploit them to their own ends.

The result is that there is much cruelty and inhumanity in the situation and this is very unpleasant to remember and to see on the screen. But it is never painted as a black-and-white case. There is baseness and nobility on both sides, and also the hope for change in the younger generation.

There is redemption of a sort, in the end, when Puro has to make a hard choice between a man who has ruined her life, but also truly loved her, and her family which has disowned her, then later come looking for her. But by that point, she has no option that is without great pain for her.

This film carries the message that both Muslims and Hindus have their grave faults, and also that both can be dignified and caring people. The reality of partition makes that realisation all the more wrenching, since there can never be real reconciliation across the India/Pakistan border. In that sense, it is similar to "Mr & Mrs Iyer".

In the end, we were glad to have seen the film, even though the resolution was heartbreakng. If the UK and US could deal with their own histories of racism with this kind of frankness, they would certainly be better off.'

Basically as per usual, we can either use factory methods or the data block API to create a data bunch. So here's the quick way to create a data bunch from a CSV of texts.

```
data_lm = TextDataBunch.from_csv(path, 'texts.csv')
```

At this point I could create a learner and start training it, but we're going to show you a little bit more detail which we mainly going to look at next week. The steps that actually happen when you create these data bunches, there's a few steps:

1. **Tokenization:** it takes those words and converts them into a standard form of tokens. Basically each token represents a word.

But it does things like, see how "didn't" has been turned here into two separate words (did and n't)? And everything has been lowercased. See how "you're" has been turned into two separate words (you and 're)? So tokenization is trying to make sure that each "token" (i.e. each thing that we've got with spaces around it) represents a single linguistic concept. Also it finds words that are really rare (e.g. really rare names) and replaces them with a special token called unknown (xxunk). Anything's starting with xx in fastai is some special token. This is tokenization, so we end up with something where we've got a list of tokenized words. You'll also see that things like punctuation end up with spaces around them to make sure that they're separate tokens.

2. Numericalization: The next thing we do is we take a complete unique list of all of the possible tokens - that's called the vocab which gets created for us.

```
data.vocab.itos[:10]
```

```
['xxunk', 'xxpad', 'the', ',', '.', 'and', 'a', 'of', 'to', 'is']
```

So here is every possible token (the first ten of them) that appear in all of the movie reviews. We then replace every movie review with a list of numbers.

```
data.train_ds[0][0].data[:10]
```

```
array([ 43,  44,  40,  34, 171,  62,    6, 352,   3,  47])
```

The list of numbers simply says what numbered thing in the vocab is in this place.

So through tokenization and numericalization, this is the standard way in NLP of turning a document into a list of numbers.

We can do that with the data block API:

```
data = (TextList.from_csv(path, 'texts.csv', cols='text')
        .split_from_df(col=2)
        .label_from_df(cols=0)
        .databunch())
```

This time, it's not ImageFilesList, it's TextList from a CSV and create a data bunch. At that point, we can start to create a model.

As we learn about it next week, when we do NLP classification, we actually create two models:

1. The first model is something called a **language model** which we train in a kind of a usual way.

```
learn = language_model_learner(data_lm, pretrained_model=URLs.WT103, drop_mlu
```

We say we want to create a language model learner, train it, save it, and we unfreeze, train some more.

2. After we've created a language model, we fine-tune it to create the **classifier**. We create the data bunch of the classifier, create a learner, train it and we end up with some accuracy.

That's the really quick version. We're going to go through it in more detail next week, but you can see the basic idea of training an NLP classifier is very similar to creating every other model we've seen so far. The current state of the art for IMDB classification is actually the algorithm that we built and published with colleague named Sebastian Ruder and what I just showed you is pretty much the state of the art algorithm with some minor tweaks. You can get this up to about 95% if you try really hard. So this is very close to the state of the art accuracy that we developed.

Question: For a dataset very different than ImageNet like the satellite images or genomic images shown in lesson 2, we should use our own stats.

Jeremy once said:

If you're using a pretrained model you need to use the same stats it was trained with.

Why is that? Isn't it that, normalized dataset with its own stats will have roughly the same distribution like ImageNet? The only thing I can think of, which may differ is skewness. Is it the possibility of skewness or something else the reason of your statement? And does that mean you don't recommend using pre-trained model with very different dataset like the one-point mutation that you showed us in lesson 2?

[1:46:53]

Nope. As you can see, I've used pre-trained models for all of those things. Every time I've used an ImageNet pre-trained model, I've used ImageNet stats. Why is that? Because that model was trained with those stats. For example, imagine you're trying to classify different types of green frogs. If you were to use your own per-channel means from your dataset, you would end up converting them to a mean of zero, a standard deviation of one for each of your red, green, and blue channels. Which means they don't look like green frogs anymore. They now look like grey frogs. But ImageNet expects frogs to be green. So you need to normalize with the same stats that the ImageNet training people normalized with. Otherwise the unique characteristics of your dataset won't appear anymore - you've actually normalized them out in terms of the per-channel statistics. So you should always use the same stats that the model was trained with.

In every case, what we're doing here is we're using gradient descent with mini batches (i.e. stochastic gradient descent) to fit some parameters of a model. And those parameters are parameters to matrix multiplications. The second half of this part, we're actually going to learn about a little tweak called convolutions, but it's basically a type of matrix multiplication.

The thing is though, no amount of matrix multiplications is possibly going to create something that can read IMDB movie reviews and decide if it's positive or negative or look at satellite imagery and decide whether it's got a road in it - that's far more than a linear classifier can do. Now we know these are deep neural networks. Deep neural networks contain lots of these matrix multiplications, but every matrix multiplication is just a linear model. A linear function on top of a linear function is just another linear function. If you remember back to your high school math, you might remember that if you have a $y = ax + b$ and then you stick another $cy + d$ on top of that, it's still just another slope and another intercept. So no amount of stacking matrix multiplications is going to help in the slightest.

So what are these models actually? What are we actually doing? And here's the interesting thing - all we're actually doing is we literally do have a matrix multiplication (or a slight variation like a convolution that we'll learn about) but after each one, we do something called a non-linearity or an **activation function**. An activation function is something that takes the result of that matrix multiplication and sticks it through some function. These are some of the functions that we use ([by Sagar Sharma](#)):

In the old days, the most common function that we used to use was **sigmoid**. And they have particular mathematical definitions. Nowadays, we almost never use those for these between each matrix multiply. Nowadays, we nearly always use this one - it's called a **rectified linear unit (ReLU)**. It's very important, when you're doing deep learning, to use big long words that sound impressive. Otherwise normal people might think they can do it too 😂. But just between you and me, a rectified linear unit is defined using the following function:

```
max(x, 0)
```

That's it. And if you want to be really exclusive, of course, you then shorten the long version and you call it a ReLU to show that you're really in the exclusive team. So this is a ReLU activation.

Here's the crazy thing. If you take your red green blue pixel inputs, and you chuck them through a matrix modification, and then you replace the negatives with zero, and you put it through another matrix modification, replace the negatives at zero, and you keep doing that again and again, you have a deep learning neural network. That's it.

⌚ Universal approximation theorem [1:52:27]

So how the heck does that work? An extremely cool guy called Michael Nielsen showed how this works. He has a very nice website (actually a book)

<http://neuralnetworksanddeeplearning.com> and he has these beautiful little JavaScript things where you can get to play around. Because this was back in the old days, this was back when we used to use sigmoids. What he shows is that if you have enough little matrix multiplications followed by sigmoids (exactly the same thing works for a matrix multiplication followed by a ReLU), you can actually create arbitrary shapes. So this idea that these combinations of linear functions and nonlinearities can create arbitrary shapes actually has a name and this name is the universal approximation theorem.

What it says is that if you have stacks of linear functions and nonlinearities, the thing you end up with can approximate any function arbitrarily closely. So you just need to make sure that you have a big enough matrix to multiply by, or enough of them. If you have this function which is just a sequence of matrix multiplies and nonlinearities where the nonlinearities can be basically any of these activation functions, if that can approximate anything, then all you need is some way to find the particular values of the weight matrices in your matrix multiplies that solve the problem you want to solve. We already know how to find the values of parameters. We can use gradient descent. So that's actually it.

And this is the bit I find the hardest thing normally to explain to students is that we're actually done now. People often come up to me after this lesson and they say "what's the rest? Please explain to me the rest of deep learning." But no, there's no rest. We have a function where we take our input pixels or whatever, we multiply them by some weight matrix, we replace the negatives with zeros, we multiply it by another weight matrix, replace the negative zeros, we do that a few times. We see how close it is to our target and then we use gradient descent to update our weight matrices using the derivatives, and we do that a few times. And eventually, we end up with something that can classify movie reviews or can recognize pictures of ragdoll cats. That's actually it.

The reason it's hard to understand intuitively is because we're talking about weight matrices that have (once you add them all up) something like a hundred million parameters. They're very big weight matrices. So your intuition about what multiplying something by a linear model and replacing the negative zeros a bunch of times can do, your intuition doesn't hold. You just have to accept empirically the truth is doing that works really well.

In part two of the course, we're actually going to build these from scratch. But just to skip ahead, you basically will find that it's going to be five lines of code. It's going to be a little for loop that goes `t = x @ w1` , `t2 = max(t, 0)` , stick that in a for loop that goes through each weight matrix, and at the end calculate the loss function. Of course, we're not going to calculate the gradients ourselves because PyTorch does that for us. And that's about it.

Question: There's a question about tokenization. I'm curious about how tokenizing words works when they depend on each other such as San Francisco. [1:56:45]

How do you tokenize something like San Francisco. San Francisco contains two tokens `San` `Francisco` . That's it. That's how you tokenize San Francisco. The question may be coming from people who have done traditional NLP which often need to use these things called n-grams. N-grams are this idea of a lot of NLP in the old days was all built on top of linear models where you basically counted how many times particular strings of text appeared like the phrase San Francisco. That would be a bi-gram for an n-gram with an n of 2. The cool thing is that with deep learning, we don't have to worry about that. Like with many things, a lot of the complex feature engineering disappears when you do deep learning. So with deep learning, each token is literally just a word (or in the case that the word really consists of two words like `you're` you split it into two words) and then what we're going to do is we're going to then let the deep learning model figure out how best to combine words together. Now when we see like let the deep learning model figure it out, of course all we really mean is find the weight matrices using gradient descent that gives the right answer. There's not really much more to it than that.

Again, there's some minor tweaks. In the second half of the course, we're going to be learning about the particular tweak for image models which is using a convolution that'll be a CNN, for language there's a particular tweak we do called using recurrent models or an RNN, but they're very minor tweaks on what we've just described. So basically it turns out with an RNN, that it can learn that San plus Francisco has a different meaning when those two things are together.

Question: Some satellite images have 4 channels. How can we deal with data that has 4 channels or 2 channels when using pre-trained models? [1:59:09]

I think that's something that we're going to try and incorporate into fast AI. So hopefully, by the time you watch this video, there'll be easier ways to do this. But the basic idea is a pre-trained ImageNet model expects a red green and blue pixels. So if you've only got two channels, there's a few things you can do but basically you'll want to create a third channel. You can create the third channel as either being all zeros, or it could be the average of the other two channels. So you can just use you know normal PyTorch arithmetic to create that third channel. You could either do that ahead of time in a little loop and save your three channel versions, or you could create a custom dataset class that does that on demand.

For 4 channel, you probably don't want to get rid of the 4th channel. So instead, what you'd have to do is to actually modify the model itself. So to know how to do that, we'll only know how to do in a couple more lessons time. But basically the idea is that the initial weight matrix (weight matrix is really the wrong term, they're not weight matrices; their weight tensors so they can have more than just two dimensions), so that initial weight tensor in the neural net, one of its axes is going to have three slices in it. So you would just have to change that to add an extra slice, which I would generally just initialize to zero or to some random numbers. So that's the short version. But really to understand exactly what I meant by that, we're going to need a couple more lessons to get there.

Wrapping up [2:01:19]

What have we looked at today? We started out by saying it's really easy now to create web apps. We've got starter kits for you that show you how to create web apps, and people have created some really cool web apps using what we've learned so far which is single label classification.

But the cool thing is the exact same steps we use to do single label classification, you can also do to:

- Multi-label classification such as in the planet dataset.
- Image segmentation.
- Any kind of image regression.
- NLP classification.

- and a lot more.

In each case, all we're actually doing is:

- Gradient descent
- Non-linearity

Universal approximation theorem tells us it lets us arbitrarily accurately approximate any given function including functions such as:

- Converting a spoken waveform into the thing the person was saying.
- Converting a sentence in Japanese to a sentence in English.
- Converting a picture of a dog into the word dog.

These are all mathematical functions that we can learn using this approach.

So this week, see if you can come up with an interesting idea of a problem that you would like to solve which is either multi-label classification, image regression, image segmentation, or something like that and see if you can try to solve that problem. You will probably find the hardest part of solving that problem is creating the data bunch and so then you'll need to dig into the data block API to try to figure out how to create the data bunch from the data you have. With some practice, you will start to get pretty good at that. It's not a huge API. There's a small number of pieces. It's also very easy to add your own, but for now, ask on the forum if you try something and you get stuck.

Next week, we're going to come back and we're going to look at some more NLP. We're going to learn some more about some details about how we actually train with SGD quickly. We're going to learn about things like Adam and RMSProp and so forth. And hopefully, we're also going to show off lots of really cool web apps and models that you've all built during the week, so I'll see you then. Thanks!



shiv-u modify the dep_var (#33)

a21c432 5 days ago

3 contributors

1064 lines (676 sloc) 98.4 KB

Raw

Blame

History



⌚ Lesson 4

[Video / Lesson Forum](#)

Welcome to Lesson 4! We are going to finish our journey through these key applications. We've already looked at a range of vision applications. We've looked at classification, localization, image regression. We briefly touched on NLP. We're going to do a deeper dive into NLP transfer learning today. We're going to then look at tabular data and collaborative filtering which are both super useful applications.

Then we're going to take a complete u-turn. We're going to take that collaborative filtering example and dive deeply into it to understand exactly what's happening mathematically - exactly what's happening in the computer. And we're going to use that to gradually go back in reverse order through the applications again in order to understand exactly what's going on behind the scenes of all of those applications.

⌚ Correction on CamVid result

Before we do, somebody on the forum is kind enough to point out that when we compared ourselves to what we think might be the state of the art or was recently the state of the art for CamVid, there wasn't a fair comparison because the paper actually used a small subset of the classes, and we used all of the classes. So Jason in our study group was kind enough to rerun the experiments with the correct subset of classes from the paper, and our accuracy went up to 94% compared to 91.5% of the paper. So I think that's a really cool result. and a great example of how pretty much just using the defaults nowadays can get you far beyond what was the best of a year or two ago. It was certainly the best last year when we were doing this course because we started it quite intensely. So that's really exciting.

⌚ Natural Language Processing (NLP) [2:00]

What I wanted to start with is going back over NLP a little bit to understand really what was going on there.

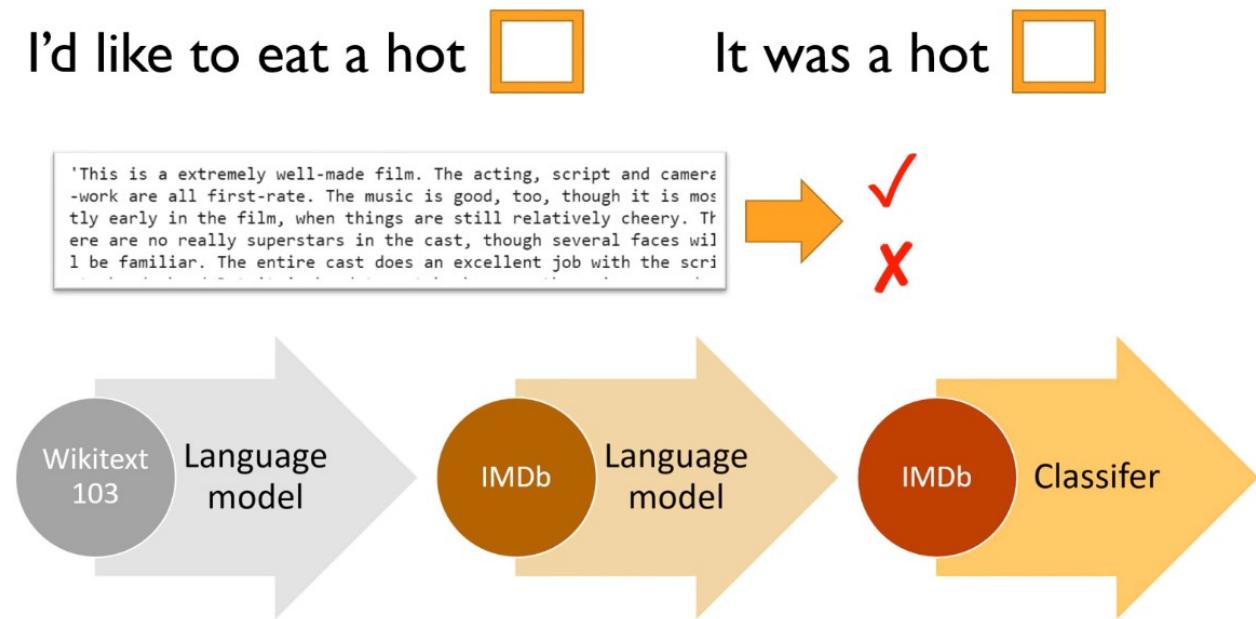
➲ A quick review

So first of all, a quick review. Remember NLP is natural language processing. It's about taking text and doing something with it. Text classification is particularly useful - practically useful applications. It's what we're going to start off focusing on. Because classifying a text or classifying a document can be used for anything from:

- Spam prevention
- Identifying fake news
- Finding a diagnosis from medical reports
- Finding mentions of your product in Twitter

So it's pretty interesting. And actually there was a great example during the week from one of our students [@howkhang](#) who is a lawyer and he mentioned on [the forum](#) that he had a really great results from classifying legal texts using this NLP approach. And I thought this was a great example. This is the post that they presented at an academic conference this week describing the approach:

This series of three steps that you see here (and I'm sure you recognize this classification matrix) is what we're going to start by digging into.



We're going to start out with a movie review like this one and decide whether it's positive or negative sentiment about the movie. That is the problem. We have, in the training set, 25,000 movie reviews and for each one we have like one bit of information: they liked it, or they didn't like it. That's what we're going to look into a lot more detail today and in the current lessons. Our neural networks (remember, they're just a bunch of matrix multiplies and simple nonlinearities - particularly replacing negatives with zeros), those weight matrices start out random. So if you start out with some random parameters and try to train those parameters to learn how to recognize positive vs. negative movie reviews, you literally have 25,000 ones and zeros to actually tell you I like this one I don't like that one. That's clearly not enough information to learn, basically, how to speak English - how to speak English well enough to recognize they liked this or they didn't like this. Sometimes that can be pretty nuanced. Particularly with movie reviews because these are like online movie reviews on IMDB, people can often use sarcasm. It could be really quite tricky.

Until very recently, in fact, this year, neural nets didn't do a good job at all of this kind of classification problem. And that was why - there's not enough information available. So the trick, hopefully you can all guess, is to use transfer learning. It's always the trick.

Last year in this course I tried something crazy which was I thought what if I try transform learning to demonstrate that it can work for NLP as well. I tried it out and it worked extraordinarily well. So here we are, a year later, and transfer learning in NLP is absolutely the hit thing. And I'm going to describe to you what happens.

⌚ Transfer learning in NLP [6:04]

The key thing is we're going to start with the same kind of thing that we used for computer vision - a pre-trained model that's been trained to do something different to what we're doing with it. For ImageNet, that was originally built as a model to predict which of a thousand categories each photo falls into. And people then fine-tune that for all kinds of different things as you've seen. So we're going to start with a pre-trained model that's going to do something else. Not movie review classification. We're going to start with a pre-trained model which is called a language model.

A language model has a very specific meaning in NLP and it's this. A language model is a model that learns to predict the next word of a sentence. To predict the next word of a sentence, you actually have to know quite a lot about English (assuming you're doing it in English) and quite a lot of world knowledge. By world knowledge, I'll give you an example.

Here's your language model and it has read:

- "I'd like to eat a hot __": Obviously, "dog", right?
- "It was a hot __": Probably "day"

Now previous approaches to NLP use something called n-grams largely which is basically saying how often do these pairs or triplets of words tend to appear next to each other. And n-grams are terrible at this kind of thing. As you can see, there's not enough information here to decide what the next word probably is. But with a neural net, you absolutely can.

So here's the nice thing. If you train a neural net to predict the next word of a sentence then you actually have a lot of information. Rather than having a single bit to every 2,000 word movie review: "liked it" or "didn't like it", every single word, you can try and predict the next word. So in a 2,000 word movie review, there are 1,999 opportunities to predict the next word. Better still, you don't just have to look at movie reviews. Because really the hard thing isn't so much as "does this person like the movie or not?" but "how do you speak English?". So you can learn "how do you speak English?" (roughly) from some much bigger set of documents. So what we did was we started with Wikipedia.

⌚ Wikitext 103 [8:30]

Stephen Merity and some of his colleagues built something called Wikitext 103 dataset which is simply a subset of most of the largest articles from Wikipedia with a little bit of pre-processing that's available for download. So you're basically grabbing Wikipedia and then I built a language model on all of Wikipedia. So I just built a neural net which would predict the next word in every significantly sized Wikipedia article. That's a lot of information. If I remember correctly, it's something like a billion tokens. So we've got a billion separate things to predict. Every time we make a mistake on one of those predictions, we get the loss, we get gradients from that, and we can update our weights, and they can better and better until we can get pretty good at predicting the next word of Wikipedia.

Why is that useful? Because at that point, I've got a model that knows probably how to complete sentences like this, so it knows quite a lot about English and quite a lot about how the world works - what kinds of things tend to be hot in different situations, for instance. Ideally, it would learn things like "in 1996 in a speech to the United Nations, United States president ____ said "... Now that would be a really good language model, because it would actually have to know who is this United States president in that year. So getting really good at training language models is a great way to teach a neural-net a lot about what is our world, what's in our world, how do things work in our world. It's a really fascinating topic, and it's actually one that philosophers have been studying for hundreds of years now. There's actually a whole theory of philosophy which is about what can be learned from studying language alone. So it turns out, apparently, quite a lot.

So here's the interesting thing. You can start by training a language model on all of Wikipedia, and then we can make that available to all of you. Just like a pre-trained ImageNet model for vision, we've now made available a pre-trained Wikitext model for NLP not because it's particularly useful of itself (predicting the next word of sentences is somewhat useful, but not normally what we want to do), but it's a model that understands a lot about language and a lot about what language describes. So then, we can take that and we can do transfer learning to create a new language model that's specifically good at predicting the next word of movie reviews.

⌚ Fine-tuning Wikitext to create a new language model [11:10]

If we can build a language model that's good at predicting the next word of movie reviews pre-trained with the Wikitext model, then that's going to understand a lot about "my favorite actor is Tom ____." Or "I thought the photography was fantastic but I wasn't really so happy about the ____ (director)." It's going to learn a lot about specifically how movie reviews are written. It'll even learn things like what are the names of some popular movies.

That would then mean we can still use a huge corpus of lots of movie reviews even if we don't know whether they're positive or negative to learn a lot about how movie reviews are written. So for all of this pre-training and all of this language model fine-tuning, we don't need any labels at all. It is what the researcher Yann LeCun calls **self supervised learning**. In other words, it's a classic supervised model - we have labels, but the labels are not things that somebody else have created. They're built into the dataset itself. So this is really really neat. Because at this point, we've now got something that's good at understanding movie reviews and we can fine-tune that with transfer learning to do the thing we want to do which in this case is to classify movie reviews to be positive or negative. So my hope was (when I tried this last year) that at that point, 25,000 ones and zeros would be enough feedback to fine-tune that model and it turned out it absolutely was.

Question: Does the language model approach work for text in forums that are informal English, misspelled words or slangs or shortforms like s6 instead of Samsung S 6? [12:47]

Yes, absolutely it does. Particularly if you start with your wikitext model and then fine-tune it with your "target" corpus. Corpus is just a bunch of documents (emails, tweets, medical reports, or whatever). You could fine-tune it so it can learn a bit about the specifics of the slang, abbreviations, or whatever that didn't appear in the full corpus. So interestingly, this is one of the big things that people were surprised about when we did this research last year. People thought that learning from something like Wikipedia wouldn't be that helpful because it's not that representative of how people tend to write. But it turns out it's extremely helpful because there's a much a difference between Wikipedia and random words than there is between like Wikipedia and reddit. So it kind of gets you 99% of the way there.

So language models themselves can be quite powerful. For example there was a [blog post](#) from SwiftKey (the folks that do the mobile-phone predictive text keyboard) and they describe how they kind of rewrote their underlying model to use neural nets. This was a year or two ago. Now most phone keyboards seem to do this. You'll be typing away on your mobile phone, and in the prediction there will be something telling you what word you might want next. So that's a language model in your phone.

Another example was the researcher Andrej Karpathy who now runs all this stuff at Tesla, back when he was a PhD student, he created [a language model of text in LaTeX documents](#) and created these automatic generation of LaTeX documents that then became these automatically generated papers. That's pretty cute.

We're not really that interested in the output of the language model ourselves. We're just interested in it because it's helpful with this process.

⌚ Review of the basic process [15:14]

We briefly looked at the process last week. The basic process is, we're going to start with the data in some format. So for example, we've prepared a little IMDB sample that you can use which is in CSV file. You can read it in with Pandas and there's negative or positive, the text of each movie review, and boolean of is it in the validation set or the training set.

```
path = untar_data(URLs.IMDB_SAMPLE)
path.ls()
```

```
[PosixPath('/home/jhoward/.fastai/data/imdb_sample/texts.csv'),
 PosixPath('/home/jhoward/.fastai/data/imdb_sample/models')]
```

```
df = pd.read_csv(path/'texts.csv')
df.head()
```

	label	text	is_valid
0	negative	Un-bleeping-believable! Meg Ryan doesn't even ...	False
1	positive	This is a extremely well-made film. The acting...	False
2	negative	Every once in a long while a movie will come a...	False
3	positive	Name just says it all. I watched this movie wi...	False
4	negative	This movie succeeds at being one of the most u...	False

So there's an example of a movie review:

```
df['text'][1]
```

'This is a extremely well-made film. The acting, script and camera-work are all first-rate. The music is good, too, though it is mostly early in the film, when things are still relatively cheery. There are no really superstars in the cast, though several faces will be familiar. The entire cast does an excellent job with the script.

But it is hard to watch, because there is no good end to a situation like the one presented. It is now fashionable to blame the British for setting Hindus and Muslims against each other, and then cruelly separating them into two countries. There is some merit in this view, but it's also true that no one forced Hindus and Muslims in the region to mistreat each other as they did around the time of partition. It seems more likely that the British simply saw the tensions between the religions and were clever enough to exploit them to their own ends.

The result is that there is much cruelty and inhumanity in the situation and this is very unpleasant to remember and to see on the screen. But it is never painted as a black-and-white case. There is baseness and nobility on both sides, and also the hope for change in the younger generation.

There is redemption of a sort, in the end, when Puro has to make a hard choice between a man who has ruined her life, but also truly loved her, and her family which has disowned her, then later come looking for her. But by that point, she has no option that is without great pain for her.

This film carries the message that both Muslims and Hindus have their grave faults, and also that both can be dignified and caring people. The reality of partition makes that realisation all the more wrenching, since there can never be real reconciliation across the India/Pakistan border. In that sense, it is similar to "Mr & Mrs Iyer".

In the end, we were glad to have seen the film, even though the resolution was heartbreakng. If the UK and US could deal with their own histories of racism with this kind of frankness, they would certainly be better off.'

So you can just go `TextDataBunch.from_csv` to grab a language model specific data bunch:

```
data_lm = TextDataBunch.from_csv(path, 'texts.csv')
```

And then you can create a learner from that in the usual way and fit it.

```
data_lm.save()
```

You can save the data bunch which means that the pre-processing that is done, you don't have to do it again. You can just load it.

```
data = TextDataBunch.load(path)
```

What happens behind the scenes if we now load it as a classification data bunch (that's going to allow us to see the labels as well)?

```
data = TextClasDataBunch.load(path)
data.show_batch()
```

text	label
xxbos xxfld 1 raising victor vargas : a review \n\n you know , raising victor vargas is like sticking your hands into a big , xxunk bowl of xxunk . it 's warm and gooey , but you 're not sure if it feels right . try as i might	negative
xxbos xxfld 1 now that che(2008) has finished its relatively short australian cinema run (extremely limited xxunk screen in xxunk , after xxunk) , i can xxunk join both xxunk of " at the movies " in taking steven soderbergh to task . \n\n it 's usually	negative
xxbos xxfld 1 many xxunk that this is n't just a classic due to the fact that it 's the first xxup 3d game , or even the first xxunk - up . it 's also one of the first xxunk games , one of the xxunk definitely the first	positive
xxbos xxfld 1 i really wanted to love this show . i truly , honestly did . \n\n for the first time , gay viewers get their own version of the " the bachelor " . with the help of his obligatory " hag " xxunk , james , a	negative
xxbos xxfld 1 this film sat on my xxunk for weeks before i watched it . i xxunk a self - indulgent xxunk flick about relationships gone bad . i was wrong ; this was an xxunk xxunk into the xxunk - up xxunk of new xxunk . \n\n the	positive

As we described, it basically creates a separate unit (i.e. a "token") for each separate part of a word. So most of them are just for words, but sometimes if it's like an 's from it's , it will get its own token. Every bit of punctuation tends to get its own token (a comma, a full stop, and so forth).

Then the next thing that we do is a numericalization which is where we find what are all of the unique tokens that appear here, and we create a big list of them. Here's the first ten in order of frequency:

```
data.vocab.itos[:10]
```

```
['xxunk', 'xxpad', 'the', ',', '.', 'and', 'a', 'of', 'to', 'is']
```

And that big list of unique possible tokens is called the vocabulary which we just call it a "vocab". So what we then do is we replace the tokens with the ID of where is that token in the vocab:

```
data.train_ds[0][0]
```

Text xxbos xxfld 1 he now has a name , an identity , some memories and a a lost girlfriend . all he wanted was to disappear , but still , they xxunk him and destroyed the world he hardly built . now he wants some explanation , and to get ride of the people how made him what he is . yeah , jason bourne is back , and this time , he 's here with a vengeance .

```
data.train_ds[0][0].data[:10]
```

```
array([ 43,  44,  40,  34, 171,  62,    6, 352,    3,  47])
```

That's numericalization. Here's the thing though. As you'll learn, every word in our vocab is going to require a separate row in a weight matrix in our neural net. So to avoid that weight matrix getting too huge, we restrict the vocab to no more than (by default) 60,000 words. And if a word doesn't appear more than two times, we don't put it in the vocab either. So we keep the vocab to a reasonable size in that way. When you see these xxunk , that's an unknown token. It just means this was something that was not a common enough word to appear in our vocab.

We also have a couple of other special tokens like (see `fastai.text.transform.py` for up-to-date info):

- `xxfld` : This is a special thing where if you've got like title, summary, abstract, body, (i. e. separate parts of a document), each one will get a separate field and so they will get numbered (e.g. `xxfld 2`).
- `xxup` : If there's something in all caps, it gets lower cased and a token called `xxup` will get added to it.

⌚ With the data block API [18:31]

Personally, I more often use the data block API because there's less to remember about exactly what data bunch to use, and what parameters and so forth, and it can be a bit more flexible.

```
data = (TextList.from_csv(path, 'texts.csv', cols='text')
        .split_from_df(col=2)
        .label_from_df(cols=0)
```

```
.databunch()
```

So another approach to doing this is to just decide:

- What kind of list you're creating (i.e. what's your independent variable)? So in this case, my independent variable is text.
- What is it coming from? A CSV.
- How do you want to split it into validation versus training? So in this case, column number two was the `is_valid` flag.
- How do you want to label it? With positive or negative sentiment, for example. So column zero had that.
- Then turn that into a data bunch.

That's going to do the same thing.

```
path = untar_data(URLs.IMDB)
path.ls()
```

```
[PosixPath('/home/jhoward/.fastai/data/imdb/imdb.vocab'),
 PosixPath('/home/jhoward/.fastai/data/imdb/models'),
 PosixPath('/home/jhoward/.fastai/data/imdb/tmp_lm'),
 PosixPath('/home/jhoward/.fastai/data/imdb/train'),
 PosixPath('/home/jhoward/.fastai/data/imdb/test'),
 PosixPath('/home/jhoward/.fastai/data/imdb/README'),
 PosixPath('/home/jhoward/.fastai/data/imdb/tmp_clas')]
```

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

```
[PosixPath('/home/jhoward/.fastai/data/imdb/train/pos'),
 PosixPath('/home/jhoward/.fastai/data/imdb/train/unsup'),
 PosixPath('/home/jhoward/.fastai/data/imdb/train/unsupBow.feat'),
 PosixPath('/home/jhoward/.fastai/data/imdb/train/labeledBow.feat'),
 PosixPath('/home/jhoward/.fastai/data/imdb/train/neg')]
```

Now let's grab the whole data set which has:

- 25,000 reviews in training
- 25,000 interviews in validation
- 50,000 unsupervised movie reviews (50,000 movie reviews that haven't been scored at all)

We're going to start with the language model. Now the good news is, we don't have to train the Wikitext 103 language model. Not that it's difficult - you can just download the wikitext 103 corpus, and run the same code. But it takes two or three days on a decent GPU, so not much point in you doing it. You may as well start with ours. Even if you've got a big corpus of like medical documents or legal documents, you should still start with Wikitext 103. There's just no reason to start with random weights. It's always good to use transfer learning if you can.

So we're gonna start fine-tuning our IMDB language model.

bs=48

```
data_lm = (TextList.from_folder(path)
            #Inputs: all the text files in path
            .filter_by_folder(include=['train', 'test'])
            #We may have other temp folders that contain text files so we only keep
            .random_split_by_pct(0.1)
            #We randomly split and keep 10% (10,000 reviews) for validation
            .label_for_lm()
            #We want to do a language model so we label accordingly
            .databunch(bs=bs))
data_lm.save('tmp_lm')
```

We can say:

- It's a list of text files - the full IMDB actually is not in a CSV. Each document is a separate text file.
- Say where it is - in this case we have to make sure we just to include the `train` and `test` folders.
- We randomly split it by 0.1.

Now this is interesting - 10%. Why are we randomly splitting it by 10% rather than using the predefined train and test they gave us? This is one of the cool things about transfer learning. Even though our validation set has to be held aside, it's actually only the labels that we have to keep aside. So we're not allowed to use the labels in the test set. If you think about in a Kaggle competition, you certainly can't use the labels because they don't even give them to you. But you can certainly use the independent variables. So in this case, you could absolutely use the text that is in the test set to train your language model. This is a good trick - when you do the language model, concatenate the training and test set together, and then just split out a smaller validation set so you've got more data to train your language model. So that's a little trick.

So if you're doing NLP stuff on Kaggle, for example, or you've just got a smaller subset of labeled data, make sure that you use all of the text you have to train in your language model, because there's no reason not to.

- How are we going to label it? Remember, a language model kind of has its own labels. The text itself is labels so label for a language model (`label_for_lm`) does that for us.
- And create a data bunch and save it. That takes a few minutes to tokenize and numericalize.

Since it takes some few minutes, we save it. Later on you can just load it. No need to run it again.

```
data_lm = TextLMDataBunch.load(path, 'tmp_lm', bs=bs)
```

```
data_lm.show_batch()
```

idx	text
0	xxbos after seeing the truman show , i wanted to see the other films by weir . i would say this is a good one to start with . the plot : \n\n the wife of a doctor (who is trying to impress his bosses , so he can get xxunk trying to finish her written course , while he s at work . but one day a strange man , who says that he s a plumber , tells her he s been called out to repair some pipes in there flat .
1	and turn to the wisdom of homeless people & ghosts . that 's a good plan . i would never recommend this movie ; partly because the sexual content is unnecessarily graphic , but also because it really does n't offer any valuable insight . check out " yentl " if you want to see a much more useful treatment of jewish tradition at odds with society . xxbos creep is the story of kate (potente) , an intensely unlikeable bourgeois bitch that finds herself somehow sleeping through the noise of the last
2	been done before but there is something about the way its done here that lifts it up from the rest of the pack . \n\n 8 out of 10 for dinosaur / monster lovers . xxbos i rented this movie to see how the sony xxunk camera shoots , (i recently purchased the same camera) and was blown away by the story and the acting . the directing , acting , editing was all above what i expected from what appeared at first glance to be a " low budget " type of
3	troubles . nigel and xxunk are the perfect team , i 'd watch their show any day ! i was so crushed when they removed it , and anytime they had it on xxup tv after that i was over the moon ! they put it on on demand one summer (only the first eight episodes or so) and i 'd spend whole afternoons watching them one after the other ... but the worst part ? it is now back on a channel called xxunk - and xxup it xxup 's xxup on

idx	text
4	movie !) the movie is about edward , a obsessive - compulsive , nice guy , who happens to be a film editor . he is then lent to another department in the building , and he is sent to the posh yet violent world of sam campbell , the splatter and gore department . sam campbell , eddy 's new boss , is telling eddy about the big break on his movies , the gruesome loose limbs series , and he needs eddy to make the movie somewhat less violent so they can

⌚ Training [22:29]

At this point things are going to look very familiar. We create a learner:

```
learn = language_model_learner(data_lm, pretrained_model=URLs.WT103, drop_mult=0.
```

But instead of creating a CNN learner, we're going to create a language model learner. So behind the scenes, this is actually not going to create a CNN (a convolutional neural network), it's going to create an RNN (a recurrent neural network). We're going to be learning exactly how they're built over the coming lessons, but in short they're the same basic structure. The input goes into a weight matrix (i.e. a matrix multiply), that then you replace the negatives with zeros, and it goes into another matrix multiply, and so forth a bunch of times. So it's the same basic structure.

As usual, when we create a learner, you have to pass in two things:

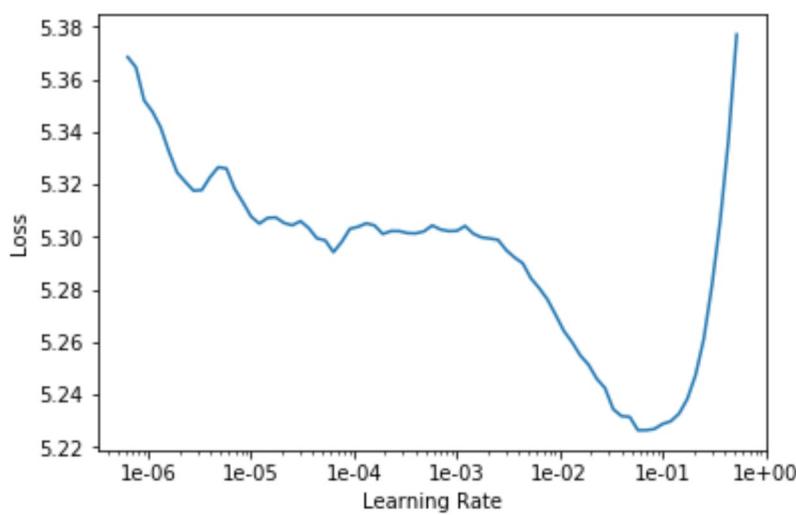
- The data: so here's our language model data
- What pre-trained model we want to use: here, the pre-trained model is the Wikitext 103 model that will be downloaded for you from fastai if you haven't used it before just like ImageNet pre-trained models are downloaded for you.

This here (`drop_mult=0.3`) sets the amount of dropout. We haven't talked about that yet. We've talked briefly about this idea that there is something called regularization and you can reduce the regularization to avoid underfitting. So for now, just know that by using a number lower than one is because when I first tried to run this, I was under fitting. So if you reduced that number, then it will avoid under fitting.

Okay. so we've got a learner, we can `lr_find` and looks pretty standard:

```
learn.lr_find()
```

```
learn.recorder.plot(skip_end=15)
```



Then we can fit one cycle.

```
learn.fit_one_cycle(1, 1e-2, mom=(0.8,0.7))
```

```
Total time: 12:42
epoch  train_loss  valid_loss  accuracy
1      4.591534    4.429290    0.251909  (12:42)
```

What's happening here is we are just fine-tuning the last layers. Normally after we fine-tune the last layers, the next thing we do is we go `unfreeze` and train the whole thing. So here it is:

```
learn.unfreeze()
learn.fit_one_cycle(10, 1e-3, mom=(0.8,0.7))
```

```
Total time: 2:22:17
epoch  train_loss  valid_loss  accuracy
1      4.307920    4.245430    0.271067  (14:14)
2      4.253745    4.162714    0.281017  (14:13)
3      4.166390    4.114120    0.287092  (14:14)
4      4.099329    4.068735    0.292060  (14:10)
5      4.048801    4.035339    0.295645  (14:12)
6      3.980410    4.009860    0.298551  (14:12)
7      3.947437    3.991286    0.300850  (14:14)
8      3.897383    3.977569    0.302463  (14:15)
9      3.866736    3.972447    0.303147  (14:14)
10     3.847952    3.972852    0.303105  (14:15)
```

As you can see, even on a pretty beefy GPU that takes two or three hours. In fact, I'm still under fitting. So probably tonight, I might train it overnight and try and do a little bit better. I'm guessing I could probably train this a bit longer because you can see the accuracy hasn't started going down again. So I wouldn't mind try to train that a bit longer. But the accuracy, it's interesting. 0.3 means we're guessing the next word of the movie review correctly about a third of the time. That sounds like a pretty high number - the idea that you can actually guess the next word that often. So it's a good sign that my language model is doing pretty well. For more limited domain documents (like medical transcripts and legal transcripts), you'll often find this accuracy gets a lot higher. So sometimes this can be even 50% or more. But 0.3 or more is pretty good.

⌚ Predicting with Language Model [25:43]

You can now run `learn.predict` and pass in the start of a sentence, and it will try and finish off that sentence for you.

```
learn.predict('I liked this movie because ', 100, temperature=1.1, min_p=0.001)
```

Total time: 00:10

```
'I liked this movie because of course after yeah funny later that the world  
reason settings - the movie that perfect the kill of the same plot - a mention  
of the most of course . do xxup diamonds and the " xxup disappeared kill of  
course and the movie niece , from the care more the story of the let character  
, " i was a lot \'s the little performance is not only . the excellent for the  
most of course , with the minutes night on the into movies ( ! , in the movie  
its the first ever ! \n\n a'
```

Now I should mention, this is not designed to be a good text generation system. This is really more designed to check that it seems to be creating something that's vaguely sensible. There's a lot lot of tricks that you can use to generate much higher quality text - none of which we're using here. But you can kind of see that it's certainly not random words that it's generating. It sounds vaguely English like even though it doesn't make any sense.

At this point, we have a movie review model. So now we're going to save that in order to load it into our classifier (i.e. to be a pre-trained model for the classifier). But I actually don't want to save the whole thing. A lot of the second half of the language model is all about predicting the next word rather than about understanding the sentence so far. So the bit which is specifically about understanding the sentence so far is called the **encoder**, so I just save that (i.e. the bit that understands the sentence rather than the bit that generates the word).

```
learn.save_encoder('fine_tuned_enc')
```

⌚ Classifier [27:18]

Now we're ready to create our classifier. Step one, as per usual, is to create a data bunch, and we're going to do basically exactly the same thing:

```
data_clas = (TextList.from_folder(path, vocab=data_lm.vocab)
              #grab all the text files in path
              .split_by_folder(valid='test')
              #split by train and valid folder (that only keeps 'train' and 'test'
              .label_from_folder(classes=['neg', 'pos'])
              #remove docs with labels not in above list (i.e. 'unsup')
              .filter_missing_y()
              #label them all with their folders
              .databunch(bs=50))
data_clas.save('tmp_clas')
```

But we want to make sure that it uses exactly the same vocab that are used for the language model. If word number 10 was the in the language model, we need to make sure that word number 10 is the in the classifier. Because otherwise, the pre-trained model is going to be totally meaningless. So that's why we pass in the vocab from the language model to make sure that this data bunch is going to have exactly the same vocab. That's an important step.

`split_by_folder` - remember, the last time we had split randomly, but this time we need to make sure that the labels of the test set are not touched. So we split by folder.

And then this time we label it not for a language model but we label these classes (`['neg', 'pos']`). Then finally create a data bunch.

Sometimes you'll find that you ran out of GPU memory. I was running this in an 11G machine, so you should make sure this number (`bs`) is a bit lower if you run out of memory. You may also want to make sure you restart the notebook and kind of start it just from here (classifier section). Batch size 50 is as high as I could get on an 11G card. If you're using a p2 or p3 on Amazon or the K80 on Google, for example, I think you'll get 16G so you might be able to make this bit higher, get it up to 64. So you can find whatever batch size fits on your card.

So here is our data bunch:

```
data_clas = TextClasDataBunch.load(path, 'tmp_clas', bs=bs)
data_clas.show_batch()
```

text	label
xxfld 1 match 1 : tag team table match bubba ray and spike dudley vs eddie guerrero and chris benoit bubba ray and spike dudley started things off with a tag team table match against eddie guerrero and chris benoit . according to the rules of the match , both	pos
xxfld 1 i have never seen any of spike lee 's prior films , as their trailers never caught my interest . i have seen , and admire denzel washington , and jodie foster 's work , and have several of their dvds . i was , however , entirely	neg
xxfld 1 pier paolo pasolini , or pee - pee - pee as i prefer to call him (due to his love of showing male genitals) , is perhaps xxup the most overrated european marxist director - and they are thick on the ground . how anyone can	neg
xxfld 1 chris rock deserves better than he gives himself in " down to earth . " as directed by brothers chris & paul weitz of " american pie " fame , this uninspired remake of warren beatty 's 1978 fantasy " heaven can wait , " itself a rehash	neg
xxfld 1 yesterday , i went to the monthly antique flea market that comes to town . i really have no interest in such things , but i went for the fellowship of friends who do have such an interest . looking over the hundreds of vendor , passing many	pos

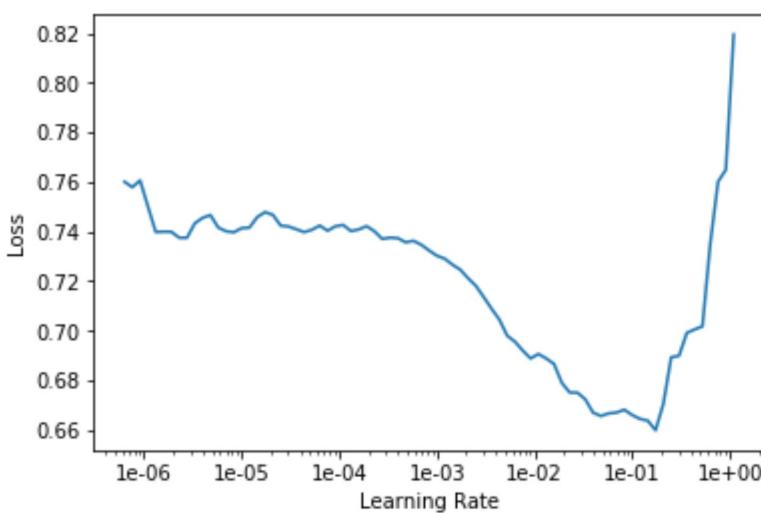
```
learn = text_classifier_learner(data_clas, drop_mult=0.5)
learn.load_encoder('fine_tuned_enc')
learn.freeze()
```

This time, rather than creating a language model learner, we're creating a text classifier learner. But again, same thing - pass in the data that we want, figure out how much regularization we need. If you're overfitting then you can increase this number (`drop_mult`). If you're underfitting, you can decrease the number. And most importantly, load in our pre train model. Remember, specifically it's this half of the model called the encoder which is the bit that we want to load in.

Then freeze, lr_find, find the learning rate and fit for a little bit.

```
learn.lr_find()
```

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



```
learn.fit_one_cycle(1, 2e-2, mom=(0.8,0.7))
```

```
Total time: 02:46
epoch  train_loss  valid_loss  accuracy
1      0.294225    0.210385    0.918960  (02:46)
```

We're already up nearly to 92% accuracy after less than three minutes of training. So this is a nice thing. In your particular domain (whether it be law, medicine, journalism, government, or whatever), you probably only need to train your domain's language model once. And that might take overnight to train well. But once you've got it, you can now very quickly create all kinds of different classifiers and models with that. In this case, already a pretty good model after three minutes. So when you first start doing this, you might find it a bit annoying that your first models take four hours or more to create that language model. But the key thing to remember is you only have to do that once for your entire domain of stuff that you're interested in. And then you can build lots of different classifiers and other models on top of that in a few minutes.

```
learn.save('first')
```

```
learn.load('first');
```

```
learn.freeze_to(-2)
learn.fit_one_cycle(1, slice(1e-2/(2.6**4),1e-2), mom=(0.8,0.7))
```

```
Total time: 03:03
epoch  train_loss  valid_loss  accuracy
1      0.268781    0.180993    0.930760  (03:03)
```

We can save that to make sure we don't have to run it again.

And then, here's something interesting. I'm not going to say `unfreeze`. Instead, I'm going to say `freeze_to`. What that says is unfreeze the last two layers, don't unfreeze the whole thing. We've just found it really helps with these text classification not to unfreeze the whole thing, but to unfreeze one layer at a time.

- unfreeze the last two layers
- train it a little bit more
- unfreeze the next layer again
- train it a little bit more
- unfreeze the whole thing
- train it a little bit more

```
learn.save('second')
```

```
learn.load('second');
```

```
learn.freeze_to(-3)
learn.fit_one_cycle(1, slice(5e-3/(2.6**4),5e-3), mom=(0.8,0.7))
```

```
Total time: 04:06
epoch  train_loss  valid_loss  accuracy
1      0.211133    0.161494    0.941280  (04:06)
```

```
learn.save('third')
```

```
learn.load('third');
```

```
learn.unfreeze()
learn.fit_one_cycle(2, slice(1e-3/(2.6**4),1e-3), mom=(0.8,0.7))
```

```
Total time: 10:01
epoch  train_loss  valid_loss  accuracy
1      0.188145    0.155038    0.942480  (05:00)
2      0.159475    0.153531    0.944040  (05:01)
```

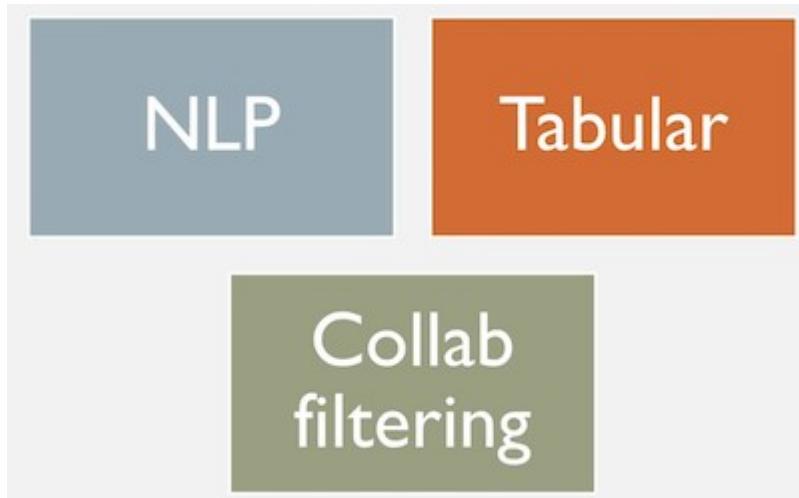
You also see I'm passing in this thing (`moms=(0.8, 0.7)`) - momentums equals 0.8,0.7. We are going to learn exactly what that means probably next week. We may even automate it. So maybe by the time you watch the video of this, this won't even be necessary anymore. Basically we found for training recurrent neural networks (RNNs), it really helps to decrease the momentum a little bit. So that's what that is.

That gets us a 94.4% accuracy after about half an hour or less of training. There's quite a lot less of training the actual classifier. We can actually get this quite a bit better with a few tricks. I don't know if we'll learn all the tricks this part. It might be next part. But even this very simple standard approach is pretty great.

IMDb		
	<i>BCN+Char+CoVe [Ours]</i>	91.8
	SA-LSTM [Dai and Le, 2015]	92.8
	bmLSTM [Radford et al., 2017]	92.9
	TRNN [Dieng et al., 2016]	93.8
	oh-LSTM [Johnson and Zhang, 2016]	94.1
	Virtual [Miyato et al., 2017]	94.1

If we compare it to last year's state of the art on IMDb, this is from The [CoVe paper](#) from McCann et al. at Salesforce Research. Their paper was 91.8% accurate. And the best paper they could find, they found a fairly domain-specific sentiment analysis paper from 2017, they've got 94.1%. And here, we've got 94.4%. And the best models I've been able to build since have been about 95.1%. So if you're looking to do text classification, this really standardized transfer learning approach works super well.

⌚ Tabular [33:10]



So that was NLP. We'll be learning more about NLP later in this course. But now, I wanted to switch over and look at tabular. Now tabular data is pretty interesting because it's the stuff that, for a lot of you, is actually what you use day-to-day at work in spreadsheets, in relational databases, etc.

Question: Where does the magic number of 2.6^4 in the learning rate come from?

[33:38]

```
learn.fit_one_cycle(2, slice(1e-3/(2.6**4),1e-3), mom=(0.8,0.7))
```

Good question. So the learning rate is various things divided by 2.6 to the fourth. The reason it's to the fourth, you will learn about at the end of today. So let's focus on the 2.6. Why 2.6? Basically, as we're going to see in more detail later today, this number, the difference between the bottom of the slice and the top of the slice is basically what's the difference between how quickly the lowest layer of the model learns versus the highest layer of the model learns. So this is called discriminative learning rates. So really the question is as you go from layer to layer, how much do I decrease the learning rate by? And we found out that for NLP RNNs, the answer is 2.6.

How do we find out that it's 2.6? I ran lots and lots of different models using lots of different sets of hyper parameters of various types (dropout, learning rates, and discriminative learning rate and so forth), and then I created something called a random forest which is a kind of model where I attempted to predict how accurate my NLP classifier would be based on the hyper parameters. And then I used random forest interpretation methods to basically figure out what the optimal parameter settings were, and I found out that the answer for this number was 2.6. So that's actually not something I've published or I don't think I've even talked about it before, so there's a new piece of information. Actually, a few months after I did this, Stephen Merity and somebody else did publish a paper describing a similar approach, so the basic idea may be out there already.

Some of that idea comes from a researcher named Frank Hutter and one of his collaborators. They did some interesting work showing how you can use random forests to actually find optimal hyperparameters. So it's kind of a neat trick. A lot of people are very interested in this thing called Auto ML which is this idea of like building models to figure out how to train your model. We're not big fans of it on the whole. But we do find that building models to better understand how your hyper parameters work, and then finding those rules of thumb like oh basically it can always be 2.6 quite helpful. So there's just something we've kind of been playing with.

↪ Back to Tabular [36:41]

Let's talk about tabular data. Tabular data such as you might see in a spreadsheet, a relational database, or financial report, it can contain all kinds of different things. I tried to make a little list of some of the kinds of things that I've seen tabular data analysis used for:



Using neural nets for analyzing tabular data - when we first presented this, people were deeply skeptical. They thought it was a terrible idea to use neural nets to analyze tabular data, because everybody knows that you should use logistic regression, random forests, or gradient boosting machines (all of which have their place for certain types of things). But since that time, it's become clear that the commonly held wisdom is wrong. It's not true that neural nets are not useful for tabular data ,in fact they are extremely useful. We've shown this in quite a few of our courses, but what's really helped is that some really effective organizations have started publishing papers and posts describing how they've been using neural nets for analyzing tabular data.

One of the key things that comes up again and again is that although feature engineering doesn't go away, it certainly becomes simpler. So Pinterest, for example, replaced the gradient boosting machines that they were using to decide how to put stuff on their homepage with neural nets. And they presented at a conference this approach, and they described how it really made engineering a lot easier because a lot of the hand created features weren't necessary anymore. You still need some, but it was just simpler. So they ended up with something that was more accurate, but perhaps even more importantly, it required less maintenance. So I wouldn't say you it's the only tool that you need in your toolbox for analyzing tabular data. But where else, I used to use random forests 99% of the time when I was doing machine learning with tabular data, I now use neural nets 90% of the time. It's my standard first go-to approach now, and it tends to be pretty reliable and effective.

One of the things that's made it difficult is that until now there hasn't been an easy way to create and train tabular neural nets. Nobody has really made it available in a library. So we've actually just created `fastai.tabular` and I think this is pretty much the first time that's become really easy to use neural nets with tabular data. So let me show you how easy it is.

⌚ Tabular examples [39:51]

[lesson4-tabular.ipynb](#)

This is actually coming directly from the examples folder in the fastai repo. I haven't changed it at all. As per usual, as well as importing fastai, import your application - so in this case, it's tabular.

```
from fastai import *
from fastai.tabular import *
```

We assume that your data is in a Pandas DataFrame. Pandas DataFrame is the standard format for tabular data in Python. There are lots of ways to get it in there, but probably the most common might be `pd.read_csv`. But whatever your data is in, you can probably get it into a Pandas data frame easily enough.

```
path = untar_data(URLS.ADULT_SAMPLE)
df = pd.read_csv(path/'adult.csv')
```

Question: What are the 10% of cases where you would not default to neural nets?

[40:41]

Good question. I guess I still tend to give them a try. But yeah, I don't know. It's kind of like as you do things for a while, you start to get a sense of the areas where things don't quite work as well. I have to think about that during the week. I don't think I have a rule of thumb. But I would say, you may as well try both. I would say try a random forest and try a neural net. They're both pretty quick and easy to run, and see how it looks. If they're roughly similar, I might dig into each and see if I can make them better. But if the random forest is doing way better, I'd probably just stick with that. Use whatever works.

So we start with the data in a data frame, and so we've got an adult sample - it's a classic old dataset. It's a pretty small simple old dataset that's good for experimenting with. And it's a CSV file, so you can read it into a data frame with Pandas read CSV (`pd.read_csv`). If your data is in a relational database, Pandas can read from that. If it's in spark or Hadoop, Pandas can read from that. Pandas can read from most stuff that you can throw at it. So that's why we use it as a default starting point.

```

dep_var = 'salary'
cat_names = ['workclass', 'education', 'marital-status', 'occupation', 'relations']
cont_names = ['age', 'fnlwgt', 'education-num']
procs = [FillMissing, Categorify, Normalize]

test = TabularList.from_df(df.iloc[800:1000].copy(), path=path, cat_names=cat_names)

data = (TabularList.from_df(df, path=path, cat_names=cat_names, cont_names=cont_names)
        .split_by_idx(list(range(800,1000)))
        .label_from_df(cols=dep_var)
        .add_test(test, label=0)
        .databunch())

```

As per usual, I think it's nice to use the data block API. So in this case, the list that we're trying to create is a tabular list and we're going to create it from a data frame. So you can tell it:

- What the data frame is.
- What the path that you're going to use to save models and intermediate steps is.
- Then you need to tell it what are your categorical variables and what are your continuous variables.

⌚ Continuous vs. Categorical [43:07]

We're going to be learning a lot more about what that means to the neural net next week, but for now the quick summary is this. Your independent variables are the things that you're using to make predictions with. So things like education, marital status, age, and so forth. Some of those variables like age are basically numbers. They could be any number. You could be 13.36 years old or 19.4 years old or whatever. Where else, things like marital status are options that can be selected from a discrete group: married, single, divorced, whatever. Sometimes those options might be quite a lot more, like occupation. There's a lot of possible occupations. And sometimes, they might be binary (i.e. true or false). But anything which you can select the answer from a small group of possibilities is called a **categorical variable**. So we're going to need to use a different approach in the neural net to modeling categorical variables to what we use for continuous variables. For categorical variables, we're going to be using something called **embeddings** which we'll be learning about later today. For continuous variables, they could just be sent into the neural net just like pixels in a neural net can. Because pixels in a neural net are already numbers; these continuous things are already numbers as well. So that's easy.

So that's why you have to tell the tabular list from data frame which ones are which. There are some other ways to do that by pre-processing them in Pandas to make things categorical variables, but it's kind of nice to have one API for doing everything; you don't have to think too much about it.

⌚ Processor [45:04]

Then we've got something which is a lot like transforms in computer vision. Transforms in computer vision do things like flip a photo on its axis, turn it a bit, brighten it, or normalize it. But for tabular data, instead of having transforms, we have things called processes. And they're nearly identical but the key difference, which is quite important, is that a processor is something that happens ahead of time. So we basically pre-process the data frame rather than doing it as we go. So transformations are really for data augmentation - we want to randomize it and do it differently each time. Or else, processes are the things that you want to do once, ahead of time.

```
procs = [FillMissing, Categorify, Normalize]
```

We have a number of processes in the fastai library. And the ones we're going to use this time are:

- `FillMissing` : Look for missing values and deal with them some way.
- `Categorify` : Find categorical variables and turn them into Pandas categories
- `Normalize` : Do a normalization ahead of time which is to take continuous variables and subtract their mean and divide by their standard deviation so they are zero-one variables.

The way we deal with missing data, we'll talk more about next week, but in short, we replace it with the median and add a new column which is a binary column of saying whether that was missing or not.

For all of these things, whatever you do to the training set, you need to do exactly the same thing to the validation set and the test set. So whatever you replaced your missing values with, you need to replace them with exactly the same thing in the validation set. So fastai handles all these details for you. They are the kinds of things that if you have to do it manually, if you're like me, you'll screw it up lots of times until you finally get it right. So that's what these processes are here.

Then we're going to split into training versus validation sets. And in this case, we do it by providing a list of indexes so the indexes from 800 to a thousand. It's very common. I don't quite remember the details of this dataset, but it's very common for wanting to keep your validation sets to be contiguous groups of things. If they're map tiles, they should be the map tiles that are next to each other, if their time periods, they should be days that are next to each other, if they are video frames, they should be video frames next to each other. Because otherwise you're kind of cheating. So it's often a good idea to use `split_by_idx` and to grab a range that's next to each other if your data has some kind of structure like that or find some other way to structure it in that way.

All right, so that's now given us a training and a validation set. We now need to add labels. In this case, the labels can come straight from the data frame we grabbed earlier, so we just have to tell it which column it is. So the dependent variable is whether they're making over \$50,000 salary. That's the thing we're trying to predict.

We'll talk about test sets later, but in this case we can add a test set. And finally get our data bunch. At that point, we have something that looks like this:

```
data.show_batch(rows=10)
```

workclass	education	marital-status	occupation	relationship	race	educationnum
Private	Prof-school	Married-civ-spouse	Prof-specialty	Husband	White	False
Self-emp-inc	Bachelors	Married-civ-spouse	Farming-fishing	Husband	White	False
Private	HS-grad	Never-married	Adm-clerical	Other-relative	Black	False
Private	10th	Married-civ-spouse	Sales	Own-child	White	False
Private	Some-college	Never-married	Handlers-cleaners	Own-child	White	False
Private	Some-college	Married-civ-spouse	Prof-specialty	Husband	White	False
?	Some-	Never-	?	Own-child	White	False

workclass	education	marital-status	occupation	relationship	race	education-num
	college	married				
Self-emp-not-inc	5th-6th	Married-civ-spouse	Sales	Husband	White	False
Private	Some-college	Married-civ-spouse	Sales	Husband	White	False
Local-gov	Some-college	Never-married	Handlers-cleaners	Own-child	White	False

There is our data. Then to use it, it looks very familiar. You get a learner, in this case it's a tabular learner, passing in the data, some information about your architecture, and some metrics. And you then call fit.

```
learn = tabular_learner(data, layers=[200,100], metrics=accuracy)
```

```
learn.fit(1, 1e-2)
```

```
Total time: 00:03
epoch  train_loss  valid_loss  accuracy
1      0.362837   0.413169   0.785000  (00:03)
```

Question: How to combine NLP (tokenized) data with meta data (tabular data) with Fastai? For instance, for IMBb classification, how to use information like who the actors are, year made, genre, etc. [49:14]

Yeah, we're not quite up to that yet. So we need to learn a little bit more about how neural net architectures work as well. But conceptually, it's kind of the same as the way we combine categorical variables and continuous variables. Basically in the neural network, you can have two different sets of inputs merging together into some layer. It could go into an early layer or into a later layer, it kind of depends. If it's like text and an image and some metadata, you probably want the text going into an RNN, the image going into a CNN, the metadata going into some kind of tabular model like this. And then you'd have them basically all concatenated together, and then go through some fully connected layers and train them end to end. We will probably largely get into that in part two. In fact we might entirely get into that in part two. I'm not sure if we have time to cover it in part one. But conceptually, it's a fairly simple extension of what we'll be learning in the next three weeks.

Question: Do you think that things like `scikit-learn` and `xgboost` will eventually become outdated? Will everyone will use deep learning tools in the future? Except for maybe small datasets? [50:36]

I have no idea. I'm not good at making predictions. I'm not a machine learning model. I mean `xgboost` is a really nice piece of software. There's quite a few really nice pieces of software for gradient boosting in particular. Actually, random forests in particular has some really nice features for interpretation which I'm sure we'll find similar versions for neural nets, but they don't necessarily exist yet. So I don't know. For now, they're both useful tools. `scikit-learn` is a library that's often used for pre-processing and running models. Again, it's hard to predict where things will end up. In some ways, it's more focused on some older approaches to modeling, but I don't know. They keep on adding new things, so we'll see. I keep trying to incorporate more `scikit-learn` stuff into `fastai` and then I keep finding ways I think I can do it better and I throw it away again, so that's why there's still no `scikit-learn` dependencies in `fastai`. I keep finding other ways to do stuff.

[52:12]

```
learn = tabular_learner(data, layers=[200,100], metrics=accuracy)
```

We're gonna learn what `layers=` means either towards the end of class today or the start of class next week, but this is where we're basically defining our architecture just like when we chose ResNet 34 or whatever for conv nets. We'll look at more about metrics in a moment, but just to remind you, metrics are just the things that get printed out. They don't change our model at all. So in this case, we're saying I want you to print out the accuracy to see how we're doing.

So that's how to do tabular. This is going to work really well because we're gonna hit our break soon. And the idea was that after three and a half lessons, we're going to hit the end of all of the quick overview of applications, and then I'm going to go down on the other side. I think we're going to be to the minute, we're going to hit it. Because the next one is collaborative filtering.

↪ Collaborative Filtering [53:08]

Collaborative filtering is where you have information about who bought what, or who liked what - it's basically something where you have something like a user, a reviewer, or whatever and information about what they've bought, what they've written about, or what they reviewed. So in the most basic version of collaborative filtering, you just have two columns: something like user ID and movie ID and that just says this user bought that movie. So for example, Amazon has a really big list of user IDs and product IDs like what did you buy. Then you can add additional information to that table such as oh, they left a review, what review did they give it? So it's now like user ID, movie ID, number of stars. You could add a timecode so this user bought this product at this time and gave it this review. But they are all basically the same structure.

There are two ways you could draw that collaborative filtering structure. One is a two-column approach where you've got user and movie. And you've got user ID, movie ID - each pair basically describes that user watch that movie, possibly also number of stars (3, 4, etc). The other way you could write it would be you could have like all the users down here and all the movies along here. And then, you can look and find a particular cell in there to find out what could be the rating of that user for that movie, or there's just a 1 there if that user watched that movie, or whatever.



So there are two different ways of representing the same information. Conceptually, it's often easier to think of it this way (the matrix on the right), but most of the time you won't store it that way. Explicitly because most of the time, you'll have what's called a very sparse matrix which is to say most users haven't watched most movies or most customers haven't purchased most products. So if you store it as a matrix where every combination of customer and product is a separate cell in that matrix, it's going to be enormous. So you tend to store it like the left or you can store it as a matrix using some kind of special sparse matrix format. If that sounds interesting, you should check out [Rachel's computational linear algebra course](#) on fastai where we have lots and lots of information about sparse matrix storage approaches. For now though, we're just going to kind of keep it in this format on left hand side.

[56:38]

[lesson4-collab.ipynb](#)

For collaborative filtering, there's a really nice dataset called MovieLens created by GroupLens group and you can download various different sizes (20 million ratings, 100,000 ratings). We've actually created an extra small version for playing around with which is what we'll start with today. And then probably next week, we'll use the bigger version.

```
from fastai import *
from fastai.collab import *
from fastai.tabular import *
```

You can grab the small version using `URLs.ML_SAMPLE`:

```
user,item,title = 'userId','movieId','title'

path = untar_data(URLs.ML_SAMPLE)
path

PosixPath('/home/jhoward/.fastai/data/movie_lens_sample')

ratings = pd.read_csv(path/'ratings.csv')
ratings.head()
```

	userId	movieId	rating	timestamp
0	73	1097	4.0	1255504951
1	561	924	3.5	1172695223
2	157	260	3.5	1291598691
3	358	1210	5.0	957481884
4	130	316	2.0	1138999234

It's a CSV so you can read it with Pandas and here it is. It's basically a list of user IDs - we don't actually know anything about who these users are. There's some movie IDs. There is some information about what the movies are, but we won't look at that until next week. Then there's the rating and the timestamp. We're going to ignore the timestamp for now. So that's a subset of our data. `head` in Pandas is just the first rows.

So now that we've got a data frame, the nice thing about collaborative filtering is it's incredibly simple.

```
data = CollabDataBunch.from_df(ratings, seed=42)

y_range = [0, 5.5]

learn = collab_learner(data, n_factors=50, y_range=y_range)
```

That's all the data that we need. So you can now go ahead and say get `collab_learner` and you can pass in the data bunch. The architecture, you have to tell it how many factors you want to use, and we're going to learn what that means after the break. And then something that could be helpful is to tell it what the range of scores are. We're going to see how that helps after the break as well. So in this case, the minimum score is 0, the maximum score is 5.

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

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

Now that you've got a learner, you can go ahead and call `fit_one_cycle` and trains for a few epochs, and there it is. So at the end of it, you now have something where you can pick a user ID and a movie ID, and guess whether or not that user will like that movie.

⌚ Cold start problem [58:55]

This is obviously a super useful application that a lot of you are probably going to try during the week. In past classes, a lot of people have taken this collaborative filtering approach back to their workplaces and discovered that using it in practice is much more tricky than this. Because in practice, you have something called the cold start problem. So the cold start problem is that the time you particularly want to be good at recommending movies is when you have a new user, and the time you particularly care about recommending a movie is when it's a new movie. But at that point, you don't have any data in your collaborative filtering system and it's really hard.

As I say this, we don't currently have anything built into fastai to handle the cold start problem and that's really because the cold start problem, the only way I know of to solve it (in fact, the only way I think that conceptually can solve it) is to have a second model which is not a collaborative filtering model but a metadata driven model for new users or new movies.

I don't know if Netflix still does this, but certainly what they used to do when I signed up to Netflix was they started showing me lots of movies and saying "have you seen this?" "did you like it?" - so they fixed the cold start problem through the UX, so there was no cold start problem. They found like 20 really common movies and asked me if I liked them, they used my replies to those 20 to show me 20 more that I might have seen, and by the time I had gone through 60, there was no cold start problem anymore.

For new movies, it's not really a problem because like the first hundred users who haven't seen the movie go in and say whether they liked it, and then the next hundred thousand, the next million, it's not a cold start problem anymore.

The other thing you can do if you, for whatever reason, can't go through that UX of asking people did you like those things (for example if you're selling products and you don't really want to show them a big selection of your products and say did you like this because you just want them to buy), you can instead try and use a metadata based tabular model what geography did they come from maybe you know their age and sex, you can try and make some guesses about the initial recommendations.

So collaborative filtering is specifically for once you have a bit of information about your users and movies or customers and products or whatever.

[1:01:37]

Question: How does the language model trained in this manner perform on code switched data (Hindi written in English words), or text with a lot of emojis?

Text with emojis, it'll be fine. There's not many emojis in Wikipedia and where they are at Wikipedia it's more like a Wikipedia page about the emoji rather than the emoji being used in a sensible place. But you can (and should) do this language model fine-tuning where you take a corpus of text where people are using emojis in usual ways, and so you fine-tune the Wikitext language model to your reddit or Twitter or whatever language model. And there aren't that many emojis if you think about it. There are hundreds of thousands of possible words that people can be using, but a small number of possible emojis. So it'll very quickly learn how those emojis are being used. So that's a piece of cake.

I'm not really familiar with Hindi, but I'll take an example I'm very familiar with which is Mandarin. In Mandarin, you could have a model that's trained with Chinese characters. There are about five or six thousand Chinese characters in common use, but there's also a romanization of those characters called pinyin. It's a bit tricky because although there's a nearly direct mapping from the character to the pinyin (I mean there is a direct mapping but that pronunciations are not exactly direct), there isn't direct mapping from the pinyin to the character because one pinyin corresponds to multiple characters.

So the first thing to note is that if you're going to use this approach for Chinese, you would need to start with a Chinese language model.

Actually fastai has something called [Language Model Zoo](#) where we're adding more and more language models for different languages, and also increasingly for different domain areas like English medical texts or even language models for things other than NLP like genome sequences, molecular data, musical MIDI notes, and so forth. So you would obviously start there.

To then convert that (in either simplified or traditional Chinese) into pinyin, you could either map the vocab directly, or as you'll learn, these multi-layer models - it's only the first layer that basically converts the tokens into a set of vectors, you can actually throw that away and fine-tune just the first layer of the model. So that second part is going to require a few more weeks of learning before you exactly understand how to do that and so forth, but if this is something you're interested in doing, we can talk about it on the forum because it's a nice test of understanding.

Question: What about time series on tabular data? is there any RNN model involved in `tabular.models` ? [\[1:05:09\]](#)

We're going to look at time series tabular data next week, but the short answer is generally speaking you don't use a RNN for time series tabular data but instead, you extract a bunch of columns for things like day of week, is it a weekend, is it a holiday, was the store open, stuff like that. It turns out that adding those extra columns which you can do somewhat automatically basically gives you state-of-the-art results. There are some good uses of RNNs for time series, but not really for these kind of tabular style time series (like retail store logistics databases, etc).

Question: Is there a source to learn more about the cold start problem? [\[1:06:14\]](#)

I'm gonna have to look that up. If you know a good resource, please mention it on the forums.

⌚ The half way point [\[1:06:34\]](#)

That is both the break in the middle of lesson 4, it's the halfway point of the course, and it's the point at which we have now seen an example of all the key applications. So the rest of this course is going to be digging deeper into how they actually work behind the scenes, more of the theory, more of how the source code is written, and so forth. So it's a good time to have a nice break. Furthermore, it's my birthday today, so it's a really special moment.

↪ Collaborative filter with Microsoft Excel [1:07:25]

[collab_filter.xlsx](#)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1						movielid	27	49	57	72	79	89	92	99	143	179	180	197	402	417	505	
2						userId	14	3	5	1	3	4	4	5	2	5	5	4	5	5	2	5
3							29	5	5	5	4	5	4	4	5	4	4	5	5	3	4	5
4							72	4	5	5	4	5	3	4.5	5	4.5	5	5	5	4.5	5	4
5							211	5	4	4	3	5	3	4	4.5	4	3	3	5	5	3	
6							212	2.5		2	5		4	2.5		5	5	3	3	4	3	2
7							293	3		4	4	4	3		3	4	4	4.5	4	4.5	4	
8							310	3	3	5	4.5	5	4.5	2	4.5	4	3	4.5	4.5	4	3	4
9							379	5	5	5	4		4	5	4	4	4		3	5	4	4
10							451	4	5	4	5	4	4	5	5	4	4	4	4	2	3.5	5
11							467	3	3.5	3	2.5			3	3.5	3.5	3	3.5	3	3	4	4
12							508	5	5	4	3	5	2	4	4	5	5	5	3	4.5	3	4.5
13							546	5	2	3	5		5	5		2.5	2	3.5	3.5	3.5	5	
14							563	1	5	3	5	4	5	5		2	5	5	3	3	4	5
15							579	4.5	4.5	3.5	3	4	4.5	4	4	4	4	3.5	3	4.5	4	4.5
16							623	5	3	3			3	5		5	5	5	2	5	4	
17																						
18																						
19																						
20																						
21																						
22																						
23																						
24							userId	27	49	57	72	79	89	92	99	143	179	180	197	402	417	505
25	0.21	1.61	2.89	-1.26			0.82	14														
26	1.55	0.75	0.22	1.62			1.26	29														
27	1.50	1.17	0.22	1.08			1.49	72														
28	0.47	0.89	1.32	1.13			0.77	211														

Microsoft Excel is one of my favorite ways to explore data and understand models. I'll make sure I put this in the repo, and actually this one, we can probably largely do in Google sheets. I've tried to move as much as I can over the last few weeks into Google sheets, but I just keep finding this is such a terrible product, so please try to find a copy of Microsoft Excel because there's nothing close, I've tried everything. Anyway, spreadsheets get a bad rap from people that basically don't know how to use them. Just like people who spend their life on Excel and then they start using Python, and they're like what the heck is this stupid thing. It takes thousands of hours to get really good at spreadsheets, but a few dozen hours to get confident at them. Once you're confident at them, you can see everything in front of you. It's all laid out, it's really great.

↪ Jeremy's spreadsheet tip of the day! [1:08:37]

I'll give you one spreadsheet tip today which is if you hold down the `ctrl` key or `command` key on your keyboard and press the arrow keys, here's `ctrl + →`, it takes you to the end of a block of a table that you're in. And it's by far the best way to move around the place, so there you go.

In this case, I want to skip around through this table, so I can hit **ctrl + ↓ →** to get to the bottom right, **ctrl + ← ↑** to get to the top left. Skip around and see what's going on.

So here's some data, and as we talked about, one way to look at collaborative filtering data is like this:

F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
userId	movielid	27	49	57	72	79	89	92	99	143	179	180	197	402	417	505
14	27	3	5	1	3	4	4	5	2	5	5	4	5	5	2	5
29	49	5	5	5	4	5	4	4	5	4	4	5	5	3	4	5
72	57	4	5	5	4	5	3	4.5	5	4.5	5	5	5	4.5	5	4
211	72	5	4	4	3	5	3	4	4.5	4	3	3	3	5	3	3
212	79	2.5		2	5		4	2.5		5	5	3	3	4	3	2
293	89	3		4	4	4	3		3	4	4	4.5	4	4.5	4	
310	92	3	3	5	4.5	5	4.5	2	4.5	4	3	4.5	4.5	4	3	4
379	99	5	5	5	4		4	5	4	4	4		3	5	4	4
451	143	4	5	4	5	4	4	5	5	4	4	4	4	2	3.5	5
467	179	3	3.5	3	2.5			3	3.5	3.5	3	3.5	3	3	4	4
508	180	5	5	4	3	5	2	4	4	5	5	5	3	4.5	3	4.5
546	197	5	2	3	5			5	5		2.5	2	3.5	3.5	3.5	5
563	402	1	5	3	5	4	5	5		2	5	5	3	3	4	5
579	417	4.5	4.5	3.5	3	4	4.5	4	4	4	4	3.5	3	4.5	4	4.5
623	505	5	3	3		3	5		5	5	5	5	2	5	4	

What we did was we grabbed from the MovieLens data the people that watched the most movies and the movies that were the most watched, and just limited the dataset down to those 15. As you can see, when you do it that way, it's not sparse anymore. There's just a small number of gaps.

This is something that we can now build a model with. How can we build a model? What we want to do is we want to create something which can predict for user 293, will they like movie 49, for example. So we've got to come up with some function that can represent that decision.

Here's a simple possible approach. We're going to take this idea of doing some matrix multiplications. So I've created here a random matrix. So here's one matrix of random numbers (the left). And I've created here another matrix of random numbers (the top). More specifically, for each movie, I've created five random numbers, and for each user, I've created five random numbers.

NB: These are initialized to random numbers					-1.69	1.49	-0.14	1.95	-0.09	1.80	1.74	0.68	0.22	1.92						
Then we use Solver to optimize them					1.01	0.12	1.36	1.49	1.17	0.73	-0.20	-0.01	2.06	1.40						
with gradient descent					0.82	1.48	0.02	0.53	1.07	1.24	1.64	0.95	0.43	0.82						
					1.89	0.50	1.74	0.41	1.57	0.49	0.20	1.54	0.43	-0.22						
					2.39	1.13	1.15	-0.74	1.14	-0.63	0.90	1.24	1.11	0.19						
					movielid	27	49	57	72	79	89	92	99	143	179					
					userId	0.21	1.61	2.89	-1.26	0.82	14	=IF(H2="",0,MMULT(\$B\$25:\$F\$25,H\$19:H\$23))	4.98	5.45						
					1.55	0.75	0.22	1.62	1.26	29	4.40	4.98	5.08	3.99	4.95	3.61	4.37	5.32	4.08	4.10
					1.50	1.17	0.22	1.08	1.49	72	4.43	4.94	4.98	4.13	4.86	3.42	4.30	4.74	4.96	4.75
					0.47	0.89	1.32	1.13	0.77	211	5.16	4.21	4.01	2.83	5.06	3.19	3.74	4.27	3.84	0.00
					0.31	2.10	1.47	-0.29	-0.15	212	1.91	0.00	2.18	4.52	0.00	3.87	2.35	0.00	4.74	4.77
					1.00	1.45	0.37	0.83	0.67	293	3.24	0.00	4.05	4.14	4.05	3.28	0.00	3.12	4.46	4.19
					1.16	1.16	0.19	2.16	-0.03	310	3.37	3.19	5.14	4.98	4.80	4.23	2.49	4.24	3.60	3.51
					0.79	1.07	1.30	1.29	0.70	379	4.92	4.68	4.41	3.84	0.00	4.00	4.19	4.62	4.26	3.93

So we could say, then, that user 14, movie 27; did they like it or not? Well, the rating, what we could do would be to multiply together this vector (red) and that vector (purple). We could do a dot product, and here's the dot product. Then we can basically do that for every possible thing in here. And thanks to spreadsheets, we can just do that in one place and copy it over, and it fills in the whole thing for us. Why would we do it this way? Well, this is the basic starting point of a neural net, isn't it? A basic starting point of a neural net is that you take the matrix multiplication of two matrices, and that's what your first layer always is. So we just have to come up with some way of saying what are two matrices that we can multiply. Clearly, you need a vector for a user (a matrix for all the users) and a vector for a movie (a matrix for all the movies) and multiply them together, and you get some numbers. So they don't mean anything yet. They're just random. But we can now use gradient descent to try to make these numbers (top) and these numbers (left) give us results that are closer to what we wanted.

So how do we do that? Well, we set this up now as a linear model, so the next thing we need is a loss function. We can calculate our loss function by saying well okay movie 27 for user ID 14 should have been a rating of 3. With this random matrices, it's actually a rating of 0.91, so we can find the sum of squared errors would be $(3 - 0.91)^2$ and then we can add them up. So there's actually a sum squared in Excel already sum X minus y squared (`SUMXMY2`), so we can use just sum X minus y squared function, passing in those two ranges and then divide by the count to get the mean.

Here is a number that is the square root of the mean squared error. You sometimes you'll see people talk about MSE so that's the Mean Squared Error, sometimes you'll see RMSE that's the Root Mean Squared Error. Since I've got a square root at the front, this is the square root mean square error.

⌚ Excel Solver [1:14:30]

We have a loss, so now all we need to do is use gradient descent to try to modify our weight matrices to make that loss smaller. Excel will do that for me.

If you don't have solver, go to Excel Options → Add-ins, and enable "Solver Add-in".

The gradient descent solver in Excel is called "Solver" and it just does normal gradient descent. You just go Data → Solver (you need to make sure that in your settings that you've enabled the solver extension which comes with Excel) and all you need to do is say which cell represents my loss function. So there it is, cell V41. Which cells contain your variables, and so you can see here, I've got H19 to V23 which is up here, and B25 to F39 which is over there, then you can just say "okay, set your loss function to a minimum by changing those cells" and click on Solve:

The screenshot shows the 'Solver Parameters' dialog box in Excel. The 'Set Objective' field is set to '\$V\$41' with the 'Min' radio button selected. The 'By Changing Variable Cells' field contains '\$H\$19:\$V\$23,\$B\$25:\$F\$39'. The 'Subject to the Constraints' section is empty. At the bottom, the 'Solving Method' dropdown is set to 'GRG Nonlinear'. A video overlay of a man speaking is visible in the top right corner.

You'll see the starts a 2.81, and you can see the numbers going down. And all that's doing is using gradient descent exactly the same way that we did when we did it manually in the notebook the other day. But it's rather than solving the mean squared error for $a @ x$ in Python, instead it is solving the loss function here which is the mean squared error of the dot product of each of those vectors by each of these vectors.

We'll let that run for a little while and see what happens. But basically in micro, here is a simple way of creating a neural network which is really in this case, it's like just a single linear layer with gradient descent to solve a collaborative filtering problem.

Let's go back and see what we do over here.

```
data = CollabDataBunch.from_df(ratings, seed=42)

y_range = [0,5.5]

learn = collab_learner(data, n_factors=50, y_range=y_range)

learn.fit_one_cycle(3, 5e-3)

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

So over here we used `collab_learner` to get a model. So the function that was called in the notebook was `collab_learner` and as you dig deeper into deep learning, one of the really good ways to dig deeper into deep learning is to dig into the fastai source code and see what's going on. So if you're going to be able to do that, you need to know how to use your editor well enough to dig through the source code. Basically there are two main things you need to know how to do:

1. Jump to a particular "symbol", like a particular class or function by its name
2. When you're looking at a particular symbol, to be able to jump to its implementation

For example in this case, I want to find `def collab_learner`. In most editors including the one I use, vim, you can set it up so that you can hit tab or something and it jumps through all the possible completions, and you can hit enter and it jumps straight to the definition for you. So here is the definition of `collab_learner`. As you can see, it's pretty small as these things tend to be, and the key thing it does is to create model of a particular kind which is an `EmbeddingDotBias` model passing in the various things you asked for. So you want to find out in your editor how you jump to the definition of that, which in vim you just hit `ctrl +]` and here is the definition of `EmbeddingDotBias`.

```

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

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

class CollabDataBunch(DataBunch): ...

class CollabLearner(Learner): ...

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

```

Now we have everything on screen at once, and as you can see there's not much going on. The models that are being created for you by fastai are actually PyTorch models. And a PyTorch model is called an `nn.Module` that's the name in PyTorch of their models. It's a little more nuanced than that, but that's a good starting point for now. When a PyTorch `nn.Module` is run (when you calculate the result of that layer, neural net, etc), specifically, it always calls a method for you called `forward`. So it's in here that you get to find out how this thing is actually calculated.

When the model is built at the start, it calls this thing called `__init__` as we've briefly mentioned before in Python people tend to call this "dunder init". So dunder init is how we create the model, and forward is how we run the model.

One thing if you're watching carefully, you might notice is there's nothing here saying how to calculate the gradients of the model, and that's because PyTorch does it for us. So you only have to tell it how to calculate the output of your model, and PyTorch will go ahead and calculate the gradients for you.

So in this case, the model contains:

- a set of weights for a user
- a set of weights for an item
- a set of biases for a user
- a set of biases for an item

And each one of those is coming from this thing called `embedding`. Here is the definition of `embedding`:

```

def embedding(ni:int,nf:int) -> nn.Module:
    "Create an embedding layer."
    emb = nn.Embedding(ni, nf)
    # See https://arxiv.org/abs/1711.09160
    with torch.no_grad(): trunc_normal_(emb.weight, std=0.01)
    return emb

```

All it does is it calls this PyTorch thing called `nn.Embedding`. In PyTorch, they have a lot of standard neural network layers set up for you. So it creates an embedding. And then this thing here (`trunc_normal_`) is it just randomizes it. This is something which creates normal random numbers for the embedding.

↪ Embedding [1:21:41]

So what's an embedding? An embedding, not surprisingly, is a matrix of weights. Specifically, an embedding is a matrix of weights that looks something like this:

NB: These are initialized to random numbers										0.71	0.92	0.68	0.83	0.60	0.18	0.26	0.91	0.99	
Then we use Solver to optimize them →										0.81	0.55	0.28	0.88	0.50	0.31	0.08	0.47	0.94	
with gradient descent										0.74	0.86	0.53	0.33	0.81	0.68	0.92	0.61	0.46	
										0.04	0.44	0.16	0.41	0.73	0.39	0.29	0.94	0.12	
										0.04	0.80	0.94	0.24	0.53	0.09	0.74	0.13	0.39	
										movield	0.04	0.80	0.94	0.24	0.53	0.09	0.74	0.13	0.39
										userId	27	49	57	72	79	89	92	99	143
0.19	0.63	0.31	0.44	0.51	14					0.91	1.40	1.02	1.12	1.27	0.66	0.89	1.13	1.18	
0.25	0.83	0.71	0.96	0.59	29					1.44	2.20	1.49	1.71	2.16	1.22	1.49	2.04	1.71	
0.30	0.44	0.19	0.00	0.72	72					0.73	1.26	1.10	0.87	0.93	0.38	0.82	0.68	1.07	
0.02	0.72	0.69	0.35	0.25	211					1.12	1.36	0.86	1.08	1.31	0.85	0.97	1.14	1.15	
0.60	0.87	0.76	0.30	0.04	212					1.71	0.00	1.14	1.65	0.00	1.02	1.03	0.00	1.82	
0.73	0.70	0.44	0.47	0.29	293					1.44	0.00	1.27	1.63	1.64	0.86	0.00	1.74	1.76	
0.23	0.81	0.36	0.47	0.12	310					1.10	1.27	0.76	1.24	1.24	0.73	0.67	1.26	1.26	
0.68	0.90	0.20	0.92	0.74	379					1.43	2.30	1.67	1.98	0.00	0.96	1.24	2.13	2.02	
0.81	0.41	0.81	0.15	0.17	451					1.52	1.88	1.28	1.41	1.55	0.90	1.15	1.59	1.66	
0.70	0.61	0.90	0.89	0.24	467					1.70	2.35	1.49	1.84	0.00	0.00	1.49	2.34	1.89	
0.50	0.27	0.73	0.44	0.83	83					1.16	2.10	1.65	1.28	1.79	0.92	1.56	1.55	1.47	

It's a matrix of weights which you can basically look up into, and grab one item out of it. So basically an embedding matrix is just a weight matrix that is designed to be something that you index into it as an array, and grab one vector out of it. That's what an embedding matrix is. In our case, we have an embedding matrix for a user and an embedding matrix for a movie. And here, we have been taking the dot product of them:

NB: These are initialized to random numbers										0.71	0.92	0.68	0.83	0.60	0.18	0.26	0.91	0.99	0.52	
Then we use Solver to optimize them →										0.81	0.55	0.28	0.88	0.50	0.31	0.08	0.47	0.94	0.70	
with gradient descent										0.74	0.86	0.53	0.33	0.81	0.68	0.92	0.61	0.46	0.64	
										0.04	0.44	0.16	0.41	0.73	0.39	0.29	0.94	0.12	0.67	
										movield	0.04	0.80	0.94	0.24	0.53	0.09	0.74	0.13	0.39	0.44
										userId	27	49	57	72	79	89	92	99	143	179
0.19	0.63	0.31	0.44	0.51	14					0.91	1.40	1.02	1.12	1.27	0.66	0.89	1.13	1.18	1.27	
0.25	0.83	0.71	0.96	0.59	29					1.44	2.20	1.49	1.71	2.16	1.22	1.49	2.04	1.71	2.08	
0.30	0.44	0.19	0.00	0.72	72					0.73	1.26	1.10	0.87	0.93	0.38	0.82	0.68	1.07	0.90	
0.02	0.72	0.69	0.35	0.25	211					1.12	1.36	0.86	1.08	1.31	0.85	0.97	1.14	1.15	0.00	
0.60	0.87	0.76	0.30	0.04	212					1.71	0.00	1.14	1.65	0.00	1.02	1.03	0.00	1.82	1.64	
0.73	0.70	0.44	0.47	0.29	293					1.44	0.00	1.27	=IF(K7="" , MMULT(\$B30:\$F30, K\$19:K\$23))							
0.23	0.81	0.36	0.47	0.12	310					1.10	1.27	0.76	1.24	0.73	0.67	1.26	1.26	1.29		
0.68	0.90	0.20	0.92	0.74	379					1.43	2.30	1.67	1.98	0.00	0.96	1.24	2.13	2.02	2.07	

But if you think about it, that's not quite enough. Because we're missing this idea that maybe there are certain movies that everybody likes more. Maybe there are some users that just tend to like movies more. So I don't really just want to multiply these two vectors together, but I really want to add a single number of like how popular is this movie, and add a single number of like how much does this user like movies in general. So those are called "bias" terms. Remember how I said there's this idea of bias and the way we dealt with that in our gradient descent notebook was we added a column of 1's. But what we tend to do in practice is we actually explicitly say I want to add a bias term. So we don't just want to have prediction equals dot product of these two things, we want to say it's the dot product of those two things plus a bias term for a movie plus a bias term for user ID.

↪ [Back to code \[1:23:55\]](#)

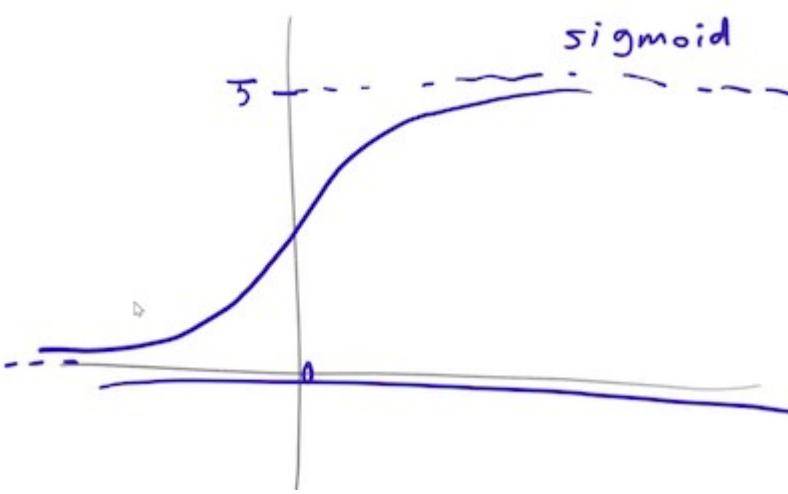
So that's basically what happens. When we set up the model, we set up the embedding matrix for the users and the embedding matrix for the items. And then we also set up the bias vector for the users and the bias vector for the items.

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

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

Then when we calculate the model, we literally just multiply the two together. Just like we did. We just take that product, we call it `dot`. Then we add the bias, and (putting aside `y_range` for a moment) that's what we return. So you can see that our model is literally doing what we did in the spreadsheet with the tweak that we're also adding the bias. So it's an incredibly simple linear model. For these kinds of collaborative filtering problems, this kind of simple linear model actually tends to work pretty well.

Then there's one tweak that we do at the end which is that in our case we said that there's `y range` of between 0 and 5.5. So here's something to point out. So you do that dot product and you add on the two biases and that could give you any possible number along the number line from very negative through to very positive numbers. But we know that we always want to end up with a number between zero and five. What if we mapped that number line like so, to this function. The shape of that function is called a sigmoid. And so, it's gonna asymptote to five and it's gonna asymptote to zero.



That way, whatever number comes out of our dot product and adding the biases, if we then stick it through this function, it's never going to be higher than 5 and never going to be smaller than 0. Now strictly speaking, that's not necessary. Because our parameters could learn a set of weights that gives about the right number. So why would we do this extra thing if it's not necessary? The reason is, we want to make its life as easy for our model as possible. If we actually set it up so it's impossible for it to ever predict too much or too little, then it can spend more of its weights predicting the thing we care about which is deciding who's going to like what movie. So this is an idea we're going to keep coming back to when it comes to like making neural network's work better. It's about all these little decisions that we make to basically make it easier for the network to learn the right thing. So that's the last tweak here:

```
return torch.sigmoid(res) * (self.y_range[1]-self.y_range[0]) + self.y_range[0]
```

We take the result of this dot product plus biases, we put it through a sigmoid. A

$\frac{1}{1+e^{-x}}$ sigmoid is just a function which is basically $\frac{1}{1+e^{-x}}$, but the definition doesn't much matter. But it just has the shape that I just mentioned, and that goes between 0 and 1. If you then multiply that by `y_range[1]` minus `y_range[0]` plus `y_range[0]`, then that's going to give you something that's between `y_range[0]` and `y_range[1]`.

So that means that this tiny little neural network, I mean it's a push to call it a neural network. But it is a neural network with one weight matrix and no nonlinearities. So it's kind of the world's most boring neural network with a sigmoid at the end. I guess it does have a non-linearity. The sigmoid at the end is the non-linearity, it only has one layer of weights. That actually turns out to give close to state-of-the-art performance. I've looked up online to find out like what are the best results people have on this MovieLens 100k database, and the results I get from this little thing is better than any of the results I can find from the standard commercial products that you can download that are specialized for this. And the trick seems to be that adding this little sigmoid makes a big difference.

Question: There was a question about how you set up your vim, and I've already linked to your [.vimrc](#) but I wanted know if you had more to say about. They really like your setup 😊

You like my setup? There's almost nothing in my setup. It's pretty bare honestly. I mean whatever you're doing with your editor, you probably want it to look like this which is when you've got a class that you're not currently working on it should be this is called folded/folding - it should be closed up so you can't see it. So you basically want something where it's easy to close and open folds, so vim already does all this for you. Then as I mentioned, you also want something where you can jump to the definition of things which in vim called using tags (e.g. to jump to the definition of `Learner`, position the cursor over `Learner` and hit `ctrl +]`). Basically vim already does all this for you. You just have to read instructions. My `.vimrc` is minimal. I basically hardly use any extensions or anything. Another great editor to use is a [Visual Studio Code](#). It's free and it's awesome and it has all the same features that you're seeing that vim does, basically VS Code does all of those things as well. I quite like using vim because I can use it on the remote machine and play around, but you can of course just clone the git repo into your local computer and open it up with VS Code to play around with. Just don't try and look through the code just on github or something. That's going to drive you crazy. You need to be able to open it and close it and jump and jump back. Maybe people can create some threads on the forum for vim tips, VS Code tips, Sublime tips, whatever. For me, I would if you're gonna pick an editor, if you want to use something on your local, I would go with the VS Code today. I think it's the best. If you want to use something on the terminal side, I would go with VIM or Emacs, to me they're clear winners.

⌚ Overview of important terminology [1:31:24]

So what I wanted to close with today is, to take this collaborative filtering example and describe how we're going to build on top of it for the next three lessons to create the more complex neural networks we've been seeing. Roughly speaking, this is the bunch of concepts that we need to learn about:

- Inputs
- Weights/parameters
 - Random
- Activations
- Activation functions / nonlinearities
- Output
- Loss
- Metric
- Cross-entropy
- Softmax
- Fine tuning

- Layer deletion and random weights
- Freezing & unfreezing

Let's think about what happens when you're using a neural network to do image recognition. Let's take a single pixel. You've got lots of pixels, but let's take a single pixel. So you've got a red a green and a blue pixel. Each one of those is some number between 0 and 255, or we normalize them so they have the mean of zero and standard deviation of one. But let's just do 0 to 255 version. So red: 10, green: 20, blue 30. So what do we do with these? Well, what we do is we basically treat that as a vector, and we multiply it by a matrix. So this matrix (depending on how you think of the rows and the columns), let's treat the matrix is having three rows and then how many columns? You get to pick. Just like with the collaborative filtering version, I decided to pick a vector of size five for each of my embedding vectors. So that would mean that's an embedding of size 5. You get to pick how big your weight matrix is. So let's make it size 5. This is 3 by 5.

Initially, this weight matrix contains random numbers. Remember we looked at embedding weight matrix just now?

```
def embedding(ni:int,nf:int) -> nn.Module:
    "Create an embedding layer."
    emb = nn.Embedding(ni, nf)
    # See https://arxiv.org/abs/1711.09160
    with torch.no_grad(): trunc_normal_(emb.weight, std=0.01)
    return emb
```

There were two lines; the first line was create the matrix, and the second was fill it with random numbers? That's all we do. I mean it all gets hidden behind the scenes by fastai and PyTorch, but that's all it's doing. So it's creating a matrix of random numbers when you set it up. The number of rows has to be 3 to match the input, and the number of columns can be as big as you like. So after you multiply the input vector by that weight matrix, you're going to end up with a vector of size 5.

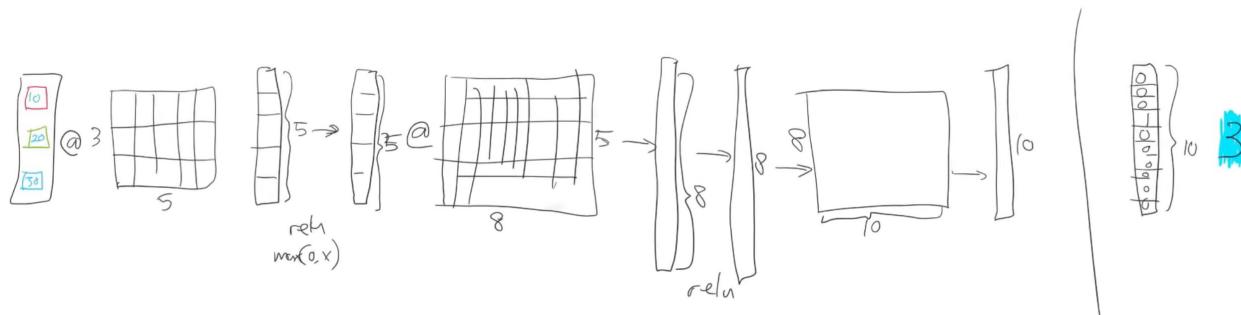
People often ask how much linear algebra do I need to know to be able to do deep learning. This is the amount you need. And if you're not familiar with this, that's fine. You need to know about matrix products. You don't need to know a lot about them, you just need to know like computationally what are they and what do they do. You've got to be very comfortable with if a matrix of size blah times a matrix of size blah gives a matrix or size blah (i.e. how do the dimensions match up). So if you have 3, and they remember in numpy and PyTorch, we use @ times 3 by 5 gives a vector of size 5.

Then what happens next; it goes through an activation function such as ReLU which is just `max(0,x)` and spits out a new vector which is, of course, going to be exactly the same size because **no activation function changes the size - it only changes the contents**. So that's still of size 5.

What happens next? We multiply by another matrix. Again, it can be any number of columns, but the number of rows has to map nicely. So it's going to be 5 by whatever. Maybe this one has 5, let's say, by 10. That's going to give some output - it should be size 10 and again we put that through ReLU, and again that gives us something of the same size.

Then we can put that through another matrix. Actually, just to make this a bit clearer (you'll see why in a moment), I'm going to use 8 not 10.

Let's say we're doing digit recognition. There are ten possible digits, so my last weight matrix has to be 10 in size. Because then that's going to mean my final output is a vector of 10 in size. Remember if you're doing that digit recognition, we take our actuals which is 10 in size. And if the number we're trying to predict was the number 3, then that means that there is a 1 in the third position ([0,0,0,1,0,...]).

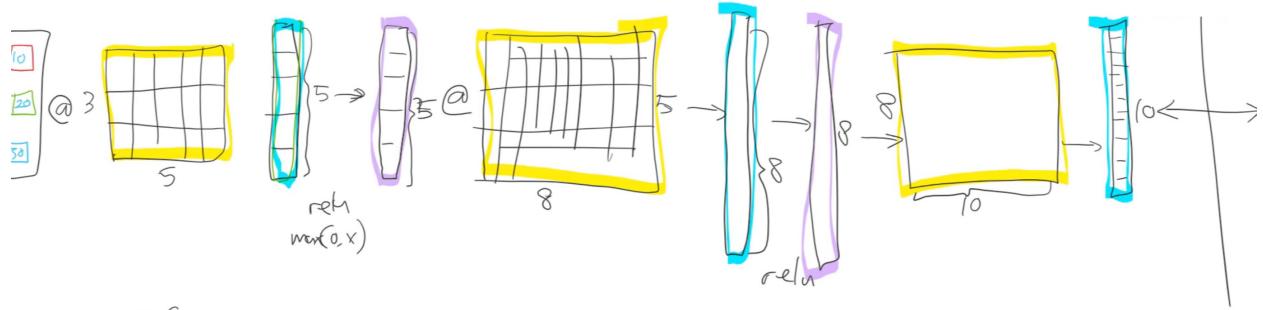


So what happens is our neural net runs along starting with our input, and going weight matrix → ReLU → weight matrix → ReLU → weight matrix → final output. Then we compare these two together to see how close they are (i.e. how close they match) using some loss function and we'll learn about all the loss functions that we use next week. For now, the only one we've learned is mean squared error. And we compare the output (you can think of them as probabilities for each of the 10) to the actual each of the 10 to get a loss, and then we find the gradients of every one of the weight matrices with respect to that, and we update the weight matrices.

The main thing I wanted to show right now is the terminology we use because it's really important.

These things (yellow) contain numbers. Specifically they initially are matrices containing random numbers. And we can refer to these yellow things, in PyTorch, they're called parameters. Sometimes we'll refer to them as weights, although weights is slightly less accurate because they can also be biases. But we kind of use the terms a little bit interchangeably. Strictly speaking, we should call them parameters.

Then after each of those matrix products, that calculates a vector of numbers. Here are some numbers (blue) that are calculated by a weight matrix multiply. And then there's some other set of numbers (purple) that are calculated as a result of a ReLU as well as the activation function. Either one is called activations.



{parameters
weights}

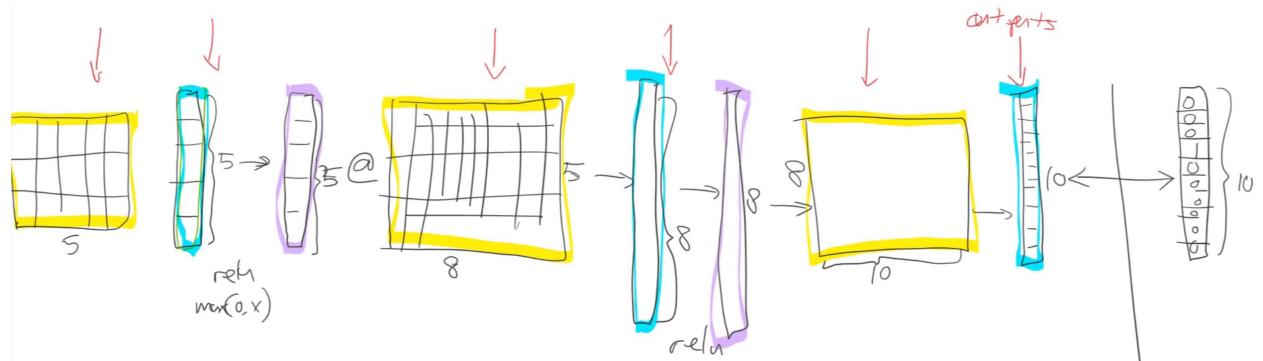
{activations}

Activations and parameters, both refer to numbers. They are numbers. But **Parameters** are numbers that are stored, they are used to make a calculation. **Activations** are the result of a calculation - the numbers that are calculated. So they're the two key things you need to remember.

So use these terms, and use them correctly and accurately. And if you read these terms, they mean these very specific things. So don't mix them up in your head. And remember, they're nothing weird and magical - they are very simple things.

- An activation is the result of either a matrix multiply or an activation function.
- Parameters are the numbers inside the matrices that we multiply by.

That's it. Then there are some special layers. Every one of these things that does a calculation, all of these things that does a calculation (red arrow), are all called layers. They're the layers of our neural net. So every layer results in a set of activations because there's a calculation that results in a set of results.



{parameters
weights}

{activations}

There's a special layer at the start which is called the input layer, and then at the end you just have a set of activations and we can refer to those special numbers (I mean they're not special mathematically but they're semantically special); we can call those the outputs. The important point to realize here is the outputs of a neural net are not actually mathematically special, they're just the activations of a layer.

So what we did in our collaborative filtering example, we did something interesting. We actually added an additional activation function right at the very end. We added an extra activation function which was sigmoid, specifically it was a scaled sigmoid which goes between 0 and 5. It's very common to have an activation function as your last layer, and it's almost never going to be a ReLU because it's very unlikely that what you actually want is something that truncates at zero. It's very often going to be a sigmoid or something similar because it's very likely that actually what you want is something that's between two values and kind of scaled in that way.

So that's nearly it. Inputs, weights, activations, activation functions (which we sometimes call nonlinearities), output, and then the function that compares those two things together is called the loss function, which so far we've used MSE.

That's enough for today. So what we're going to do next week is we're going to kind of add in a few more extra bits which is we're going to learn the loss function that's used for classification called **cross-entropy**, we're going to use the activation function that's used for single label classification called **softmax**, and we're also going to learn exactly what happens when we do fine-tuning in terms of how these layers actually, what happens with unfreeze, and what happens when we create transfer learning. Thanks everybody! Looking forward to seeing you next week.



lukemerrick Corrects a few typos in the lesson5 file (#30)

c8d93bd on Mar 28

4 contributors



1304 lines (834 sloc) 120 KB

Raw

Blame

History



⌚ Lesson 5

[Video / Lesson Forum](#)

Welcome everybody to lesson 5. And so we have officially peaked, and everything is down hill here from here as of halfway through the last lesson.

We started with computer vision because it's the most mature out-of-the-box ready to use deep learning application. It's something which if you're not using deep learning, you won't be getting good results. So the difference, hopefully, between not during lesson one versus doing lesson one, you've gained a new capability you didn't have before. And you kind of get to see a lot of the tradecraft of training and effective neural net.

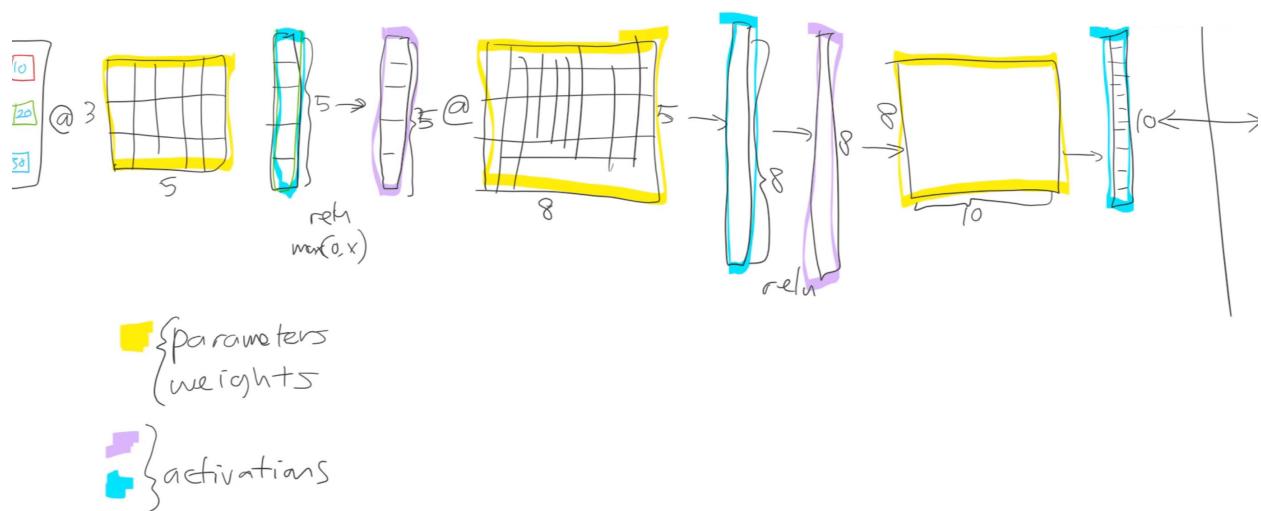
So then we moved into NLP because text is another one which you really can't do really well without deep learning generally speaking. It's just got to the point where it works pretty well now. In fact, the New York Times just featured an article about the latest advances in deep learning for text yesterday and talked quite a lot about the work that we've done in that area along with Open AI, Google, and Allen Institute of artificial intelligence.

Then we've kind of finished our application journey with tabula and collaborative filtering, partly because tabular and collaborative filtering are things that you can still do pretty well without deep learning. So it's not such a big step. It's not a whole new thing that you could do that you couldn't used to do. And also because we're going to try to get to a point where we understand pretty much every line of code and the implementations of these things, and the implementations of those things is much less intricate than vision and NLP. So as we come down this other side of the journey which is all the stuff we've just done, how does it actually work by starting where we just ended which is starting with collaborative filtering and then tabular data. We're going to be able to see what all those lines of code do by the end of today's lesson. That's our goal.

Particularly this lesson, you should not expect to come away knowing how to do applications you couldn't do before. But instead, you should have a better understanding of how we've actually been solving the applications we've seen so far. Particularly we're going to understand a lot more about regularization which is how we go about managing over versus under fitting. So hopefully you can use some of the tools from this lesson to go back to your previous projects and get a little bit more performance, or handle models where previously maybe you felt like your data was not enough, or maybe you were underfitting and so forth. It's also going to lay the groundwork for understanding convolutional neural networks and recurrent neural networks that we will do deep dives into in the next two lessons. As we do that, we're also going to look at some new applications - two new vision and NLP applications.

Review of last week [3:32]

Let's start where we left off last week. Do you remember this picture?



We were looking at what does a deep neural net look like, and we had various layers. The first thing we pointed out is that there are only and exactly two types of layer. There are layers that contain parameters, and there are layers that contain activations. Parameters are the things that your model learns. They're the things that you use gradient descent to go `parameters -= learning rate * parameters.grad`. That's what we do. And those parameters are used by multiplying them by input activations doing a matrix product.

So the yellow things are our weight matrices, your weight tensors, more generally, but that's close enough. We take some input activations or some layer activations and we multiply it by weight matrix to get a bunch of activations. So activations are numbers but these are numbers that are calculated. I find in our study group, I keep getting questions about where does that number come from. And I always answer it in the same way. "You tell me. Is it a parameter or is it an activation? Because it's one of those two things." That's where numbers come from. I guess input is a kind of a special activation. They're not calculated. They're just there, so maybe that's a special case. So maybe it's an input, a parameter, or an activation.

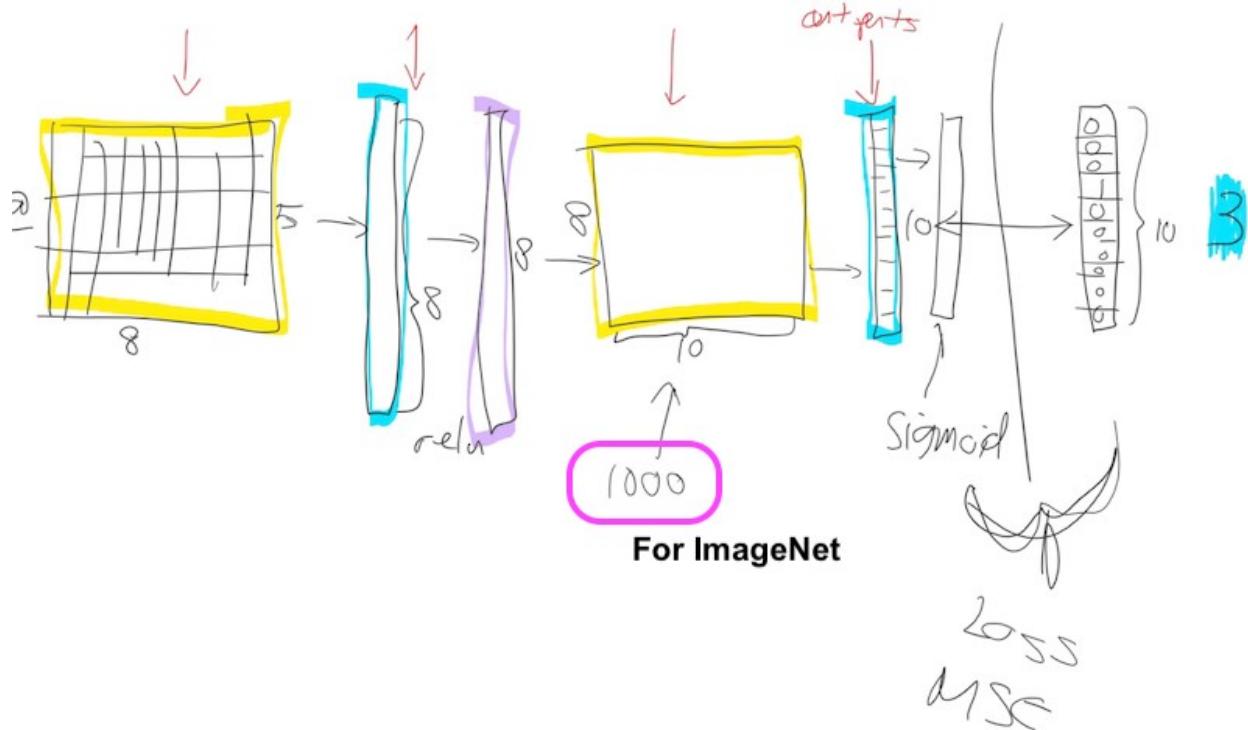
Activations don't only come out of matrix multiplications, they also come out of activation functions. And the most important thing to remember about an activation function is that it's an element-wise function. So it's a function that is applied to each element of the input, activations in turn, and creates one activation for each input element. So if it starts with a 20 long vector it creates a 20 long vector. By looking at each one of those in turn, doing one thing to it, and spitting out the answer. So an element-wise function. ReLU is the main one we've looked at, and honestly it doesn't too much matter which you pick. So we don't spend much time talking about activation functions because if you just use ReLU, you'll get a pretty good answer pretty much all the time.

Then we learnt that this combination of matrix multiplications followed by ReLUs stack together has this amazing mathematical property called the universal approximation theorem. If you have big enough weight matrices and enough of them, it can solve any arbitrarily complex mathematical function to any arbitrarily high level of accuracy (assuming that you can train the parameters, both in terms of time and data availability and so forth). That's the bit which I find particularly more advanced computer scientists get really confused about. They're always asking where's the next bit? What's the trick? How does it work? But that's it. You just do those things, and you pass back the gradients, and you update the weights with the learning rate, and that's it.

So that piece where we take the loss function between the actual targets and the output of the final layer (i.e. the final activations), we calculate the gradients with respect to all of these yellow things, and then we update those yellow things by subtracting learning rate times the gradient. That process of calculating those gradients and then subtracting like that is called **back propagation**. So when you hear the term back propagation, it's one of these terms that neural networking folks love to use - it sounds very impressive but you can replace it in your head with `weights -= weight.grad * learning rate` or parameters, I should say, rather than weights (a bit more general). So that's what we covered last week. Then I mentioned last week that we're going to cover a couple more things. I'm going to come back to these ones "cross-entropy" and "softmax" later today.

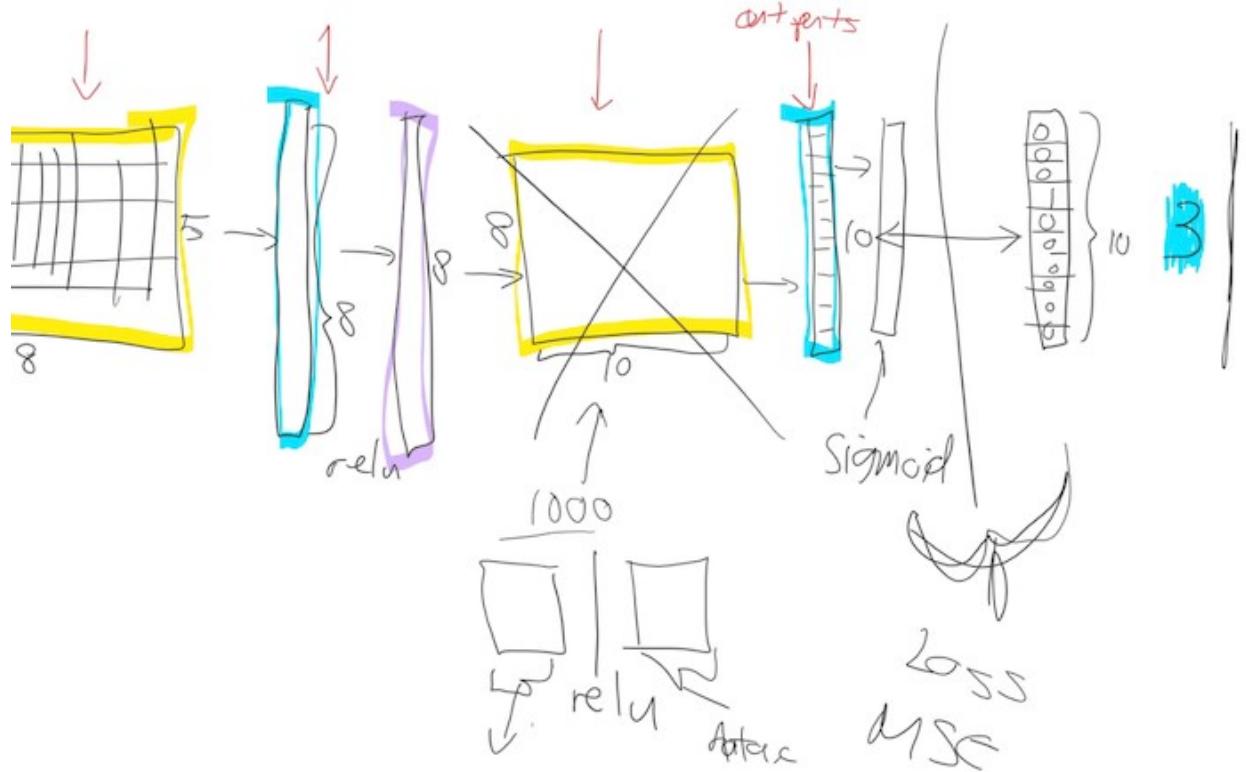
⌚ Fine tuning [8:45]

Let's talk about fine-tuning. So what happens when we take a ResNet 34 and we do transfer learning? What's actually going on? The first thing to notice is the ResNet34 we grabbed from ImageNet has a very specific weight matrix at the end. It's a weight matrix that has 1000 columns:



Why is that? Because the problem they asked you to solve in the ImageNet competition is please figure out which one of these 1000 image categories this picture is. So that's why they need a 1000 things here because in ImageNet, this target vector is length 1000. You've got to pick the probability that it's which one of those thousand things.

There's a couple of reasons this weight matrix is no good to you when you're doing transfer learning. The first is that you probably don't have a thousand categories. I was trying to do teddy bears, black bears, or brown bears. So I don't want a thousand categories. The second is even if I did have exactly a thousand categories, they're not the same thousand categories that are in ImageNet. So basically this whole weight matrix is a waste of time for me. So what do we do? We throw it away. When you go `create_cnn` in fastai, it deletes that. And what does it do instead? Instead, it puts in two new weight matrices in there for you with a ReLU in between.



There are some defaults as to what size this first one is, but the second one the size there is as big as you need it to be. So in your data bunch which you passed to your learner, from that we know how many activations you need. If you're doing classification, it's how many ever classes you have, if you're doing regression it's how many ever numbers you're trying to predict in the regression problem. So remember, if your data bunch is called `data` that'll be called `data.c`. So we'll add for you this weight matrix of size `data.c` by however much was in the previous layer.

[11:08]

Okay so now we need to train those because initially these weight matrices are full of random numbers. Because new weight matrices are always full of random numbers if they are new. And these ones are new. We're just we've grabbed them and thrown them in there, so we need to train them. But the other layers are not new. The other layers are good at something. What are they good at? Let's remember that [Zeiler and Fergus paper](#). Here are examples of some visualization of some filters some weight matrices in the first layer and some examples of some things that they found.

Visualizing and Understanding Convolutional Networks

Matthew D. Zeiler

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

ZEILER@CS.NYU.EDU

Rob Fergus

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

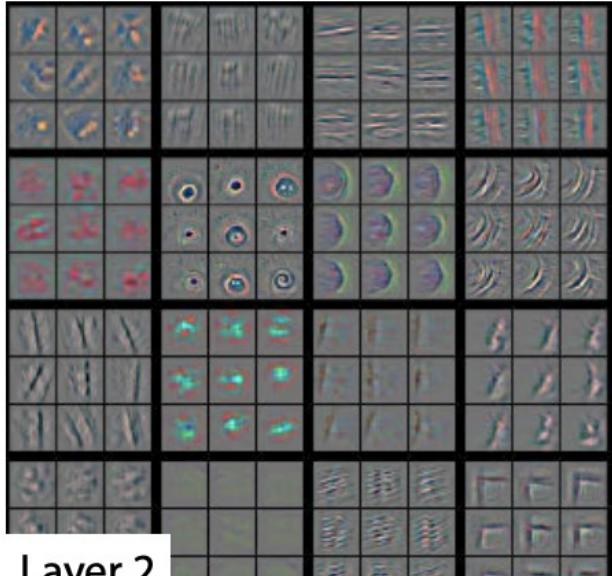
FERGUS@CS.NYU.EDU



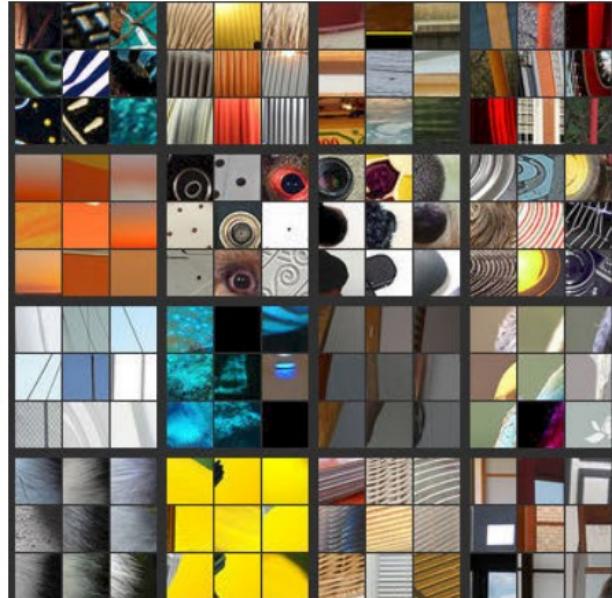
Layer 1



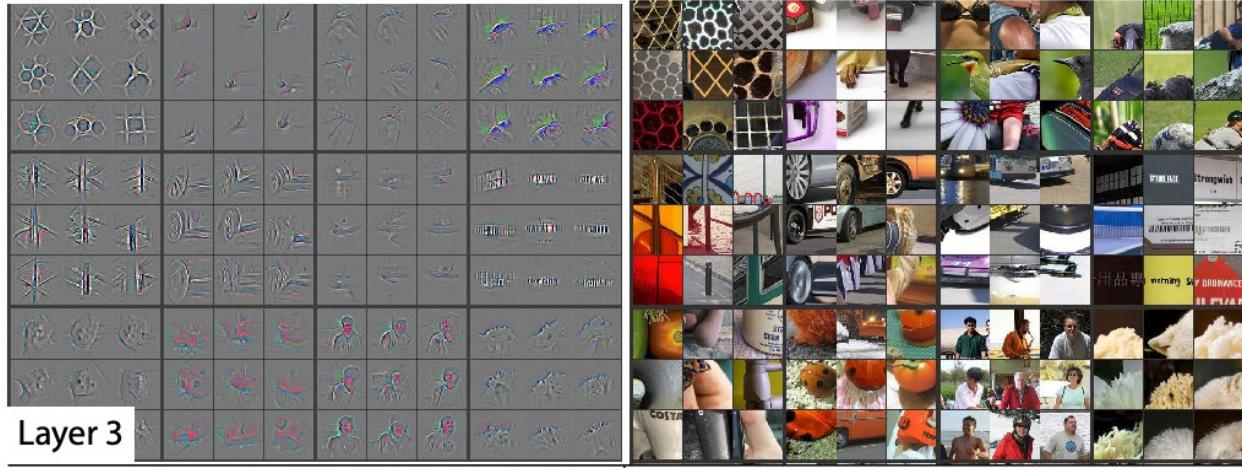
So the first layer had one part of the weight matrix was good at finding diagonal edges in this direction.



Layer 2



And then in layer 2, one of the filters was good at finding corners in the top left.

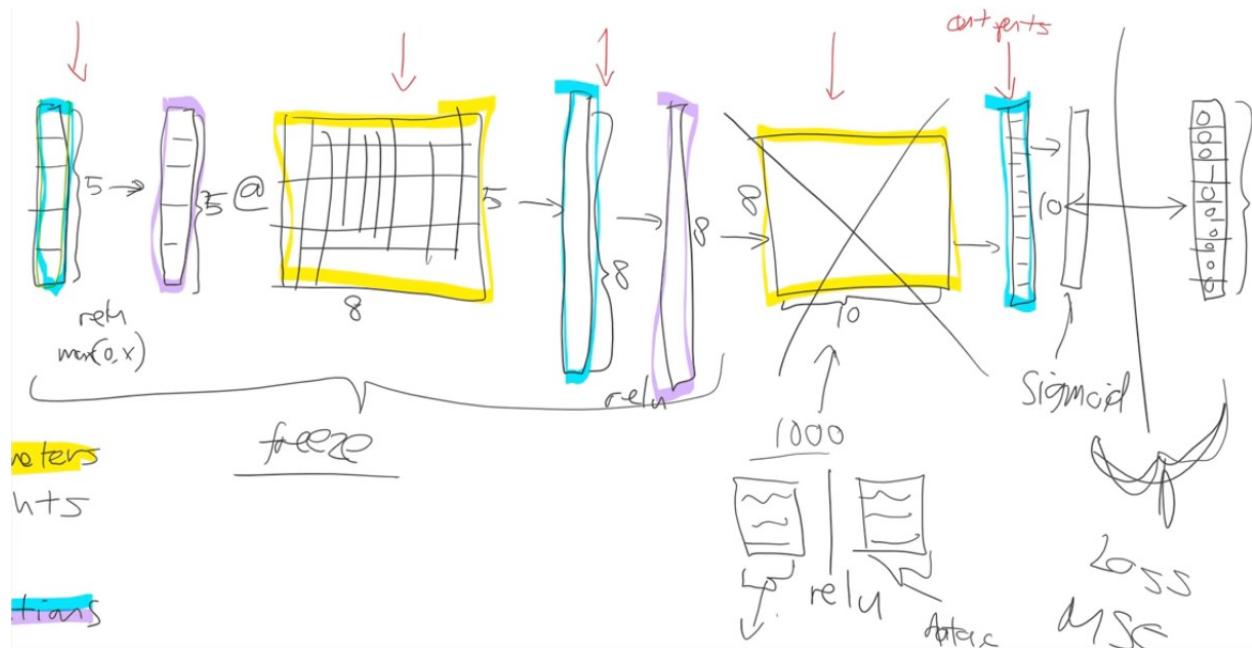


Then in layer 3 one of the filters was good at finding repeating patterns, another one was good at finding round orange things, another one was good at finding kind of like hairy or floral textures.

So as we go up, they're becoming more sophisticated, but also more specific. So like layer 4, I think, was finding like eyeballs, for instance. Now if you're wanting to transfer and learn to something for histopathology slides, there's probably going to be no eyeballs in that, right? So the later layers are no good for you. But there'll certainly be some repeating patterns and diagonal edges. So the earlier you go in the model, the more likely it is that you want those weights to stay as they are.

⌚ Freezing layers [13:00]

To start with, we definitely need to train these new weights because they're random. So let's not bother training any of the other weights at all to start with. So what we do is we basically say let's freeze.



Let's freeze all of those other layers. So what does that mean? All that means is that we're asking fastai and PyTorch that when we train (however many epochs we do), when we call `fit`, don't back propagate the gradients back into those layers. In other words, when you go `parameters=parameters - learning rate * gradient`, only do it for the new layers, don't bother doing it for the other layers. That's what freezing means - just means don't update those parameters.

So it'll be a little bit faster as well because there's a few less calculations to do. It'll take up a little bit less memory because there's a few less gradients that we have to store. But most importantly it's not going to change weights that are already better than nothing - they're better than random at the very least.

So that's what happens when you call `freeze`. It doesn't freeze the whole thing. It freezes everything except the randomly generated added layers that we put on for you.

⌚ Unfreezing and Using Discriminative Learning Rates

Then what happens next? After a while we say "okay this is looking pretty good. We probably should train the rest of the network now". So we unfreeze. Now we're gonna chain the whole thing, but we still have a pretty good sense that these new layers we added to the end probably need more training, and these ones right at the start (e.g. diagonal edges) probably don't need much training at all. So we split our model into a few sections. And we say "let's give different parts of the model different learning rates." So the earlier part of the model, we might give a learning rate of `1e- 5`, and the later part of the model we might give a learning rate of `1e-3`, for example.

So what's gonna happen now is that we can keep training the entire network. But because the learning rate for the early layers is smaller, it's going to move them around less because we think they're already pretty good and also if it's already pretty good to the optimal value, if you used a higher learning rate, it could kick it out - it could actually make it worse which we really don't want to happen. So this process is called using **discriminative learning rates**. You won't find much online about it because I think we were kind of the first to use it for this purpose (or at least talked about it extensively). Maybe other people used it without writing it down. So most of the stuff you'll find about this will be fastai students. But it's starting to get more well-known slowly now. It's a really really important concept. For transfer learning, without using this, you just can't get nearly as good results.

How do we do discriminative learning rates in fastai? Anywhere you can put a learning rate in fastai such as with the `fit` function. The first thing you put in is the number of epochs and then the second thing you put in is learning rate (the same if you use `fit_one_cycle`). The learning rate, you can put a number of things there:

- You can put a single number (e.g. `1e-3`): Every layer gets the same learning rate. So you're not using discriminative learning rates.

- You can write a slice. So you can write slice with a single number (e.g. `slice(1e-3)`): The final layers get a learning rate of whatever you wrote down (`1e-3`), and then all the other layers get the same learning rate which is that divided by 3. So all of the other layers will be $1e-3/3$. The last layers will be `1e-3`.
- You can write slice with two numbers (e.g. `slice(1e-5, 1e-3)`). The final layers (these randomly added layers) will still be again `1e-3`. The first layers will get `1e-5`, and the other layers will get learning rates that are equally spread between those two - so multiplicatively equal. If there were three layers, there would be `1e-5, 1e-4, 1e-3`, so equal multiples each time.

One slight tweak - to make things a little bit simpler to manage, we don't actually give a different learning rate to every layer. We give a different learning rate to every "layer group" which is just we decided to put the groups together for you. Specifically what we do is, the randomly added extra layers we call those one layer group. This is by default. You can modify it. Then all the rest, we split in half into two layer groups.

By default (at least with a CNN), you'll get three layer groups. If you say `slice(1e-5, 1e-3)`, you will get `1e-5` learning rate for the first layer group, `1e-4` for the second, `1e-3` for the third. So now if you go back and look at the way that we're training, hopefully you'll see that this makes a lot of sense.

This divided by three thing, it's a little weird and we won't talk about why that is until part two of the course. It's a specific quirk around batch normalization. So we can discuss that in the advanced topic if anybody's interested.

That is fine tuning. Hopefully that makes that a little bit less mysterious.

[19:49]

So we were looking at collaborative filtering last week.

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

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

In [ ]: y_range = [0,5.5]

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

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

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

And in the collaborative filtering example, we called `fit_one_cycle` and we passed in just a single number. That makes sense because in collaborative filtering, we only have one layer. There's a few different pieces in it, but there isn't a matrix multiply followed by an activation function followed by another matrix multiply.

⌚ Affine Function [20:24]

I'm going to introduce another piece of jargon here. They're not always exactly matrix multiplications. There's something very similar. They're linear functions that we add together, but the more general term for these things that are more general than matrix multiplications is **affine functions**. So if you hear me say the word affine function, you can replace it in your head with matrix multiplication. But as we'll see when we do convolutions, convolutions are matrix multiplications where some of the weights are tied. So it would be slightly more accurate to call them affine functions. And I'd like to introduce a little bit more jargon in each lesson so that when you read books or papers or watch other courses or read documentation, there will be more of the words you recognize. So when you see affine function, it just means a linear function. It means something very very close to matrix multiplication. And matrix multiplication is the most common kind of affine function, at least in deep learning.

Specifically for collaborative filtering, the model we were using was this one:

NB: These are initialized to random numbers	0.71	0.92	0.68	0.83	0.60	0.18	0.26	0.91	0.99					
Then we use Solver to optimize them with gradient descent	0.81	0.55	0.28	0.88	0.50	0.31	0.08	0.47	0.94					
	0.74	0.86	0.53	0.33	0.81	0.68	0.92	0.61	0.46					
	0.04	0.44	0.16	0.41	0.73	0.39	0.29	0.94	0.12					
	0.04	0.80	0.94	0.24	0.53	0.09	0.74	0.13	0.39					
userId	27	49	57	72	79	89	92	99	143					
0.19	0.63	0.31	0.44	0.51	14	0.91	1.40	1.02	1.12	1.27	0.66	0.89	1.13	1.18
0.25	0.83	0.71	0.96	0.59	29	1.44	2.20	1.49	1.71	2.16	1.22	1.49	2.04	1.71
0.30	0.44	0.19	0.00	0.72	72	0.73	1.26	1.10	0.87	0.93	0.38	0.82	0.68	1.07
0.02	0.72	0.69	0.35	0.25	211	1.12	1.36	0.86	1.08	1.31	0.85	0.97	1.14	1.15
0.60	0.87	0.76	0.30	0.04	212	1.71	0.00	1.14	1.65	0.00	1.02	1.03	0.00	1.82
0.73	0.70	0.44	0.47	0.29	293	1.44	0.00	1.27	1.63	1.64	0.86	0.00	1.74	1.76
0.23	0.81	0.36	0.47	0.12	310	1.10	1.27	0.76	1.24	1.24	0.73	0.67	1.26	1.26
0.68	0.90	0.20	0.92	0.74	379	1.43	2.30	1.67	1.98	0.00	0.96	1.24	2.13	2.02
0.81	0.41	0.81	0.15	0.17	451	1.52	1.88	1.28	1.41	1.55	0.90	1.15	1.59	1.66
0.70	0.61	0.90	0.89	0.24	467	1.70	2.35	1.49	1.84	0.00	0.00	1.49	2.34	1.89
0.50	0.27	0.73	0.44	0.83	3	1.16	2.10	1.65	1.28	1.79	0.92	1.56	1.55	1.47

It was where we had a bunch of numbers here (left) and a bunch of numbers here (top), and we took the dot product of them. And given that one here is a row and one is a column, that's the same as a matrix product. So `MMULT` in Excel multiplies matrices, so here's the matrix product of those two.

I started this training last week by using solver in Excel and we never actually went back to see how it went. So let's go and have a look now. The average sum of squared error got down to 0.39. We're trying to predict something on a scale of 0.5 to 5, so on average we're being wrong by about 0.4. That's pretty good. And you can kind of see it's pretty good, if you look at like 3, 5, 1 is what it meant to be, 3.23, 5.1, 0.98 - that's pretty close. So you get the general idea.

↪ Embedding [22:51]

Then I started to talk about this idea of embedding matrices. In order to understand that, let's put this worksheet aside and look at [another worksheet](#) ("movielens_1hot" tab).

Here's another worksheet. What I've done here is I have copied over those two weight matrices from the previous worksheet. Here's the one for users, and here's the one for movies. And the movies one I've transposed it, so it's now got exactly the same dimensions as the users one.

So here are two weight matrices (in orange). Initially they were random. We can train them with gradient descent. In the original data, the user IDs and movie IDs were numbers like these. To make life more convenient, I've converted them to numbers from 1 to 15 (`user_idx` and `movie_idx`). So in these columns, for every rating, I've got user ID movie ID rating using these mapped numbers so that they're contiguous starting at one.

Now I'm going to replace user ID number 1 with this vector - the vector contains a 1 followed by 14 zeros. Then user number 2, I'm going to replace with a vector of 0 and then 1 and then 13 zeros. So movie ID 14, I've also replaced with another vector which is 13 zeros and then a 1 and then a 0. These are called one-hot encodings, by the way. This is not part of a neural net. This is just like some input pre-processing where I'm literally making this my new input:

So this is my new inputs for my movies, this is my new inputs for my users. These are the inputs to a neural net.

What I'm going to do now is I'm going to take this input matrix and I'm going to do a matrix multiplied by the weight matrix. That'll work because weight matrix has 15 rows, and this (one-hot encoding) has 15 columns. I can multiply those two matrices together because they match. You can do matrix multiplication in Excel using the `MMULT` function. Just be careful if you're using Excel. Because this is a function that returns multiple numbers, you can't just hit enter when you finish with it; you have to hit `ctrl + shift + enter`. `ctrl + shift + enter` means this is an array function - something that returns multiple values.

Here (`User activations`) is the matrix product of this input matrix of inputs, and this parameter matrix or weight matrix. So that's just a normal neural network layer. It's just a regular matrix multiply. So we can do the same thing for movies, and so here's the matrix multiply for movies.

Well, here's the thing. This input, we claim, is this one hot encoded version of user ID number 1, and these activations are the activations for user ID number one. Why is that? Because if you think about it, the matrix multiplication between a one hot encoded vector and some matrix is actually going to find the Nth row of that matrix when the one is in position N. So what we've done here is we've actually got a matrix multiply that is creating these output activations. But it's doing it in a very interesting way - it's effectively finding a particular row in the input matrix.

Having done that, we can then multiply those two sets together (just a dot product), and we can then find the loss squared, and then we can find the average loss.

f _x	=SQRT(AVERAGE(BH3:BH209))	BA	BB	BC	BD	BE	BF	BG	BH																			
A	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK	ALAMANAO	APA	QARAS	AT	AU	AV	AW	AX	AY	AZ	Movie activations	Preds	Loss					
0	0	0	1.16	1.16	0.19	2.16	-0.03	0	0	0	0	0	0	0	0	0	1	0	0	0	1.69	0.91	0.71	0.19	0.43	3.54	0.93	
0	0	0	0.79	1.07	1.30	1.29	0.70	0	0	0	0	0	0	0	0	0	1	0	0	0	1.69	0.91	0.71	0.19	0.43	3.77	0.60	
0	0	0	1.52	0.54	0.64	1.36	0.94	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1.69	0.91	0.71	0.19	0.43	4.17	0.03
0	0	0	1.00	0.69	0.41	0.75	1.02	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1.69	0.91	0.71	0.19	0.43	3.19	0.03
0	0	0	0.86	1.29	0.80	0.19	1.79	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1.69	0.91	0.71	0.19	0.43	3.99	0.97
0	0	0	0.61	-0.09	2.40	1.57	-0.18	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1.69	0.91	0.71	0.19	0.43	2.87	0.39
1	0	0	1.45	0.59	1.40	1.29	-0.13	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1.69	0.91	0.71	0.19	0.43	4.15	1.33
0	1	0	0.68	0.95	1.53	0.84	0.64	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1.69	0.91	0.71	0.19	0.43	3.52	0.27
0	0	1	1.70	1.00	0.20	-0.25	2.05	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1.69	0.91	0.71	0.19	0.43	4.75	0.06
0	0	0	0.21	1.61	2.89	-1.26	0.82	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	4.90	0.01
0	0	0	1.55	0.75	0.22	1.62	1.26	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	3.53	0.28
0	0	0	1.50	1.17	0.22	1.08	1.49	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	3.90	0.36
0	0	0	0.47	0.89	1.32	1.13	0.77	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	4.91	0.01
0	0	0	0.31	2.10	1.47	-0.29	-0.15	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	4.59	0.35
0	0	0	1.00	1.45	0.37	0.83	0.67	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	3.90	0.37
0	0	0	1.16	1.16	0.19	2.16	-0.03	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	4.06	0.00
0	0	0	0.79	1.07	1.30	1.29	0.70	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	5.01	0.00
0	0	0	1.52	0.54	0.64	1.36	0.94	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	2.86	0.73
0	0	0	1.00	0.69	0.41	0.75	1.02	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	2.76	0.06
0	0	0	0.86	1.29	0.80	0.19	1.79	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	4.54	0.00
0	0	0	0.61	-0.09	2.40	1.57	-0.18	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	3.49	0.00
1	0	0	1.45	0.59	1.40	1.29	-0.13	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	2.50	0.25
0	1	0	0.68	0.95	1.53	0.84	0.64	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	4.45	0.00
0	0	1	1.70	1.00	0.20	-0.25	2.05	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-1.16	1.93	0.99	1.39	1.11	2.08	0.01

And lo and behold, that number 0.39 is the same as this number (from the solver) because they're doing the same thing.

This one ("dotprod" version) was finding this particular user's embedding vector, this one ("movielens_1hot" version) is just doing a matrix multiply, and therefore we know they are **mathematically identical**.

↪ Embedding once over [27:57]

So let's lay that out again. Here's our final version (*recommend watching a video for this section*):

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W				
1																											
2		Users	Embeddings										original data		user embedding												
3	idx	Row Lab											userid	movielid	rating	user idx	1	2	3	4	5	movie idx	1	2	3	4	5
4	1	14		0.21	1.61	2.89	-1.26	0.82					14	417	2	1	0.21	1.61	2.89	-1.26	0.82	14	1.66	0.66	0.57	0.46	0.47
5	2	29		1.55	0.75	0.22	1.62	1.26					29	417	4	2	1.55	0.75	0.22	1.62	1.26	14	1.66	0.66	0.57	0.46	0.47
6	3	72		1.50	1.17	0.22	1.08	1.49					72	417	5	3	1.50	1.17	0.22	1.08	1.49	14	1.66	0.66	0.57	0.46	0.47
7	4	211		0.47	0.47	1.32	1.13	0.77					211	417	3	4	0.47	0.89	1.32	1.13	0.77	14	1.66	0.66	0.57	0.46	0.47
8	5	212		0.31	2.10	1.47	-0.29	-0.15					212	417	3	5	0.31	2.10	1.47	-0.29	-0.15	14	1.66	0.66	0.57	0.46	0.47
9	6	293		1.00	1.45	0.37	0.83	0.67					293	417	4	6	1.00	1.45	0.37	0.83	0.67	14	1.66	0.66	0.57	0.46	0.47
10	7	310		1.16	1.16	0.19	2.16	-0.03					310	417	3	7	1.16	1.16	0.19	2.16	-0.03	14	1.66	0.66	0.57	0.46	0.47
11	8	379		0.79	1.07	1.30	1.29	0.70					379	417	4	8	0.79	1.07	1.30	1.29	0.70	14	1.66	0.66	0.57	0.46	0.47
12	9	451		1.52	0.54	0.64	1.36	0.94					451	417	3.5	9	1.52	0.54	0.64	1.36	0.94	14	1.66	0.66	0.57	0.46	0.47
13	10	467		1.00	0.69	0.41	0.75	1.02					467	417	4	10	1.00	0.69	0.41	0.75	1.02	14	1.66	0.66	0.57	0.46	0.47
14	11	508		0.86	1.29	0.80	0.19	1.79					508	417	3	11	0.86	1.29	0.80	0.19	1.79	14	1.66	0.66	0.57	0.46	0.47
15	12	546		0.61	-0.09	2.40	1.57	-0.18					546	417	3.5	12	0.61	-0.09	2.40	1.57	-0.18	14	1.66	0.66	0.57	0.46	0.47
16	13	563		1.45	0.59	1.40	1.29	-0.13					563	417	4	13	1.45	0.59	1.40	1.29	-0.13	14	1.66	0.66	0.57	0.46	0.47
17	14	579		0.68	0.95	1.53	0.84	0.64					579	417	4	14	0.68	0.95	1.53	0.84	0.64	14	1.66	0.66	0.57	0.46	0.47
18	15	623		1.70	1.00	0.20	-0.25	2.05					623	417	5	15	1.70	1.00	0.20	-0.25	2.05	14	1.66	0.66	0.57	0.46	0.47
19													14	27	3	1	0.21	1.61	2.89	-1.26	0.82	1	-1.69	1.01	0.82	1.89	2.39
20		Movies											29	27	5	2	1.55	0.75	0.22	1.62	1.26	1	-1.69	1.01	0.82	1.89	2.39
21	idx	Row Lab											72	27	4	3	1.50	1.17	0.22	1.08	1.49	1	-1.69	1.01	0.82	1.89	2.39
22	1	27		-1.69	1.01	0.82	1.89	2.39					211	27	5	4	0.47	0.89	1.32	1.13	0.77	1	-1.69	1.01	0.82	1.89	2.39
23	2	49		1.49	0.12	1.48	0.50	1.13					212	27	2.5	5	0.31	2.10	1.47	-0.29	-0.15	1	-1.69	1.01	0.82	1.89	2.39
7:02 PM 11/19/2018				dotprod	movielens_1hot	movielens_emb	bias	word_emb					293	27	3	6	1.00	1.45	0.37	0.83	0.67	1	-1.69	1.01	0.82	1.89	2.39

This is the same weight matrices again - exactly the same I've copied them over. And here's those user IDs and movie IDs again. But this time, I've laid them out just in a normal tabular form just like you would expect to see in the input to your model. And this time, I have got exactly the same set of activations here (user embedding) that I had in movielens_1hot. But in this case I've calculated these activations using Excel's `OFFSET` function which is an array look up. This version (`movielens_emb`) is identical to `movielens_1hot` version, but obviously it's much less memory intensive and much faster because I don't actually create the one hot encoded matrix and I don't actually do a matrix multiply. That matrix multiply is nearly all multiplying by zero which is a total waste of time.

So in other words, multiplying by a one hot encoded matrix is identical to doing an array lookup. Therefore **we should always do the array lookup version**, and therefore we have a specific way of saying I want to do a matrix multiplication by a one hot encoded matrix without ever actually creating it. I'm just instead going to pass in a bunch of integers and pretend they're one hot encoded. And that is called an **embedding**.

You might have heard this word "embedding" all over the places as if it's some magic advanced mathy thing, but embedding means look something up in an array. But it's interesting to know that looking something up in an array is mathematically identical to doing a matrix product by a one hot encoded matrix. And therefore, an embedding fits very nicely in our standard model of our neural networks work.

Now suddenly it's as if we have another whole kind of layer. It's a kind of layer where we get to look things up in an array. But we actually didn't do anything special. We just added this computational shortcut - this thing called an embedding which is simply a fast memory efficient way of multiplying by hot encoded matrix.

So this is really important. Because when you hear people say embedding, you need to replace it in your head with "an array lookup" which we know is mathematically identical to matrix multiply by a one hot encoded matrix.

Here's the thing though, it has kind of interesting semantics. Because when you do multiply something by a one hot encoded matrix, you get this nice feature where the rows of your weight matrix, the values only appear (for row number one, for example) where you get user ID number one in your inputs. So in other words you kind of end up with this weight matrix where certain rows of weights correspond to certain values of your input. And that's pretty interesting. It's particularly interesting here because (going back to a kind of most convenient way to look at this) because the only way that we can calculate an output activation is by doing a dot product of these two input vectors. That means that they kind of have to correspond with each other. There has to be some way of saying if this number for a user is high and this number for a movie is high, then the user will like the movie. So the only way that can possibly make sense is if **these numbers represent features of personal taste and corresponding features of movies**. For example, the movie has John Travolta in it and user ID likes John Travolta, then you'll like this movie.

We're not actually deciding the rows mean anything. We're not doing anything to make the rows mean anything. But the only way that this gradient descent could possibly come up with a good answer is if it figures out what the aspects of movie taste are and the corresponding features of movies are. So those underlying kind of features that appear that are called **latent factors** or **latent features**. They're these hidden things that were there all along, and once we train this neural net, they suddenly appear.

⌚ Bias [33:08]

Now here's the problem. No one's going to like *Battlefield Earth*. It's not a good movie even though it has John Travolta in it. So how are we going to deal with that? Because there's this feature called I like John Travolta movies, and this feature called this movie has John Travolta, and so this is now like you're gonna like the movie. But we need to save some way to say "unless it's *Battlefield Earth*" or "you're a Scientologist" - either one. So how do we do that? We need to add in **bias**.

I28	fx	=IF(I4="",0,MMULT(\$B28:\$F28,I\$20:I\$24))+I\$19+\$A28
19		2.63 -2.01 -3.19 0.87 -1.11 1.19 -1.69 1.68 4.29 -2.15 -3.16 1.96 -5.56 0.36 -1.40
20		-2.10 1.77 0.12 2.19 0.02 1.99 2.15 0.09 -0.51 2.54 2.27 1.59 -0.58 1.44 1.68
21		-0.73 0.08 2.76 1.49 1.74 0.42 -0.40 -0.20 0.08 2.11 2.39 0.56 4.18 0.79 0.02
22		0.96 2.12 0.62 0.26 1.11 0.87 2.24 0.60 0.19 1.15 1.07 0.22 1.62 0.49 1.95
23		1.69 0.88 2.25 0.08 1.69 0.13 0.50 1.10 0.20 0.15 0.82 -0.23 2.61 0.41 0.91
24		movielid 2.58 1.77 1.38 -1.23 1.34 -0.89 1.45 1.33 0.61 0.14 0.66 -0.07 1.67 0.45 1.30
25		userid 27 49 57 72 79 89 92 99 143 179 180 197 402 417 505
26		-0.20 -0.18 2.03 3.75 -1.96 0.64 14 3.26 4.97 0.94 3.32 3.92 3.92 5.23 1.99 5.08 5.55 3.87 3.80 4.85 2.80 4.50
27		-0.32 1.86 0.66 -0.12 2.06 1.31 29 4.68 4.91 4.91 4.13 4.87 3.82 4.37 5.33 4.27 4.00 4.76 4.37 3.18 4.63 4.76
28		-0.26 1.66 1.37 -0.04 0.81 1.95 72 4.25 =IF(I4="",0,MMULT(\$B28:\$F28,I\$20:I\$24))+I\$19+\$A28 4.77 4.25 4.77 4.35
29		-0.84 0.75 0.96 1.51 1.62 0.59 211 5.21 4.22 4.08 2.91 4.95 3.23 3.71 4.17 4.12 -2.99 3.52 2.76 4.82 3.03 -2.24
30		0.26 -0.06 1.87 2.29 -0.67 -0.18 212 2.25 -1.75 1.89 4.57 -0.84 4.20 2.22 1.94 4.94 4.42 3.22 3.83 4.24 2.78 2.44
31		-0.59 1.08 1.67 0.45 0.62 1.08 293 2.83 -2.60 4.11 3.97 4.22 2.96 -2.28 3.24 4.16 4.27 4.39 3.90 4.34 3.61 -1.98
32		-0.45 1.44 0.86 0.14 2.62 0.12 310 3.40 2.99 5.05 4.96 4.71 4.31 2.41 4.31 3.81 3.45 4.10 3.71 4.00 3.87 3.41
33		-0.80 1.15 1.01 1.33 1.82 0.65 379 4.71 4.89 4.74 3.79 -1.91 3.91 4.40 4.44 4.01 3.99 -3.96 3.38 5.17 3.71 4.85
34		-0.53 1.82 0.39 0.50 2.15 0.72 451 3.95 4.95 3.72 4.34 4.24 4.51 4.76 4.85 3.83 3.77 4.16 4.10 2.11 4.22 5.00
35		-1.43 1.41 1.06 0.58 1.17 1.28 467 3.31 3.70 3.24 2.79 -2.54 -0.24 3.24 3.51 3.36 3.27 3.59 3.13 2.77 3.16 3.44
36		-0.73 0.93 1.62 1.17 -0.05 1.94 508 4.81 4.92 3.96 2.52 4.84 2.29 4.34 3.93 4.63 4.51 4.61 3.74 4.98 3.69 4.24
37		1.43 -0.31 0.50 2.61 0.38 0.09 546 4.06 4.94 2.15 2.98 4.84 2.62 5.03 5.08 5.72 2.62 1.91 3.65 3.50 3.22 5.08
38		-2.14 2.05 0.88 1.60 2.09 0.14 563 0.96 5.02 3.21 4.95 3.81 5.02 5.02 -0.47 1.99 4.93 4.95 3.42 3.04 3.57 5.11
39		-0.66 0.84 0.99 1.77 1.26 0.42 579 4.39 4.51 3.48 3.58 4.63 3.95 4.25 3.90 4.14 3.69 3.65 3.25 4.27 3.27 4.52
40		0.00 1.62 1.28 0.07 -0.43 2.48 623 2.63 5.11 3.02 3.25 -1.11 2.74 4.80 1.67 5.01 5.01 4.94 5.18 1.97 4.67 4.30
41		
42		
43		0.32

Here is the same thing again, the same construct, same shape of everything. But this time you've got an extra row. So now this is not just the matrix product of that and that, but I'm also adding on this number and this number. Which means, now each movie can have an overall "this is a great movie" versus "this isn't a great movie" and every user can have an overall "this user rates movies highly" or "this user doesn't rate movies highly" - that's called the bias. So this is hopefully going to look very familiar. This is the same usual linear model concept or linear layer concept from a neural net that you have a matrix product and a bias.

And remember from lesson 2 SGD notebook, you never actually need a bias. You could always just add a column of ones to your input data and then that gives you bias for free, but that's pretty inefficient. So in practice, all neural networks library explicitly have a concept of bias. We don't actually add the column of ones.

So what does that do? Well just before I came in today, I ran data solver on this as well, and we can check the RMSE. So the root mean squared here is 0.32 versus the version without bias was 0.39. So you can see that this slightly better model gives us a better result. And it's better because it's giving both more flexibility and it also just makes sense semantically that you need to be able to say whether I'd like the movie is not just about the combination of what actors it has, whether it's dialogue-driven, and how much action is in it but just is it a good movie or am I somebody who rates movies highly.

So there's all the pieces of this collaborative filtering model.

Question: When we load a pre-trained model, can we explore the activation grids to see what they might be good at recognizing? [36:11]

Yes, you can. And we will learn how to (should be) in the next lesson.

Question: Can we have an explanation of what the first argument in `fit_one_cycle` actually represents? Is it equivalent to an epoch?

Yes, the first argument to `fit_one_cycle` or `fit` is number of epochs. In other words, an epoch is looking at every input once. If you do 10 epochs, you're looking at every input ten times. So there's a chance you might start overfitting if you've got lots of lots of parameters and a high learning rate. If you only do one epoch, it's impossible to overfit, and so that's why it's kind of useful to remember how many epochs you're doing.

Question: What is an affine function?

An affine function is a linear function. I don't know if we need much more detail than that. If you're multiplying things together and adding them up, it's an affine function. I'm not going to bother with the exact mathematical definition, partly because I'm a terrible mathematician and partly because it doesn't matter. But if you just remember that you're multiplying things together and then adding them up, that's the most important thing. It's linear. And therefore if you put an affine function on top of an affine function, that's just another affine function. You haven't won anything at all. That's a total waste of time. So you need to sandwich it with any kind of non-linearity pretty much works - including replacing the negatives with zeros which we call ReLU. So if you do affine, ReLU, affine, ReLU, affine, ReLU, you have a deep neural network.

[38:25]

Let's go back to the [collaborative filtering notebook](#). And this time we're going to grab the whole MovieLens 100k dataset. There's also a 20 million dataset, by the way, so really a great project available made by this group called GroupLens. They actually update the MovieLens datasets on a regular basis, but they helpfully provide the original one. We're going to use the original one because that means that we can compare to baselines. Because everybody basically they say, hey if you're going to compare the baselines make sure you all use the same dataset, here's the one you should use. Unfortunately it means that we're going to be restricted to movies that are before 1998. So maybe you won't have seen them all. but that's the price we pay. You can replace this with `ml-latest` when you download it and use it if you want to play around with movies that are up to date:

```
path=Path('data/ml-100k/')
```

The original MovieLens dataset, the more recent ones are in a CSV file it's super convenient to use. The original one is a slightly messy. First of all they don't use commas for delimiters, they use tabs. So in Pandas you can just say what's the delimiter and you loaded in. The second is they don't add a header row so that you know what column is what, so you have to tell Pandas there's no header row. Then since there's no header row, you have to tell Pandas what are the names of four columns. Rather than that, that's all we need.

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

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

So we can then have a look at `head` which remember is the first few rows and there is our ratings; user, movie, rating.

Let's make it more fun. Let's see what the movies actually are.

```
movies = pd.read_csv(path/'u.item', delimiter='|', encoding='latin-1', header=None,
                      names=[item, 'title', 'date', 'N', 'url', *[f'g{i}' for i in range(16)])
movies.head()
```

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

5 rows × 24 columns

I'll just point something out here, which is there's this thing called `encoding=`. I'm going to get rid of it and I get this error:

```
-----
UnicodeDecodeError                                Traceback (most recent call last)
pandas/_libs/parsers.pyx  in pandas._libs.parsers.TextReader._convert_tokens()
pandas/_libs/parsers.pyx  in pandas._libs.parsers.TextReader._convert_with_dtype()
pandas/_libs/parsers.pyx  in pandas._libs.parsers.TextReader._string_convert()
pandas/_libs/parsers.pyx  in pandas._libs.parsers._string_box_utf8()

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 3: invalid c

During handling of the above exception, another exception occurred:
UnicodeDecodeError                                Traceback (most recent call last)
<ipython-input-15-d6ba3ac593ed>  in <module>
    1 movies = pd.read_csv(path/'u.item', delimiter='|', header=None,
--> 2                               names=[item, 'title', 'date', 'N', 'url', *[f'g{i}' f
    3 movies.head()

~/src/miniconda3/envs/fastai/lib/python3.6/site-packages/pandas/io/parsers.py  in
    676                               skip_blank_lines=skip_blank_lines)
    677
--> 678         return _read(filepath_or_buffer, kwds)
    679
    680     parser_f.__name__ = name

~/src/miniconda3/envs/fastai/lib/python3.6/site-packages/pandas/io/parsers.py  in .
    444
    445     try:
--> 446         data = parser.read(nrows)
    447     finally:
    448         parser.close()

~/src/miniconda3/envs/fastai/lib/python3.6/site-packages/pandas/io/parsers.py  in
    1034             raise ValueError('skipfooter not supported for iteration')
    1035
-> 1036         ret = self._engine.read(nrows)
    1037
    1038         # May alter columns / col_dict

~/src/miniconda3/envs/fastai/lib/python3.6/site-packages/pandas/io/parsers.py  in
    1846     def read(self, nrows=None):
    1847         try:
--> 1848             data = self._reader.read(nrows)
    1849         except StopIteration:
    1850             if self._first_chunk:

pandas/_libs/parsers.pyx  in pandas._libs.parsers.TextReader.read()
pandas/_libs/parsers.pyx  in pandas._libs.parsers.TextReader._read_low_memory()
pandas/_libs/parsers.pyx  in pandas._libs.parsers.TextReader._read_rows()
pandas/_libs/parsers.pyx  in pandas._libs.parsers.TextReader._convert_column_data()
```

```
pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._convert_tokens()
pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._convert_with_dtype()
pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._string_convert()
pandas/_libs/parsers.pyx in pandas._libs.parsers._string_box_utf8()
```

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 3: invalid c
```

I just want to point this out because you'll all see this at some point in your lives. codec can't decode blah blah blah . What this means is that this is not a Unicode file. This will be quite common when you're using datasets are a little bit older. Back before us English people in the West really realized that there are people that use languages other than English. Nowadays we're much better at handling different languages. We use this standard called Unicode for it. Python very helpfully uses Unicode by default. So if you try to load an old file that's not Unicode, you actually (believe it or not) have to guess how it was coded. But since it's really likely that it was created by some Western European or American person, they almost certainly used Latin-1. So if you just pop in encoding='latin-1' if you use file open in Python or Pandas open or whatever, that will generally get around your problem.

Again, they didn't have the names so we had to list what the names are. This is kind of interesting. They had a separate column for every one of however many genres they had ...19 genres. And you'll see this looks one hot encoded, but it's actually not. It's actually N hot encoded. In other words, a movie can be in multiple genres. We're not going to look at genres today, but it's just interesting to point out that this is a way that sometimes people will represent something like genre. The more recent version, they actually listed the genres directly which is much more convenient.

```
len(ratings)
```

```
100000
```

[42:07]

We got a hundred thousand ratings. I find life is easier when you're modeling when you actually denormalize the data. I actually want the movie title directly in my ratings, so Pandas has a merge function to let us do that. Here's the ratings table with actual titles.

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

	userId	movieId	rating	timestamp	title
0	196	242	3	881250949	Kolya (1996)

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

As per usual, we can create a data bunch for our application, so a CollabDataBunch for the collab application. From what? From a data frame. There's our data frame. Set aside some validation data. Really we should use the validation sets and cross validation approach that they used if you're going to properly compare with a benchmark. So take these comparisons with a grain of salt.

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

data.show_batch()
```

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

By default, CollabDataBunch assumes that your first column is user, second column is item, the third column is rating. But now we're actually going to use the title column as item, so we have to tell it what the item column name is (`item_name=title`). Then all of our data bunches support show batch, so you can just check what's in there, and there it is.

↪ Jeremy's tricks for getting better results [43:18]

I'm going to try and get as good a result as I can, so I'm gonna try and use whatever tricks I can come up with to get a good answer. Now one of the tricks is to use the Y range. Remember the Y range was the thing that made the final activation function a sigmoid. And specifically, last week we said "let's have a sigmoid that goes from naught to 5" and that way it's going to ensure that it is going to help the neural network predict things that are in the range.

Actually I didn't do that in my Excel version and so you can see I've actually got some negatives and there's also some things bigger than five. So if you want to beat me in Excel, you could add the sigmoid to excel and train this, and you'll get a slightly better answer.

Now the problem is that a sigmoid actually asymptotes at whatever the maximum is (we said 5) which means you can never actually predict 5. But plenty of movies have a rating of 5, so that's a problem. So actually **it's slightly better to make your Y range go from a little bit less than the minimum to a little bit more than the maximum**. The minimum of this data is 0.5 and the maximum is 5, so this range is just a little bit further. So that's a that's one little trick to get a little bit more accuracy.

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

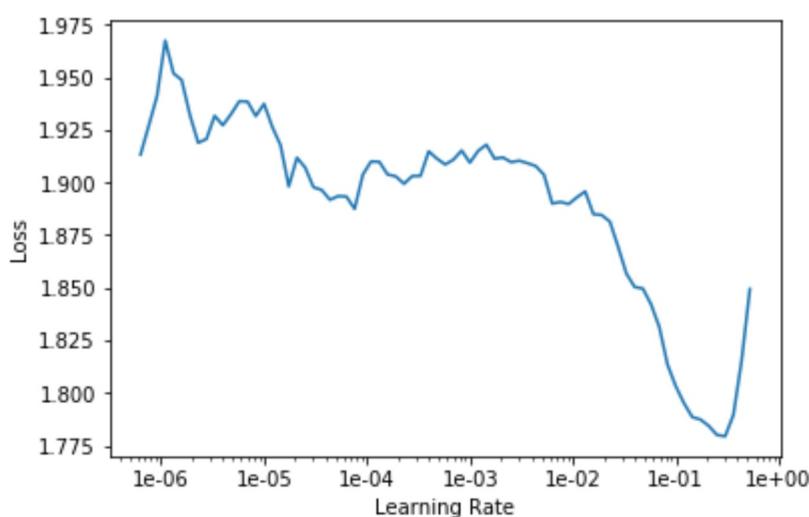
The other trick I used is to add something called weight decay, and we're going to look at that next . After this section, we are going to learn about weight decay.

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

How many factors do you want or what are factors? The number of factors is the width of the embedding matrix. So why don't we say embedding size? Maybe we should, but in the world of collaborative filtering they don't use that word. They use the word "factors" because of this idea of latent factors, and because the standard way of doing collaborative filtering has been with something called **matrix factorization**. In fact what we just saw happens to actually be a way of doing matrix factorization. So we've actually accidentally learned how to do matrix factorization today. So this is a term that's kind of specific to this domain. But you can just remember it as the width of the embedding matrix.

Why 40? Well this is one of these architectural decisions you have to play around with and see what works. So I tried 10, 20, 40, and 80 and I found 40 seems to work pretty well. And it trained really quickly, so you can chuck it in a little for loop to try a few things and see what looks best.

```
learn.lr_find()  
learn.recorder.plot(skip_end=15)
```



Then for learning rates, here's the learning rate finder as usual. `5e-3` seemed to work pretty well. Remember this is just a rule of thumb. `5e-3` is a bit lower than both Sylvain's rule and my rule - so Sylvain's rule is find the bottom and go back by ten, so his rule would be more like `2e-2`, I reckon. My rule is kind of find about the steepest section which is about here, which again often it agrees with Sylvain's so that would be about `2e-2`. I tried that and I always like to try like 10 X less and 10x more just to check. And actually I found a bit less was helpful. So the answer to the question like "should I do blah?" is always "try blah and see." Now that's how you actually become a good practitioner.

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

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

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

So that gave me 0.813. And as usual, you can save the result to save you another 33 seconds from having to do it again later.

There's a library called LibRec and they published [some benchmarks for MovieLens 100k](#) and there's a root mean squared error section, and about 0.91 is about as good as they seem to have been able to get. 0.91 is the root mean square error. We use the mean square error, not the root, so we have to go to point 0.91^2 which is 0.83 and we're getting 0.81, so that's cool. With this very simple model, we're doing a little bit better, quite a lot better actually. Although as I said, take it with a grain of salt because we're not doing the same splits and the same cross validation. So we're at least highly competitive with their approaches.

We're going to look at the Python code that does this in a moment, but for now just take my word for it that we're going to see something that's just doing this:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
1																						
2	Users	Embeddings																				
3	idx	Row Lab																				
4	1	14	0.21	1.61	2.89	-1.26	0.82	14	417	2	1	0.21	1.61	2.89	-1.26	0.82	14	1.66	0.66	0.57	0.46	0.47
5	2	29	1.55	0.75	0.22	1.62	1.26	29	417	4	2	1.55	0.75	0.22	1.62	1.26	14	1.66	0.66	0.57	0.46	0.47
6	3	72	1.50	1.17	0.22	1.08	1.49	72	417	5	3	1.50	1.17	0.22	1.08	1.49	14	1.66	0.66	0.57	0.46	0.47
7	4	211	0.47	0.89	1.32	1.13	0.77	211	417	3	4	0.47	0.89	1.32	1.13	0.77	14	1.66	0.66	0.57	0.46	0.47
8	5	212	0.31	2.10	1.47	-0.29	-0.15	212	417	3	5	0.31	2.10	1.47	-0.29	-0.15	14	1.66	0.66	0.57	0.46	0.47
9	6	293	1.00	1.45	0.37	0.83	0.67	293	417	4	6	1.00	1.45	0.37	0.83	0.67	14	1.66	0.66	0.57	0.46	0.47
10	7	310	1.16	1.16	0.19	2.16	-0.03	310	417	3	7	1.16	1.16	0.19	2.16	-0.03	14	1.66	0.66	0.57	0.46	0.47
11	8	379	0.79	1.07	1.30	1.29	0.70	379	417	4	8	0.79	1.07	1.30	1.29	0.70	14	1.66	0.66	0.57	0.46	0.47
12	9	451	1.52	0.54	0.64	1.36	0.94	451	417	3.5	9	1.52	0.54	0.64	1.36	0.94	14	1.66	0.66	0.57	0.46	0.47
13	10	467	1.00	0.69	0.41	0.75	1.02	467	417	4	10	1.00	0.69	0.41	0.75	1.02	14	1.66	0.66	0.57	0.46	0.47
14	11	508	0.86	1.29	0.80	0.19	1.79	508	417	3	11	0.86	1.29	0.80	0.19	1.79	14	1.66	0.66	0.57	0.46	0.47
15	12	546	0.61	-0.09	2.40	1.57	-0.18	546	417	3.5	12	0.61	-0.09	2.40	1.57	-0.18	14	1.66	0.66	0.57	0.46	0.47
16	13	563	1.45	0.59	1.40	1.29	-0.13	563	417	4	13	1.45	0.59	1.40	1.29	-0.13	14	1.66	0.66	0.57	0.46	0.47
17	14	579	0.68	0.95	1.53	0.84	0.64	579	417	4	14	0.68	0.95	1.53	0.84	0.64	14	1.66	0.66	0.57	0.46	0.47
18	15	623	1.70	1.00	0.20	-0.25	2.05	623	417	5	15	1.70	1.00	0.20	-0.25	2.05	14	1.66	0.66	0.57	0.46	0.47
19								14	27	3	1	0.21	1.61	2.89	-1.26	0.82	1	-1.69	1.01	0.82	1.89	2.39
20		Movies						29	27	5	2	1.55	0.75	0.22	1.62	1.26	1	-1.69	1.01	0.82	1.89	2.39
21	idx	Row Lab						72	27	4	3	1.50	1.17	0.22	1.08	1.49	1	-1.69	1.01	0.82	1.89	2.39
22	1	27	-1.69	1.01	0.82	1.89	2.39	211	27	5	4	0.47	0.89	1.32	1.13	0.77	1	-1.69	1.01	0.82	1.89	2.39
23	2	49	1.49	0.12	1.48	0.50	1.13	212	27	2.5	5	0.31	2.10	1.47	-0.29	-0.15	1	-1.69	1.01	0.82	1.89	2.39
								293	27	3	6	1.00	1.45	0.37	0.83	0.67	1	-1.69	1.01	0.82	1.89	2.39

Looking things up in an array, and then multiplying them together, adding them up, and doing the mean square error loss function. Given that and given that we noticed that the only way that can do anything interesting is by trying to find these latent factors. It makes sense to look and see what they found. Particularly since as well as finding latent factors, we also now have a specific bias number for every user and every movie.

Now, you could just say what's the average rating for each movie. But there's a few issues with that. In particular, this is something you see a lot with like anime. People who like anime just love anime, and so they're watching lots of anime and then they just rate all the anime highly. So very often on kind of charts of movies, you'll see a lot of anime at the top. Particularly if it's like a hundred long series of anime, you'll find every single item of that series in the top thousand movie list or something.

⌚ Interpreting bias [49:29]

So how do we deal with that? Well the nice thing is that instead if we look at the movie bias, once we've included the user bias (which for an anime lover might be a very high number because they're just rating a lot of movies highly) and once we account for the specifics of this kind of movie (which again might be people love anime), what's left over is something specific to that movie itself. So it's kind of interesting to look at movie bias numbers as a way of saying what are the best movies or what do people really like as movies even if those people don't rate movies very highly or even if that movie doesn't have the kind of features that people tend to rate highly. So it's kind of nice, it's funny to say this 😊, by using the bias, we get an unbiased movie score.

How do we do that? To make it interesting particularly because this dataset only goes to 1998, let's only look at movies that are plenty of people watch. So we'll use Pandas to grab our `rating_movie` table, group it by title, and then count the number of ratings. Not measuring how high their rating, just how many ratings do they have.

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

learn.model

EmbeddingDotBias(
    (u_weight): Embedding(944, 40)
    (i_weight): Embedding(1654, 40)
    (u_bias): Embedding(944, 1)
    (i_bias): Embedding(1654, 1)
)

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

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

So the top thousand are the movies that have been rated the most, and so there hopefully movies that we might have seen. That's the only reason I'm doing this. So I've called this `top_movies` by which I mean not good movies, just movies we likely to have seen.

Not surprisingly, Star Wars is the one, at that point, the most people had put a rating to. Independence Day, there you go. We can then take our learner that we trained and asked it for the bias of the items listed here.

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

```
torch.Size([1000])
```

So `is_item=True`, you would pass `True` to say I want the items or `False` to say I want the users. So this is kind of like a pretty common piece of nomenclature for collaborative filtering - these IDs (users) tend to be called users, these IDs (movies) tend to be called items, even if your problem has got nothing to do with users and items at all. We just use these names for convenience. So they're just words. In our case, we want the items. This (`top_movies`) is the list of items we want, we want the bias. So this is specific to collaborative filtering.

And so that's going to give us back a thousand numbers back because we asked for this has a thousand movies in it. Just for comparison, let's also group the titles by the mean rating. Then we can zip through (i.e. going through together) each of the movies along with the bias and grab their rating, the bias, and the movie. Then we can sort them all by the zero index thing which is the bias.

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

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

Here are the lowest numbers:

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

```
[(tensor(-0.3264),
  'Children of the Corn: The Gathering (1996)',
  1.3157894736842106),
 (tensor(-0.3241),
  'Lawnmower Man 2: Beyond Cyberspace (1996)',
  1.7142857142857142),
 (tensor(-0.2799), 'Island of Dr. Moreau, The (1996)', 2.1578947368421053),
 (tensor(-0.2761), 'Mortal Kombat: Annihilation (1997)', 1.9534883720930232),
 (tensor(-0.2703), 'Cable Guy, The (1996)', 2.339622641509434),
 (tensor(-0.2484), 'Leave It to Beaver (1997)', 1.8409090909090908),
```

```
(tensor(-0.2413), 'Crow: City of Angels, The (1996)', 1.9487179487179487),  
(tensor(-0.2395), 'Striptease (1996)', 2.2388059701492535),  
(tensor(-0.2389), 'Free Willy 3: The Rescue (1997)', 1.7407407407407407),  
(tensor(-0.2346), 'Barb Wire (1996)', 1.9333333333333333),  
(tensor(-0.2325), 'Grease 2 (1982)', 2.0),  
(tensor(-0.2294), 'Beverly Hills Ninja (1997)', 2.3125),  
(tensor(-0.2223), "Joe's Apartment (1996)", 2.2444444444444445),  
(tensor(-0.2218), 'Bio-Dome (1996)', 1.903225806451613),  
(tensor(-0.2117), "Stephen King's The Langoliers (1995)", 2.413793103448276)]
```

I can say you know Mortal Kombat Annihilation, not a great movie. Lawnmower Man 2, not a great movie. I haven't seen Children of the Corn, but we did have a long discussion at SF study group today and people who have seen it agree, not a great movie. And you can kind of see like some of them actually have pretty decent ratings. So this one's actually got a much higher rating (Island of Dr. Moreau, The (1996)) than the next one. But that's kind of saying well the kind of actors that were in this, the kind of movie that this was, and the kind of people who watch it, you would expect it to be higher.

Then here's the sort by reverse:

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

```
[(tensor(0.6105), "Schindler's List (1993)", 4.466442953020135),  
(tensor(0.5817), 'Titanic (1997)', 4.2457142857142856),  
(tensor(0.5685), 'Shawshank Redemption, The (1994)', 4.445229681978798),  
(tensor(0.5451), 'L.A. Confidential (1997)', 4.161616161616162),  
(tensor(0.5350), 'Rear Window (1954)', 4.3875598086124405),  
(tensor(0.5341), 'Silence of the Lambs, The (1991)', 4.28974358974359),  
(tensor(0.5330), 'Star Wars (1977)', 4.3584905660377355),  
(tensor(0.5227), 'Good Will Hunting (1997)', 4.262626262626263),  
(tensor(0.5114), 'As Good As It Gets (1997)', 4.196428571428571),  
(tensor(0.4800), 'Casablanca (1942)', 4.45679012345679),  
(tensor(0.4698), 'Boot, Das (1981)', 4.203980099502488),  
(tensor(0.4589), 'Close Shave, A (1995)', 4.491071428571429),  
(tensor(0.4567), 'Apt Pupil (1998)', 4.1),  
(tensor(0.4566), 'Vertigo (1958)', 4.251396648044692),  
(tensor(0.4542), 'Godfather, The (1972)', 4.283292978208232)]
```

Schindler's List, Titanic, Shawshank Redemption - seems reasonable. Again you can kind of look for ones where the rating isn't that high but it's still very high here. So that's kind of like at least in 1998, people weren't that into Leonardo DiCaprio, people aren't that into dialogue-driven movies, or people aren't that into romances or whatever. But still people liked it more than you would have expected. It's interesting to interpret our models in this way.

⌚ Interpreting Weights [54:27]

We can go a bit further and grab not just the biases but the weights.

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

```
torch.Size([1000, 40])
```

```
movie_pca = movie_w.pca(3)  
movie_pca.shape
```

```
torch.Size([1000, 3])
```

Again we're going to grab the weights for the items for our top movies. That is a thousand by forty because we asked for 40 factors, so rather than having a width of 5, we have a width of 40.

Often, really, there isn't really conceptually 40 latent factors involved in taste, and so trying to look at the 40 can be not that intuitive. So what we want to do is, we want to squish those 40 down to just 3. And there's something that we're not going to look into called PCA stands for Principal Components Analysis. This `movie_w` is a torch tensor and fastai adds the PCA method to torch tensors. What Principal Components Analysis does is it's a simple linear transformation that takes an input matrix and tries to find a smaller number of columns that cover a lot of the space of that original matrix. If that sounds interesting, which it totally is, you should check out our course, computational linear algebra, which Rachel teaches where we will show you how to calculate PCA from scratch and why you'd want to do it and lots of stuff like that. It's absolutely not a prerequisite for anything in this course, but it's definitely worth knowing that taking layers of neural nets and chucking them through PCA is very often a good idea. Because very often you have way more activations than you want in a layer, and there's all kinds of reasons you would might want to play with it. For example, Francisco who's sitting next to me today has been working on something to do with image similarity. And for image similarity, a nice way to do that is to compare activations from a model, but often those activations will be huge and therefore your thing could be really slow and unwieldy. So people often, for something like image similarity, will chuck it through a PCA first and that's kind of cool. In our case, we're just going to do it so that we take our 40 components down to 3 components, so hopefully they'll be easier for us to interpret.

```
fac0,fac1,fac2 = movie_pca.t()
```

```
movie_comp = [(f, i) for f,i in zip(fac0, top_movies)]
```

We can grab each of those three factors will call them `fac0`, `fac1`, and `fac2`. Let's grab that movie components and then sort. Now the thing is, we have no idea what this is going to mean. But we're pretty sure it's going to be some aspect of taste and movie feature. So if we print it out the top and the bottom, we can see that the highest ranked things on this feature, you would kind of describe them as I guess "connoisseur movies".

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

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

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

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

Chinatown - really classic Jack Nicholson movie. Everybody knows Casablanca, and even like Wrong Trousers is like this classic claymation movie and so forth. So yeah, this is definitely measuring like things that are very high on the connoisseur level. Where else, maybe Home Alone 3, not such a favorite with connoisseurs, perhaps. It's just not to say that there aren't people who don't like it, but probably not the same kind of people that would appreciate Secrets & Lies. So you can kind of see this idea that this has found some feature of movies and a corresponding feature of the kind of things people like.

Let's look at another feature.

```

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

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

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

```

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

```

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

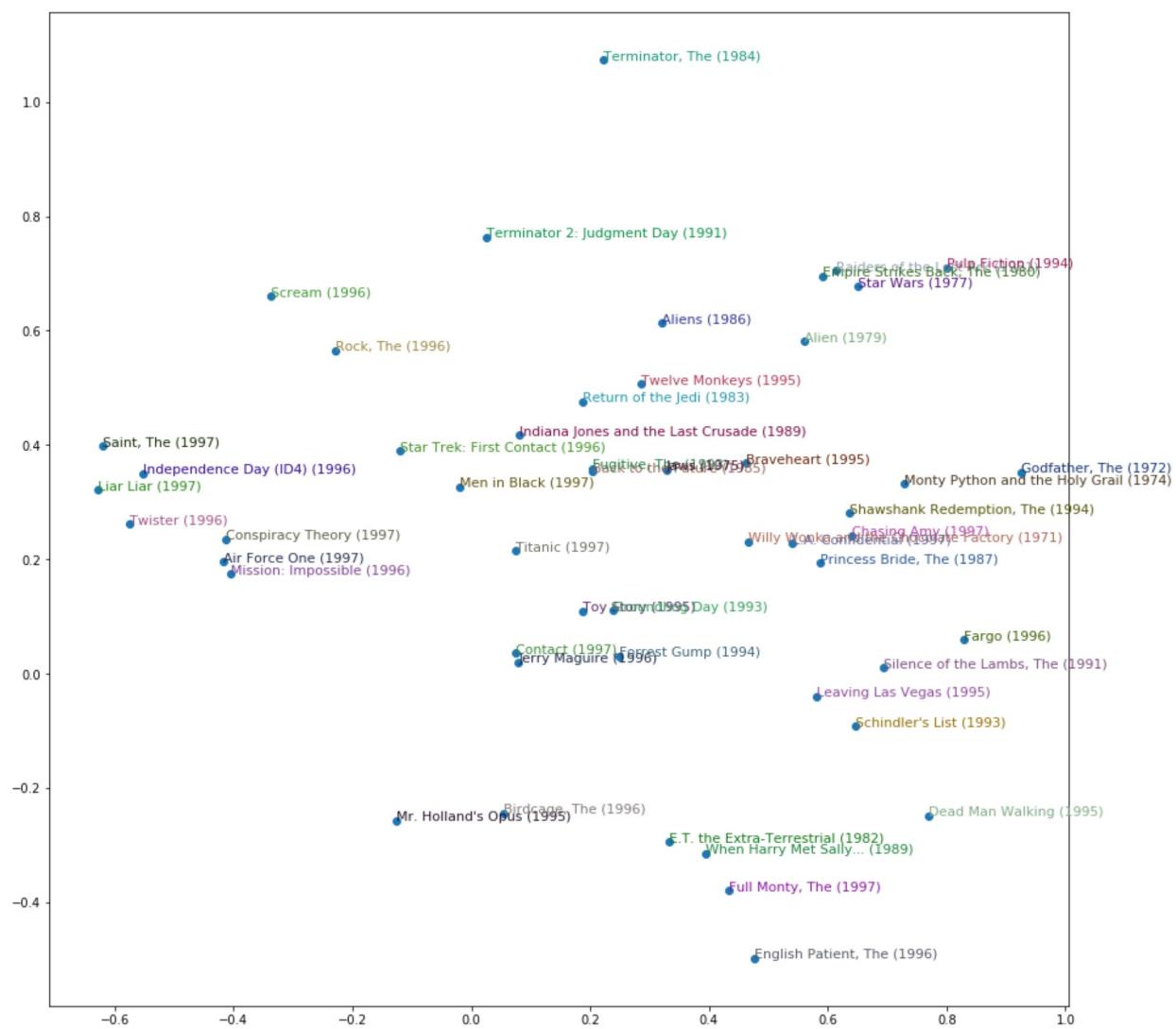
```

Here's factor number one. This seems to have found... okay these are just big hits that you could watch with the family (the latter). These are definitely not that - Trainspotting very gritty thing. So again, it's kind of found this interesting feature of taste. And we could even like draw them on a graph.

```

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

```



I've just cuddled them randomly to make them easier to see. This is just the top 50 most popular movies by how many times they've been rated. On this one factor, you've got The Terminators really high up here, and The English Patient and Schindler's List at the other end. Then The Godfather and Monty Python over here (on the right), and Independence Day and Liar Liar over there (on the left). So you get the idea. It's kind of fun. It would be interesting to see if you can come up with some stuff at work or other kind of datasets where you could try to pull out some features and play with them.

Question: Why am I sometimes getting negative loss when training? [59:49]

You shouldn't be. So you're doing something wrong. Particularly since people are uploading this, I guess other people have seen it too, so put it on the forum. We're going to be learning about cross entropy and negative log likelihood after the break today. They are loss functions that have very specific expectations about what your input looks like. And if your input doesn't look like that, then they're going to give very weird answers, so probably you press the wrong buttons. So don't do that.

↪ **collab_learner** [1:00:43]

```

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

```

Here is the `collab_learner` function. The `collab_learner` function as per usual takes a data bunch. And normally learners also take something where you ask for particular architectural details. In this case, there's only one thing which does that which is basically do you want to use a multi-layer neural net or do you want to use a classic collaborative filtering. We're only going to look at the classic collaborative filtering today, or maybe we'll briefly look at the other one too, we'll see.

So what actually happens here? Well basically we create an `EmbeddingDotBias` model, and then we pass back a learner which has our data and that model. So obviously all the interesting stuff is happening here in `EmbeddingDotBias`, so let's take a look at that.

```

class EmbeddingDotBias(nn.Module):
    "Base dot model for collaborative filtering."
    def __init__(self, n_factors:int, n_users:int, n_items:int, y_range:Tuple[float,float]=None):
        super().__init__()
        self.y_range = y_range
        (self.u_weight, self.i_weight, self.u_bias, self.i_bias) = [embedding(*o) for o in [
            (n_users, n_factors), (n_items, n_factors), (n_users, 1), (n_items, 1)
        ]]
    def forward(self, users:LongTensor, items:LongTensor) -> Tensor:
        dot = self.u_weight(users)* self.i_weight(items)
        res = dot.sum(1) + self.u_bias(users).squeeze() + self.i_bias(items).squeeze()
        if self.y_range is None: return res
        return torch.sigmoid(res) * (self.y_range[1]-self.y_range[0]) + self.y_range[0]

```

Here's our `EmbeddingDotBias` model. It is a `nn.Module`, so in PyTorch, to remind you, all PyTorch layers and models are `nn.Module`'s. They are things that, once you create them, look exactly like a function. You call them with parentheses and you pass them arguments. But they're not functions. They don't even have `__call__`. Normally in Python, to make something look like a function, you have to give it a method called dunder call. Remember that means `__call__`, which doesn't exist here. The reason is that PyTorch actually expects you to have something called `forward` and that's what PyTorch will call for you when you call it like a function.

So when this model is being trained, to get the predictions it's actually going to call forward for us. So this (`forward`) is where we calculate our predictions. So this is where you can see, we grab our... Why is this `users` rather than `user`? That's because everything's done a mini-batch at a time. When I read the `forward` in a PyTorch module, I tend to ignore in my head the fact that there's a mini batch. And I pretend there's just one. Because PyTorch automatically handles all of the stuff about doing it to everything in the mini batch for you. So let's pretend there's just one user. So grab that user and what is this `self.u_weight`? `self.u_weight` is an embedding. We create an embedding for each of users by factors, items by factors, users by one, items by one. That makes sense, right? So users by one is the user's bias. Then users by factors is feature/embedding. So users by factors is the first tuple, so that's going to go in `u_weight` and `(n_users,1)` is the third, so that's going to go in `u_bias`.

Remember, when PyTorch creates our `nn.Module`, it calls dunder init. So this is where we have to create our weight matrices. We don't normally create the actual weight matrix tensors. We normally use PyTorch's convenience functions to do that for us, and we're going to see some of that after the break. For now, just recognize that this function is going to create an embedding matrix for us. It's going to be a PyTorch `nn.Module` as well, so therefore to actually pass stuff into that embedding matrix and get activations out, you treat it as if it was a function - stick it in parentheses. So if you want to look in the PyTorch source code and find `nn.Embedding`, you will find there's something called `.forward` in there which will do this array lookup for us.

[1:05:29]

Here's where we grab the users (`self.u_weight(users)`), here's where we grab the items (`self.i_weight(items)`). So we've now got the embeddings for each. So at this point, we multiply them together and sum them up, and then we add on the user bias and the item bias. Then if we've got a `y_range`, then we do our sigmoid trick. So the nice thing is, you now understand the entirety of this model. This is not just any model. This is a model that we just found which is at the very least highly competitive with and perhaps slightly better than some published table of pretty good numbers from a software group that does nothing but this. So you're doing well. This is nice.

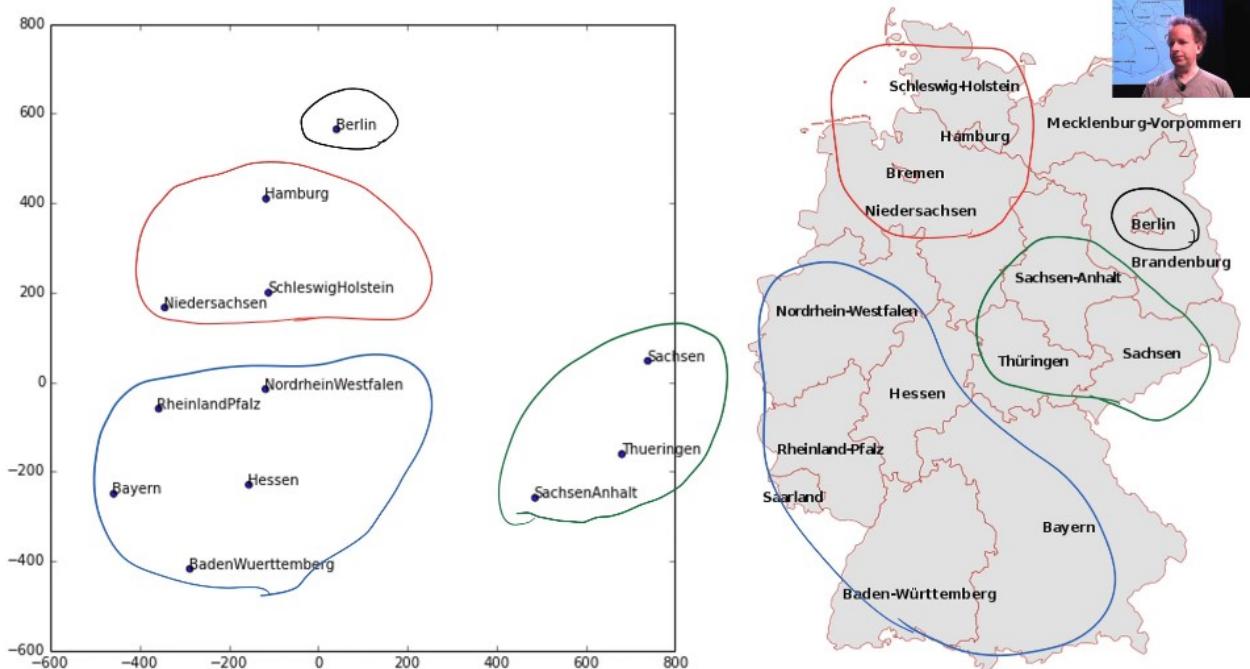
↪ **Embeddings are amazing** [1:07:03]

This idea of interpreting embeddings is really interesting. As we'll see later in this lesson, the things that we create for categorical variables more generally in tabular data sets are also embedding matrices. And again, that's just a normal matrix multiplied by a one hot encoded input where we skip the computational and memory burden of it by doing it in a more efficient way, and it happens to end up with these interesting semantics kind of accidentally. There was [this really interesting paper](#) by these folks who came second in a Kaggle competition for something called Rossman. We will probably look in more detail at the Rossman competition in part two. I think we're gonna run out of time in part one. But it's basically this pretty standard tabular stuff. The main interesting stuffs in the pre-processing. And it was interesting because they came second despite the fact that the person who came first and pretty much everybody else who was the top of the leaderboard did a massive amount of highly specific feature engineering. Where else, these folks did way less feature engineering than anybody else. But instead they used a neural net, and this was at a time in 2016 when just no one did that. No one was doing neural nets for tabular data.

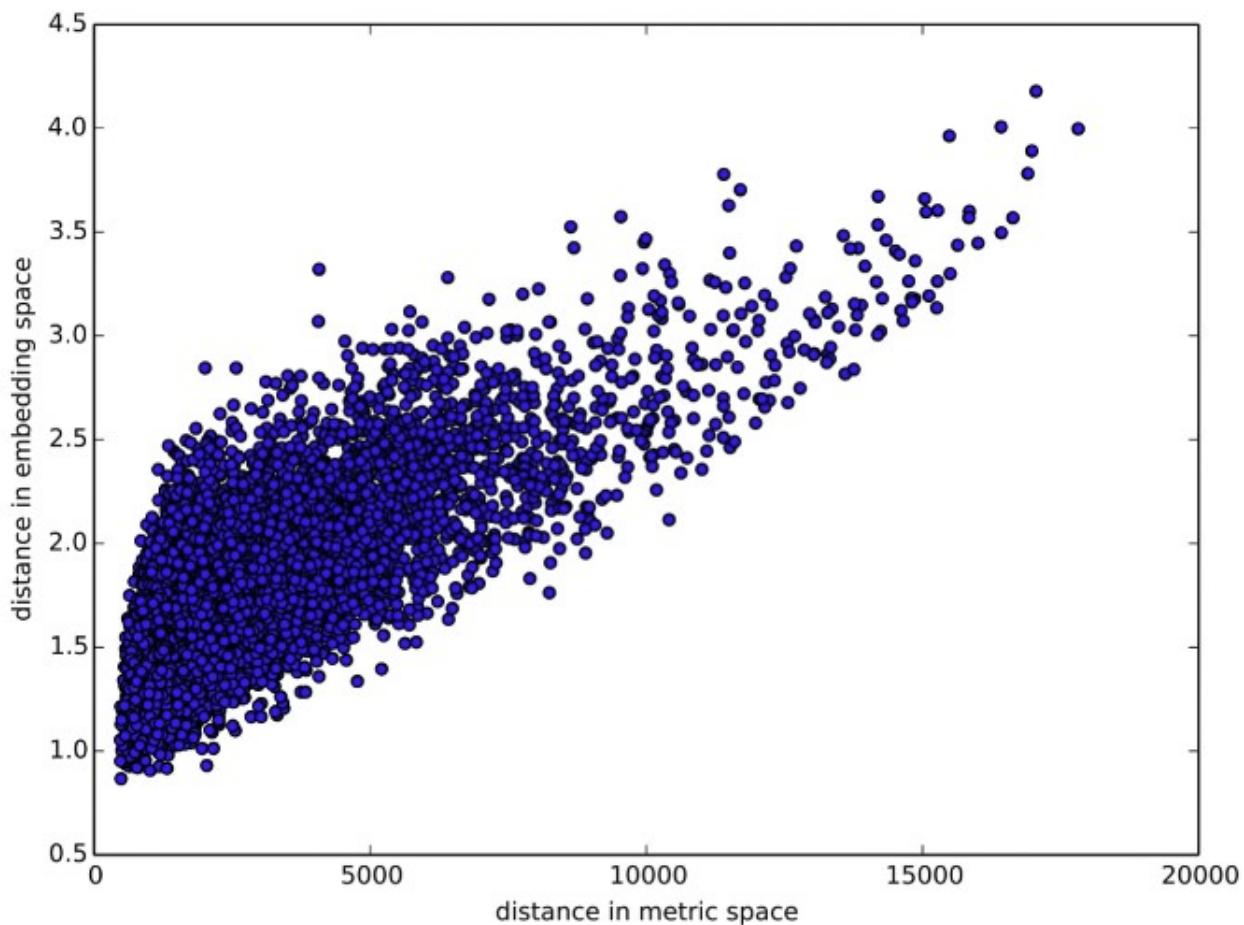
So the kind of stuff that we've been talking about kind of arose there or at least was kind of popularized there. And when I say popularized, I mean only popularized a tiny bit - still most people are unaware of this idea. But it's pretty cool because in their paper they showed that the main average percentage error for various techniques K nearest neighbors, random forests, and gradient boosted trees:

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

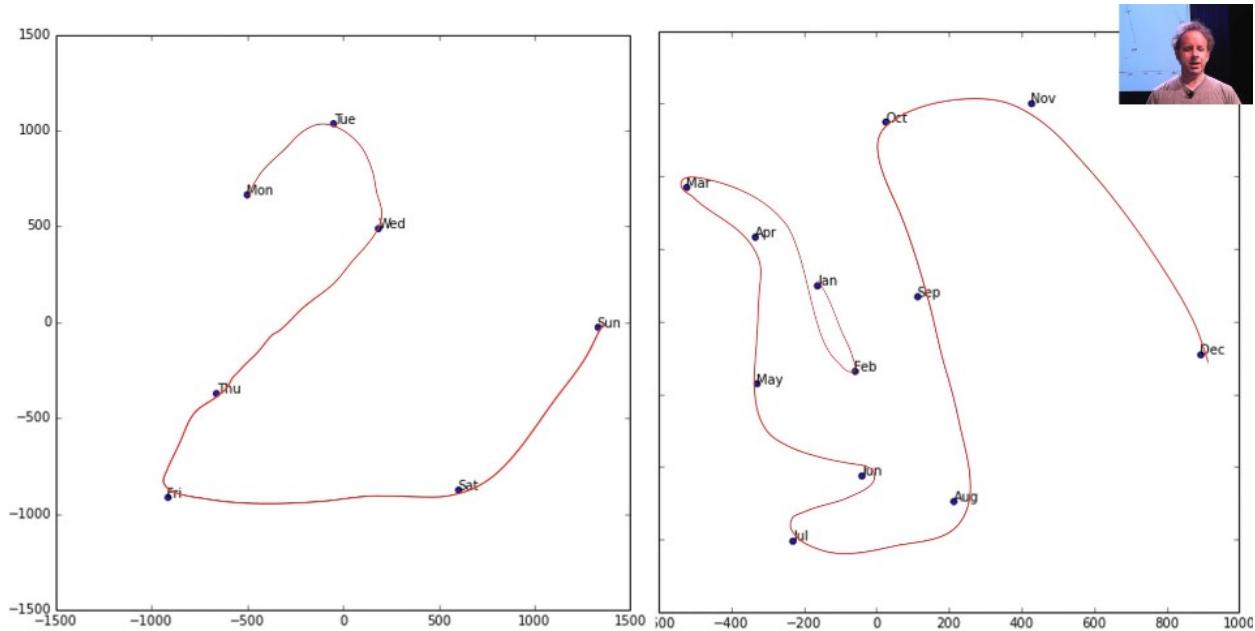
First, you know, neural nets just worked a lot better but then with entity embeddings (which is what they call this using entity matrices in tabular data), they actually added the entity embeddings to all of these different tasks after training them and they all got way better. So neural nets with entity embeddings are still the best but a random forest with empty embeddings was not at all far behind. That's kind of nice because you could train these entity matrices for products or stores or genome motifs or whatever and then use them in lots of different models, possibly using faster things like random forests but getting a lot of the benefits.



But here is something interesting. They took a two-dimensional projection of their embedding matrix for German state because this was a German supermarket chain using the same kind of approach we did - I don't remember if they use PCA or something else slightly different. And then here's the interesting thing. I've circled here a few things in this embedding space, and I've circled it with the same color over here and it's like "oh my god, the embedding projection has actually discovered geography." They didn't do that but it's found things that are near by each other in grocery purchasing patterns because this was about predicting how many sales there will be. There is some Geographic element of that.



In fact, here is a graph of the distance between two embedding vectors. So you can just take an embedding vector and say what's the sum of squared compared to some other embedding vector. That's the Euclidean distance (i.e. what's the distance in embedding space) and then plotted against the distance in real life between shops, and you get this very strong positive correlation.



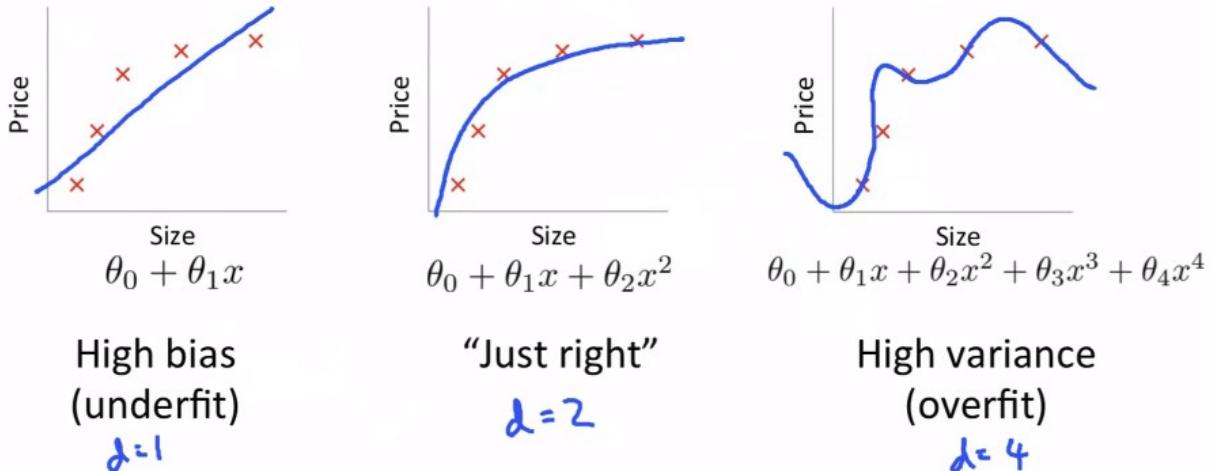
Here is an embedding space for the days of the week, and as you can see there's a very clear path through them. Here's the embedding space for the month of the year, and again there's a very clear path through them.

Embeddings are amazing, and I don't feel like anybody's even close to exploring the kind of interpretation that you could get. So if you've got genome motifs or plant species or products that your shop sells or whatever, it would be really interesting to train a few models, try and fine tune some embeddings, and then start looking at them in these ways in terms of similarity to other ones and clustering them and projecting them into 2d spaces and whatever. I think is really interesting.

⌚ Regularization: Weight Decay [1:12:09]

We were trying to make sure we understood what every line of code did in this some pretty good collab learner model we built. The one piece missing is this `wd` piece, and `wd` stands for weight decay. So what is weight decay? Weight decay is a type of regularization. What is regularization?

Bias/variance



Andrew Ng

Let's start by going back to this nice little chart that Andrew Ng did in his terrific machine learning course where he plotted some data and then showed a few different lines through it. This one here, because Andrew's at Stanford he has to use Greek letters. We can say this is $a + bx$ but if you want to go there $\theta_0 + \theta_1 x$ is a line. It's a line even if it's got a Greek letters. Here's a second-degree polynomial $a + bx + cx^2$ - bit of curve, and here's a high degree polynomial which is curvy as anything.

So models with more parameters tend to look more like this. In traditional statistics, we say "let's use less parameters" because we don't want it to look like this. Because if it looks like this, then the predictions far left and far right, they're going to be all wrong. It's not going to generalize well. We're overfitting. So we avoid overfitting by using less parameters. So if any of you are unlucky enough to have been brainwashed by a background in statistics or psychology or econometrics or any of these kinds of courses, you're gonna have to unlearn the idea that you need less parameters. Because what you instead need to realize is you were fed this lie that you need less parameters because it's a convenient fiction for the real truth which is you don't want your function to be too complex. Having less parameters is one way of making it less complex. But what if you had a thousand parameters and 999 of those parameters were $1e-9$? What if they were 0? If they were 0, they're not really there. Or if they were $1e-9$, they're hardly there. So why can't I have lots of parameters if lots of them are really small? And the answer is you can. So this thing of counting the number of parameters is how we limit complexity is actually extremely limiting. It's a fiction that really has a lot of problems. So if in your head complexity is scored by how many parameters you have, you're doing it all wrong. Score it properly.

So why do we care? Why would I want to use more parameters? Because more parameters means more nonlinearities, more interactions, more curvy bits. And real life is full of curvy bits. Real life does not look like a straight line. But we don't want them to be more curvy than necessary or more interacting than necessary. Therefore let's use lots of parameters and then penalize complexity. So one way to penalize complexity (as I kind of suggested before) is let's sum up the value of your parameters. Now that doesn't quite work because some parameters are positive and some are negative. So what if we sum up the square of the parameters, and that's actually a really good idea.

Let's actually create a model, and in the loss function we're going to add the sum of the square of the parameters. Now here's a problem with that though. Maybe that number is way too big, and it's so big that the best loss is to set all of the parameters to zero. That would be no good. So we want to make sure that doesn't happen, so therefore let's not just add the sum of the squares of the parameters to the model but let's multiply that by some number that we choose. That number that we choose in fastai is called `wd`. That's what we are going to do. We are going take our loss function and we're going to add to it the sum of the squares of parameters multiplied by some number `wd`.

What should that number be? Generally, it should be 0.1. People with fancy machine learning PhDs are extremely skeptical and dismissive of any claims that a learning rate can be 3e-3 most of the time or a weight decay can be 0.1 most of the time. But here's the thing - we've done a lot of experiments on a lot of datasets, and we've had a lot of trouble finding anywhere a weight decay of 0.1 isn't great. However we don't make that the default. We actually make the default 0.01. Why? Because in those rare occasions where you have too much weight decay, no matter how much you train it just never quite fits well enough. Where else if you have too little weight decay, you can still train well. You'll just start to overfit, so you just have to stop a little bit early.

So we've been a little bit conservative with our defaults, but my suggestion to you is this. Now that you know that every learner has a `wd` argument and I should mention you won't always see it in this list:

```
learn = collab_learner(data, n_factors=40, y_range=y_range, wd='1e-1')
Signature: collab_learner(data, n_factors:int=None, use_nn:bool=False, metrics=None, szs:Dict[str, int]=None, wd:float=0.01, **kwargs) -> fastai.basic_train.Learner
Docstring: Create a Learner for collaborative filtering.
File: /data1/jhoward/git/fastai/fastai/collab.py
```

Because there's this concept of `**kwargs` in Python which is basically parameters that are going to get passed up the chain to the next thing that we call. So basically all of the learners will call eventually this constructor:

```
Learner
Init signature: Learner(data:fastai.basic_data.DataBunch, model:torch.nn.modules.  
.Module, opt_func:Callable=functools.partial(<class 'torch.optim.adam.Adam'>, betas=(0.  
9, 0.99)), loss_func:Callable=None, metrics:Collection[Callable]=None, true_wd:bool=True  
e, bn_wd:bool=True, wd:Union[float, Collection[float]]=0.01, train_bn:bool=True, path:s  
tr=None, model_dir:str='models', callback_fns:Collection[Callable]=None, callbacks:Coll  
ection[fastai.callback.Callback]=<factory>, layer_groups:Collection[torch.nn.modules.mo  
dule.Module]=None) -> None
Docstring: Train `model` using `data` to minimize `loss_func` with optimizer `opt_`  
`func`.
File: /data1/jhoward/git/fastai/fastai/basic_train.py
Type: type
```

And this constructor has a `wd`. So this is just one of those things that you can either look in the docs or you now know it. Anytime you're constructing a learner from pretty much any kind of function in fastai, you can pass `wd`. And passing 0.1 instead of the default 0.01 will often help, so give it a go.

⌚ Going back to Lesson2 SGD notebook [1:19:16]

So what's really going on here? It would be helpful to go back to [lesson 2 SGD](#) because everything we're doing for the rest of today really is based on this.

We created some data, we added at a loss function MSE, and then we created a function called `update` which calculated our predictions. That's our weight make matrix multiply:

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

This is just a one layer so there's no ReLU. We calculated our loss using that mean squared error. We calculated the gradients using `loss.backward`. We then subtracted in place the learning rate times the gradients, and that is gradient descent. If you haven't reviewed lesson two SGD, please do because this is our starting point. So if you don't get this, then none of this is going to make sense. If you watching the video, maybe pause now go back, re-watch this part of lesson 2, make sure you get it.

Remember `a.sub_` is basically the same as `a -=` because `a.sub` is subtract and everything in PyTorch, if you add an underscore to it means do it in place. So this is updating our `a` parameters which started out as `[-1., 1.]` - we just arbitrary picked those numbers and it gradually makes them better.

Let's write that down so we are trying to calculate the parameters (I'm going to call them weights because this is more common) in epoch t or time t . And they're going to be equal to whatever the weights were in the previous epoch minus our learning rate multiplied by the derivative of our loss function with respect to our weights at time $t - 1$

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

That's what this is doing:

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

We don't have to calculate the derivative because it's boring and because computers do it for us fast, and then they store it in `grad` for us, so we're good to go. Make sure you're exceptionally comfortable with either that equation or that line of code because they are the same thing.

What's our loss? Our loss is some function of our independent variables X and our weights ($L(x, w)$). In our case, we're using mean squared error, for example, and it's between our predictions and our actuals.

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

Where does X and W come in? Well our predictions come from running some model (we'll call it m) on those predictions and that model contains some weights. So that's what our loss function might be:

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

And this might be all kinds of other loss functions, we will see some more today. So that's what ends up creating `a.grad` over here.

We're going to do something else. We're going to add weight decay which in our case is 0.1 times the sum of weights squared.

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

↪ MNIST SGD [1:23:59]

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

So let's do that and let's make it interesting by not using synthetic data but let's use some real data. We're going to use MNIST - the hand-drawn digits. But we're going to do this as a standard fully connected net, not as a convolutional net because we haven't learnt the details of how to really create one of those from scratch. So in this case, is actually [deeplearning.net](#) provides MNIST as a Python pickle file, in other words it's a file that Python can just open up and it'll give you numpy arrays straight away. They're flat numpy arrays, we don't have to do anything to them. So go grab that and it's a gzip file so you can actually just `gzip.open` it directly and then you can `pickle.load` it directly, and again `encoding='latin-1'`.

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

```
path.ls()
```

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

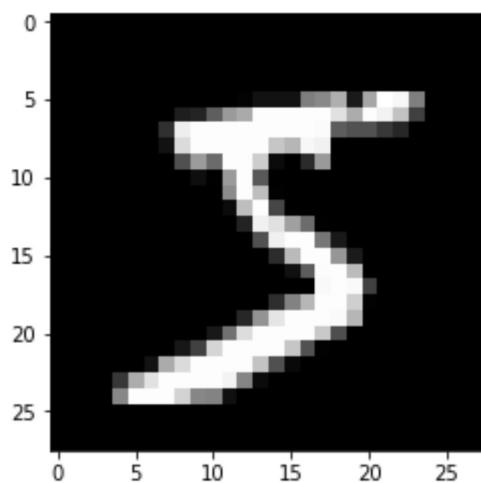
```
with gzip.open(path/'mnist.pkl.gz', 'rb') as f:  
    ((x_train, y_train), (x_valid, y_valid), _) = pickle.load(f, encoding='latin-
```

That'll give us the training, the validation, and the test set. I don't care about the test set, so generally in Python if there's something you don't care about, you tend to use this special variable called underscore (`_`). There's no reason you have to. It's just people know you mean I don't care about this. So there's our training `x & y`, and a valid `x & y`.

Now this actually comes in as the shape 50,000 rows by 784 columns, but those 784 columns are actually 28 by 28 pixel pictures. So if I reshape one of them into a 28 by 28 pixel picture and plot it, then you can see it's the number five:

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

```
(50000, 784)
```



So that's our data. We've seen MNIST before in its pre-reshaped version, here it is in flattened version. So I'm going to be using it in its flattened version.

Currently they are numpy arrays. I need them to be tensors. So I can just map `torch.tensor` across all of them, and so now they're tensors.

```
x_train,y_train,x_valid,y_valid = map(torch.tensor, (x_train,y_train,x_valid,y_valid))
n,c = x_train.shape
x_train.shape, y_train.min(), y_train.max()

(torch.Size([50000, 784]), tensor(0), tensor(9))
```

I may as well create a variable with the number of things I have which we normally call `n`. Here, we use `c` to mean the number of columns (that's not a great name for it sorry). So there we are. Then the `y` not surprisingly the minimum value is 0 and the maximum value is 9 because that's the actual number we're gonna predict.

[1:26:38]

In lesson 2 SGD, we created a data where we actually added a column of 1's on so that we didn't have to worry about bias:

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

We're not going to do that. We're going to have PyTorch to do that implicitly for us. We had to write our own MSE function, we're not going to do that. We had to write our own little matrix multiplication thing, we're not going to do that. We're gonna have PyTorch do all this stuff for us now.

What's more and really important, we're going to do mini batches because this is a big enough dataset we probably don't want to do it all at once. So if you want to do mini batches, so we're not going to use too much fastai stuff here, PyTorch has something called `TensorDataset` that basically grabs two tensors and creates a dataset, Remember a dataset is something where if you index into it, you get back an x value and a y value - just one of them. It looks a lot like a list of xy tuples.

```
bs=64
train_ds = TensorDataset(x_train, y_train)
valid_ds = TensorDataset(x_valid, y_valid)
data = DataBunch.create(train_ds, valid_ds, bs=bs)
```

Once you have a dataset, then you can use a little bit of convenience by calling `DataBunch.create` and what that is going to do is it's going to create data loaders for you. A data loader is something which you don't say I want the first thing or the fifth thing, you just say I want the "next" thing, and it will give you a mini batch of whatever size you asked for. Specifically it'll give you the X and the y of a mini batch. So if I just grab the `next` of the iterator (this is just standard Python). Here's my training data loader (`data.train_dl`) that `DataBunch.create` creates for you. You can check that as you would expect the X is 64 by 784 because there's 784 pixels flattened out, 64 in a mini batch and the Y is just 64 numbers - they are things we're trying to predict.

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

If you look at the source code for `DataBunch.create`, you'll see there's not much there, so feel free to do so. We just make sure that your training set gets randomly shuffled for you. We make sure that the data is put on the GPU for you. Just a couple of little convenience things like that. But don't let it be magic. If it feels magic check out the source code to make sure you see what's going on.

Rather than do this `y_hat = x@a` thing, we're going to create an `nn.Module`. If you want to create an `nn.Module` that does something different to what's already out there, you have to subclass it. So sub classing is very very very normal in PyTorch. So if you're not comfortable with sub classing stuff in Python, go read a couple of tutorials to make sure you are. The main thing is you have to override the constructor dunder init (`__init__`) and make sure that you call the super class' constructor (`super().__init__()`) because `nn.Module` super class' constructor is going to set it all up to be a proper `nn.Module` for you. So if you're trying to create your own PyTorch subclass and things don't work, it's almost certainly because you forgot this line of code.

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

```
super().__init__()  
self.lin = nn.Linear(784, 10, bias=True)  
  
def forward(self, xb): return self.lin(xb)
```

[1:30:04]

So the only thing we want to add is we want to create an attribute in our class which contains a linear layer an `nn.Linear` module. What is an `nn.Linear` module? It's something which does `x@a`, but actually it doesn't only do that it actually is `x@a + b`. So in other words, we don't have to add the column of ones. That's all it does. If you want to play around, why don't you try and create your own `nn.Linear` class? You could create something called `MyLinear` and it'll take you (depending on your PyTorch background) an hour or two. We don't want any of this to be magic, and you know all of the things necessary to create this now. So these are the kind of things that you should be doing for your assignments this week. Not so much new applications but try to start writing more of these things from scratch and get them to work. Learn how to debug them, check what's going in and out and so forth.

But we could just use `nn.Linear` and that's this going to do so it's going to have a `def forward` in it that goes `a@x + b`. Then in our `forward`, how do we calculate the result of this? Remember, every `nn.Module` looks like a function, so we pass our X mini-batch so I tend to use `xb` to mean a batch of X to `self.lin` and that's going to give us back the result of the `a@x + b` on this mini batch.

So this is a logistic regression model. A logistic regression model is also known as a neural net with no hidden layers, so it's a one layer neural net, no nonlinearities.

Because we're doing stuff ourself a little bit we have to put the weight matrices (i.e. the parameters) onto the GPU manually. So just type `.cuda()` to do that.

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

```
model
```

```
Mnist_Logistic(  
    (lin): Linear(in_features=784, out_features=10, bias=True)  
)
```

Here's our model. As you can see the `nn.Module` machinery has automatically given us a representation of it. It's automatically stored the `.lin` thing, and it's telling us what's inside it.

```
model.lin  
  
Linear(in_features=784, out_features=10, bias=True)  
  
model(x).shape  
  
torch.Size([64, 10])  
  
[p.shape for p in model.parameters()  
  
[torch.Size([10, 784]), torch.Size([10])]
```

So there's a lot of little conveniences that PyTorch does for us. If you look now at `model.lin`, you can see, not surprisingly, here it is.

Perhaps the most interesting thing to point out is that our model automatically gets a bunch of methods and properties. And perhaps the most interesting one is the one called `parameters` which contains all of the yellow squares from our picture. It contains our parameters. It contains our weight matrices and bias matrices in as much as they're different. So if we have a look at `p.shape for p in model.parameters()`, there's something of 10 by 784, and there's something of 10. So what are they? 10 by 784 - so that's the thing that's going to take in 784 dimensional input and spit out a 10 dimensional output. That's handy because our input is 784 dimensional and we need something that's going to give us a probability of 10 numbers. After that happens we've got ten activations which we then want to add the bias to, so there we go. Here's a vector of length 10. So you can see why this model we've created has exactly the stuff that we need to do our `a@x+b`.

[1:33:40]

`lr=2e-2`

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

Let's grab a learning rate. We're going to come back to this loss function in a moment but we can't really use MSE for this because we're not trying to see "how close are you". Did you predict 3 and actually it was 4, gosh you were really close. No, 3 is just as far away from 4 as 0 is away from 4 when you're trying to predict what number did somebody draw. So we're not going to use MSE, we're going to use cross-entropy loss which we'll look at in a moment.

```
def update(x,y,lr):
    wd = 1e-5
    y_hat = model(x)
    # weight decay
    w2 = 0.
    for p in model.parameters(): w2 += (p**2).sum()
    # add to regular loss
    loss = loss_func(y_hat, y) + w2*wd
    loss.backward()
    with torch.no_grad():
        for p in model.parameters():
            p.sub_(lr * p.grad)
            p.grad.zero_()
    return loss.item()
```

Here's our `update` function. I copied it from lesson 2 SGD, but now we're calling our model rather than going `a@x`. We're calling our model as if it was a function to get `y_hat` and we're calling our `loss_func` rather than calling MSE to get our loss. Then the rest is all the same as before except rather than going through each parameter and going `parameter. sub_(learning_rate*gradient)`, we loop through the parameters. Because very nicely for us, PyTorch will automatically create this list of the parameters of anything that we created in our dunder init.

And look, I've added something else. I've got this thing called `w2`, I go through each `p` in `model.parameters()` and I add to `w2` the sum of squares. So `w2` now contains my sum of squared weights. Then I multiply it by some number which I set to `1e-5`. So now I just implemented weight decay. So when people talk about weight decay, it's not an amazing magic complex thing containing thousands of lines of CUDA C++ code. It's those two lines of Python:

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

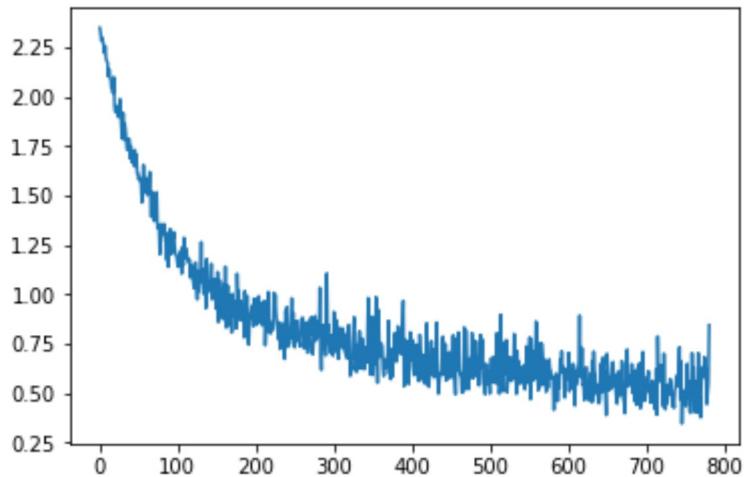
That's weight decay. This is not a simplified version that's just enough for now, this *is* weight decay. That's it.

So here's the thing. There's a really interesting kind of dual way of thinking about weight decay. One is that we're adding the sum of squared weights and that seems like a very sound thing to do and it is. Well, let's go ahead and run this.

```
losses = [update(x,y,lr) for x,y in data.train_d1]
```

Here I've just got a list comprehension that's going through my data loader. The data loader gives you back one mini batch for the whole thing giving you XY each time, I'm gonna call update for each. Each one returns loss. Now PyTorch tensors, since I did it all on the GPU that's sitting in the GPU. And it's got all these stuff attached to it to calculate gradients, it's going to use up a lot of memory. So if you called `.item()` on a scalar tensor, it turns it into an actual normal Python number. So this is just means I'm returning back normal Python numbers.

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



And then I can plot them, and there you go. My loss function is going down. It's really nice to try this stuff to see it behaves as you expect. We thought this is what would happen - as we get closer and closer to the answer it bounces around more and more, because we're kind of close to where we should be. It's probably getting flatter in weight space, so we kind of jumping further. So you can see why we would probably want to be reducing our learning rate as we go (i.e. learning rate annealing).

Now here's the thing.

$$w_+ = w_{+-} - \beta \times \frac{dL}{dw_{+-}}$$

$$L(x, w) = \underline{\text{mse}(m(x, w), y)} + \text{wd. } \underline{\sum w^2}$$

That ($wd \cdot \sum w^2$) is only interesting for training a neural net because it appears here (dL). Because we take the gradient of it. That's the thing that actually updates the weights. So actually the only thing interesting about $wd \cdot \sum w^2$ is its gradient. So we don't do a lot of math here, but I think we can handle that. The gradient of this whole thing if you remember back to your high school math is equal to the gradient of each part taken separately and then add them together. So let's just take the gradient of that ($wd \cdot \sum w^2$) because we already know the gradient of this ($\text{mse}(m(x, w), y)$) is just whatever we had before. So what's the gradient of $wd \cdot \sum w^2$?

Let's remove the sum and pretend there's just one parameter. It doesn't change the generality of it. So the gradient of $wd \cdot w^2$ - what's the gradient of that with respect to w ?

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

It's just $2wd \cdot w$. So remember this (wd) is our constant which in that little loop was $1e-5$. And w is our weights. We could replace wd with like $2wd$ without loss of generality, so let's throw away the 2. So in other words, all weight decay does is it subtracts some constant times the weights every time we do a batch. That's why it's called weight decay.

- When it's in this form ($wd \cdot w^2$) where we add the square to the loss function, that's called **L2 regularization**.
- When it's in this form ($wd \cdot w$) where we subtract wd times weights from the gradients, that's called **weight decay**.

They are kind of mathematically identical. For everything we've seen so far, in fact they are mathematically identical. And we'll see in a moment a place where they're not - where things get interesting. So this is just a really important tool you now have in your toolbox. You can make giant neural networks and still avoid overfitting by adding more weight decay. Or you could use really small datasets with moderately large sized models and avoid overfitting with weight decay. It's not magic. You might still find you don't have enough data in which case you get to the point where you're not overfitting by adding lots of weight decay and it's just not training very well - that can happen. But at least this is something that you can now play around with.

↪ MNIST neural network [1:40:33]

Now that we've got this `update` function, we could replace this `Mnist_Logistic` with MNIST neural network and build a neural network from scratch.

```
class Mnist_NN(nn.Module):
    def __init__(self):
        super().__init__()
        self.lin1 = nn.Linear(784, 50, bias=True)
        self.lin2 = nn.Linear(50, 10, bias=True)

    def forward(self, xb):
        x = self.lin1(xb)
        x = F.relu(x)
        return self.lin2(x)
```

Now we just need two linear layers. In the first one, we could use a weight matrix of size 50. We need to make sure that the second linear layer has an input of size 50 so it matches. The final layer has to have an output of size 10 because that's the number of classes we're predicting. So now our `forward` just goes:

- do a linear layer
- calculate ReLU
- do a second linear layer

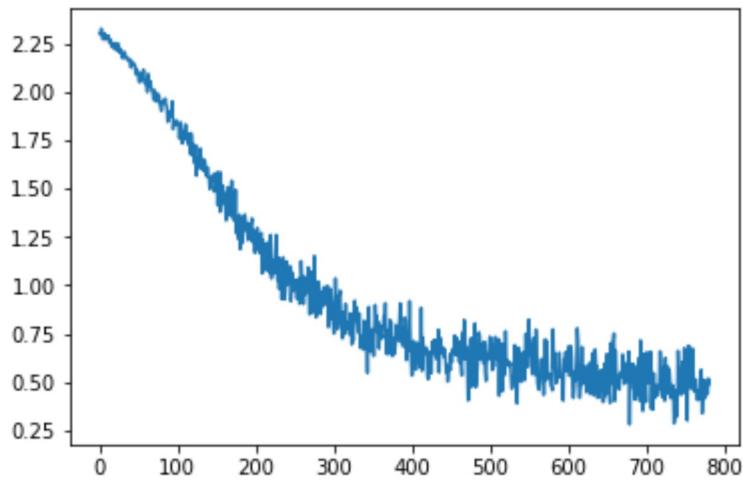
Now we've actually created a neural net from scratch. I mean we didn't write `nn.Linear` but you can write it yourself or you could do the matrices directly - you know how to.

Again we can go model dot CUDA, and then we can calculate losses for the exact same `update` function, there it goes.

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

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

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



So this is why this idea of neural nets is so easy. Once you have something that can do gradient descent, then you can try different models. Then you can start to add more PyTorch stuff. Rather than doing all this stuff yourself (`update` function), why not just go `opt = optim.something`? So the "something" we've done so far is SGD.

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

Now you're saying to PyTorch I want you to take these parameters and optimize them using SGD. So this now, rather than saying `for p in parameters: p -= lr * p.grad`, you just say `opt.step()`. It's the same thing. It's just less code and it does the same thing. But the reason it's kind of particularly interesting is that now you can replace `SGD` with `Adam` for example and you can even add things like weight decay because there's more stuff in these things for you. So that's why we tend to use `optim.blah`. So behind the scenes, this is actually what we do in fastai.

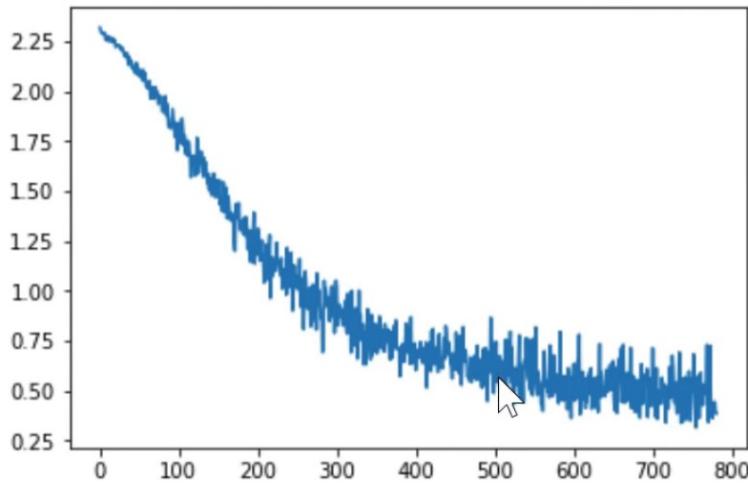
[1:42:54]

So if I go `optim.SGD`, the plot looks like before:

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

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

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

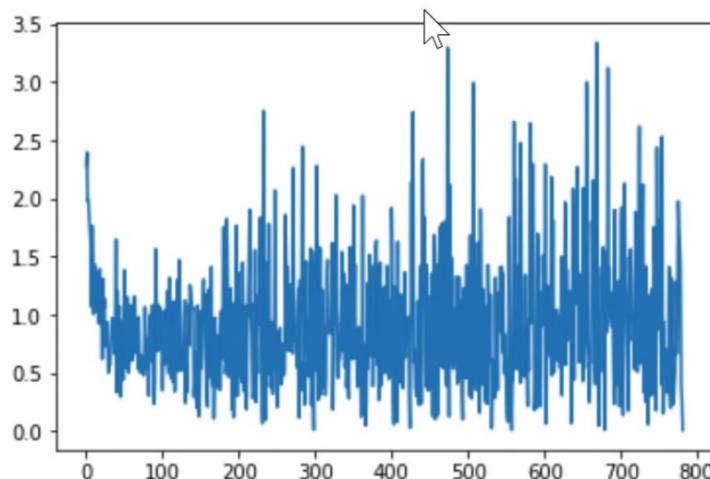


But if we change to a different optimizer, look what happened:

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

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

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

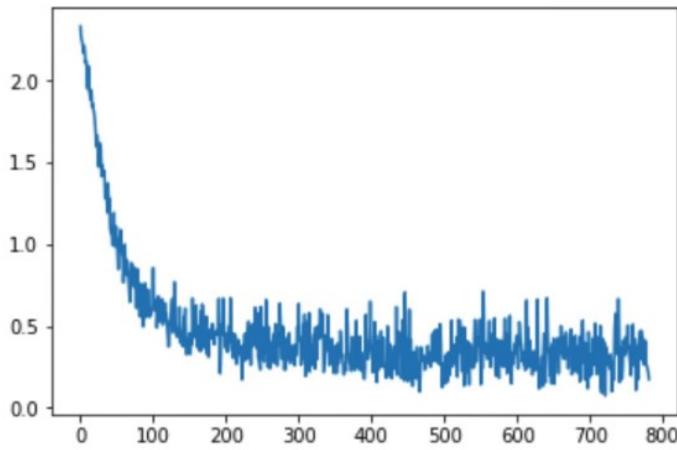


It diverged. We've seen a great picture of that from one of our students who showed what divergence looks like. This is what it looks like when you try to train something. Since we're using a different optimizer, we need a different learning rate. And you can't just continue training because by the time it's diverged, the weights are really really big and really really small - they're not going to come back. So start again.

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

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

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



Okay, there's a better learning rate. But look at this - we're down underneath 0.5 by about epoch 200. Whereas before (with SGD), I'm not even sure we ever got to quite that level. So what's going on? What's Adam? Let me show you.

⌚ Adam [1:43:56]

[graddesc.xlsx](#)

We're gonna do gradient descent in Excel because why wouldn't you. So here is some randomly generated data:

	A	B	C	D
1	b	const	30	
2	a	slope	2	
3				
4		x	y	
5		18	66	
6		28	86	
7		85	200	
8		38	106	
9		28	86	
10		40	110	
11		77	184	
12		9	48	
13		30	90	
14		61	152	
15		8	46	
16		40	110	
17		55	140	
18		90	210	
19		18	66	
20		32	94	
21		13	56	
22		55	140	
23		56	142	
24		93	216	
25		89	208	
26		47	124	
27		75	180	
28		67	164	

They're randomly generated X's' and the Y's are all calculated by doing $ax + b$ where a is 2 and b is 30. So this is some data that we have to try and match. Here is SGD:

AutoSave [On] File Home Insert Draw Page Layout Formulas Data Review View Developer Add-ins Help Team Tell me what you want to do

graddesc.xlsx - Excel Jeremy Howard

A4 : fx 14

	A	B	C	D	E	F	G	H	I	J	K	L
1	Reset	intercept	1	learn	0.0001				de/db=2(ax+b-y)			
2	Run	slope	1				e=(ax+b-y)^2 de/da=x^2(ax+b-y)					
3	x	y	intercept	slope	y_pred	err^2	errb1	est de/db	erra1	est de/da	de/db	de/da
4	14	58	1	1	15	1,849.00	1,848.14	-85.99	1,836.98	-1,202.04	-86.00	-1,204.00
5	86	202	1.0086	1.1204	97.363	10,948.90	10,946.81	-209.26	10,769.67	-17,923.60	-209.27	-17,997.56
6	28	86	1.029527	2.9202	82.7939	10.28	10.22	-6.40	8.56	-171.70	-6.41	-179.54
7	51	132	1.030169	2.9381	150.874	356.22	356.60	37.76	375.73	1,951.14	37.75	1,925.13
8	28	86	1.026394	2.7456	77.9031	65.56	65.40	-16.18	61.10	-445.58	-16.19	-453.42
9	29	88	1.028013	2.7909	81.9653	36.42	36.30	-12.06	33.00	-341.60	-12.07	-350.01
10	72	174	1.02922	2.8259	204.497	930.07	930.68	61.00	974.50	4,443.41	60.99	4,391.57
11	62	154	1.023121	2.3868	149.004	24.96	24.86	-9.98	19.15	-581.09	-9.99	-619.53
12	84	198	1.02412	2.4487	206.718	76.01	76.18	17.45	91.36	1,535.20	17.44	1,464.64
13	15	60	1.022376	2.3023	35.5565	597.49	597.00	-48.88	590.17	-731.06	-48.89	-733.31
14	42	114	1.027265	2.3756	100.803	174.17	173.91	-26.38	163.26	-1,090.94	-26.39	-1,108.58
15	62	154	1.029905	2.4865	155.191	1.42	1.44	2.39	3.28	186.07	2.38	147.63
16	47	124	1.029666	2.4717	117.2	46.25	46.11	-13.59	40.07	-617.15	-13.60	-639.24
17	35	100	1.031027	2.5356	89.7779	104.49	104.29	-20.43	97.46	-703.30	-20.44	-715.55
18	9	48	1.033071	2.6072	24.4977	552.36	551.89	-46.99	548.14	-422.23	-47.00	-423.04
19	38	106	1.037771	2.6495	101.718	18.33	18.25	-8.55	15.22	-310.98	-8.56	-325.42
20	44	118	1.038628	2.682	119.048	1.10	1.12	2.11	2.21	111.56	2.10	92.20
21	99	228	1.038418	2.6728	265.646	1,417.23	1,417.98	75.30	1,492.75	7,551.94	75.29	7,453.93
		adagrad_ann	adam_ann	adam	rmsprop	adagrad	momentum	basic	data			

So we have to do it with SGD. Now in our lesson 2 SGD notebook, we did the whole dataset at once as a batch. In the notebook we just looked at, we did mini batches. In this spreadsheet, we're going to do online gradient descent which means every single row of data is a batch. So it's kind of a batch size of one.

As per usual, we're going to start by picking an intercept and slope kind of arbitrarily, so I'm just going to pick 1 - doesn't really matter. Here I've copied over the data. This is my x and y and my intercept and slope, as I said, is 1. I'm just literally referring back to cell (C1) here.

So my prediction for this particular intercept and slope would be 14 times one plus one which is 15, and there is my squared error:

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

Now I need to calculate the gradient so that I can update. There's two ways you can calculate the gradient. One is analytically and so you know you can just look them up on Wolfram Alpha or whatever so there's the gradients ($de/db=2(ax+b-y)$) if you write it out by hand or look it up.

Or you can do something called finite differencing because remember gradient is just how far the outcome moves divided by how far your change was for really small changes. So let's just make a really small change.

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

Here we've taken our intercept and added 0.01 to it, and then calculated our loss. You can see that our loss went down a little bit and we added 0.01 here, so our derivative is that difference divided by that 0.01:

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

That's called finite differencing. You can always do derivatives over finite differencing. It's slow. We don't do it in practice, but it's nice for just checking stuff out. So we can do the same thing for our a term, add 0.01 to that, take the difference and divide by 0.01.

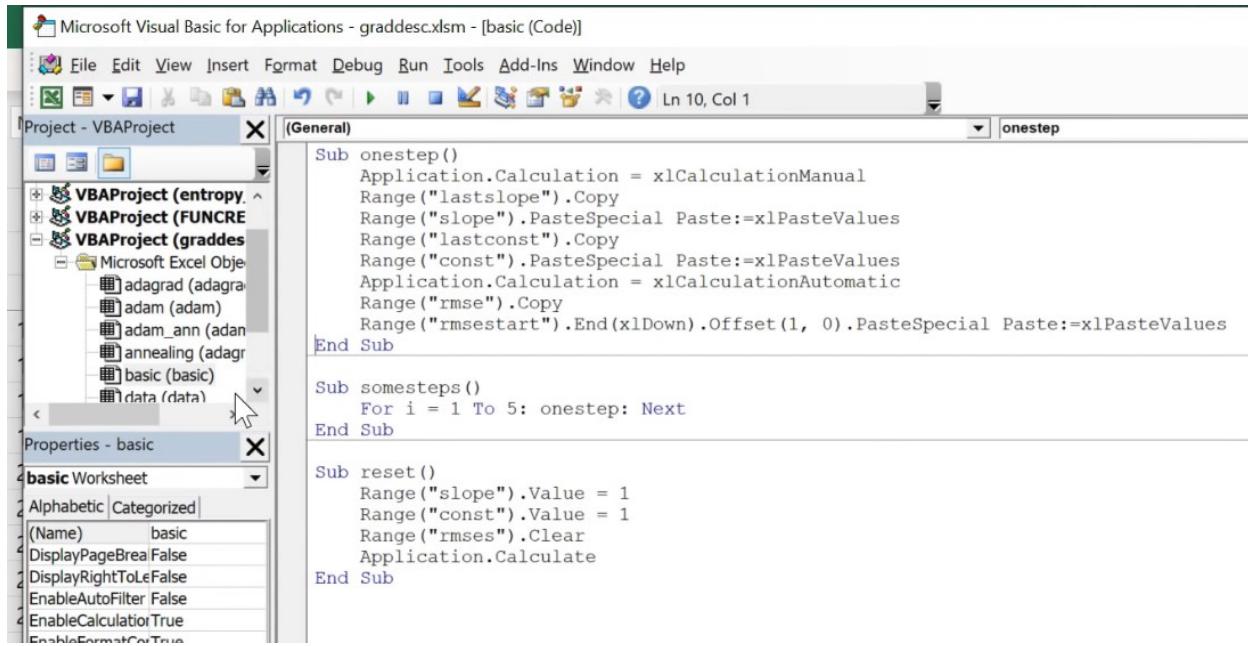
	intercept	1	learn	0.0001		de/db=2(ax+b-y)							
	slope	1				e=(ax+b-y)^2	de/da=x^2(ax+b-y)						
x	y	intercept	slope	y_pred	err^2	errb1	est de/db	erra1	est de/da	de/db	de/da	new a	new b
14	58	1	1	15	1,849.00	1,848.14	-85.99	1,836.98	-1,202.04	-86.00	-1,204.00	1.12	1.01
86	202	1.0086	1.1204	97.363	10,948.90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

Or as I say, we can calculate it directly using the actual derivative analytical and you can see est de/db and de/db are as you'd expect very similar (as well as est de/da and de/da).

So gradient descent then just says let's take our current value of that weight (slope) and subtract the learning rate times the derivative - there it is (new a , new b). And so now we can copy that intercept and that slope to the next row, and do it again. And do it lots of times, and at the end we've done one epoch.

At the end of that epoch, we could say "oh great so this is our slope, so let's copy that over to where it says slope, and this is our intercept so I'll copy it to where it says intercept, and now it's done another epoch."

So that's kind of boring I'm copying and pasting so I created a very sophisticated macro which copies and pastes for you (I just recorded it) and then I created a very sophisticated for loop that goes through and does it five times:



```

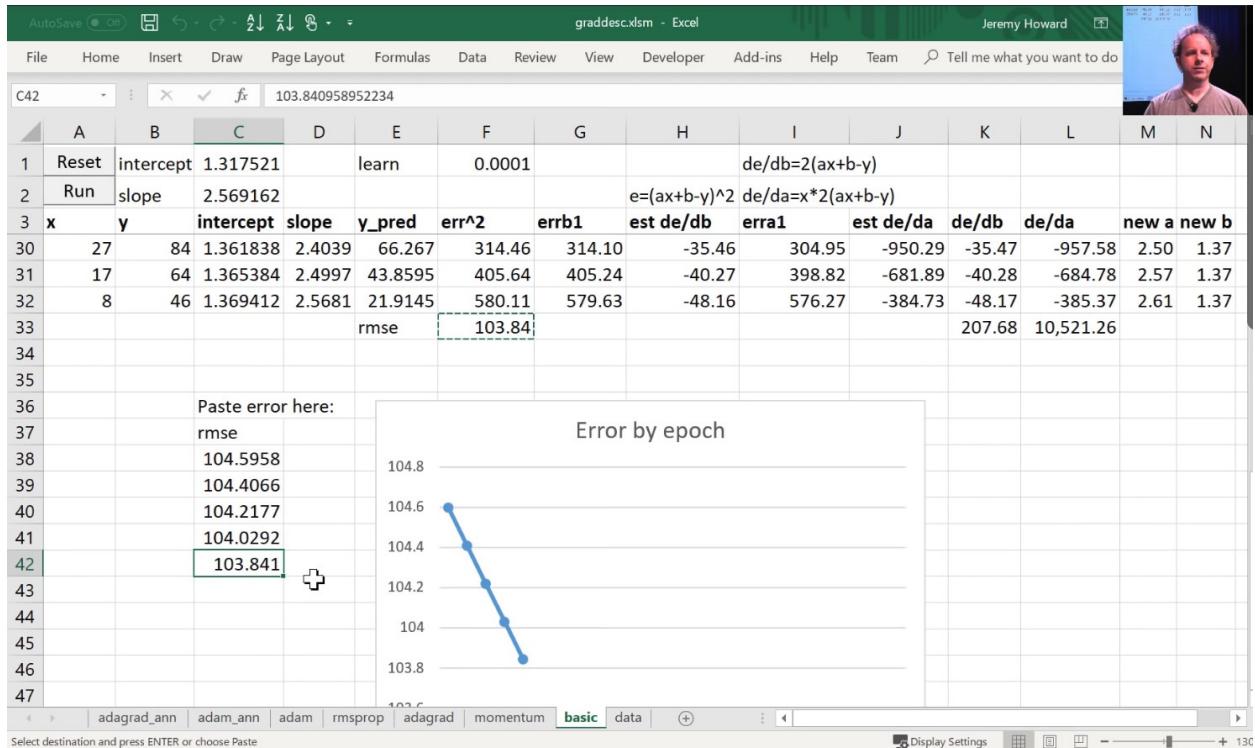
Sub onestep()
    Application.Calculation = xlCalculationManual
    Range("lastslope").Copy
    Range("slope").PasteSpecial Paste:=xlPasteValues
    Range("lastconst").Copy
    Range("const").PasteSpecial Paste:=xlPasteValues
    Application.Calculation = xlCalculationAutomatic
    Range("rmse").Copy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

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

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

```

I attach that to the Run button, so if I press run, it'll go ahead and do it five times and just keep track of the error each time.



So that is SGD. As you can see, it is just infuriatingly slow like particularly the intercept is meant to be 30 and we're still only up to 1.57, and it's just going so slowly. So let's speed it up.

⌚ Momentum [1:48:40]

The first thing we can do to speed it up is to use something called momentum. Here's the exact same spreadsheet as the last worksheet. I've removed the finite differencing version of the derivatives because they're not that useful, just the analytical ones here. $\frac{de}{db}$ where I take the derivative and I'm going to update by the derivative.

	A	B	C	D	E	F	G	H	I	J	K	L
1	b	1.0000	learn						beta	0.9	0.1	
2	a	1.0000	0.0001									
3	x	y	b	a	pred	$\frac{de}{db}$	$\frac{de}{da}$	new b	new a	-18.33	98.25	err^2
4	14	58	1.0000	1.0000	15.0000	-86.00	-1204.00	1.00	1.00	-25.1	-31.98	1849
5	86	202	1.0025	1.0032	87.2775	-229.44	-19732.27	1.01	1.20	-45.53	-2002	13161.25
6	28	86	1.0071	1.2034	34.7022	-102.60	-2872.68	1.01	1.41	-51.24	-2089	2631.462
7	51	132	1.0122	1.4123	73.0398	-117.92	-6013.94	1.02	1.66	-57.91	-2482	3476.306
8	28	86	1.0180	1.6605	47.5109	-76.98	-2155.39	1.02	1.91	-59.82	-2449	1481.409
9	29	88	1.0240	1.9054	56.2793	-63.44	-1839.80	=J8*\$J\$1+\$K\$1*F9		-2388	1006.203	
10	72	174	1.0300	2.1442	155.4095	-37.18	-2677.04	1.04	2.39	-57.88	-2417	345.6083
11	62	154	1.0358	2.3859	148.9586	-10.08	-625.13	1.04	2.61	-53.1	-2238	25.41554
12	84	198	1.0411	2.6096	220.2498	44.50	3737.96	1.05	2.77	-43.34	-1640	495.0529
13	15	60	1.0454	2.7736	42.6501	-34.70	-520.50	1.05	2.93	-42.48	-1528	301.0194
14	42	114	1.0497	2.9265	123.9612	19.92	836.74	1.05	3.06	-36.24	-1292	99.22647
15	62	154	1.0533	3.0556	190.5028	73.01	4526.35	1.06	3.13	-25.31	-709.9	1332.457
16	47	124	1.0558	3.1266	148.0074	48.01	2256.69	1.06	3.17	-17.98	-413.2	576.3537
17	35	100	1.0576	3.1680	111.9360	23.87	835.52	1.06	3.20	-13.79	-288.4	142.4678
18	9	48	1.0590	3.1968	29.8301	-36.34	-327.06	1.06	3.23	-16.05	-292.2	330.145
19	38	106	1.0606	3.2260	123.6492	35.30	1341.34	1.06	3.24	-10.91	-128.9	311.4928
20	44	118	1.0617	3.2389	143.5734	51.15	2250.46	1.06	3.23	-4.708	109.1	654
21	99	228	1.0622	3.2280	320.6340	185.27	18341.52	1.06	3.03	14.29	1932	8581.05
22	13	56	1.0607	3.0348	40.5127	-30.97	-402.67	1.06	2.86	9.763	1699	239.8561
23	21	72	1.0598	2.8649	61.2224	-21.56	-452.66	1.06	2.72	6.632	1484	116.1567
24	28	86	1.0591	2.7165	77.1217	-17.76	-497.18	1.06	2.59	4.193	1286	78.824
25	20	70	1.0587	2.5880	52.8180	-34.36	-687.28	1.06	2.48	0.337	1088	295.2221
26	8	46	1.0586	2.4791	20.8917	-50.22	-401.73	1.06	2.39	-4.718	939.3	630.4254
27	64	158	1.0591	2.3852	153.7124	-8.58	-548.81	1.06	2.31	-5.104	790.5	18.38337
28	99	228	1.0596	2.3062	229.3695	2.74	271.16	1.06	2.23	-4.32	738.5	1.875537
29	70	170	1.0600	2.2323	157.3215	-25.36	-1774.99	1.06	2.18	-6.423	487.2	160.745
30	27	84	1.0607	2.1836	60.0175	-47.96	-1295.05	1.06	2.15	-10.58	309	575.1583
31	17	64	1.0617	2.1527	37.6575	-52.69	-895.65	1.06	2.13	-14.79	188.5	693.928
32	8	46	1.0632	2.1338	18.1339	-55.73	-445.86	1.07	2.12	-18.88	125.1	776.5169
33										-18.33	98.25	200.9652

But what I do is I take the derivative and I multiply it by 0.1. And what I do is I look at the previous update and I multiply that by 0.9 and I add the two together. So in other words, the update that I do is not just based on the derivative but 1/10 of it is the derivative and 90% of it is just the same direction I went last time. This is called momentum. What it means is, remember how we thought about what might happen if you're trying to find the minimum of this.



You were here and your learning rate was too small, and you just keep doing the same steps. Or if you keep doing the same steps, then if you also add in the step you took last time, and your steps are going to get bigger and bigger until eventually they go too far. But now, of course, your gradient is pointing the other direction to where your momentum is pointing. So you might just take a little step over here, and then you'll start going small steps, bigger steps, bigger steps, small steps, bigger steps, like that. That's kind of what momentum does.

If you're going too far like this which is also slow all, then the average of your last few steps is actually somewhere between the two, isn't it? So this is a really common idea - when you have something that says my step at time T equals some number (people often use alpha because gotta love these Greek letters) times the actual thing I want to do (in this case it's the gradient) plus one minus alpha times whatever you had last time (S_{t-1}):

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

This is called an **exponentially weighted moving average**. The reason why is that, if you think about it, these $(1 - \alpha)$ are going to multiply. So if S_{t-2} is in here with $(1 - \alpha)^2$ and S_{t-3} is in there with $(1 - \alpha)^3$.

So in other words S_t ends up being the actual thing I want ($\alpha \cdot g$) plus a weighted average of the last few time periods where the most recent ones are exponentially higher weighted. And this is going to keep popping up again and again. So that's what momentum is. It says I want to go based on the current gradient plus the exponentially weighted moving average of my last few steps. So that's useful. That's called SGD with momentum, and we can do it by changing:

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

to

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

Momentum 0.9 is really common. It's so common it's always 0.9 (just about) four basic stuff. So that's how you do SGD with momentum. And again I didn't show you some simplified version, I showed you "the" version. That is SGD. Again you can write your own. Try it out. That would be a great assignment would be to take lesson 2 SGD and add momentum to it; or even the new notebook we've got MNIST, get rid of the `optim.` and write your own update function with momentum.

↪ RMSProp [1:53:30]

Then there's a cool thing called RMSProp. One of the really cool things about RMSProp is that Geoffrey Hinton created it (a famous neural net guy). Everybody uses it. It's like really popular and common. The correct citation for RMSProp is the Coursera online free MOOC. That's where he first mentioned RMSProp so I love this thing that cool new things appear in MOOCs not a paper.

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

Sheet1 | data | basic | momentum | rmsprop | adam | adam_ann |

So RMSProp is very similar to momentum but this time we have an exponentially weighted moving average not of the gradient updates but of F_8 squared - that's the gradient squared. So what the gradient squared times 0.1 plus the previous value times 0.9. This is an exponentially weighted moving average of the gradient squared. So what's this number going to mean? Well if my gradient is really small and consistently really small, this will be a small number. If my gradient is highly volatile, it's going to be a big number. Or if it's just really big all the time, it'll be a big number.

Why is that interesting? Because when we do an update this time we say weight minus learning rate times gradient divided by the square root of this (shown as x below).

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

So in other words, if our gradient is consistently very small and not volatile, let's take bigger jumps. That's kind of what we want, right? When we watched how the intercept moves so darn slowly, it's like obviously you need to just try to go faster.

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

So if I now run this, after just 5 epochs, this is already up to 3. Where else, with the basic version after five epochs it's still at 1.27. Remember, we have to get to 30.

Adam [1:55:44]

So the obvious thing to do (and by obvious I mean only a couple of years ago did anybody actually figure this out) is do both. So that's called **Adam**. So Adam is simply keep track of the exponentially weighted moving average of the gradient squared (RMSProp) and also keep track of the exponentially weighted moving average of my steps (momentum). And both divided by the exponentially weighted moving average of the squared terms and take 0.9 of a step in the same direction as last time. So it's momentum and RMSProp - that's called Adam. And look at this - 5 steps, we're at 25.

These optimizes, people call them dynamic learning rates. A lot of people have the misunderstanding that you don't have to set a learning rate. Of course, you do. It's just like trying to identify parameters that need to move faster or consistently go in the same direction. It doesn't mean you don't need learning rates. We still have a learning rate. In fact, if I run this again, it's getting better but eventually it's just moving around the same place. So you can see what's happened is the learning rate is too high. So we could just drop it down and run it some more. Getting pretty close now, right?

So you can see, how you still need learning rate annealing even with Adam. That spreadsheet is fun to play around with. I do have a [Google sheets version](#) of basic SGD that actually works and the macros work and everything. Google sheet is so awful and I went so insane making that work that I gave up I'm making the other ones work. So I'll share a link to the Google sheets version. Oh my god, they do have a macro language but it's just ridiculous. Anyway, if somebody feels like fighting it to actually get all the other ones to work, they'll work. It's just annoying. So maybe somebody can get this working on Google sheets too.

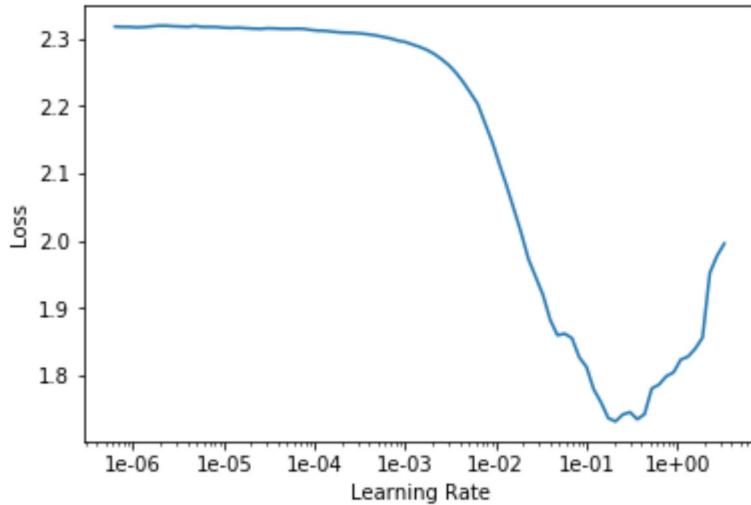
[1:58:37]

So that's weight decay and Adam, and Adam is amazingly fast.

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

But we don't tend to use `optim.` whatever and create the optimizer ourselves and all that stuff. Because instead, we had to use learner. But learn is just doing those things for you. Again, there's no magic. So if you create a learner you say here's my data bunch, here's my PyTorch `nn.Module` instance, here's my loss function, and here are my metrics. Remember, the metrics are just stuff to print out. That's it. Then you just get a few nice things like `learn.lr_find` starts working and it starts recording this:

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



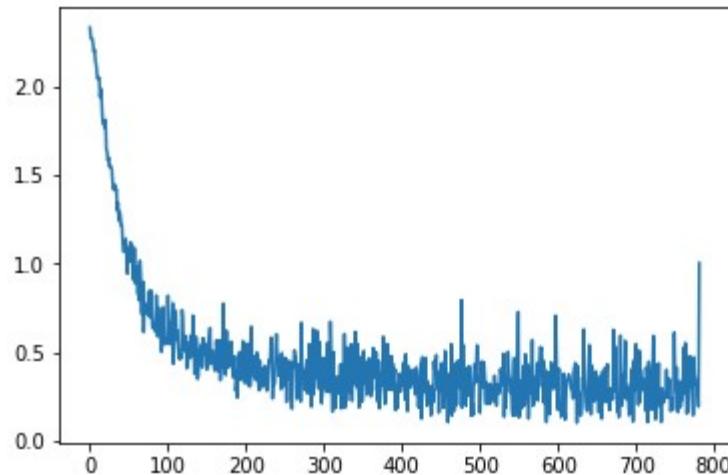
And you can say `fit_one_cycle` instead of just `fit`. These things really help a lot like.

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

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

By using the learning rate finder, I found a good learning rate. Then like look at this, my loss here 0.13. Here I wasn't getting much beneath 0.5:

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

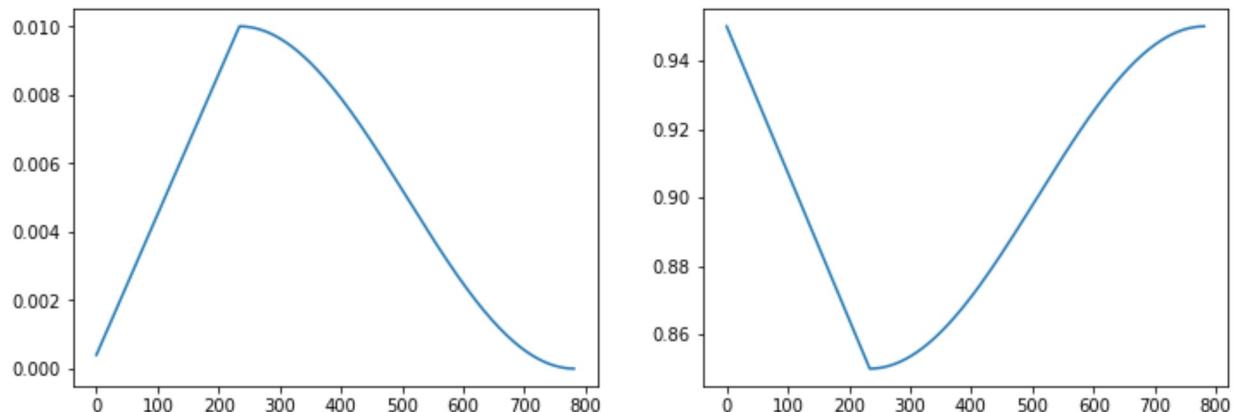


So these tweeks make huge differences; not tiny differences. And this is still just one one epoch.

⌚ Fit one cycle [2:00:02]

Now what does fit one cycle do? What does it really do? This is what it really does:

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

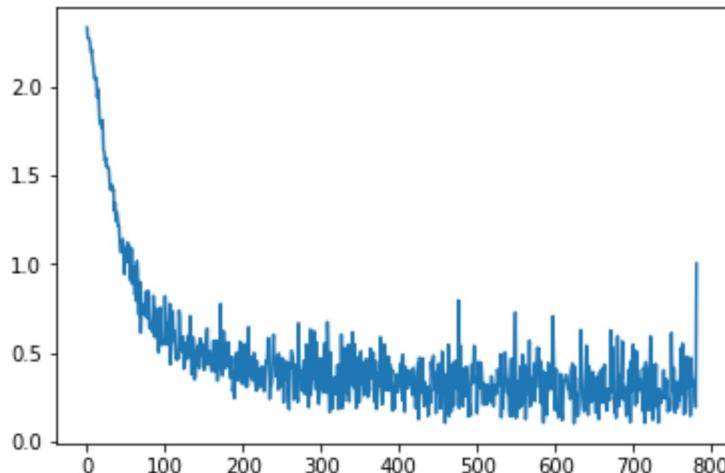


We've seen this chart on the left before. Just to remind you, this is plotting the learning rate per batch. Remember, Adam has a learning rate and we use Adam by default (or minor variation which we might try to talk about). So the learning rate starts really low and it increases about half the time, and then it decreases about half the time. Because at the very start, we don't know where we are. So we're in some part of function space, it's just bumpy as all heck. So if you start jumping around, those bumps have big gradients and it will throw you into crazy parts of the space. So start slow. Then you'll gradually move into parts of the weight space that is sensible. And as you get to the points where they're sensible, you can increase the learning rate because the gradients are actually in the direction you want to go. Then as we've discussed a few times, as you get close to the final answer you need to anneal your learning rate to hone in on it.

But here's the interesting thing - on the right is the momentum plot. Every time our learning rate is small, our momentum is high. Why is that? Because I do have a learning small learning rate, but you keep going in the same direction, you may as well go faster. But if you're jumping really far, don't like jump really far because it's going to throw you off. Then as you get to the end again, you're fine tuning in but actually if you keep going the same direction again and again, go faster. So this combination is called one cycle and it's a simple thing but it's astonishing. This can help you get what's called super convergence that can let you train 10 times faster.

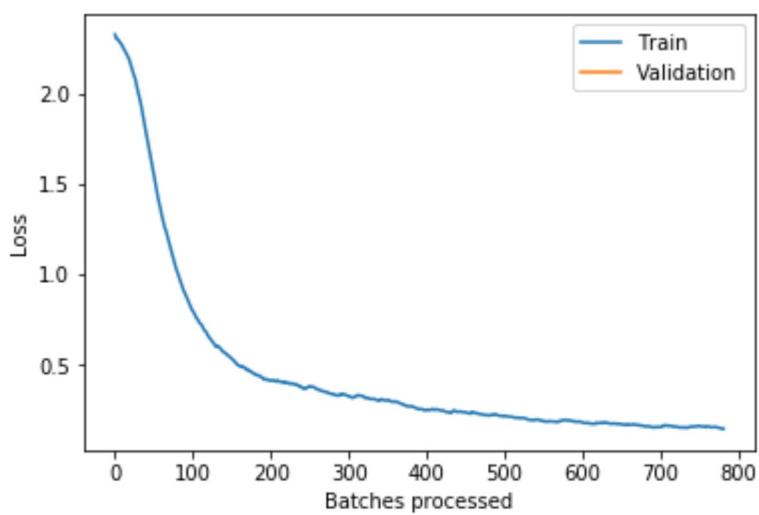
This was just last year's paper. Some of you may have seen [the interview with Leslie Smith](#) that I did last week. An amazing guy, incredibly humble and also I should say somebody who is doing groundbreaking research well into his 60's and all of these things are inspiring.

I'll show you something else interesting. When you plot the losses with fastai, it doesn't look like that:



It looks like that:

```
learn.recorder.plot_losses()
```



Why is that? Because fastai calculates the exponentially weighted moving average of the losses for you. So this concept of exponentially weighted stuff, it's just really handy and I use it all the time. And one of the things that is to make it easier to read these charts. It does mean that these charts from fastai might be a batch or two behind where they should be. There's that slight downside when you use an exponentially weighted moving average is you've got a little bit of history in there as well. But it can make it much easier to see what's going on.

↪ Back to Tabular [2:03:15]

[Notebook](#)

We're now at a point coming to the end of this collab and tabular section where we're going to try to understand all of the code in our tabular model. Remember, the tabular model uses this data set called adult which is trying to predict who's going to make more money. It's a classification problem and we've got a number of categorical variables and a number of continuous variables.

```
from fastai.tabular import *

path = untar_data(URLs.ADULT_SAMPLE)
df = pd.read_csv(path/'adult.csv')

dep_var = '>=50k'
cat_names = ['workclass', 'education', 'marital-status', 'occupation', 'relations']
cont_names = ['age', 'fnlwgt', 'education-num']
procs = [FillMissing, Categorify, Normalize]
```

The first thing we realize is we actually don't know how to predict a categorical variable yet. Because so far, we did some hand waving around the fact that our loss function was `nn.CrossEntropyLoss`. What is that? Let's find out. And of course we're going to find out by looking at [Microsoft Excel](#).

Cross-entropy loss is just another loss function. You already know one loss function which is mean squared error $(\hat{y} - y)^2$. That's not a good loss function for us because in our case we have, for MNIST, 10 possible digits and we have 10 activations each with a probability of that digit. So we need something where predicting the right thing correctly and confidently should have very little loss; predicting the wrong thing confidently should have a lot of loss. So that's what we want.

Here's an example:

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

Here is cat versus dog one hot encoded. Here are my two activations for each one from some model that I built - probability cat, probability dog. The first row is not very confident of anything. The second row is very confident of being a cat and that's right. The third row is very confident for being a cat and it's wrong. So we want a loss that for the first row should be a moderate loss because not predicting anything confidently is not really what we want, so here's 0.3. The second row is predicting the correct thing very confidently, so 0.01. The third row is predicting the wrong thing very confidently, so 1.0.

How do we do that? This is the cross entropy loss:

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

It is equal to whether it's a cat multiplied by the log of the cat activation, negative that, minus is it a dog times the log of the dog activation. That's it. So in other words, it's the sum of all of your one hot encoded variables times all of your activations.

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

Interestingly these ones here (column G) - exactly the same numbers as the column F, but I've written it differently. I've written it with an if function because the zeros don't actually add anything so actually it's exactly the same as saying if it's a cat, then take the log of cattiness and if it's a dog (i.e. otherwise) take the log of one minus cattiness (in other words, the log of dogginess). So the sum of the one hot encoded times the activations is the same as an `if` function. If you think about it, because this is just a matrix multiply, it is the same as an index lookup (as we now know from our embedding discussion). So to do cross entropy, you can also just look up the log of the activation for the correct answer.

Now that's only going to work if these rows add up to one. This is one reason that you can get screwy cross-entropy numbers is (that's why I said you press the wrong button) if they don't add up to 1 you've got a trouble. So how do you make sure that they add up to 1? You make sure they add up to 1 by using the correct activation function in your last layer. And the correct activation function to use for this is **softmax**. Softmax is an activation function where:

- all of the activations add up to 1
- all of the activations are greater than 0
- all of the activations are less than 1

So that's what we want. That's what we need. How do you do that? Let's say we were predicting one of five things: cat, dog, plane, fish, building, and these were the numbers that came out of our neural net for one set of predictions (output).

What if I did e to the power of that? That's one step in the right direction because e to the power of something is always bigger than zero so there's a bunch of numbers that are always bigger than zero. Here's the sum of those numbers (12.14). Here is e to the number divided by the sum of e to the number:

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

Now this number is always less than one because all of the things were positive so you can't possibly have one of the pieces be bigger than 100% of its sum. And all of those things must add up to 1 because each one of them was just that percentage of the total. That's it. So this thing `softmax` is equal to e to the activation divided by the sum of e to the activations. That's called softmax.

So when we're doing single label multi-class classification, you generally want softmax as your activation function and you generally want cross-entropy as your loss. Because these things go together in such friendly ways, PyTorch will do them both for you. So you might have noticed that in this MNIST example, I never added a softmax here:

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

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

That's because if you ask for cross entropy loss (`nn.CrossEntropyLoss`), it actually does the softmax inside the loss function. So it's not really just cross entropy loss, it's actually softmax then cross entropy loss.

So you've probably noticed this, but sometimes your predictions from your models will come out looking more like this:

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

Pretty big numbers with negatives in, rather than this (softmax column) - numbers between 0 to 1 that add up to 1. The reason would be that it's a PyTorch model that doesn't have a softmax in because we're using cross entropy loss and so you might have to do the softmax for it.

Fastai is getting increasingly good at knowing when this is happening. Generally if you're using a loss function that we recognize, when you get the predictions, we will try to add the softmax in there for you. But particularly if you're using a custom loss function that might call `nn.CrossEntropyLoss` behind the scenes or something like that, you might find yourself with this situation.

We only have 3 minutes less, but I'm going to point something out to you. Next week when we finish off tabular which we'll do in like the first 10 minutes, this is `forward` in tabular:

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

It basically goes through a bunch of embeddings. It's going to call each one of those embeddings `e` and you can use it like a function, of course. So it's going to pass each categorical variable to each embedding, it's going to concatenate them together into a single matrix. It's going to then call a bunch of layers which are basically a bunch of linear layers. And then it's going to do our sigmoid trick. There's only two new things we'll need to learn. One is dropout and the other is batch norm (`bn_cont`). These are two additional regularization strategies. BatchNorm does more than just regularization, but amongst other things it does regularization. And the basic ways you regularize your model are weight decay, batch norm, and dropout. Then you can also avoid overfitting using something called data augmentation. So batch norm and dropout, we're going to touch on at the start of next week. And we're also going to look at data augmentation and then we're also going to look at what convolutions are. And we're going to learn some new computer vision architectures and some new computer vision applications. But basically we're very nearly there. You already know how the entirety of `collab.py` (`fastai.collab`) works. You know why it's there and what it does and you're very close to knowing what the entirety of tabular model does. And this tabular model is actually the one that, if you run it on Rossmann, you'll get the same answer that I showed you in that paper. You'll get that second place result. In fact, even a little bit better. I'll show you next week (if I remember) how I actually ran some additional experiments where I figured out some minor tweaks that can do even slightly better than that. We'll see you next week. Thanks very much and enjoy the smoke outside.

 hiromis fixed typo

d0836b3 on Jan 7

1 contributor

1413 lines (937 sloc) 138 KB

Raw

Blame

History



⌚ Lesson 6

[Video / Course Forum](#)

Welcome to lesson 6 where we're going to do a deep dive into computer vision, convolutional neural networks, what is a convolution, and we're also going to learn the final regularization tricks after last lesson learning about weight decay/L2 regularization.

⌚ Platform.ai

I want to start by showing you something that I'm really excited about and I've had a small hand and helping to create. For those of you that saw [my talk on ted.com](#), you might have noticed this really interesting demo that we did about four years ago showing a way to quickly build models with unlabeled data. It's been four years but we're finally at a point where we're ready to put this out in the world and let people use it. And the first people we're going to let use it are you folks.

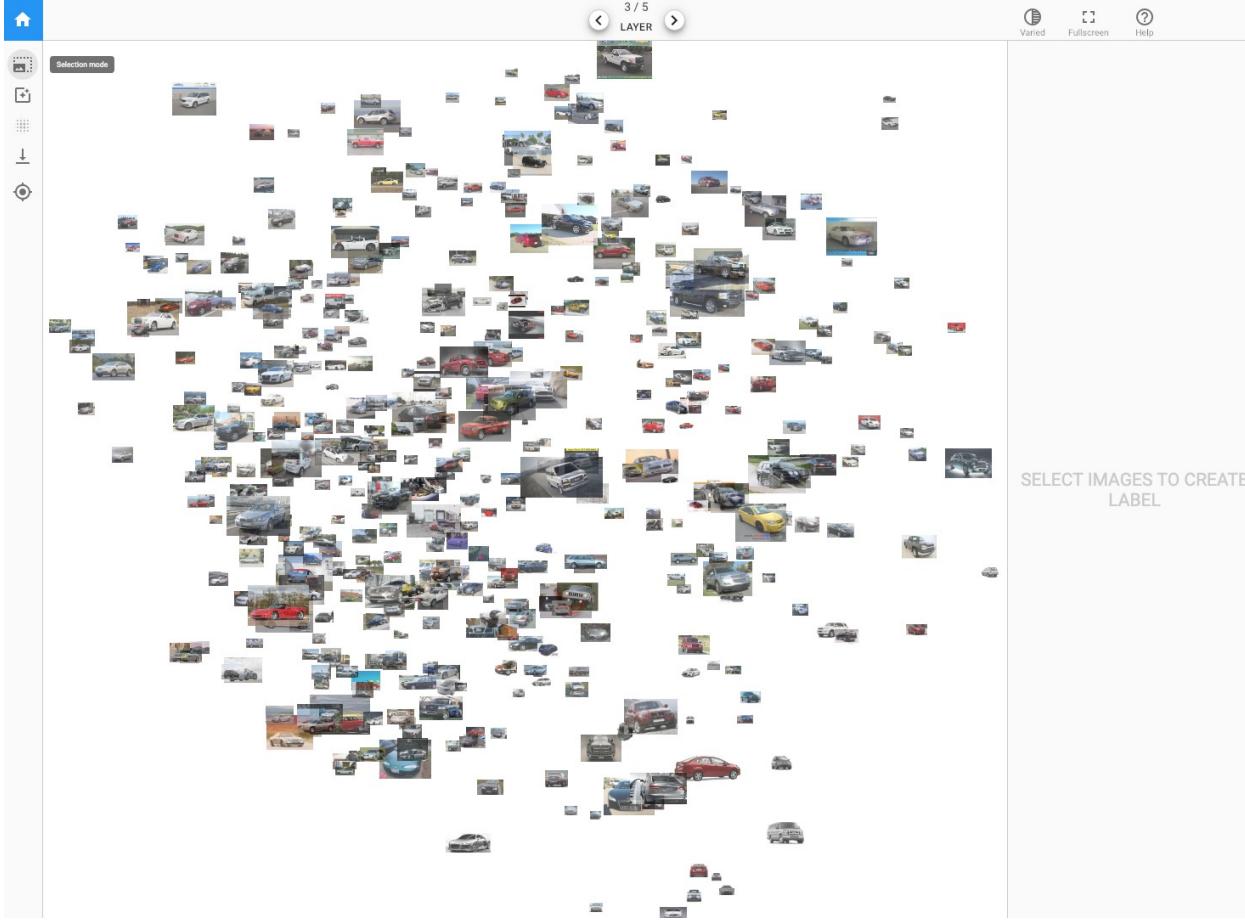
So the company is called [platform.ai](#) and the reason I'm mentioning it here is that it's going to let you create models on different types of datasets to what you can do now, that is to say datasets that you don't have labels for yet. We're actually going to help you label them. So this is the first time this has been shown before, so I'm pretty thrilled about it. Let me give you a quick demo.

If you'd go to [platform.ai](#) and choose "get started" you'll be able to create a new project. And if you create a new project you can either upload your own images. Uploading it at 500 or so works pretty well. You can upload a few thousand, but to start, upload 500 or so. They all have to be in a single folder. So we're assuming that you've got a whole bunch of images that you haven't got any labels for or you can start with one of the existing collections if you want to play around, so I've started with the cars collection kind of going back to what we did four years ago.

This is what happens when you first go into platform.ai and look at the collection of images you've uploaded - a random sample of them will appear on the screen. As you'll recognize, they are projected from a deep learning space into a 2D space using a pre-trained model. For this initial version, it's an ImageNet model we're using. As things move along, we'll be adding more and more pre train models. And what I'm going to do is I want to add labels to this data set representing which angle a photo of the car was taken from which is something that actually ImageNet is going to be really bad at because ImageNet has learnt to recognize the difference between cars versus bicycles and ImageNet knows that the angle you take a photo on actually doesn't matter. So we want to try and create labels using the kind of thing that actually ImageNet specifically learn to ignore.

So the projection that you see, we can click these layer buttons at the top to switch to user projection using a different layer of the neural net. Here's the last layer which is going to be a total waste of time for us because it's really going to be projecting things based on what kind of thing it thinks it is. The first layer is probably going to be a waste of time for us as well because there's very little interesting semantic content there. But if I go into the middle, in layer 3, we may well be able to find some differences there.

Then what you can do is you can click on the projection button here (you can actually just press up and down rather than just pressing the the arrows at the top) to switch between projections or left and right so switch between layers. And what you can do is you can basically look around until you notice that there's a projection which is kind of separated out things you're interested in. So this one actually I notice that it's got a whole bunch of cars that are from the front right over here. So if we zoom in a little bit, we can double check - "yeah that looks pretty good, they're all kind of front right." So we can click on here to go to selection mode, and we can grab a few, and then you should check:



What we're doing here is we're trying to take advantage of the combination of human plus machine. The machine is pretty good at quickly doing calculations, but as a human I'm pretty good at looking at a lot of things at once and seeing the odd one out. So in this case I'm looking for cars that aren't front right, and so by laying them in front of me, I can do that really quickly. It's like "okay definitely that one" so just click on the ones that you don't want. All right, it's all good.

Then you can just go back. Then what you can do is you can either put them into a new category by typing in "create a new label" or you can click on one of the existing ones. So before I came, I just created a few. So here's front right, so I just click on it here.

The basic idea is that you keep flicking through different layers or projections to try and find groups that represent the things you're interested in, and then over time you'll start to realize that there are some things that are a little bit harder. For example, I'm having trouble finding sides, so what I can do is I can see over here there's a few sides, so I can zoom in here and click on a couple of them. Then I'll say "find similar" and this is going to basically look in that projection space and not just at the images that are currently displayed but all of the images that you uploaded, and hopefully I might be able to label a few more side images at that point. It's going through and checking all of the images that you uploaded to see if any of them have projections in this space which are similar to the ones I've selected. Hopefully we'll find a few more of what I'm interested in.

Now if I want to try to find a projection that separates the sides from the front right, I can click on each of those two and then over here this button is now called "switch to the projection that maximizes the distance between the labels." What this is going to do is it's going to try and find the best projection that separates out those classes. The goal here is to help me visually inspect and quickly find a bunch of things that I can use to label.

They're the kind of the key features and it's done a good job. You can see down here, we've now got a whole bunch of sides which I can now grab because I was having a lot of trouble finding them before. And it's always worth double-checking. It's kind of interesting to see how the neural nets behave - like there seems to be more sports cars in this group than average as well. So it's kind of found side angles of sports cars, so that's kind of interesting. So I've got those, now I clicks "side" and there we go.

Once you've done that a few times, I find if you've got a hundred or so labels, you can then click on the train model button, and it'll take a couple of minutes, and come back and show you your train model. After it's trained, which I did it on a smaller number of labels earlier, you can then switch this vary opacity button, and it'll actually fade out the ones that are already predicted pretty well. It'll also give you a estimate as to how accurate it thinks the model is. The main reason I mentioned this for you is so that you can now click the download button and it'll download the predictions, which is what we hope will be interesting to most people. But what I think will be interesting to you as deep learning students is it'll download your labels. So now you can use that labeled subset of data along with the unlabeled set that you haven't labeled yet to see if you can build a better model than platform.ai has done for you. See if you can use that initial set of data to get going, creating models which you weren't able to label before.

Clearly, there are some things that this system is better at than others. For things that require really zooming in closely and taking a very very close inspection, this isn't going to work very well. This is really designed for things that the human eye can kind of pick up fairly readily. But we'd love to get feedback as well, and you can click on the Help button to give feedback. Also there's a [platform.ai discussion topic](#) in our forum. So Arshak if you can stand up, Arshak is the CEO of the company. He'll be there helping out answering questions and so forth. I hope people find that useful. It's been many years getting to this point, and I'm glad we're finally there.

⌚ Finishing up regularization for the Tabular Learner[9:48]

One of the reasons I wanted to mention this today is that we're going to be doing a big dive into convolutions later in this lesson. So I'm going to circle back to this to try and explain a little bit more about how that is working under the hood, and give you a kind of a sense of what's going on. But before we do, we have to finish off last week's discussion of regularization. We were talking about regularization specifically in the context of the tabular learner because the tabular learner, this is the init method in the tabular learner:

```

ps = ifnone(ps, [0]*len(layers))
ps = listify(ps, layers)
self.embeds = nn.ModuleList([embedding(ni, nf) for ni,nf in emb_szs])
self.emb_drop = nn.Dropout(emb_drop)
self.bn_cont = nn.BatchNorm1d(n_cont)
n_emb = sum(e.embedding_dim for e in self.embeds)
self.n_emb,self.n_cont,self.y_range = n_emb,n_cont,y_range
sizes = self.get_sizes(layers, out_sz)
actns = [nn.ReLU(inplace=True)] * (len(sizes)-2) + [None]
layers = []
for i,(n_in,n_out,dp,act) in enumerate(zip(sizes[:-1],sizes[1:], [0.]+ps,actns)):
    layers += bn_drop_lin(n_in, n_out, bn=use_bn and i!=0, p=dp, actn=act)
if bn_final: layers.append(nn.BatchNorm1d(sizes[-1]))
self.layers = nn.Sequential(*layers)

```

And our goal was to understand everything here, and we're not quite there yet. Last week we were looking at the adult data set which is a really simple (kind of over simple) data set that's just for toy purposes. So this week, let's look at a data set that's much more interesting - a Kaggle competition data set so we know what the best in the world and Kaggle competitions' results tend to be much harder to beat than academic state-of-the-art results tend to be because a lot more people work on Kaggle competitions than most academic data sets. So it's a really good challenge to try and do well on a Kaggle competition data set.

The rossmann data set is they've got 3,000 drug stores in Europe and you're trying to predict how many products they're going to sell in the next couple of weeks. One of the interesting things about this is that the test set for this is from a time period that is more recent than the training set. This is really common. If you want to predict things, there's no point predicting things that are in the middle of your training set. You want to predict things in the future.

Another interesting thing about it is the evaluation metric they provided is the root mean squared percent error.

$$\text{RMSPE} = \sqrt{\frac{1}{n} \sum_{i=1}^n \left(\frac{y_i - \hat{y}_i}{y_i} \right)^2}$$

This is just a normal root mean squared error except we go actual minus prediction divided by actual, so in other words it's the "percent" error that we're taking the root mean squared of. There's a couple of interesting features.

Always interesting to look at the leaderboard. So the leaderboard, the winner was 0.1. The paper that we've roughly replicated was 0.105 ~ 0.106, and the 10th place out of 3,000 was 0.11ish - a bit less.

We're gonna skip over a little bit. The data that was provided here was they provided a small number of files but they also let competitors provide additional external data as long as they shared it with all the competitors. So in practice the data set we're going to use contains six or seven tables. The way that you join tables and stuff isn't really part of a deep learning course. So I'm going to skip over it, and instead I'm going to refer you to [Introduction to Machine Learning for Coders](#) which will take you step-by-step through the data preparation for this. We've provided it for you in [rossman_data_clean.ipynb](#) so you'll see the whole process there. You'll need to run through that notebook to create these pickle files that we read here ([lesson6-rossmann.ipynb](#)):

```
%reload_ext autoreload
%autoreload 2

from fastai.tabular import *

path = Path('data/rossmann/')
train_df = pd.read_pickle(path/'train_clean')
```

⌚ Time Series and `add_datepart` [13:21]

I just want to mention one particularly interesting part of the rossmann data clean notebook which is you'll see there's something that says `add_datepart` and I wanted to explain what's going on here.

```
add_datepart(train, "Date", drop=False)
add_datepart(test, "Date", drop=False)
```

I've been mentioning for a while that we're going to look at time series. Pretty much everybody whom I've spoken to about it has assumed that I'm going to do some kind of recurrent neural network. But I'm not. Interestingly, the main academic group that studies time series is econometrics but they tend to study one very specific kind of time series which is where the only data you have is a sequence of time points of one thing. That's the only thing you have is one sequence. In real life, that's almost never the case. Normally, we would have some information about the store that represents or the people that it represents. We'd have metadata, we'd have sequences of other things measured at similar time periods or different time periods. So most of the time, I find in practice the state-of-the-art results when it comes to competitions on more real-world data sets don't tend to use recurrent neural networks. But instead, they tend to take the time piece which in this case it was a date we were given in the data, and they add a whole bunch of metadata. So in our case, for example, we've added day of week. We were given a date. We've added a day of week, year, month, week of year, day of month, day of week, day of year, and then a bunch of booleans is it at the month start/end, quarter year start/end, elapsed time since 1970, so forth.

If you run this one function `add_datepart` and pass it a date, it'll add all of these columns to your data set for you. What that means is that, let's take a very reasonable example. Purchasing behavior probably changes on payday. Payday might be the fifteenth of the month. So if you have a thing here called this is day of month, then it'll be able to recognize every time something is a fifteen there and associated it with a higher, in this case, embedding matrix value. Basically, we can't expect a neural net to do all of our feature engineering for us. We can expect it to find nonlinearities and interactions and stuff like that. But for something like taking a date like this (2015-07-31 00:00:00) and figuring out that the fifteenth of the month is something when interesting things happen. It's much better if we can provide that information for it.

So this is a really useful function to use. Once you've done this, you can treat many kinds of time-series problems as regular tabular problems. I say "many" kinds not "all". If there's very complex kind of state involved in a time series such as equity trading or something like that, this probably won't be the case or this won't be the only thing you need. But in this case, it'll get us a really good result and in practice, most of the time I find this works well.

Tabular data is normally in Pandas, so we just stored them as standard Python pickle files. We can read them in. We can take a look at the first five records.

```
train_df.head().T
```

	0	1	2
index	0	1	2

	0	1	2
Store	1	2	3
DayOfWeek	5	5	5
Date	2015-07-31 00:00:00	2015-07-31 00:00:00	2015-07-31 00:00:00
Sales	5263	6064	8314
Customers	555	625	821
Open	1	1	1
Promo	1	1	1
StateHoliday	False	False	False
SchoolHoliday	1	1	1
Year	2015	2015	2015
Month	7	7	7
Week	31	31	31
Day	31	31	31
Dayofweek	4	4	4
Dayofyear	212	212	212
Is_month_end	True	True	True
Is_month_start	False	False	False
Is_quarter_end	False	False	False
Is_quarter_start	False	False	False
Is_year_end	False	False	False
Is_year_start	False	False	False
Elapsed	1438300800	1438300800	1438300800
StoreType	c	a	a
Assortment	a	a	a
CompetitionDistance	1270	570	14130
CompetitionOpenSinceMonth	9	11	12
CompetitionOpenSinceYear	2008	2007	2006

	0	1	2
Promo2	0	1	1
Promo2SinceWeek	1	13	14
...
Min_Sea_Level_PressurehPa	1015	1017	1017
Max_VisibilityKm	31	10	31
Mean_VisibilityKm	15	10	14
Min_VisibilitykM	10	10	10
Max_Wind_SpeedKm_h	24	14	14
Mean_Wind_SpeedKm_h	11	11	5
Max_Gust_SpeedKm_h	NaN	NaN	NaN
Precipitationmm	0	0	0
CloudCover	1	4	2
Events	Fog	Fog	Fog
WindDirDegrees	13	309	354
StateName	Hessen	Thueringen	NordrheinWestfalen
CompetitionOpenSince	2008-09-15 00:00:00	2007-11-15 00:00:00	2006-12-15 00:00:00
CompetitionDaysOpen	2510	2815	3150
CompetitionMonthsOpen	24	24	24
Promo2Since	1900-01-01 00:00:00	2010-03-29 00:00:00	2011-04-04 00:00:00
Promo2Days	0	1950	1579
Promo2Weeks	0	25	25
AfterSchoolHoliday	0	0	0
BeforeSchoolHoliday	0	0	0
AfterStateHoliday	57	67	57
BeforeStateHoliday	0	0	0
AfterPromo	0	0	0

	0	1	2
BeforePromo	0	0	0
SchoolHoliday_bw	5	5	5
StateHoliday_bw	0	0	0
Promo_bw	5	5	5
SchoolHoliday_fw	7	1	5
StateHoliday_fw	0	0	0
Promo_fw	5	1	5

93 rows × 5 columns

The key thing here is that we're trying to predict a particular date for a particular store ID, we want to predict the number of sales. Sales is the dependent variable.

⌚ Preprocesses [16:52]

The first thing I'm going to show you is something called pre-processes. You've already learned about transforms. **Transforms** are bits of code that run every time something is grabbed from a data set so it's really good for data augmentation that we'll learn about today, which is that it's going to get a different random value every time it's sampled. **Preprocesses** are like transforms, but they're a little bit different which is that they run once before you do any training. Really importantly, they run once on the training set and then any kind of state or metadata that's created is then shared with the validation and test set.

Let me give you an example. When we've been doing image recognition and we've had a set of classes to all the different pet breeds and they've been turned into numbers. The thing that's actually doing that for us is a preprocessor that's being created in the background. That makes sure that the classes for the training set are the same as the classes for the validation and the classes of the test set. So we're going to do something very similar here. For example, if we create a little small subset of a data for playing with. This is a really good idea when you start with a new data set.

```
idx = np.random.permutation(range(n))[:2000]
idx.sort()
small_train_df = train_df.iloc[idx[:1000]]
small_test_df = train_df.iloc[idx[1000:]]
small_cont_vars = ['CompetitionDistance', 'Mean_Humidity']
small_cat_vars = ['Store', 'DayOfWeek', 'PromoInterval']
small_train_df = small_train_df[small_cat_vars + small_cont_vars + ['Sales']]
small_test_df = small_test_df[small_cat_vars + small_cont_vars + ['Sales']]
```

I've just grabbed 2,000 IDs at random. Then I'm just going to grab a little training set and a little test set - half and half of those 2,000 IDs, and it's going to grab five columns. Then we can just play around with this. Nice and easy. Here's the first few of those from the training set:

```
small_train_df.head()
```

	Store	DayOfWeek	PromoInterval	CompetitionDistance	Mean_Hu
280	281	5	NaN	6970.0	61
584	586	5	NaN	250.0	61
588	590	5	Jan,Apr,Jul,Oct	4520.0	51
847	849	5	NaN	5000.0	67
896	899	5	Jan,Apr,Jul,Oct	2590.0	55

You can see, one of them is called promo interval and it has these strings, and sometimes it's missing. In Pandas, missing is `NaN`.

⌚ Preprocessor: Categorify [18:39]

The first preprocessor I'll show you is Categorify.

```
categorify = Categorify(small_cat_vars, small_cont_vars)
categorify(small_train_df)
categorify(small_test_df, test=True)
```

Categorify does basically the same thing that `.classes` thing for image recognition does for a dependent variable. It's going to take these strings, it's going to find all of the possible unique values of it, and it's going to create a list of them, and then it's going to turn the strings into numbers. So if I call it on my training set, that'll create categories there (`small_train_df`) and then I call it on my test set passing in `test=True`, that makes sure it's going to use the same categories that I had before. Now when I say `.head`, it looks exactly the same:

```
small_test_df.head()
```

	Store	DayOfWeek	PromoInterval	CompetitionDistance	Me
428412	NaN	2	NaN	840.0	89
428541	1050.0	2	Mar,Jun,Sept,Dec	13170.0	78

	Store	DayOfWeek	PromoInterval	CompetitionDistance	Mo
428813	NaN	1	Jan,Apr,Jul,Oct	11680.0	85
430157	414.0	6	Jan,Apr,Jul,Oct	6210.0	88
431137	285.0	5	NaN	2410.0	57

That's because Pandas has turned this into a categorical variable which internally is storing numbers but externally is showing me the strings. But I can look inside promo interval to look at the `cat.categories`, this is all standard Pandas here, to show me a list of all of what we would call "classes" in fast.ai or would be called just "categories" in Pandas.

```
small_train_df.PromoInterval.cat.categories
```

```
Index(['Feb,May,Aug,Nov', 'Jan,Apr,Jul,Oct', 'Mar,Jun,Sept,Dec'],
      dtype='object')
```

```
small_train_df['PromoInterval'].cat.codes[:5]
```

```
280    -1
584    -1
588     1
847    -1
896     1
dtype: int8
```

So then if I look at the `cat.codes`, you can see here this list here is the numbers that are actually stored (-1, -1, 1, -1, 1). What are these minus ones? The minus ones represent `NaN` - they represent "missing". So Pandas uses the special code `-1` to be mean missing.

As you know, these are going to end up in an embedding matrix, and we can't look up item -1 in an embedding matrix. So internally in fast.ai, we add one to all of these.

⌚ Preprocessor: Fill Missing [20:18]

Another useful preprocessor is `FillMissing`. Again, you can call it on the data frame, you can call on the test passing in `test=true`.

```
fill_missing = FillMissing(small_cat_vars, small_cont_vars)
fill_missing(small_train_df)
```

```
fill_missing(small_test_df, test=True)
```

```
small_train_df[small_train_df['CompetitionDistance_na'] == True]
```

	Store	DayOfWeek	PromoInterval	CompetitionDistance	Me
78375	622	5	NaN	2380.0	71
161185	622	6	NaN	2380.0	91
363369	879	4	Feb,May,Aug,Nov	2380.0	73

This will create, for anything that has a missing value, it'll create an additional column with the column name underscore na (e.g. `CompetitionDistance_na`) and it will set it for true for any time that was missing. Then what we do is, we replace competition distance with the median for those. Why do we do this? Well, because very commonly the fact that something's missing is of itself interesting (i.e. it turns out the fact that this is missing helps you predict your outcome). So we certainly want to keep that information in a convenient boolean column, so that our deep learning model can use it to predict things.

But then, we need competition distance to be a continuous variable so we can use it in the continuous variable part of our model. So we can replace it with almost any number because if it turns out that the missingness is important, it can use the interaction of `CompetitionDistance_na` and `CompetitionDistance` to make predictions. So that's what `FillMissing` does.

[21:31]

You don't have to manually call preprocesses yourself. When you call any kind of item list creator, you can pass in a list of pre processes which you can create like this:

```
procs=[FillMissing, Categorify, Normalize]
```

```
data = (TabularList.from_df(df, path=path, cat_names=cat_vars, cont_names=cont_va
                            .split_by_idx(valid_idx)
                            .label_from_df(cols=dep_var, label_cls=FloatList, log=True)
                            .databunch())
```

This is saying "ok, I want to fill missing, I want to categorify, I want to normalize (i.e. for continuous variables, it'll subtract the mean and divide by the standard deviation to help a train more easily)." So you just say, those are my procs and then you can just pass it in there and that's it.

Later on, you can go `data.export` and it'll save all the metadata for that data bunch so you can, later on, load it in knowing exactly what your category codes are, exactly what median values used for replacing the missing values, and exactly what means and standard deviations you normalize by.

⌚ Categorical and Continuous Variables [22:23]

The main thing you have to do if you want to create a data bunch of tabular data is tell it what are your categorical variables and what are your continuous variables. As we discussed last week briefly, your categorical variables are not just strings and things, but also I include things like day of week and month and day of month. Even though they're numbers, I make them categorical variables. Because, for example, day of month, I don't think it's going to have a nice smooth curve. I think that the fifteenth of the month and the first of the month and the 30th of the month are probably going to have different purchasing behavior to other days of the month. Therefore, if I make it a categorical variable, it's going to end up creating an embedding matrix and those different days of the month can get different behaviors.

You've actually got to think carefully about which things should be categorical variables. On the whole, if in doubt and there are not too many levels in your category (that's called the **cardinality**), if your cardinality is not too high, I would put it as a categorical variable. You can always try an each and see which works best.

```
cat_vars = ['Store', 'DayOfWeek', 'Year', 'Month', 'Day', 'StateHoliday',
            'CompetitionMonthsOpen', 'Promo2Weeks', 'StoreType', 'Assortment',
            'PromoInterval', 'CompetitionOpenSinceYear', 'Promo2SinceYear', 'Stat
            'Week', 'Events', 'Promo_fw', 'Promo_bw', 'StateHoliday_fw',
            'StateHoliday_bw', 'SchoolHoliday_fw', 'SchoolHoliday_bw']

cont_vars = ['CompetitionDistance', 'Max_TemperatureC', 'Mean_TemperatureC',
            'Min_TemperatureC', 'Max_Humidity', 'Mean_Humidity', 'Min_Humidity',
            'Max_Wind_SpeedKm_h', 'Mean_Wind_SpeedKm_h', 'CloudCover', 'trend',
            'trend_DE', 'AfterStateHoliday', 'BeforeStateHoliday', 'Promo',
            'SchoolHoliday']
```

Our final data frame that we're going to pass in is going to be a training set with the categorical variables, the continuous variables, the dependent variable, and the date. The date, we're just going to use to create a validation set where we are basically going to say the validation set is going to be the same number of records at the end of the time period that the test set is for Kaggle. That way, we should be able to validate our model nicely.

```
dep_var = 'Sales'
df = train_df[cat_vars + cont_vars + [dep_var, 'Date']].copy()
```

```
test_df['Date'].min(), test_df['Date'].max()

(Timestamp('2015-08-01 00:00:00'), Timestamp('2015-09-17 00:00:00'))
```



```
cut = train_df['Date'][((train_df['Date'] == train_df['Date'][len(test_df)])].index[0]
cut
```

41395

```
valid_idx = range(cut)
```

```
df[dep_var].head()
```

```
0    5263
1    6064
2    8314
3   13995
4    4822
Name: Sales, dtype: int64
```

Now we can create a tabular list.

```
data = (TabularList.from_df(df, path=path, cat_names=cat_vars, cont_names=cont_vars)
        .split_by_idx(valid_idx)
        .label_from_df(cols=dep_var, label_cls=FloatList, log=True)
        .databunch())
```

This is our standard data block API that you've seen a few times:

- From a data frame, passing all of that information.
- Split it into valid vs. train.
- Label it with a dependent variable.

Here's something I don't think you've seen before - label class (`label_cls=FloatList`). This is our dependent variable (`df[dep_var].head()` above), and as you can see, this is sales. It's not a float. It's `int64`. **If this was a float, then fast.ai would automatically guess that you want to do a regression.** But this is not a float, it's an int. So fast.ai is going to assume you want to do a classification. So when we label it, we have to tell it that the class of the labels we want is a list of floats, not a list of categories (which would otherwise be the default). **So this is the thing that's going to automatically turn this into a regression problem for us.** Then we create a data bunch.

⌚ Reminder about Doc [25:09]

```
doc(FloatList)
```

I wanted to remind you again about `doc` which is how we find out more information about this stuff. In this case, all of the labeling functions in the data blocks API will pass on any keywords they don't recognize to the label class. So one of the things I've passed in here is `log` and so that's actually going to end up in `FloatList` and so if I go `doc(FloatList)`, I can see a summary:

The screenshot shows a Jupyter Notebook cell with the following content:

```
In [31]: doc(FloatList)
```

Below the cell, the documentation for `FloatList` is displayed:

Model

```
In [29]: max_log_y = np.log(np.max(train_df['Sales'])*1.2)
y_range = torch.tensor([0, max_log_y], device=defaults.device)
```

class FloatList [source]

```
FloatList(items: Iterator, log: bool = False, kwargs) :: ItemList
```

ItemList suitable for storing the floats in items for regression. Will add a `log` if True
Show in docs

And I can even jump into the full documentation, and it shows me here that `log` is something which if true, it's going to take the logarithm of my dependent variable. Why am I doing that? So this is the thing that's actually going to automatically take the log of my y . The reason I'm doing that is because as I mentioned before, the evaluation metric is root mean squared percentage error.

$$\text{RMSPE} = \sqrt{\frac{1}{n} \sum_{i=1}^n \left(\frac{y_i - \hat{y}_i}{y_i} \right)^2}$$

Neither fast.ai nor PyTorch has a root mean squared percentage error loss function built-in. I don't even know if such a loss function would work super well. But if you want to spend the time thinking about it, you'll notice that this ratio if you first take the log of y and \hat{y} , then becomes a difference rather than the ratio. In other words, if you take the log of y then RMSPE becomes root mean squared error. So that's what we're going to do. We're going to take the log of y and then we're just going to use root mean square error which is the default for a regression problems we won't even have to mention it.

```
In [24]: data = (TabularList.from_df(df, path=path, cat_names=cat_vars, cont_names=cont_vars, procs=procs)
               .split_by_idx(valid_idx)
               .label_from_df(cols=dep_var, label_cls=FloatList, log=True)
               .databunch())
```

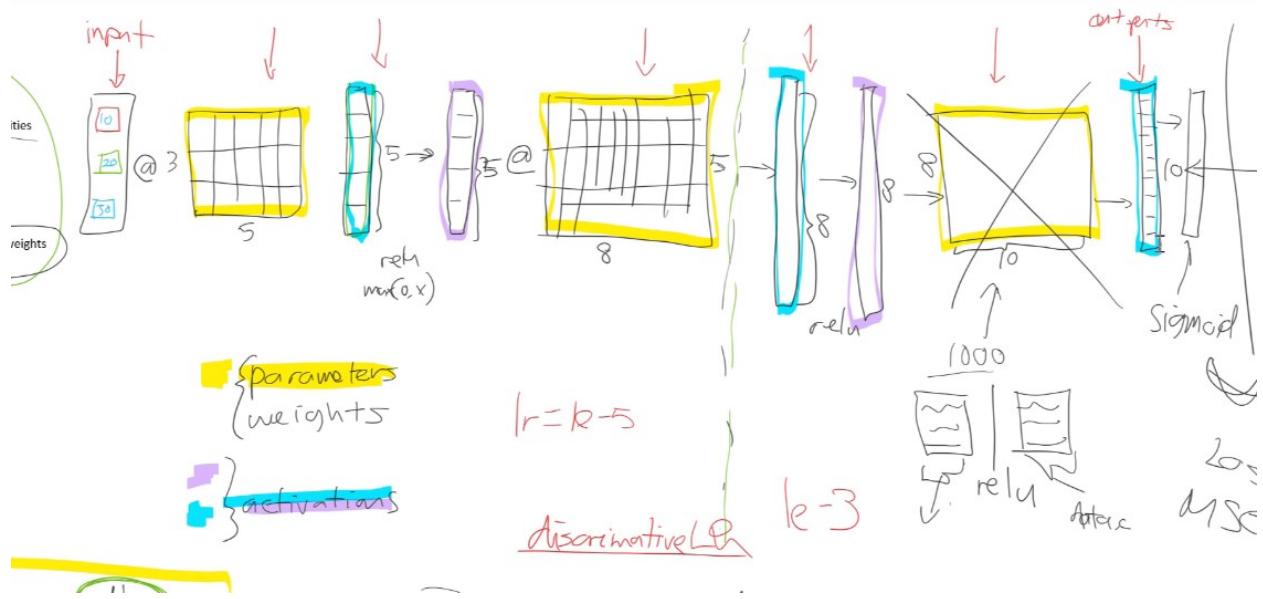
The reason that we have this (`log=True`) here is because this is so common. Basically anytime you're trying to predict something like a population or a dollar amount of sales, these kind of things tend to have long tail distributions where you care more about percentage differences and exact/absolute differences. So you're very likely to want to do things with `log=True` and to measure the root mean squared percent error.

⌚ `y_range` [27:12]

```
max_log_y = np.log(np.max(train_df['Sales'])*1.2)
y_range = torch.tensor([0, max_log_y], device=defaults.device)
```

We've learned about the `y_range` before which is going to use that sigmoid to help us get in the right range. Because this time the y values are going to be taken the log of it first, we need to make sure that the `y_range` we want is also the log. So I'm going to take the maximum of the sales column. I'm going to multiply it by a little bit because remember how we said it's nice if your range is a bit wider than the range of the data. Then we're going to take the log. That's going to be our maximum. Then our `y_range` will be from zero to a bit more than the maximum.

Now we've got our data bunch, we can create a tabular learner from it. Then we have to pass in our architecture. As we briefly discussed, for a tabular model, our architecture is literally the most basic fully connected network - just like we showed in this picture:



It's an input, matrix multiply, non-linearity, matrix multiply, non-linearity, matrix multiply, non-linearity, done. What are the interesting things about this is that this competition is three years old, but I'm not aware of any significant advances at least in terms of architecture that would cause me to choose something different to what the third-placed folks did three years ago. We're still basically using simple fully connected models for this problem.

```
learn = tabular_learner(data, layers=[1000,500], ps=[0.001,0.01], emb_drop=0.04,
                        y_range=y_range, metrics=exp_rmspe)
```

Now the intermediate weight matrix is going to have to go from a 1000 activation input to a 500 activation output, which means it's going to have to be 500,000 elements in that weight matrix. That's an awful lot for a data set with only a few hundred thousand rows. So this is going to overfit, and we need to make sure it doesn't. The way to make sure it doesn't is to **use regularization; not to reduce the number of parameters**. So one way to do that will be to use weight decay which fast.ai will use automatically, and you can vary it to something other than the default if you wish. It turns out in this case, we're going to want more regularization. So we're going to pass in something called `ps`. This is going to provide dropout. And also this one here `emb_drop` - this is going to provide embedding dropout.

Dropout [29:47]

Let's learn about what is dropout. The short version is dropout is a kind of regularization. This is [the dropout paper](#) Nitish Srivastava it was Srivastava's master's thesis under Geoffrey Hinton.

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava
Geoffrey Hinton
Alex Krizhevsky
Ilya Sutskever
Ruslan Salakhutdinov

NITISH@CS.TORONTO.EDU
HINTON@CS.TORONTO.EDU
KRIZ@CS.TORONTO.EDU
ILYA@CS.TORONTO.EDU
RSALAKHU@CS.TORONTO.EDU

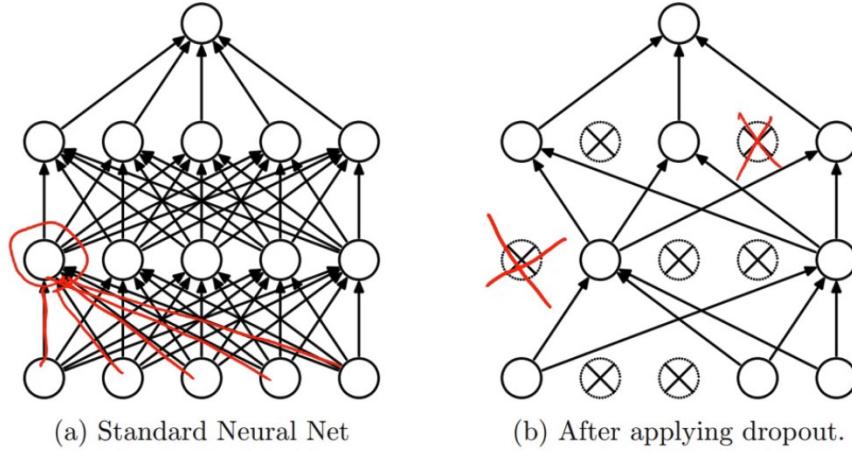


Figure 1: Dropout Neural Net Model. **Left:** A standard neural net with 2 hidden layers. **Right:** An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

This picture from the original paper is a really good picture of what's going on. This first picture is a picture of a standard fully connected network and what each line shows is a multiplication of an activation times a weight. Then when you've got multiple arrows coming in, that represents a sum. So this activation here (circled in red) is the sum of all of these inputs times all of these activations. So that's what a normal fully connected neural net looks like.

For dropout, we throw that away. At random, we **throw away some percentage of the activations** not the weights, not the parameters. Remember, there's **only two types of number in a neural net - parameters** also called weights (kind of) and **activations**. So we're going to throw away some activations.

So you can see that when we throw away this activation, all of the things that were connected to it are gone too. For each mini batch, we throw away a different subset of activations. How many do we throw away? We throw each one away with a probability p . A common value of p is 0.5. So what does that mean? And you'll see in this case, not only have they deleted at random some of these hidden layers, but they've actually deleted some of the inputs as well. Deleting the inputs is pretty unusual. Normally, we only delete activations in the hidden layers. So what does this do? Well, every time I have a mini batch going through, I, at random, throw away some of the activations. And then the next mini batch, I put them back and I throw away some different ones.

It means that no 1 activation can memorize some part of the input because that's what happens if we over fit. If we over fit, some part of the model is basically learning to recognize a particular image rather than a feature in general or a particular item. With dropout, it's going to be very hard for it to do that. In fact, Geoffrey Hinton described part of the thinking behind this as follows:

I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting.

Hinton: Reddit AMA

He noticed every time he went to his bank that all the tellers and staff moved around, and he realized the reason for this must be that they're trying to avoid fraud. If they keep moving them around, nobody can specialize so much in that one thing that they're doing that they can figure out a conspiracy to defraud the bank. Now, of course, depends when you ask Hinton. At other times he says that the reason for this was because he thought about how spiking neurons work and he's a neuroscientist by training:

We don't really know why neurons spike. One theory is that they want to be noisy so as to regularize, because we have many more parameters than we have data points. The idea of dropout is that if you have noisy activations, you can afford to use a much bigger model.

Hinton: O'Reilly

There's a view that spiking neurons might help regularization, and dropout is a way of matching this idea of spiking neurons. It's interesting. When you actually ask people where did your idea for some algorithm come from, it basically never comes from math; it always comes from intuition and thinking about physical analogies and stuff like that.

Anyway the truth is a bunch of ideas I guess all flowing around and they came up with this idea of dropout. But the important thing to know is it worked really really well. So we can use it in our models to get generalization for free.

Now too much dropout, of course, is reducing the capacity of your model, so it's going to under fit. So you've got to play around with different dropout values for each of your layers to decide.

In pretty much every fast.ai learner, there's a parameter called `ps` which will be the p-value for the dropout for each layer. So you can just pass in a list, or you can pass it an int and it'll create a list with that value everywhere. Sometimes it's a little different. For CNN, for example, if you pass in an int, it will use that for the last layer, and half that value for the earlier layers. We basically try to do things represent best practice. But you can always pass in your own list to get exactly the dropout that you want.

⌚ Dropout and test time [34:47]

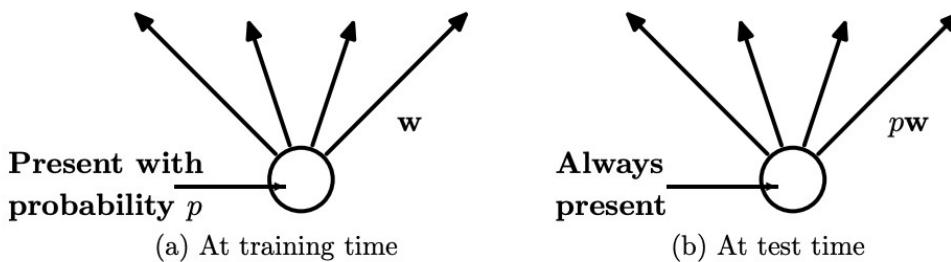


Figure 2: **Left:** A unit at training time that is present with probability p and is connected to units in the next layer with weights w . **Right:** At test time, the unit is always present and the weights are multiplied by p . The output at test time is same as the expected output at training time.

There is an interesting feature of dropout. We talk about training time and test time (we also call inference time). Training time is when we're actually doing that those weight updates - the backpropagation. The training time, dropout works the way we just saw. At test time we turn off dropout. We're not going to do dropout anymore because we wanted to be as accurate as possible. We're not training so we can't cause it to overfit when we're doing inference. So we remove dropout. But what that means is if previously p was 0.5, then half the activations were being removed. Which means when they're all there, now our overall activation level is twice of what it used to be. Therefore, in the paper, they suggest multiplying all of your weights at test time by p .

Interestingly, you can dig into the PyTorch source code and you can find the actual C code where dropout is implemented.

```
noise.bernoulli_(1 - p);
noise.div_(1 - p);
return multiply<inplace>(input, noise);
```

And you can see what they're doing is something quite interesting. They first of all do a Bernoulli trial. So a Bernoulli trial is with probability $1 - p$, return the value 1 otherwise return the value 0. That's all it means. In this case, p is the probability of dropout, so $1 - p$ is a probability that we keep the activation. So we end up here with either a 1 or a 0. Then (this is interesting) we divide in place (remember underscore means "in place" in PyTorch) we divide in place that 1 or 0 by $1 - p$. If it's a 0 nothing happens it's still 0. If it's a 1 and p was 0.5, that one now becomes 2. Then finally, we multiply in place our input by this noise (i.e. this dropout mask).

So in other words, in PyTorch, we don't do the change at test time. We actually do the change at training time - which means that you don't have to do anything special at inference time with PyTorch. It's not just PyTorch, it's quite a common pattern. But it's kind of nice to look inside the PyTorch source code and see dropout; this incredibly cool, incredibly valuable thing, is really just these three lines of code which they do in C because I guess it ends up a bit faster when it's all fused together. But lots of libraries do it in Python and that works well as well. You can even write your own dropout layer, and it should give exactly the same results as this. That'd be a good exercise to try. See if you can create your own dropout layer in Python, and see if you can replicate the results that we get with this dropout layer.

[37:38]

```
learn = tabular_learner(data, layers=[1000,500], ps=[0.001,0.01], emb_drop=0.04,  
                        y_range=y_range, metrics=exp_rmspe)
```

So that's dropout. In this case, we're going to use a tiny bit of dropout on the first layer (0.001) and a little bit of dropout on the next layer (0.01), and then we're going to use special dropout on the embedding layer. Now why do we do special dropout on the embedding layer? If you look inside the fast.ai source code, here is our tabular model:

```

class TabularModel(nn.Module):
    "Basic model for tabular data."
    def __init__(self, emb_szs:ListSizes, n_cont:int, out_sz:int, layers:Collection[int], ps:Collection[float]=None,
                 emb_drop:float=0., y_range:OptRange=None, use_bn:bool=True, bn_final:bool=False):
        super().__init__()
        ps = ifnone(ps, [0]*len(layers))
        ps = listify(ps, layers)
        self.embeds = nn.ModuleList([embedding(ni, nf) for ni,nf in emb_szs])
        self.emb_drop = nn.Dropout(emb_drop)
        self.bn_cont = nn.BatchNorm1d(n_cont)
        n_emb = sum(e.embedding_dim for e in self.embeds)
        self.n_emb,self.n_cont,y_range = n_emb,n_cont,y_range
        sizes = self.get_sizes(layers, out_sz)
        actns = [nn.ReLU(inplace=True)] * (len(sizes)-2) + [None]
        layers = []
        for i,(n_in,n_out,dp,act) in enumerate(zip(sizes[:-1],sizes[1:],ps,actns)):
            layers += bn_drop_lin(n_in, n_out, bn=use_bn and i!=0, p=dp, actn=act)
        if bn_final: layers.append(nn.BatchNorm1d(sizes[-1]))
        self.layers = nn.Sequential(*layers)

    def get_sizes(self, layers, out_sz):
        return [self.n_emb + self.n_cont] + layers + [out_sz]

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

```

You'll see that in the section that checks that there's some embeddings (`if self.n_emb != 0: in forward`),

- we call each embedding
- we concatenate the embeddings into a single matrix
- then we call embedding dropout

An embedding dropout is simply just a dropout. So it's just an instance of a dropout module. This kind of makes sense, right? For continuous variables, that continuous variable is just in one column. You wouldn't want to do dropout on that because you're literally deleting the existence of that whole input which is almost certainly not what you want. But for an embedding, and embedding is just effectively a matrix multiplied by a one hot encoded matrix, so it's just another layer. So it makes perfect sense to have dropout on the output of the embedding, because you're putting dropout on those activations of that layer. So you're basically saying let's delete at random some of the results of that embedding (i.e. some of those activations). So that makes sense.

The other reason we do it that way is because I did very extensive experiments about a year ago where on this data set I tried lots of different ways of doing kind of everything. And you can actually see it here:

Avg Score	Column L	No scale	No scale Total	Scaled	Scaled Total	Grand Total
Row Labels	Eq	Dict		Eq	Dict	
■ No init	0.0088	0.0087	0.0087	0.0089	0.0089	0.0088
Dense	0.0087	0.0088	0.0087	0.0090	0.0088	0.0089
Split	0.0089	0.0086	0.0087	0.0088	0.0090	0.0089
■ Init	0.0086	0.0090	0.0088	0.0086	0.0087	0.0087
Dense	0.0085	0.0090	0.0087	0.0086	0.0087	0.0086
Split	0.0088	0.0089	0.0089	0.0087	0.0087	0.0087
Grand Total	0.0087	0.0088	0.0088	0.0088	0.0088	0.0088

I put it all in a spreadsheet (of course Microsoft Excel), put them into a pivot table to summarize them all together to find out which different choices, hyper parameters, and architectures worked well and worked less well. Then I created all these little graphs:



These are like little summary training graphs for different combinations of high parameters and architectures. And I found that there was one of them which ended up consistently getting a good predictive accuracy, the bumpiness of the training was pretty low, and you can see, it was just a nice smooth curve.

This is an example of the experiments that I do that end up in the fastai library. So embedding dropout was one of those things that I just found work really well. Basically the results of these experiments is why it looks like this rather than something else. Well, it's a combination of these experiments but then why did I do these particular experiments? Well because it was very influenced by what worked well in that Kaggle prize winner's paper. But there are quite a few parts of that paper I thought "there were some other choices they could have made, I wonder why they didn't" and I tried them out and found out what actually works and what doesn't work as well, and found a few little improvements. So that's the kind of experiments that you can play around with as well when you try different models and architectures; different dropouts, layer numbers, number of activations, and so forth.

[41:02]

Having created our learner, we can type `learn.model` to take a look at it:

```
learn.model
```

```
TabularModel(  
    (embeds): ModuleList(  
        (0): Embedding(1116, 50)  
        (1): Embedding(8, 5)  
        (2): Embedding(4, 3)  
        (3): Embedding(13, 7)  
        (4): Embedding(32, 17)  
        (5): Embedding(3, 2)  
        (6): Embedding(26, 14)  
        (7): Embedding(27, 14)  
        (8): Embedding(5, 3)  
        (9): Embedding(4, 3)  
        (10): Embedding(4, 3)  
        (11): Embedding(24, 13)  
        (12): Embedding(9, 5)  
        (13): Embedding(13, 7)  
        (14): Embedding(53, 27)  
        (15): Embedding(22, 12)  
        (16): Embedding(7, 4)  
        (17): Embedding(7, 4)  
        (18): Embedding(4, 3)  
        (19): Embedding(4, 3)  
        (20): Embedding(9, 5)  
        (21): Embedding(9, 5)  
        (22): Embedding(3, 2)  
        (23): Embedding(3, 2)  
    )  
    (emb_drop): Dropout(p=0.04)  
    (bn_cont): BatchNorm1d(16, eps=1e-05, momentum=0.1, affine=True,  
    track_running_stats=True)
```

```

(layers): Sequential(
    (0): Linear(in_features=229, out_features=1000, bias=True)
    (1): ReLU(inplace)
    (2): BatchNorm1d(1000, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (3): Dropout(p=0.001)
    (4): Linear(in_features=1000, out_features=500, bias=True)
    (5): ReLU(inplace)
    (6): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (7): Dropout(p=0.01)
    (8): Linear(in_features=500, out_features=1, bias=True)
)
)

```

As you would expect, in that, there is a whole bunch of embeddings. Each of those embedding matrices tells you the number of levels for each input (the first number). You can match these with your list `cat_vars`. So the first one will be `store`, so that's not surprising there are 1,116 stores. Then the second number, of course, is the size of the embedding. That's a number that you get to choose.

Fast.ai has some defaults which actually work really really well nearly all the time. So I almost never changed them. But when you create your `tabular_lerner`, you can absolutely pass in an embedding size dictionary which maps variable names to embedding sizes for anything where you want to override the defaults.

Then we've got our embedding dropout layer, and then we've got a batch norm layer with 16 inputs. The 16 inputs make sense because we have 16 continuous variables.

```
len(data.train_ds.cont_names)
```

16

The length of `cont_names` is 16. So this is something for our continuous variables. Specifically, it's over here, `bn_cont` on our continuous variables:

```

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

```

And `bn_cont` is a `nn.BatchNorm1d`. What's that? The first short answer is it's one of the things that I experimented with as to having batchnorm not, and I found that it worked really well. Then specifically what it is is extremely unclear. Let me describe it to you.

- It's kind of a bit of regularization
- It's kind of a bit of training helper

It's called batch normalization and it comes from [this paper](#).

[43:06]

Actually before I do this, I just want to mention one other really funny thing dropout. I mentioned it was a master's thesis. Not only was it a master's thesis, one of the most influential papers of the last ten years.



Chris Gorgolewski

@ChrisFiloG

Follow



Did you know that Dropout was originally introduced in a Master's thesis and was rejected from NIPS? Was disseminated via #arxiv! #OHBM2018

The slide has a yellow-to-green gradient background. At the top left, the word "Dropout" is written in white. Below it, there is a block of text and a bulleted list. At the bottom, there is another text block.

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

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

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

It was rejected from the main neural nets conference what was then called NIPS, now called NeurIPS. I think it's very interesting because it's just a reminder that our academic community is generally extremely poor at recognizing which things are going to turn out to be important. Generally, people are looking for stuff that are in the field that they're working on and understand. So dropout kind of came out of left field. It's kind of hard to understand what's going on. So that's kind of interesting.

It's a reminder that if you just follow as you develop beyond being just a practitioner into actually doing your own research, don't just focus on the stuff everybody's talking about. Focus on the stuff you think might be interesting. Because the stuff everybody's talking about generally turns out not to be very interesting. The community is very poor at recognizing high-impact papers when they come out.

⌚ Batch Normalization [44:28]

Batch normalization, on the other hand, was immediately recognized as high-impact. I definitely remember everybody talking about it in 2015 when it came out. That was because it's so obvious, they showed this picture:

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., sioffe@google.com

Christian Szegedy
Google Inc., szegedy@google.com

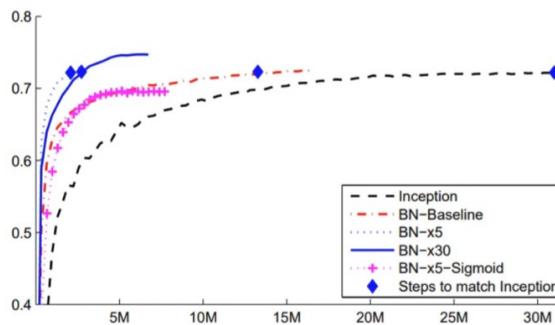


Figure 2: Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

```


$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$


$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$$


$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$


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


```

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Showing the current then state of the art ImageNet model Inception. This is how long it took them to get a pretty good result, and then they tried the same thing with this new thing called batch norm, and they just did it way way way quickly. That was enough for pretty much everybody to go "wow, this is interesting."

Specifically they said this thing is called batch normalization and it's accelerating training by reducing internal covariate shift. So what is internal covariate shift? Well, it doesn't matter. Because this is one of those things where researchers came up with some intuition and some idea about this thing they wanted to try. They did it, it worked well, they then post hoc added on some mathematical analysis to try and claim why it worked. And it turned out they were totally wrong.

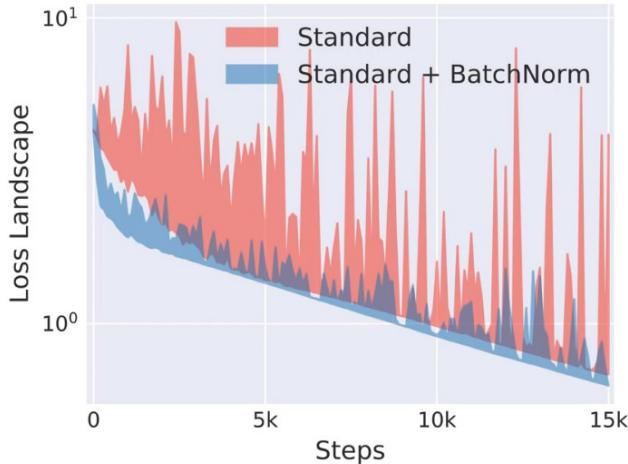
How Does Batch Normalization Help Optimization?

Shibani Santurkar*
MIT
shibani@mit.edu

Dimitris Tsipras*
MIT
tsipras@mit.edu

Andrew Ilyas*
MIT
ailyas@mit.edu

Aleksander Mądry
MIT
madry@mit.edu



“the positive impact of BatchNorm on training might be somewhat serendipitous”

In the last two months, there's been two papers (so it took three years for people to really figure this out), in the last two months, there's been two papers that have shown batch normalization doesn't reduce covariate shift at all. And even if it did, that has nothing to do with why it works. I think that's an interesting insight, again, which is why we should be focusing on being practitioners and experimentalists and developing an intuition.

What batch norm does is what you see in this picture here in this paper. Here are steps or batches (x-axis). And here is loss (y-axis). The red line is what happens when you train without batch norm - very very bumpy. And here, the blue line is what happens when you train with batch norm - not very bumpy at all. What that means is, you can increase your learning rate with batch norm. Because these big bumps represent times that you're really at risk of your set of weights jumping off into some awful part of the weight space that it can never get out of again. So if it's less bumpy, then you can train at a higher learning rate. So that's actually what's going on.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\begin{aligned}\mu_{\mathcal{B}} &\leftarrow \frac{1}{m} \sum_{i=1}^m x_i && // \text{mini-batch mean} \\ \sigma_{\mathcal{B}}^2 &\leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 && // \text{mini-batch variance} \\ \hat{x}_i &\leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} && // \text{normalize} \\ y_i &\leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) && // \text{scale and shift}\end{aligned}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

This is the algorithm, and it's really simple. The algorithm is going to take a mini batch. So we have a mini batch, and remember this is a layer, so the thing coming into it is activations. Batch norm is a layer, and it's going to take in some activations. So the activations are what it's calling x_1, x_2, x_3 and so forth.

1. The first thing we do is we find the mean with those activations - sum divided by the count that is just the mean.
2. The second thing we do is we find the variance of those activations - a difference squared divided by the mean is the variance.
3. Then we normalize - the values minus the mean divided by the standard deviation is the normalized version. It turns out that bit is actually not that important. We used to think it was - it turns out it's not. The really important bit is the next bit.
4. We take those values and we add a vector of biases (they call it beta here). We've seen that before. We've used a bias term before. So we're just going to add a bias term as per usual. Then we're going to use another thing that's a lot like a bias term, but rather than adding it, we're going to multiply by it. So there's these parameters gamma γ and beta β which are learnable parameters.

Remember, in a neural net there's only two kinds of number; activations and parameters. These are parameters. They're things that are learnt with gradient descent. β is just a normal bias layer and γ is a multiplicative bias layer. Nobody calls it that, but that's all it is. It's just like bias, but we multiply rather than add. That's what batch norm is. That's what the layer does.

So why is that able to achieve this fantastic result? I'm not sure anybody has exactly written this down before. If they have, I apologize for failing to site it because I haven't seen it. But let me explain. What's actually going on here. The value of our predictions \hat{y} is some function of our various weights. There could be millions of them (weight 1 million) and it's also a function, of course, of the inputs to our layer.

$$\hat{y} = f(w_1, w_2 \dots w_{1000000}, \vec{x})$$

This function f is our neural net function whatever is going on in our neural net. Then our loss, let's say it's mean squared error, is just our actuals minus our predicted squared.

$$L = \sum (y - \hat{y})^2$$

Let's say we're trying to predict movie review outcomes, and they're between 1 and 5. And we've been trying to train our model and the activations at the very end currently between -1 and 1. So they're way off where they need to be. The scale is off, the mean is off, so what can we do? One thing we could do would be to try and come up with a new set of weights that cause the spread to increase, and cause the mean to increase as well. But that's going to be really hard to do, because remember all these weights interact in very intricate ways. We've got all those nonlinearities, and they all combine together. So to just move up, it's going to require navigating through this complex landscape and we use all these tricks like momentum and Adam and stuff like that to help us, but it still requires a lot of twiddling around to get there. So that's going to take a long time, and it's going to be bumpy.

But what if we did this? What if we went times g plus b ?

$$\hat{y} = f(w_1, w_2 \dots w_{1000000}, \vec{x}) \times g + b$$

We added 2 more parameter vectors. Now it's really easy. In order to increase the scale, that number g has a direct gradient to increase the scale. To change the mean, that number b has a direct gradient to change the mean. There's no interactions or complexities, it's just straight up and down, straight in and out. That's what batch norm does. Batch norm is basically making it easier for it to do this really important thing which is to shift the outputs up and down, and in and out. And that's why we end up with these results.

Those details, in some ways, don't matter terribly. The really important thing to know is **you definitely want to use it**. Or if not it, something like it. There's various other types of normalization around nowadays, but batch norm works great. The other main normalization type we use in fast.ai is something called weight norm which is much more just in the last few months' development.

```

class TabularModel(nn.Module):
    "Basic model for tabular data."
    def __init__(self, emb_szs:ListSizes, n_cont:int, out_sz:int, layers:Collection[int], ps:Collection[float]=None,
                 emb_drop:float=0., y_range:OptRange=None, use_bn:bool=True, bn_final:bool=False):
        super().__init__()
        ps = ifnone(ps, [0]*len(layers))
        ps = listify(ps, layers)
        self.embeds = nn.ModuleList([embedding(ni, nf) for ni,nf in emb_szs])
        self.emb_drop = nn.Dropout(emb_drop)
        self.bn_cont = nn.BatchNorm1d(n_cont)
        n_emb = sum(e.embedding_dim for e in self.embeds)
        self.n_emb,self.n_cont,self.y_range = n_emb,n_cont,y_range
        sizes = self.get_sizes(layers, out_sz)
        actns = [nn.ReLU(inplace=True)] * (len(sizes)-2) + [None]
        layers = []
        for i,(n_in,n_out,dp,act) in enumerate(zip(sizes[:-1],sizes[1:],[0.]*ps,actns)):
            layers += bn_drop_lin(n_in, n_out, bn=use_bn and i!=0, p=dp, actn=act)
        if bn_final: layers.append(nn.BatchNorm1d(sizes[-1]))
        self.layers = nn.Sequential(*layers)

    def get_sizes(self, layers, out_sz):
        return [self.n_emb + self.n_cont] + layers + [out_sz]

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

```

So that's batch norm and so what we do is we create a batch norm layer for every continuous variable. `n_cont` is a number of continuous variables. In fast.ai, `n_something` always means the count of that thing, `cont` always means continuous. Then here is where we use it. We grab our continuous variables and we throw them through a batch norm layer.

So then over here you can see it in our model.

```

File Edit View Insert Cell Kernel Navigate Widgets Help Trusted
(17): Embedding(7, 4)
(18): Embedding(4, 3)
(19): Embedding(4, 3)
(20): Embedding(9, 5)
(21): Embedding(9, 5)
(22): Embedding(3, 2)
(23): Embedding(3, 2)
)
(emb_drop): Dropout(p=0.04)
(bn_cont): BatchNorm1d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(layers): Sequential(
    (0): Linear(in_features=229, out_features=1000, bias=True)
    (1): ReLU(inplace)
    (2): BatchNorm1d(1000, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): Dropout(p=0.001)
    (4): Linear(in_features=1000, out_features=500, bias=True)
    (5): ReLU(inplace)
    (6): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): Dropout(p=0.01)
    (8): Linear(in_features=500, out_features=1, bias=True)
)

```

One interesting thing is this momentum here. This is not momentum like in optimization, but this is momentum as in exponentially weighted moving average. Specifically this mean and standard deviation (in batch norm algorithm), we don't actually use a different mean and standard deviation for every mini batch. If we did, it would vary so much that it be very hard to train. So instead, we take an exponentially weighted moving average of the mean and standard deviation. If you don't remember what I mean by that, look back at last week's lesson to remind yourself about exponentially weighted moving averages which we implemented in excel for the momentum and Adam gradient squared terms.

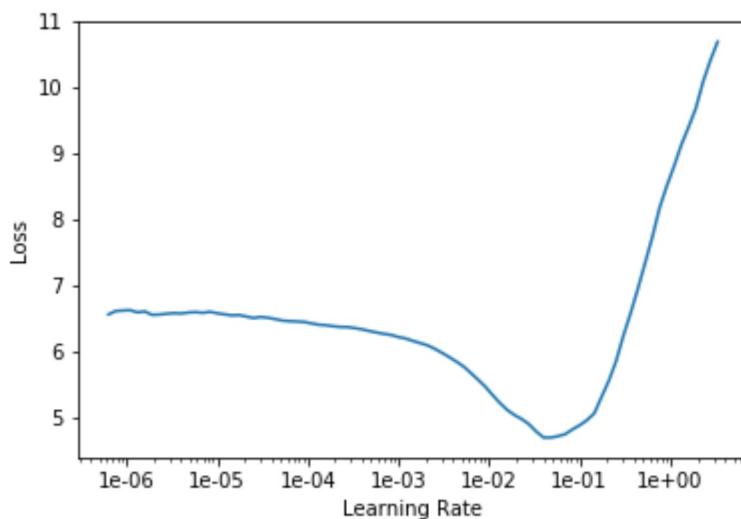
[53:10]

You can vary the amount of momentum in a batch norm layer by passing a different value to the constructor in PyTorch. If you use a smaller number, it means that the mean and standard deviation will vary less from mini batch to mini batch, and that will have less of a regularization effect. A larger number will mean the variation will be greater for a mini batch to mini batch, that will have more of a regularization effect. So as well as this thing of training more nicely because it's parameterised better, this momentum term in the mean and standard deviation is the thing that adds this nice regularization piece.

When you add batch norm, you should also be able to use a higher learning rate. So that's our model. So then you can go `lr_find`, you can have a look:

```
learn.lr_find()
```

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

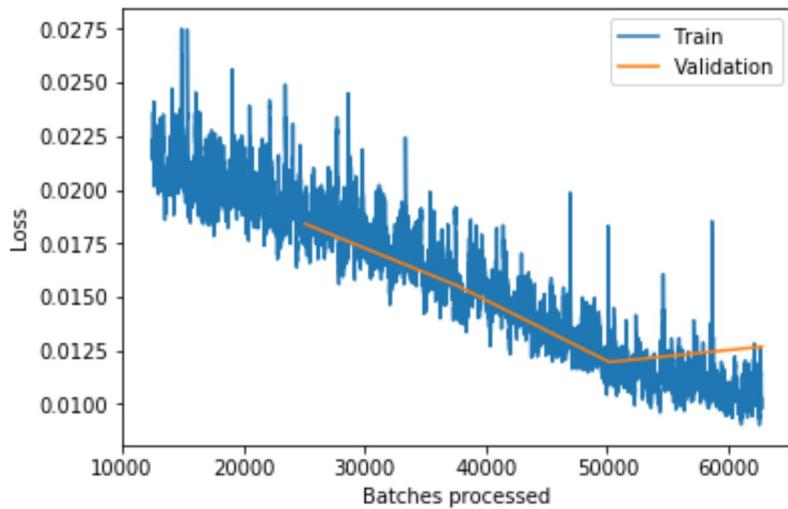


```
learn.fit_one_cycle(5, 1e-3, wd=0.2)
```

```
Total time: 14:18
epoch  train_loss  valid_loss  exp_rmspe
1      0.021467    0.023627    0.149858    (02:49)
2      0.017700    0.018403    0.128610    (02:52)
3      0.014242    0.015516    0.116233    (02:51)
4      0.012754    0.011944    0.108742    (02:53)
5      0.010238    0.012665    0.105895    (02:52)
```

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

```
learn.recorder.plot_losses(last=-1)
```



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

```
learn.fit_one_cycle(5, 3e-4)
```

```
Total time: 13:52
epoch  train_loss  valid_loss  exp_rmspe
1      0.018280    0.021080    0.118549    (02:49)
2      0.018260    0.015992    0.121107    (02:50)
3      0.015710    0.015826    0.113787    (02:44)
4      0.011987    0.013806    0.109169    (02:43)
5      0.011023    0.011944    0.104263    (02:42)
```

```
learn.fit_one_cycle(5, 3e-4)
```

Total time: 14:41

epoch	train_loss	valid_loss	exp_rmspe
1	0.012831	0.012518	0.106848
2	0.011145	0.013722	0.109208
3	0.011676	0.015752	0.115598
4	0.009419	0.012901	0.107179
5	0.009156	0.011122	0.103746

(10th place in the competition was 0.108)

We end up 0.103. 10th place in the competition was 0.108, so it's looking good. Again, take it with a slight grain of salt because what you actually need to do is use the real training set and submit it to Kaggle, but you can see we're very much amongst the cutting-edge of models at least as of 2015. As I say, they haven't really been any architectural improvements since then. There wasn't batch norm when this was around, so the fact we added batch norm means that we should get better results and certainly more quickly. If I remember correctly, in their model, they had to train at a lower learning rate for quite a lot longer. As you can see, this is less than 45 minutes of training. So that's nice and fast.

Question: In what proportion would you use dropout vs. other regularization errors, like, weight decay, L2 norms, etc.? [54:49]

So remember that L2 regularization and weight decay are kind of two ways of doing the same thing? We should always use the weight decay version, not the L2 regularization version. So there's weight decay. There's batch norm which kind of has a regularizing effect. There's data augmentation which we'll see soon, and there's dropout. So batch norm, we pretty much always want. So that's easy. Data augmentation, we'll see in a moment. So then it's really between dropout versus weight decay. I have no idea. I don't think I've seen anybody to provide a compelling study of how to combine those two things. Can you always use one instead of the other? Why? Why not? I don't think anybody has figured that out. I think in practice, it seems that you generally want a bit of both. You pretty much always want some weight decay, but you often also want a bit of dropout. But honestly, I don't know why. I've not seen anybody really explain why or how to decide. So this is one of these things you have to try out and kind of get a feel for what tends to work for your kinds of problems. I think the defaults that we provide in most of our learners should work pretty well in most situations. But yeah, definitely play around with it.

⌚ Data augmentation [56:45]

The next kind of regularization we're going to look at is data augmentation. Data augmentation is one of the least well studied types of regularization, but it's the kind that I think I'm kind of the most excited about. The reason I'm kind of the most about it is that there's basically almost no cost to it. You can do data augmentation and get better generalization without it taking longer to train, without underfitting (to an extent, at least). So let me explain.

[lesson6-pets-more.ipynb](#)

What we're going to do now is we're going to come back to a computer vision, and we're going to come back to our pets data set again. So let's load it in. Our pets data set, the images are inside the `images` subfolder:

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

from fastai.vision import *

bs = 64

path = untar_data(URLs.PETS) / 'images'
```

I'm going to call `get_transforms` as per usual, but when we call `get_transforms` there's a whole long list of things that we can provide:

```
tfms = get_transforms(max_rotate=20, max_zoom=1.3, max_lighting=0.4,
                     max_warp=0.4, p_affine=1., p_lighting=1.)
```

So far, we haven't been varying that much at all. But in order to really understand data augmentation, I'm going to kind of ratchet up all of the defaults. There's a parameter here for what's the probability of an affine transform happening, what's the probability of a lighting transfer happening, so I set them both to 1. So they're all gonna get transformed, I'm going to do more rotation, more zoom, more lighting transforms, and more warping.

What are all those mean? Well, you should check the documentation, and to do that, by typing doc and there's the brief documentation:

In [5]: doc(get_transforms)

get_transforms [source]

```
get_transforms(`do_flip`: bool =`True`, `flip_vert`: bool =`False`, `max_rotate`: float =`10.0`,  
`max_zoom`: float =`1.1`, `max_lighting`: float =`0.2`, `max_warp`: float =`0.2`, `p_affine`: float =`0.75`,  
`p_lighting`: float =`0.75`, `xtra_tfms`: Optional [Collection [Transform]] =`None`) →  
Collection [Transform]
```

Utility func to easily create a list of flip, rotate, zoom, warp, lighting transforms.

[Show in docs](#)

But the real documentation is in docs. so I'll click on [Show in docs](#) and here it is. This tells you what all those do, but generally the most interesting parts of the docs tend to be at the top where you kind of get the summaries of what's going on.

Here, there's something called [List of transforms](#) and you can see every transform has something showing you lots of different values of it.

List of transforms

Here is the list of all the deterministic functions on which the transforms are built. As explained before, each of those can have a probability `p` of being executed, and any time an argument is type-annotated with a random function, it's possible to randomize it via that function.

brightness

[\[source\]](#)

```
brightness(`x`, `change`: uniform) → Image :: TfmLighting
```

Apply `change` in brightness of image `x`.

This transform adjusts the brightness of the image depending on the value in `change`. A `change` of 0 will transform the image to black and a `change` of 1 will transform the image to white. `change` =0.5 doesn't do adjust the brightness.

```
fig, axs = plt.subplots(1,5, figsize=(12,4))  
for change, ax in zip(np.linspace(0.1,0.9,5), axs):  
    brightness(get_ex(), change).show(ax=ax, title=f'change={change:.1f}')
```



Here's brightness. So make sure you read these, and remember these notebooks, you can open up and run this code yourself and get this output. All of these HTML documentation documents are auto-generated from the notebooks in the `docs_source` directory in the fast.ai repo. So you will see the exact same cats, if you try this. Sylvain really likes cats, so there's a lot of cats in the documentation, and because he's been so awesome at creating great documentation, he gets to pick the cats.

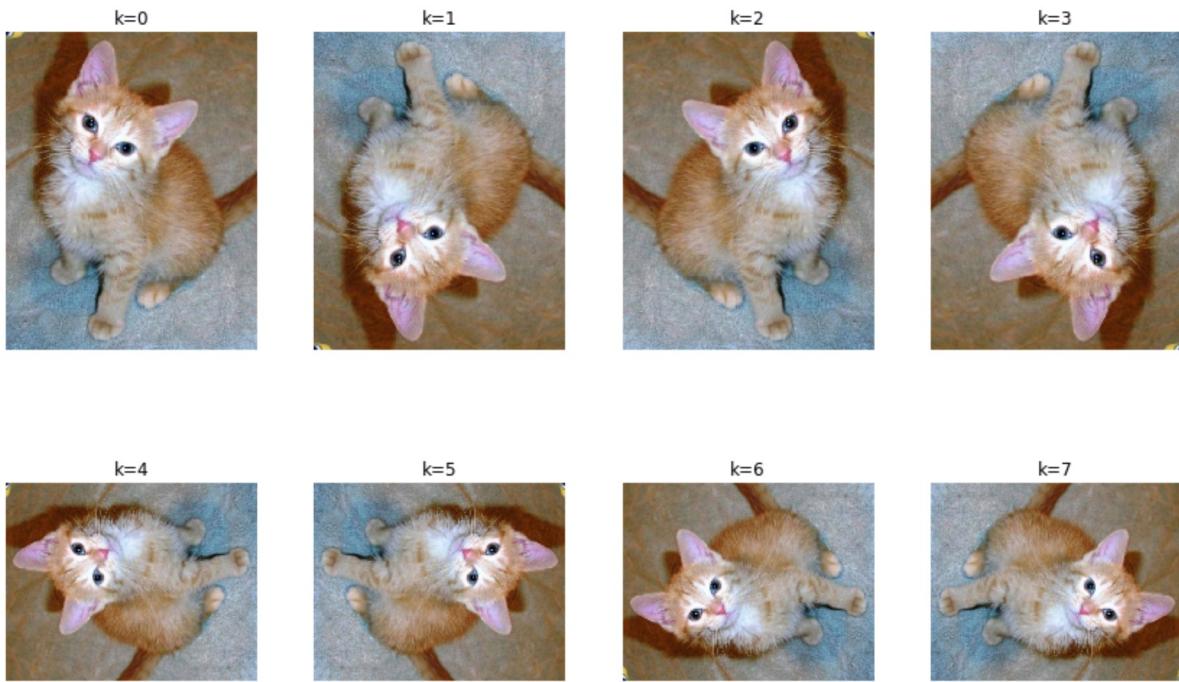
So for example, looking at different values of brightness, what I do here is I look to see two things. The first is for which of these levels of transformation is it still clear what the picture is a picture of. The left most one is kind of getting to a point where it's pretty unclear, the right most one is possibly getting a little unclear. The second thing I do is I look at the actual data set that I'm modeling or particularly the data set that I'll be using as validation set, and I try to get a sense of what the variation (in this case) in lighting is.

[1:00:12]

So when they are nearly all professionally taking photos, I would probably want them all to be about in the middle. But if the photos are taken by some pretty amateur photographers, there are likely to be some that are very overexposed, some very underexposed. So you should pick a value of this data augmentation for brightness that both allows the image to still be seen clearly, and also represents the kind of data that you're going to be using this to model it in practice.



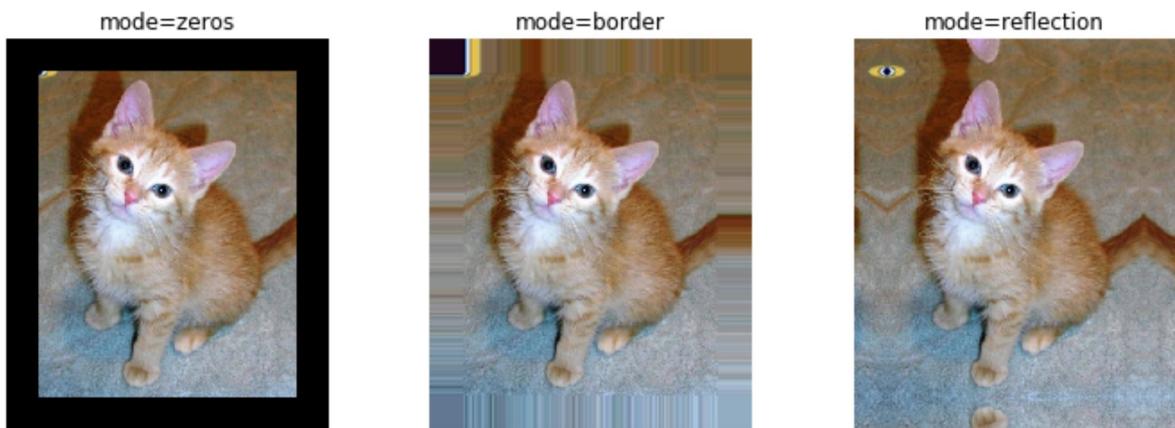
You kind of see the same thing for contrast. It'd be unusual to have a data set with such ridiculous contrast, but perhaps you do - in which case, you should use data augmentation up to that level. But if you don't, then you shouldn't.



This one called `dihedral` is just one that does every possible rotation and flip. So obviously most of your pictures are not going to be upside down cats. So you probably would say "hey, this doesn't make sense. I won't use this for this data set." But if you're looking at satellite images, of course you would.

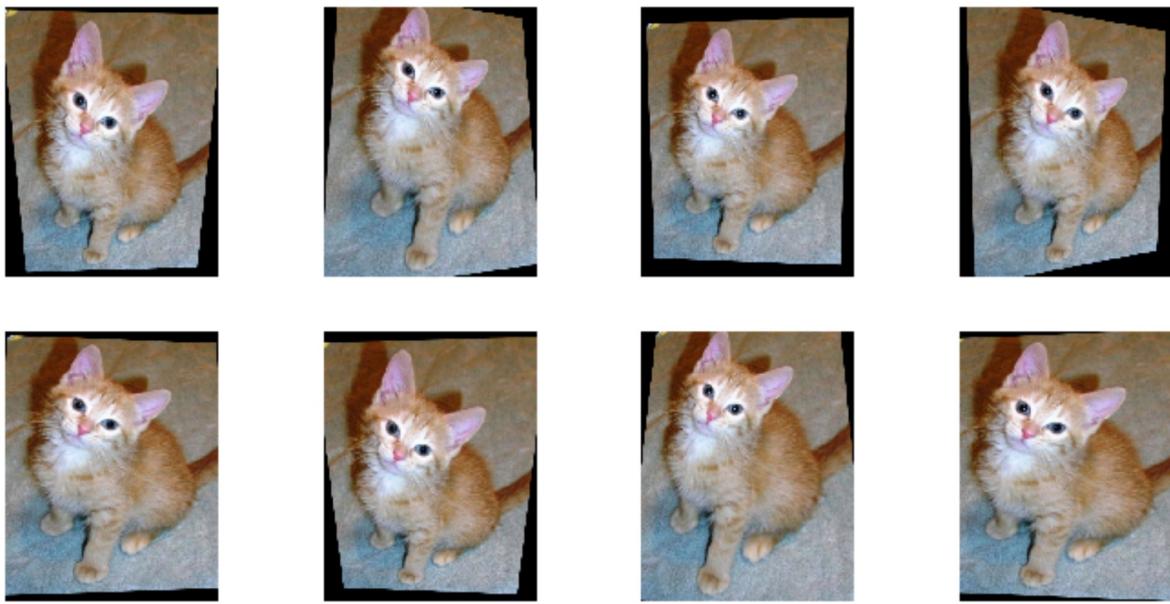
On the other hand, `flip` makes perfect sense. So you would include that.

A lot of things that you can do with fast.ai lets you pick a padding mode, and this is what padding mode looks like:



You can pick zeros, you can pick border which just replicates, or you can pick reflection which as you can see is it's as if the last little few pixels are in a mirror. **Reflection is nearly always better**, by the way. I don't know that anybody else has really studied this, but we have studied it in some depth. We haven't actually written a paper about it, but just enough for our own purposes to say reflection works best most of the time. So that's the default.

Then there's a really cool bunch of perspective warping ones which I'll probably show you by using symmetric warp.



We've added black borders to this so it's more obvious for what's going on. As you can see, what symmetric warp is doing is as if the camera is being moved above or to the side of the object, and literally warping the whole thing like that. The cool thing is that as you can see, each of these pictures is as if this cat was being taken kind of from different angles, so they're all kind of optically sensible. And this is a really great type of data augmentation. It's also one which I don't know of any other library that does it or at least certainly one that does it in a way that's both fast and keeps the image crisp as it is in fast.ai, so this is like if you're looking to win a Kaggle competition, this is the kind of thing that's going to get you above the people that aren't using the fast.ai library.

Having looked at all that, we are going to have a little `get_data` function that just does the usual data block stuff, but we're going to add padding mode explicitly so that we can turn on padding mode of zeros just so we can see what's going on better.

```
src = ImageItemList.from_folder(path).random_split_by_pct(0.2, seed=2)
```

```
def get_data(size, bs, padding_mode='reflection'):
    return (src.label_from_re(r'([^\_]+)\_\d+.jpg$')
            .transform(tfms, size=size, padding_mode=padding_mode)
            .databunch(bs=bs).normalize(imagenet_stats))
```

```
data = get_data(224, bs, 'zeros')
```

```
def _plot(i,j,ax):
```

```
x,y = data.train_ds[3]
x.show(ax, y=y)

plot_multi(_plot, 3, 3, figsize=(8,8))
```

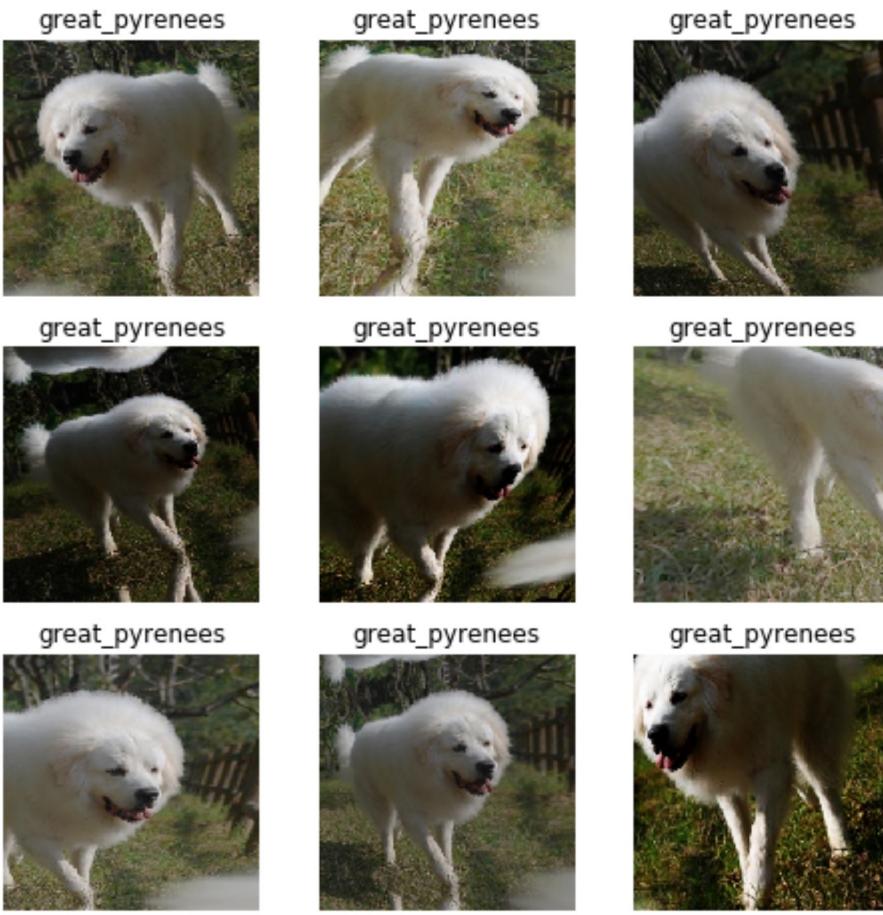


Fast.ai has this handy little function called `plot_multi` which is going to create a 3 by 3 grid of plots, and each one will contain the result of calling this (`_plot`) function which will receive the plot coordinates and the axis. So I'm actually going to plot the exact same thing in every box, but because this is a training data set, it's going to use data augmentation. You can see the same doggie using lots of different kinds of data augmentation. So you can see why this is going to work really well. Because these pictures all look pretty different. But we didn't have to do any extra hand labeling or anything. They're like free extra data. So data augmentation is really really great.

One of the big opportunities for research is to figure out ways to do data augmentation in other domains. So how can you do data augmentation with text data, or genomic data, or histopathology data, or whatever. Almost nobody's looking at that, and to me, it's one of the biggest opportunities that could let you decrease data requirements by like five to ten X.

```
data = get_data(224,bs)
```

```
plot_multi(_plot, 3, 3, figsize=(8,8))
```



Here's the same thing again, but with reflection padding instead of zero padding. and you can kind of see like this doggies legs are actually being reflected at the bottom (bottom center). So reflection padding tends to create images that are much more naturally reasonable. In the real world, you don't get black borders. So they do seem to work better.

⌚ Convolutional Neural Network [1:05:14]

Because we're going to study convolutional neural networks, we are going to create a convolutional neural network. You know how to create them, so I'll go ahead and create one. I will fit it for a little bit. I will unfreeze it, I will then create a larger version of the data set 352 by 352, and fit for a little bit more, and I will save it.

```
gc.collect()  
learn = create_cnn(data, models.resnet34, metrics=error_rate, bn_final=True)  
  
learn.fit_one_cycle(3, slice(1e-2), pct_start=0.8)
```

```
Total time: 00:52
epoch  train_loss  valid_loss  error_rate
1      2.413196   1.091087   0.191475   (00:18)
2      1.397552   0.331309   0.081867   (00:17)
3      0.889401   0.269724   0.068336   (00:17)
```

```
learn.unfreeze()
learn.fit_one_cycle(2, max_lr=slice(1e-6,1e-3), pct_start=0.8)
```

```
Total time: 00:44
epoch  train_loss  valid_loss  error_rate
1      0.695697   0.286645   0.064276   (00:22)
2      0.636241   0.295290   0.066982   (00:21)
```

```
data = get_data(352,bs)
learn.data = data
```

```
learn.fit_one_cycle(2, max_lr=slice(1e-6,1e-4))
```

```
Total time: 01:32
epoch  train_loss  valid_loss  error_rate
1      0.626780   0.264292   0.056834   (00:47)
2      0.585733   0.261575   0.048038   (00:45)
```

```
learn.save('352')
```

We have a CNN. And we're going to try and figure out what's going on in our CNN. The way we're going to try and figure it out is specifically that we're going to try to learn how to create this picture:



This is a heat map. This is a picture which shows me what part of the image did the CNN focus on when it was trying to decide what this picture is. We're going to make this heat map from scratch.

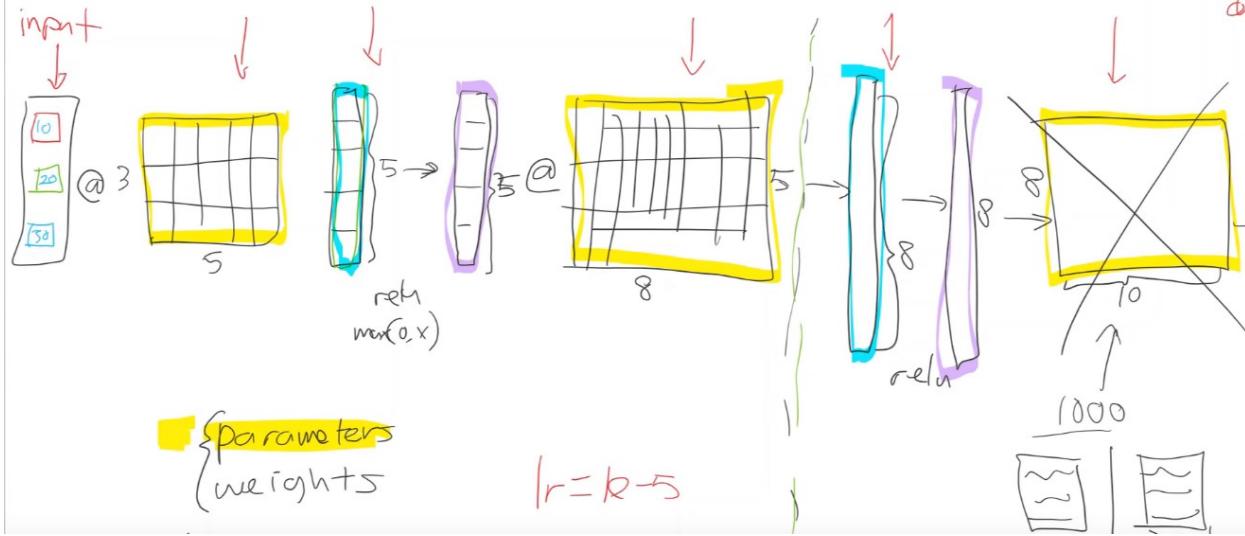
We're kind of at a point now in the course where I'm assuming that if you've got to this point and you're still here, thank you, then you're interested enough that you're prepared to dig into some of these details. So we're actually going to learn how to create this heat map without almost any fast.ai stuff. We're going to use pure tensor arithmetic in PyTorch, and we're going to try and use that to really understand what's going on.

To warn you, none of it is rocket science, but a lot of it is going to look really new so don't expect to get it the first time, but expect to listen, jump into the notebook, try a few things, test things out, look particularly at tensor shapes and inputs and outputs to check your understanding, then go back and listen again. Try it a few times because you will get there. It's just that there's going to be a lot of new concepts because we haven't done that much stuff in pure PyTorch.

[1:07:32]

Let's learn about convolutional neural networks. The funny thing is it's pretty unusual to get close to the end of a course, and only then look at convolutions. But when you think about it, knowing actually how batch norm works, how dropout works, or how convolutions work isn't nearly as important as knowing how it all goes together, what to do with them, and how to figure out how to do those things better. But we're at a point now where we want to be able to do things like heatmap. And although we're adding this functionality directly into the library so you can run a function to do that, the more you do, the more you'll find things that you want to do a little bit differently to how we do them, or there'll be something in your domain where you think "oh, I could do a slight variation of that." So you're getting to a point in your experience now where it helps to know how to do more stuff yourself, and that means you need to understand what's really going on behind the scenes.

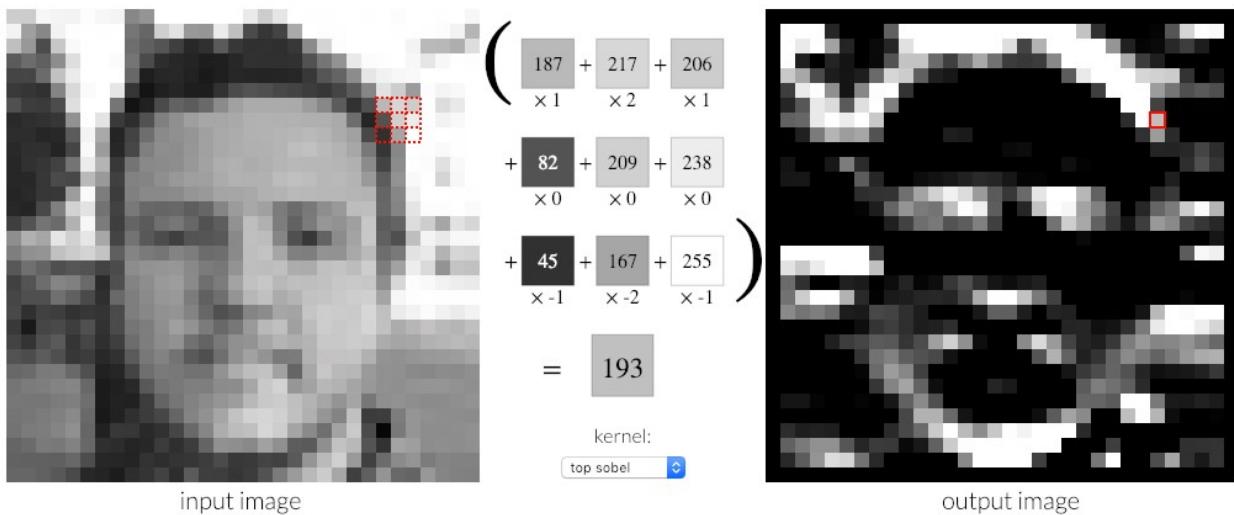
What's really going on behind the scenes is that we are creating a neural network that looks a lot like this:



But rather than doing a matrix multiply, we're actually going to do, instead, a convolution. A convolution is just a kind of matrix multiply which has some interesting properties.

$$\begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

Below, for each 3x3 block of pixels in the image on the left, we multiply each pixel by the corresponding entry of the kernel and then take the sum. That sum becomes a new pixel in the image on the right. Hover over a pixel on either image to see how its value is computed.



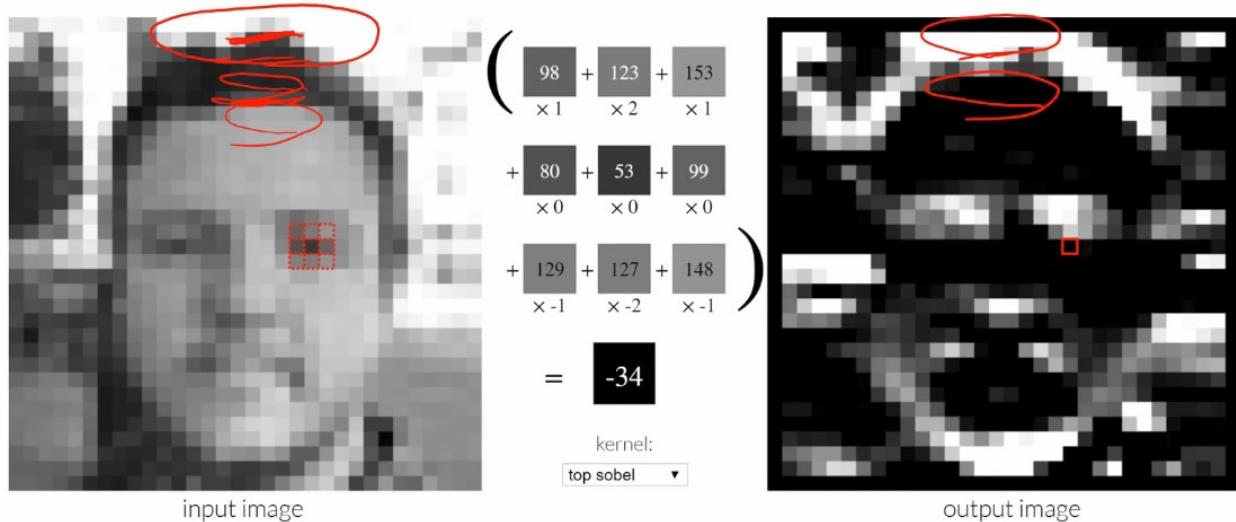
You should definitely check out this website <http://setosa.io/ev/image-kernels/> (ev stands for explain visually) where we have stolen this beautiful animation. It's actually a JavaScript thing that you can actually play around with yourself in order to show you how convolutions work. It's actually showing you a convolution as we move around these little red squares.

Here's a picture - a black and white or grayscale picture. Each 3x3 bit of this picture as this red thing moves around, it shows you a different 3x3 part (the bottom matrix). It shows you over here the value of the pixels. In fast.ai's case, our pixel values are between nought to one, in this case they are between nought to 255. So here are nine pixel values (the bottom matrix). This area is pretty white, so they're pretty high numbers.

As we move around, you can see the nine big numbers change, and you can also see their colors change. Up here is another nine numbers, and you can see those in the little $1 \times 1 \times 1$ in the bottom matrix. What you might see going on is as we move this little red block, as these numbers change, we then multiply them by the corresponding numbers in the upper matrix. So let's start using some nomenclature.

The thing up here, we are going to call the kernel - the convolutional kernel. So we're going to take each little 3x3 part of this image, and we're going to do an element-wise multiplication of each of the 9 pixels that we are mousing over with each of the 9 items in our kernel. Once we multiply each set together, we can then add them all up. And that is what's shown on the right. As the little bunch of red things move on the left, you can see there's one red thing that appears on the right. The reason there's one red thing over here is because each set of 9, after getting through the element-wise multiplication with the kernel, get added together to create one output. Therefore the size of the left image has one pixel less on each edge than the original, as you can see. See how there's black borders on it? That's because at the edge the 3x3 kernel, can't quite go any further. So the furthest you can go is to end up with a dot in the middle just off the corner.

So why are we doing this? Well, perhaps you can see what's happened. This face has turned into some white parts outlining the horizontal edges. How? Well, the how is just by doing this element wise multiplication of each set of 9 pixels with this kernel, adding them together, and sticking the result in the corresponding spot over here. Why is that creating white spots where the horizontal edges are? Well, let's think about it. Let's look up here (the top of the head):

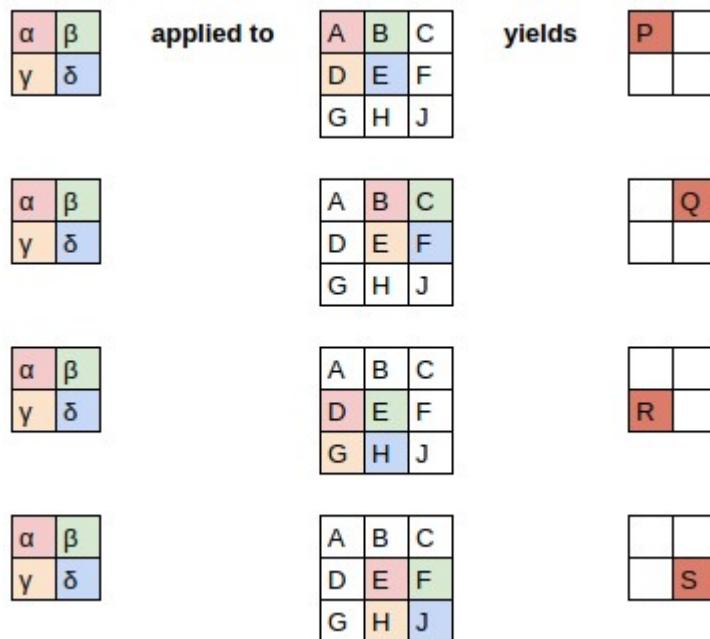


If we're just in this little bit here, then the spots above it are all pretty white, so they have high numbers. so the bits above it (i.e. the big numbers) are getting multiplied by (1 2 1). So that's going to create a big number. And the ones in the middle are all zeros, so don't care about that. And then the ones underneath are all small numbers because they're all close to 0, so that really doesn't do much at all. Therefore that little set there is going to end up with bright white. Whereas on the other side right down here (the hairline), you've got light pixels underneath, so they're going to get a lot of negative; dark pixels on top which are very small, so not much happens. Therefore, we're going to end up with very negative output.

This thing where we take each 3x3 area, and element wise multiply them with a kernel, and add each of those up together to create one output is called a **convolution**. That's it. That's a convolution. That might look familiar to you, because what we did back a while ago is we looked at that Zeiler and Fergus paper where we saw like each different layer and we visualized what the weights were doing. Remember how the first layer was basically finding diagonal edges and gradient? That's because that's all a convolution can do. Each of our layers is just a convolution. So the first layer can do nothing more than this kind of thing (e.g. finding top edges). But the nice thing is, the next layer could then take the results of this, and it could kind of combine one channel (the output of one convolutional field is called a channel), so it could take one channel that found top edges and another channel that finds left edges, and then the layer above that could take those two as input and create something that finds top left corners as we saw when we looked at Zeiler and Fergus visualizations.

[1:15:02]

Let's take a look at this from another angle or quite a few other angles. We're going to look at [a fantastic post from Matt Kleinsmith](#) who was actually a student in the first year we did this course. He wrote this as a part of his project work back then.



What he's going to show here is here is our image. It's a 3 by 3 image (middle) and our kernel is a 2 by 2 kernel (left). What we're going to do is we're going to apply this kernel to the top left 2x2 part of this image. So the pink bit will be correspondingly multiplied by the pink bit, the green by the green, and so forth. And they all get added up together to create this top left in the output. In other words:

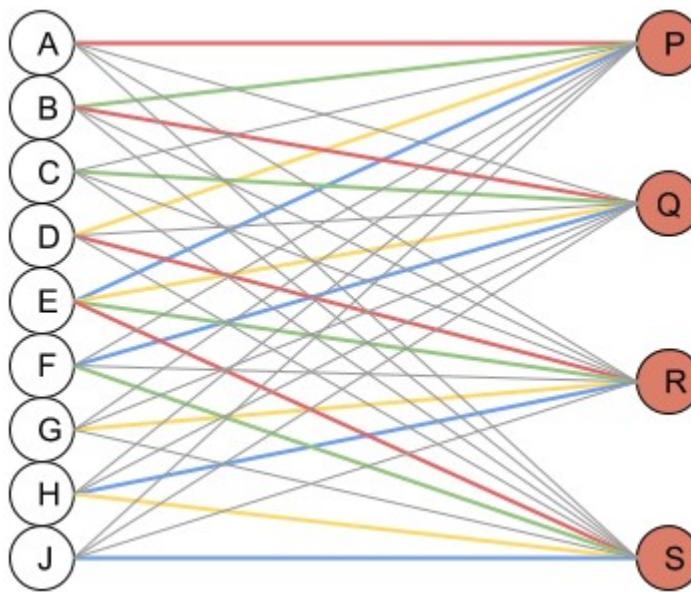
$$\alpha A + \beta B + \gamma D + \delta E + b = P$$

$$\alpha B + \beta C + \gamma E + \delta F + b = Q$$

$$\alpha D + \beta E + \gamma G + \delta H + b = R$$

$$\alpha E + \beta F + \gamma H + \delta J + b = S$$

b is a bias so that's fine. That's just a normal bias. So you can see how basically each of these output pixels is the result of some different linear equation. And you can see these same four weights are being moved around, because this is our convolutional kernel.



Here is another way of looking at it which is here is a classic neural network view. Now P is result of multiplying every one of these inputs by a weight and then adding them all together, except the gray ones are going to have a value of zero. Because remember, P was only connected to A B D and E. In other words, remembering that this represents a matrix multiplication, therefore we can represent this as a matrix multiplication.

α	β	0	γ	δ	0	0	0	0
0	α	β	0	γ	δ	0	0	0
0	0	0	α	β	0	γ	δ	0
0	0	0	0	α	β	0	γ	δ

A	b
B	b
C	b
D	b
E	b
F	
G	
H	
J	

$$* \quad + \quad = \quad = \quad =$$

$\alpha A + \beta B + 0C + \gamma D + \delta E + 0F + 0G + 0H + 0J + b$
$0A + \alpha B + \beta C + 0D + \gamma E + \delta F + 0G + 0H + 0J + b$
$0A + 0B + 0C + \alpha D + \beta E + 0F + \gamma G + \delta H + 0J + b$
$0A + 0B + 0C + 0D + \alpha E + \beta F + 0G + \gamma H + \delta J + b$

$\alpha A + \beta B + \gamma D + \delta E + b$
$\alpha B + \beta C + \gamma E + \delta F + b$
$\alpha D + \beta E + \gamma G + \delta H + b$
$\alpha E + \beta F + \gamma H + \delta J + b$

P
Q
R
S

A B C D E F G H J

Here is our list of pixels in our 3×3 image flattened out into a vector, and here is a matrix vector multiplication plus bias. Then, a whole bunch of them, we're just going to set to zero. So you can see here we've got:

$\alpha \beta \circ \gamma \delta \circ \circ \circ \circ$

which corresponds to

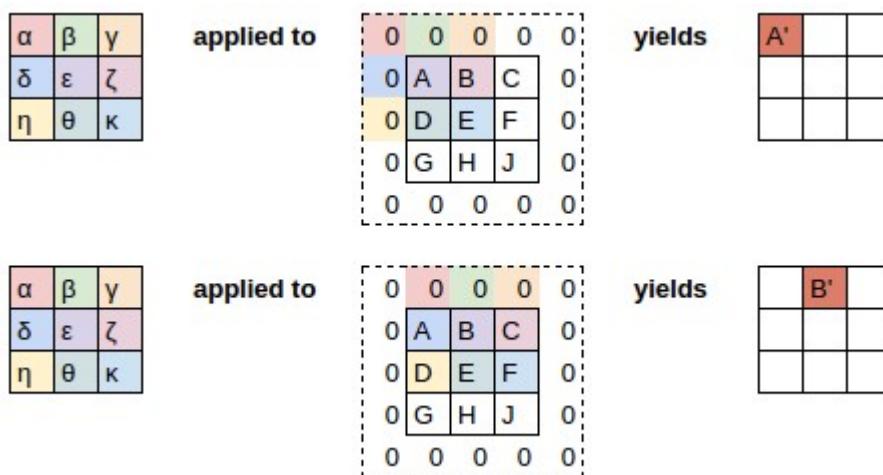
A	B	C
D	E	F
G	H	J

In other words, a convolution is just a matrix multiplication where two things happen:

- some of the entries are set to zero all the time
 - all of the ones are the same color, always have the same weight

So when you've got multiple things with the same weight, that's called **weight tying**.

Clearly we could implement a convolution using matrix multiplication, but we don't because it's slow. So in practice, our libraries have specific convolution functions that we use. And they're basically doing [this](#), which is [this](#) which is [this equation](#) which is the same as [this matrix multiplication](#).



As we discussed, we have to think about padding because if you have a 3 by 3 kernel and a 3 by 3 image, then that can only create one pixel of output. There's only one place that this 3x3 can go. So if we want to create more than one pixel of output, we have to do something called padding which is to put additional numbers all around the outside. What most libraries do is that they just put a bunch of zeros of all around the outside. So for 3x3 kernel, a single zero on every edge piece here. Once you've padded it like that, you can now move your 3x3 kernel all the way across, and give you the same output size that you started with.

As we mentioned, in fast.ai, we don't normally necessarily use zero padding. Where possible, we use reflection padding, although for these simple convolutions, we often use zero padding because it doesn't matter too much in a big image. It doesn't make too much difference.

So that's what a convolution is. A convolutional neural network wouldn't be very interesting if it can only create top edges. So we have to take it a little bit further.

If we have an input, and it might be a standard kind of red-green-blue picture. Then we can create a 3x3 kernel like so, and then we could pass that kernel over all of the different pixels. But if you think about it, we actually don't have a 2D input anymore, we have a 3D input (i.e. a rank 3 tensor). So we probably don't want to use the same kernel values for each of red and green and blue, because for example, if we're creating a green frog detector, we would want more activations on the green than we would on the blue. Or if we're trying to find something that could actually find a gradient that goes from green to blue, then the different kernels for each channel need to have different values in. Therefore, we need to create a 3 by 3 by 3 kernel. This is still our kernel, and we're still good a very it across the height and the width. But rather than doing an element-wise multiplication of 9 things, we're going to do an element-wise multiplication of 27 things (3 by 3 by 3) and we're still going to then add them up into a single number. As we pass this cube over this and the kind of like a little bit that's going to be sitting behind it (the yellow cube). As we do that part of the convolution, it's still going to create just one number because we do an element-wise multiplication of all 27 and add them all together.

We can do that across the whole single unit padded input. We started with 5 by 5, so we're going to end up with an output that's also 5 by 5. But now our input was 3 channels and our output is only one channel. We're not going to be able to do very much with just one channel, because all we've done now is found the top edge. How are we going to find a side edge? and a gradient? and an area of constant white? Well, we're going to have to create another kernel, and we're going to have to do that convolved over the input, and that's going to create another 5x5. Then we can just stack those together across this as another axis, and we can do that lots and lots of times and that's going to give us another rank 3 tensor output.

That's what happens in practice. In practice, we start with an input which is H by W by (for images) 3. We pass it through a bunch of convolutional kernels, and we get to pick how many we want, and it gives us back an output of height by width by however many kernels we had, and often that might be something like 16 in the first layer. Now we've got 16 channels representing things like how much left edge was on this pixel, how much top edge was in this pixel, how much blue to red gradient was on this set of 9 pixels each with RGB.

Then you can just do the same thing. You can have another bunch of kernels, and that's going to create another output ranked 3 tensor (again height by width by whatever - might still be 16). Now what we really like to do is, as we get deeper in the network, we actually want to have more and more channels. We want to be able to find a richer and richer set of features so that as we saw in the Zeiler and Fergus paper, by layer 4 or 5, we've got eyeball detectors and fur detectors and things, so you really need a lot of channels.

In order to avoid our memory going out of control, from time to time we create a convolution where we don't step over every single set of 3x3, but instead we skip over two at a time. We would start with a 3x3 centered at (2, 2) and then we'd jump over to (2, 4), (2, 6), (2, 8), and so forth. That's called a **stride 2 convolution**. What that does is, it looks exactly the same, it's still just a bunch of kernels, but we're just jumping over 2 at a time. We're skipping every alternate input pixel. So the output from that will be H/2 by W/2. When we do that, we generally create twice as many kernels, so we can now have 32 activations in each of those spots. That's what modern convolutional neural networks tend to look like.

[1:26:26]

We can actually see that if we go into our pets notebook. We grab our CNN, and we're going to take a look at this particular cat:

```
data = get_data(352, 16)
```

```
learn = create_cnn(data, models.resnet34, metrics=error_rate,
```

```
bn_final=True).load('352')
```

```
idx=0
x,y = data.valid_ds[idx]
x.show()
data.valid_ds.y[idx]
```

Category Maine_Coon



so if we go `x,y = data.valid_ds` some index, so it's just grab the 0th, we'll go `.show` and we would print out the value of `y`. Apparently this cat is of category Main Coon. Until a week ago, I was not at all familiar that there's a cat called a Maine Coon. Having spent all week with this particular cat, I am now deeply familiar with this Maine Coon.

```
learn.summary()
```

```
Input Size override by Learner.data.train_dl
Input Size passed in: 16
```

Layer (type)	Output Shape	Param #
Conv2d	[16, 64, 176, 176]	9408
BatchNorm2d	[16, 64, 176, 176]	128
ReLU	[16, 64, 176, 176]	0
MaxPool2d	[16, 64, 88, 88]	0
Conv2d	[16, 64, 88, 88]	36864
BatchNorm2d	[16, 64, 88, 88]	128
ReLU	[16, 64, 88, 88]	0

...

If we go `learn.summary()`, remember that our input we asked for was 352 by 352 pixels, generally speaking, the very first convolution tends to have a stride 2. So after the first layer, it's 176 by 176. `learn.summary()` will print out for you the output shape up to every layer.

The first set of convolutions has 64 activations. We can actually see that if we type in `learn.model`:

```
learn.model
```

```
Sequential(  
    (0): Sequential(  
        (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),  
        bias=False)  
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,  
        track_running_stats=True)  
        (2): ReLU(inplace)  
        (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,  
        ceil_mode=False)  
        (4): Sequential(  
            (0): BasicBlock(  
                (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,  
                1), bias=False)  
                (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,  
                track_running_stats=True)  
                (relu): ReLU(inplace)  
                (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,  
                1), bias=False)  
                (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,  
                track_running_stats=True)  
            )  
            (1): BasicBlock(  
                (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,  
                1), bias=False)  
                (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,  
                track_running_stats=True)  
                (relu): ReLU(inplace)  
                (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,  
                1), bias=False)  
                (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,  
                track_running_stats=True)  
            )
```

You can see here it's a 2D conv with 3 input channels and 64 output channels, and the stride of 2. Interestingly, it actually starts with a kernel size of 7 by 7. Nearly all of the convolutions are 3 by 3. See they are all 3 by 3? For reasons we'll talk about in part 2, we often use a larger kernel for the very first one. If you use a larger kernel, you have to use more padding, so we have to use `kernel_size int divide by 2 padding` to make sure we don't lose anything.

Anyway, we're now have 64 output channels, and since it was stride 2, it's now 176 by 176. Then, as we go along, you'll see that from time to time we halve (e.g. go from 88 by 88 to 44 by 44 grid size, so that was a 2D conv) and then when we do that we generally double the number of channels.

So we keep going through a few more convs and as you can see, they've got batch norm and ReLU, that's kind of pretty standard. And eventually we do it again - another stride 2 conv which again doubles. we now got 512 by 11 by 11. And that's basically where we finish the main part of the network. We end up with 512 channels 11 by 11.

⌚ Manual Convolutions [1:29:24]

We're actually at a point where we're going to be able to do this heat map now. So let's try and work through it. Before we do, I want to show you how you can do your own manual convolutions because it's kind of fun.



```
k = tensor([
    [0., -5/3, 1],
    [-5/3, -5/3, 1],
    [1., 1, 1],
]).expand(1,3,3,3)/6
```

We're going to start with this picture of a Maine Coon, and I've created a convolutional kernel. As you can see, this one has a right edge and a bottom edge with positive numbers, and just inside that, it's got negative numbers. So I'm thinking this should show me bottom-right edges. So that's my tensor.

One complexity is that that 3x3 kernel cannot be used for this purpose, because I need two more dimensions. The first is I need the third dimension to say how to combine the red green and blue. So what I do is I say `.expand`, this is my 3x3 and I pop another three on the start. What `.expand` does is it says create a 3 by 3 by 3 tensor by simply copying this one 3 times. I mean honestly it doesn't actually copy it, it pretends to have copied it but it just basically refers to the same block of memory, so it kind of copies it in a memory efficient way. So this one here is now 3 copies of that:

k

```
tensor([[[[ 0.0000, -0.2778,  0.1667],  
         [-0.2778, -0.2778,  0.1667],  
         [ 0.1667,  0.1667,  0.1667]],  
  
        [[ 0.0000, -0.2778,  0.1667],  
         [-0.2778, -0.2778,  0.1667],  
         [ 0.1667,  0.1667,  0.1667]],  
  
        [[ 0.0000, -0.2778,  0.1667],  
         [-0.2778, -0.2778,  0.1667],  
         [ 0.1667,  0.1667,  0.1667]]]))
```

And the reason for that is that I want to treat red and green and blue the same way for this little manual kernel I'm showing you. Then we need one more axis because rather than actually having a separate kernel like I've kind of drawn these as if they were multiple kernels, what we actually do is we use a rank 4 tensor. The very first axis is for the every separate kernel that we have.

[1:31:30]

In this case, I'm just going to create one kernel. To do a convolution I still have to put this unit axis on the front. So you can see `k.shape` is now `[1, 3, 3, 3]`:

`k.shape`

```
torch.Size([1, 3, 3, 3])
```

It's a 3 by 3 kernel. There are three of them, and then that's just the one kernel that I have. It takes awhile to get the feel for these higher dimensional tensors because we're not used to writing out the 4D tensor, but just think of them like this - the 4D tensor is just a bunch of 3D tensors sitting on top of each other.

So this is our 4D tensor, and then you can just call `conv2d`, passing in some image, and so the image I'm going to use is the first part of my validation data set, and the kernel.

```
t = data.valid_ds[0][0].data; t.shape
```

```
torch.Size([3, 352, 352])
```

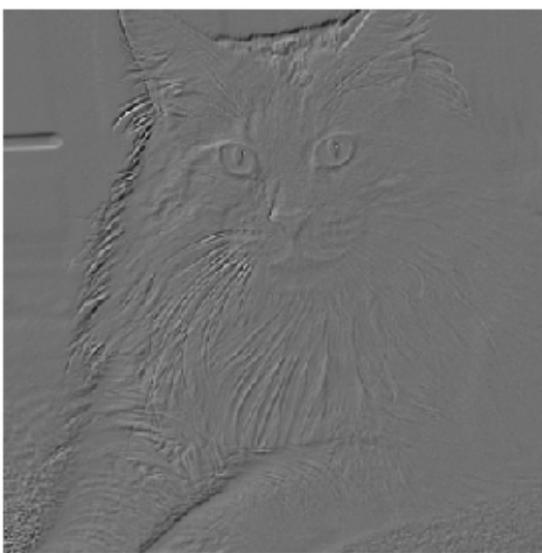
```
t[None].shape
```

```
torch.Size([1, 3, 352, 352])
```

```
edge = F.conv2d(t[None], k)
```

There's one more trick which is that in PyTorch, pretty much everything is expecting to work on a mini-batch, not on an individual thing. So in our case, we have to create a mini-batch of size 1. Our original image is 3 channels by 352 by 352 (height by width). Remember, PyTorch is channel by height by width. I need to create a rank 4 tensor where the first axis is 1. In other words, it's a mini batch of size 1, because that's what PyTorch expects. So there's something you can do in both PyTorch and numpy which is you can index into an array or a tensor with a special value `None`, and that creates a new unit axis in that point. So `t` is my image of dimensions 3 by 352 by 352. `t[None]` is a rank 4 tensor, a mini batch of one image of 1 by 3 by 352 by 352.

```
show_image(edge[0], figsize=(5,5));
```



Now I can go `conv2d` and get back a cat, specifically my Maine Coon. So that's how you can play around with convolutions yourself. So how are we going to do this to create a heat map?

Creating Heat Map [1:33:50]

This is where things get fun. Remember mentioned was that I basically have my input red green blue. It goes through a bunch of convolutional layers (let us write a little line to say a convolutional layer) to create activations which have more and more channels and smaller and smaller height by widths. Until eventually, remember we looked at the summary, we ended up with something which was 11 by 11 by 512. There's a whole bunch more layers that we skipped over.

Now there are 37 classes because `data.c` is the number of classes we have. And we can see that at the end here, we end up with 37 features in our model. So that means that we end up with a probability for every one of the 37 breeds of cat and dog. So it's a vector of length 37 - that's our final output that we need because that's what we're going to compare implicitly to our one hot encoded matrix which will have a 1 in the location for Maine Coon.

So somehow we need to get from this 11 by 11 by 512 to this 37. The way we do it is we actually take the average of every one of these 11 by 11 faces. We just take the mean. We're going to take the mean of this first face, take the mean, that gets this one value. Then we'll take second of the 512 faces, and take that mean, and that'll give us one more value. So we'll do that for every face, and that will give us a 512 long vector.

Now all we need to do is pop that through a single matrix multiply of 512 by 37 and that's going to give us an output vector of length 37. This step here where we take the average of each face is called **average pooling**.



[1:36:52]

Let's go back to our model and take a look.

```

(2): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
)
(1): Sequential(
    (0): AdaptiveConcatPool2d(
        (ap): AdaptiveAvgPool2d(output_size=1)
        (mp): AdaptiveMaxPool2d(output_size=1)
    )
    (1): Lambda()
    (2): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (3): Dropout(p=0.25)
    (4): Linear(in_features=1024, out_features=512, bias=True)
    (5): ReLU(inplace)
    (6): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): Dropout(p=0.5)
    (8): Linear(in_features=512, out_features=37, bias=True)
    (9): BatchNorm1d(37, eps=1e-05, momentum=0.01, affine=True, track_running_stats=True)
)
)

```

Here is our final 512. We will talk about what a concat pooling is in part 2, for now, we'll just focus on that this is a fast.ai specialty. Everybody else just does this

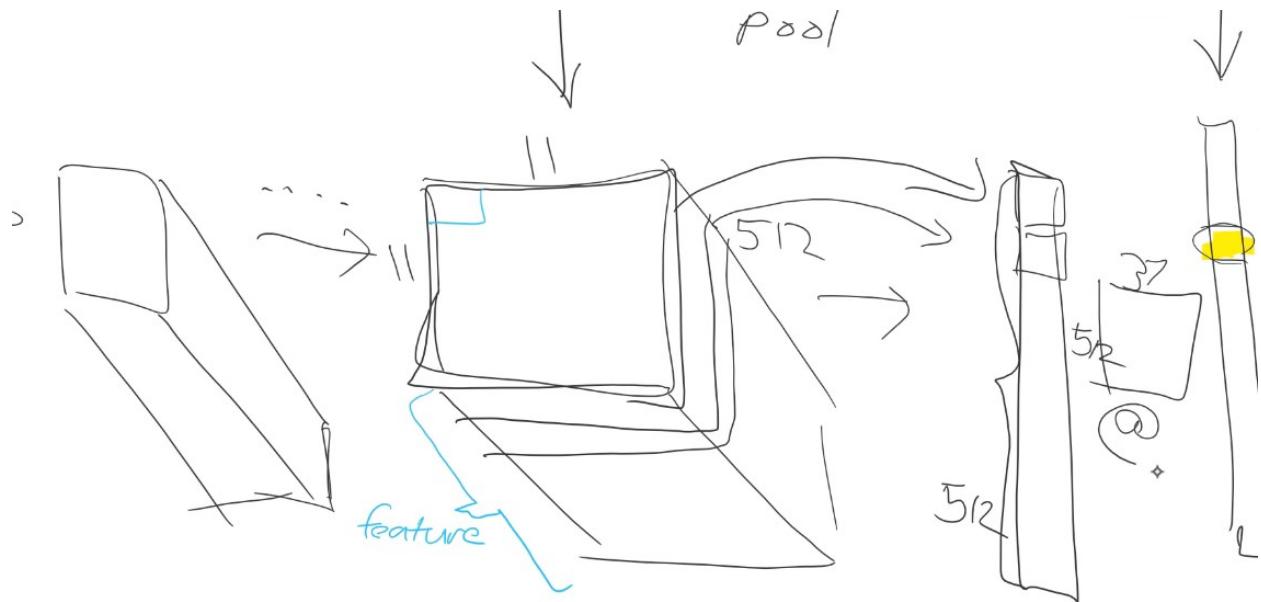
`AdaptiveAvgPool2` with an output size of one.

Again, there's a bit of a special fast.ai thing that we actually have two layers here, but normally people then just have the one `Linear` layer with the input of 512 and the output of 37.



What that means is that, this little box over here (output layer) where we want a one for Maine Coon, we've got to have a box over here (last layer before output) which needs to have a high value in that place so that the loss will be low. So if we're going to have a high value there, the only way to get it is with this matrix multiplication is that it's going to represent a simple weighted linear combination of all of the 512 values here. So if we're going to be able to say I'm pretty confident this is a Maine Coon, just by taking the weighted sum of a bunch of inputs, those inputs are going to have to represent features like how fluffy is it, what color is its nose, how long is its legs, how pointy is its ears - all the kinds of things that can be used. Because for the other thing which figures out is this a bulldog, it's going to use exactly the same kind of 512 inputs with a different set of weights. Because that's all a matrix multiplication is. It's just a bunch of weighted sums - a different weighted sum for each output.

Therefore, we know that this potentially dozens or even hundreds of layers of convolutions must have eventually come up with an 11 by 11 face for each of these features saying in this little bit here, how much is that part of the image like a pointy ear, how much is it fluffy, how much is it like a long leg, how much is it like a very red nodes. That's what all of those things must represent. So each face represents a different feature. The outputs of these we can think of as different features.



What we really want to know then is not so much what's the average across the 11 by 11 to get this set of outputs. But what we really want to know is what's in each of these 11 by 11 spots. So what if instead of averaging across the 11 by 11, let's instead average across the 512. If we average across the 512, that's going to give us a single 11 by 11 matrix and each grid point in that 11 by 11 matrix will be the average of how activated was that area. When it came to figuring out that this was a Maine Coon, how many signs of Maine Coon-ishness was there in that part of the 11 by 11 grid.

That's actually what we do to create our heat map. I think maybe the easiest way is to kind of work backwards. Here's our heat map and it comes from something called average activations (avg_acts).

```
show_heatmap(avg_acts)
```



It's just a little bit of matplotlib and fast.ai.

```
def show_heatmap(hm):
    _,ax = plt.subplots()
    xb_im.show(ax)
    ax.imshow(hm, alpha=0.6, extent=(0,352,352,0),
              interpolation='bilinear', cmap='magma');
```

Fast.ai to show the image, and then matplotlib to take the heat map which we passed in which was called average activations. `hm` for heat map. `alpha=0.6` means make it a bit transparent, and matplotlib `extent` means expand it from 11 by 11 to 352 by 352. Use bilinear interpolation so it's not all blocky, and use a different color map to kind of highlight things. That's just matplotlib - it's not important.

The key thing here is that average activations is the 11 by 11 matrix we wanted. Here it is:

```
avg_acts = acts.mean(0)
avg_acts.shape
```

```
torch.Size([11, 11])
```

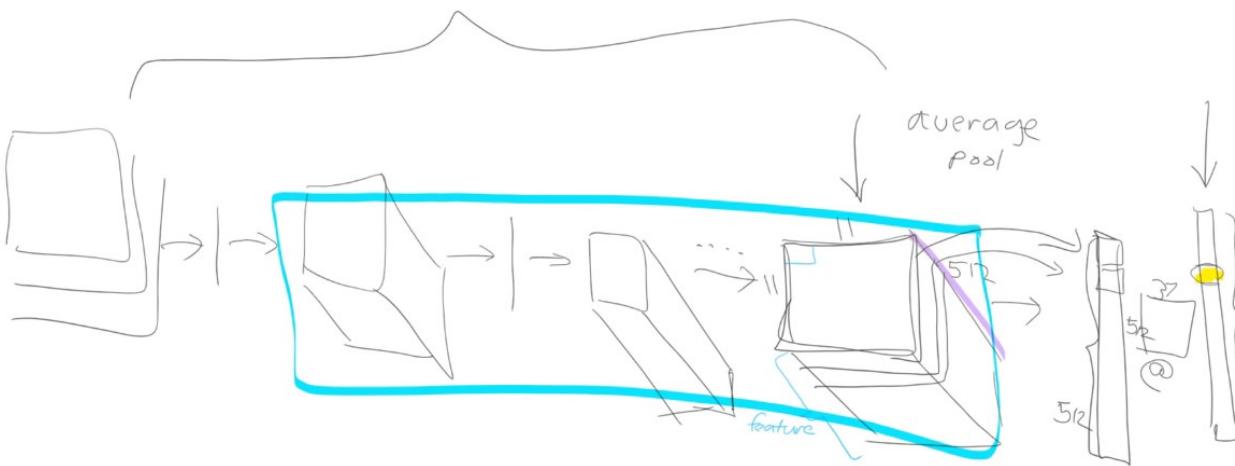
Average activations' shape is 11 by 11. To get there, we took the mean of activations across dimension 0 which is what I just said - in PyTorch, the channel dimension is the first dimension, so the mean across dimension 0 took us from something of size 512 by 11 by 11 to something of 11 by 11. Therefore activations `acts` contains the activations we're averaging. Where did they come from?

```
acts = hook_a.stored[0].cpu()
```

```
acts.shape
```

```
torch.Size([512, 11, 11])
```

They came from something called a hook. A hook is a really cool, more advanced PyTorch feature that lets you (as the name suggests) hook into the PyTorch machinery itself, and run any arbitrary Python code you want to. It's a really amazing and nifty thing. Because normally when we do a forward pass through a PyTorch module, it gives us this set of outputs. But we know that in the process, it's calculated these (512 by 11 by 11 features). So what I would like to do is I would like to hook into that forward pass and tell PyTorch "hey, when you calculate this, can you store it for me please." So what is "this"? This is the output of the convolutional part of the model. So the convolutional part of the model which is everything before the average pool is basically all of that:



So thinking back to transfer learning, remember with transfer learning, we actually cut off everything after the convolutional part of the model, and replaced it with our own little bit. With fast.ai, the original convolutional part of the model is always going to be the first thing in the model. Specifically, it's always going to be called `m[0]`. In this case, I'm taking my model and I'm just going to call it `m`. So you can see `m` is this big thing:

```
In [34]: m = learn.model.eval();
In [35]: m
Out[35]: Sequential(
    (0): Sequential(
        (0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (2): ReLU(inplace)
        (3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
        (4): Sequential(
            (0): BasicBlock(
                (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
                (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
                (relu): ReLU(inplace)
                (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
                (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            )
        (1): BasicBlock(
            (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
            (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (relu): ReLU(inplace)
            (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        )
    )
)
```

But always (at least in fast.ai), `m[0]` will be the convolutional part of the model. So in this case, we created a `resnet34`, so the main part of the ResNet34 (the pre-trained bit we hold on to) is in `m[0]`. So this is basically it. This is a printout of the ResNet34, and at the end of it there is the 512 activations.

```
m = learn.model.eval();

xb, _ = data.one_item(x)
xb_im = Image(data.denorm(xb)[0])
xb = xb.cuda()

from fastai.callbacks.hooks import *

def hooked_backward(cat=y):
    with hook_output(m[0]) as hook_a:
        with hook_output(m[0], grad=True) as hook_g:
            preds = m(xb)
            preds[0,int(cat)].backward()
    return hook_a, hook_g

hook_a, hook_g = hooked_backward()
```

In other words, what we want to do is we want to grab `m[0]` and we want to hook its output:

```
with hook_output(m[0]) as hook_a:
```

This is a really useful thing to be able to do. So fast.ai has actually created something to do it for you which is literally you say `hook_output` and you pass in the PyTorch module that you want to hook the output of. Most likely, the thing you want to hook is the convolutional part of the model, and that's always going to be `m[0]` or `learn.model[0]`.

We give that hook a name (`hook_a`). Don't worry about this part (`with hook_output(m[0], grad=True) as hook_g:`). We'll learn about it next week. Having hooked the output, we now need to actually do the forward pass. So remember, in PyTorch, to actually get it to calculate something (i.e. doing the forward pass), you just act as if the model is a function. We just pass in our X mini-batch.

We already had a Maine Coon image called `x`, but we can't quite pass that into our model. It has to be normalized, turned into a mini batch, and put on to the GPU. Fast.ai has a thing called a data bunch which we have in `data`, and you can always say `data.one_item(x)` to create a mini batch with one thing in it.

As an exercise at home, you could try to create a mini batch without using `data.one_item` to make sure that you learn how to normalize and stuff yourself. if you want to. But this is how you can create a mini batch with just one thing in it. Then I can pop that onto the GPU by saying `.cuda()`. That's what I passed to my model.

The predictions that I get out, I actually don't care about because the predictions is this thing (37 long vector) which is not what I want. So I'm not actually going to do anything with the predictions. The thing I care about is the hook that it just created.

Now, one thing to be aware of is that when you hook something in PyTorch, that means every single time you run that model (assuming you're hooking outputs), it's storing those outputs. So you want to remove the hook when you've got what you want, because otherwise if you use the model again, it's going to keep hooking more and more outputs which will be slow and memory intensive. So we've created this thing (Python calls that a context manager), you can use any hook as a context manager, at the end of that `with` block, it'll remove the hook.

We've got our hook, so now fast.ai hooks (at least the output hooks) always give you something called `.stored` which is where it stores away the thing you asked it to hook. So that's where the activations now are.

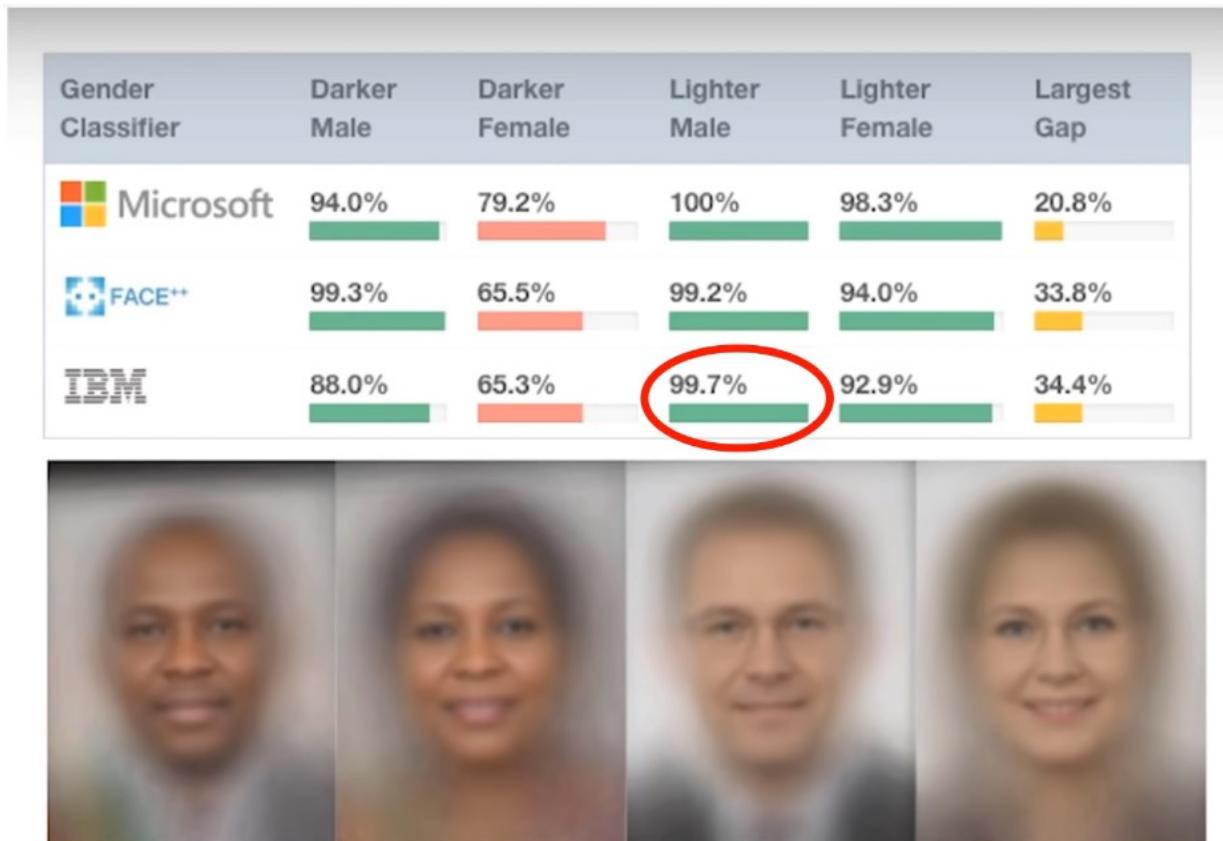
1. We did a forward pass after hooking the output of the convolutional section of the model.
2. We grabbed what it stored.
3. We check the shape - it was 512 by 11 by 11 as we predicted.
4. We then took the mean of the channel axis to get an 11 by 11 tensor.
5. Then, if we look at that, that's our picture.

There's a lot to unpack. But if you take your time going through these two sections; the **Convolution Kernel** section and the **Heatmap** section of this notebook, like running those lines of code and changing them around a little bit, and remember the most important thing to look at is shape. You might have noticed. When I'm showing you these notebooks, I very often print out the shape. And when you look at this shape, you want to be looking at how many axes are there (that's the rank of the tensor) and how many things are there in each axis, and try and think why. Try going back to the print out of the summary, try going back to the actual list of the layers, and try and go back and think about the actual picture we drew, and think about what's actually going on. So that's a lot of technical content, so what I'm going to do now is switch from technical content to something much more important.

In the next lesson, we're going to be looking at generative models both text and image generative models. Generative models are where you can create a new piece of text or a new image or a new video or a new sound. As you probably are aware, this is the area that deep learning has developed the most in the last 12 months. We're now at a point where we can generate realistic looking videos, images, audio, and to some extent even text. There are many things in this journey which have ethical considerations, but perhaps this area of generative modeling is one of the largest ones. So before I got into it, I wanted to specifically touch on ethics and data science.

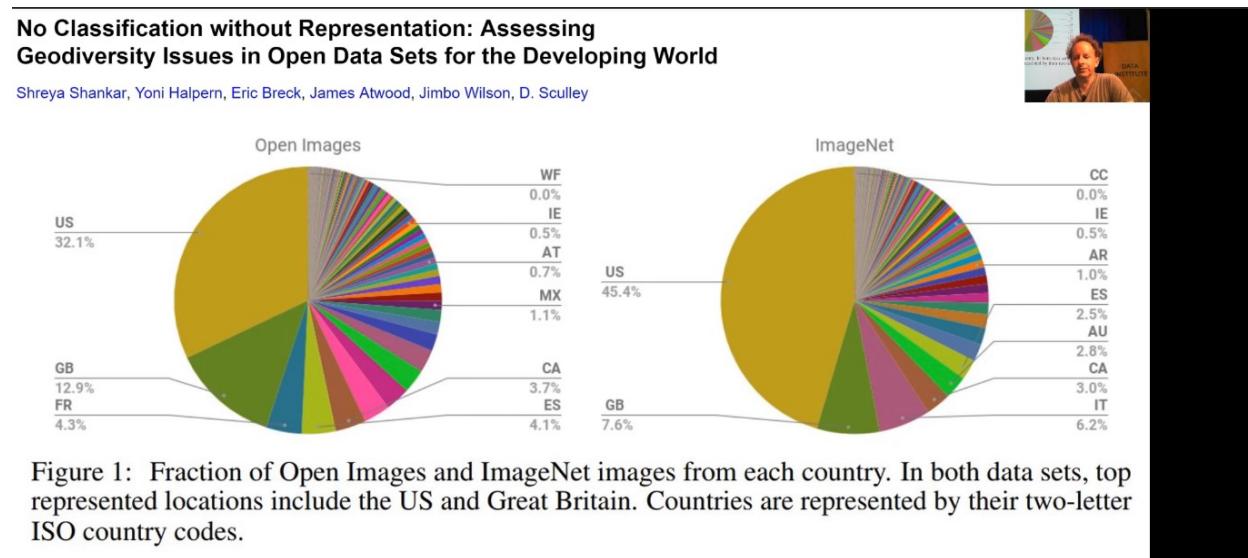
Most of the stuff I'm showing you actually comes from Rachel, and Rachel has a really cool [TEDx San Francisco talk](#) that you can check out on YouTube. A more extensive analysis of ethical principles and bias principles in AI which you can find at this talk [here](#), and she has a playlist that you can check out.

We've already touched on an example of bias which was this gender shades study where, for example, lighter male skin people on IBM's main computer vision system are 99.7% accurate, and darker females are some hundreds of times less accurate in terms of error (so extraordinary differences). It's, first of all, important to be aware that not only can this happen technically, but this can happen on a massive companies rolled out publicly available highly marketed system that hundreds of quality control people have studied and lots of people are using. It's out there in the wild.



Joy Buolamwini & Timnit Gebru

They all look kind of crazy, right? So it's interesting to think about why. One of the reasons why is that the data we feed these things. We tend to use, me included, a lot of these datasets kind of unthinkingly. But like ImageNet which is the basis of like a lot of the computer vision stuff we do is over half American and Great Britain.



When it comes to the countries that actually have most of the population in the world, I can't even see them here. They're somewhere in these impossibly thin lines. Because remember, these datasets are being created almost exclusively by people in US, Great Britain, and nowadays increasingly also China. So there's a lot of bias in the content we're creating, because of a bias in the people that are creating that content even when, *in theory*, it's being created in a very kind of neutral way, but you can't argue with the data. It's obviously not neutral at all.



When you have biased data creating biased algorithms, you then need to ask "what are we doing with that?" We've been spending a lot of time talking about image recognition. A couple of years ago, this company DeepGlint advertised their image recognition system which can be used to do mass surveillance on large crowds of people, find any person passing through who is a person of interest in theory. So putting aside even the question of "is it a good idea to have such a system?" you kind of think "is it a good idea to have such a system where certain kinds of people are 300 times more likely to be misidentified?"

PALANTIR HAS SECRETLY BEEN USING NEW ORLEANS TO TEST ITS PREDICTIVE POLICING TECHNOLOGY



Palantir deployed a predictive policing system in New Orleans that even city council members don't know about

By Ali Winston | Feb 27, 2018, 3:25pm EST

Amazon is selling police departments a real-time facial recognition system

New documents obtained by the ACLU shed light on Amazon's Rekognition project

By Russell Brandom | @russellbrandom | May 22, 2018, 11:06am EDT

Amazon's Face Recognition Falsely Matched 28 Members of Congress With Mugshots

By Jacob Snow, Technology & Civil Liberties Attorney, ACLU of Northern California
JULY 26, 2018 | 8:00 AM

Then thinking about it, so this is now starting to happen in America - these systems are being rolled out. There are now systems in America that will identify a person of interest in a video and send a ping to the local police. These systems are extremely inaccurate, and extremely biased. And what happens then, of course, is if you're in a predominantly black neighborhood where the probability of successfully recognizing you is much lower, and you're much more likely to be surrounded by black people, and so suddenly all of these black people are popping up as persons of interest, or in a video of a person of interest. All the people in the video are all recognized as in the vicinity of a person of interest, you suddenly get all these pings going off the local police department causing the police to run down there. Therefore likely to lead to a larger number of arrests, which is then likely to feed back into the data being used to develop the systems.

So this is happening right now. Thankfully, a very small number of people are actually bothering to look into these things. I mean ridiculously small, but at least it's better than nothing. One of the best ways that people get publicity is to do "funny" experiments like let's try the mug shot image recognition system that's being widely used and try it against the members of Congress, and find out that there are 28 black members of Congress who would have been identified by this system (obviously incorrectly).



She is a doctor.
He is a nurse.

O bir doktor.
O bir hemşire.



31/5000



O bir doktor.
O bir hemşire

He is a doctor.
She is a nurse ✓



28/5000

Google

We see this kind of bias in a lot of the systems we use, not just image recognition but text translation when you convert "She is a doctor. He is a nurse" into Turkish, you quite correctly get a gender inspecific pronoun because that's what Turkish uses. You could then take that and feed it back into Turkish with your gender in specific pronoun, and you will now get "He is a doctor. She is a nurse." So the bias, again, this is in a massively widely rolled out, carefully studied system. It's not like even these kind of things (a little one-off things) then get fixed quickly, these issues have been identified in Google Translate for a very long time, and they're still there. They don't get fixed.

The kind of results of this are, in my opinion, quite terrifying. Because what's happening is that in many countries including America where I'm speaking from now, algorithms are increasingly being used for all kinds of public policy, judicial, and so forth purposes.



PRO PUBLICA



Machine Bias

There's software used across the country to predict future criminals. And it's biased against blacks.

Prediction Fails Differently for Black Defendants

	WHITE	AFRICAN AMERICAN
Labeled Higher Risk, But Didn't Re-Offend	23.5%	44.9%
Labeled Lower Risk, Yet Did Re-Offend	47.7%	28.0%

For example, there's a system called COMPAS which is very widely used to decide who's going to jail. It does that in a couple of ways; it tells judges what sentencing guidelines they should use for particular cases, and it tells them also which people (the system says) should be let out on bail. But here's the thing. White people, it keeps on saying let this person out even though they end up reoffending, and vice versa. It's systematically out by double compared to what it should be in terms of getting it wrong with white people versus black people. So this is kind of horrifying because, amongst other things, the data that it's using in this system is literally asking people questions about things like "did any of your parents ever go to jail?" or "do any of your friends do drugs?" They're asking questions about other people who they have no control over. So not only are these systems very systematically biased, but they're also being done on the basis of data which is totally out of your control. "Are your parents divorced?" is another question that's being used to decide whether you go to jail or not.

When we raise these issues on Twitter or in talks or whatever, there's always a few people (always white men) who will always say like "that's just the way the world is." "That's just reflecting what the data shows." But when you actually look at it, it's not. It's actually systematically erroneous. And systematically erroneous against people of color, minorities - the people who are less involved in creating the systems that these products are based on.

Sometimes this can go a really long way. For example, in Myanmar there was a genocide of Rohingya people. That genocide was very heavily created by Facebook. Not because anybody at Facebook wanted it, I mean heavens no, I know a lot of people at Facebook. I have a lot of friends at Facebook. They're really trying to do the right thing. They're really trying to create a product that people like, but not in a thoughtful enough way. Because when you roll out something where literally in Myanmar - a country that maybe half of people didn't have electricity until very recently. And you say "hey, you can all have free internet as long as it's just Facebook", you must think carefully about what you're doing. Then you use algorithms to feed people the stuff they will click on. Of course what people click on is stuff which is controversial, stuff that makes their blood boil. So when they actually started asking the generals in the Myanmar army that were literally throwing babies onto bonfires, they were saying "we know that these are not humans. We know that they are animals, because we read the news. We read the internet." Because this is the stories that the algorithms are pushing. The algorithms are pushing the stories, because the algorithms are good. They know how to create eyeballs, how to get people watching, and how to get people clicking. Again, nobody at Facebook said let's cause a massive genocide in Myanmar. They said let's maximize the engagement of people in this new market on our platform.

TIMOTHY MCLAUGHLIN 07.06.18 7:00 AM

HOW FACEBOOK'S RISE FUELED CHAOS AND CONFUSION IN MYANMAR



Inside Facebook's Myanmar operation

Hatebook

Why Facebook is losing the war on hate speech in Myanmar

"That's not 20/20 hindsight. The scale of this problem was significant and it was already apparent."

NEWS / ROHINGYA

UN: Facebook had a 'role' in Rohingya genocide

'I'm afraid that Facebook has now turned into a beast.'

13 Mar 2018 [f](#) [t](#)

They very successfully maximized engagement. It's important to note people warned executives of Facebook how the platform was being used to incite violence as far back as 2013, 2014, 2015. And 2015, someone even warned executives that Facebook could be used in Myanmar in the same way that the radio broadcast were used in Rwanda during the Rwandan genocide. As of 2015, Facebook only had four contractors who spoke Burmese working for them. They really did not put many resources into the issue at all, even though they were getting very very alarming warnings about it.

Ethics is complicated

- Many ethical issues are complex and don't have clear or easy answers.
- I'm not here to tell you what to do or provide answers.
- It's good to think about how you'd handle a situation BEFORE you are in it.
- Volkswagen engineer sentenced to prison for creating software to cheat on emissions tests (he was following his boss's orders)

So why does this happen? The part of the issue is that ethics is complicated, and you will not find Rachel or I telling you how to do ethics, how do you fix this - we don't know. We can just give you things to think about. Another part of a problem we keep hearing is, it's not my problem, I'm just a researcher, I am just a techie, I'm just building a data set, I'm not part of a problem, I'm part of this foundation that's far enough away that I can imagine that I'm not part of this. But if you're creating ImageNet, and you want it to be successful, you want lots of people to use it, you want lots of people to build products on it, lots people to do research on top of it. If you're trying to create something that people are using, you want them to use, then please try to make it something that won't cause massive amounts of harm, and doesn't have massive amounts of bias.

It can actually come back and bite you in the arse. The Volkswagen engineer who ended up actually encoding the thing that made them systematically cheat on their diesel emissions tests on their pollution tests ended up in jail. Not because it was their decision to cheat on the tests, but because their manager told them to write their code, and they wrote the code. Therefore they were the ones that ended up being criminally responsible, and they were the ones that were jailed. So if you do, in some way, a crappy thing that ends up causing trouble, that can absolutely come back around and get you in trouble as well.

In the concentration camps, IBM's code for Jews was 8. Its code for Gypsies was 12. General executions were coded as 4, death in the gas chambers as 6. Only Jews and Gypsies were systematically murdered in [gas chambers](#).

The Swiss judge ruled "It does not thus seem unreasonable to deduce that IBM's technical facilitated the tasks of the Nazis in the commission of their crimes against humanity, acts also involving accountancy and classification by IBM machines and utilized in the concentration camps themselves."

The Nazi Party: IBM & "Death's Calculator"
by Edwin Black

Sometimes it can cause huge amounts of trouble. If we go back to World War II, then this was one of the first great opportunities for IBM to show off their amazing tabulating system. And they had a huge client in Nazi Germany. And Nazi Germany used this amazing new tabulating system to encode all of the different types of Jews that they had in the country and all the different types of problem people. So Jews were 8, gypsies were 12, then different outcomes were coded; executions were 4, death in a gas chamber was 6.

A Swiss judge ruled that IBM was actively involved facilitating the commission of these crimes against humanity. There are absolutely plenty of examples of people building data processing technology that are directly causing deaths, sometimes millions of deaths. We don't want to be one of those people. You might have thought "oh you know, I'm just creating some data processing software" and somebody else is thinking "I'm just the sales person" and somebody else is thinking "I'm just the biz dev person opening new markets." But it all comes together. So we need to care.

Myth of neutral platforms



- Revenue model
- Design: do friends' posts, sponsored content, etc have same format
- Controls and filters available to users, available to advertisers
- Experiments
- Whether to have human moderators & how many
- Who gets banned

Facebook fires human editors, algorithm immediately posts fake news

Facebook makes its Trending feature fully automated, with mixed results.

ANNALEE NEWITZ - 8/29/2016, 11:20 AM

One of the things we need to care about is getting humans back in the loop. When we pull humans out of the loop is one of the first times that trouble happens. I don't know if you remember, I remember this very clearly when I first heard that Facebook was firing the human editors that were responsible for basically curating the news that ended up on the Facebook pages. And I've gotta say, at the time, I thought that's a recipe for disaster. Because I've seen again and again that humans can be the person in the loop that can realize this isn't right. It's very hard to create an algorithm that can recognize "this isn't right." Or else, humans are very good at that. And we saw that's what happened. After Facebook fired the human editors, the nature of stories on Facebook dramatically changed. You started seeing this proliferation of conspiracy theories, and the kind of the algorithms went crazy with recommending more and more controversial topics. And of course, that changed people's consumption behavior causing them to want more and more controversial topics.

One of the really interesting places this comes in, Cathy O'Neil (who's got a great book called Weapons of Math Destruction) and many others have pointed out. What happens to algorithms is that they end up impacting people. For example, COMPAS sentencing guidelines go to a judge. Now you can say the algorithm is very good. I mean in COMPAS' case, it isn't. It actually turned out to be about as bad as random because it's a black box and all that. But even if it was very good, you could then say "well, the judge is getting the algorithm; otherwise they're just be getting a person - people also give bad advice. So what?" Humans respond differently to algorithms. It's very common, particularly for a human that is not very familiar with the technology themselves like a judge to see like "oh that's what the computer says." The computer looked it up and it figured this out.

It's extremely difficult to get a non-technical audience to look at a computer recommendation and come up with a nuanced decision-making process. So what we see is that algorithms are often put into place with no appeals process. They're often used to massively scale up decision making systems because they're cheap. Then the people that are using those algorithms tend to give them more credence than they deserve because very often they're being used by people that don't have the technical competence to judge them themselves.

Algorithms are used differently than human decision makers:

- Algorithms are more likely to be implemented with **no appeals process** in place.
- Algorithms are often used **at scale**.
- Algorithmic systems are **cheap**.
- People are more likely to assume algorithms are **objective or error-free** (even if they're given the option of a human override)



The privileged are processed by people; the poor are processed by algorithms. (Cathy O'Neil)

A great example was here's an example of somebody who lost their health care. They lost their health care because of an error in a new algorithm that was systematically failing to recognize that there are many people that need help with cerebral palsy and diabetes. So this system which had this error that was later discovered was cutting off these people from the home care that they needed. So that cerebral palsy victims no longer had the care they needed. So their life was destroyed basically.

When the person that created that algorithm with the error was asked about this and more specifically said should they have found a better way to communicate the system, the strengths, the failures, and so forth, he said "yeah, I should probably also dust under my bed." That was the level of interest they had.

This is extremely common. I hear this all the time. And it's much easier to see it from afar and say "okay, after the problems happened I can see that that's a really shitty thing to say." But it can be very difficult when you're kind of in the middle of it.

[Rachel] I just want to say one more thing about that example. This was a case where it was separate; there was someone who created the algorithm, then I think different people implemented the software, and this is now in use in over half of the 50 states, then there was also the particular policy decisions made by that state. So this is one of those situations where nobody felt responsible because the algorithm creators are like "oh no, it's the policy decisions of the state that were bad." And the state can be like "oh no, it's the ones who implemented the software" and so everyone's just kind of pointing fingers and not taking responsibility.

And you know, in some ways maybe it's unfair, but I would argue the person who is creating the data set and the person who is implementing the algorithm is the person best placed to get out there and say "hey here are the things you need to be careful of" and make sure that they are part of the implementation process.

How Algorithms Can Learn to Discredit the Media

Defamation is efficient, and AIs may have already figured it out

Guillaume Chaslot



AIs Are Designed to Maximize Watch Time

At YouTube, we used a complex AI to pursue a simple goal: maximize watch time. Google explains this focus in [the following statement](#):

If viewers are watching more YouTube, it signals to us that they're happier with the content they've found. It means that creators are attracting more engaged audiences. It also opens up more opportunities to generate revenue for our partners.

How an ex-YouTube insider investigated its secret algorithm

We've also seen this with YouTube. It's similar to what happened with Facebook and we've heard examples of students watching the fast.ai courses who say "hey Jeremy and Rachel, watching the fast.ai courses, really enjoyed them and at the end of one of them the YouTube autoplay fed me across to a conspiracy theory." What happens is that once the system decides that you like the conspiracy theories, it's going to just feed you more and more.

[Rachel] Just briefly. You don't even have to like conspiracy theories. The goal is to get as many people hooked on conspiracy theories as possible is what the algorithms trying to do whether or not you've expressed interest.

The interesting thing again is I know plenty of people involved in YouTube's recommendation systems. None of them are wanting to promote conspiracy theories. But people click on them, and people share them, and what tends to happen is also people that are into conspiracy theories consume a lot more YouTube media. So it actually is very good at finding a market that watches a lot of hours of YouTube and then it makes that market watch even more.

THE WALL STREET JOURNAL

How YouTube Drives People to the Internet's Darkest Corners

Google's video site often recommends divisive or misleading material, despite recent changes designed to fix the problem

"YouTube may be the most powerful radicalizing instrument of the 21st century." – Zeynep Tufekci, The New York Times

How Algorithms Can Learn to Discredit the Media

How an ex-YouTube insider investigated its secret algorithm

So this is an example of a feedback loop. The New York Times is now describing YouTube as perhaps the most powerful radicalizing instrument of the 21st century. I can tell you my friends that worked on the YouTube recommendation system did not think they were creating the most powerful radicalizing instrument of the 21st century. And to be honest, most of them today when I talk to them still think they're not. They think it's all bull crap. Not all of them, but a lot of them now are at the point where they just feel like they're the victims here, people are unfairly ... you know, they don't get it, they don't understand what we're trying to do. It's very very difficult when you are right out there in the heart of it.

So you've got to be thinking from right at the start. What are the possible unintended consequences of what you're working on? And as the technical people involved, how can you get out in front and make sure that people are aware of them.

[Rachel] I just also need to say that in particular, many of these conspiracy theories are promoting white supremacy, they're kind of far-right ethno-nationalism, anti-science, and i think maybe five or ten years ago, I would have thought conspiracy theories are more fringe thing, but we're seeing huge societal impact it can have for many people to believe these.

And you know, partly it's you see them on YouTube all the time, it starts to feel a lot more normal. So one of the things that people are doing to try to say how to fix this problem is to explicitly get involved in talking to the people who might or will be impacted by the kind of decision making processes that you're enabling.

Talk to domain experts & those impacted.



A screenshot of a video player interface. The title bar reads "Secure | https://fatconference.org/2018/livestream_vh210.html". Below the title bar, there are tabs for "FAT* Conference 2019" and "2018". To the right of the tabs are links for "Organization" and "Resources". The main content area shows a presentation slide titled "Tutorials - Translating to Computer Science - Vanderbilt Hall 210" and "FAT Breakout Room_210". The slide features a flowchart illustrating the bail process. The flowchart starts with "Stopped by NYPD", followed by "Arrested", "Lawyer Interview", and "Arraignment/Bail Hearing". From the hearing, two paths emerge: "BAIL SET" (leading to "Transported to jail", "Wait 48 hrs for next appearance", "Prepare for trial from jail", and finally "Trial, Dismiss, or Plea") and "RELEASED" (leading directly to "Go home"). Below the slide, a video frame shows a person speaking to an audience in a lecture hall. The video player has a progress bar at the bottom indicating "34:13 / 2:06:44".

Kristian Lum,
Elizabeth Bender, &
Terrence Wilkerson

For example, there was a really cool thing recently where literally statisticians and data scientists got together with people who had been inside the criminal system (i.e. had gone through the bail and sentencing process of criminals themselves) and talking to the lawyers who worked with them, and put them together with the data scientists, and actually put together a timeline of how exactly does it work, and where exactly the other places that there are inputs, and how do people respond to them, and who's involved.

This is really cool. This is the only way for you as a data product developer to actually know how your data product is going to be working.



Choosing NOT to just maximize a metric

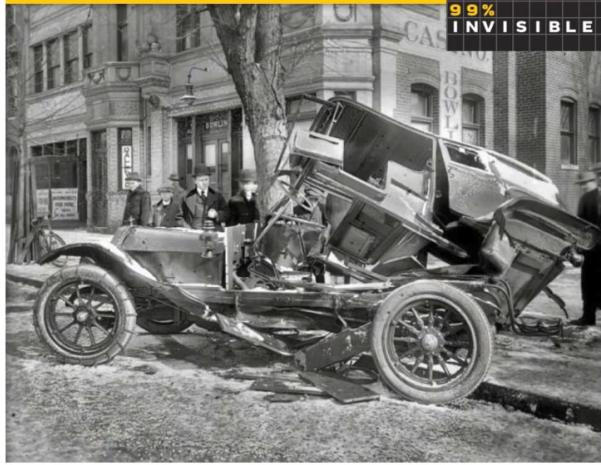
The video player displays a presentation slide with the title "When Recommendation Systems Go Bad" in large red font. Below the title, it says "Evan Estola 5/20/16". The video progress bar shows 0:06 / 23:39. The video controls include play, pause, volume, and full-screen icons. The overall interface is white with black side bars.

Evan Estola - When Recommendations Systems Go Bad - MLconf SEA 2016

A really great example of somebody who did a great job here was Evan Estola at Meetup who said "hey, a lot of men are going to our tech meetups and if we use a recommendation system naively, it's going to recommend more tech meetups to men, which is going to cause more men to go to them, and then when women do try to go, they'll be like "oh my god, there's so many men here" which is going to cause more men to go to the tech meetups. So showing recommendations to men, and therefore not showing them to women.

What Evan and meetup decided was to make an explicit product decision that this would not even be representing the actual true preferences of people. It would be creating a runaway feedback loop. So let's explicitly stop it before it happens, and not recommend less tech meetups women and more tech meetups to men. So I think it's really cool. It's like it's saying, we don't have to be slaves to the algorithm. We actually get to decide.

The Nut Behind the Wheel

99%
INVISIBLE

Datasheets for Datasets*

Timnit Gebru¹, Jamie Morgenstern², Briana Vecchione³,
 Jennifer Wortman Vaughan¹, Hanna Wallach¹,
 Hal Daumé III^{1,4}, and Kate Crawford^{1,5}

Early cars:

- sharp metal knobs on dashboard that could lodge in people's skulls in crash
- non-collapsible steering columns would frequently impale drivers
- belief that cars were dangerous because of **the people** driving them

Another thing that people can do to help is regulation. Normally, when we talk about regulation, there's a natural reaction of like "how do you regulate these things? That's ridiculous - you can't regulate AI." But actually when you look at it, again and again, and this fantastic paper called [Datasheets for Datasets](#) has lots of examples of this. There are many many examples of industries where people thought they couldn't be regulated, people thought that's just how it was. Like cars. People died in cars all the time because they literally had sharp metal knobs on dashboards, steering columns weren't collapsible, and all of the discussion in the community was "that's just how cars are" and when people died in cars, it's because of the people. But then eventually the regulations did come in. And today, driving is dramatically safer - dozens and dozens of times safer than it was before. So often there are things we can do through policy.

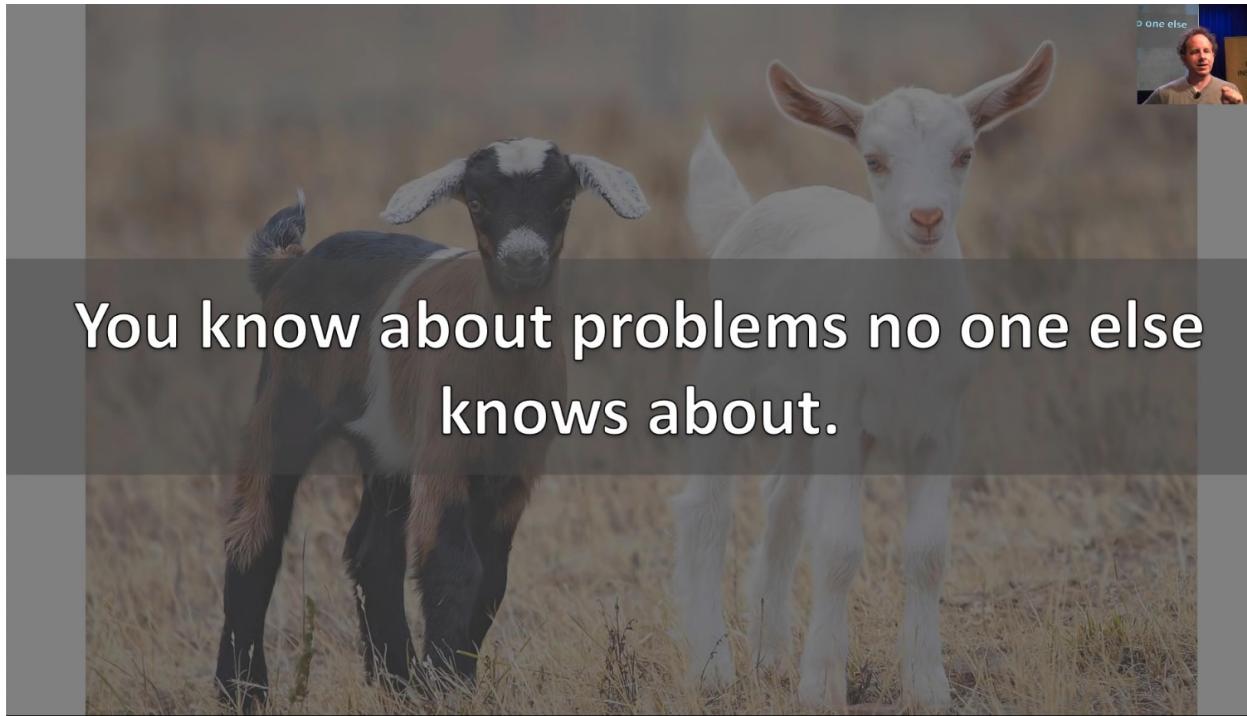


Let's try to make the world a *better* place.

Only 0.3-0.5% of the global population knows how to code.



To summarize, we are part of the 0.3 to 0.5% of the world that knows how to code. We have a skill that very few other people do. Not only that, we now know how to code deep learning algorithms which is like the most powerful kind of code I know. So I'm hoping that we can explicitly think about at least not making the world worse, and perhaps explicitly making it better.



So why is this interesting to you as an audience in particular? That's because fast.ai in particular is trying to make it easy for domain experts to use deep learning. This picture of the goats here is an example of one of our international fellows from a previous course who is a goat dairy farmer and told us that they were going to use deep learning on their remote Canadian island to help study udder disease in goats. To me, this is a great example of a domain experts problem which nobody else even knows about, let alone know that it's a computer vision problem that can be solved with deep learning. So in your field, whatever it is, you probably know a lot more now about the opportunities in your field to make it a hell of a lot better than it was before. You're probably able to come up with all kinds of cool product ideas, maybe build a startup or create a new product group in your company or whatever. But also, please be thinking about what that's going to mean in practice, and think about where can you put humans in the loop? Where can you put those pressure release valves? Who are the people you can talk to who could be impacted who could help you understand? And get the humanities folks involved to understand history and psychology and sociology and so forth.

That's our plea to you. If you've got this far, you're definitely at a point now where you're ready to make a serious impact on the world, so I hope we can make sure that that's a positive impact. See you next week!



hiromis Lesson 7.

e68f003 on Jan 21

1 contributor

2586 lines (1940 sloc) 171 KB

Raw

Blame

History



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.

The screenshot shows the Google Play Store interface for the "Food Classifier" app. The app is categorized under Education and is rated Everyone. It has a compatibility note indicating it's compatible with all devices. The main image features a slice of pizza being examined through a magnifying glass. Below the image, there are two screenshots of the app's interface, both titled "What Food Is It?". The left screenshot shows a camera view with a food item, and the right screenshot shows a list of predictions with "baklava" as the most likely result. At the bottom, there's a comment from user "reshama" about the mobile app.

Food Classifier

TJND Education

Everyone

This app is compatible with all of your devices.

Add to Wishlist Install

reshama Reshma Shaikh

Food Classifier: Mobile App

Here is a 2-minute [video demonstration](#) of our food classifier app on mobile, developed by @npatta01 and I.

We are working on a blog with instructions. We were able to deploy it on Android without any cost.

Predictions

Most likely: *baklava*

Other possibilities

output: 14671.4	output: 571.5
output: 424.7	

I wanted to start by showing some cool work done by a couple of students; Reshma 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 Reshma 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, Reshma 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'` would 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),
Image (1, 28, 28)]...
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.

```
i1[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.

```
sd
```

```
ItemLists;

Train: ImageItemList (60000 items)
[Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28), Image (1, 28, 28),
Image (1, 28, 28)]...
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),
Image (1, 28, 28)]...
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),
Image (1, 28, 28)]...
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),
Image (1, 28, 28)]...
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.

```
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()
```

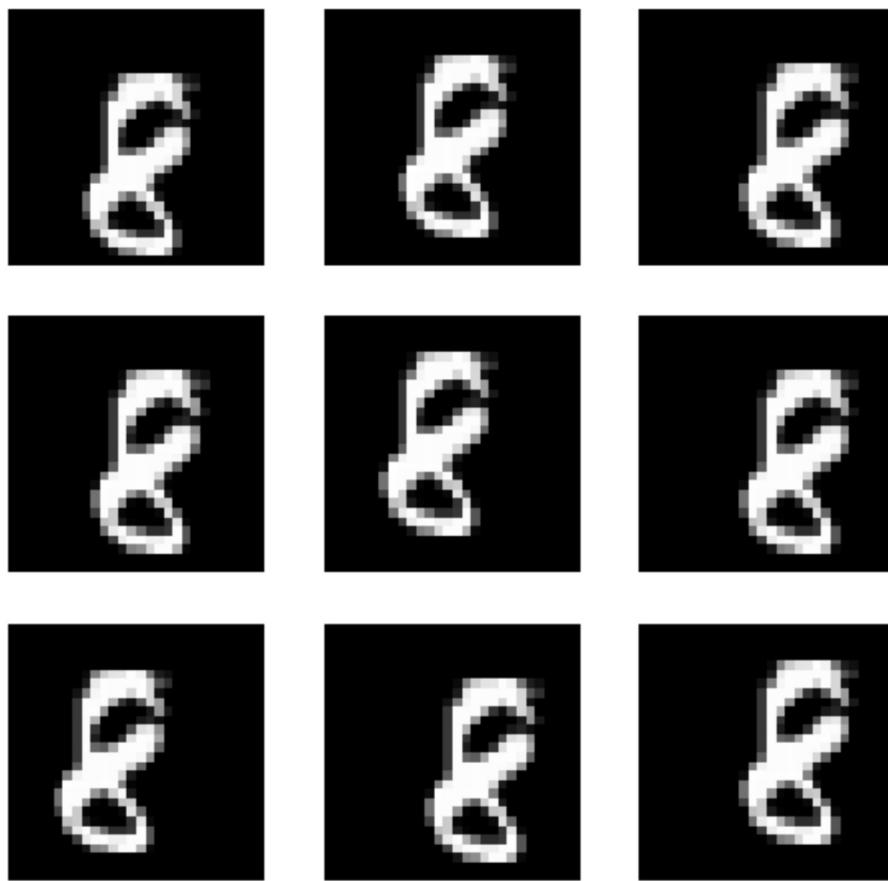
We can choose `normalize`. In this case, we're not using a pre-trained model, so there's no reason to use ImageNet stats here. So if you call `normalize` like this without passing in stats, it will grab a batch of data at random and use that to decide what normalization stats to use. That's a good idea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

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



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.ceil` 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()
```

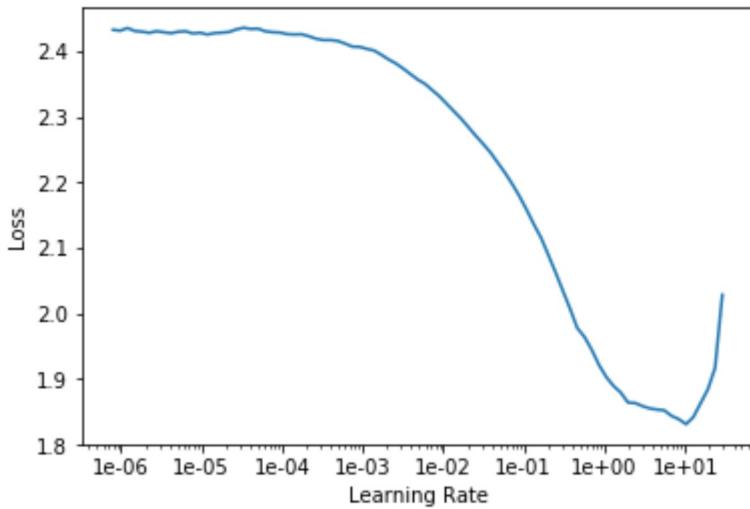
```
model(xb).shape
```

```
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.

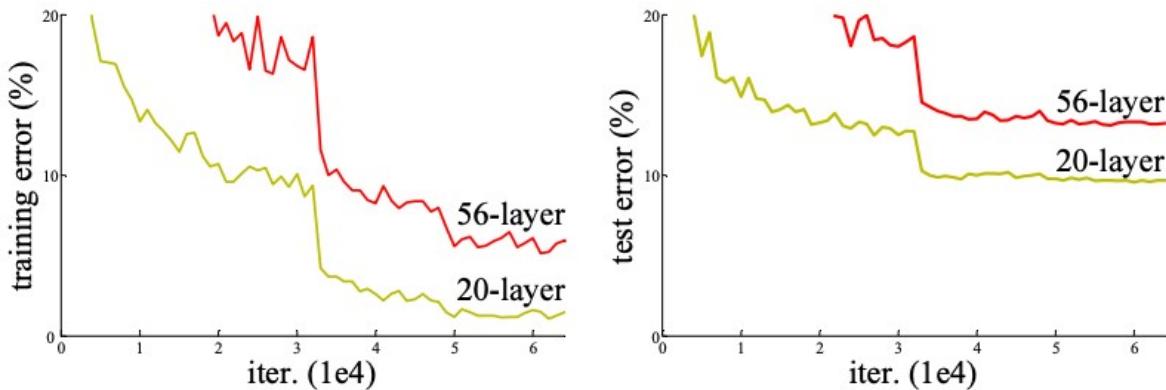


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

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:

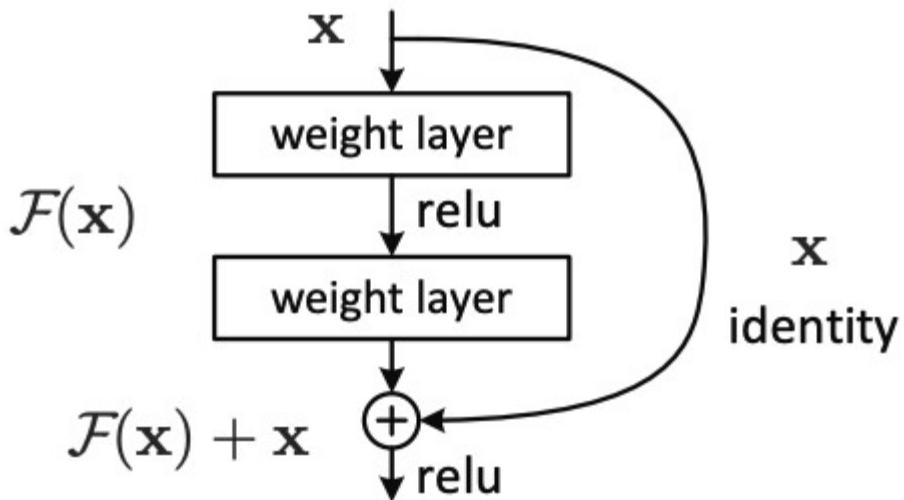


Figure 2. Residual learning: a building block.

Every time 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:

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

Instead, he's saying:

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

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.

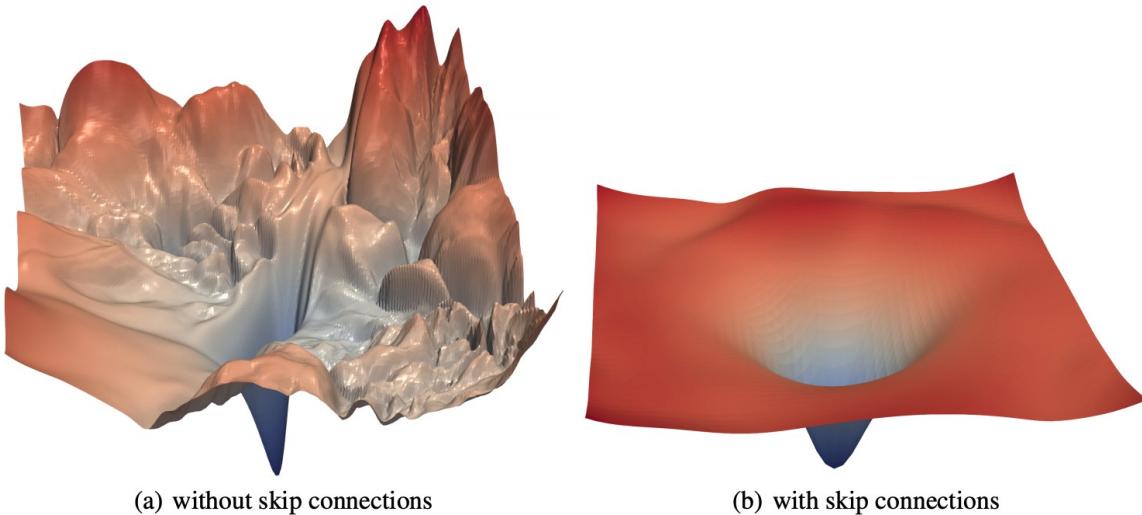


Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

The big picture was this one. Here's what happens if you 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]=<NormType.Batch: 1>, bottle:bool=False, **kwargs)
    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()  
)
```

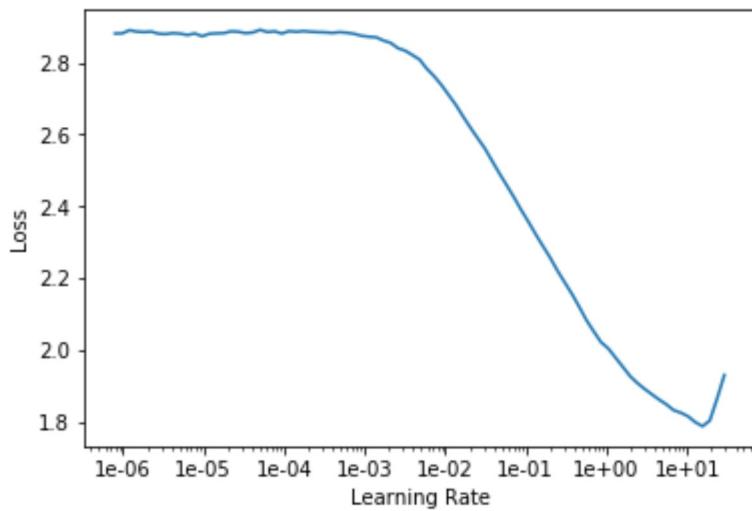
Keep refactoring your architectures if you're trying novel architectures because you'll make less mistakes. Very few people do this. Most research code you look at is clunky as all heck and people often make mistakes in that way, so don't do that. You're all coders, so use your coding skills to make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

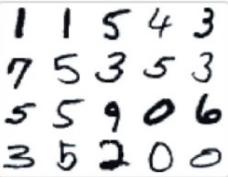
epoch	train_loss	valid_loss	accuracy
5	0.074078	0.063749	0.980800
6	0.057730	0.090142	0.973800
7	0.054202	0.034091	0.989200
8	0.043408	0.042037	0.986000
9	0.033529	0.023126	0.992800
10	0.023253	0.017727	0.994400
11	0.019803	0.016165	0.994900
12	0.023228	0.015396	0.995400

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

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

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)

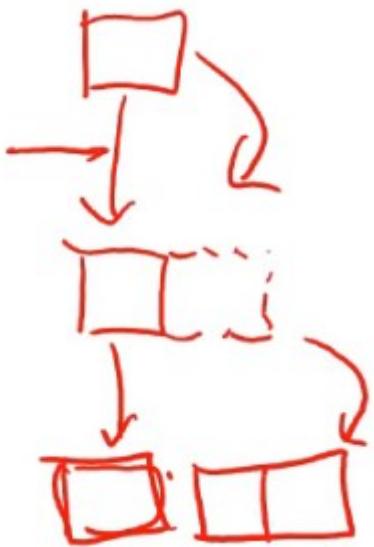
def res_block(nf, dense:bool=False, norm_type:Optional[NormType]=NormType.Batch,
             "Resnet block of `nf` features.")
    norm2 = norm_type
    if not dense and (norm_type==NormType.Batch): norm2 = NormType.BatchZero
    nf_inner = nf//2 if bottle else nf
    return SequentialEx(conv_layer(nf,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

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



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

¹Montreal Institute for Data Science, University of Montreal, Montréal, Québec, Canada

²Image & Vision Group, University of Guelph, Waterloo, Ontario, Canada

³Image & Vision Group, University of Guelph, Waterloo, Ontario, Canada

⁴Image & Vision Group, University of Guelph, Waterloo, Ontario, Canada

Model	Pretrained	# parameters (M)
SegNet [1]	✓	29.5
Ravensian SeeNet [5]	✓	29.5

learn.fit_one_cycle(10, lr)

Total time: 04:57

epoch	train_loss	valid_loss	acc_camvid
1	0.163259	0.226014	0.939663
2	0.159221	0.223871	0.940497

Sidewalk	Cyclist	Mean IoU	Global accuracy
60.5	24.8	46.4	62.5
63.1	26.9	46.9	66.9

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

Model	Pretrained	# parameters (M)
DeepLab-LFOV [5]	✓	37.3
Dilation8 [37]	✓	140.8
Dilation8 + FSO [17]	✓	140.8
Classic Upsampling	✗	20
FC-DenseNet56 (k=12)	✗	1.5
FC-DenseNet67 (k=16)	✗	3.5
FC-DenseNet103 (k=16)	✗	9.4

Table 3. Result

epoch	train_loss	valid_loss	acc_camvid
3	0.150215	0.227715	0.941239
6	0.155152	0.226728	0.941032
7	0.150818	0.230083	0.940657
8	0.149479	0.229187	0.940948
9	0.148236	0.229072	0.941316
10	0.148074	0.234124	0.940629

CN8 model

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.

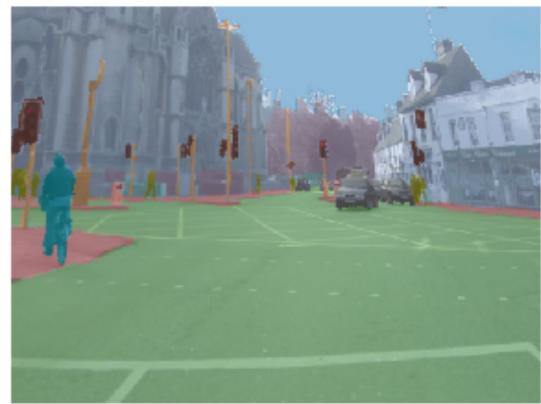
What we're going to do to get there is we're going to use this U-Net. We've used a U-Net before. We used it when we did the CamVid segmentation but we didn't understand what it was doing. So we're now in a position where we can understand what it was doing. The first thing we need to do is to understand 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=classes))
```

```
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?

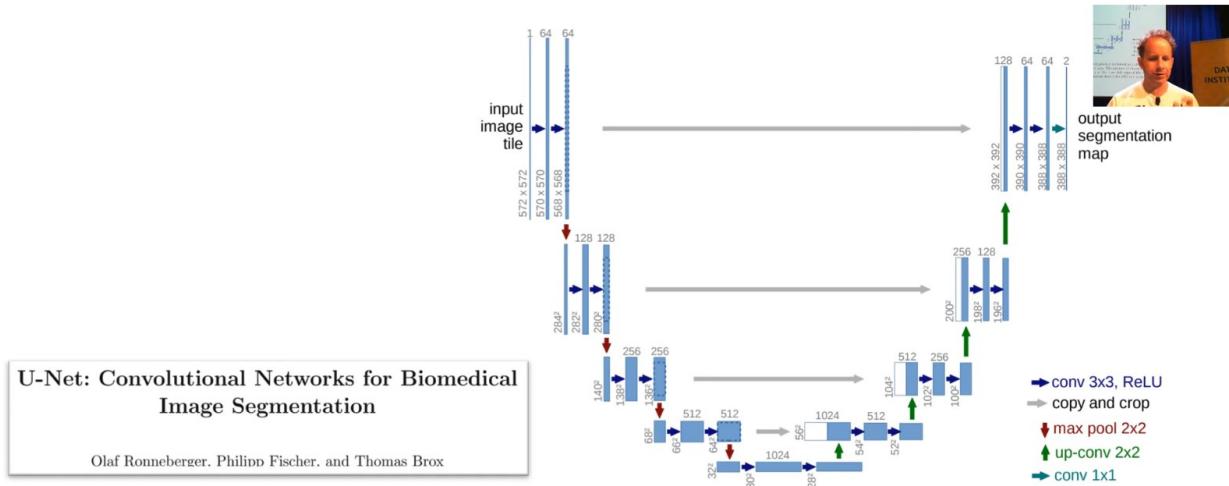


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows 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 keep grid size or a stride 2 conv to do computation and halve the grid size.

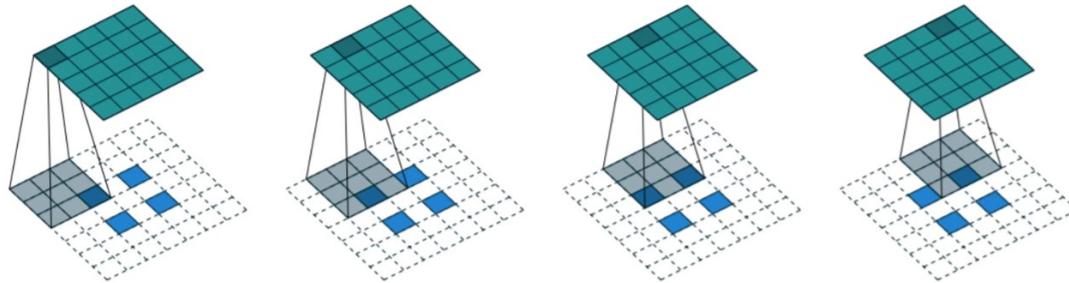
[35:58]

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**.

A guide to convolution arithmetic for deep learning

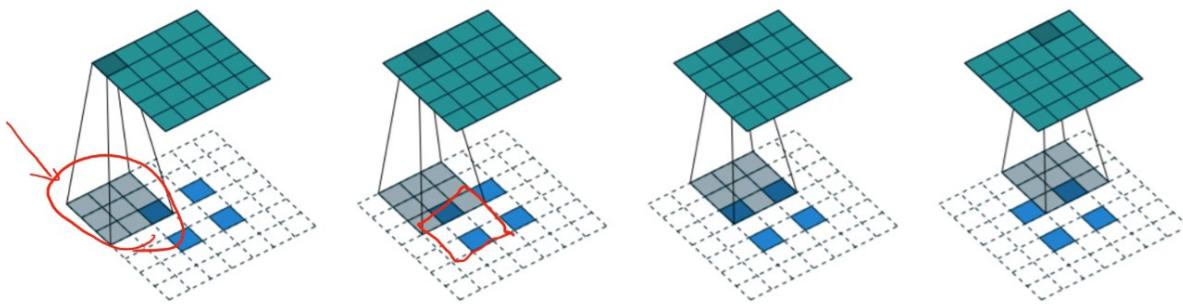


Vincent Dumoulin^{1★} and Francesco Visin^{2†}



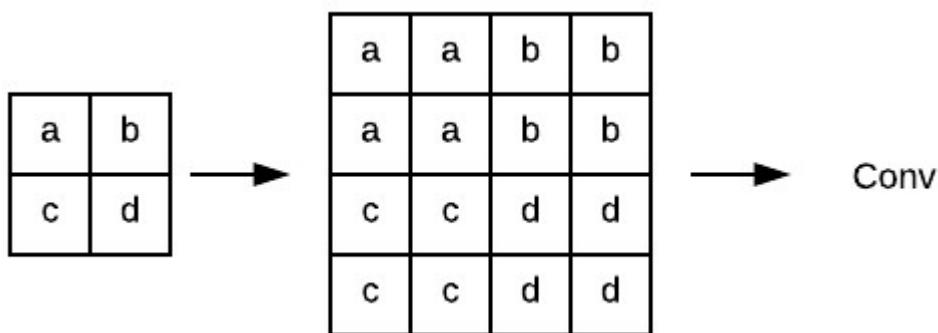
There is a fantastic paper called [A guide to convolution arithmetic for deep learning](#) that shows a great picture of exactly what does a 3x3 kernel stride half conv look like. And it's literally this. If you have a 2x2 input, so the blue squares are the 2x2 input, you add not only 2 pixels of padding 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 3×3 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, 2×2 input with these are your pixel values (a, b, c, d) and you want to create a 4×4 , why not just do this?



So I've now up scaled from 2×2 to 4×4 . 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 4×4 , 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.

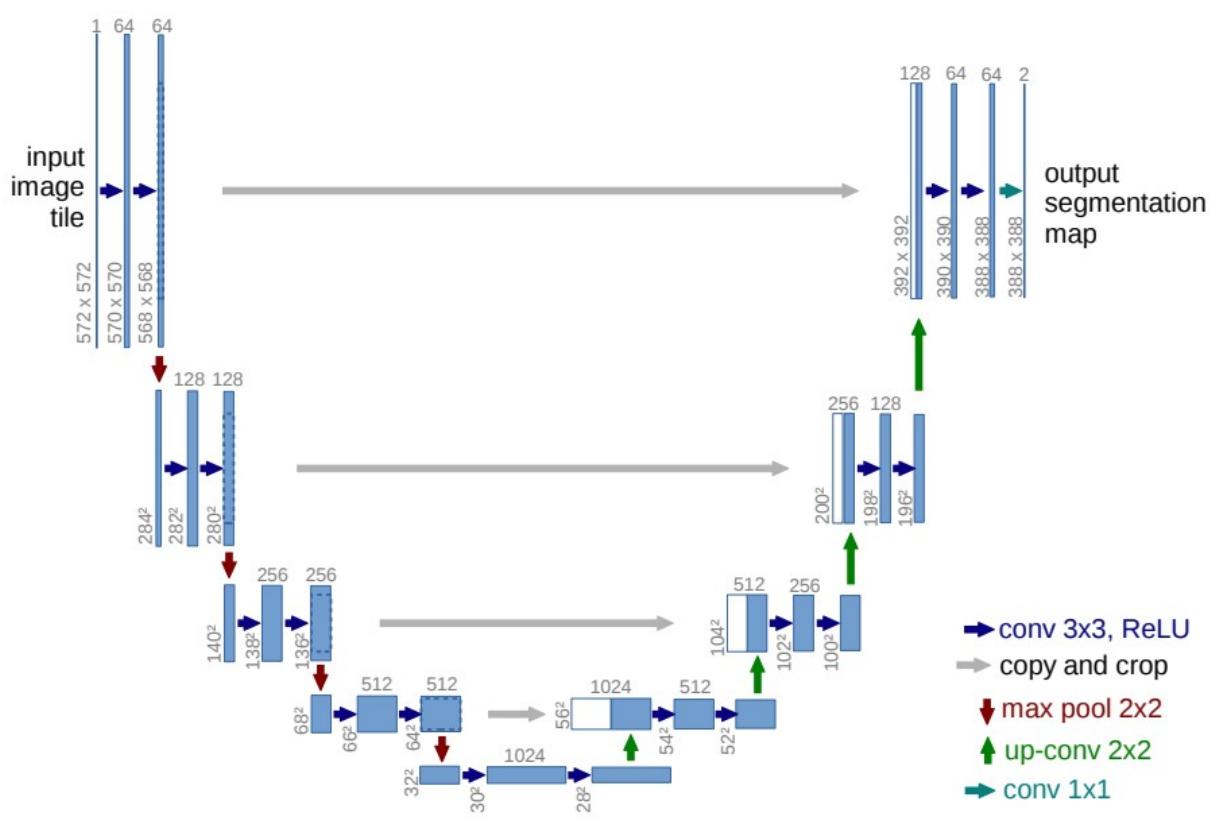
Another approach is instead of using nearest neighbor interpolation, you can use bilinear interpolation which basically means instead of copying A to all those different cells you take a weighted average of the cells around it.

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

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]

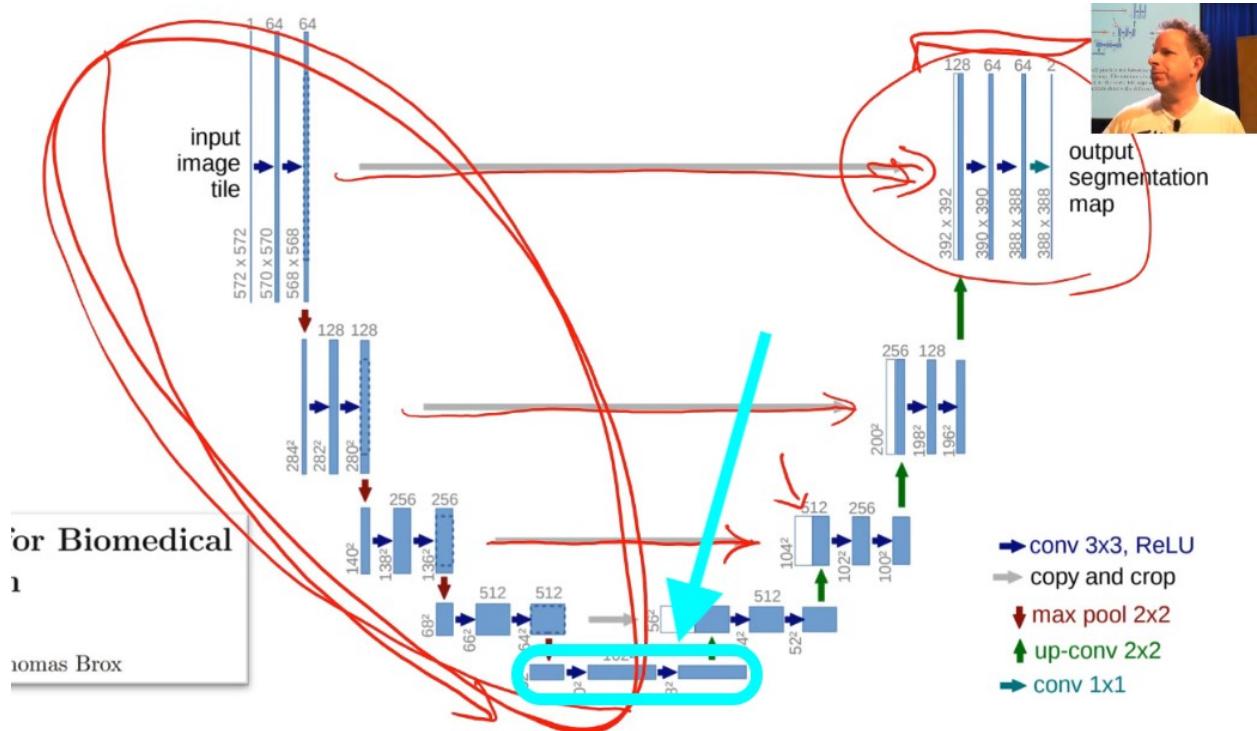
for i, idx in enumerate(sfs_ids):
    not_final = i!=len(sfs_ids)-1
    up_in_c, x_in_c = int(x.shape[1]), int(sfs_szs[idx][1])
    do.blur = blur and (not_final or blur_final)
    sa = self_attention and (i==len(sfs_ids)-3)
    unet_block = UnetBlock(up_in_c, x_in_c, self.sfs[i], final_div=not_final,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

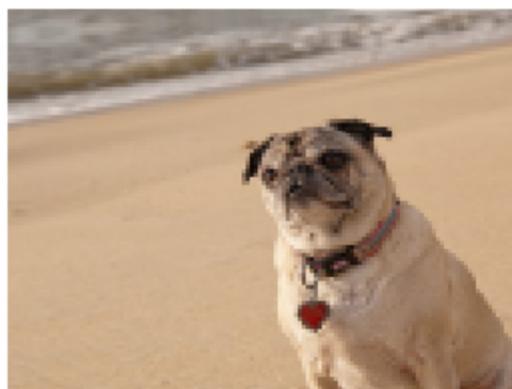
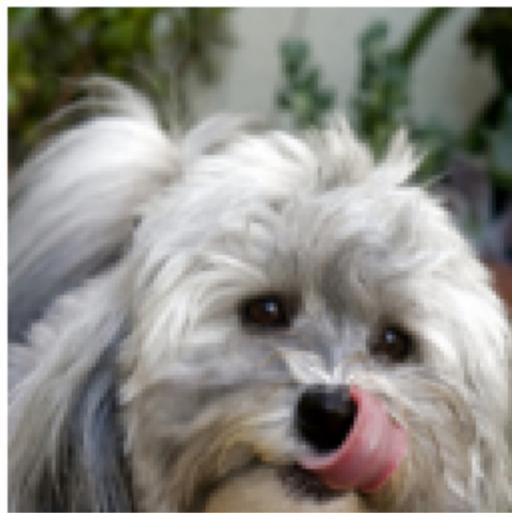
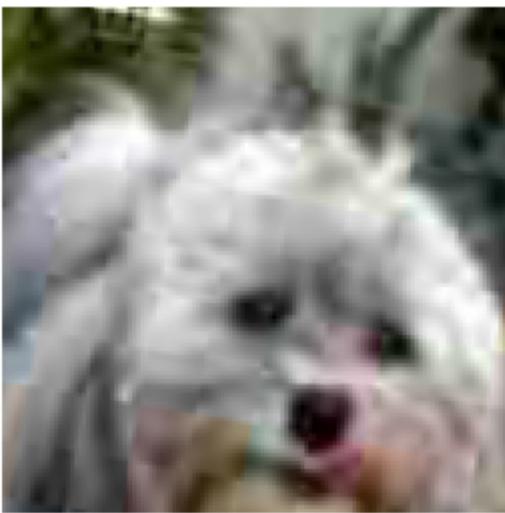
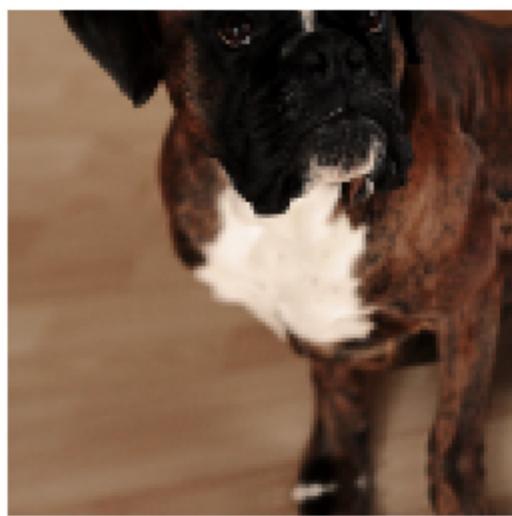
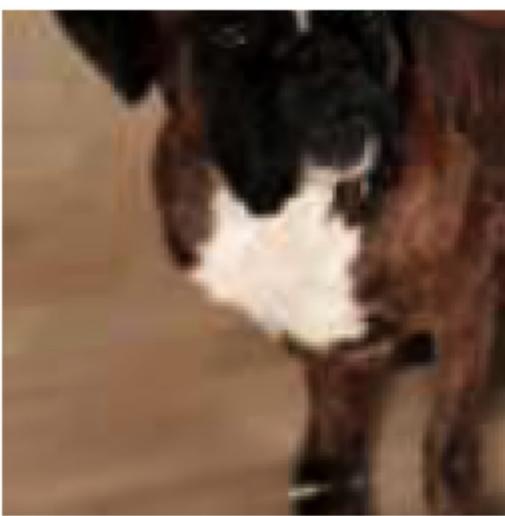
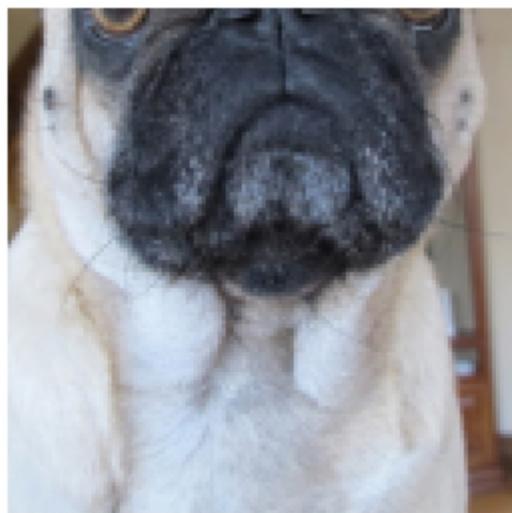
```
from PIL import Image, ImageDraw, ImageFont

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.Weight,
                        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()
```

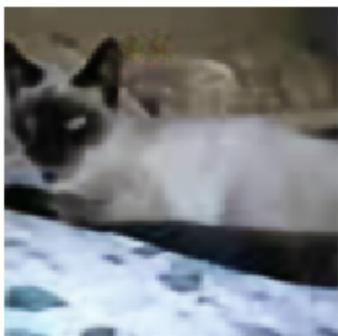
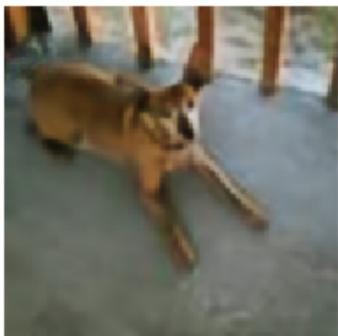
```
learn_gen.fit_one_cycle(3, slice(1e-6,1e-3))
```

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

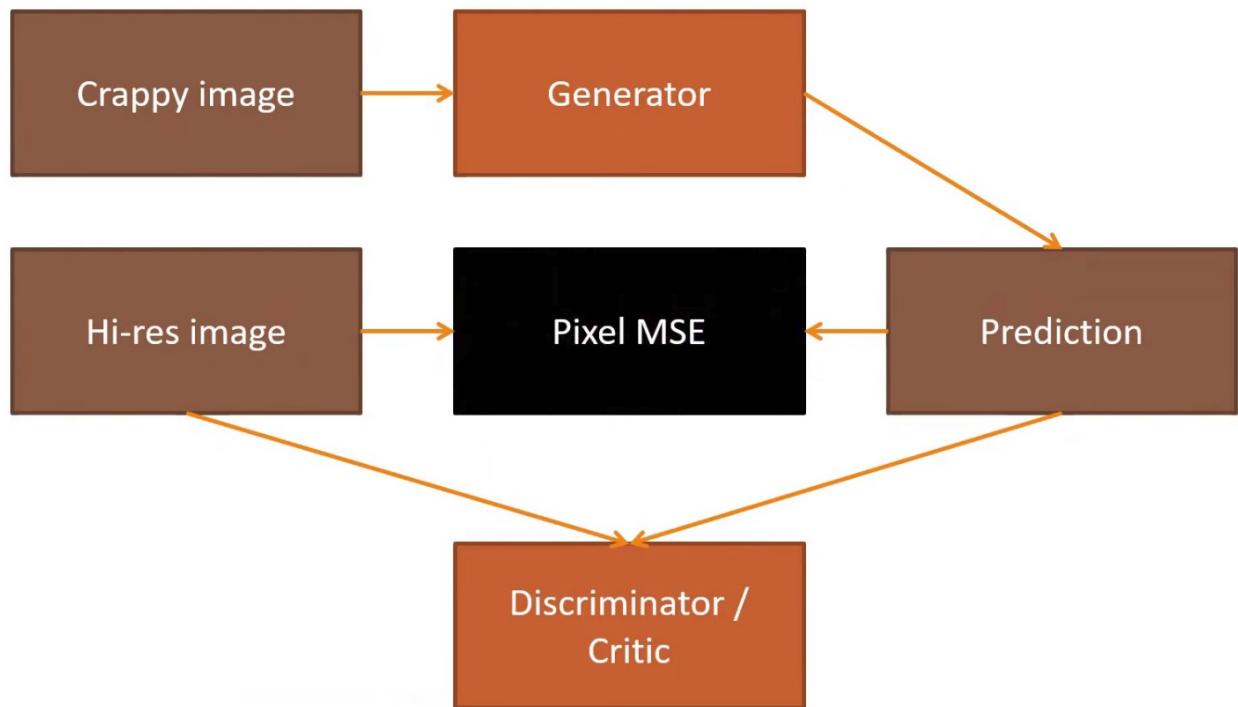


Let's unfreeze that and train a little more and look at that! With four minutes of training, we've got something which is basically doing a perfect job of removing numbers. It's certainly not doing a good job of upsampling, but it's definitely doing a nice job. Sometimes when it removes a number, it 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 loss 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 as 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 fooling 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.

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.

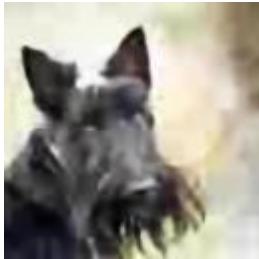
```
name_gen = 'image_gen'  
path_gen = path/name_gen  
  
# shutil.rmtree(path_gen)  
  
path_gen.mkdir(exist_ok=True)  
  
  
def save_preds(dl):  
    i=0  
    names = dl.dataset.items  
    for b in dl:  
        preds = learn_gen.pred_batch(batch=b, reconstruct=True)  
        for o in preds:  
            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()
```

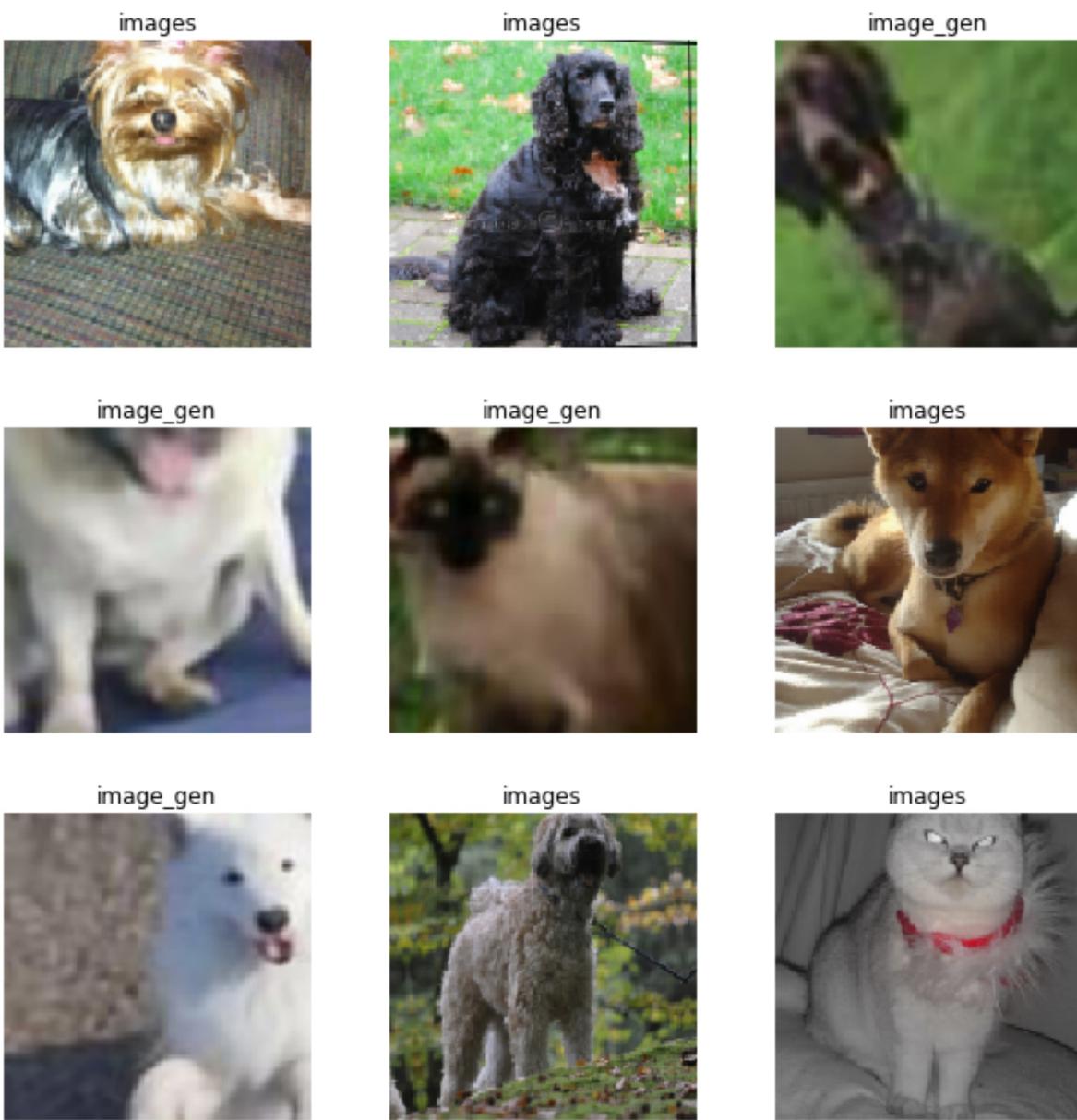
If you're using `nvidia-smi` to actually look at your GPU memory, you won't see it clear because PyTorch still has a kind of allocated cache, but it makes it available. So you should find this is how you can avoid restarting your notebook.

```
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.), show_img=False,
                                  opt_func=partial(optim.Adam, betas=(0., 0.99)), wd=wd)
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=False,
                        switcher=switcher, opt_func=partial(optim.Adam, betas=(0., 0.99)),
                        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
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

epoch	train_loss	gen_loss	disc_loss
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

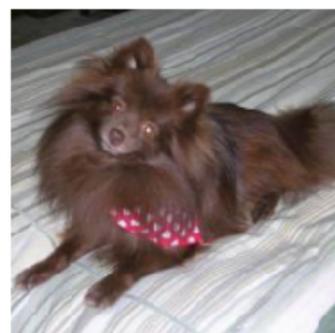
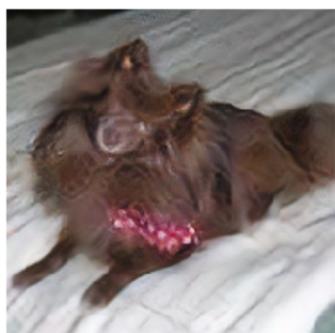
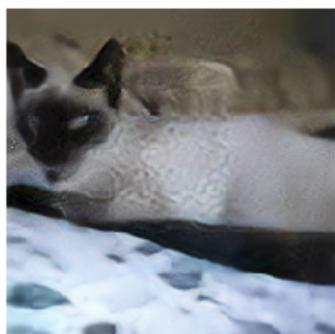
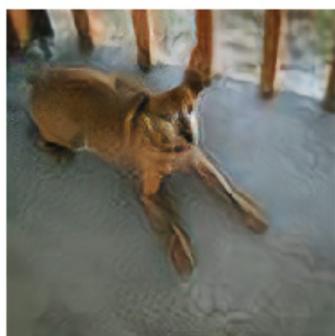
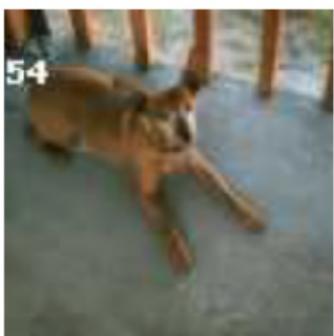
One of the tough things about GANs is that these loss numbers, they're meaningless. You can't expect them to go down because as the generator gets better, it gets harder for the discriminator (i.e. the critic) and then as the critic gets better, it's harder for the generator. So the numbers should stay 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=False
                           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 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

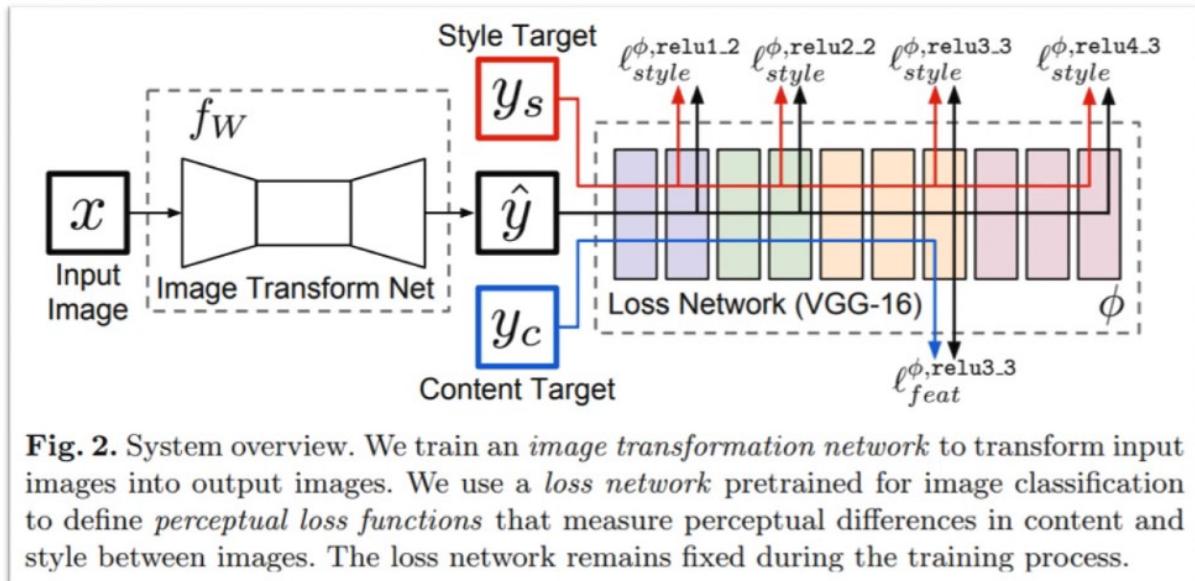


Fig. 2. System overview. We train an *image transformation network* to transform input images into output images. We use a *loss network* pretrained for image classification to define *perceptual loss functions* that measure perceptual differences in content and style between images. The loss network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

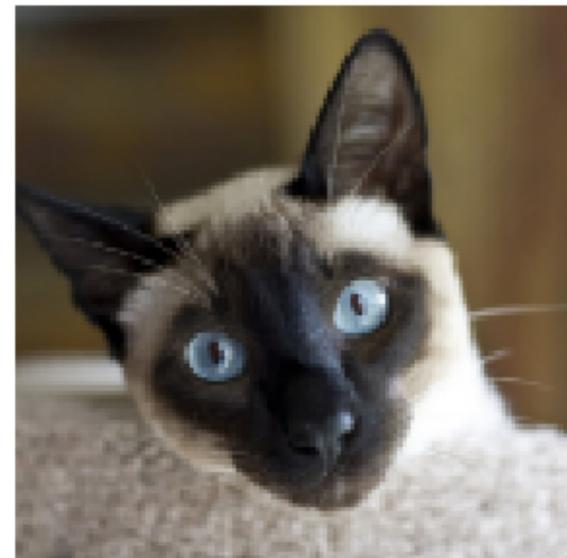
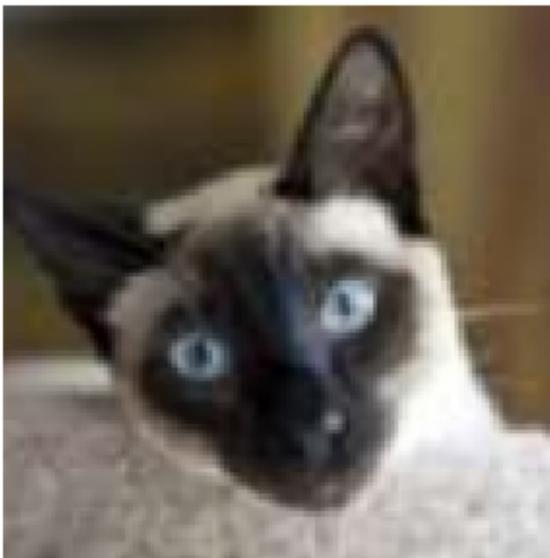

def get_data(bs,size):
    data = (src.label_from_func(lambda x: path_hr/x.name)
            .transform(get_transforms(max_zoom=2.), size=size, tfm_y=True)
            .databunch(bs=bs).normalize(imagenet_stats, do_y=True))

    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).

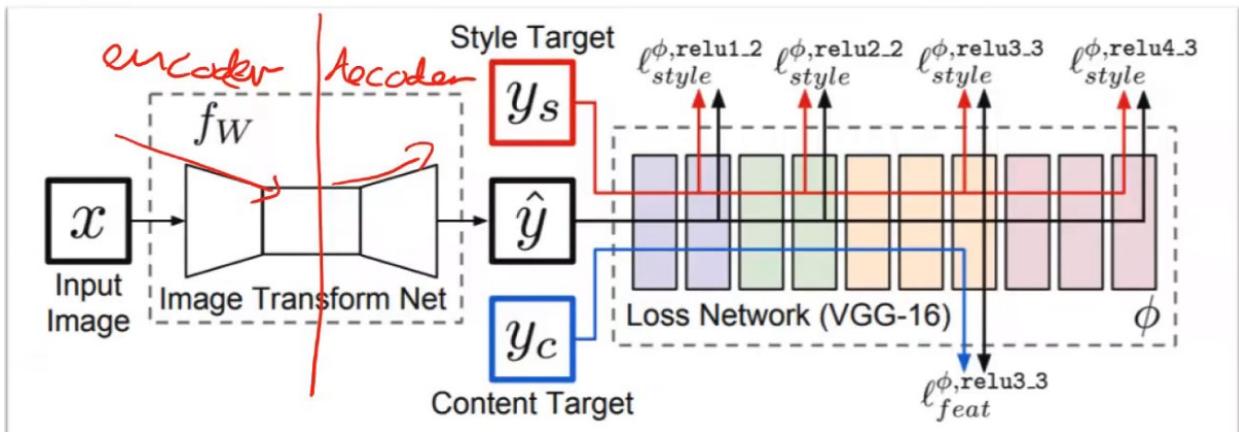
```

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.wgts)]
        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.wgts)]
        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 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.

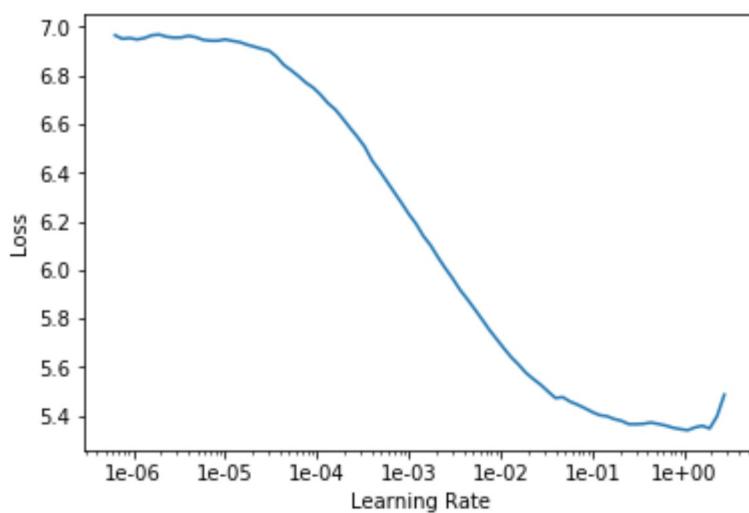
That's going to end up in this list called `feat_losses`, then sum them all up. By the way, the reason I do it as a list is because we've got this nice little callback that if you put them into thing called `.metrics` in your loss function, it'll print out all of the separate layer loss amounts for you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

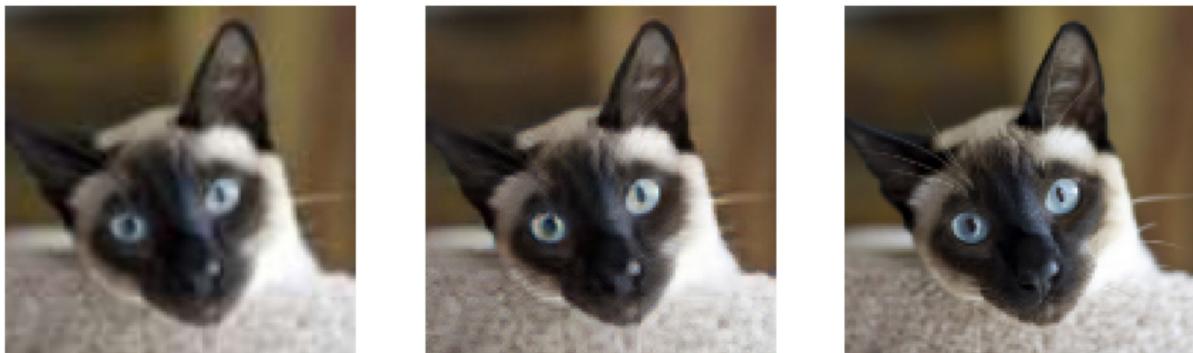
```
do_fit('1a', slice(lr*10))
```

Total time: 11:16

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2
1	3.873667	3.759143	0.144560	0.229806	0.314573	0.226204
2	3.756051	3.650393	0.145068	0.228509	0.308807	0.218000
3	3.688726	3.628370	0.157359	0.226753	0.304955	0.215417
4	3.628276	3.524132	0.145285	0.225455	0.300169	0.211110
5	3.586930	3.422895	0.145161	0.224946	0.294471	0.205117
6	3.528042	3.394804	0.142262	0.220709	0.289961	0.201980
7	3.522416	3.361185	0.139654	0.220379	0.288046	0.200114
8	3.469142	3.338554	0.142112	0.219271	0.287442	0.199255
9	3.418641	3.318710	0.146493	0.219915	0.284979	0.197340

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2
10	3.356641	3.187186	0.135588	0.215685	0.277398	0.189562

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()
```

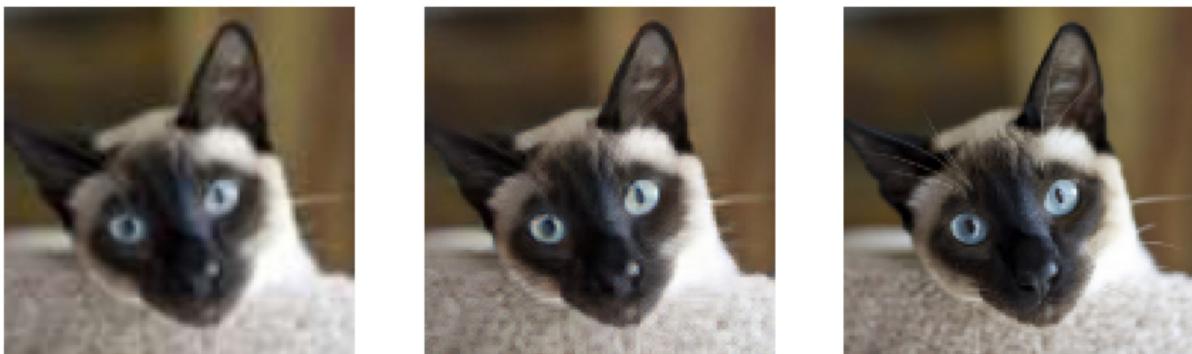
```
do_fit('1b', slice(1e-5,lr))
```

Total time: 11:39

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2
1	3.303951	3.179916	0.135630	0.216009	0.277359	0.189097
2	3.308164	3.174482	0.135740	0.215970	0.277178	0.188737
3	3.294504	3.169184	0.135216	0.215401	0.276744	0.188395
4	3.282376	3.160698	0.134830	0.215049	0.275767	0.187716
5	3.301212	3.168623	0.135134	0.215388	0.276196	0.188382
6	3.299340	3.159537	0.135039	0.214692	0.275285	0.187554
7	3.291041	3.159207	0.134602	0.214618	0.275053	0.187660
8	3.285271	3.147745	0.134923	0.214514	0.274702	0.187147

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2
9	3.279353	3.138624	0.136035	0.213191	0.273899	0.186854
10	3.261495	3.124737	0.135016	0.213681	0.273402	0.185922

Input / Prediction / Target



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');
```

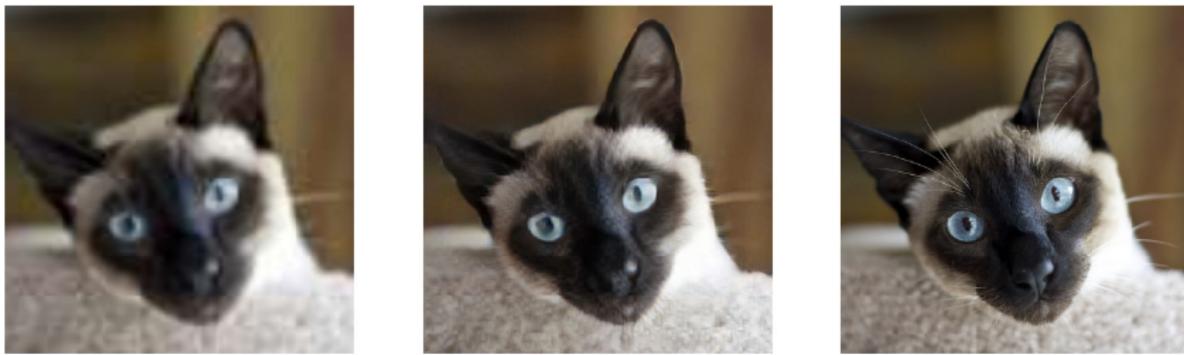
```
do_fit('2a')
```

Total time: 43:44

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2
1	2.249253	2.214517	0.164514	0.260366	0.294164	0.155227
2	2.205854	2.194439	0.165290	0.260485	0.293195	0.154746
3	2.184805	2.165699	0.165945	0.260999	0.291515	0.153438
4	2.145655	2.159977	0.167295	0.260605	0.290226	0.152415
5	2.141847	2.134954	0.168590	0.260219	0.288206	0.151237

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2
6	2.145108	2.128984	0.164906	0.259023	0.286386	0.150245
7	2.115003	2.125632	0.169696	0.259949	0.286435	0.150898
8	2.109859	2.111335	0.166503	0.258512	0.283750	0.148191
9	2.092685	2.097898	0.169842	0.259169	0.284757	0.148156
10	2.061421	2.080940	0.167636	0.257998	0.282682	0.147471

Input / Prediction / Target



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

```
learn.unfreeze()
```

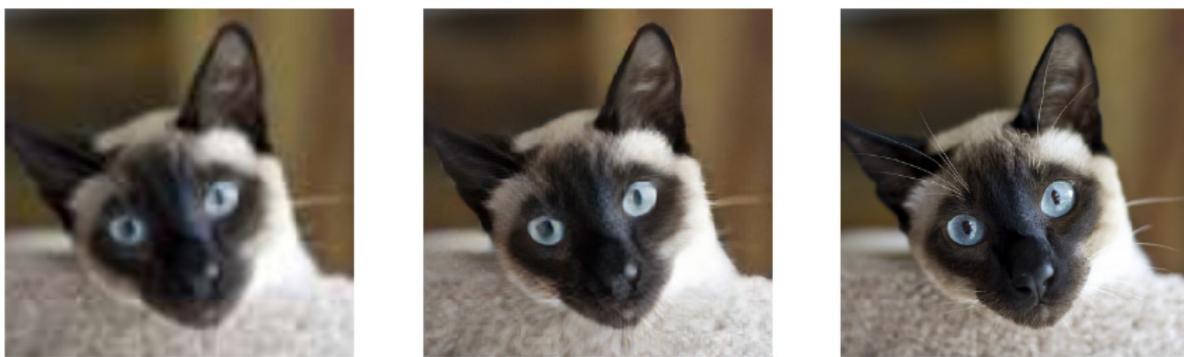
```
do_fit('2b', slice(1e-6,1e-4), pct_start=0.3)
```

Total time: 45:19

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2
1	2.061799	2.078714	0.167578	0.257674	0.282523	0.147208
2	2.063589	2.077507	0.167022	0.257501	0.282275	0.146879
3	2.057191	2.074605	0.167656	0.257041	0.282204	0.146925
4	2.050781	2.073395	0.166610	0.256625	0.281680	0.146585
5	2.054705	2.068747	0.167527	0.257295	0.281612	0.146392
6	2.052745	2.067573	0.167166	0.256741	0.281354	0.146101
7	2.051863	2.067076	0.167222	0.257276	0.281607	0.146188

epoch	train_loss	valid_loss	pixel	feat_0	feat_1	feat_2
8	2.046788	2.064326	0.167110	0.257002	0.281313	0.146055
9	2.054460	2.065581	0.167222	0.257077	0.281246	0.146016
10	2.052605	2.064459	0.166879	0.256835	0.281252	0.146135

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/Siamese_178.jpg')

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.



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

Dropout

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

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

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

5:30 PM - 20 Jun 2018

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.

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



- Srivastava's Master's(!) thesis.

- Training

- Usually gives a small performance boost.

- Mysterious.

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

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

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

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

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.

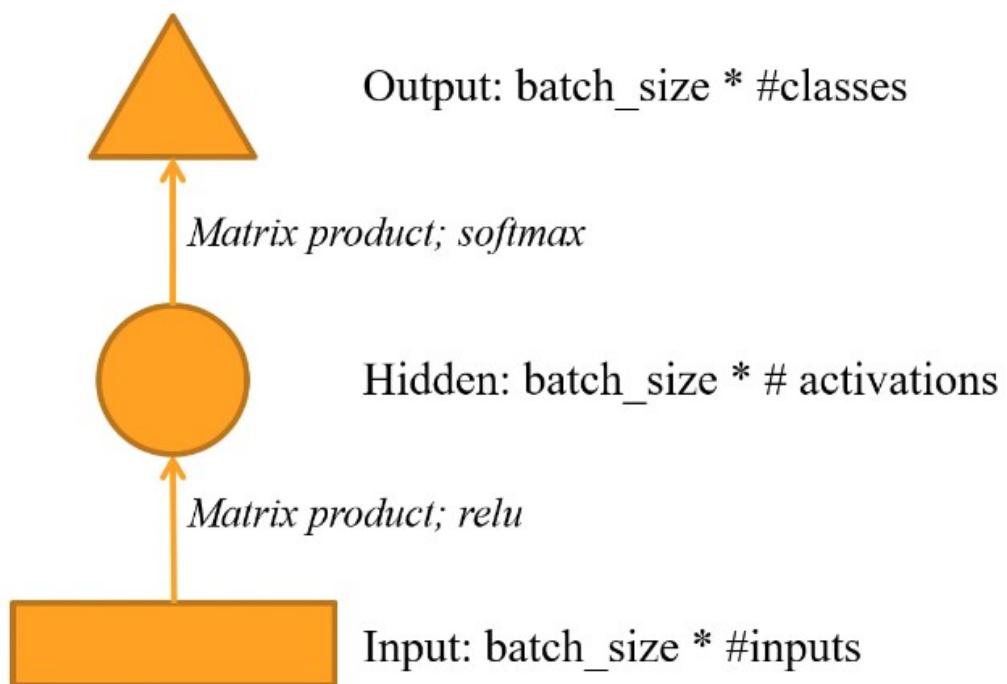
Then today, we've learned about Res/Dense blocks. We obviously learnt a lot about image classification and regression, embeddings, categorical and continuous variables, collaborative filtering, language models, and NLP classification. Then segmentation, U-Net and GANs.

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.

Basic NN with single hidden layer



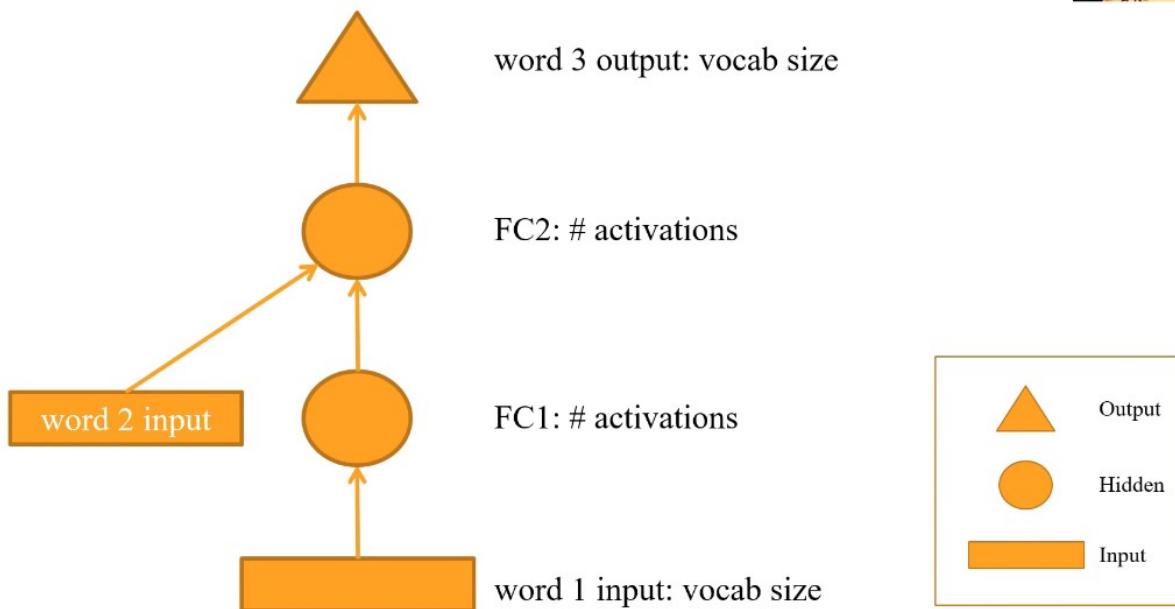
Rectangle means an input. That'll be batch size by number of inputs. An **arrow** means a layer (broadly defined) such as matrix product followed by ReLU. A **circle** is activations. So this case, we have one set of hidden activations and so given that the input was number of inputs, this here (the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

Predicting word 3 using words 1 & 2

*NB: layer operations
remember that arrows represent layers*



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 add. And that's going to

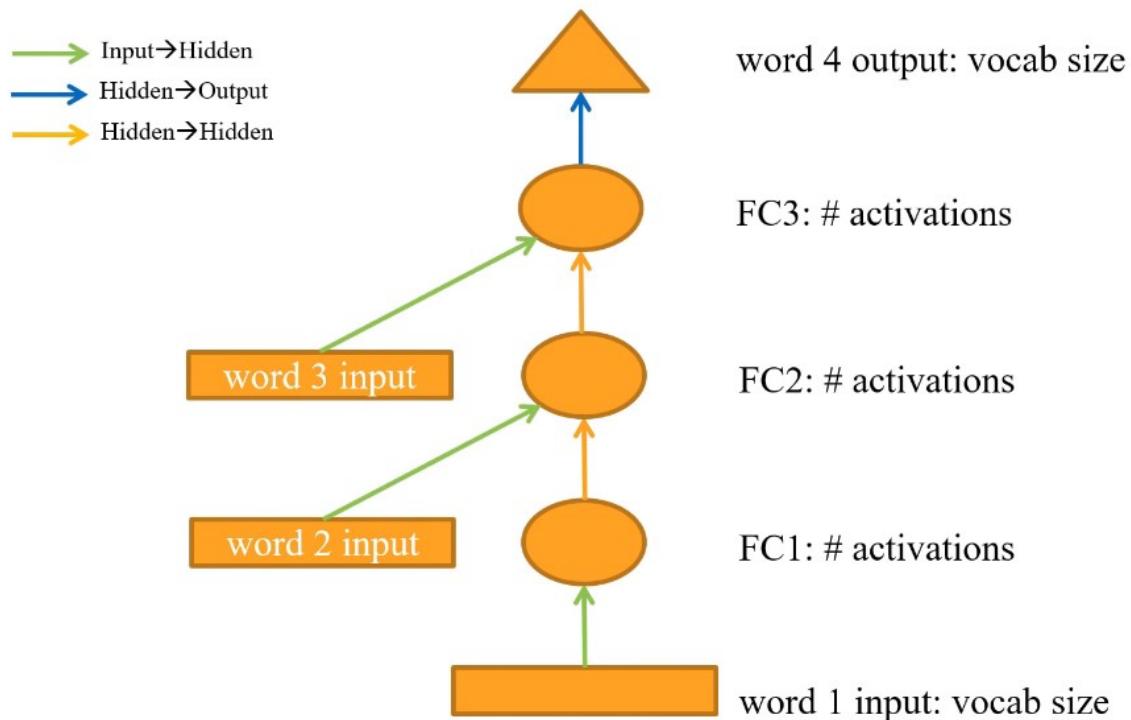
create the second bunch of activations.

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.

Predicting word 4 using words 1, 2 & 3



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.

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 or 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/batch_size
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

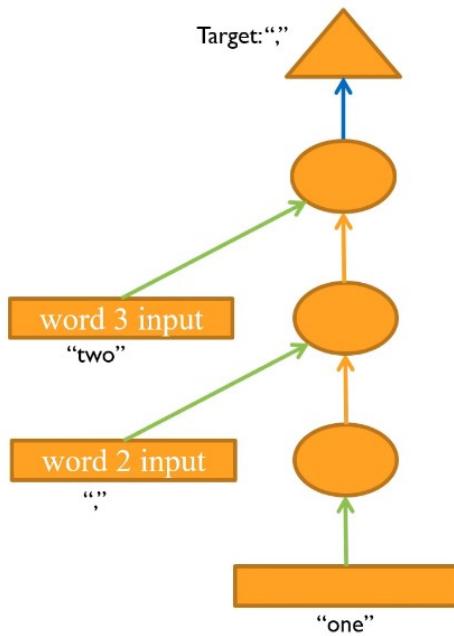
nh=64

```
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:



```

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)

```

It 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

epoch	train_loss	valid_loss	acc4
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.

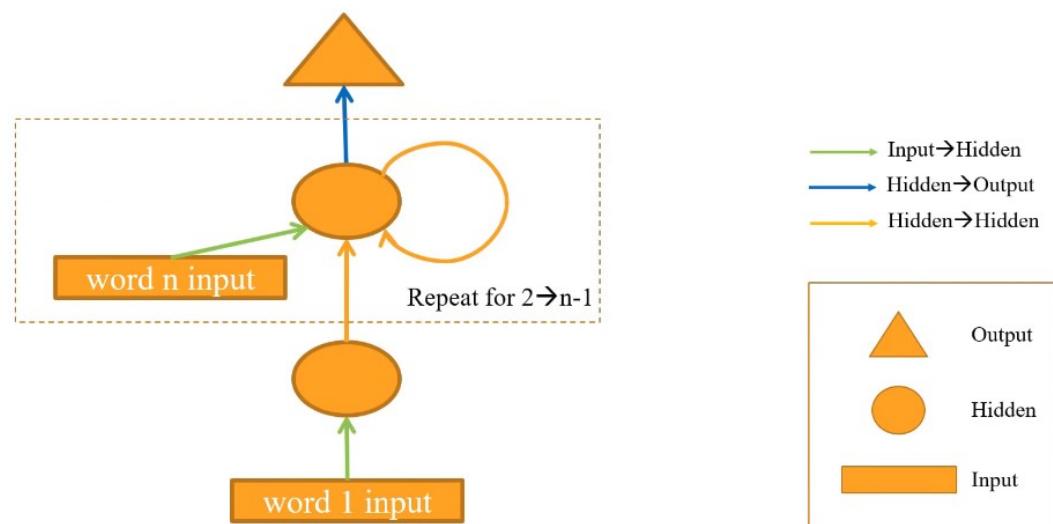
```
class Model1(nn.Module):
    def __init__(self):
        super().__init__()
        self.i_h = nn.Embedding(nv,nh) # green arrow
        self.h_h = nn.Linear(nh,nh) # brown arrow
        self.h_o = nn.Linear(nh,nv) # blue arrow
        self.bn = nn.BatchNorm1d(nh)

    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:

Predicting word n using words 1 to n-1

NB: no hidden/output layer



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

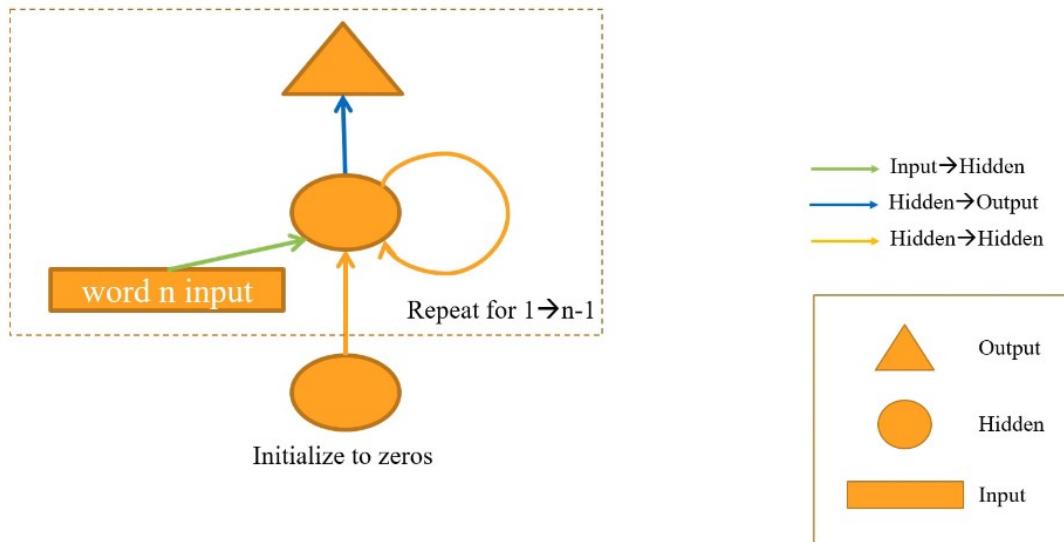
epoch	train_loss	valid_loss	acc4
1	3.463261	3.436951	0.172881
2	2.937433	2.903948	0.385984
3	2.405134	2.457942	0.454827
4	2.100047	2.231621	0.459468
5	1.981868	2.155234	0.460860
6	1.957631	2.144365	0.461324

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 `n`th 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 in `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:



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.

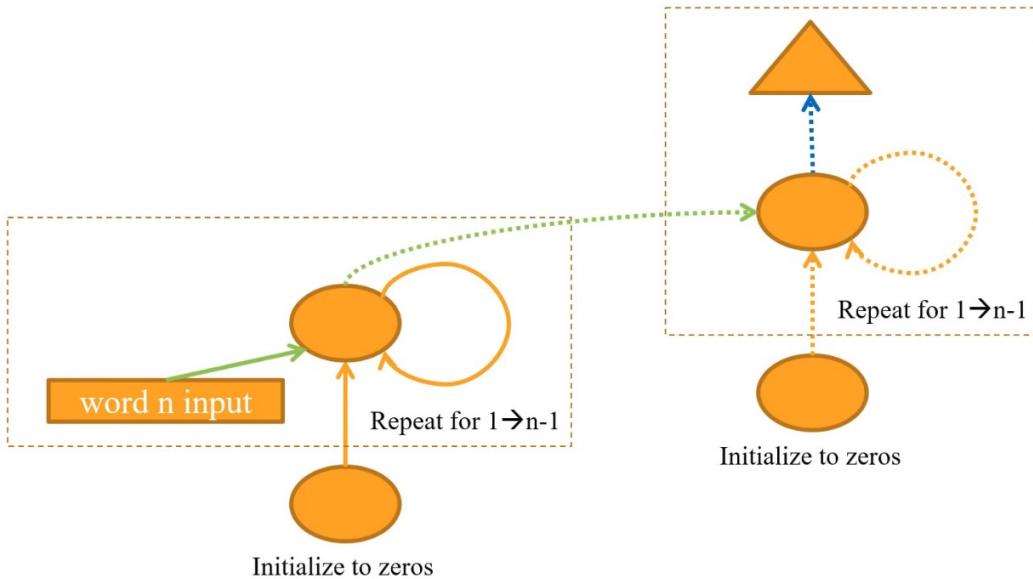
```
learn = Learner(data, Model3(), metrics=accuracy)
```

```
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.



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)
```

```

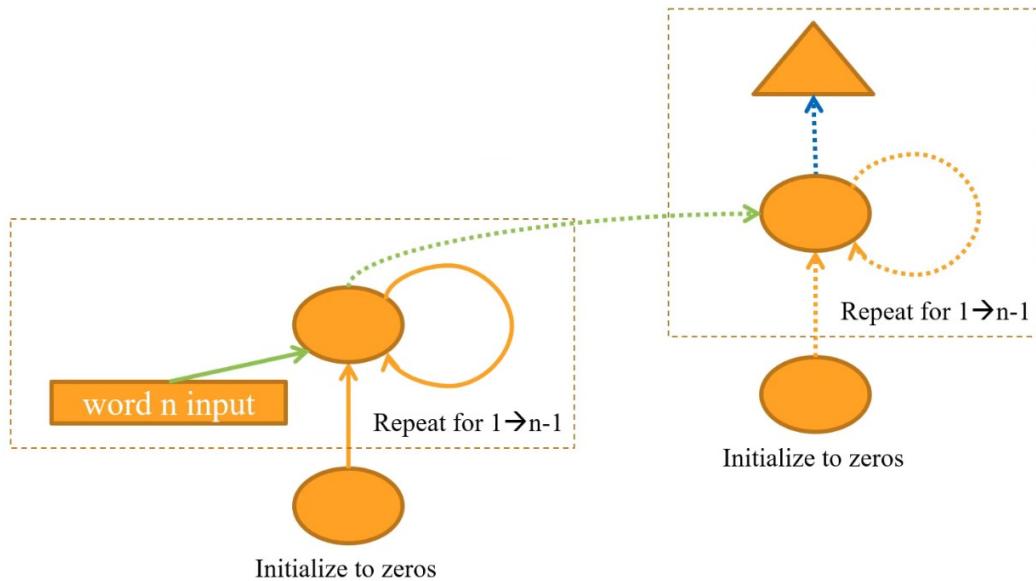
self.rnn = nn.GRU(nh,nh,2)
self.h_o = nn.Linear(nh,nv)
self.bn = nn.BatchNorm1d(nh)
self.h = torch.zeros(2, bs, nh).cuda()

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))

```

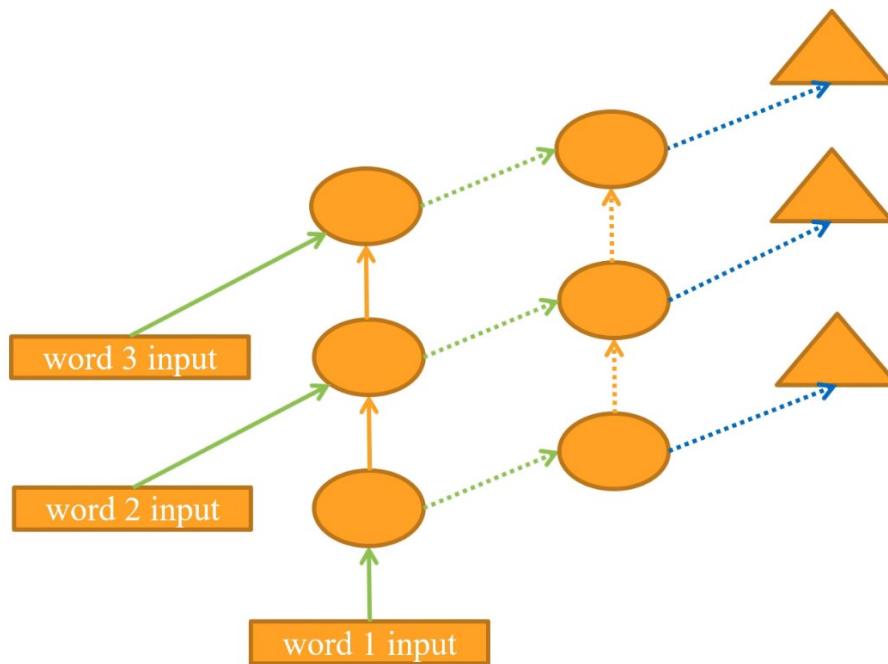
But here's the thing. When you think about this:

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

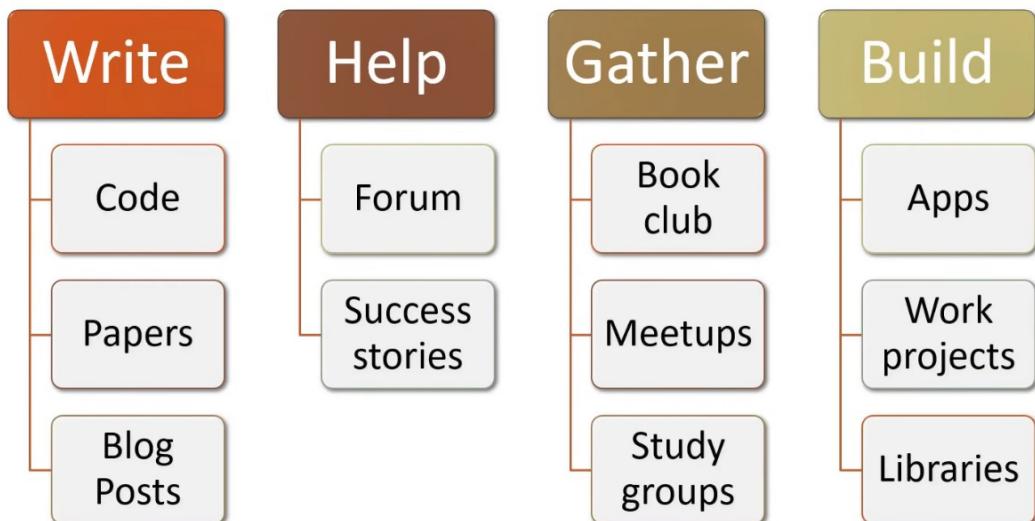
epoch	train_loss	valid_loss	accuracy
1	3.010502	2.602906	0.428063
2	2.087371	1.765773	0.532410
3	1.311803	1.571677	0.643796
4	0.675637	1.594766	0.730275
5	0.363373	1.432574	0.760873
6	0.188198	1.704319	0.762454
7	0.108004	1.716183	0.755837
8	0.064206	1.510942	0.763404
9	0.040955	1.633394	0.754179
10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

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



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.

**Elon has landed on Mars!!
And his tweets are
totally bonkers!**

FastAI makes neural

Generate Tweet

FastAI makes neural net open to any number of possible civilizations. in the past most population regions are still within range of the world's largest

#Elon_1_SEC_0 #IronMan #FastAI



Tweet me your favorite tweet!



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."



OCData_nerd Dave Smith



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

<https://deepelon.com/> 2

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

Head [here](#) 2 to generate your own tweet or review the code (my first commits ever were today!). Thanks to [@rachel](#) and [@jeremy](#) for teaching a finance guy how to build an app in 8 weeks 😊

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.

Coming up: part 2! Cutting Edge Deep Learning



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.