



Hiromi Suenaga [Follow](#)
Jan 12 · 12 min read

Deep Learning 2: Part 1 Lesson 1

My personal notes from fast.ai course. These notes will continue to be updated and improved as I continue to review the course to “really” understand it. Much appreciation to Jeremy and Rachel who gave me this opportunity to learn.

. . .

Lesson 1

Getting started [0:00]:

- In order to train a neural network, you will most certainly need Graphics Processing Unit (GPU)—specifically NVIDIA GPU because it is the only one that supports CUDA (the language and framework that nearly all deep learning libraries and practitioners use).
- There are several ways to rent GPU: Crestle [04:06], Paperspace [06:10]

Introduction to Jupyter Notebook and Dogs vs. Cats [12:39]

- You can run a cell by selecting it and hitting `shift+enter` (you can hold down `shift` and hit `enter` multiple times to keep going down the cells), or you can click on Run button at the top. A cell can contain code, text, picture, video, etc.
- Fast.ai requires Python 3

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

# This file contains all the main external libs we'll use
from fastai.imports import *

from fastai.transforms import *
from fastai.conv_learner import *
from fastai.model import *
from fastai.dataset import *
from fastai.sgdr import *
from fastai.plots import *

PATH = "data/dogscats/"
sz=224
```

First look at pictures [15:39]

```
!ls {PATH}

models  sample  test1  tmp  train  valid
```

- `!` tells to use bash (shell) instead of python
- If you are not familiar with training set and validation set, check out Practical Machine Learning class (or read [Rachel's blog](#))

```
!ls {PATH}valid

cats  dogs

files = !ls {PATH}valid/cats | head
files

['cat.10016.jpg',
 'cat.1001.jpg',
 'cat.10026.jpg',
 'cat.10048.jpg',
 'cat.10050.jpg',
 'cat.10064.jpg',
 'cat.10071.jpg',
 'cat.10091.jpg',
 'cat.10103.jpg',
 'cat.10104.jpg']
```

- This folder structure is the most common approach for how image classification dataset is shared and provided. Each folder tells you the label (e.g. `dogs` or `cats`).

```
img = plt.imread(f'{PATH}valid/cats/{files[0]}')
plt.imshow(img);
```



- `f'{PATH}valid/cats/{files[0]}'` —This is a Python 3.6. format string which is a convenient to format a string.

```
img.shape  
  
(198, 179, 3)  
  
img[:4,:4]  
  
array([[[ 29,  20,  23],  
       [ 31,  22,  25],  
       [ 34,  25,  28],  
       [ 37,  28,  31]],  
  
      [[ 60,  51,  54],  
       [ 58,  49,  52],  
       [ 62,  53,  56],  
       [ 65,  55,  58]]])
```

```
[ 56,  47,  50],  
[ 55,  46,  49]],  
  
[[ 93,  84,  87],  
 [ 89,  80,  83],  
 [ 85,  76,  79],  
 [ 81,  72,  75]],  
  
[[104,  95,  98],  
 [103,  94,  97],  
 [102,  93,  96],  
 [102,  93,  96]]], dtype=uint8)
```

- `img` is a 3 dimensional array (a.k.a. rank 3 tensor)
- The three items (e.g. `[29, 20, 23]`) represents Red Green Blue pixel values between 0 and 255
- The idea is to take these numbers and use them to predict whether those numbers represent a cat or a dog based on looking at lots of pictures of cats and dogs.
- This dataset comes from [Kaggle competition](#), and when it was released (back in 2013) the state-of-the-art was 80% accurate.

Let's train a model [20:21]

Here are the three lines of code necessary to train a model:

```
data = ImageClassifierData.from_paths(PATH,  
tfms=tfms_from_model(resnet34, sz))  
learn = ConvLearner.pretrained(resnet34, data,
```

```
precompute=True)
learn.fit(0.01, 3)
```

```
[ 0.      0.04955  0.02605  0.98975]
[ 1.      0.03977  0.02916  0.99219]
[ 2.      0.03372  0.02929  0.98975]
```



- This will do 3 **epochs** which means it is going to look at the entire set of images three times.
- The last of three numbers in the output is the accuracy on the validation set.
- The first two are the value of loss function (in this case the cross-entropy loss) for the training set and the validation set.
- The start (e.g. 0. , 1.) is the epoch number.
- We achieved ~99% (which would have won the Kaggle competition back in 2013) in 17 seconds with 3 lines of code!
[21:49]
- A lot of people assume that deep learning takes a huge amount of time, lots of resources, and lots of data—that, in general, is not true!

Fast.ai Library [22:24]

- The library takes all of the best practices and approaches they can find—each time a paper comes out that looks interesting, they test it out and if it works well for a variety of datasets and they can figure out how to tune it, it gets implemented in the library.

- Fast.ai curates all these best practices and packages up for you, and most of the time, figures out the best way to handle things automatically.
- Fast.ai sits on top of a library called PyTorch which is a really flexible deep learning, machine learning, and GPU computation library written by Facebook.
- Most people are more familiar with TensorFlow than PyTorch, but most of the top researchers Jeremy knows nowadays have switched across to PyTorch.
- Fast.ai is flexible that you can use all these curated best practices as much or as little as you want. It is easy to hook in at any point and write your own data augmentation, loss function, network architecture, etc, and we will learn all that in this course.

Analyzing results [24:21]

This is what the validation dataset label (think of it as the correct answers) looks like:

```
data.val_y  
  
array([0, 0, 0, ..., 1, 1, 1])
```

What do these 0's and 1's represent?

```
data.classes  
['cats', 'dogs']
```

- `data` contains the validation and training data
- `learn` contains the model

Let's make predictions for the validation set (predictions are in log scale):

```
log_preds = learn.predict()  
log_preds.shape  
  
(2000, 2)  
  
log_preds[:10]  
  
array([[ -0.00002, -11.07446],  
       [ -0.00138, -6.58385],  
       [ -0.00083, -7.09025],  
       [ -0.00029, -8.13645],  
       [ -0.00035, -7.9663 ],  
       [ -0.00029, -8.15125],  
       [ -0.00002, -10.82139],  
       [ -0.00003, -10.33846],  
       [ -0.00323, -5.73731],  
       [ -0.0001 , -9.21326]], dtype=float32)
```

- The output represents a prediction for cats, and prediction for dogs

```
preds = np.argmax(log_preds, axis=1) # from log
probabilities to 0 or 1
probs = np.exp(log_preds[:,1]) # pr(dog)
```

- In PyTorch and Fast.ai, most models return the log of the predictions rather than the probabilities themselves (we will learn why later in the course). For now, just know that to get probabilities, you have to do `np.exp()`

Definition of natural logarithm

When

$$e^y = x$$

Then base e logarithm of x is

$$\ln(x) = \log_e(x) = y$$

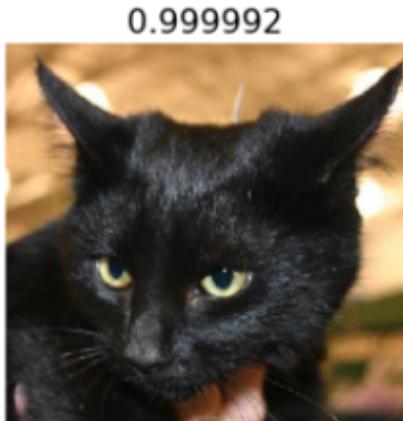
- Make sure you familiarize yourself with numpy (`np`)

```
# 1. A few correct labels at random
plot_val_with_title(rand_by_correct(True), "Correctly
classified")
```

- The number above the image is the probability of being a dog

```
# 2. A few incorrect labels at random
plot_val_with_title(rand_by_correct(False), "Incorrectly
classified")
```

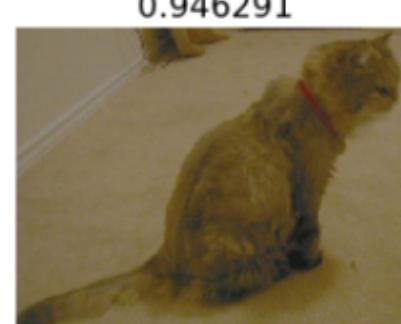
Incorrectly classified



0.999992



0.906738



0.946291



0.504476

```
plot_val_with_title(most_by_correct(0, True), "Most correct cats")
```

Most correct cats

1.59057e-07



3.41795e-07



5.54502e-07



5.64355e-07



```
plot_val_with_title(most_by_correct(1, True), "Most correct dogs")
```

Most correct dogs

1.0



1.0



1.0



1.0



More interestingly, here are what the model thought it was definitely a dog but turns out to be a cat, or vice versa:

```
plot_val_with_title(most_by_correct(0, False), "Most  
incorrect cats")
```

Most incorrect cats

0.999992



0.999984



0.999887



0.998166



The Caring Containment Professionals.SM

```
plot_val_with_title(most_by_correct(1, False), "Most  
incorrect dogs")
```

Most incorrect dogs

0.0372201



0.0434048



0.0733013



0.134101



```
most_uncertain = np.argsort(np.abs(probs -0.5))[:4]
plot_val_with_title(most_uncertain, "Most uncertain
predictions")
```

Most uncertain predictions

0.504476



0.522612



0.52306



0.456155



- Why is it important to look at these images? The first thing Jeremy does after he builds a model is to find a way to visualize what it has built. Because if he wants to make the model better, then he needs to take advantage of the things that is doing well and fix the things that is doing badly.

- In this case, we have learned something about the dataset itself which is that there are some images that are in here that probably should not be. But it is also clear that this model has room to improve (e.g. data augmentation—which we will learn later).
- Now you are ready to build your own image classifier (for regular photos—maybe not CT scan)! For example, [here](#) is what one of the students did.
- Check out [this forum post](#) for different way of visualizing the results (e.g. when there are more than 2 categories, etc)

Top-down vs Bottom-up [30:52]

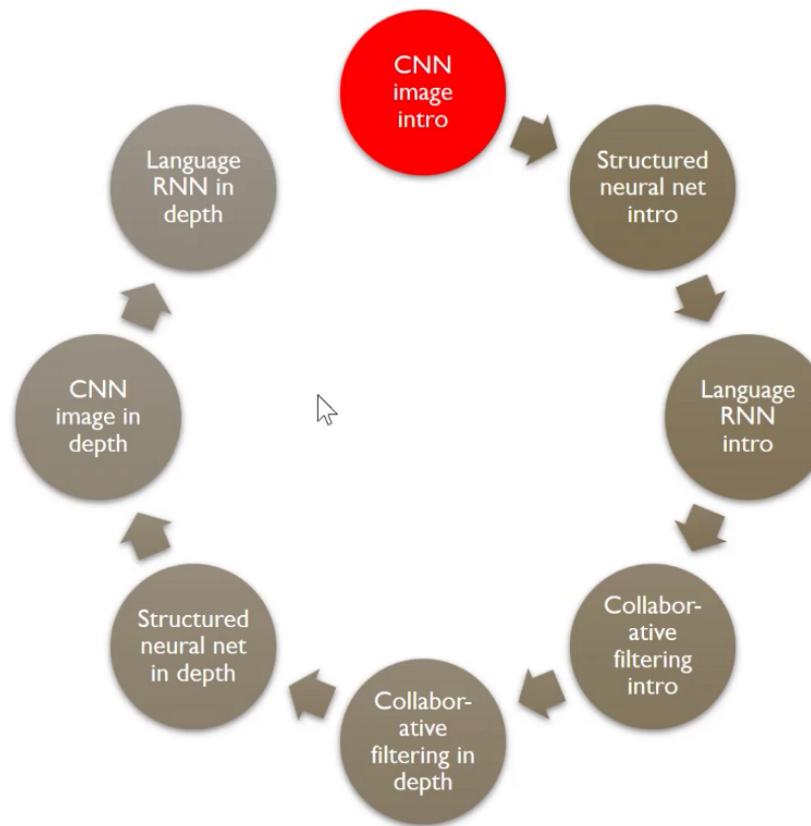
Bottom-up: learn each building block you need, and eventually put them together

- Hard to maintain motivation
- Hard to know the “big picture”
- Hard to know which pieces you’ll actually need

fast.ai: Get students using a neural net right away, getting results ASAP

- Gradually peel back the layers, modify, look under the hood

Course Structure [33:53]



1. Image classifier with deep learning (with fewest lines of code)
2. Multi-label classification and different kinds of images (e.g. satellite images)
3. Structured data (e.g. sales forecasting)—structured data is what comes from database or spreadsheet
4. Language: NLP classifier (e.g. movie review classification)
5. Collaborative filtering (e.g. recommendation engine)

6. Generative language model: How to write your own Nietzsche philosophy from scratch character by character
7. Back to computer vision—not just recognize a cat photo, but find where the cat is in the photo (heat map) and also learn how to write our own architecture from scratch (ResNet)

Image Classifier Examples:

Image classification algorithm is useful for lots and lots of things.

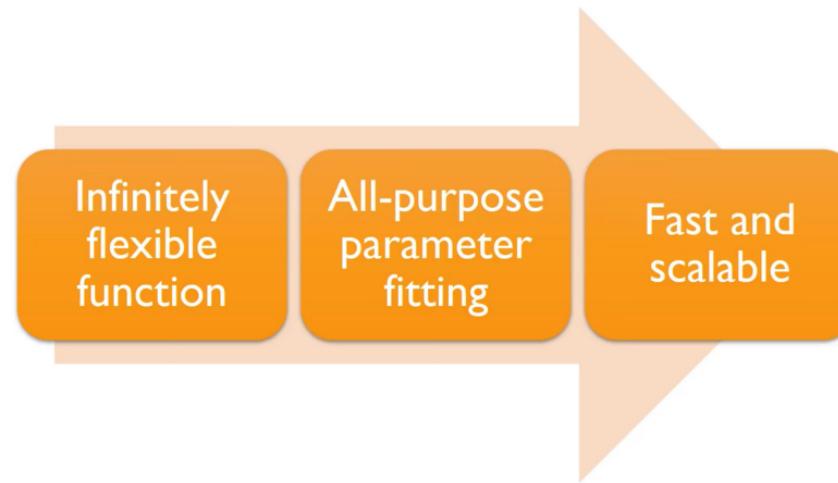
- For example, AlphaGo [42:20] looked at thousands and thousands of go boards and each one had a label saying whether the go board ended up being the winning or the losing player's. So it learnt an image classification that was able to look at a go board and figure out whether it was a good or bad—which is the most important step in playing go well: to know which move is better.
- Another example is an earlier student created an image classifier of mouse movement images and detected fraudulent transactions.

Deep Learning ≠ Machine Learning [44:26]

- Deep learning is a kind of machine learning
- Machine learning was invented by Arthur Samuel. In the late 50s, he got an IBM mainframe to play checkers better than he could by inventing machine learning. He made the mainframe to play against itself lots of times and figure out which kind of things led to victories, and used that to, in a way, write its own program. In 1962, Arthur Samuel said one day, the vast majority of computer software would be written using this machine learning approach rather than written by hand.

- C-Path (Computational Pathologist) [45:42] is an example of traditional machine learning approach. He took pathology slides of breast cancer biopsies, consulted many pathologists on ideas about what kinds of patterns or features might be associated with long-term survival. Then they wrote specialist algorithms to calculate these features, run through logistic regression, and predicted the survival rate. It outperformed pathologists, but it took domain experts and computer experts many years of work to build.

A better way [47:35]



- A class of algorithm that have these three properties is Deep Learning.

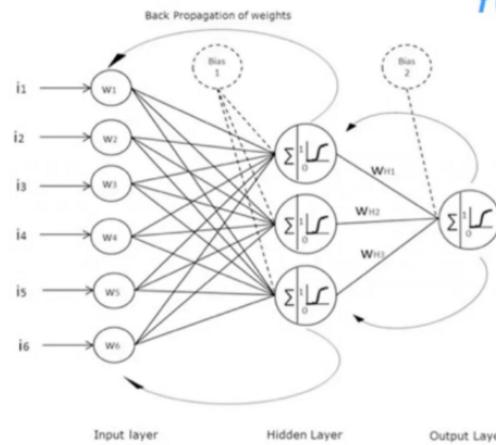
Infinitely flexible function: Neural Network [48:43]

Underlying function that deep learning uses is called the neural network:

Neural Network

$$f(X) = \text{nonlinear function composed of } \Sigma, \Pi, S$$

$$S(x) = 1 / (1 - e^x)$$

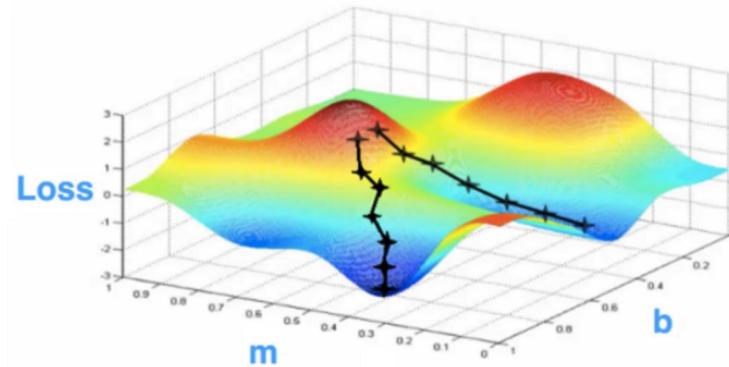


- All you need to know for now is that it consists of a number of simple linear layers interspersed with a number of simple non-linear layers. When you intersperse these layers, you get something called the universal approximation theorem. What universal approximation theorem says is that this kind of function can solve any given problem to arbitrarily close accuracy as long as you add enough parameters.

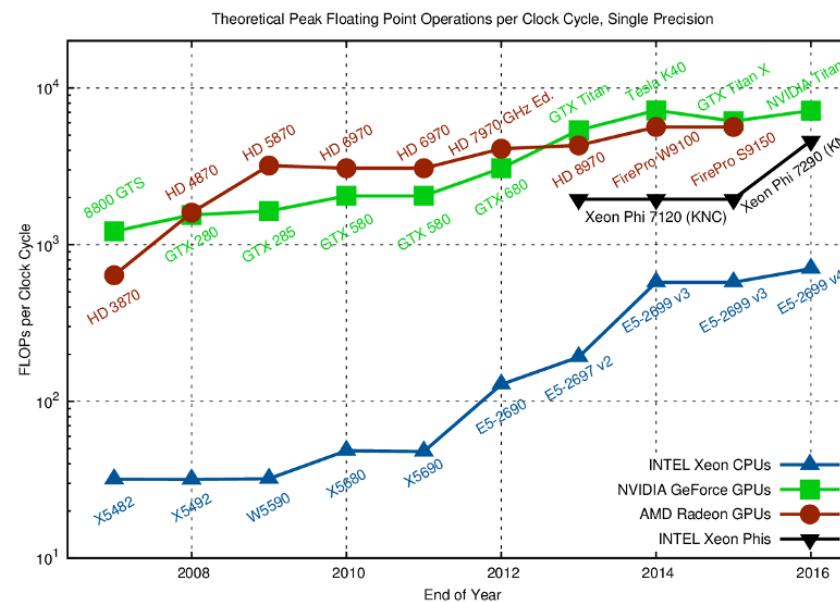
All purpose parameter fitting: Gradient Descent [49:39]

Gradient Descent

$$f(x) = \text{nonlinear function of } x$$



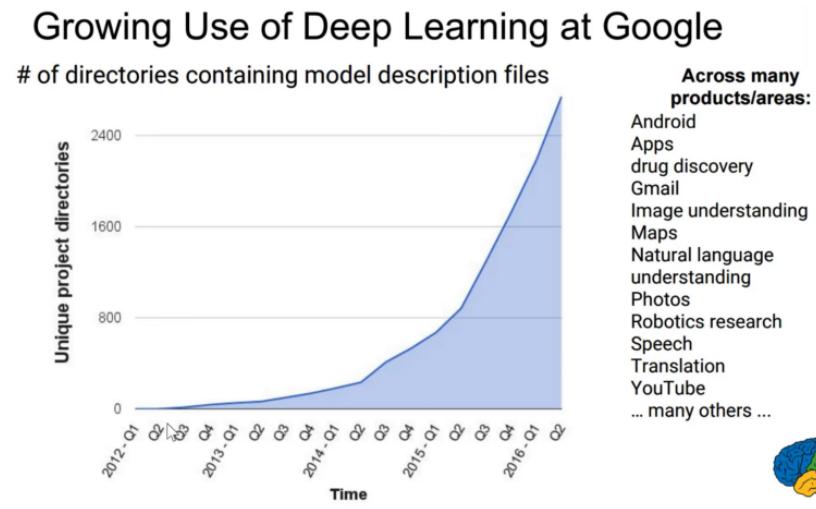
Fast and scalable: GPU [51:05]



The neural network example shown above has one hidden layer.

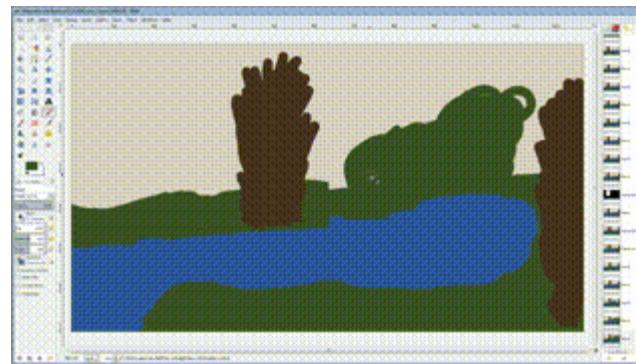
Something what we learned in the past few years is that these kind of neural network was not fast or scalable unless we added multiple hidden layers—hence called “Deep” learning.

Putting all together [53:40]



Here are some fo the examples:

- <https://research.googleblog.com/2015/11/computer-respond-to-this-email.html>
- <https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/>
- <https://www.skype.com/en/features/skype-translator/>
- <https://arxiv.org/abs/1603.01768>



Diagnosing lung cancer [56:55]

	False Positive Rate	False Negative Rate
Panel of 4 Human Radiologists	66.3%	7.0%
Enlitic Algorithm	47.5%	0.0%

CNN Money Could this computer save your life? By Jillian Eugenios | @jillianeugenios

Meet the computer diagnosing cancer.

Cancer is good at hiding. It's so good that sometimes sick patients are sent home with a clean bill of health. And screenings don't always help. A 2013 study by Oxford University found "no evidence" that screening programs are responsible for the decline in breast cancer, and a study by the Huntsman Cancer Institute last year found that colon cancer is missed in about 6% of colonoscopies.

A company is looking to change that margin of error by bringing a super-smart computer into the examination room.

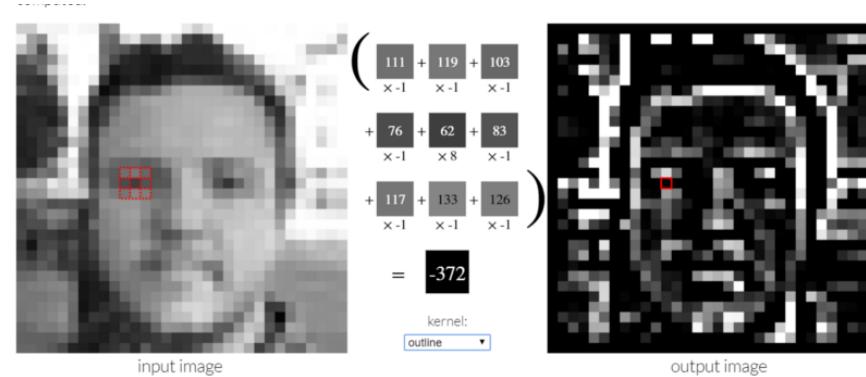
Other current applications:



Convolutional Neural Network [59:13]

Linear Layer

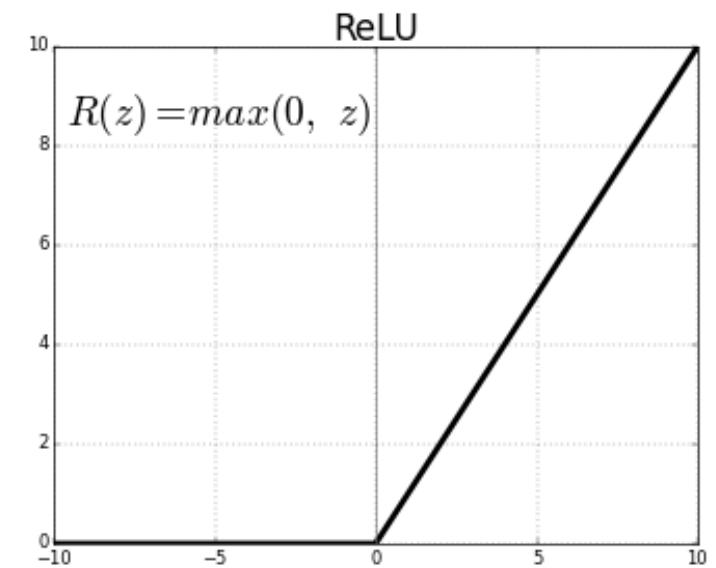
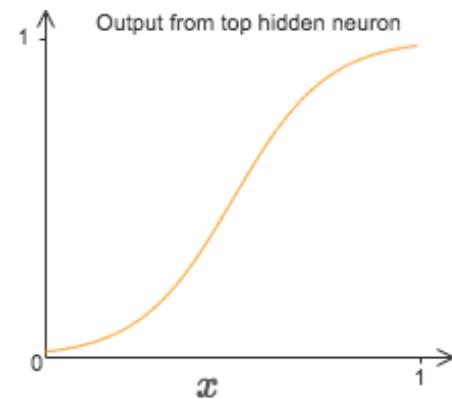
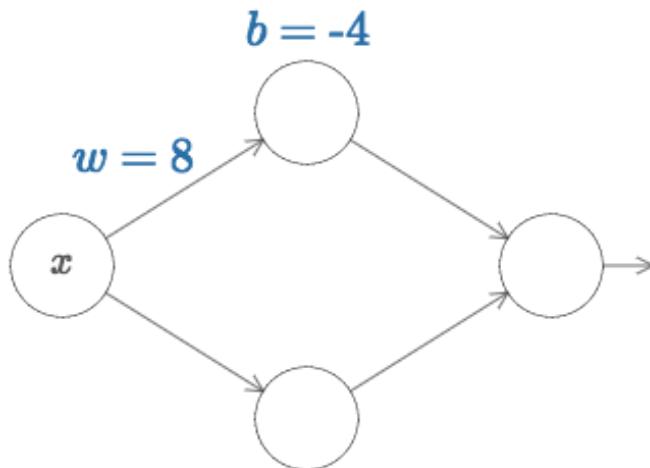
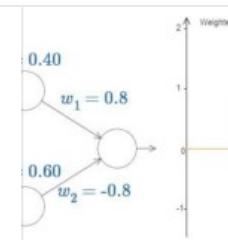
<http://setosa.io/ev/image-kernels/>



Nonlinear Layer [01:02:12]

Neural networks and deep learning

In this chapter I give a simple and mostly visual explanation of the universality theorem. We'll go...
neuralnetworksanddeeplearning.com



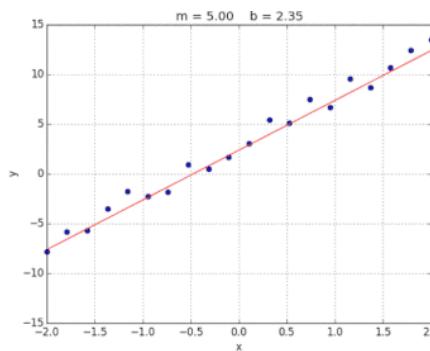
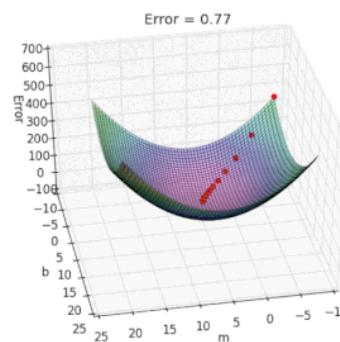
Sigmoid and ReLU

- A combination of linear layer followed by an element-wise nonlinear function allows us to create arbitrarily complex shapes —this is the essence of the universal approximation theorem.

How to set these parameters to solve problems

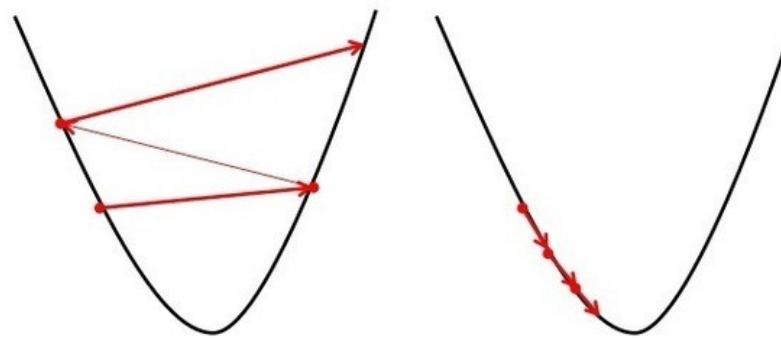
[01:04:25]

- Stochastic Gradient Descent—we take small steps down the hill.
The step size is called **learning rate**



Gradient Descent

Big learning rate

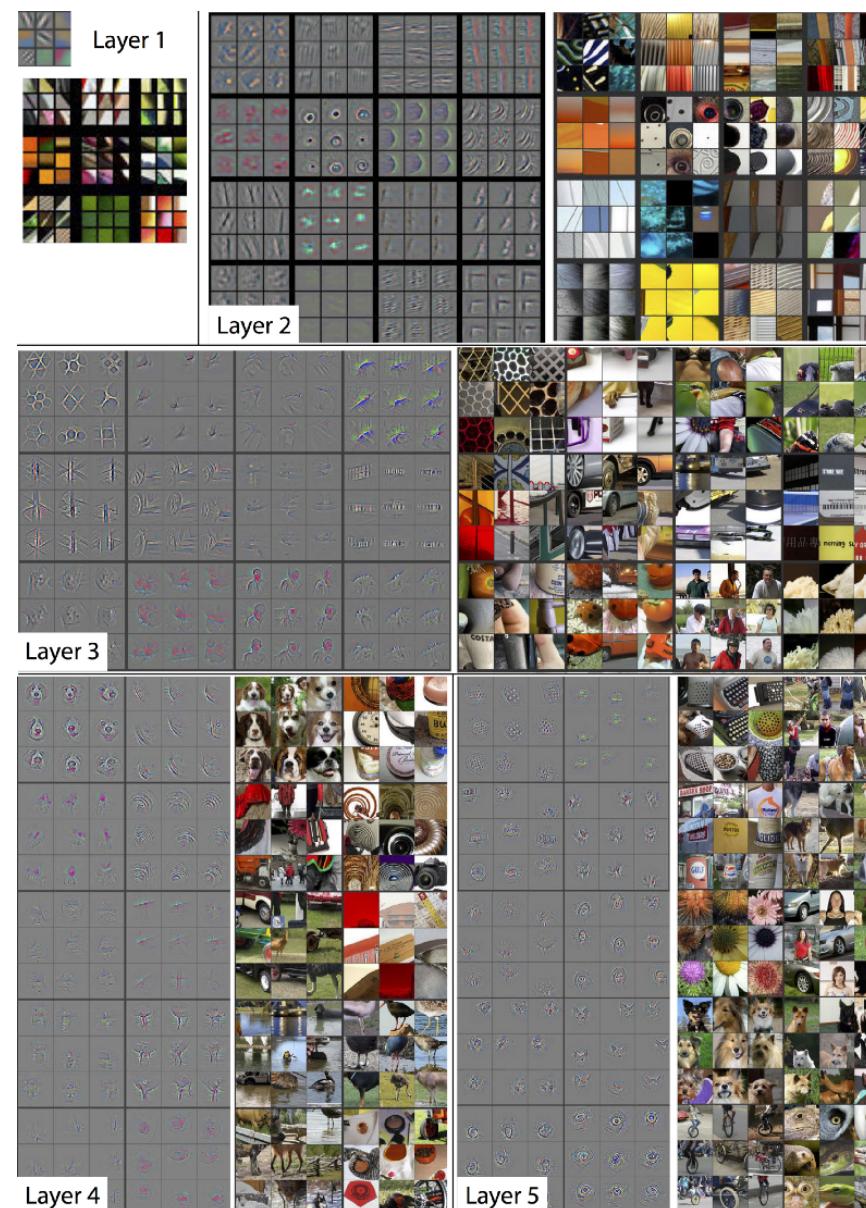


Small learning rate

- If learning rate is too large, it will diverge instead of converge

- If learning rate is too small, it will take forever

Visualizing and Understanding Convolutional Networks [01:08:27]



We started with something incredibly simple but if we use it as a big enough scale, thanks to the universal approximation theorem and the use of multiple hidden layers in deep learning, we actually get the very very rich capabilities. This is actually what we used when we used when we trained our dog vs cat recognizer.

Dog vs. Cat Revisited—Choosing a learning rate [01:11:41]

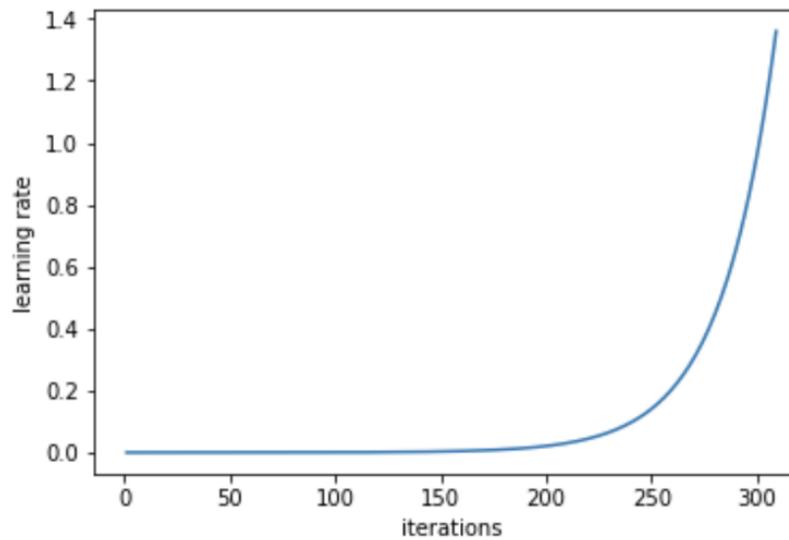
```
learn.fit(0.01, 3)
```

- The first number `0.01` is the learning rate.
- The *learning rate* determines how quickly or how slowly you want to update the *weights* (or *parameters*). Learning rate is one of the most difficult parameters to set, because it significantly affect model performance.
- The method `learn.lr_find()` helps you find an optimal learning rate. It uses the technique developed in the 2015 paper [Cyclical Learning Rates for Training Neural Networks](#), where we simply keep increasing the learning rate from a very small value, until the loss starts decreasing. We can plot the learning rate across batches to see what this looks like.

```
learn = ConvLearner.pretrained(arch, data, precompute=True)
learn.lr_find()
```

Our `learn` object contains an attribute `sched` that contains our learning rate scheduler, and has some convenient plotting functionality including this one:

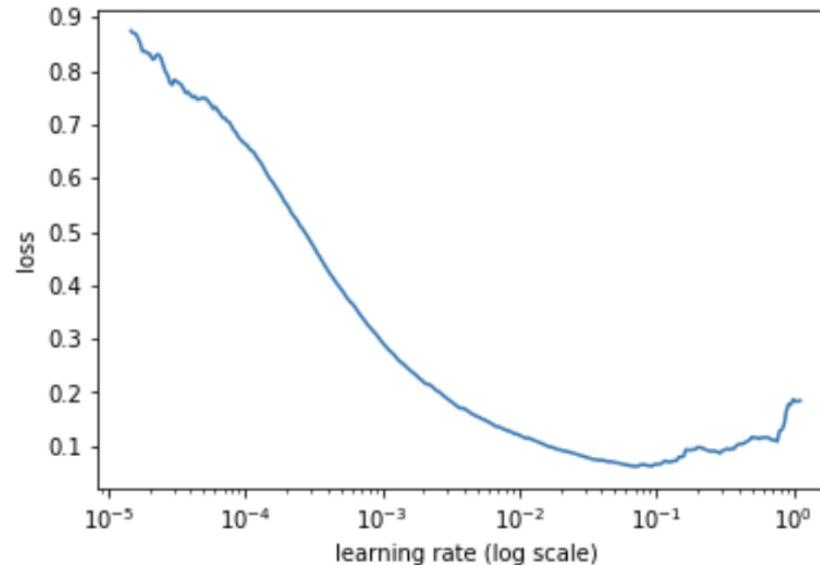
```
learn.sched.plot_lr()
```



- Jeremy is currently experimenting with increasing the learning rate exponentially vs. linearly.

We can see the plot of loss versus learning rate to see where our loss stops decreasing:

```
learn.sched.plot()
```



- We then pick the learning rate where the loss is still clearly improving—in this case `1e-2` (0.01)

Choosing number of epochs [1:18:49]

```
[ 0.      0.04955  0.02605  0.98975]
[ 1.      0.03977  0.02916  0.99219]
[ 2.      0.03372  0.02929  0.98975]
```

- As many as you would like, but accuracy might start getting worse if you run it for too long. It is something called “overfitting” and we will learn more about it later.
- Another consideration is the time available to you.

Tips and Tricks [1:21:40]

1. `Tab` —it will do an auto complete when you cannot remember the function name

The screenshot shows a Jupyter Notebook interface. In the top cell (In []), the user has typed `learn.p`. A dropdown menu appears, listing several methods from the `learn` module: `learn.precompute`, `learn.predict`, `learn.predict_array`, `learn.predict_dl`, `learn.predict_with_targs`, `learn.pretrained`, and `learn.precompute`. Below this, the notebook shows two more cells: In [16] and Out[16], and In [17] and Out[17]. The Out[16] cell contains the result of the previous command, which includes the method names listed in the dropdown. The Out[17] cell shows the continuation of the list.

```
In [ ]: learn.p
In [16]: learn.precompute
learn.predict
learn.predict_array
learn.predict_dl
learn.predict_with_targs
Out[16]: learn.pretrained
learn.precompute
In [17]: learn.predict
learn.predict_array
learn.predict_dl
Out[17]: [ -0.00150, -0.00046], 46], 85],
```

2. `Shift + Tab` —it will show you the arguments of a function

In []: `learn.predict()`

In [16]:

```
Signature: learn.predict(is_test=False)
Docstring: <no docstring>
File:      ~/fastai/courses/dl1/fastai/learner.py
Type:      method
```

Out[16]:

3. Shift + Tab + Tab —it will bring up a documentation (i.e. docstring)

In []: `np.exp()`

In [16]:

```
Call signature: np.exp(*args, **kwargs)
Type:          ufunc
String form:   <ufunc 'exp'>
File:          ~/anaconda3/envs/fastai/lib/python3.6/site-packages/numpy/__init__.py
Docstring:
```

Out[16]:

```
exp(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj])
```

In [17]:

Out[17]:

Calculate the exponential of all elements in the input array.

4. Shift + Tab + Tab + Tab —it will open a separate window with the same information.

In [15]: # from here we know that 'cats' is label 0 and 'dogs' is label 1.
`data.classes`

Out[15]: ['cats', 'dogs']

a shape that the inputs broadcast to. If not provided or 'None', a freshly-allocated array is returned. A tuple (possibly only as a keyword argument) must have length equal to the number of outputs.

where : array_like, optional
Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone.

**kwargs
For other keyword-only arguments, see the :ref:`ufunc docs <ufuncs.kwargs>`.

Returns

out : ndarray
Output array, element-wise exponential of `'x'`.

Typing `?` followed by a function name in a cell and running it will do the same as `shift + tab (3 times)`

```
In [31]: ?np.exp
```

```
In [16]: # this gives prediction for validation set.
```

5. Typing two question mark will display the source code

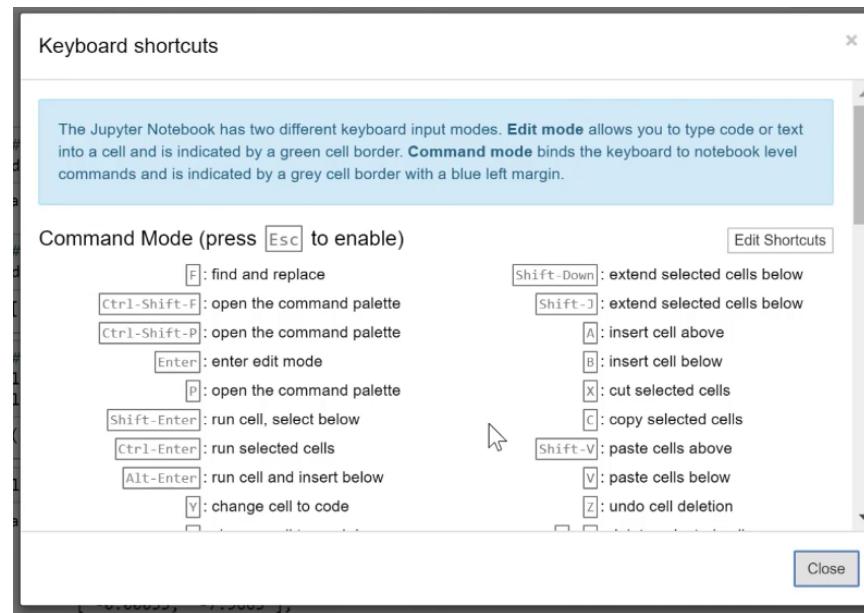
```
In [32]: ??learn.predict
```

```
In [16]: # this gives prediction for validation set. Predictions are in log scale
log_preds = learn.predict()
log_preds.shape
```

```
Out[16]: (2000, 2)
```

```
Signature: learn.predict(is_test=False)
Source:     def predict(self, is_test=False): return self.predict_with_targs(is_test)[@]
File:      ~/fastai/courses/dl1/fastai/learner.py
Type:     method
```

6. Typing `H` in Jupyter Notebook will open up a window with keyboard shortcuts. Try learning 4 or 5 shortcuts a day



7. Stop Paperspace, Crestle, AWS—otherwise you'll be charged \$\$

8. Please remember about the forum and <http://course.fast.ai/> (for each lesson) for up-to-date information.



Hiromi Suenaga [Follow](#)
Jan 14 · 22 min read

Deep Learning 2: Part 1 Lesson 2

My personal notes from fast.ai course. These notes will continue to be updated and improved as I continue to review the course to “really” understand it. Much appreciation to Jeremy and Rachel who gave me this opportunity to learn.

• • •

Lesson 2

Notebook

Review of last lesson [01:02]

- We used 3 lines of code to build an image classifier.
- In order to train the model, data needs to be organized in a certain way under `PATH` (in this case `data/dogscats/`):

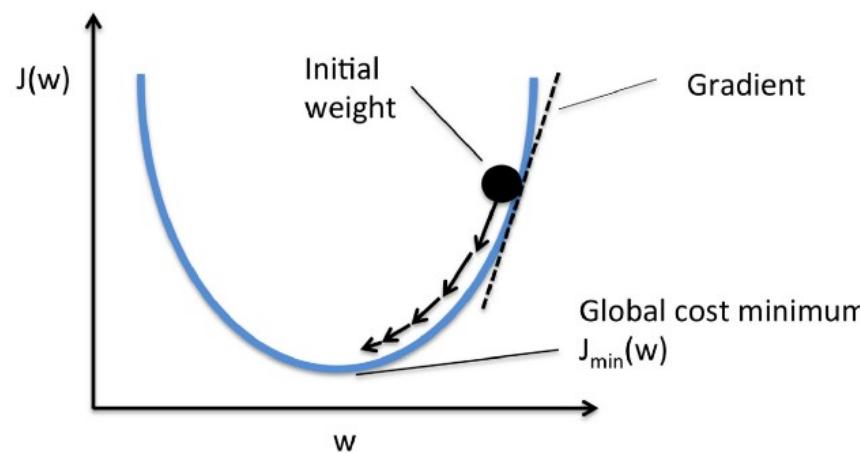
```
data/dogscats/
├── sample
│   ├── models
│   ├── tmp
│   ├── train
│   │   ├── cats
│   │   └── dogs
│   └── valid
│       ├── cats
│       └── dogs
└── test1
    ├── train
    │   ├── cats
    │   └── dogs
    └── valid
        ├── cats
        └── dogs
: Deep Learning v2
```

- There should be `train` folder and `valid` folder, and under each of these, folders with classification labels (i.e. `cats` and `dogs` for this example) with corresponding images in them.
- The training output: $[\text{epoch \#}, \text{training loss}, \text{validation loss}, \text{accuracy}]$

```
[ 0.          0.04955  0.02605  0.98975]
```

Learning Rate [4:54]

- The basic idea of learning rate is that it is going to decide how quickly we zoom/hone in on the solution.



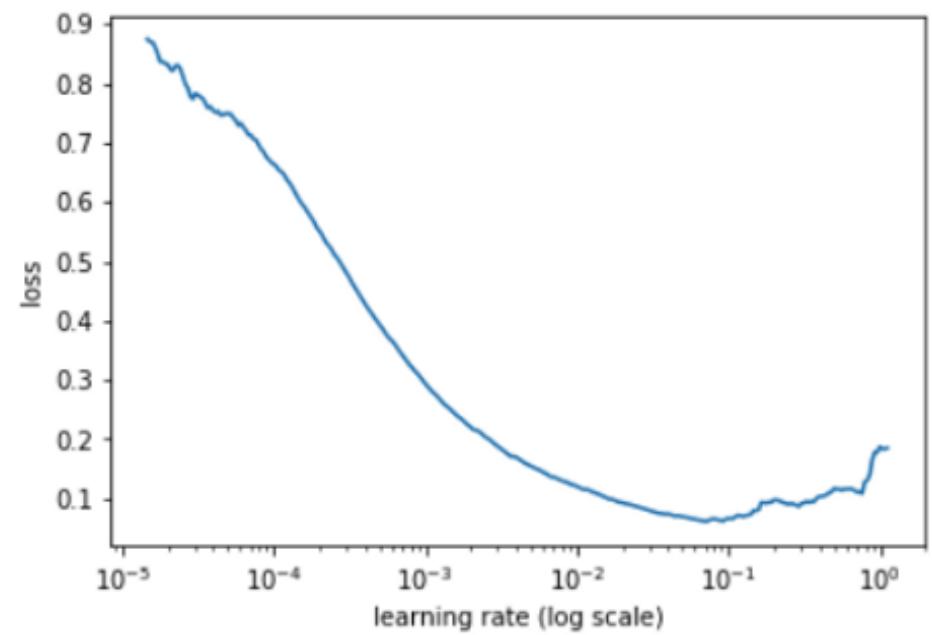
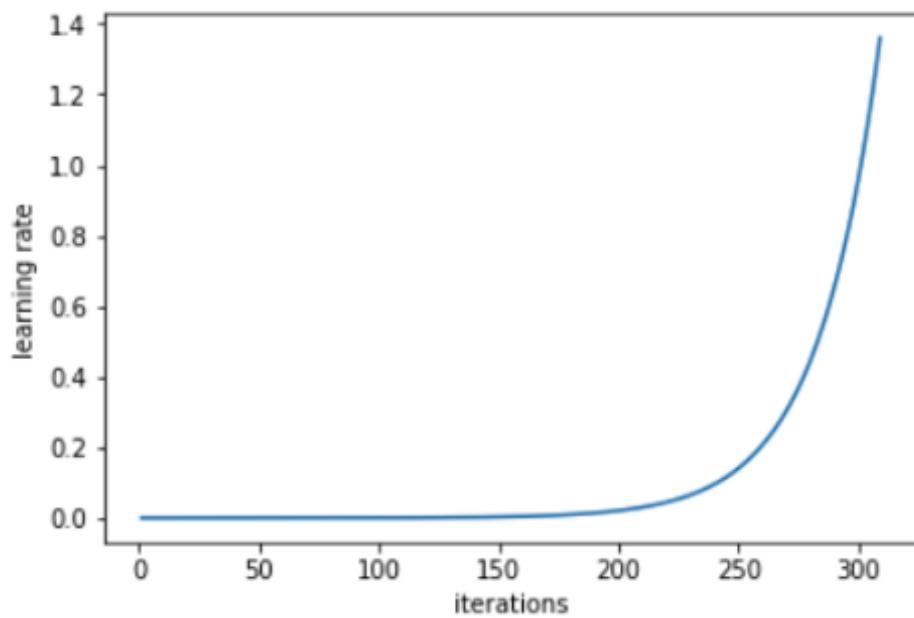
- If the learning rate is too small, it will take very long time to get to the bottom
- If the learning rate is too big, it could get oscillate away from the bottom.
- Learning rate finder (`learn.lr_find`) will increase the learning rate after each mini-batch. Eventually, the learning rate is too high that loss will get worse. We then look at the plot of learning rate against loss, and determine the lowest point and go back by one magnitude and choose that as a learning rate (`1e-2` in the example below).
- Mini-batch is a set of few images we look at each time so that we are using the parallel processing power of the GPU effectively

(generally 64 or 128 images at a time)

- In Python:

$$10^{-1} = 0.1 = 1e-1$$

$$10^{-2} = 0.01 = 1e-2$$

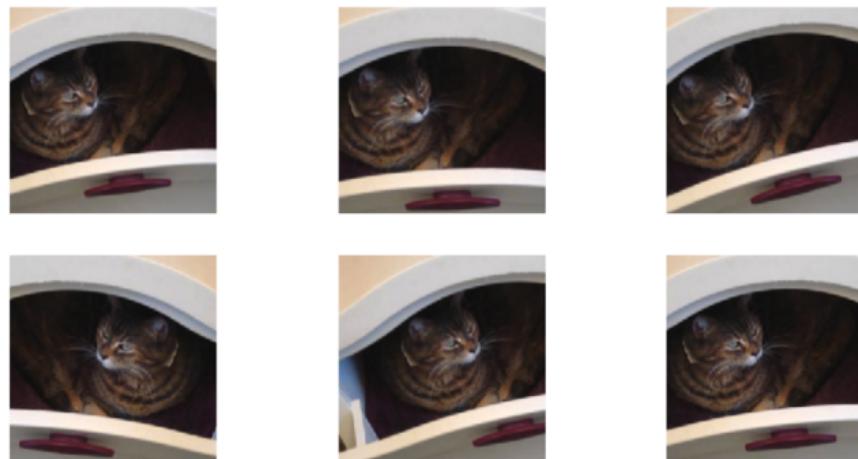


- By adjusting this one number, you should be able to get pretty good results. fast.ai library picks the rest of the hyper parameters for you. But as the course progresses, we will learn that there are

some more things we can tweak to get slightly better results. But learning rate is the key number for us to set.

- Learning rate finder sits on top of other optimizers (e.g. momentum, Adam, etc) and help you choose the best learning rate given what other tweaks you are using (such as advanced optimizers but not limited to optimizers).
- Questions: what happens for optimizers that changes learning rate during the epoch? Is this finder choosing an initial learning rate? [14:05] We will learn about optimizers in details later, but the basic answer is no. Even Adam has a learning rate which gets divided by the average previous gradient and also the recent sum of squared gradients. Even those so-called “dynamic learning rate” methods have a learning rate.
- The most important thing you can do to make the model better is to give it more data. Since these models have millions of parameters, if you train them for a while, they start to do what is called “overfitting”.
- Overfitting—the model is starting to see the specific details of the images in the training set rather than learning something general that can be transferred across to the validation set.
- We can collect more data, but another easy way is data augmentation.

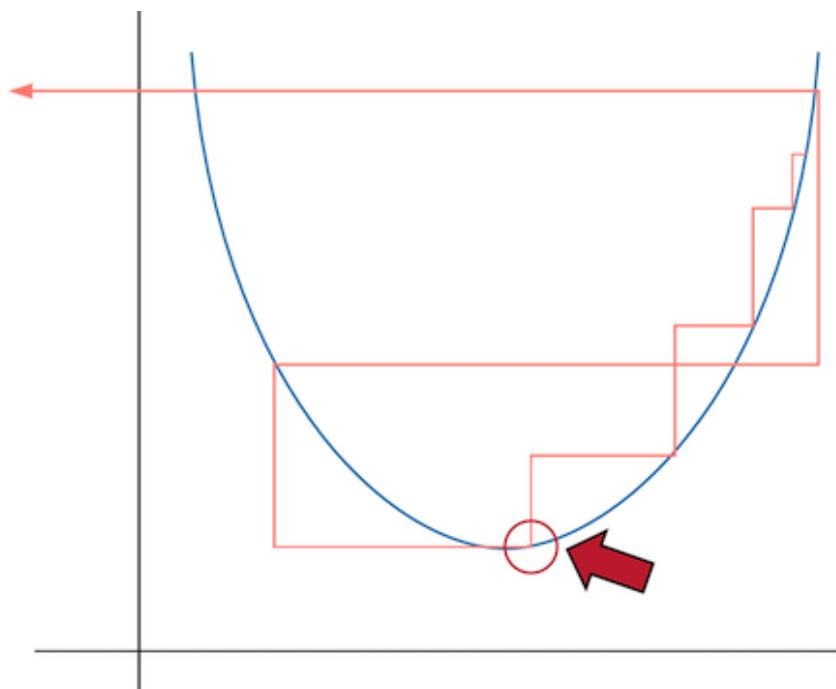
Data Augmentation [15:50]



- Every epoch, we will randomly change the image a little bit. In other words, the model is going to see slightly different version of the image each epoch.
- You want to use different types of data augmentation for different types of image (flip horizontally, vertically, zoom in, zoom out, vary contrast and brightness, and many more).

Learning Rate Finder Questions [19:11]:

- Why not pick the bottom? The point at which the loss was lowest is where the red circle is. But that learning rate was actually too large at that point and will not likely to converge. So the one before that would be a better choice (it is always better to pick a learning rate that is smaller than too big)



- When should we learn `lr_find`? [23:02] Run it once at the start, and maybe after unfreezing layers (we will learn it later). Also when I change the thing I am training or change the way I am training it. Never any harm in running it.

Back to Data Augmentation [24:10]

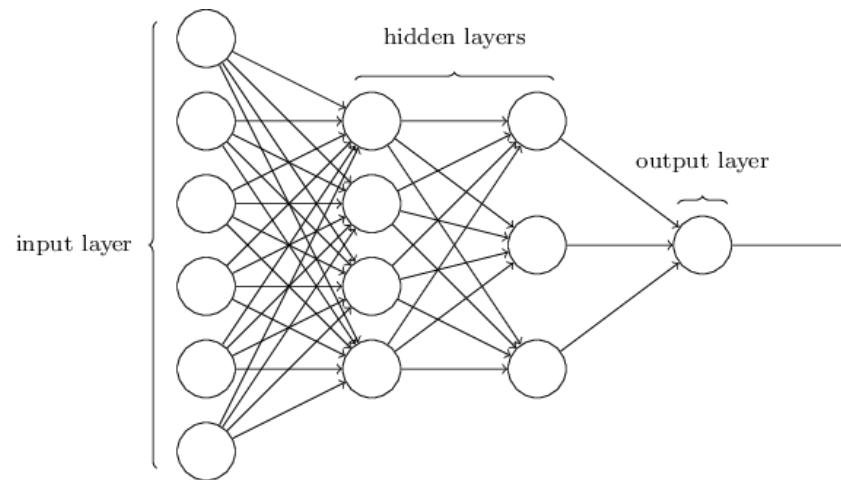
```
tfms = tfms_from_model(resnet34, sz,  
aug_tfms=transforms_side_on, max_zoom=1.1)
```

- `transform_side_on` —a predefined set of transformations for side-on photos (there is also `transform_top_down`). Later we will learn how to create custom transform lists.
- It is not exactly creating new data, but allows the convolutional neural net to learn how to recognize cats or dogs from somewhat different angles.

```
data = ImageClassifierData.from_paths(PATH, tfms=tfms)
learn = ConvLearner.pretrained(arch, data, precompute=True)

learn.fit(1e-2, 1)
```

- Now we created a new `data` object that includes augmentation. Initially, the augmentations actually do nothing because of `precompute=True`.
- Convolutional neural network have these things called “activations.” An activation is a number that says “this feature is in this place with this level of confidence (probability)”. We are using a pre-trained network which has already learned to recognize features (i.e. we do not want to change hyper parameters it learned), so what we can do is to pre-compute activations for hidden layers and just train the final linear portion.



- This is why when you train your model for the first time, it takes longer—it is pre-computing these activations.
- Even though we are trying to show a different version of the cat each time, we had already pre-computed the activations for a particular version of the cat (i.e. we are not re-calculating the activations with the altered version).
- To use data augmentation, we have to do

```
learn.precompute=False :
```

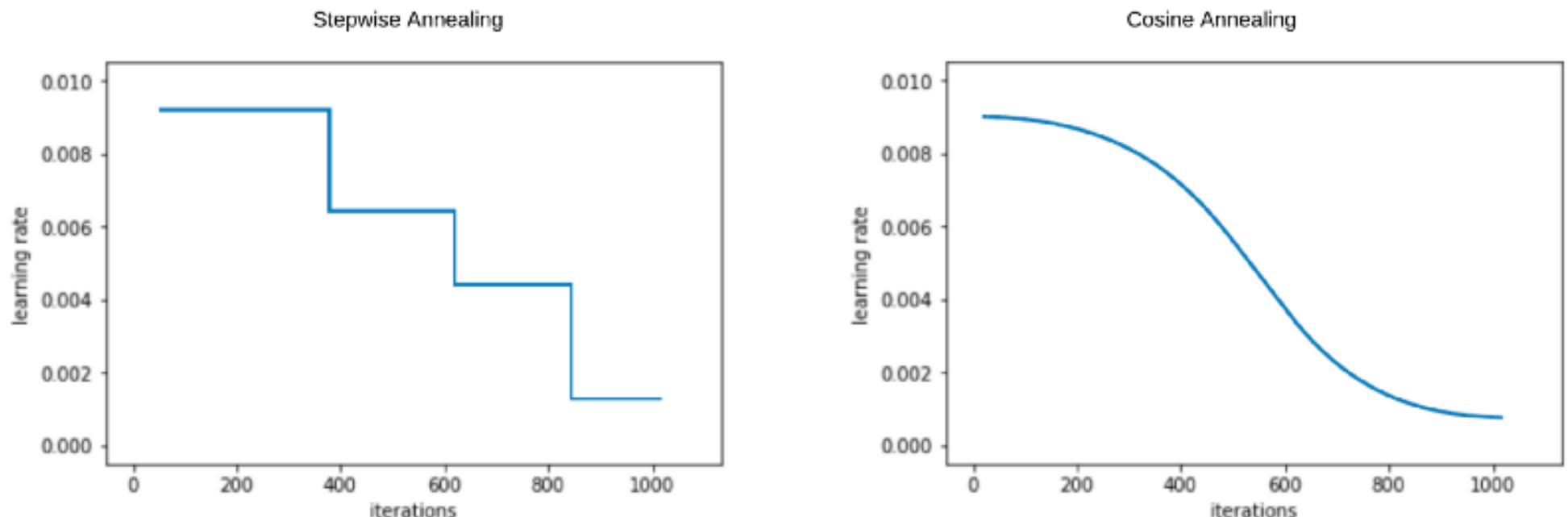
```
learn.precompute=False

learn.fit(1e-2, 3, cycle_len=1)

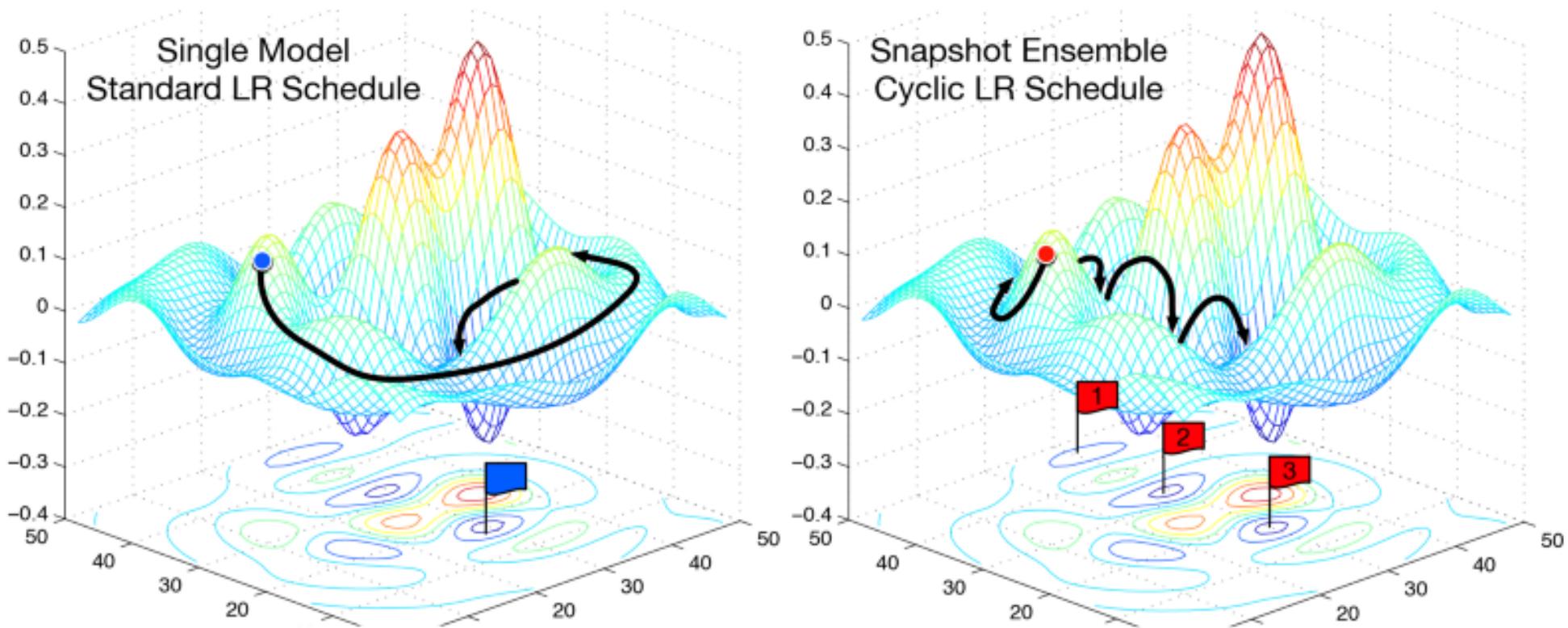
[ 0.          0.03597  0.01879  0.99365]
[ 1.          0.02605  0.01836  0.99365]
[ 2.          0.02189  0.0196   0.99316]
```



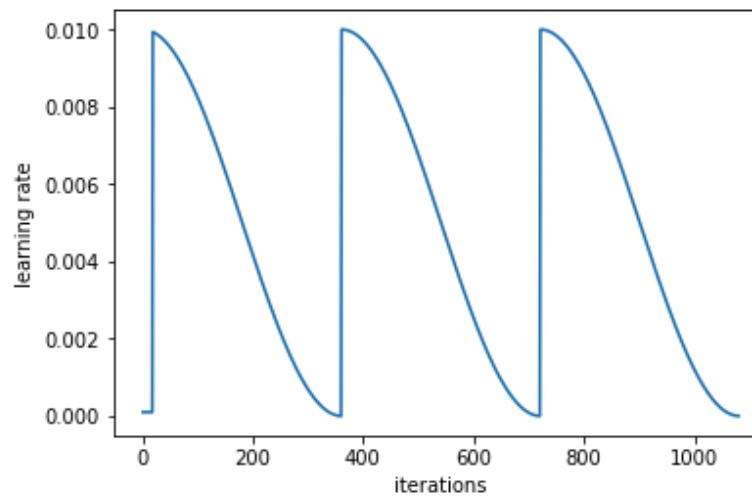
- Bad news is that accuracy is not improving. Training loss is decreasing but validation loss is not, but we are not overfitting. Overfitting when the training loss is much lower than the validation loss. In other words, when your model is doing much better job on the training set than it is on the validation set, that means your model is not generalizing.
- `cycle_len=1 [30:17]`: This enables **stochastic gradient descent with restarts (SGDR)**. The basic idea is as you get closer and closer to the spot with the minimal loss, you may want to start decrease the learning rate (taking smaller steps) in order to get to exactly the right spot.
- The idea of decreasing the learning rate as you train is called **learning rate annealing** which is very common. Most common and “hacky” way to do this is to train a model with a certain learning rate for a while, and when it stops improving, manually drop down the learning rate (stepwise annealing).
- A better approach is simply to pick some kind of functional form—turns out the really good functional form is one half of the cosign curve which maintains the high learning rate for a while at the beginning, then drop quickly when you get closer.



- However, we may find ourselves in a part of the weight space that isn't very resilient—that is, small changes to the weights may result in big changes to the loss. We want to encourage our model to find parts of the weight space that are both accurate and stable. Therefore, from time to time we increase the learning rate (this is the ‘restarts’ in ‘SGDR’), which will force the model to jump to a different part of the weight space if the current area is “spiky”. Here's a picture of how that might look if we reset the learning rates 3 times (in this paper they call it a “cyclic LR schedule”):



- The number of epochs between resetting the learning rate is set by `cycle_len`, and the number of times this happens is referred to as the *number of cycles*, and is what we're actually passing as the 2nd parameter to `fit()`. So here's what our actual learning rates looked like:



- Question: Could we get the same effect by using random starting point? [35:40] Before SGDR was created, people used to create “ensembles” where they would relearn a whole new model ten times in the hope that one of them would end up being better. In SGDR, once we get close enough to the optimal and stable area, resetting will not actually “reset” but the weights keeps better. So SGDR will give you better results than just randomly try a few different starting points.
- It is important to pick a learning rate (which is the highest learning rate SGDR uses) that is big enough to allow the reset to jump to a different part of the function. [37:25]
- SGDR reduces the learning rate every mini-batch, and reset occurs every `cycle_len` epoch (in this case it is set to 1).
- Question: Our main goal is to generalize and not end up in the narrow optima. In this method, are we keeping track of the

minima and averaging them and ensembling them? [39:27] That is another level of sophistication and you see “Snapshot Ensemble” in the diagram. We are not currently doing that but if you wanted it to generalize even better, you can save the weights right before the resets and take the average. But for now, we are just going to pick the last one.

- If you want to skip ahead, there is a parameter called `cycle_save_name` which you can add as well as `cycle_len`, which will save a set of weights at the end of every learning rate cycle and then you can ensemble them [40:14].

Saving model [40:31]

```
learn.save('224_lastlayer')

learn.load('224_lastlayer')
```

- When you precompute activations or create resized images (we will learn about it soon), various temporary files get created which you see under `data/dogcats/tmp` folder. If you are getting weird errors, it might be because of precomputed activations that are only half completed or are in some way incompatible with what you are doing. So you can always go ahead and delete this `/tmp` folder to see if it makes the error go away (fast.ai equivalent of turning it off and then on again).

- You will also see there is a directory called `/models` that is where models get saved when you say `learn.save`

```
jhoward@usf2:/data1/jhoward/git/fastai/courses/dl1/data/dogscats$ ls tmp
x_act_dn121_0_224.bc          x_act_test_inception_4_0_224.bc      x_act_val_inception_4_0_224.bc
x_act_inception_4_0_224.bc      x_act_test_inceptionresnet_2_0_224.bc  x_act_val_inceptionresnet_2_0_224.bc
x_act_inceptionresnet_2_0_224.bc x_act_test_resnet34_0_224.bc      x_act_val_resnet34_0_224.bc
x_act_resnet34_0_224.bc         x_act_test_resnet34_0_299.bc      x_act_val_resnet34_0_299.bc
x_act_resnet34_0_299.bc         x_act_test_resnext101_0_224.bc    x_act_val_resnext101_0_224.bc
x_act_resnext101_0_224.bc       x_act_test_resnext101_0_299.bc    x_act_val_resnext101_0_299.bc
x_act_resnext101_0_299.bc       x_act_test_resnext101_64_0_224.bc  x_act_val_resnext101_64_0_224.bc
x_act_resnext101_64_0_224.bc    x_act_test_resnext50_0_224.bc   x_act_val_resnext50_0_224.bc
x_act_resnext50_0_224.bc        x_act_test_resnext50_0_299.bc   x_act_val_resnext50_0_299.bc
x_act_resnext50_0_299.bc        x_act_test_wrn_0_224.bc       x_act_val_wrn_0_224.bc
x_act_test_dn121_0_224.bc       x_act_val_dn121_0_224.bc     x_act_wrn_0_224.bc
jhoward@usf2:/data1/jhoward/git/fastai/courses/dl1/data/dogscats$ ls models
224_all_101.h5  224_all.h5      224_lastlayer_50.h5  299_fc.h5  tmp.h5
224_all_50.h5   224_lastlayer_101.h5  224_lastlayer.h5  fc.h5
jhoward@usf2:/data1/jhoward/git/fastai/courses/dl1/data/dogscats$ [0] 0:bash*MZ "usf2"
```

Fine Tuning and Differential Learning Rate [43:49]

- So far, we have not retrained any of pre-trained features—specifically, any of those weights in the convolutional kernels. All we have done is we added some new layers on top and learned how to mix and match pre-trained features.
- Images like satellite images, CT scans, etc have totally different kinds of features all together (compare to ImageNet images), so you want to re-train many layers.

- For dogs and cats, images are similar to what the model was pre-trained with, but we still may find it is helpful to slightly tune some of the later layers.
- Here is how you tell the learner that we want to start actually changing the convolutional filters themselves:

```
learn.unfreeze()
```

- “frozen” layer is a layer which is not being trained/updated.
`unfreeze` unfreezes all the layers.
- Earlier layers like the first layer (which detects diagonal edges or gradient) or the second layer (which recognizes corners or curves) probably do not need to change by much, if at all.
- Later layers are much more likely to need more learning. So we create an array of learning rates (differential learning rate):

```
lr=np.array([1e-4,1e-3,1e-2])
```

- `1e-4` : for the first few layers (basic geometric features)
- `1e-3` : for the middle layers (sophisticated convolutional features)
- `1e-2` : for layers we added on top

- Why 3? Actually they are 3 ResNet blocks but for now, think of it as a group of layers.

Question: What if I have a bigger images than the model is trained with? [50:30] The short answer is, with this library and modern architectures we are using, we can use any size we like.

Question: Can we unfreeze just specific layers? [51:03] We are not doing it yet, but if you wanted, you can do `learn.unfreeze_to(n)` (which will unfreeze layers from layer `n` onwards). Jeremy almost never finds it helpful and he thinks it is because we are using differential learning rates, and the optimizer can learn just as much as it needs to. The one place he found it helpful is if he is using a really big memory intensive model and he is running out of GPU, the less layers you unfreeze, the less memory and time it takes.

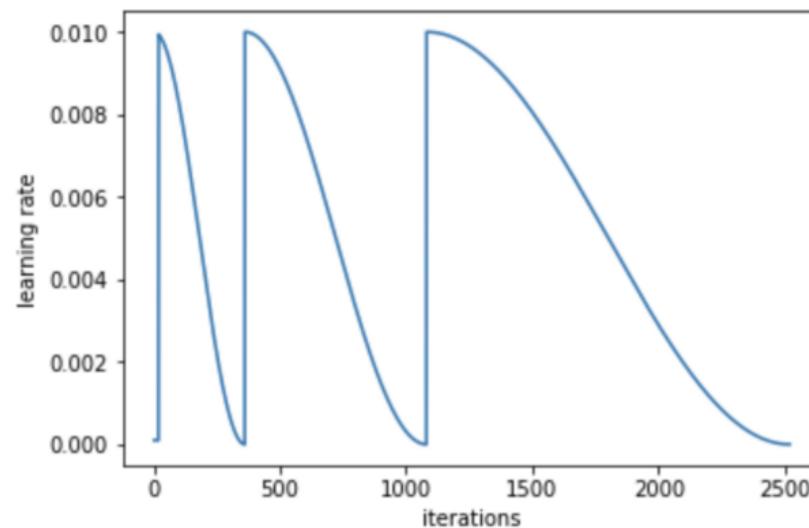
Using differential learning rate, we are up to 99.5%! [52:28]

```
learn.fit(lr, 3, cycle_len=1, cycle_mult=2)

[ 0.      0.04538  0.01965  0.99268]
[ 1.      0.03385  0.01807  0.99268]
[ 2.      0.03194  0.01714  0.99316]
[ 3.      0.0358   0.0166   0.99463]
[ 4.      0.02157  0.01504  0.99463]
[ 5.      0.0196   0.0151   0.99512]
[ 6.      0.01356  0.01518  0.9956 ]
```



- Earlier we said `3` is the number of epochs, but it is actually **cycles**. So if `cycle_len=2`, it will do 3 cycles where each cycle is 2 epochs (i.e. 6 epochs). Then why did it 7? It is because of `cycle_mult`
- `cycle_mult=2` : this multiplies the length of the cycle after each cycle (1 epoch + 2 epochs + 4 epochs = 7 epochs).

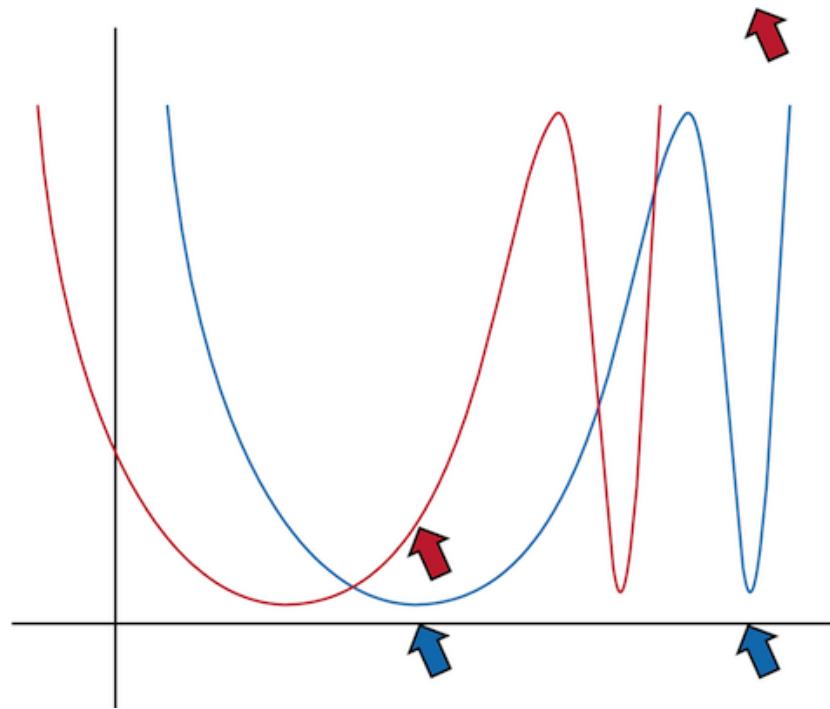


Intuitively speaking [53:57], if the cycle length is too short, it starts going down to find a good spot, then pops out, and goes down trying to find a good spot and pops out, and never actually get to find a good spot. Earlier on, you want it to do that because it is trying to find a spot that is smoother, but later on, you want it to do more exploring. That is why `cycle_mult=2` seems to be a good approach.

We are introducing more and more hyper parameters having told you that there are not many. You can get away with just choosing a good learning rate, but then adding these extra tweaks helps get that extra level-up without any effort. In general, good starting points are:

- `n_cycle=3, cycle_len=1, cycle_mult=2`
- `n_cycle=3, cycle_len=2 (no cycle_mult)`

Question: why do smoother surfaces correlate to more generalized networks? [55:28]



Say you have something spiky (blue line). X-axis is showing how good this is at recognizing dogs vs. cats as you change this particular parameter. Something to be generalizable means that we want it to work when we give it a slightly different dataset. Slightly different dataset may have a slightly different relationship between this parameter and how cat-like vs. dog-like it is. It may, instead look like the red line. In other words, if we end up at the blue pointy part, then it will not go to do a good job on this slightly different dataset. Or else, if we end up on the wider blue part, it will still do a good job on the red dataset.

- Here is some interesting discussion about spiky minima.

Test Time Augmentation (TTA) [56:49]

Our model has achieved 99.5%. But can we make it better still? Let's take a look at pictures we predicted incorrectly:

```
In [23]: plot_val_with_title(most_by_correct(1, False), "Most incorrect dogs")
```

Most incorrect dogs



Here, Jeremy printed out the whole of these pictures. When we do the validation set, all of our inputs to our model must be square. The reason is kind of a minor technical detail, but GPU does not go very quickly if you have different dimensions for different images. It needs to be consistent so that every part of the GPU can do the same thing. This may probably be fixable but for now that is the state of the technology we have.

To make it square, we just pick out the square in the middle—as you can see below, it is understandable why this picture was classified incorrectly:



We are going to do what is called “**Test Time Augmentation**”. What this means is that we are going to take 4 data augmentations at random as well as the un-augmented original (center-cropped). We will then calculate predictions for all these images, take the average, and make that our final prediction. Note that this is only for validation set and/or test set.

To do this, all you have to do is `learn.TTA()` —which brings up the accuracy to 99.65%!

```
log_preds,y = learn.TTA()
probs = np.mean(np.exp(log_preds),0)

accuracy(probs, y)

0.9965000000000005
```

Questions on augmentation approach[01:01:36]: Why not border or padding to make it square? Typically Jeremy does not do much padding, but instead he does a little bit of **zooming**. There is a thing called **reflection padding** that works well with satellite imagery. Generally speaking, using TTA plus data augmentation, the best thing to do is try to use as large image as possible. Also, having fixed crop locations plus random contrast, brightness, rotation changes might be better for TTA.

Question: Data augmentation for non-image dataset? [01:03:35] No one seems to know. It seems like it would be helpful, but there are very few number of examples. In natural language processing, people tried replacing synonyms for instance, but on the whole the area is under researched and under developed.

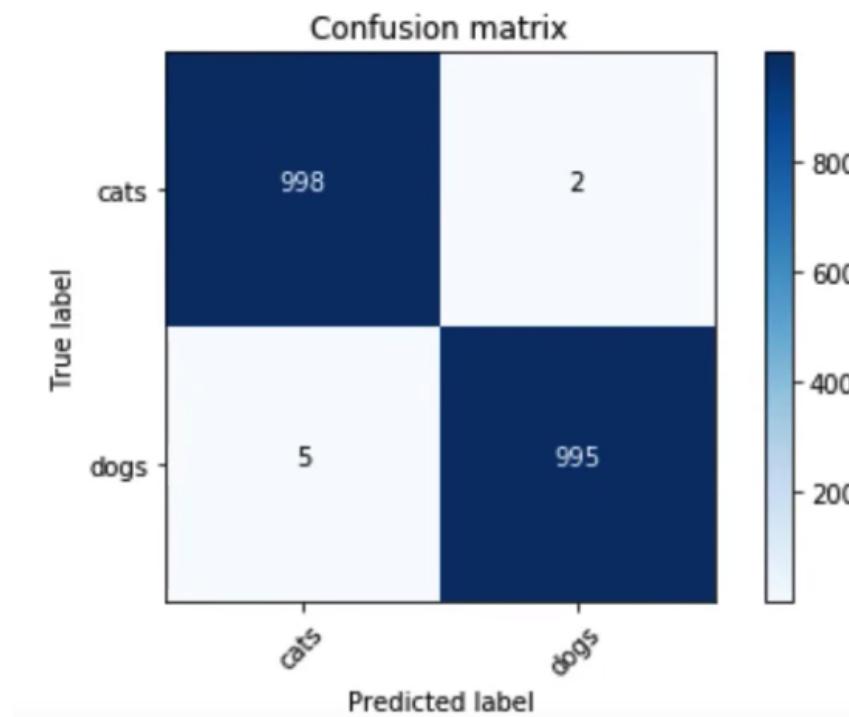
Question: Is fast.ai library open source?[01:05:34] Yes. He then covered the reason why Fast.ai switched from Keras + TensorFlow to PyTorch

Random note: PyTorch is much more than just a deep learning library. It actually lets us write arbitrary GPU accelerated algorithms from scratch—Pyro is a great example of what people are now doing with PyTorch outside of deep learning.

Analyzing results [01:11:50]

Confusion matrix

The simple way to look at the result of a classification is called confusion matrix—which is used not only for deep learning but in any kind of machine learning classifier. It is helpful particularly if there are four or five classes you are trying to predict to see which group you are having the most trouble with.



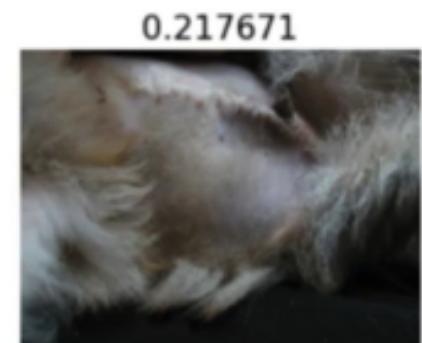
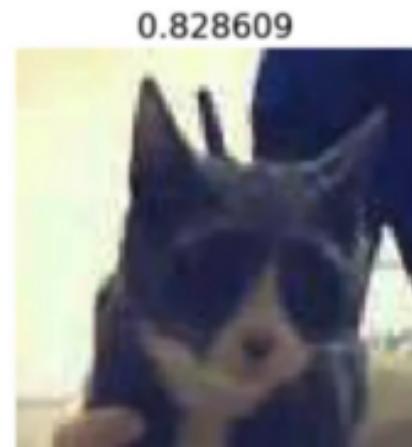
```
preds = np.argmax(probs, axis=1)
probs = probs[:,1]

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y, preds)

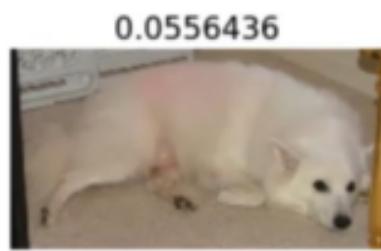
plot_confusion_matrix(cm, data.classes)
```

Let's look at the pictures again [01:13:00]

Most incorrect cats (only the left two were incorrect—it displays 4 by default):



Most incorrect dots:



**Review: easy steps to train a world-class
image classifier [01:14:09]**

1. Enable data augmentation, and `precompute=True`
2. Use `lr_find()` to find highest learning rate where loss is still clearly improving
3. Train last layer from precomputed activations for 1–2 epochs
4. Train last layer with data augmentation (i.e. `precompute=False`) for 2–3 epochs with `cycle_len=1`
5. Unfreeze all layers
6. Set earlier layers to 3x-10x lower learning rate than next higher layer. Rule of thumb: 10x for ImageNet like images, 3x for satellite or medical imaging
7. Use `lr_find()` again (Note: if you call `lr_find` having set differential learning rates, what it prints out is the learning rate of the last layers.)
8. Train full network with `cycle_mult=2` until over-fitting

Let's do it again: Dog Breed Challenge [01:16:37]

- You can use Kaggle CLI to download data for Kaggle competitions
- Notebook is not made public since it is an active competition

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

from fastai.imports import *
from fastai.transforms import *
```

```
from fastai.conv_learner import *
from fastai.model import *
from fastai.dataset import *
from fastai.sgdr import *
from fastai.plots import *

PATH = 'data/dogbreed/'
sz = 224
arch = resnext101_64
bs=16

label_csv = f'{PATH}labels.csv'
n = len(list(open(label_csv)))-1
val_idxs = get_cv_idxs(n)

!ls {PATH}
```

```
labels.csv      sample_submission.csv      test      train
labels.csv.zip  sample_submission.csv.zip  test.zip  train.zip
```

This is a little bit different to our previous dataset. Instead of `train` folder which has a separate folder for each breed of dog, it has a CSV file with the correct labels. We will read CSV file with Pandas. Pandas is what we use in Python to do structured data analysis like CSV and usually imported as `pd` :

```
label_df = pd.read_csv(label_csv)
label_df.head()
```

		id	breed
0	000bec180eb18c7604dcecc8fe0dba07		boston_bull
1	001513dfcb2ffafc82cccf4d8bbaba97		dingo
2	001cdf01b096e06d78e9e5112d419397		pekinese
3	00214f311d5d2247d5dfe4fe24b2303d		bluetick
4	0021f9ceb3235effd7fcde7f7538ed62		golden_retriever

```
label_df.pivot_table(index='breed',  
aggfunc=len).sort_values('id', ascending=False)
```

	id
	breed
scottish_deerhound	126
maltese_dog	117
afghan_hound	116
entlebucher	115
bernese_mountain_dog	114
shih-tzu	112
great_pyrenees	111
pomeranian	111
basenji	110
samoyed	109
airedale	107
tibetan_terrier	107
leonberg	106
cairn	106
beagle	105
japanese_spaniel	105
australian_terrier	102
blenheim_spaniel	102
miniature_pinscher	102

How many dog images per breed

```
tfms = tfms_from_model(arch, sz,
                      aug_tfms=transforms_side_on,
                      max_zoom=1.1)

data = ImageClassifierData.from_csv(PATH, 'train',
                                    f'{PATH}labels.csv', test_name='test',
                                    val_idxs=val_idxs, suffix='.jpg',
                                    tfms=tfms, bs=bs)
```

- `max_zoom` —we will zoom in up to 1.1 times
- `ImageClassifierData.from_csv` —last time, we used `from_paths` but since the labels are in CSV file, we will call `from_csv` instead.
- `test_name` —we need to specify where the test set is if you want to submit to Kaggle competitions
- `val_idx` —there is no `validation` folder but we still want to track how good our performance is locally. So above you will see:

`n = len(list(open(label_csv)))-1` : Open CSV file, create a list of rows, then take the length. `-1` because the first row is a header. Hence `n` is the number of images we have.

`val_idxs = get_cv_idxs(n)` : “get cross validation indexes”—this will return, by default, random 20% of the rows (indexes to be precise) to use as a validation set. You can also send `val_pct` to get different amount.

```
In [76]: n
```

```
Out[76]: 10222
```

```
In [75]: val_idxs
```

```
Out[75]: array([3694, 1573, 6281, ..., 5734, 5191, 5390])
```

- `suffix='jpg'` —File names has `.jpg` at the end, but CSV file does not. So we will set `suffix` so it knows the full file names.

```
fn = PATH + data.trn_ds.fnames[0]; fn  
  
'data/dogbreed/train/001513dfcb2ffa fc82cccf4d8bbaba97.jpg'
```

- You can access to training dataset by saying `data.trn_ds` and `trn_ds` contains a lot of things including file names (`fnames`)

```
img = PIL.Image.open(fn); img
```



```
img.size
```

```
(500, 375)
```

- Now we check image size. If they are huge, then you have to think really carefully about how to deal with them. If they are tiny, it is also challenging. Most of ImageNet models are trained on either 224 by 224 or 299 by 299 images

```
size_d = {k: PIL.Image.open(PATH+k).size for k in  
data.trn_ds.fnames}
```

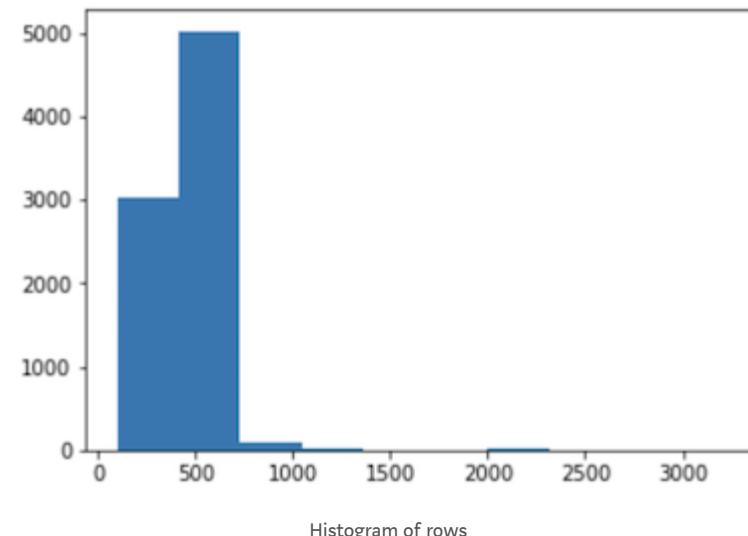
- Dictionary comprehension—

```
key: name of the file , value:  
size of the file
```

```
row_sz, col_sz = list(zip(*size_d.values()))
```

- `*size_d.values()` will unpack a list. `zip` will pair up elements of tuples to create a list of tuples.

```
plt.hist(row_sz);
```



- Matplotlib is something you want to be very familiar with if you do any kind of data science or machine learning in Python. Matplotlib is always referred to as `plt` .

Question: How many images should we use as a validation set?

[01:26:28] Using 20% is fine unless the dataset is small—then 20% is not enough. If you train the same model multiple times and you are getting very different validation set results, then your validation set is too small. If the validation set is smaller than a thousand, it is hard to interpret how well you are doing. If you care about the third decimal place of accuracy and you only have a thousand things in your validation set, a single image changes the accuracy. If you care about the difference between 0.01 and 0.02, you want that to represent 10 or 20 rows. Normally 20% seems to work fine.

```
def get_data(sz, bs):
    tfms = tfms_from_model(arch, sz,
                           aug_tfms=transforms_side_on,
                           max_zoom=1.1)
    data = ImageClassifierData.from_csv(PATH, 'train',
                                         f'{PATH}labels.csv', test_name='test',
                                         num_workers=4,
                                         val_idxs=val_idxs, suffix='.jpg', tfms=tfms,
                                         bs=bs)

    return data if sz>300 else data.resize(340, 'tmp')
```

- Here is the regular two lines of code. When we start working with new dataset, we want everything to go super fast. So we made it possible to specify the size and start with something like 64 which

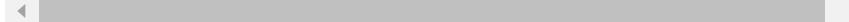
will run fast. Later, we will use bigger images and bigger architectures at which point, you may run out of GPU memory. If you see CUDA out of memory error, the first thing you need to do is to restart kernel (you cannot recover from it), then make the batch size smaller.

```
data = get_data(224, bs)

learn = ConvLearner.pretrained(arch, data, precompute=True)

learn.fit(1e-2, 5)

[0.      1.99245 1.0733  0.76178]
[1.      1.09107 0.7014   0.8181  ]
[2.      0.80813 0.60066  0.82148]
[3.      0.66967 0.55302  0.83125]
[4.      0.57405 0.52974  0.83564]
```



- 83% for 120 classes is pretty good.

```
learn.precompute = False

learn.fit(1e-2, 5, cycle_len=1)
```

- Reminder: a `epoch` is one pass through the data, a `cycle` is how many epochs you said is in a cycle

```
learn.save('224_pre')
learn.load('224_pre')
```

Increase image size [1:32:55]

```
learn.set_data(get_data(299, bs))
```

- If you trained a model on smaller size images, you can then call `learn.set_data` and pass in a larger size dataset. That is going to take your model, however it has been trained so far, and it is going to let you continue to train on larger images.

Starting training on small images for a few epochs, then switching to bigger images, and continuing training is an amazingly effective way to avoid overfitting.

```
learn.fit(1e-2, 3, cycle_len=1)

[0.      0.35614 0.22239 0.93018]
[1.      0.28341 0.2274   0.92627]
[2.      0.28341 0.2274   0.92627]
```

- As you see, validation set loss (0.2274) is much lower than training set loss (0.28341)—which means it is **under fitting**. When you are under fitting, it means `cycle_len=1` is too short (learning rate is getting reset before it had the chance to zoom in properly). So we will add `cycle_mult=2` (i.e. 1st cycle is 1 epoch, 2nd cycle is 2 epochs, and 3rd cycle is 4 epochs)

```
learn.fit(1e-2, 3, cycle_len=1, cycle_mult=2)  
  
[0.      0.27171 0.2118  0.93192]  
[1.      0.28743 0.21008 0.9324 ]  
[2.      0.25328 0.20953 0.93288]  
[3.      0.23716 0.20868 0.93001]  
[4.      0.23306 0.20557 0.93384]  
[5.      0.22175 0.205    0.9324 ]  
[6.      0.2067  0.20275 0.9348 ]
```

- Now the validation loss and training loss are about the same—this is about the right track. Then we try `TTA` :

```
log_preds, y = learn.TTA()  
probs = np.exp(log_preds)  
accuracy(log_preds,y), metrics.log_loss(y, probs)  
  
(0.9393346379647749, 0.20101565705592733)
```

Other things to try:

- Try running one more cycle of 2 epochs
- Unfreezing (in this case, training convolutional layers did not help in the slightest since the images actually came from ImageNet)
- Remove validation set and just re-run the same steps, and submit that—which lets us use 100% of the data.

Question: How do we deal with unbalanced dataset? [01:38:46] This dataset is not totally balanced (between 60 and 100) but it is not unbalanced enough that Jeremy would give it a second thought. A recent paper says the best way to deal with very unbalanced dataset is to make copies of the rare cases.

Question: Difference between `precompute=True` and `unfreeze` ?

- We started with a pre-trained network
- We added a couple of layers on the end of it which start out random. With everything frozen and `precompute=True`, all we are learning is the layers we have added.
- With `precompute=True`, data augmentation does not do anything because we are showing exactly the same activations each time.
- We then set `precompute=False` which means we are still only training the layers we added because it is frozen but data augmentation is now working because it is actually going through and recalculating all of the activations from scratch.
- Then finally, we unfreeze which is saying “okay, now you can go ahead and change all of these earlier convolutional filters”.

Question: Why not just set `precompute=False` from the beginning?

The only reason to have `precompute=True` is it is much faster (10 or more times). If you are working with quite a large dataset, it can save quite a bit of time. There is no accuracy reason ever to use

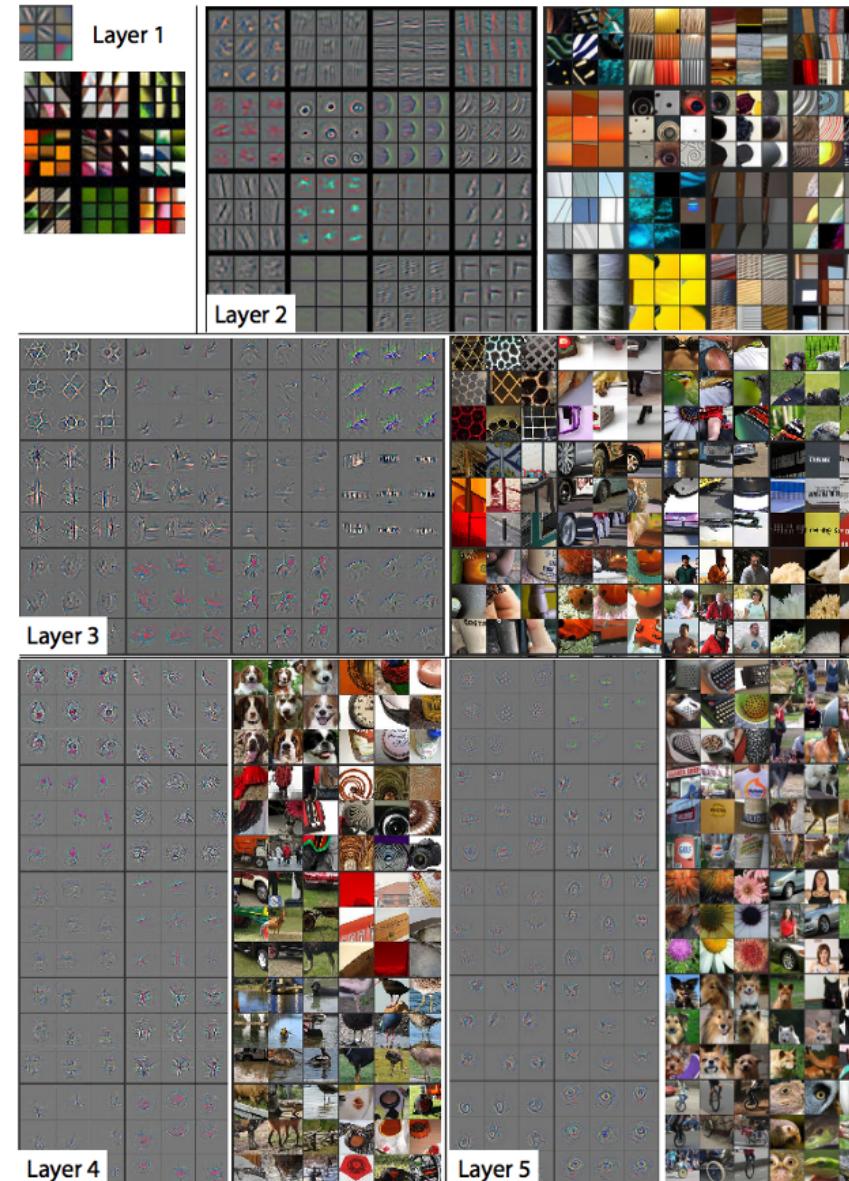
`precompute=True`.

Minimal steps to get good results:

1. Use `lr_find()` to find highest learning rate where loss is still clearly improving
2. Train last layer with data augmentation (i.e. `precompute=False`) for 2–3 epochs with `cycle_len=1`
3. Unfreeze all layers
4. Set earlier layers to 3x–10x lower learning rate than next higher layer
5. Train full network with `cycle_mult=2` until over-fitting

Question: Does reducing the batch size only affect the speed of training? [1:43:34] Yes, pretty much. If you are showing it less images each time, then it is calculating the gradient with less images—hence less accurate. In other words, knowing which direction to go and how far to go in that direction is less accurate. So as you make the batch size smaller, you are making it more volatile. It impacts the optimal learning rate that you would need to use, but in practice, dividing the batch size by 2 vs. 4 does not seem to change things very much. If you change the batch size by much, you can re-run learning rate finder to check.

Question: What are the grey images vs. the ones on the right?



Visualizing and Understanding Convolutional Networks

Layer 1, they are exactly what the filters look like. It is easy to visualize because input to it are pixels. Later on, it gets harder because inputs are themselves activations which is a combination of activations. Zeiler and Fergus came up with a clever technique to show what the filters tend to look like on average—called **deconvolution** (we will learn in Part 2). Ones on the right are the examples of patches of image which activated that filter highly.

Question: What would you have done if the dog was off to the corner or tiny (re: dog breed identification)? [01:47:16] We will learn about it in Part 2, but there is a technique that allows you to figure out roughly which parts of an image most likely have the interesting things in them. Then you can crop out that area.

Further improvement [01:48:16]

Two things we can do immediately to make it better:

1. Assuming the size of images you were using is smaller than the average size of images you have been given, you can increase the size. As we have seen before, you can increase it during training.
2. Use better architecture. There are different ways of putting together what size convolutional filters and how they are connected to each other, and different architectures have different number of layers, size of kernels, filters, etc.

We have been using ResNet34—a great starting point and often a good finishing point because it does not have too many parameters and works well with small dataset. There is another architecture called ResNext which was the second-place winner in last year’s ImageNet

competition.ResNext50 takes twice as long and 2–4 times more memory than ResNet34.

Here is the notebook which is almost identical to the original dogs. vs. cats. which uses ResNext50 which achieved 99.75% accuracy.

Satellite Imagery [01:53:01]

Notebook

```
In [5]: from planet import f2  
  
metrics=[f2]  
f_model = resnet34
```

```
In [6]: label_csv = f'{PATH}train_v2.csv'  
n = len(list(open(label_csv)))-1  
val_idxs = get_cv_idxs(n)
```

We use a different set of data augmentations for this dataset - we also allow vertical flips, since we don't expect vertical orientation of satellite images to change our classifications.

```
In [7]: def get_data(sz):  
    tfms = tfms_from_model(f_model, sz, aug_tfms=transforms_top_down, max_zoom=1.05)  
    return ImageClassifierData.from_csv(PATH, 'train-jpg', label_csv, tfms=tfms,  
                                         suffix='.jpg', val_idxs=val_idxs, test_name='test-jpg')
```

Code is pretty much the same as what we had seen before. Here are some differences:

- `transforms_top_down` —Since they satellite imagery, they still make sense when they were flipped vertically.
- Much higher learning rate—something to do with this particular dataset
- `lrs = np.array([lr/9, lr/3, lr])` —differential learning rate now change by 3x because images are quite different from ImageNet images
- `sz=64` —this helped to avoid over fitting for satellite images but he would not do that for dogs. vs. cats or dog breed (similar images to ImageNet) as 64 by 64 is quite tiny and might destroy pre-trained weights.

How to get your AWS setup [01:58:54]

You can follow along the video or here is a great write up by one of the students.



Hiromi Suenaga [Follow](#)
Jan 16 · 19 min read

Deep Learning 2: Part 1 Lesson 3

My personal notes from fast.ai course. These notes will continue to be updated and improved as I continue to review the course to “really” understand it. Much appreciation to Jeremy and Rachel who gave me this opportunity to learn.

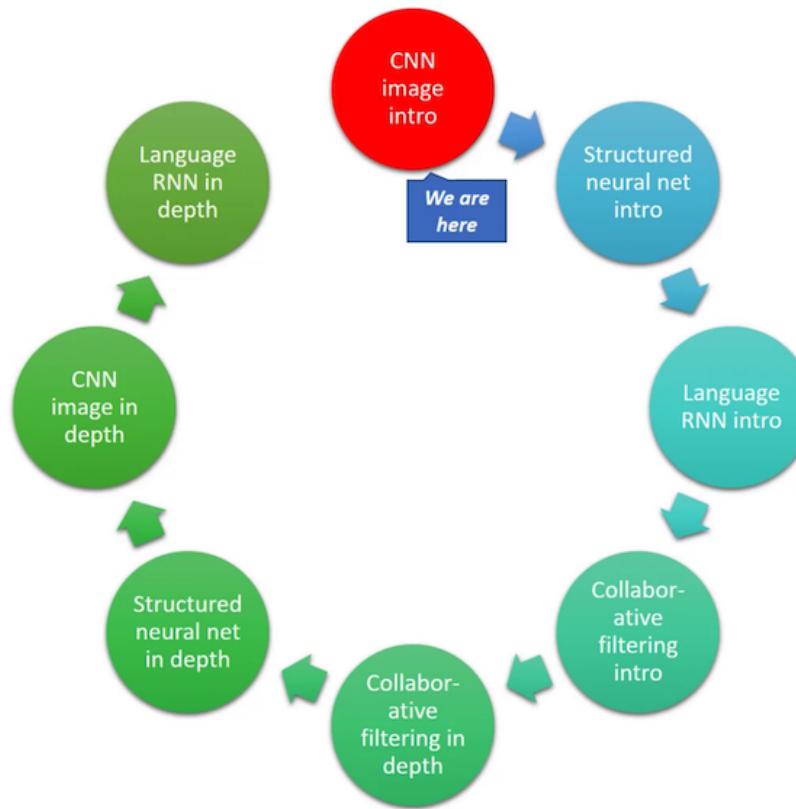
Lesson 3

Helpful materials created by students:

- AWS how-to
- Tmux
- Lesson 2 summary
- Learning rate finder
- PyTorch
- Learning rate vs. Batch size
- Smoother area of the error surface vs. generalization
- Convolutional Neural Network in 5 minutes
- Decoding ResNet Architecture

- Yet Another ResNet Tutorial

Where we go from here:



Review [08:24]:

Kaggle CLI : How to download data 1:

[Kaggle CLI](#) is a good tool to use when you are downloading from Kaggle. Because it is downloading data from Kaggle website (through screen scraping), it breaks when the website changes. When that happens, run `pip install kaggle-cli --upgrade`.

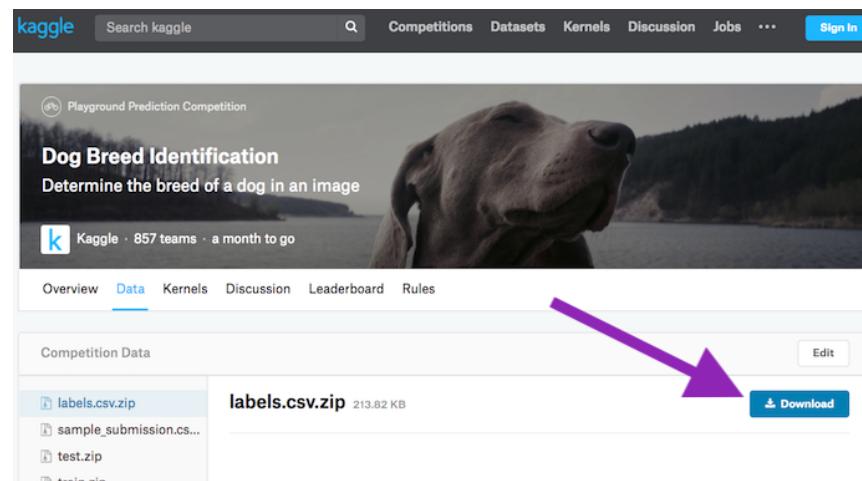
Then you can run:

```
$ kg download -u <username> -p <password> -c <competition>
```

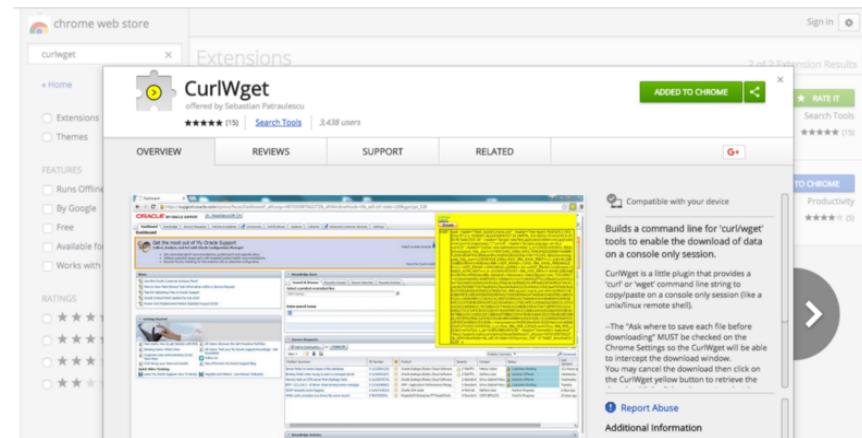
Replace `<username>`, `<password>` with your credential and `<competition>` is what follows `/c/` in the URL. For example, if you are trying to download dog breed data from <https://www.kaggle.com/c/dog-breed-identification> the command would look like:

```
$ kg download -u john.doe -p mypassword -c dog-breed-identification
```

Make sure you had clicked on the `Download` button from your computer once and accepted the rules:



CurWget (Chrome extension): How to download data 2:



Quick Dogs vs. Cats [13:39]

```
from fastai.conv_learner import *
PATH = 'data/dogscats/'
sz=224; bs=64
```

Often the notebook assumes that your data is in `data` folder. But maybe you want to put them somewhere else. In that case, you can use symbolic link (symlink for short):

```
jhoward@usf2:/data1/jhoward/git/fastai$ ls -l courses/dl1
total 10048
lrwxrwxrwx 1 jhoward jhoward      21 Sep 10 16:02 data -> /data2/datasets/part1
drwxrwxr-x 2 jhoward jhoward     4096 Nov  6 16:11 excel
lrwxrwxrwx 1 jhoward jhoward      12 Sep 10 16:05 fastai -> ../../fastai
-rw-rw-r-- 1 jhoward jhoward   337926 Oct  5 17:14 fish.ipynb
drwxrwxr-x 2 jhoward jhoward     4096 Sep 18 10:18 images
-rw-rw-r-- 1 jhoward jhoward    7231 Nov 13 14:43 keras_lesson1.ipynb
```

Here is an end to end process to get a state of the art result for dogs vs. cats:

▼ 1 Quick Dogs v Cats

```
In [9]: from fastai.conv_learner import *
PATH = "data/dogscats/"
sz=224; bs=64
```

```
In [10]: tfms = tfms_from_model(resnet50, sz, aug_tfms=transforms_side_on, max_zoom=1.1)
data = ImageClassifierData.from_paths(PATH, tfms=tfms, bs=bs)
learn = ConvLearner.pretrained(resnet50, data)
%time learn.fit(1e-2, 3, cycle_len=1)
```

...

```
In [11]: learn.unfreeze()
learn.bn_freeze(True)
%time learn.fit([1e-5,1e-4,1e-2], 1, cycle_len=1)
```

...

```
In [12]: %time log_preds,y = learn.TTA()
metrics.log_loss(y,np.exp(log_preds)), accuracy(log_preds,y)
```

CPU times: user 13.6 s, sys: 5.86 s, total: 19.5 s
Wall time: 23.8 s

```
Out[12]: (0.015632241148641549, 0.9945000000000005)
```

Quick Dogs v Cats

A little further analysis:

```
data = ImageClassifierData.from_paths(PATH, tfms= tfms,  
bs=bs, test_name='test')
```

- `from_paths` : Indicates that subfolder names are the labels. If your `train` folder or `valid` folder has a different name, you can send `trn_name` and `val_name` argument.
- `test_name` : If you want to submit to Kaggle competition, you will need to fill in the name of the folder where the test set is.

```
learn = ConvLearner.pretrained(resnet50, data)
```

- Notice that we did not set `pre_compue=True`. It is just a shortcut which caches some of the intermediate steps that do not have to be recalculated each time. If you are at all confused about it, you can just leave it off.
- Remember, when `pre_compute=True`, data augmentation does not work.

```
learn.unfreeze()  
learn.bn_freeze(True)  
%time learn.fit([1e-5, 1e-4, 1e-2], 1, cycle_len=1)
```

- `bn_freeze` : If you are using a bigger deeper model like ResNet50 or ResNext101 (anything with number bigger than 34) on a dataset that is very similar to ImageNet (i.e. side-on photos of standard object whose size is similar to ImageNet between 200–500 pixels), you should add this line. We will learn more in the second half of the course, but it is causing the batch normalization moving averages to not be updated.

How to use other libraries—Keras [20:02]

It is important to understand how to use libraries other than Fast.ai. Keras is a good example to look at because just like Fast.ai sits on top of PyTorch, it sits on top of varieties of libraries such as TensorFlow, MXNet, CNTK, etc.

If you want to run [the notebook](#), run `pip install tensorflow-gpu`
`keras`

1. Define data generators

```
train_data_dir = f'{PATH}train'  
validation_data_dir = f'{PATH}valid'  
  
train_datagen = ImageDataGenerator(rescale=1. / 255,  
shear_range=0.2, zoom_range=0.2, horizontal_flip=True)  
  
test_datagen = ImageDataGenerator(rescale=1. / 255)  
  
train_generator =  
train_datagen.flow_from_directory(train_data_dir,  
target_size=(sz, sz),  
batch_size=batch_size, class_mode='binary')
```

```
validation_generator = test_datagen.flow_from_directory(  
    validation_data_dir,  
    shuffle=False,  
    target_size=(sz, sz),  
    batch_size=batch_size, class_mode='binary')
```

- The idea of train folder and validation folder with subfolders with the label names is commonly done, and Keras also does it.
- Keras requires much more code and many more parameters to be set.
- Rather than creating a single data object, in Keras you define `DataGenerator` and specify what kind of data augmentation we want it to do and also what kind of normalization to do. In other words, in Fast.ai, we can just say “whatever ResNet50 requires, just dot hat for me please” but in Keras, you need to know what is expected. There is no standard set of augmentations.
- You have to then create a validation data generator in which you are responsible to create a generator that does not have data augmentation. And you also have to tell it not to shuffle the dataset for validation because otherwise you cannot keep track of how well you are doing.

2. Create a model

```
base_model = ResNet50(weights='imagenet', include_top=False)  
x = base_model.output  
x = GlobalAveragePooling2D()(x)
```

```
x = Dense(1024, activation='relu')(x)
predictions = Dense(1, activation='sigmoid')(x)
```

- The reason Jeremy used ResNet50 for Quick Dogs and Cats was because Keras does not have ResNet34. We want to compare apple to apple.
- You cannot ask it to construct a model that is suitable for a particular dataset, so you have to do it by hand.
- First you create a base model, then you construct layers you want to add on top of it.

3. Freeze layers and compile

```
model = Model(inputs=base_model.input, outputs=predictions)

for layer in base_model.layers: layer.trainable = False

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

- Loop through layers and freeze them manually by calling
`layer.trainable=False`
- You need to compile a model
- Pass the type of optimizer, loss, and metrics

4. Fit

```
model.fit_generator(train_generator,  
                    train_generator.n//batch_size,  
                    epochs=3, workers=4,  
                    validation_data=validation_generator,  
                    validation_steps=validation_generator.n // batch_size)
```

- Keras expects to know how many batches there are per epoch.
- `workers` : how many processors to use

5. Fine-tune: Unfreeze some layers, compile, then fit again

```
split_at = 140  
  
for layer in model.layers[:split_at]: layer.trainable =  
    False  
for layer in model.layers[split_at:]: layer.trainable = True  
  
model.compile(optimizer='rmsprop',  
              loss='binary_crossentropy',  
              metrics=['accuracy'])  
  
%%time model.fit_generator(train_generator,  
                           train_generator.n // batch_size, epochs=1, workers=3,  
                           validation_data=validation_generator,  
                           validation_steps=validation_generator.n // batch_size)
```

Pytorch—If you want to deploy to mobile devices, PyTorch is still very early.

Tensorflow—If you want to convert things you learned in this class, do more work with Keras, but it would take a bit more work and is hard to get the same level of results. Maybe there will be TensorFlow compatible version of Fast.ai in future. We will see.

Create Submission file for Kaggle [32:45]

To create the submission files, we need two pieces of information:

- `data.classes` : contains all the different classes
- `data.test_ds.fnames` : test file names

```
log_preds, y = learn.TTA(is_test=True)
probs = np.exp(log_preds)
```

It is always good idea to use `TTA`:

- `is_test=True` : it will give you predictions on the test set rather than the validation set
- By default, PyTorch models will give you back the log of the predictions, so you need to do `np.exp(log_preds)` to get the probability.

```
ds = pd.DataFrame(probs)
ds.columns = data.classes
```

- Create Pandas `DataFrame`
- Set the column name as `data.classes`

```
df.insert(0, 'id', [o[5:-4] for o in data.test_ds.fnames])
```

- Insert a new column at position zero named `id`. Remove first 5 and last 4 letters since we just need IDs (a file name looks like

```
test/0042d6bf3e5f3700865886db32689436.jpg )
```

```
df.head()
```

		id	affenpinscher	afghan_hound	african_hunting_dog	airedale	american_staffordshire_terrier
0	bd817f883c35e1c77516f60e3e2bfcaf		0.000119	4.331197e-04	1.980616e-05	0.005610	7.083308e-
1	cd3a666d082b6cb9a44ddb1dca5eedc9		0.000731	3.163513e-03	4.280231e-04	0.000139	2.575643e-
2	a7b2cc836cca8fc7c844fb4cccc5401a		0.000001	5.223153e-07	1.764584e-07	0.740886	6.112249e-
3	5366834111cdfdc1876d2ce66ca8dead		0.000362	8.363432e-04	4.421286e-04	0.002472	5.485309e-
4	66f8bb9969fce49747c9b9a47228ad		0.000010	1.971873e-05	6.064667e-06	0.000022	1.028777e-

5 rows × 121 columns

```
SUBM = f'{PATH}sub/'
os.makedirs(SUBM, exist_ok=True)
df.to_csv(f'{SUBM}subm.gz', compression='gzip', index=False)
```

- Now you can call `df.to_csv` to create a CSV file and `compression='gzip'` will zip it up on the server.

```
FileLink(f'{SUBM}subm.gz')
```

- You can use Kaggle CLI to submit from the server directly, or you can use `FileLink` which will give you a link to download the file from the server to your computer.

Individual prediction [39:32]

What if we want to run a single image through a model to get a prediction?

```
fn = data.val_ds.fnames[0]; fn  
  
'train/001513dfcb2ffafc82cccf4d8bbaba97.jpg'  
  
Image.open(PATH + fn)
```



- We will pick a first file from the validation set.

This is the shortest way to get a prediction:

```
trn_tfms, val_tfms = tfms_from_model(arch, sz)

im = val_tfms(Image.open(PATH+fn))

preds = learn.predict_array(im[None])

np.argmax(preds)
```

- Image must be transformed. `tfms_from_model` returns training transforms and validation transforms. In this case, we will use validation transform.
- Everything that gets passed to or returned from a model is generally assumed to be in a mini-batch. Here we only have one image, but we have to turn that into a mini-batch of a single image. In other words, we need to create a tensor that is not just `[rows, columns, channels]`, but `[number of images, rows, columns, channels]`.
- `im[None]`: Numpy trick to add additional unit axis to the start.

Theory: What is actually going on behind the scenes with convolutional neural network [42:17]

- We saw a little bit of theory in Lesson 1—
<http://setosa.io/ev/image-kernels/>
- Convolution is something where we have a little matrix (nearly always 3x3 in deep learning) and multiply every element of that matrix by every element of 3x3 section of an image and add them all together to get the result of that convolution at one point.

Otavio's fantastic visualization (he created Word Lens):

A visual and intuitive understanding of deep learn...



Jeremy's visualization: [Spreadsheet \[49:51\]](#)

42	U	U	U	1	U	U	U	U	1	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	
43	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	3	2	0	0	0	0
44	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	3	0	0	0	0
45	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	3	1	0	0	0
46	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	3	2	0	0	0
47	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	3	2	0	0	0
48	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	2	3	1	0	0	0
49	0	0	0	0	0	0	0	0	1	1	1	0	1	0	1	0	0	0	0	0	0	0	2	3	1	0	0	0
50	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	3	3	0	0	0	0
51	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	3	2	0	0	0
52	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	3	1	0	0	0
53	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	3	2	0	0	0
54	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	1	0	0	0
55	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	1	0	0	0
56	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	0	0	0	0
57	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0
58	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

I used <https://office.live.com/start/Excel.aspx>

- This data is from MNIST
- **Activation:** A number that is calculated by applying some kind of linear operation to some numbers in the input.
- **Rectified Linear Unit (ReLU):** Throw away negative—i.e. $\text{MAX}(0, x)$
- **Filter/Kernel:** A 3x3 slice of a 3D tensor you used for convolution
- **Tensor:** Multidimensional array or matrix Hidden Layer A layer that is neither input nor output
- **Max pooling:** A (2,2) max pooling will halve the resolution in both height and width—think of it as a summary
- **Fully connected layer:** Give a weight to each and every single activation and calculate the sum product. Weight matrix is as big as the entire input.

- Note: There are many things you can do after the max pooling layer. One of them is to do another max pool across the entire size. In older architectures or structured data, we do fully connected layer. Architecture that make heavy use of fully connected layers are prone to overfitting and are slower. ResNet and ResNext do not use very large fully connected layers.

Question: What happens if the input had 3 channels? [1:05:30] It will look something similar to the Conv1 layer which has 2 channels—therefore, filters have 2 channels per filter. Pre-trained ImageNet models use 3 channels. Some of the techniques you can use when you do when you do have less than 3 channel is to either duplicate one of the channels to make it 3, or if you have 2, then get an average and consider that as the third channel. If you have 4 channels, you could add extra level to the convolutional kernel with all zeros.

What happens next? [1:08:47]

We have gotten as far as fully connected layer (it does classic matrix product). In the excel sheet, there is one activation. If we want to look at which one of ten digit the input is, we actually want to calculate 10 numbers.

Let's look at an example where we are trying to predict whether a picture is a cat, a dog, or a plane, or fish, or a building. Our goal is:

1. Take output from the fully connected layer (no ReLU so there may be negatives)
2. Calculate 5 numbers where each of them is between 0 and 1 and they add up to 1.

To do this, we need a different kind of activation function (a function applied to an activation).

Why do we need non-linearity? If you stack multiple linear layers, it is still just a linear layer. By adding non-linear layers, we can fit arbitrarily complex shapes. The non-linear activation function we used was ReLU.

Softmax [01:14:08]

Softmax only ever occurs in the final layer. It outputs numbers between 0 and 1, and they add up to 1. In theory, this is not strictly necessary—we could ask our neural net to learn a set of kernels which give probabilities that line up as closely as possible with what we want. In general with deep learning, if you can construct your architecture so that the desired characteristics are as easy to express as possible, you will end up with better models (learn more quickly and with less parameters).

	A	B	C	D
1		output	exp	softmax
2	cat	-1.83	0.16	0.00
3	dog	2.85	17.25	0.09
4	plane	3.86	47.54	0.26
5	fish	4.08	59.03	0.32
6	building	4.07	58.78	0.32
7		182.75		1.00

- Get rid of negatives by e^x because we cannot have negative probabilities. It also accentuates the value difference (2.85 : 4.08 → 17.25 : 59.03)

All the math that you need to be familiar with to do deep learning:

$$\ln(x \cdot y) = \ln(x) + \ln(y)$$

$$\ln\left(\frac{x}{y}\right) = \ln(x) - \ln(y)$$

$$\ln(x) = y \rightarrow e^y = x$$

↓
inverse

- We then add up the \exp column (182.75), and divide the e^x by the sum. The result will always be positive since we divided positive by positive. Each number will be between 0 and 1, and the total will be 1.

Question: What kind of activation function do we use if we want to classify the picture as cat and dog? [1:20:27] It so happens that we are going to do that right now. One reason we might want to do that is to do multi-label classification.

Planet Competition [01:20:54]

Notebook / Kaggle page

I would definitely recommend anthropomorphizing your activation functions. They have personalities.

Softmax does not like to predicting multiple things. It wants to pick one thing.

Fast.ai library will automatically switch into multi-label mode if there is more than one label. So you do not have to do anything. But here is what happens behind the scene:

```
from planet import f2

metrics=[f2]
f_model = resnet34

label_csv = f'{PATH}train_v2.csv'
n = len(list(open(label_csv)))-1
val_idxs = get_cv_idxs(n)

def get_data(sz):
    tfms = tfms_from_model(f_model, sz,
                           aug_tfms=transforms_top_down, max_zoom=1.05)

    return ImageClassifierData.from_csv(PATH, 'train-jpg',
                                        label_csv, tfms=tfms, suffix='.jpg',
                                        val_idxs=val_idxs, test_name='test-jpg')

data = get_data(256)
```

- Multi-label classification cannot be done with Keras style approach where subfolder is the name of the label. So we use `from_csv`
- `transform_top_down` : it does more than just a vertical flip. There are 8 possible symmetries for a square—it can be rotated through 0, 90, 180, 270 degrees and for each of those, it can be flipped (**dihedral group of eight**)

```
x,y = next(iter(data.val_dl))
```

- We had seen `data.val_ds` , `test_ds` , `train_ds` (`ds` : dataset) for which you can get an individual image by `data.train_ds[0]` , for example.
- `dl` is a data loader which will give you a mini-batch, specifically *transformed* mini-batch. With a data loader, you cannot ask for a particular mini-batch; you can only get back the `next` mini-batch. In Python, it is called “generator” or “iterator”. PyTorch really leverages modern Python methodologies.

If you know Python well, PyTorch comes very naturally. If you don't know Python well, PyTorch is a good reason to learn Python well.

- `x` : a mini-batch of images, `y` : a mini-batch of labels.

If you are never sure what arguments a function takes, hit `shift+tab` .

```
list(zip(data.classes, y[0]))  
  
[('agriculture', 1.0),  
 ('artisinal_mine', 0.0),  
 ('bare_ground', 0.0),  
 ('blooming', 0.0),  
 ('blow_down', 0.0),  
 ('clear', 1.0),  
 ('cloudy', 0.0),  
 ('conventional_mine', 0.0),  
 ('cultivation', 0.0),  
 ('habitation', 0.0),  
 ('haze', 0.0),  
 ('partly_cloudy', 0.0),  
 ('primary', 1.0),  
 ('road', 0.0),  
 ('selective_logging', 0.0),  
 ('slash_burn', 1.0),  
 ('water', 1.0)]
```

Behind the scene, PyTorch and fast.ai convert our labels into one-hot-encoded labels. If the actual label is dog, it will look like:

	A	B	C	D	E
1		output	exp	softmax	actuals
2	cat	1.60	4.94	0.48	0
3	dog	1.10	3.01	0.30	1
4	plane	-0.54	0.58	0.06	0
5	fish	-3.71	0.02	0.00	0
6	building	0.50	1.65	0.16	0
7			10.20	1.00	

We take the difference between `actuals` and `softmax`, add them up to say how much error there is (i.e. loss function).

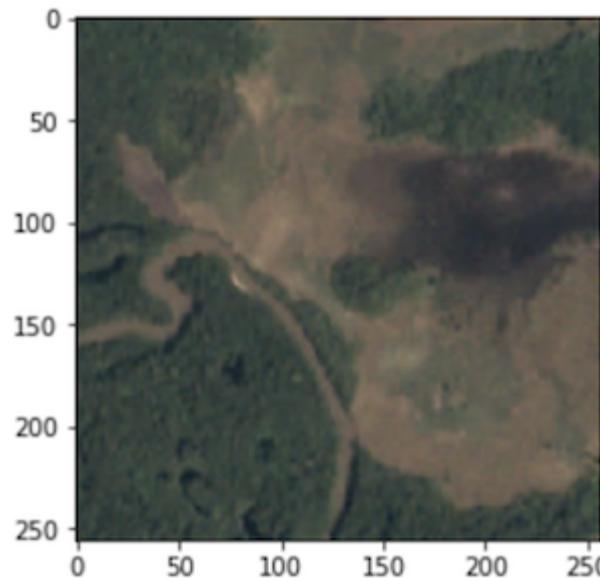
One-hot-encoding is not efficient for single label classification problems, so you will see an index (single integer) rather than 0's and 1's for the target value (`y`). But internally, PyTorch is converting the index to one-hot-encoded label. So, technically we are doing the same thing for both single label classification and multi label classification.

Question: Does it make sense to change the base of log for softmax?

[01:32:55] No, changing the base is just a linear scaling which neural net can learn easily:

$$\log_b a = \frac{\log_d(a)}{\log_d(b)}$$

```
plt.imshow(data.val_ds.denorm(to_np(x))[0]*1.4);
```



- `*1.4` : The image was washed out, so making it more visible (“brightening it up a bit”). Images are just matrices of numbers, so we can do things like this.
- It is good to experiment images like this because these images are not at all like ImageNet. The vast majority of things you do involving convolutional neural net will not actually be anything like ImageNet (medical imaging, classifying different kinds of steel tube, satellite images, etc)

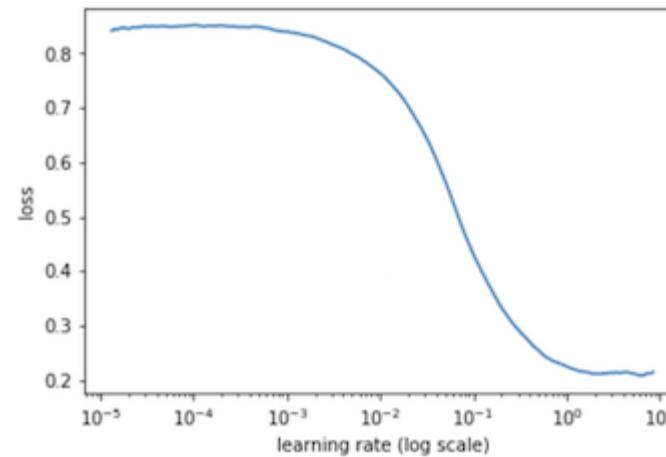
```
sz=64

data = get_data(sz)
data = data.resize(int(sz*1.3), 'tmp')
```

- We will not use `sz=64` for cats and dogs competition because we started with pre-trained ImageNet network which starts off nearly perfect. If we re-trained the whole set with 64 by 64 images, we would destroy the weights that are already very good. Remember, most of ImageNet models are trained with 224 by 224 or 299 by 299 images.
- There is no images in ImageNet that looks like the one above. And only the first couple layers are useful to us. So starting out with smaller images works well in this case.

```
learn = ConvLearner.pretrained(f_model, data,
metrics=metrics)

lrf=learn.lr_find()
learn.sched.plot()
```



```
lr = 0.2
learn.fit(lr, 3, cycle_len=1, cycle_mult=2)
```

```
[ 0.      0.14882  0.13552  0.87878]
[ 1.      0.14237  0.13048  0.88251]
[ 2.      0.13675  0.12779  0.88796]
[ 3.      0.13528  0.12834  0.88419]
[ 4.      0.13428  0.12581  0.88879]
[ 5.      0.13237  0.12361  0.89141]
[ 6.      0.13179  0.12472  0.8896 ]
```

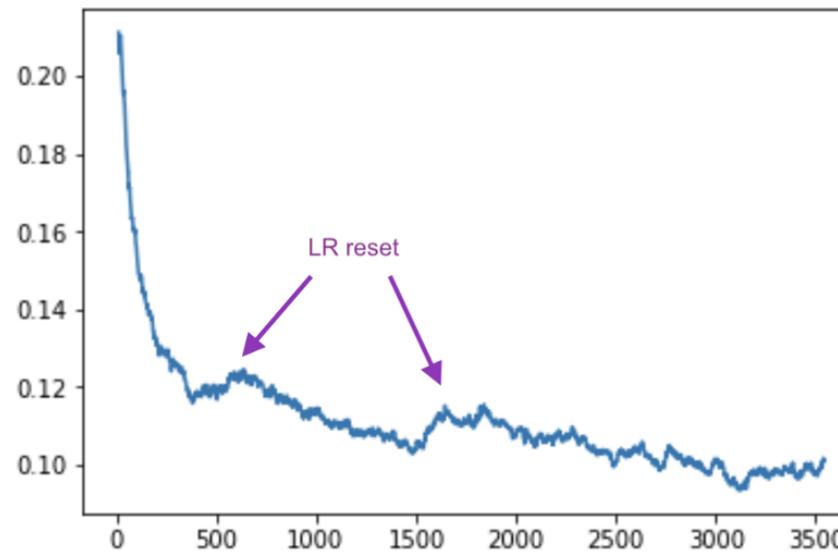
```
lrs = np.array([lr/9, lr/3, lr])

learn.unfreeze()
learn.fit(lrs, 3, cycle_len=1, cycle_mult=2)
```

```
[ 0.      0.12534  0.10926  0.90892]
[ 1.      0.12035  0.10086  0.91635]
[ 2.      0.11001  0.09792  0.91894]
[ 3.      0.1144   0.09972  0.91748]
[ 4.      0.11055  0.09617  0.92016]
[ 5.      0.10348  0.0935   0.92267]
[ 6.      0.10502  0.09345  0.92281]
```

- `[lr/9, lr/3, lr]` —this is because the images are unlike ImageNet image and earlier layers are probably not as close to what they need to be.

```
learn.sched.plot_loss()
```



```
sz = 128
learn.set_data(get_data(sz))
learn.freeze()
learn.fit(lr, 3, cycle_len=1, cycle_mult=2)
```

```
[ 0.      0.09729  0.09375  0.91885]
[ 1.      0.10118  0.09243  0.92075]
[ 2.      0.09805  0.09143  0.92235]
[ 3.      0.09834  0.09134  0.92263]
[ 4.      0.096     0.09046  0.9231 ]
[ 5.      0.09584  0.09035  0.92403]
[ 6.      0.09262  0.09059  0.92358]
```

```
◀ [ 0.      0.09623  0.08693  0.92696]
[ 1.      0.09371  0.08621  0.92887] ▶
learn.unfreeze()
learn.fit(lrs, 3, cycle_len=1, cycle_mult=2)
learn.save(f'{sz}')
```

```
[ 0.      0.09623  0.08693  0.92696]
[ 1.      0.09371  0.08621  0.92887]
```

```
[ 2.      0.08919  0.08296  0.93113]
[ 3.      0.09221  0.08579  0.92709]
[ 4.      0.08994  0.08575  0.92862]
[ 5.      0.08729  0.08248  0.93108]
[ 6.      0.08218  0.08315  0.92971]
```

```
sz = 256
learn.set_data(get_data(sz))
learn.freeze()
learn.fit(lr, 3, cycle_len=1, cycle_mult=2)
```

```
[ 0.      0.09161  0.08651  0.92712]
[ 1.      0.08933  0.08665  0.92677]
[ 2.      0.09125  0.08584  0.92719]
[ 3.      0.08732  0.08532  0.92812]
[ 4.      0.08736  0.08479  0.92854]
[ 5.      0.08807  0.08471  0.92835]
[ 6.      0.08942  0.08448  0.9289 ]
```

```
learn.unfreeze()
learn.fit(lrs, 3, cycle_len=1, cycle_mult=2)
learn.save(f'{sz}')
```

```
[ 0.      0.08932  0.08218  0.9324 ]
[ 1.      0.08654  0.08195  0.93313]
[ 2.      0.08468  0.08024  0.93391]
[ 3.      0.08596  0.08141  0.93287]
[ 4.      0.08211  0.08152  0.93401]
[ 5.      0.07971  0.08001  0.93377]
[ 6.      0.07928  0.0792   0.93554]
```

```
log_preds,y = learn.TTA()
preds = np.mean(np.exp(log_preds),0)
f2(preds,y)
```

```
0.93626519738612801
```

A couple of questions people have asked what this does [01:38:46]:

```
data = data.resize(int(sz*1.3), 'tmp')
```

When we specify what transforms to apply, we send a size:

```
tfms = tfms_from_model(f_model, sz,
                      aug_tfms=transforms_top_down, max_zoom=1.05)
```

One of the things the data loader does is to resize the images on-demand. This has nothing to do with `data.resize`. If the initial image is 1000 by 1000, reading that JPEG and resizing it to 64 by 64 take more time than training the convolutional net. `data.resize` tells it that we will not use images bigger than `sz*1.3` so go through once and create new JPEGs of this size. Since images are rectangular, so new JPEGs whose smallest edge is `sz*1.3` (center-cropped). It will save you a lot of time.

```
metrics=[f2]
```

Instead of `accuracy`, we used `F-beta` for this notebook—it is a way of weighing false negatives and false positives. The reason we are using it is because this particular Kaggle competition wants to use it. Take a

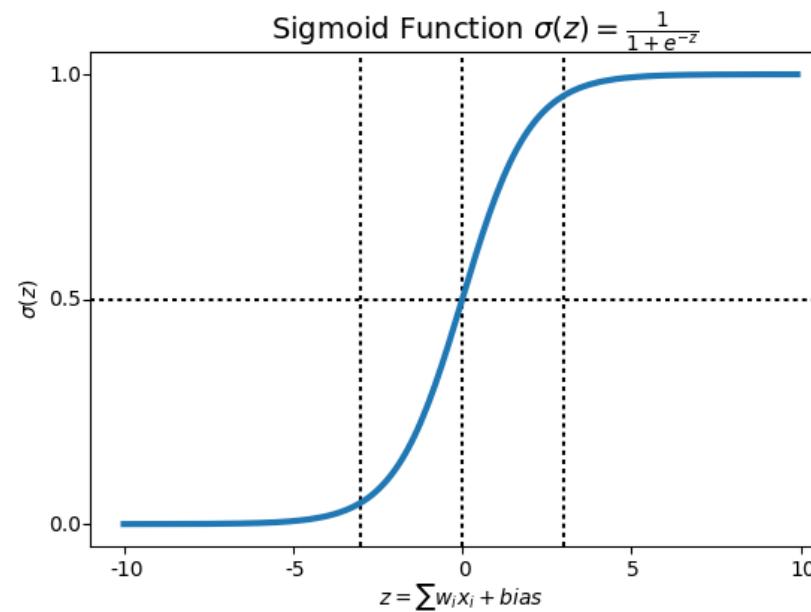
look at [planet.py](#) to see how you can create your own metrics function.

This is what gets printed out at the end [0. 0.08932 0.08218 0.9324]

Activation function for multi-label classification [01:44:25]

Activation function for multi-label classification is called **sigmoid**.

	A	B	C	D	E	F	G	H
1		output	exp	softmax	actuals	index		
2	cat	-4.16	0.02	0.00	0	0	=C2/(1+C2)	
3	dog	-0.97	0.38	0.08	1	1		0
4	plane	-4.21	0.01	0.00	0	2		1
5	fish	1.31	3.69	0.75	0	3		1
6	building	-0.22	0.80	0.16	0	4		0
7			4.90	1.00				



Question: Why don't we start training with differential learning rate rather than training the last layers alone? [01:50:30]

```
In [10]: tfms = tfms_from_model(resnet50, sz, aug_tfms=transforms_side_on, max_zoom=1.1)
data = ImageClassifierData.from_paths(PATH, tfms=tfms, bs=bs)
learn = ConvLearner.pretrained(resnet50, data)
%time learn.fit(1e-2, 3, cycle_len=1)
```

```
In [11]: learn.unfreeze()
learn.bn_freeze(True)
%time learn.fit([1e-5,1e-4,1e-2], 1, cycle_len=1)
```

You can skip training just the last layer and go straight to differential learning rates, but you probably do not want to. Convolutional layers all contain pre-trained weights, so they are not random—for things that are close to ImageNet, they are really good; for things that are not close to ImageNet, they are better than nothing. All of our fully connected layers, however, are totally random. Therefore, you would always want to make the fully connected weights better than random by training them a bit first. Otherwise if you go straight to unfreeze, then you are actually going to be fiddling around with those early layer weights when the later ones are still random—which is probably not what you want.

Question: When you use the differential learning rates, do those three learning rates spread evenly across the layers? [01:55:35] We will talk more about this later in the course but the fast.ai library, there is a concept of “layer groups”. In something like ResNet50, there are hundreds of layers and you probably do not want to write hundreds of learning rates, so the library decided for you how to split them and the last one always refers to just the fully connected layers that we have randomly initialized and added.

Visualizing the layers [01:56:42]

```
learn.summary()

[('Conv2d-1',
  OrderedDict([('input_shape', [-1, 3, 64, 64]),
              ('output_shape', [-1, 64, 32, 32]),
              ('trainable', False),
              ('nb_params', 9408)])),
 ('BatchNorm2d-2',
  OrderedDict([('input_shape', [-1, 64, 32, 32]),
              ('output_shape', [-1, 64, 32, 32]),
              ('trainable', False),
              ('nb_params', 128)])),
 ('ReLU-3',
  OrderedDict([('input_shape', [-1, 64, 32, 32]),
              ('output_shape', [-1, 64, 32, 32]),
              ('nb_params', 0)])),
 ('MaxPool2d-4',
  OrderedDict([('input_shape', [-1, 64, 32, 32]),
              ('output_shape', [-1, 64, 16, 16]),
              ('nb_params', 0)])),
 ('Conv2d-5',
  OrderedDict([('input_shape', [-1, 64, 16, 16]),
              ('output_shape', [-1, 64, 16, 16]),
              ('trainable', False),
              ('nb_params', 36864)]))
 ...
```

- `'input_shape', [-1, 3, 64, 64]`—PyTorch lists channel before the image size. Some of the GPU computations run faster when it is in that order. This is done behind scene by the transformation step.
- `-1` : indicates however big the batch size is. Keras uses `None` .

- `'output_shape', [-1, 64, 32, 32]` —64 is the number of kernels

Question: Learning rate finder for a very small dataset returned strange number and the plot was empty [01:58:57]—The learning rate finder will go through a mini-batch at a time. If you have a tiny dataset, there is just not enough mini-batches. So the trick is to make your batch size very small like 4 or 8.

Structured Data [01:59:48]

There are two types of dataset we use in machine learning:

- **Unstructured**—Audio, images, natural language text where all of the things inside an object are all the same kind of things—pixels, amplitude of waveform, or words.
- **Structured**—Profit and loss statement, information about a Facebook user where each column is structurally quite different. “Structured” refers to columnar data as you might find in a database or a spreadsheet where different columns represent different kinds of things, and each row represents an observation.

Structured data is often ignored in academics because it is pretty hard to get published in fancy conference proceedings if you have a better logistics model. But it is the thing that makes the world goes round, makes everybody money and efficiency. We will not ignore it because we are doing practical deep learning, and Kaggle does not either because people put prize money up on Kaggle to solve real-world problems:

- Corporación Favorita Grocery Sales Forecasting—which is currently running
- Rossmann Store Sales—almost identical to above but completed competition.

Rossmann Store Sale [02:02:42]

Notebook

```
from fastai.structured import *
from fastai.column_data import *
np.set_printoptions(threshold=50, edgeitems=20)

PATH='data/rossmann/'
```

- `fastai.structured` —not PyTorch specific and also used in machine learning course doing random forests with no PyTorch at all. It can be used on its own without any of the other parts of Fast.ai library.
- `fastai.column_data` —allows us to do Fast.ai and PyTorch stuff with columnar structured data.
- For structured data need to use **Pandas** a lot. Pandas is an attempt to replicate R's data frames in Python (If you are not familiar with Pandas, here is a good book—Python for Data Analysis, 2nd Edition)

There are a lot of data pre-processing. This notebook contains the entire pipeline from the third place winner (Entity Embeddings of Categorical

Variables). Data processing is not covered in this course, but is covered in machine learning course in some detail because feature engineering is very important.

Looking at CSV files

```
table_names = ['train', 'store', 'store_states',
'state_names',
'googletrend', 'weather', 'test']

tables = [pd.read_csv(f'{PATH}{fname}.csv',
low_memory=False) for fname in table_names]

for t in tables: display(t.head())
```

```
for t in tables: display(t.head())
```

	Store	DayOfWeek	Date	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday
0	1	5	2015-07-31	5263	555	1	1	0	1
1	2	5	2015-07-31	6064	625	1	1	0	1
2	3	5	2015-07-31	8314	821	1	1	0	1
3	4	5	2015-07-31	13995	1498	1	1	0	1
4	5	5	2015-07-31	4822	559	1	1	0	1
	Store	StoreType	Assortment	CompetitionDistance	CompetitionOpenSinceMonth	CompetitionOpenSinceYear	Promo2	Promo2SinceWeek	Promo2SinceYear
0	1	c	a	1270.0		9.0	2008.0	0	NaN
1	2	a	a	570.0		11.0	2007.0	1	13.0
2	3	a	a	14130.0		12.0	2006.0	1	14.0
3	4	c	c	620.0		9.0	2009.0	0	NaN
4	5	a	a	29910.0		4.0	2015.0	0	NaN

- `StoreType` —you often get datasets where some columns contain “code”. It really does not matter what the code means. Stay away from learning too much about it and see what the data says first.

Joining tables

This is a relational dataset, and you have join quite a few tables together—which is easy to do with Pandas’ `merge` :

```
def join_df(left, right, left_on, right_on=None,
suffix='_y'):
    if right_on is None: right_on = left_on

    return left.merge(right, how='left', left_on=left_on,
                      right_on=right_on, suffixes=("", suffix))
```

From Fast.ai library:

```
add_datepart(train, "Date", drop=False)
```

- Take a date and pull out a bunch of columns such as “day of week”, “start of a quarter”, “month of year” and so on and add them all to the dataset.
- Duration section will calculate things like how long until the next holiday, how long it has been since the last holiday, etc.

```
joined.to_feather(f'{PATH}joined')
```

- `to_feather` : Saves a Pandas' data frame into a “feather” format which takes it as it sits in RAM and dumps it to the disk. So it is really really fast. Ecuadorian grocery competition has 350 million records, so you will care about how long it takes to save.

Next week

- split columns into two types: categorical and continuous.
Categorical column will be represented as one hot encoding, and continuous column gets fed into fully connected layer as is.
- categorical: store #1 and store #2 are not numerically related to each other. Similarly, day of week Monday (day 0) and Tuesday (day 1).
- continuous: Things like distance in kilometers to the nearest competitor is a number we treat numerically.
- `ColumnarModelData`



Hiromi Suenaga [Follow](#)
Jan 19 · 36 min read

Deep Learning 2: Part 1 Lesson 4

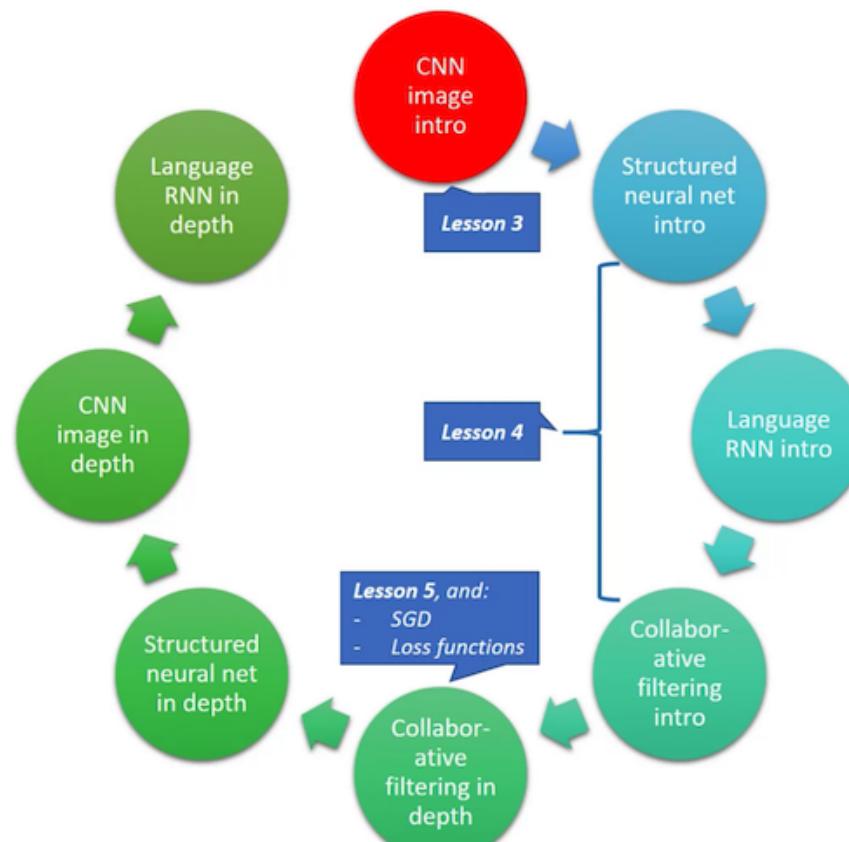
My personal notes from fast.ai course. These notes will continue to be updated and improved as I continue to review the course to “really” understand it. Much appreciation to Jeremy and Rachel who gave me this opportunity to learn.

• • •

Lesson 4

Articles by students:

- [Improving the way we work with learning rate](#)
- [The Cyclical Learning Rate technique](#)
- [Exploring Stochastic Gradient Descent with Restarts \(SGDR\)](#)
- [Transfer Learning using differential learning rates](#)
- [Getting Computers To See Better Than Humans](#)



Dropout [04:59]

```
learn = ConvLearner.pretrained(arch, data, ps=0.5,  
precompute=True)
```

- `precompute=True` : Pre-compute the activations that come out of the last convolutional layer. Remember, activation is a number

that is calculated based on some weights/parameter that makes up kernels/filters, and they get applied to the previous layer's activations or inputs.

```
learn

Sequential(
    (0): BatchNorm1d(1024, eps=1e-05, momentum=0.1,
affine=True)
    (1): Dropout(p=0.5)
    (2): Linear(in_features=1024, out_features=512)
    (3): ReLU()
    (4): BatchNorm1d(512, eps=1e-05, momentum=0.1,
affine=True)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=512, out_features=120)
    (7): LogSoftmax()
)
```

`learn` —This will display the layers we added at the end. These are the layers we train when `precompute=True`

(0), (4): `BatchNorm` will be covered in the last lesson

(1), (5): `Dropout`

(2): `Linear` layer simply means a matrix multiply. This is a matrix which has 1024 rows and 512 columns, so it will take in 1024 activations and spit out 512 activations.

(3): `ReLU` —just replace negatives with zero

(6): `Linear` —the second linear layer that takes those 512 activations from the previous linear layer and put them through a new matrix multiply 512 by 120 and outputs 120 activations

(7): `Softmax` —The activation function that returns numbers that adds up to 1 and each of them is between 0 and 1:

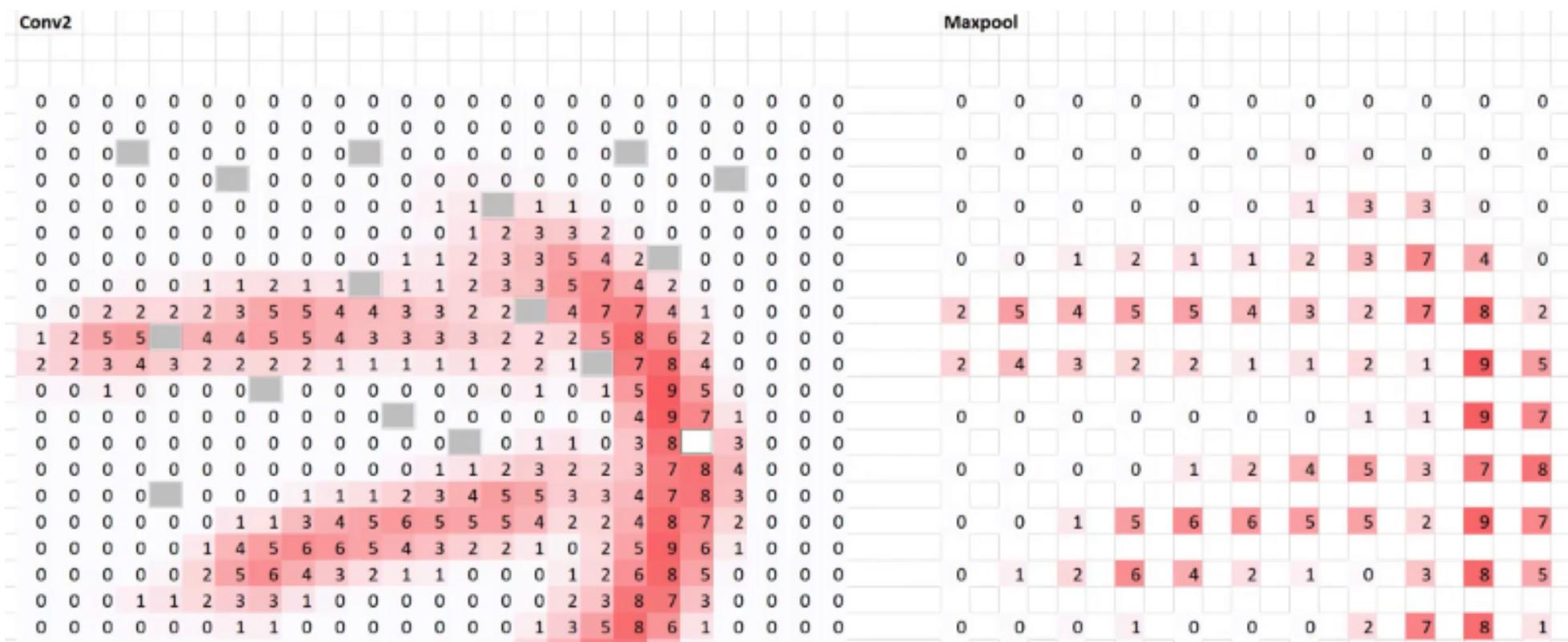
$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

For minor numerical precision reasons, it turns out to be better to take the log of the softmax than softmax directly [15:03]. That is why when we get predictions out of our models, we have to do

```
np.exp(log_preds) .
```

What is `Dropout` and what is `p`? [08:17]

Dropout (p=0.5)



If we applied dropout with `p=0.5` to `Conv2` layer, it would look like the above. We go through, pick an activation, and delete it with 50% chance. So `p=0.5` is the probability of deleting that cell. Output does not actually change by very much, just a little bit.

Randomly throwing away half of the activations in a layer has an interesting effect. An important thing to note is for each mini-batch, we throw away a different random half of activations in that layer. It forces it to not overfit. In other words, when a particular activation that learned just that exact dog or exact cat gets dropped out, the model has

to try and find a representation that continues to work even as random half of the activations get thrown away every time.

This has been absolutely critical in making modern deep learning work and just about solve the problem of generalization. Geoffrey Hinton and his colleagues came up with this idea loosely inspired by the way the brain works.

- `p=0.01` will throw away 1% of the activations. It will not change things up very much at all, and will not prevent overfitting (not generalized).
- `p=0.99` will throw away 99% of the activations. Not going to overfit and great for generalization, but will kill your accuracy.
- By default, the first layer is `0.25` and second layer is `0.5` [17:54]. If you find it is overfitting, start bumping it up—try setting all to `0.5`, still overfitting, try `0.7` etc. If you are underfitting, you can try making it lower but is unlikely you would need to make it much lower.
- ResNet34 has less parameters so it does not overfit as much, but for bigger architecture like ResNet50, you often need to increase dropout.

Have you wondered why the validation losses better than the training losses particularly early in the training? [12:32] This is because we turn off dropout when we run inference (i.e. making prediction) on the validation set. We want to be using the best model we can.

Question: Do you have to do anything to accommodate for the fact that you are throwing away activations? [13:26] We do not, but PyTorch

does two things when you say `p=0.5`. It throws away half of the activations, and it doubles all the activations that are already there so that average activation does not change.

In Fast.ai, you can pass in `ps` which is the `p` value for all of the added layers. It will not change the dropout in the pre-trained network since it should have been already trained with some appropriate level of dropout:

```
learn = ConvLearner.pretrained(arch, data, ps=0.5,  
precompute=True)
```

You can remove dropout by setting `ps=0.` but even after a couple epochs, we start to massively overfit (training loss << validation loss):

```
[2.      0.3521   0.55247  0.84189]
```

When `ps=0.` , dropout layers are not even added to the model:

```
Sequential(  
    (0): BatchNorm1d(4096, eps=1e-05, momentum=0.1,  
    affine=True)  
    (1): Linear(in_features=4096, out_features=512)  
    (2): ReLU()  
    (3): BatchNorm1d(512, eps=1e-05, momentum=0.1,  
    affine=True)  
    (4): Linear(in_features=512, out_features=120))
```

```
(5): LogSoftmax()  
)
```

You may have noticed, it has been adding two `Linear` layers [16:19].

We do not have to do that. There is `xtra_fc` parameter you can set.

Note: you do need at least one which takes the output of the convolutional layer (4096 in this example) and turns it into the number of classes (120 dog breeds):

```
learn = ConvLearner.pretrained(arch, data, ps=0.,  
precompute=True,  
      xtra_fc=[]); learn  
  
Sequential(  
    (0): BatchNorm1d(1024, eps=1e-05, momentum=0.1,  
affine=True)  
    (1): Linear(in_features=1024, out_features=120)  
    (2): LogSoftmax()  
)  
  
learn = ConvLearner.pretrained(arch, data, ps=0.,  
precompute=True,  
      xtra_fc=[700, 300]); learn  
  
Sequential(  
    (0): BatchNorm1d(1024, eps=1e-05, momentum=0.1,  
affine=True)  
    (1): Linear(in_features=1024, out_features=700)  
    (2): ReLU()  
    (3): BatchNorm1d(700, eps=1e-05, momentum=0.1,  
affine=True)  
    (4): Linear(in_features=700, out_features=300)  
    (5): ReLU()  
    (6): BatchNorm1d(300, eps=1e-05, momentum=0.1,  
affine=True)
```

```
(7): Linear(in_features=300, out_features=120)
(8): LogSoftmax()
)
```

Question: Is there a particular way in which you can determine if it is overfitted? [19:53]. Yes, you can see the training loss is much lower than the validation loss. You cannot tell if it is *too* overfitted. Zero overfitting is not generally optimal. The only thing you are trying to do is to get the validation loss low, so you need to play around with a few things and see what makes the validation loss low. You will get a feel for it overtime for your particular problem what too much overfitting looks like.

Question: Why does average activation matter? [21:15] If we just deleted a half of activations, the next activation who takes them as input will also get halved, and everything after that. For example, fluffy ears are fluffy if this is greater than 0.6, and now it is only fluffy if it is greater than 0.3—which is changing the meaning. The goal here is delete activations without changing the meanings.

Question: Can we have different level of dropout by layer? [22:41] Yes, that is why it is called `ps` :

```
learn = ConvLearner.pretrained(arch, data, ps=[0., 0.2],
                               precompute=True, xtra_fc=[512]); learn

Sequential(
    (0): BatchNorm1d(4096, eps=1e-05, momentum=0.1,
affine=True)
    (1): Linear(in_features=4096, out_features=512)
    (2): ReLU()
```

```
(3): BatchNorm1d(512, eps=1e-05, momentum=0.1,
affine=True)
(4): Dropout(p=0.2)
(5): Linear(in_features=512, out_features=120)
(6): LogSoftmax()
)
```

- There is no rule of thumb for when earlier or later layers should have different amounts of dropout yet.
- If in doubt, use the same dropout for every fully connected layer.
- Often people only put dropout on the very last linear layer.

Question: Why monitor loss and not accuracy? [23:53] Loss is the only thing that we can see for both the validation set and the training set. As we learn later, the loss is the thing that we are actually optimizing so it is easier to monitor and understand what that means.

Question: Do we need to adjust the learning rate after adding dropouts? [24:33] It does not seem to impact the learning rate enough to notice. In theory, it might but not enough to affect us.

Structured and Time Series Data [25:03]

[Notebook](#) / [Kaggle](#)

	Store	DayOfWeek	Date	Sales	Customers	Open	Promo	StateHoliday	SchoolHoliday
0	1	5	2015-07-31	5263	555	1	1	0	1
1	2	5	2015-07-31	6064	625	1	1	0	1
2	3	5	2015-07-31	8314	821	1	1	0	1
3	4	5	2015-07-31	13995	1498	1	1	0	1
4	5	5	2015-07-31	4822	559	1	1	0	1
	Store	StoreType	Assortment	CompetitionDistance	CompetitionOpenSinceMonth	CompetitionOpenSinceYear	Promo2	Promo2SinceWeek	Promo2SinceYear
0	1	c	a	1270.0		9.0	2008.0	0	NaN
1	2	a	a	570.0		11.0	2007.0	1	13.0
2	3	a	a	14130.0		12.0	2006.0	1	14.0
3	4	c	c	620.0		9.0	2009.0	0	NaN
4	5	a	a	29910.0		4.0	2015.0	0	NaN

There are two types of columns:

- Categorical—It has a number of “levels” e.g. StoreType, Assortment
- Continuous—It has a number where differences or ratios of that numbers have some kind of meanings e.g. CompetitionDistance

```
cat_vars = ['Store', 'DayOfWeek', 'Year', 'Month', 'Day',
           'StateHoliday', 'CompetitionMonthsOpen',
           'Promo2Weeks',
           'StoreType', 'Assortment', 'PromoInterval',
```

```
'CompetitionOpenSinceYear', 'Promo2SinceYear',
'State',
'Week', 'Events', 'Promo_fw', 'Promo_bw',
'StateHoliday_fw', 'StateHoliday_bw',
'SchoolHoliday_fw', 'SchoolHoliday_bw']

contin_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']
```

n = len(joined); n

- Numbers like `Year`, `Month`, although we could treat them as continuous, we do not have to. If we decide to make `Year` a categorical variable, we are telling our neural net that for every different “level” of `Year` (2000, 2001, 2002), you can treat it totally differently; where-else if we say it is continuous, it has to come up with some kind of smooth function to fit them. So often things that actually are continuous but do not have many distinct levels (e.g. `Year`, `DayOfWeek`), it often works better to treat them as categorical.
- Choosing categorical vs. continuous variable is a modeling decision you get to make. In summary, if it is categorical in the data, it has to be categorical. If it is continuous in the data, you get to pick whether to make it continuous or categorical in the model.

- Generally, floating point numbers are hard to make categorical as there are many levels (we call number of levels “**Cardinality**

Question: Do you ever *bin* continuous variables? [31:02] Jeremy does not bin variables but one thing we could do with, say max temperature, is to group into 0–10, 10–20, 20–30, and call that categorical.

Interestingly, a paper just came out last week in which a group of researchers found that sometimes binning can be helpful.

Question: If you are using year as a category, what happens when a model encounters a year it has never seen before? [31:47] We will get there, but the short answer is that it will be treated as an unknown category. Pandas has a special category called unknown and if it sees a category it has not seen before, it gets treated as unknown.

```
for v in cat_vars:  
    joined[v] =  
        joined[v].astype('category').cat.as_ordered()  
  
for v in contin_vars:  
    joined[v] = joined[v].astype('float32')  
  
dep = 'Sales'  
joined = joined[cat_vars+contin_vars+[dep, 'Date']].copy()
```

- Loop through `cat_vars` and turn applicable data frame columns into categorical columns.

- Loop through `contin_vars` and set them as `float32` (32 bit floating point) because that is what PyTorch expects.

Start with a small sample [34:29]

```
idxs = get_cv_idxs(n, val_pct=150000/n)
joined_samp = joined.iloc[idxs].set_index("Date")
samp_size = len(joined_samp); samp_size
```

```
joined_samp.head(2)
```

	Store	DayOfWeek	Year	Month	Day	StateHoliday	CompetitionMonthsOpen	Promo2Weeks	StoreType	Assortment	...	Max_Wind_SpeedKm_h	Mean_I
Date													
2014-01-08	781	3	2014	1	8	False	24	0	a	a	...	29.0	
2014-11-22	626	6	2014	11	22	False	12	0	c	c	...	23.0	

2 rows × 39 columns

Here is what our data looks like. Even though we set some of the columns as “category” (e.g. ‘StoreType’, ‘Year’), Pandas still display as string in the notebook.

```
df, y, nas, mapper = proc_df(joined_samp, 'Sales',
do_scale=True)
```

```
    yl = np.log(y)
```

`proc_df` (process data frame)—A function in Fast.ai that does a few things:

1. Pulls out the dependent variable, puts it into a separate variable, and deletes it from the original data frame. In other words, `df` do not have `Sales` column, and `y` only contains `Sales` column.
2. `do_scale` : Neural nets really like to have the input data to all be somewhere around zero with a standard deviation of somewhere around 1. So we take our data, subtract the mean, and divide by the standard deviation to make that happen. It returns a special object which keeps track of what mean and standard deviation it used for that normalization so you can do the same to the test set later (`mapper`).
3. It also handles missing values—for categorical variable, it becomes ID: 0 and other categories become 1, 2, 3, and so on. For continuous variable, it replaces the missing value with the median and create a new boolean column that says whether it was missing or not.

```
df.head(2)
```

	Store	DayOfWeek	Year	Month	Day	StateHoliday	CompetitionMonthsOpen	Promo2Weeks	StoreType	Assortment	...	Mean_Wind_SpeedKm_h	Cloud
Date													
2014-01-08	781	3	2	1	8	1	25	1	1	1	...	0.367717	1.4
2014-11-22	626	6	2	11	22	1	13	1	3	3	...	0.367717	-0.3

2 rows × 40 columns

After processing, year 2014 for example becomes 2 since categorical variables have been replaced with contiguous integers starting at zero. The reason for that is, we are going to put them into a matrix later, and we would not want the matrix to be 2014 rows long when it could just be two rows.

Now we have a data frame which does not contain the dependent variable and where everything is a number. That is where we need to get to do deep learning. Check out Machine Learning course on further details. Another thing that is covered in Machine Learning course is validation sets. In this case, we need to predict the next two weeks of sales therefore we should create a validation set which is the last two weeks of our training set:

```
val_idx =
np.flatnonzero((df.index<=datetime.datetime(2014,9,17)) &
(df.index>=datetime.datetime(2014,8,1)))
```

- How (and why) to create a good validation set

Let's get straight to the deep learning action [39:48]

For any Kaggle competitions, it is important that you have a strong understanding of your metric—how you are going to be judged. In this competition, we are going to be judged on Root Mean Square Percentage Error (RMSPE).

$$\text{RMSPE} = \sqrt{\frac{1}{n} \sum_{i=1}^n \left(\frac{y_i - \hat{y}_i}{y_i} \right)^2}$$

$$\frac{\sum \left(\frac{\hat{y}}{y} \right)}{n}$$

$$\ln\left(\frac{a'}{b'}\right) = \ln(a') - \ln(b')$$

```
def inv_y(a): return np.exp(a)

def exp_rmspe(y_pred, targ):
    targ = inv_y(targ)
    pct_var = (targ - inv_y(y_pred))/targ
    return math.sqrt((pct_var**2).mean())

max_log_y = np.max(y1)
y_range = (0, max_log_y*1.2)
```

- When you take the log of the data, getting the root mean squared error will actually get you the root mean square percentage error.

```
md = ColumnarModelData.from_data_frame(PATH, val_idx, df,
                                         yl.astype(np.float32), cat flds=cat_vars, bs=128,
                                         test_df=df_test)
```

- As per usual, we will start by creating model data object which has a validation set, training set, and optional test set built into it. From that, we will get a learner, we will then optionally call `lr_find`, then call `learn.fit` and so forth.
- The difference here is we are not using `ImageClassifierData.from_csv` or `.from_paths`, we need a different kind of model data called `ColumnarModelData` and we call `from_data_frame`.
- `PATH` : Specifies where to store model files etc
- `val_idx` : A list of the indexes of the rows that we want to put in the validation set
- `df` : data frame that contains independent variable
- `yl` : We took the dependent variable `y` returned by `proc_df` and took the log of that (i.e. `np.log(y)`)
- `cat flds` : which columns to be treated as categorical. Remember, by this time, everything is a number, so unless we specify, it will treat them all as continuous.

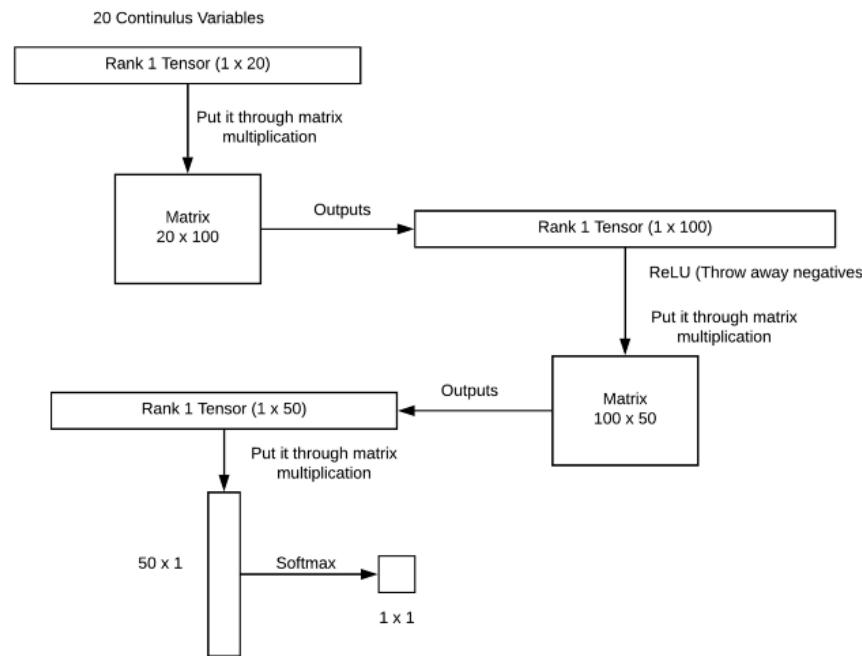
Now we have a standard model data object which we are familiar with and contains `train_dl` , `val_dl` , `train_ds` , `val_ds` , etc.

```
m = md.get_learner(emb_szs, len(df.columns)-len(cat_vars),  
                    0.04, 1, [1000,500], [0.001,0.01],  
                    y_range=y_range)
```

- Here, we are asking it to create a learner that is suitable for our model data.
- `0.04` : how much dropout to use
- `[1000,500]` : how many activations to have in each layer
- `[0.001,0.01]` : how many dropout to use at later layers

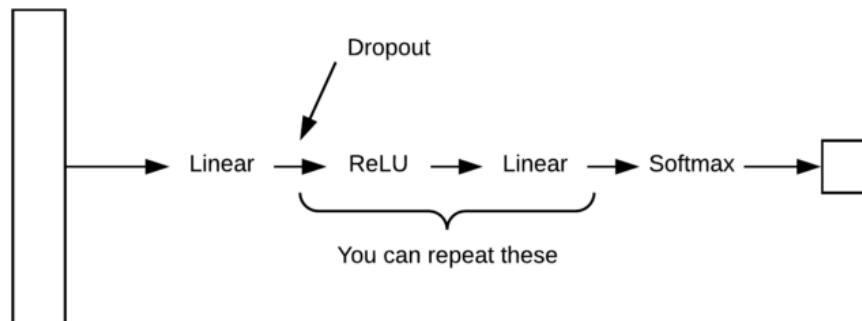
Key New Concept: Embeddings [45:39]

Let's forget about categorical variables for a moment:



Remember, you never want to put ReLU in the last layer because softmax needs negatives to create low probabilities.

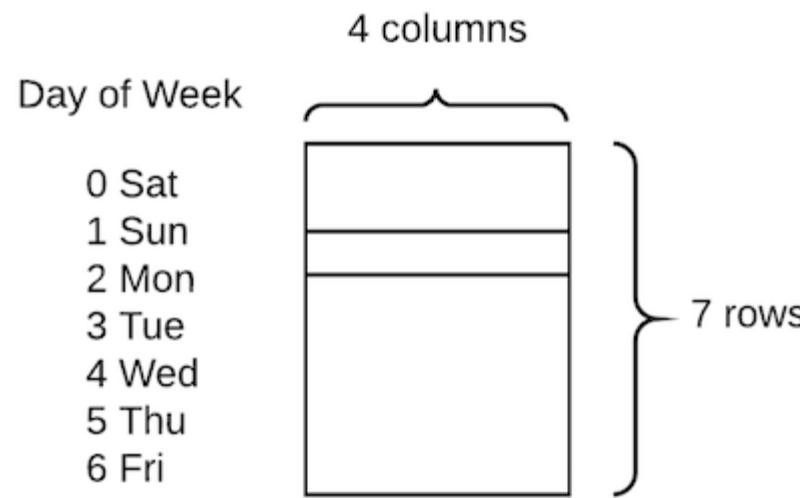
Simple view of fully connected neural net [49:13]:



For regression problems (not classification), you can even skip the softmax layer.

Categorical variables [50:49]

We create a new matrix of 7 rows and as many columns as we choose (4, for example) and fill it with floating numbers. To add “Sunday” to our rank 1 tensor with continuous variables, we do a look up to this matrix, which will return 4 floating numbers, and we use them as “Sunday”.



Initially, these numbers are random. But we can put them through a neural net and update them in a way that reduces the loss. In other words, this matrix is just another bunch of weights in our neural net. And matrices of this type are called “**embedding matrices**”. An embedding matrix is something where we start out with an integer between zero and maximum number of levels of that category. We

index into the matrix to find a particular row, and we append it to all of our continuous variables, and everything after that is just the same as before (linear → ReLU → etc).

Question: What do those 4 numbers represent? [55:12] We will learn more about that when we look at collaborative filtering, but for now, they are just parameters that we are learning that happen to end up giving us a good loss. We will discover later that these particular parameters often are human interpretable and quite interesting but that a side effect.

Question: Do you have good heuristics for the dimensionality of the embedding matrix? [55:57] I sure do! Let's take a look.

```
cat_sz = [(c, len(joined_samp[c].cat.categories)+1)
           for c in cat_vars]
cat_sz

[('Store', 1116),
 ('DayOfWeek', 8),
 ('Year', 4),
 ('Month', 13),
 ('Day', 32),
 ('StateHoliday', 3),
 ('CompetitionMonthsOpen', 26),
 ('Promo2Weeks', 27),
 ('StoreType', 5),
 ('Assortment', 4),
 ('PromoInterval', 4),
 ('CompetitionOpenSinceYear', 24),
 ('Promo2SinceYear', 9),
 ('State', 13),
 ('Week', 53),
 ('Events', 22),
 ('Promo_fw', 7),
```

```
('Promo_bw', 7),  
('StateHoliday_fw', 4),  
('StateHoliday_bw', 4),  
('SchoolHoliday_fw', 9),  
('SchoolHoliday_bw', 9)]
```

- Here is a list of every categorical variable and its cardinality.
- Even if there were no missing values in the original data, you should still set aside one for unknown just in case.
- The rule of thumb for determining the embedding size is the cardinality size divided by 2, but no bigger than 50.

```
emb_szs = [(c, min(50, (c+1)//2)) for _,c in cat_sz]  
emb_szs  
  
[(1116, 50),  
(8, 4),  
(4, 2),  
(13, 7),  
(32, 16),  
(3, 2),  
(26, 13),  
(27, 14),  
(5, 3),  
(4, 2),  
(4, 2),  
(24, 12),  
(9, 5),  
(13, 7),  
(53, 27),  
(22, 11),  
(7, 4),  
(7, 4),  
(4, 2),  
(4, 2),
```

```
(9, 5),  
(9, 5)]
```

Then pass the embedding size to the learner:

```
m = md.get_learner(emb_szs, len(df.columns)-len(cat_vars),  
0.04, 1,  
[1000,500], [0.001,0.01],  
y_range=y_range)
```

Question: Is there a way to initialize embedding matrices besides random? [58:14] We will probably talk about pre-trained more later in the course, but the basic idea is if somebody else at Rossmann had already trained a neural network to predict cheese sales, you may as well start with their embedding matrix of stores to predict liquor sales. This is what happens, for example, at Pinterest and Instacart. Instacart uses this technique for routing their shoppers, and Pinterest uses it for deciding what to display on a webpage. They have embedding matrices of products/stores that get shared in the organization so people do not have to train new ones.

Question: What is the advantage of using embedding matrices over one-hot-encoding? [59:23] For the day of week example above, instead of the 4 numbers, we could have easily passed 7 numbers (e.g. [0, 1, 0, 0, 0, 0, 0] for Sunday). That also is a list of floats and that would totally work—and that is how, generally speaking, categorical variables have been used in statistics for many years (called “dummy variables”). The problem is, the concept of Sunday could only ever be associated with a

single floating-point number. So it gets this kind of linear behavior—it says Sunday is more or less of a single thing. With embeddings, Sunday is a concept in four dimensional space. What we tend to find happen is that these embedding vectors tend to get these rich semantic concepts. For example, if it turns out that weekends have a different behavior, you tend to see that Saturday and Sunday will have some particular number higher.

By having higher dimensionality vector rather than just a single number, it gives the deep learning network a chance to learn these rich representations.

The idea of an embedding is what is called a “distributed representation”—the most fundamental concept of neural networks. This is the idea that a concept in neural network has a high dimensional representation which can be hard to interpret. These numbers in this vector does not even have to have just one meaning. It could mean one thing if this is low and that one is high, and something else if that one is high and that one is low because it is going through this rich nonlinear function. It is this rich representation that allows it to learn such interesting relationships.

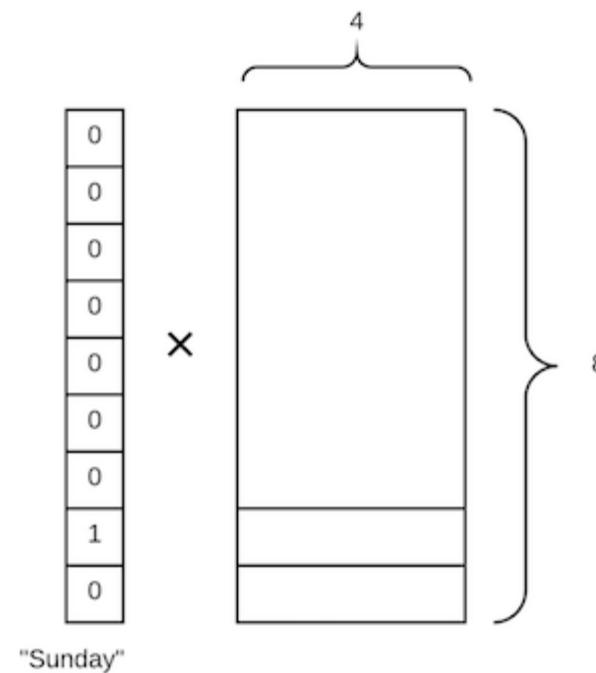
Question: Are embeddings suitable for certain types of variables?

[01:02:45] Embedding is suitable for any categorical variables. The only thing it cannot work well for would be something with too high cardinality. If you had 600,000 rows and a variable had 600,000 levels, that is just not a useful categorical variable. But in general, the third winner in this competition really decided that everything that was not too high cardinality, they put them all as categorical. The good rule of

thumb is if you can make a categorical variable, you may as well because that way it can learn this rich distributed representation; where else if you leave it as continuous, the most it can do is to try and find a single functional form that fits it well.

Matrix algebra behind the scene [01:04:47]

Looking up an embedding with an index is identical to doing a matrix product between a one-hot encoded vector and the embedding matrix. But doing so is terribly inefficient, so modern libraries implement this as taking an integer and doing a look up into an array.



Question: Could you touch on using dates and times as categorical and how that affects seasonality? [01:06:59] There is a Fast.ai function called `add_datepart` which takes a data frame and a column name. It optionally removes the column from the data frame and replaces it with lots of columns representing all of the useful information about that date such as day of week, day of month, month of year, etc (basically everything Pandas gives us).

```
add_datepart(weather, "Date", drop=False)
add_datepart(googletrend, "Date", drop=False)
add_datepart(train, "Date", drop=False)
add_datepart(test, "Date", drop=False)
```

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
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

So for example, day of week now becomes eight rows by four columns embedding matrix. Conceptually this allows our model to create some interesting time series models. If there is something that has a seven day period cycle that goes up on Mondays and down on Wednesdays but only for daily and only in Berlin, it can totally do that—it has all the information it needs. This is a fantastic way to deal with time series. You just need to make sure that the cycle indicator in your time series

exists as a column. If you did not have a column called day of week, it would be very difficult for the neural network to learn to do mod seven and look up in an embedding matrix. It is not impossible but really hard. If you are predicting sales of beverages in San Francisco, you probably want a list of when the ball game is on at AT&T park because that is going to impact how many people are drinking beer in SoMa. So you need to make sure that the basic indicators or periodicity is in your data, and as long as they are there, neural net is going to learn to use them.

Learner [01:10:13]

```
m = md.get_learner(emb_szs, len(df.columns)-len(cat_vars),  
0.04, 1,  
[1000,500], [0.001,0.01],  
y_range=y_range)  
lr = 1e-3
```

- `emb_szs` : embedding size
- `len(df.columns)-len(cat_vars)` : number of continuous variables in the data frame
- `0.04` : embedding matrix has its own dropout and this is the dropout rate
- `1` : how many outputs we want to create (output of the last linear layer)

- `[1000, 500]` : number of activations in the first linear layer, and the second linear layer
- `[0.001, 0.01]` : dropout in the first linear layer, and the second linear layer
- `y_range` : we will not worry about that for now

A screenshot of a Jupyter Notebook cell. The code is:

```
m.fit(lr, 3, metrics=[exp_rmspe])
```

The output is:

A Jupyter Widget

```
[ 0.        0.02479  0.02205  0.19309]
[ 1.        0.02044  0.01751  0.18301]
[ 2.        0.01598  0.01571  0.17248]
```

Below the output is a horizontal scroll bar.

- `metrics` : this is a custom metric which specifies a function to be called at the end of every epoch and prints out a result

A screenshot of a Jupyter Notebook cell. The code is:

```
m.fit(lr, 1, metrics=[exp_rmspe], cycle_len=1)
```

The output is:

```
[ 0.        0.00676  0.01041  0.09711]
```

By using all of the training data, we achieved a RMSPE around 0.09711. There is a big difference between public leader board and

private leader board, but we are certainly in the top end of this competition.

So this is a technique for dealing with time series and structured data. Interestingly, compared to the group that used this technique ([Entity Embeddings of Categorical Variables](#)), the second place winner did way more feature engineering. The winners of this competition were actually subject matter experts in logistics sales forecasting so they had their own code to create lots and lots of features. Folks at Pinterest who build a very similar model for recommendations also said that when they switched from gradient boosting machines to deep learning, they did way less feature engineering and it was much simpler model which requires less maintenance. So this is one of the big benefits of using this approach to deep learning—you can get state of the art results but with a lot less work.

Question: Are we using any time series in any of these? [01:15:01]

Indirectly, yes. As we just saw, we have a day of week, month of year, etc in our columns and most of them are being treated as categories, so we are building a distributed representation of January, Sunday, and so on. We are not using any classic time series techniques, all we are doing is true fully connected layers in a neural net. The embedding matrix is able to deal with things like day of week periodicity in a much richer way than any standard time series techniques.

Question regarding the difference between image models and this model [01:15:59]: There is a difference in a way we are calling `get_learner`. In imaging we just did `Learner.trained` and pass the data:

```
learn = ConvLearner.pretrained(arch, data, ps=0.,
                               precompute=True)
```

For these kinds of models, in fact for a lot of the models, the model we build depends on the data. In this case, we need to know what embedding matrices we have. So in this case, the data objects creates the learner (upside down to what we have seen before):

```
m = md.get_learner(emb_szs, len(df.columns)-len(cat_vars),
                    0.04, 1,
                    [1000,500], [0.001,0.01],
                    y_range=y_range)
```

Summary of steps (if you want to use this for your own dataset)
[01:17:56]:

Step 1. List categorical variable names, and list continuous variable names, and put them in a Pandas data frame

Step 2. Create a list of which row indexes you want in your validation set

Step 3. Call this exact line of code:

```
md = ColumnarModelData.from_data_frame(PATH, val_idx, df,
                                         yl.astype(np.float32), cat_flds=cat_vars, bs=128,
                                         test_df=df_test)
```

Step 4. Create a list of how big you want each embedding matrix to be

Step 5. Call `get_learner` —you can use these exact parameters to start with:

```
m = md.get_learner(emb_szs, len(df.columns)-len(cat_vars),  
0.04, 1,  
[1000,500], [0.001,0.01],  
y_range=y_range)
```

Step 6. Call `m.fit`

Question: How to use data augmentation for this type of data, and how does dropout work? [\[01:18:59\]](#) No idea. Jeremy thinks it has to be domain-specific, but he has never seen any paper or anybody in industry doing data augmentation with structured data and deep learning. He thinks it can be done but has not seen it done. What dropout is doing is exactly the same as before.

Question: What is the downside? Almost no one is using this. Why not? [\[01:20:41\]](#) Basically the answer is as we discussed before, no one in academia almost is working on this because it is not something that people publish on. As a result, there have not been really great examples people could look at and say “oh here is a technique that works well so let’s have our company implement it”. But perhaps equally importantly, until now with this Fast.ai library, there has not been any way to do it conveniently. If you wanted to implement one of

these models, you had to write all the custom code yourself. There are a lot of big commercial and scientific opportunity to use this and solve problems that previously haven't been solved very well.

Natural Language Processing [01:23:37]

The most up-and-coming area of deep learning and it is two or three years behind computer vision. The state of software and some of the concepts is much less mature than it is for computer vision. One of the things you find in NLP is there are particular problems you can solve and they have particular names. There is a particular kind of problem in NLP called “language modeling” and it has a very specific definition —it means build a model where given a few words of a sentence, can you predict what the next word is going to be.

Language Modeling [01:25:48]

Notebook

Here we have 18 months worth of papers from arXiv (arXiv.org) and this is an example:

```
' '.join(md.trn_ds[0].text[:150])  
  
'<cat> csni <summ> the exploitation of mm - wave bands is  
one of the key - enabler for 5 g mobile \n radio networks .  
however , the introduction of mm - wave technologies in  
cellular \n networks is not straightforward due to harsh  
propagation conditions that limit \n the mm - wave access  
availability . mm - wave technologies require high - gain  
antenna \n systems to compensate for high path loss and  
limited power . as a consequence , \n directional
```

transmissions must be used for cell discovery and synchronization \n processes : this can lead to a non - negligible access delay caused by the \n exploration of the cell area with multiple transmissions along different \n directions . \n the integration of mm - wave technologies and conventional wireless access \n networks with the objective of speeding up the cell search process requires new \n '

- <cat> —category of the paper. CSNI is Computer Science and Networking
- <summ> —abstract of the paper

Here are what the output of a trained language model looks like. We did simple little tests in which you pass some priming text and see what the model thinks should come next:

```
sample_model(m, "<CAT> csni <SUMM> algorithms that")  
  
...use the same network as a single node are not able to  
achieve the same performance as the traditional network -  
based routing algorithms . in this paper , we propose a  
novel routing scheme for routing protocols in wireless  
networks . the proposed scheme is based ...
```

It learned by reading arXiv papers that somebody who is writing about computer networking would talk like this. Remember, it started out not knowing English at all. It started out with an embedding matrix for every word in English that was random. By reading lots of arXiv papers, it learned what kind of words followed others.

Here we tried specifying a category to be computer vision:

```
sample_model(m, "<CAT> cscv <SUMM> algorithms that")  
  
...use the same data to perform image classification are  
increasingly being used to improve the performance of image  
classification algorithms . in this paper , we propose a  
novel method for image classification using a deep  
convolutional neural network ( cnn ) . the proposed method  
is ...
```

It not only learned how to write English pretty well, but also after you say something like “convolutional neural network” you should then use parenthesis to specify an acronym “(CNN)”.

```
sample_model(m,"<CAT> cscv <SUMM> algorithms. <TITLE> on ")  
  
...the performance of deep learning for image classification  
<eos>  
  
sample_model(m,"<CAT> csni <SUMM> algorithms. <TITLE> on ")  
  
...the performance of wireless networks <eos>  
  
sample_model(m,"<CAT> cscv <SUMM> algorithms. <TITLE>  
towards ")  
  
...a new approach to image classification <eos>  
  
sample_model(m,"<CAT> csni <SUMM> algorithms. <TITLE>  
towards ")
```

...a new approach to the analysis of wireless networks <eos>

A language model can be incredibly deep and subtle, so we are going to try and build that—not because we care about this at all, but because we are trying to create a pre-trained model which is used to do some other tasks. For example, given an IMDB movie review, we will figure out whether they are positive or negative. It is a lot like cats vs. dogs—a classification problem. So we would really like to use a pre-trained network which at least knows how to read English. So we will train a model that predicts a next word of a sentence (i.e. language model), and just like in computer vision, stick some new layers on the end and ask it to predict whether something is positive or negative.

IMDB [1:31:11]

Notebook

What we are going to do is to train a language model, making that the pre-trained model for a classification model. In other words, we are trying to leverage exactly what we learned in our computer vision which is how to do fine-tuning to create powerful classification models.

Question: why would doing directly what you want to do not work?

[01:31:34] It just turns out it doesn't empirically. There are several reasons. First of all, we know fine-tuning a pre-trained network is really powerful. So if we can get it to learn some related tasks first, then we can use all that information to try and help it on the second task. The other is IMDB movie reviews are up to a thousands words long. So after reading a thousands words knowing nothing about how English is

structured or concept of a word or punctuation, all you get is a 1 or a 0 (positive or negative). Trying to learn the entire structure of English and then how it expresses positive and negative sentiments from a single number is just too much to expect.

Question: Is this similar to Char-RNN by Karpathy? [01:33:09] This is somewhat similar to Char-RNN which predicts the next letter given a number of previous letters. Language model generally work at a word level (but they do not have to), and we will focus on word level modeling in this course.

Question: To what extent are these generated words/sentences actual copies of what it found in the training set? [01:33:44] Words are definitely words it has seen before because it is not a character level so it can only give us the word it has seen before. Sentences, there are rigorous ways of doing it but the easiest would be by looking at examples like above, you get a sense of it. Most importantly, when we train the language model, we will have a validation set so that we are trying to predict the next word of something that has never seen before. There are tricks to using language models to generate text like beam search.

Use cases of text classification:

- For hedge fund, identify things in articles or Twitter that caused massive market drops in the past.
- Identify customer service queries which tend to be associated with people who cancel their contracts in the next month

- Organize documents into whether they are part of legal discovery or not.

```
from fastai.learner import *

import torchtext
from torchtext import vocab, data
from torchtext.datasets import language_modeling

from fastai.rnn_reg import *
from fastai.rnn_train import *
from fastai.nlp import *
from fastai.lm_rnn import *

import dill as pickle
```

- `torchtext` —PyTorch's NLP library

Data [01:37:05]

IMDB [Large Movie Review Dataset](#)

```
PATH = 'data/aclImdb/'

TRN_PATH = 'train/all/'
VAL_PATH = 'test/all/'
TRN = f'{PATH}{TRN_PATH}'
VAL = f'{PATH}{VAL_PATH}'

%ls {PATH}
```

```
imdbEr.txt  imdb.vocab  models/  README  test/  tmp/  train/
```

We do not have separate test and validation in this case. Just like in vision, the training directory has bunch of files in it:

```
trn_files = !ls {TRN}
trn_files[:10]

['0_0.txt',
 '0_3.txt',
 '0_9.txt',
 '10000_0.txt',
 '10000_4.txt',
 '10000_8.txt',
 '1000_0.txt',
 '10001_0.txt',
 '10001_10.txt',
 '10001_4.txt']

review = !cat {TRN}{trn_files[6]}
review[0]

"I have to say when a name like Zombiegeddon and an atom bomb on the front cover I was expecting a flat out chopsocky fung-ku, but what I got instead was a comedy. So, it wasn't quite what I was expecting, but I really liked it anyway! The best scene ever was the main cop dude pulling those kids over and pulling a Bad Lieutenant on them!! I was laughing my ass off. I mean, the cops were just so bad! And when I say bad, I mean The Shield Vic Macky bad. But unlike that show I was laughing when they shot people and smoked dope.<br /><br />Felissa Rose...man, oh man. What can you say about that hottie. She was great and put those other actresses to shame. She should work more often!!!! I also really liked the fight scene outside of the building. That was done really well. Lots of fighting and people getting their heads banged up. FUN! Last, but not least Joe Estevez
```

*and William Smith were great as the...well, I wasn't sure what they were, but they seemed to be having fun and throwing out lines. I mean, some of it didn't make sense with the rest of the flick, but who cares when you're laughing so hard! All in all the film wasn't the greatest thing since sliced bread, but I wasn't expecting that. It was a Troma flick so I figured it would totally suck. It's nice when something surprises you but not totally sucking.

Rent it if you want to get stoned on a Friday night and laugh with your buddies. Don't rent it if you are an uptight weenie or want a zombie movie with lots of flesh eating.

P.S. Uwe Boil was a nice touch."*

Now we will check how many words are in the dataset:

```
!find {TRN} -name '*.txt' | xargs cat | wc -w
```

```
17486581
```

```
!find {VAL} -name '*.txt' | xargs cat | wc -w
```

```
5686719
```

Before we can do anything with text, we have to turn it into a list of tokens. Token is basically like a word. Eventually we will turn them into a list of numbers, but the first step is to turn it into a list of words—this is called “tokenization” in NLP. A good tokenizer will do a good job of recognizing pieces in your sentence. Each separated piece of punctuation will be separated, and each part of multi-part word will be separated as appropriate. Spacy does a lot of NLP stuff, and it has the

best tokenizer Jeremy knows. So Fast.ai library is designed to work well with the Spacey tokenizer as with torchtext.

Creating a field [01:41:01]

A field is a definition of how to pre-process some text.

```
TEXT = data.Field(lower=True, tokenize=spacy_tok)
```

- `lower=True` —lowercase the text
- `tokenize=spacy_tok` —tokenize with `spacy_tok`

Now we create the usual Fast.ai model data object:

```
bs=64; bptt=70

FILES = dict(train=TRN_PATH, validation=VAL_PATH,
test=VAL_PATH)
md = LanguageModelData.from_text_files(PATH, TEXT, **FILES,
bs=bs,
                                bptt=bptt,
min_freq=10)
```

- `PATH` : as per usual where the data is, where to save models, etc
- `TEXT` : torchtext's Field definition

- `**FILES` : list of all of the files we have: training, validation, and test (to keep things simple, we do not have a separate validation and test set, so both points to validation folder)
- `bs` : batch size
- `bptt` : Back Prop Through Time. It means how long a sentence we will stick on the GPU at once
- `min_freq=10` : In a moment, we are going to be replacing words with integers (a unique index for every word). If there are any words that occur less than 10 times, just call it unknown.

After building our `ModelData` object, it automatically fills the `TEXT` object with a very important attribute: `TEXT.vocab`. This is a *vocabulary*, which stores which unique words (or *tokens*) have been seen in the text, and how each word will be mapped to a unique integer id.

```
# 'itos': 'int-to-string'
TEXT.vocab.itos[:12]

['<unk>', '<pad>', 'the', ',', '.', 'and', 'a', 'of', 'to',
 'is', 'it', 'in']

# 'stoi': 'string to int'
TEXT.vocab.stoi['the']
```

2

`itos` is sorted by frequency except for the first two special ones. Using `vocab`, torchtext will turn words into integer IDs for us :

```
md.trn_ds[0].text[:12]

['i',
 'have',
 'always',
 'loved',
 'this',
 'story',
 '-',
 'the',
 'hopeful',
 'theme',
 ',',
 'the']

TEXT.numericalize([md.trn_ds[0].text[:12]])

Variable containing:
12
35
227
480
13
76
17
2
7319
769
3
2
[torch.cuda.LongTensor of size 12x1 (GPU 0)]
```

Question: Is it common to do any stemming or lemma-tizing?

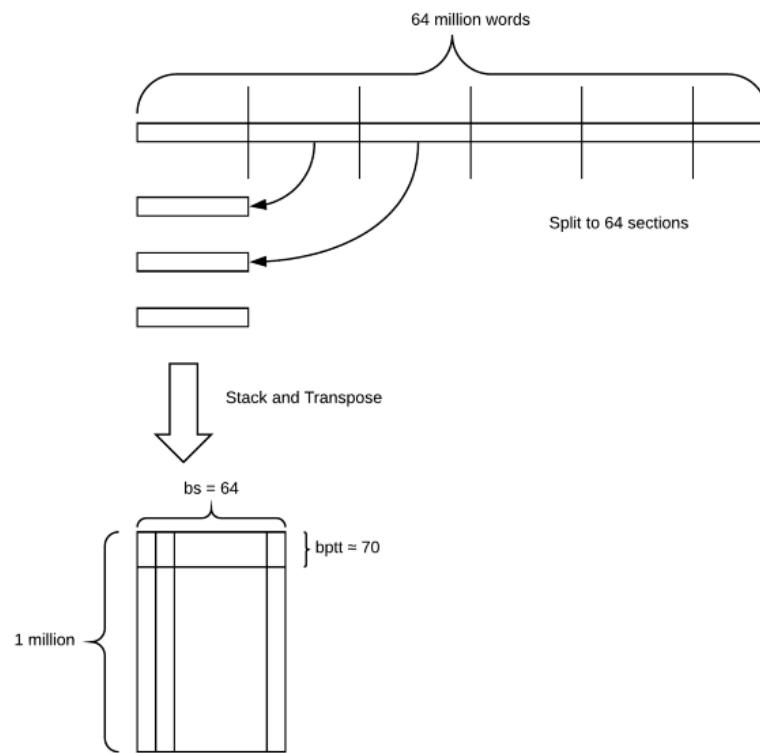
[01:45:47] Not really, no. Generally tokenization is what we want. To keep it as general as possible, we want to know what is coming next so whether it is future tense or past tense or plural or singular, we don't really know which things are going to be interesting and which are not, so it seems that it is generally best to leave it alone as much as possible.

Question: When dealing with natural language, isn't context important? Why are we tokenizing and looking at individual word?

[01:46:38] No, we are not looking at individual word—they are still in order. Just because we replaced I with a number 12, they are still in that order. There is a different way of dealing with natural language called “bag of words” and they do throw away the order and context. In the Machine Learning course, we will be learning about working with bag of words representations but my belief is that they are no longer useful or in the verge of becoming no longer useful. We are starting to learn how to use deep learning to use context properly.

Batch size and BPTT [01:47:40]

What happens in a language model is even though we have lots of movie reviews, they all get concatenated together into one big block of text. So we predict the next word in this huge long thing which is all of the IMDB movie reviews concatenated together.



- We split up the concatenated reviews into batches. In this case, we will split it to 64 sections
- We then move each section underneath the previous one, and transpose it.
- We end up with a matrix which is 1 million by 64.
- We then grab a little chunk at time and those chunk lengths are **approximately** equal to BPTT. Here, we grab a little 70 long

section and that is the first thing we chuck into our GPU (i.e. the batch).

```
next(iter(md.trn_dl))

(Variable containing:
 12      567      3   ...    2118      4    2399
35      7      33   ...        6    148      55
 227     103     533   ...    4892      31     10
  ...
 19     8879     33   ...      41      24    733
 552    8250     57   ...    219      57   1777
 5       19      2   ...    3099      8     48
[torch.cuda.LongTensor of size 75x64 (GPU 0)], Variable
containing:
35
7
33
:
 22
 3885
 21587
[torch.cuda.LongTensor of size 4800 (GPU 0)])
```

- We grab our first training batch by wrapping data loader with `iter` then calling `next`.
- We got back a 75 by 64 tensor (approximately 70 rows but not exactly)
- A neat trick torchtext does is to randomly change the `bptt` number every time so each epoch it is getting slightly different bits of text—similar to shuffling images in computer vision. We cannot

randomly shuffle the words because they need to be in the right order, so instead, we randomly move their breakpoints a little bit.

- The target value is also 75 by 64 but for minor technical reasons it is flattened out into a single vector.

Question: Why not split by a sentence? [01:53:40] Not really.

Remember, we are using columns. So each of our column is of length about 1 million, so although it is true that those columns are not always exactly finishing on a full stop, they are so darn long we do not care. Each column contains multiple sentences.

Pertaining to this question, Jeremy found what is in this language model matrix a little mind-bending for quite a while, so do not worry if it takes a while and you have to ask a thousands questions.

Create a model [01:55:46]

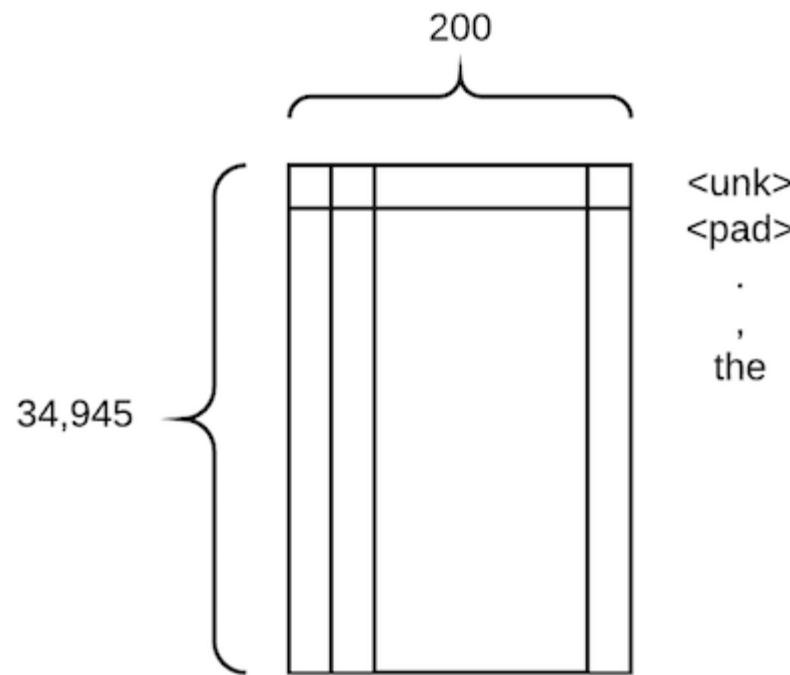
Now that we have a model data object that can feed us batches, we can create a model. First, we are going to create an embedding matrix.

Here are the: # batches; # unique tokens in the vocab; length of the dataset; # of words

```
len(md.trn_dl), md.nt, len(md.trn_ds),
len(md.trn_ds[0].text)

(4602, 34945, 1, 20621966)
```

This is our embedding matrix looks like:



- It is a high cardinality categorical variable and furthermore, it is the only variable—this is typical in NLP
- The embedding size is 200 which is much bigger than our previous embedding vectors. Not surprising because a word has a lot more nuance to it than the concept of Sunday. **Generally, an embedding size for a word will be somewhere between 50 and 600.**

```
em_sz = 200 # size of each embedding vector
nh = 500     # number of hidden activations per layer
nl = 3       # number of layers
```

Researchers have found that large amounts of *momentum* (which we'll learn about later) don't work well with these kinds of *RNN* models, so we create a version of the *Adam* optimizer with less momentum than its default of `0.9`. Any time you are doing NLP, you should probably include this line:

```
opt_fn = partial(optim.Adam, betas=(0.7, 0.99))
```

Fast.ai uses a variant of the state of the art AWD LSTM Language Model developed by Stephen Merity. A key feature of this model is that it provides excellent regularization through Dropout. There is no simple way known (yet!) to find the best values of the dropout parameters below—you just have to experiment...

However, the other parameters (`alpha`, `beta`, and `clip`) shouldn't generally need tuning.

```
learner = md.get_model(opt_fn, em_sz, nh, nl, dropouti=0.05,
                      dropout=0.05, wdrop=0.1,
                      dropoute=0.02,
                      dropouth=0.05)
learner.reg_fn = partial(seq2seq_reg, alpha=2, beta=1)
learner.clip=0.3
```

- In the last lecture, we will learn what the architecture is and what all these dropouts are. For now, just know it is the same as per usual, if you try to build an NLP model and you are under-fitting, then decrease all these dropouts, if overfitting, then increase all these dropouts in roughly this ratio. Since this is such a recent paper so there is not a lot of guidance but these ratios worked well —it is what Stephen has been using as well.
- There is another kind of way we can avoid overfitting that we will talk about in the last class. For now, `learner.reg_fn = partial(seq2seq_reg, alpha=2, beta=1)` works reliably so all of your NLP models probably want this particular line.
- `learner.clip=0.3` : when you look at your gradients and you multiply them by the learning rate to decide how much to update your weights by, this will not allow them be more than 0.3. This is a cool little trick to prevent us from taking too big of a step.
- Details do not matter too much right now, so you can use them as they are.

Question: There are word embedding out there such as Word2vec or GloVe. How are they different from this? And why not initialize the weights with those initially? [02:02:29] People have pre-trained these embedding matrices before to do various other tasks. They are not called pre-trained models; they are just a pre-trained embedding matrix and you can download them. There is no reason we could not download them. I found that building a whole pre-trained model in this way did not seem to benefit much if at all from using pre-trained word

vectors; where else using a whole pre-trained language model made a much bigger difference. Maybe we can combine both to make them a little better still.

Question: What is the architecture of the model? [02:03:55] We will be learning about the model architecture in the last lesson but for now, it is a recurrent neural network using something called LSTM (Long Short Term Memory).

Fitting [02:04:24]

```
learner.fit(3e-3, 4, wds=1e-6, cycle_len=1, cycle_mult=2)

learner.save_encoder('adam1_enc')

learner.fit(3e-3, 4, wds=1e-6, cycle_len=10,
            cycle_save_name='adam3_10')

learner.save_encoder('adam3_10_enc')

learner.fit(3e-3, 1, wds=1e-6, cycle_len=20,
            cycle_save_name='adam3_20')

learner.load_cycle('adam3_20', 0)
```

In the sentiment analysis section, we'll just need half of the language model - the *encoder*, so we save that part.

```
learner.save_encoder('adam3_20_enc')

learner.load_encoder('adam3_20_enc')
```

Language modeling accuracy is generally measured using the metric *perplexity*, which is simply `exp()` of the loss function we used.

```
math.exp(4.165)

64.3926824434624

pickle.dump(TEXT, open(f'{PATH}models/TEXT.pkl', 'wb'))
```

Testing [02:04:53]

We can play around with our language model a bit to check it seems to be working OK. First, let's create a short bit of text to 'prime' a set of predictions. We'll use our torchtext field to numericalize it so we can feed it to our language model.

```
m=learner.model
ss=""". So, it wasn't quite what I was expecting, but I
really liked it anyway! The best"""

s = [spacy_tok(ss)]
t=TEXT.numericalize(s)
' '.join(s[0])
```

". So , it was n't quite was I was expecting , but I really liked it anyway ! The best"

We haven't yet added methods to make it easy to test a language model, so we'll need to manually go through the steps.

```
# Set batch size to 1
m[0].bs=1
# Turn off dropout
m.eval()
# Reset hidden state
m.reset()
# Get predictions from model
res,*_ = m(t)
# Put the batch size back to what it was
m[0].bs=bs
```

Let's see what the top 10 predictions were for the next word after our short text:

```
nexts = torch.topk(res[-1], 10)[1]
[TEXT.vocab.itos[o] for o in to_np(nexts)]  
  
['film',
 'movie',
 'of',
 'thing',
 'part',
 '<unk>',
 'performance',
 'scene',
```

```
', ',  
'actor']
```

...and let's see if our model can generate a bit more text all by itself!

```
print(ss, "\n")
for i in range(50):
    n=res[-1].topk(2)[1]
    n = n[1] if n.data[0]==0 else n[0]
    print(TEXT.vocab.itos[n.data[0]], end=' ')
    res,*_ = m(n[0].unsqueeze(0))
print('...')

. So, it wasn't quite was I was expecting, but I really
liked it anyway! The best

film ever ! <eos> i saw this movie at the toronto
international film festival . i was very impressed . i was
very impressed with the acting . i was very impressed with
the acting . i was surprised to see that the actors were not
in the movie . ...
```

Sentiment [02:05:09]

So we had pre-trained a language model and now we want to fine-tune it to do sentiment classification.

To use a pre-trained model, we will need to the saved vocab from the language model, since we need to ensure the same words map to the same IDs.

```
TEXT = pickle.load(open(f'{PATH}models/TEXT.pkl', 'rb'))
```

`sequential=False` tells torchtext that a text field should be tokenized (in this case, we just want to store the 'positive' or 'negative' single label).

```
IMDB_LABEL = data.Field(sequential=False)
```

This time, we need to not treat the whole thing as one big piece of text but every review is separate because each one has a different sentiment attached to it.

`splits` is a torchtext method that creates train, test, and validation sets. The IMDB dataset is built into torchtext, so we can take advantage of that. Take a look at `lang_model-archiv.ipynb` to see how to define your own fastai/torchtext datasets.

```
splits = torchtext.datasets.IMDB.splits(TEXT, IMDB_LABEL,
'data/')

t = splits[0].examples[0]

t.label, ' '.join(t.text[:16])

('pos', 'ashanti is a very 70s sort of film ( 1979 , to be
precise ) .')
```

fastai can create a `ModelData` object directly from torchtext `splits`.

```
md2 = TextData.from_splits(PATH, splits, bs)
```

Now you can go ahead and call `get_model` that gets us our learner.
Then we can load into it the pre-trained language model
(`load_encoder`).

```
m3 = md2.get_model(opt_fn, 1500, bptt, emb_sz=em_sz,  
n_hid=nh,  
n_layers=nl, dropout=0.1, dropouti=0.4,  
wdrop=0.5, dropoute=0.05, dropouth=0.3)  
  
m3.reg_fn = partial(seq2seq_reg, alpha=2, beta=1)  
  
m3.load_encoder(f'adam3_20_enc')
```

Because we're fine-tuning a pretrained model, we'll use differential learning rates, and also increase the max gradient for clipping, to allow the SGDR to work better.

```
m3.clip=25.  
lrs=np.array([1e-4,1e-3,1e-2])
```

```
m3.freeze_to(-1)
m3.fit(lrs/2, 1, metrics=[accuracy])
m3.unfreeze()
m3.fit(lrs, 1, metrics=[accuracy], cycle_len=1)

[ 0.        0.45074  0.28424  0.88458]

[ 0.        0.29202  0.19023  0.92768]
```

We make sure all except the last layer is frozen. Then we train a bit, unfreeze it, train it a bit. The nice thing is once you have got a pre-trained language model, it actually trains really fast.

```
m3.fit(lrs, 7, metrics=[accuracy], cycle_len=2,
       cycle_save_name='imdb2')

[ 0.        0.29053  0.18292  0.93241]
[ 1.        0.24058  0.18233  0.93313]
[ 2.        0.24244  0.17261  0.93714]
[ 3.        0.21166  0.17143  0.93866]
[ 4.        0.2062   0.17143  0.94042]
[ 5.        0.18951  0.16591  0.94083]
[ 6.        0.20527  0.16631  0.9393 ]
[ 7.        0.17372  0.16162  0.94159]
[ 8.        0.17434  0.17213  0.94063]
[ 9.        0.16285  0.16073  0.94311]
[ 10.       0.16327   0.17851   0.93998]
[ 11.       0.15795   0.16042   0.94267]
[ 12.       0.1602    0.16015   0.94199]
[ 13.       0.15503   0.1624    0.94171]
```

```
m3.load_cycle('imdb2', 4)
```

```
accuracy(*m3.predict_with_targs())
```

0.94310897435897434

A recent paper from Bradbury et al, Learned in translation: [contextualized word vectors](#), has a handy summary of the latest academic research in solving this IMDB sentiment analysis problem. Many of the latest algorithms shown are tuned for this specific problem.

IMDb		
	<i>BCN+Char+Cove (Ours)</i>	91.8
	SA-LSTM [Dai and Le, 2015]	92.8
	bmLSTM [Radford et al., 2017]	92.9
	TRNN [Dieng et al., 2016]	93.8
	oh-LSTM [Johnson and Zhang, 2016]	94.1
	Virtual [Miyato et al., 2017]	94.1

As you see, we just got a new state of the art result in sentiment analysis, decreasing the error from 5.9% to 5.5%! You should be able to get similarly world-class results on other NLP classification problems using the same basic steps.

There are many opportunities to further improve this, although we won't be able to get to them until part 2 of this course.

- For example we could start training language models that look at lots of medical journals and then we could make a downloadable

medical language model that then anybody could use to fine-tune on a prostate cancer subset of medical literature.

- We could also combine this with pre-trained word vectors
- We could have pre-trained a Wikipedia corpus language model and then fine-tuned it into an IMDB language model, and then fine-tune that into an IMDB sentiment analysis model and we would have gotten something better than this.

There is a really fantastic researcher called Sebastian Ruder who is the only NLP researcher who has been really writing a lot about pre-training, fine-tuning, and transfer learning in NLP. Jeremy was asking him why this is not happening more, and his view was it is because there is not a software to make it easy. Hopefully Fast.ai will change that.

Collaborative Filtering Introduction [02:11:38]

Notebook

Data available from <http://files.grouplens.org/datasets/movielens/ml-latest-small.zip>

```
path='data/ml-latest-small/'  
  
ratings = pd.read_csv(path+'ratings.csv')  
ratings.head()
```

The dataset looks like this:

	userId	movieId	rating	timestamp
0	1	31	2.5	1260759144
1	1	1029	3.0	1260759179
2	1	1061	3.0	1260759182
3	1	1129	2.0	1260759185
4	1	1172	4.0	1260759205

It contains ratings by users. Our goal will be for some user-movie combination we have not seen before, we have to predict a rating.

```
movies = pd.read_csv(path+'movies.csv')
movies.head()
```

	movieId	title	genres
0	1	Toy Story (1995)	Adventure Animation Children Comedy Fantasy
1	2	Jumanji (1995)	Adventure Children Fantasy
2	3	Grumpier Old Men (1995)	Comedy Romance
3	4	Waiting to Exhale (1995)	Comedy Drama Romance
4	5	Father of the Bride Part II (1995)	Comedy

To make it more interesting, we will also actually download a list of movies so that we can interpret what is actually in these embedding matrices.

```

g=ratings.groupby('userId')['rating'].count()
topUsers=g.sort_values(ascending=False)[:15]

g=ratings.groupby('movieId')['rating'].count()
topMovies=g.sort_values(ascending=False)[:15]

top_r = ratings.join(topUsers, rsuffix='_r', how='inner',
                     on='userId')
top_r = top_r.join(topMovies, rsuffix='_r', how='inner',
                     on='movieId')

pd.crosstab(top_r.userId, top_r.movieId, top_r.rating,
            aggfunc=np.sum)

```

movieId	1	110	260	296	318	356	480	527	589	593	608	1196	1198	1270	2571
userId															
15	2.0	3.0	5.0	5.0	2.0	1.0	3.0	4.0	4.0	5.0	5.0	5.0	4.0	5.0	5.0
30	4.0	5.0	4.0	5.0	5.0	5.0	4.0	5.0	4.0	4.0	5.0	4.0	5.0	5.0	3.0
73	5.0	4.0	4.5	5.0	5.0	5.0	4.0	5.0	3.0	4.5	4.0	5.0	5.0	5.0	4.5
212	3.0	5.0	4.0	4.0	4.5	4.0	3.0	5.0	3.0	4.0	NaN	NaN	3.0	3.0	5.0
213	3.0	2.5	5.0	NaN	NaN	2.0	5.0	NaN	4.0	2.5	2.0	5.0	3.0	3.0	4.0
294	4.0	3.0	4.0	NaN	3.0	4.0	4.0	4.0	3.0	NaN	NaN	4.0	4.5	4.0	4.5
311	3.0	3.0	4.0	3.0	4.5	5.0	4.5	5.0	4.5	2.0	4.0	3.0	4.5	4.5	4.0
380	4.0	5.0	4.0	5.0	4.0	5.0	4.0	NaN	4.0	5.0	4.0	4.0	NaN	3.0	5.0
452	3.5	4.0	4.0	5.0	5.0	4.0	5.0	4.0	4.0	5.0	5.0	4.0	4.0	4.0	2.0
468	4.0	3.0	3.5	3.5	3.5	3.0	2.5	NaN	NaN	3.0	4.0	3.0	3.5	3.0	3.0
509	3.0	5.0	5.0	5.0	4.0	4.0	3.0	5.0	2.0	4.0	4.5	5.0	5.0	3.0	4.5
547	3.5	NaN	NaN	5.0	5.0	2.0	3.0	5.0	NaN	5.0	5.0	2.5	2.0	3.5	3.5
564	4.0	1.0	2.0	5.0	NaN	3.0	5.0	4.0	5.0	5.0	5.0	5.0	5.0	3.0	3.0
580	4.0	4.5	4.0	4.5	4.0	3.5	3.0	4.0	4.5	4.0	4.5	4.0	3.5	3.0	4.5
624	5.0	NaN	5.0	5.0	NaN	3.0	3.0	NaN	3.0	5.0	4.0	5.0	5.0	5.0	2.0

This is what we are creating—this kind of cross tab of users by movies.

Feel free to look ahead and you will find that most of the steps are familiar to you already.



Hiromi Suenaga [Follow](#)
Jan 11 · 13 min read

Deep Learning 2: Part 1 Lesson 5

My personal notes from fast.ai course. These notes will continue to be updated and improved as I continue to review the course to “really” understand it. Much appreciation to Jeremy and Rachel who gave me this opportunity to learn.

• • •

Lesson 5

I. Introduction

There is not enough publications on structured deep learning, but it is definitely happening in industries:

Structured Deep Learning

by Kerem Turgutlu

towardsdatascience.com



You can download images from Google by using this tool and solve your own problems:

Fun with small image data-sets (Part 2)

by Nikhil B

towardsdatascience.com



Introduction on how to train Neural Net (a great technical writing):

How do we 'train' neural networks ?

by Vitaly Bushaev

towardsdatascience.com



Students are competing with Jeremy in Kaggle seedling classification competition.

II. Collaborative Filtering—using MovieLens dataset

The notebook discussed can be found here(lesson5-movielens.ipynb).

Let's take a look at the data. We will use `userId` (categorical),
`movieId` (categorical) and `rating` (dependent) for modeling.

```
ratings = pd.read_csv(path+'ratings.csv')
ratings.head()
```

	userId	movieId	rating	timestamp
0	1	31	2.5	1260759144
1	1	1029	3.0	1260759179
2	1	1061	3.0	1260759182
3	1	1129	2.0	1260759185
4	1	1172	4.0	1260759205

Create subset for Excel

We create a crosstab of the most popular movies and most movie-addicted users which we will copy into Excel for visualization.

```
g=ratings.groupby('userId')['rating'].count()
topUsers=g.sort_values(ascending=False)[:15]

g=ratings.groupby('movieId')['rating'].count()
topMovies=g.sort_values(ascending=False)[:15]

top_r = ratings.join(topUsers, rsuffix='_r', how='inner',
on='userId')
top_r = top_r.join(topMovies, rsuffix='_r', how='inner',
on='movieId')

pd.crosstab(top_r.userId, top_r.movieId, top_r.rating,
aggfunc=np.sum)
```

movielid	1	110	260	296	318	356	480	527	589	593	608	1196	1198	1270	2571
userId															
15	2.0	3.0	5.0	5.0	2.0	1.0	3.0	4.0	4.0	5.0	5.0	5.0	4.0	5.0	5.0
30	4.0	5.0	4.0	5.0	5.0	5.0	4.0	5.0	4.0	4.0	5.0	4.0	5.0	5.0	3.0
73	5.0	4.0	4.5	5.0	5.0	5.0	4.0	5.0	3.0	4.5	4.0	5.0	5.0	5.0	4.5
212	3.0	5.0	4.0	4.0	4.5	4.0	3.0	5.0	3.0	4.0	NaN	NaN	3.0	3.0	5.0
213	3.0	2.5	5.0	NaN	NaN	2.0	5.0	NaN	4.0	2.5	2.0	5.0	3.0	3.0	4.0
294	4.0	3.0	4.0	NaN	3.0	4.0	4.0	4.0	3.0	NaN	NaN	4.0	4.5	4.0	4.5
311	3.0	3.0	4.0	3.0	4.5	5.0	4.5	5.0	4.5	2.0	4.0	3.0	4.5	4.5	4.0
380	4.0	5.0	4.0	5.0	4.0	5.0	4.0	NaN	4.0	5.0	4.0	4.0	NaN	3.0	5.0
452	3.5	4.0	4.0	5.0	5.0	4.0	5.0	4.0	4.0	5.0	5.0	4.0	4.0	4.0	2.0
468	4.0	3.0	3.5	3.5	3.5	3.0	2.5	NaN	NaN	3.0	4.0	3.0	3.5	3.0	3.0
509	3.0	5.0	5.0	5.0	4.0	4.0	3.0	5.0	2.0	4.0	4.5	5.0	5.0	3.0	4.5
547	3.5	NaN	NaN	5.0	5.0	2.0	3.0	5.0	NaN	5.0	5.0	2.5	2.0	3.5	3.5
564	4.0	1.0	2.0	5.0	NaN	3.0	5.0	4.0	5.0	5.0	5.0	5.0	5.0	3.0	3.0
580	4.0	4.5	4.0	4.5	4.0	3.5	3.0	4.0	4.5	4.0	4.5	4.0	3.5	3.0	4.5
624	5.0	NaN	5.0	5.0	NaN	3.0	3.0	NaN	3.0	5.0	4.0	5.0	5.0	5.0	2.0

This is the excel file with above information. To begin with, we will use **matrix factorization/decomposition** instead of building a neural net.

f_x	=SQRT(SUMXMY2(H2:V16,H25:V39)/COUNT(H2:V16))																					
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	
1						movielid	27	49	57	72	79	89	92	99	143	179	180	197	402	417	505	
2					userId	14	3	5	1	3	4	4	5	2	5	5	4	5	5	2	5	
3						29	5	5	5	4	5	4	4	5	4	4	5	5	3	4	5	
4						72	4	5	5	4	5	3	4.5	5	4.5	5	5	5	4.5	5	4	
5						211	5	4	4	3	5	3	4	4.5	4	3	3	5	3	3	2	
6						212	2.5		2	5		4	2.5		5	5	3	3	4	3	2	
7						293	3		4	4	4	3		3	4	4	4.5	4	4.5	4		
8						310	3	3	5	4.5	5	4.5	2	4.5	4	3	4.5	4.5	4	3	4	
9						379	5	5	5	4		4	5	4	4	4		3	5	4	4	
10						451	4	5	4	5	4	4	5	5	4	4	4	4	2	3.5	5	
11						467	3	3.5	3	2.5			3	3.5	3.5	3	3.5	3	3	4	4	
12						508	5	5	4	3	5	2	4	4	5	5	5	3	4.5	3	4.5	
13						546		5	2	3	5		5	5		2.5	2	3.5	3.5	3.5	5	
14						563	1	5	3	5	4	5	5		2	5	5	3	3	4	5	
15						579	4.5	4.5	3.5	3	4	4.5	4	4	4	4	3.5	3	4.5	4	4.5	
16						623		5	3	3		3	5		5	5	5	2	5	4		
17																						
18																						
19						NB: These are initialized to random numbers		0.71	0.92	0.68	0.83	0.60	0.18	0.26	0.91	0.99	0.52	0.91	0.53	0.23	0.75	0.43
20						Then we use Solver to optimize them		0.81	0.55	0.28	0.88	0.50	0.31	0.08	0.47	0.94	0.70	0.11	0.87	0.20	0.47	0.81
21						with gradient descent		0.74	0.86	0.53	0.33	0.81	0.68	0.92	0.61	0.46	0.64	0.24	0.25	0.83	0.05	0.17
22								0.04	0.44	0.16	0.41	0.73	0.39	0.29	0.94	0.12	0.67	0.54	0.57	0.53	0.91	0.30
23						movielid		0.04	0.80	0.94	0.24	0.53	0.09	0.74	0.13	0.39	0.44	0.81	0.80	0.23	0.59	0.29
24						userId		27	49	57	72	79	89	92	99	143	179	180	197	402	417	505
25	0.19	0.63	0.31	0.44	0.51	14		0.91	1.40	1.02	1.12	1.27	0.66	0.89	1.13	1.18	1.27	0.96	1.39	0.77	1.15	0.92
26	0.25	0.83	0.71	0.96	0.59	29		1.44	2.20	1.49	1.71	2.16	1.22	1.49	2.04	1.71	2.08	1.48	2.06	1.46	1.84	1.36
27	0.30	0.44	0.19	0.00	0.72	72		0.73	1.26	1.10	0.87	0.93	0.38	0.82	0.68	1.07	0.90	0.95	1.16	0.47	0.86	0.72
28	0.02	0.72	0.69	0.35	0.25	211		1.12	1.36	0.86	1.08	1.31	0.85	0.97	1.14	1.15	0.00	0.65	1.21	0.96	0.85	0.00
29	0.60	0.87	0.76	0.30	0.04	212		1.71	0.00	1.14	1.65	0.00	1.02	1.03	0.00	1.82	1.64	1.01	1.47	1.11	1.19	1.20
30	0.73	0.70	0.44	0.47	0.29	293		1.44	0.00	1.27	1.63	1.64	0.86	0.00	1.74	1.76	1.60	1.33	1.61	0.98	1.50	0.00
31	0.23	0.81	0.36	0.47	0.12	310		1.10	1.27	0.76	1.24	1.24	0.73	0.67	1.26	1.26	1.29	0.73	1.28	0.79	1.07	0.99
32	0.68	0.90	0.20	0.92	0.74	379		1.43	2.30	1.67	1.98	0.00	0.96	1.24	2.13	2.02	2.07	0.00	2.32	1.15	2.22	1.55
33	0.81	0.41	0.81	0.15	0.17	451		1.52	1.88	1.28	1.41	1.55	0.90	1.15	1.59	1.66	1.42	1.20	1.22	1.05	1.09	0.92
34	0.70	0.61	0.00	0.80	0.24	467		1.70	2.25	1.40	1.61	2.00	1.00	1.40	2.01	2.00	2.00	1.50	1.61	1.62	1.50	

- Blue cells—the actual rating
 - Purple cells—our predictions
 - Red cell—our loss function i.e. Root Mean Squared Error (RMSE)
 - Green cells—movie embeddings (randomly initialized)
 - Orange cells—user embeddings (randomly initialized)

Each prediction is a dot product of movie embedding vector and user embedding vector. In linear algebra term, it is equivalent of matrix product as one is a row and one is a column. If there is no actual rating we set the prediction to zero (think of this as test data—not training data).

SUM	B	C	D	E	F	G	H	I	J	K	L	M	N	
	19	NB: These are initialized to random numbers					0.71	0.92	0.68	0.83	0.60	0.18	0.2	
	20	Then we use Solver to optimize them					0.81	0.55	0.28	0.88	0.50	0.31	0.0	
	21	with gradient descent					0.74	0.86	0.53	0.33	0.81	0.68	0.9	
	22						0.04	0.44	0.16	0.41	0.73	0.39	0.2	
	23					movielid	0.04	0.80	0.94	0.24	0.53	0.09	0.7	
	24				userId		27	49	57	72	79	89	92	
	25	0.19	0.63	0.31	0.44	0.51	14	=IF(H2="",0,MMULT(\$B\$25:\$F\$25,H\$19:H\$23))						
	26	0.25	0.83	0.71	0.96	0.59	29	1.44	2.20	1.49	1.71	2.16	1.22	1.49

We then use Gradient Descent to minimize our loss. Microsoft excel has a “solver” in the add-ins that would minimize a variable by changing selected cells (`GRG Nonlinear` is the method you want to use).

This can be called “shallow learning” (as opposed to deep learning) as there is no nonlinear layer or a second linear layer. So what did we just do intuitively? The five numbers for each movie is called “embeddings” (latent factors)—the first number might represent how much it is sci-fi and fantasy, the second might be how much special effect is used for a movie, the third might be how dialog driven it is, etc. Similarly, each user also has 5 numbers representing, for example, how much does the user like sci-fi fantasy, special effects, and dialog-driven in movies. Our prediction is a cross product of these vectors. Since we do not have every movie review for every user, we are trying to figure out which movies are similar this movie and how other users who rated other movies similarly to this user rate this movie (hence the name “collaborative”).

What do we do with a new user or a new movie—do we have to retrain a model? We do not have a time to cover this now, but basically you need to have a new user model or a new movie model that you would use initially and over time you will need to re-train the model.

Simple Python version

This should look familiar by now. We create a validation set by picking random set of ID's. `wd` is a weight decay for L2 regularization, and `n_factors` is how big an embedding matrix we want.

```
val_idxs = get_cv_idxs(len(ratings))
wd = 2e-4
n_factors = 50
```

We create a model data object from CSV file:

```
cf = CollabFilterDataset.from_csv(path, 'ratings.csv',
'userId', 'movieId', 'rating')
```

We then get a learner that is suitable for the model data, and fit the model:

```
learn = cf.get_learner(n_factors, val_idxs, 64,
opt_fn=optim.Adam)

learn.fit(1e-2, 2, wds=wd, cycle_len=1, cycle_mult=2)
```

```
[ 0.          0.77809   0.80824]
[ 1.          0.78936   0.77727]
[ 2.          0.63006   0.765    ]
```

Output MSE

Since the output is Mean Squared Error, you can take RMSE by:

```
math.sqrt(0.765)
```

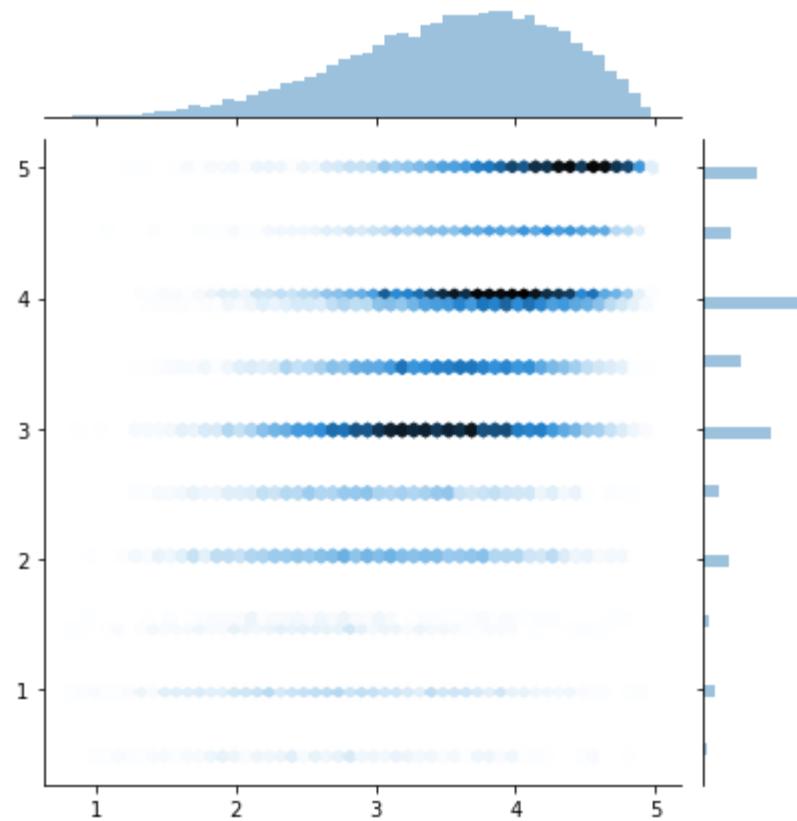
The output is about 0.88 which outperforms the bench mark of 0.91.

You can get a prediction in a usual way:

```
preds = learn.predict()
```

And you can also plot using seaborn `sns` (built on top of matplotlib):

```
y = learn.data.val_y
sns.jointplot(preds, y, kind='hex', stat_func=None)
```



Dot product with Python

Dot Product

tensor = multidimensional array

$$\begin{bmatrix} a & b \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = [ax + by]$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

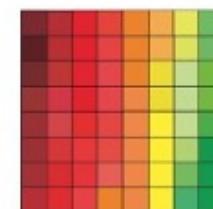
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw + by & ax + bz \\ cw + dy & cx + dz \end{bmatrix}$$

vector



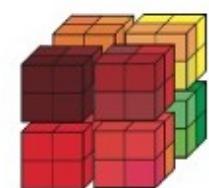
$\mathbf{v} \in \mathbb{R}^{64}$

matrix



$X \in \mathbb{R}^{8 \times 8}$

tensor



$\mathcal{X} \in \mathbb{R}^{4 \times 4 \times 4}$

`T` is a tensor in Torch

```
a = T([[1., 2], [3, 4]])
b = T([[2., 2], [10, 10]])
```

When we have a mathematical operator between tensors in numpy or PyTorch, it will do element-wise assuming that they both have the same dimensionality. The below is how you would calculate the dot product of two vectors (e.g. $(1, 2) \cdot (2, 2) = 6$ —the first rows of matrix a and b):

```
(a*b).sum(1)  
  
6  
70  
[torch.FloatTensor of size 2]
```

Building our first custom layer (i.e. PyTorch module)

We do this by creating a Python class that extends `nn.Module` and override `forward` function.

```
class DotProduct (nn.Module):  
    def forward(self, u, m): return (u*m).sum(1)
```

Now we can call it and get the expected result (notice that we do not need to say `model.forward(a, b)` to call the `forward` function—it is a PyTorch magic.):

```
model = DotProduct()  
model(a,b)  
  
6  
70  
[torch.FloatTensor of size 2]
```

Building more complex module

This implementation has two additions to the `DotProduct` class:

- Two `nn.Embedding` matrices
- Look up our users and movies in above embedding matrices

It is quite possible that user ID's are not contiguous which makes it hard to use as an index of embedding matrix. So we will start by creating indexes that starts from zero and contiguous and replace

`ratings.userId` column with the index by using Panda's `apply` function with an anonymous function `lambda` and do the same for `ratings.movieId`.

```
u_uniq = ratings.userId.unique()
user2idx = {o:i for i,o in enumerate(u_uniq)}
ratings.userId = ratings.userId.apply(lambda x: user2idx[x])

m_uniq = ratings.movieId.unique()
movie2idx = {o:i for i,o in enumerate(m_uniq)}
ratings.movieId = ratings.movieId.apply(lambda x:
    movie2idx[x])

n_users=int(ratings.userId.nunique())
n_movies=int(ratings.movieId.nunique())
```

Tip: `{o:i for i,o in enumerate(u_uniq)}` is a handy line of code to keep in your tool belt!

```
class EmbeddingDot(nn.Module):
    def __init__(self, n_users, n_movies):
```

```
super().__init__()  
self.u = nn.Embedding(n_users, n_factors)  
self.m = nn.Embedding(n_movies, n_factors)  
self.u.weight.data.uniform_(0,0.05)  
self.m.weight.data.uniform_(0,0.05)  
  
def forward(self, cats, conts):  
    users,movies = cats[:,0],cats[:,1]  
    u,m = self.u(users),self.m(movies)  
    return (u*m).sum(1)
```

Note that `__init__` is a constructor which is now needed because our class needs to keep track of “states” (how many movies, how many users, how many factors, etc). We initialized the weights to random numbers between 0 and 0.05 and you can find more information about a standard algorithm for weight initialization, “Kaiming Initialization” here (PyTorch has He initialization utility function but we are trying to do things from scratch here).

`Embedding` is not a tensor but a **variable**. A variable does the exact same operations as a tensor but it also does automatic differentiation. To pull a tensor out of a variable, call `data` attribute. All the tensor functions have a variation with trailing underscore (e.g. `uniform_`) will do things in-place.

```
x = ratings.drop(['rating', 'timestamp'],axis=1)  
y = ratings['rating'].astype(np.float32)  
data = ColumnarModelData.from_data_frame(path, val_idxs, x,  
y, ['userId', 'movieId'], 64)
```

We are reusing `ColumnarModelData` (from fast.ai library) from Rossmann notebook, and that is the reason behind why there are both categorical and continuous variables in `def forward(self, cats, conts)` function in `EmbeddingDot` class. Since we do not have continuous variable in this case, we will ignore `conts` and use the first and second columns of `cats` as `users` and `movies`. Note that they are mini-batches of users and movies. It is important not to manually loop through mini-batches because you will not get GPU acceleration, instead, process a whole mini-batch at a time as you see in line 3 and 4 of `forward` function above [51:00–52:05].

```
wd=1e-5
model = EmbeddingDot(n_users, n_movies).cuda()
opt = optim.SGD(model.parameters(), 1e-1, weight_decay=wd,
momentum=0.9)
```

`optim` is what gives us the optimizers in PyTorch.
`model.parameters()` is one of the function inherited from `nn.Modules` that gives us all the weight to be updated/learned.

```
fit(model, data, 3, opt, F.mse_loss)
```

This function is from fast.ai library [54:40] and is closer to regular PyTorch approach compared to `learner.fit()` we have been using. It will not give you features like “stochastic gradient descent with restarts” or “differential learning rate” out of box.

Let's improve our model

Bias—to adjust to generally popular movies or generally enthusiastic users.

```
min_rating,max_rating =
    ratings.rating.min(),ratings.rating.max()
min_rating,max_rating

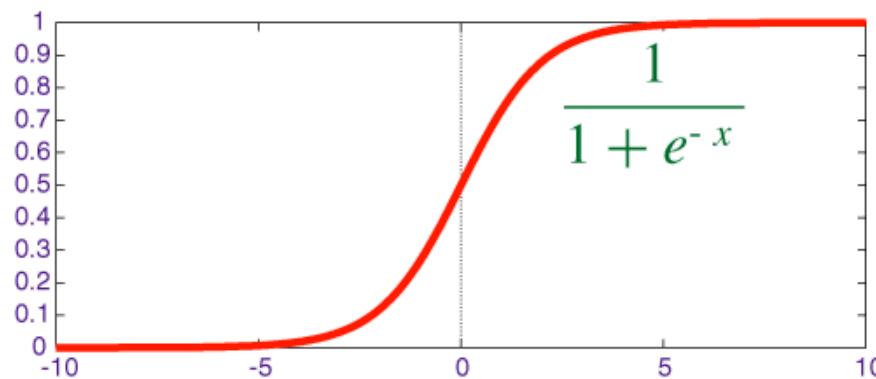
def get_emb(ni,nf):
    e = nn.Embedding(ni, nf)
    e.weight.data.uniform_(-0.01,0.01)
    return e

class EmbeddingDotBias(nn.Module):
    def __init__(self, n_users, n_movies):
        super().__init__()
        (self.u, self.m, self.ub, self.mb) = [get_emb(*o)
for o in [
    (n_users, n_factors), (n_movies, n_factors),
    (n_users,1), (n_movies,1)
    ]]

    def forward(self, cats, conts):
        users,movies = cats[:,0],cats[:,1]
        um = (self.u(users)* self.m(movies)).sum(1)
        res = um + self.ub(users).squeeze() +
self.mb(movies).squeeze()
        res = F.sigmoid(res) * (max_rating-min_rating) +
min_rating
        return res
```

`squeeze` is PyTorch version of *broadcasting* [1:04:11] for more information, see Machine Learning class or numpy documentation.

Can we squish the ratings so that it is between 1 and 5? Yes! By putting the prediction through sigmoid function will result in number between 1 and 0. So in our case, we can multiply that by 4 and add 1—which will result in number between 1 and 5.



`F` is a PyTorch functional (`torch.nn.functional`) that contains all functions for tensors, and is imported as `F` in most cases.

```
wd=2e-4
model = EmbeddingDotBias(cf.n_users, cf.n_items).cuda()
opt = optim.SGD(model.parameters(), 1e-1, weight_decay=wd,
momentum=0.9)

fit(model, data, 3, opt, F.mse_loss)
[ 0.        0.85056  0.83742]
[ 1.        0.79628  0.81775]
[ 2.        0.8012   0.80994]
```



Let's take a look at fast.ai code [1:13:44] we used in our **Simple Python version**. In `column_data.py` file,
`CollabFilterDataSet.get_leaner` calls `get_model` function that creates `EmbeddingDotBias` class that is identical to what we created.

Neural Net Version [1:17:21]

We go back to excel sheet to understand the intuition. Notice that we create `user_idx` to look up Embeddings just like we did in the python code earlier. If we were to one-hot-encode the `user_idx` and multiply it by user embeddings, we will get the applicable row for the user. If it is just matrix multiplication, why do we need Embeddings? It is for computational performance optimization purposes.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	
1	Users Embeddings							original data					user embedding					movie embedding										
2	idx	Row	Label							userId	movieId	rating	user idx	1	2	3	4	5	movie idx	1	2	3	4	5	predict	error^2	rmse	
3	14	0.19	0.63	0.31	0.44	0.51	29	417	4	2	0.25	0.83	0.71	0.96	0.59	14	0.75	0.47	0.05	0.91	0.59	1.15	0.72	2.81				
4	29	0.25	0.83	0.71	0.96	0.59	72	417	5	3	0.30	0.44	0.19	0.00	0.72	14	0.75	0.47	0.05	0.91	0.59	1.84	4.69					
5	72	0.30	0.44	0.19	0.00	0.72	211	417	3	4	0.02	0.72	0.69	0.35	0.25	14	0.75	0.47	0.05	0.91	0.59	0.86	17.14					
6	211	0.02	0.72	0.69	0.35	0.25	212	417	3	5	0.60	0.87	0.76	0.30	0.04	14	0.75	0.47	0.05	0.91	0.59	0.85	4.61					
7	212	0.60	0.87	0.76	0.30	0.04	293	417	4	6	0.73	0.70	0.44	0.47	0.29	14	0.75	0.47	0.05	0.91	0.59	1.50	6.26					
8	293	0.73	0.70	0.44	0.47	0.29	310	417	3	7	0.23	0.81	0.36	0.47	0.12	14	0.75	0.47	0.05	0.91	0.59	1.07	3.73					
9	310	0.23	0.81	0.36	0.47	0.12	379	417	4	8	0.68	0.90	0.20	0.92	0.74	14	0.75	0.47	0.05	0.91	0.59	2.22	3.18					
10	379	0.68	0.90	0.20	0.92	0.74	451	417	3.5	9	0.81	0.41	0.81	0.15	0.17	14	0.75	0.47	0.05	0.91	0.59	1.09	5.83					
11	451	0.81	0.41	0.81	0.15	0.17	467	417	4	10	0.70	0.61	0.90	0.89	0.24	14	0.75	0.47	0.05	0.91	0.59	1.81	4.79					
12	467	0.70	0.61	0.90	0.89	0.24	508	417	3	11	0.50	0.27	0.73	0.44	0.83	14	0.75	0.47	0.05	0.91	0.59	1.43	2.48					
13	508	0.50	0.27	0.73	0.44	0.83	546	417	3.5	12	0.16	0.21	0.75	0.48	0.98	14	0.75	0.47	0.05	0.91	0.59	1.26	5.00					
14	546	0.16	0.21	0.75	0.48	0.98	563	417	4	13	0.91	0.75	0.75	0.24	0.06	14	0.75	0.47	0.05	0.91	0.59	1.32	7.17					
15	563	0.91	0.75	0.75	0.24	0.06	579	417	4	14	0.55	0.58	0.68	0.93	0.66	14	0.75	0.47	0.05	0.91	0.59	1.96	4.18					
16	579	0.55	0.58	0.68	0.93	0.66	623	417	5	15	0.94	0.25	0.46	0.16	0.30	14	0.75	0.47	0.05	0.91	0.59	1.16	14.76					
17	623	0.94	0.25	0.46	0.16	0.30	14	27	3	1	0.19	0.63	0.31	0.44	0.51	1	0.71	0.81	0.74	0.04	0.04	0.91	4.35					
18	27	0.71	0.81	0.74	0.04	0.04	29	27	5	2	0.25	0.83	0.71	0.96	0.59	1	0.71	0.81	0.74	0.04	0.04	1.44	12.67					
19	Movies Embeddings							72	27	4	3	0.30	0.44	0.19	0.00	0.72	1	0.71	0.81	0.74	0.04	0.04	0.73	10.69				
20	27	0.92	0.55	0.86	0.44	0.80	211	27	5	4	0.02	0.72	0.69	0.35	0.25	1	0.71	0.81	0.74	0.04	0.04	1.12	15.02					
21	49	0.68	0.28	0.53	0.16	0.94	212	27	2.5	5	0.60	0.87	0.76	0.30	0.04	1	0.71	0.81	0.74	0.04	0.04	1.71	0.63					
22	57	0.83	0.88	0.33	0.41	0.24	293	27	3	6	0.73	0.70	0.44	0.47	0.29	1	0.71	0.81	0.74	0.04	0.04	1.44	2.44					
23	72	0.83	0.88	0.33	0.41	0.24	310	27	3	7	0.23	0.81	0.36	0.47	0.12	1	0.71	0.81	0.74	0.04	0.04	1.10	3.59					
24	379	0.68	0.81	0.74	0.04	0.04	379	27	5	8	0.68	0.90	0.20	0.92	0.74	1	0.71	0.81	0.74	0.04	0.04	1.43	12.76					
25	27	0.71	0.81	0.74	0.04	0.04	29	27	5	2	0.25	0.83	0.71	0.96	0.59	1	0.71	0.81	0.74	0.04	0.04	1.44	12.67					

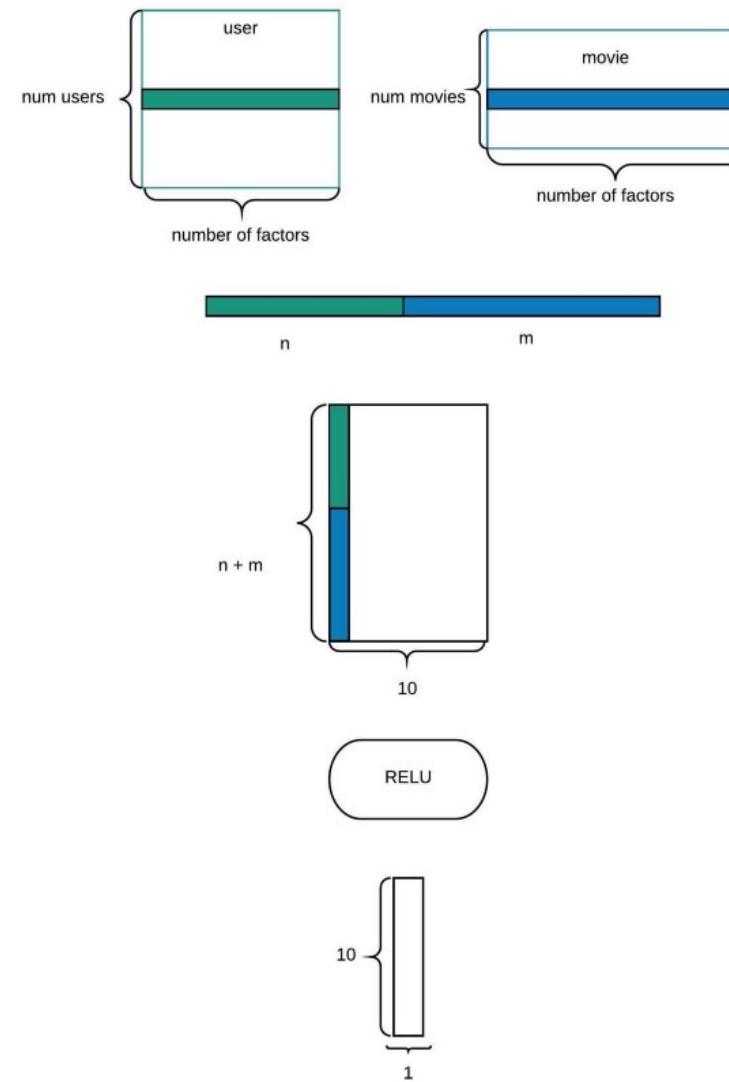
Rather than calculating the dot product of user embedding vector and movie embedding vector to get a prediction, we will concatenate the two and feed it through neural net.

```
class EmbeddingNet(nn.Module):
    def __init__(self, n_users, n_movies, nh=10, p1=0.5,
p2=0.5):
```

```
super().__init__()
(self.u, self.m) = [get_emb(*o) for o in [
    (n_users, n_factors), (n_movies, n_factors)]]
self.lin1 = nn.Linear(n_factors*2, nh)
self.lin2 = nn.Linear(nh, 1)
self.drop1 = nn.Dropout(p1)
self.drop2 = nn.Dropout(p2)

def forward(self, cats, conts):
    users,movies = cats[:,0],cats[:,1]
    x =
    self.drop1(torch.cat([self.u(users),self.m(movies)], dim=1))
    x = self.drop2(F.relu(self.lin1(x)))
    return F.sigmoid(self.lin2(x)) * (max_rating-
min_rating+1) + min_rating-0.5
```

Notice that we no longer has bias terms since `Linear` layer in PyTorch already has a build in bias. `nh` is a number of activations a linear layer creates (Jeremy calls it “num hidden”).



It only has one hidden layer, so maybe not “deep”, but this is definitely a neural network.

```
wd=1e-5
model = EmbeddingNet(n_users, n_movies).cuda()
opt = optim.Adam(model.parameters(), 1e-3, weight_decay=wd)
fit(model, data, 3, opt, F.mse_loss)

A Jupyter Widget

[ 0. 0.88043 0.82363]
[ 1. 0.8941 0.81264]
[ 2. 0.86179 0.80706]
```

Notice that the loss functions are also in `F` (here, it's mean squared loss).

Now that we have neural net, there are many things we can try:

- Add dropouts
- Use different embedding sizes for user embedding and movie embedding
- Not only user and movie embeddings, but append movie genre embedding and/or timestamp from the original data.
- Increase/decrease number of hidden layers and activations
- Increase/decrease regularization

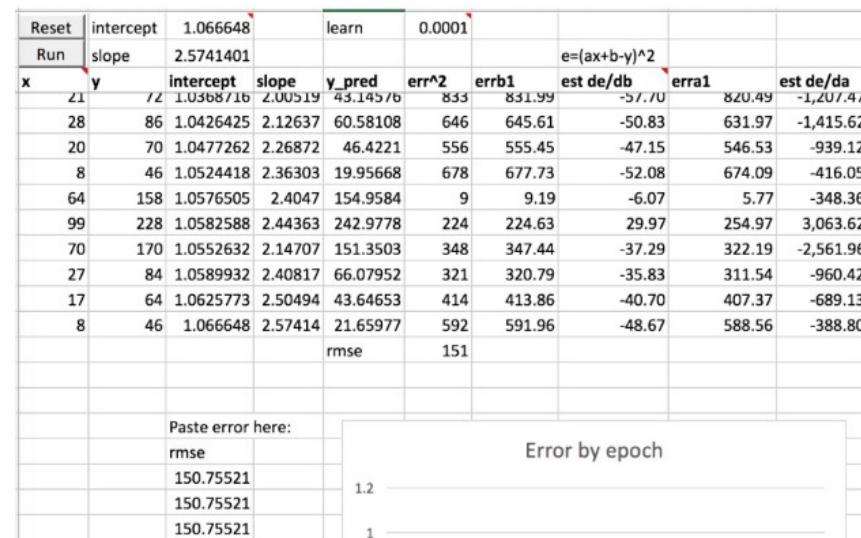
What is happening in the training loop? [1:33:21]

Currently, we are passing off the updating of weights to PyTorch's optimizer. What does an optimizer do? and what is a `momentum`?

```
opt = optim.SGD(model.parameters(), 1e-1, weight_decay=wd,  
momentum=0.9)
```

We are going to implement gradient descent in an excel sheet (graddesc.xlsx)—see worksheets right to left. First we create a random x 's, and y 's that are linearly correlated with the x 's (e.g. $y = a*x + b$). By using sets of x 's and y 's, we will try to learn a and b .

weights:	
b	const
a	slope
x	$y=a*x + b$
29	88
7	44
76	182
75	\$C\$3
23	76
34	98
50	130



To calculate the error, we first need a prediction, and square the difference:

Reset	intercept	1	learn	0.0001			de/db=2(ax+b-y)						
Run	slope	1			e=(ax+b-y)^2	de/da=x*2(ax+b-y)							
x	y	intercept	slope	y_pred	err^2	errb1	est	de/db	erra1	est de/da	de/db	de/da	new a new b
14	58	1	1	15	= $(B4-E4)^2$	1,848.14	-85.99	1,836.98	-1,202.04	-86.00	-1,204.00	1.12	1.01
86	202	1.0086	1.1204	97.363	10,948.90	10,946.81	-209.26	10,769.67	-17,923.60	-209.27	-17,997.56	2.92	1.03

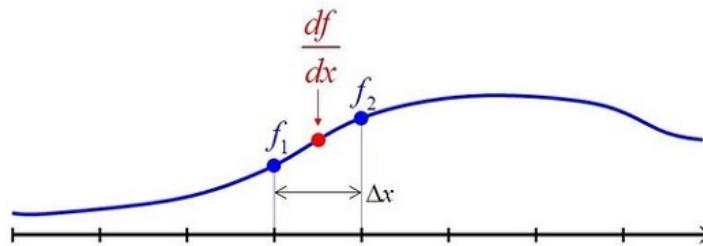
To reduce the error, we increase/decrease a and b a little bit and figure out what would make the error decrease. This is called finding the derivative through finite differencing.

The Basic Finite-Difference Approximation



$$\frac{df_{1.5}}{dx} \approx \frac{f_2 - f_1}{\Delta x}$$

second-order accurate
first-order derivative



This is the only finite-difference approximation we will use in this course!

Lecture 9

Slide 11

Finite differencing gets complicated in high dimensional spaces [1:41:46], and it becomes very memory intensive and takes a long time. So we want to find some way to do this more quickly. It is worthwhile to look up things like Jacobian and Hessian (Deep Learning book: section 4.3.1 page 84).

Chain Rule and Backpropagation

The faster approach is to do this analytically [1:45:27]. For this, we need a chain rule:

$$\frac{d}{dx} f(g(t)) = \frac{df}{dg} \frac{dg}{dt} = f'(g(t))g'(t)$$

Overview of chain rule

Here is a great article by Chris Olah on Backpropagation as a chain rule.

Now we replace the finite-difference with an actual derivative
 WolframAlpha gave us (notice that finite-difference output is fairly close to the actual derivative and good way to do quick sanity check if you need to calculate your own derivative):

					de/da=x^2(ax+b-y)			
	e=(ax+b-y)^2				de/db=2(ax+b-y)			
	errb1	est de/db	erra1	est de/da	de/db	de/da	new a	new b
9	1,848.14	-85.99	1,836.98	-1,202.04	-86.00	-1,204.00	1.12	1.01
9	10,946.81	-209.26	10,769.67	-17,923.60	-209.27	-17,997.56	2.92	1.03
0	10.22	-6.40	8.56	-171.70	-6.41	-179.54	2.94	1.03
5	356.60	37.76	375.73	1,951.14	37.75	1,925.13	2.75	1.03
5	65.40	-16.18	61.10	-445.58	-16.19	-453.42	2.79	1.03

- “Online” training—mini-batch with size 1

And this is how you do SGD with excel sheet. If you were to change the prediction value with the output from CNN spreadsheet, we can train CNN with SGD.

Momentum [1:53:47]

Come on, take a hint—that’s a good direction. Please keep doing that but more.

With this approach, we will use a linear interpolation between the current mini-batch's derivative and the step (and direction) we took after the last mini-batch (cell K9):

	A	B	C	D	E	F	G	H	I	J	K	L
1	Reset	b	1.0000	learn							0.98	0.02
2	Run	a	1.0000		0.0001							
3	x	y	b	a	pred	de/db	de/da	new b	new a	-11.97	-174.9	err^2
4	14	58	1.0000	1.0000	15.0000	-86.00	-1204.00	1.00	1.02	-13.45	-195.4	1849
5	86	202	1.0013	1.0195	88.6822	-226.64	-19490.66	1.00	1.08	-17.71	-581.4	12840.93
6	28	86	1.0031	1.0777	31.1782	-109.64	-3070.02	1.01	1.14	-19.55	-631.1	3005.435
7	51	132	1.0051	1.1408	59.1855	-145.63	-7427.08	1.01	1.22	-22.07	-767	5301.954
8	28	86	1.0073	1.2175	35.0972	-101.81	-2850.56	1.01	1.30	-23.67	-808.7	2591.096
9	29	88	1.0096	1.2984	38.6623	-98.68	-2861.59	1.01	1.38	-25.17	=KB*\$J\$1+\$K\$1*G9	
10	72	174	1.0122	1.3833	100.6130	-146.77	-10567.72	1.01	1.49	-27.6	-1044	5385.646

Compared to de/db whose sign (+/-) is random, the one with momentum will keep going the same direction a little bit faster up till certain point. This will reduce a number of epochs required for training.

Adam [1:59:04]

Adam is much faster but the issue has been that final predictions are not as good as they are with SGD with momentum. It seems as though that it was due to the combined usage of Adam and weight decay. The new version that fixes this issue is called **AdamW**.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Reset	b	1.0000	learn					beta	0.7	0.3	0.9	0.1
2	Run	a	1.0000		1								
3	x	y	b	a	pred	de/db	de/da	new b	new a	-19.37	-637	5490	27555653.26
4	14	58	1.0000	1.0000	15.0000	-86.00	-1204.00	1.52	1.16	-39.36	-807.1	5680.6	24945049.53
5	86	202	1.5222	1.1616	101.4192	-201.16	-17299.90	2.44	1.96	-87.9	-5755	9159.14	52379213.4
6	28	86	2.4407	1.9568	57.2300	-57.54	-1611.12	3.29	2.61	-78.79	-4512	8574.31	47400862.32
7	51	132	3.2916	2.6121	136.5079	9.02	459.81	3.89	3.07	-52.45	-3020	7725.01	42681918.29
8	28	86	3.8883	3.0744	89.9713	7.94	222.39	=C8-J8*\$D\$2/SQRT(L8)		-2047	6958.82		38418672.22
9	29	88	4.2999	3.4047	103.0369	30.07	872.14	4.49	3.60	-15.01	-1172	6353.38	34652867.48

- cell J8 : a linear interpolation of derivative and previous direction (identical to what we had in momentum)
- cell L8 : a linear interpolation of derivative squared + derivative squared from last step (cell L7)
- The idea is called “exponentially weighted moving average” (in another words, average with previous values multiplicatively decreased)

Learning rate is much higher than before because we are dividing it by square root of L8 .

If you take a look at fast.ai library (model.py), you will notice that in fit function, it does not just calculate average loss, but it is calculating the **exponentially weighted moving average of loss**.

```
avg_loss = avg_loss * avg_mom + loss * (1-avg_mom)
```

Another helpful concept is whenever you see ` $\alpha(\dots) + (1-\alpha)(\dots)$ `, immediately think **linear interpolation**.

Some intuitions

- We calculated exponentially weighted moving average of gradient squared, take a square root of that, and divided the learning rate by it.
- Gradient squared is always positive.
- When there is high variance in gradients, gradient squared will be large.
- When the gradients are constant, gradient squared will be small.
- If gradients are changing a lot, we want to be careful and divide the learning rate by a big number (slow down)
- If gradients are not changing much, we will take a bigger step by dividing the learning rate with a small number
- **Adaptive learning rate**—keep track of the average of the squares of the gradients and use that to adjust the learning rate. So there is just one learning rage, but effectively every parameter at every epoch is getting a bigger jump if the gradient is constant; smaller jump otherwise.
- There are two momentums—one for gradient, and the other for gradient squared (in PyTorch, it is called a beta which is a tuple of two numbers)

AdamW[2:11:18]

When there are much more parameters than data points, regularizations become important. We had seen dropout previously, and weight decay is another type of regularization. Weight decay (L2 regularization) penalizes large weights by adding squared weights (times weight decay multiplier) to the loss. Now the loss function wants to keep the weights small because increasing the weights will increase the loss; hence only doing so when the loss improves by more than the penalty.

The problem is that since we added the squared weights to the loss function, this affects the moving average of gradients and the moving average of the squared gradients for Adam. This result in decreasing the amount of weight decay when there is high variance in gradients, and increasing the amount of weight decay when there is little variation. In other words, “penalize large weights unless gradients varies a lot” which is not what we intended. AdamW removed the weight decay out of the loss function, and added it directly when updating the weights.



Hiromi Suenaga [Follow](#)
Jan 11 · 22 min read

Deep Learning 2: Part 1 Lesson 6

My personal notes from fast.ai course. These notes will continue to be updated and improved as I continue to review the course to “really” understand it. Much appreciation to Jeremy and Rachel who gave me this opportunity to learn.

• • •

Lesson 6

Optimization for Deep Learning Highlights in 2017

Table of contents: Deep Learning ultimately is about finding a minimum that generalizes well -...
[ruder.io](#)

Review from last week [2:15]

We took a deep dive to collaborative filtering last week, and we ended up re-creating `EmbeddingDotBias` class (`column_data.py`) in fast.ai library. Let's visualize what the embeddings look like [notebook].

Inside of a learner `learn`, you can get a PyTorch model itself by calling `learn.model`. `@property` looks like a regular function, but requires no parenthesis when you call it.

```
@property  
def model(self): return self.models.model
```

`learn.models` is an instance of `CollabFilterModel` which is a thin wrapper of PyTorch model that allows us to use “layer groups” which is not a concept available in PyTorch and fast.ai uses it to apply different learning rates to different sets of layers (layer group).

PyTorch model prints out the layers nicely including layer name which is what we called them in the code.

```
m=learn.model; m  
  
EmbeddingDotBias (  
    (u): Embedding(671, 50)  
    (i): Embedding(9066, 50)  
    (ub): Embedding(671, 1)  
    (ib): Embedding(9066, 1)  
)
```

```
class EmbeddingDotBias(nn.Module):
    def __init__(self, n_factors, n_users, n_items, min_score, max_score):
        super().__init__()
        self.min_score, self.max_score = min_score, max_score
        (self.u, self.i, self.ub, self.ib) = [get_emb(*o) for o in [
            (n_users, n_factors), (n_items, n_factors), (n_users, 1), (n_items, 1)
        ]]
    def forward(self, users, items):
        um = self.u(users)* self.i(items)
        res = um.sum(1) + self.ub(users).squeeze() + self.ib(items).squeeze()
        return F.sigmoid(res) * (self.max_score - self.min_score) + self.min_score
```

`m.ib` refers to an embedding layer for an item bias—movie bias, in our case. What is nice about PyTorch models and layers is that we can call them as if they are functions. So if you want to get a prediction, you call `m(...)` and pass in variables.

Layers require variables not tensors because it needs to keep track of the derivatives—that is the reason for `V(...)` to convert tensor to variable. PyTorch 0.4 will get rid of variables and we will be able to use tensors directly.

```
movie_bias = to_np(m.ib(V(topMovieIdx)))
```

The `to_np` function will take a variable or a tensor (regardless of being on the CPU or GPU) and returns a numpy array. Jeremy's approach [12:03] is to use numpy for everything except when he explicitly needs something to run on the GPU or he needs its derivatives—in which case he uses PyTorch. Numpy has been around longer than PyTorch and works well with other libraries such as OpenCV, Pandas, etc.

A question regarding CPU vs. GPU in production. The suggested approach is to do inference on CPU as it is more scalable and you do not need to put things in batches. You can move a model onto the CPU by typing `m.cpu()`, similarly a variable by typing `v.topMovieIndex.cpu()` (from CPU to GPU would be `m.cuda()`). If your server does not have GPU, it will run inference on CPU automatically. For loading a saved model that was trained on GPU, take a look at this line of code in `torch_imports.py`:

```
def load_model(m, p): m.load_state_dict(torch.load(p,
map_location=lambda storage, loc: storage))
```

Now that we have movie bias for top 3000 movies, and let's take a look at ratings:

```
movie_ratings = [(b[0], movie_names[i]) for i,b in
zip(topMovies,movie_bias)]
```

`zip` will allow you to iterate through multiple lists at the same time.

Worst movies

About sorting key—Python has `itemgetter` function but plain `lambda` is just one more character.

```
sorted(movie_ratings, key=lambda o: o[0])[:15]

[(-0.96070349, 'Battlefield Earth (2000)'),
 (-0.76858485, 'Speed 2: Cruise Control (1997)'),
 (-0.73675376, 'Wild Wild West (1999)'),
 (-0.73655486, 'Anaconda (1997)'),
 ...]

sorted(movie_ratings, key=itemgetter(0))[:15]
```

Best movies

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

[(1.3070084, 'Shawshank Redemption, The (1994)'),
 (1.1196285, 'Godfather, The (1972)'),
 (1.0844109, 'Usual Suspects, The (1995)'),
 (0.96578616, "Schindler's List (1993)'),
 ...]
```

Embedding interpretation [18:42]

Each movie has 50 embeddings and it is hard to visualize 50 dimensional space, so we will turn it into a three dimensional space. We can compress dimensions using several techniques: Principal Component Analysis (PCA) (Rachel's Computational Linear Algebra class covers this in detail—which is almost identical to Singular Value Decomposition (SVD))

```
movie_emb = to_np(m.i(V(topMovieIdx)))
movie_emb.shape

(3000, 50)

from sklearn.decomposition import PCA
pca = PCA(n_components=3)
movie_pca = pca.fit(movie_emb.T).components_
movie_pca.shape

(3, 3000)
```

We will take a look at the first dimension “easy watching vs. serious” (we do not know what it represents but can certainly speculate by looking at them):

```
fac0 = movie_pca[0]
movie_comp = [(f, movie_names[i]) for f,i in zip(fac0,
topMovies)]
sorted(movie_comp, key=itemgetter(0), reverse=True)[:10]

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

[(0.06748189, 'Independence Day (a.k.a. ID4) (1996)'),
(0.061572548, 'Police Academy 4: Citizens on Patrol
(1987)'),
(0.061050549, 'Waterworld (1995)'),
(0.057877172, 'Rocky V (1990)'),
...
]

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

```
[(-0.078433245, 'Godfather: Part II, The (1974)'),
 (-0.072180331, 'Fargo (1996)'),
 (-0.071351372, 'Pulp Fiction (1994)'),
 (-0.068537779, 'Goodfellas (1990)'),
 ...
]
```

The second dimension “dialog driven vs. CGI”

```
fac1 = movie_pca[1]
movie_comp = [(f, movie_names[i]) for f,i in zip(fac1,
topMovies)]
sorted(movie_comp, key=itemgetter(0), reverse=True)[:10]

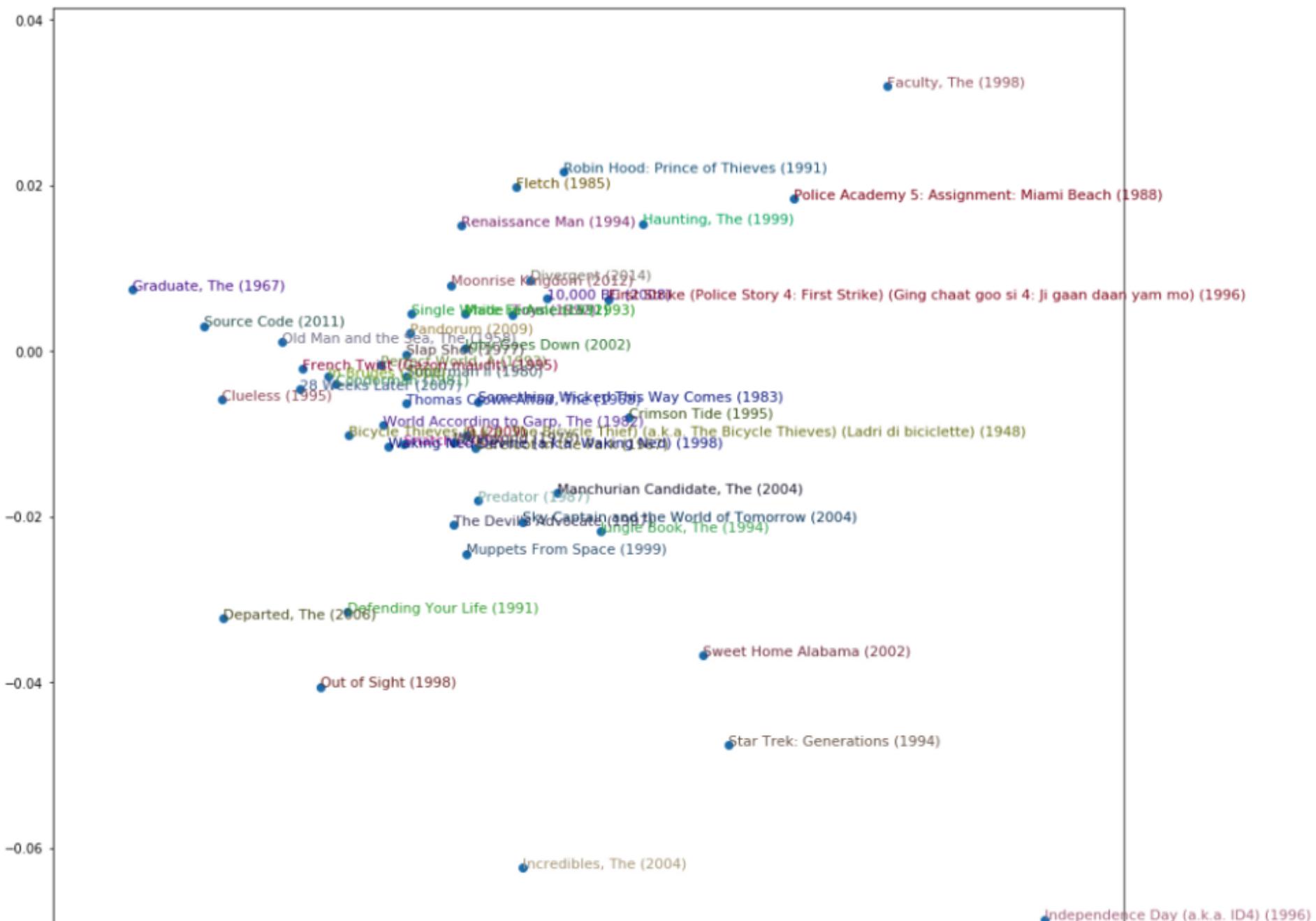
[(0.058975246, 'Bonfire of the Vanities (1990)'),
(0.055992026, '2001: A Space Odyssey (1968)'),
(0.054682467, 'Tank Girl (1995)'),
(0.054429606, 'Purple Rose of Cairo, The (1985'),
...
]

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

[(-0.1064609, 'Lord of the Rings: The Return of the King,
The (2003)'),
(-0.090635143, 'Aladdin (1992)'),
(-0.089208141, 'Star Wars: Episode V - The Empire Strikes
Back (1980)'),
(-0.088854566, 'Star Wars: Episode IV - A New Hope
(1977)'),
...
]
```

Plot

```
idxs = np.random.choice(len(topMovies), 50, replace=False)
X = fac0[idxs]
Y = fac1[idxs]
plt.figure(figsize=(15,15))
plt.scatter(X, Y)
for i, x, y in zip(topMovies[idxs], X, Y):
    plt.text(x,y,movie_names[i],
    color=np.random.rand(3)*0.7, fontsize=11)
plt.show()
```





What actually happens when you say `learn.fit` ?

Entity Embeddings of Categorical Variables [24:42]

The second paper to talk about categorical embeddings. FIG. 1. caption should sound familiar as they talk about how entity embedding layers are equivalent to one-hot encoding followed by a matrix multiplication.

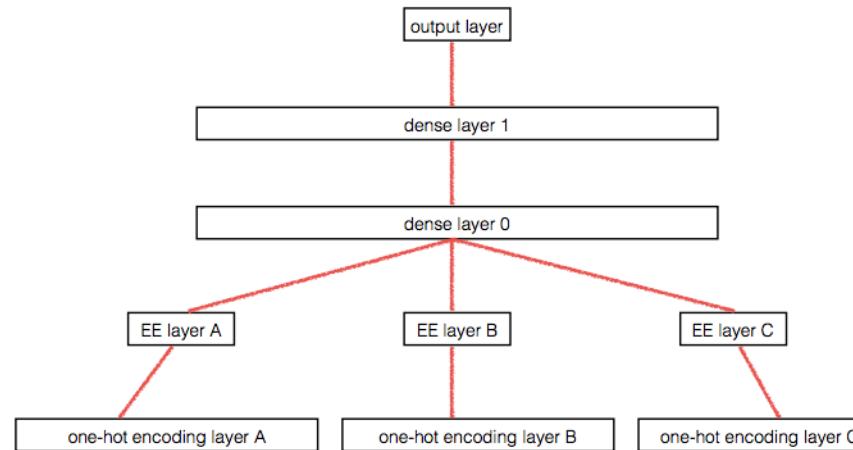


FIG. 1. Illustration that entity embedding layers are equivalent to extra layers on top of each one-hot encoded input.

The interesting thing they did was, they took the entity embeddings trained by a neural network, replaced each categorical variable with the learned entity embeddings, then fed that into Gradient Boosting Machine (GBM), Random Forest (RF), and KNN—which reduced the error to something almost as good as neural network (NN). This is a great way to give the power of neural net within your organization without forcing others to learn deep learning because they can continue to use what they currently use and use the embeddings as input. GBM and RF train much faster than NN.

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

They also plotted the embeddings of states in Germany which interestingly (“whackingly enough” as Jeremy would call it) resembled an actual map.

They also plotted the distances of stores in physical space and embedding space—which showed a beautiful and clear correlation.

There also seems to be correlation between days of the week, or months of the year. Visualizing embeddings can be interesting as it shows you what you expected see or what you didn’t.

A question about Skip-Gram to generate embeddings [31:31]

Skip-Gram is specific to NLP. A good way to turn an unlabeled problem into a labeled problem is to “invent” labels. Word2Vec’s approach was to take a sentence of 11 words, delete the middle word, and replace it with a random word. Then they gave a label 1 to the original sentence; 0 to the fake one, and built a machine learning model to find the fake sentences. As a result, they now have embeddings they can use for other purposes. If you do this as a single matrix multiplier (shallow model) rather than deep neural net, you can train this very quickly—the disadvantage is that it is a less predictive model, but the advantages are that you can train on a very large dataset and more importantly, the resulting embeddings have *linear characteristics* which allow us to add, subtract, or draw nicely. In NLP, we should move past Word2Vec and Glove (i.e. linear based methods) because these embeddings are less predictive. The state of the art language model uses deep RNN.

To learn any kind of feature space, you either need labeled data or you need to invent a fake task [35:45]

- Is one fake task better than another? Not well studied yet.
- Intuitively, we want a task which helps a machine to learn the kinds of relationships that you care about.
- In computer vision, a type of fake task people use is to apply unreal and unreasonable data augmentations.
- If you can’t come up with great fake tasks, just use crappy one—it is often surprising how little you need.

- **Autoencoder** [38:10]—it recently won an insurance claim competition. Take a single policy, run it through neural net, and have it reconstruct itself (make sure that intermediate layers have less activations than the input variable). Basically, it is a task whose input = output which works surprisingly well as a fake task.

In computer vision, you can train on cats and dogs and use it for CT scans. Maybe it might work for language/NLP! (future research)

Rossmann [41:04]

- A way to use test set properly was added to the notebook.
- For more detailed explanations, see Machine Learning course.
- `apply_cats(joined_test, joined)` is used to make sure that the test set and the training set have the same categorical codes.
- Keep track of `mapper` which contains the mean and standard deviation of each continuous column, and apply the same `mapper` to the test set.
- Do not rely on Kaggle public board—rely on your own thoughtfully created validation set.

Going over a good Kernel for Rossmann

- Sunday effect on sales

There is a jump on sales before and after the store closing. 3rd place winner deleted closed store rows before they started any analysis.

Don't touch your data unless you, first of all, analyze to see what you are doing is okay—no assumptions.

Vim tricks [49:12]

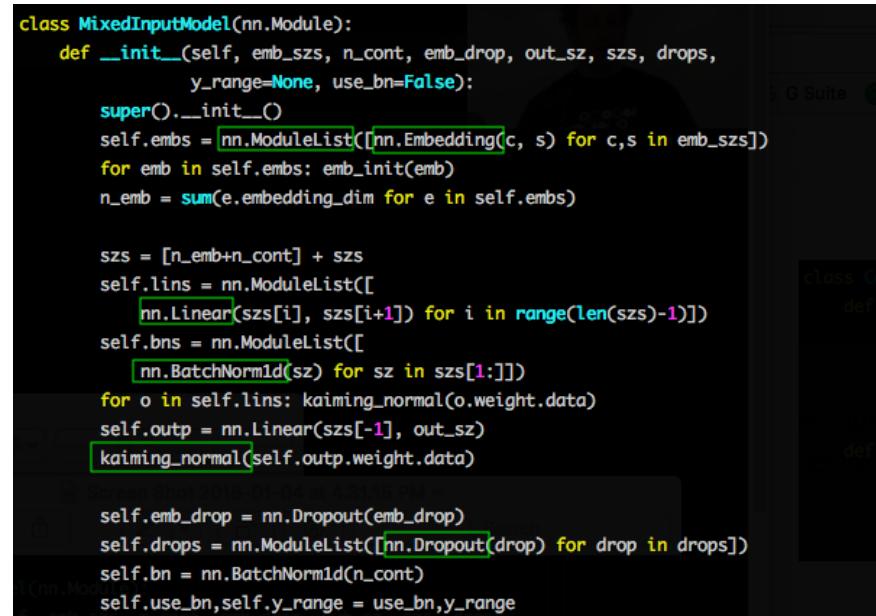
- `:tag ColumnarModelData` will take you to the class definition
- `ctrl +]` will take you to a definition of what's under the cursor
- `ctrl + t` to go back
- `*` to find the usage of what's under the cursor
- You can switch between tabs with `:tabn` and `:tabp`, With `:tabe <filepath>` you can add a new tab; and with a regular `:q` or `:wq` you close a tab. If you map `:tabn` and `:tabp` to your F7/F8 keys you can easily switch between files.

Inside of ColumnarModelData [51:01]

Slowly but surely, what used to be just “magic” start to look familiar. As you can see, `get_learner` returns `Learner` which is fast.ai concept that wraps data and PyTorch model:

```
class ColumnarModelData(ModelData):
    def __init__(self, path, trn_ds, val_ds, bs, test_ds=None, shuffle=True):
        test_dl = DataLoader(test_ds, bs, shuffle=False, num_workers=1) if test_ds is not None
        super().__init__(path, DataLoader(trn_ds, bs, shuffle=shuffle, num_workers=1),
                        DataLoader(val_ds, bs*2, shuffle=False, num_workers=1), test_dl)
    def get_learner(self, emb_szs, n_cont, emb_drop, out_sz, szs, drops,
                   y_range=None, use_bn=False, **kwargs):
        model = MixedInputModel(emb_szs, n_cont, emb_drop, out_sz, szs, drops, y_range, use_bn)
        return StructuredLearner(self, StructuredModel(to_gpu(model)), opt_fn=optim.Adam, **kwargs)
```

Inside of `MixedInputModel` you see how it is creating `Embedding` which we now know more about. `nn.ModuleList` is used to register a list of layers. We will talk about `BatchNorm` next week, but rest, we have seen before.



```
class MixedInputModel(nn.Module):
    def __init__(self, emb_szs, n_cont, emb_drop, out_sz, szs, drops,
                 y_range=None, use_bn=False):
        super().__init__()
        self.embs = nn.ModuleList([nn.Embedding(c, s) for c,s in emb_szs])
        for emb in self.embs: emb_init(emb)
        n_emb = sum(e.embedding_dim for e in self.embs)

        szs = [n_emb+n_cont] + szs
        self.lins = nn.ModuleList([
            nn.Linear(szs[i], szs[i+1]) for i in range(len(szs)-1)])
        self.bns = nn.ModuleList([
            nn.BatchNorm1d(sz) for sz in szs[1:]])
        for o in self.lins: kaiming_normal(o.weight.data)
        self.outp = nn.Linear(szs[-1], out_sz)
        kaiming_normal(self.outp.weight.data)

        self.emb_drop = nn.Dropout(emb_drop)
        self.drops = nn.ModuleList([nn.Dropout(drop) for drop in drops])
        self.bn = nn.BatchNorm1d(n_cont)
        self.use_bn, self.y_range = use_bn, y_range
```

Similarly, we now understand what's going on in the `forward` function.

- call embedding layer with i th categorical variable and concatenate them all together
- put that through dropout
- go through each one of our linear layers, call it, apply relu and dropout

- then final linear layer has a size of 1
- if `y_range` is passed in, apply sigmoid and fit the output within a range (which we learned last week)

```

def forward(self, x_cat, x_cont):
    x = [e(x_cat[:, i]) for i, e in enumerate(self.embs)]
    x = torch.cat(x, 1)
    x2 = self.bn(x_cont)
    x = self.emb_drop(x)
    x = torch.cat([x, x2], 1)
    for l,d,b in zip(self.lins, self.drops, self.bns):
        x = F.relu(l(x))
        if self.use_bn: x = b(x)
        x = d(x)
    x = self.outp(x)
    if self.y_range:
        x = F.sigmoid(x)
        x = x*(self.y_range[1] - self.y_range[0])
        x = x+self.y_range[0]
    return x

```

Stochastic Gradient Descent—SGD [59:56]

To make sure we are totally comfortable with SGD, we will use it to learn `y = ax + b`. If we can solve something with 2 parameters, we can use the same technique to solve 100 million parameters.

```

# Here we generate some fake data
def lin(a,b,x): return a*x+b

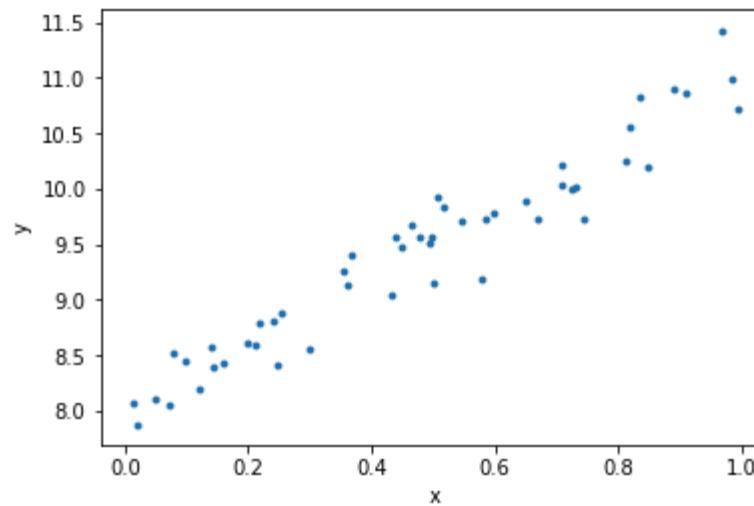
def gen_fake_data(n, a, b):
    x = s = np.random.uniform(0,1,n)

```

```
y = lin(a,b,x) + 0.1 * np.random.normal(0,3,n)
return x, y

x, y = gen_fake_data(50, 3., 8.)

plt.scatter(x,y, s=8); plt.xlabel("x"); plt.ylabel("y");
```



To get started, we need a loss function. This is a regression problem since the output is continuous output, and the most common loss function is the mean squared error (MSE).

Regression—the target output is a real number or a whole vector of real numbers

Classification—the target output is a class label

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

def mse_loss(a, b, x, y): return mse(lin(a,b,x), y)
```

- `y_hat` —predictions

We will make 10,000 more fake data and turn them into PyTorch variables because Jeremy doesn't like taking derivatives and PyTorch can do that for him:

```
x, y = gen_fake_data(10000, 3., 8.)
x,y = V(x),V(y)
```

Then create random weight for `a` and `b`, they are the variables we want to learn, so set `requires_grad=True`.

```
a = V(np.random.randn(1), requires_grad=True)
b = V(np.random.randn(1), requires_grad=True)
```

Then set the learning rate and do 10,000 epoch of full gradient descent (not SGD as each epoch will look at all of the data):

```
learning_rate = 1e-3
for t in range(10000):
    # Forward pass: compute predicted y using operations on
    # Variables
    loss = mse_loss(a,b,x,y)
    if t % 1000 == 0: print(loss.data[0])

    # Computes the gradient of loss with respect to all
    # Variables with requires_grad=True.
    # After this call a.grad and b.grad will be Variables
    # holding the gradient
    # of the loss with respect to a and b respectively
    loss.backward()

    # Update a and b using gradient descent; a.data and
    # b.data are Tensors,
    # a.grad and b.grad are Variables and a.grad.data and
    # b.grad.data are Tensors
    a.data -= learning_rate * a.grad.data
    b.data -= learning_rate * b.grad.data

    # Zero the gradients
    a.grad.data.zero_()
    b.grad.data.zero_()
```

```
In [11]: learning_rate = 1e-3
for t in range(10000):
    # Forward pass: compute predicted y using operations on Variables
    loss = mse_loss(a,b,x,y)
    if t % 1000 == 0: print(loss.data[0])

    # Computes the gradient of loss with respect to all Variables with requires_grad=True.
    # After this call a.grad and b.grad will be Variables holding the gradient
    # of the loss with respect to a and b respectively
    loss.backward()

    # Update a and b using gradient descent; a.data and b.data are Tensors,
    # a.grad and b.grad are Variables and a.grad.data and b.grad.data are Tensors
    a.data -= learning_rate * a.grad.data
    b.data -= learning_rate * b.grad.data

    # Zero the gradients
    a.grad.data.zero_()
    b.grad.data.zero_()
```

```
89.19391632080078
0.6885505318641663
0.11982045322656631
0.11007291823625565
0.10528462380170822
```

- calculate the loss (remember, `a` and `b` are set to random initially)
- from time to time (every 1000 epochs), print out the loss
- `loss.backward()` will calculate gradients for all variables with `requires_grad=True` and fill in `.grad` property

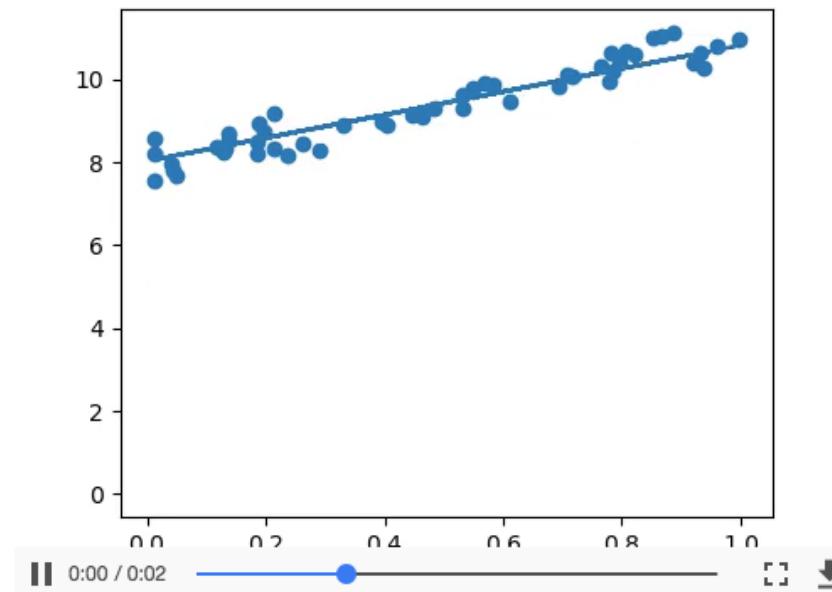
- update `a` to whatever it was minus $\text{LR} * \text{grad}$ (`.data` accesses a tensor inside of a variable)
- when there are multiple loss functions or many output layers contributing to the gradient, PyTorch will add them together. So you need to tell when to set gradients back to zero (`zero_()` in the `_` means that the variable is changed in-place).
- The last 4 lines of code is what is wrapped in `optim.SGD.step` function

Let's do this just Numpy (without PyTorch) [1:07:01]

We actually have to do calculus, but everything else should look similar:

```
x, y = gen_fake_data(50, 3., 8.)  
  
a_guess, b_guess = -1., 1.  
mse_loss(y, a_guess, b_guess, x)  
  
lr=0.01  
def upd():  
    global a_guess, b_guess  
    y_pred = lin(a_guess, b_guess, x)  
    dydb = 2 * (y_pred - y)  
    dyda = x*dydb  
    a_guess -= lr*dyda.mean()  
    b_guess -= lr*dydb.mean()
```

Just for fun, you can use `matplotlib.animation.FuncAnimation` to animate:



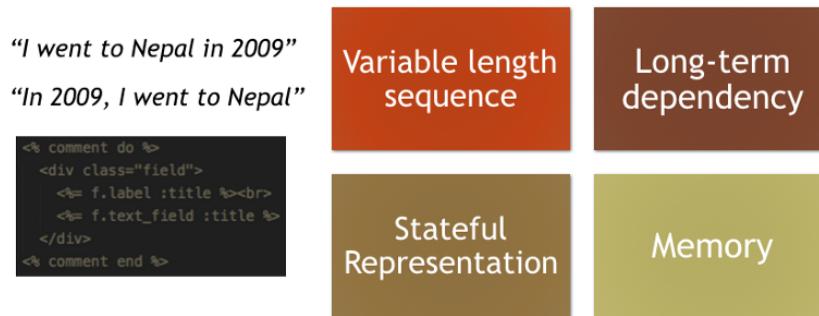
Tip: Fast.ai AMI did not come with `ffmpeg`. So if you see `KeyError:`
`'ffmpeg'`

- Run `print(animation.writers.list())` and print out a list of available MovieWriters
- If `ffmpeg` is among it. Otherwise install it.

Recursive Neural Network—RNN [1:09:16]

Let's learn how to write philosophy like Nietzsche. This is similar to a language model we learned in lesson 4, but this time, we will do it one character at a time. RNN is no different from what we have already learned.

Why we need RNNs



Some examples:

- SwiftKey
- Andrej Karpathy LaTex generator

Basic NN with single hidden layer

All shapes are activations (an activation is a number that has been calculated by a relu, matrix product, etc.). An arrow is a layer operation (possibly more than one). Check out Machine Learning course lesson 9–11 for creating this from scratch.

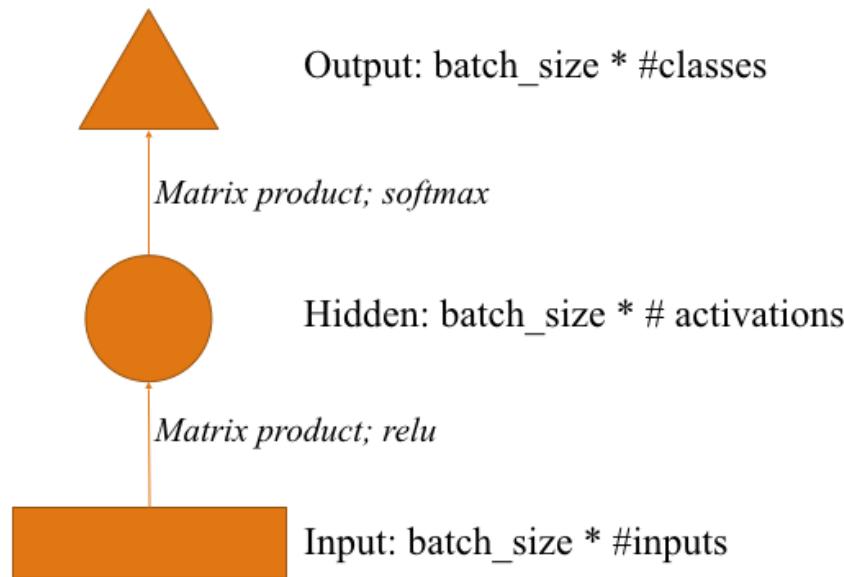
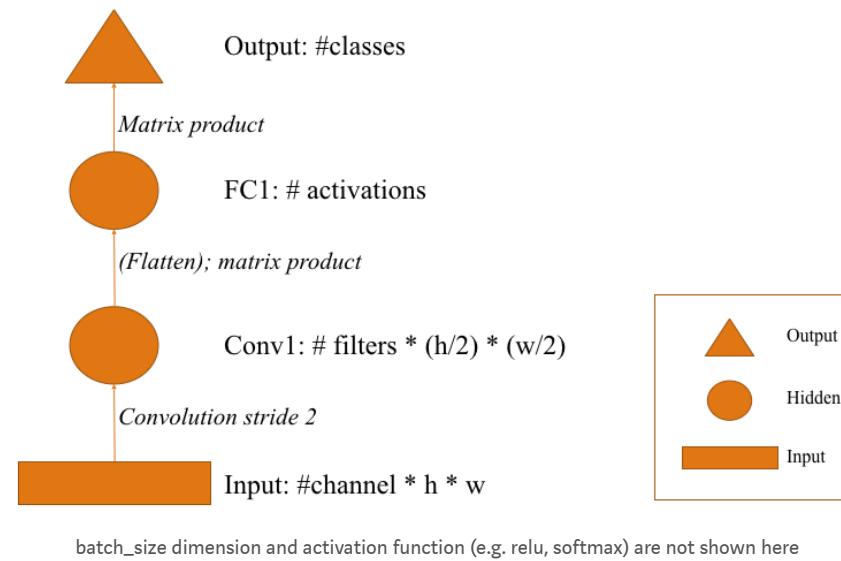


Image CNN with single dense hidden layer

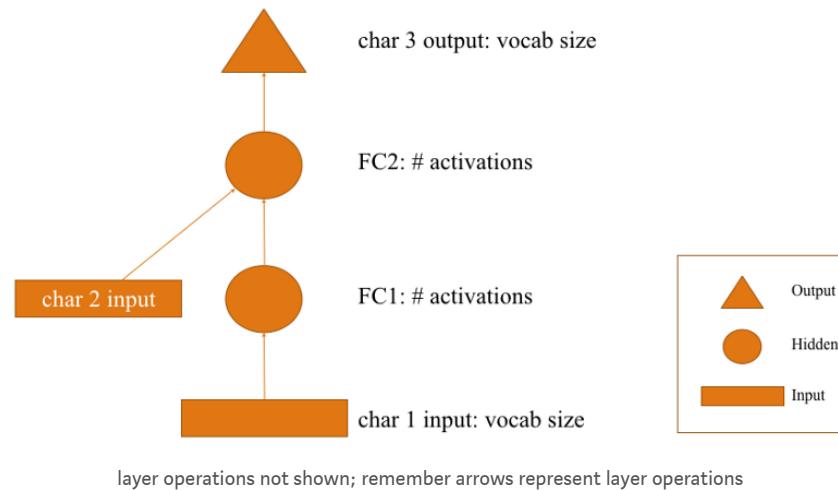
We will cover how to flatten a layer next week more, but the main method is called “adaptive max pooling”—where we average across the height and the width and turn it into a vector.



Predicting char 3 using chars 1 & 2 [1:18:04]

We are going to implement this one for NLP.

- Input can be one-hot-encoded character (length of vector = # of unique characters) or a single integer and pretend it is one-hot-encoded by using an embedding layer.
- The difference from the CNN one is that then char 2 inputs gets added.



Let's implement this without torchtext or fast.ai library so we can see.

- `set` will return all unique characters.

```
text = open(f'{PATH}nietzsche.txt').read()
print(text[:400])

'PREFACE\n\nSUPPOSING that Truth is a woman--what then? Is
there not ground\nfor suspecting that all philosophers, in
so far as they have been\nDogmatists, have failed to
understand women--that the terrible\nseriousness and clumsy
importunity with which they have usually paid\ntheir
addresses to Truth, have been unskilled and unseemly methods
for\nwinning a woman? Certainly she has never allowed
herself '
```

```
chars = sorted(list(set(text)))
vocab_size = len(chars)+1
print('total chars:', vocab_size)
```

```
total chars: 85
```

- Always good to put a null or an empty character for padding.

```
chars.insert(0, "\0")
```

Mapping of every character to a unique ID, and a unique ID to a character

```
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))
```

Now we can represent the text with its ID's:

```
idx = [char_indices[c] for c in text]
idx[:10]

[40, 42, 29, 30, 25, 27, 29, 1, 1, 1]
```

Question: Character based model vs. word based model [1:22:30]

- Generally, you want to combine character level model and word level model (e.g. for translation).
- Character level model is useful when a vocabulary contains unusual words—which word level model will just treat as “unknown”. When you see a word you have not seen before, you can use a character level model.
- There is also something in between that is called Byte Pair Encoding (BPE) which looks at n-gram of characters.

Create inputs [1:23:48]

```
cs = 3
c1_dat = [idx[i] for i in range(0, len(idx)-cs, cs)]
c2_dat = [idx[i+1] for i in range(0, len(idx)-cs, cs)]
c3_dat = [idx[i+2] for i in range(0, len(idx)-cs, cs)]
c4_dat = [idx[i+3] for i in range(0, len(idx)-cs, cs)]
```

Note that `c1_dat[n+1] == c4_dat[n]` since we are skipping by 3 (the third argument of `range`)

```
x1 = np.stack(c1_dat)
x2 = np.stack(c2_dat)
x3 = np.stack(c3_dat)
y = np.stack(c4_dat)
```

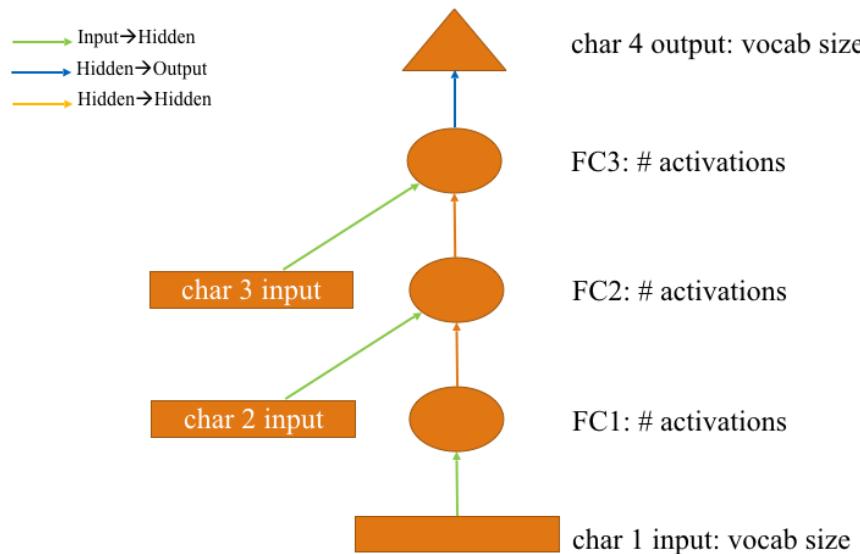
`x`'s are our inputs, `y` is our target value.

Build a model [1:26:08]

```
n_hidden = 256  
n_fac = 42
```

- `n_hiddein` —“# activations” in the diagram.
- `n_fac` —the size of the embedding matrix.

Here is the updated version of the previous diagram. Notice that now arrows are colored. All the arrows with the same color will use the same weight matrix. The idea here is that a character would not have different meaning (semantically or conceptually) depending on whether it is the first, the second, or the third item in a sequence, so treat them the same.



```

class Char3Model(nn.Module):
    def __init__(self, vocab_size, n_fac):
        super().__init__()

        self.e = nn.Embedding(vocab_size, n_fac)

        self.l_in = nn.Linear(n_fac, n_hidden)
        self.l_hidden = nn.Linear(n_hidden, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)

    def forward(self, c1, c2, c3):
        in1 = F.relu(self.l_in(self.e(c1)))
        in2 = F.relu(self.l_in(self.e(c2)))
        in3 = F.relu(self.l_in(self.e(c3)))

        h = V(torch.zeros(in1.size()).cuda())
        h = F.tanh(self.l_hidden(h+in1))
        h = F.tanh(self.l_hidden(h+in2))
        h = F.tanh(self.l_hidden(h+in3))

```

```
    return F.log_softmax(self.l_out(h))
```

```
class Char3Model(nn.Module):
    def __init__(self, vocab_size, n_fac):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)

        # The 'green arrow' from our diagram - the layer operation from input to hidden
        self.l_in = nn.Linear(n_fac, n_hidden)

        # The 'orange arrow' from our diagram - the layer operation from hidden to hidden
        self.l_hidden = nn.Linear(n_hidden, n_hidden)

        # The 'blue arrow' from our diagram - the layer operation from hidden to output
        self.l_out = nn.Linear(n_hidden, vocab_size)

    def forward(self, c1, c2, c3):
        in1 = F.relu(self.l_in(self.e(c1)))
        in2 = F.relu(self.l_in(self.e(c2)))
        in3 = F.relu(self.l_in(self.e(c3)))

        h = V(torch.zeros(in1.size()).cuda())
        h = F.tanh(self.l_hidden(h+in1))
        h = F.tanh(self.l_hidden(h+in2))
        h = F.tanh(self.l_hidden(h+in3))

    return F.log_softmax(self.l_out(h))
```

Video [1:27:57]

- [1:29:58] It is important that this `l_hidden` uses a square weight matrix whose size matches the output of `l_in`. Then `h` and `in2` will be the same shape allowing us to sum them together as you see in `self.l_hidden(h+in2)`
- `V(torch.zeros(in1.size()).cuda())` is only there to make the three lines identical to make it easier to put in a for loop later.

```
md = ColumnarModelData.from_arrays('. ', [-1],
np.stack([x1, x2, x3], axis=1), y, bs=512)
```

We will reuse `ColumnarModelData` [1:32:20]. If we stack `x1`, `x2`, and `x3`, we will get `c1`, `c2`, `c3` in the `forward` method.

`ColumnarModelData.from_arrays` will come in handy when you want to train a model in raw-er approach, what you put in `[x1, x2, x3]`, you will get back in `def forward(self, c1, c2, c3)`

```
m = Char3Model(vocab_size, n_fac).cuda()
```

- We create a standard PyTorch model (not `Learner`)
- Because it is a standard PyTorch model, don't forget `.cuda`

```
it = iter(md.trn_dl)
*xs, yt = next(it)
t = m(*V(xs))
```

- `iter` to grab an iterator
- `next` returns a mini-batch
- “Variabize” the `xs` tensor, and put it through the model—which will give us 512x85 tensor containing prediction (batch size * unique character)

```
opt = optim.Adam(m.parameters(), 1e-2)
```

- Create a standard PyTorch optimizer—for which you need to pass in a list of things to optimize, which is returned by `m.parameters()`

```
fit(m, md, 1, opt, F.nll_loss)
set_lrs(opt, 0.001)
fit(m, md, 1, opt, F.nll_loss)
```

- We do not find a learning rate finder and SGDR because we are not using `Learner`, so we would need to manually do learning rate annealing (set LR a little bit lower)

Testing a model [1:35:58]

```
def get_next(inp):
    idxs = T(np.array([char_indices[c] for c in inp]))
    p = m(*VV(idxs))
    i = np.argmax(to_np(p))
    return chars[i]
```

This function takes three characters and return what the model predict as the fourth. Note: `np.argmax` returns index of the maximum values.

```
get_next('y. ')
'T'

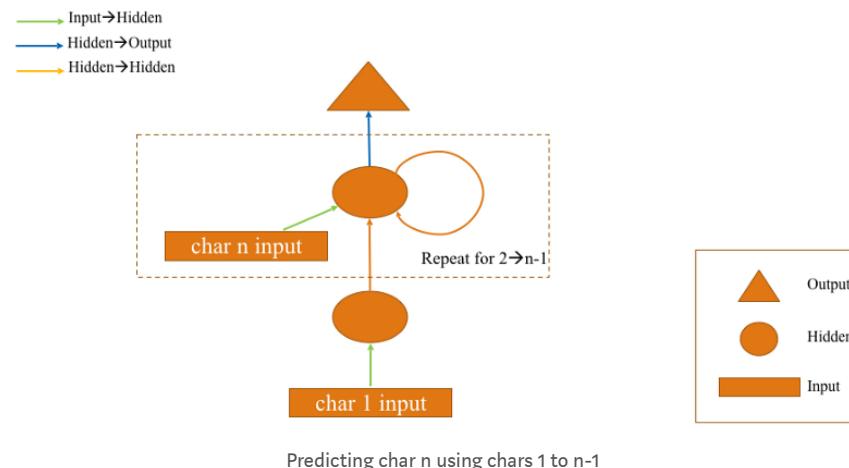
get_next('ppl')
'e'

get_next(' th')
'e'

get_next('and')
'
```

Let's create our first RNN [1:37:45]

We can simplify the previous diagram as below:



Let's implement this. This time, we will use the first 8 characters to predict the 9th. Here is how we create inputs and output just like the last time:

```
cs = 8

c_in_dat = [[idx[i+j] for i in range(cs)] for j in
range(len(idx)-cs)]

c_out_dat = [idx[j+cs] for j in range(len(idx)-cs)]

xs = np.stack(c_in_dat, axis=0)

y = np.stack(c_out_dat)

xs[:cs,:cs]
array([[40, 42, 29, 30, 25, 27, 29, 1],
       [42, 29, 30, 25, 27, 29, 1, 1],
       [29, 30, 25, 27, 29, 1, 1, 1],
       [30, 25, 27, 29, 1, 1, 1, 43],
```

```
[25, 27, 29, 1, 1, 1, 43, 45],
[27, 29, 1, 1, 1, 43, 45, 40],
[29, 1, 1, 1, 43, 45, 40, 40],
[1, 1, 1, 43, 45, 40, 40, 39]])
```

```
y[:cs]
array([1, 1, 43, 45, 40, 40, 39, 43])
```

Notice that they are overlaps (i.e. 0–7 to predict 8, 1–8 to predict 9).

```
val_idx = get_cv_idxs(len(idx)-cs-1)
md = ColumnarModelData.from_arrays('.', val_idx, xs, y,
bs=512)
```

Create the model [1:43:03]

```
class CharLoopModel(nn.Module):
    # This is an RNN!
    def __init__(self, vocab_size, n_fac):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
        self.l_in = nn.Linear(n_fac, n_hidden)
        self.l_hidden = nn.Linear(n_hidden, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)

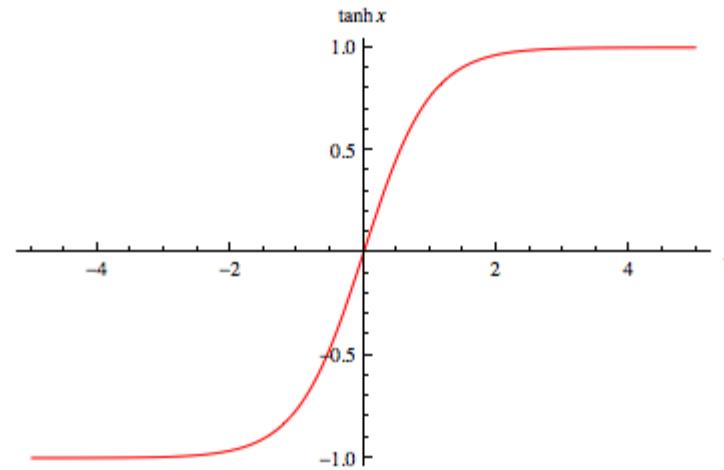
    def forward(self, *cs):
        bs = cs[0].size(0)
        h = V(torch.zeros(bs, n_hidden).cuda())
        for c in cs:
            inp = F.relu(self.l_in(self.e(c)))
            h = F.tanh(self.l_hidden(h+inp))

    return F.log_softmax(self.l_out(h), dim=-1)
```

Most of the code is the same as before. You will notice that there is one `for` loop in `forward` function.

Hyperbolic Tangent (Tanh) [1:43:43]

It is a sigmoid that is offset. It is common to use hyperbolic tanh in the hidden state to hidden state transition because it stops it from flying off too high or too low. For other purposes, relu is more common.



This now is a quite deep network as it uses 8 characters instead of 2.
And as networks get deeper, they become harder to train.

```
m = CharLoopModel(vocab_size, n_fac).cuda()
opt = optim.Adam(m.parameters(), 1e-2)
fit(m, md, 1, opt, F.nll_loss)
set_lrs(opt, 0.001)
fit(m, md, 1, opt, F.nll_loss)
```

Adding vs. Contatenating

We now will try something else for `self.l_hidden(h+inp)` [1:46:04].

The reason is that the input state and the hidden state are qualitatively different. Input is the encoding of a character, and h is an encoding of series of characters. So adding them together, we might lose information. Let's concatenate them instead. Don't forget to change the input to match the shape (`n_fac+n_hidden` instead of `n_fac`).

```
class CharLoopConcatModel(nn.Module):
    def __init__(self, vocab_size, n_fac):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
        self.l_in = nn.Linear(n_fac+n_hidden, n_hidden)
        self.l_hidden = nn.Linear(n_hidden, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)

    def forward(self, *cs):
        bs = cs[0].size(0)
        h = V(torch.zeros(bs, n_hidden).cuda())
        for c in cs:
            inp = torch.cat((h, self.e(c)), 1)
            inp = F.relu(self.l_in(inp))
            h = F.tanh(self.l_hidden(inp))

    return F.log_softmax(self.l_out(h), dim=-1)
```

This gives some improvement.

RNN with PyTorch [1:48:47]

PyTorch will write the `for` loop automatically for us and also the linear input layer.

```
class CharRnn(nn.Module):
    def __init__(self, vocab_size, n_fac):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
        self.rnn = nn.RNN(n_fac, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)

    def forward(self, *cs):
        bs = cs[0].size(0)
        h = V(torch.zeros(1, bs, n_hidden))
        inp = self.e(torch.stack(cs))
        outp,h = self.rnn(inp, h)

    return F.log_softmax(self.l_out(outp[-1]), dim=-1)
```

- For reasons that will become apparent later on, `self.rnn` will return not only the output but also the hidden state.
- The minor difference in PyTorch is that `self.rnn` will append a new hidden state to a tensor instead of replacing (in other words, it will give back all ellipses in the diagram). We only want the final one so we do `outp[-1]`

```
m = CharRnn(vocab_size, n_fac).cuda()
opt = optim.Adam(m.parameters(), 1e-3)
```

```
ht = v(torch.zeros(1, 512,n_hidden))
outp, hn = m.rnn(t, ht)
outp.size(), hn.size()

(torch.Size([8, 512, 256]), torch.Size([1, 512, 256]))
```

In PyTorch version, a hidden state is rank 3 tensor `h = v(torch.zeros(1, bs, n_hidden))` (in our version, it was rank 2 tensor [1:51:58]. We will learn more about this later, but it turns out you can have a second RNN that goes backwards. The idea is that it is going to be better at finding relationships that go backwards—it is called “bi-directional RNN”. Also you can have an RNN feeds to an RNN which is called “multi layer RNN”. For these RNN’s, you will need the additional axis in the tensor to keep track of additional layers of hidden state. For now, we will just have 1 there, and get back 1.

Test the model

```
def get_next(inp):
    idxs = T(np.array([char_indices[c] for c in inp]))
    p = m(*VV(idxs))
    i = np.argmax(to_np(p))
    return chars[i]

def get_next_n(inp, n):
    res = inp
    for i in range(n):
        c = get_next(inp)
        res += c
        inp = inp[1:]+c
    return res
```

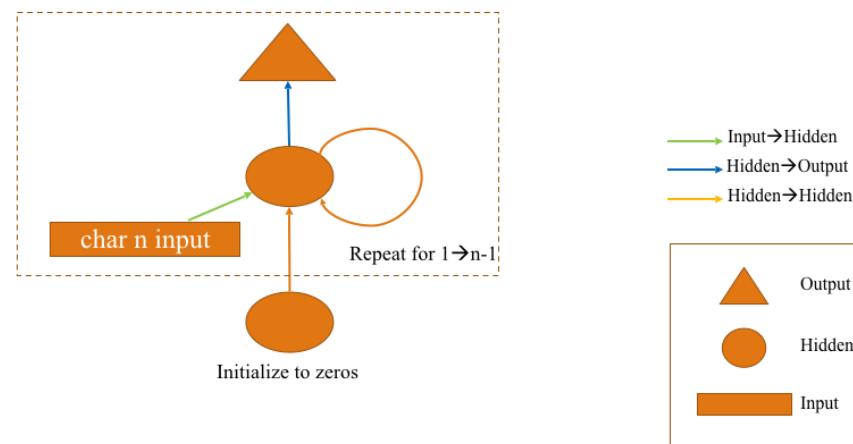
```
get_next_n('for thos', 40)
'for those the same the same the same th'
```

This time, we loop `n` times calling `get_next` each time, and each time we will replace our input by removing the first character and adding the character we just predicted.

For an interesting homework, try writing your own `nn.RNN` “`JeremysRNN`” without looking at PyTorch source code.

Multi-output [1:55:31]

From the last diagram, we can simplify even further by treating char 1 the same as char 2 to n-1. You notice the triangle (the output) also moved inside of the loop, in other words, we create a prediction after each character.



One of the reasons we may want to do this is the redundancies we had seen before:

```
array([[40, 42, 29, 30, 25, 27, 29, 1],  
       [42, 29, 30, 25, 27, 29, 1, 1],  
       [29, 30, 25, 27, 29, 1, 1, 1],  
       [30, 25, 27, 29, 1, 1, 1, 43],  
       [25, 27, 29, 1, 1, 1, 43, 45],  
       [27, 29, 1, 1, 1, 43, 45, 40],  
       [29, 1, 1, 1, 43, 45, 40, 40],  
       [1, 1, 1, 43, 45, 40, 40, 39]])
```

We can make it more efficient by taking **non-overlapping** sets of character this time. Because we are doing multi-output, for an input char 0 to 7, the output would be the predictions for char 1 to 8.

```
xs[:cs,:cs]  
  
array([[40, 42, 29, 30, 25, 27, 29, 1],  
       [1, 1, 43, 45, 40, 40, 39, 43],  
       [33, 38, 31, 2, 73, 61, 54, 73],  
       [2, 44, 71, 74, 73, 61, 2, 62],  
       [72, 2, 54, 2, 76, 68, 66, 54],  
       [67, 9, 9, 76, 61, 54, 73, 2],  
       [73, 61, 58, 67, 24, 2, 33, 72],  
       [2, 73, 61, 58, 71, 58, 2, 67]])  
  
ys[:cs,:cs]  
array([[42, 29, 30, 25, 27, 29, 1, 1],  
       [1, 43, 45, 40, 40, 39, 43, 33],  
       [38, 31, 2, 73, 61, 54, 73, 2],  
       [44, 71, 74, 73, 61, 2, 62, 72],  
       [2, 54, 2, 76, 68, 66, 54, 67],
```

```
[ 9,  9, 76, 61, 54, 73,  2, 73],  
[61, 58, 67, 24,  2, 33, 72,  2],  
[73, 61, 58, 71, 58,  2, 67, 68]])
```

This will not make our model any more accurate, but we can train it more efficiently.

```
class CharSeqRnn(nn.Module):  
    def __init__(self, vocab_size, n_fac):  
        super().__init__()  
        self.e = nn.Embedding(vocab_size, n_fac)  
        self.rnn = nn.RNN(n_fac, n_hidden)  
        self.l_out = nn.Linear(n_hidden, vocab_size)  
  
    def forward(self, *cs):  
        bs = cs[0].size(0)  
        h = V(torch.zeros(1, bs, n_hidden))  
        inp = self.e(torch.stack(cs))  
        outp,h = self.rnn(inp, h)  
        return F.log_softmax(self.l_out(outp), dim=-1)
```

Notice that we are no longer doing `outp[-1]` since we want to keep all of them. But everything else is identical. One complexity[2:00:37] is that we want to use the negative log-likelihood loss function as before, but it expects two rank 2 tensors (two mini-batches of vectors). But here, we have rank 3 tensor:

- 8 characters (time steps)
- 84 probabilities
- for 512 minibatch

Let's write a custom loss function [2:02:10]:

```
def nll_loss_seq(inp, targ):
    sl,bs,nh = inp.size()
    targ = targ.transpose(0,1).contiguous().view(-1)
    return F.nll_loss(inp.view(-1,nh), targ)
```

- `F.nll_loss` is the PyTorch loss function.
- Flatten our inputs and targets.
- Transpose the first two axes because PyTorch expects 1. sequence length (how many time steps), 2. batch size, 3. hidden state itself.
`yt.size()` is 512 by 8, whereas `sl, bs` is 8 by 512.
- PyTorch does not generally actually shuffle the memory order when you do things like ‘transpose’, but instead it keeps some internal metadata to treat it as if it is transposed. When you transpose a matrix, PyTorch just updates the metadata . If you ever see an error that says “this tensor is not continuous” , add `.contiguous()` after it and error goes away.
- `.view` is same as `np.reshape` . `-1` indicates as long as it needs to be.

```
fit(m, md, 4, opt, null_loss_seq)
```

Remember that `fit(...)` is the lowest level fast.ai abstraction that implements the training loop. So all the arguments are standard PyTorch things except for `md` which is our model data object which wraps up the test set, the training set, and the validation set.

Question [2:06:04]: Now that we put a triangle inside of the loop, do we need a bigger sequence size?

- If we have a short sequence like 8, the first character has nothing to go on. It starts with an empty hidden state of zeros.
- We will learn how to avoid that problem next week.
- The basic idea is “why should we reset the hidden state to zeros every time?” (see code below). If we can line up these mini-batches somehow so that the next mini-batch joins up correctly representing the next letter in Nietzsche’s works, then we can move `h = V(torch.zeros(1, bs, n_hidden))` to the constructor.

```
class CharSeqRnn(nn.Module):  
    def __init__(self, vocab_size, n_fac):  
        super().__init__()  
        self.e = nn.Embedding(vocab_size, n_fac)  
        self.rnn = nn.RNN(n_fac, n_hidden)  
        self.l_out = nn.Linear(n_hidden, vocab_size)  
  
    def forward(self, *cs):  
        bs = cs[0].size(0)  
        h = V(torch.zeros(1, bs, n_hidden))  
        inp = self.e(torch.stack(cs))  
        outp,h = self.rnn(inp, h)  
        return F.log_softmax(self.l_out(outp), dim=-1)
```

Gradient Explosion [2:08:21]

`self.rnn(inp, h)` is a loop applying the same matrix multiply again and again. If that matrix multiply tends to increase the activations each time, we are effectively doing that to the power of 8—we call this a gradient explosion. We want to make sure the initial `l_hidden` will not cause our activations on average to increase or decrease.

A nice matrix that does exactly that is called identity matrix:

$$I_1 = [1], I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \dots, I_n = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

We can overwrite the randomly initialized hidden-hidden weight with an identity matrix:

```
m.rnn.weight_hh_10.data.copy_(torch.eye(n_hidden))
```

This was introduced by Geoffrey Hinton et. al. in 2015 (A Simple Way to Initialize Recurrent Networks of Rectified Linear Units)—after RNN has been around for decades. It works very well, and you can use higher learning rate since it is well behaved.



Hiromi Suenaga [Follow](#)
Jan 11 · 34 min read

Deep Learning 2: Part 1 Lesson 7

My personal notes from fast.ai course. These notes will continue to be updated and improved as I continue to review the course to “really” understand it. Much appreciation to Jeremy and Rachel who gave me this opportunity to learn.

. . .

Lesson 7

The theme of Part 1 is:

- classification and regression with deep learning
- identifying and learning best and established practices
- focus is on classification and regression which is predicting “a thing” (e.g. a number, a small number of labels)

Part 2 of the course:

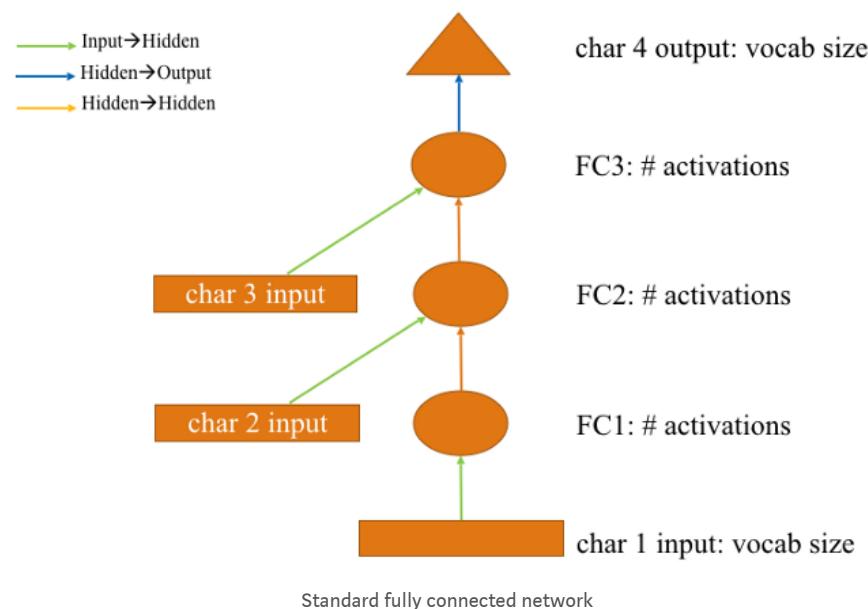
- focus is on generative modeling which means predicting “lots of things”—for example, creating a sentence as in neural translation, image captioning, or question answering while creating an image

such as in style transfer, super-resolution, segmentation and so forth.

- not as much best practices but a little more speculative from recent papers that may not be fully tested.

Review of Char3Model [02:49]

Reminder: RNN is not in any way different or unusual or magical—just a standard fully connected network.



- Arrows represent one or more layer operations—generally speaking a linear followed by a non-linear function, in this case matrix multiplications followed by `relu` or `tanh`

- Arrows of the same color represent exactly the same weight matrix being used.
- One slight difference from previous is that there are inputs coming in at the second and third layers. We tried two approaches—concatenating and adding these inputs to the current activations.

```
class Char3Model(nn.Module):  
    def __init__(self, vocab_size, n_fac):  
        super().__init__()  
        self.e = nn.Embedding(vocab_size, n_fac)  
  
        # The 'green arrow' from our diagram  
        self.l_in = nn.Linear(n_fac, n_hidden)  
  
        # The 'orange arrow' from our diagram  
        self.l_hidden = nn.Linear(n_hidden, n_hidden)  
  
        # The 'blue arrow' from our diagram  
        self.l_out = nn.Linear(n_hidden, vocab_size)  
  
    def forward(self, c1, c2, c3):  
        in1 = F.relu(self.l_in(self.e(c1)))  
        in2 = F.relu(self.l_in(self.e(c2)))  
        in3 = F.relu(self.l_in(self.e(c3)))  
  
        h = V(torch.zeros(in1.size()).cuda())  
        h = F.tanh(self.l_hidden(h+in1))  
        h = F.tanh(self.l_hidden(h+in2))  
        h = F.tanh(self.l_hidden(h+in3))  
  
        return F.log_softmax(self.l_out(h))
```

- By using `nn.Linear` we get both the weight matrix and the bias vector wrapped up for free for us.

- To deal with the fact that there is no orange arrow coming in for the first ellipse , we invented an empty matrix

```

class CharLoopModel(nn.Module):
    # This is an RNN!
    def __init__(self, vocab_size, n_fac):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
        self.l_in = nn.Linear(n_fac, n_hidden)
        self.l_hidden = nn.Linear(n_hidden, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)

    def forward(self, *cs):
        bs = cs[0].size(0)
        h = V(torch.zeros(bs, n_hidden).cuda())
        for c in cs:
            inp = F.relu(self.l_in(self.e(c)))
            h = F.tanh(self.l_hidden(h+inp))

    return F.log_softmax(self.l_out(h), dim=-1)

```

- Almost identical except for the `for` loop

```

class CharRnn(nn.Module):
    def __init__(self, vocab_size, n_fac):
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
        self.rnn = nn.RNN(n_fac, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)

    def forward(self, *cs):
        bs = cs[0].size(0)
        h = V(torch.zeros(1, bs, n_hidden))
        inp = self.e(torch.stack(cs))
        outp,h = self.rnn(inp, h)

```

```
return F.log_softmax(self.l_out(outp[-1]), dim=-1)
```

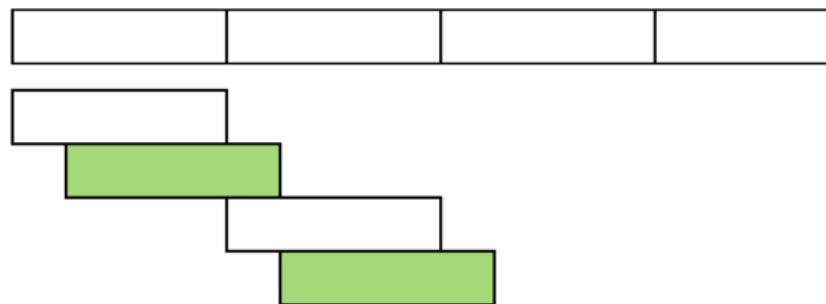
- PyTorch version—`nn.RNN` will create the loop and keep track of `h` as it goes along.
- We are using white section to predict the green character—which seems wasteful as the next section mostly overlaps with the current section.

First $n-1$ characters to predict n th character



- We then tried splitting it into non-overlapping pieces in multi-output model:

First $n-1$ characters to predict $1 - n$ th characters



- In this approach, we are throwing away our h activation after processing each section and started a new one. In order to predict the second character using the first one in the next section, it has nothing to go on but a default activation. Let's not throw away h .

Stateful RNN [08:52]

```
class CharSeqStatefulRnn(nn.Module):
    def __init__(self, vocab_size, n_fac, bs):
        self.vocab_size = vocab_size
        super().__init__()
        self.e = nn.Embedding(vocab_size, n_fac)
        self.rnn = nn.RNN(n_fac, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)
        self.init_hidden(bs)

    def forward(self, cs):
        bs = cs[0].size(0)
        if self.h.size(1) != bs: self.init_hidden(bs)
        outp,h = self.rnn(self.e(cs), self.h)
        self.h = repackage_var(h)
        return F.log_softmax(self.l_out(outp),
                             dim=-1).view(-1, self.vocab_size)
```

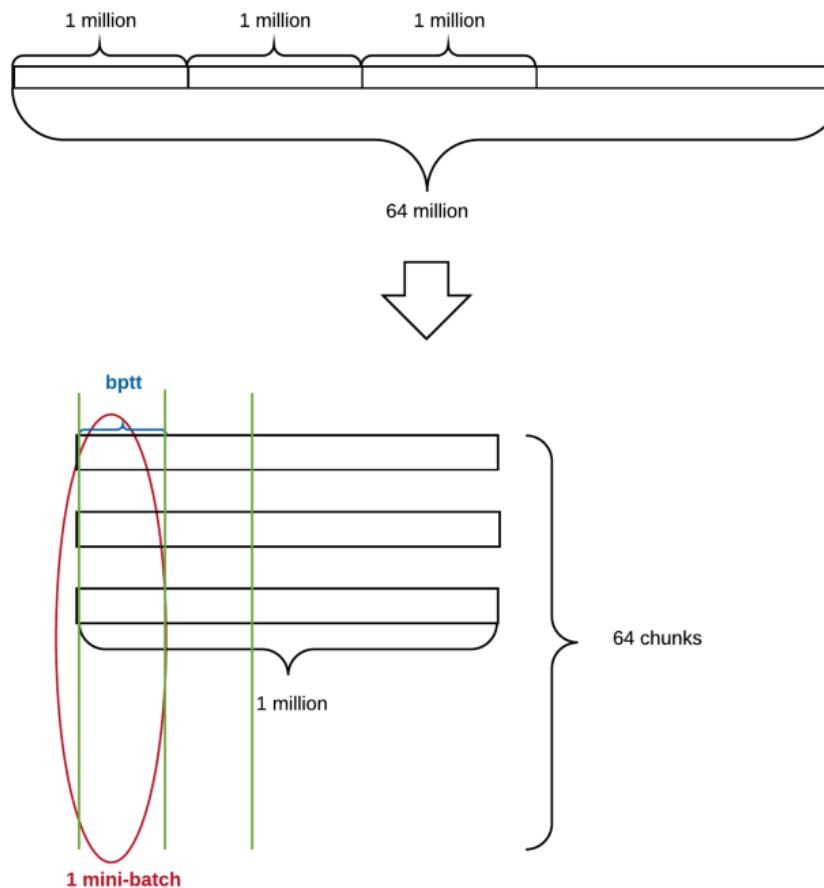
```
def init_hidden(self, bs): self.h = V(torch.zeros(1, bs,
n_hidden))
```

- One additional line in constructor. `self.init_hidden(bs)` sets `self.h` to bunch of zeros.
- **Wrinkle #1 [10:51]**—if we were to simply do `self.h = h`, and we trained on a document that is a million characters long, then the size of unrolled version of the RNN has a million layers (ellipses). One million layer fully connected network is going to be very memory intensive because in order to do a chain rule, we have to multiply one million layers while remembering all one million gradients every batch.
- To avoid this, we tell it to forget its history from time to time. We can still remember the state (the values in our hidden matrix) without remembering everything about how we got there.

```
def repackage_var(h):
    return Variable(h.data) if type(h) == Variable else
tuple(repackage_var(v) for v in h)
```

- Grab the tensor out of `Variable h` (remember, a tensor itself does not have any concept of history), and create a new `Variable` out of that. The new variable has the same value but no history of operations, therefore when it tries to back-propagate, it will stop there.

- `forward` will process 8 characters, it then back propagate through eight layers, keep track of the values in our hidden state, but it will throw away its history of operations. This is called **back-prop through time (bptt)**.
- In other words, after the `for` loop, just throw away the history of operations and start afresh. So we are keeping our hidden state but we are not keeping our hidden state history.
- Another good reason not to back-propagate through too many layers is that if you have any kind of gradient instability (e.g. gradient explosion or gradient vanishing), the more layers you have, the harder the network gets to train (slower and less resilient).
- On the other hand, the longer `bptt` means that you are able to explicitly capture a longer memory and more state.
- **Wrinkle #2 [16:00]**—how to create mini-batches. We do not want to process one section at a time, but a bunch in parallel at a time.
- When we started looking at TorchText for the first time, we talked about how it creates these mini-batches.
- Jeremy said we take a whole long document consisting of the entire works of Nietzsche or all of the IMDB reviews concatenated together, we split this into 64 equal sized chunks (NOT chunks of size 64).



- For a document that is 64 million characters long, each “chunk” will be 1 million characters. We stack them together and now split them by `bptt` — 1 mini-batch consists of 64 by `bptt` matrix.
- The first character of the second chunk(1,000,001th character) is likely be in the middle of a sentence. But it is okay since it only happens once every million characters.

Question: Data augmentation for this kind of dataset? [20:34]

There is no known good way. Somebody recently won a Kaggle competition by doing data augmentation which randomly inserted parts of different rows—something like that may be useful here. But there has not been any recent state-of-the-art NLP papers that are doing this kind of data augmentation.

Question: How do we choose the size of bptt? [21:36]

There are a couple things to think about:

- the first is that mini-batch matrix has a size of `bs` (# of chunks) by `bptt` so your GPU RAM must be able to fit that by your embedding matrix. So if you get CUDA out of memory error, you need reduce one of these.
- If your training is unstable (e.g. your loss is shooting off to NaN suddenly), then you could try decreasing your `bptt` because you have less layers to gradient explode through.
- If it is too slow [22:44], try decreasing your `bptt` because it will do one of those steps at a time. `for` loop cannot be parallelized (for the current version). There is a recent thing called QRNN (Quasi-Recurrent Neural Network) which does parallelize it and we hope to cover in part 2.
- So pick the highest number that satisfies all these.

Stateful RNN & TorchText [23:23]

When using an existing API which expects data to be certain format, you can either change your data to fit that format or you can write your own dataset sub-class to handle the format that your data is already in. Either is fine, but in this case, we will put our data in the format TorchText already support. Fast.ai wrapper around TorchText already has something where you can have a training path and validation path, and one or more text files in each path containing bunch of text that are concatenated together for your language model.

```
from torchtext import vocab, data

from fastai.nlp import *
from fastai.lm_rnn import *

PATH='data/nietzsche/'

TRN_PATH = 'trn/'
VAL_PATH = 'val/'
TRN = f'{PATH}{TRN_PATH}'
VAL = f'{PATH}{VAL_PATH}'

%ls {PATH}
models/ nietzsche.txt trn/ val/

%ls {PATH}trn
trn.txt
```

- Made a copy of Nietzsche file, pasted into training and validation directory. Then deleted the last 20% of the rows from training set, and deleted everything but the last 20% from the validation set [25:15].

- The other benefit of doing it this way is that it seems like it is more realistic to have a validation set that was not a random shuffled set of rows of text, but it was totally separate part of the corpus.
- When you are doing a language model, you do not really need separate files. You can have multiple files but they just get concatenated together anyway.

```
TEXT = data.Field(lower=True, tokenize=list)
bs=64; bptt=8; n_fac=42; n_hidden=256

FILES = dict(train=TRN_PATH, validation=VAL_PATH,
test=VAL_PATH)
md = LanguageModelData.from_text_files(PATH, TEXT, **FILES,
bs=bs, bptt=bptt, min_freq=3)

len(md.trn_dl), md.nt, len(md.trn_ds),
len(md.trn_ds[0].text)
(963, 56, 1, 493747)
```

- In TorchText, we make this thing called `Field` and initially `Field` is just a description of how to go about pre-processing the text.
- `lower` —we told it to lowercase the text
- `tokenize` —Last time, we used a function that splits on whitespace that gave us a word model. This time, we want a character model, so use `list` function to tokenize strings. Remember, in Python, `list('abc')` will return `['a', 'b', 'c']`.

- `bs` : batch size, `bptt` : we renamed `cs` , `n_fac` : size of embedding, `n_hidden` : size of our hidden state
- We do not have a separate test set, so we'll just use validation set for testing
- TorchText randomize the length of `bptt` a little bit each time. It does not always give us exactly 8 characters; 5% of the time, it will cut it in half and add on a small standard deviation to make it slightly bigger or smaller than 8. We cannot shuffle the data since it needs to be contiguous, so this is a way to introduce some randomness.
- Question [31:46]: Does the size remain constant per mini-batch? Yes, we need to do matrix multiplication with `h` weight matrix, so mini-batch size must remain constant. But sequence length can change no problem.
- `len(md.trn_dl)` : length of data loader (i.e. how many mini-batches), `md.nt` : number of tokens (i.e. how many unique things are in the vocabulary)
- Once you run `LanguageModelData.from_text_files` , `TEXT` will contain an extra attribute called `vocab` . `TEXT.vocab.itos` list of unique items in the vocabulary, and `TEXT.vocab.stoi` is a reverse mapping from each item to number.

```
class CharSeqStatefulRnn(nn.Module):  
    def __init__(self, vocab_size, n_fac, bs):  
        self.vocab_size = vocab_size  
        super().__init__()  
        self.e = nn.Embedding(vocab_size, n_fac)
```

```
        self.rnn = nn.RNN(n_fac, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)
        self.init_hidden(bs)

    def forward(self, cs):
        bs = cs[0].size(0)
        if self.h.size(1) != bs: self.init_hidden(bs)
        outp,h = self.rnn(self.e(cs), self.h)
        self.h = repackage_var(h)
        return F.log_softmax(self.l_out(outp),
                             dim=-1).view(-1, self.vocab_size)

    def init_hidden(self, bs): self.h = V(torch.zeros(1, bs,
                                                    n_hidden))
```

- **Wrinkle #3 [33:51]:** Jeremy lied to us when he said that mini-batch size remains constant. It is very likely that the last mini-batch is shorter than the rest unless the dataset is exactly divisible by `bptt` times `bs`. That is why we check whether `self.h` ‘s second dimension is the same as `bs` of the input. If it is not the same, set it back to zero with the input’s `bs`. This happens at the end of the epoch and the beginning of the epoch (setting back to the full batch size).
- **Wrinkle #4 [35:44]:** The last wrinkle is something that slightly sucks about PyTorch and maybe somebody can be nice enough to try and fix it with a PR. Loss functions are not happy receiving a rank 3 tensor (i.e. three dimensional array). There is no particular reason they ought to not be happy receiving a rank 3 tensor (sequence length by batch size by results—so you can just calculate loss for each of the two initial axis). Works for rank 2 or 4, but not 3.

- `.view` will reshape rank 3 tensor into rank 2 of `-1` (however big as necessary) by `vocab_size`. TorchText automatically changes the **target** to be flattened out, so we do not need to do that for actual values (when we looked at a mini-batch in lesson 4, we noticed that it was flattened. Jeremy said we will learn about why later, so later is now.)
- PyTorch (as of 0.3), `log_softmax` requires us to specify which axis we want to do the softmax over (i.e. which axis we want to sum to one). In this case we want to do it over the last axis `dim = -1`.

```
m = CharSeqStatefulRnn(md.nt, n_fac, 512).cuda()
opt = optim.Adam(m.parameters(), 1e-3)

fit(m, md, 4, opt, F.nll_loss)
```

Let's gain more insight by unpacking RNN [42:48]

We remove the use of `nn.RNN` and replace it with `nn.RNNCell`. PyTorch source code looks like the following. You should be able to read and understand (Note: they do not concatenate the input and the hidden state, but they sum them together—which was our first approach):

```
def RNNCell(input, hidden, w_ih, w_hh, b_ih, b_hh):
    return F.tanh(F.linear(input, w_ih, b_ih) +
                  F.linear(hidden, w_hh, b_hh))
```

Question about `tanh` [44:06]: As we have seen last week, `tanh` is forcing the value to be between -1 and 1. Since we are multiplying by this weight matrix again and again, we would worry that `relu` (since it is unbounded) might have more gradient explosion problem. Having said that, you can specify `RNNCell` to use different `nonlineality` whose default is `tanh` and ask it to use `relu` if you wanted to.

```
class CharSeqStatefulRnn2(nn.Module):
    def __init__(self, vocab_size, n_fac, bs):
        super().__init__()
        self.vocab_size = vocab_size
        self.e = nn.Embedding(vocab_size, n_fac)
        self.rnn = nn.RNNCell(n_fac, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)
        self.init_hidden(bs)

    def forward(self, cs):
        bs = cs[0].size(0)
        if self.h.size(1) != bs: self.init_hidden(bs)
        outp = []
        o = self.h
        for c in cs:
            o = self.rnn(self.e(c), o)
            outp.append(o)
        outp = self.l_out(torch.stack(outp))
        self.h = repackage_var(o)
        return F.log_softmax(outp, dim=-1).view(-1,
self.vocab_size)

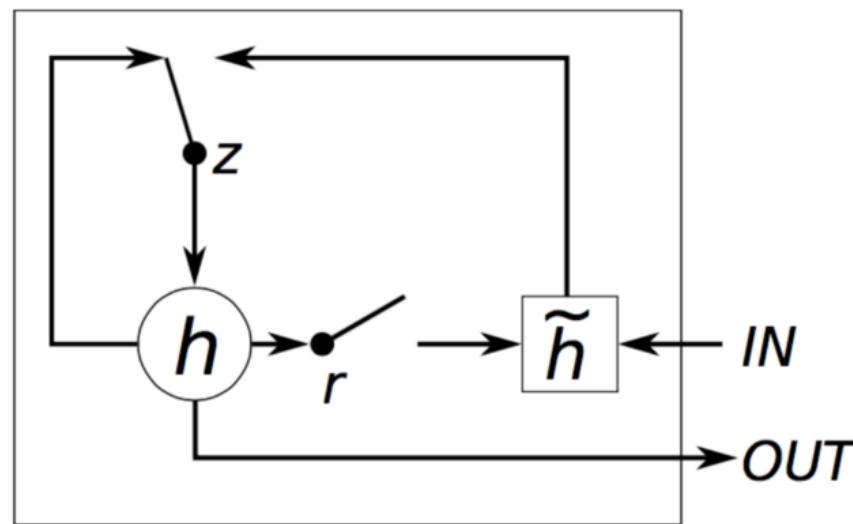
    def init_hidden(self, bs): self.h = V(torch.zeros(1, bs,
n_hidden))
```

- `for` loop is back and append the result of linear function to a list —which in end gets stacked up together.

- fast.ai library actually does exactly this in order to use regularization approaches that are not supported by PyTorch.

Gated Recurrent Unit (GRU) [46:44]

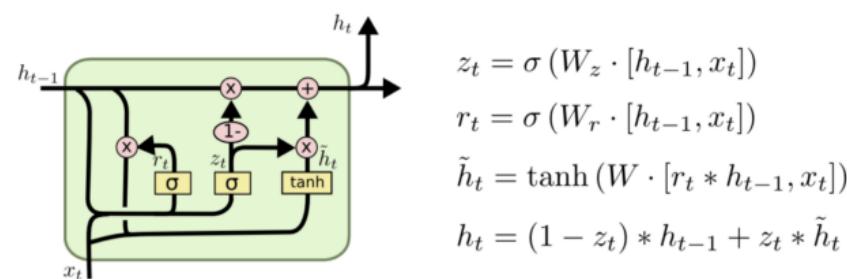
In practice, nobody really uses `RNNCell` since even with `tanh` , gradient explosions are still a problem and we need use low learning rate and small `bptt` to get them to train. So what we do is to replace `RNNCell` with something like `GRUCell` .



<http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-gru-lstm-rnn-with-python-and-theano/>

- Normally, the input gets multiplied by a weight matrix to create new activations `h` and get added to the existing activations straight away. That is not what happens here.

- Input goes into \tilde{h}_t and it doesn't just get added to the previous activations, but the previous activation gets multiplied by r_t (reset gate) which has a value of 0 or 1.
- r_t is calculated as below—matrix multiplication of some weight matrix and the concatenation of our previous hidden state and new input. In other words, this is a little one hidden layer neural net. It gets put through the sigmoid function as well. This mini neural net learns to determine how much of the hidden states to remember (maybe forget it all when it sees a full-stop character—beginning of a new sentence).
- z_t gate (update gate) determines what degree to use \tilde{h}_t (the new input version of hidden states) and what degree to leave the hidden state the same as before.



<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- Linear interpolation

```
def GRUCell(input, hidden, w_ih, w_hh, b_ih, b_hh):
    gi = F.linear(input, w_ih, b_ih)
    gh = F.linear(hidden, w_hh, b_hh)
    i_r, i_i, i_n = gi.chunk(3, 1)
    h_r, h_i, h_n = gh.chunk(3, 1)

    resetgate = F.sigmoid(i_r + h_r)
    inputgate = F.sigmoid(i_i + h_i)
    newgate = F.tanh(i_n + resetgate * h_n)
    return newgate + inputgate * (hidden - newgate)
```

Above is what `GRUCell` code looks like, and our new model that utilize this is below:

```
class CharSeqStatefulGRU(nn.Module):
    def __init__(self, vocab_size, n_fac, bs):
        super().__init__()
        self.vocab_size = vocab_size
        self.e = nn.Embedding(vocab_size, n_fac)
        self.rnn = nn.GRU(n_fac, n_hidden)
        self.l_out = nn.Linear(n_hidden, vocab_size)
        self.init_hidden(bs)

    def forward(self, cs):
        bs = cs[0].size(0)
        if self.h.size(1) != bs: self.init_hidden(bs)
        outp,h = self.rnn(self.e(cs), self.h)
        self.h = repackage_var(h)
        return F.log_softmax(self.l_out(outp),
                             dim=-1).view(-1, self.vocab_size)

    def init_hidden(self, bs): self.h = V(torch.zeros(1, bs,
n_hidden))
```

As a result, we can lower the loss down to 1.36 (`RNNCell` one was 1.54). In practice, GRU and LSTM are what people uses.

Putting it all together: Long Short-Term Memory [54:09]

LSTM has one more piece of state in it called “cell state” (not just hidden state), so if you do use a LSTM, you have to return a tuple of matrices in `init_hidden` (exactly the same size as hidden state):

```
from fastai import sgdr

n_hidden=512

class CharSeqStatefulLSTM(nn.Module):
    def __init__(self, vocab_size, n_fac, bs, nl):
        super().__init__()
        self.vocab_size, self.nl = vocab_size, nl
        self.e = nn.Embedding(vocab_size, n_fac)
        self.rnn = nn.LSTM(n_fac, n_hidden, nl, dropout=0.5)
        self.l_out = nn.Linear(n_hidden, vocab_size)
        self.init_hidden(bs)

    def forward(self, cs):
        bs = cs[0].size(0)
        if self.h[0].size(1) != bs: self.init_hidden(bs)
        outp,h = self.rnn(self.e(cs), self.h)
        self.h = repackage_var(h)
        return F.log_softmax(self.l_out(outp),
dim=-1).view(-1, self.vocab_size)

    def init_hidden(self, bs):
        self.h = (V(torch.zeros(self.nl, bs, n_hidden)),
                  V(torch.zeros(self.nl, bs, n_hidden)))
```

The code is identical to GRU one. The one thing that was added was `dropout` which does dropout after each time step and doubled the hidden layer—in a hope that it will be able to learn more and be resilient as it does so.

Callbacks (specifically SGDR) without Learner class [55:23]

```
m = CharSeqStatefullLSTM(md.nt, n_fac, 512, 2).cuda()
lo = LayerOptimizer(optim.Adam, m, 1e-2, 1e-5)
```

- After creating a standard PyTorch model, we usually do something like `opt = optim.Adam(m.parameters(), 1e-3)`. Instead, we will use fast.ai `LayerOptimizer` which takes an optimizer `optim.Adam`, our model `m`, learning rate `1e-2`, and optionally weight decay `1e-5`.
- A key reason `LayerOptimizer` exists is to do differential learning rates and differential weight decay. The reason we need to use it is that all of the mechanics inside fast.ai assumes that you have one of these. If you want to use callbacks or SGDR in code you are not using the Learner class, you need to use this.
- `lo.opt` returns the optimizer.

```
on_end = lambda sched, cycle: save_model(m,
f'{PATH}models/cyc_{cycle}')
```

```
cb = [CosAnneal(lo, len(md.trn_dl), cycle_mult=2,
on_cycle_end=on_end)]  
  
fit(m, md, 2**4-1, lo.opt, F.nll_loss, callbacks=cb)
```

- When we call `fit`, we can now pass the `LayerOptimizer` and also `callbacks`.
- Here, we use cosine annealing callback—which requires a `LayerOptimizer` object. It does cosine annealing by changing learning rate in side the `lo` object.
- Concept: Create a cosine annealing callback which is going to update the learning rates in the layer optimizer `lo`. The length of an epoch is equal to `len(md.trn_dl)` —how many mini-batches are there in an epoch is the length of the data loader. Since it is doing cosine annealing, it needs to know how often to reset. You can pass in `cycle_mult` in usual way. We can even save our model automatically just like we did with `cycle_save_name` in `Learner.fit`.
- We can do callback at a start of a training, epoch or a batch, or at the end of a training, an epoch, or a batch.
- It has been used for `CosAnneal` (SGDR), and decoupled weight decay (AdamW), loss-over-time graph, etc.

Testing [59:55]

```
def get_next(inp):
    idxs = TEXT.numericalize(inp)
    p = m(VV(idxs.transpose(0,1)))
    r = torch.multinomial(p[-1].exp(), 1)
    return TEXT.vocab.itos[to_np(r)[0]]


def get_next_n(inp, n):
    res = inp
    for i in range(n):
        c = get_next(inp)
        res += c
        inp = inp[1:]+c
    return res


print(get_next_n('for thos', 400))

for those the skemps), or imaginates, though they deceives.
it should so each ourselvess and new present, step
absolutely for the science." the contradity and measuring,
the whole!

293. perhaps, that every life a values of blood of
intercourse when it senses there is unscrupulus, his very
rights, and still impulse, love? just after that thereby how
made with the way anything, and set for harmless philos
```

- In lesson 6, when we were testing `CharRnn` model, we noticed that it repeated itself over and over. `torch.multinomial` used in this new version deals with this problem. `p[-1]` to get the final output (the triangle), `exp` to convert log probability to probability. We then use `torch.multinomial` function which will give us a sample using the given probabilities. If probability is [0, 1, 0, 0] and ask it to give us a sample, it will always return the second item. If it was [0.5, 0, 0.5], it will give the first item 50% of

the time, and second item . 50% of the time (review of multinomial distribution)

- To play around with training character based language models like this, try running `get_next_n` at different levels of loss to get a sense of what it looks like. The example above is at 1.25, but at 1.3, it looks like a total junk.
- When you are playing around with NLP, particularly generative model like this, and the results are kind of okay but not great, do not be disheartened because that means you are actually very VERY nearly there!

Back to computer vision: CIFAR 10 [1:01:58]

CIFAR 10 is an old and well known dataset in academia—well before ImageNet, there was CIFAR 10. It is small both in terms of number of images and size of images which makes it interesting and challenging. You will likely be working with thousands of images rather than one and a half million images. Also a lot of the things we are looking at like in medical imaging, we are looking at a specific area where there is a lung nodule, you are probably looking at 32 by 32 pixels at most.

It also runs quickly, so it is much better to test your algorithms. As Ali Rahini mentioned in NIPS 2017, Jeremy has the concern that many people are not doing carefully tuned and thought-about experiments in deep learning, but instead, they throw lots of GPUs and TPUs or lots of data and consider that a day. It is important to test many versions of your algorithm on dataset like CIFAR 10 rather than ImageNet that takes weeks. MNIST is also good for studies and experiments even though people tend to complain about it.

CIFAR 10 data in image format is available here

```
from fastai.conv_learner import *
PATH = "data/cifar10/"
os.makedirs(PATH, exist_ok=True)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog',
'frog', 'horse', 'ship', 'truck')
stats = (np.array([ 0.4914 ,  0.48216,  0.44653]),  
np.array([ 0.24703,  0.24349,  0.26159]))

def get_data(sz,bs):
    tfms = tfms_from_stats(stats, sz, aug_tfms=
[RandomFlipXY()], pad=sz//8)
    return ImageClassifierData.from_paths(PATH,
val_name='test', tfms=tfms, bs=bs)

bs=256
```

- `classes` —image labels
- `stats` —When we use pre-trained models, you can call `tfms_from_model` which creates the necessary transforms to convert our data set into a normalized dataset based on the means and standard deviations of each channel in the original model that was trained in. Since we are training a model from scratch, we need to tell it the mean and standard deviation of our data to normalize it. Make sure you can calculate the mean and the standard deviation for each channel.
- `tfms` —For CIFAR 10 data augmentation, people typically do horizontal flip and black padding around the edge and randomly

select 32 by 32 area within the padded image.

```
data = get_data(32,bs)  
lr=1e-2
```

From this notebook by our student Kerem Turgutlu:

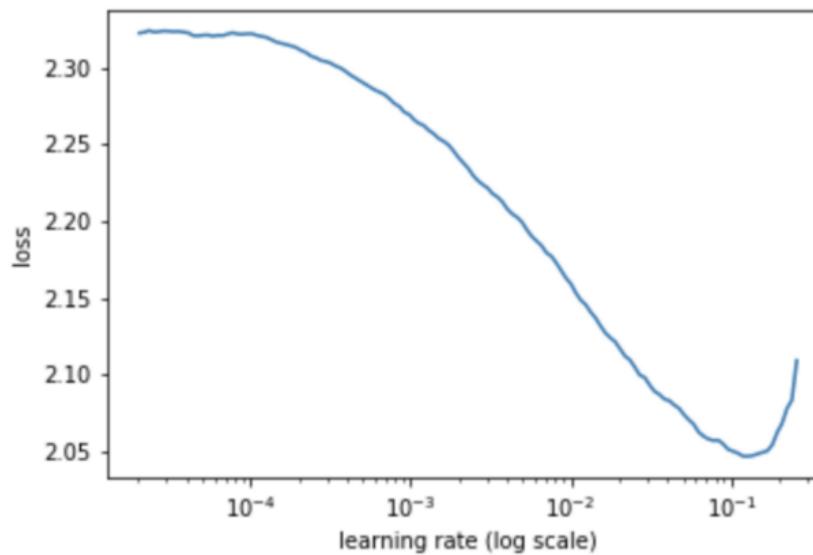
```
class SimpleNet(nn.Module):  
    def __init__(self, layers):  
        super().__init__()  
        self.layers = nn.ModuleList([  
            nn.Linear(layers[i], layers[i + 1]) for i in  
            range(len(layers) - 1)])  
  
    def forward(self, x):  
        x = x.view(x.size(0), -1)  
        for l in self.layers:  
            l_x = l(x)  
            x = F.relu(l_x)  
        return F.log_softmax(l_x, dim=-1)
```

- `nn.ModuleList` —whenever you create a list of layers in PyTorch, you have to wrap it in `ModuleList` to register these as attributes.

```
learn = ConvLearner.from_model_data(SimpleNet([32*32*3,  
40,10]), data)
```

- Now we step up one level of API higher—rather than calling `fit` function, we create a `learn` object *from a custom model*.
`ConfLearner.from_model_data` takes standard PyTorch model and model data object.

```
learn, [o.numel() for o in learn.model.parameters()]  
  
(SimpleNet(  
    (layers): ModuleList(  
        (0): Linear(in_features=3072, out_features=40)  
        (1): Linear(in_features=40, out_features=10)  
    )  
, [122880, 40, 400, 10])  
  
learn.summary()  
  
OrderedDict([('Linear-1',  
    OrderedDict([('input_shape', [-1, 3072]),  
                ('output_shape', [-1, 40]),  
                ('trainable', True),  
                ('nb_params', 122920)])),  
            ('Linear-2',  
    OrderedDict([('input_shape', [-1, 40]),  
                ('output_shape', [-1, 10]),  
                ('trainable', True),  
                ('nb_params', 410)])))  
  
learn.lr_find()  
  
learn.sched.plot()
```



```
%time learn.fit(lr, 2)

A Jupyter Widget

[ 0.        1.7658   1.64148  0.42129]
[ 1.        1.68074   1.57897  0.44131]

CPU times: user 1min 11s, sys: 32.3 s, total: 1min 44s
Wall time: 55.1 s
```

```
%time learn.fit(lr, 2, cycle_len=1)
```

```
A Jupyter Widget
```

```
[ 0.        1.60857   1.51711  0.46631]
[ 1.        1.59361   1.50341  0.46924]
```

```
CPU times: user 1min 12s, sys: 31.8 s, total: 1min 44s
Wall time: 55.3 s
```

With a simple one hidden layer model with 122,880 parameters, we achieved 46.9% accuracy. Let's improve this and gradually build up to a basic ResNet architecture.

CNN [01:12:30]

- Let's replace a fully connected model with a convolutional model. Fully connected layer is simply doing a dot product. That is why the weight matrix is big ($3072 \text{ input} * 40 = 122880$). We are not using the parameters very efficiently because every single pixel in the input has a different weight. What we want to do is a group of 3 by 3 pixels that have particular patterns to them (i.e. convolution).
- We will use a filter with three by three kernel. When there are multiple filters, the output will have additional dimension.

```
class ConvNet(nn.Module):
    def __init__(self, layers, c):
        super().__init__()
        self.layers = nn.ModuleList([
            nn.Conv2d(layers[i], layers[i + 1],
                     kernel_size=3, stride=2)
            for i in range(len(layers) - 1)])
        self.pool = nn.AdaptiveMaxPool2d(1)
        self.out = nn.Linear(layers[-1], c)

    def forward(self, x):
        for l in self.layers: x = F.relu(l(x))
        x = self.pool(x)
```

```
x = x.view(x.size(0), -1)
return F.log_softmax(self.out(x), dim=-1)
```

- Replace `nn.Linear` with `nn.Conv2d`
- First two parameters are exactly the same as `nn.Linear` —the number of features coming in, and the number of features coming out
- `kernel_size=3`, the size of the filter
- `stride=2` will use every other 3 by 3 area which will halve the output resolution in each dimension (i.e. it has the same effect as 2 by 2 max-pooling)

```
learn = ConvLearner.from_model_data(ConvNet([3, 20, 40, 80],
10), data)

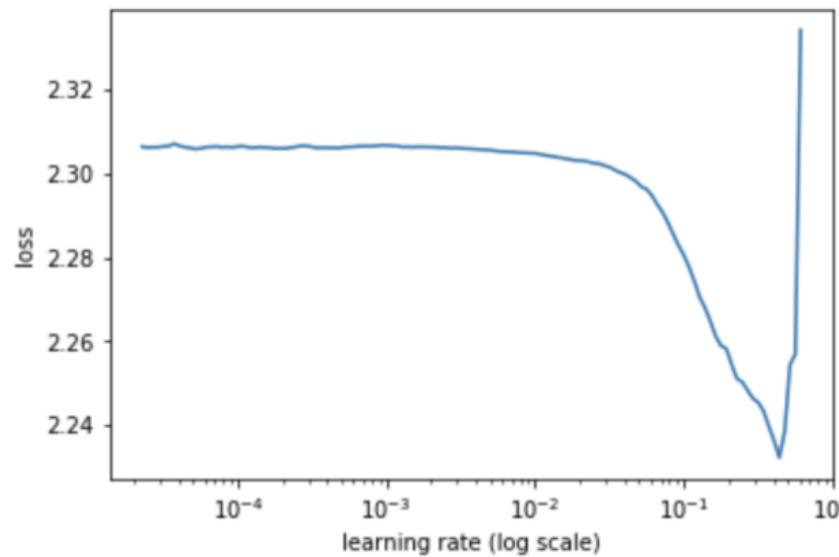
learn.summary()

OrderedDict([('Conv2d-1',
    OrderedDict([('input_shape', [-1, 3, 32, 32]),
                ('output_shape', [-1, 20, 15,
15]),
                ('trainable', True),
                ('nb_params', 560)])),
('Conv2d-2',
    OrderedDict([('input_shape', [-1, 20, 15,
15]),
                ('output_shape', [-1, 40, 7, 7]),
                ('trainable', True),
                ('nb_params', 7240)])),
('Conv2d-3',
    OrderedDict([('input_shape', [-1, 40, 7, 7]),
                ('output_shape', [-1, 80, 3, 3]),
```

```
('trainable', True),  
        ('nb_params', 28880))]),  
        ('AdaptiveMaxPool2d-4',  
         OrderedDict([('input_shape', [-1, 80, 3, 3]),  
                      ('output_shape', [-1, 80, 1, 1]),  
                      ('nb_params', 0)])),  
        ('Linear-5',  
         OrderedDict([('input_shape', [-1, 80]),  
                      ('output_shape', [-1, 10]),  
                      ('trainable', True),  
                      ('nb_params', 810)])))
```

- `ConvNet([3, 20, 40, 80], 10)` —It start with 3 RGB channels, 20, 40, 80 features, then 10 classes to predict.
- `AdaptiveMaxPool2d` —This followed by a linear layer is how you get from 3 by 3 down to a prediction of one of 10 classes and is now a standard for state-of-the-art algorithms. The very last layer, we do a special kind of max-pooling for which you specify the output activation resolution rather than how big of an area to poll. In other words, here we do 3 by 3 max-pool which is equivalent of 1 by 1 *adaptive* max-pool.
- `x = x.view(x.size(0), -1)` — `x` has a shape of # of the features by 1 by 1, so it will remove the last two layers.
- This model is called “fully convolutional network”—where every layer is convolutional except for the very last.

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



- The default final learning rate `lr_find` tries is 10. If the loss is still getting better at that point, you can overwrite by specifying `end_lr`.

```
%time learn.fit(1e-1, 2)

A Jupyter Widget

[ 0.      1.72594  1.63399  0.41338]
[ 1.      1.51599  1.49687  0.45723]

CPU times: user 1min 14s, sys: 32.3 s, total: 1min 46s
Wall time: 56.5 s
```

```
%time learn.fit(1e-1, 4, cycle_len=1)
```

A Jupyter Widget

```
[ 0.      1.36734  1.28901  0.53418]
[ 1.      1.28854  1.21991  0.56143]
[ 2.      1.22854  1.15514  0.58398]
[ 3.      1.17904  1.12523  0.59922]

CPU times: user 2min 21s, sys: 1min 3s, total: 3min 24s
Wall time: 1min 46s
```

- It flattened out around 60% accuracy. Considering it uses about 30,000 parameters (compared to 47% with 122k parameters)
- Time per epoch is about the same since their architectures are both simple and most of time is spent doing memory transfer.

Refactored [01:21:57]

Simplify `forward` function by creating `ConvLayer` (our first custom layer!). In PyTorch, layer definition and neural network definitions are identical. Anytime you have a layer, you can use it as a neural net, when you have a neural net, you can use it as a layer.

```
class ConvLayer(nn.Module):
    def __init__(self, ni, nf):
        super().__init__()
        self.conv = nn.Conv2d(ni, nf, kernel_size=3,
                           stride=2, padding=1)

    def forward(self, x): return F.relu(self.conv(x))
```

- `padding=1` —When you do convolution the image shrink by 1 pixel on each side. So it does not go from 32 by 32 to 16 by 16 but

actually 15 by 15. `padding` will add a border so we can keep the edge pixel information. It is not as big of a deal for a big image, but when it's down to 4 by 4, you really don't want to throw away a whole piece.

```
class ConvNet2(nn.Module):
    def __init__(self, layers, c):
        super().__init__()
        self.layers = nn.ModuleList([ConvLayer(layers[i],
                                             layers[i + 1])
                                    for i in range(len(layers) - 1)])
        self.out = nn.Linear(layers[-1], c)

    def forward(self, x):
        for l in self.layers: x = l(x)
        x = F.adaptive_max_pool2d(x, 1)
        x = x.view(x.size(0), -1)
        return F.log_softmax(self.out(x), dim=-1)
```

- Another difference from the last model is that `nn.AdaptiveMaxPool2d` does not have any state (i.e. no weights). So we can just call it as a function `F.adaptive_max_pool2d` .

BatchNorm [1:25:10]

- The last model, when we tried to add more layers, we had trouble training. The reason we had trouble training was that if we used larger learning rates, it would go off to NaN and if we used smaller learning rate, it would take forever and doesn't have a chance to explore properly—so it was not resilient.

- To make it resilient, we will use something called batch normalization. BatchNorm came out about two years ago and it has been quite transformative since it suddenly makes it really easy to train deeper networks.
- We can simply use `nn.BatchNorm` but to learn about it, we will write it from scratch.
- It is unlikely that the weight matrices on average are not going to cause your activations to keep getting smaller and smaller or keep getting bigger and bigger. It is important to keep them at reasonable scale. So we start things off with zero-mean standard deviation one by normalizing the input. What we really want to do is to do this for all layers, not just the inputs.

```
class BnLayer(nn.Module):
    def __init__(self, ni, nf, stride=2, kernel_size=3):
        super().__init__()
        self.conv = nn.Conv2d(ni, nf,
kernel_size=kernel_size,
                           stride=stride, bias=False,
padding=1)
        self.a = nn.Parameter(torch.zeros(nf,1,1))
        self.m = nn.Parameter(torch.ones(nf,1,1))

    def forward(self, x):
        x = F.relu(self.conv(x))
        x_chan =
x.transpose(0,1).contiguous().view(x.size(1), -1)
        if self.training:
            self.means = x_chan.mean(1)[:,None,None]
            self.stds = x_chan.std (1)[:,None,None]
        return (x-self.means) / self.stds *self.m + self.a
```

- Calculate the mean of each channel or each filter and standard deviation of each channel or each filter. Then subtract the means and divide by the standard deviations.
- We no longer need to normalize our input because it is normalizing it per channel or for later layers it is normalizing per filter.
- Turns out this is not enough since SGD is bloody-minded [01:29:20]. If SGD decided that it wants matrix to be bigger/smaller overall, doing `(x=self.means) / self.stds` is not enough because SGD will undo it and try to do it again in the next mini-batch. So we will add two parameters: `a` —adder (initial value zeros) and `m` —multiplier (initial value ones) for each channel.
- `Parameter` tells PyTorch that it is allowed to learn these as weights.
- Why does this work? If it wants to scale the layer up, it does not have to scale up every single value in the matrix. It can just scale up this single trio of numbers `self.m`, if it wants to shift it all up or down a bit, it does not have to shift the entire weight matrix, they can just shift this trio of numbers `self.a`. Intuition: We are normalizing the data and then we are saying you can then shift it and scale it using far fewer parameters than would have been necessary if it were to actually shift and scale the entire set of convolutional filters. In practice, it allows us to increase our learning rates, it increase the resilience of training, and it allows us to add more layers and still train effectively.

- The other thing batch norm does is that it regularizes, in other words, you can often decrease or remove dropout or weight decay. The reason why is each mini-batch is going to have a different mean and a different standard deviation to the previous mini-batch. So they keep changing and it is changing the meaning of the filters in a subtle way acting as a noise (i.e. regularization).
- In real version, it does not use this batch's mean and standard deviation but takes an exponentially weighted moving average standard deviation and mean.
- `if self.training`—this is important because when you are going through the validation set, you do not want to be changing the meaning of the model. There are some types of layer that are actually sensitive to what the mode of the network is whether it is in training mode or evaluation/test mode. There was a bug when we implemented mini net for MovieLens that dropout was applied during the validation—which was fixed. In PyTorch, there are two such layer: dropout and batch norm. `nn.Dropout` already does the check.
- [01:37:01] The key difference in fast.ai which no other library does is that these means and standard deviations get updated in training mode in every other library as soon as you basically say I am training, regardless of whether that layer is set to trainable or not. With a pre-trained network, that is a terrible idea. If you have a pre-trained network for specific values of those means and standard deviations in batch norm, if you change them, it changes the meaning of those pre-trained layers. In fast.ai, always by default, it will not touch those means and standard deviations if your layer is frozen. As soon as you un-freeze it, it will start

updating them unless you set `learn.bn_freeze=True`. In practice, this often seems to work a lot better for pre-trained models particularly if you are working with data that is quite similar to what the pre-trained model was trained with.

- Where do you put batch-norm layer? We will talk more in a moment, but for now, after `relu`

Ablation Study [01:39:41]

It is something where you try turning on and off different pieces of your model to see which bits make which impacts, and one of the things that wasn't done in the original batch norm paper was any kind of effective ablation. And one of the things therefore that was missing was this question which was just asked—where to put the batch norm. That oversight caused a lot of problems because it turned out the original paper did not actually put it in the best spot. Other people since then have now figured that out and when Jeremy show people code where it is actually in the spot that is better, people say his batch norm is in the wrong spot.

- Try and always use batch norm on every layer if you can
- Don't stop normalizing your data so that people using your data will know how you normalized your data. Other libraries might not deal with batch norm for pre-trained models correctly, so when people start re-training, it might cause problems.

```
class ConvBnNet(nn.Module):
    def __init__(self, layers, c):
        super().__init__()
```

```

    self.conv1 = nn.Conv2d(3, 10, kernel_size=5,
    stride=1, padding=2)
    self.layers = nn.ModuleList([BnLayer(layers[i],
    layers[i + 1])
        for i in range(len(layers) - 1)])
    self.out = nn.Linear(layers[-1], c)

    def forward(self, x):
        x = self.conv1(x)
        for l in self.layers: x = l(x)
        x = F.adaptive_max_pool2d(x, 1)
        x = x.view(x.size(0), -1)
        return F.log_softmax(self.out(x), dim=-1)

```

- Rest of the code is similar—Using `BnLayer` instead of `ConvLayer`
- A single convolutional layer was added at the start trying to get closer to the modern approaches. It has a bigger kernel size and a stride of 1. The basic idea is that we want the first layer to have a richer input. It does convolution using the 5 by 5 area which allows it to try and find more interesting richer features in that 5 by 5 area, then spit out bigger output (in this case, it's 10 by 5 by 5 filters). Typically it is 5 by 5 or 7 by 7, or even 11 by 11 convolution with quite a few filters coming out (e.g. 32 filters).
- Since `padding = kernel_size-1 / 2` and `stride=1`, the input size is the same as the output size—just more filters.
- It is a good way of trying to create a richer starting point.

Deep BatchNorm [01:50:52]

Let's increase the depth of the model. We cannot just add more of stride 2 layers since it halves the size of the image each time. Instead, after each stride 2 layer, we insert a stride 1 layer.

```
class ConvBnNet2(nn.Module):
    def __init__(self, layers, c):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 10, kernel_size=5,
                           stride=1, padding=2)
        self.layers = nn.ModuleList([BnLayer(layers[i],
                                             layers[i+1])
                                    for i in range(len(layers) - 1)])
        self.layers2 = nn.ModuleList([BnLayer(layers[i+1],
                                             layers[i + 1], 1)
                                    for i in range(len(layers) - 1)])
        self.out = nn.Linear(layers[-1], c)

    def forward(self, x):
        x = self.conv1(x)
        for l, l2 in zip(self.layers, self.layers2):
            x = l(x)
            x = l2(x)
        x = F.adaptive_max_pool2d(x, 1)
        x = x.view(x.size(0), -1)
        return F.log_softmax(self.out(x), dim=-1)

learn = ConvLearner.from_model_data((ConvBnNet2([10, 20, 40,
                                                 80, 160], 10), data))

%time learn.fit(1e-2, 2)
```

A Jupyter Widget

```
[ 0.          1.53499  1.43782  0.47588]
[ 1.          1.28867  1.22616  0.55537]
```

CPU times: user 1min 22s, sys: 34.5 s, total: 1min 56s
Wall time: 58.2 s

```
%time learn.fit(1e-2, 2, cycle_len=1)
```

A Jupyter Widget

```
[ 0.       1.10933  1.06439  0.61582]
[ 1.       1.04663  0.98608  0.64609]

CPU times: user 1min 21s, sys: 32.9 s, total: 1min 54s
Wall time: 57.6 s
```

The accuracy remained the same as before. This is now 12 layers deep, and it is too deep even for batch norm to handle. It is possible to train 12 layer deep conv net but it starts to get difficult. And it does not seem to be helping much if at all.

ResNet [01:52:43]

```
class ResnetLayer(BnLayer):
    def forward(self, x): return x + super().forward(x)

class Resnet(nn.Module):
    def __init__(self, layers, c):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 10, kernel_size=5,
                           stride=1, padding=2)
        self.layers = nn.ModuleList([BnLayer(layers[i],
                                             layers[i+1])
                                    for i in range(len(layers) - 1)])
        self.layers2 =
        nn.ModuleList([ResnetLayer(layers[i+1], layers[i + 1], 1)
                      for i in range(len(layers) - 1)])
        self.layers3 =
        nn.ModuleList([ResnetLayer(layers[i+1], layers[i + 1], 1)
                      for i in range(len(layers) - 1)])
        self.out = nn.Linear(layers[-1], c)

    def forward(self, x):
        x = self.conv1(x)
        for l1,l2,l3 in zip(self.layers, self.layers2,
```

```
self.layers3):
    x = 13(l2(l(x)))
    x = F.adaptive_max_pool2d(x, 1)
    x = x.view(x.size(0), -1)
    return F.log_softmax(self.out(x), dim=-1)
```

- `ResnetLayer` inherit from `BnLayer` and override `forward`.
- Then add bunch of layers and make it 3 times deeper, ad it still trains beautifully just because of `x + super().forward(x)`.

```
learn = ConvLearner.from_model_data(Resnet([10, 20, 40, 80,
160], 10), data)
```

```
wd=1e-5
```

```
%time learn.fit(1e-2, 2, wds=wd)
```

A Jupyter Widget

```
[ 0.      1.58191  1.40258  0.49131]
[ 1.      1.33134  1.21739  0.55625]
```

```
CPU times: user 1min 27s, sys: 34.3 s, total: 2min 1s
Wall time: 1min 3s
```

```
%time learn.fit(1e-2, 3, cycle_len=1, cycle_mult=2, wds=wd)
```

A Jupyter Widget

```
[ 0.      1.11534  1.05117  0.62549]
[ 1.      1.06272  0.97874  0.65185]
[ 2.      0.92913  0.90472  0.68154]
[ 3.      0.97932  0.94404  0.67227]
```

```
[ 4.      0.88057  0.84372  0.70654]
[ 5.      0.77817  0.77815  0.73018]
[ 6.      0.73235  0.76302  0.73633]
```

```
CPU times: user 5min 2s, sys: 1min 59s, total: 7min 1s
Wall time: 3min 39s
```

```
%time learn.fit(1e-2, 8, cycle_len=4, wds=wd)
```

A Jupyter Widget

```
[ 0.      0.8307   0.83635  0.7126 ]
[ 1.      0.74295  0.73682  0.74189]
[ 2.      0.66492  0.69554  0.75996]
[ 3.      0.62392  0.67166  0.7625 ]
[ 4.      0.73479  0.80425  0.72861]
[ 5.      0.65423  0.68876  0.76318]
[ 6.      0.58608  0.64105  0.77783]
[ 7.      0.55738  0.62641  0.78721]
[ 8.      0.66163  0.74154  0.7501 ]
[ 9.      0.59444  0.64253  0.78106]
[ 10.     0.53       0.61772  0.79385]
[ 11.     0.49747  0.65968  0.77832]
[ 12.     0.59463  0.67915  0.77422]
[ 13.     0.55023  0.65815  0.78106]
[ 14.     0.48959  0.59035  0.80273]
[ 15.     0.4459    0.61823  0.79336]
[ 16.     0.55848  0.64115  0.78018]
[ 17.     0.50268  0.61795  0.79541]
[ 18.     0.45084  0.57577  0.80654]
[ 19.     0.40726  0.5708   0.80947]
[ 20.     0.51177  0.66771  0.78232]
[ 21.     0.46516  0.6116   0.79932]
[ 22.     0.40966  0.56865  0.81172]
[ 23.     0.3852   0.58161  0.80967]
[ 24.     0.48268  0.59944  0.79551]
[ 25.     0.43282  0.56429  0.81182]
[ 26.     0.37634  0.54724  0.81797]
[ 27.     0.34953  0.54169  0.82129]
[ 28.     0.46053  0.58128  0.80342]
[ 29.     0.4041   0.55185  0.82295]
```

```
[ 30.      0.3599  0.53953  0.82861]
[ 31.      0.32937 0.55605  0.82227]

CPU times: user 22min 52s, sys: 8min 58s, total: 31min 51s
Wall time: 16min 38s
```

ResNet block [01:53:18]

```
return x + super().forward(x)
```

$$y = x + f(x)$$

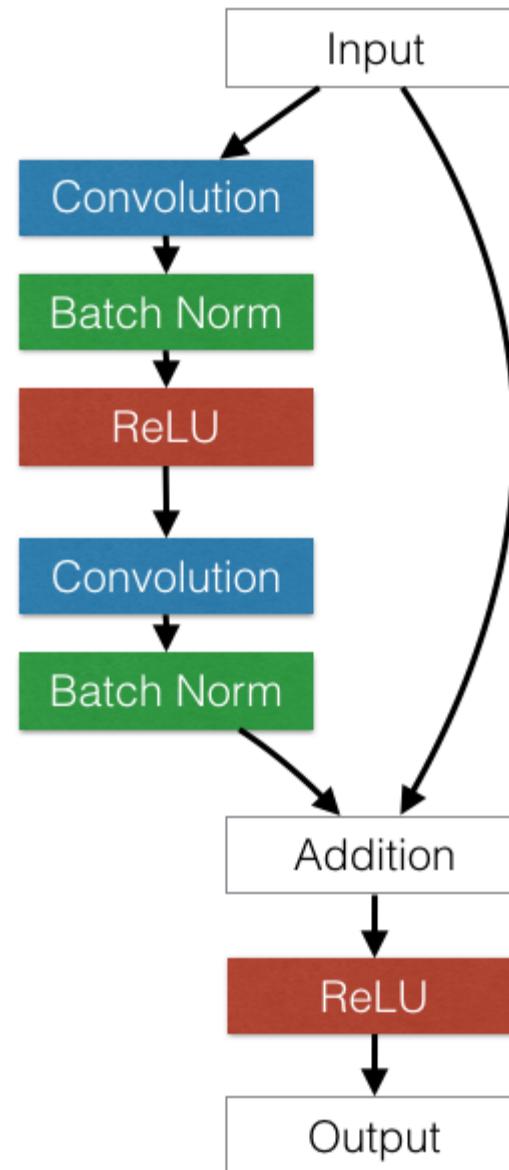
Where x is prediction from the previous layer, y is prediction from the current layer. Shuffle around the formula and we get: formula shuffle

$$f(x) = y - x$$

The difference $y - x$ is **residual**. The residual is the error in terms of what we have calculated so far. What this is saying is that try to find a set of convolutional weights that attempts to fill in the amount we were off by. So in other words, we have an input, and we have a function which tries to predict the error (i.e. how much we are off by). Then we add a prediction of how much we were wrong by to the input, then add another prediction of how much we were wrong by that time, and repeat that layer after layer—zooming into the correct answer. This is based on a theory called **boosting**.

- The full ResNet does two convolutions before it gets added back to the original input (we did just one here).

- In every block `x = 13(12(1(x)))`, one of the layers is not a `ResnetLayer` but a standard convolution with `stride=2` —this is called a “bottleneck layer”. ResNet does not convolutional layer but a different form of bottleneck block which we will cover in Part 2.



ResNet 2 [01:59:33]

Here, we increased the size of features and added dropout.

```
class Resnet2(nn.Module):
    def __init__(self, layers, c, p=0.5):
        super().__init__()
        self.conv1 = BnLayer(3, 16, stride=1, kernel_size=7)
        self.layers = nn.ModuleList([BnLayer(layers[i],
                                             layers[i+1])
                                    for i in range(len(layers) - 1)])
        self.layers2 =
            nn.ModuleList([ResnetLayer(layers[i+1], layers[i + 1], 1)
                          for i in range(len(layers) - 1)])
        self.layers3 =
            nn.ModuleList([ResnetLayer(layers[i+1], layers[i + 1], 1)
                          for i in range(len(layers) - 1)])
        self.out = nn.Linear(layers[-1], c)
        self.drop = nn.Dropout(p)

    def forward(self, x):
        x = self.conv1(x)
        for l1,l2,l3 in zip(self.layers, self.layers2,
                            self.layers3):
            x = l3(l2(l1(x)))
            x = F.adaptive_max_pool2d(x, 1)
            x = x.view(x.size(0), -1)
            x = self.drop(x)
        return F.log_softmax(self.out(x), dim=-1)

learn = ConvLearner.from_model_data(Resnet2([16, 32, 64,
                                              128, 256], 10, 0.2), data)

wd=1e-6

%time learn.fit(1e-2, 2, wds=wd)
%time learn.fit(1e-2, 3, cycle_len=1, cycle_mult=2, wds=wd)
%time learn.fit(1e-2, 8, cycle_len=4, wds=wd)

log_preds,y = learn.TTA()
preds = np.mean(np.exp(log_preds),0)
```

```
metrics.log_loss(y,preds), accuracy(preds,y)
(0.44507397166057938, 0.8490999999999997)
```

85% was a state-of-the-art back in 2012 or 2013 for CIFAR 10.

Nowadays, it is up to 97% so there is a room for improvement but all based on these techniques:

- Better approaches to data augmentation
- Better approaches to regularization
- Some tweaks on ResNet

Question [02:01:07]: Can we apply “training on the residual” approach for non-image problem? Yes! But it has been ignored everywhere else. In NLP, “transformer architecture” recently appeared and was shown to be the state of the art for translation, and it has a simple ResNet structure in it. This general approach is called “skip connection” (i.e. the idea of skipping over a layer) and appears a lot in computer vision, but nobody else much seems to be using it even though there is nothing computer vision specific about it. Good opportunity!

Dogs vs. Cats [02:02:03]

Going back dogs and cats. We will create resnet34 (if you are interested in what the trailing number means, see here—just different parameters).

```
PATH = "data/dogscats/"
sz = 224
arch = resnet34 # <-- Name of the function
bs = 64

m = arch(pretrained=True) # Get a model w/ pre-trained
weight loaded
m

ResNet(
    (conv1): Conv2d (3, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1,
affine=True)
    (relu): ReLU(inplace)
    (maxpool): MaxPool2d(kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), dilation=(1, 1))
    (layer1): 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)
            (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)
        )
        (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)
            (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)
        )
        (2): 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)
        (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)
    )
)
(layer2): Sequential(
    (0): BasicBlock(
        (conv1): Conv2d (64, 128, kernel_size=(3, 3), stride=
(2, 2), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1,
affine=True)
        (relu): ReLU(inplace)
        (conv2): Conv2d (128, 128, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1,
affine=True)
    )
    (downsample): Sequential(
        (0): Conv2d (64, 128, kernel_size=(1, 1), stride=(2,
2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1,
affine=True)
    )
)
(1): BasicBlock(
    (conv1): Conv2d (128, 128, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1,
affine=True)
    (relu): ReLU(inplace)
    (conv2): Conv2d (128, 128, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1,
affine=True)
)
(2): BasicBlock(
    (conv1): Conv2d (128, 128, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1,
affine=True)
```

```
(relu): ReLU(inplace)
(conv2): Conv2d (128, 128, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
(bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1,
affine=True)
)
(3): BasicBlock(
(conv1): Conv2d (128, 128, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
(bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1,
affine=True)
(relu): ReLU(inplace)
(conv2): Conv2d (128, 128, kernel_size=(3, 3), stride=
(1, 1), padding=(1, 1), bias=False)
(bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1,
affine=True)
)
)
...
(avgpool): AvgPool2d(kernel_size=7, stride=7, padding=0,
ceil_mode=False, count_include_pad=True)
(fc): Linear(in_features=512, out_features=1000)
)
```

Our ResNet model had Relu → BatchNorm. TorchVision does BatchNorm → Relu. There are three different versions of ResNet floating around, and the best one is PreAct (<https://arxiv.org/pdf/1603.05027.pdf>).

- Currently, the final layer has a thousands features because ImageNet has 1000 features, so we need to get rid of it.
- When you use fast.ai's `ConvLearner`, it deletes the last two layers for you. fast.ai replaces `AvgPool2d` with Adaptive Average Pooling

and Adaptive Max Pooling and concatenate the two together.

- For this exercise, we will do a simple version.

```
m = nn.Sequential(*children(m)[:-2],  
                  nn.Conv2d(512, 2, 3, padding=1),  
                  nn.AdaptiveAvgPool2d(1), Flatten(),  
                  nn.LogSoftmax())
```

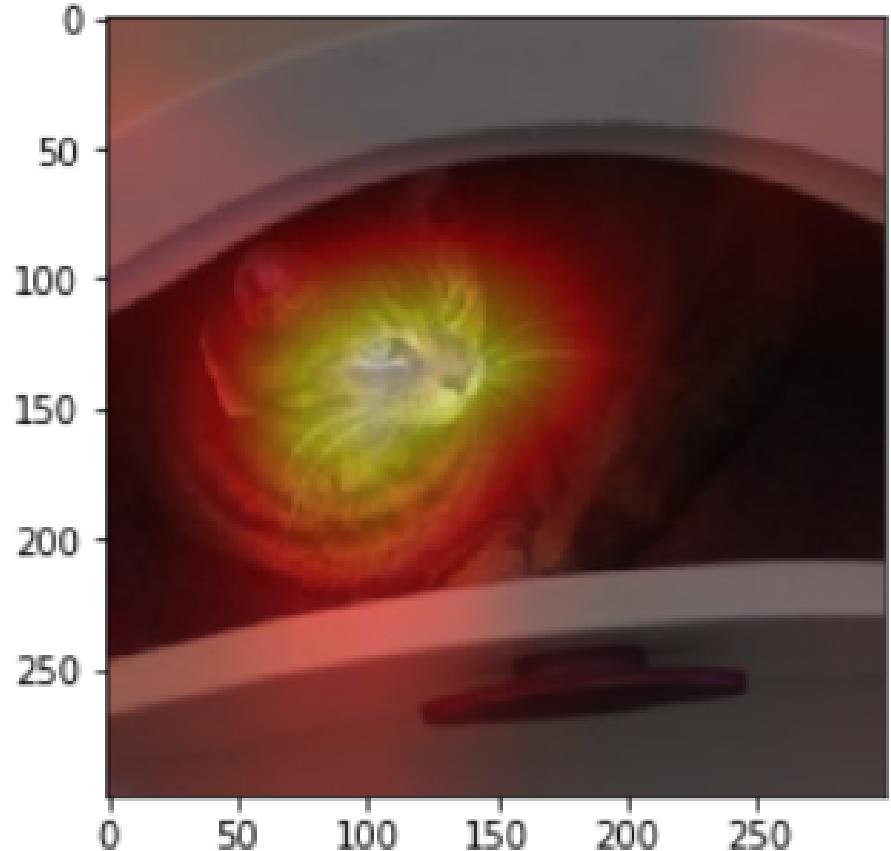
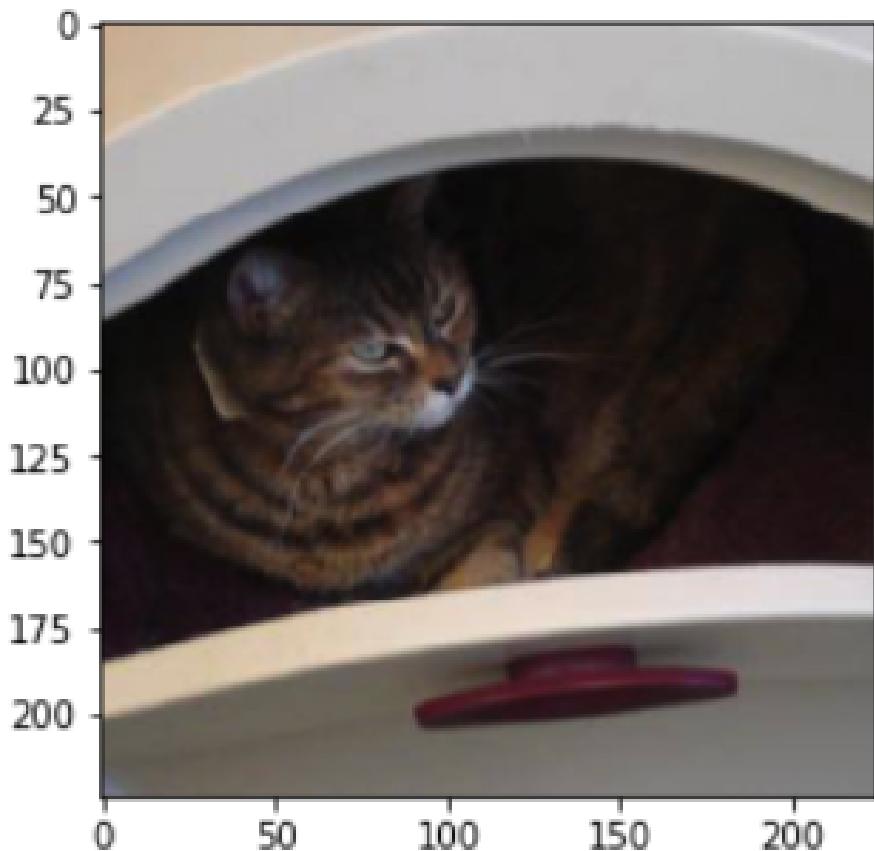
- Remove the last two layers
- Add a convolution which just has 2 outputs.
- Do average pooling then softmax
- There is no linear layer at the end. This is a different way of producing just two numbers—which allows us to do CAM!

```
tfms = tfms_from_model(arch, sz,  
                      aug_tfms=transforms_side_on, max_zoom=1.1)  
data = ImageClassifierData.from_paths(PATH, tfms=tfms,  
                                     bs=bs)  
  
learn = ConvLearner.from_model_data(m, data)  
  
learn.freeze_to(-4)  
  
learn.fit(0.01, 1)  
learn.fit(0.01, 1, cycle_len=1)
```

- `ConvLearner.from_model` is what we learned about earlier—allows us to create a Learner object with custom model.
- Then freeze the layer except the ones we just added.

Class Activation Maps (CAM) [02:08:55]

We pick a specific image, and use a technique called CAM where we take a model and we ask it which parts of the image turned out to be important.



How did it do this? Let's work backwards. The way it did it was by producing this matrix:

```
array([[ 0.06796,  0.09529,  0.04571,  0.02226,  0.01783,  0.02682,  0.      ],
       [ 0.18074,  0.25366,  0.21349,  0.17134,  0.12896,  0.07998,  0.02269],
       [ 0.32468,  0.53808,  0.58196,  0.51675,  0.36953,  0.20348,  0.06563],
       [ 0.4753 ,  0.82032,  0.93738,  0.84066,  0.58015,  0.30568,  0.09089],
       [ 0.53621,  0.90491,  1.      ,  0.87108,  0.60574,  0.33723,  0.11391],
       [ 0.44609,  0.70038,  0.72927,  0.60194,  0.42627,  0.26258,  0.10247],
       [ 0.21954,  0.3172 ,  0.30497,  0.21987,  0.15255,  0.08741,  0.00709]], dtype=float32)
```

Big numbers correspond to the cat. So what is this matrix? This matrix simply equals to the value of feature matrix `feat` times `py` vector:

```
f2=np.dot(np.rollaxis(feat,0,3), py)
f2-=f2.min()
f2/=f2.max()
f2
```

`py` vector is the predictions that says “I am 100% confident it’s a cat.” `feat` is the values ($2 \times 7 \times 7$) coming out of the final convolutional layer (the `Conv2d` layer we added). If we multiply `feat` by `py`, we get all of the first channel and none of the second channel. Therefore, it is going to return the value of the last convolutional layers for the section which lines up with being a cat. In other words, if we multiply `feat` by `[0, 1]`, it will line up with being a dog.

```
sf = SaveFeatures(m[-4])
py = m(Variiable(x.cuda()))
sf.remove()
```

```
py = np.exp(to_np(py)[0]); py  
  
array([ 1.,  0.], dtype=float32)  
  
feat = np.maximum(0, sf.features[0])  
feat.shape
```

Put it in another way, in the model, the only thing that happened after the convolutional layer was an average pooling layer. The average pooling layer took the 7 by 7 grid and averaged out how much each part is “cat-like”. We then took the “cattyness” matrix, resized it to be the same size as the original cat image, and overlaid it on top, then you get the heat map.

The way you can use this technique at home is

1. when you have a large image, you can calculate this matrix on a quick small little convolutional net
2. zoom into the area that has the highest value
3. re-run it just on that part

We skipped this over quickly as we ran out of time, but we will learn more about these kind of approaches in Part 2.

“Hook” is the mechanism that lets us ask the model to return the matrix.

`register_forward_hook` asks PyTorch that every time it calculates a layer it runs the function given—sort of like a callback that happens

every time it calculates a layer. In the following case, it saves the value of the particular layer we were interested in:

```
class SaveFeatures():
    features=None
    def __init__(self, m):
        self.hook = m.register_forward_hook(self.hook_fn)
    def hook_fn(self, module, input, output):
        self.features = to_np(output)
    def remove(self): self.hook.remove()
```

Questions to Jeremy [02:14:27]: "Your journey into Deep Learning" and "How to keep up with important research for practitioners"

"If you intend to come to Part 2, you are expected to master all the techniques you have learned in Part 1". Here are something you can do:

1. Watch each of the video at least 3 times.
2. Make sure you can re-create the notebooks without watching the videos—maybe do so with different datasets to make it more interesting.
3. Keep an eye on the forum for recent papers, recent advances.
4. Be tenacious and keep working at it!

