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 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.

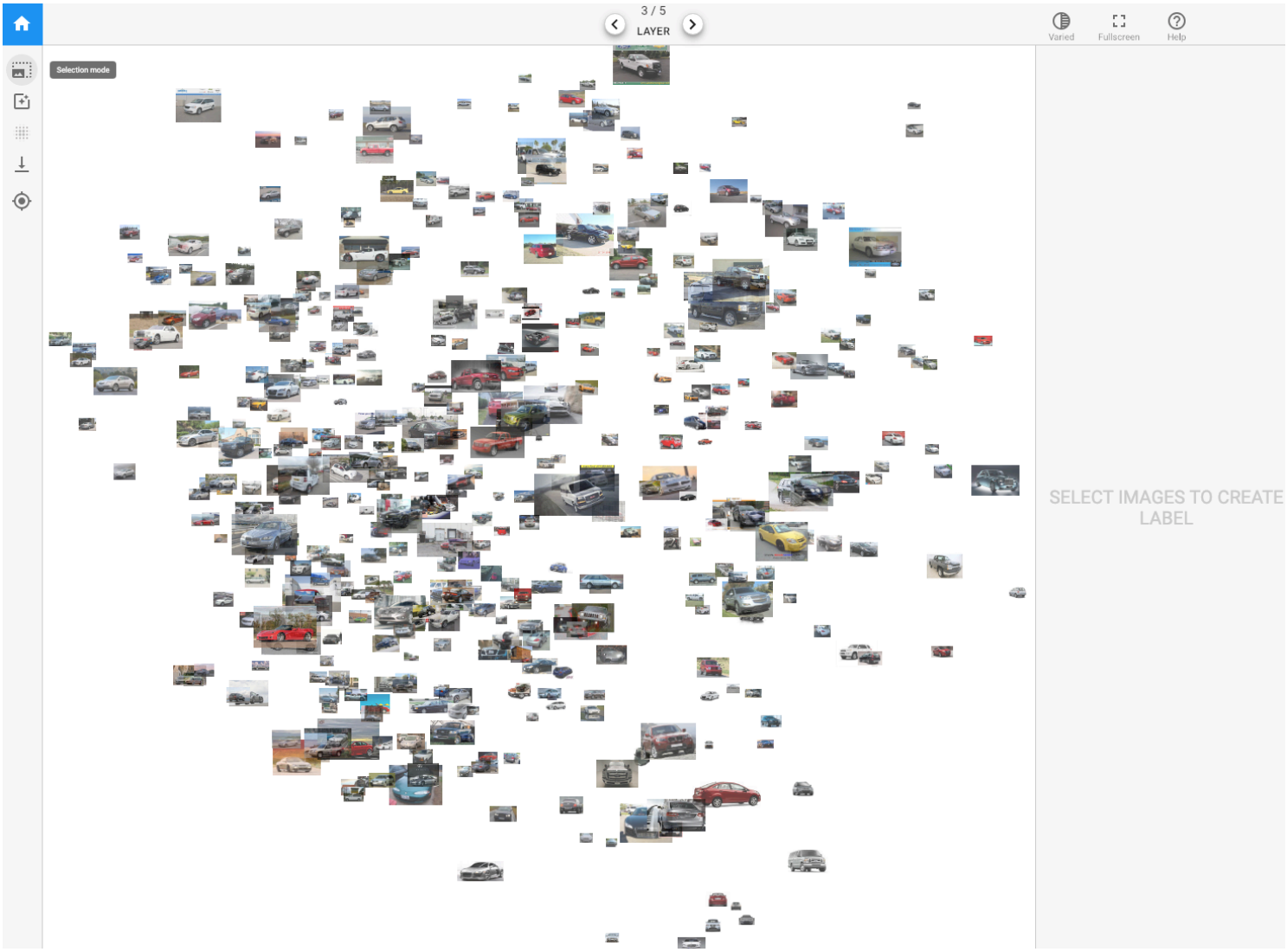
So the company is called platform.ai and the reason I'm mentioning it here is that it's going to let you create models on different types of datasets to what you can do now, that is to say datasets that you don't have labels for yet. We're actually going to help you label them. So this is the first time 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 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 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 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	3	4
index	0	1	2	3	4
Store	1	2	3	4	5
DayOfWeek	5	5	5	5	5
Date	2015-07-31 00:00:00	2015-07-31 00:00:00	2015-07-31 00:00:00	2015-07-31 00:00:00	2015-07-31 00:00:00
Sales	5263	6064	8314	13995	4822
Customers	555	625	821	1498	559
Open	1	1	1	1	1
Promo	1	1	1	1	1
StateHoliday	False	False	False	False	False
SchoolHoliday	1	1	1	1	1
Year	2015	2015	2015	2015	2015
Month	7	7	7	7	7
Week	31	31	31	31	31
Day	31	31	31	31	31
Dayofweek	4	4	4	4	4
Dayofyear	212	212	212	212	212

	0	1	2	3	4
Is_month_end	True	True	True	True	True
Is_month_start	False	False	False	False	False
Is_quarter_end	False	False	False	False	False
Is_quarter_start	False	False	False	False	False
Is_year_end	False	False	False	False	False
Is_year_start	False	False	False	False	False
Elapsed	1438300800	1438300800	1438300800	1438300800	1438300800
StoreType	c	a	a	c	a
Assortment	a	a	a	c	a
CompetitionDistance	1270	570	14130	620	29910
CompetitionOpenSinceMonth	9	11	12	9	4
CompetitionOpenSinceYear	2008	2007	2006	2009	2015
Promo2	0	1	1	0	0
Promo2SinceWeek	1	13	14	1	1
…	…	…	…	…	…
Min_Sea_Level_PressurehPa	1015	1017	1017	1014	1016
Max_VisibilityKm	31	10	31	10	10
Mean_VisibilityKm	15	10	14	10	10
Min_VisibilitykM	10	10	10	10	10
Max_Wind_SpeedKm_h	24	14	14	23	14
Mean_Wind_SpeedKm_h	11	11	5	16	11
Max_Gust_SpeedKm_h	NaN	NaN	NaN	NaN	NaN
Precipitationmm	0	0	0	0	0
CloudCover	1	4	2	6	4
Events	Fog	Fog	Fog	NaN	NaN
WindDirDegrees	13	309	354	282	290
StateName	Hessen	Thueringen	NordrheinWestfalen	Berlin	Sachsen
CompetitionOpenSince	2008-09-15 00:00:00	2007-11-15 00:00:00	2006-12-15 00:00:00	2009-09-15 00:00:00	2015-04-15 00:00:00
CompetitionDaysOpen	2510	2815	3150	2145	107
CompetitionMonthsOpen	24	24	24	24	3
Promo2Since	1900-01-01 00:00:00	2010-03-29 00:00:00	2011-04-04 00:00:00	1900-01-01 00:00:00	1900-01-01 00:00:00
Promo2Days	0	1950	1579	0	0
Promo2Weeks	0	25	25	0	0
AfterSchoolHoliday	0	0	0	0	0
BeforeSchoolHoliday	0	0	0	0	0
AfterStateHoliday	57	67	57	67	57
BeforeStateHoliday	0	0	0	0	0
AfterPromo	0	0	0	0	0
BeforePromo	0	0	0	0	0
SchoolHoliday_bw	5	5	5	5	5
StateHoliday_bw	0	0	0	0	0
Promo_bw	5	5	5	5	5
SchoolHoliday_fw	7	1	5	1	1

	0	1	2	3	4
StateHoliday_fw	0	0	0	0	0
Promo_fw	5	1	5	1	1

93 rows × 5 columns

The key thing here is that we're trying to on 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_Humidity	Sales
280	281	5	NaN	6970.0	61	8053
584	586	5	NaN	250.0	61	17879
588	590	5	Jan,Apr,Jul,Oct	4520.0	51	7250
847	849	5	NaN	5000.0	67	10829
896	899	5	Jan,Apr,Jul,Oct	2590.0	55	5952

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	Mean_Humidity	Sales
428412	NaN	2	NaN	840.0	89	8343
428541	1050.0	2	Mar,Jun,Sept,Dec	13170.0	78	4945
428813	NaN	1	Jan,Apr,Jul,Oct	11680.0	85	4946
430157	414.0	6	Jan,Apr,Jul,Oct	6210.0	88	6952
431137	285.0	5	NaN	2410.0	57	5377

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	Mean_Humidity	Sales	CompetitionDistance_na
78375	622	5	NaN	2380.0	71	5390	True
161185	622	6	NaN	2380.0	91	2659	True
363369	879	4	Feb,May,Aug,Nov	2380.0	73	4788	True

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)])].inde
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_va
                    .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:

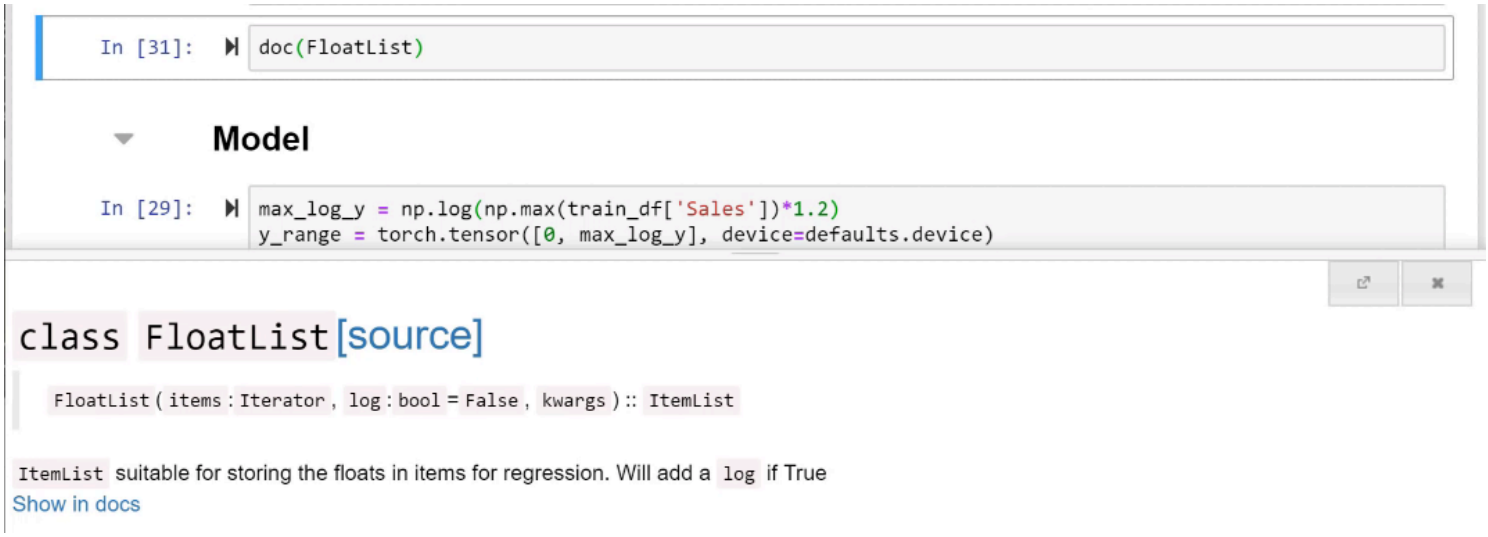
- From a data frame, passing all of that information.
- Split it into valid vs. train.
- Label it with a dependent variable.

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:



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())
```

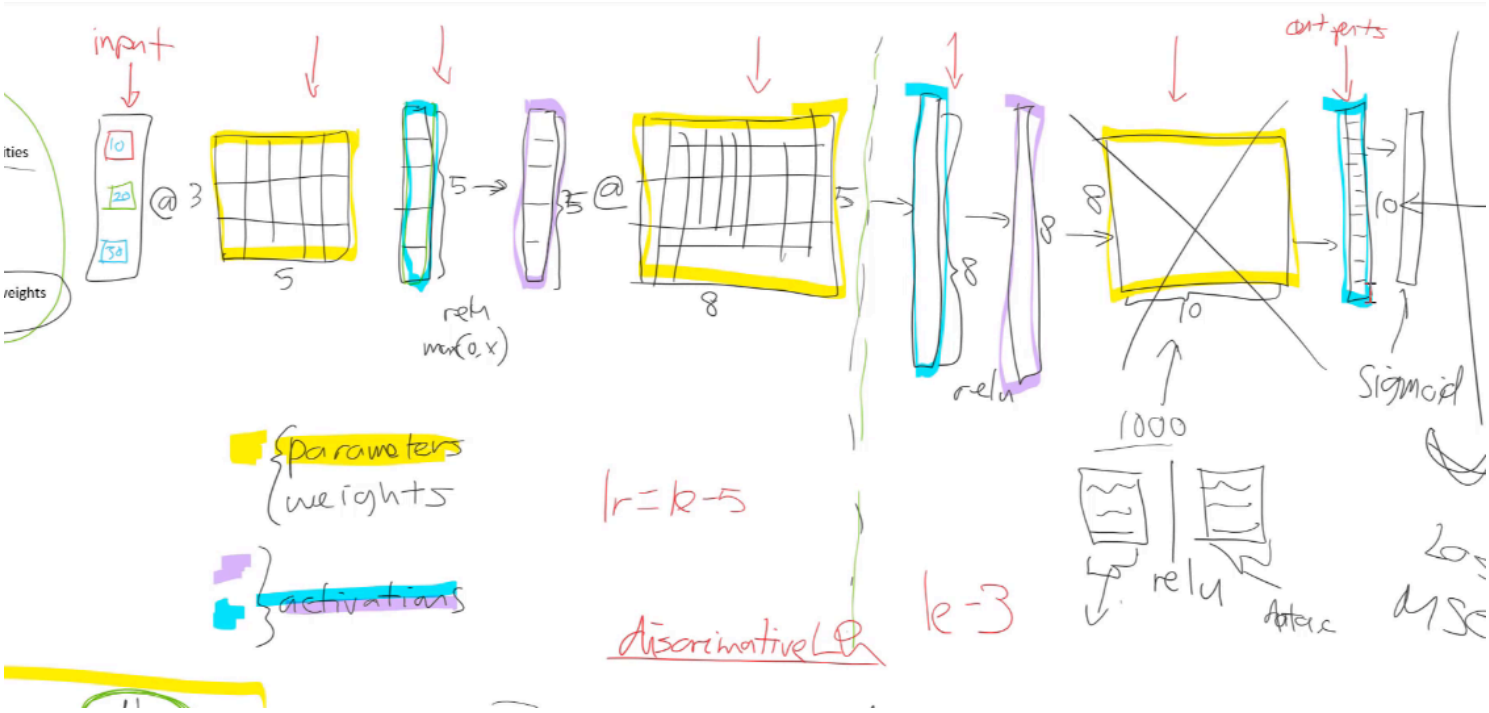
The reason that we have this (`log=True`) here is because this is so common. Basically anytime you're trying to predict something like a population or a dollar amount of sales, these kind of things tend to have long tail distributions where you care more about percentage differences and exact/absolute 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)
```

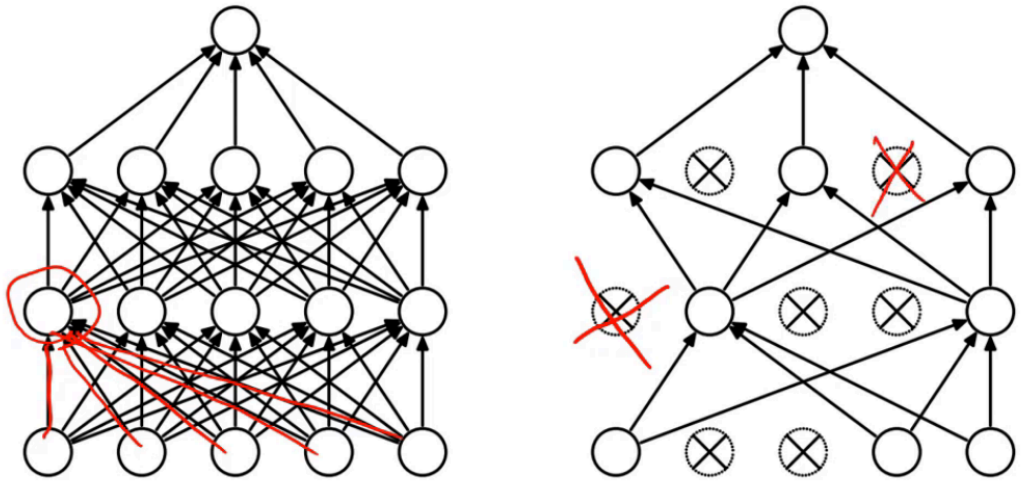
Now the intermediate weight matrix is going to have to go from a 1000 activation input to a 500 activation output, which means it's going to have to be 500,000 elements in that weight matrix. That's an awful lot for a data set with only a few hundred thousand rows. So this is going to overfit, and we 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 `embb_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 NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU



(a) Standard Neural Net (b) After applying dropout.

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

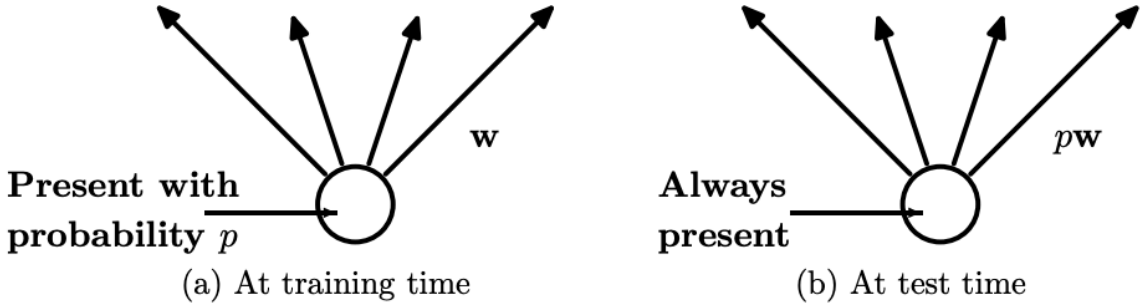
There's a view that spiking neurons might help regularization, and dropout is a way of matching this idea of spiking neurons. It's interesting. When you actually ask people where did your idea for some algorithm come from, it basically never comes from math; it always comes from intuition and thinking 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]



(a) At training time (b) At test time

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,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
```

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

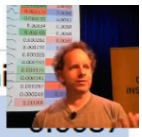
An embedding dropout is simply just a dropout. So it's just an instance of a dropout module. This kind of makes sense, right? For continuous variables, that continuous variable is just in one column. You wouldn't want to do dropout on that because you're literally deleting the existence of that whole 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 La						
	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**	**0.0089**	**0.0088**
Dense	0.0087	0.0088	0.0087	0.0090	0.0088	0.0089	0.0088
Split	0.0089	0.0086	0.0087	0.0088	0.0090	0.0089	0.0088
Init	**0.0086**	**0.0090**	**0.0088**	**0.0086**	**0.0087**	**0.0087**	**0.0087**
Dense	0.0085	0.0090	0.0087	0.0086	0.0087	0.0086	0.0087
Split	0.0088	0.0089	0.0089	0.0087	0.0087	0.0087	0.0088
Grand Total	**0.0087**	**0.0088**	**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:



									var	mi
0.0103	0.0104	0.0102	0.009	0.0091	0.0093	0.0088	0.0087		0.000262	
0.0108	0.0108	0.0095	0.0093	0.009	0.0091	0.0091	0.009		4.71E-05	0.009
0.0104	0.01	0.0093	0.0093	0.0096	0.009	0.009	0.0089		4.71E-05	0.0089
0.0107	0.0101	0.0093	0.0096	0.0089	0.0111	0.0086	0.0086		0.001179	0.0086
0.0103	0.0099	0.0104	0.009	0.0095	0.009	0.0087	0.0089		0.000125	0.0087
0.0111	0.0113	0.0095	0.0096	0.0088	0.0097	0.0091	0.0089		0.00034	0.0088
0.0101	0.0099	0.0142	0.0096	0.0099	0.009	0.0088	0.0088		9.43E-05	0.0088
0.01	0.0106	0.0099	0.0095	0.0093	0.0094	0.0095	0.0089		0.000262	0.0089
0.0105	0.0112	0.0093	0.0092	0.0107	0.0092	0.0087	0.0088		0.000216	0.0087
0.0098	0.0096	0.0092	0.009	0.0093	0.0093	0.0091	0.0088		0.000205	0.0088
0.0109	0.0099	0.0097	0.0095	0.0093	0.009	0.0094	0.0092		0.000163	0.009
0.0101	0.0107	0.0098	0.0099	0.0094	0.009	0.0091	0.0088		0.000125	0.0088
0.0104	0.0115	0.01	0.0095	0.0093	0.0089	0.0089	0.0086		0.000141	0.0086
0.0108	0.01	0.0099	0.0102	0.009	0.0093	0.009	0.0097		0.000287	0.009
0.0105	0.0108	0.0095	0.0098	0.0095	0.0089	0.0091	0.0085		0.000249	0.0085
0.01	0.0095	0.0094	0.0099	0.0091	0.0087	0.0087	0.0101		0.00066	0.0087

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_
  (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_st
```

```
    (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_sta
    (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
```

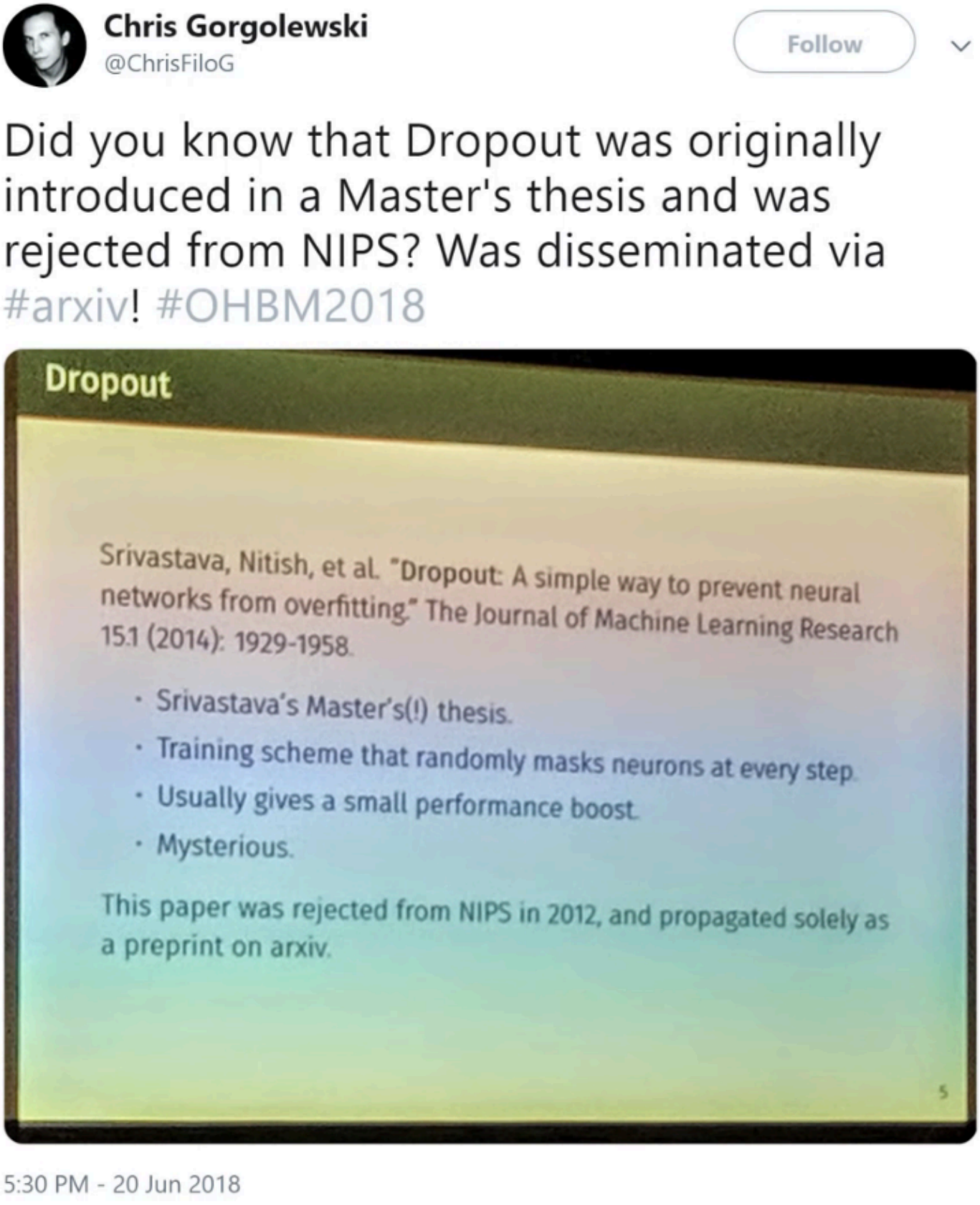
And `bn_cont` is a `nn.BatchNorm1d`. What's that? The first short answer is it's one of the things that I experimented with as to having batchnorm not, and I found that it worked really well. Then specifically what it is is extremely unclear. Let me describe it to you.

- 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.



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*

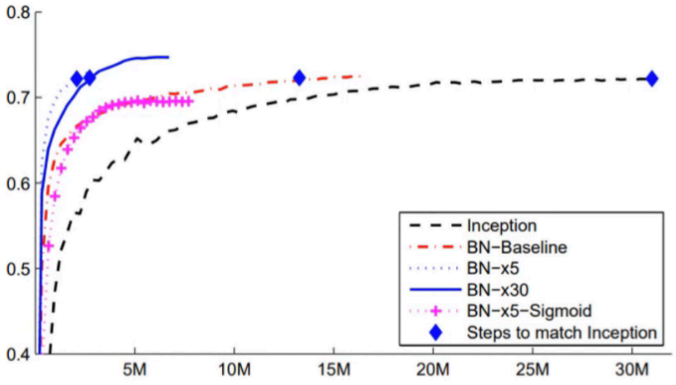


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

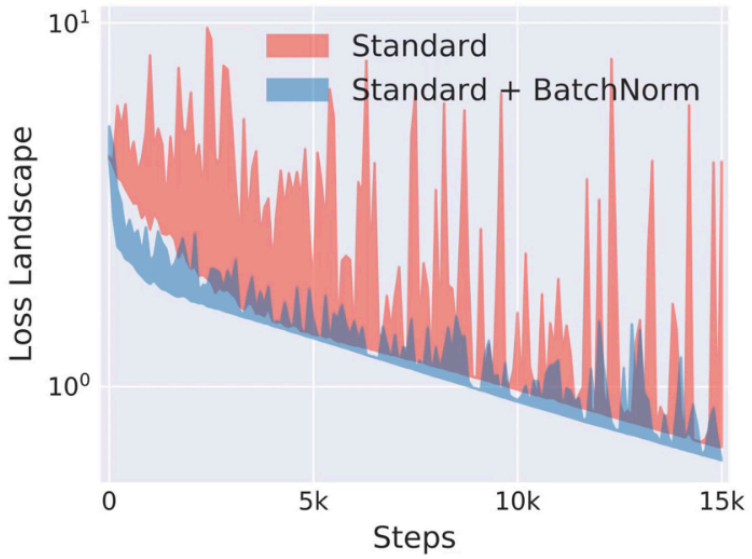
Showing the current then state of the art ImageNet model Inception. This is how long it took them to get a pretty good result, and then they tried the same thing with this new thing called batch norm, and they just did it way way way quickly. That was enough for pretty much everybody to go "wow, this 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.

[45:29]

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: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

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 y-hat 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.

$$\hat{y} = f(w_1, w_2 ... w_{1000000}, \vec{x})$$

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?

$$\hat{y} = f(w_1, w_2 ... w_{1000000}, \vec{x}) \times g + b$$

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.

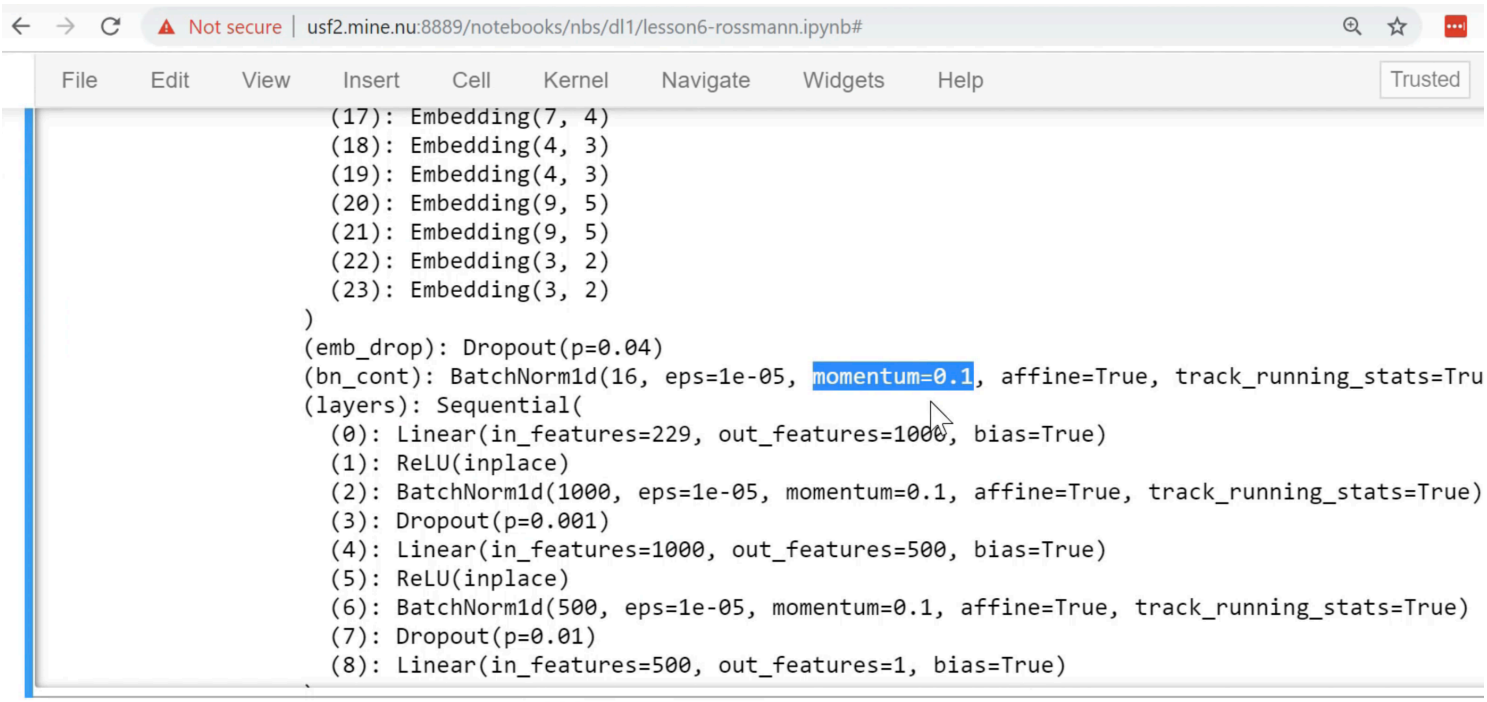
Those details, in some ways, don't matter terribly. The really important thing to know is **you definitely want to use it**. Or if not it, something like it. There's various other types of normalization around nowadays, but batch norm works great. The other main normalization type we use in fast.ai is something called weight norm which is much more just in the last few months' development.

[51:50]

```
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.



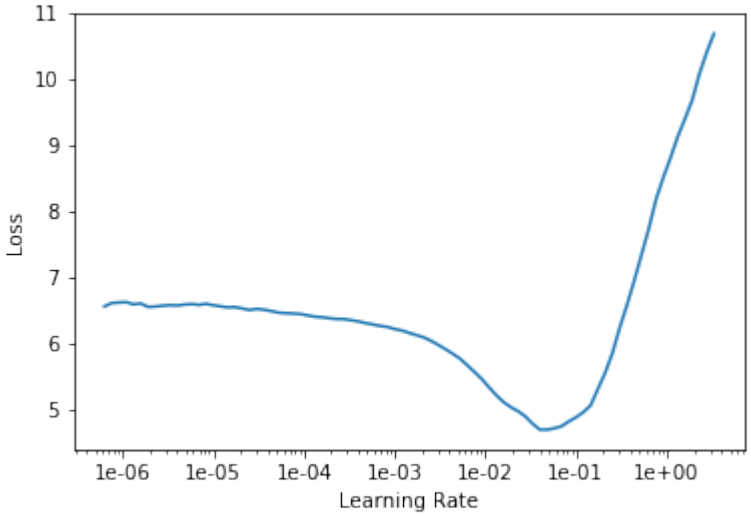
One interesting thing is this momentum here. This is not momentum like in optimization, but this is momentum as in exponentially weighted moving average. Specifically this mean and standard deviation (in batch norm algorithm), we don't actually use a different mean and standard deviation for every mini 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()
```

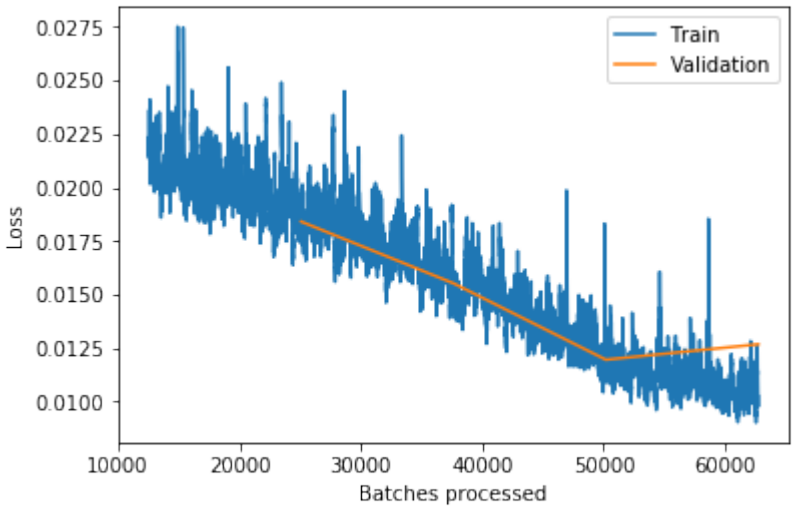



```
learn.fit_one_cycle(5, 1e-3, wd=0.2)
```

```
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:

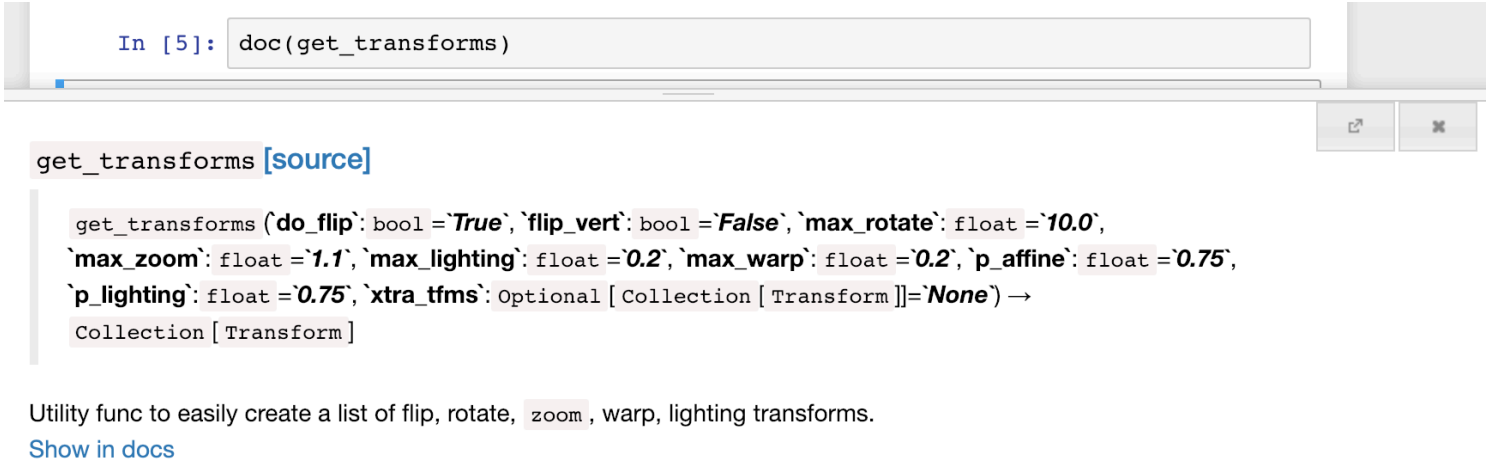
```
%reload_ext autoreload
%autoreload 2
%matplotlib inline

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:



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.

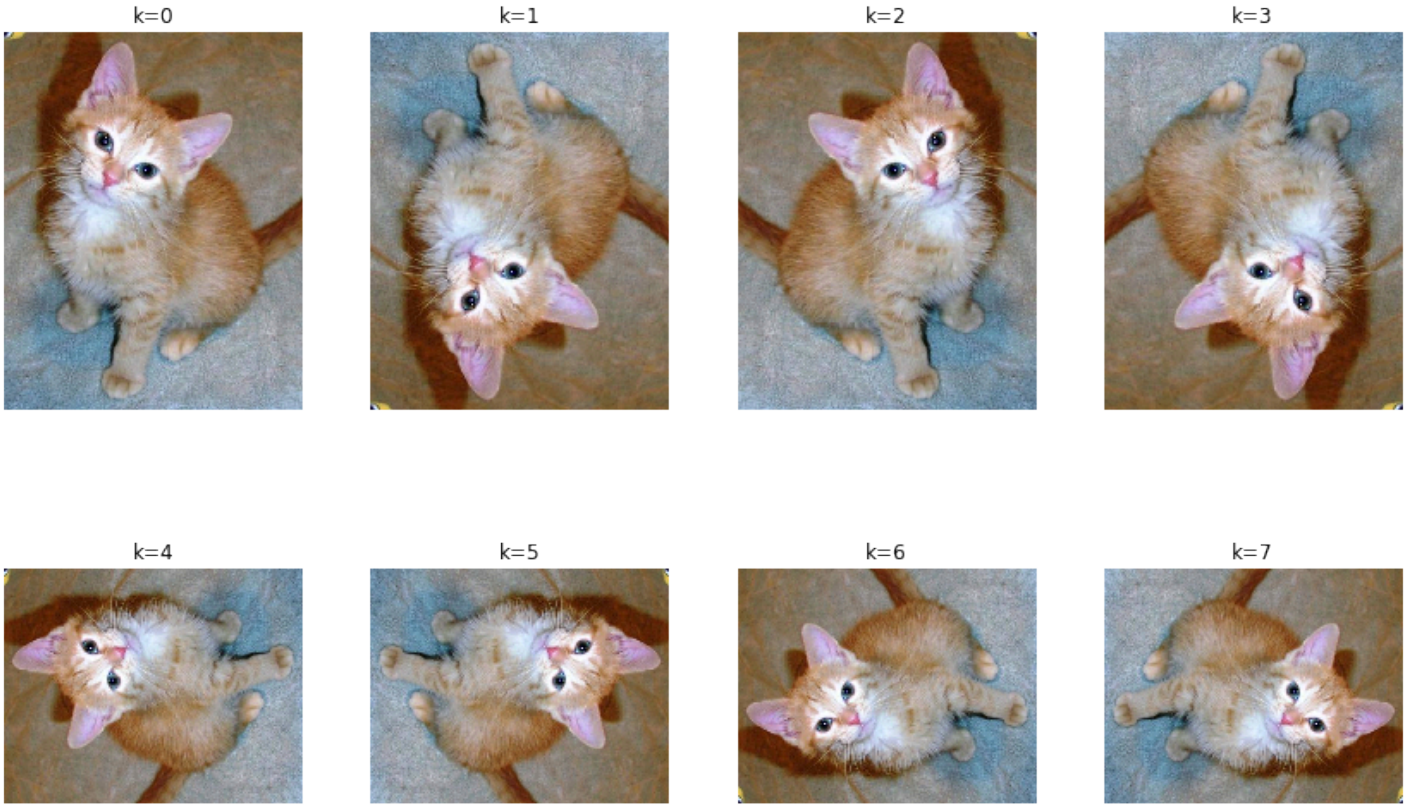
So for example, looking at different values of brightness, what I do here is I look to see two things. The first is for which of these levels of transformation is it still clear what the picture is a picture of. The left most one is kind of getting to a point where it's pretty unclear, the right most 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

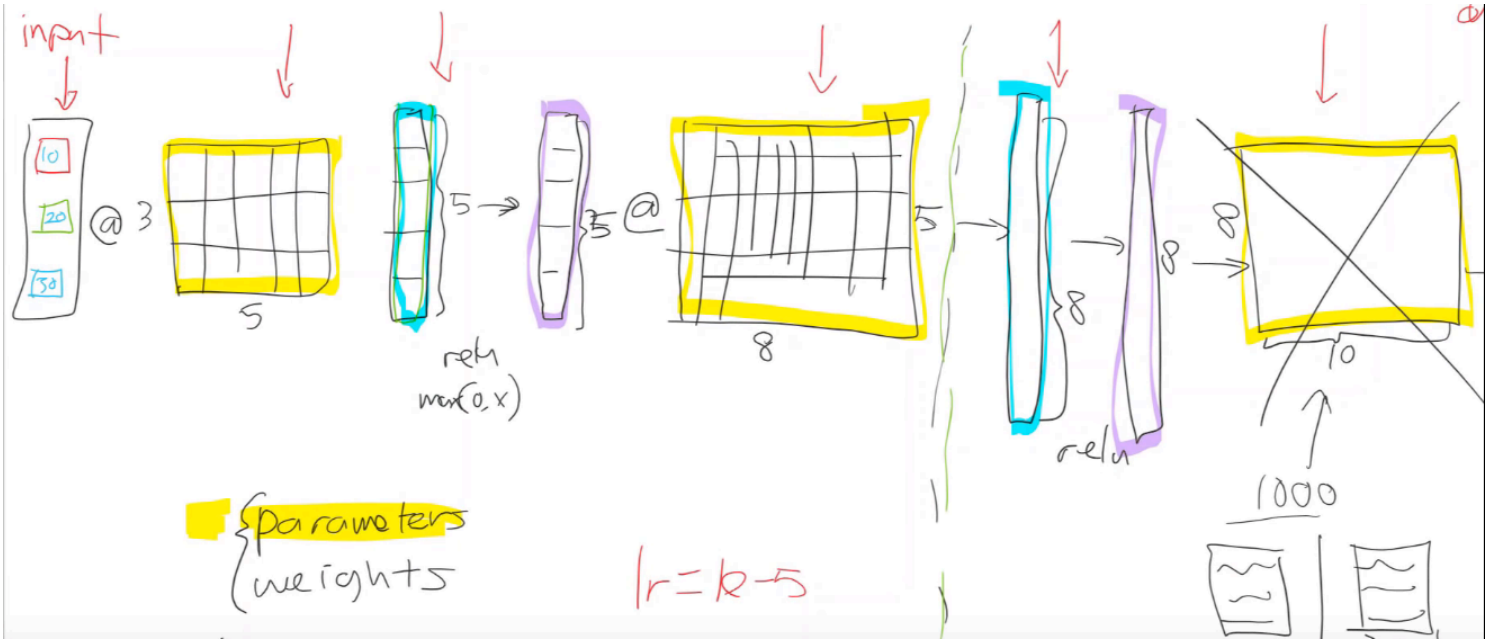
arithmetic in PyTorch, and we're going to try and use that to really understand what's going on.

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 x1 x2 x1 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.

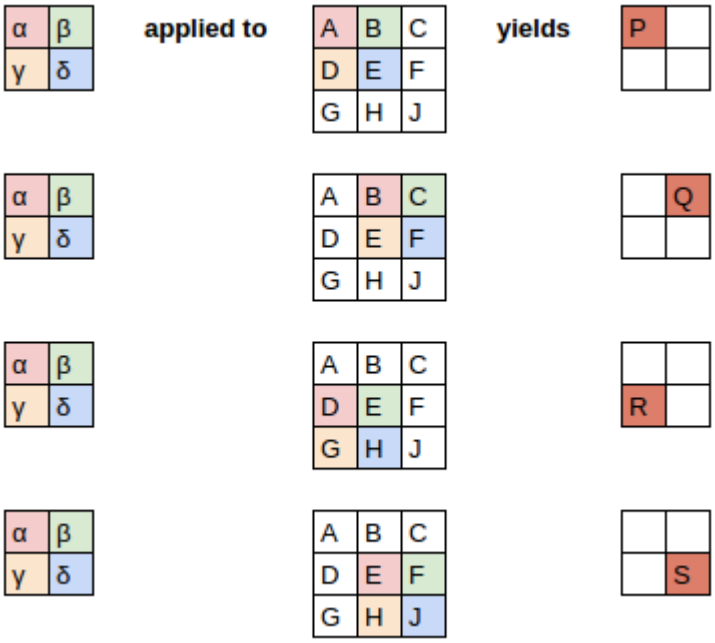
So why are we doing this? Well, perhaps you can see what's happened. This face has turned into some white parts outlining the horizontal edges. How? Well, the how is just by doing this element wise multiplication of each set of 9 pixels with this kernel, adding them together, and sticking the result 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]

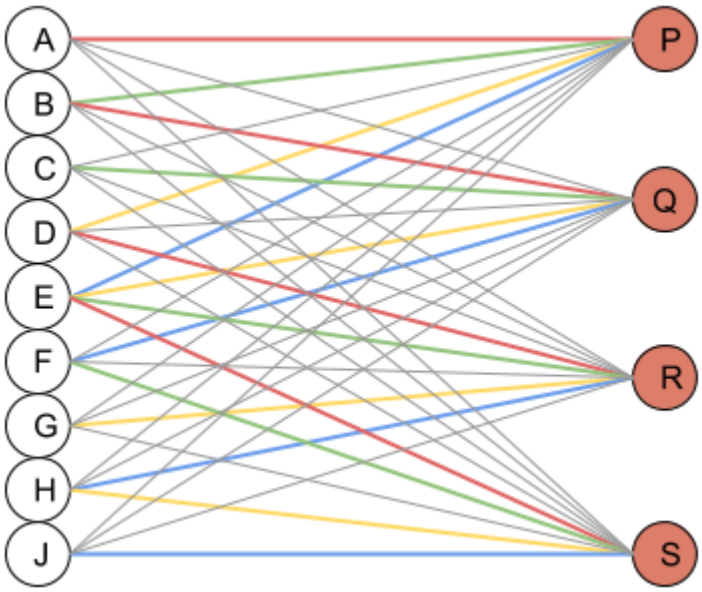
Let's take a look at this from another angle or quite a few other angles. We're going to look at a fantastic post from Matt Kleinsmith who was actually a student in the first year we did this course. He wrote this as a part of his project work back then.



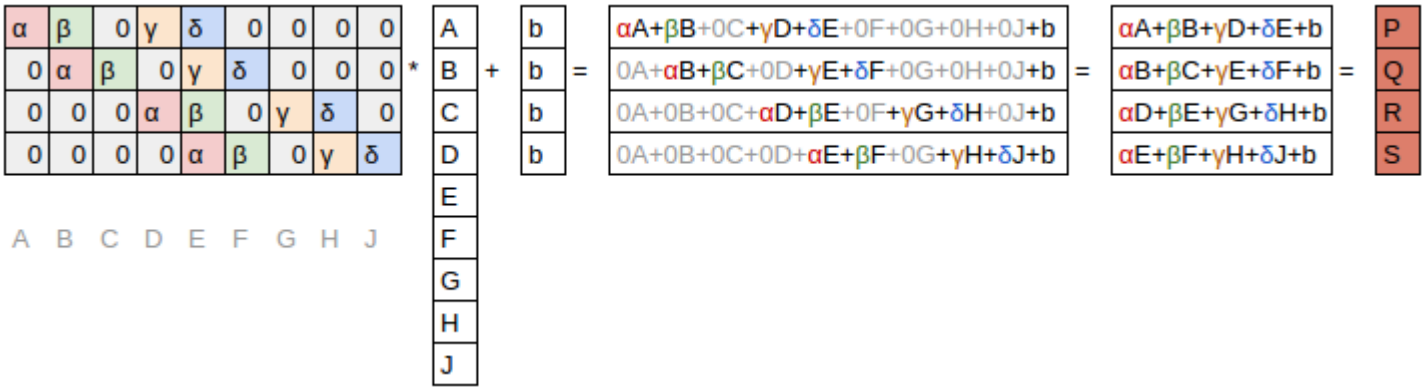
What he's going to show here is here is our image. It's a 3 by 3 image (middle) and our kernel is a 2 by 2 kernel (left). What we're going to do is we're going to apply this kernel to the top left 2x2 part of this image. So the pink bit will be correspondingly multiplied by the pink bit, the green by the green, and so forth. And they all get added up together to create this top left in the output. In other words:

$$\alpha A+\beta B+\gamma D+\delta E+b = P$$
$$\alpha B+\beta C+\gamma E+\delta F+b = Q$$
$$\alpha D+\beta E+\gamma G+\delta H+b = R$$
$$\alpha E+\beta F+\gamma H+\delta J+b = S$$

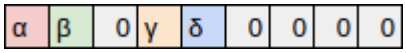
b is a bias so that's fine. That's just a normal bias. So you can see how basically each of these output pixels is the result of some different linear equation. And you can see these same four weights are being moved around, because this is our convolutional kernel.



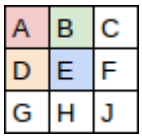
Here is another way of looking at it which is here is a classic neural network view. Now P is result of multiplying every one of these inputs by a weight and then adding them all together, except the gray ones are going to have a value of zero. Because remember, P was only connected to A B D and E. In other words, remembering that this represents a matrix multiplication, therefore we can represent this as a matrix multiplication.



Here is our list of pixels in our 3x3 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:



which corresponds to

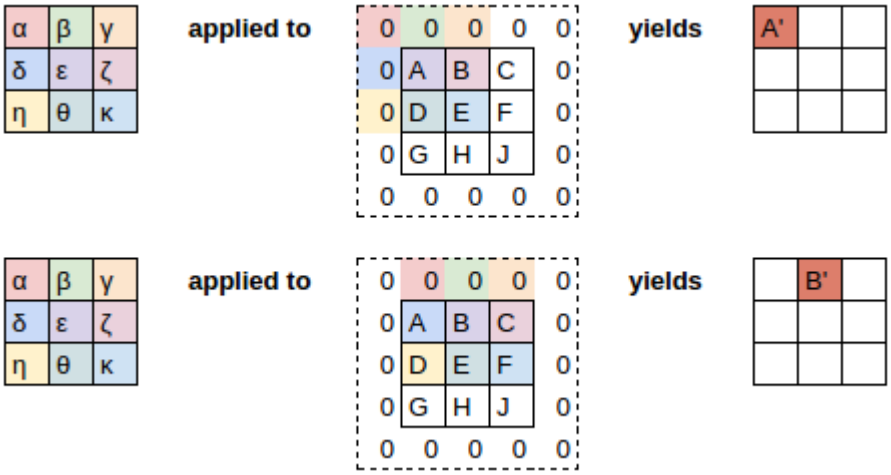


In other words, a convolution is just a matrix multiplication where two things happen:

- 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

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=Fa
(1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stat
(2): ReLU(inplace)
(3): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=Fals
(4): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
    (relu): ReLU(inplace)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
  )
  (1): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
    (relu): ReLU(inplace)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_runnin
  )
```

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])
```

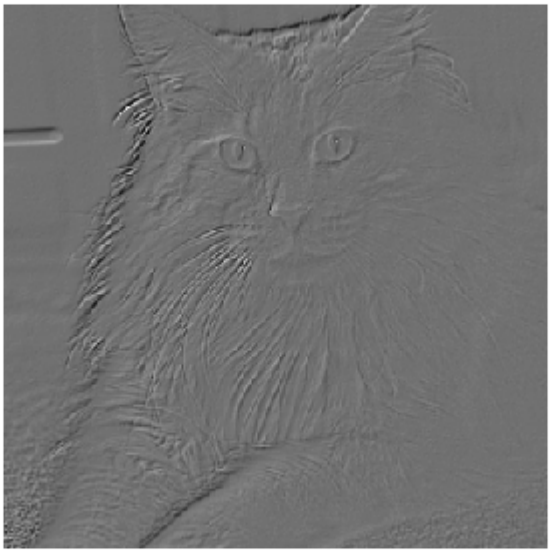
```
t[None].shape
```

```
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]

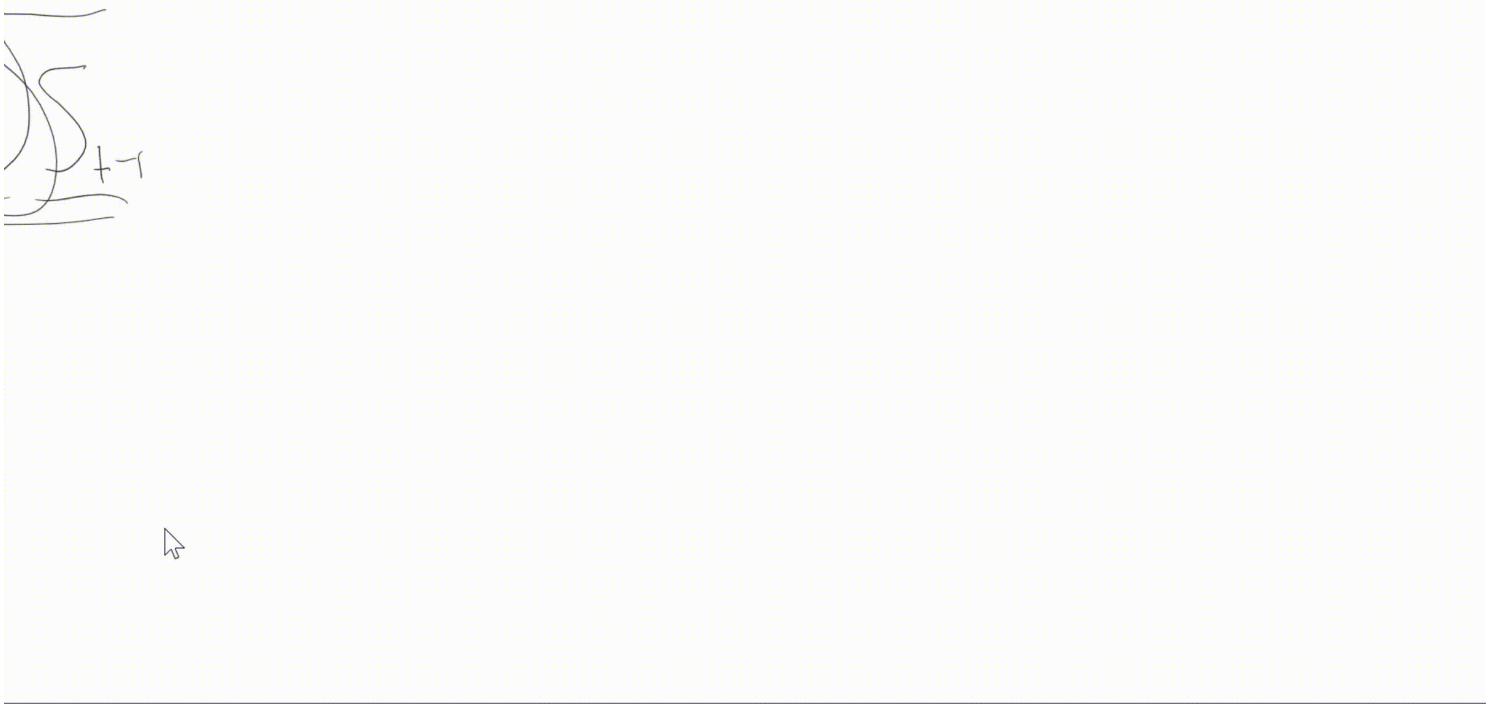
This is where things get fun. Remember mentioned was that I basically have my input red green blue. It goes through a bunch of convolutional layers (let us write a little line to say a convolutional layer) to create activations which have more and more channels and smaller and smaller height by widths. 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**.



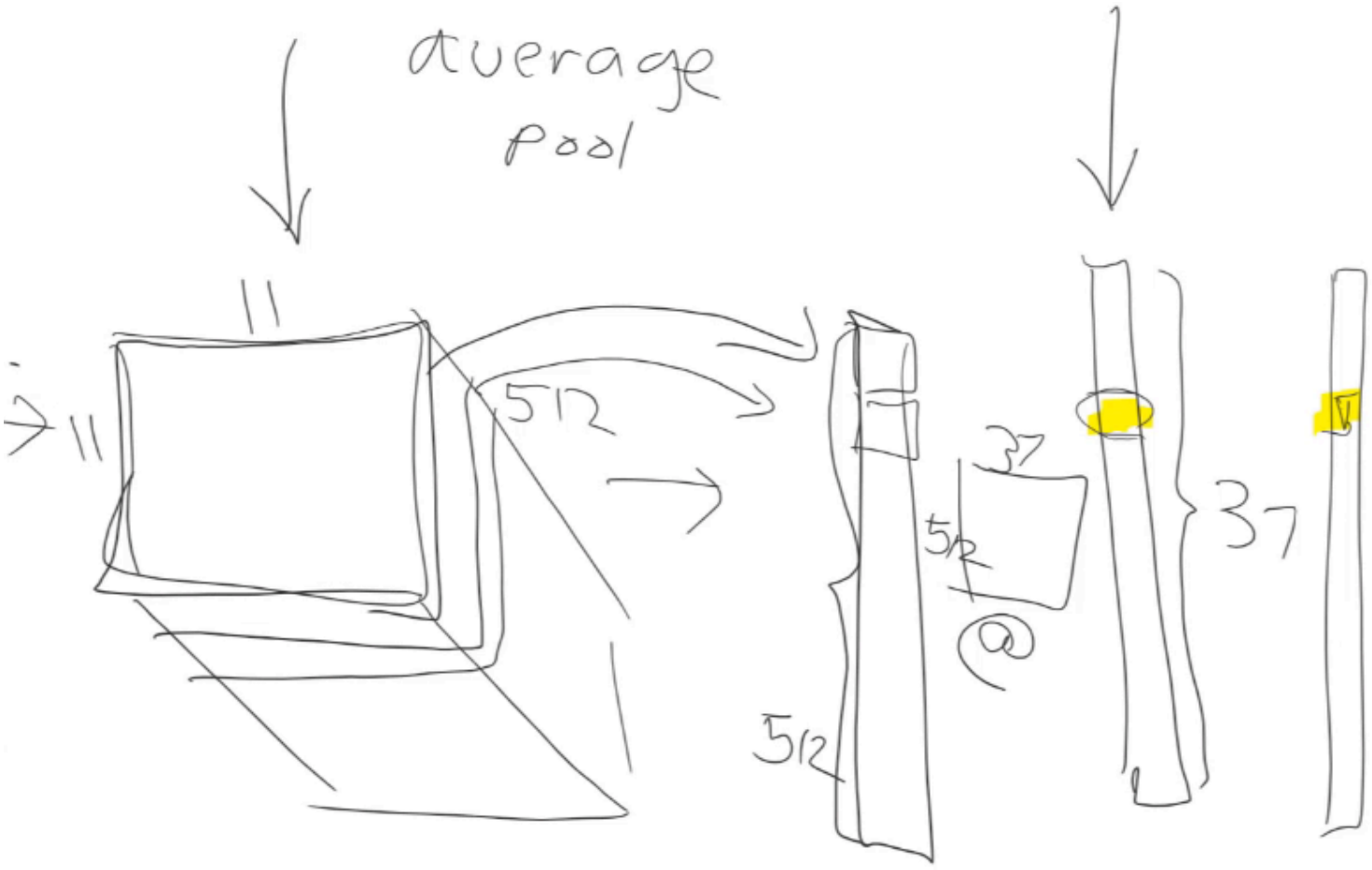
[1:36:52]

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)
  )
)
```

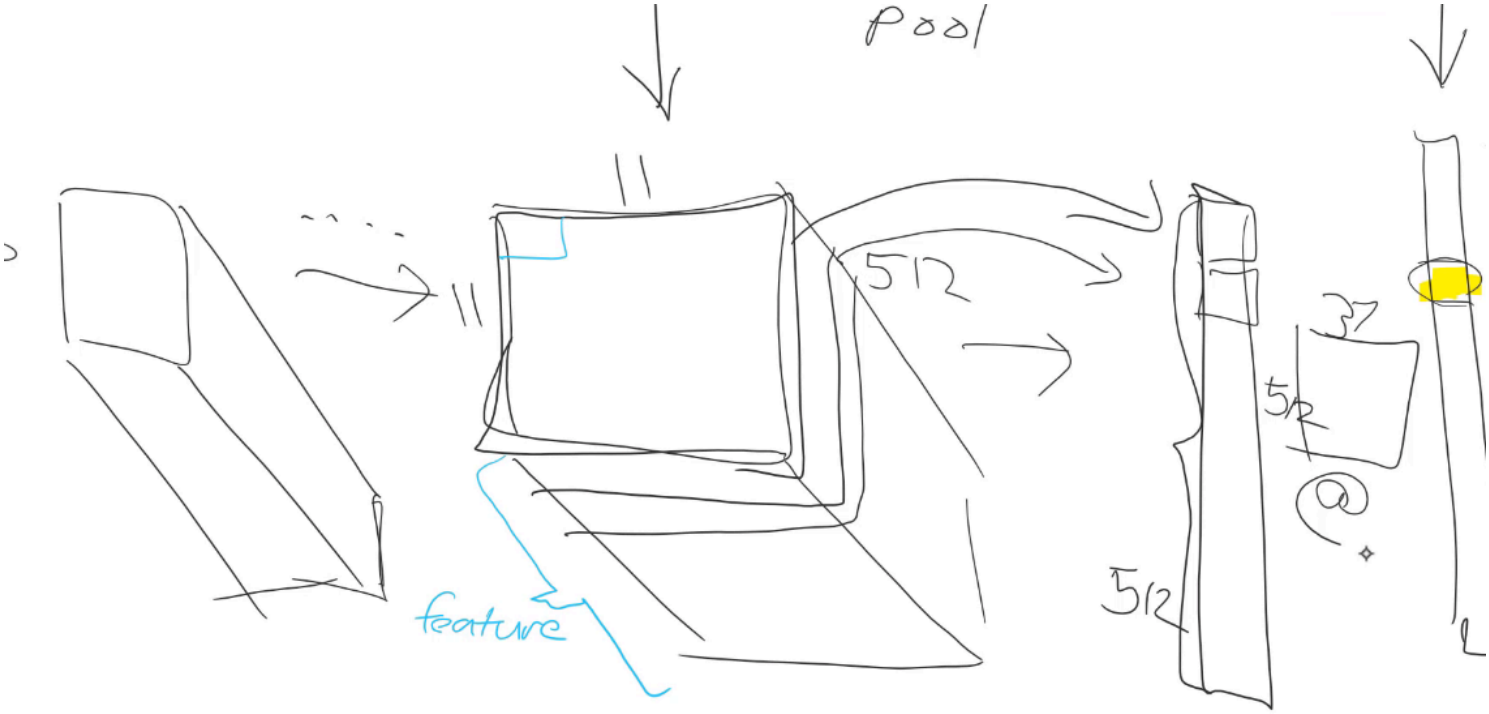
Here is our final 512. We will talk about what a concat pooling is in part 2, for now, we'll just focus on that this is a fast.ai specialty. Everybody else just does this `AdaptiveAvgPool2` with an output size of one.

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
```

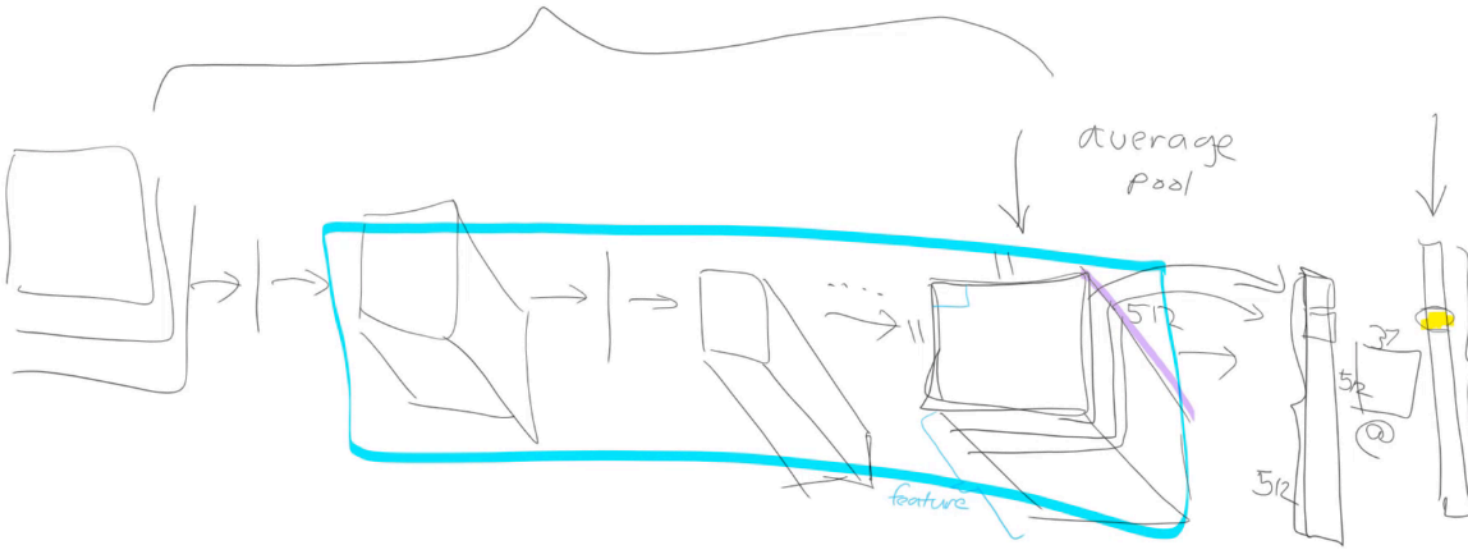
```
torch.Size([11, 11])
```

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.

Ethics and Data Science [1:49:10]

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 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.

Gender Classifier	Darker Male	Darker Female	Lighter Male	Lighter Female	Largest Gap
Microsoft	94.0%	79.2%	100%	98.3%	20.8%
FACE++	99.3%	65.5%	99.2%	94.0%	33.8%
IBM	88.0%	65.3%	99.7%	92.9%	34.4%

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.

No Classification without Representation: Assessing Geodiversity Issues in Open Data Sets for the Developing World

Shreya Shankar, Yoni Halpern, Eric Breck, James Atwood, Jimbo Wilson, D. Sculley

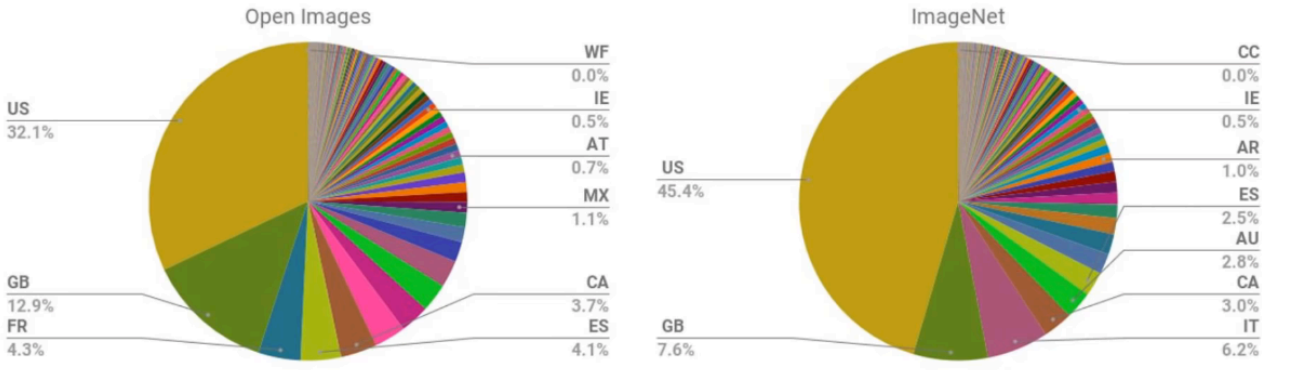
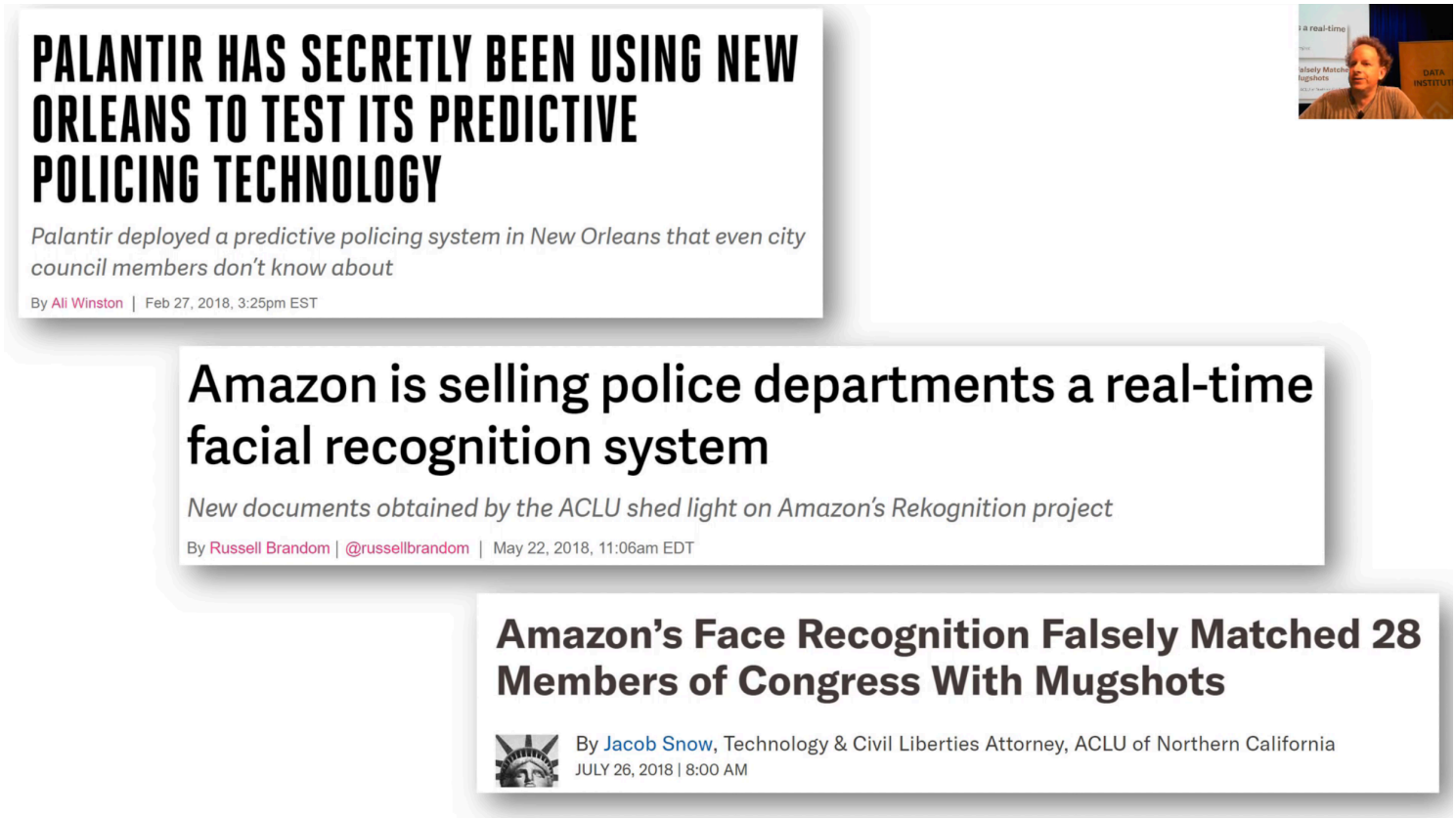


Figure 1: Fraction of Open Images and ImageNet images from each country. In both data sets, top represented locations include the US and Great Britain. Countries are represented by their two-letter ISO country codes.

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.

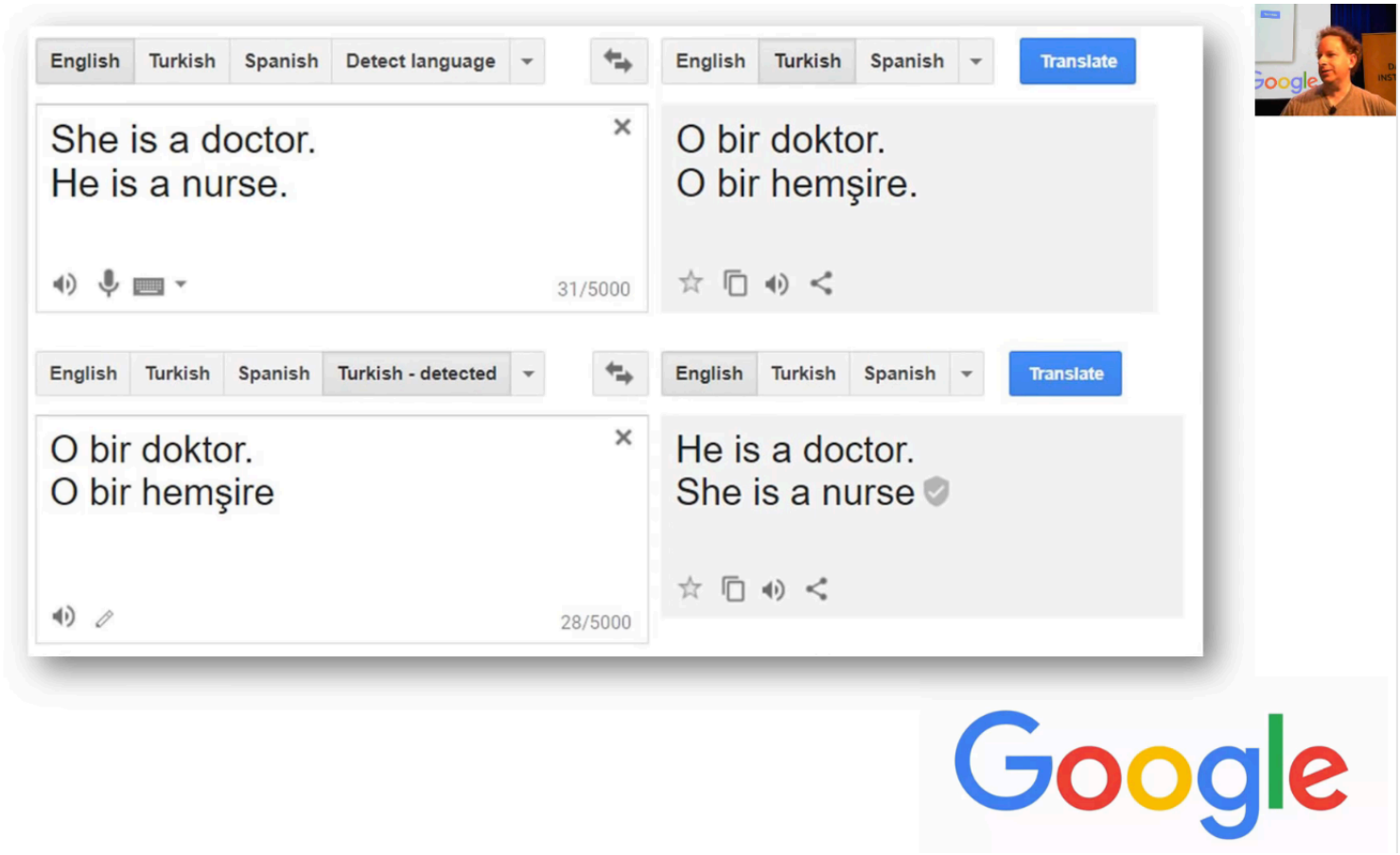
When you have biased data creating biased algorithms, you then need to ask "what are we doing with that?" We've been spending a lot of time talking about image recognition. A couple of years ago, this company DeepGlint advertised their image recognition system which can be used to do mass surveillance 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?"



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

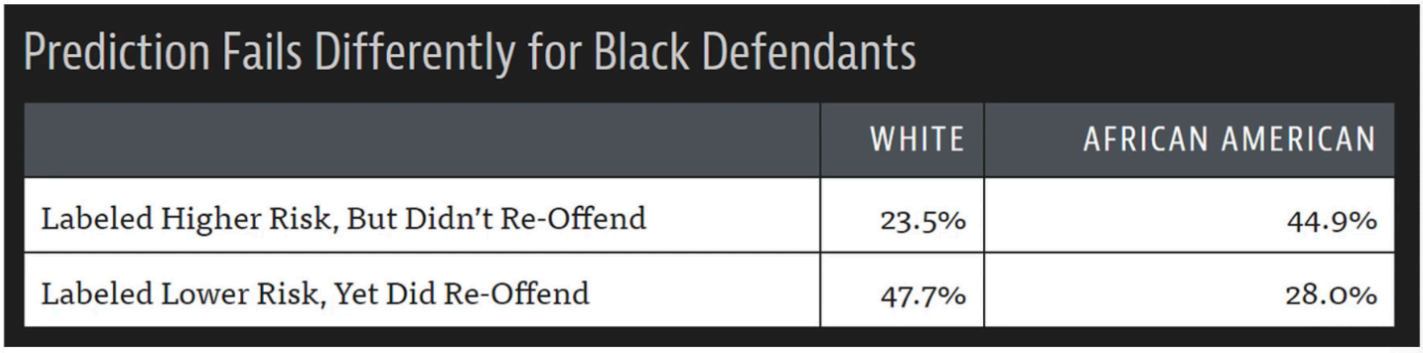
systems are extremely inaccurate, and extremely biased. And what happens then, of course, is if you're in a predominantly black neighborhood where the probability of successfully recognizing you is much lower, and you're much more likely to be surrounded by black people, and so suddenly all of these 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).



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.

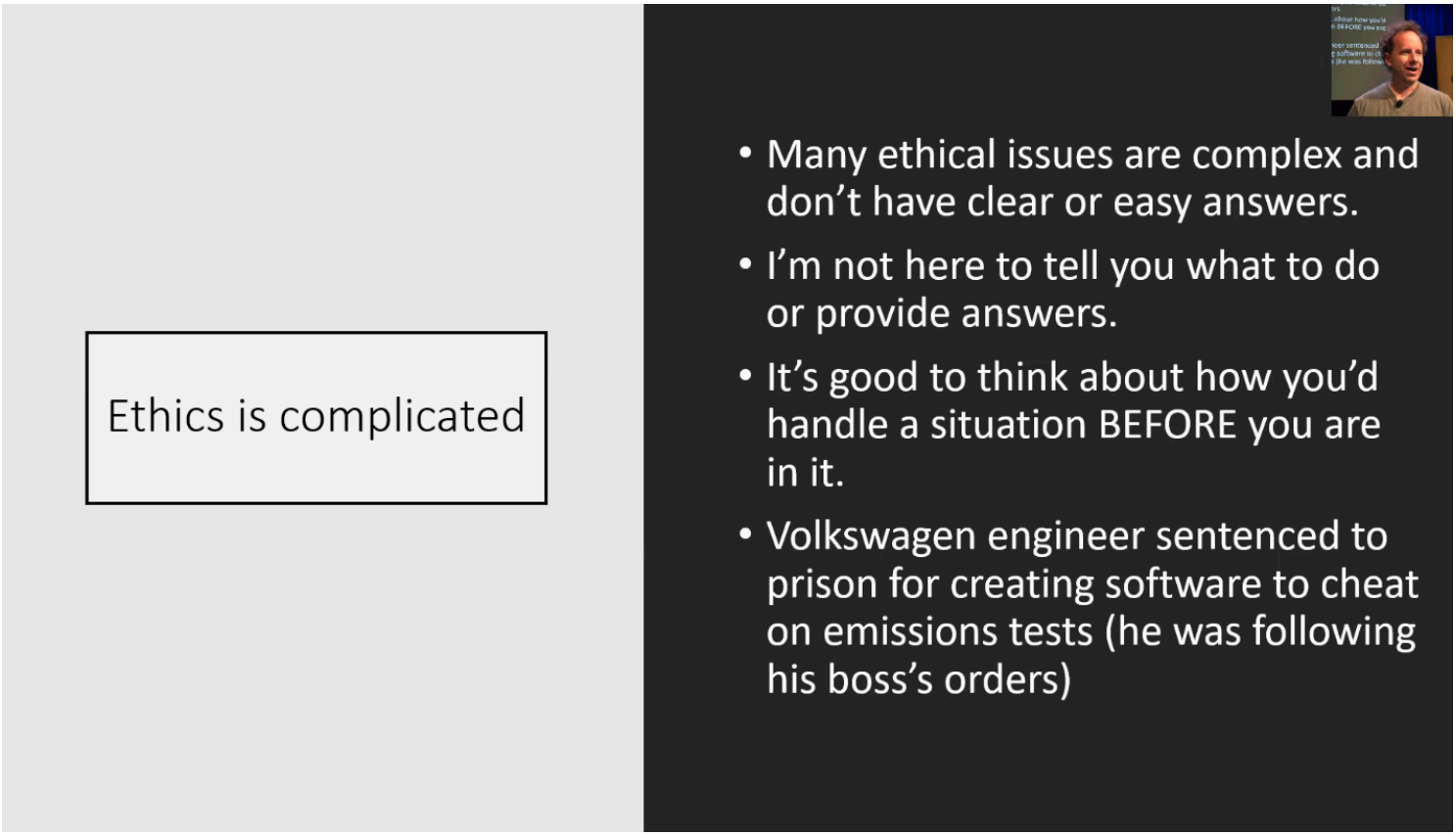


For example, there's a system called COMPAS which is very widely used to decide who's going to jail. It does that in a couple of ways; it tells judges what sentencing guidelines they should use for particular cases, and it tells them also which people (the system says) should be let out on bail. But 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.

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.



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.

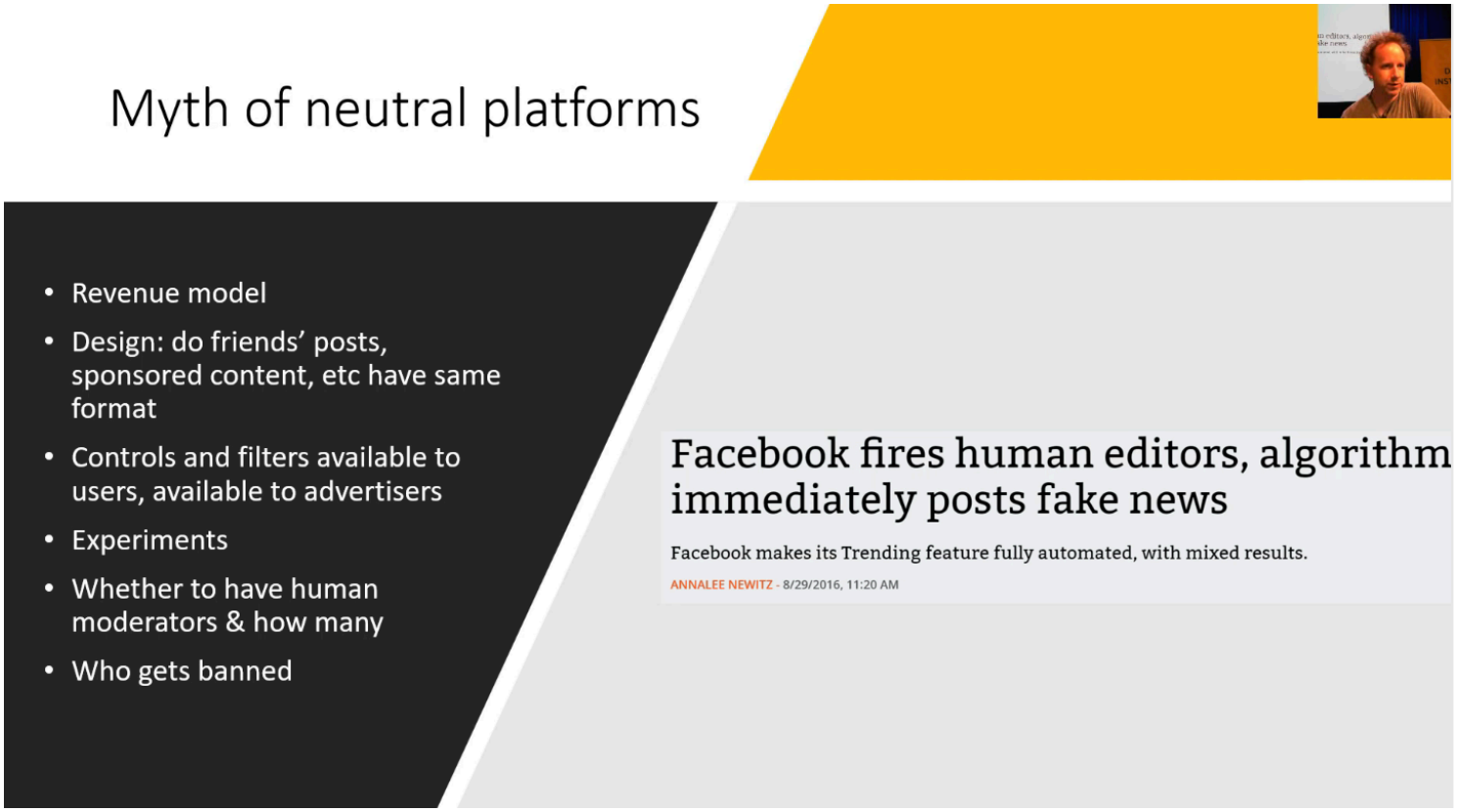
In the concentration camps, IBM's code for Jews was 8. Its code for Gypsies was 12. General executions were coded as 4, death in the gas chambers as 6. Only Jews and Gypsies were systematically murdered in gas chambers.

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."

"To the blind technocrat, the means were more important than the ends. The destruction of the Jewish people became even less important because the invigorating nature of IBM's technical achievement was only heightened by the fantastical profits to be made at a time when bread lines stretched across the world."

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.



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)

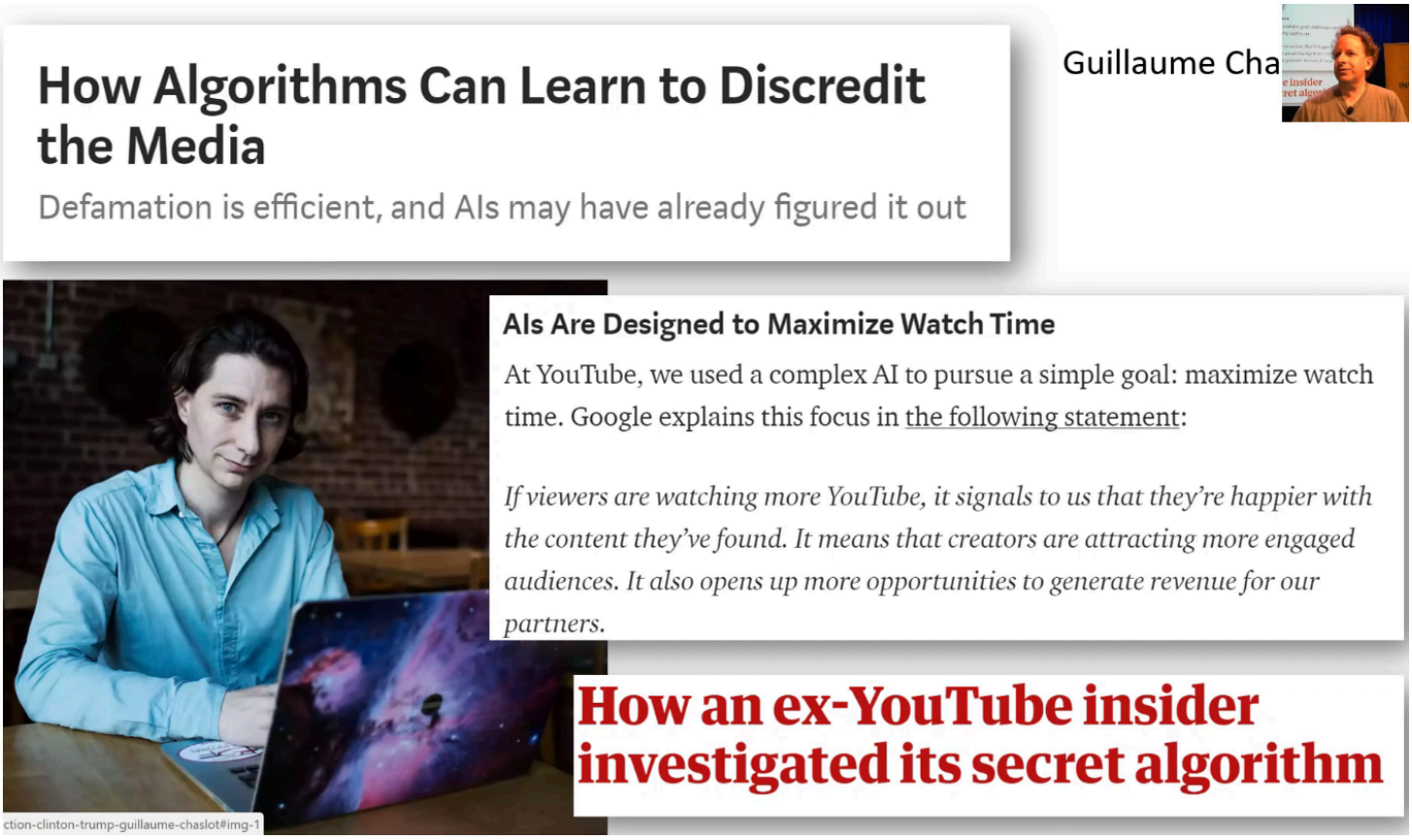
A great example was here's an example of somebody who lost their health care. They lost their health care because of an error in a new algorithm that was systematically failing to recognize that there are many people that need help with cerebral palsy and diabetes. So this system which had this error 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.

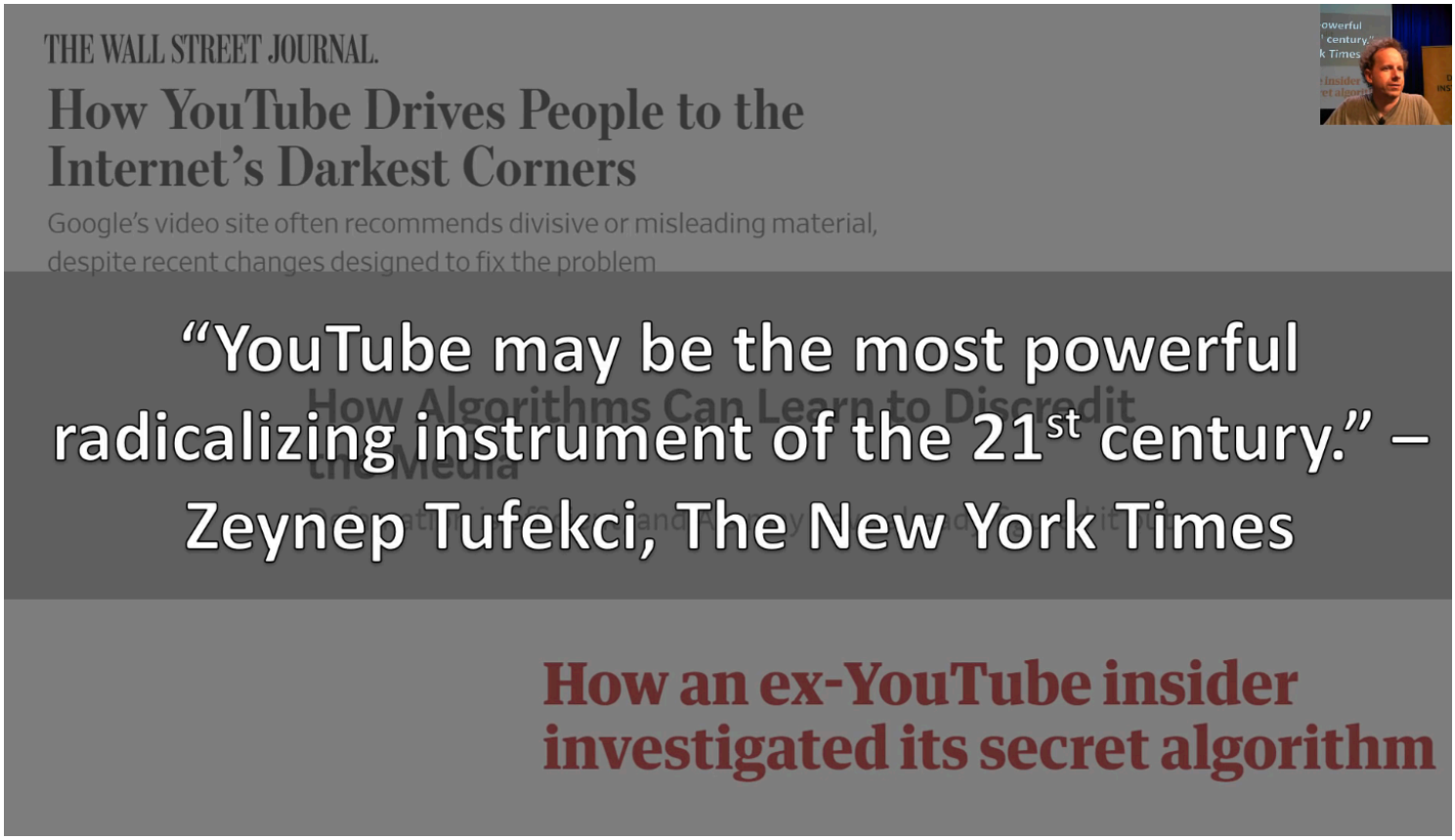
And you know, in some ways maybe it's unfair, but I would argue the person who is creating the data set and the person who is implementing the algorithm is the person best placed to get out there and say "hey here are the things you need to be careful of" and make sure that they are part of the implementation process.



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.



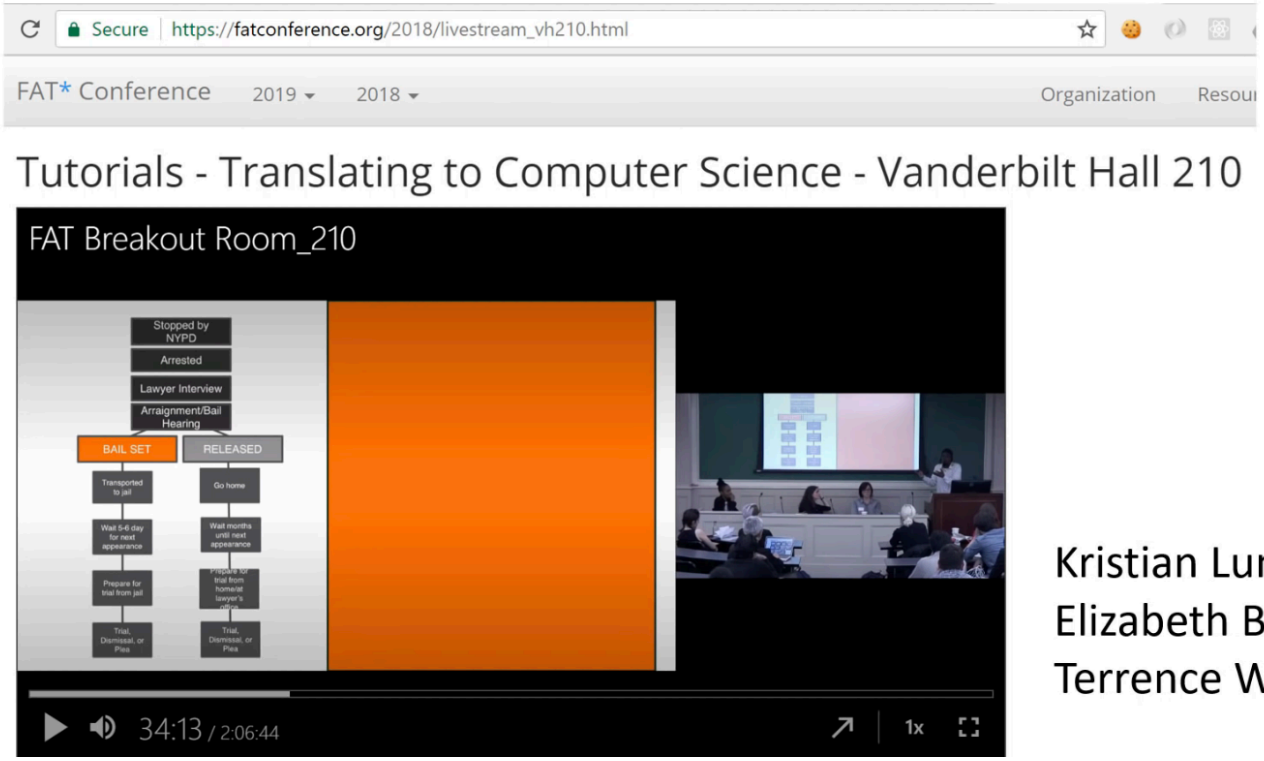
So this is an example of a feedback loop. The New York Times is now describing YouTube as perhaps the most powerful radicalizing instrument of the 21st century. I can tell you my friends that worked on the YouTube recommendation system did not think they were creating the most powerful radicalizing instrument 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.

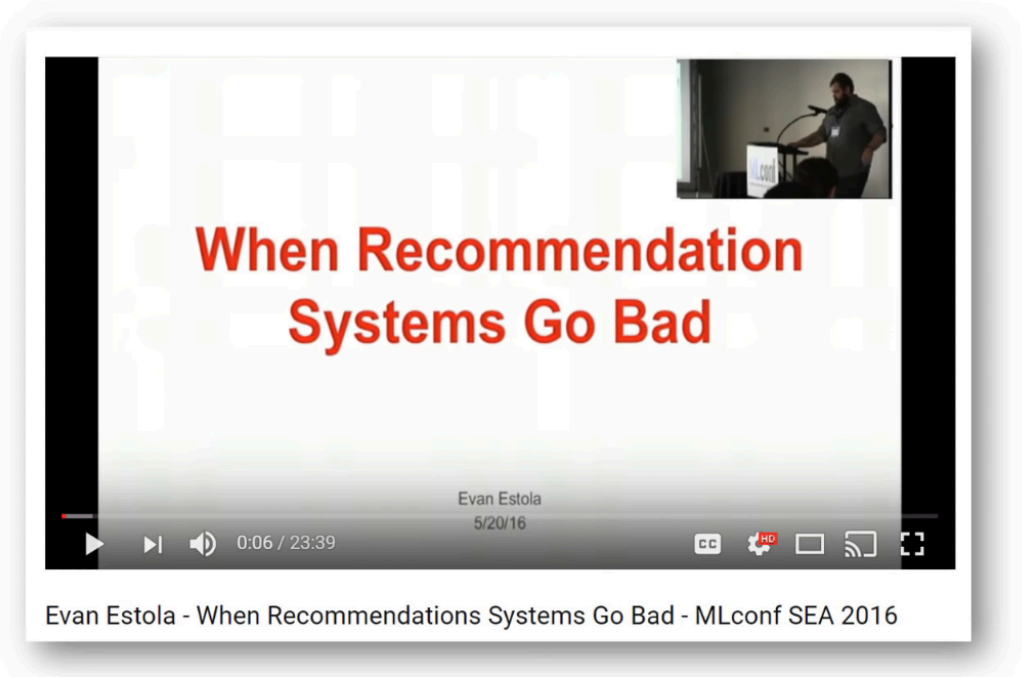


Kristian Lum,
Elizabeth Bender, &
Terrence Wilkerson

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



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 man, 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.

Datasheets for Datasets*

Timnit Gebru[1], Jamie Morgenstern[2], Briana Vecchione[3],
Jennifer Wortman Vaughan[1], Hanna Wallach[1],
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 the0.3 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!